THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Pro Git

Scott Chacon, Ben Straub

Versão 2.1.47-7-ge30df8c, 2024-10-28

Índice

Prefácio por Scott Chacon

Bem vindo à segunda edição do Pro Git. A primeira edição foi publicada há mais de quatro anos. Desde então muita coisa mudou e ainda muitas coisas importantes não mudaram. Enquanto a maioria dos comandos e conceitos fundamentais ainda hoje são válidos por conta da equipe do Git ser fantástica em manter as coisas retrocompatíveis, houveram algumas adições e modificações significativas na comunidade que envolve o Git. A segunda edição deste livro é destinada a pontuar essas mudanças e atualizar o livro para que ele possa ser mais proveitoso aos novos usuários.

Quando eu escrevi a primeira edição deste livro, Git ainda era uma ferramenta relativamente difícil de se usar e pouco adotada pelos hackers mais hardcore. Ele estava começando a ganhar força em certas comunidades, mas não tinha chegado nem perto da ubiquidade que tem hoje. Desde então, praticamente todas as comunidades open source o adotaram. Git obteve um progresso incrível no Windows, com a explosão de interfaces gráficas criadas para ele, no suporte em IDEs e em uso corporativo. O Pro Git de quatro anos atrás não sabia de praticamente nada disso. Um dos principais objetivos dessa nova edição é abordar todas essas novas fronteiras na comunidade Git.

A comunidade Open Source usando Git também explodiu. Quando eu originalmente sentei para escrever o livro cerca de cinco anos atrás (tomou-me um tempo para publicar a primeira versão), eu tinha acabado de conseguir emprego numa companhia pouco conhecida desenvolvendo um website de hospedagem de Git, chamado GitHub. Na época da publicação haviam talvez algumas milhares de pessoas usando o site e somente quatro de nós trabalhando nele. Enquanto escrevo essa introdução, GitHub está anunciando nosso décimo milionésimo projeto hospedado, com aproximadamente 5 milhões de contas de desenvolvedores cadastradas e mais de 230 funcionários. Ame-o ou odeie-o, GitHub mudou drasticamente grandes esferas da comunidade Open Source de uma forma que era praticamente inconcebível quando me sentei para escrever a primeira edição.

Escrevi uma pequena seção na versão original de Pro Git sobre o GitHub como um exemplo de hospedagem Git, com a qual nunca estive muito confortável. Eu não gostava muito do fato que eu estava escrevendo algo que eu sentia que era essencialmente um recurso da comunidade, falando sobre a minha empresa no meio. Embora eu ainda não ame esse conflito de interesses, a importância do GitHub na comunidade Git é inevitável. Ao invés de usar como exemplo de hospedagem Git, eu decidi transformar aquela parte do livro em uma descrição mais aprofundada sobre o que GitHub é e como usá-lo efetivamente. Se você irá aprender como usar o Git, saber como usar o GitHub ajudará você a fazer parte de uma imensa comunidade, o que é valioso independentemente de qual serviço de hospedagem Git você decidir usar para seus códigos.

A outra grande mudança desde a época da última publicação tem sido o desenvolvimento a ascensão do protocolo HTTP para transações em rede do Git. A maioria dos exemplos do livro foi mudada do SSH para o HTTP por ele ser muito mais simples.

Tem sido maravilhoso assistir o Git crescer nos últimos anos, indo de um sistema de controle de versões relativamente obscuro para basicamente dominar os sistemas de controle de versão comerciais e open source. Eu estou feliz que Pro Git foi tão bem sucedido e também foi capaz de ser um dos poucos livros técnicos no mercado que ambos obteve um sucesso considerável e é completamente open source.

Espero que você aprecie esta versão atualizada de Pro Git.

Prefácio por Ben Straub

A primeira edição deste livro foi o que me viciou em Git. Essa foi minha introdução a um estilo de fazer software que parecia mais natural que qualquer coisa que eu já tinha visto. Eu já era programador há vários anos na época, mas essa foi a virada certa que me colocou num caminho muito mais interessante do que o que eu estava.

Agora, anos depois, eu sou um contribuidor de uma das principais implementações de Git, trabalhei para a maior companhia de hospedagem Git, e viajei o mundo ensinando as pessoas sobre Git. Quando Scott me perguntou se eu tinha interesse em trabalhar na segunda edição, eu não precisei nem pensar.

Tem sido um grande prazer e privilégio trabalhar neste livro. Espero que ele te ajude tanto quanto ele me ajudou.

Dedicatória

À minha esposa, Becky, sem a qual essa aventura nunca teria começado. — Ben

Essa edição é dedicada às minhas meninas. À minha esposa Jessica que me apoiou todos esses anos e à minha filha Josephine, que me apoiará quando eu estiver velho demais para entender o que está acontecendo. — Scott

Introdução

Você está prestes a passar várias horas de sua vida lendo sobre Git. Vamos fazer uma pequena pausa para explicar o que está diante de você. Abaixo um resumo dos dez capítulos e dos três apêndices deste livro.

No **Capítulo 1**, nós vamos falar sobre Sistemas de Controle de Versão (VCSs - **Version Control System**) e o básico do Git - nada técnico, somente o que é o Git, porque ele apareceu em uma terra cheia de VCSs, o que o diferencia, e porque tantas pessoas estão usando. Então, nós vamos explicar como baixar Git e configurá-lo pela primeira vez se você já não o tem em seu sistema.

No **Capítulo 2**, nós falaremos do uso básico do Git - como usar o Git em 80% dos casos que você vai encontrar com mais frequência. Após ter lido este capítulo, você será capaz de clonar um repositório, ver o que aconteceu na história do projeto, modificar arquivos e contribuir com alterações. Se o livro entrar em combustão espontânea neste ponto, você já será bem capaz de usar o Git durante o tempo que levará até conseguir outra cópia do livro.

Capítulo 3 é sobre o modelo de ramificações **(branching)** no Git, frequentemente descrito como a característica mais sagaz do Git. Aqui você vai aprender o que diferencia o Git do resto. Quando você tiver terminado de ler este capítulo, você vai precisar de um tempo meditando sobre como você viveu antes de as ramificações do Git fazerem parte da sua vida.

Capítulo 4 fala sobre Git em servidores. Este capítulo é para aqueles que querem configurar Git na sua empresa ou em seu servidor pessoal. Nós também falaremos sobre várias opções disponíveis na internet se você preferir deixar alguém cuidar disso para você.

Capítulo 5 cobrirá em detalhes vários fluxos de trabalho distribuídos e como desenvolvê-los com o Git. Ao terminar este capítulo, você será capaz de trabalhar confortavelmente com múltiplos repositórios remotos, usar Git através de e-mail e sagazmente usar vários ramos remotos e correções de colaboradores.

Capítulo 6 fala em detalhes sobre o serviço de hospedagem GitHub e suas ferramentas. Nós falaremos sobre como se inscrever e gerenciar uma conta, criar e usar repositórios Git, fluxos de trabalho comuns para colaborar com projetos e para aceitar contribuições em seus projetos, a interface programática do GitHub e várias dicas gerais para deixar sua vida mais fácil.

Capítulo 7 trata de comandos avançados do Git. Você vai aprender sobre assuntos como dominar o temido comando *reset*, usar procura binária para identificar erros, editar a história, seleção de revisão em detalhes, e muito mais. Este capítulo complementa seu conhecimento sobre o Git e fará de você um verdadeiro mestre.

Capítulo 8 é sobre configurar o seu próprio ambiente Git personalizado. Isto inclui configurar **hook scripts** para forçar ou incentivar diretivas personalizadas e usar definições de configurações ambiental tal que você possa trabalhar da forma como prefira. Nós também falamos sobre criar seu conjunto de scripts para reforçar diretivas personalizadas de comprometimento **(committing)**.

Capítulo 9 trata sobre o Git e outros VCSs. Isto inclui o uso do Git em um mundo Subversion (SVN) e converter projetos de outras VCSs para o Git. Muitas organizações ainda usam SVN e são pretendem migrar, mas neste ponto você terá aprendido o incrível poder do Git - e este capítulo mostra como

enfrentar o desafio de usar um servidor SVN. Nós também mostramos como importar projetos de vários sistemas diferentes caso você não convença todo mundo a usar o Git.

Capítulo 10 penetra nas sombrias porém bonitas profundezas do Git. Agora que você sabe tudo sobre o Git e pode usá-lo com poder e graça, você pode continuar e aprender como Git armazena seus objetos, o que é o modelo de objeto, detalhes sobre pacotes de arquivos, protocolos de servidor, e mais. Através do livro, serão feitas referências a seções deste capítulo caso você sinta vontade de ir mais fundo no assunto; mas se você for como nós e quiser mergulhar em detalhes técnicos, talvez você prefira ler o Capítulo 10 primeiro. Você decide.

No **Apêndice A** nós olhamos para vários exemplos de uso do Git em ambientes específicos. Nós cobrimos um número de ambientes de programação GUI e IDE que você talvez queira usar com o Git e o que há de disponível para você. Se você quiser ter uma vista geral de como usar Git no seu sistema, no Visual Studio ou Eclipse, dê uma olhada aqui.

No **Apêndice** B nós exploramos scripts e extensões do Git fazendo uso de ferramentas como libgit2 e JGit. Se você tiver interesse em desenvolver ferramentas personalizadas complexas e rápidas e precisar de acesso ao Git em baixo nível, você encontrará isso neste apêndice.

Finalmente no **Apêndice C** nós cobrimos todos os comandos Git mais importantes, um de cada vez, e revemos onde eles foram vistos e o que foi feito com eles. Se você quiser saber onde cada comando foi introduzido e usado, olhe aqui.

Vamos começar.

Começando

Este capítulo abordará como começar com o Git. Nós vamos começar explicando sobre algumas ferramentas de controle de versão, então vamos falar de como ter o Git rodando no seu sistema e finalmente como configurá-lo para começar a trabalhar com ele. Ao fim deste capítulo você deve entender o porquê de o Git estar em todos os lugares, por que utilizá-lo e você estará ápto a fazé-lo.

Sobre Controle de Versão

O que é "controle de versão", e porque eu deveria me importar?

Controle de versão é um sistema que registra alterações em um arquivo ou conjunto de arquivos ao longo do tempo para que você possa lembrar versões específicas mais tarde. Para os exemplos neste livro você irá utilizar o código-fonte de software com arquivos que possuam controle de versão, embora na realidade você possa fazer isso com quase qualquer tipo de arquivo em um computador.

Se você é um designer gráfico ou *web designer* e quer manter todas as versões de uma imagem ou *layout* (o que você certamente deveria querer), um sistema de controle de versão (VCS) é a coisa correta a ser usada. Ele permite que você reverta para estado anterior determinados arquivos ou um projeto inteiro, compare as mudanças ao longo do tempo, veja quem modificou pela última vez algo que pode estar causando um problema, quem introduziu um problema, quando, e muito mais. Usar um VCS também significa que se você estragar tudo ou perder arquivos, você pode facilmente recuperar. Além disso, você tem tudo isso com muito pouco trabalho.

Sistemas Locais de Controle de Versão

O método de controle de versão de muitas pessoas é copiar os arquivos para outro diretório (talvez um diretório com carimbo de tempo, se eles forem espertos). Esta abordagem é muito comum porque é muito simples, mas também é incrivelmente propensa a erros. É fácil esquecer em qual diretório você está e acidentalmente sobreescrever o arquivo errado ou copiar arquivos que não quer.

Para lidar com este problema, programadores há muito tempo desenvolveram VCSs locais que tem um banco de dados simples que mantêm todas as alterações nos arquivos sob controle de revisão.

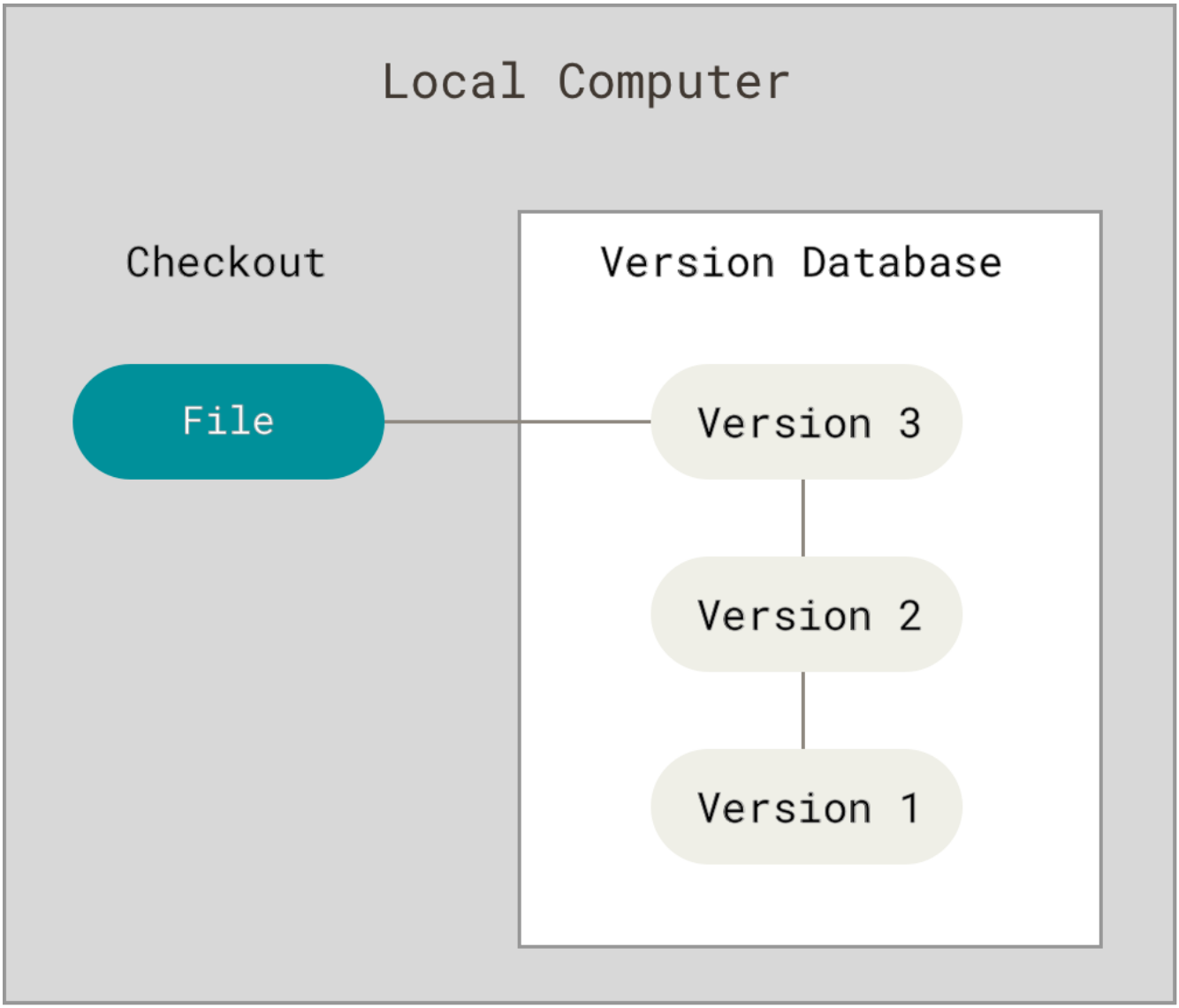


Figura 1. Controle de versão local.

Uma das ferramentas VCS mais populares foi um sistema chamado RCS, que ainda é distribuído com muitos computadores hoje. Até mesmo o popular sistema operacional Mac OS X inclui o comando `rcs` quando você instala as Ferramentas de Desenvolvimento. RCS funciona mantendo conjuntos de alterações (ou seja, as diferenças entre os arquivos) em um formato especial no disco; ele pode, em seguida, re-criar como qualquer arquivo se parecia em qualquer ponto no tempo, adicionando-se todas as alterações.

Sistemas Centralizados de Controle de Versão

A próxima questão importante que as pessoas encontram é que elas precisam colaborar com desenvolvedores em outros sistemas. Para lidar com este problema, Sistemas Centralizados de Controle de Versão (CVCSs) foram desenvolvidos. Estes sistemas, tais como CVS, Subversion e Perforce, têm um único servidor que contém todos os arquivos de controle de versão, e um número de clientes que usam arquivos a partir desse lugar central. Por muitos anos, este tem sido o padrão para controle de versão.

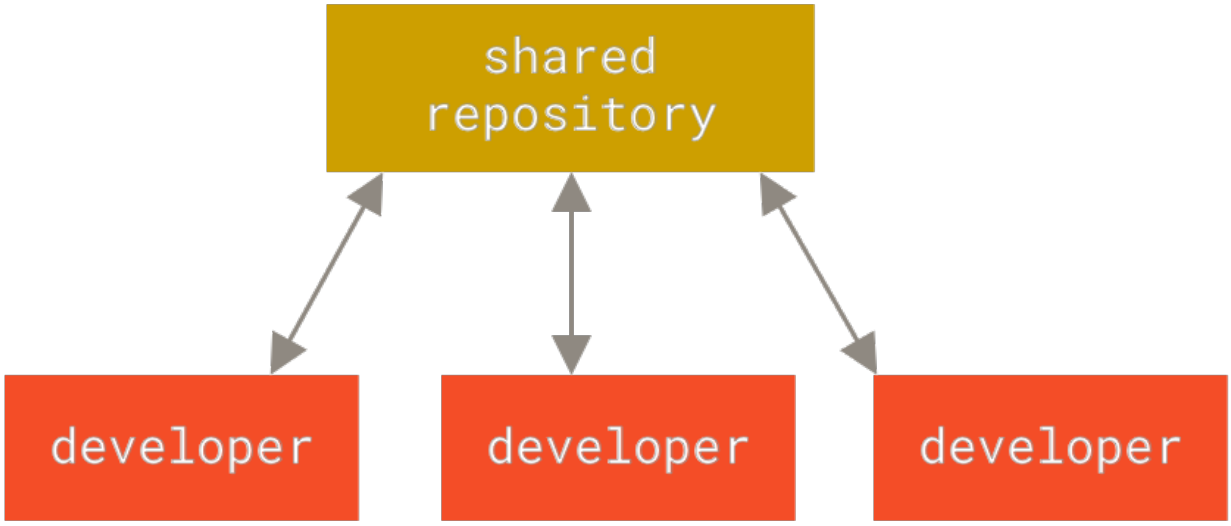


Figura 2. Controle de versão centralizado.

Esta configuração oferece muitas vantagens, especialmente sobre VCSs locais. Por exemplo, todo mundo sabe, até certo ponto o que todo mundo no projeto está fazendo. Os administradores têm controle refinado sobre quem pode fazer o que; e é muito mais fácil de administrar um CVCS do que lidar com bancos de dados locais em cada cliente.

No entanto, esta configuração também tem algumas desvantagens graves. O mais óbvio é o ponto único de falha que o servidor centralizado representa. Se esse servidor der problema por uma hora, durante essa hora ninguém pode colaborar ou salvar as alterações de versão para o que quer que eles estejam trabalhando. Se o disco rígido do banco de dados central for corrompido, e backups apropriados não foram mantidos, você perde absolutamente tudo - toda a história do projeto, exceto imagens pontuais que desenvolvedores possam ter em suas máquinas locais. Sistemas VCS locais sofrem com esse mesmo problema - sempre que você tenha toda a história do projeto em um único lugar, há o risco de perder tudo.

Sistemas Distribuídos de Controle de Versão

É aqui que Sistemas Distribuídos de Controle de Versão (DVCS) entram em cena. Em um DVCS (como Git, Mercurial, Bazaar ou Darcs), clientes não somente usam o estado mais recente dos arquivos: eles duplicam localmente o repositório completo. Assim, se qualquer servidor morrer, e esses sistemas estiverem colaborando por meio dele, qualquer um dos repositórios de clientes podem ser copiado de volta para o servidor para restaurá-lo. Cada clone é de fato um backup completo de todos os dados.

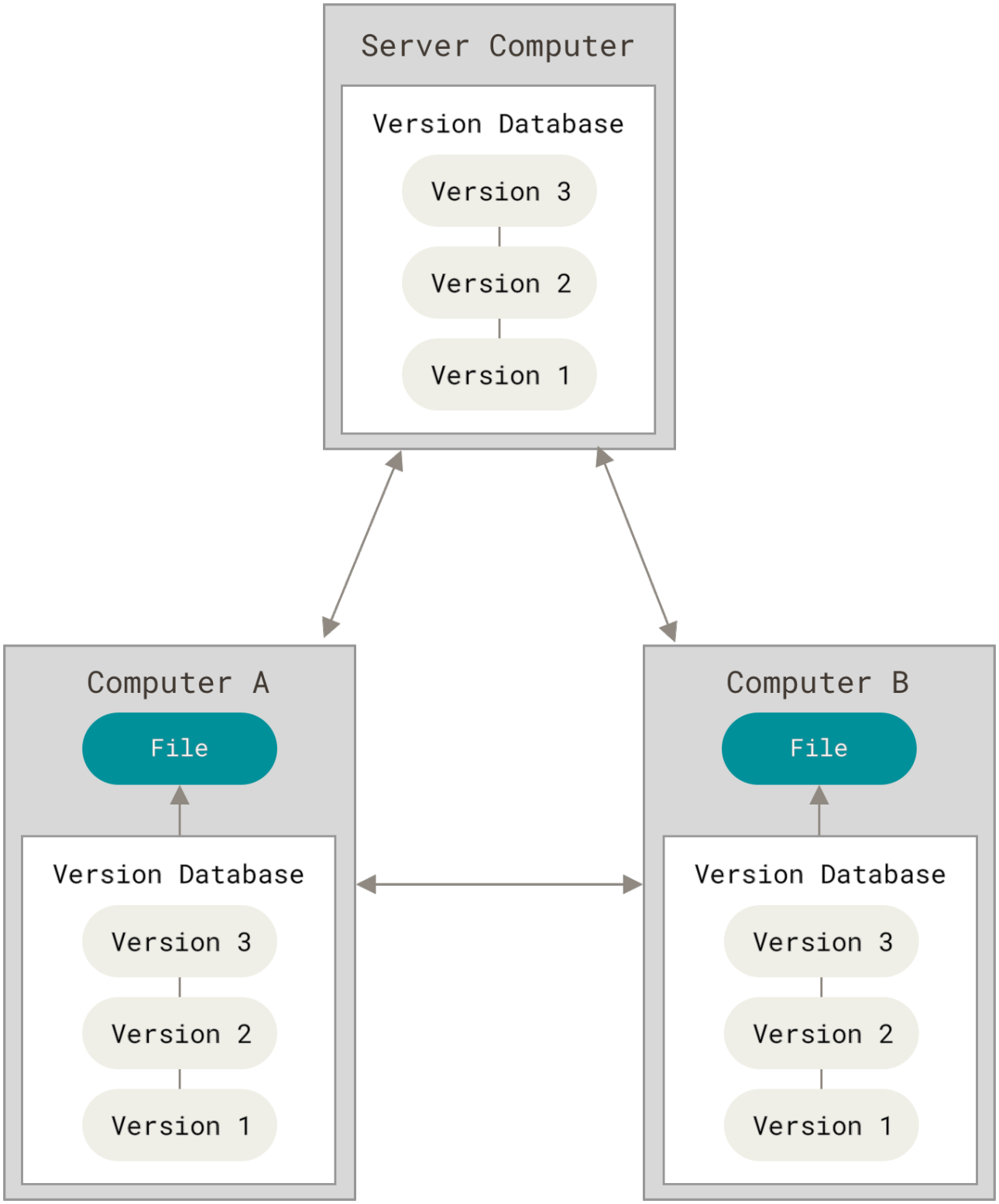


Figura 3. Controle de versão distribuído.

Além disso, muitos desses sistemas trabalham muito bem com vários repositórios remotos, tal que você possa colaborar em diferentes grupos de pessoas de maneiras diferentes ao mesmo tempo dentro do mesmo projeto. Isso permite que você configure vários tipos de fluxos de trabalho que não são possíveis em sistemas centralizados, como modelos hierárquicos.

Uma Breve História do Git

Como muitas coisas na vida, o Git começou com um pouco de destruição criativa e uma ardente controvérsia.

O núcleo (kernel) do Linux é um projeto de código aberto com um escopo bastante grande. A maior parte da vida da manutenção do núcleo o Linux (1991-2002), as mudanças no código eram compartilhadas como correções e arquivos. Em 2002, o projeto do núcleo do Linux começou usar uma DVCS proprietária chamada BitKeeper.

Em 2005, a relação entre a comunidade que desenvolveu o núcleo do Linux e a empresa que desenvolveu BitKeeper quebrou em pedaços, e a ferramenta passou a ser paga. Isto alertou a comunidade que desenvolvia o Linux (e especialmente Linus Torvalds, o criador do Linux) a desenvolver a sua própria ferramenta baseada em lições aprendidas ao usar o BitKeeper. Algumas metas do novo sistema era os seguintes:

- Velocidade
- Projeto simples
- Forte suporte para desenvolvimento não-linear (milhares de ramos paralelos)
- Completamente distribuído
- Capaz de lidar com projetos grandes como o núcleo o Linux com eficiência (velocidade e tamanho dos dados)

Desde seu nascimento em 2005, Git evoluiu e amadureceu para ser fácil de usar e ainda reter essas qualidades iniciais. Ele é incrivelmente rápido, é muito eficiente com projetos grandes, e ele tem um incrível sistema de ramos para desenvolvimento não linear (Veja Branches no Git).

O Básico do Git

Então, em poucas palavras, o que é o Git ? Esta é uma parte que é importante aprender, porque se você entender o que o Git é e os fundamentos de como ele funciona, em seguida, provavelmente usar efetivamente o Git será muito mais fácil para você. Enquanto você estiver aprendendo sobre o Git, tente esquecer das coisas que você pode saber sobre outros VCSs, como Subversion e Perforce; isso vai ajudá-lo a evitar a confusão sutil ao usar a ferramenta. O Git armazena e vê informações de forma muito diferente do que esses outros sistemas, mesmo que a interface do usuário seja bem semelhante, e entender essas diferenças o ajudará a não ficar confuso. (Perforce)

Imagens, Não Diferenças

A principal diferença entre o Git e qualquer outro VCS (Subversion e similares) é a maneira como o Git trata seus dados. Conceitualmente, a maioria dos outros sistemas armazenam informação como uma lista de mudanças nos arquivos. Estes sistemas (CVS, Subversion, Perforce, Bazaar, e assim por diante) tratam a informação como um conjunto de arquivos e as mudanças feitas em cada arquivo ao longo do tempo.

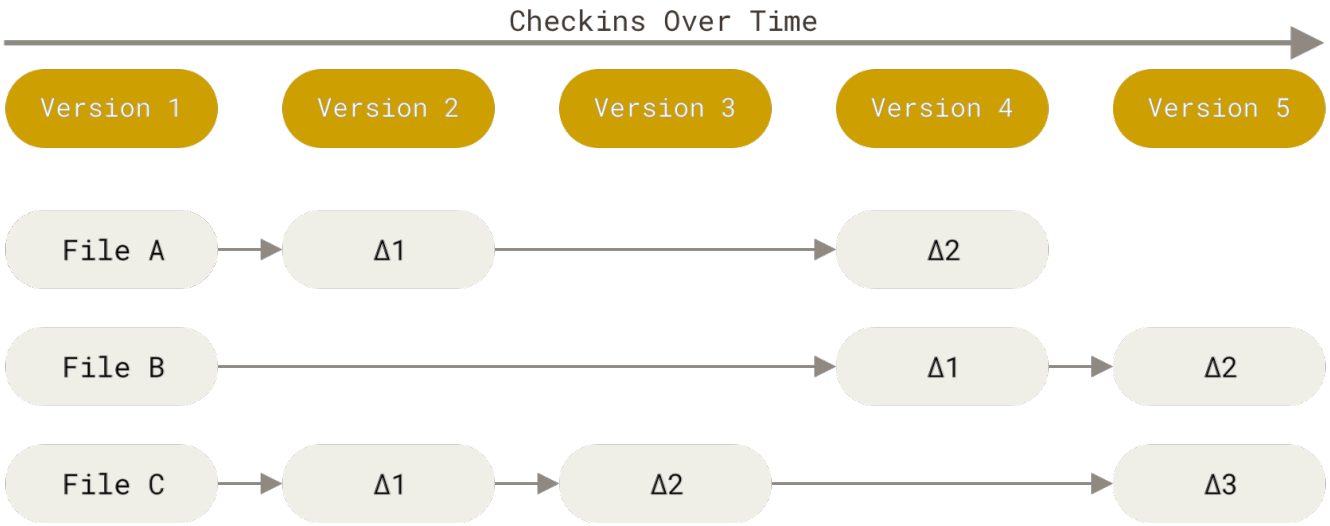


Figura 4. Armazenando dados como alterações em uma versão básica de cada arquivo.

O Git não trata nem armazena seus dados desta forma. Em vez disso, o Git trata seus dados mais como um conjunto de imagens de um sistema de arquivos em miniatura. Toda vez que você fizer um *commit*, ou salvar o estado de seu projeto no Git, ele basicamente tira uma foto de todos os seus arquivos e armazena uma referência para esse conjunto de arquivos. Para ser eficiente, se os arquivos não foram alterados, o Git não armazena o arquivo novamente, apenas um link para o arquivo idêntico anterior já armazenado. O Git trata seus dados mais como um **fluxo do estado dos arquivos**.

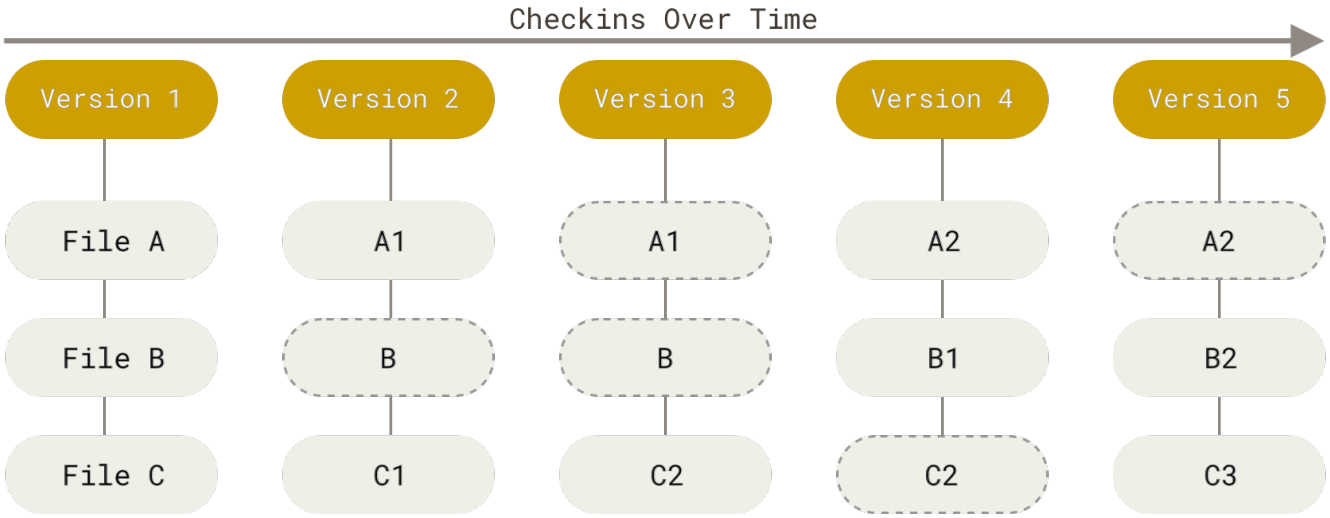


Figura 5. Armazenando dados como um estado do conjunto de arquivos do projeto ao longo do tempo.

Esta é uma diferença importante entre o Git e quase todos os outros VCSs. Isto faz o Git reconsiderar quase todos os aspectos de controle de versão que a maioria dos outros sistemas copiaram da geração anterior. Isso faz com que o Git seja mais como um mini sistema de arquivos com algumas ferramentas incrivelmente poderosas, ao invés de simplesmente um VCS. Vamos explorar alguns dos benefícios que você ganha ao tratar seus dados desta forma quando cobrirmos ramificações no Git Branches no Git.

Quase Todas as Operações são Locais

A maioria das operações no Git só precisa de arquivos e recursos locais para operar - geralmente nenhuma informação é necessária de outro computador da rede. Se você estiver acostumado com um CVCS onde a maioria das operações têm aquela demora causada pela latência da rede, este

aspecto do Git vai fazer você pensar que os deuses da velocidade abençoaram o Git com poderes extraterrestres. Como você tem toda a história do projeto ali mesmo em seu disco local, a maioria das operações parecem quase instantâneas.

Por exemplo, para pesquisar o histórico do projeto, o Git não precisa sair para o servidor para obter a história e exibi-lo para você - ele simplesmente lê diretamente do seu banco de dados local. Isto significa que você vê o histórico do projeto quase que instantaneamente. Se você quiser ver as alterações introduzidas entre a versão atual de um arquivo e o arquivo de um mês atrás, o Git pode procurar o arquivo de um mês atrás e fazer um cálculo de diferença local, em vez de ter que quer pedir a um servidor remoto para fazê-lo ou puxar uma versão mais antiga do arquivo do servidor remoto para fazê-lo localmente.

Isto também significa que há muito pouco que você não pode fazer se você estiver desconectado ou sem VPN. Se você estiver em um avião ou um trem e quiser trabalhar um pouco, você pode fazer *commits* alegremente até conseguir conexão de rede e enviar os arquivos. Se você chegar em casa e não conseguir conectar ao VPN, você ainda poderá trabalhar. Em muitos outros sistemas, isso é impossível ou doloroso. No Perforce, por exemplo, você não pode fazer quase nada se você não estiver conectado ao servidor; e no Subversion e CVS, você pode editar os arquivos, mas não poderá enviar *commits* das alterações ao seu banco de dados (porque você não está conectado ao seu banco de dados). Isso pode não parecer muito, mas você poderá se surpreender com a grande diferença que isso pode fazer.

Git Tem Integridade

Tudo no Git passa por uma soma de verificações (*checksum*) antes de ser armazenado e é referenciado por esse *checksum*. Isto significa que é impossível mudar o conteúdo de qualquer arquivo ou pasta sem que Git saiba. Esta funcionalidade está incorporada no Git nos níveis mais baixos e é parte integrante de sua filosofia. Você não perderá informação durante a transferência e não receberá um arquivo corrompido sem que o Git seja capaz de detectar.

O mecanismo que o Git utiliza para esta soma de verificação é chamado um *hash* SHA-1. Esta é uma sequência de 40 caracteres composta de caracteres hexadecimais (0-9 e-f) e é calculada com base no conteúdo de uma estrutura de arquivo ou diretório no Git. Um *hash* SHA-1 é algo como o seguinte:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Você vai ver esses valores de *hash* em todo o lugar no Git porque ele os usa com frequência. Na verdade, o Git armazena tudo em seu banco de dados não pelo nome do arquivo, mas pelo valor de *hash* do seu conteúdo.

O Git Geralmente Somente Adiciona Dados

Quando você faz algo no Git, quase sempre dados são adicionados no banco de dados do Git - e não removidos. É difícil fazer algo no sistema que não seja reversível ou fazê-lo apagar dados de forma alguma. Como em qualquer VCS, você pode perder alterações que ainda não tenham sido adicionadas em um *commit*; mas depois de fazer o *commit* no Git do estado atual das alterações, é muito difícil que haja alguma perda, especialmente se você enviar regularmente o seu banco de dados para outro repositório.

Isso faz com que o uso do Git seja somente alegria, porque sabemos que podemos experimentar sem o perigo de estragar algo. Para um olhar mais aprofundado de como o Git armazena seus dados e como você pode recuperar dados que parecem perdidos, consulte Desfazendo coisas.

Os Três Estados

Agora, preste atenção. Esta é a principal coisa a lembrar sobre Git se você quiser que o resto do seu processo de aprendizagem ocorra sem problemas. O Git tem três estados principais que seus arquivos podem estar: *committed*, modificado (*modified*) e preparado (*staged*). *Committed* significa que os dados estão armazenados de forma segura em seu banco de dados local. Modificado significa que você alterou o arquivo, mas ainda não fez o *commit* no seu banco de dados. Preparado significa que você marcou a versão atual de um arquivo modificado para fazer parte de seu próximo *commit*.

Isso nos leva a três seções principais de um projeto Git: o diretório Git, o diretório de trabalho e área de preparo.

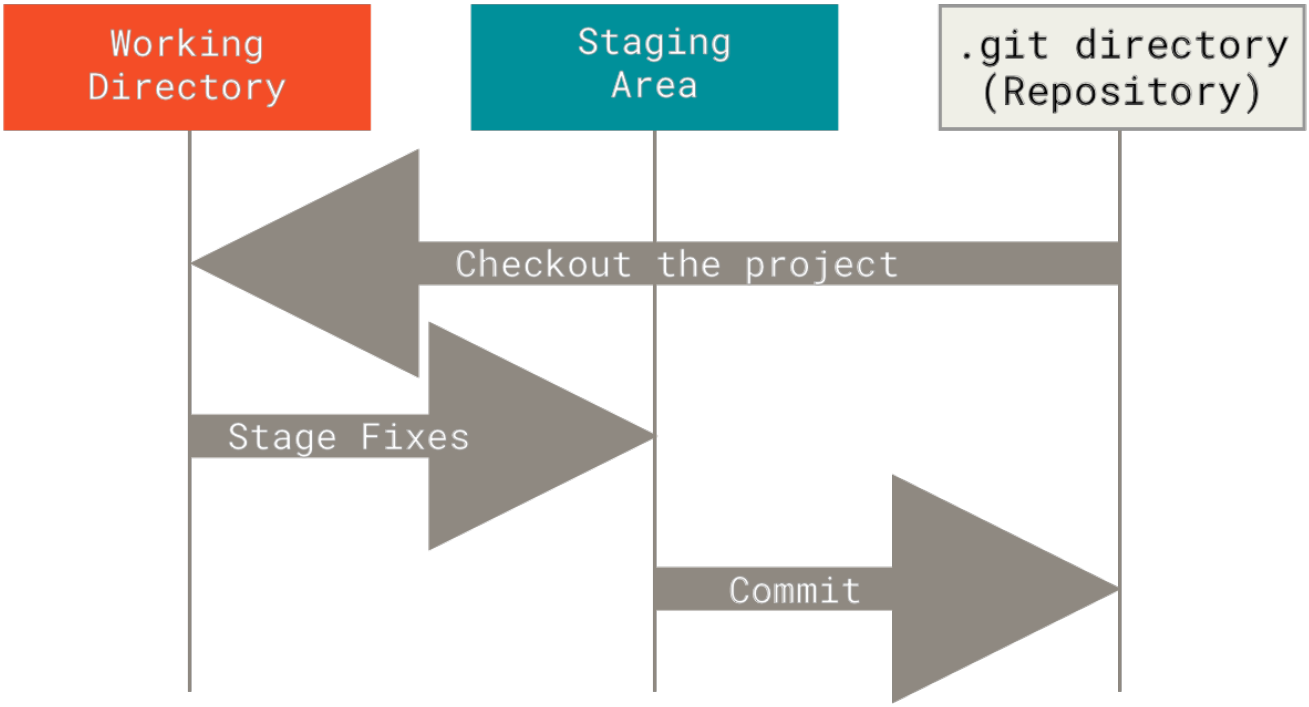


Figura 6. Diretório de trabalho, área de preparo, e o diretório Git.

O diretório Git é onde o Git armazena os metadados e o banco de dados de objetos de seu projeto. Esta é a parte mais importante do Git, e é o que é copiado quando você clona um repositório de outro computador.

O diretório de trabalho é uma simples cópia de uma versão do projeto. Esses arquivos são pegos do banco de dados compactado no diretório Git e colocados no disco para você usar ou modificar.

A área de preparo é um arquivo, geralmente contido em seu diretório Git, que armazena informações sobre o que vai entrar em seu próximo *commit*. É por vezes referido como o "índice", mas também é comum referir-se a ele como área de preparo (*staging area*).

O fluxo de trabalho básico Git é algo assim:

1. Você modifica arquivos no seu diretório de trabalho.
2. Você prepara os arquivos, adicionando imagens deles à sua área de preparo.
3. Você faz *commit*, o que leva os arquivos como eles estão na área de preparo e armazena essa imagens de forma permanente para o diretório do Git.

Se uma versão específica de um arquivo está no diretório Git, é considerado *commited*. Se for modificado, mas foi adicionado à área de preparo, é considerado preparado. E se ele for alterado depois de ter sido carregado, mas não foi preparado, ele é considerado modificado. Em Fundamentos de Git, você vai aprender mais sobre esses estados e como você pode tirar proveito deles ou pular a parte de preparação inteiramente.

A Linha de Comando

Existem várias formas diferentes de usar o Git. Existem as ferramentas originais de linha de comando, e existem várias interfaces gráficas de usuário (GUI - Graphical User Interface) com opções variadas. Para este livro, nós usaremos o Git na linha de comando. A linha de comando é o único lugar onde você pode rodar **todos** os comandos do Git - a maioria das GUI implementa somente um subconjunto das funcionalidades do Git. Se você sabe como usar o Git na linha de comando, você provavelmente descobrirá como rodar versões GUI, enquanto o oposto não é necessariamente verdade. Além disso, enquanto a sua escolha da interface gráfica é uma questão de gosto pessoal, *todos* os usuário terão as ferramentas de linha de comando instaladas e disponíveis.

Então, nós esperamos que você saiba como abrir um Terminal no Mac, ou Linha de Comando ou Powershell no Windows. Se você não sabe do que estamos falando, talvez você precise parar e pesquisar isso antes de continuar, para poder seguir os exemplos descritos neste livro.

Instalando o Git

Antes de começar a usar o Git, você tem que torná-lo disponível em seu computador. Mesmo se ele já tiver sido instalado, é provavelmente uma boa idéia atualizar para a versão mais recente. Você pode instalá-lo como um pacote ou através de outro instalador, ou baixar o código fonte e compilá-lo.

NOTA	Este livro foi escrito usando a versão **2.0.0** do Git. Embora a maioria dos comandos usados deve funcionar mesmo em versões antigas do Git, alguns deles podem não funcionar, ou podem agir de forma ligeiramente diferente se você estiver usando uma versão mais antiga. Como o Git é excelente para preservar compatibilidade com versões anteriores, qualquer versão após 2.0 deve funcionar muito bem.

Instalando no Linux

Se você deseja instalar o Git no Linux através de um instalador binário, você pode geralmente fazê-lo através da ferramenta básica de gerenciamento de pacotes que vem com sua distribuição. Se você usar Fedora por exemplo, você pode usar o yum:

```
$ sudo yum install git-all
```

Se você usar uma distribuição baseada em Debian como o Ubuntu, use o apt-get:

```
$ sudo apt-get install git-all
```

Para mais opções de instruções de como instalar o Git em outros vários sistemas Unix, veja na página do Git, em http://git-scm.com/download/linux.

Instalando no Mac

Existem várias maneiras de instalar o Git em um Mac. O mais fácil é provavelmente instalar as ferramentas de linha de comando Xcode. No Mavericks (10,9) ou acima, você pode fazer isso simplesmente rodando *git* a partir do Terminal pela primeira vez. Se você não tiver o Git instalado, ele irá pedir-lhe para instalá-lo.

Se você quiser uma versão mais atualizada, você também pode instalá-lo através de um instalador binário. Um instalador OSX Git é mantido e disponível para download no site do Git, pelo http://git-scm.com/download/mac.

Figura 7. Instalador do Git no OS X.

Você também pode instalá-lo como parte do instalador GitHub para Mac. Sua ferramenta GUI Git

tem uma opção para instalar as ferramentas de linha de comando. Você pode baixar essa ferramenta a partir da página GitHub para Mac, em http://mac.github.com.

Instalando no Windows

Há também algumas maneiras de instalar o Git no Windows. A compilação mais oficial está disponível para download no site do Git. Basta ir ao http://git-scm.com/download/win e o download começará automaticamente. Note que este é um projeto chamado Git para Windows (também chamado msysGit), que é algo separado do Git; para mais informações sobre isso, vá para http://msysgit.github.io/.

Para fazer uma instalação automatizada, você pode usar o pacote Git do Chocolatey. Note que o pacote Chocolatey é mantido pela comunidade.

Outra forma fácil de obter Git instalada é através da instalação de GitHub para Windows. O instalador inclui uma versão de linha de comando do Git, bem como a GUI. Ele também funciona bem com o PowerShell, e configura o cache de credenciais sólidas e as devidas configurações CRLF. Vamos saber mais sobre isso um pouco mais tarde, por enquanto basta dizer que estas são coisas que você deveria ter. Você pode baixá-lo da página GitHub para Windows, em http://windows.github.com.

Instalando da Fonte

Algumas pessoas podem achar interessante instalar Git a partir da fonte, para ter a versão mais recente. Os instaladores binários tendem a ficar um pouco atrás, embora após o Git ter amadurecido nos últimos anos, isso faz cada vez menos diferença.

Se você deseja instalar o Git a partir da fonte, você precisa ter as seguintes bibliotecas das quais o Git: curl, zlib, openssl, expat, e libiconv. Por exemplo, se você estiver em um sistema que tem yum (como o Fedora) ou apt-get (tal como um sistema baseado em Debian), você pode usar um destes comandos para instalar as dependências mínimas para compilar e instalar os binários do Git:

source,console]

```
$ sudo yum install dh-autoreconf curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
  gettext libz-dev libssl-dev
```

Para incluir a documentação em vários formatos (doc, html, info), essas dependências adicionais são necessárias:

```
$ sudo yum install asciidoc xmlto docbook2X getopt
$ sudo apt-get install asciidoc xmlto docbook2x getopt
```

Além disso, se estiver usando o Fedora/RHEL/derivados-do-RHEL, vai precisar executar

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

devido a diferenças nos nomes dos binários.

Quando você tiver todas as dependências necessárias, você poderá baixar o tarball com a última versão de vários lugares. Você pode obtê-lo através da página Kernel.org, em https://www.kernel.org/pub/software/scm/git, ou no espelho no site do GitHub, em https://github.com/git/git/releases. Em geral, é um pouco mais claro qual é a versão mais recente na página do GitHub, mas a página kernel.org também tem assinaturas se você quiser verificar o seu download.

Então, compile e instale:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Depois de ter feito isso, você poderá atualizar o Git através dele mesmo:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Configuração Inicial do Git

Agora que você tem o Git em seu sistema, você deve fazer algumas coisas para personalizar o ambiente Git. Você fará isso apenas uma vez por computador e o efeito se manterá após atualizações. Você também pode mudá-las em qualquer momento rodando esses comandos novamente.

O Git vem com uma ferramenta chamada `git config` que permite ver e atribuir variáveis de configuração que controlam todos os aspectos de como o Git aparece e opera. Estas variáveis podem ser armazenadas em três lugares diferentes:

1. `/etc/gitconfig`: válido para todos os usuários no sistema e todos os seus repositórios. Se você passar a opção `--system` para `git config`, ele lê e escreve neste arquivo.
2. `~/.gitconfig` ou `~/.config/git/config`: Somente para o seu usuário. Você pode fazer o Git ler e escrever neste arquivo passando a opção `--global`.
3. `config` no diretório Git (ou seja, `.git/config`) de qualquer repositório que você esteja usando: específico para este repositório.

Cada nível sobrescreve os valores no nível anterior, ou seja, valores em `.git/config` prevalecem sobre `/etc/gitconfig`.

No Windows, Git procura pelo arquivo `.gitconfig` no diretório `$HOME` (`C:\Users\$USER` para a maioria). Ele também procura por `/etc/gitconfig`, mesmo sendo relativo à raiz do sistema, que é onde quer que você tenha instalado Git no seu sistema.

Sua Identidade

A primeira coisa que você deve fazer ao instalar Git é configurar seu nome de usuário e endereço de e-mail. Isto é importante porque cada *commit* usa esta informação, e ela é carimbada de forma imutável nos *commits* que você começa a criar:

```
$ git config --global user.name "Fulano de Tal"
$ git config --global user.email fulanodetal@exemplo.br
```

Reiterando, você precisará fazer isso somente uma vez se tiver usado a opção `--global`, porque então o Git usará esta informação para qualquer coisa que você fizer naquele sistema. Se você quiser substituir essa informação com nome diferente para um projeto específico, você pode rodar o comando sem a opção `--global` dentro daquele projeto.

Muitas ferramentas GUI o ajudarão com isso quando forem usadas pela primeira vez.

Seu Editor

Agora que a sua identidade está configurada, você pode escolher o editor de texto padrão que será chamado quando Git precisar que você entre uma mensagem. Se não for configurado, o Git usará o editor padrão, que normalmente é o Vim. Se você quiser usar um editor de texto diferente, como o Emacs, você pode fazer o seguinte:

```
$ git config --global core.editor emacs
```

AVISO	Vim e Emacs são editores de texto populares comumente usados por desenvolvedores em sistemas baseados em Unix como Linux e Max. Se você não for acostumado com estes editores ou estiver em um sistema Windows, você precisará procurar por instruções de como configurar o seu editor preferido com Git. Se você não configurar o seu editor preferido e não sabe usar o Vim ou Emacs, é provável que você fique bastante confuso ao entrar neles.

Testando Suas Configurações

Se você quiser testar as suas configurações, você pode usar o comando `git config --list` para listar todas as configurações que o Git conseguir encontrar naquele momento:

```
$ git config --list
user.name=Fulano de Tal
user.email=fulanodetal@exemplo.br
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Pode ser que algumas palavras chave apareçam mais de uma vez, porque Git lê as mesmas chaves de arquivos diferentes (`/etc/gitconfig` e `~/.gitconfig`, por exemplo). Neste caso, Git usa o último valor para cada chave única que ele vê.

Você pode também testar o que Git tem em uma chave específica digitando `git config <key>`:

```
$ git config user.name
Fulano de Tal
```

Pedindo Ajuda

Se você precisar de ajuda para usar o Git, há três formas de acessar a página do manual de ajuda (*manpage*) para qualquer um dos comandos Git:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Por exemplo, você pode ver a *manpage* do commando config rodando

```
$ git help config
```

Estes comandos podem ser acessados de qualquer lugar, mesmo desconectado. Se as *manpages* e este livro não forem suficientes e você precisar de ajuda personalizada, você pode tentar os canais `#git` ou `#github` no servidor IRC Freenode (irc.freenode.net). Estes canais estão sempre cheios com centenas de pessoas que são bem versadas com o Git e dispostas a ajudar.

Sumário

Você deve ter um entendimento básico do que o Git é e como ele é diferente de sistemas de controle de versão centralizados que você talvez tenha usado anteriormente. Você já deve ter também uma versão do Git funcionando na sua máquina que está configurada com o seu toque pessoal. Agora é hora de começar a aprender o básico sobre Git.

Fundamentos de Git

Se você pode ler apenas um capítulo antes de começar a usar o Git, este é ele. Este capítulo cobre cada comando básico que você precisa para fazer a maior parte das coisas com as quais eventualmente você vai se deparar durante seu uso do Git. No final deste capítulo, você será capaz de configurar e inicializar um repositório, iniciar e interromper o rastreamento de arquivos, usar a área de *stage* e realizar *commits* das alterações. Também mostraremos como configurar o Git para ignorar certos arquivos e padrões de arquivo, como desfazer erros de maneira rápida e fácil, como navegar no histórico do seu projeto e visualizar alterações entre *commits* e como fazer *push* e *pull* em repositórios remotos.

Obtendo um Repositório Git

Você pode obter um projeto Git utilizando duas formas principais. 1. Você pode pegar um diretório local que atualmente não está sob controle de versão e transformá-lo em um repositório Git, ou 2. Você pode fazer um *clone* de um repositório Git existente em outro lugar.

Inicializando um Repositório em um Diretório Existente

Para você começar a monitorar um projeto existente com Git, você deve ir para o diretório desse projeto. Se você nunca fez isso, use o comando a seguir, que terá uma pequena diferença dependendo do sistema em que está executando:

para Linux:

```
$ cd /home/user/your_repository
```

para Mac:

```
$ cd /Users/user/your_repository
```

para Windows:

```
$ cd /c/user/your_repository
```

depois digite:

```
$ git init
```

Isso cria um novo subdiretório chamado `.git` que contém todos os arquivos necessários de seu repositório – um esqueleto de repositório Git. Neste ponto, nada em seu projeto é monitorado ainda. (Veja Funcionamento Interno do Git para mais informações sobre quais arquivos estão contidos no diretório `.git` que foi criado.)

Se você quer começar a controlar o versionamento dos arquivos existentes (ao contrário de um diretório vazio), você provavelmente deve começar a monitorar esses arquivos e fazer um *commit* inicial. Você pode fazer isso com alguns comandos `git add` que especificam os arquivos que você quer monitorar, seguido de um `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

Nós já veremos o que esses comandos fazem. Mas neste ponto você já tem um repositório Git com arquivos monitorados e um *commit* inicial.

Clonando um Repositório Existente

Caso você queira obter a cópia de um repositório Git existente – por exemplo, um projeto que você queira contribuir – o comando para isso é `git clone`. Se você estiver familiarizado com outros sistemas VCS, tal como Subversion, você vai notar que o comando é `clone` e não `checkout`. Essa é uma diferença importante – em vez de receber apenas uma cópia para trabalho, o Git recebe uma cópia completa de praticamente todos os dados que o servidor possui. Cada versão de cada arquivo no histórico do projeto é obtida por padrão quando você executa `git clone`. De fato, se o disco do servidor ficar corrompido, em geral você pode usar qualquer uma das cópias de qualquer um dos clientes para reverter o servidor ao estado em que estava quando foi clonado (talvez você perca algumas configurações do servidor, mas todos os dados versionados estarão lá — veja Getting Git on a Server para mais detalhes).

Você clona um repositório com `git clone [url]`. Por exemplo, caso você queria clonar a biblioteca Git Linkable chamada libgit2, você pode fazer da seguinte forma:

```
$ git clone https://github.com/libgit2/libgit2
```

Isso cria um diretório chamado `libgit2`, inicializa um diretório `.git` dentro dele, recebe todos os dados deste repositório e deixa disponível para trabalho a cópia da última versão. Se você entrar no novo diretório `libgit2`, você verá os arquivos do projeto nele, pronto para serem editados ou utilizados. Caso você queira clonar o repositório em um diretório diferente de `libgit2`, é possível especificar esse diretório utilizando a opção abaixo:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Este comando faz exatamente a mesma coisa que o anterior, mas o diretório de destino será chamado `mylibgit`.

O Git possui diversos protocolos de transferência que você pode utilizar. O exemplo anterior usa o protocolo `https://`, mas você também pode ver `git://` ou `user@server:path/to/repo.git`, que usam o protocolo de transferência SSH. Em Getting Git on a Server é apresentado todas as opções disponíveis com as quais o servidor pode ser configurado para acessar o seu repositório Git, e os prós e contras de cada uma.

Gravando Alterações em Seu Repositório

Você tem um verdadeiro repositório Git e um "checkout" ou cópia de trabalho dos arquivos para aquele projeto. Você precisa fazer algumas alterações e adicionar commits dessas alterações em seu repositório a cada vez que o projeto chegar a um estado que você queira registrar.

Lembre-se que cada arquivo em seu diretório de trabalho pode estar em um dos seguintes estados: rastreado e não-rastreado. Arquivos rastreados são arquivos que foram incluídos no último *snapshot*; eles podem ser não modificados, modificados ou preparados (adicionados ao *stage*). Em resumo, arquivos rastreados são os arquivos que o Git conhece.

Arquivos não rastreados são todos os outros - quaisquer arquivos em seu diretório de trabalho que não foram incluídos em seu último snapshot e não estão na área de stage. Quando você clona um repositório pela primeira vez, todos os seus arquivos serão rastreados e não modificados já que o Git acabou de obtê-los e você ainda não editou nada.

Assim que você edita alguns arquivos, Git os considera modificados, porque você os editou desde o seu último commit. Você prepara os arquivos editados e então faz commit das suas alterações, e o ciclo se repete.

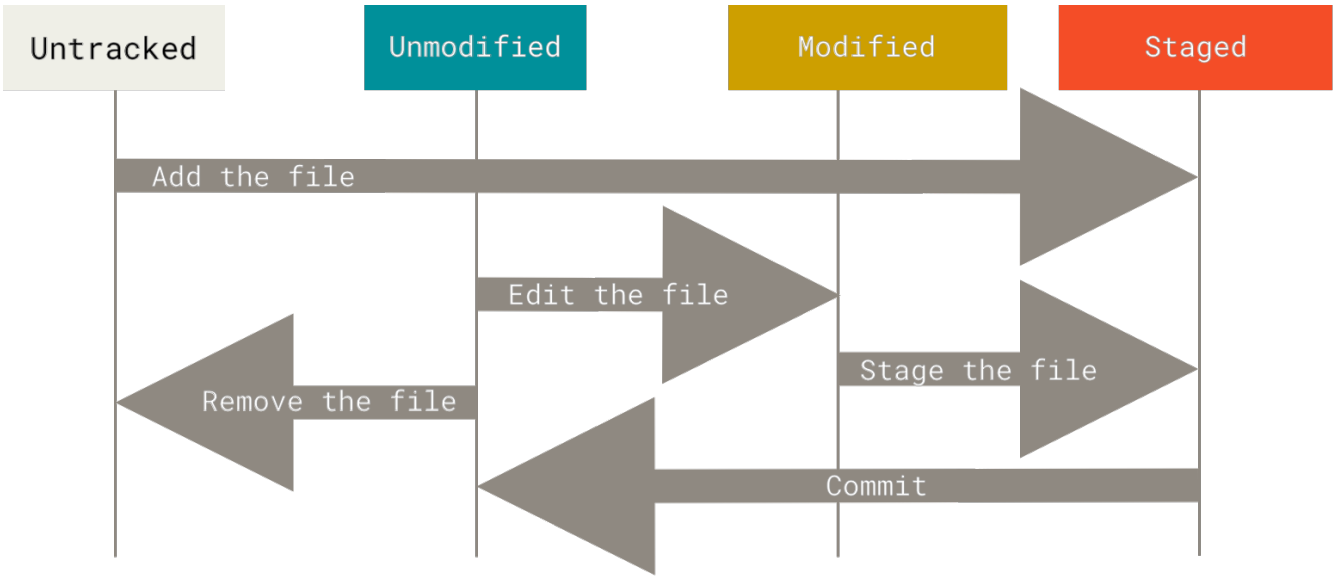


Figura 8. O ciclo de vida dos status de seus arquivos.

Verificando os Status de Seus Arquivos

A principal ferramenta que você vai usar para determinar quais arquivos estão em qual estado é o comando `git status`. Se você executar esse comando imediatamente após clonar um repositório, você vai ver algo assim:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Isso significa que você tem um diretório de trabalho limpo - em outras palavras, nenhum de seus

arquivos rastreados foi modificado. O Git também não está vendo nenhum arquivo não rastreado, senão eles estariam listados aqui. Finalmente, o comando lhe diz qual o branch que você está e diz que ele não divergiu do mesmo branch no servidor. Por enquanto, esse branch é sempre “master”, que é o padrão; você não precisa se preocupar com isso agora. Branches no Git vai abordar branches e referências em detalhe.

Digamos que você adiciona um novo arquivo no seu projeto, um simples arquivo README. Se o arquivo não existia antes, e você executar `git status`, você verá seu arquivo não rastreado da seguinte forma:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

Você pode ver que o seu novo arquivo README é um arquivo não rastreado, porque está abaixo do subtítulo “Untracked files” na saída do seu status. "Não rastreado" basicamente significa que o Git vê um arquivo que você não tinha no snapshot (commit) anterior; o Git não vai passar a incluir o arquivo nos seus commits a não ser que você o mande fazer isso explicitamente. O comportamento do Git é dessa forma para que você não inclua acidentalmente arquivos binários gerados automaticamente ou outros arquivos que você não deseja incluir. Você *quer* incluir o arquivo README, então vamos começaçar a rastreá-lo.

Rastreando Arquivos Novos

Para começar a rastrear um novo arquivo, você deve usar o comando `git add`. Para começar a rastrear o arquivo README, você deve executar o seguinte:

```
$ git add README
```

Executando o comando *status* novamente, você pode ver que seu README agora está sendo rastreado e preparado (*staged*) para o *commit*:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

É possível saber que o arquivo está preparado porque ele aparece sob o título “Changes to be committed”. Se você fizer um *commit* neste momento, a versão do arquivo que existia no instante em que você executou `git add`, é a que será armazenada no histórico de *snapshots*. Você deve se lembrar que, quando executou `git init` anteriormente, em seguida, você também executou `git add (arquivos)` - isso foi para começar a rastrear os arquivos em seu diretório. O comando `git add` recebe o caminho de um arquivo ou de um diretório. Se for um diretório, o comando adiciona todos os arquivos contidos nesse diretório recursivamente.

Preparando Arquivos Modificados

Vamos modificar um arquivo que já estava sendo rastreado. Se você modificar o arquivo `CONTRIBUTING.md`, que já era rastreado, e então executar `git status` novamente, você deve ver algo como:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

O arquivo `CONTRIBUTING.md` aparece sob a seção “Changes not staged for commit” — que indica que um arquivo rastreado foi modificado no diretório de trabalho mas ainda não foi mandado para o *stage* (preparado). Para mandá-lo para o *stage*, você precisa executar o comando `git add`. O `git add` é um comando de múltiplos propósitos: serve para começar a rastrear arquivos e também para outras coisas, como marcar arquivos que estão em conflito de mesclagem como resolvidos. Pode ser útil pensar nesse comando mais como “adicione este conteúdo ao próximo *commit*”. Vamos executar `git add` agora, para mandar o arquivo `CONTRIBUTING.md` para o *stage*, e então executar `git status` novamente:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Ambos os arquivos estão preparados (no *stage*) e entrarão no seu próximo *commit*. Neste momento, suponha que você se lembre de uma pequena mudança que quer fazer no arquivo `CONTRIBUTING.md` antes de fazer o *commit*. Você abre o arquivo, faz a mudança e está pronto para fazer o *commit*. No entanto, vamos executar `git status` mais uma vez:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Que negócio é esse? Agora o `CONTRIBUTING.md` está listado como preparado (*staged*) e também como não-preparado (*unstaged*). Como isso é possível? Acontece que o Git põe um arquivo no *stage* exatamente como ele está no momento em que você executa o comando `git add`. Se você executar `git commit` agora, a versão do `CONTRIBUTING.md` que vai para o repositório é aquela de quando você executou `git add`, não a versão que está no seu diretório de trabalho. Se você modificar um arquivo depois de executar `git add`, você tem que executar `git add` de novo para por sua versão mais recente no *stage*:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Status Curto

Ao mesmo tempo que a saída do `git status` é bem completa, ela também é bastante verbosa. O Git também tem uma *flag* para status curto, que permite que você veja suas alterações de forma mais compacta. Se você executar `git status -s` ou `git status --short` a saída do comando vai ser bem mais simples:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Arquivos novos que não são rastreados têm um `??` do lado, novos arquivos que foram adicionados à área de *stage* têm um `A`, arquivos modificados têm um `M` e assim por diante. Há duas colunas de status na saída: a coluna da esquerda indica o status da área de *stage* e a coluna da direita indica o status do diretório de trabalho. No exemplo anterior, o arquivo `README` foi modificado no diretório de trabalho mas ainda não foi para o *stage*, enquanto o arquivo `lib/simplegit.rb` foi modificado e foi para o *stage*. O arquivo `Rakefile` foi modificado, foi para o *stage* e foi modificado de novo, de maneira que há alterações para ele tanto no estado preparado quanto no estado não-preparado.

Ignorando Arquivos

Frequentemente você terá uma classe de arquivos que não quer que sejam adicionados automaticamente pelo Git e nem mesmo que ele os mostre como não-rastreados. Geralmente, esses arquivos são gerados automaticamente, tais como arquivos de *log* ou arquivos produzidos pelo seu sistema de compilação (*build*). Nesses casos, você pode criar um arquivo chamado `.gitignore`, contendo uma lista de padrões de nomes de arquivo que devem ser ignorados. Aqui está um exemplo de arquivo `.gitignore`:

```
$ cat .gitignore
*.[oa]
*~
```

A primeira linha diz ao Git para ignorar todos os arquivos que terminem com “.o” ou “.a” – arquivos objeto ou de arquivamento, que podem ser produtos do processo de compilação. A segunda linha diz ao Git para ignorar todos os arquivos cujo nome termine com um til (`~`), que é usado por muitos editores de texto, como o Emacs, para marcar arquivos temporários. Você também pode incluir diretórios `log`, `tmp` ou `pid`; documentação gerada automaticamente; e assim por diante. Configurar um arquivo `.gitignore`, antes de você começar um repositório, geralmente é uma boa ideia para que você não inclua acidentalmente em seu repositório Git arquivos que você não quer.

As regras para os padrões que podem ser usados no arquivo `.gitignore` são as seguintes:

- Linhas em branco ou começando com `#` são ignoradas.
- Os padrões que normalmente são usados para nomes de arquivos funcionam.
- Você pode iniciar padrões com uma barra (`/`) para evitar recursividade.
- Você pode terminar padrões com uma barra (`/`) para especificar um diretório.
- Você pode negar um padrão ao fazê-lo iniciar com um ponto de exclamação (`!`).

Padrões de nome de arquivo são como expressões regulares simplificadas usadas em ambiente

shell. Um asterisco (`*`) casa com zero ou mais caracteres; `[abc]` casa com qualquer caracter dentro dos colchetes (neste caso, a, b ou c); um ponto de interrogação (`?`) casa com um único caracter qualquer; e caracteres entre colchetes separados por hífen (`[0-9]`) casam com qualquer caracter entre eles (neste caso, de 0 a 9). Você também pode usar dois asteriscos para criar uma expressão que case com diretórios aninhados; `a/**/z` casaria com `a/z`, `a/b/z`, `a/b/c/z`, e assim por diante.

Aqui está outro exemplo de arquivo `.gitignore`:

```
# ignorar arquivos com extensão .a
*.a

# mas rastrear o arquivo lib.a, mesmo que você esteja ignorando os arquivos .a acima
!lib.a

# ignorar o arquivo TODO apenas no diretório atual, mas não em subdir/TODO
/TODO

# ignorar todos os arquivos no diretório build/
build/

# ignorar doc/notes.txt, mas não doc/server/arch.txt
doc/*.txt

# ignorar todos os arquivos .pdf no diretório doc/
doc/**/*.pdf
```

DICA

O GitHub mantém uma lista bem abrangente com bons exemplos de arquivo `.gitignore` para vários projetos e linguagens em https://github.com/github/gitignore, se você quiser um ponto de partida para o seu projeto.

NOTA

Em casos simples, um repositório deve ter um único arquivo `.gitignore` em seu diretório raiz, o qual é aplicado recursivamente a todo o repositório. Contudo, também é possível ter arquivos `.gitignore` adicionais em subdiretórios. As regras definidas nesses `.gitignore` internos se aplicam somente aos arquivos contidos no diretório em que eles estão localizados. (O repositório do kernel do Linux tem 206 arquivos `.gitignore`.)

Está fora do escopo deste livro explicar os detalhes de múltiplos arquivos `.gitignore`; veja `man gitignore` para mais informações.

Visualizando Suas Alterações Dentro e Fora do Stage

Se o comando `git status` for vago demais para você — você quer saber exatamente o que você alterou, não apenas quais arquivos foram alterados — você pode usar o comando `git diff`. Nós explicaremos o `git diff` em detalhes mais tarde, mas provavelmente você vai usá-lo com maior frequência para responder a essas duas perguntas: O que você alterou mas ainda não mandou para o *stage* (estado preparado)? E o que está no *stage*, pronto para o *commit*? Apesar de o `git status` responder a essas perguntas de forma genérica, listando os nomes dos arquivos, o `git diff` exibe

exatamente as linhas que foram adicionadas e removidas — o *patch*, como costumava se chamar.

Digamos que você altere o arquivo `README` e o mande para o *stage* e então altere o arquivo `CONTRIBUTING.md` sem mandá-lo para o *stage*. Se você executar o comando `git status`, você verá mais uma vez alguma coisa como o seguinte:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Para ver o que você alterou mas ainda não mandou para o *stage*, digite o comando `git diff` sem nenhum argumento:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Esse comando compara o que está no seu diretório de trabalho com o que está no *stage*. O resultado permite que você saiba quais alterações você fez que ainda não foram mandadas para o *stage*.

Se você quiser ver as alterações que você mandou para o *stage* e que entrarão no seu próximo *commit*, você pode usar `git diff --staged`. Este comando compara as alterações que estão no seu *stage* com o seu último *commit*:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

É importante notar que o `git diff` sozinho não mostra todas as alterações feitas desde o seu último *commit* — apenas as alterações que ainda não estão no *stage* (não-preparado). Isso pode ser confuso porque, se você já tiver mandado todas as suas alterações para o *stage*, a saída do `git diff` vai ser vazia.

Um outro exemplo: se você mandar o arquivo `CONTRIBUTING.md` para o *stage* e então alterá-lo, você pode usar o `git diff` para ver as alterações no arquivo que estão no *stage* e também as que não estão. Se o nosso ambiente se parecer com isso:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Agora você poderá usar o `git diff` para ver o que ainda não foi mandado para o *stage*:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects
list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

e `git diff --cached` para ver o que você já mandou para o *stage* até agora (`--staged` e `--cached` são sinônimos):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

NOTA

Git Diff em uma Ferramenta Externa

Nós continuaremos usando o comando `git diff` de várias maneiras pelo resto do livro. Há outra maneira de ver essas diferenças, se você preferir usar uma ferramenta gráfica ou um programa externo. Se você executar `git difftool` em vez de `git diff`, você pode ver qualquer dessas diferenças em um software como emerge, vimdiff e muitos outros (incluindo produtos comerciais). Execute `git difftool --tool-help` para ver o que há disponível em seu sistema.

Fazendo Commit das Suas Alterações

Agora que sua área de *stage* está preparada do jeito que você quer, você pode fazer *commit* das suas alterações. Lembre-se que qualquer coisa que ainda não foi enviada para o *stage* — qualquer arquivo que você tenha criado ou alterado e que ainda não tenha sido adicionado com `git add` — não entrará nesse *commit*. Esses arquivos permanecerão no seu disco como arquivos alterados. Nesse caso, digamos que, da última vez que você executou `git status`, você viu que tudo estava no *stage*, então você está pronto para fazer *commit* de suas alterações. O jeito mais simples de fazer *commit* é digitar `git commit`:

```
$ git commit
```

Fazendo isso, será aberto o editor de sua escolha.

NOTA

O editor é determinado pela variável de ambiente `EDITOR` — normalmente o vim ou emacs, mas você pode escolher qualquer editor que quiser usando o comando `git config --global core.editor` como você viu em Começando.

O editor mostra o seguinte texto (este é um exemplo da tela do Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Você pode ver que a mensagem de *commit* padrão contém a saída mais recente do comando `git status`, comentada, e uma linha em branco no topo. Você pode remover esses comentários e digitar sua mensagem de *commit*, ou você pode deixá-los lá para ajudá-lo a lembrar o que faz parte do *commit*.

NOTA	Para um lembrete ainda mais explícito do que você alterou, você pode passar a opção `-v` para o `git commit`. Isso inclui as diferenças (*diff*) da sua alteração no editor, para que você possa ver exatamente quais alterações estão entrando no *commit*.

Quando você sair do editor, o Git criará seu *commit* com essa mensagem (com os comentários e diferenças removidos).

Alternativamente, você pode digitar sua mensagem de *commit* diretamente na linha de comando, depois da opção `-m` do comando `commit`, assim:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Você acaba de criar seu primeiro *commit*! Veja que a saída do comando fornece algumas informações: em qual *branch* foi feito o *commit* (`master`), seu *checksum* SHA-1 (`463dc4f`), quantos arquivos foram alterados e estatísticas sobre o número de linhas adicionadas e removidas.

Lembre-se de que o *commit* grava o *snapshot* que você deixou na área de *stage*. Qualquer alteração que você não tiver mandado para o *stage* permanecerá como estava, em seu lugar; você pode executar outro *commit* para adicioná-la ao seu histórico. Toda vez que você executa um *commit*, você está gravando um *snapshot* do seu projeto que você pode usar posteriormente para fazer comparações, ou mesmo restaurá-lo.

Pulando a Área de Stage

Mesmo sendo incrivelmente útil para preparar *commits* exatamente como você quer, a área de

stage algumas vezes é um pouco mais complexa do que o necessário. Se você quiser pular a área de *stage*, o Git fornece um atalho simples. A opção `-a`, do comando `git commit`, faz o Git mandar todos arquivos rastreados para o *stage* automaticamente, antes de fazer o *commit*, permitindo que você pule a parte do `git add`:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Perceba que, nesse caso, você não tem que executar `git add` antes, para adicionar o arquivo `CONTRIBUTING.md` ao *commit*. Isso ocorre porque a opção `-a` inclui todos os arquivos alterados. Isso é conveniente, mas cuidado; algumas vezes esta opção fará você incluir alterações indesejadas.

Removendo Arquivos

Para remover um arquivo do Git, você tem que removê-lo dos seus arquivos rastreados (mais precisamente, removê-lo da sua área de *stage*) e então fazer um *commit*. O comando `git rm` faz isso, e também remove o arquivo do seu diretório de trabalho para que você não o veja como um arquivo não-rastreado nas suas próximas interações.

Se você simplesmente remover o arquivo do seu diretório, ele aparecerá sob a área “Changes not staged for commit” (isto é, fora do *stage*) da saída do seu `git status`:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Mas, se você executar `git rm`, o arquivo será preparado para remoção (retirado do *stage*):

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

Da próxima vez que você fizer um *commit*, o arquivo será eliminado e não será mais rastreado. Se o arquivo tiver sido alterado ou se já tiver adicionado à área de *stage*, você terá que forçar a remoção com a opção `-f`. Essa é uma medida de segurança para prevenir a exclusão acidental de dados que ainda não tenham sido gravados em um *snapshot* e que não poderão ser recuperados do histórico.

Outra coisa útil que você pode querer fazer é manter o arquivo no seu diretório de trabalho, mas removê-lo da sua área de *stage*. Em outras palavras, você pode querer manter o arquivo no seu disco rígido, mas não deixá-lo mais sob controle do Git. Isso é particularmente útil se você esquecer de adicionar alguma coisa ao seu arquivo `.gitignore` e, acidentalmente, mandá-la para o *stage*, como um grande arquivo de *log* ou um monte de arquivos compilados `.a`. Para fazer isso, use a opção `--cached`:

```
$ git rm --cached README
```

Você pode passar arquivos, diretórios e padrões de nomes para o comando `git rm`. Isso quer dizer que você pode fazer coisas como:

```
$ git rm log/\*.log
```

Note a barra invertida (`\`) na frente do `*`. Isso é necessário porque o Git faz sua própria expansão de nomes de arquivos em adição a que é feita pela sua *shell*. Esse comando remove todos os arquivos que tenham a extensão `.log` do diretório `log/`. Ou, você pode fazer algo como o seguinte:

```
$ git rm \*~
```

Esse comando remove todos os arquivos cujos nomes terminem com um `~`.

Movendo Arquivos

Diferentemente de outros sistemas de controle de versão, o Git não rastreia explicitamente a movimentação de arquivos. Se você renomear um arquivo no Git, ele não armazena metadados indicando que determinado arquivo foi renomeado. Porém, o Git é bastante esperto para perceber isso depois do fato ocorrido — nós trataremos de movimentação de arquivos daqui a pouco.

Assim, é um pouco confuso o fato de o Git ter um comando `mv`. Se você quiser renomear um arquivo

no Git, você pode executar alguma coisa como:

```
$ git mv arq_origem arq_destino
```

e vai funcionar bem. Na verdade, se você executar alguma coisa assim e verificar o *status*, você vai ver que o Git considera que arquivo foi renomeado:

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Contudo, isso é equivalente a executar algo como:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

O Git percebe que, implicitamente, se trata de um arquivo renomeado, de maneira que não importa se você renomear um arquivo desse jeito ou com o comando `mv`. A única diferença real é que `git mv` é um comando em vez de três — é uma função de conveniência. Mais importante, você pode usar qualquer ferramenta que quiser para renomear um arquivo e cuidar do `add`/`rm` depois, antes de fazer o *commit*.

Vendo o histórico de Commits

Depois de você ter criado vários commits ou se você clonou um repositório com um histórico de commits pré-existente, você vai provavelmente querer olhar para trás e ver o que aconteceu. A ferramenta mais básica e poderosa para fazer isso é o comando `git log`.

Esses exemplos usam um projeto muito simples chamando “simplegit”. Para conseguir o projeto, execute

```
$ git clone https://github.com/schacon/simplegit-progit
```

Quando você executa `git log` neste projeto, você deve receber um retorno que se parece com algo assim:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Por padrão, sem argumentos, `git log` lista os commits feitos neste repositório em ordem cronológica inversa; isto é, o commit mais recente aparece primeiro. Como você pode ver, esse comando lista cada commit com o seu checksum SHA-1, o nome e email do autor, data de inserção, e a mensagem do commit.

Está disponível um enorme número e variedade de opções para o comando `git log` a fim de lhe mostrar exatamente aquilo pelo que está procurando. Aqui, vamos mostrar a você algumas das mais populares.

Uma das opções que mais ajuda é `-p`, que mostra as diferenças introduzidas em cada commit. Você pode também usar `-2`, que lista no retorno apenas os dois últimos itens:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

Essa opção mostrar a mesma informação mas com um diff diretamente após cada item. Isso é de muita ajuda para revisão de código ou para rapidamente procurar o que aconteceu durante uma série de commits que uma colaborador tenha adicionado. Você pode também usar uma série de opções resumidas com o `git log`. Por exemplo, se você quer ver algumas estatísticas abreviadas para cada commit, você pode usar a opção `--stat`:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

Como você pode ver, a opção `--stat` apresenta abaixo de cada commit uma lista dos arquivos modificados, quantos arquivos foram modificados, e quantas linhas nestes arquivos foram adicionadas e removidas. Por último ela também colocar um resumo das informações.

Uma outra opção realmente muito util é `--pretty`. Essa opção modifica os registros retornados para formar outro formato diferente do padrão. Algumas opções pré-definidas estão disponíveis para você usar. A opção `oneline` mostra cada commit em uma única linha, esta é de muita ajuda se você está olhando para muitos commits. Em adição, as opções `short`, `full`, e `fuller` apresentam o retorno quase no mesmo formato porem com menos ou mais informações, respectivamente:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

A opção mais interessante é `format`, a qual permite a você especificar seu próprio formato de registros de retorno. Isso é especialmente útil quando você esta gerando um retorno para uma máquina analisar – pois você especifica o formato explicitamente, você sabe que isso não irá mudar

com as atualizações do Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Useful options for `git log --pretty=format` lista algumas das opções mais uteis que `format` gera.

Tabela 1. Useful options for `git log --pretty=format`

Opção	Descrição da saída
`%H`	Hash do commit
`%h`	Hash do commit abreviado
`%T`	Hash da árvore
`%t`	Hash da árvore abreviado
`%P`	Hashes dos pais
`%p`	Hashes dos pais abreviado
`%an`	Nome do Autor
`%ae`	Email do Autor
`%ad`	Data do Autor (o formato segue a opção --date=option)
`%ar`	Data do Autor, relativa
`%cn`	Nome do Committer
`%ce`	Email do Committer
`%cd`	Data do Committer
`%cr`	Data do Committer, relativa
`%s`	Comentário

Você talvez esteja imaginando qual a diferença entre *author* e *committer*. O autor é a pessoa que escreveu originalmente o trabalho, ao passo que a pessoa que submeteu o trabalho é o committer. Então, se você criar uma correção para um projeto e um dos membros principais submete a correção, ambos receberão crédito – você como autor, e o membro principal como commiter. Nós vamos abordar esta distinção um pouco mais em Distributed Git.

As opções `oneline` e `format` são particularmente úteis juntamente com uma outra opção de `log` chamada `--graph`. Esta opção adiciona um pequeno gráfico ASCII mostrando seu histórico de branch e merge:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

Esse tipo de retorno se tornará mais interessante conforme formos criando branches e merges no próximo capitulo.

Essas são apenas algumas opções simples de formatações de retorno para `git log` – existem muitas mais. Common options to `git log` lista as opções que nós já abordamos, assim como algumas outras opções de formatação mais comuns que talvez sejam muito úteis, juntamente com o como ela mudam o retorno do comando log.

Tabela 2. Common options to `git log`

Opções	Descrição
`-p`	Mostra o patch introduzido com cada commit.
`--stat`	Mostra estatísticas de arquivos modificados em cada commit.
`--shortstat`	Exibe apenas a linha informando a alteração, inserção e exclusão do comando --stat.
`--name-only`	Mostra a lista de arquivos modificados após as informações de commit.
`--name-status`	Mostra também a lista de arquivos que sofreram modificação com informações adicionadas / modificadas / excluídas.
`--abbrev-commit`	Mostra apenas os primeiros caracteres da soma de verificação SHA-1 em vez de todos os 40.
`--relative-date`	Exibe a data em um formato relativo (por exemplo, ‘` 2 semanas atrás ’’) em vez de usar o formato de data completo.
`--graph`	Exibe um gráfico ASCII do histórico de branches e merges ao lado da saída do log.
`--pretty`	Mostra os commits em um formato alternativo. As opções incluem oneline, short, full, fuller e format (onde você especifica seu próprio formato).

Limitando o retorno do comando Log

Em adição ás opções de formatação do retorno, `git log` leva um número útil de opções de limitação – que são, opções que lhe permitem mostrar apenas um subconjunto de commits. Você já viu essa opção antes – a opção `-2`, a qual mostra apenas os dois últimos commits. De fato, você pode usar `-<n>`, onde `n` é qualquer número inteiro para mostra os ultimos `n` commits. Na verdade, você não gostará de usar isso frequentemente, pois o Git por padrão enquadra todo o retorno através de uma

página então você vê apenas uma página de registros por vez.

Entretanto, as opções de lina do tempo tais como `--since` e `--until` são muito uteis. Por exemplo, esse comando retorna a lista de commits feitos nas últimas duas semanas:

```
$ git log --since=2.weeks
```

Esse comando funciona com um grande número de formatos – você pode determinar uma data específica como `"2008-01-15"`, ou uma data relativa tal como `"2 anos 1 dia 3 minutos atrás"`.

Você pode também filtrar a lista de commits que combinam com algum critério de busca. A opção `--author` permite você filtrar por um autor especifico, e a opção `--grep` permite você procurar por palavras chaves na mensagem do commit. (Note que se você quer especificar ambas as opções autor e grep, você tem que adicionar `--all-match` ou o comando irá combinar com qualquer uma delas.)

Uma outra opção de filtro que realmente ajuda muito é `-S` A qual pega um conjunto de caracteres e mostra apenas os commits que introduzem uma mudança no código onde esse conjunto é adicionado ou removido. Por exemplo, se você quer encontrar o último commit que adicionou ou removeu uma referência a uma função especifica, você poderia chamar:

```
$ git log -Sfunction_name
```

A última opção realmente útil para passar ao `git log` como filtro é o caminho. Se você especificar um diretório ou nome de arquivo, você pode limitar os registros retornados referentes aos commits que introduziram uma mudança a estes arquivos. Issa é sempre a última opção e é geralmente precedida por dois traços (`--`) para separa os caminhos das opções dos comandos.

Em Options to limit the output of `git log` nós vamos listar estes e algumas outras opções comuns para sua referência.

Tabela 3. Options to limit the output of `git log`

Opção	Descrição
`-(n)`	Exibe somente os últimos n commits
`--since`, `--after`	Limita os commits para aqueles feitos após a data especificada.
`--until`, `--before`	Limita os commits aos feitos antes da data especificada.
`--author`	Mostra apenas os commits nos quais a entrada do autor corresponde à string especificada.
`--committer`	Mostra apenas os commits nos quais a entrada do committer corresponde à string especificada.
`--grep`	Mostra apenas os commits com uma mensagem de commit contendo a string
`-S`	mostrar apenas commits adicionando ou removendo o código que corresponde à string

Por exemplo, se você quer ver quais commits estão modificando arquivos de testes no histórico do código fonte do Git que sofreram commit por Junio Hamano no mês de Outubro de 2008 e não são commits de merge, você pode executar algo semelhante a isso:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
branch
```

Dos quase 40.000 commits no histórico do código fonte do Git, esse comando mostra os 6 que combinam com esses critérios.

Desfazendo coisas

Em qualquer estágio, você talvez queira desfazer algo. Aqui, vamos rever algumas ferramentas básicas para desfazer modificações que porventura tenha feito. Seja cuidadoso, porque nem sempre você pode voltar uma alteração desfeita. Essa é uma das poucas áreas do Git onde pode perder algum trabalho feito se você cometer algum engano.

Um dos motivos mais comuns para desfazer um comando, aparece quando você executa um commit muito cedo e possivelmente esquecendo de adicionar alguns arquivos ou você escreveu a mensagem do commit de forma equivocada. Se você quiser refazer este commit, execute o commit novamente usando a opção `--amend`:

```
$ git commit --amend
```

Esse comando pega a área stage e a usa para realizar o commit. Se você não fez nenhuma alteração desde o último commit (por exemplo, se você executar o comando imediatamente depois do commit anterior), então sua imagem dos arquivos irá ser exatamente a mesma, e tudo o que você alterará será a mensagem do commit.

O mesmo editor de mensagens de commit é acionado, porém o commit anterior já possui uma mensagem. Você pode editar a mensagem como sempre, porém esta sobrescreve a mensagem do commit anterior.

Por exemplo, se você fizer um commit e então lembrar que esqueceu de colocar no stage as modificações de um arquivo que você quer adicionar no commit, você pode fazer algo semelhante a isto:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

No final das contas você termina com um único commit – O segundo commit substitui o resultante do primeiro.

Retirando um arquivo do Stage

A próxima sessão demonstra como trabalhar com modificações na área stage e work directory. A boa notícia é que o comando que você usa para verificar o estado dessas duas áreas, também te relembra como desfazer as modificações aplicadas. Por exemplo, vamos supor que você alterou dois arquivos, e deseja realizar o commit deles separadamente, porém você acidentalmente digitou `git add *` adicionando ambos ao stage. Como você pode retirar um deles do stage? O comando `git status` lhe lembrará de como fazer isso:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Logo abaixo do texto “Changes to be committed”, diz `git reset HEAD <file>...` para retirar o arquivo do stage. Então, vamos usar esta sugestão para retirar o arquivo `CONTRIBUTING.md` do stage:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M   CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

O comando é um tanto quanto estranho, mas funciona. O arquivo `CONTRIBUTING.md` volta ao estado modificado porem está novamente fora do stage.

NOTA	É verdade que o comando `git reset` pode ser perigoso, especialmente se você usar a opção `--hard`. Entretanto, no cenário descrito acima, o arquivo no working directory está inalterado, então é relativamente seguro utilizar o comando.

Essa mágica usando o `git reset` é tudo que você precisa saber por enquanto sobre este comando. Nós vamos entra mais no detalhe sobre o que o comando `reset` faz e como usá-lo de forma a fazer coisas realmente interessantes em Reset Demystified.

Desfazendo as Modificações de um Arquivo

E se você se der conta de que na verdade não quer manter as modificações do arquivo `CONTRIBUTING.md`? Como você pode reverter as modificações, voltando a ser como era quando foi realizado o último commit (ou clone inicial, ou seja como for que você chegou ao seu working directory)? Felizmente, `git status` diz a você como fazer isso também. Neste último exemplo, a área fora do stage parece com isso:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Isso lhe diz de forma explicita como descartar as modificações que você fez. Vamos fazer o que o comando nos sugeriu:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Você pode notar que as modificações foram revertidas.

IMPORTANTE	É importante entender que o `git checkout -- <file>` é um a comando perigoso. Qualquer modificação que você fez no arquivo se foi — O Git apenas substitui o arquivo pela última versão (mais recente) que sofreu commit. Nunca use este comando a não ser que você saiba com certeza que não quer salvar as modificações do arquivo.

Se você gostaria de manter as modificações que fez no arquivo, porem precisa tirá-lo do caminho por enquanto, sugerimos que pule para a documentação sobre BranchesBranches no Git; esta geralmente é a melhor forma de fazer isso.

Lembre-se, qualquer coisa que sobre commit com Git pode quase sempre ser recuperada. Até mesmo commits que estava em algum branches que foram deletados ou commits que forma sobre

escritos através de um `--amend` podem ser recuperados (vejaData Recovery para recuperação de dados). Contudo, qualquer coisa que você perder que nunca sofreu commit pode ser considerada praticamente perdida.

Trabalhando de Forma Remota

Para colaborar com qualquer projeto Git, você precisará saber como gerenciar seus repositórios remotos. Repositórios remotos são versões de seu repositório hospedado na Internet ou em uma rede qualquer. Você pode ter vários deles, cada um dos quais geralmente é ou somente leitura ou leitura/escrita. Colaborar com outras pessoas envolve o gerenciamento destes repositórios remotos, fazer *pushing*(atualizar) e *pulling*(obter) de dados para e deles quando você precisar compartilhar seu trabalho. Gerenciar repositórios remotos inclui saber como adicioná-los remotamente, remover aqueles que não são mais válidos, gerenciar vários *branches*(ramos) e definí-los como rastreados ou não e muito mais. Nesta seção, abordaremos algumas destas habilidades de gereciamento remoto.

Exibindo seus repositórios remotos

Para ver quais servidores remotos você configurou, você pode executar o comando `git remote`. Ele lista os nomes abreviados de cada repositório remoto manejado que você especificou. Se você clonou seu repositório, você deve pelo menos ver *origin*(origem) – que é o nome padrão dado pelo Git ao servidor que você clonou:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Você também pode especificar `-v`, que mostra as URLs que o Git tem armazenado pelo nome abreviado a ser usado para ler ou gravar naquele repositório remoto:

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Se você tem mais de um repositório remoto, o comando lista todos eles. Por exemplo, um repositório com diversos repositórios remotos para trabalhar com vários colaboradores pode ser algo parecido com isto:

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Isto significa que nós podemos obter(*pull*) contribuições de qualquer um desses usuários muito facilmente. Nós podemos, adicionalmente, ter a permissão de atualizar(*push*) um ou mais destes, embora não possamos dizer isso nesse caso.

Note que estes repositórios remotos usam uma variedade de protocolos e nós falaremos mais sobre isso em Getting Git on a Server.

Adicionando Repositórios Remotos

Nós mencionamos e demos algumas demonstrações de como o comando `clone` implicitamente adiciona a origem(`origin`) remota para você. Aqui está como adicionar um novo repositório remoto explicitamente. Para adicionar um novo repositório Git remoto como um nome curto que você pode referenciar facilmente, execute `git remote add <shortname> <url>`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

Agora você pode usar a string `pb` na linha de comando no lugar de uma URL completa. Por exemplo, se você quiser buscar toda a informação que Paul tem, mas você ainda não tem em seu repositório, você pode executar `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

O *master branch*(ramo mestre) do Paul agora está acessível localmente como `pb/master` – você pode fundí-lo(*merge*) dentro de uma de suas ramificações(*branches*) ou você pode checar fora da ramificação local se você quiser inspecioná-lo. (Nós abordaremos o que são ramificações(*branches*) e como usá-las mais detalhadamente em Branches no Git.)

Buscando e Obtendo de seus Repositórios Remotos

Como você viu, para obter dados de seus projetos remotos, você pode executar:

```
$ git fetch [remote-name]
```

O comando vai até aquele projeto remoto e extrai todos os dados daquele projeto que você ainda não tem. Depois que você faz isso, você deve ter como referência todos as ramificações(*branches*) daquele repositório remoto, que você pode mesclar(*merge*) com o atual ou inspecionar a qualquer momento.

Se você clonar um repositório, o comando automaticamente adiciona àquele repositório remoto com o nome `origin`. Então, `git fetch origin` busca qualquer novo trabalho que tenha sido enviado para aquele servidor desde que você o clonou ou fez a última busca(*fetch*). É importante notar que o comando `git fetch` só baixa os dados para o seu repositório local - ele não é automaticamente mesclado(*merge*) com nenhum trabalho seu ou modificação que você esteja trabalhando atualmente. Você deve mesclá-los manualmente dentro de seu trabalho quando você estiver pronto.

Se o `branch` atual é configurando para rastrear um `branch` remoto (veja a próxima seção e Branches no Git para mais informação), você pode usar o comando `git pull` para buscar(*fetch*) e então mesclar(*merge*) automaticamente aquele `branch` remoto dentro do seu `branch` atual. Este pode ser um fluxo de trabalho mais fácil e mais confortável para você, e por padrão, o comando `git clone` automaticamente configura a sua `master branch` local para rastrear a `master branch` remota ou qualquer que seja o nome do `branch` padrão no servidor de onde você o clonou. Executar `git pull` comumente busca os dados do servidor de onde você originalmente clonou e automaticamente tenta mesclá-lo dentro do código que você está atualmente trabalhando.

Pushing to Your Remotes

Quando você tem seu projeto em um ponto que deseja compartilhar, é necessário enviá-lo para o servidor remoto. O comando para isso é simples: `git push [remote-name] [branch-name]`. Se você quiser enviar sua ramificacão (branch) master para o servidor `origin` (novamente, a clonagem geralmente configura ambos os nomes para você automaticamente), então você pode executar isso

para enviar quaisquer commits feitos para o servidor:

```
$ git push origin master
```

Este comando funciona apenas se você clonou de um servidor ao qual você tem acesso de escrita (write-access) e se ninguém mais utilizou o comando push nesse meio-tempo. Se você e outra pessoa clonarem o repositório ao mesmo tempo e ela utilizar o comando push e, em seguida, você tentar utilizar, seu envio será rejeitado. Primeiro você terá que atualizar localmente, incorporando o trabalho dela ao seu, só assim você poderá utilizar o comando push. Veja Branches no Git para informações mais detalhadas sobre como enviar para servidores remotos.

Inspecionando o Servidor Remoto

Se você quiser ver mais informações sobre um servidor remoto em particular, você pode usar o comando `git remote show [nome-remoto]`. Ao executar este comando com um nome abreviado específico, como `origin`, obterá algo assim:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Ele lista a URL para o repositório remoto, bem como as informações de rastreamento do branch. O comando, de forma útil, comunica que se você estiver no branch master e executar `git pull`, ele irá mesclar (merge) automaticamente no branch master do servidor após buscar (fetch) todas as referências remotas. Ele também lista todas as referências remotas recebidas.

Esse é um exemplo simples que você provavelmente encontrará. Quando você usa o Git mais intensamente, no entanto, pode ver muito mais informações com `git remote show`:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to
date)
    markdown-strip                 pushes to markdown-strip                 (up to
date)
    master                         pushes to master                         (up to
date)
```

Este comando mostra para qual ramificação (branch) é enviada automaticamente quando você executa `git push` enquanto em certas ramificações. Ele também mostra quais branches remotos do servidor você ainda não tem, quais você tem que foram removidos do servidor e várias branches locais que são capazes de se fundir automaticamente com seu branch de rastreamento remoto quando você executa `git pull`.

Removendo e Renomeando Remotes

Você pode utilizar o `git remote rename` para alterar o nome curto de servidores remotos. Por exemplo, se você deseja renomear `pb` para ` paul `, você pode fazer isso com `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Vale a pena mencionar que isso muda todos os nomes de ramificações de rastreamento remoto também. O que costumava ser referenciado em `pb/master` agora está em `paul/master`.

Se você quiser remover um servidor remoto por algum motivo - e você anteriormente moveu o servidor ou não está mais usando um em particular, ou talvez um contribuidor não esteja mais contribuindo - você pode usar `git remote remove` ou `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

Criando Tags

Da mesma forma que a maioria dos VCSs, o Git tem a habilidade de marcar pontos específicos na história como sendo importantes. Normalmente as pessoas usam essa funcionalidade para marcar pontos onde foram feitas releases (v1.0 e assim por diante). Nessa sessão, você irá aprender como listar as tags existentes, como criar novas tags e quais são os diferentes tipos de tags.

Listando suas Tags

Listar tags disponíveis no Git não tem rodeios. Apenas escreva `git tag`:

```
$ git tag
v0.1
v1.3
```

Esse comando lista as tags em ordem alfabética; a ordem na qual eles aparecerem não tem real importância.

Você pode também procurar por tags com algum padrão em especifico. O repositório fonte do Git, por exemplo, contém mais de 500 tags. Se você deseja procurar apenas a série 1.8.5, você pode executar isso:

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Criando Tags

O Git usa dois tipos de tags: Leve e Anotada.

Uma tag do tipo leve é muito parecida com um branch que não muda – Ela apenas aponta para um commit em especifico.

Tags anotadas, entretanto, são um armazenamento completo de objetos no banco de dados do Git.

Elas têm checksum, contem marcações de nome, email e data; têm uma mensagem de tag; e podem ser assinadas e asseguradas pela GPG (GNU Privacy Guard). É geralmente recomendado que você crie tags anotadas assim você tem todas as informações; mas se você quer um tag temporária ou por alguma razão não quer manter todas as informações, tags do tipo leve esta disponíveis para isso.

Tags Anotadas

Criar uma tag anotada no Git é simples. A forma mais fácil é por especificar o parâmetro `-a` quando você executa o comando `tag`:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

O `-m` define a mensagem de tag, a qual é armazenada junto com a tag. Se você não especificar uma mensagem para uma tag anotada, o Git abre seu editor para que você possa digitar nele.

Você pode ver os dados da tag juntamente com o commit onde ela foi realizada usando o comando `git show`:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Este mostra as informações de tag, a data do commit atrelado a tag, e a mensagem antes mesmo de mostrar as informações do commit.

Tags de Tipo Leve

Uma outra forma de submeter uma tag é usando o tipo leve. Isso é basicamente o checksum de commit armazenando em um arquivo – nenhuma outra informação é mantida. Para criar uma tag do tipo leve, não forneça as opções `-a`, `-s`, or `-m`:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Dessa vez, se você executar `git show` nessa tag, você não verá as informações extras. O comando apenas mostrará o commit:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Criando Tags posteriormente

Você pode também criar uma tag após já ter realizado o commit. Suponha que seu histórico de commits seja semelhante a este:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Agora, suponha que você esqueceu de criar a tag na versão v1.2, a qual equivale ao commit “updated rakefile”. Você pode adicionar isso posteriormente. Para criar uma tag neste commit, você deve informar o checksum do commit (ou parte dele) ao final do comando:

```
$ git tag -a v1.2 9fceb02
```

Você pode ver que criou uma tag para o commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

Compartilhando Tags

Por padrão, o comando `git push` não envia as tags para os servidores remoto. Você terá que explicitamente enviar as tags para o servidor de compartilhamento depois de tê-las criado. Esse processo é semelhante a compartilhar branches remotos – você pode executar `git push origin [tagname]`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

Se você tem muitas tags que você quer enviar de uma vez, você pode também usar a opção `--tags` atrelada ao comando `git push`. Isso vai transferir todas as suas tags ainda não enviadas para o servidor remoto.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

Agora, quando alguém executar um clone ou pull a partir do seu repositório, ele vai puxar também todas as tags. ==== Checando as Tags

Você não pode realizar o checkout de uma tag no Git, uma vez que elas não podem ser movidas. Se você quer deixar uma versão do seu repositório idêntica a uma tag especifica em seu diretório de trabalho, você pode criar um novo branch em uma tag especifica com o comando `git checkout -b [branchname] [tagname]`:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Claro que se você fazer isso e realizar um commit, seu branch `version2` será ligeiramente diferente do que a tag `v2.0.0` assim você irá seguir em frente com suas novas modificações, então seja cuidadoso.

Apelidos Git

Antes de finalizarmos o capítulo básico sobre o Git, temos apenas uma pequena dica que pode tornar sua experiência com o Git mais simples, fácil e familiar: apelidos(aliases). Não iremos nos referir a eles ou assumir que você já os tenha usado mais adiante neste livro, porém você provavelmente deve saber como usá-los.

Git não infere automaticamente seu comando se você digitá-lo parcialmente. Se você não quiser digitar o texto inteiro de cada comando, você pode facilmente configurar um apelido para cada comando usando `git config`.

Aqui estão alguns exemplos que talvez você queira configurar:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Isto significa que, por exemplo, ao invés de digitar `git commit`, você só precisa digitar `git ci`. A medida que você for usando o Git, provavelmente usará outros comandos com frequência também. Então, não hesite em criar novos apelidos.

Esta técnica também pode ser muito útil na criação de comandos que você acredita que deveriam

existir. Por exemplo, para corrigir o problema de usabilidade que você encontrou em um arquivo *unstaging*, você pode adicionar seu próprio apelido unstage ao Git:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Isto faz dos dois comandos a seguir equivalentes

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

Este parece um pouco mais claro. Também é comum adicionar ao comando `last` algo como:

```
$ git config --global alias.last 'log -1 HEAD'
```

Deste jeito, você pode ver o último *commit* mais facilmente:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

Como você pode ver, o Git simplesmente substitui o novo comando por qualquer apelido que você escolher. Entretanto, talvez você queira executar um comando externo em vez de um subcomando Git. Nesse caso, você inicia o comando com um caracter de exclamação (`!`). Isto é muito útil se você escreve suas próprias *tools* que trabalham com um repositório Git. Podemos demonstrar isso apelidando `git visual` com `gitk`:

```
$ git config --global alias.visual '!gitk'
```

Sumário

Nesse ponto, você já pode executar todas as operações locais básicas do Git - como criar ou clonar um repositório, fazer alterações, usar *stage* e *commit* para essas alterações e visualizar o histórico de todas as alterações pelas quais o repositório passou. A seguir, abordaremos o recurso matador do Git: seu modelo de ramificação (*branch*).

Branches no Git

Quase todo Sistema de Controle de Versionamento tem alguma forma de suporte a ramificações (Branches). Ramificação significa que você diverge da linha principal de desenvolvimento e continua a trabalhar sem alterar essa linha principal. Em muitas ferramentas de versionamento, este é um processo um tanto difícil, geralmente exigindo que você crie uma nova cópia do diretório do código-fonte, o que pode demorar muito em projetos maiores.

Algumas pessoas se referem ao modelo de ramificação do Git como seu “recurso matador” e certamente diferencia o Git na comunidade de sistemas de versionamento. Por que isso é tão especial? A forma como o Git cria branches é incrivelmente leve, tornando as operações de ramificação quase instantâneas, alternando entre os branches geralmente com a mesma rapidez. Ao contrário de muitos outros sistemas, o Git incentiva fluxos de trabalho que se ramificam e se fundem com frequência, até mesmo várias vezes ao dia. Compreender e dominar esse recurso oferece uma ferramenta poderosa e única e pode mudar totalmente a maneira como você desenvolve.

Branches em poucas palavras

Para realmente entender como o Git trabalha com Branches, precisamos dar um passo atrás e examinar como o Git armazena seus dados.

Como você deve se lembrar de << ch01-introdução # ch01-introdução >>, o Git não armazena dados como uma série de mudanças ou diferenças, mas sim como uma série de snapshots (instantâneos de um momento) .

Quando você faz um commit, o Git armazena um objeto de commit que contém um ponteiro para o snapshot do conteúdo que você testou. Este objeto também contém o nome do autor e o e-mail, a mensagem que você digitou e ponteiros para o commit ou commits que vieram antes desse commit (seu pai ou pais): sem pai para o commit inicial, um pai para um commit normal, e vários pais para um commit que resulta de uma fusão de dois ou mais branches.

Para verificar isso, vamos assumir que você tem um diretório contendo três arquivos, e você seleciona todos eles e efetua o commit. Ele Prepara os arquivos e calcula uma verificação para cada um (o hash SHA-1 que mencionamos em << ch01-introdução # ch01-introdução >>), armazena essa versão do arquivo no repositório Git (Git se refere a eles como blobs), e adiciona esse hash de verificação à área de preparação (staging area):

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

Quando você faz um commit executando `git commit`, o Git verifica cada subdiretório (neste caso, apenas o diretório raiz do projeto) e armazena esses objetos no repositório do Git. O Git então cria um objeto de commit que possui os metadados e um ponteiro para a raiz do projeto para que ele possa recriar aquele snapshot quando necessário.

Seu repositório Git agora contém cinco objetos: um blob para o conteúdo de cada um dos seus três

arquivos, uma árvore que lista o conteúdo do diretório e especifica quais nomes de arquivo são armazenados e quais seus blobs e um commit com o ponteiro para essa árvore e todos os metadados de commit.

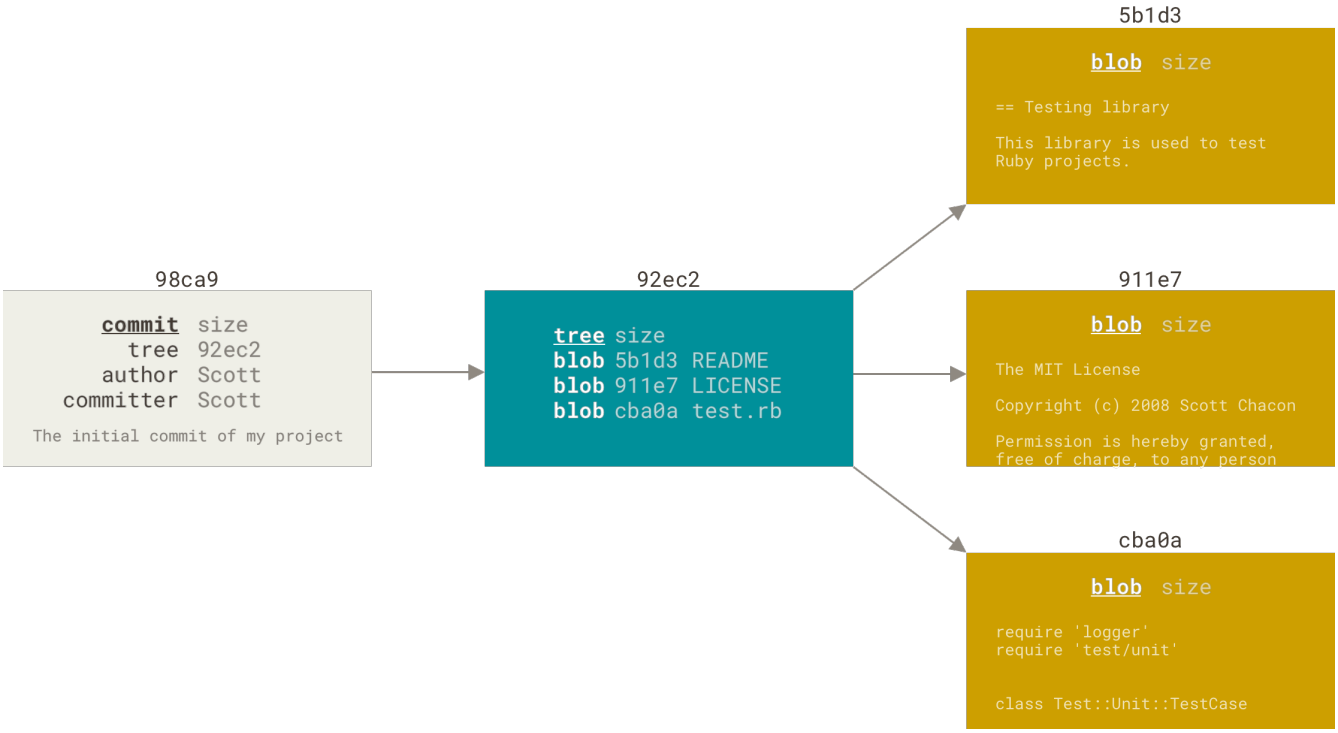


Figura 9. Um commit e sua árvore

Se você fizer algumas mudanças e confirmar novamente, o próximo commit armazena um ponteiro para o commit que veio imediatamente antes dele.

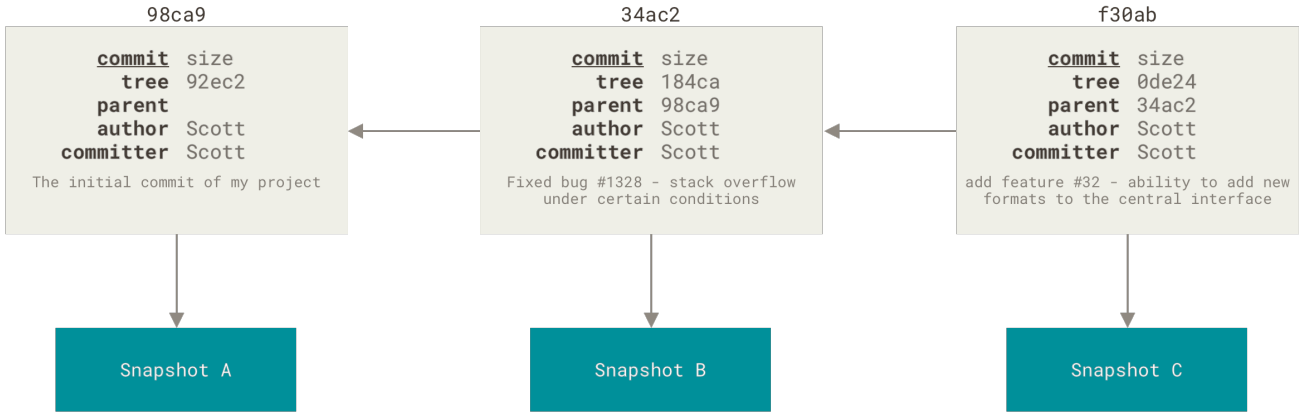


Figura 10. Commits e seus pais

Um branch no Git é simplesmente um ponteiro móvel para um desses commits. O nome do branch padrão no Git é `master`. Conforme você começa a fazer commits, você recebe um branch `master` que aponta para o último commit que você fez. Cada vez que você faz um novo commit, ele avança automaticamente.

NOTA

O branch '`master`' no Git não é um branch especial. É exatamente como qualquer outra ramificação. A única razão pela qual quase todo repositório tem um é que o comando `git init` o cria por padrão e a maioria das pessoas não se preocupa em alterá-lo.

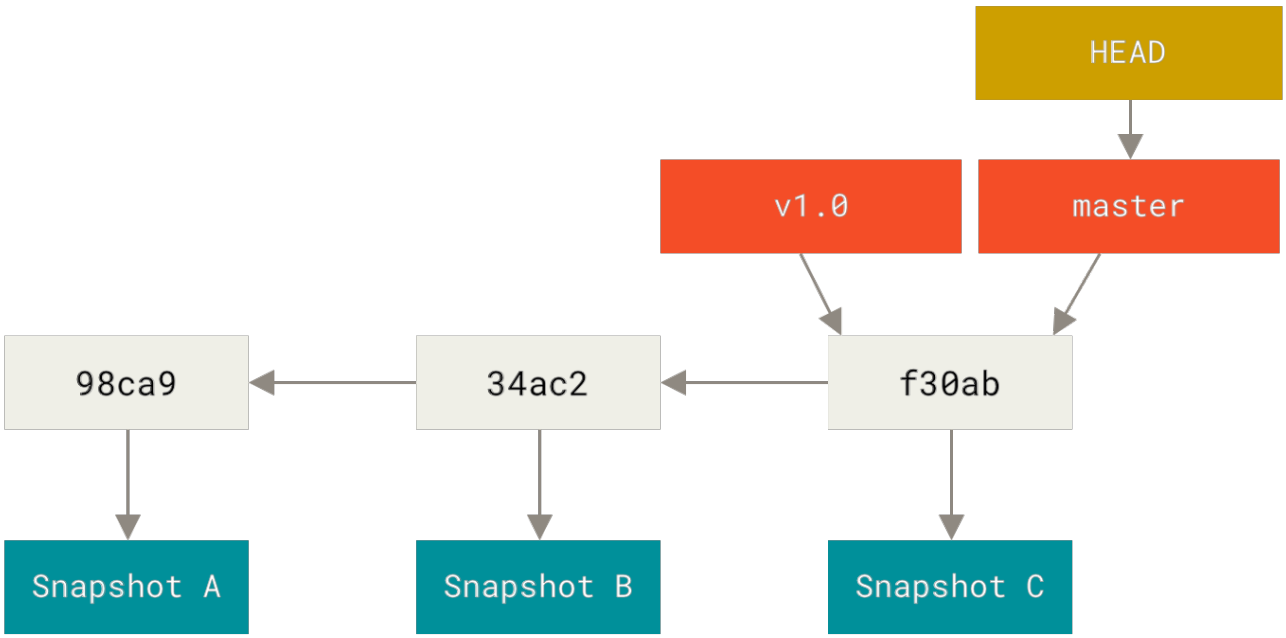


Figura 11. Um branch e seu histórico de commits

Criando um Novo Branch

O que acontece se você criar um novo branch? Bem, fazer isso cria um novo ponteiro para você mover. Digamos que você crie um novo branch chamado: testing. Você faz isso com o comando `git branch` :

```
$ git branch testing
```

Isso cria um novo ponteiro para o mesmo commit em que você está atualmente.

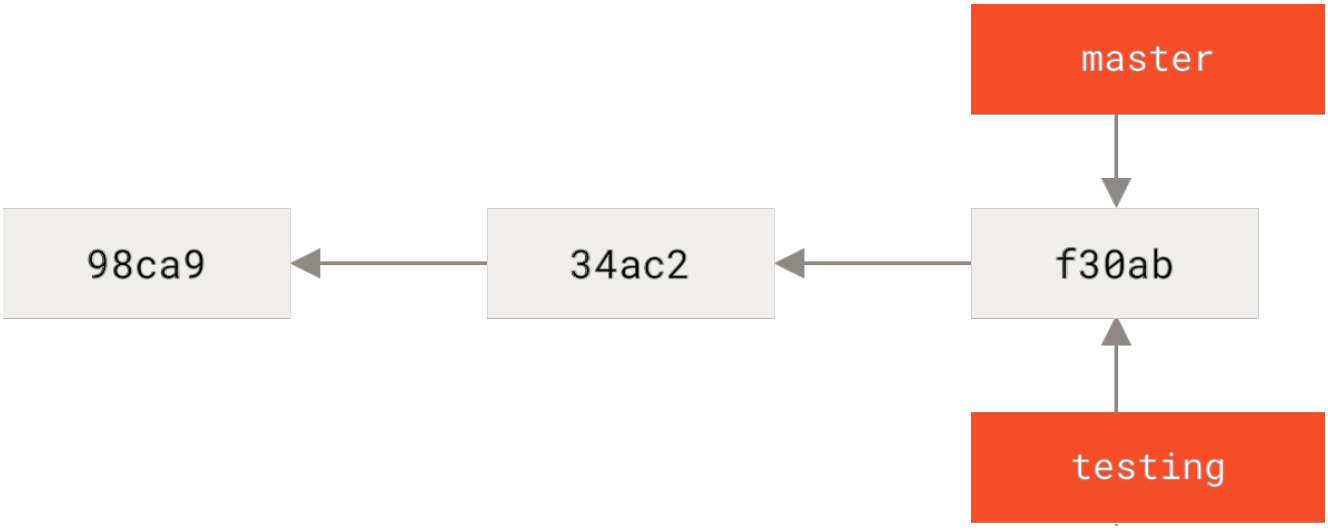


Figura 12. Duas branches apontando para a mesma série de commits

Como o Git sabe em qual branch você está atualmente? Ele mantém um ponteiro especial chamado `HEAD`. Note que isso é muito diferente do conceito de `HEAD` em outros sistemas de versionamento com os quais você pode estar acostumado, como Subversion ou CVS. No Git, isso é um ponteiro para o branch local em que você está. Neste caso, você ainda está em `master`. O comando `git branch` apenas

criou um novo branch - ele não mudou para aquele branch.

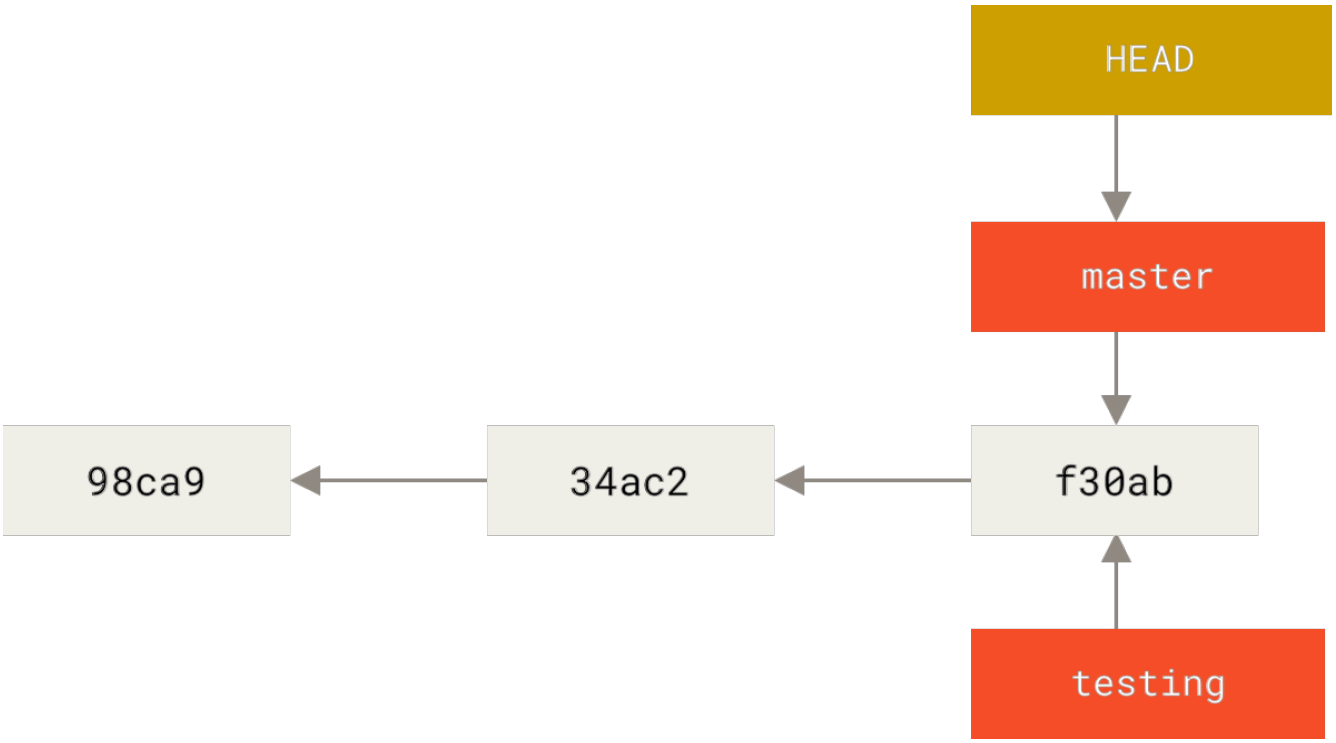


Figura 13. HEAD apontando para um branch

Você pode ver isso facilmente executando um simples comando `git log` que mostra para onde os ponteiros do branch estão apontando. Esta opção é chamada de `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the
central interface
34ac2 Fixed bug #1328 - stack overflow under certain conditions
98ca9 The initial commit of my project
```

Você pode ver os branches `master` e `testing` que estão bem ali ao lado do commit `f30ab`.

Alternando entre Branches

Para mudar para um branch existente, você executa o comando `git checkout`. Vamos mudar para o novo branch `testing`:

```
$ git checkout testing
```

Isso move o `HEAD` e o aponta para o branch `testing`.

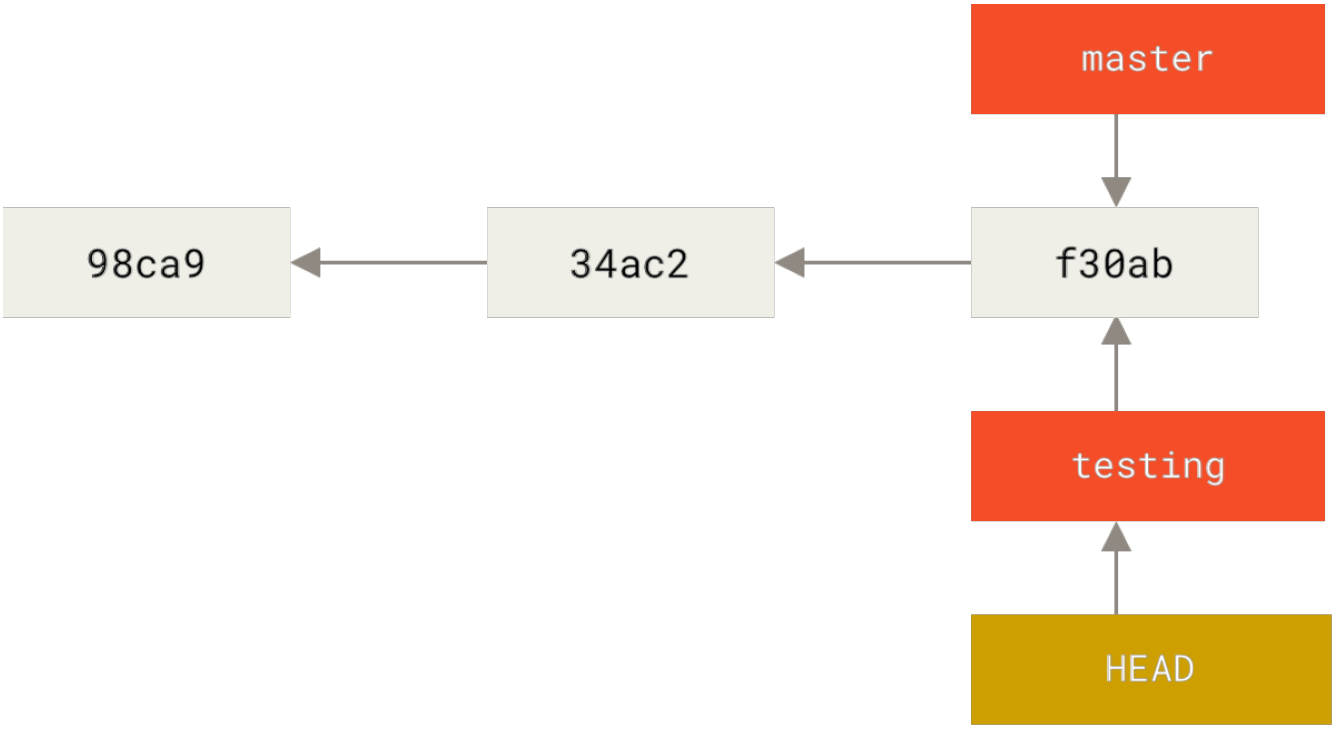


Figura 14. HEAD aponta para o branch atual

O que isso significa? Bem, vamos fazer outro commit:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

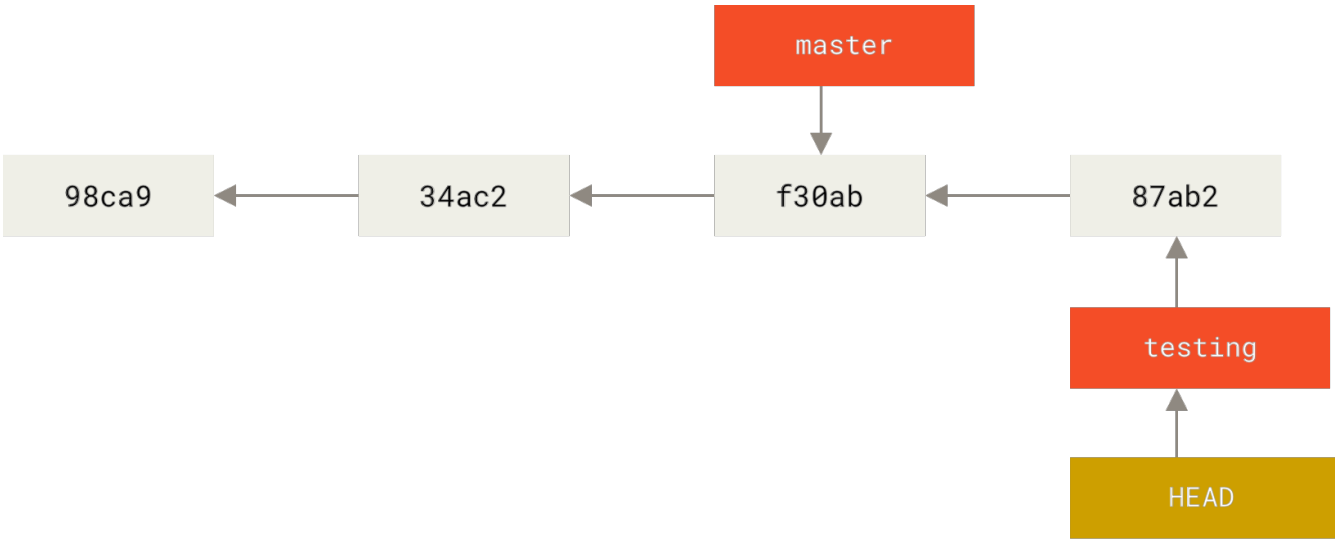


Figura 15. O branch do HEAD avança quando um commit é feito

Isso é interessante, porque agora seu branch `testing` avançou, mas seu branch `master` ainda aponta para o commit em que você estava quando executou `git checkout` para alternar entre os branches. Vamos voltar para o branch `master`:

```
$ git checkout master
```

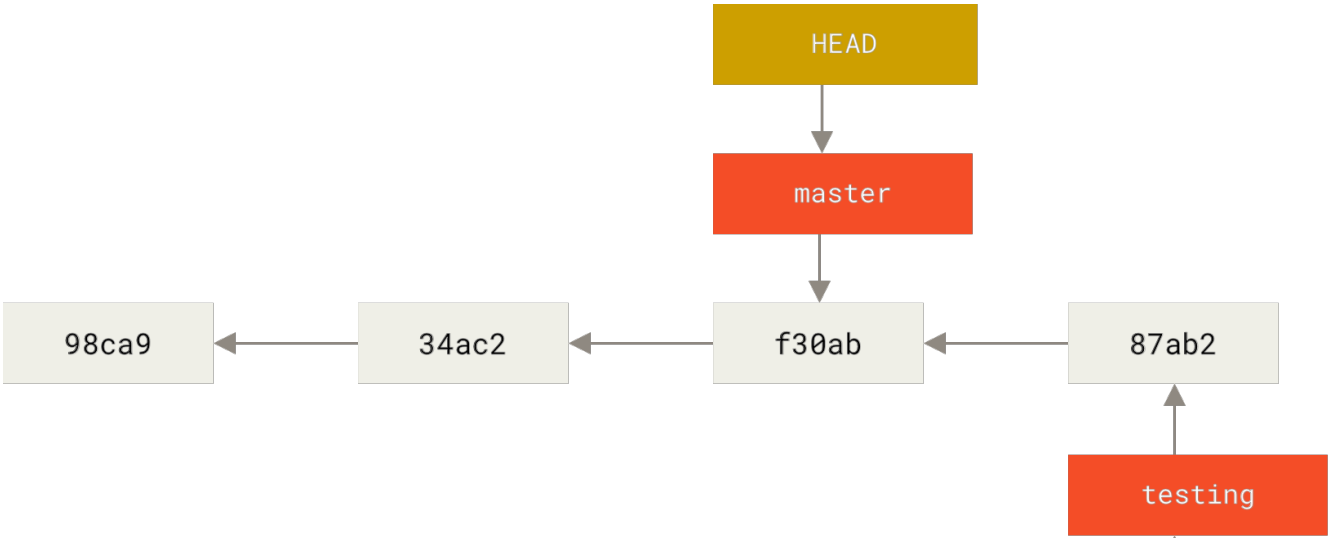


Figura 16. O HEAD se move quando você faz o checkout

Esse comando fez duas coisas. Ele moveu o ponteiro HEAD de volta para apontar para o branch `master`, e reverteu os arquivos em seu diretório de trabalho de volta para o snapshot para o qual `master` aponta. Isso também significa que as alterações feitas a partir deste ponto irão divergir de uma versão mais antiga do projeto. Essencialmente, ele retrocede o trabalho que você fez em seu branch `testing` para que você possa ir em uma direção diferente.

NOTA

A troca de branches muda os arquivos em seu diretório de trabalho

É importante notar que quando você muda de branches no Git, os arquivos em seu diretório de trabalho mudam. Se você mudar para um branch mais antigo, seu diretório de trabalho será revertido para se parecer com a última vez que você fez commit naquele branch. Se o Git não puder fazer, ele não permitirá que você faça a troca.

Vamos fazer algumas mudanças e confirmar novamente:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Agora o histórico do seu projeto divergiu (consulte Histórico de diferenças). Você criou e mudou para um branch, fez algum trabalho nele e, em seguida, voltou para o seu branch principal e fez outro trabalho. Ambas as mudanças são isoladas em branches separados: você pode alternar entre os branches e mesclá-los quando estiver pronto. E você fez tudo isso com comandos simples `branch`, `checkout` e `commit`.

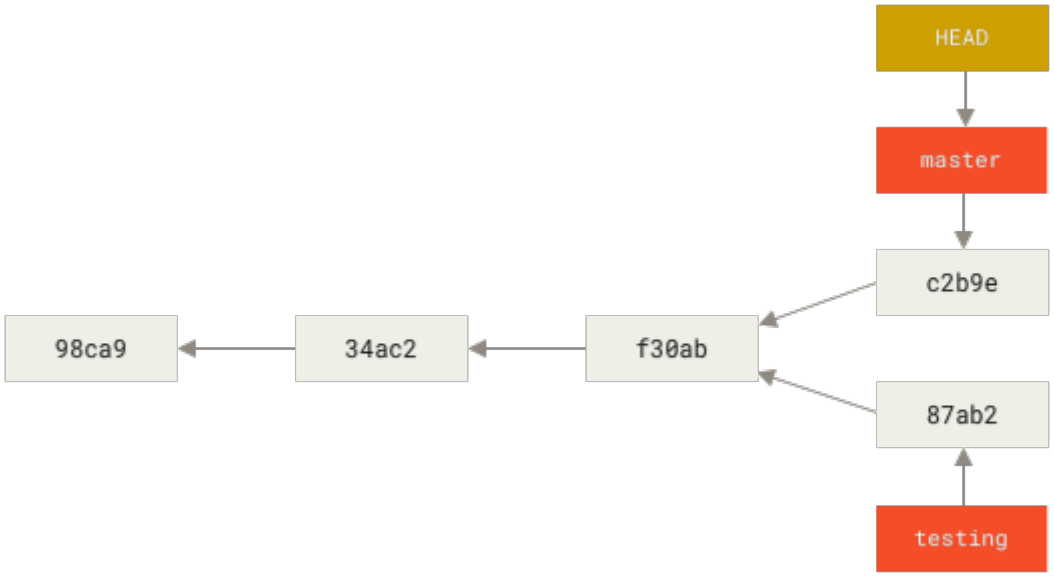


Figura 17. Histórico de diferenças

Você também pode ver isso facilmente com o comando `git log`. Se você executar `git log --oneline --decorate --graph --all`, ele mostrará o histórico de seus commits, exibindo onde estão seus ponteiros de branch e como seu histórico divergiu.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Como um branch no Git é na verdade um arquivo simples que contém a verificação SHA-1 de 40 caracteres do commit para o qual ele aponta, branches são fáceis de criar e destruir. Criar um novo branch é tão rápido e simples quanto escrever 41 bytes em um arquivo (40 caracteres e uma nova linha).

Isso está em nítido contraste com a forma como as ferramentas de versionamento mais antigas se ramificam, o que envolve a cópia de todos os arquivos do projeto em um segundo diretório. Isso pode levar vários segundos ou até minutos, dependendo do tamanho do projeto, enquanto no Git o processo é sempre instantâneo. Além disso, como estamos gravando os pais quando fazemos o commit, encontrar uma base adequada para a mesclagem é feito automaticamente para nós e geralmente é muito fácil de fazer. Esses recursos ajudam a incentivar os desenvolvedores a criar e usar branches com frequência.

Vamos ver por que você deve fazer isso.

O básico de Ramificação (Branch) e Mesclagem (Merge)

Vamos ver um exemplo simples de ramificação (*branching*) e mesclagem (*merging*) com um fluxo de trabalho que você pode vir a usar no mundo real. Você seguirá os seguintes passos:

1. Trabalhar um pouco em um website.
2. Criar um *branch* para um nova história de usuário na qual você está trabalhando.
3. Trabalhar um pouco neste novo *branch*.

Nesse ponto, você vai receber uma mensagem dizendo que outro problema é crítico e você precisa fazer a correção. Você fará o seguinte:

1. Mudar para o seu branch de produção.
2. Criar um novo branch para fazer a correção.
3. Após testar, fazer o merge do branch de correção, e fazer push para produção.
4. Voltar para sua história de usuário original e continuar trabalhando.

Ramificação Básica

Primeiramente, digamos que você esteja trabalhando em seu projeto e já tenha alguns commits no branch `master`.

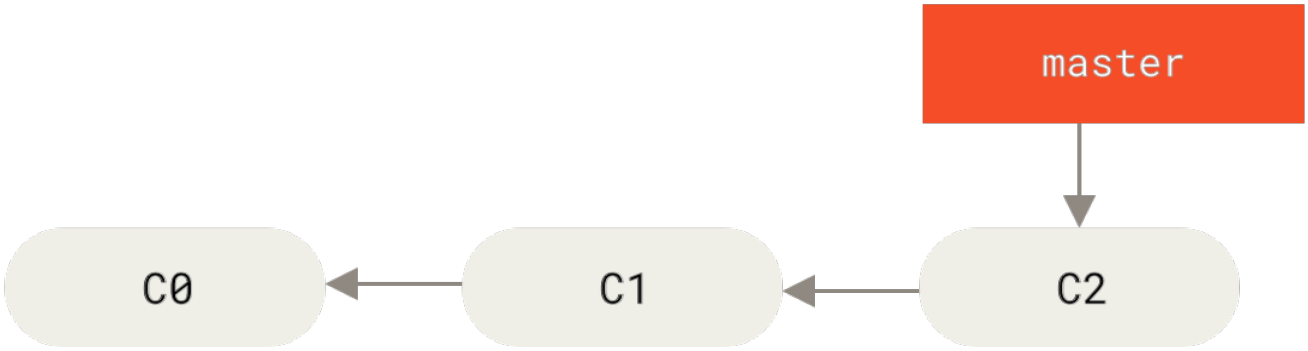


Figura 18. Um histórico de commits simples

Você decidiu que você vai trabalhar no chamado #53 em qualquer que seja o sistema de gerenciamento de chamados que a sua empresa usa.

Para criar um novo branch e mudar para ele ao mesmo tempo, você pode executar o comando `git checkout` com o parâmetro `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Esta é a abreviação de:

```
$ git branch iss53
$ git checkout iss53
```

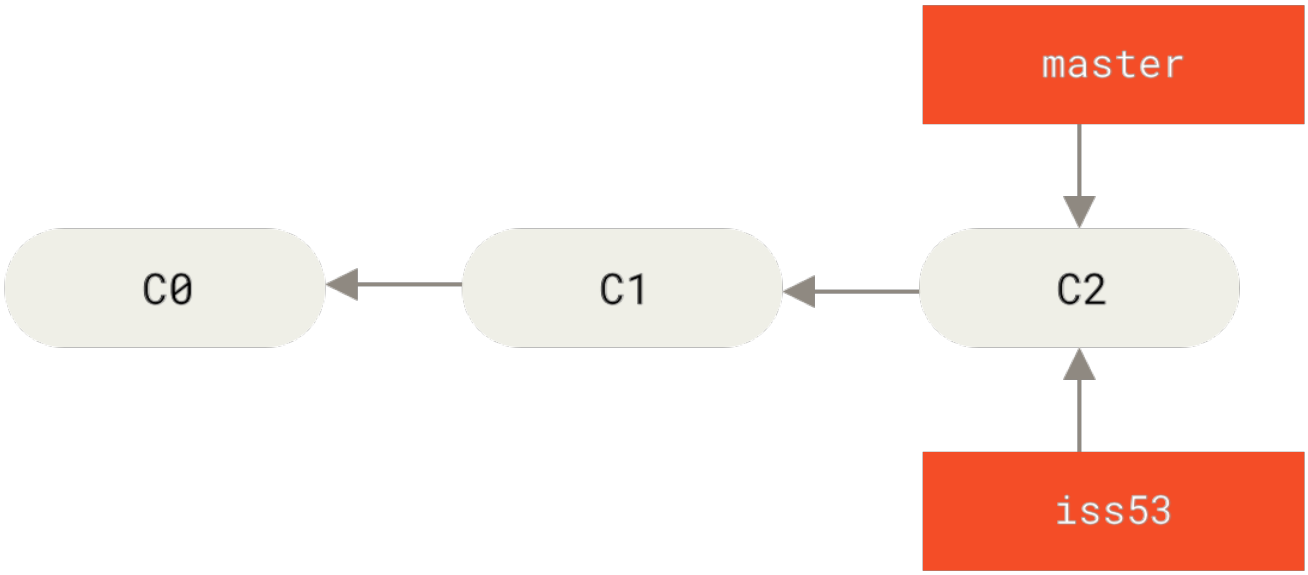


Figura 19. Criando um novo ponteiro de branch

Você trabalha no seu website e adiciona alguns commits.

Ao fazer isso, você move o branch `iss53` para a frente, pois este é o branch que está selecionado, ou *checked out*(isto é, seu `HEAD` está apontando para ele):

```
$ vim index.html
$ git commit -a -m 'Create new footer [issue 53]'
```

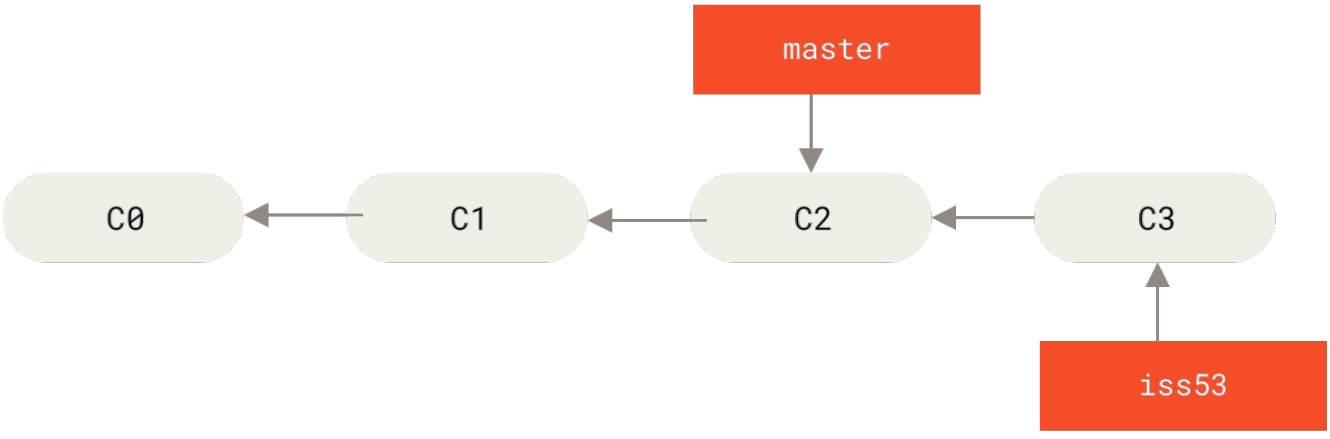


Figura 20. O branch `iss53` moveu para a frente graças ao seu trabalho

Agora você recebe a ligação dizendo que há um problema com o site, e que você precisa corrigí-lo imediatamente. Com o Git, você não precisa enviar sua correção junto com as alterações do branch `iss53` que já fez. Você também não precisa se esforçar muito para desfazer essas alterações antes de poder trabalhar na correção do erro em produção. Tudo o que você precisa fazer é voltar para o seu branch `master`.

Entretanto, antes de fazer isso, note que se seu diretório de trabalho ou stage possui alterações ainda não commitadas que conflitam com o branch que você quer usar, o Git não deixará que você troque de branch. O melhor é que seu estado de trabalho atual esteja limpo antes de trocar de branches. Há maneiras de contornar isso (a saber, o comando stash e commit com a opção --ammend) que iremos cobrir mais tarde, em Stashing and Cleaning. Por agora, vamos considerar que você fez commit de todas as suas alterações, de forma que você pode voltar para o branch `master`:

```
$ git checkout master
Switched to branch 'master'
```

Neste ponto, o diretório de trabalho de seu projeto está exatamente da forma como estava antes de você começar a trabalhar no chamado #53, e você pode se concentrar na correção. Isso é importante de se ter em mente: quando você troca de branches, o Git reseta seu diretório de trabalho para a forma que era na última vez que você commitou naquele branch. O Git adiciona, remove e modifica arquivos automaticamente para se assegurar que a sua cópia de trabalho seja igual ao estado do branch após você adicionar o último commit a ele.

Seu próximo passo é fazer a correção necessária; Vamos criar um branch chamado `hotfix` no qual trabalharemos até a correção estar pronta:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```

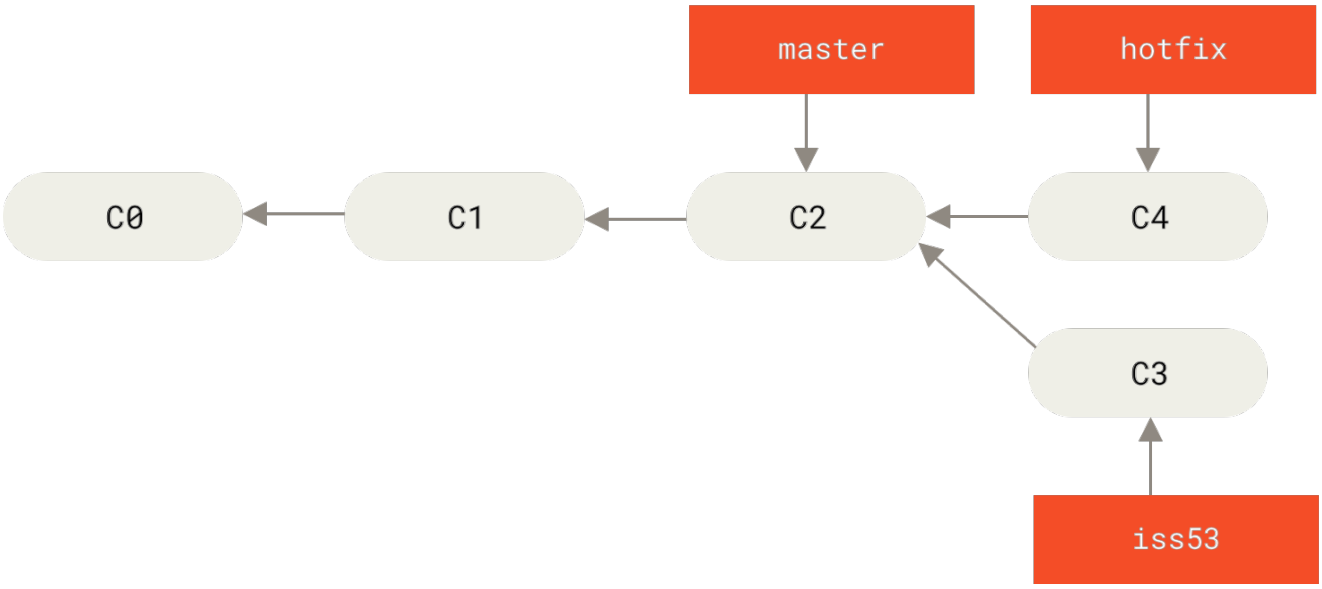


Figura 21. Branch de correção (hotfix) baseado em `master`

Você pode executar seus testes, se assegurar que a correção está do jeito que você quer, e

finalmente mesclar o branch `hotfix` de volta para o branch `master` para poder enviar para produção. Para isso, você usa o comando `git merge` command:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Você vai notar a expressão “fast-forward” nesse merge. Já que o branch `hotfix` que você mesclou aponta para o commit `C4` que está diretamente à frente do commit `C2` no qual você está agora, o Git simplesmente move o ponteiro para a frente. Em outras palavras, quando você tenta mesclar um commit com outro commit que pode ser alcançado por meio do histórico do primeiro commit, o Git simplifica as coisas e apenas move o ponteiro para a frente porque não há nenhum alteração divergente para mesclar — isso é conhecido como um merge “fast-forward.”

Agora, a sua alteração está no snapshot do commmit para o qual o branch `master` aponta, e você pode enviar a correção.

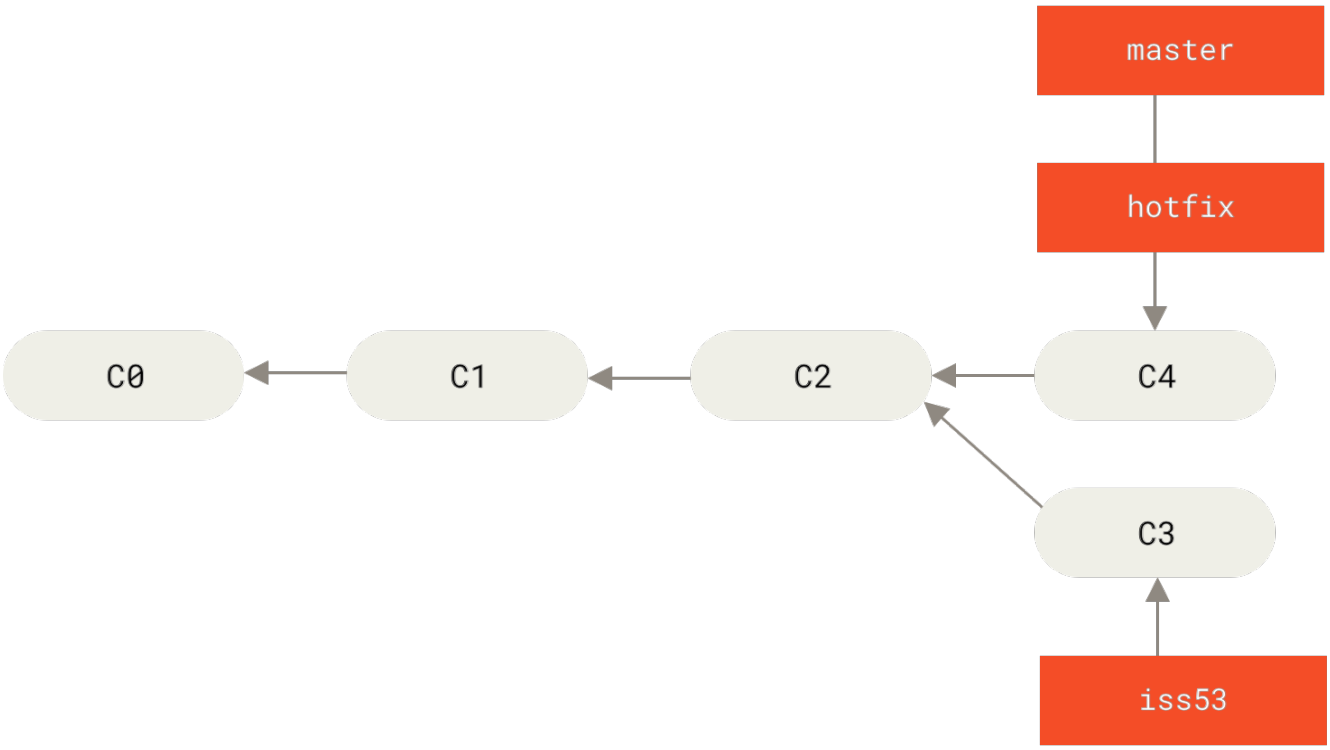


Figura 22. o branch `master` sofre um "fast-forward" até `hotfix`

Assim que a sua correção importantíssima é entregue, você já pode voltar para o trabalho que estava fazendo antes da interrupção. Porém, você irá antes excluir o branch `hotfix`, pois ele já não é mais necessário — o branch `master` aponta para o mesmo lugar. Você pode remover o branch usando a opção `-d` com o comando `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Agora você pode retornar ao branch com seu trabalho em progresso na *issue* #53 e continuar trabalhando.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'Finish the new footer [issue 53]'
[iss53 ad82d7a] Finish the new footer [issue 53]
1 file changed, 1 insertion(+)
```

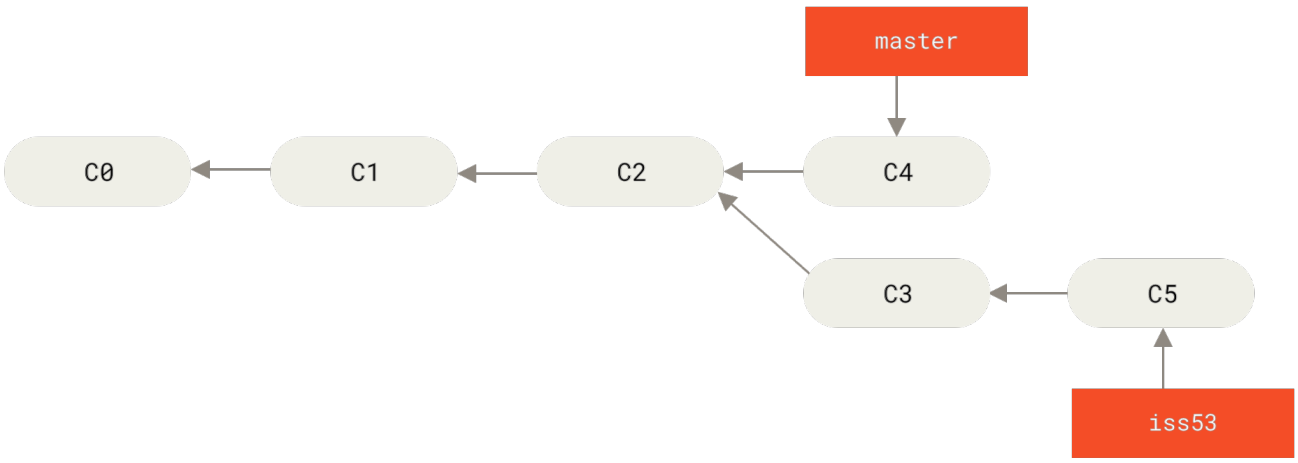


Figura 23. Continuando o trabalho no branch `iss53`

É importante frisar que o trabalho que você fez no seu branch `hotfix` não está contido nos arquivos do seu branch `iss53`. Caso você precise dessas alterações, você pode fazer o merge do branch `master` no branch `iss53` executando `git merge master`, ou você pode esperar para integrar essas alterações até que você decida mesclar o branch `iss53` de volta para `master` mais tarde.

Mesclagem Básica

Digamos que você decidiu que o seu trabalho no chamado #53 está completo e pronto para ser mesclado de volta para o branch `master`. Para fazer isso, você precisa fazer o merge do branch `iss53`, da mesma forma com que você mesclou o branch `hotfix` anteriormente. Tudo o que você precisa fazer é mudar para o branch que receberá as alterações e executar o comando `git merge`:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Isso é um pouco diferente do merge anterior que você fez com o branch `hotfix`. Neste caso, o histórico de desenvolvimento divergiu de um ponto mais antigo. O Git precisa trabalhar um pouco mais, devido ao fato de que o commit no seu branch atual não é um ancestral direto do branch cujas alterações você quer integrar. Neste caso, o Git faz uma simples mesclagem de três vias (*three-way merge*), usando os dois snapshots referenciados pela ponta de cada branch e o ancestral em comum dos dois.

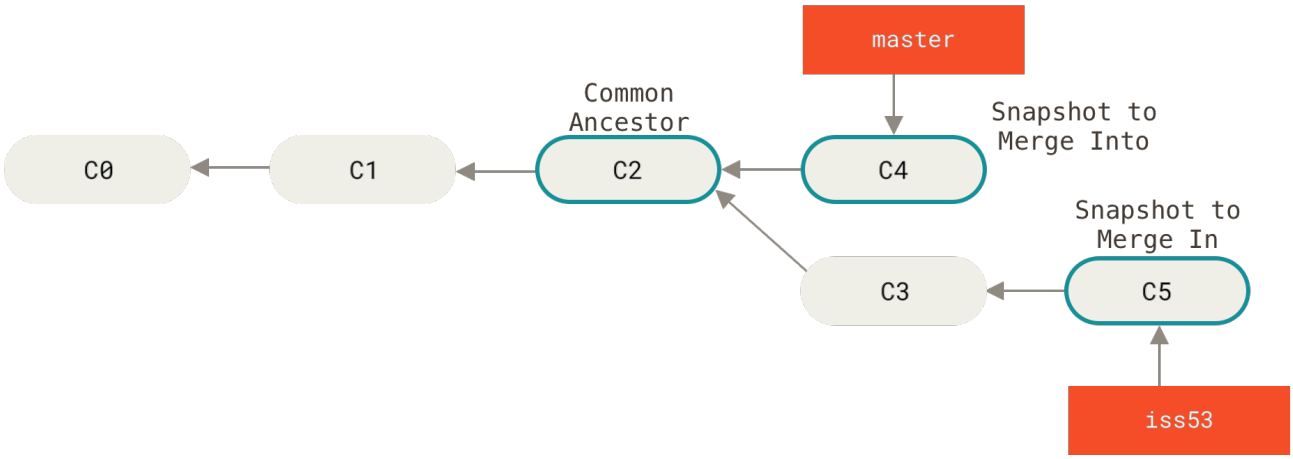


Figura 24. Três snapshots usados em um merge típico

Ao invés de apenas mover o ponteiro do branch para a frente, o Git cria um novo snapshot que resulta desse merge em três vias e automaticamente cria um novo commit que aponta para este snapshot. Esse tipo de commit é chamado de commit de merge, e é especial porque tem mais de um pai.

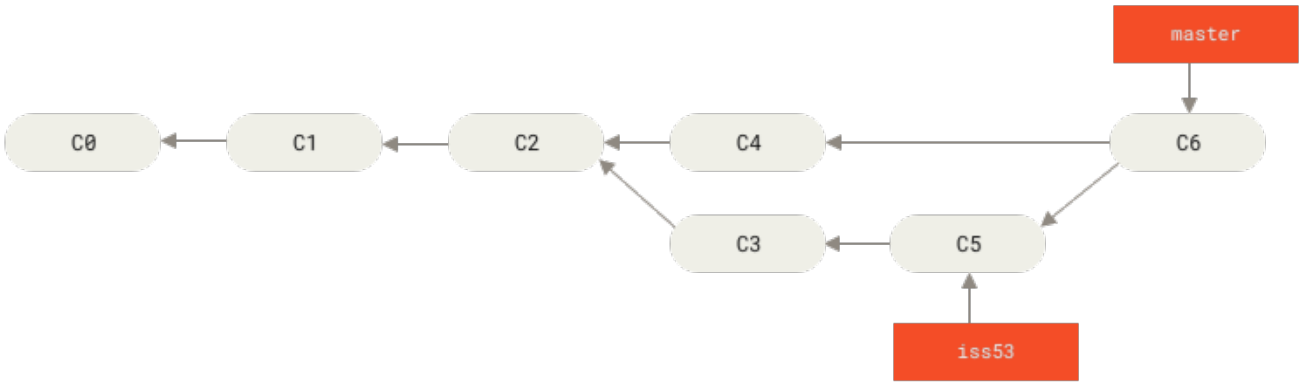


Figura 25. Um commit de merge

Agora que seu trabalho foi integrado, você não precisa mais do branch `iss53`. Você pode encerrar o chamado no seu sistema e excluir o branch:

```
$ git branch -d iss53
```

Conflitos Básicos de Merge

De vez em quando, esse processo não acontece de maneira tão tranquila. Se você mudou a mesma parte do mesmo arquivo de maneiras diferentes nos dois branches que você está tentando mesclar, o Git não vai conseguir integrá-los de maneira limpa. Se a sua correção para o problema #53 modificou a mesma parte de um arquivo que também foi modificado em `hotfix`, você vai ter um conflito de merge que se parece com isso:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

O Git não criou automaticamente um novo commit de merge. Ele pausou o processo enquanto você soluciona o conflito. Para ver quais arquivos não foram mesclados a qualquer momento durante um conflito de merge, você pode executar `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Qualquer arquivo que tenha conflitos que não foram solucionados é listado como *unmerged*("não mesclado"). O Git adiciona símbolos padrão de resolução de conflitos nos arquivos que têm conflitos, para que você possa abrí-los manualmente e solucionar os conflitos. O seu arquivo contém uma seção que se parece com isso:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Isso significa que a versão em `HEAD` (o seu branch `master`, porque era o que estava selecionado quando você executou o comando merge) é a parte superior daquele bloco (tudo acima de `=======`), enquanto que a versão no branch `iss53` contém a versão na parte de baixo. Para solucionar o conflito, você precisa escolher um dos lados ou mesclar os conteúdos diretamente. Por exemplo, você pode resolver o conflito substituindo o bloco completo por isso:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Essa solução tem um pouco de cada versão, e as linhas com os símbolos `<<<<<<<`, `=======`, e `>>>>>>>` foram completamente removidas. Após solucionar cada uma dessas seções em cada arquivo com conflito, execute `git add` em cada arquivo para marcá-lo como solucionado. Adicionar o arquivo ao stage o marca como resolvido para o Git.

Se você quiser usar uma ferramenta gráfica para resolver os conflitos, você pode executar `git mergetool`, que inicia uma ferramenta de mesclagem visual apropriada e guia você através dos conflitos:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Caso você queira usar uma ferramente de merge diferente da padrão (o Git escolheu `opendiff` neste caso porque o comando foi executado em um Mac), você pode ver todas as ferramentas suportadas listadas acima após “one of the following tools.” Você só tem que digitar o nome da ferramenta que você prefere usar.

NOTA Se você precisa de ferramentas mais avançadas para resolver conflitos mais complicados, nós abordamos mais sobre merge em Advanced Merging.

Após você sair da ferramenta, o Git pergunta se a operação foi bem sucedida. Se você responder que sim, o Git adiciona o arquivo ao stage para marcá-lo como resolvido. Você pode executar `git status` novamente para verificar que todos os conflitos foram resolvidos:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Se você estiver satisfeito e verificar que tudo o que havia conflitos foi testado, você pode digitar `git commit` para finalizar o commit. A mensagem de confirmação por padrão é semelhante a esta:

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

Se você acha que seria útil para outras pessoas olhar para este merge no futuro, você pode modificar esta mensagem de confirmação com detalhes sobre como você resolveu o conflito e explicar por que você fez as mudanças que você fez se elas não forem óbvias.

Gestão de Branches

Agora que você criou, mesclou e excluiu alguns branches, vamos dar uma olhada em algumas ferramentas de gerenciamento de branches que serão úteis quando você começar a usar o tempo todo.

O comando `git branch` faz mais do que apenas criar e excluir branches. Se você executá-lo sem argumentos, obterá uma lista simples de seus branches atuais:

```
$ git branch
  iss53
* master
  testing
```

Observe o caractere `*` que no início do `master`: ele indica o branch que você fez check-out (ou seja, o branch para o qual `HEAD` aponta). Isso significa que se você fizer commit neste ponto, o branch `master` será movido para frente com seu novo trabalho. Para ver o último commit em cada branch, você pode executar `git branch -v`:

```
$ git branch -v
  iss53   93b412c Fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 Add scott to the author list in the readme
```

As opções `--merged` e `--no-merged` podem filtrar esta lista para branches que você tem ou ainda não mesclou no branch em que está atualmente. Para ver quais branches já estão mesclados no branch em que você está, você pode executar `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Como você já mesclou o `iss53` anteriormente, você o vê na sua lista. Branches que aparecem na lista sem o `*` na frente deles geralmente podem ser deletados com `git branch -d`; você já incorporou o trabalho deles em outro branch, então não vai perder nada.

Para ver todos os branches que contêm trabalhos que você ainda não mesclou, você pode executar `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

Isso mostra seu outro branch. Por conter trabalho que ainda não foi mesclado, tentar excluí-lo com `git branch -d` irá não irá executar:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Se você realmente deseja excluir o branch e perder esse trabalho, pode forçá-lo com `-D`, como mostra a mensagem.

DICA

As opções descritas acima, `--merged` e `--no-merged` irão, se não for dado um nome de commit ou branch como argumento, mostrar o que é, respectivamente, mesclado ou não mesclado em seu branch *current*.

Você sempre pode fornecer um argumento adicional para perguntar sobre o estado de mesclagem em relação a algum outro branch sem verificar esse outro branch primeiro, como: O que não foi feito merge no branch `master`?

```
$ git checkout testing
$ git branch --no-merged master
  topicA
  featureB
```

Fluxo de Branches

Agora que você tem o básico sobre branches e merges, o que você pode ou deve fazer com eles? Nesta seção, cobriremos alguns fluxos de trabalho comuns que o branch torna possível, para que você possa decidir se deseja incorporá-los em seu próprio ciclo de desenvolvimento.

Branches de longa duração

Como o Git usa uma mesclagem simples de três vias, mesclar de um branch para outro várias vezes durante um longo período é geralmente fácil de fazer. Isso significa que você pode ter vários branches que estão sempre abertos e que você usa para diferentes estágios do seu ciclo de desenvolvimento; você pode mesclar regularmente alguns deles com outros.

Muitos desenvolvedores Git têm um fluxo de trabalho que adota essa abordagem, como ter apenas código totalmente estável em seu branch `master` - possivelmente apenas código que foi ou será lançado. Eles têm outro branch paralelo chamado `developers` ou `next`, a partir do qual trabalham ou usam para testar a estabilidade - não é necessáriamente sempre estável, mas sempre que chega a um estado estável, pode ser mesclado com o `master`. É usado para puxar branches de tópico (de curta duração, como seu anterior `iss53`) quando eles estão prontos, para garantir que eles passem em todos os testes e não introduzam bugs.

Na realidade, estamos falando sobre indicadores que aumentam a linha de commits que você está fazendo. Os branches estáveis estão mais abaixo na linha em seu histórico de commit, e os branches mais avançados estão mais adiante no histórico.

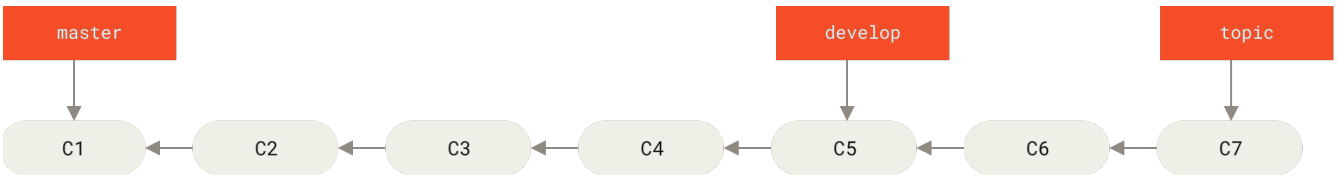


Figura 26. Uma visão linear de branches de estabilidade progressiva

Geralmente é mais fácil pensar neles como silos de trabalho, onde conjuntos de commits passam para um silo mais estável quando são totalmente testados.

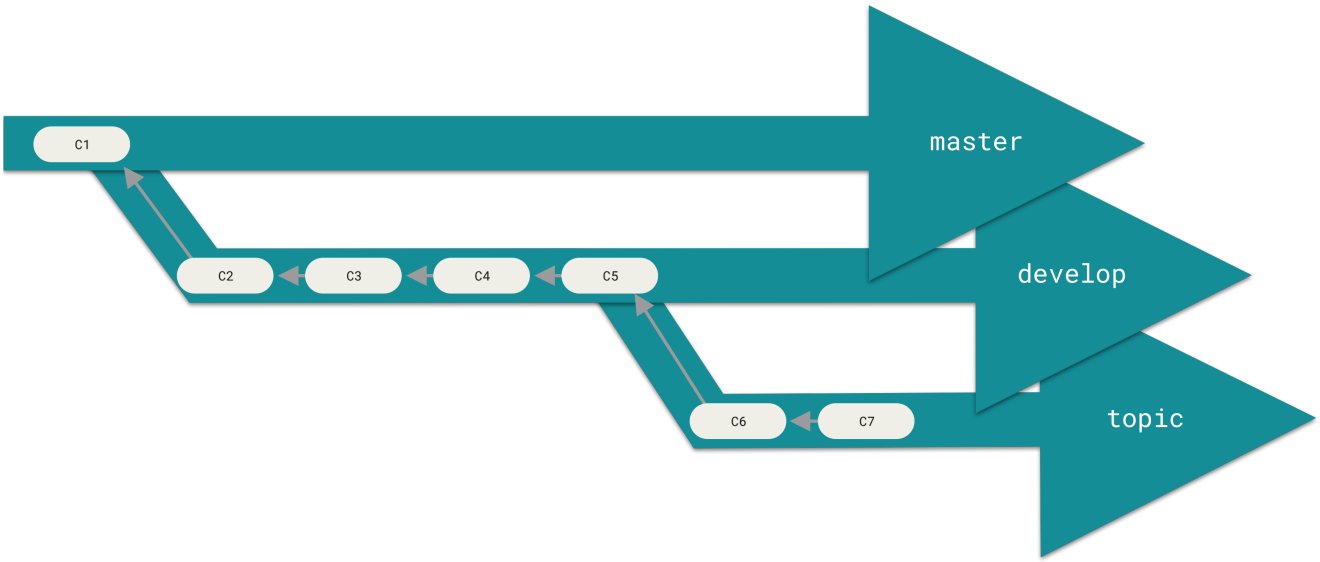


Figura 27. A visão de "silo" de branches de estabilidade progressiva

Você pode continuar fazendo isso por vários níveis de estabilidade. Alguns projetos maiores também têm um ramo `proposto` ou `pu` (proposed updates) que tem branches integrados que podem não estar prontos para ir para o branch `next` ou `master`. A ideia é que seus ramos estejam em vários níveis de estabilidade; quando eles atingem um nível mais estável, eles são mesclados no ramo acima deles. Novamente, não é necessário ter vários branches de longa duração, mas geralmente é útil, especialmente quando você está lidando com projetos muito grandes ou complexos.

Branches por tópicos

Branches de tópicos, no entanto, são úteis em projetos de qualquer tamanho. Um branch de tópico é um branch de curta duração que você cria e usa para um único recurso específico ou trabalho relacionado. Isso é algo que você provavelmente nunca fez com um VCS antes porque geralmente é muito difícil para criar e mesclar branches. Mas no Git é comum criar, trabalhar, mesclar e excluir branches várias vezes ao dia.

Você viu isso na última seção com os branches `iss53` e `hotfix` que você criou. Você fez alguns commits neles e os deletou diretamente após fundi-los em seu branch principal. Esta técnica permite que você mude de contexto de forma rápida e completa - porque seu trabalho é separado em silos onde todas as mudanças naquele branch têm a ver com aquele tópico, é mais fácil ver o que aconteceu durante a revisão de código. Você pode manter as alterações lá por minutos, dias ou meses e mesclá-las quando estiverem prontas, independentemente da ordem em que foram criadas ou trabalhadas.

Considere um exemplo de como fazer algum trabalho (no `master`), ramificando para um problema (`iss91`), trabalhando nisso um pouco, ramificando o segundo branch para tentar outra maneira de lidar com a mesma coisa (`iss91v2`), voltando ao seu branch `master` e trabalhando lá por um tempo, e então ramificando para fazer algum trabalho que você não tem certeza se é uma boa ideia (branch `dumbidea`). Seu histórico de commits será parecido com isto:

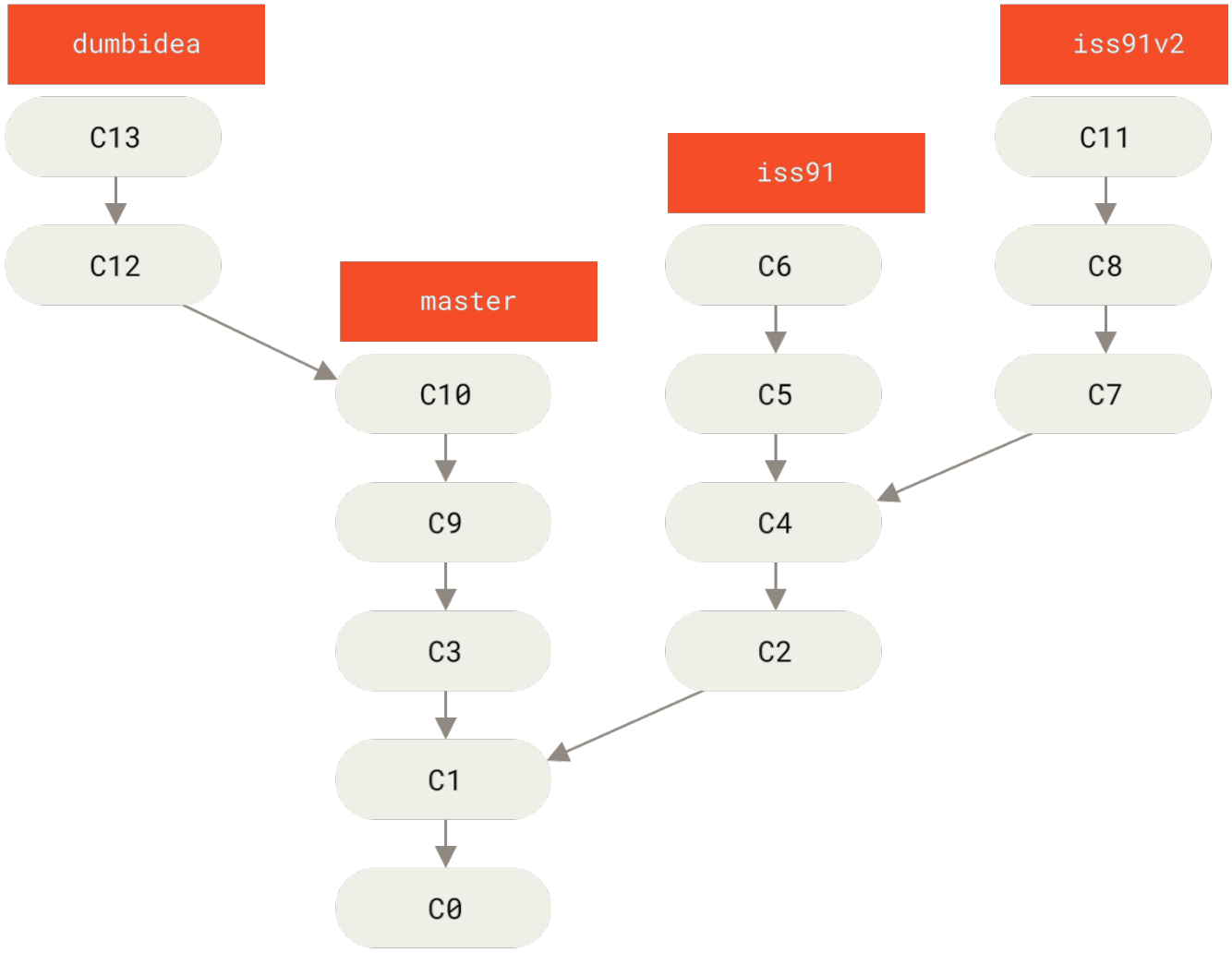


Figura 28. Branches de tópico múltiplos

Agora, digamos que você decida que prefere a segunda solução para o seu problema (`iss91v2`); e você mostrou o branch `dumbidea` para seus colegas de trabalho, e acabou sendo um gênio. Você pode descartar o branch `iss91` original (perdendo commits `C5` e `C6`) e mesclar os outros dois. Seu histórico será assim:

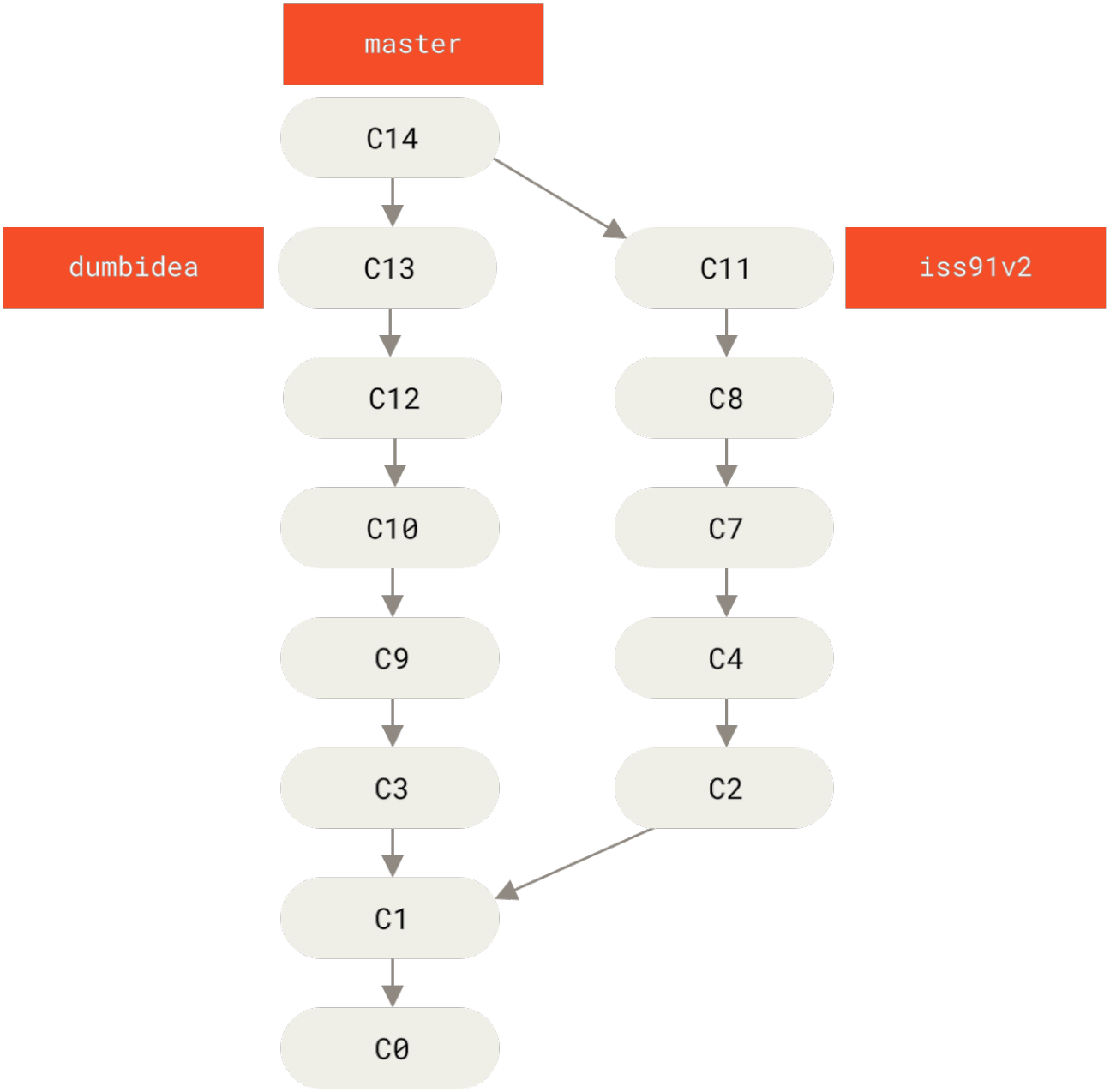


Figura 29. Histórico após mesclar `dumbidea` *and* `iss91v2`

Iremos entrar em mais detalhes sobre os vários fluxos de trabalho possíveis para seu projeto Git em Distributed Git, portanto, antes de decidir qual esquema de ramificação seu próximo projeto usará, certifique-se de ler esse capítulo.

É importante lembrar quando você estiver fazendo tudo isso que esses branches são completamente locais. Quando você está ramificando e mesclando, tudo está sendo feito apenas em seu repositório Git - nenhuma comunicação com o servidor está acontecendo.

Branches remotos

Referências remotas são referências (ponteiros) em seus repositórios remotos, incluindo branches, tags e assim por diante. Você pode obter uma lista completa de referências remotas explicitamente com `git ls-remote [remote]` ou `git remote show [remote]` para branches remotos, bem como mais informações. No entanto, uma forma mais comum é aproveitar as branches de rastreamento

remoto.

Branches de rastreamento remoto são referências ao estado de branches remotas. São referências locais que você não pode mover; eles são movidos automaticamente para você sempre que você fizer qualquer comunicação de rede. Branches de rastreamento remoto agem como marcadores para lembrá-lo de onde estavam as branches em seus repositórios remotos da última vez que você se conectou a eles.

Eles assumem a forma `(remoto)/(branch)`. Por exemplo, se você quiser ver como era o branch `master` em seu branch remoto `origin` da última vez que você se comunicou com ele, você deve verificar o branch `origin/master`. Se você estava trabalhando em um problema com um parceiro e ele enviou um push branch `iss53`, você pode ter seu próprio branch local `iss53`; mas o branch no servidor apontaria para o commit em `origin/iss53`.

Isso pode ser um pouco confuso, então vamos ver um exemplo. Digamos que você tenha um servidor Git em sua rede em `git.ourcompany.com`. Se você clonar a partir dele, o comando `clone` do Git automaticamente o nomeia como `origin` para você, puxa todos os seus dados, cria um ponteiro para onde seu branch `master` está e o nomeia `origin/master` localmente. O Git também fornece seu próprio branch `master` local começando no mesmo lugar que o branch `master` de origem, então você tem algo a partir do qual trabalhar.

NOTA

“origin” não é especial

Assim como o nome do branch “master” não tem nenhum significado especial no Git, tampouco o “origin”. Enquanto “master” é o nome padrão para um branch inicial quando você executa `git init` que é a única razão pela qual é amplamente usado,“origin” é o nome padrão para um repositório remoto quando você executa `git clone`. Se você executar `git clone -o booyah` em vez disso, terá `booyah/master` como seu branch remoto padrão.

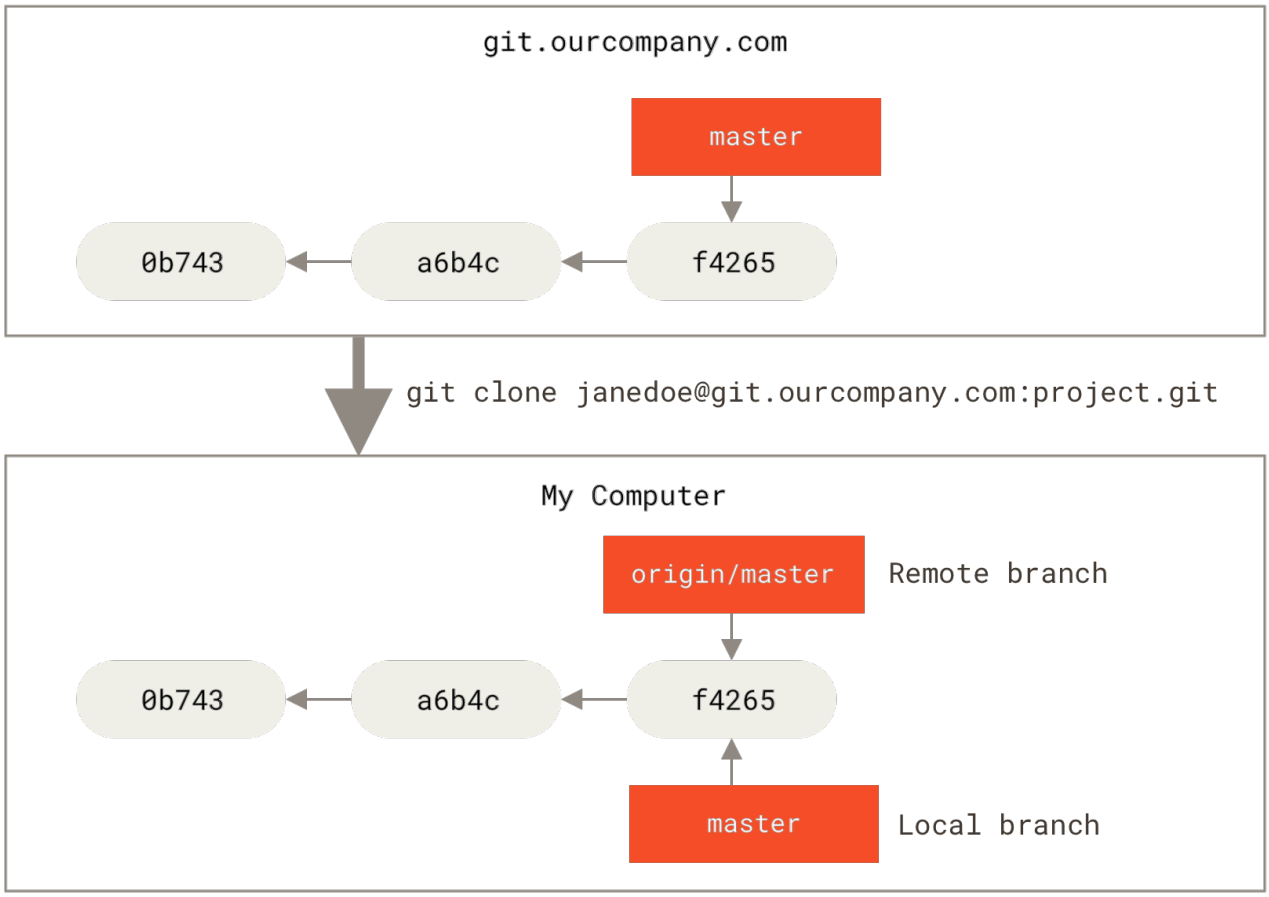


Figura 30. Repositório local e servidor após o clone

Se você fizer algum trabalho em seu branch `master` local e, nesse ínterim, outra pessoa enviar um push para `git.ourcompany.com` e atualizar seu branch `master`, então seus históricos avançam de forma diferente. Além disso, contanto que você fique fora de contato com o servidor de origem, o ponteiro `origin/master` não se move.

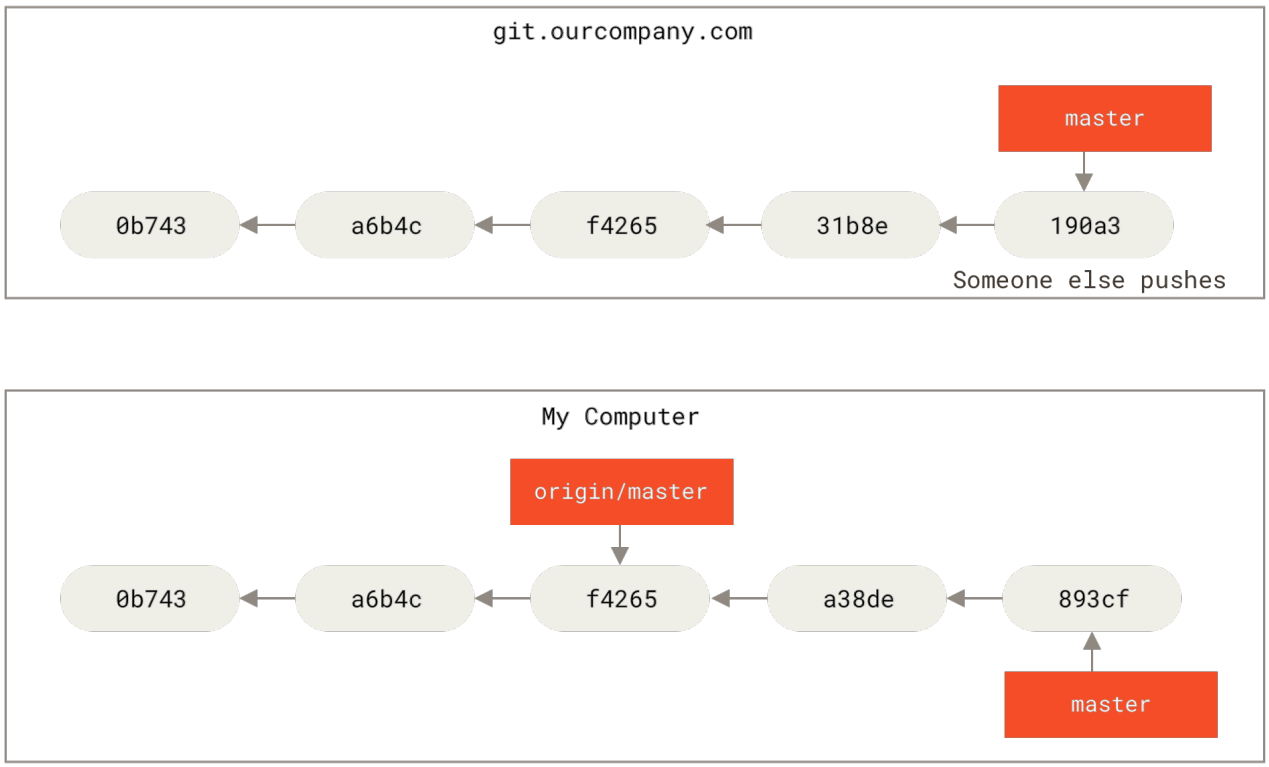


Figura 31. Repositório local e remoto podem divergir

Para sincronizar seu trabalho, você executa um comando `git fetch origin`. Este comando procura em qual servidor está a "origin" (neste caso, é `git.nossaempresa.com`), busca quaisquer dados que você ainda não tenha, e atualiza seu banco de dados local, movendo seu ponteiro `origin/master` para sua nova posição mais atualizada.

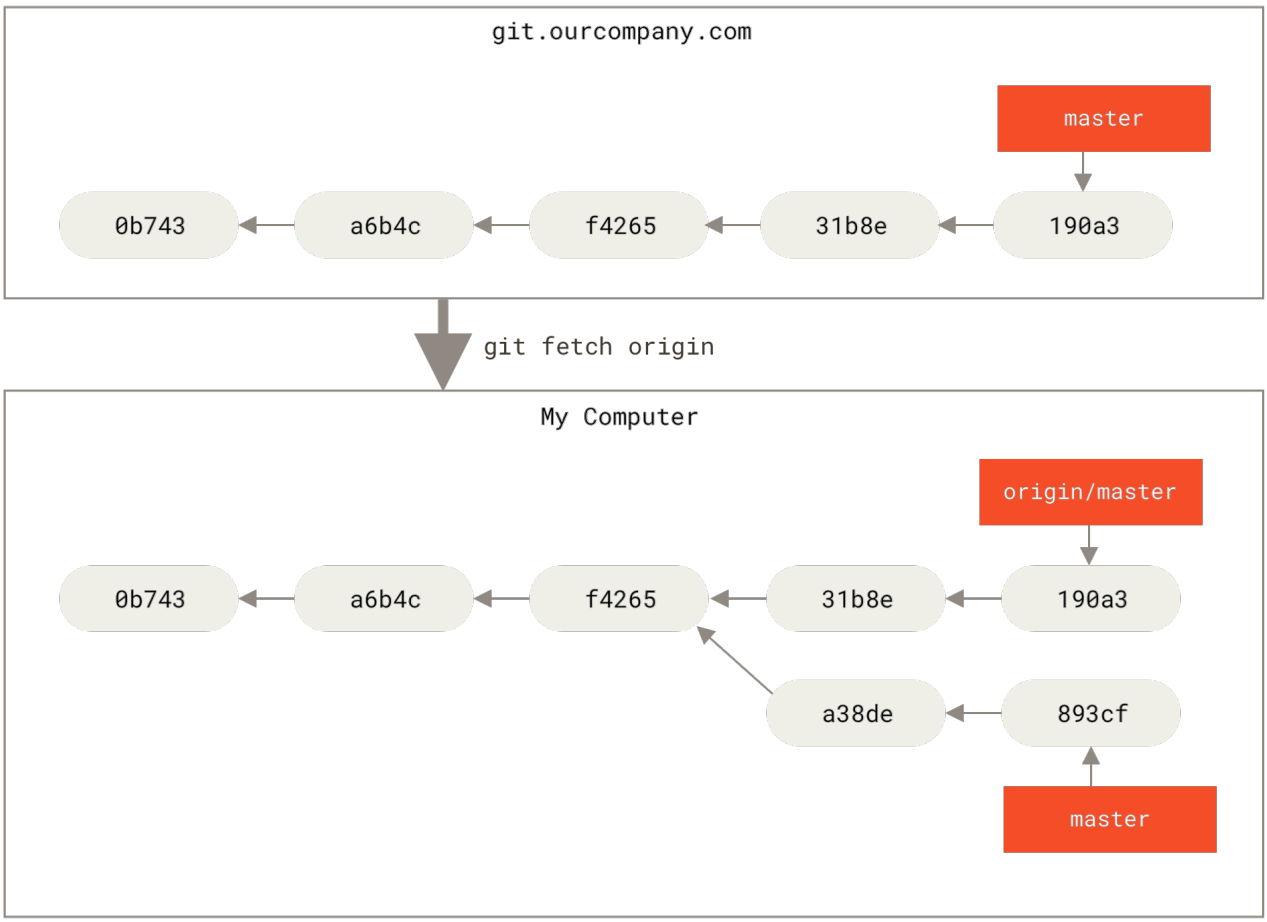


Figura 32. `git fetch` *atualiza suas preferências remotas*

Para demonstrar a existência de vários servidores remotos e como as branches remotas desses projetos remotos se parecem, vamos supor que você tenha outro servidor Git interno usado apenas para desenvolvimento por uma de suas equipes. Este servidor está em `git.team1.ourcompany.com`. Você pode adicioná-lo como uma nova referência remota ao projeto em que está trabalhando atualmente executando o comando `git remote add` conforme abordamos em Fundamentos de Git. Nomeie este servidor remoto como `teamone`, que será o seu apelido para toda a URL.

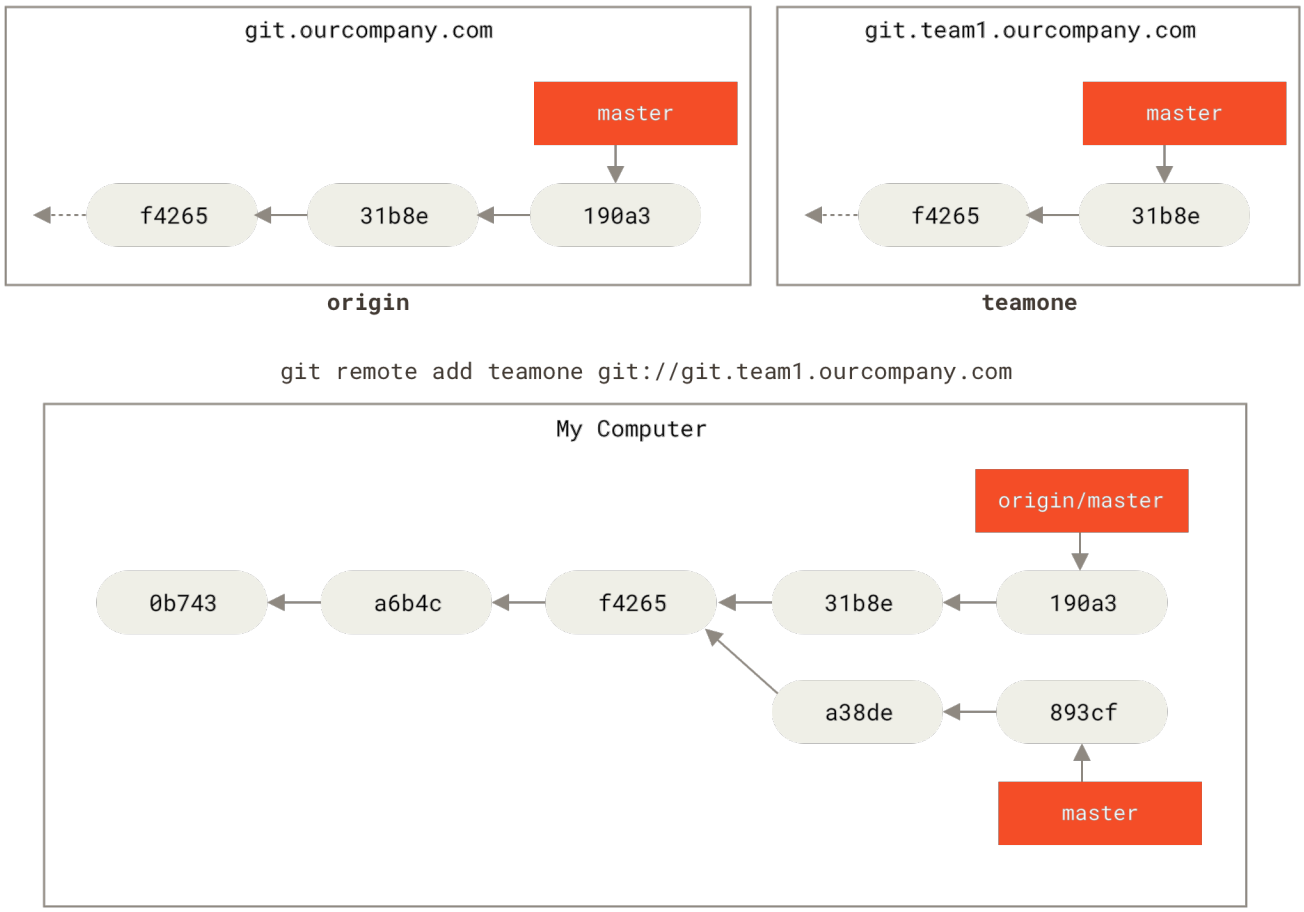


Figura 33. Adicionando outro servidor como remoto

Agora, você pode executar `git fetch teamone` para buscar tudo o que o servidor remoto `teamone` tem que você ainda não tem. Porque esse servidor tem um subconjunto dos dados que seu servidor `origin` tem agora, o Git não busca nenhum dado, mas define um branch remoto de rastreamento chamado `teamone/master` para apontar para o commit que `teamone` tem como seu branch `master`.

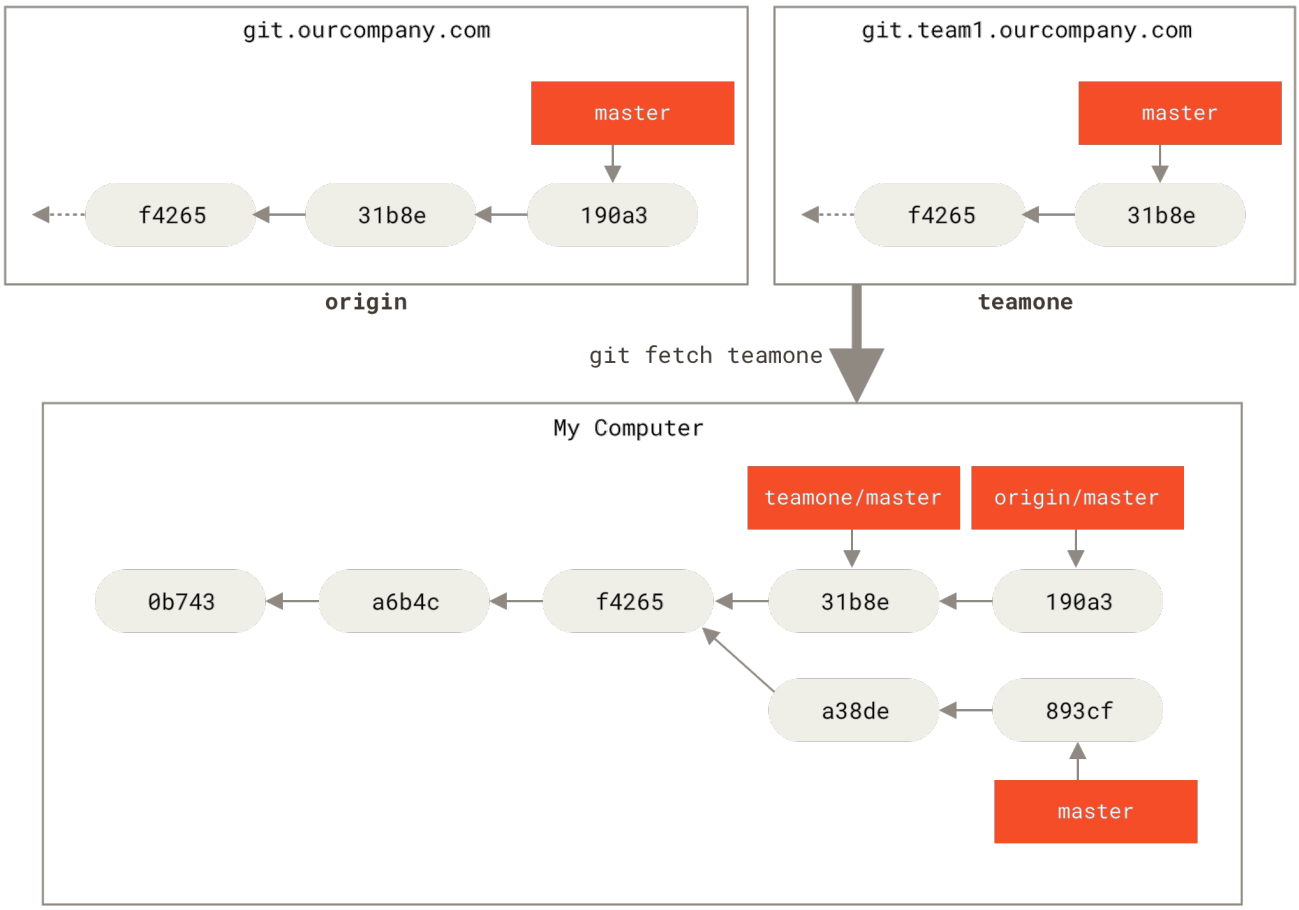


Figura 34. Branch remoto de rastreamento para `teamone/master`

Empurrando (Push)

Quando você deseja compartilhar um branch com o mundo, você precisa transferi-lo para um servidor remoto ao qual você tenha acesso de gravação. Seus branches locais não são sincronizados automaticamente com os branches remotos para os quais você escreve - você tem que empurrar explicitamente os branches que deseja compartilhar. Dessa forma, você pode usar branches privados para o trabalho que não deseja compartilhar e fazer o push apenas dos branches de tópicos nos quais deseja colaborar.

Se você tem um branch chamado `serverfix` que deseja trabalhar com outros, pode enviá-lo da mesma forma que fez o push do primeiro branch. Execute `git push <remote> <branch>`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Este é um atalho. O Git expande automaticamente o branch `serverfix` para `refs/heads/serverfix:refs/heads/serverfix`, o que significa, “Pegue meu branch local serverfix e

empurre-o para atualizar o branch serverfix remoto". Veremos a parte `refs/heads/` em detalhes em Funcionamento Interno do Git, mas geralmente você pode deixá-la desativada. Você também pode fazer `git push origin serverfix:serverfix`, que faz a mesma coisa - "Pegue meu serverfix e torne-o o serverfix remoto." Você pode usar esse formato para enviar um branch local para um branch remoto com um nome diferente. Se você não quiser que ele seja chamado de `serverfix` no remoto, você pode executar `git push origin serverfix:awesomebranch` para enviar seu branch local `serverfix` para o branch `awesomebranch` no projeto remoto.

Não digite sua senha todo o tempo

Se você estiver usando uma URL HTTPS para fazer push, o servidor Git solicitará seu nome de usuário e senha para autenticação. Por padrão, ele solicitará essas informações no terminal para que o servidor possa dizer se você tem permissão para fazer push.

NOTA

Se você não quiser digitá-lo toda vez que for fazer o push, você pode configurar um "credential cache". O mais simples é mantê-lo na memória por alguns minutos, o que você pode configurar facilmente executando `git config --global credential.helper cache`.

Para obter mais informações sobre as várias opções de "credential cache" disponíveis, consulte Credential Storage.

Na próxima vez que um de seus colaboradores buscar no servidor, eles obterão uma referência de onde a versão do servidor do `serverfix` está no branch remoto `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

É importante notar que quando você faz uma busca que desativa novos branches remotos de rastreamento, você não tem automaticamente cópias locais editáveis deles. Em outras palavras, neste caso, você não tem um novo branch `serverfix` - você só tem um ponteiro `origin/serverfix` que você não pode modificar.

Para mesclar este trabalho em seu branch atual, você pode executar `git merge origin/serverfix`. Se você quiser seu próprio branch `serverfix` com o qual possa trabalhar, pode basear-se em seu branch remoto:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Isso lhe dá um branch local no qual você pode trabalhar que inicia onde está o `origin/serverfix`.

Rastreando Branches

Fazer check-out de um branch local a partir de um branch remoto cria automaticamente o que é chamado de "tracking branch" (e o branch que ele rastreia é chamado de "branch upstream"). "Tracking branch" são branches locais que têm um relacionamento direto com um branch remoto. Se você estiver em um branch de rastreamento e digitar `git pull`, o Git saberá automaticamente de qual servidor buscar o branch para fazer o merge.

Quando você clona um repositório, geralmente ele cria automaticamente um branch `master` que rastreia `origin/master`. No entanto, você pode configurar outros branches de rastreamento se desejar - aqueles que rastreiam branches em outros branches remotos, ou não rastreiam o branch `master`. O caso simples é o exemplo que você acabou de ver, executando `git checkout -b [branch] [remotename]/[branch]`. Esta é uma operação suficiente para que o Git forneça a abreviação `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Na verdade, isso é tão comum que existe até um atalho para isso. Se o nome do branch que você está tentando verificar (a) não existe e (b) corresponde exatamente a um nome em apenas um repositório remoto, o Git criará um branch de rastreamento para você:

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Para configurar um branch local com um nome diferente do branch remoto, você pode usar facilmente a primeira versão com um nome de branch local diferente:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

Agora, seu branch local `sf` irá puxar automaticamente de `origin/serverfix`.

Se você já tem um branch local e deseja configurá-lo para um branch remoto que acabou de puxar, ou deseja alterar o branch upstream que está rastreando, você pode usar o método `-u` ou `--set-upstream-to` para `git branch` para configurá-lo explicitamente a qualquer momento.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

NOTA

O Atalho Upstream

Quando você tem um branch de rastreamento configurado, pode referenciar seu branch upstream com as abreviações `@{upstream}` ou `@{u}`. Então, se você está no branch `master` e está rastreando `origin/master`, você pode dizer algo como `git merge @{u}` ao invés de `git merge origin/master` se desejar.

Se você quiser ver quais branches de rastreamento você configurou, você pode usar a opção `-vv` para `git branch`. Isso listará seus branches locais com mais informações, incluindo o que cada filial está rastreando e se sua filial local está à frente, atrás ou ambos.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Portanto, aqui podemos ver que nosso branch `iss53` está rastreando `origin/iss53` e está “ahead” por dois, o que significa que temos dois commits localmente que não são enviados ao servidor. Também podemos ver que nosso branch `master` está rastreando `origin/master` e está atualizado. Em seguida, podemos ver que nosso branch `serverfix` está rastreando o branch `server-fix-good` em nosso servidor `teamone` e está à frente de três e atrás de um, o que significa que há um commit no servidor que não foi mesclado ainda e três commits localmente que não foram enviados por push. Finalmente, podemos ver que nosso branch `testing` não está rastreando nenhum branch remoto.

É importante observar que esses números são apenas desde a última vez que você fez um fetch em cada servidor. Este comando não chega aos servidores, ele está informando sobre o que armazenou localmente em cache. Se você quiser ficar totalmente atualizado com os números à frente e atrás, precisará buscar em todos os seus servidores remotos antes de executá-lo. Você poderia fazer isso assim:

```
$ git fetch --all; git branch -vv
```

Fazendo o Pull

Embora o comando `git fetch` baixe todas as alterações no servidor que você ainda não tem, ele não modificará seu diretório de trabalho. Ele simplesmente obterá os dados para você e permitirá que você mesmo os mescle. No entanto, existe um comando chamado `git pull` que é essencialmente um `git fetch` seguido imediatamente por um `git merge` na maioria dos casos. Se você tiver um branch de rastreamento configurado conforme demonstrado na última seção, seja explicitamente configurando-o ou tendo-o criado para você pelos comandos `clone` ou `checkout`, `git pull` irá procurar qual servidor e branch seu branch atual está rastreando, buscará naquele servidor e, em seguida, tentará mesclar nesse branch remoto.

Geralmente é melhor usar os comandos `fetch` e `merge` explicitamente, já que o `git pull` muitas vezes pode ser confuso.

Removendo Branches remotos

Suponha que você terminou com um branch remoto - digamos que você e seus colaboradores terminaram com um recurso e o fundiram no branch `master` do seu servidor remoto (ou em qualquer branch em que sua linha de código estável esteja). Você pode deletar um branch remoto usando a opção `--delete` para `git push`. Se você deseja excluir seu branch `serverfix` do servidor, execute o seguinte:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

Basicamente, tudo o que isso faz é remover o ponteiro do servidor. O servidor Git geralmente mantém os dados lá por um tempo até que seja removido definitivamente pelo Git, então, se ele foi excluído acidentalmente, geralmente é fácil de recuperar.

Rebase

No Git, existem duas maneiras principais de integrar as mudanças de um branch para outro: o `merge` e o `rebase`. Nesta seção, você aprenderá o que é o rebase, como fazê-lo, por que é uma ferramenta incrível e em que casos não vai querer usá-la.

O básico do Rebase

Se você voltar a um exemplo anterior de Mesclagem Básica, você pode ver que o seu trabalho divergiu e fez commits em dois branches diferentes.

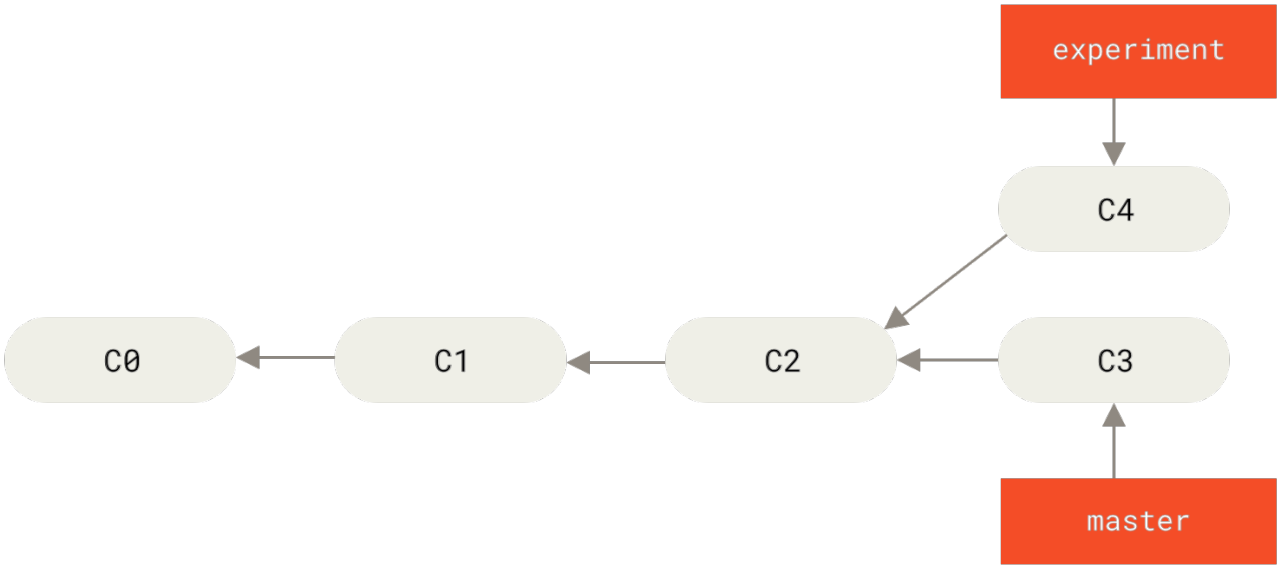


Figura 35. Um simples histórico de divergência

A maneira mais fácil de integrar os branches, como já vimos, é o comando `merge`. Ele realiza uma fusão de três vias entre os dois últimos snapshots de branch (`C3` e `C4`) e o ancestral comum mais recente dos dois (`C2`), criando um novo snapshot (e commit).

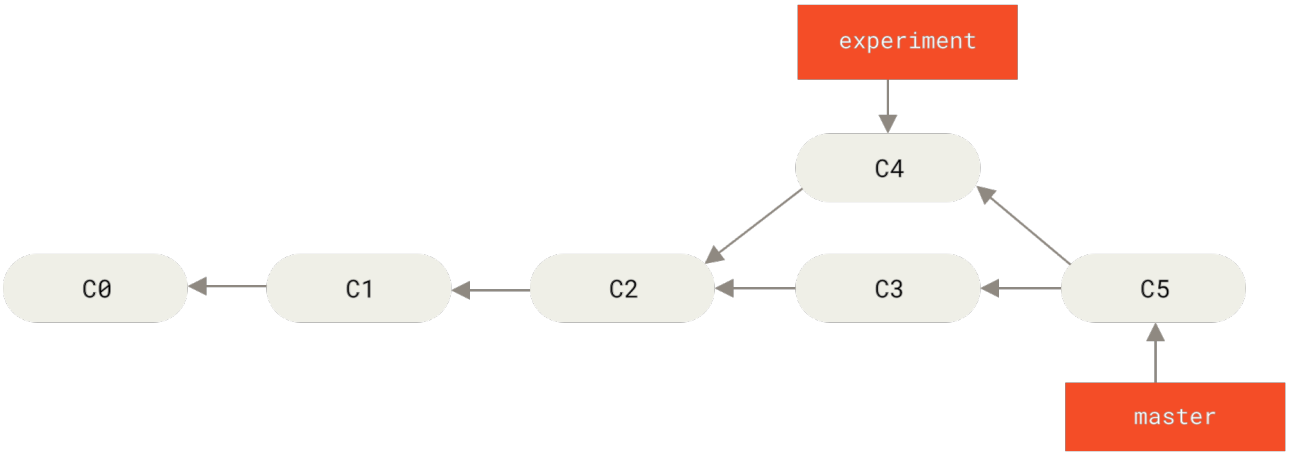


Figura 36. Fazendo um merge para integrar áreas de trabalho que divergiram

No entanto, há outra maneira: você pode pegar o patch da mudança que foi introduzida no `C4` e reaplicá-lo em cima do `C3`. No Git, isso é chamado de *rebasing*. Com o comando `rebase`, você pode pegar todas as alterações que foram confirmadas em um branch e reproduzi-las em outro.

Neste exemplo, você executaria o seguinte:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Ele funciona indo para o ancestral comum dos dois branches (aquele em que você está e aquele em que você está fazendo o rebase), obtendo o diff introduzido por cada commit do branch em que você está, salvando esses diffs em arquivos temporários, redefinindo o branch atual para o mesmo commit do branch no qual você está fazendo o rebase e, finalmente, aplicando cada mudança por vez.

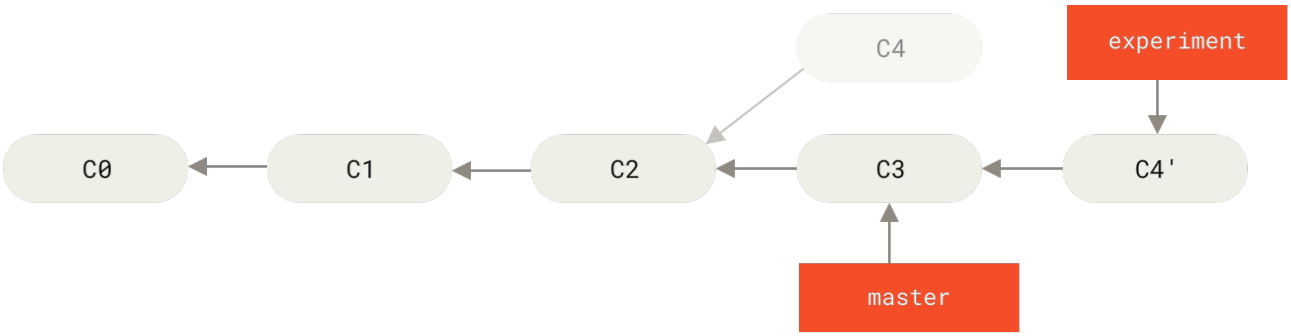


Figura 37. Fazendo o Rebase da mudança introduzida no `C4` em `C3`

Neste ponto, você pode voltar ao branch `master` e fazer uma fusão rápida.

```
$ git checkout master
$ git merge experiment
```

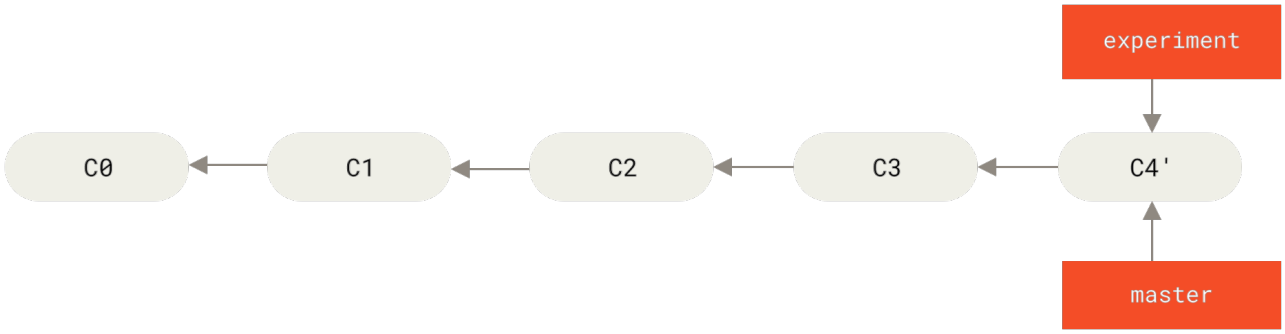


Figura 38. Fazendo uma fusão rápida no branch master

Agora, o snapshot apontado por `C4'` é exatamente o mesmo que foi apontado por `C5` no exemplo de merge. Não há diferença no produto final da integração, mas o rebase contribui para um histórico mais limpo. Se você examinar o log de um branch que foi feito rebase, parece uma registro linear: parece que todo o trabalho aconteceu em série, mesmo quando originalmente aconteceu em paralelo.

Frequentemente, você fará isso para garantir que seus commits sejam aplicados de forma limpa em um branch remoto - talvez em um projeto para o qual você está tentando contribuir, mas que não mantém. Neste caso, você faria seu trabalho em um branch e então realocaria seu trabalho em `origin/master` quando estivesse pronto para enviar seus patches para o projeto principal. Dessa forma, o mantenedor não precisa fazer nenhum trabalho de integração - apenas um fusão rápida ou uma aplicação limpa.

Observe que o snapshot apontado pelo commit final com o qual você termina, seja o último dos commits para um rebase ou o commit final de mesclagem após um merge, é o mesmo snapshot - é apenas o histórico que é diferente. O Rebase reproduz as alterações de uma linha de trabalho para outra na ordem em que foram introduzidas, enquanto a mesclagem pega os finais e os mescla.

Rebases mais interessantes

Você também pode fazer o replay do rebase em algo diferente do branch de destino. Pegue um histórico como Um histórico com um tópico de branch de outro branch, por exemplo. Você ramificou um branch de tópico (`server`) para adicionar alguma funcionalidade do lado do servidor ao seu projeto e fez um commit. Então, você ramificou isso para fazer as alterações do lado do cliente (`client`) e fez commit algumas vezes. Finalmente, você voltou ao seu branch de servidor e fez mais alguns commits.

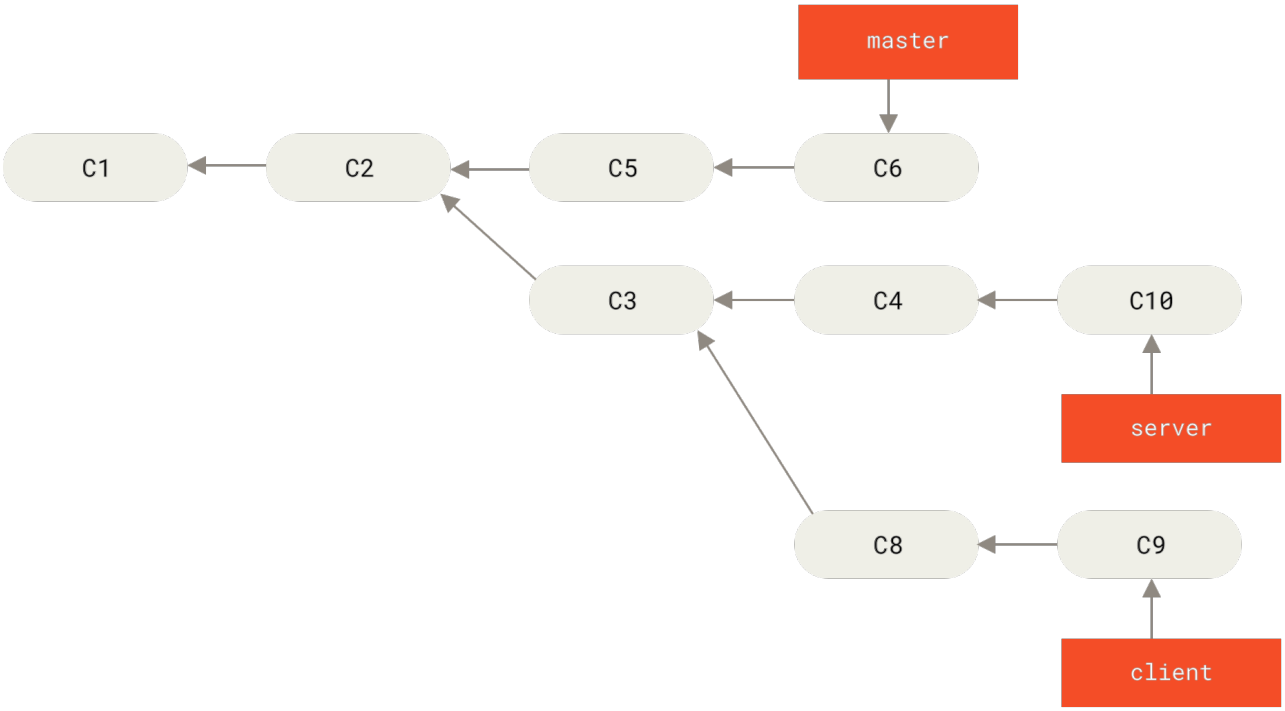


Figura 39. Um histórico com um tópico de branch de outro branch

Suponha que você decida que deseja mesclar suas alterações do lado do cliente em sua linha principal para um lançamento, mas deseja adiar as alterações do lado do servidor até que seja testado mais profundamente. Você pode pegar as mudanças no cliente que não estão no servidor (`C8` e `C9`) e reproduzi-las em seu branch `master` usando a opção `--onto` do `git rebase`:

```
$ git rebase --onto master server client
```

Isso basicamente diz: "Pegue o branch `client`, descubra os patches, desde que divergiu do branch `server`, e repita esses patches no branch `client` como se fosse baseado diretamente no branch `master`". É um pouco complexo, mas o resultado é bem legal.

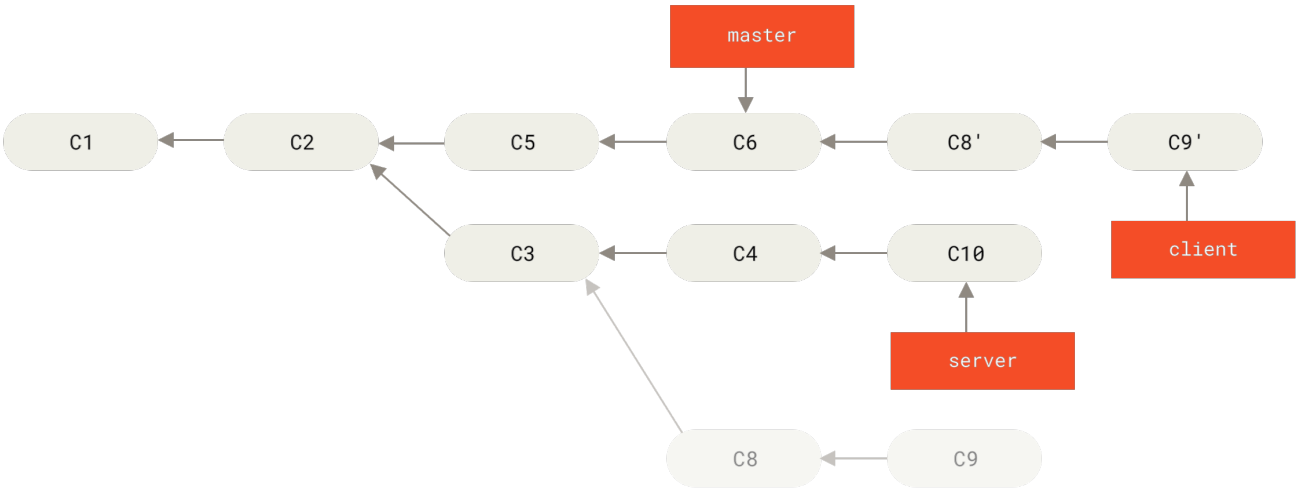


Figura 40. Rebase o tópico de um branch de outro branch

Agora, você pode fazer uma fusão rápida no branch master (veja Avanço rápido de seu branch principal para incluir as alterações da branch do cliente):

```
$ git checkout master
$ git merge client
```

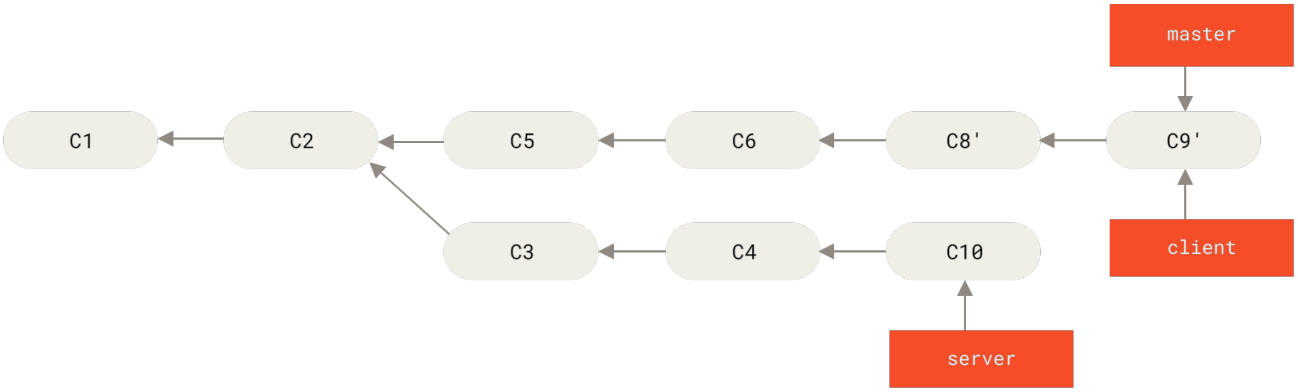


Figura 41. Avanço rápido de seu branch principal para incluir as alterações da branch do cliente

Digamos que você decida puxar seu branch de servidor também. Você pode realocar o branch do servidor no branch `master` sem ter que verificá-lo primeiro executando `git rebase [basebranch] [topicbranch]` - que verifica o branch do tópico (neste caso, `server`) para você e repete no branch base (`master`):

```
$ git rebase master server
```

Isso reproduz o trabalho do `server` em cima do trabalho do `master`, como mostrado em Rebase o branch server por cima do branch master.

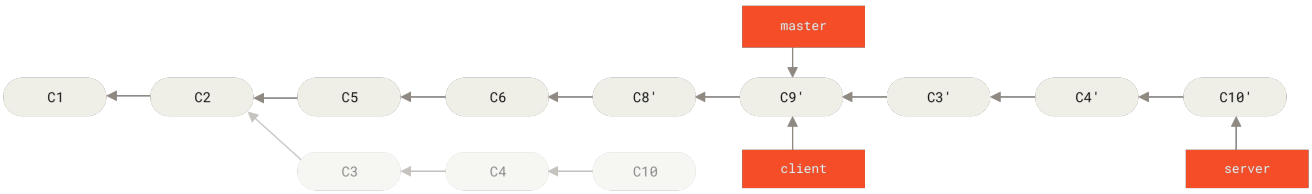


Figura 42. Rebase o branch server por cima do branch master

Então, você pode avançar o branch base (`master`):

```
$ git checkout master
$ git merge server
```

Você pode remover os branches `client` e `server` porque todo o trabalho foi integrado e você não precisa mais deles, deixando seu histórico para todo o processo parecido com Histórico final de commits:

```
$ git branch -d client
$ git branch -d server
```

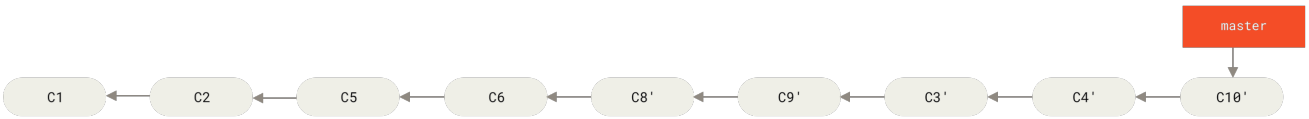


Figura 43. Histórico final de commits

Os perigos do Rebase

Ahh, mas a felicidade do rebase não vem sem suas desvantagens, que podem ser resumidas em uma única linha:

Não faça rebase de commits que existam fora do seu repositório.

Se você seguir essa diretriz, ficará bem. Do contrário, as pessoas irão odiá-lo e você será desprezado por amigos e familiares.

Quando você faz o rebase, você está abandonando commits existentes e criando novos que são semelhantes, mas diferentes. Se você enviar commits para algum lugar e outras pessoas fizerem o pull e basearem seu trabalho neles, e então você reescrever esses commits com `git rebase` e enviá-los novamente, seus colaboradores terão que fazer um novo merge em seus trabalhos e as coisas ficarão complicadas quando você tentar puxar o trabalho deles de volta para o seu.

Vejamos um exemplo de como o rebase de um trabalho que você tornou público pode causar problemas. Suponha que você clone de um servidor central e faça algum trabalho a partir dele. Seu histórico de commits é parecido com este:

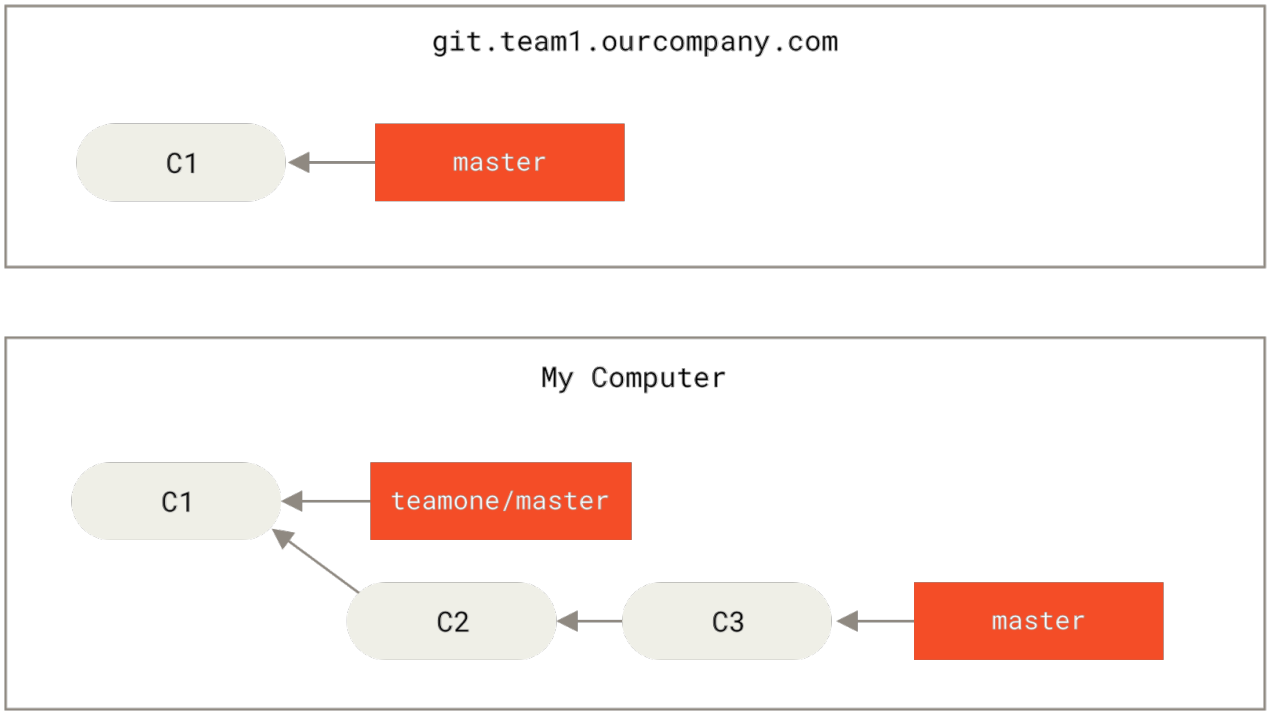


Figura 44. Fazendo o clone de um repositório e trabalhando com ele

Agora, outra pessoa faz mais alterações que inclui um merge e envia esse trabalho para o servidor central. Você o busca e mescla o novo branch remoto em seu trabalho, fazendo com que seu histórico se pareça com isto:

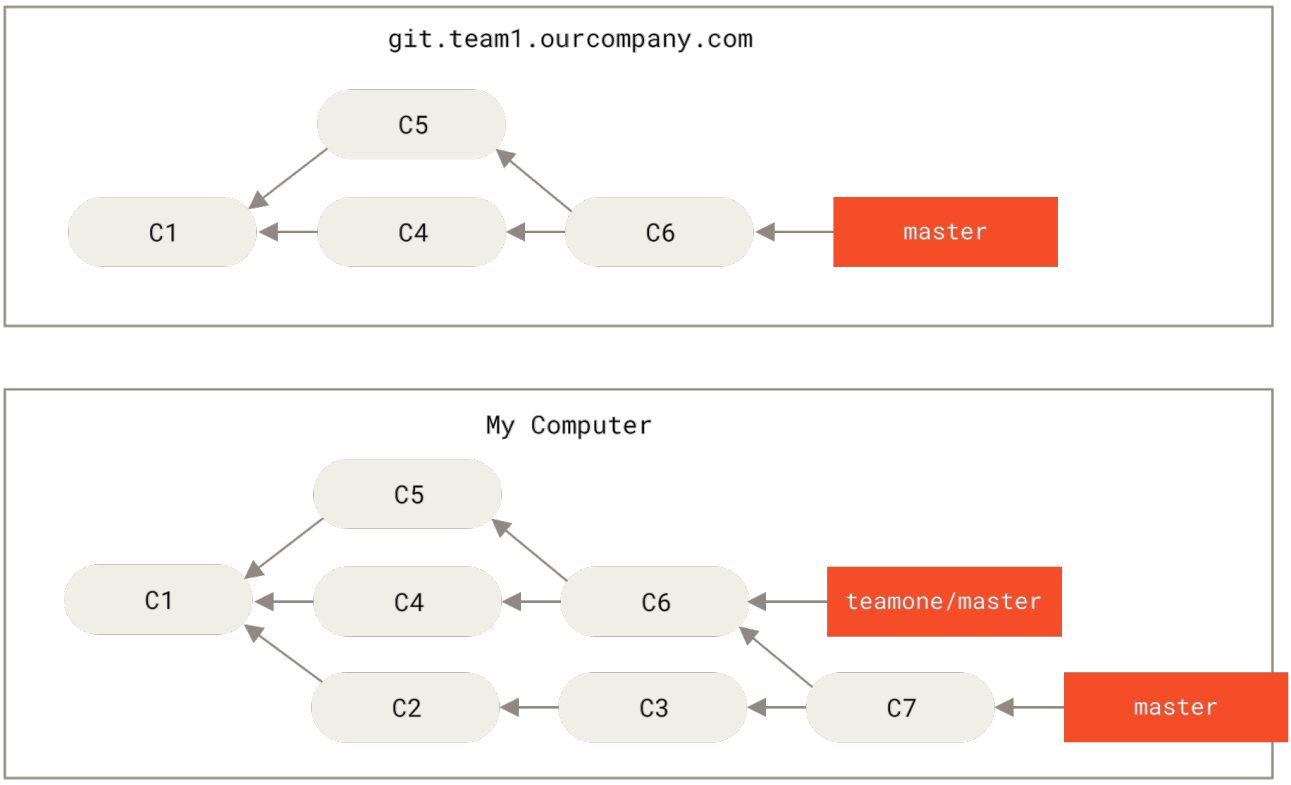


Figura 45. Buscar mais commits e fazer merge em seu trabalho

Em seguida, a pessoa que enviou o trabalho mesclado decide voltar atrás e realizar um rebase no trabalho em vez disso; ele faz um `git push --force` para sobrescrever o histórico no servidor. Você então busca daquele servidor, derrubando os novos commits.

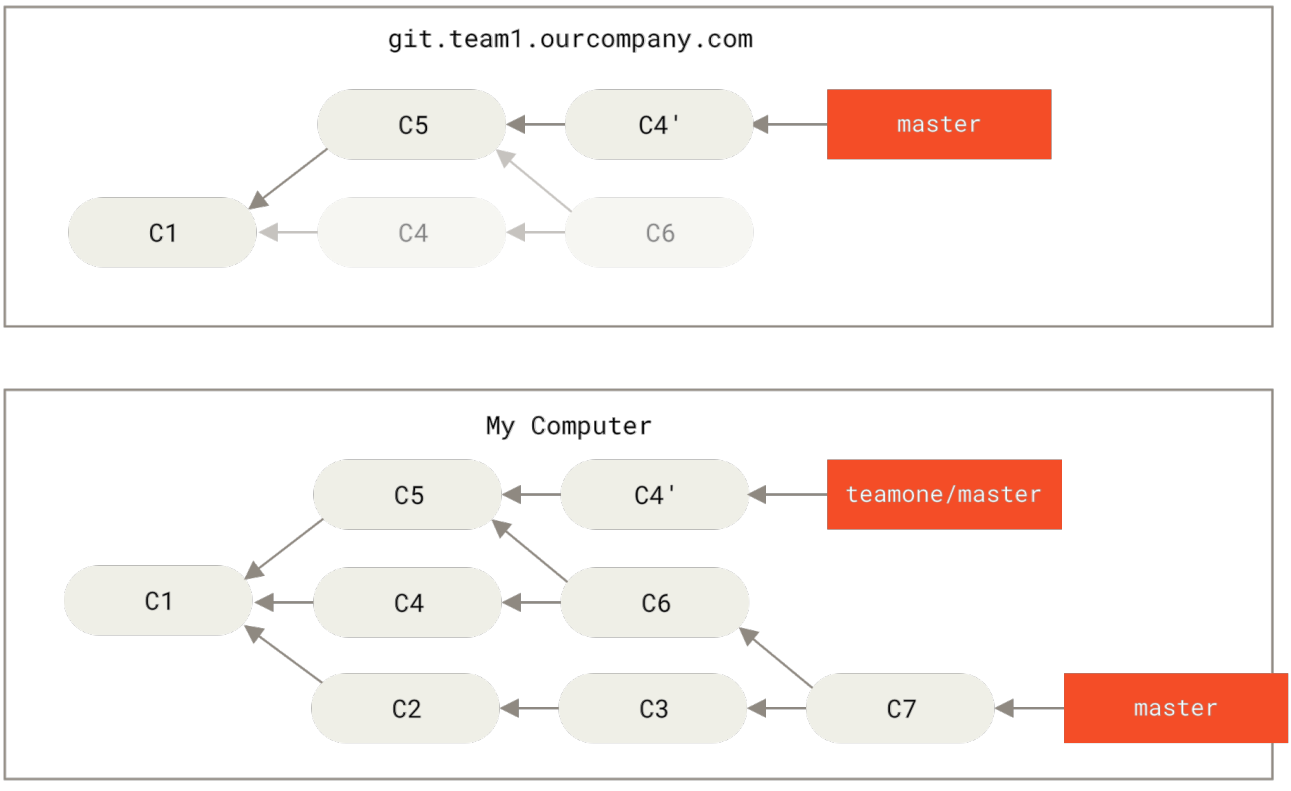


Figura 46. Alguém empurra commits que foram feitos rebase, abandonando commits nos quais você baseou seu trabalho

Agora você está em apuros. Se você fizer um `git pull`, você criará um commit de merge que inclui as duas linhas do histórico, e seu repositório ficará assim:

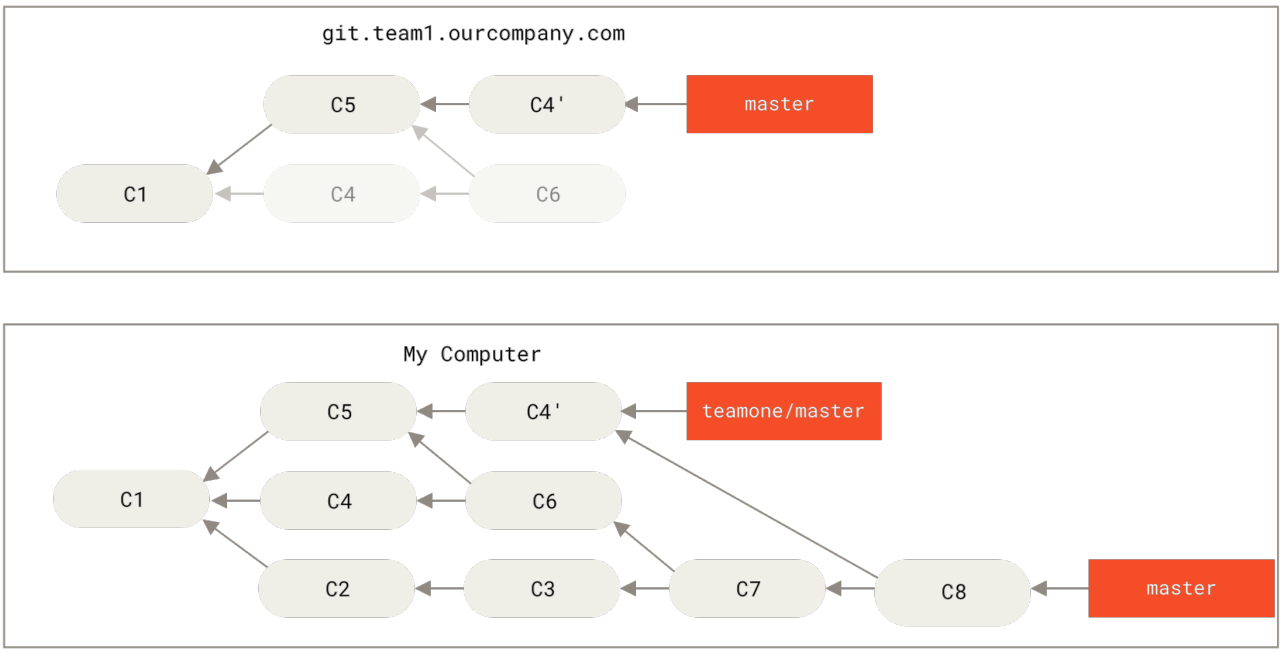


Figura 47. Você faz o merge do mesmo trabalho novamente em um novo commit de merge

Se você executar um `git log` quando seu histórico estiver assim, você verá dois commits com o mesmo autor, data e mensagem, o que será confuso. Além disso, se você enviar esse histórico de volta ao servidor, reintroduzirá todos os commits realocados no servidor central, o que pode confundir ainda mais as pessoas. É bastante seguro assumir que o outro desenvolvedor não quer que `C4` e `C6` apareçam na história; é por isso que eles fizeram um rebase antes.

Rebase quando vocês faz Rebase

Se você **realmente** se encontrar em uma situação como essa, o Git tem mais alguma mágica que pode te ajudar. Se alguém em sua equipe forçar mudanças que substituam o trabalho no qual você se baseou, seu desafio é descobrir o que é seu e o que eles reescreveram.

Acontece que, além da soma de verificação SHA-1 de commit, o Git também calcula uma soma de verificação que é baseada apenas no patch introduzido com a confirmação. Isso é chamado de "patch-id".

Se você puxar o trabalho que foi reescrito e fazer o rebase sobre os novos commits de seu parceiro, o Git pode muitas vezes descobrir o que é exclusivamente seu e aplicá-lo de volta ao novo branch.

Por exemplo, no cenário anterior, se em vez de fazer uma fusão quando estamos em Alguém empurra commits que foram feitos rebase, abandonando commits nos quais você baseou seu trabalho executarmos `git rebase teamone/master`, Git irá:

- Determinar qual trabalho é exclusivo para nosso branch (C2, C3, C4, C6, C7)
- Determinar quais não são confirmações de merge (C2, C3, C4)
- Determinar quais não foram reescritos no branch de destino (apenas C2 e C3, uma vez que C4 é o mesmo patch que C4')

- Aplicar esses commits no topo de `teamone/master`

Então, em vez do resultado que vemos em Você faz o merge do mesmo trabalho novamente em um novo commit de merge, acabaríamos com algo mais parecido com Rebase on top of force-pushed rebase work..

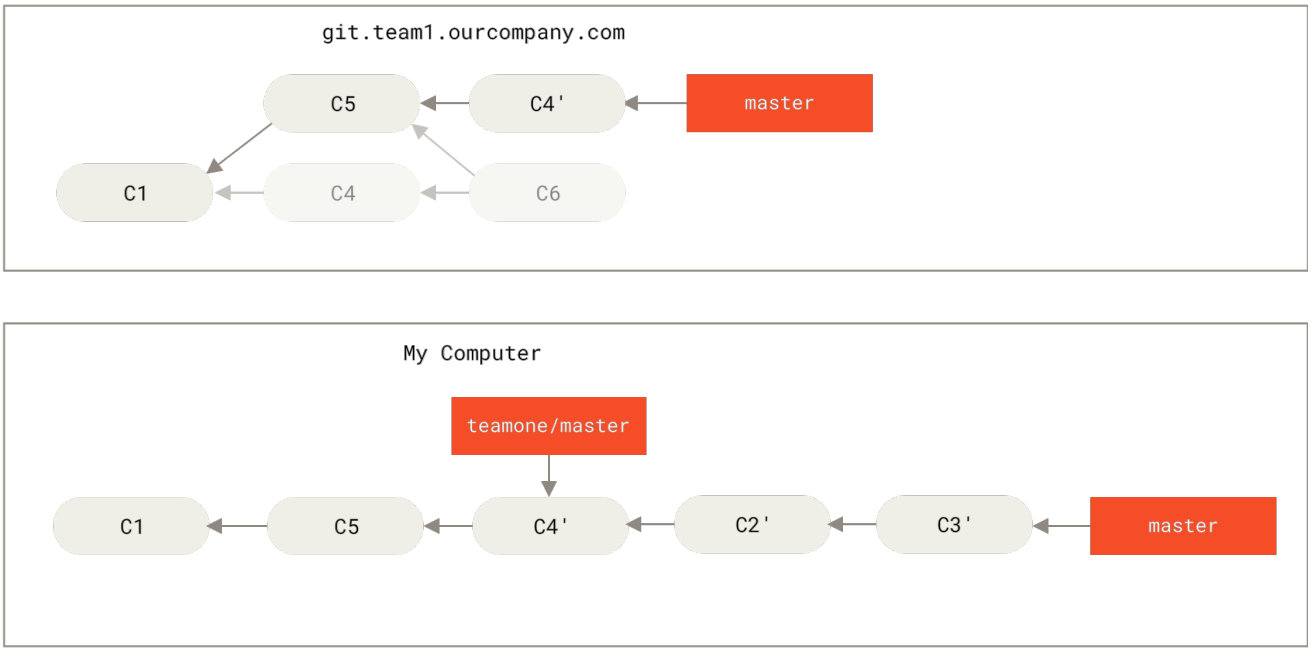


Figura 48. Rebase on top of force-pushed rebase work.

Isso só funciona se C4 e C4' que seu parceiro fez forem quase exatamente o mesmo patch. Caso contrário, o rebase não será capaz de dizer o que é uma duplicata e adicionará outro patch semelhante ao C4 (que provavelmente não será aplicado, uma vez que as alterações já estariam pelo menos um pouco lá).

Você também pode simplificar isso executando um `git pull --rebase` em vez de um `git pull` normal. Ou você poderia fazer isso manualmente com um `git fetch` seguido por um `git rebase teamone/master` neste caso.

Se você estiver usando `git pull` e quiser tornar `--rebase` o padrão, você pode definir o valor de configuração `pull.rebase` com algo como `git config --global pull.rebase true`.

Se você tratar o rebase como uma forma de limpar e trabalhar com commits antes de enviá-los, e se você apenas fazer o rebase dos commits que nunca estiveram disponíveis publicamente, então você ficará bem. Se você fizer o rebase dos commits que já foram enviados publicamente, e as pessoas podem ter baseado o trabalho nesses commits, então você pode ter alguns problemas frustrantes e o desprezo de seus companheiros de equipe.

Se você ou um parceiro achar necessário em algum ponto, certifique-se de que todos saibam executar `git pull --rebase` para tentar tornar a dor depois que ela acontecer um pouco mais simples.

Rebase vs. Merge

Agora que você viu o rebase e o merge em ação, pode estar se perguntando qual é o melhor. Antes de respondermos, vamos voltar um pouco e falar sobre o que a história significa.

Um ponto de vista sobre isso é que o histórico de commit do seu repositório é um **registro do que realmente aconteceu.** É um documento histórico, valioso por si só, e não deve ser alterado. Desse ângulo, mudar o histórico de commits é quase uma blasfêmia; você está mentindo sobre o que realmente aconteceu. E daí se houvesse uma série confusa de commits de merge? Foi assim que aconteceu, e o repositório deve preservar isso para a posteridade.

O ponto de vista oposto é que o histórico de commits é a **história de como seu projeto foi feito.** Você não publicaria o primeiro rascunho de um livro, e o manual de como manter seu software merece uma edição cuidadosa. Este é o campo que usa ferramentas como rebase e filter-branch para contar a história da maneira que for melhor para futuros leitores.

Agora, à questão de saber se merge ou rebase é melhor: espero que você veja que não é tão simples. O Git é uma ferramenta poderosa e permite que você faça muitas coisas para e com sua história, mas cada equipe e cada projeto são diferentes. Agora que você sabe como essas duas coisas funcionam, cabe a você decidir qual é a melhor para sua situação específica.

Em geral, a maneira de obter o melhor dos dois mundos é fazer o rebase nas mudanças locais que você fez, mas não compartilhou ainda antes de empurrá-las para limpar seu histórico, mas nunca faça rebase em algo que você empurrou em algum lugar.

Sumário

Nós iremos mostrar as funções de Branches e Merges básicas no Git. Sinta-se confortável para criar e alternar para novos branches, alternar entre branches e mesclar branches locais. Você também deve ser capaz de compartilhar seus branches enviando-os (push) para um servidor compartilhado, trabalhando com outras pessoas em branches compartilhados e rebaseando (rebase) seus branches antes de serem compartilhados. A seguir, vamos apresentar o que você precisa para executar seu próprio servidor para hospedagem do repositório Git.

Git no servidor

Nesse ponto, você deve ser capaz de fazer a maioria das suas tarefas diárias usando Git. Contudo, para fazer qualquer colaboração no Git, você precisará de um repositório remoto Git. Embora tecnicamente você possa enviar e baixar alterações de repositórios individuais, isso é desencorajado porque você provavelmente confundirá o que está sendo trabalhando em cada um deles se você não for cuidadoso.

Além disso, você quer que seus colaboradores sejam capazes de acessar seu repositório mesmo se seu computador estiver offline - geralmente é útil ter um repositório comum mais confiável. Portanto, o método preferido para colaborar com alguém é definir um repositório intermediário que vocês possam enviar e baixar dele.

Executar um servidor Git é bastante simples. Primeiro, você escolhe quais protocolos de comunicação que você quer que seu servidor utilize. A primeira seção desse capítulo cobrirá os protocolos disponíveis e seus prós e contras. As próximas seções explicarão algumas configurações típicas usando estes protocolos e como fazer seu servidor executá-las. Por último, nós veremos algumas opções de hospedagem, caso você não se importe de hospedar seu código no servidor de outra pessoa e não quiser lidar com problemas configurando e mantendo seu próprio servidor.

Se você não tem interesse em executar seu próprio servidor, você pode pular para a última seção deste capítulo para ver algumas opções para configurar uma conta hospedada e passar para o próximo capítulo, onde nós discutimos vários prós e contras de trabalhar em ambientes de controle de código distribuídos.

Um repositório remoto geralmente é repositório vazio (*bare repository*) - um repositório Git que não tem um diretório de trabalho. Porque o repositório é usado apenas como ponto de colaboração, não há razão para ter uma cópia verificada em disco; são apenas dados do Git. Simplificando, o repositório vazio (bare repository) é o conteúdo do diretório `.git` do seu projeto e nada mais.

Os Protocolos

Git pode usar os quatro principais protocolos de transferência de dados: Local, HTTP, Secure Shell (SSH) e Git. Aqui nós discutiremos o que eles são e em que circunstâncias você poderia querer (ou não) usá-los.

Protocolo Local

O mais básico é o *Protocolo Local*, em que o repositório remoto é outro diretório no disco. Isso é frequentemente usado se todo seu time tem acesso à um sistema de arquivos compartilhado como uma montagem NFS, ou menos provavelmente no caso em que todos acessem o mesmo computador. O último não seria o ideal, porque todas as intâncias do seu repositório de código ficariam no mesmo computador, fazendo com que uma perda catastrófica seja muito mais provável.

Se você tem um sistema de arquivos montado e compartilhado, então você pode clonar, enviar e

baixar do repositório local. Para clonar um repositório como esse ou adicionar um como remoto em um projeto existente, use o caminho para o repositório como uma URL.

Por exemplo, para clonar um repositório local, você pode executar algo assim:

```
$ git clone /srv/git/project.git
```

Ou você pode fazer assim:

```
$ git clone file:///srv/git/project.git
```

Git funciona um pouco diferente se você explicitamente especificar `file://` no começo da URL. Se você apenas especificar o caminho, o Git tenta usar caminhos absolutos ou diretamente copiar os arquivos que ele precisa. Se você especificar `file://`, o Git dispara um processo que ele normalmente usa para transferir dados pela rede que normalmente é um método de transferência de dados muito menos eficiente. O principal motivo para especificar o prefixo `file://` é se você quer uma cópia limpa do repositório com referências estranhar ou objetos deixados de fora – geralmente depois de importar de outro sistema de controle de versão or algo similar (veja Funcionamento Interno do Git para tarefas de manutenção). Nós vamos usar o caminho normal porque fazendo isso é quase sempre mais rápido.

Para adicionar um repositório local em um projeto Git já existente, você pode executar algo assim:

```
$ git remote add local_proj /srv/git/project.git
```

Então, você pode enviar e baixar desse repositório como se você estivesse fazendo pela rede.

Os Prós

Os prós dos repositórios baseados em arquivos são que eles são simples e usam permissões de arquivo e acessos de rede já existentes. Se você já tem um sistema de arquivos compartilhado que seu time inteiro tem acesso, configurar um repositório é muito fácil. Você cola a cópia do repositório vazio (bare repository) em algum lugar que todos tem acesso para ler/escrever permissões como você faria com qualquer outro diretório.

Nós discutiremos como exportar uma cópia de um repositório vazio (bare repository) para esse propósito em Getting Git on a Server.

Esta é uma boa opção para rapidamente pegar o trabalho do repositório de trabalho de outra pessoa. Se você ou seu colega de trabalho estão trabalhando no mesmo projeto e ele quiser que você verifique alguma coisa, executar um comando como `git pull /home/john/project` geralmente é mais fácil que enviar para um servidor remoto e baixar.

Os Contras

Os contras desse método são que geralmente acesso compartilhado é mais difícil de configurar e acessar de múltiplos locais do que acessos básicos à rede. Se você quer enviar do seu notebook

quando você estiver em casa , você tem que montar uma unidade remota, que pode pode ser mais difícil e lenta comparado ao acesso baseado em rede.

É importante mencionar que essa não é necessariamente a opção mais rápida se você está usando montagens compartilhadas de alguma forma. Um repositório local é mais rápido apenas se você tiver acesso aos dados. Um repositório NFS geralmente é mais lento que um repositório em SSH no mesmo servidor, permitindo que o Git execute discos locais em cada sisteme.

Finalmente, este protocolo não prote o repositório contra danos acidentais. Todo usuário tem acesso total no diretório "remoto" pelo shell, e isso não os previne de mudar ou remover arquivos internos do Git e corromper o repositório.

O Protocolo HTTP

Git pode se comunicar por HTTP em dois jeitos diferentes.

Antes do Git 1.6.6 havia apenas uma maneira de fazer isso, que era muito simples e geralmente somente leitura. Na versão 1.6.6 foi introduzido um protocolo novo e mais inteligente, que tornava o Git capaz de inteligentemente negociar a transferência de dados de modo similar ao que faz por SSH. Nos últimos anos, esse novo protocolo HTTP se tornou muito popular por ser mais simples para o usuário e mais inteligente na forma como se comunica. A versão mais recente é gerealmente referida como protocolo HTTP "Smart" e a anterior como HTTP "Dumb". Nós cobriremos o mais novo protocolo HTTP "Smart" primeiro.

HTTP Smart

O protocolo HTTP "Smart" funciona muito semelhantemente ao protocolo SSH ou Git, mas funciona no padrão das portas HTTP/S e pode usar vários mecanismos de autenticação HTTP, isso significa que geralmente é mais fácil para o usuário do que outros, como SSH, já que você pode usar coisas como usuário e senha para autenticação ao invés de ter que configurar chaves SSH.

Ele se tornou provavelmente o jeito mais popular de usar o Git agora, já que pode ser configurado para servir aninomamente como protocolo `git://`, e também pode ser enviado com autenticação e criptografia como o protocolo SSH. Ao invés de ter que configurar diferentes URLs para essas coisas, você pode usar uma única URL para ambos. Se você tentar enviar e o repositório requerer autenticação (o que ele normalmente deveria fazer), o servidor pode pedir usuário e senha. O mesmo vale para acessos de leitura.

De fato, para serviços como GitHub, a URL que você vê o repositório online (por exemplo, "https://github.com/schacon/simplegit[]") é a mesma URL que você usa para clonar e, se você tiver acesso, enviar.

HTTP Dumb

Se o servidor não responder com serviço Git HTTP Smart, o cliente Git vai tentar voltar para o protocolo mais simples, o HTTP "Dumb". O protocolo Dumb espera que o repositório vazio (bare

repository) do Git sirva os arquivos como um servidor web normal. A beleza do protocolo HTTP Dumb é sua simplicidade de configuração. Basicamente, tudo que você tem que fazer é colocar o repositório vazio (bare repository) sob o documento raiz do seu HTTP e configurar um hook `post-update` específico, e você está pronto (Veja Git Hooks). Nesse ponto, qualquer um que possa acessar o servidor web no qual você colocou o repositório também pode cloná-lo. Para acessos de leitura para seu repositório por HTTP, faça algo como:

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Isso é tudo. O hook `post-update` que vem com o Git por padrão executa o comando apropriado (`git update-server-info`) para fazer uma busca HTTP e clonagem do trabalho propriamente. Esse comando executa quando você envia para esse repositório (por SSH talvez); então, outras pessoas podem cloná-lo com algo como

```
$ git clone https://example.com/gitproject.git
```

Nesse caso em particular, nós estamos usando o caminho `/var/www/htdocs` que é comum para configurações Apache, mas você pode usar qualquer servidor web estático – apenas colocando o repositório vazio (bare repository) nesse caminho. Os dados do Git são disponibilizados como simples arquivos estáticos (veja Funcionamento Interno do Git para mais detalhes sobre como exatamente eles são disponibilizados).

Geralmente você escolheria executar um servidor HTTP Smart de leitura/gravação ou simplesmente ter os arquivos acessíveis como somente leitura da maneira burra. É raro executar uma mistura dos dois serviços.

Os Prós

Nós vamos nos concentrar nos prós da versão Smart do protocolo HTTP.

A simplicidade de ter uma única URL para todos os tipos de acesso e ter o prompt do servidor apenas quando a autenticação é necessária torna as coisas muito fáceis para o usuário final. Ser capaz de se autenticar com usuário e senha é também grande vantagem sobre o SSH, já que usuários não precisam gerar uma chave SSH localmente e fazer upload de sua chave pública para o servidor antes de ser capaz de interagir com ele. Para usuários menos sofisticados, os usuários de sistemas onde SSH é menos comum, esta é a maior vantagem na usabilidade. Ele também é um protocolo muito rápido e eficiente, parecido com o próprio SSH.

Você também pode disponibilizar seus repositórios somente-leitura por HTTPS, que significa que você pode encriptar a transferência de conteúdo; ou você pode ir mais longe e fazer os clientes usarem certificados SSL assinados específicos.

Outra coisa legal é que HTTP/S são protocolos tão comumente usado que firewalls corporativos

costumam ser configurados para permitir o tráfego por essas portas.

Os Contras

Por HTTP/S o Git pode ser um pouco complicado para configurar comparado com alguns servidores SSH. Fora isso, há muitas pequenas vantagens que outros protocolos tem sobre o protocolo HTTP "Smart" para servir o Git.

Se você está usando HTTP para envios autenticados, inserir suas credenciais é algumas vezes mais complicado que usar chaves por SSH. Há contudo várias ferramentas para armazenar credenciais que você pode usar, incluindo o acesso por Keychain no OSX e o Gerenciados de Credenciais no Windows, para fazer isso bastante indolor. Leia Credential Storage para ver como configurar um gerenciador de credenciais no seu sistema.

O Protocolo SSH

Um protocolo comum de transporte quando o Git está auto hospedado é o SSH. Isso é porque o acesso SSH para servidores já é configurado na maioria dos lugares – e se não for, é fácil fazê-lo. SSH também é um protocolo de rede autenticada; e por causa disso ele é onipresente, ele é geralmente mais fácil de configurar e usar.

Para clonar um repositório Git por SSH você pode especificar a URL ssh:// assim:

```
$ git clone ssh://user@server/project.git
```

Ou para encurtar você pode usar a sintáxe scp para o protocolo SSH:

```
$ git clone user@server:project.git
```

Você também pode não especificar o usuário e o Git assumirá que o usuário é o atualmente logado.

Os Prós

São muitos os prós de usar SSH. Primeiro, SSH é relativamente fácil de configurar – serviços SSH são comuns, muitos administradores de rede tem experiência com eles, e muitas distribuições de SO são configurados com eles ou tem ferramentas para gerenciá-los. Depois, acessar por SSH é seguro – toda a transferência de dados é criptografada e autenticada. Por último, como HTTP/S, Git e o protocolo Local, SSH é eficiente, compactando os dados o quanto possível antes de transferí-los.

Os Contras

O aspecto negativo de SSH é que você não pode usar acesso anômino ao seu repositório por ele. As pessoas precisam ter acesso à sua máquina por SSH para acessá-la, mesmo que seja em modo somente leitura, o que não torna o acesso SSH propício para projetos de código aberto. Se você está usando ele somente na sua rede corporativa, o SSH pode ser o único protococolo que você precisa para ligar com isso. Se você quer permitir acesso anonimo somente leitura para seus projetos e

também quer usar SSH, você precisará configurar o SSH para você enviar, mas outra coisa para os outros buscarem.

O Protocolo Git

O próximo é o protocolo Git. Esse serviço especial vem empacotado com Git; Ele escuta uma porta dedicada (9418) que provê um serviço parecido com o protocolo SSH, mas absolutamente sem autenticação. Para um repositório ser usado pelo protocolo Git, você precisa criar o arquivo `git-daemon-export-ok` – o serviço não usará o repositório sem que o arquivo esteja nele – mas não há qualquer segurança além disso. O repositório Git estará disponível para todo mundo para clonar ou não está. Isso significa que não haverá download por esse protocolo. Você pode habilitar o acesso para envio; mas dada a falta de autenticação, se você você habilitar o envio, qualquer um na internet poderia encontrar a URL dos seus projetos e enviar para eles. Basta dizer que isso é raro.

Os prós

O protocolo Git geralmente é o protocolo de rende mais rápido disponível. Se você está usando muito tráfego para um projeto público ou trabalhando em um projeto muito grande que não requer atenticação dos usuários para acesso de leitura, é provável que você quererá definir um serviço Git para seu projeto. Ele usa o mesmo mecanismo de transferência de dados que o protocolo SSH, mas sem sobrecarga de criptografia ou atenticação.

Os Contra

A desvantagem do protocolo Git é a ausência de autenticação. Geralmente, é indesejável que o protocolo Git seja o único acesso ao seu projeto. Geralmente você o usará em conjunto com um acesso SSH ou HTTPS para alguns desenvolvedores que terão acesso para enviar e todos os outros usarão `git://` para acesso somente leitura. Ele também é o provavelmente o protocolo mais difícil de configurar. Ele precisa executar seu próprio serviço, o que requer configuração `xinetd` ou similar, o que não é sempre fácil. Ele também querer um acesso do firewall à porta 9418, o que não é uma porta que por padrão os firewall corporativos sempre permitem. Grandes firewalls corporativos normalmente comumente bloqueiam essa porta obscura.

Getting Git on a Server

Now we'll cover setting up a Git service running these protocols on your own server.

NOTA	Here we'll be demonstrating the commands and steps needed to do basic, simplified installations on a Linux based server, though it's also possible to run these services on Mac or Windows servers. Actually setting up a production server within your infrastructure will certainly entail differences in security measures or operating system tools, but hopefully this will give you the general idea of what's involved.

In order to initially set up any Git server, you have to export an existing repository into a new bare repository – a repository that doesn't contain a working directory. This is generally straightforward to do. In order to clone your repository to create a new bare repository, you run the clone

command with the `--bare` option. By convention, bare repository directories end in `.git`, like so:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

You should now have a copy of the Git directory data in your `my_project.git` directory.

This is roughly equivalent to something like

```
$ cp -Rf my_project/.git my_project.git
```

There are a couple of minor differences in the configuration file; but for your purpose, this is close to the same thing. It takes the Git repository by itself, without a working directory, and creates a directory specifically for it alone.

Putting the Bare Repository on a Server

Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols. Let's say you've set up a server called `git.example.com` that you have SSH access to, and you want to store all your Git repositories under the `/srv/git` directory. Assuming that `/srv/git` exists on that server, you can set up your new repository by copying your bare repository over:

```
$ scp -r my_project.git user@git.example.com:/srv/git
```

At this point, other users who have SSH access to the same server which has read-access to the `/srv/git` directory can clone your repository by running

```
$ git clone user@git.example.com:/srv/git/my_project.git
```

If a user SSHs into a server and has write access to the `/srv/git/my_project.git` directory, they will also automatically have push access.

Git will automatically add group write permissions to a repository properly if you run the `git init` command with the `--shared` option.

```
$ ssh user@git.example.com
$ cd /srv/git/my_project.git
$ git init --bare --shared
```

You see how easy it is to take a Git repository, create a bare version, and place it on a server to which you and your collaborators have SSH access. Now you're ready to collaborate on the same project.

It's important to note that this is literally all you need to do to run a useful Git server to which several people have access – just add SSH-able accounts on a server, and stick a bare repository somewhere that all those users have read and write access to. You're ready to go – nothing else needed.

In the next few sections, you'll see how to expand to more sophisticated setups. This discussion will include not having to create user accounts for each user, adding public read access to repositories, setting up web UIs and more. However, keep in mind that to collaborate with a couple of people on a private project, all you *need* is an SSH server and a bare repository.

Small Setups

If you're a small outfit or are just trying out Git in your organization and have only a few developers, things can be simple for you. One of the most complicated aspects of setting up a Git server is user management. If you want some repositories to be read-only to certain users and read/write to others, access and permissions can be a bit more difficult to arrange.

SSH Access

If you have a server to which all your developers already have SSH access, it's generally easiest to set up your first repository there, because you have to do almost no work (as we covered in the last section). If you want more complex access control type permissions on your repositories, you can handle them with the normal filesystem permissions of the operating system your server runs.

If you want to place your repositories on a server that doesn't have accounts for everyone on your team whom you want to have write access, then you must set up SSH access for them. We assume that if you have a server with which to do this, you already have an SSH server installed, and that's how you're accessing the server.

There are a few ways you can give access to everyone on your team. The first is to set up accounts for everybody, which is straightforward but can be cumbersome. You may not want to run `adduser` and set temporary passwords for every user.

A second method is to create a single *git* user on the machine, ask every user who is to have write access to send you an SSH public key, and add that key to the `~/.ssh/authorized_keys` file of your new *git* user. At that point, everyone will be able to access that machine via the *git* user. This doesn't affect the commit data in any way – the SSH user you connect as doesn't affect the commits you've recorded.

Another way to do it is to have your SSH server authenticate from an LDAP server or some other centralized authentication source that you may already have set up. As long as each user can get shell access on the machine, any SSH authentication mechanism you can think of should work.

Gerando Sua Chave Pública SSH

Dito isto, muitos servidores de Git efetuam autenticação utilizando chaves públicas SSH. Para prover uma chave pública, cada usuário do seu sistema deve gerar uma chave se ainda não possui uma. Este processo é similar em todos os sistemas operacionais. Primeiro, você deve verificar e se certificar de que ainda não tem uma chave. Por padrão, as chaves de SSH de um determinado

usuário são armazenadas no diretório `~/.ssh` daquele usuário. Você pode facilmente verificar se não já possui uma chave indo neste diretório e listando seu conteúdo:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

O que você precisa encontrar é um par de arquivos, um com o nome parecido com `id_dsa` ou `id_rsa`, e o outro de nome correspondente, porém com a extensão `.pub` no final. O arquivo de extensão `.pub` é sua chave pública, e o outro arquivo é sua chave privada. Caso você não tenha estes arquivos (ou sequer um diretório `.ssh`), você pode criá-los rodando um programa chamado `ssh-keygen`, que vem incluído no pacote SSH em sistemas Linux/Mac, e também no Git para Windows:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Primeiro, o programa confirma onde você deseja salvar a chave (`.ssh/id_rsa`), e então pede duas vezes por uma frase secreta, a qual você pode deixar vazia se não deseja digitar uma senha quando utiliza as chaves. Entretanto, se você utilizar uma senha, lembre-se de adicionar a opção `-o`; ela salva a chave privada em um formato que é mais resistente a ataques de força bruta do que o formato padrão. Você também pode utilizar a ferramenta `ssh-agent` para não ter de entrar com sua senha toda vez. Agora, cada usuário que tenha feito os passos acima deve enviar sua chave pública a você ou a quem estiver administrando o servidor de Git (assumindo que um servidor SSH que requeira chaves públicas esteja sendo utilizado). Tudo que o administrador precisa fazer é copiar o conteúdo do arquivo `.pub` e enviá-lo por e-mail. A chave tem mais ou menos esta aparência:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Para ver um tutorial mais detalhado sobre a criação de chaves SSH em diversos sistemas operacionais, veja o guia do GitHub sobre chaves SSH em https://docs.github.com/pt/authentication/connecting-to-github-with-ssh.

Setting Up the Server

Let's walk through setting up SSH access on the server side. In this example, you'll use the `authorized_keys` method for authenticating your users. We also assume you're running a standard Linux distribution like Ubuntu. First, you create a `git` user and a `.ssh` directory for that user.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Next, you need to add some developer SSH public keys to the `authorized_keys` file for the `git` user. Let's assume you have some trusted public keys and have saved them to temporary files. Again, the public keys look something like this:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

You just append them to the `git` user's `authorized_keys` file in its `.ssh` directory:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Now, you can set up an empty repository for them by running `git init` with the `--bare` option, which initializes the repository without a working directory:

```
$ cd /srv/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /srv/git/project.git/
```

Then, John, Josie, or Jessica can push the first version of their project into that repository by adding it as a remote and pushing up a branch. Note that someone must shell onto the machine and create a bare repository every time you want to add a project. Let's use `gitserver` as the hostname of the server on which you've set up your `git` user and repository. If you're running it internally, and you set up DNS for `gitserver` to point to that server, then you can use the commands pretty much as is (assuming that `myproject` is an existing project with files in it):

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/srv/git/project.git
$ git push origin master
```

At this point, the others can clone it down and push changes back up just as easily:

```
$ git clone git@gitserver:/srv/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

With this method, you can quickly get a read/write Git server up and running for a handful of developers.

You should note that currently all these users can also log into the server and get a shell as the `git` user. If you want to restrict that, you will have to change the shell to something else in the `passwd` file.

You can easily restrict the `git` user to only doing Git activities with a limited shell tool called `git-shell` that comes with Git. If you set this as your `git` user's login shell, then the `git` user can't have normal shell access to your server. To use this, specify `git-shell` instead of bash or csh for your user's login shell. To do so, you must first add `git-shell` to `/etc/shells` if it's not already there:

```
$ cat /etc/shells   # see if `git-shell` is already in there.  If not...
$ which git-shell   # make sure git-shell is installed on your system.
$ sudo vim /etc/shells  # and add the path to git-shell from last command
```

Now you can edit the shell for a user using `chsh <username>`:

```
$ sudo chsh git  # and enter the path to git-shell, usually: /usr/bin/git-shell
```

Now, the `git` user can only use the SSH connection to push and pull Git repositories and can't shell onto the machine. If you try, you'll see a login rejection like this:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

Now Git network commands will still work just fine but the users won't be able to get a shell. As the

output states, you can also set up a directory in the `git` user's home directory that customizes the `git-shell` command a bit. For instance, you can restrict the Git commands that the server will accept or you can customize the message that users see if they try to SSH in like that. Run `git help shell` for more information on customizing the shell.

Git Daemon

Next we'll set up a daemon serving repositories over the "Git" protocol. This is common choice for fast, unauthenticated access to your Git data. Remember that since it's not an authenticated service, anything you serve over this protocol is public within its network.

If you're running this on a server outside your firewall, it should only be used for projects that are publicly visible to the world. If the server you're running it on is inside your firewall, you might use it for projects that a large number of people or computers (continuous integration or build servers) have read-only access to, when you don't want to have to add an SSH key for each.

In any case, the Git protocol is relatively easy to set up. Basically, you need to run this command in a daemonized manner:

```
$ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/
```

`--reuseaddr` allows the server to restart without waiting for old connections to time out, the `--base-path` option allows people to clone projects without specifying the entire path, and the path at the end tells the Git daemon where to look for repositories to export. If you're running a firewall, you'll also need to punch a hole in it at port 9418 on the box you're setting this up on.

You can daemonize this process a number of ways, depending on the operating system you're running. On an Ubuntu machine, you can use an Upstart script. So, in the following file

```
/etc/init/local-git-daemon.conf
```

you put this script:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/srv/git/ \
    /srv/git/
respawn
```

For security reasons, it is strongly encouraged to have this daemon run as a user with read-only permissions to the repositories – you can easily do this by creating a new user *git-ro* and running the daemon as them. For the sake of simplicity we'll simply run it as the same *git* user that `git-shell` is running as.

When you restart your machine, your Git daemon will start automatically and respawn if it goes down. To get it running without having to reboot, you can run this:

```
$ initctl start local-git-daemon
```

On other systems, you may want to use `xinetd`, a script in your `sysvinit` system, or something else – as long as you get that command daemonized and watched somehow.

Next, you have to tell Git which repositories to allow unauthenticated Git server-based access to. You can do this in each repository by creating a file named `git-daemon-export-ok`.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

The presence of that file tells Git that it's OK to serve this project without authentication.

Smart HTTP

We now have authenticated access through SSH and unauthenticated access through `git://`, but there is also a protocol that can do both at the same time. Setting up Smart HTTP is basically just enabling a CGI script that is provided with Git called `git-http-backend` on the server. This CGI will read the path and headers sent by a `git fetch` or `git push` to an HTTP URL and determine if the client can communicate over HTTP (which is true for any client since version 1.6.6). If the CGI sees that the client is smart, it will communicate smartly with it, otherwise it will fall back to the dumb behavior (so it is backward compatible for reads with older clients).

Let's walk through a very basic setup. We'll set this up with Apache as the CGI server. If you don't have Apache setup, you can do so on a Linux box with something like this:

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env rewrite
```

This also enables the `mod_cgi`, `mod_alias`, `mod_env`, and `mod_rewrite` modules, which are all needed for this to work properly.

You'll also need to set the Unix user group of the `/srv/git` directories to `www-data` so your web server can read- and write-access the repositories, because the Apache instance running the CGI script will (by default) be running as that user:

```
$ chgrp -R www-data /srv/git
```

Next we need to add some things to the Apache configuration to run the `git-http-backend` as the handler for anything coming into the `/git` path of your web server.

```
SetEnv GIT_PROJECT_ROOT /srv/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

If you leave out `GIT_HTTP_EXPORT_ALL` environment variable, then Git will only serve to unauthenticated clients the repositories with the `git-daemon-export-ok` file in them, just like the Git daemon did.

Finally you'll want to tell Apache to allow requests to `git-http-backend` and make writes be authenticated somehow, possibly with an Auth block like this:

```
RewriteEngine On
RewriteCond %{QUERY_STRING} service=git-receive-pack [OR]
RewriteCond %{REQUEST_URI} /git-receive-pack$
RewriteRule ^/git/ - [E=AUTHREQUIRED]

<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /srv/git/.htpasswd
    Require valid-user
    Order deny,allow
    Deny from env=AUTHREQUIRED
    Satisfy any
</Files>
```

That will require you to create a `.htpasswd` file containing the passwords of all the valid users. Here is an example of adding a "schacon" user to the file:

```
$ htpasswd -c /srv/git/.htpasswd schacon
```

There are tons of ways to have Apache authenticate users, you'll have to choose and implement one of them. This is just the simplest example we could come up with. You'll also almost certainly want to set this up over SSL so all this data is encrypted.

We don't want to go too far down the rabbit hole of Apache configuration specifics, since you could well be using a different server or have different authentication needs. The idea is that Git comes with a CGI called `git-http-backend` that when invoked will do all the negotiation to send and receive data over HTTP. It does not implement any authentication itself, but that can easily be controlled at the layer of the web server that invokes it. You can do this with nearly any CGI-capable web server, so go with the one that you know best.

NOTA For more information on configuring authentication in Apache, check out the Apache docs here: http://httpd.apache.org/docs/current/howto/auth.html

GitWeb

Now that you have basic read/write and read-only access to your project, you may want to set up a simple web-based visualizer. Git comes with a CGI script called GitWeb that is sometimes used for this.

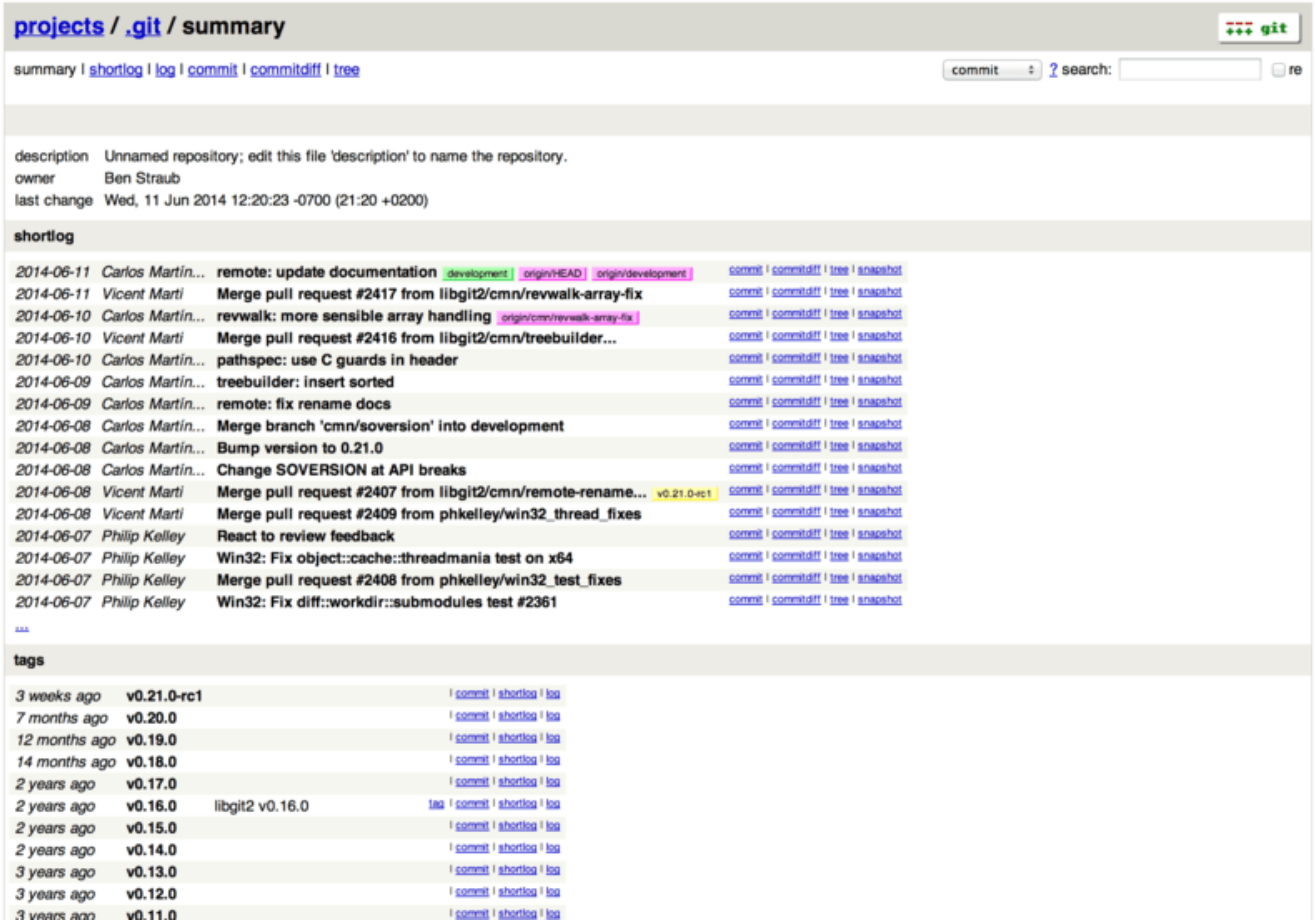


Figura 49. The GitWeb web-based user interface.

If you want to check out what GitWeb would look like for your project, Git comes with a command to fire up a temporary instance if you have a lightweight server on your system like `lighttpd` or `webrick`. On Linux machines, `lighttpd` is often installed, so you may be able to get it to run by typing `git instaweb` in your project directory. If you're running a Mac, Leopard comes preinstalled with Ruby, so `webrick` may be your best bet. To start `instaweb` with a non-lighttpd handler, you can run it with the `--httpd` option.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

That starts up an HTTPD server on port 1234 and then automatically starts a web browser that opens on that page. It's pretty easy on your part. When you're done and want to shut down the server, you can run the same command with the `--stop` option:

```
$ git instaweb --httpd=webrick --stop
```

If you want to run the web interface on a server all the time for your team or for an open source project you're hosting, you'll need to set up the CGI script to be served by your normal web server. Some Linux distributions have a `gitweb` package that you may be able to install via `apt` or `yum`, so you may want to try that first. We'll walk through installing GitWeb manually very quickly. First, you need to get the Git source code, which GitWeb comes with, and generate the custom CGI script:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/srv/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Notice that you have to tell the command where to find your Git repositories with the `GITWEB_PROJECTROOT` variable. Now, you need to make Apache use CGI for that script, for which you can add a VirtualHost:

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Again, GitWeb can be served with any CGI or Perl capable web server; if you prefer to use something else, it shouldn't be difficult to set up. At this point, you should be able to visit `http://gitserver/` to view your repositories online.

GitLab

Embora o GitWeb seja bastante simplista, se você estiver procurando por um servidor Git mais moderno e completo, existem algumas soluções de código aberto lá fora que você pode instalar em vez deste. Como o GitLab é um dos mais populares, vamos cobrir a instalação e usá-lo como um exemplo. Este é um pouco mais complexo do que a opção GitWeb e provavelmente requer mais manutenção, mas é uma opção muito mais completa.

Instalação

O GitLab é um aplicativo da Web baseado em banco de dados, por isso sua instalação é um pouco mais trabalhosa do que alguns outros servidores Git. Felizmente, este processo é muito bem documentado e apoiado.

Existem alguns métodos que você pode seguir para instalar o GitLab. Para obter algo em execução rapidamente, você pode baixar uma imagem de máquina virtual ou um instalador de um clique em https://bitnami.com/stack/gitlab e ajustar a configuração para que corresponda ao seu ambiente particular.. Um toque agradável que Bitnami incluiu é a tela de login (acessada digitando alt-→); Ele informa o endereço IP e o nome de usuário e senha padrão para o GitLab instalado.

Figura 50. A tela de login da máquina virtual Bitnami GitLab.

Para qualquer outra coisa, siga as orientações no readme do GitLab Community Edition, que pode ser encontrado em https://gitlab.com/gitlab-org/gitlab-ce/tree/master. Lá você encontrará assistência na instalação do GitLab usando receitas do Chef, a máquina virtual no Digital Ocean, e os pacotes RPM e DEB (que, no momento em que este texto foi escrito, estava na versão estável Omnibus, para os sistemas operacionais Ubuntu 14.04, Ubuntu 16.04, Debian 7, Debian 8, CentOS 6, CentOS 7, OpenSUSE 42.1 e Raspberry PI 2 em Raspbian). Há também guias com orientações “não oficiais” para que o GitLab funcione em sistemas operacionais e bancos de dados diferentes dos citados acima, um script para instalação completamente manual e muitos outros tópicos.

Administração

A interface de administração do GitLab é acessada através da web. Basta apontar o seu navegador para o nome do host ou endereço IP onde o GitLab está instalado e efetuar login como um usuário admin. O nome de usuário padrão é `admin@local.host` e a senha padrão é `5iveL!fe` (que você será solicitado a alterar assim que você entrar nele). Depois de efetuar login, clique no ícone “Área de administração” no menu no canto superior direito.

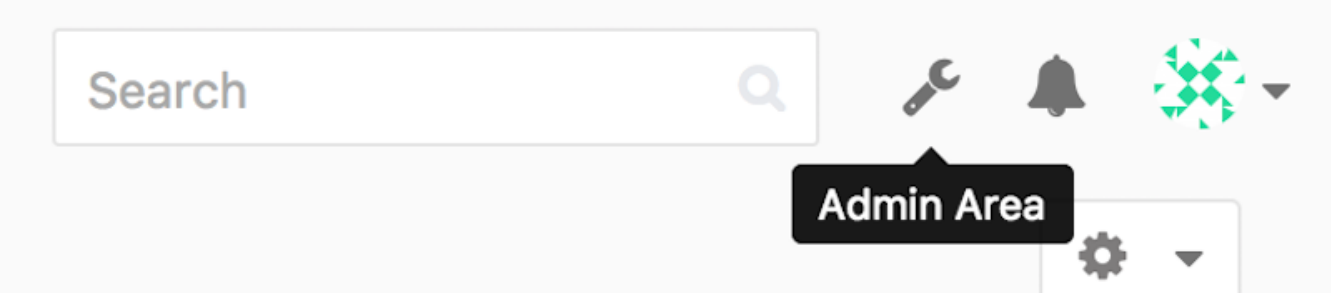


Figura 51. O item "Área de administração" no menu GitLab.

Usuários

Usuários no GitLab são contas que correspondem a pessoas. As contas de usuário não têm muita complexidade; A conta de usuário é uma coleção de informações pessoais anexadas aos dados de login. Cada conta de usuário vem com um **namespace**, que é um agrupamento lógico de projetos que pertencem a esse usuário. Se o usuário `jane` tivesse um projeto chamado `project`, o URL do projeto seria `http://servidor/jane/project`.

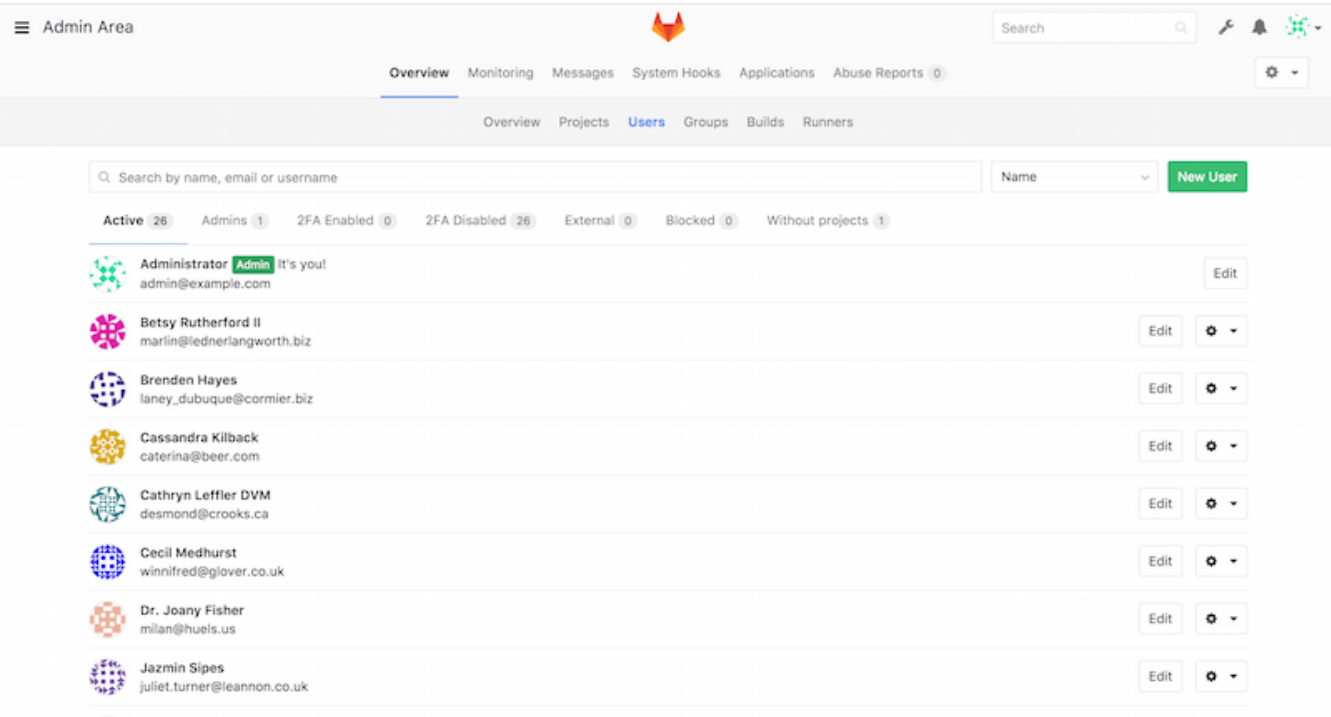


Figura 52. Tela de administração de usuários do GitLab.

Remoção de um usuário pode ser feito de duas formas. "Bloquear" um usuário os impede de logar na instância do Gitlab, mas todos os dados sob o namespace do usuário serão preservados, e os commits assinados com o e-mail do usuário ainda irão direcionar para o perfil do mesmo.

"Destruir" um usuário, por outro lado, remove o remove completamente do banco de dados e do sistema de arquivos. Todos os projetos e dados no namespace são removidos, e qualquer outro grupo que ele possua também será removido. Isso é obviamente uma ação muito mais permanete e destrutiva, e o uso disso é raro.

Grupos

Um grupo GitLab é um conjunto de projetos, juntamente com dados sobre como os usuários podem acessar esses projetos. Cada grupo tem um espaço para nome de projeto (da mesma forma que os

usuários), então se o grupo `training` tiver um projeto `materials`, sua url seria `http://servidor/training/materials`.

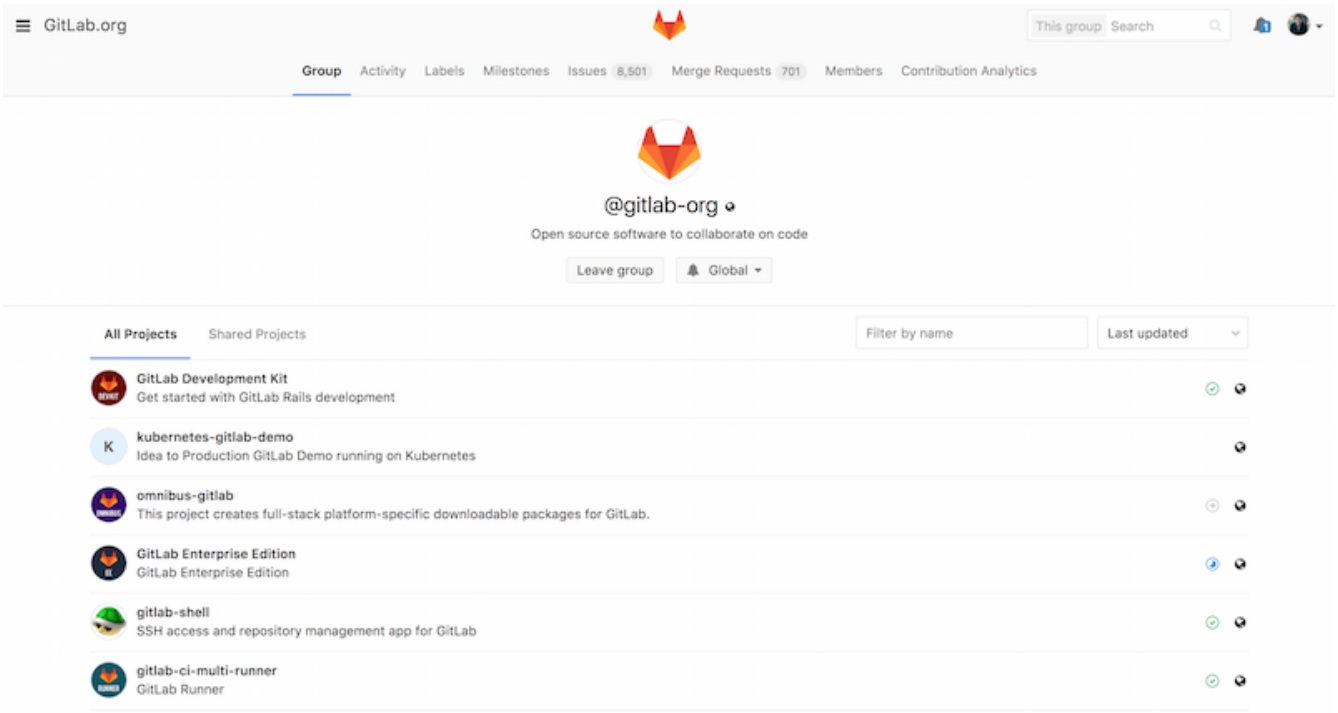


Figura 53. Tela de administração de grupos do GitLab.

Cada grupo está associado a um número de usuários, cada um com um nível de permissões para os projetos do grupo e para o próprio grupo. Estes variam de "Convidado" (problemas e bate-papo somente) a "Proprietário" (controle total do grupo, seus membros e seus projetos). Os tipos de permissões são muito numerosos para listar aqui, mas o GitLab tem um link útil na tela de administração.

Projetos

Um projeto GitLab corresponde grosso modo a um único repositório Git. Cada projeto pertence a um único namespace, a um usuário ou a um grupo. Se o projeto pertence a um usuário, o proprietário do projeto tem controle direto sobre quem tem acesso ao projeto; Se o projeto pertence a um grupo, as permissões de nível de usuário do grupo também terão efeito.

Cada projeto também tem um nível de visibilidade, que controla quem tem acesso de leitura às páginas desse projeto e ao repositório. Se um projeto for *Privado*, o proprietário do projeto deve conceder explicitamente acesso a usuários específicos. Um projeto *Interno* é visível para qualquer usuário logado, e um projeto *Público* é visível para qualquer pessoa. Observe que isso controla tanto o acesso `git fetch` quanto o acesso à interface web do usuário a esse projeto.

Ganchos (hooks)

O GitLab inclui suporte para ganchos (hooks), tanto a nível de projeto como de sistema. Para qualquer um destes, o servidor GitLab executará um HTTP POST com algum JSON descritivo sempre que ocorrerem eventos relevantes. Esta é uma ótima maneira de conectar seus repositórios Git e a instância GitLab ao resto de sua automação de desenvolvimento, como servidores CI, salas de bate-papo ou ferramentas de implantação.

Uso Básico

A primeira coisa que você vai querer fazer com o GitLab é criar um novo projeto. Isso é feito clicando no ícone "+" na barra de ferramentas. Ser-lhe-á pedido o nome do projecto, a qual namespace ele deverá pertencer e que nível de visibilidade deverá ter. A maior parte do que você especifica aqui não é permanente e pode ser reajustada posteriormente através da interface de configurações. Clique em "Criar projeto" e pronto.

Uma vez que o projeto exista, você provavelmente vai querer conectá-lo com um repositório Git local. Cada projeto é acessível através de HTTPS ou SSH, sendo que ambos podem ser usados para configurar um Git remoto. As URLs estão visíveis na parte superior da página inicial do projeto. Para um repositório local existente, este comando criará um remoto chamado `gitlab` para o local hospedado:

```
$ git remote add gitlab https://servidor/namespace/project.git
```

Se você não tem uma cópia local do repositório, você pode simplesmente fazer isso:

```
$ git clone https://servidor/namespace/project.git
```

A interface do usuário da Web fornece acesso a várias visualizações úteis do próprio repositório. A página inicial de cada projeto mostra as atividades recentes e os links ao longo do topo levam você a exibições dos arquivos do projeto e do log de commits.

Trabalhando juntos

A maneira mais simples de trabalhar juntos em um projeto GitLab é dar a outro usuário acesso direto de push (envio de commits) ao repositório Git. Você pode adicionar um usuário a um projeto indo para a seção "Membros" das configurações desse projeto e associando o novo usuário com um nível de acesso (os diferentes níveis de acesso estão um pouco descritos em Grupos). Ao fornecer a um usuário um nível de acesso de "Desenvolvedor" ou superior, esse usuário pode empurrar branches e ramificações diretamente para o repositório.

Outra maneira mais dissociada de colaboração é usar solicitações de mesclagem. Esse recurso permite que qualquer usuário que possa ver um projeto contribua para ele de forma controlada. Os usuários com acesso direto podem simplesmente criar uma ramificação, empurrá-la para ele e abrir uma solicitação de mesclagem de seu ramo de volta para `master` ou qualquer outra ramificação. Os usuários que não têm permissões push para um repositório podem "fork" (criar sua própria cópia), push commit para *aquela* cópia e abrir uma solicitação de mesclagem de sua bifurcação de volta para o projeto principal. Este modelo permite que o proprietário esteja no controle total do que entra no repositório e quando, embora permita contribuições de usuários não confiáveis.

Os pedidos de mesclagem e os problemas são as principais unidades de discussão de longa duração no GitLab. Cada solicitação de mesclagem permite uma discussão linha a linha da alteração proposta (que suporta um tipo leve de revisão de código), bem como um tópico geral de discussão geral. Ambos podem ser atribuídos a usuários ou organizados em marcos.

Esta seção é focada principalmente nos recursos relacionados ao Git do GitLab, mas como um projeto maduro, ele fornece muitos outros recursos para ajudar sua equipe a trabalhar em conjunto, como wikis de projeto e ferramentas de manutenção do sistema. Um benefício para o GitLab é que, uma vez que o servidor está configurado e em execução, você raramente precisará ajustar um arquivo de configuração ou acessar o servidor via SSH; A maior parte da administração e o uso geral podem ser realizados através da interface no navegador.

Third Party Hosted Options

If you don't want to go through all of the work involved in setting up your own Git server, you have several options for hosting your Git projects on an external dedicated hosting site. Doing so offers a number of advantages: a hosting site is generally quick to set up and easy to start projects on, and no server maintenance or monitoring is involved. Even if you set up and run your own server internally, you may still want to use a public hosting site for your open source code – it's generally easier for the open source community to find and help you with.

These days, you have a huge number of hosting options to choose from, each with different advantages and disadvantages. To see an up-to-date list, check out the GitHosting page on the main Git wiki at https://git.wiki.kernel.org/index.php/GitHosting

We'll cover using GitHub in detail in GitHub, as it is the largest Git host out there and you may need to interact with projects hosted on it in any case, but there are dozens more to choose from should you not want to set up your own Git server.

Sumário

Você tem várias opções para ter um repositório remoto Git executando para que você possa colaborar com outras pessoas e compartilhar seu trabalho.

Executar seu próprio servidor te proporcional muito controle e permite que você execute seu servidor por seu próprio firewall, mas é esse tipo de servidor geralmente requer bastante de seu tempo para configuração e manutenção. Se você colocou seus dados em um servidor hospedado é fácil configurar e manter, contudo, você tem que manter seu código no servidor de terceiros, mas algumas organizações não permitem isso.

Isso deve ser o bastante para determinar qual solução ou combinação de soluções é apropriada para você e seus colaboradores.

Distributed Git

Now that you have a remote Git repository set up as a point for all the developers to share their code, and you're familiar with basic Git commands in a local workflow, you'll look at how to utilize some of the distributed workflows that Git affords you.

In this chapter, you'll see how to work with Git in a distributed environment as a contributor and an integrator. That is, you'll learn how to contribute code successfully to a project and make it as easy on you and the project maintainer as possible, and also how to maintain a project successfully with a number of developers contributing.

Fluxos de Trabalho Distribuídos

Em contraste com Sistemas de Controle de Versão Centralizados (CVCSs), a natureza compartilhada do Git permite ser muito mais flexível na maneira que desenvolvedores colaboram em projetos. Em sistemas centralizados, cada desenvolvedor é um nó trabalhando mais ou menos pareado com o ponto central (*hub*). No Git, entretanto, cada desenvolvedor pode ser tanto um nó quanto um hub; ou seja, cada desenvolvedor pode tanto contribuir para o código de outros repositórios quanto manter um repositório público no qual outros podem basear o trabalho deles e contribuir. Isto permite várias possibilidades no fluxo de trabalho de seu projeto e/ou da sua equipe, então iremos explorar alguns paradigmas comuns que aproveitam esta flexibilidade. Cobriremos os pontos fortes e as possíveis fraquezas de cada design; você poderá escolher apenas um para usar, ou uma combinação de suas características.

Fluxo de Trabalho Centralizado

Em sistemas centralizados, geralmente há um único modelo de colaboração — o fluxo de trabalho centralizado. Um hub central, ou *repositório*, que pode aceitar código e todos sincronizam seu trabalho com ele. Alguns desenvolvedores são nós — consumidores daquele hub — e sincronizam com aquela localização central.

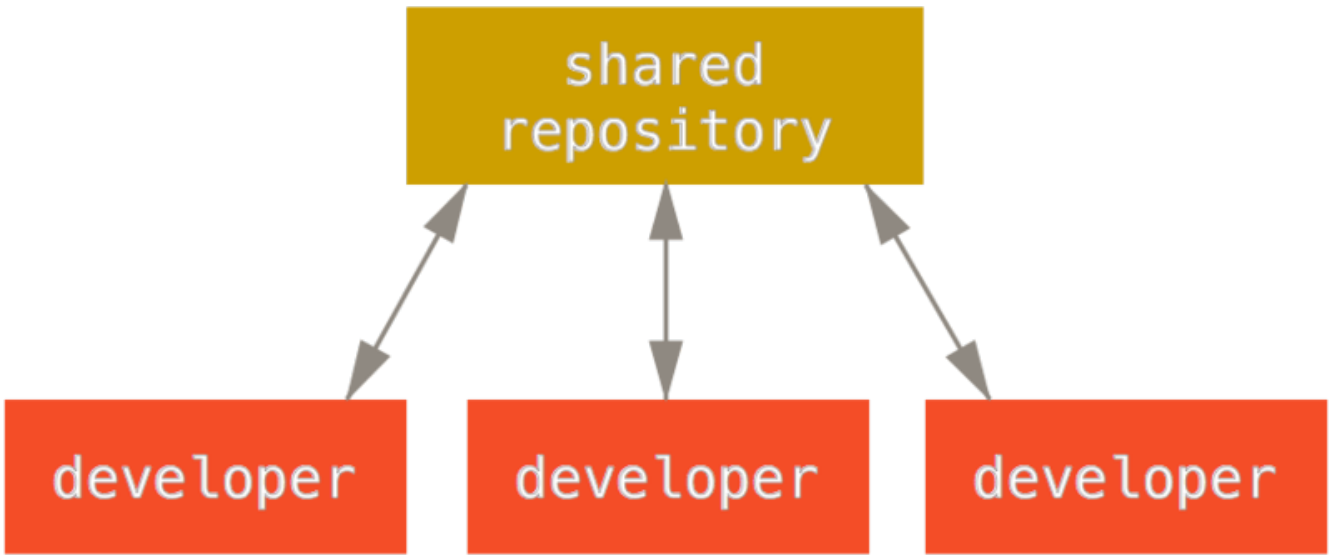


Figura 54. Fluxo de trabalho centralizado

Isto significa que se dois desenvolvedores clonarem do hub e ambos fizerem alterações, o primeiro

desenvolvedor que publicar (*push*) no servidor pode fazê-lo sem problemas. O segundo desenvolvedor deve mesclar (*merge*) com o trabalho do primeiro antes de publicar suas mudanças, para não sobrescrever as modificações do primeiro desenvolvedor. Este conceito é tão verdadeiro em Git quanto em Subversion (ou qualquer CVCS), e este modelo funciona perfeitamente bem em Git.

Se você já é confortável com um fluxo de trabalho centralizado na sua companhia ou equipe, pode facilmente continuar usando este fluxo de trabalho com o Git. Simplesmente configure um único repositório, e dê para todos no seu time permissão de publicação (*push*); Git não permitirá os usuários sobrescreverem-se.

Digamos que John e Jessica começaram a trabalhar ao mesmo tempo. John termina sua modificação e dá um push para o servidor. Então Jessica tenta dar um push das alterações dela, mas o servidor as rejeita. Ela recebe uma mensagem de que está tentando dar um push com modificações conflitantes (non-fast-forward) e que não conseguirá até as resolver e mesclar. Este fluxo de trabalho atrai várias pessoas pois já é um modelo familiar e confortável para muitos.

Isto não é limitado apenas a equipes pequenas. Com o modelo de ramificações do Git, é possível para centenas de desenvolvedores conseguirem trabalhar em um único projeto através de dúzias de ramos (*branches*) simultaneamente.

Fluxo de Trabalho Coordenado

Como o Git permite ter múltiplos repositórios remotos, é possível ter um fluxo de trabalho onde cada desenvolvedor tem permissão de escrita para o seu próprio repositório, e permissão de leitura para o de todos os outros. Este cenário geralmente inclui um repositório canônico que representa o projeto “oficial”. Para contribuir com este projeto, você cria seu próprio clone público do projeto e dá um push das suas modificações. Então você pode mandar um pedido para os coordenadores do projeto principal para aceitarem (*pull*) suas mudanças. Os coordenadores podem então adicionar seu repositório como um repositório remoto deles, testar suas mudanças localmente, mesclá-las (*merge*) nos respectivos branches e publicar (*push*) no repositório principal. O processo funciona assim (ver Fluxo de trabalho coordenado):

1. Os coordenadores do projeto publicam no repositório público.
2. Um colaborador clona o repositório e faz modificações.
3. O colaborador dá um push para a sua própria cópia pública.
4. Este contribuinte manda aos coordenadores um email pedindo para incluir as modificações.
5. Os coordenadores adicionam o repositório do colaborador como um repositório remoto e o mesclam localmente.
6. Os coordenadores publicam as alterações combinadas no repositório principal.

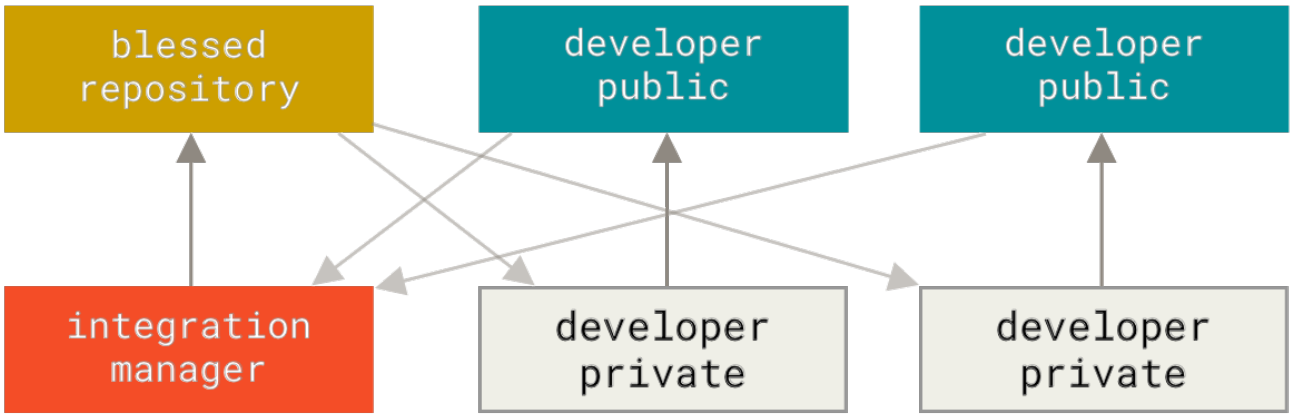


Figura 55. Fluxo de trabalho coordenado

Este é um fluxo de trabalho bastante comum em ferramentas baseadas em um hub como GitHub ou GitLab, onde é fácil bifurcar (*fork*) um projeto e publicar suas modificações no seu próprio fork para todos verem. Uma das principais vantagens desta abordagem é que você pode continuar a trabalhar, e os coordenadores do repositório principal podem incluir as suas modificações a qualquer hora. Colaboradores não tem que esperar pelo projeto para incorporar suas mudanças — cada grupo pode trabalhar na sua própria velocidade.

Fluxo de Trabalho Ditador e Tenentes

Esta é uma variante de um fluxo de trabalho com múltiplos repositórios. É geralmente usada por projetos gigantescos com centenas de colaboradores; um exemplo famoso é o kernel Linux. Vários coordenadores são responsáveis por partes específicas do repositório, eles são chamados *tenentes*. Todos os tenentes têm um coordenador conhecido como o ditador benevolente. O ditador benevolente publica (*push*) do diretório deles para um repositório de referência do qual todos os colaboradores precisam buscar (*pull*). Este processo funciona assim (ver Fluxo de trabalho do ditador benevolente):

1. Desenvolvedores comuns trabalham no seu próprio branch, baseando seu trabalho no `master`. Este branch `master` é aquele do repositório de referência no qual o ditador publica (*push*).
2. Tenentes mesclam (*merge*) cada branch dos desenvolvedores ao branch `master` deles.
3. O ditador mescla os branches `master` dos tenentes no branch `master` do ditador.
4. Finalmente, o ditador publica aquele branch `master` para o repositório de referência então os desenvolvedores podem se basear nele.

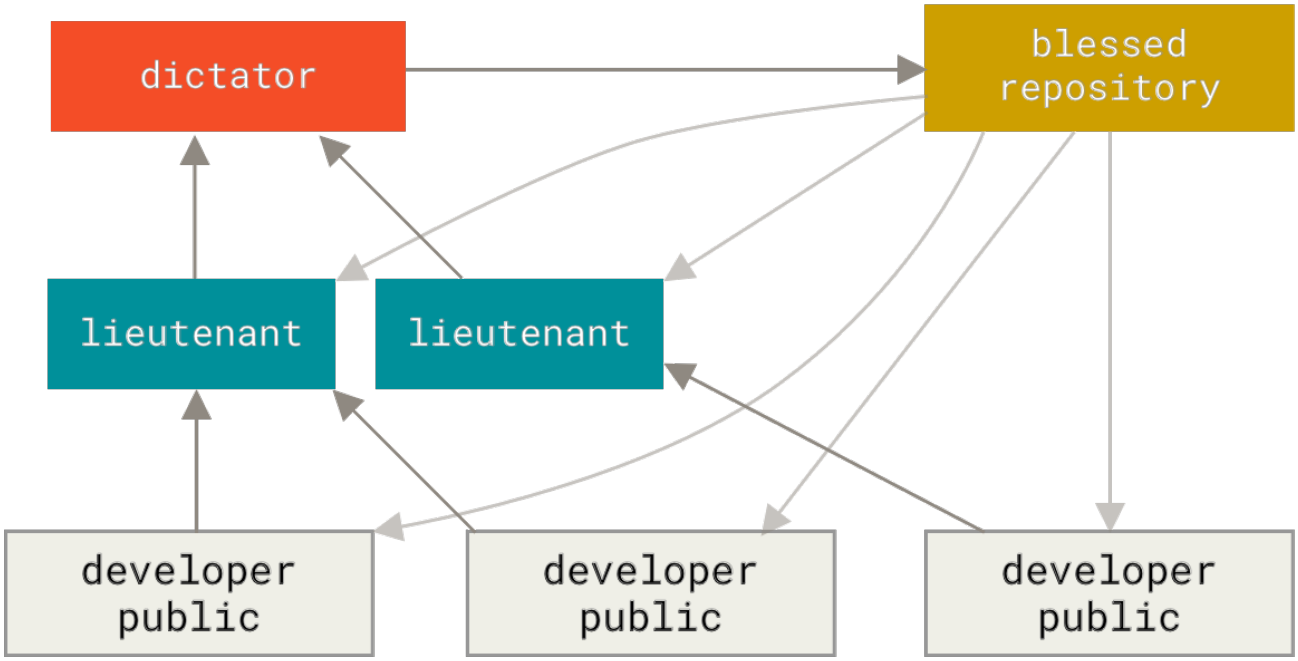


Figura 56. Fluxo de trabalho do ditador benevolente

Este tipo de fluxo de trabalho não é comum, mas pode ser útil em projetos muito grandes, ou em ambientes altamente hierárquicos. Ele permite ao líder de projeto (o ditador) delegar muitas tarefas e coletar vários pedaços de código de múltiplas fontes antes de combiná-los.

Padrões para Controlar Branches de Código Fonte

NOTA

Martin Fowler fez um manual "Patterns for Managing Source Code Branches". Este guia cobre todos os fluxos de trabalho comuns, e explica como/quando utilizá-los. Há também uma seção comparando fluxos com muitas ou poucas combinações.

https://martinfowler.com/articles/branching-patterns.html

Resumo do Fluxo de Trabalho

Estes são alguns fluxos de trabalho comumente utilizados graças a sistemas distribuídos como o Git, mas muitas variações podem ser adaptadas ao seu fluxo de trabalho no mundo real. Agora que você é capaz (tomara) de determinar qual combinação de fluxo de trabalho deve funcionar para você, iremos cobrir alguns exemplos mais específicos de como realizar as principais funções que compõem os diferentes fluxos. Na próxima seção, você irá aprender sobre alguns padrões comuns para contribuir com um projeto.

Contribuindo com um Projeto

A principal dificuldade em descrever como contribuir com um projeto é a numerosa quantidade de maneiras de contribuir. Já que Git é muito flexível, as pessoas podem e trabalham juntas de muitas maneiras, sendo problemático descrever como você deve contribuir — cada projeto é um pouco diferente. Algumas das variáveis envolvidas são a quantidade de colaboradores ativos, o fluxo de trabalho escolhido, sua permissão para fazer commit, e possivelmente o método de contribuição externa.

A primeira variável é a quantidade de colaboradores ativos — quantos usuários estão ativamente contribuindo para o código deste projeto, e em que frequência? Em muitas circunstâncias você terá dois ou três desenvolvedores com alguns commites por dia, ou possivelmente menos em projetos meio dormentes. Para grandes companhias ou projetos, o número de desenvolvedores pode estar nas centenas, com centenas ou milhares de commites chegando todos os dias. Isto é importante porque com mais e mais desenvolvedores, você se depara com mais problemas para certificar-se que seu código é aplicado diretamente ou pode ser facilmente integrado. Alterações que você entrega podem se tornar obsoletas ou severamente quebrar pelo trabalho que é mesclado enquanto você trabalha ou enquanto suas mudanças estão esperando para serem aprovadas e aplicadas. Como você pode manter seu código consistentemente atualizado e seus commites válidos?

A próxima variável é o fluxo de trabalho em uso no projeto. Ele é centralizado, com cada desenvolvedor tendo permissões de escrita iguais para o código principal? O projeto tem um mantenedor ou coordenador que checa todos os patches? Todos os patches são revisados e aprovados pelos colegas? Você está envolvido neste processo? Um sistema de tenentes está estabelecido, e você tem que enviar seu trabalho para eles antes?

A próxima variável é sua permissão de commit. O fluxo de trabalho necessário para contribuir ao projeto é muito diferente se você tem permissão de escrita ao projeto ou não tem. Se você não tem permissão de escrita, como o projeto costuma aceitar o trabalho dos colaboradores? Existe algum tipo de norma? Quanto trabalho você está enviando por vez? Com que frequência?

Todas estas questões podem afetar como você contribui efetivamente para o projeto e que fluxos de trabalho são adequados ou possíveis para você. Iremos abordar aspectos de cada uma delas em uma série de estudos de caso, indo do simples ao mais complexo; você deve ser capaz de construir fluxos de trabalho específicos para suas necessidades com estes exemplos.

Diretrizes para Fazer Commites

Antes de vermos estudos de casos específicos, uma observação rápida sobre mensagens de commit. Ter uma boa diretriz para criar commites e a seguir facilita muito trabalhar com Git e colaborar com outros. O projeto Git fornece um documento que dá várias dicas boas para criar commites ao enviar patches — você pode lê-lo no código fonte do Git no arquivo `Documentation/SubmittingPatches`.

Primeiro, seus envios não devem conter nenhum espaço em branco não proposital. Git fornece uma maneira fácil de checar isto — antes de você fazer um commit, execute `git diff --check`, que identifica possíveis espaços em branco indesejados e os lista pra você.

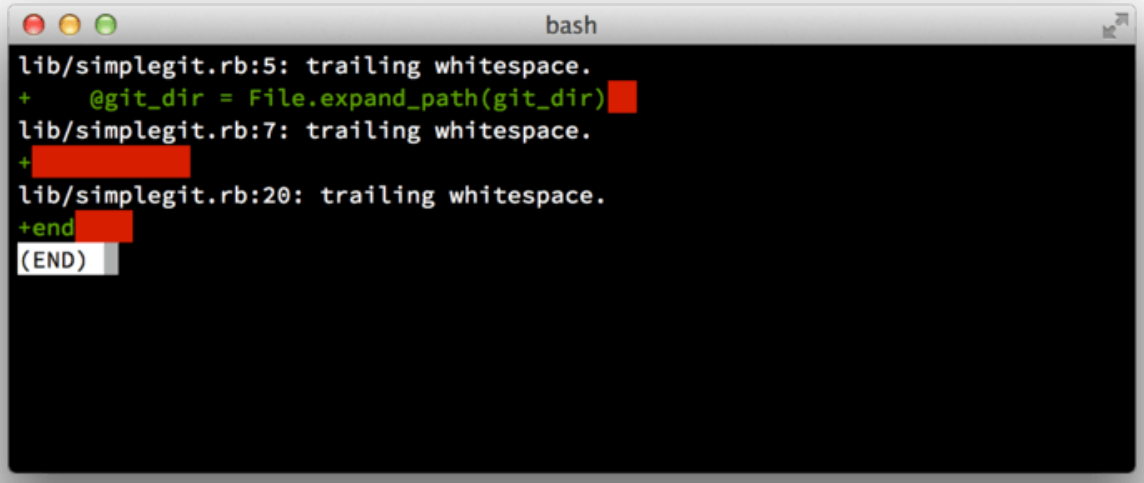


Figura 57. Resultado de `git diff --check`

Se você executar este comando antes do commit, você perceberá se está prestes a enviar espaços em branco problemáticos que podem irritar os outros desenvolvedores.

Depois tente fazer cada commit como um conjunto lógico de mudanças. Se possível, tente digerir suas modificações — não programe o final de semana inteiro em cinco diferentes problemas e então publique tudo em um commit massivo na segunda-feira. Mesmo que você não publique durante o fim de semana, use a área de stage na segunda-feira para dividir seu trabalho em ao menos um commit por assunto, com uma mensagem útil em cada commit. Se algumas alterações modificarem o mesmo arquivo, tente executar `git add --patch` para colocar na área de stage os arquivos parcialmente (explicado em detalhes em Interactive Staging). O retrato do projeto no final do branch é idêntico você fazendo um commit ou cinco, desde que todas as mudanças forem eventualmente adicionadas, então tente fazer as coisas mais fáceis para seus colegas desenvolvedores quando eles tiverem que revisar suas mudanças.

Esta abordagem também facilita retirar ou reverter uma das alterações se você precisar depois. Rewriting History descreve vários truques úteis do Git para reescrever o histórico e colocar interativamente arquivos na área de stage — utilize estas ferramentas para criar um histórico limpo e compreensível antes de enviar o trabalho para alguém.

A última coisa para ter em mente é a mensagem do commit. Manter o hábito de criar boas mensagens de commit facilita muito usar e colaborar com o Git. Como regra geral, suas mensagens devem começar com uma única linha que não tem mais que 50 caracteres e descreve as alterações concisamente, seguida de uma linha em branco, seguida de uma explicação mais detalhada. O projeto Git requer que esta explicação mais detalhada inclua sua motivação para a mudança e compare sua implementação com o comportamento anterior — esta é uma boa diretriz para seguir. Escreva sua mensagem de commit no imperativo: "Consertar o bug" e não "Bug consertado" ou "Conserta bug." Tem um modelo que você pode seguir, que adaptamos ligeiramente daqui escrito originalmente por Tim Pope:

```
Resumo curto (50 caracteres ou menos), com maiúsculas.

Mais texto explicativo, se necessário. Desenvolva por 72
caracteres aproximadamente. Em alguns contextos, a primeira linha é tratada como o
assunto do email e o resto do texto como o corpo. A linha
em branco separando o assunto do corpo é crítica (a não ser que você omita
o corpo inteiro); ferramentas como _rebase_ irão te confundir se você unir as
duas partes.

Escreva sua mensagem de commit no imperativo: "Consertar o bug" e não "Bug consertado"
ou "Conserta bug". Esta convenção combina com as mensagens de commit geradas
por comandos como `git merge` e `git revert`.

Parágrafos seguintes veem depois de linhas em branco.

- Marcadores são ok, também

- Normalmente um hífen ou asterisco é usado para o marcador, seguido de um
 único espaço, com linhas em branco entre eles, mas os protocolos podem variar aqui

- Utilize recuos alinhados
```

Se todas as suas mensagens de commit seguirem este modelo, as coisas serão muito mais fáceis para você e os desenvolvedores com que trabalha. O projeto Git tem mensagens de commit bem formatadas - tente executar `git log --no-merges` nele para ver o que um projeto com histórico de commit bem feito se parece.

NOTA

Faça o que digo, não faça o que faço.

Para o bem da concisão, muitos dos exemplos neste livro não tem mensagens de commit bem formatas como esta; ao invés, nós simplesmente usamos a opção `-m` do `git commit`.

Resumindo, faça o que digo, não faça o que faço.

Time Pequeno Privado

A configuração mais simples que você deve encontrar é um projeto privado com um ou dois outros desenvolvedores. ``Privado,`` neste contexto, significa código fechado — não acessível ao mundo exterior. Você e os outros desenvolvedores têm permissão para publicar no repositório.

Neste ambiente, você pode seguir um fluxo de trabalho similar ao que faria usando Subversion ou outro sistema centralizado. Você ainda tem vantagens como fazer commites offline e fazer branches e mesclagens infinitamente mais simples, mas o fluxo de trabalho pode ser bastante semelhante; a principal diferença é que mesclar acontece no lado do cliente ao invés do servidor na hora do commit. Veremos o que pode acontecer quando dois desenvolvedores começam a trabalhar juntos em um repositório compartilhado. O primeiro desenvolvedor, John, clona o repositório, faz uma alteração e um commit localmente. O protocolo de mensagens foi substituído por `...` nestes exemplos por simplificação.

```
# Máquina do John
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Remover valor padrão inválido'
[master 738ee87] Remover valor padrão inválido
 1 files changed, 1 insertions(+), 1 deletions(-)
```

O segundo desenvolvedor, Jessica, faz a mesma coisa — clona o repositório e faz um commit de uma alteração:

```
# Computador da Jessica
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim FAZER
$ git commit -am 'Adicionar tarefa para reiniciar'
[master fbff5bc] Adicionar tarefa para reiniciar
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Agora, Jessica dá um push do seu trabalho no servidor, o que funciona bem:

```
# Computador da Jessica
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

A última linha do resultado acima mostra uma mensagem de retorno útil da operação push. O formato básico é `<velharef>..<novaref> daref -> pararef`, onde `velharef` significa velha referência, `novaref` significa a nova referência, `daref` é o nome da referência local que está sendo publicada, e `pararef` é o nome da referência remota sendo atualizada. Você verá resultados semelhantes como este nas discussões seguintes, então ter uma ideia básica do significado irá ajudá-lo entender os vários estados dos repositórios. Mais detalhes estão disponíveis na documentação aqui git-push.

Continuando com este exemplo, pouco depois, John faz algumas modificações, um commit no seu repositório local, e tenta dar um push no mesmo servidor:

```
# Máquina do John
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

Neste caso, o push do John falhou por causa das alterações do push *que a Jessica fez* antes. Isto é especialmente importante de entender se você está acostumado ao Subversion, pois notará que os dois desenvolvedores não editaram o mesmo arquivo. Embora Subversion automaticamente faz uma mesclagem no servidor se arquivos diferentes foram editados, com Git, você deve *primeiro* mesclar os commites localmente. Em outras palavras, John deve primeiro buscar (*fetch*) as modificações anteriores feitas pela Jessica e mesclá-las no seu repositório local antes que ele possa fazer o seu push.

Como um passo inicial, John busca o trabalho de Jessica (apenas buscar o trabalho da Jessica, ainda não mescla no trabalho do John):

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

Neste ponto, o repositório local do John se parece com isso:

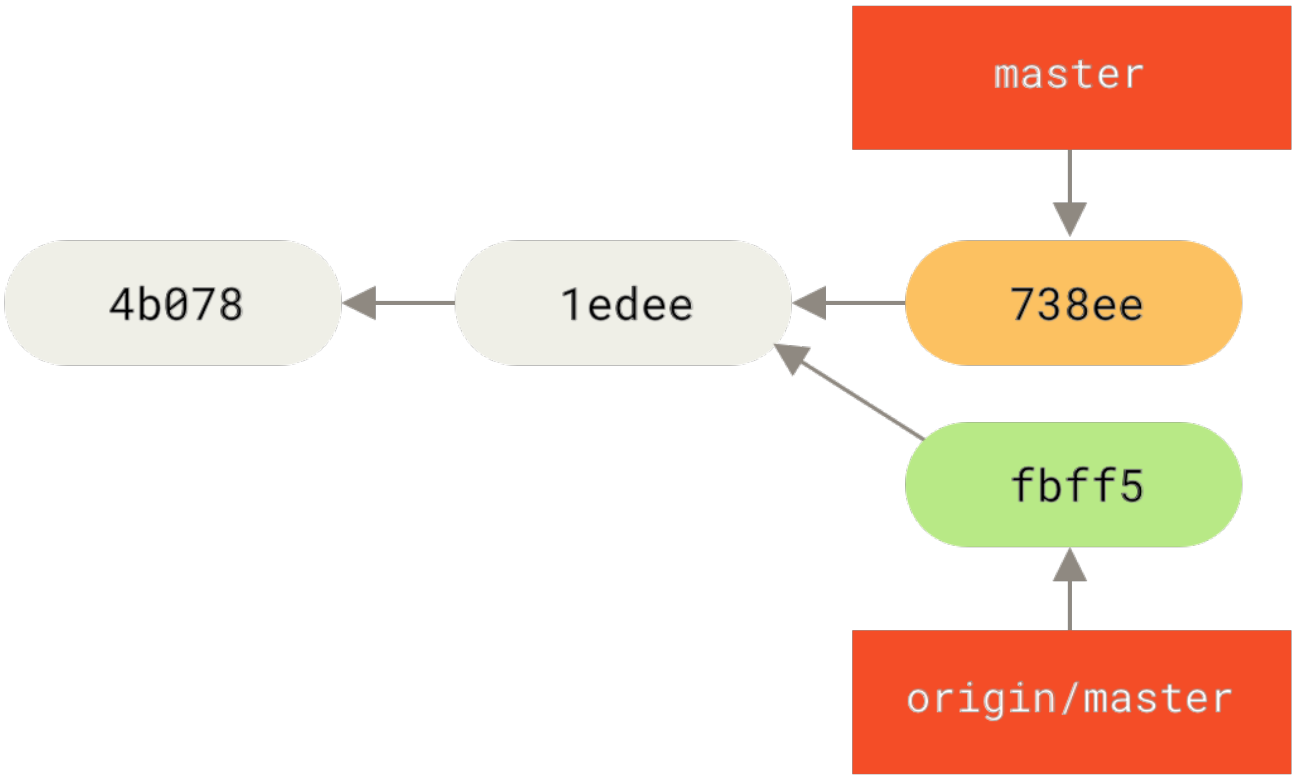


Figura 58. Histórico divergente do John

Agora John pode mesclar com o trabalho da Jessica que ele buscou no seu local de trabalho:

```
$ git merge origin/master
Merge made by the 'recursive' strategy.
 FAZER |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Contanto que a mesclagem local seja tranquila, o histórico atualizado do John será parecido com isto:

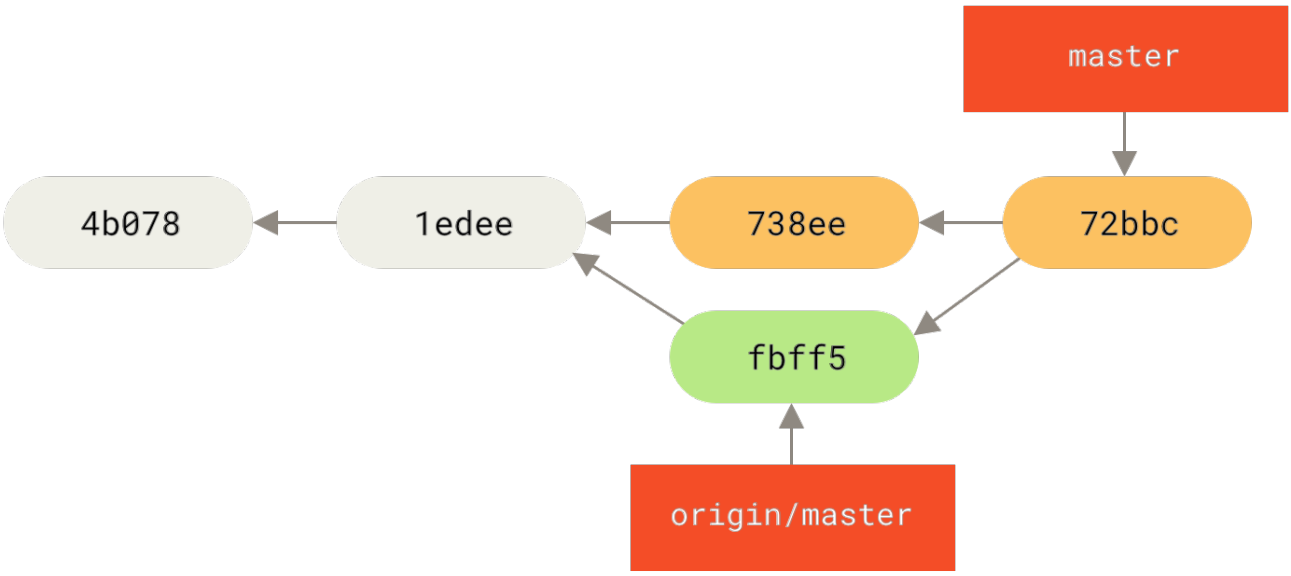


Figura 59. Repositório do John depois de mesclar `origin/master`

Neste ponto, John deve querer testar este código novo para se certificar que o trabalho da Jessica não afete o seu e, desde que tudo corra bem, ele pode finalmente publicar o novo trabalho combinado no servidor:

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

No fim, o histórico de commit do John ficará assim:

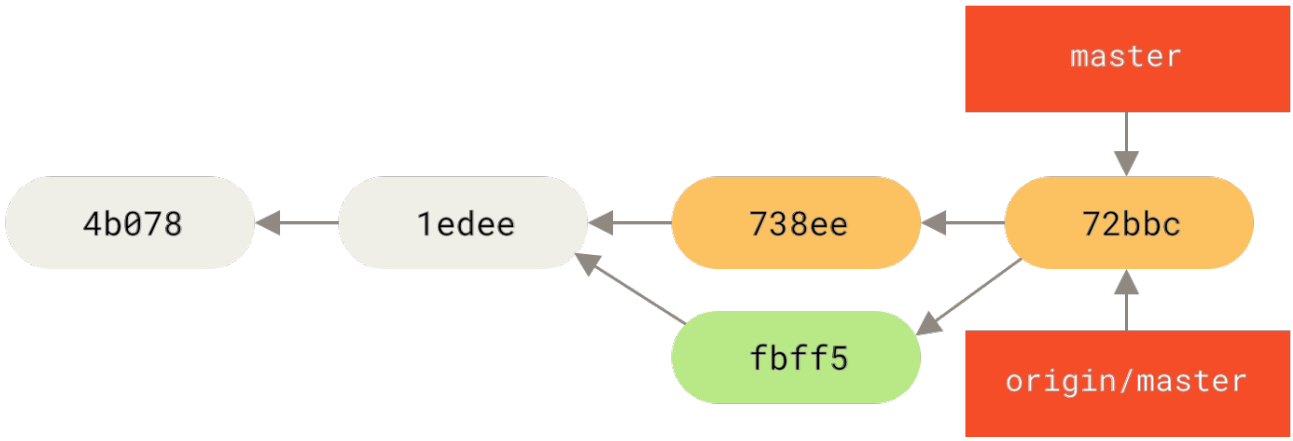


Figura 60. Histórico do John depois de publicar no servidor `origin`

Enquanto isso, Jessica criou um novo branch chamado `issue54`, e fez três commites naquele branch. Ela ainda não buscou as alterações do John, então o histórico de commites dela se parece com isso:

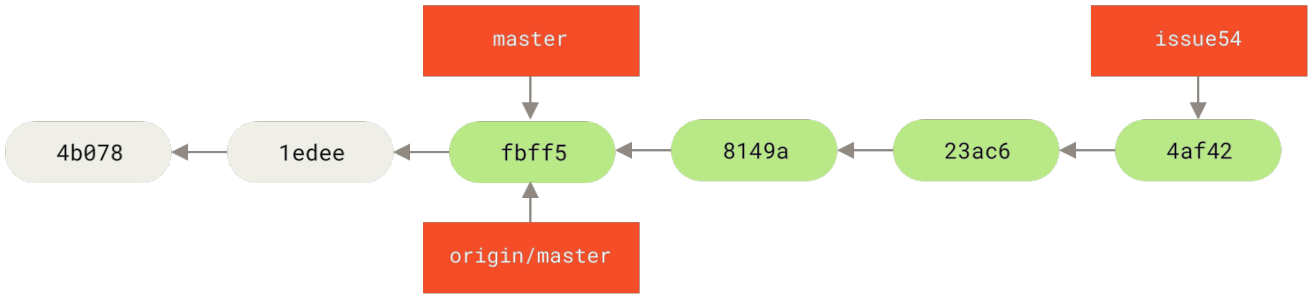


Figura 61. Branch atual da Jessica

De repente, Jessica percebe que o John publicou um trabalho novo no servidor e ela quer dar uma olhada, então ela busca todo o novo conteúdo do servidor que ela ainda não tem:

```
# Máquina da Jessica
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Isto baixa o trabalho que John publicou enquanto isso. O histórico da Jessica agora fica assim:

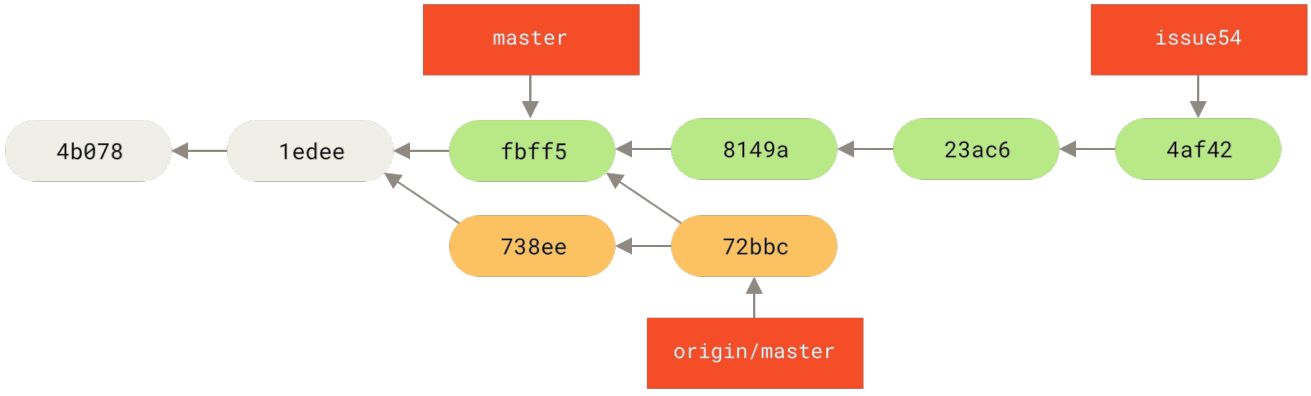


Figura 62. Histórico da Jessica depois de buscar as mudanças do John

Jessica acha que seu branch atual está pronto, mas quer saber que parte do trabalho que buscou do John ela deve combinar com o seu para publicar. Ela executa `git log` para encontrar:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   Remover valor padrão inválido
```

A sintaxe `issue54..origin/master` é um filtro de log que pede ao Git mostrar apenas estes commites que estão no branch seguinte (neste caso `origin/master`) e não estão no primeiro branch (neste caso `issue54`). Iremos cobrir esta sintaxe em detalhes em Commit Ranges.

Do resultado acima, podemos ver que há um único commit que John fez e Jessica não mesclou no trabalho local dela. Se ela mesclar `origin/master`, aquele é o único commit que irá modificar seu trabalho local.

Agora, Jessica pode mesclar seu trabalho atual no branch `master` dela, mesclar o trabalho de John (`origin/master`) no seu branch `master`, e então publicar devolta ao servidor.

Primeiro (tendo feito commit de todo o trabalho no branch atual dela `issue54`), Jessica volta para o seu branch `master` preparando-se para a integração de todo este trabalho:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commites, and can be fast-forwarded.
```

Jessica pode mesclar tanto `origin/master` ou `issue54` primeiro — ambos estão adiantados, então a ordem não importa. O retrato final deve ser idêntico não importando a ordem que ela escolher; apenas o histórico será diferente. Ela escolhe mesclar o branch `issue54` antes:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Nenhum problema ocorre; como você pode ver foi uma simples combinação direta. Jessica agora completa o processo local de combinação integrando o trabalho de John buscado anteriormente que estava esperando no branch `origin/master`:

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Tudo combina bem, e o histórico de Jessica agora se parece com isto:

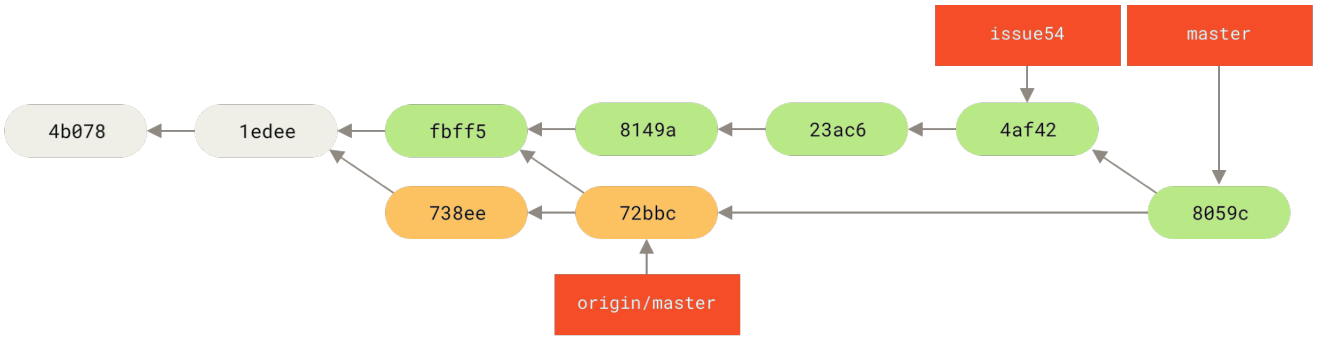


Figura 63. Histórico de Jessicadepois de integrar com as alterações de John

Agora `origin/master` é acessível para o branch `master` de Jessica, então ela deve ser capaz de publicar com sucesso (assumindo que John não publicou nenhuma outra modificação enquanto isso):

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

Cada desenvolvedor fez alguns commites e mesclou ao trabalho do outro com sucesso.

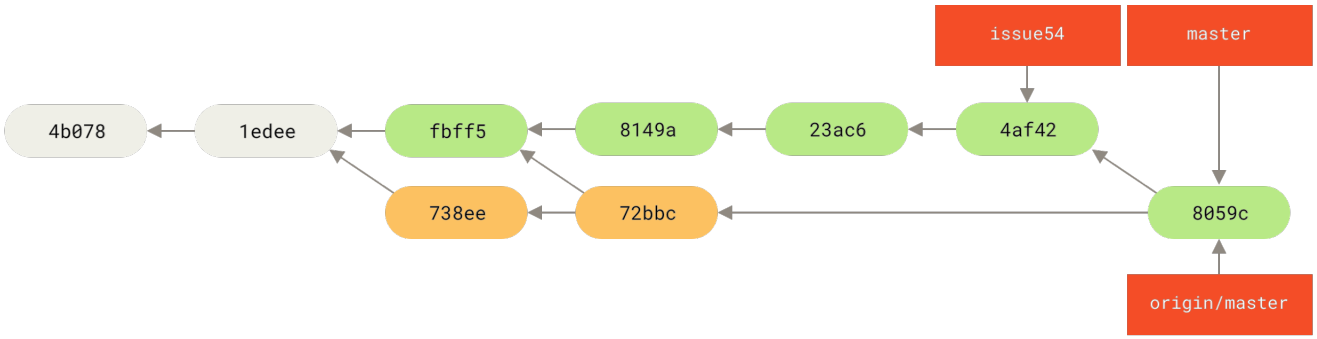


Figura 64. Histórico de Jessica depois de publicar todas alterações no servidor

Este é um dos mais simples fluxos de trabalho. Você trabalha um pouco (normalmente em um branch separado), e mescla este trabalho no seu branch `master` quando está pronto para ser integrado. Quando você quiser compartilhar este trabalho, você busca e mescla seu `master` ao `origin/master` se houve mudanças, e finalmente publica no branch `master` do servidor. A sequência comum é mais ou menos assim:

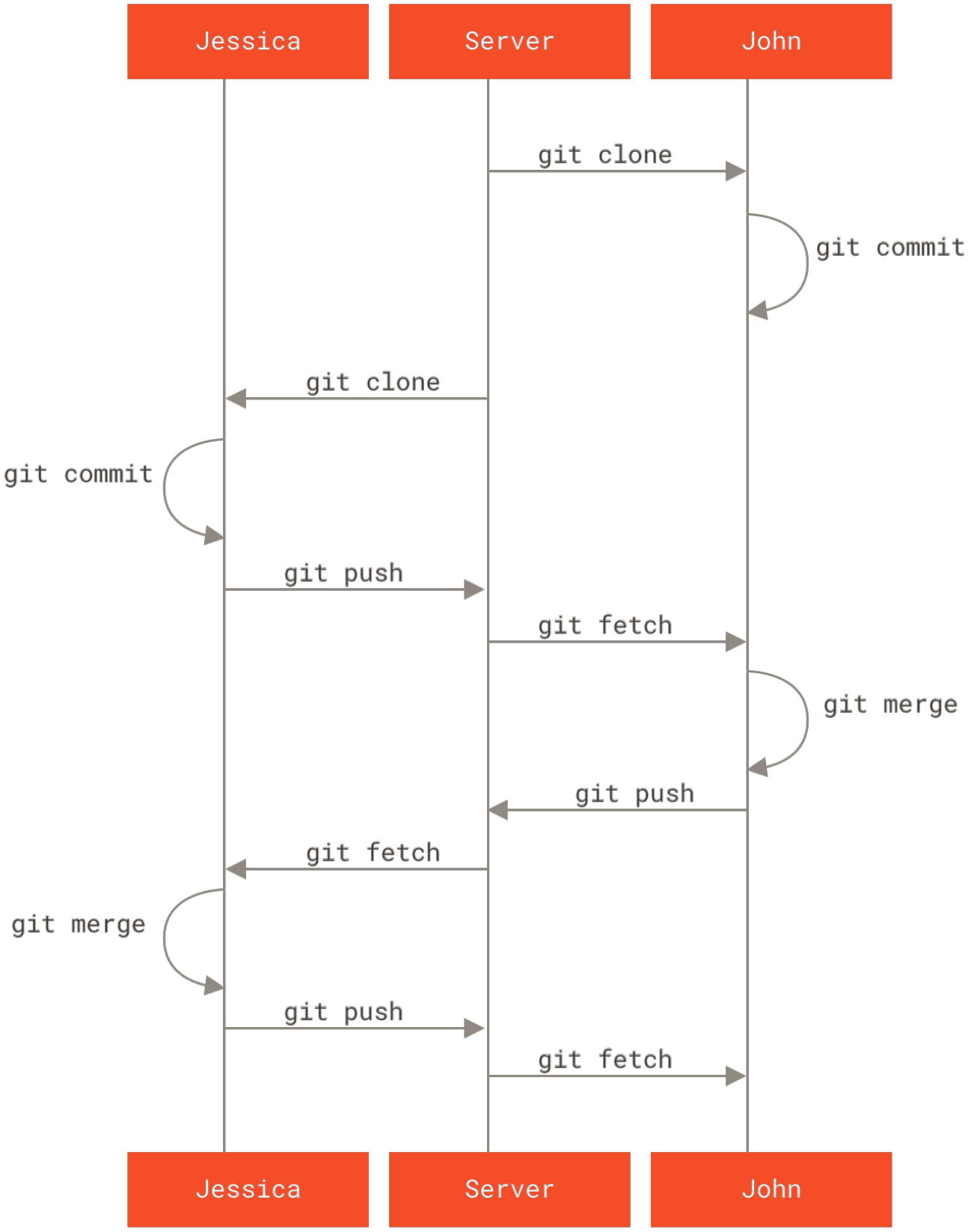


Figura 65. Sequência comum de eventos para um fluxo de trabalho simples no Git com múltiplos desenvolvedores

Time Privado Gerenciado

No próximo cenário, você irá observar os papéis de um colaborador em um grande grupo privado. Você irá aprender como trabalhar em um ambiente onde pequenos grupos colaboram em

componentes, então as contribuições deste time são integradas por outra equipe.

Digamos que John e Jessica estão trabalhando juntos em um componente (chamado `featureA`), enquanto Jessica e uma terceira desenvolvedora, Josie, estão trabalhando num outro (digamos `featureB`). Neste caso, a companhia está usando um tipo de fluxo de trabalho gerenciado onde o trabalho de grupos isolados é integrado apenas por certos engenheiros, e o branch `master` do repositório principal pode ser atualizado apenas por estes engenheiros. Neste cenário, todo trabalho é feito em branches da equipe e publicados depois pelos coordenadores.

Vamos acompanhar o fluxo de trabalho da Jessica enquanto ela trabalha em seus dois componentes, colaborando em paralelo com dois desenvolvedores neste ambiente. Assumindo que ela já tem seu repositório clonado, ela decide trabalhar no `featureA` antes. Ela cria um novo branch para o componente e trabalha um pouco nele:

```
# Máquina da Jessica
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'Adicionar limite a função log'
[featureA 3300904] Adicionar limite a função log
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Neste ponto, ela precisa compartilhar seu trabalho com John, então ela publica seus commites no branch `featureA` do servidor. Jessica não tem permissão de publicar no branch `master` — apenas os coordenadores tem — então ela publica em outro branch para trabalhar com John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica comunica John que publicou alguma coisa no branch chamado `featureA` e ele pode dar uma olhada agora. Enquanto ela espera John responder, Jessica decide começar a trabalhar no `featureB` com Josie. Para começar, ela cria um novo branch para o componente, baseando-se no branch `master` do servidor:

```
# Máquina da Jessica
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Agora Jessica faz alguns commites no branch `featureB`:

```
$ vim lib/simplegit.rb
$ git commit -am 'Tornar a função ls-tree recursiva'
[featureB e5b0fdc] Tornar a função ls-tree recursiva
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'Adicionar ls-files'
[featureB 8512791] Adicionar ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

O repositório de Jessica agora fica assim:

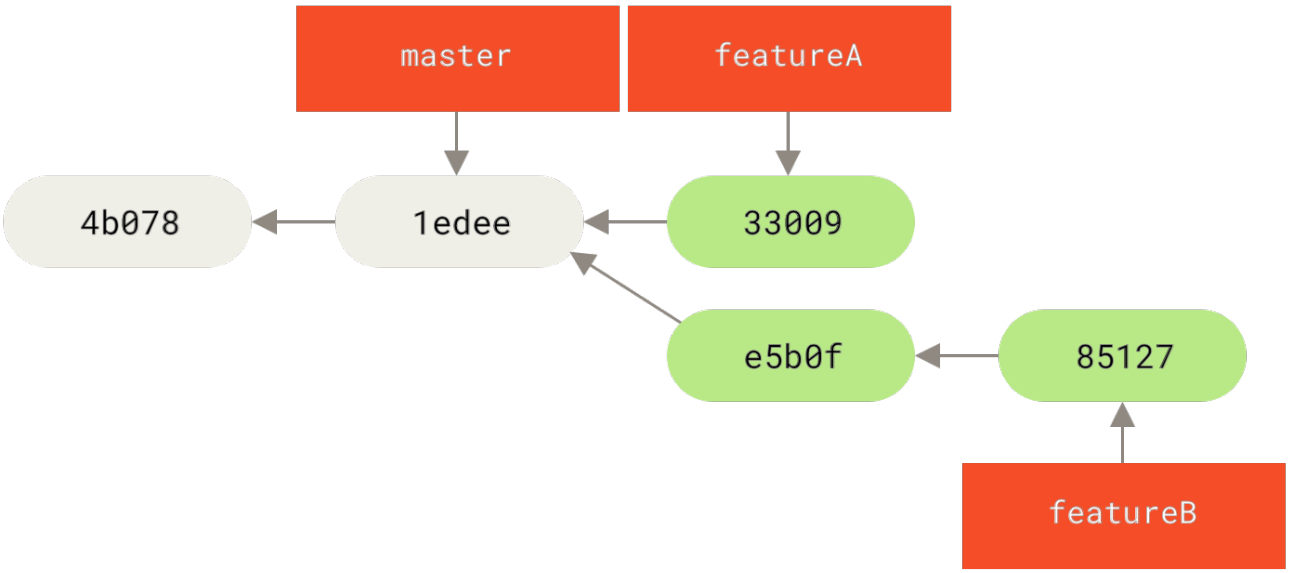


Figura 66. Histórico do commit inical de Jessica

Ela já está pronta para publicar seu trabalho, mas recebe um email de Josie dizendo que um branch contendo um `featureB` inicial já foi publicado no servidor como `featureBee`. Jessica precisa combinar estas alterações com as suas antes que ela possa publicar seu trabalho no servidor. Jessica primeiro busca as mudanças de Josie com `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Assumindo que Jessica ainda está no seu branch `featureB`, ela pode agora incorporar o trabalho de Josie neste branch com `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Neste ponto, Jessica quer publicar todo este `featureB` mesclado no servidor, mas ela não quer simplesmente publicar no seu branch `featureB`. Ao invés disso, como Josie já começou um branch `featureBee`, Jessica quer publicar *neste* branch, que ela faz assim:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

Isto é chamado de *refspec*. Veja The Refspec para uma discussão mais detalhada sobre os refspecs do Git e as diferentes coisas que você pode fazer com eles. Também perceba a flag `-u`; isto é abreviação de `--set-upstream`, que configura os branches para facilitar publicar e baixar depois.

De repente, Jessica recebe um email de John, contando que publicou algumas modificações no branch `featureA` no qual eles estavam colaborando, e ele pede a Jessica para dar uma olhada. Denovo, Jessica executa um simples `git fetch` para buscar *todo* novo conteúdo do servidor, incluindo (é claro) o último trabalho de John:

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

Jessica pode exibir o log do novo trabalho de John comparando o conteúdo do branch `featureA` recentemente buscado com sua cópia local do mesmo branch:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    Aumentar resultados do log para 30 de 25
```

Se Jessica gostar do que vê, ela pode integrar o recente trabalho de John no seu branch `featureA` local com:

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Finalmente, Jessica pode querer fazer algumas pequenas alterações de todo o conteúdo mesclado, então ela está livre para fazer estas mudanças, commitar elas no seu branch `featureA` local, e

publicar o resultado de volta ao servidor:

```
$ git commit -am 'Adicionar pequeno ajuste ao conteúdo mesclado'
[featureA 774b3ed] Adicionar pequeno ajuste ao conteúdo mesclado
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

O histórico de commites da Jessica agora está assim:

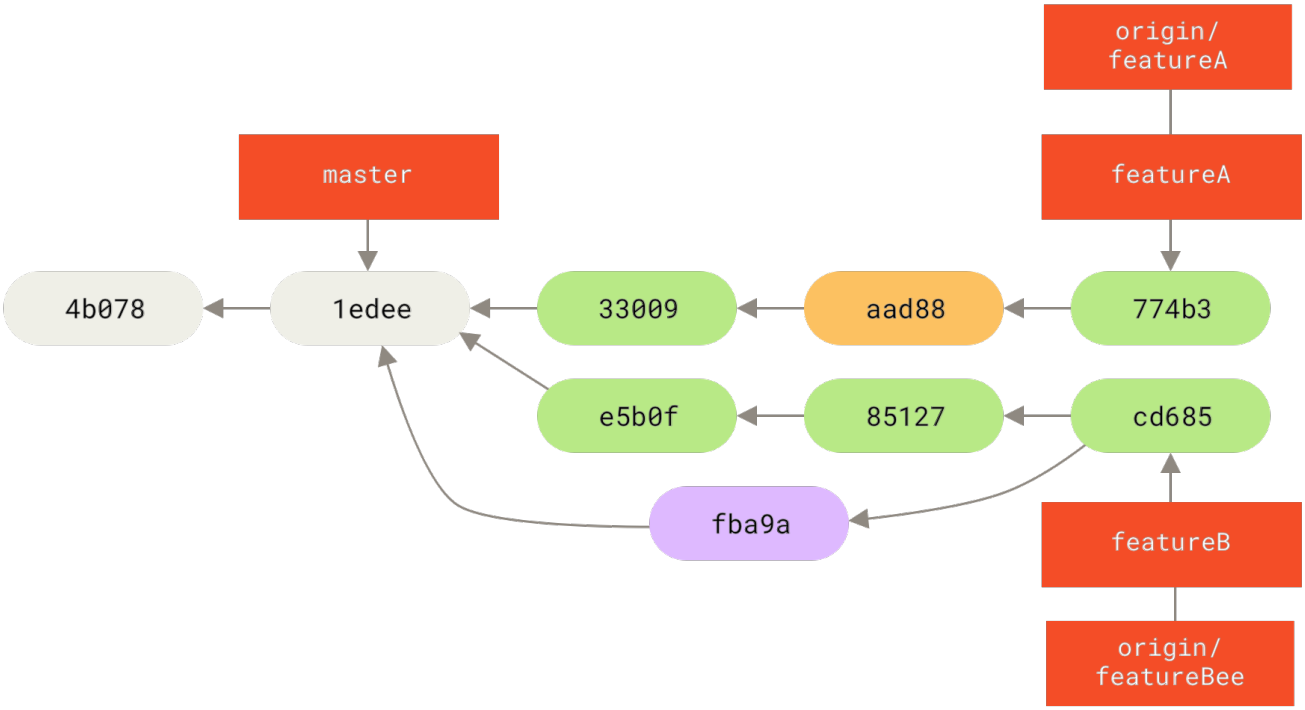


Figura 67. Histórico da Jessica depois dos commites em um branch de componentes

Em algum ponto, Jessica, Josie e John informam os coordenadores que os branches `featureA` e `featureBee` no servidor estão prontos para serem integrados no código principal. Depois dos coordenadores mesclarem estes branches no código principal, um *fetch* trará o novo commit mesclado, deixando o histórico assim:

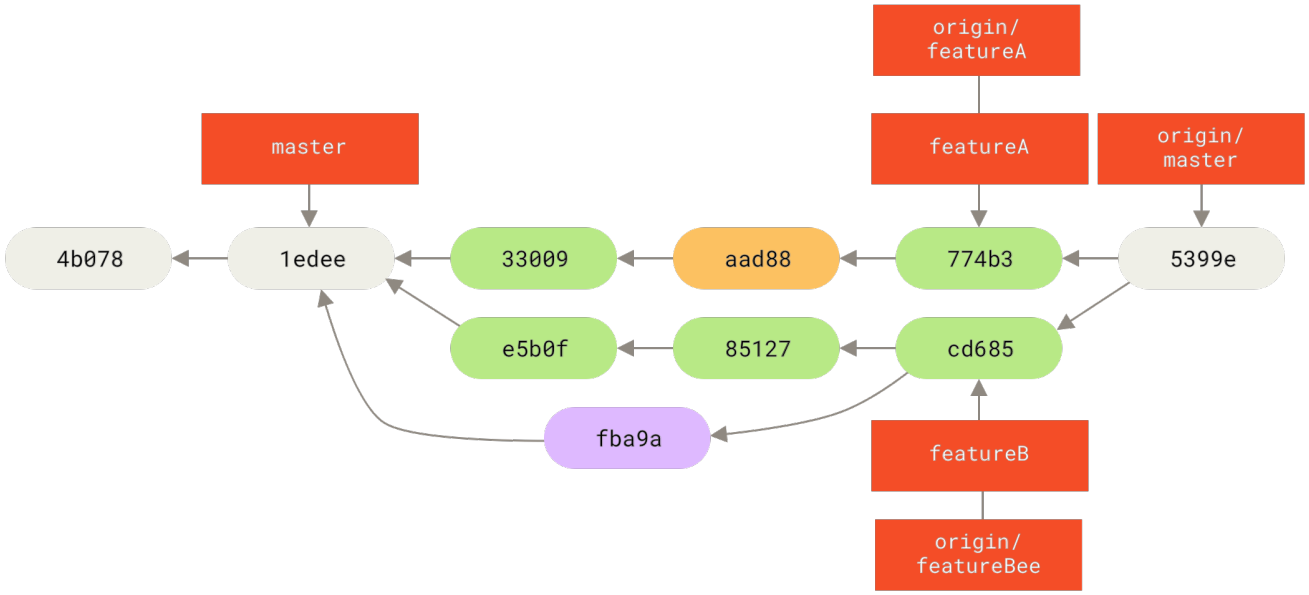


Figura 68. Histórico da Jessica depois de integrar ambos seus branches

Muitos grupos migraram para o Git pela possibilidade de ter múltiplos times trabalhando em paralelo, combinando as diferentes frentes de trabalho mais tarde no processo. A habilidade de pequenos subgrupos da equipe colaborar através de branches remotos sem necessariamente ter que envolver ou atrasar a equipe inteira é um benefício imenso do Git. A sequência do fluxo de trabalho que você viu aqui é parecida com isto:

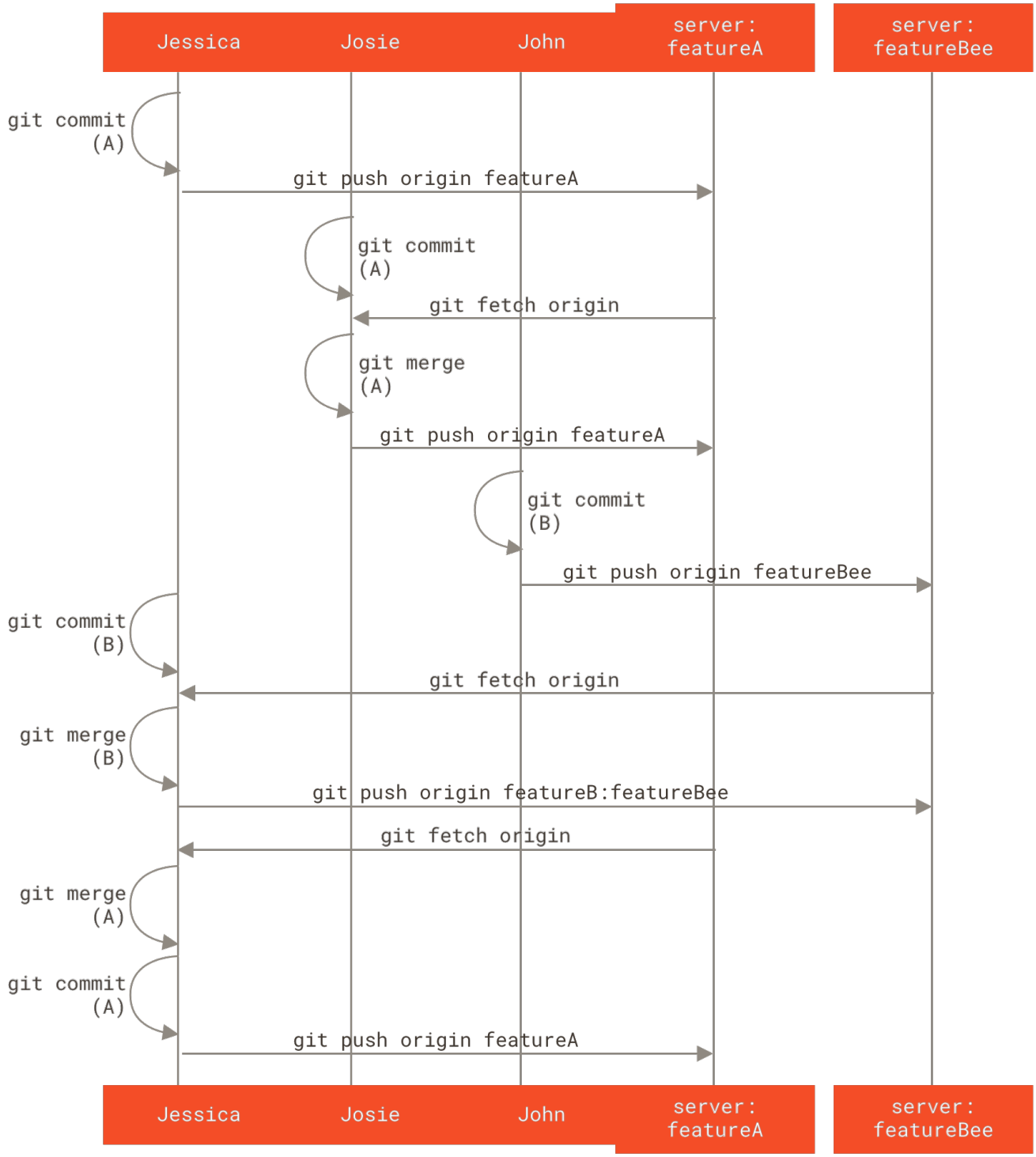


Figura 69. Sequencia básica deste fluxo de trabalho coordenado

Fork de Projeto Público

Colaborando com projetos públicos é um pouco diferente. Como você não tem as permissões para atualizar diretamente branches no projeto, você deve levar seu trabalho aos coordenadores de algum jeito diferente. O primeiro exemplo descreve como contribuir através de *forks* em um site de hospedagem Git que permite fazer forks facilmente. Muitos sites de hospedagem suportam isto (incluindo GitHub, BitBucket, repo.or.cz, e outros), e muitos mantenedores de projetos esperam este estilo de contribuição. A próxima seção lida com projetos que preferem aceitar patches de contribuição por email.

Primeiro, você provavelmente irá preferir clonar o repositório principal, criar um branch específico para o patch ou série de patchs que está planejando contribuir, e fazer o seu trabalho ali. A sequência fica basicamente assim:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
  ... trabalho ...
$ git commit
  ... trabalho ...
$ git commit
```

NOTA Você pode querer usar `rebase -i` para resumir seu trabalho a um único commit, ou rearranjar o trabalho em commites que deixarão o trabalho mais fácil para os mantenedores revisarem — veja Rewriting History para mais informações sobre rebase interativo.

Quando seu trabalho no branch é finalizado e você está pronto para mandá-lo para os mantenedores, vá para a página original do projeto e clique no botão "Fork", criando seu próprio fork editável do projeto. Você precisa então adicionar a URL deste repositório como um novo repositório remoto do seu repositório local; neste exemplo, vamos chamá-lo `meufork`:

```
$ git remote add meufork <url>
```

Você então precisa publicar seu trabalho neste repositório. É mais fácil publicar o branch em que você está trabalhando no seu repositório fork, ao invés de mesclar este trabalho no seu branch `master` e publicar assim. A razão é que se o seu trabalho não for aceito ou for selecionado a dedo (*cherry-pick*), você não tem que voltar seu branch `master` (a operação do Git `cherry-pick` é vista em mais detalhes em Rebasing and Cherry-Picking Workflows). Se os mantenedores executarem um `merge`, `rebase` ou `cherry-pick` no seu trabalho, você irá eventualmente receber seu trabalho de novo através do repositório deles de qualquer jeito.

Em qualquer um dos casos, você pode publicar seu trabalho com:

```
$ git push -u meufork featureA
```

Uma vez que seu trabalho tenha sido publicado no repositório do seu fork, você deve notificar os mantenedores do projeto original que você tem trabalho que gostaria que eles incorporassem no código. Isto é comumente chamado de *pull request*, e você tipicamente gera esta requisição ou através do website — GitHub tem seu próprio mecanismo de "Pull Request" que iremos abordar em GitHub — ou você pode executar o comando `git request-pull` e mandar um email com o resultado para o mantenedor do projeto manualmente.

O comando `git request-pull` pega o branch base no qual você quer o seu branch atual publicado e a URL do repositório Git de onde você vai buscar, e produz um resumo de todas as mudanças que você está tentando publicar. Por exemplo, se Jessica quer mandar a John uma pull request, e ela fez

dois commites no branch que ela acabou de publicar, ela pode executar:

```
$ git request-pull origin/master meufork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
        Criar nova função

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      Adicionar limite para a função log
      Aumentar a saída do log para 30 de 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Este resultado pode ser mandado para os mantenedores — ele diz de qual branch o trabalho vem, resume os commites, e identifica onde o novo trabalho será publicado.

Em um projeto em que você não é um mantenedor, é geralmente mais fácil ter um branch principal `master` sempre rastreando `origin/master` e fazer seu trabalho em branches separados que você pode facilmente descartar se eles forem rejeitados. Tendo temas de trabalho isolados em branches próprios também facilita para você realocar seu trabalho se a ponta do repositório principal se mover enquanto trabalha e seus commites não mais puderem ser aplicados diretamente. Por exemplo, se você quer publicar um segundo trabalho numa outra área do projeto, não continue trabalhando no branch que você acabou de publicar — comece um novo branch apartir do branch no repositório principal `master`:

```
$ git checkout -b featureB origin/master
  ... trabalho ...
$ git commit
$ git push meufork featureB
$ git request-pull origin/master meufork
  ... email generated request pull to maintainer ...
$ git fetch origin
```

Agora, cada um dos seus assuntos está contido em um casulo — igual a um patch na fila — que você pode reescrever, realocar, e modificar sem que os outros assuntos interfiram com ele, deste jeito:

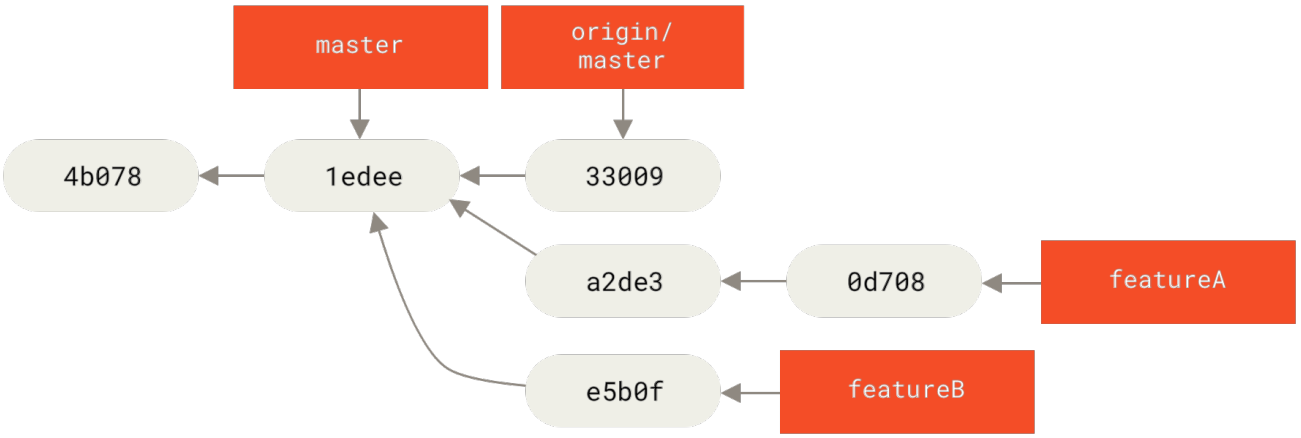


Figura 70. Histórico de commites inicial com o trabalho `featureB`

Digamos que o mantenedor do projeto tenha integrado vários outros patches e, quando tentou seu primeiro branch, seu trabalho não mais combina facilmente. Neste caso, você pode tentar realocar seu branch no topo de `origin/master`, resolver os conflitos para o mantenedor, e então republicar suas alterações:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f meufork featureA
```

Isto reescreve seu histórico que agora fica assim Histórico de commites depois do trabalho em `featureA`.

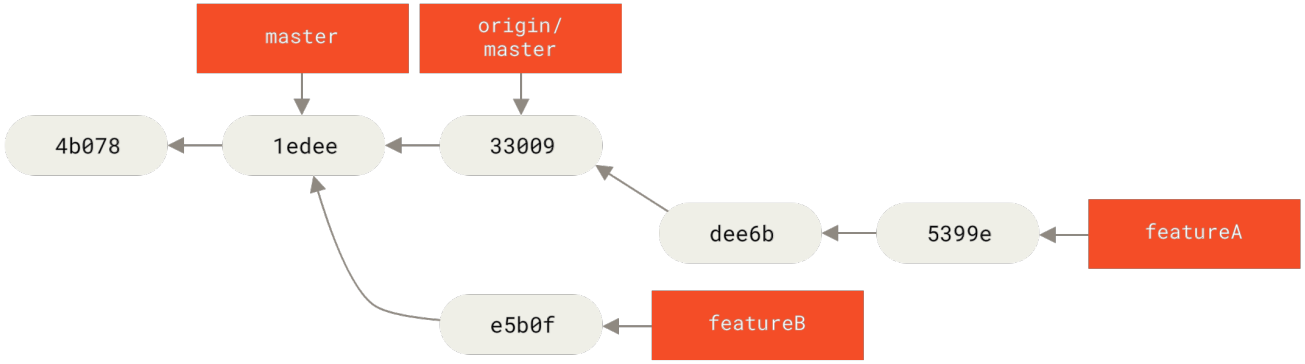


Figura 71. Histórico de commites depois do trabalho em `featureA`

Como você realocou seu branch, você deve especificar `-f` para seu comando push substituir o branch `featureA` no servidor com um commit que não é um descendente dele. Uma alternativa seria publicar seu novo trabalho em um branch diferente no servidor (talvez chamado `featureAv2`).

Vamos olhar um outro cenário possível: o mantenedor viu seu trabalho no seu branch secundário e gosta do conceito mas gostaria que você mudasse um detalhe de implementação. Você aproveita esta oportunidade para basear seu trabalho no branch `master` do projeto atual. Você inicia um novo branch baseado no branch `origin/master` atual, compacta (*squash*) as mudanças do `featureB` ali, resolve qualquer conflito, faz a mudança de implementação, e então publica isto como um novo branch:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
  ... mudando implementação ...
$ git commit
$ git push meufork featureBv2
```

A opção `--squash` pega todo o trabalho no branch mesclado e comprime em um novo conjunto gerando um novo estado de repositório como se uma mescla real tivesse acontecido, sem realmente mesclar os commites. Isto significa que seu commit futuro terá um pai apenas e te permite introduzir todas as mudanças de um outro branch e então aplicar mais alterações antes de gravar seu novo commit. A opção `--no-commit` também pode ser útil para atrasar a integração do commit no caso do processo de mesclagem padrão.

Neste ponto, você pode notificar os mantenedores que você já fez as mudanças pedidas, e que eles podem encontrá-las no branch `featureBv2`.

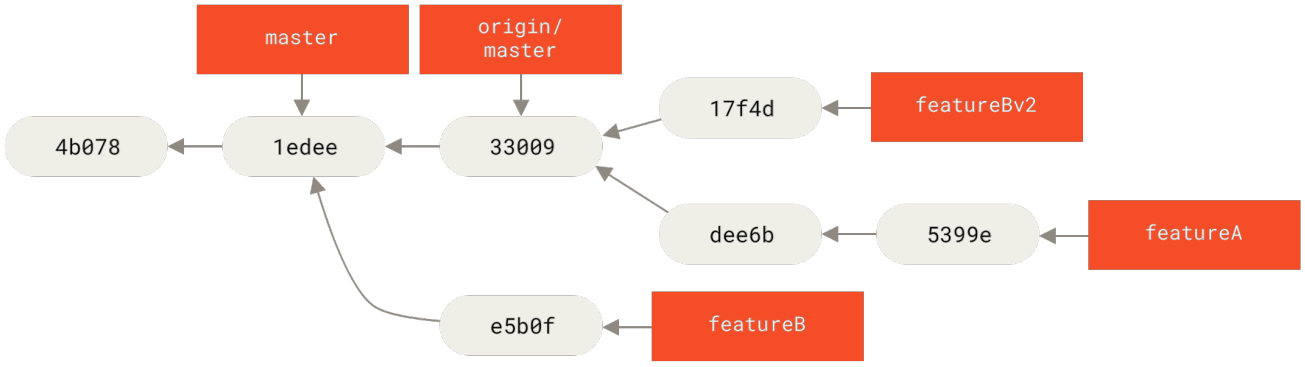


Figura 72. Histórico de commit depois do trabalho em `featureBv2`

Projeto Público através de Email

Muitos projetos têm procedimentos estabelecidos para aceitar patches — você irá precisar checar as regras específicas para cada projeto, pois elas serão diferentes. Já que existem vários projetos mais antigos, maiores, que aceitam patches através de uma lista de emails de desenvolvedores, iremos exemplificar isto agora:

O fluxo de trabalho é parecido ao caso anterior — você cria branches separados para cada série de patches em que trabalhar. A diferença é como enviá-los ao projeto. Ao invés de fazer um fork do projeto e publicar sua própria versão editável, você gera versões de email de cada série de commites e as envia para a lista de email dos desenvolvedores:

```
$ git checkout -b assuntoA
  ... trabalho ...
$ git commit
  ... trabalho ...
$ git commit
```

Agora você tem dois commites que gostaria de enviar para a lista de email. Você pode usar `git format-patch` para gerar os arquivos formatados em mbox e enviar para a lista de email — isto transforma cada commit em uma mensagem de email, com a primeira linha da mensagem do commit como o assunto, e o resto da mensagem mais o patch que o commit traz como o corpo do email. O legal disto é que aplicar um patch de email gerado com `format-patch` preserva todas as informações de commit corretamente.

```
$ git format-patch -M origin/master
0001-adicionar-limite-para-a-função-log.patch
0002-aumentar-a-saída-do-log-para-30-de-25.patch
```

O comando `format-patch` exibe os nomes dos arquivos de patch que cria. A chave `-M` diz ao Git para procurar por renomeações. Os arquivos acabam se parecendo com isto:

```
$ cat 0001-adicionar-limite-para-a-função-log.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Adicionar limite a função log

Limitar a função log aos primeiros 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```

Você também pode editar estes arquivos de patch para adicionar mais informação para a lista de email que você não gostaria de colocar na mensagem do commit. Se você adicionar texto entre a linha `---` e o começo do patch (a linha `diff --git), os desenvolvedores podem ler, mas o conteúdo é ignorado pelo processo de patch.

Para enviar este email para uma lista de emails, você pode tanto colar o arquivo no seu programa

de email quanto enviar via um programa de linha de comando. Colar o texto geralmente causa problemas de formatação, especialmente com programas ``inteligentes`` que não preservam novas linhas e espaços em branco corretamente. Felizmente, o Git fornece uma ferramenta para ajudar você enviar patches formatados adequadamente através de IMAP, o que pode ser mais fácil para você. Iremos demonstrar como enviar um patch via Gmail, no caso o veículo que conhecemos melhor; você pode ler instruções detalhadas de vários programas de email no final do acima mencionado arquivo `Documentation/SubmittingPatches` no código fonte do Git.

Primeiro, você deve configurar a seção imap no seu arquivo `~/.gitconfig`. Você pode configurar cada valor separadamente com uma série de comandos `git config`, ou adicioná-los manualmente, mas no final seu arquivo config deve ficar assim:

```
[imap]
  folder = "[Gmail]/Rascunhos"
  host = imaps://imap.gmail.com
  user = usuario@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

Se o seu servidor IMAP não usa SSL, as duas últimas linhas provavelmente não são necessárias, e o valor de host será `imap://` ao invés de `imaps://`. Quando isto estiver pronto, você poderá usar `git imap-send` para colocar os seus patches no diretório Rascunhos no servidor IMAP especificado:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

Neste ponto, você deve poder ir ao seu diretório Rascunhos, mudar o campo Para com a lista de email para qual você está mandando o patch, possivelmente copiando (*CC*) os mantenedores ou pessoas responsáveis pela seção, e enviar o patch.

Você também pode enviar os patches através de um servidor SMTP. Como antes, você pode configurar cada valor separadamente com uma série de comandos `git config`, ou você pode adicioná-los manualmente na seção sendemail no seu arquivo `~/.gitconfig`:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = usuario@gmail.com
  smtpserverport = 587
```

Depois que isto estiver pronto, você pode usar `git send-email` para enviar os seus patches:

```
$ git send-email *.patch
0001-adicionar-limite-para-a-função-log.patch
0002-aumentar-a-saída-do-log-para-30-de-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Então, o Git retorna várias informações de log, para cada patch que você está enviando:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] Adicionar limite a função log
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

DICA	Se quiser ajuda para configurar seu sistema de email, mais dicas e truques, e um ambiente controlado para enviar um patch de teste através de email, acesse git-send-email.io.

Resumo

Nesta seção nós abordamos múltiplos fluxos de trabalho, e falamos sobre as diferenças entre trabalhar como parte de uma equipe pequena em projetos de código fechado, e contribuir para um grande projeto público. Você sabe checar por erros de espaços em branco antes do seu commit, e pode escrever uma excelente mensagem nele. Você aprendeu como formatar patches, e enviá-los por email para a lista de desenvolvedores. Lidando com combinações também foi coberto no contexto de diferentes fluxos de trabalho. Você está agora bem preparado para colaborar com qualquer projeto.

A seguir, você verá como trabalhar no outro lado da moeda: mantendo um projeto Git. Aprenderá como ser um ditador benevolente ou um coordenador.

Maintaining a Project

In addition to knowing how to contribute effectively to a project, you'll likely need to know how to maintain one. This can consist of accepting and applying patches generated via `format-patch` and

emailed to you, or integrating changes in remote branches for repositories you've added as remotes to your project. Whether you maintain a canonical repository or want to help by verifying or approving patches, you need to know how to accept work in a way that is clearest for other contributors and sustainable by you over the long run.

Working in Topic Branches

When you're thinking of integrating new work, it's generally a good idea to try it out in a *topic branch* — a temporary branch specifically made to try out that new work. This way, it's easy to tweak a patch individually and leave it if it's not working until you have time to come back to it. If you create a simple branch name based on the theme of the work you're going to try, such as `ruby_client` or something similarly descriptive, you can easily remember it if you have to abandon it for a while and come back later. The maintainer of the Git project tends to namespace these branches as well — such as `sc/ruby_client`, where `sc` is short for the person who contributed the work. As you'll remember, you can create the branch based off your `master` branch like this:

```
$ git branch sc/ruby_client master
```

Or, if you want to also switch to it immediately, you can use the `checkout -b` option:

```
$ git checkout -b sc/ruby_client master
```

Now you're ready to add the contributed work that you received into this topic branch and determine if you want to merge it into your longer-term branches.

Applying Patches from Email

If you receive a patch over email that you need to integrate into your project, you need to apply the patch in your topic branch to evaluate it. There are two ways to apply an emailed patch: with `git apply` or with `git am`.

Applying a Patch with apply

If you received the patch from someone who generated it with `git diff` or some variation of the Unix `diff` command (which is not recommended; see the next section), you can apply it with the `git apply` command. Assuming you saved the patch at `/tmp/patch-ruby-client.patch`, you can apply the patch like this:

```
$ git apply /tmp/patch-ruby-client.patch
```

This modifies the files in your working directory. It's almost identical to running a `patch -p1` command to apply the patch, although it's more paranoid and accepts fewer fuzzy matches than patch. It also handles file adds, deletes, and renames if they're described in the `git diff` format, which `patch` won't do. Finally, `git apply` is an "apply all or abort all" model where either everything is applied or nothing is, whereas `patch` can partially apply patchfiles, leaving your working directory in a weird state. `git apply` is overall much more conservative than `patch`. It won't create a

commit for you — after running it, you must stage and commit the changes introduced manually.

You can also use `git apply` to see if a patch applies cleanly before you try actually applying it — you can run `git apply --check` with the patch:

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

If there is no output, then the patch should apply cleanly. This command also exits with a non-zero status if the check fails, so you can use it in scripts if you want.

Applying a Patch with `am`

If the contributor is a Git user and was good enough to use the `format-patch` command to generate their patch, then your job is easier because the patch contains author information and a commit message for you. If you can, encourage your contributors to use `format-patch` instead of `diff` to generate patches for you. You should only have to use `git apply` for legacy patches and things like that.

To apply a patch generated by `format-patch`, you use `git am` (the command is named `am` as it is used to "apply a series of patches from a mailbox"). Technically, `git am` is built to read an mbox file, which is a simple, plain-text format for storing one or more email messages in one text file. It looks something like this:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Add limit to log function

Limit log functionality to the first 20
```

This is the beginning of the output of the `git format-patch` command that you saw in the previous section; it also represents a valid mbox email format. If someone has emailed you the patch properly using `git send-email`, and you download that into an mbox format, then you can point `git am` to that mbox file, and it will start applying all the patches it sees. If you run a mail client that can save several emails out in mbox format, you can save entire patch series into a file and then use `git am` to apply them one at a time.

However, if someone uploaded a patch file generated via `git format-patch` to a ticketing system or something similar, you can save the file locally and then pass that file saved on your disk to `git am` to apply it:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

You can see that it applied cleanly and automatically created the new commit for you. The author

information is taken from the email's `From` and `Date` headers, and the message of the commit is taken from the `Subject` and body (before the patch) of the email. For example, if this patch was applied from the mbox example above, the commit generated would look something like this:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   Add limit to log function

   Limit log functionality to the first 20
```

The `Commit` information indicates the person who applied the patch and the time it was applied. The `Author` information is the individual who originally created the patch and when it was originally created.

But it's possible that the patch won't apply cleanly. Perhaps your main branch has diverged too far from the branch the patch was built from, or the patch depends on another patch you haven't applied yet. In that case, the `git am` process will fail and ask you what you want to do:

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation. You solve this issue much the same way — edit the file to resolve the conflict, stage the new file, and then run `git am --resolved` to continue to the next patch:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

If you want Git to try a bit more intelligently to resolve the conflict, you can pass a `-3` option to it, which makes Git attempt a three-way merge. This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository. If you do have that commit — if the patch was based on a public commit — then the `-3` option is generally much smarter about applying a conflicting patch:

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In this case, without the `-3` option the patch would have been considered as a conflict. Since the `-3` option was used the patch applied cleanly.

If you're applying a number of patches from an mbox, you can also run the `am` command in interactive mode, which stops at each patch it finds and asks if you want to apply it:

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
See if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

This is nice if you have a number of patches saved, because you can view the patch first if you don't remember what it is, or not apply the patch if you've already done so.

When all the patches for your topic are applied and committed into your branch, you can choose whether and how to integrate them into a longer-running branch.

Checking Out Remote Branches

If your contribution came from a Git user who set up their own repository, pushed a number of changes into it, and then sent you the URL to the repository and the name of the remote branch the changes are in, you can add them as a remote and do merges locally.

For instance, if Jessica sends you an email saying that she has a great new feature in the `ruby-client` branch of her repository, you can test it by adding the remote and checking out that branch locally:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

If she emails you again later with another branch containing another great feature, you could directly `fetch` and `checkout` because you already have the remote setup.

This is most useful if you're working with a person consistently. If someone only has a single patch to contribute once in a while, then accepting it over email may be less time consuming than requiring everyone to run their own server and having to continually add and remove remotes to get a few patches. You're also unlikely to want to have hundreds of remotes, each for someone who

contributes only a patch or two. However, scripts and hosted services may make this easier — it depends largely on how you develop and how your contributors develop.

The other advantage of this approach is that you get the history of the commits as well. Although you may have legitimate merge issues, you know where in your history their work is based; a proper three-way merge is the default rather than having to supply a `-3` and hope the patch was generated off a public commit to which you have access.

If you aren't working with a person consistently but still want to pull from them in this way, you can provide the URL of the remote repository to the `git pull` command. This does a one-time pull and doesn't save the URL as a remote reference:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by the 'recursive' strategy.
```

Determining What Is Introduced

Now you have a topic branch that contains contributed work. At this point, you can determine what you'd like to do with it. This section revisits a couple of commands so you can see how you can use them to review exactly what you'll be introducing if you merge this into your main branch.

It's often helpful to get a review of all the commits that are in this branch but that aren't in your `master` branch. You can exclude commits in the `master` branch by adding the `--not` option before the branch name. This does the same thing as the `master..contrib` format that we used earlier. For example, if your contributor sends you two patches and you create a branch called `contrib` and applied those patches there, you can run this:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    See if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    Update gemspec to hopefully work better
```

To see what changes each commit introduces, remember that you can pass the `-p` option to `git log` and it will append the diff introduced to each commit.

To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results. You may think to run this:

```
$ git diff master
```

This command gives you a diff, but it may be misleading. If your `master` branch has moved forward since you created the topic branch from it, then you'll get seemingly strange results. This happens because Git directly compares the snapshots of the last commit of the topic branch you're on and the snapshot of the last commit on the `master` branch. For example, if you've added a line in a file on the `master` branch, a direct comparison of the snapshots will look like the topic branch is going to remove that line.

If `master` is a direct ancestor of your topic branch, this isn't a problem; but if the two histories have diverged, the diff will look like you're adding all the new stuff in your topic branch and removing everything unique to the `master` branch.

What you really want to see are the changes added to the topic branch — the work you'll introduce if you merge this branch with `master`. You do that by having Git compare the last commit on your topic branch with the first common ancestor it has with the `master` branch.

Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

or, more concisely:

```
$ git diff $(git merge-base contrib master)
```

However, neither of those is particularly convenient, so Git provides another shorthand for doing the same thing: the triple-dot syntax. In the context of the `git diff` command, you can put three periods after another branch to do a `diff` between the last commit of the branch you're on and its common ancestor with another branch:

```
$ git diff master...contrib
```

This command shows you only the work your current topic branch has introduced since its common ancestor with `master`. That is a very useful syntax to remember.

Integrating Contributed Work

When all the work in your topic branch is ready to be integrated into a more mainline branch, the question is how to do it. Furthermore, what overall workflow do you want to use to maintain your project? You have a number of choices, so we'll cover a few of them.

Merging Workflows

One basic workflow is to simply merge all that work directly into your `master` branch. In this scenario, you have a `master` branch that contains basically stable code. When you have work in a topic branch that you think you've completed, or work that someone else has contributed and you've verified, you merge it into your master branch, delete that just-merged topic branch, and repeat.

For instance, if we have a repository with work in two branches named `ruby_client` and `php_client` that looks like History with several topic branches, and we merge `ruby_client` followed by `php_client`, your history will end up looking like After a topic branch merge.

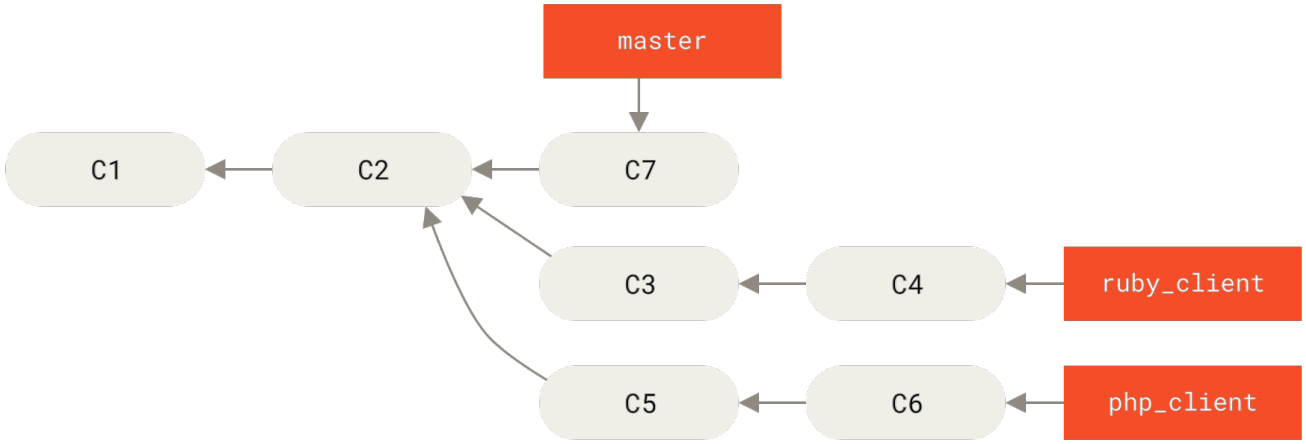


Figura 73. History with several topic branches

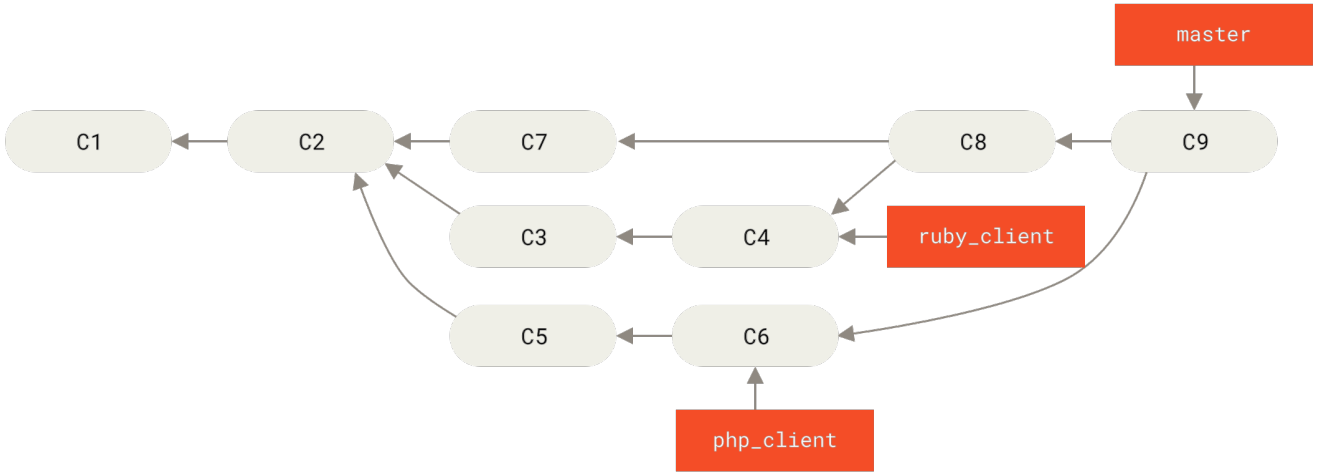


Figura 74. After a topic branch merge

That is probably the simplest workflow, but it can possibly be problematic if you're dealing with larger or more stable projects where you want to be really careful about what you introduce.

If you have a more important project, you might want to use a two-phase merge cycle. In this scenario, you have two long-running branches, `master` and `develop`, in which you determine that `master` is updated only when a very stable release is cut and all new code is integrated into the `develop` branch. You regularly push both of these branches to the public repository. Each time you have a new topic branch to merge in (Before a topic branch merge), you merge it into `develop` (After a topic branch merge); then, when you tag a release, you fast-forward `master` to wherever the now-

stable `develop` branch is (After a project release).

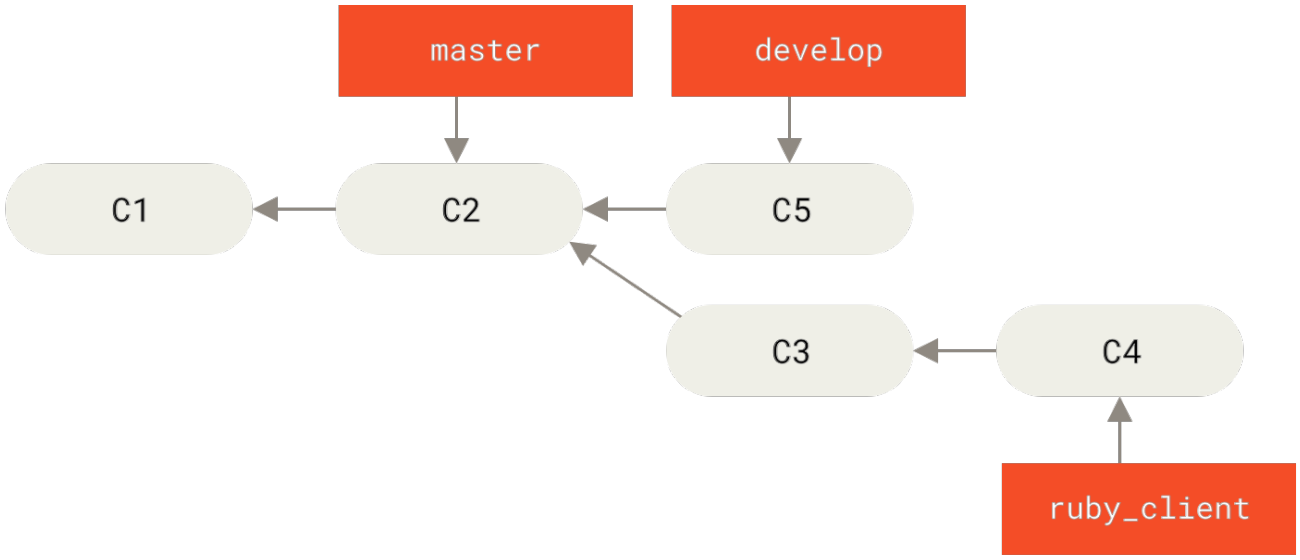


Figura 75. Before a topic branch merge

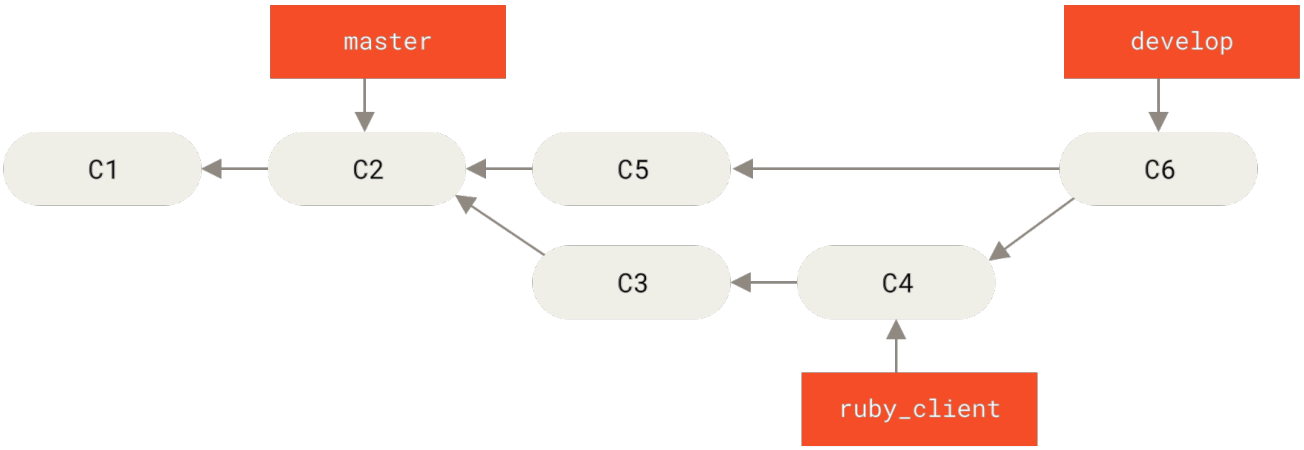


Figura 76. After a topic branch merge

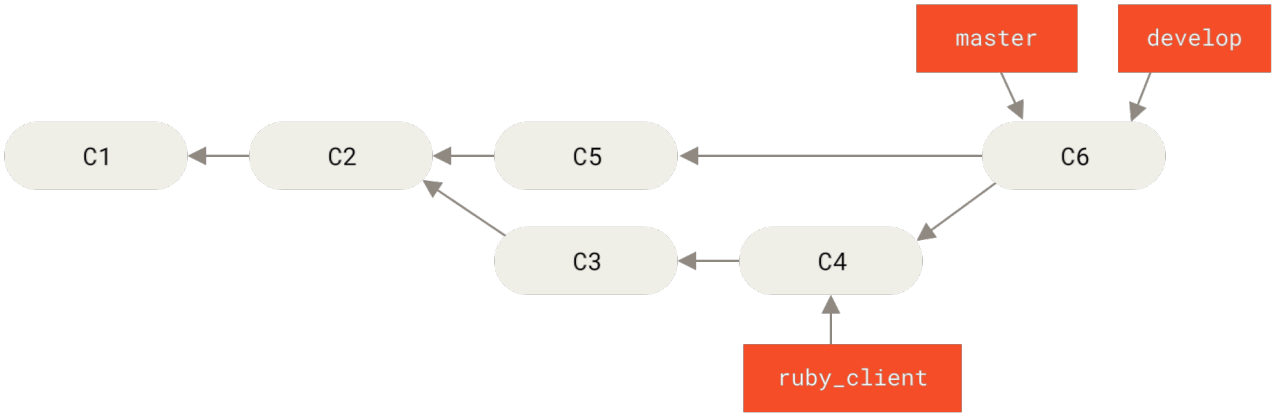


Figura 77. After a project release

This way, when people clone your project's repository, they can either check out `master` to build the latest stable version and keep up to date on that easily, or they can check out `develop`, which is the more cutting-edge content. You can also extend this concept by having an `integrate` branch where

all the work is merged together. Then, when the codebase on that branch is stable and passes tests, you merge it into a `develop` branch; and when that has proven itself stable for a while, you fast-forward your `master` branch.

Large-Merging Workflows

The Git project has four long-running branches: `master`, `next`, and `seen` (formerly *pu*—proposed updates) for new work, and `maint` for maintenance backports. When new work is introduced by contributors, it's collected into topic branches in the maintainer's repository in a manner similar to what we've described (see Managing a complex series of parallel contributed topic branches). At this point, the topics are evaluated to determine whether they're safe and ready for consumption or whether they need more work. If they're safe, they're merged into `next`, and that branch is pushed up so everyone can try the topics integrated together.

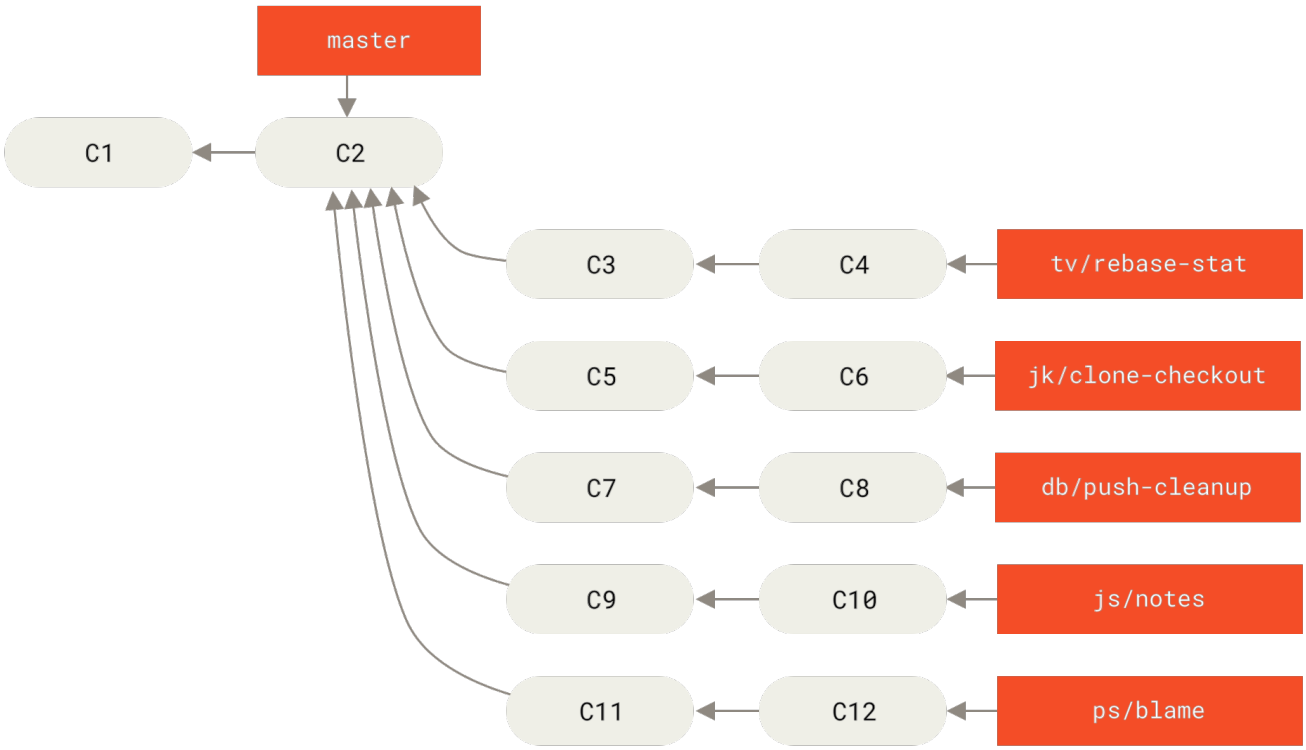


Figura 78. Managing a complex series of parallel contributed topic branches

If the topics still need work, they're merged into `seen` instead. When it's determined that they're totally stable, the topics are re-merged into `master`. The `next` and `seen` branches are then rebuilt from the `master`. This means `master` almost always moves forward, `next` is rebased occasionally, and `seen` is rebased even more often:

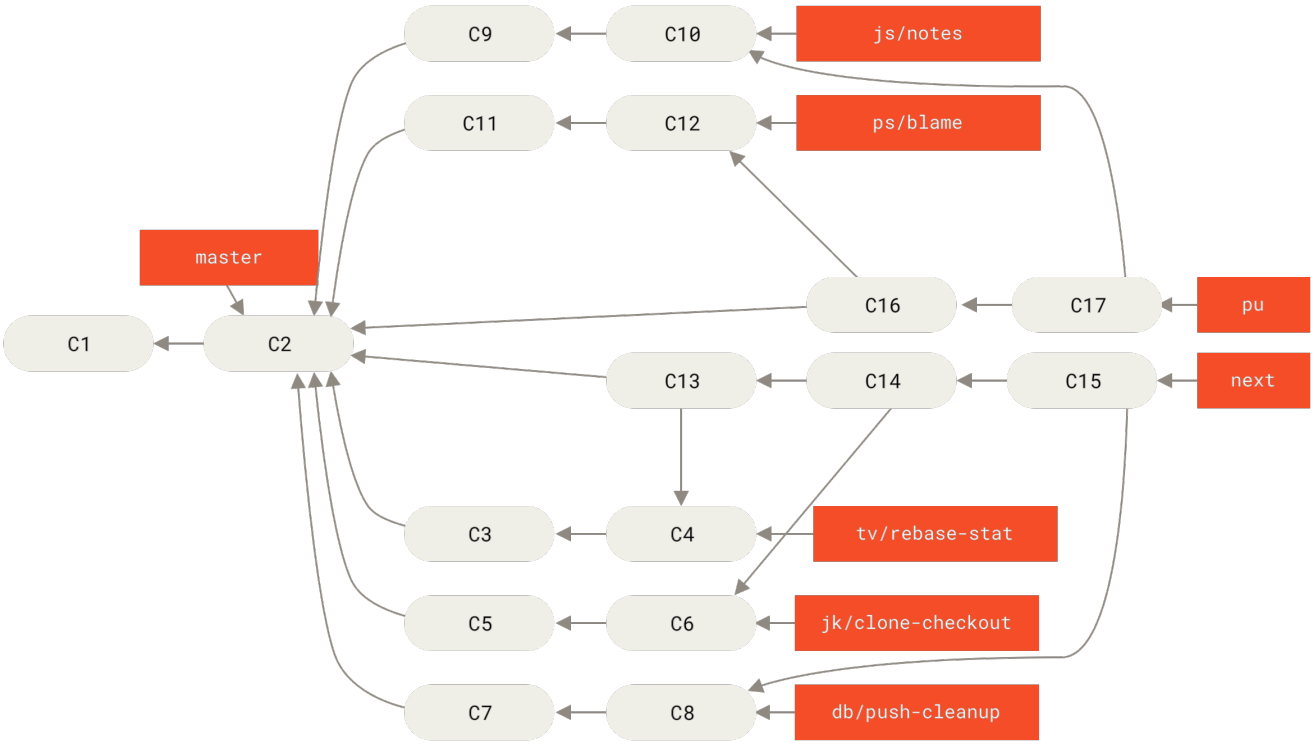


Figura 79. Merging contributed topic branches into long-term integration branches

When a topic branch has finally been merged into `master`, it's removed from the repository. The Git project also has a `maint` branch that is forked off from the last release to provide backported patches in case a maintenance release is required. Thus, when you clone the Git repository, you have four branches that you can check out to evaluate the project in different stages of development, depending on how cutting edge you want to be or how you want to contribute; and the maintainer has a structured workflow to help them vet new contributions. The Git project's workflow is specialized. To clearly understand this you could check out the Git Maintainer's guide.

Rebasing and Cherry-Picking Workflows

Other maintainers prefer to rebase or cherry-pick contributed work on top of their `master` branch, rather than merging it in, to keep a mostly linear history. When you have work in a topic branch and have determined that you want to integrate it, you move to that branch and run the rebase command to rebuild the changes on top of your current `master` (or `develop`, and so on) branch. If that works well, you can fast-forward your `master` branch, and you'll end up with a linear project history.

The other way to move introduced work from one branch to another is to cherry-pick it. A cherry-pick in Git is like a rebase for a single commit. It takes the patch that was introduced in a commit and tries to reapply it on the branch you're currently on. This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you'd prefer to cherry-pick it rather than run rebase. For example, suppose you have a project that looks like this:

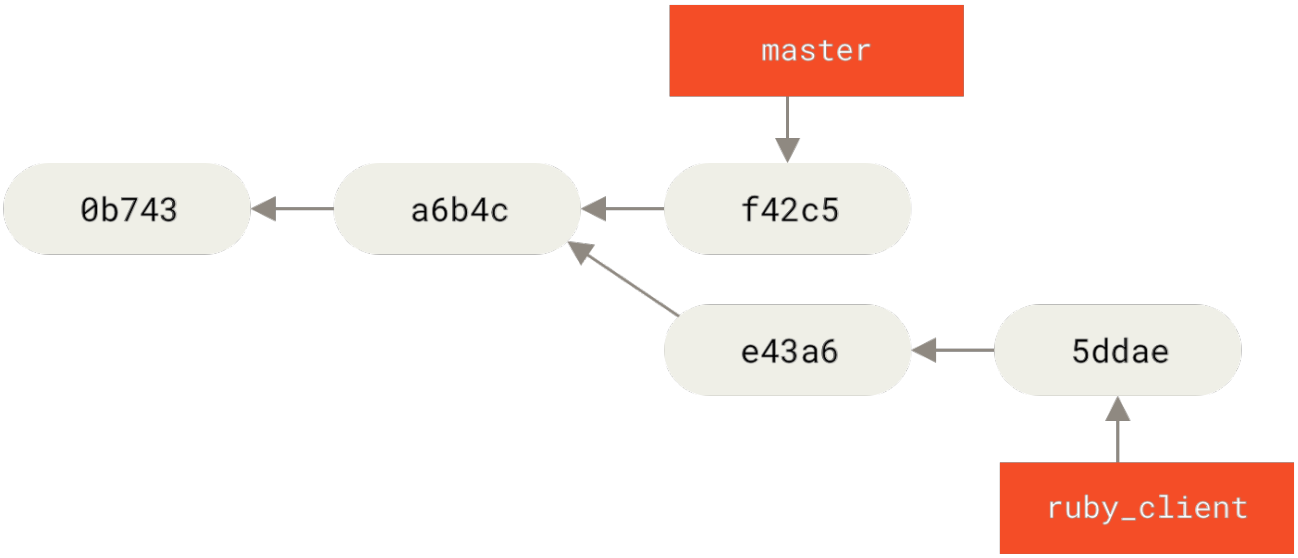


Figura 80. Example history before a cherry-pick

If you want to pull commit `e43a6` into your `master` branch, you can run:

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

This pulls the same change introduced in `e43a6`, but you get a new commit SHA-1 value, because the date applied is different. Now your history looks like this:

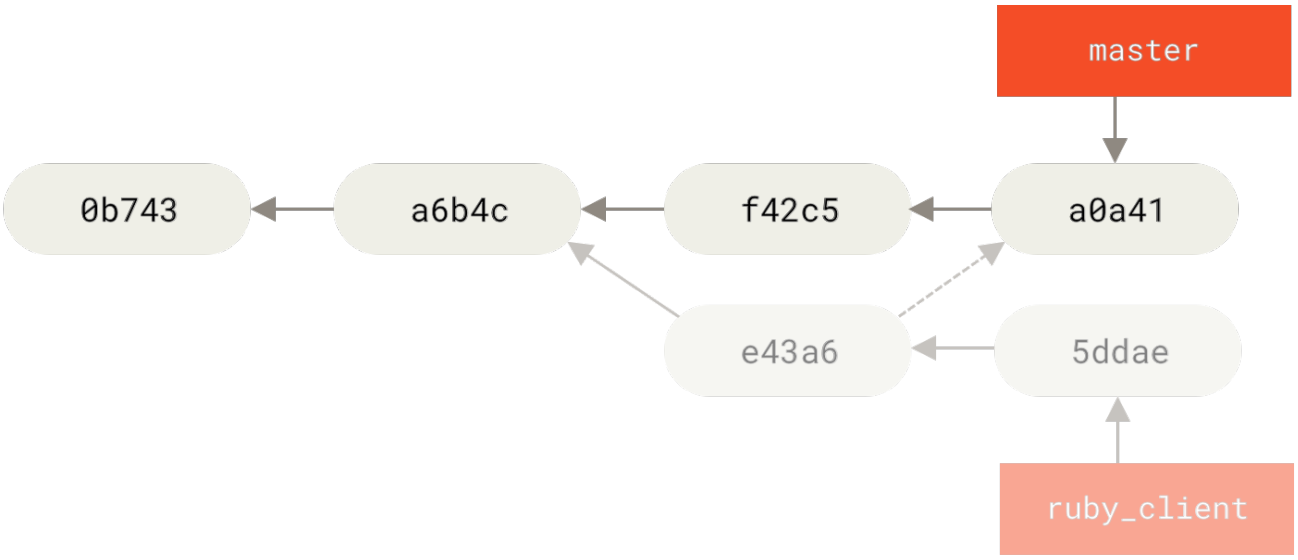


Figura 81. History after cherry-picking a commit on a topic branch

Now you can remove your topic branch and drop the commits you didn't want to pull in.

Rerere

If you're doing lots of merging and rebasing, or you're maintaining a long-lived topic branch, Git has a feature called "rerere" that can help.

Rerere stands for "reuse recorded resolution" — it's a way of shortcutting manual conflict resolution. When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there's a conflict that looks exactly like one you've already fixed, it'll just use the fix from last time, without bothering you with it.

This feature comes in two parts: a configuration setting and a command. The configuration setting is `rerere.enabled`, and it's handy enough to put in your global config:

```
$ git config --global rerere.enabled true
```

Now, whenever you do a merge that resolves conflicts, the resolution will be recorded in the cache in case you need it in the future.

If you need to, you can interact with the rerere cache using the `git rerere` command. When it's invoked alone, Git checks its database of resolutions and tries to find a match with any current merge conflicts and resolve them (although this is done automatically if `rerere.enabled` is set to `true`). There are also subcommands to see what will be recorded, to erase specific resolution from the cache, and to clear the entire cache. We will cover rerere in more detail in Rerere.

Tagging Your Releases

When you've decided to cut a release, you'll probably want to assign a tag so you can re-create that release at any point going forward. You can create a new tag as discussed in Fundamentos de Git. If you decide to sign the tag as the maintainer, the tagging may look something like this:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you do sign your tags, you may have the problem of distributing the public PGP key used to sign your tags. The maintainer of the Git project has solved this issue by including their public key as a blob in the repository and then adding a tag that points directly to that content. To do this, you can figure out which key you want by running `gpg --list-keys`:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Then, you can directly import the key into the Git database by exporting it and piping that through `git hash-object`, which writes a new blob with those contents into Git and gives you back the SHA-1 of the blob:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Now that you have the contents of your key in Git, you can create a tag that points directly to it by specifying the new SHA-1 value that the `hash-object` command gave you:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

If you run `git push --tags`, the `maintainer-pgp-pub` tag will be shared with everyone. If anyone wants to verify a tag, they can directly import your PGP key by pulling the blob directly out of the database and importing it into GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

They can use that key to verify all your signed tags. Also, if you include instructions in the tag message, running `git show <tag>` will let you give the end user more specific instructions about tag verification.

Generating a Build Number

Because Git doesn't have monotonically increasing numbers like *v123* or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run `git describe` on that commit. In response, Git generates a string consisting of the name of the most recent tag earlier than that commit, followed by the number of commits since that tag, followed finally by a partial SHA-1 value of the commit being described (prefixed with the letter "g" meaning Git):

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

This way, you can export a snapshot or build and name it something understandable to people. In fact, if you build Git from source code cloned from the Git repository, `git --version` gives you something that looks like this. If you're describing a commit that you have directly tagged, it gives you simply the tag name.

By default, the `git describe` command requires annotated tags (tags created with the `-a` or `-s` flag); if you want to take advantage of lightweight (non-annotated) tags as well, add the `--tags` option to the command. You can also use this string as the target of a `git checkout` or `git show` command, although it relies on the abbreviated SHA-1 value at the end, so it may not be valid forever. For instance, the Linux kernel recently jumped from 8 to 10 characters to ensure SHA-1 object uniqueness, so older `git describe` output names were invalidated.

Preparing a Release

Now you want to release a build. One of the things you'll want to do is create an archive of the latest snapshot of your code for those poor souls who don't use Git. The command to do this is `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

If someone opens that tarball, they get the latest snapshot of your project under a `project` directory. You can also create a zip archive in much the same way, but by passing the `--format=zip` option to `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

You now have a nice tarball and a zip archive of your project release that you can upload to your website or email to people.

The Shortlog

It's time to email your mailing list of people who want to know what's happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or email is to use the `git shortlog` command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (6):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

You get a clean summary of all the commits since v1.0.1, grouped by author, that you can email to your list.

Summary

You should feel fairly comfortable contributing to a project in Git as well as maintaining your own project or integrating other users' contributions. Congratulations on being an effective Git developer! In the next chapter, you'll learn about how to use the largest and most popular Git hosting service, GitHub.

GitHub

GitHub is the single largest host for Git repositories, and is the central point of collaboration for millions of developers and projects. A large percentage of all Git repositories are hosted on GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things. So while it's not a direct part of the Git open source project, there's a good chance that you'll want or need to interact with GitHub at some point while using Git professionally.

This chapter is about using GitHub effectively. We'll cover signing up for and managing an account, creating and using Git repositories, common workflows to contribute to projects and to accept contributions to yours, GitHub's programmatic interface and lots of little tips to make your life easier in general.

If you are not interested in using GitHub to host your own projects or to collaborate with other projects that are hosted on GitHub, you can safely skip to Git Tools.

AVISO

Interfaces Change

It's important to note that like many active websites, the UI elements in these screenshots are bound to change over time. Hopefully the general idea of what we're trying to accomplish here will still be there, but if you want more up to date versions of these screens, the online versions of this book may have newer screenshots.

Configurando uma conta

A primeira coisa que você precisa fazer é configurar uma conta gratuita de usuário. Simplesmente visite https://github.com, escolha um nome de usuário que esteja disponível, forneça uma conta de email e uma senha, então clique no grande botão verde "Sign up for GitHub"

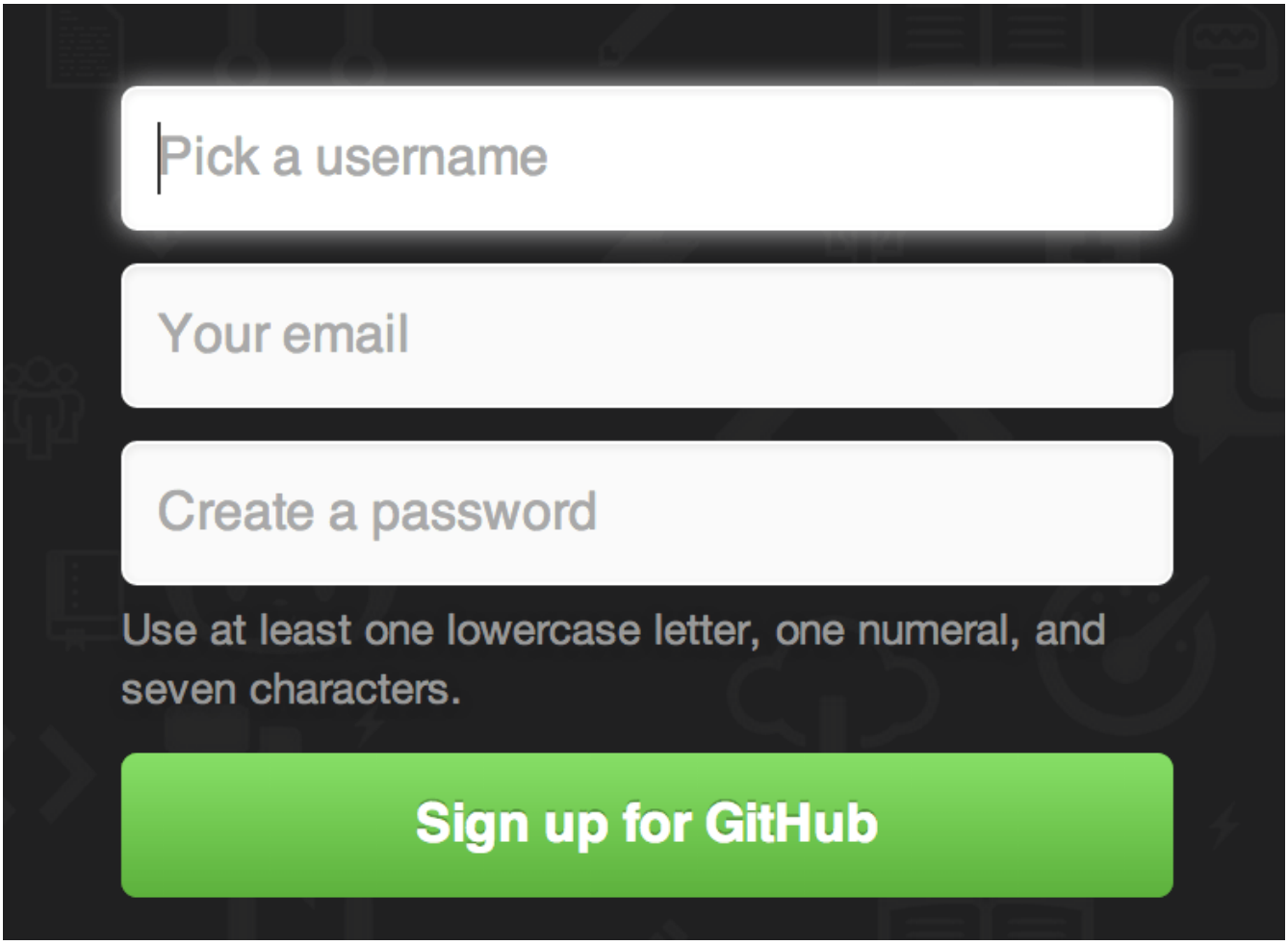


Figura 82. O formulário de Sign-in do GitHub.

A próxima coisa que você verá é a página de preços para planos melhores, mas pode ignorar isso por enquanto. O GitHub vai te enviar um email para verificar o email que você forneceu. Vá em frente e verifique, é muito importante (como veremos mais tarde).

NOTA	O GitHub oferece quase todas as suas funcionalidades com contas gratuitas, exceto algumas funções avançadas. Os planos pagos do GitHub incluem ferramentas e funcionalidades avançadas, bem como limites maiores para serviços gratuitos, mas nós não vamos abordá-los neste livro.

Clicando na logo do Octocat no canto superior esquerdo da tela você vai para a página de dashboard. Você está pronto para usar o GitHub.

Acesso SSH

A partir de agora, você é completamente capaz de se conectar aos repositórios GitHub usando o protocolo `https://`, autenticando com o nome de usuário e senha que você forneceu. Entretanto, para simplesmente clonar repositórios públicos, você não precisa nem mesmo estar logado - a conta que criamos vem à tona quando fazemos fork nos nossos projetos e dermos push em nossos forks mais para frente.

Se você quiser usar repositórios remotos via SSH, você precisa configurar uma chave pública. Se

você ainda não tiver uma, veja Gerando Sua Chave Pública SSH. Abra as configurações da sua conta usando o link no canto superior direito da janela:

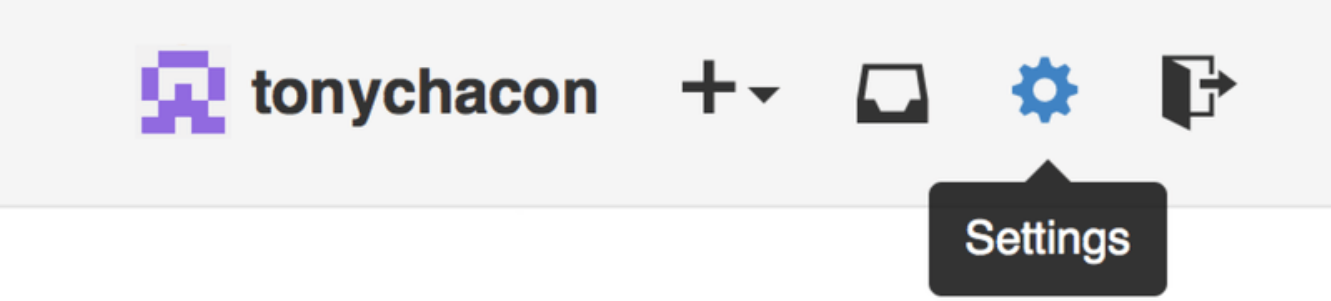


Figura 83. Link para “Account settings”.

Então selecione a seção "´SSH keys´´" no lado esquerdo.

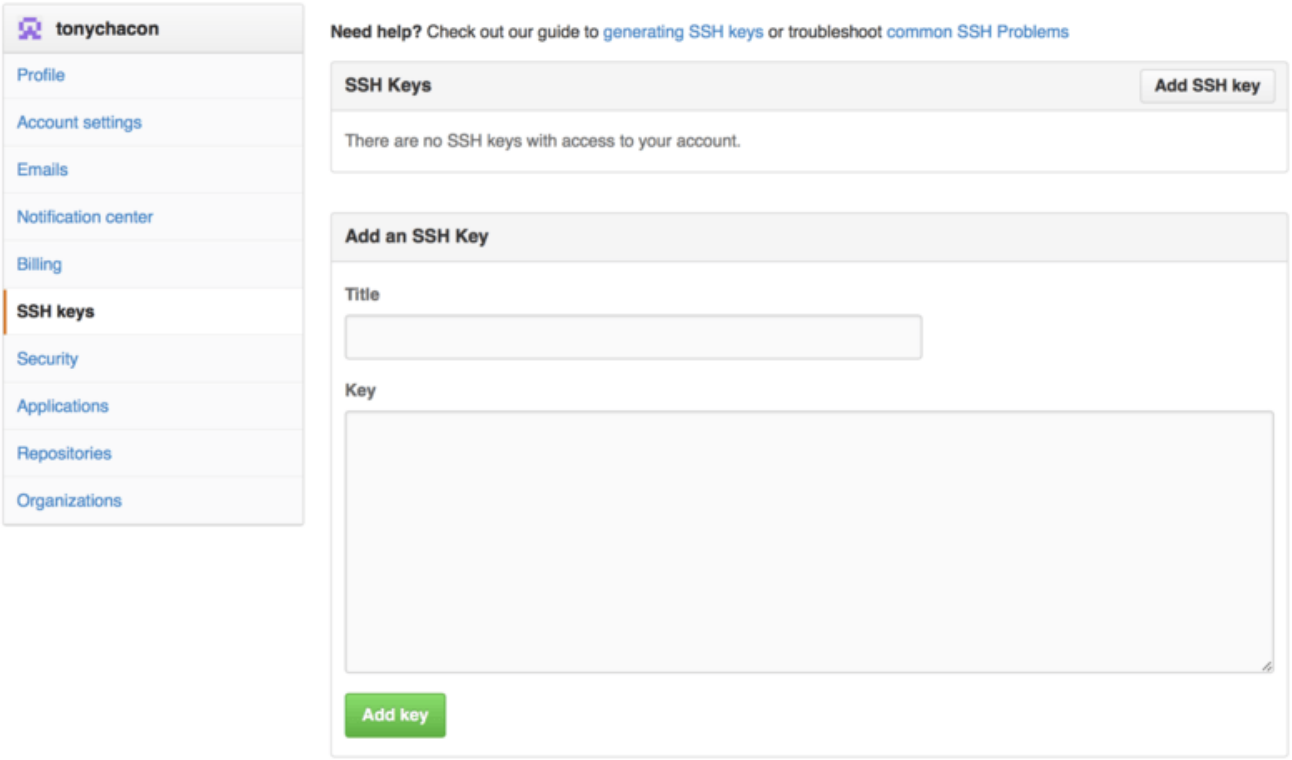


Figura 84. Link para “SSH keys”.

A partir daí, clique no botão "`Add an SSH key`", dê um nome para a chave, copie o conteúdo do seu `~/.ssh/id_rsa.pub` (ou qualquer que seja o nome) arquivo de chave pública no text area e clique “Add key”.

NOTA	Certifique-se de colocar um nome que possa lembrar para sua chave SSH. Você pode nomear cada uma das suas chaves (e.g. "Meu laptop" or "Conta atual") então se você precisar fazer revoke de uma chave você pode facilmente dizer qual você está procurando.

Seu Avatar

A seguir, caso queira, você pode substituir o avatar que é gerado para você com uma imagem de

sua escolha. Primeiro vá para a aba “Profile” (acima da aba SSH Keys) e clique em “Upload new picture”.

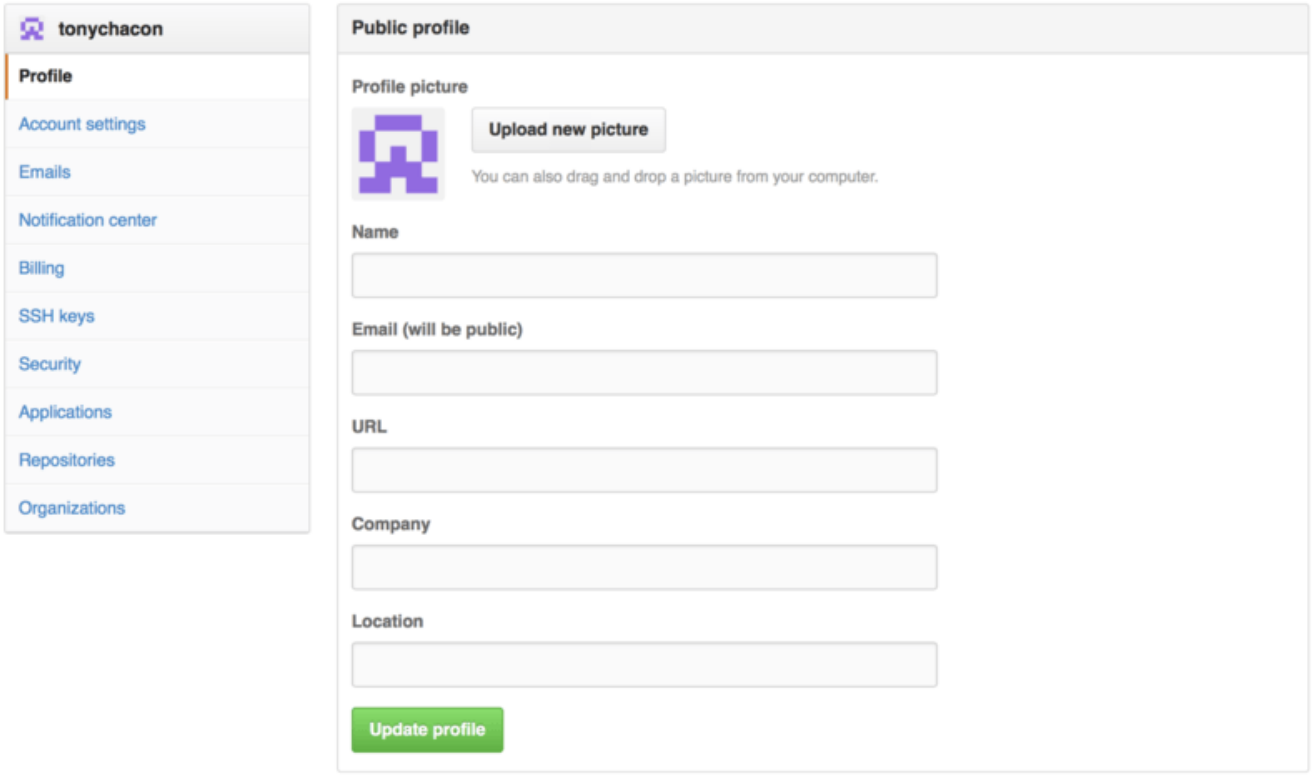


Figura 85. Link para “Profile”.

Nós vamos escolher uma cópia da logo do Git que está no nosso disco rígido e então podemos recortá-lo.

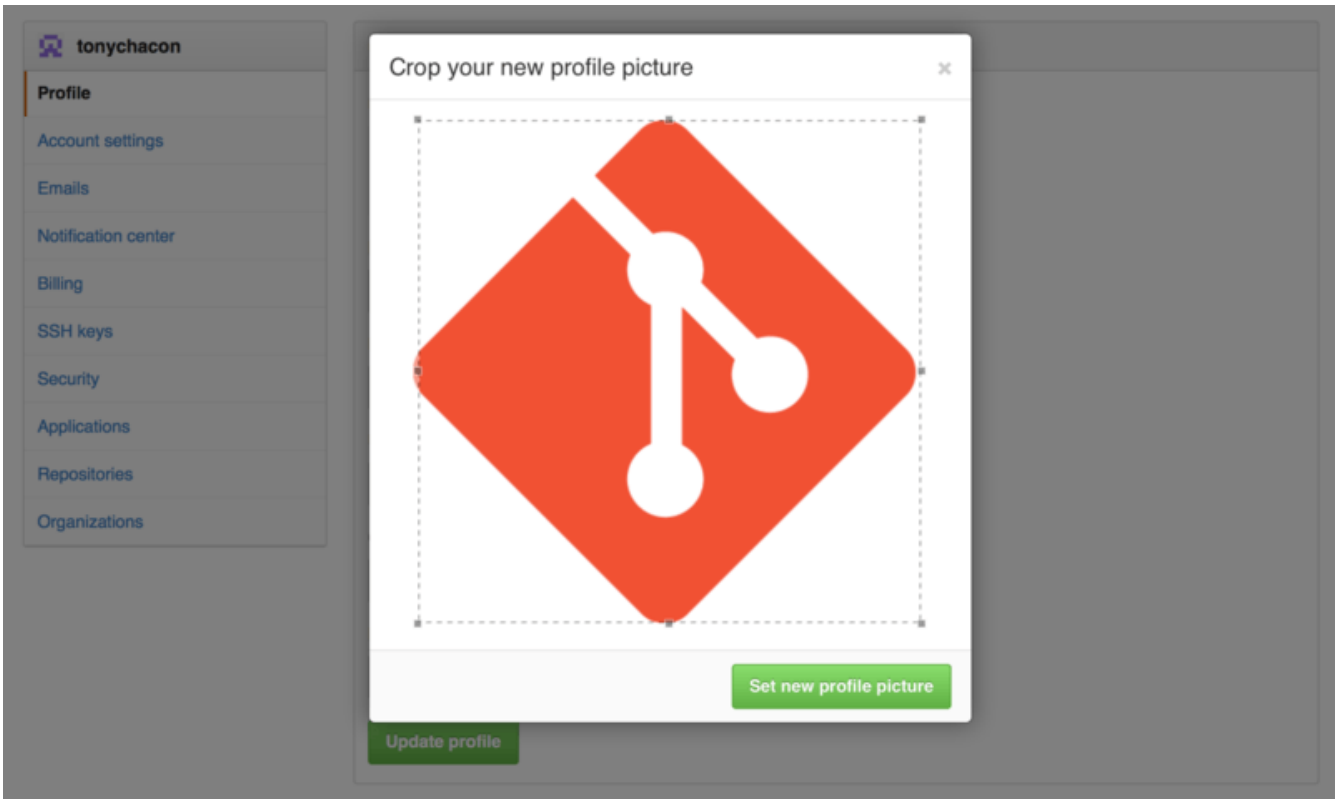


Figura 86. Recorte seu Avatar

Agora sempre que você interagir no site, as pessoas verão seu avatar próximo do seu nome.

Se de repente você tiver carregado um avatar para um serviço Gravatar (geralmente usado para contas Wordpress), este avatar será usado por padrão e você não precisa seguir este passo.

Seus endereços de email

A maneira que o GitHub mapeia seus commits do Git para seu usuário é pelo endereço de email. Se você utiliza múltiplos endereços de email nos seus commits e deseja que o GitHub os conecte adequadamente, você precisa adicionar todos os endereços de email que você usou na seção Email da seção admin.

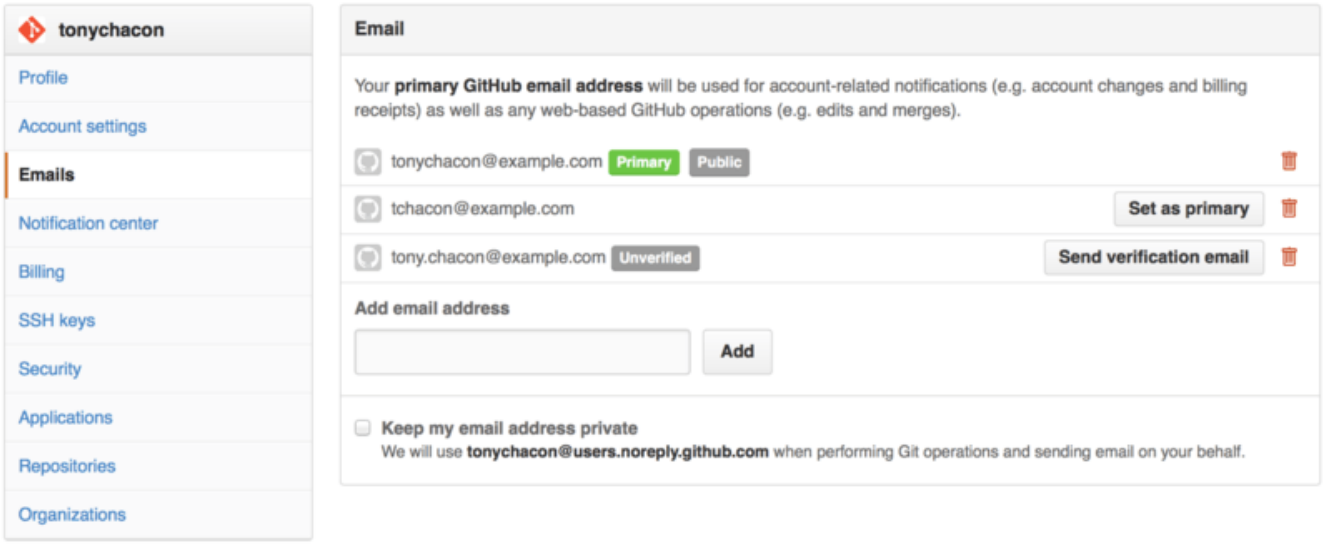


Figura 87. Adicionar endereços de email

Em Adicionar endereços de email podemos ver alguns dos diferentes estados possíveis. O endereço no topo é verificado e definido como endereço padrão, o que significa que é onde você vai receber quaisquer notificações e convites. O segundo endereço é verificado e então pode ser definido como endereço padrão caso você queira trocá-los. O último endereço não é verficado, o que significa que você não pode torná-lo seu endereço padrão. Caso GitHub encontre qualquer um desses endereços em mensagens commit em qualquer repositório no site ele será linkado para seu usuário.

Autenticação de Dois Fatores

Finalmente, para uma segurança extra, você realmente deveria usar uma Autenticação de Dois Fatores ou “2FA”. Autenticação de Dois Fatores é um mecanismo de autenticação que está se tornando cada vez mais popular para reduzir o risco de sua conta ser compromissada caso sua senha seja roubada. Utilizando esse método, o GitHub vai pedir dois métodos diferentes de autenticação, então se um deles for compromissado um invasor não terá acesso total a sua conta.

Você pode encontrar o Two-factor Authentication setup abaixo da Security tab do seu Account settings.

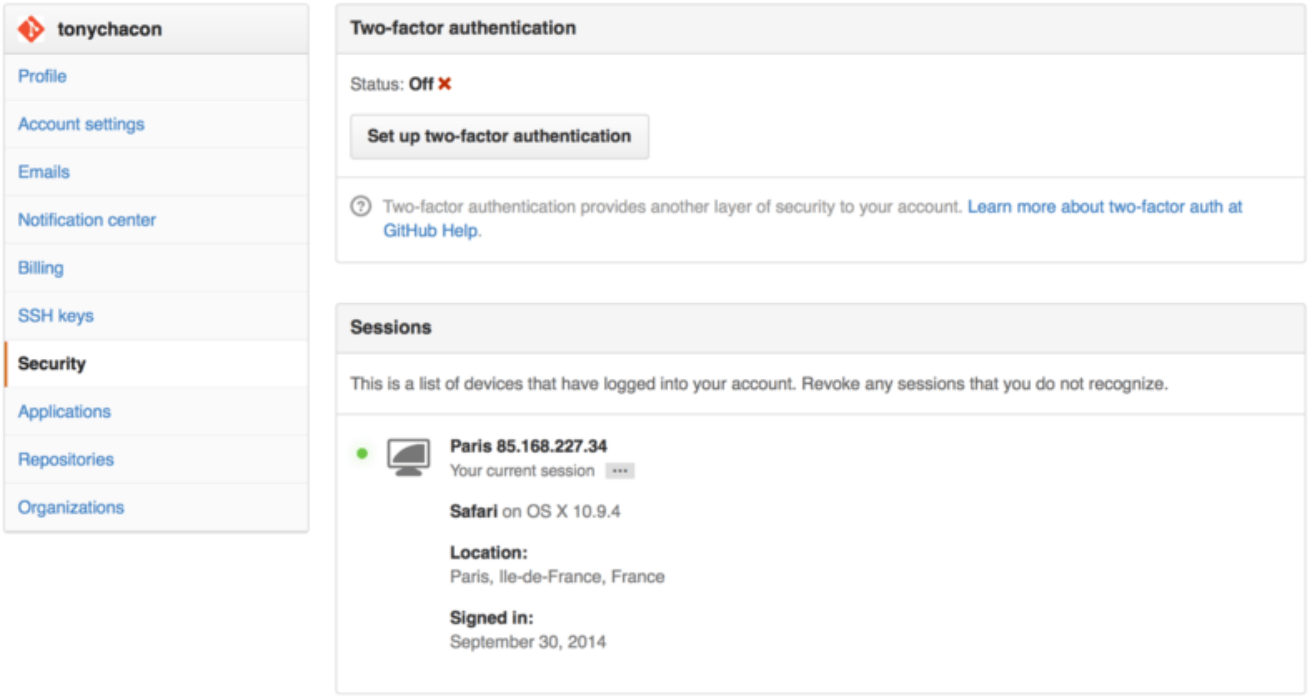


Figura 88. 2FA na Security Tab

Se você clicar no botão "Set up two-factor authentication", será levado para uma página de autenticação onde você pode escolher usar um aplicativo de celular para gerar um código secundário (uma "senha única baseada no tempo") ou mandar o GitHub enviar um código via SMS toda vez que você faz login.

Depois que escolher qual método você prefere e seguir as instruções para configurar um 2FA, sua conta ficará um pouco mais segura e você terá que fornecer um código além de sua senha sempre que você fizer login no GitHub.

Contribuindo em um projeto

Agora que a sua conta está configurada, vamos analisar alguns detalhes que podem te ajudar a contribuir em um projeto existente.

Fazendo Fork de projetos

Se você deseja contribuir em um projeto existente no qual você ainda não tem acesso de push, você pode dar um "fork" no projeto. Quando você faz "fork" de um projeto, o GitHub faz uma cópia do projeto que é inteiramente sua; ele vive no seu namespace e você pode dar push nela.

NOTA

Historicamente, o termo "fork" tinha uma conotação negativa, significando que alguém levou um projeto open source para uma direção diferente, às vezes criando um projeto competidor e dividindo os contribuintes. No GitHub, um "fork" é simplesmente o mesmo projeto no seu namespace, permitindo que você faça alterações publicamente em um projeto como uma forma mais aberta de contribuir.

Dessa forma, projetos não precisam se preocupar em adicionar colaboradores para dar acesso ao push. Qualquer um pode dar fork no projeto, fazer push e mandar suas alterações de volta para o

repositório original criando o chamado Pull Request, o qual falaremos a seguir. Isso abre uma thread de disscusão para revisar o código e para que o proprietário e o contribuinte possam se comunicar sobre as mudanças até que o proprietário esteja feliz com isso, nesse instante o proprietário pode fazer merge do código.

Para fazer fork de um projeto, visite a página do projeto e clique no botão “Fork” na parte superior direita do site.

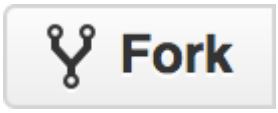


Figura 89. O botão de “Fork”

Depois de alguns segundos, você vai para a página do seu novo projeto, com sua própria cópia alterável do código.

O fluxo do GitHub

O GitHub é desenhado ao redor de uma fluxo particular de colaboção, centrado nas Pull Requests. Este fluxo funciona se você está colaborando com um time integrado em uma único repositório compartilhado, com uma empresa distribuída globalmente ou com uma rede de estranhos contribuindo para o projeto com dúzias de forks. É centrado no fluxo Branches por tópicos abordado em Branches no Git.

Aqui está como geralmente funciona:

1. Faça fork do projeto.
2. Crie um branch para alterações no *master*.
3. Faça alguns commits para aprimorar o projeto.
4. Faça push dessa branch para seu projeto no GitHub
5. Abra um Pull Request no GitHub.
6. Discusta e opicionalmente continue commitando.
7. O proprietário do projeto faz merge ou fecha o Pull Request.

Isto é basicamente o fluxo de trabalho de um Integration Manager abordado em Fluxo de Trabalho Coordenado, mas em vez de usar o email para se comunicarem e revisarem mudanças, os times usam ferramentas do GitHub basedas na web.

Vamos observar um exemplo de como propor uma alteração para um projeto open source hospedado no GitHub usando esse fluxo.

Criando um Pull Request

Tony está procurando algum código para rodar no seu microcontrolador programável em Arduino e ele achou um ótimo arquivo de programa no GitHub em https://github.com/schacon/blink.

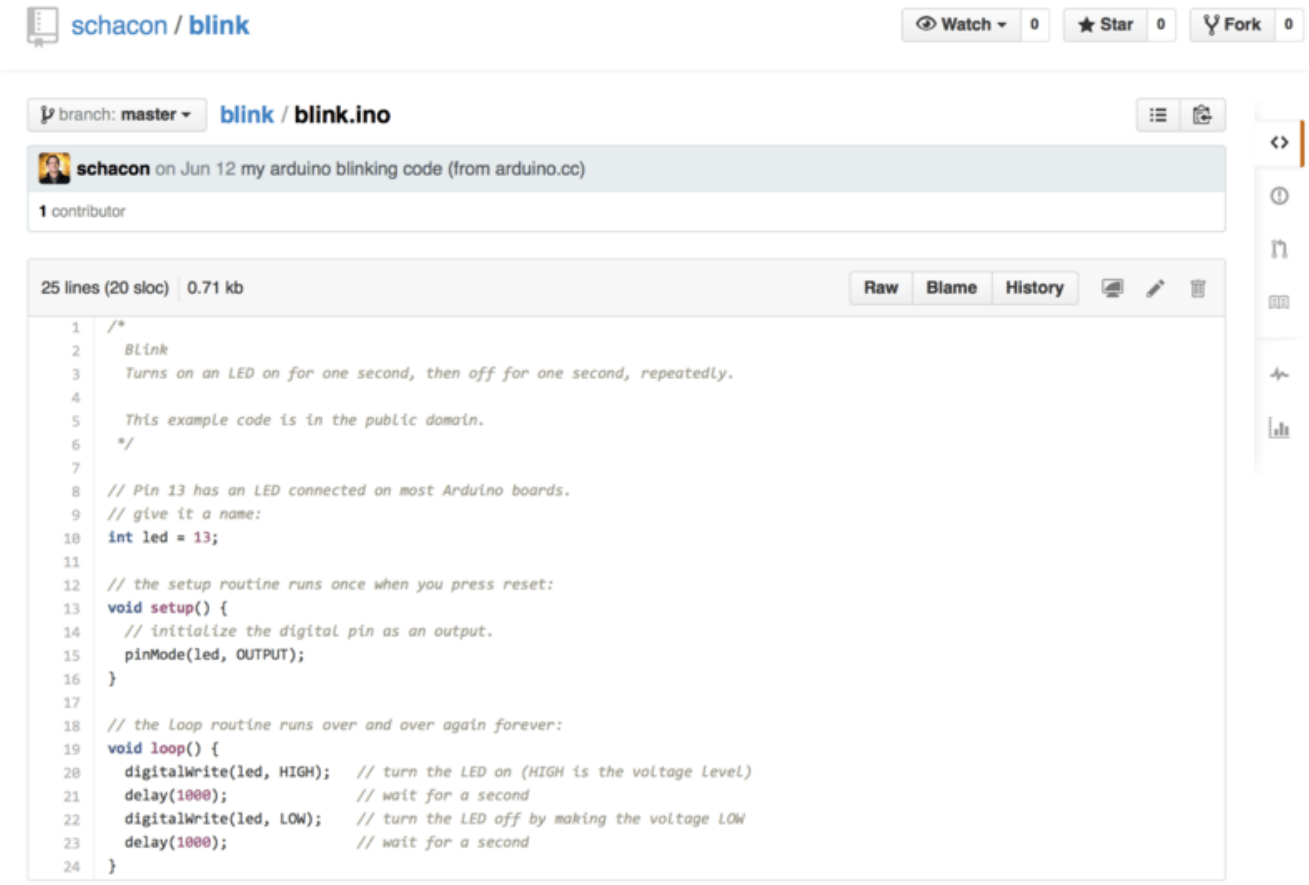


Figura 90. O projeto que queremos contribuir

O único problema é que o intervalo para o LED piscar é muito curto, nós achamos que seria muito melhor aguardar 3 segundos ao invés de apenas 1 entre cada mudança de estado. Então vamos aprimorar o programa e retornar para o projeto como uma proposta de mudança.

Primeiro, clicamos no botão de *Fork* como mencionado mais cedo para pegar nossa própria cópia do projeto. Nosso nome de usuário é “tonychacon” então nossa cópia desse projeto está em `https://github.com/tonychacon/blink` e é aí onde podemos editá-lo. Vamos cloná-lo localmente, criar um tópico de branch, fazer alterações no código e finalmente dar push nas mudanças para o GitHub.

```
$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
}

$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

① Clone seu fork do projeto localmente

② Crie um tópico de branch descritivo

③ Faça suas alterações no código

④ Confira se suas alterações são boas

⑤ Faça commit de suas alterações no tópico de branch

⑥ Faça push do seu novo tópico de branch de volta para seu fork no GitHub.

Agora se você voltar para seu fork no GitHub, podemos ver que o GitHub percebeu que fizemos push de um novo tópico de branch e nos presenteia com um grande botão verde para conferir nossas alterações e abrir um Pull Request para o projeto original.

Você pode alternativamente ir para página de “Branches” em `https://github.com/<user>/<project>/branches` para localizar seu branch e abrir um Pull Request de lá.

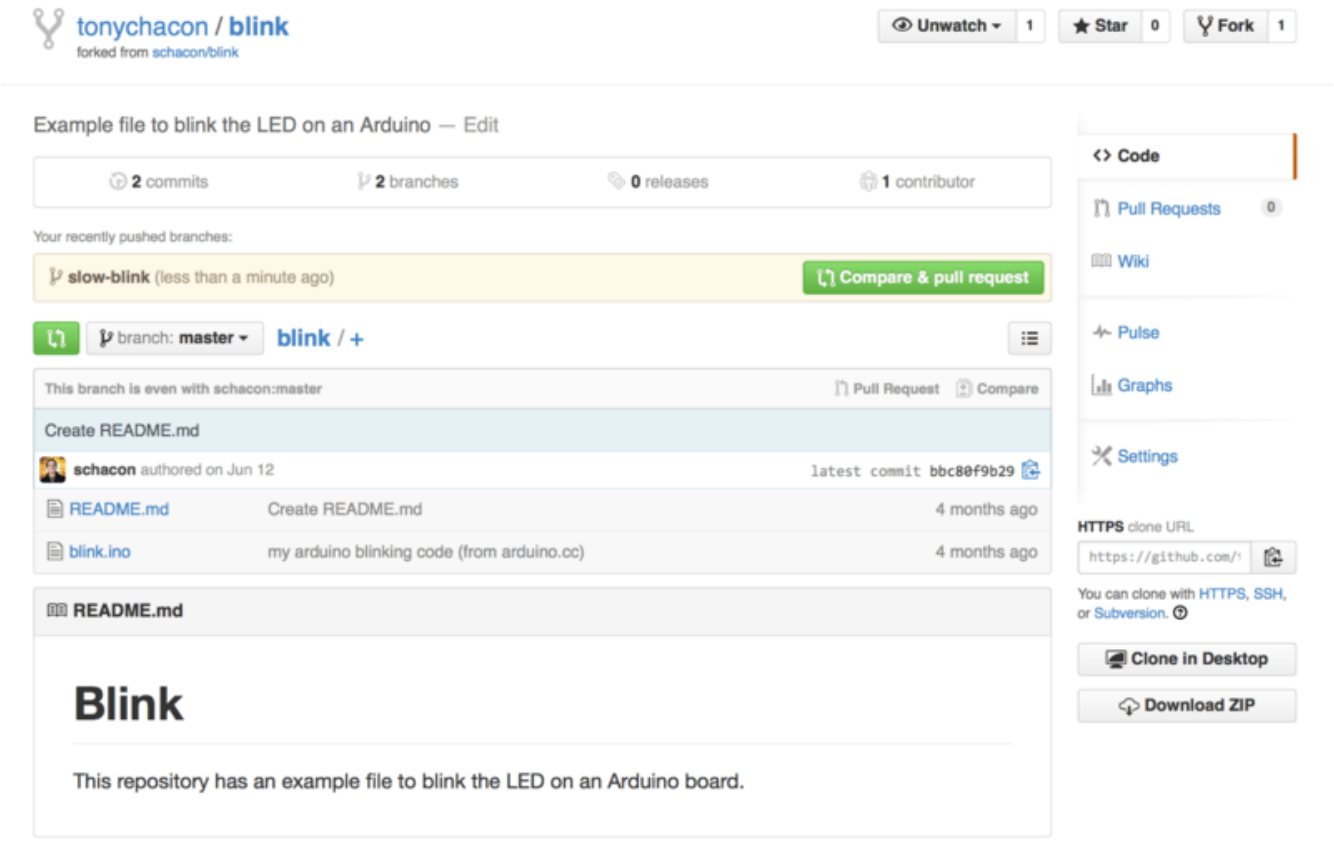


Figura 91. Botão de Pull Request

Se clicarmos no botão verde, veremos uma tela que nos pede para dar um título e uma descrição para nosso Pull Request. Quase sempre vale a pena colocar um pouco de esforço nisso, já que uma boa descriçaõo ajuda o proprietário do projeto original a determinar o que você está tentando fazer, se suas alterações propostas estão corretas e se aceitar suas alterações vai melhorar o projeto original.

Também vemos uma lista de commits no nosso tópico de branch que estão “ahead” no branch `master` (nessa caso apenas um) e uma diff unificada de todas as alterações que deveriam ser feitas nesse branch para que o proprietário do projeto faça merge.

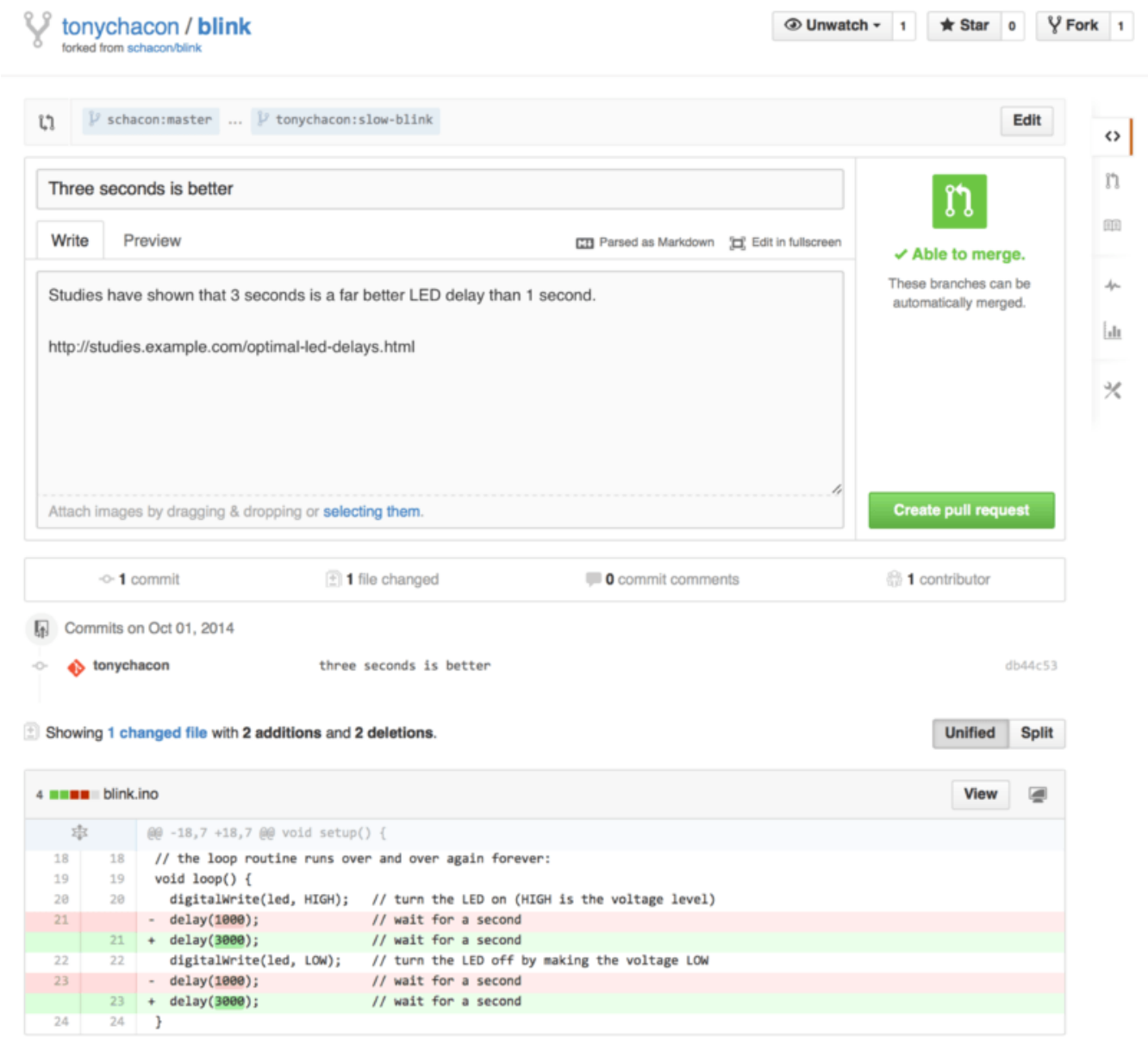


Figura 92. Página de criação de Pull Request

Quando você aperta no botão *Create pull request* na tela, o proprietário do projeto que você fez fork vai receber uma notificação de que alguém está sugerindo uma alteração e um link para a página onde está toda a informação sobre isso.

NOTA

Embora os Pull Requests sejam normalmente usados para projetos públicos nos quais o contribuinte tem uma alteração pronta para ser feita, também é comum em projetos internos *no começo* do ciclo de desenvolvimento. Uma vez que você continua fazendo push no tópico de branch mesmo **depois** que o Pull Request está aberto, normalmente é aberto cedo e usado como uma forma de interagir no trabalho como time com um contexto, em vez de aberto perto do fim do processo.

Interagindo em um Pull Request

Nesse ponto, o proprietário do projeto pode olhar para a alteração sugerida e fazer merge dela, rejeitá-la ou comentá-la. Digamos que ele goste da ideia, mas prefira um pouco mais de tempo para pensar no assunto.

Essa conversa pode ocorrer por meio de email com os workflows apresentados em Distributed Git, ou de forma online com o GitHub. O proprietário do projeto pode revisar o diff unificado e deixar um comentário clicando em qualquer linha.

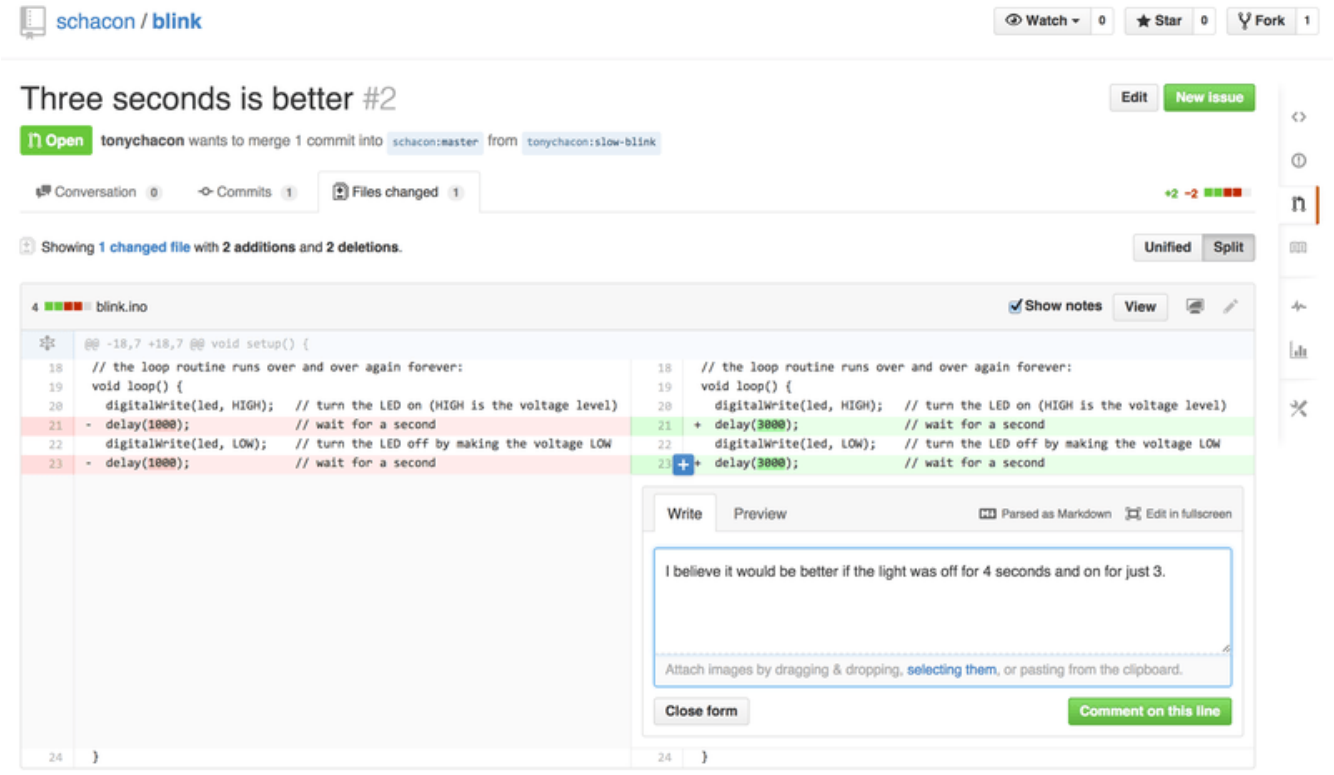


Figura 93. Comente em uma linha específica de código no Pull Request

Uma vez que o mantenedor faça este comentário, a pessoa que abriu o Pull Request (na verdade qualquer um que estiver observando o repositório) receberá uma notificação. Vamos customizar isso depois, mas se ele tivesse ativado suas configurações de email, Tony receberia um email como este:

Figura 94. Comentários enviados como notificações pelo email

Qualquer um pode deixar um comentário geral no Pull Request. Em Página de discussão de um Pull Request podemos ver um exemplo no qual o proprietário do projeto comenta uma linha de código e deixa um comentário geral na seção de discussão. Você pode ver que os comentários aparecem na discussão também.

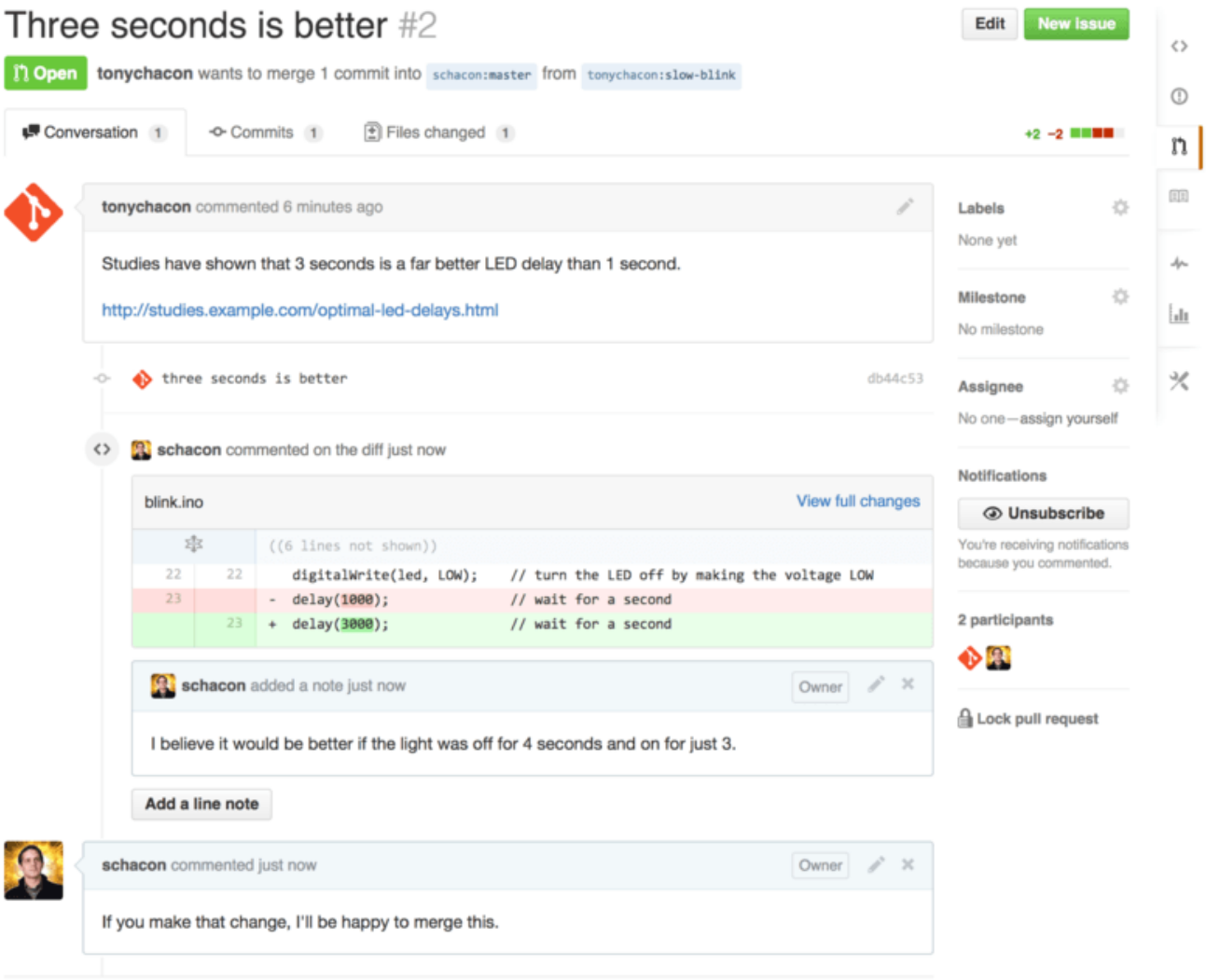


Figura 95. Página de discussão de um Pull Request

Agora o contribuinte pode ver o que ele precisa fazer para ter sua alteração aceita. Felizmente isso é muito simples. Além do email, onde você tem que fazer re-roll das suas series e reenviar para a lista de email, com o GitHub você simplesmente commita no tópico de branch de novo e faz um push que automaticamente atualiza o Pull Request. Em Pull Request final você também pode ver que o comentário do código antigo foi colapsado no Pull Request atualizado, desde que seja de uma linha que foi alterada.

Adicionar commits para um Pull Request não ativa uma notificação, então uma vez que Tony fez push de suas correções ele decide deixar um comentário para informar ao proprietário do projeto que ele fez as alterações requeridas.

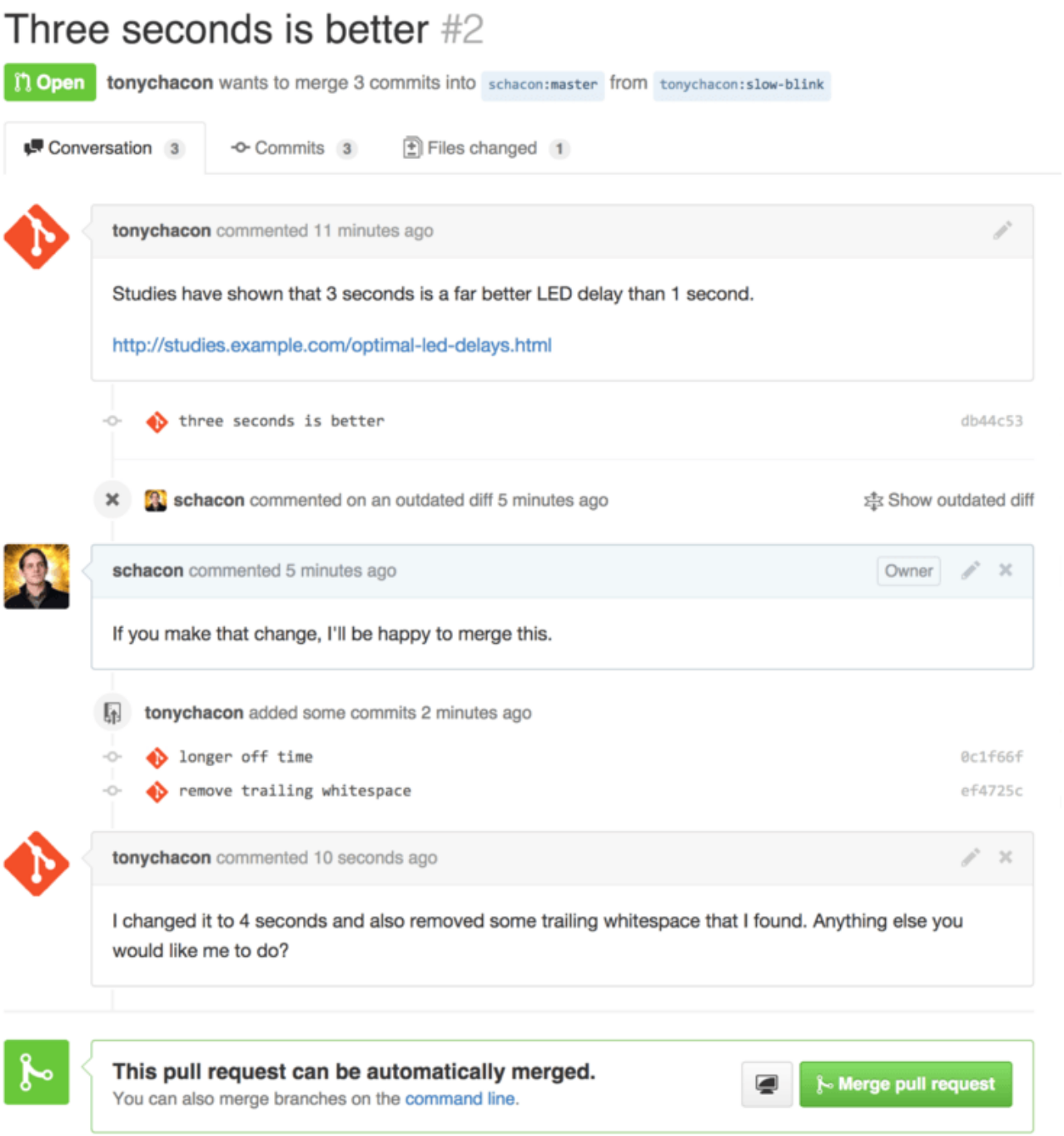


Figura 96. Pull Request final

Uma coisa interessante para se notar é que se você clicar na aba “Files Changed” nesse Pull Request, você vai ter o diff “unificado” — isto é, o agregado total das diferenças que seriam introduzidas no seu branch principal se este tópico de branch levasse merge. O `git diff` basicamente mostra automaticamente `git master...<branch>` para o branch no qual este Pull Request se baseia. Veja Determining What Is Introduced para mais informação sobre este tipo de diff.

A outra coisa de você vai notar é que o GitHub confere se o Pull Request fez um merge válido e te fornece um botão para fazer merge no servidor. Esse botão apenas mostra se você tem acesso de escrita no repositório e um merge trivial se possível. Se você clicar nele, o GitHub fornecerá um merge “no-fast-forward”, significando que mesmo se **pudesse** ser um fast-forward, será criado um commit de merge.

Se preferir, você pode simplesmente fazer pull da branch e dar merge localmente. Se você fizer

merge deste branch na branch `master` e dar push para o GitHub, o Pull Request será automaticamente fechado.

Este é o workflow que a maioria dos projetos no GitHub usa. Tópicos de branch são criados, Pull Requests são abertos neles, uma discussão começa, possivelmente mais trabalho é feito na branch e eventualmente a requisição é fechada ou sofre merge.

NOTA

Não só Forks

É importante notar que você também pode abrir um Pull Request entre duas branches no mesmo repositório. Se você está trabalhando em um recurso com alguém e vocês têm acesso de escrita no projeto, você pode fazer push de um tópico de branch para o repositório e abrir um Pull Request da branch `master` do mesmo projeto para iniciar um processo de discussão e revisão do código. Sem a necessidade de forks.

Pull Requests Avançados

Agora que cobrimos o básico sobre contribuição de um projeto no GitHub, vamos abordar algumas dicas e truques interessantes sobre Pull Requests e então você pode ser mais eficaz usando eles.

Pull Requests como Patches

É importante entender que muitos projetos não pensam realmente em Pull Requests como filas de patches perfeitos que deveriam claramente ser aplicados em ordem, assim como a maioria dos projetos baseados em listas de emails pensa em séries de patches de contribuição. A maior parte dos projetos no GitHub encara branches de Pull Requests como conversações interativas sobre uma alteração proposta, culminando em um diff unificado que é aplicado por meio de merging.

Esta é um distinção importante, porque normalmente a alteração é sugerida antes do código atingir a perfeição, o que é muito mais raro em séries de patches de contribuições baseadas em listas de emails. Isso possibilita uma conversação mais cedo com os mantenedores para chegar a uma solução adequada por meio da dedicação da comunidade. Quando código é proposto com um Pull Request e os mantenedores ou a comunidade sugerem uma alteração, a série de patches normalmente não é recarregado, mas sim a diferença sofre push como um novo commit na branch, movendo a conversação para frente com o contexto do trabalho anterior preservado.

Por exemplo, se você voltar e olhar de novo em Pull Request final, você vai notar que o contribuinte não deu rebase no seu commit e enviou outro Pull Request. Em vez disso ele adicionou novos commits e fez push deles para uma branch existente. Desta forma se você voltar e olhar esse Pull Request no futuro, você poderá facilmente encontrar todo o contexto do motivo de cada decisão. Pressionar o botão “Merge” no site intencionalmente cria commit de merge que referencia o Pull Request e torna mais fácil voltar e pesquisar a conversação original caso necessário.

Mantendo-se no Upstream

Se o seu Pull Request ficar desatualizado ou de qualquer modo não for um merge válido, você vai querer consertá-lo para que o mantenedor possa facilmente fazer merge dele. O GitHub vai testar isso para você e informar na parte inferior de cada Pull Request se o merge é trivial ou não.

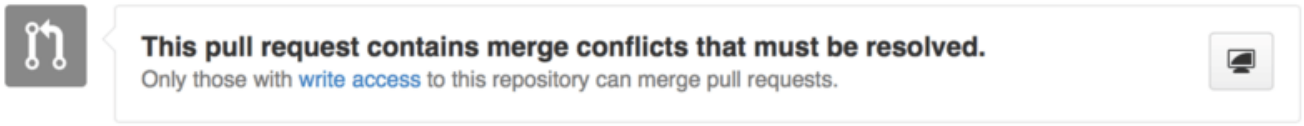


Figura 97. Pull Request que não é um merge válido

Se você ver algo como Pull Request que não é um merge válido, você vai querer consertar seu branch para que ele fique verde e o para que mantenedor não tenha trabalho extra.

Você tem duas opções principais para lidar com isso. Você pode fazer rebase da sua branch no topo ou em qualquer branch alvo (que normalmente é a branch `master` do repositório que você deu fork), ou você pode fazer merge da branch alvo na sua branch.

A maioria dos desenvolvedores no GitHub vai escolher a última opção, pelas mesmas razões que passamos por cima na seção anterior. O que importa é o histórico e o merge final, então fazer rebase não será nada mais do que olhar um histórico limpo e passar **longe** de mais dificuldade e de propensão a erros.

Se você quiser dar merge em uma branch específica, você deve adicionar o repositório original como um novo remoto, fazer fetch, e dar merge da branch principal deste repositório para seu tópico de branch, consertando quaisquer erros e finalmente fazendo push para a mesma branch na qual você abriu o Pull Request.

Por exemplo, digamos que no exemplo do “tonychacon” que nós usamos antes, o autor original fez uma alteração que criaria um conflito no Pull Request. Vamos dar uma olhada nesses passos.

```
$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master     -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
   ef4725c..3c8d735  slower-blink -> slow-blink
```

① Adicione um repositório remoto original nomeado como “upstream”

② Faça fetch do trabalho mais recente para este repositório

③ Dê merge da branch principal para seu tópico de branch

④ Conserte o conflito que ocorreu

⑤ Faça push de volta para o mesmo tópico de branch

Uma vez que você fez isso, o Pull Request será automaticamnete atualizado e re-avaliado para ver se é um merge válido.

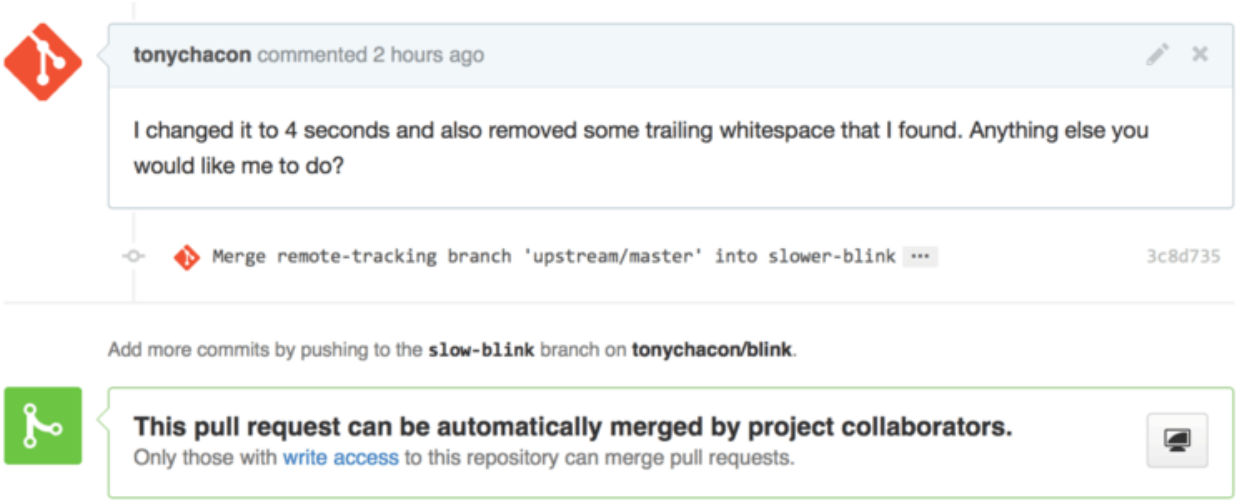


Figura 98. O Pull Request agora é válido

Um dos grandes feitos do Git é que você pode fazer isso continuamente. Se você tem um projeto muito grande, você pode facilmente fazer merge de uma branch específica de novo e de novo se preocupando apenas com conflitos que sugiram desde da última vez que você deu merge, tornando o projeto muito mais gerenciável.

Se você realmente quer fazer rebase da branch para limpá-la, você até pode fazê-lo, mas uma atitude muito mais encorajada é não forçar uma branch em um Pull Request que já está aberto. Se outra pessoa deu push e trabalhou mais no projeto, você pode conferir todos os erros destacados em Os perigos do Rebase. Em vez disso, faça push da branch rebaseada para uma nova branch no GitHub e abra um novo Pull Request referenciando o antigo, e então feche o original.

Referências

Sua próxima pergunta deve ser “Como eu referencio um Pull Request antigo?”. Na verdade existem muitas, muitas formas de referenciar quase tudo que você pode escrever no GitHub.

Vamos começar com como referenciar outro Pull Request ou Issue. Todos os Pull Requests e Issues são números atribuídos e eles são únicos dentro do projeto. Por exemplo, você não pode ter um Pull Request 3 *e* uma Issue #3. Se você quiser referenciar qualquer Pull Request ou Issue de outro repositório, você pode simplesmente #<num> em qualquer comentário ou descrição. Você também pode ser mais específico quanto se a Issue ou o Pull Request está ativo em outro lugar; escreva username<num> se você está referenciando uma Issue ou um Pull Request em algum fork do repositório em que você está, ou usernam/repo#<num> para referenciar algo em outro repositório.

Vamos olhar um exemplo. Digamos que rebaseamos a branch no exemplo anterior, criamos um novo pull request para ele e agora queremos referenciar um antigo pull request para o novo. Nós também queremos referenciar um problema no fork do repositório e um problema em um projeto completamente diferente. Podemos preencher a descrição como em Referências cruzadas em um Pull Request..

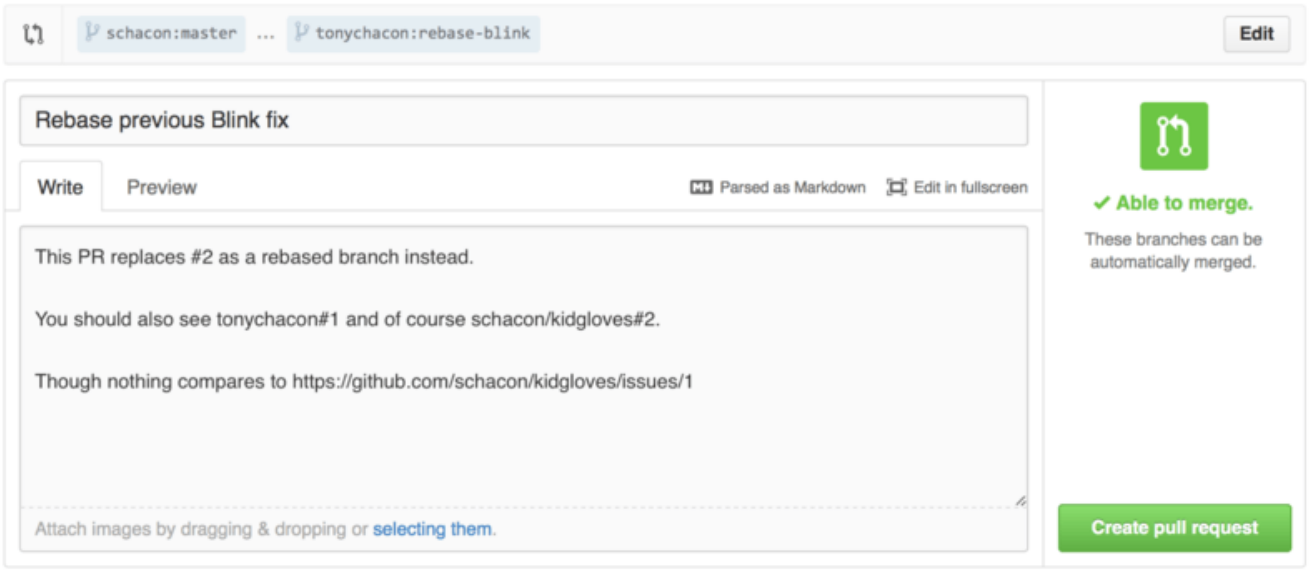


Figura 99. Referências cruzadas em um Pull Request.

Quando fazemos submit deste pull request, veremos tudo renderizado como em Referências cruzadas renderizadas em um Pull Request..

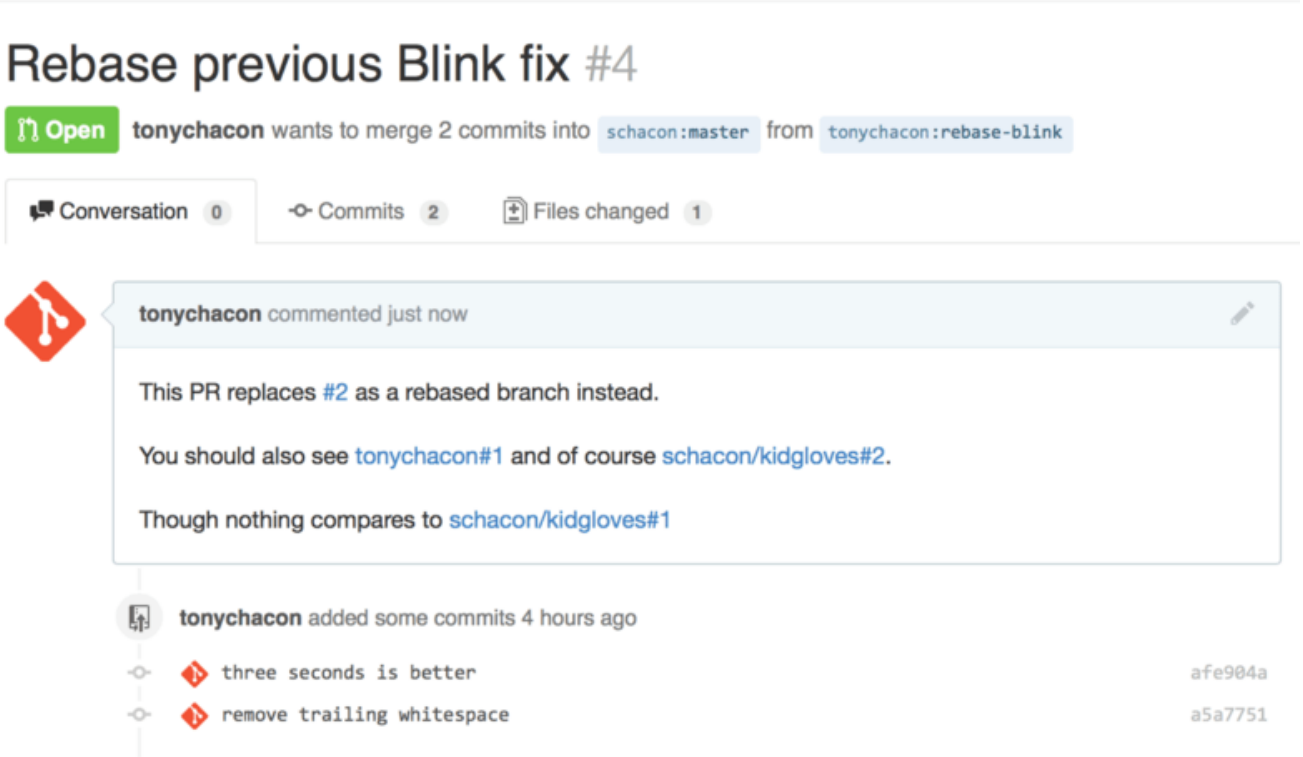


Figura 100. Referências cruzadas renderizadas em um Pull Request.

Note que a URL completa que colocamos no GitHub foi encurtada para a informação necessária.

Agora se Tony voltar e fechar o Pull Request original, poderemos ver isso sendo mencionado no novo, o GitHub automaticamente cria um evento de trackback na timeline do Pull Request. Isso significa que qualquer um que visite este Pull Request e veja que está fechado pode facilmente seguir o link para o que o sucedeu. O link parecerá algo parecido com Referências cruzadas renderizadas em um Pull Request..

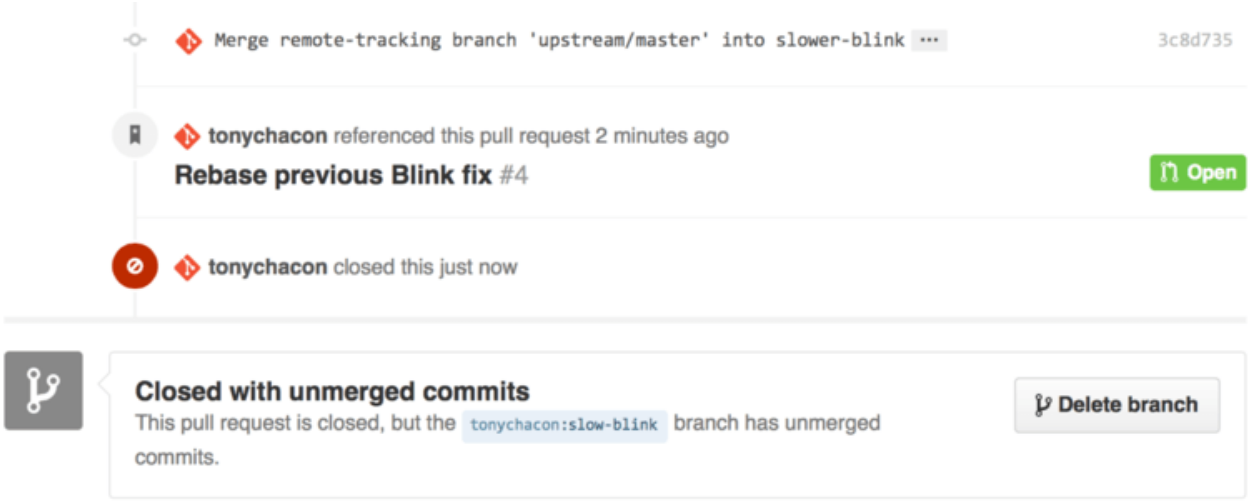


Figura 101. Referências cruzadas renderizadas em um Pull Request.

Além de números de issues, você também pode referenciar um commit específico pelo SHA-1. Você precisa especificar uma SHA-1 de 40 caracteres, mas se o GitHub vê-lo em um comentário, ele vai linkar diretamente para o commit. De novo, você pode referenciar commits em outros forks ou repositórios da mesma forma que você fez com issues.

Markdown aprimorado do GitHub

Fazer link com outras Issues é só o começo das coisas interessantes que você pode fazer com quase qualquer caixa de texto no GitHub. Em uma Issue e em descrições de Pull Request, comentários, comentários de código e outros, você pode usar o chamado “Markdown aprimorado do GitHub”. Markdown é similar a um texto comum mas renderizado de forma rica.

Veja Um exemplo escrito e renderizado do Markdown melhorado do GitHub. para um exemplo de como comentários ou textos podem ser escritos e renderizados usando Markdown.

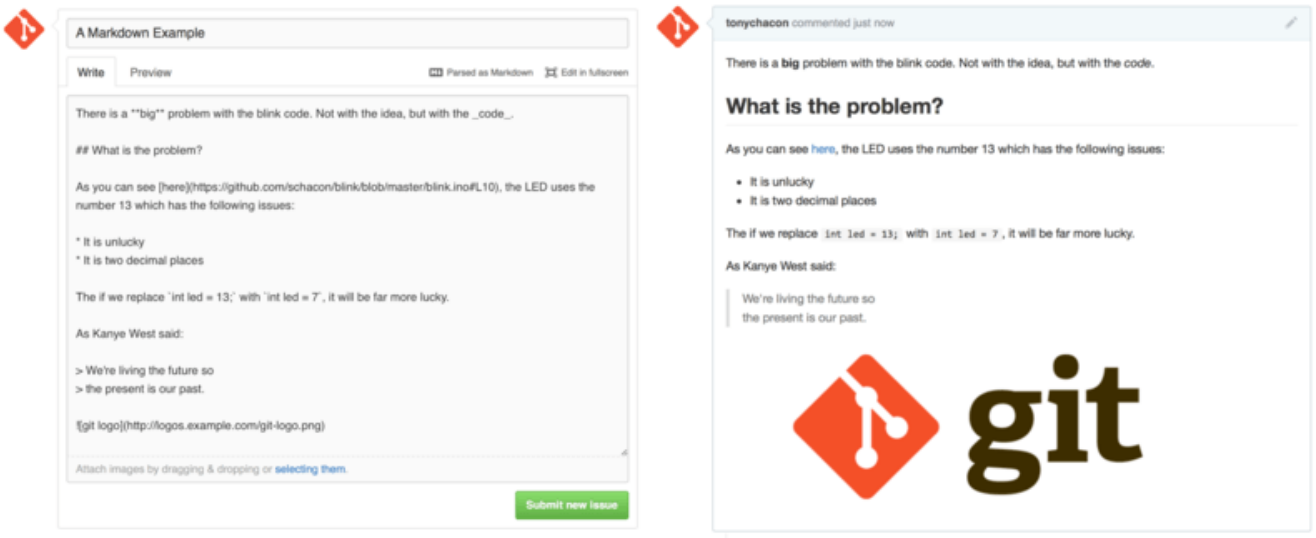


Figura 102. Um exemplo escrito e renderizado do Markdown melhorado do GitHub.

O Markdown do GitHub adiciona mais coisas para você fazer além da sintaxe básica do Markdown. Elas podem ser realmente úteis quando for criar um comentário ou uma descrição para um Pull Request ou para uma Issue.

Task Lists

O primeiro recurso realmente útil do Markdown específico do GitHub, especialmente para usar em Pull Requests, é a Task List. Uma task list é uma lista de checkboxes das coisas que você precisa fazer. Colocar elas em uma Issue ou em um Pull Request normalmente indica o que você quer fazer antes de considerar o item completo.

Você pode criar uma task list como essa:

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

Se incluírmos isso na descrição do nosso Pull Request ou Issue, nós veremos ele ser renderizado como em Task lists renderizadas em um comentário de Markdown.

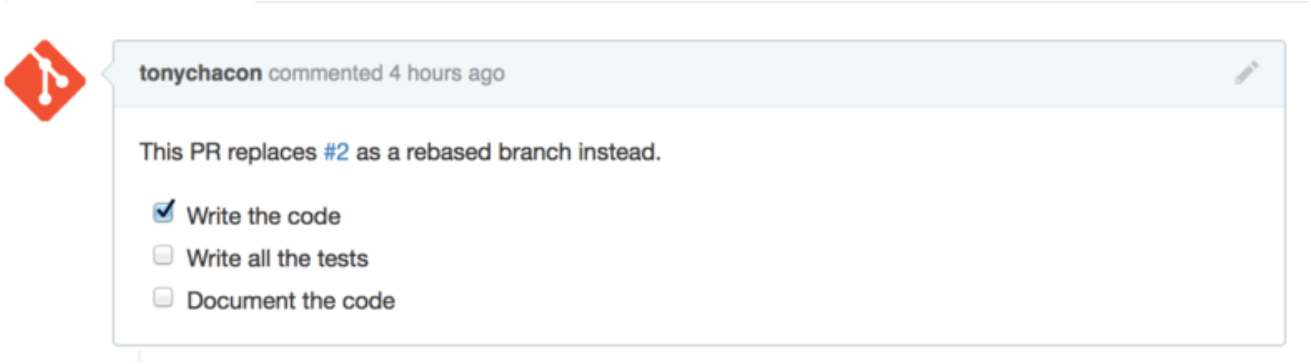


Figura 103. Task lists renderizadas em um comentário de Markdown.

Isso é normalmente usado em Pull Requests para indicar tudo aquilo que você gostaria de fazer em uma branch antes do Pull Request estar pronto para merge. A parte realmente legal é que você pode simplesmente clicar nas checkboxes para atualizar o comentário — você não precisa editar o Markdown diretamente para marcar suas tasks.

Não só isso, o GitHub vai procurar task lists nas suas Issues e Pull Requests e vai mostrá-las como metadata em uma páginas que as lista. Por exemplo, se você tem um Pull Request com tasks e você olha na página de resumo de todos os Pull Requests, você pode ver o quão longe isso vai. Isso ajuda as pessoas a dividir Pull Requests em subtasks e ajudar outras pessoas a acompanhar o progresso da branch. Você pode ver um exemplo disso em Resumo das Task lists na lista de Pull Requests..

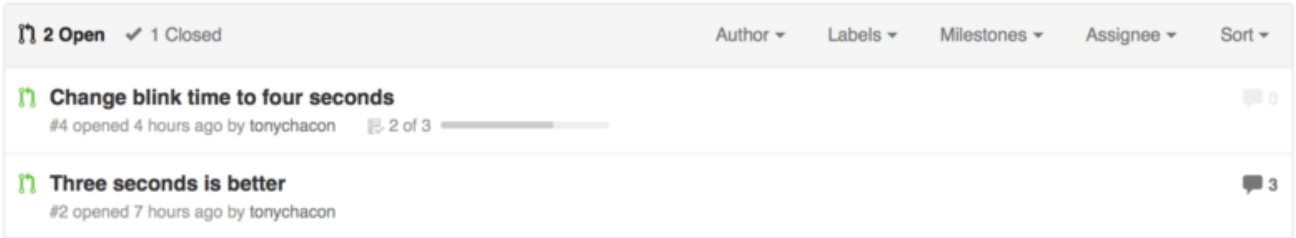


Figura 104. Resumo das Task lists na lista de Pull Requests.

Esses recursos são incrivelmente úteis quando você abre um Pull Request cedo e usa ele para traçar seu progresso por meio de implementações no recurso.

Trechos de código

Você também pode adicionar um trecho de código nos comentários. Isso é muito útil se você quer apresentar algo que *poderia* tentar fazer antes de realmente implementar como um commit na branch. Isso também é muito usado para adicionar um código de exemplo do que não está funcionando ou do que este Pull Request poderia implementar.

Para adicionar um trecho de código, você precisa cercá-lo com acentos graves.

```
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
```

Se você adicionar um nome de linguagem como fizemos com *java*, o GitHub também vai tentar destacar o trecho. No caso do exemplo acima, o código vai ser renderizado como em Um exemplo de código cercado renderizado..

Figura 105. Um exemplo de código cercado renderizado.

Citando

Se você está respondendo uma pequena parte de um comentário longo, você pode citá-la colocando o caractere `>` antes das linhas de código. De fato, isso é tão comum e útil que há uma atalho para isso. Se você destacar o texto em um comentário que você quer responder e aperta a tecla `r`, ele vai citar esse texto em uma caixa de comentário para você.

A citação vai parecer algo como isto:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

Uma vez renderizado, o comentário vai se parecer com Exemplo de citação renderizada.

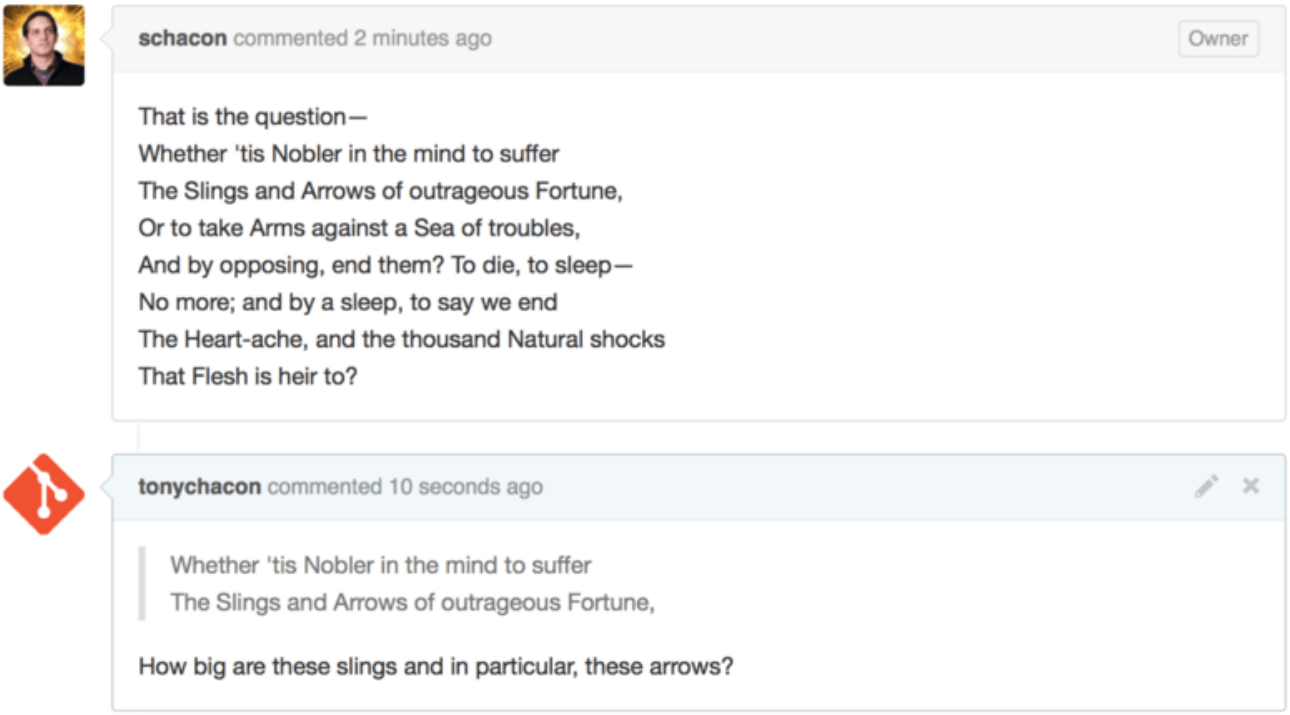


Figura 106. Exemplo de citação renderizada

Emoji

Finalmente, você também pode usar emoji nos seus comentários. Na verdade isso é usado extensivamente em comentários que você encontra em várias Issues e Pull Requests no GitHub. Também há um auxiliador de emoji no GitHub. Se você está digitando um comentário e começa com o caractere `:`, um autocompletador vai te ajudar a achar o que você está procurando.

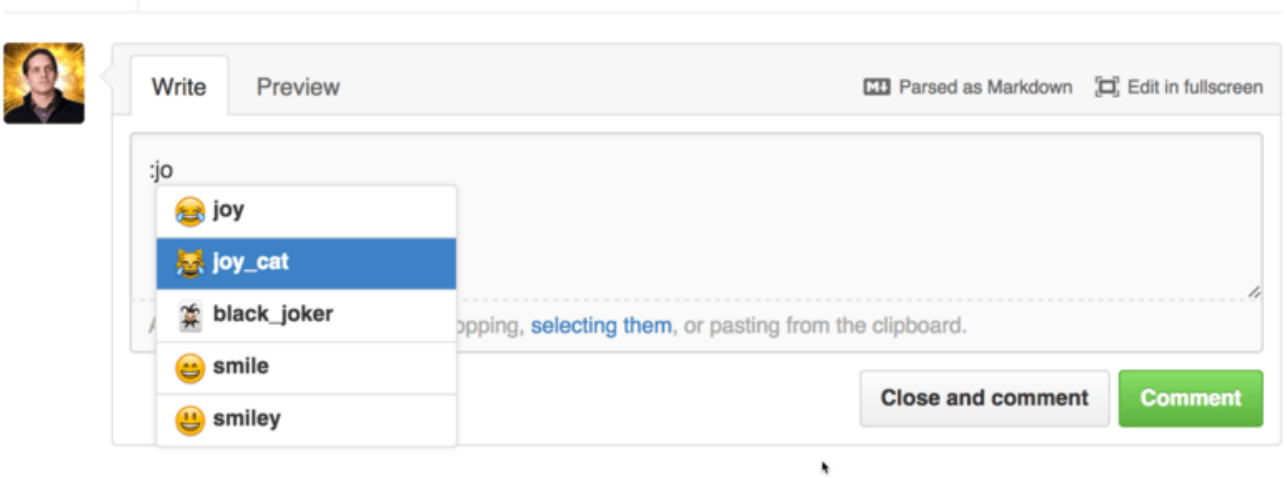


Figura 107. Autocompletador de emoji em ação.

Emojis têm a forma de `:<nome>:` em um comentário. Por exemplo, você poderia escrever algo como isto:

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

Quando renderizado, deveria se parecer com Comentando que nem louco com emoji.

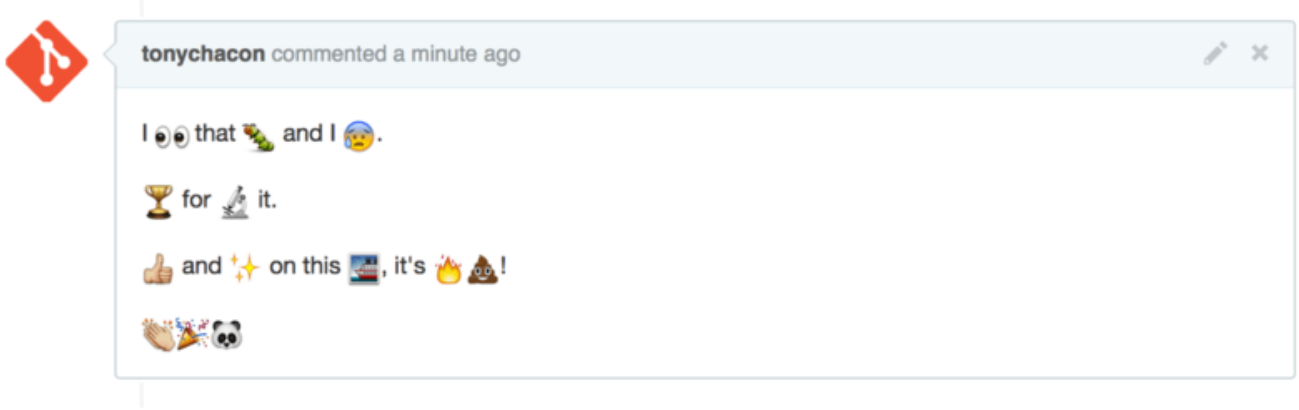


Figura 108. Comentando que nem louco com emoji

Note que isso é incrivelmente útil, mas adiciona um elemento de diversão e emoção em um meio no qual é difícil transmitir emoção.

NOTA Na verdade, há um grande número de serviços web que usam caracteres emoji esses dias. Um grande cola de referências para encontrar o emoji que expressa o que você quer dizer pode ser encontrada em:

http://www.emoji-cheat-sheet.com

Imagens

Isso tecnicamente não é o Markdown aprimorado do GitHub, mas é incrivelmente útil. Além de adicionar links Markdown de images nos comentários, o que pode ser bem difícil de encontrar e inserir as URLs, o GitHub permite arrastar e soltar imagens em áreas de texto para incorporá-las..

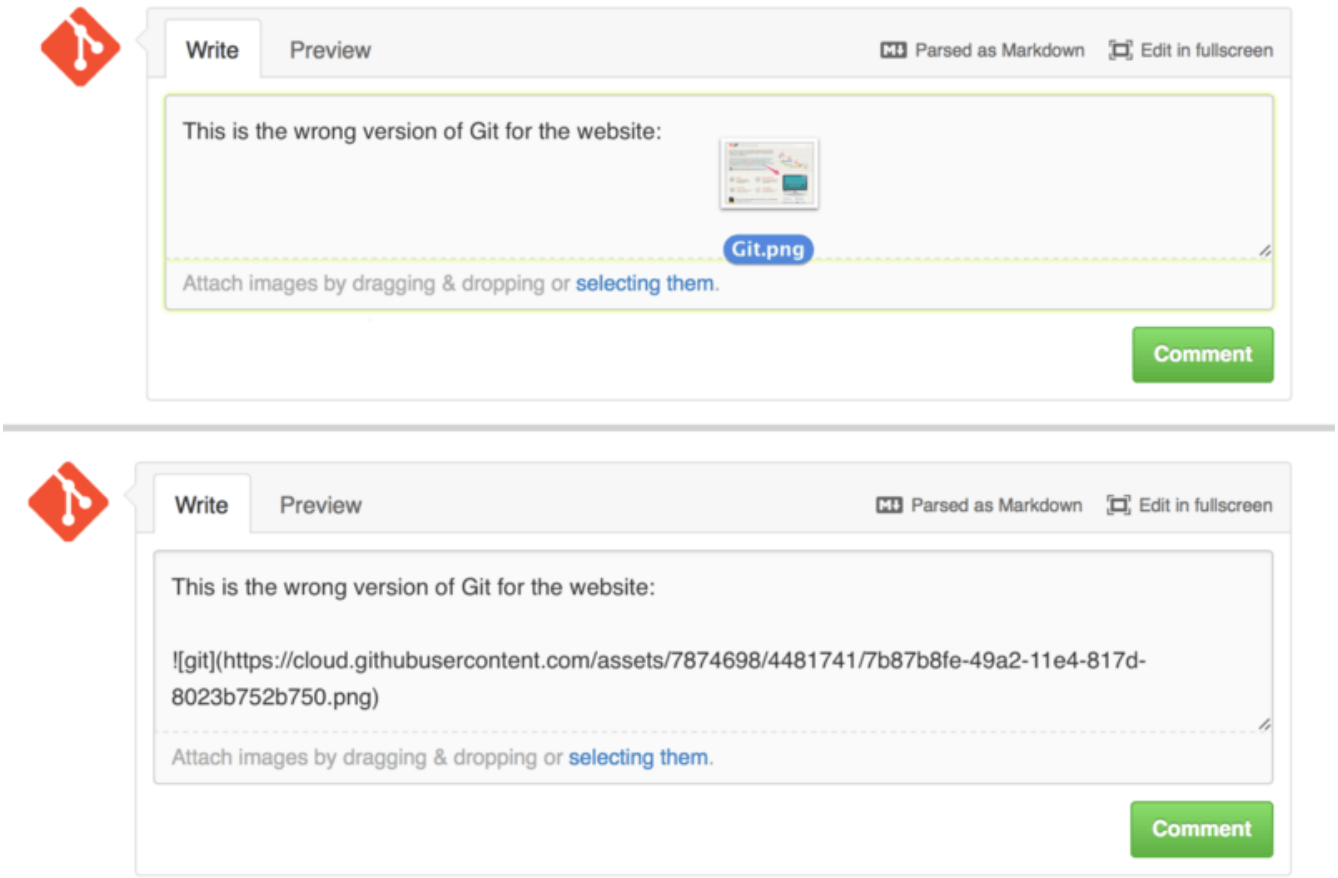


Figura 109. Arraste e solte imagens para carregá-las e incorporá-las.

Se você olhar em Arraste e solte imagens para carregá-las e incorporá-las., você pode ver um pequeno aviso “Ver como Markdown” acima da área de texto. Clicando nele gera uma cola de tudo que você pode fazer com o Markdown no GitHub.

Maintaining a Project

Now that we’re comfortable contributing to a project, let’s look at the other side: creating, maintaining and administering your own project.

Creating a New Repository

Let’s create a new repository to share our project code with. Start by clicking the “New repository” button on the right-hand side of the dashboard, or from the + button in the top toolbar next to your username as seen in The “New repository” dropdown..

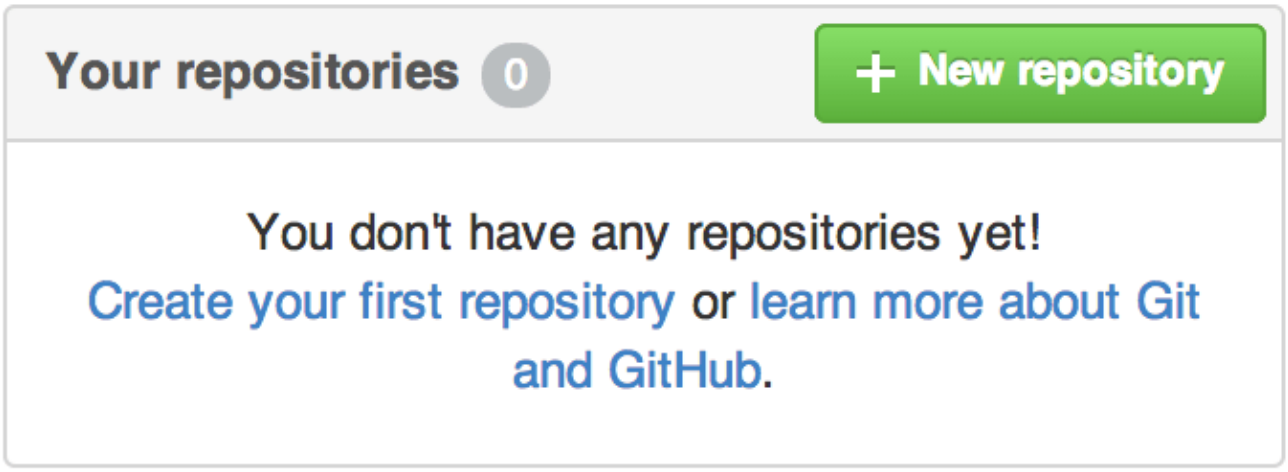


Figura 110. The “Your repositories” area.

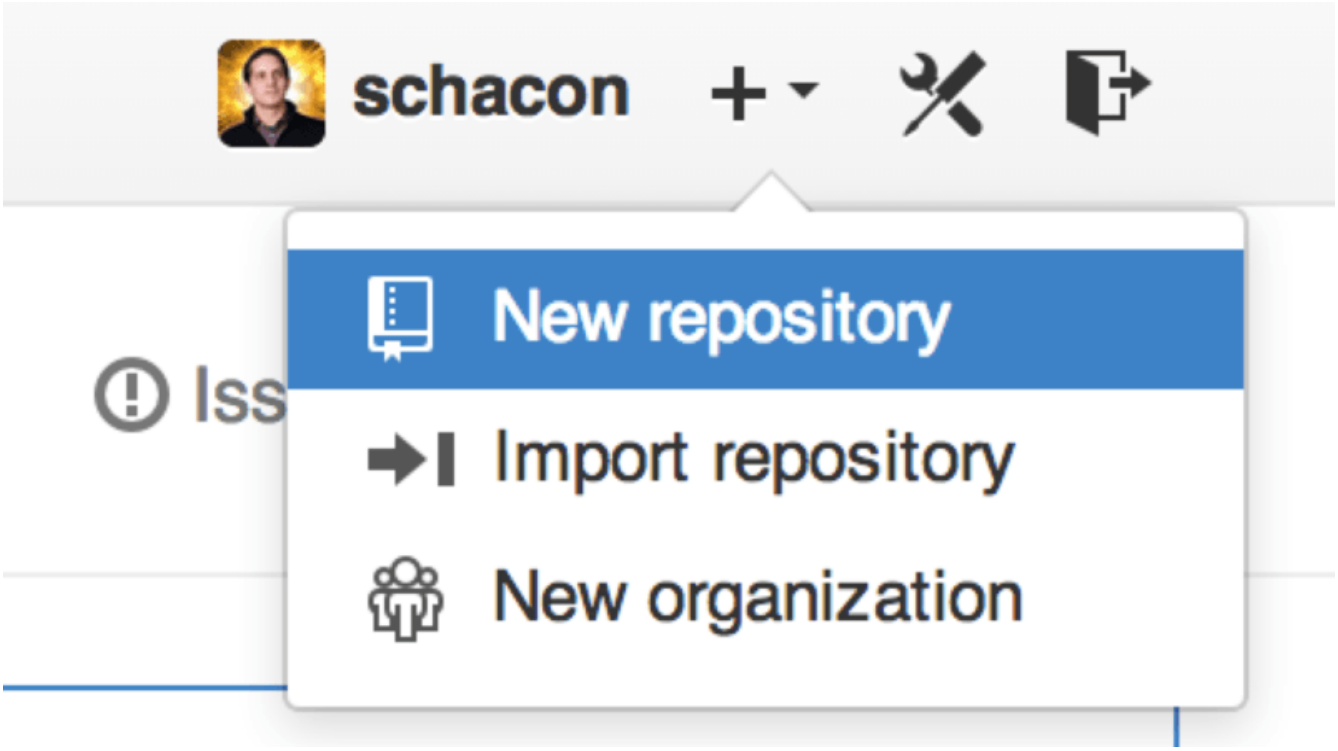


Figura 111. The “New repository” dropdown.

This takes you to the “new repository” form:

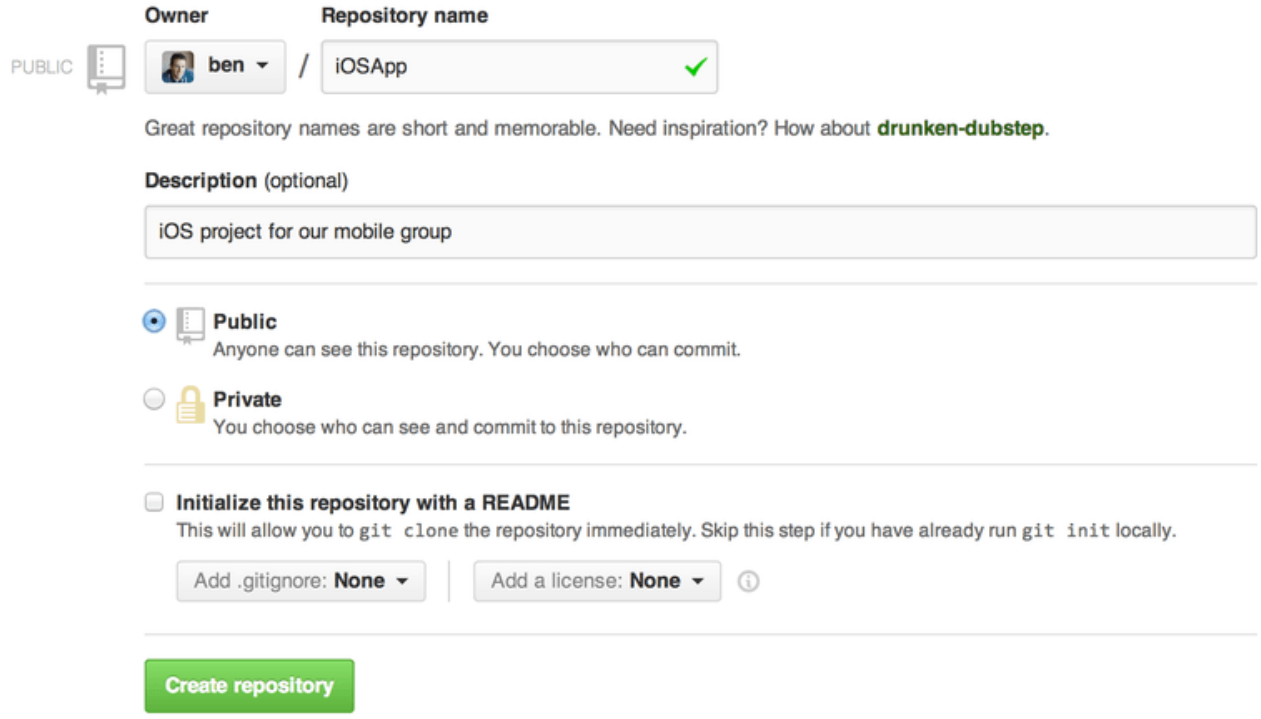


Figura 112. The "new repository" form.

All you really have to do here is provide a project name; the rest of the fields are completely optional. For now, just click the "Create Repository" button, and boom – you have a new repository on GitHub, named `<user>/<project_name>`.

Since you have no code there yet, GitHub will show you instructions for how to create a brand-new Git repository, or connect an existing Git project. We won't belabor this here; if you need a refresher, check out Fundamentos de Git.

Now that your project is hosted on GitHub, you can give the URL to anyone you want to share your project with. Every project on GitHub is accessible over HTTPS as `https://github.com/<user>/<project_name>`, and over SSH as `git@github.com:<user>/<project_name>`. Git can fetch from and push to both of these URLs, but they are access-controlled based on the credentials of the user connecting to them.

NOTA It is often preferable to share the HTTPS based URL for a public project, since the user does not have to have a GitHub account to access it for cloning. Users will have to have an account and an uploaded SSH key to access your project if you give them the SSH URL. The HTTPS one is also exactly the same URL they would paste into a browser to view the project there.

Adding Collaborators

If you're working with other people who you want to give commit access to, you need to add them as "collaborators". If Ben, Jeff, and Louise all sign up for accounts on GitHub, and you want to give them push access to your repository, you can add them to your project. Doing so will give them "push" access, which means they have both read and write access to the project and Git repository.

Click the "Settings" link at the bottom of the right-hand sidebar.

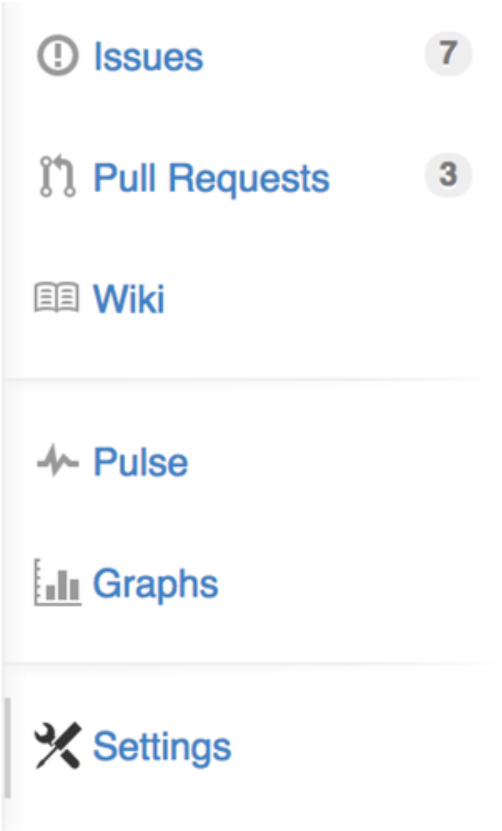


Figura 113. The repository settings link.

Then select “Collaborators” from the menu on the left-hand side. Then, just type a username into the box, and click “Add collaborator.” You can repeat this as many times as you like to grant access to everyone you like. If you need to revoke access, just click the “X” on the right-hand side of their row.

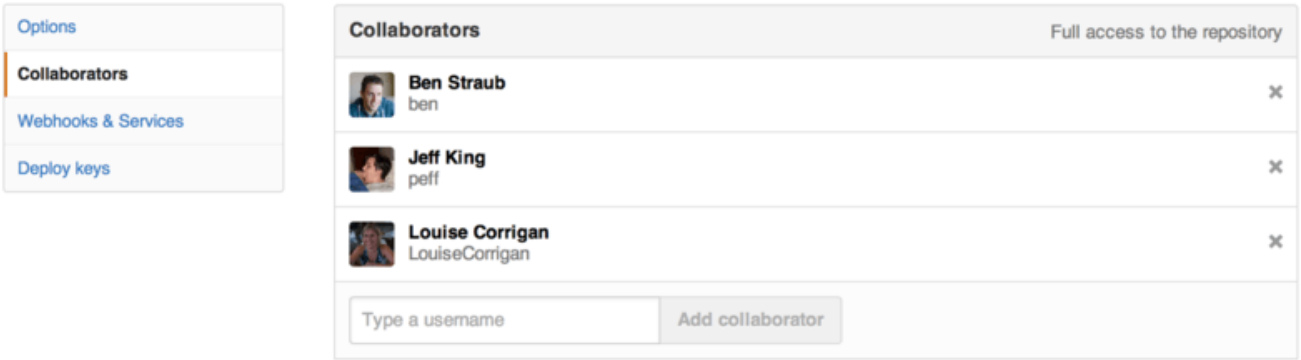


Figura 114. Repository collaborators.

Managing Pull Requests

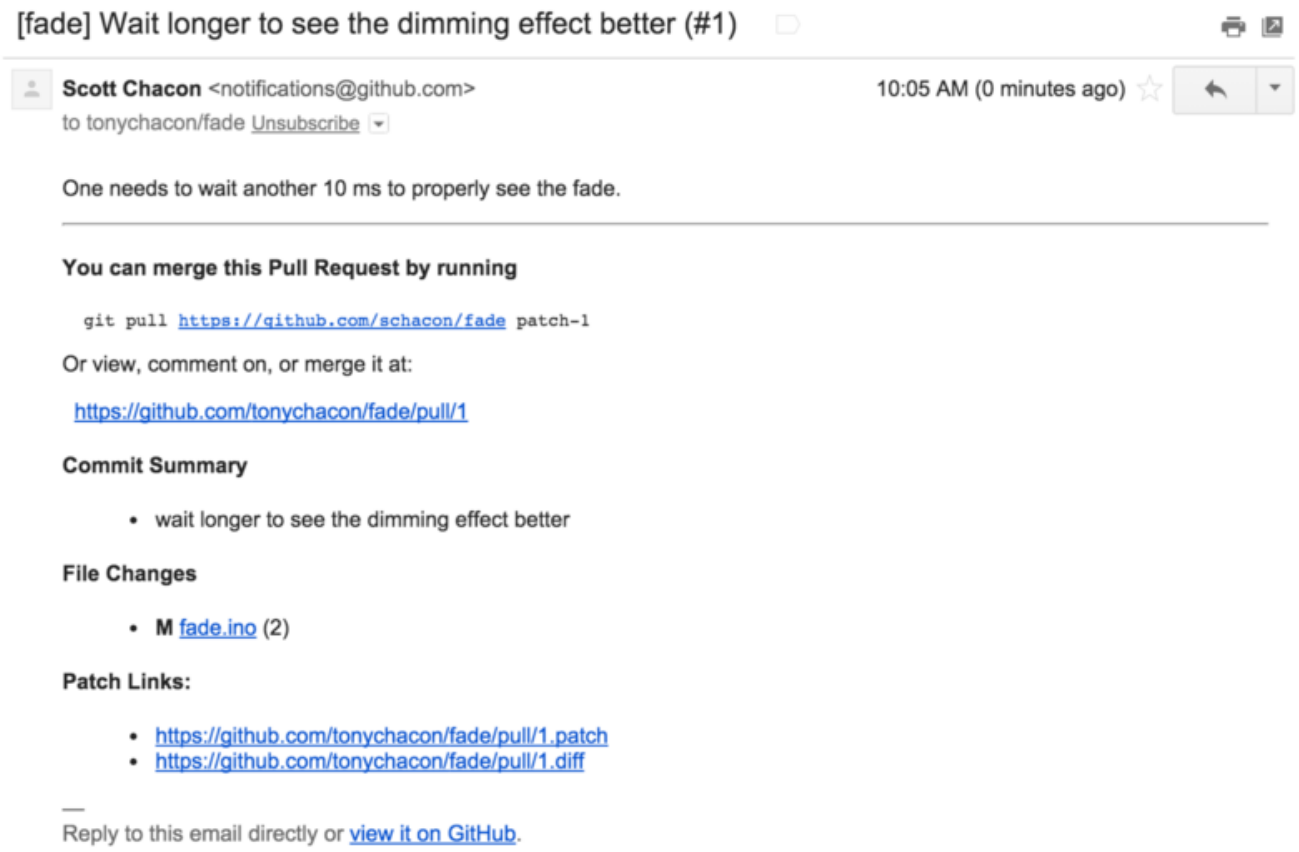
Now that you have a project with some code in it and maybe even a few collaborators who also have push access, let’s go over what to do when you get a Pull Request yourself.

Pull Requests can either come from a branch in a fork of your repository or they can come from another branch in the same repository. The only difference is that the ones in a fork are often from people where you can’t push to their branch and they can’t push to yours, whereas with internal Pull Requests generally both parties can access the branch.

For these examples, let's assume you are "tonychacon" and you've created a new Arduino code project named "fade".

Email Notifications

Someone comes along and makes a change to your code and sends you a Pull Request. You should get an email notifying you about the new Pull Request and it should look something like Email notification of a new Pull Request..


[fade] Wait longer to see the dimming effect better (#1)

Scott Chacon <notifications@github.com> 10:05 AM (0 minutes ago)
to tonychacon/fade Unsubscribe

One needs to wait another 10 ms to properly see the fade.

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

https://github.com/tonychacon/fade/pull/1

Commit Summary

- wait longer to see the dimming effect better

File Changes

- **M** fade.ino (2)

Patch Links:

- https://github.com/tonychacon/fade/pull/1.patch
- https://github.com/tonychacon/fade/pull/1.diff

—
Reply to this email directly or view it on GitHub.

Figura 115. Email notification of a new Pull Request.

There are a few things to notice about this email. It will give you a small diffstat — a list of files that have changed in the Pull Request and by how much. It gives you a link to the Pull Request on GitHub. It also gives you a few URLs that you can use from the command line.

If you notice the line that says `git pull <url> patch-1`, this is a simple way to merge in a remote branch without having to add a remote. We went over this quickly in Checking Out Remote Branches. If you wish, you can create and switch to a topic branch and then run this command to merge in the Pull Request changes.

The other interesting URLs are the `.diff` and `.patch` URLs, which as you may guess, provide unified diff and patch versions of the Pull Request. You could technically merge in the Pull Request work with something like this:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

Collaborating on the Pull Request

As we covered in O fluxo do GitHub, you can now have a conversation with the person who opened the Pull Request. You can comment on specific lines of code, comment on whole commits or comment on the entire Pull Request itself, using GitHub Flavored Markdown everywhere.

Every time someone else comments on the Pull Request you will continue to get email notifications so you know there is activity happening. They will each have a link to the Pull Request where the activity is happening and you can also directly respond to the email to comment on the Pull Request thread.

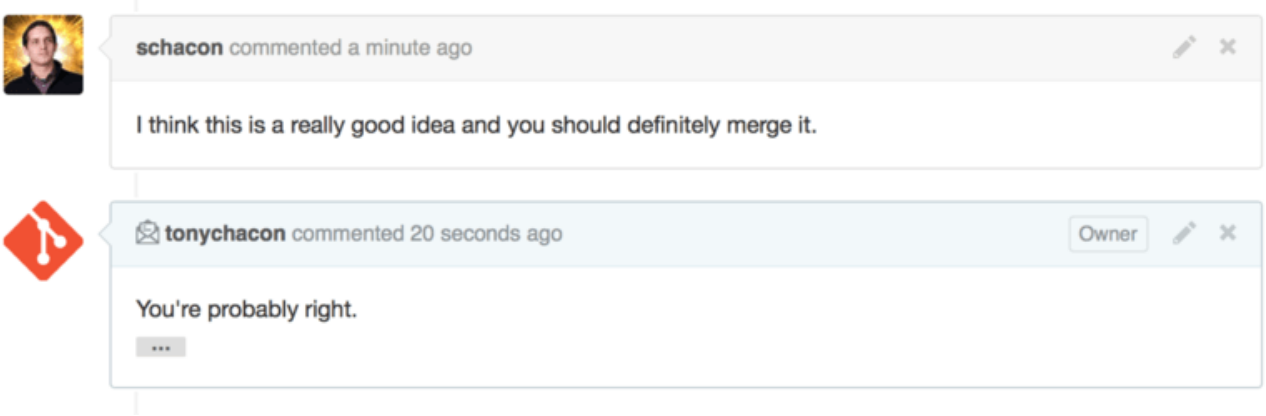


Figura 116. Responses to emails are included in the thread.

Once the code is in a place you like and want to merge it in, you can either pull the code down and merge it locally, either with the `git pull <url> <branch>` syntax we saw earlier, or by adding the fork as a remote and fetching and merging.

If the merge is trivial, you can also just hit the “Merge” button on the GitHub site. This will do a “non-fast-forward” merge, creating a merge commit even if a fast-forward merge was possible. This means that no matter what, every time you hit the merge button, a merge commit is created. As you can see in Merge button and instructions for merging a Pull Request manually., GitHub gives you all of this information if you click the hint link.

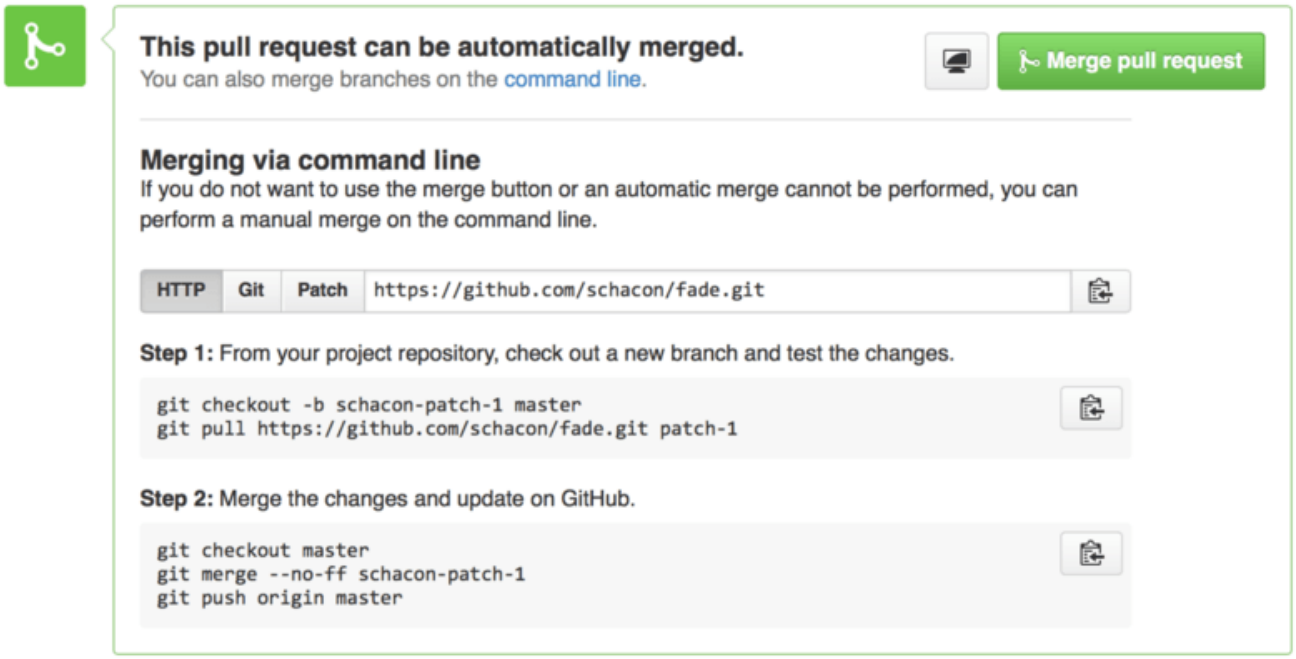


Figura 117. Merge button and instructions for merging a Pull Request manually.

If you decide you don't want to merge it, you can also just close the Pull Request and the person who opened it will be notified.

Pull Request Refs

If you're dealing with a **lot** of Pull Requests and don't want to add a bunch of remotes or do one time pulls every time, there is a neat trick that GitHub allows you to do. This is a bit of an advanced trick and we'll go over the details of this a bit more in The Refspec, but it can be pretty useful.

GitHub actually advertises the Pull Request branches for a repository as sort of pseudo-branches on the server. By default you don't get them when you clone, but they are there in an obscured way and you can access them pretty easily.

To demonstrate this, we're going to use a low-level command (often referred to as a "plumbing" command, which we'll read about more in Encanamento e Porcelana) called `ls-remote`. This command is generally not used in day-to-day Git operations but it's useful to show us what references are present on the server.

If we run this command against the "blink" repository we were using earlier, we will get a list of all the branches and tags and other references in the repository.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d    HEAD
10d539600d86723087810ec636870a504f4fee4d    refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e    refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3    refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1    refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

Of course, if you're in your repository and you run `git ls-remote origin` or whatever remote you want to check, it will show you something similar to this.

If the repository is on GitHub and you have any Pull Requests that have been opened, you'll get these references that are prefixed with `refs/pull/`. These are basically branches, but since they're not under `refs/heads/` you don't get them normally when you clone or fetch from the server — the process of fetching ignores them normally.

There are two references per Pull Request - the one that ends in `/head` points to exactly the same commit as the last commit in the Pull Request branch. So if someone opens a Pull Request in our repository and their branch is named `bug-fix` and it points to commit `a5a775`, then in **our** repository we will not have a `bug-fix` branch (since that's in their fork), but we *will* have `pull/<pr#>/head` that points to `a5a775`. This means that we can pretty easily pull down every Pull Request branch in one go without having to add a bunch of remotes.

Now, you could do something like fetching the reference directly.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

This tells Git, "Connect to the `origin` remote, and download the ref named `refs/pull/958/head`." Git happily obeys, and downloads everything you need to construct that ref, and puts a pointer to the commit you want under `.git/FETCH_HEAD`. You can follow that up with `git merge FETCH_HEAD` into a branch you want to test it in, but that merge commit message looks a bit weird. Also, if you're reviewing a **lot** of pull requests, this gets tedious.

There's also a way to fetch *all* of the pull requests, and keep them up to date whenever you connect to the remote. Open up `.git/config` in your favorite editor, and look for the `origin` remote. It should look a bit like this:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

That line that begins with `fetch =` is a "refspec." It's a way of mapping names on the remote with names in your local `.git` directory. This particular one tells Git, "the things on the remote that are

under `refs/heads` should go in my local repository under `refs/remotes/origin`." You can modify this section to add another refspec:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

That last line tells Git, "All the refs that look like `refs/pull/123/head` should be stored locally like `refs/remotes/origin/pr/123`." Now, if you save that file, and do a `git fetch`:

```
$ git fetch
# …
 * [new ref]         refs/pull/1/head -> origin/pr/1
 * [new ref]         refs/pull/2/head -> origin/pr/2
 * [new ref]         refs/pull/4/head -> origin/pr/4
# …
```

Now all of the remote pull requests are represented locally with refs that act much like tracking branches; they're read-only, and they update when you do a fetch. This makes it super easy to try the code from a pull request locally:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

The eagle-eyed among you would note the `head` on the end of the remote portion of the refspec. There's also a `refs/pull/#/merge` ref on the GitHub side, which represents the commit that would result if you push the "merge" button on the site. This can allow you to test the merge before even hitting the button.

Pull Requests on Pull Requests

Not only can you open Pull Requests that target the main or `master` branch, you can actually open a Pull Request targeting any branch in the network. In fact, you can even target another Pull Request.

If you see a Pull Request that is moving in the right direction and you have an idea for a change that depends on it or you're not sure is a good idea, or you just don't have push access to the target branch, you can open a Pull Request directly to it.

When you go to open a Pull Request, there is a box at the top of the page that specifies which branch you're requesting to pull to and which you're requesting to pull from. If you hit the "Edit" button at the right of that box you can change not only the branches but also which fork.

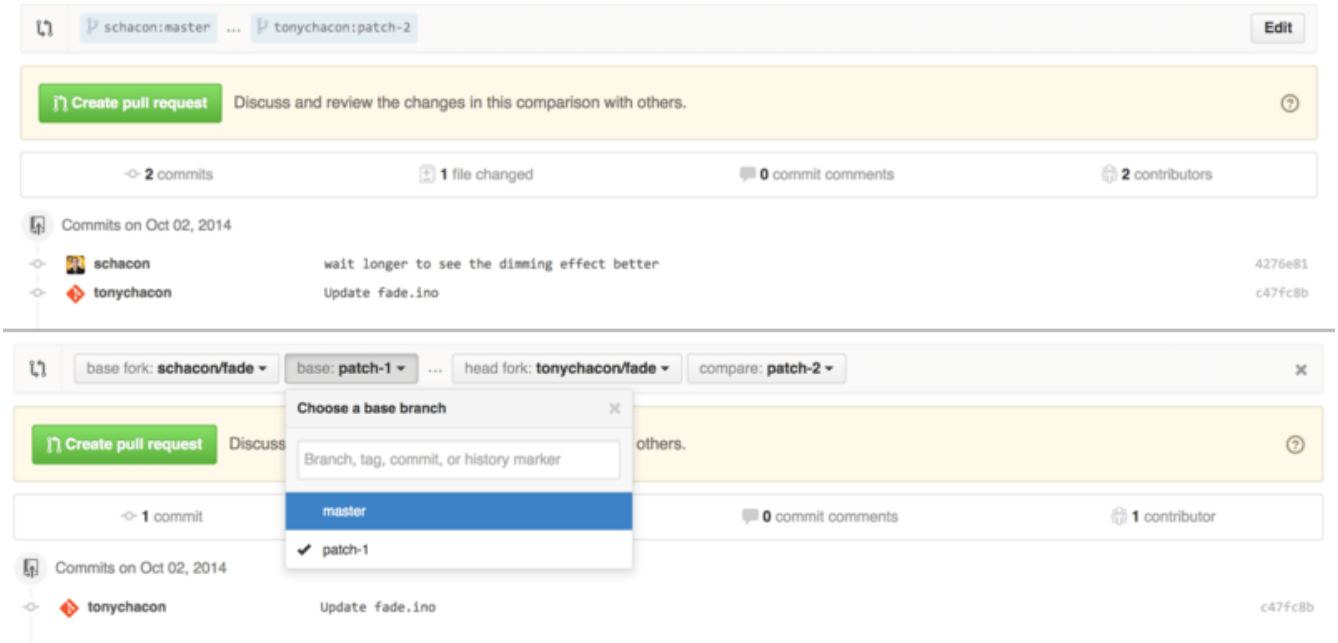


Figura 118. Manually change the Pull Request target fork and branch.

Here you can fairly easily specify to merge your new branch into another Pull Request or another fork of the project.

Mentions and Notifications

GitHub also has a pretty nice notifications system built in that can come in handy when you have questions or need feedback from specific individuals or teams.

In any comment you can start typing a `@` character and it will begin to autocomplete with the names and usernames of people who are collaborators or contributors in the project.

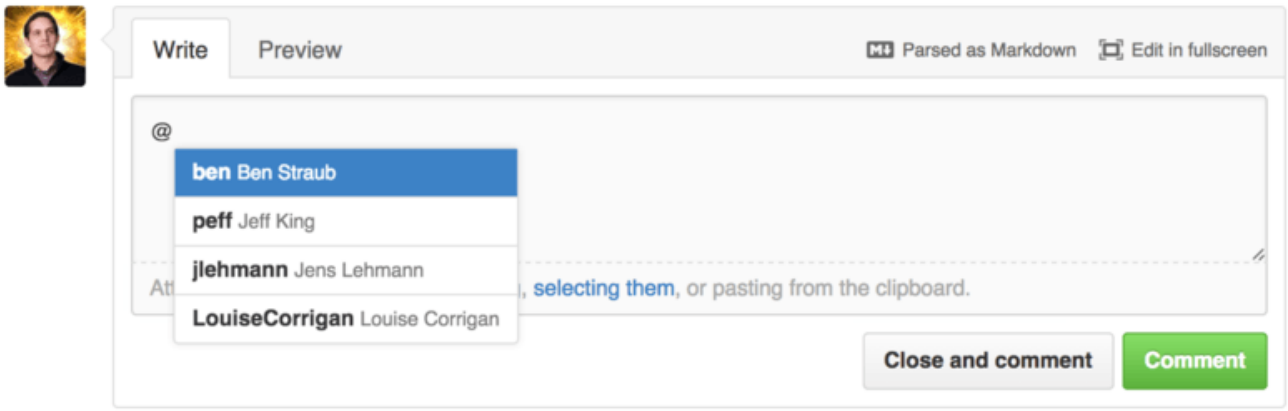


Figura 119. Start typing @ to mention someone.

You can also mention a user who is not in that dropdown, but often the autocompleter can make it faster.

Once you post a comment with a user mention, that user will be notified. This means that this can be a really effective way of pulling people into conversations rather than making them poll. Very often in Pull Requests on GitHub people will pull in other people on their teams or in their company to review an Issue or Pull Request.

If someone gets mentioned on a Pull Request or Issue, they will be “subscribed” to it and will continue getting notifications any time some activity occurs on it. You will also be subscribed to something if you opened it, if you’re watching the repository or if you comment on something. If you no longer wish to receive notifications, there is an “Unsubscribe” button on the page you can click to stop receiving updates on it.

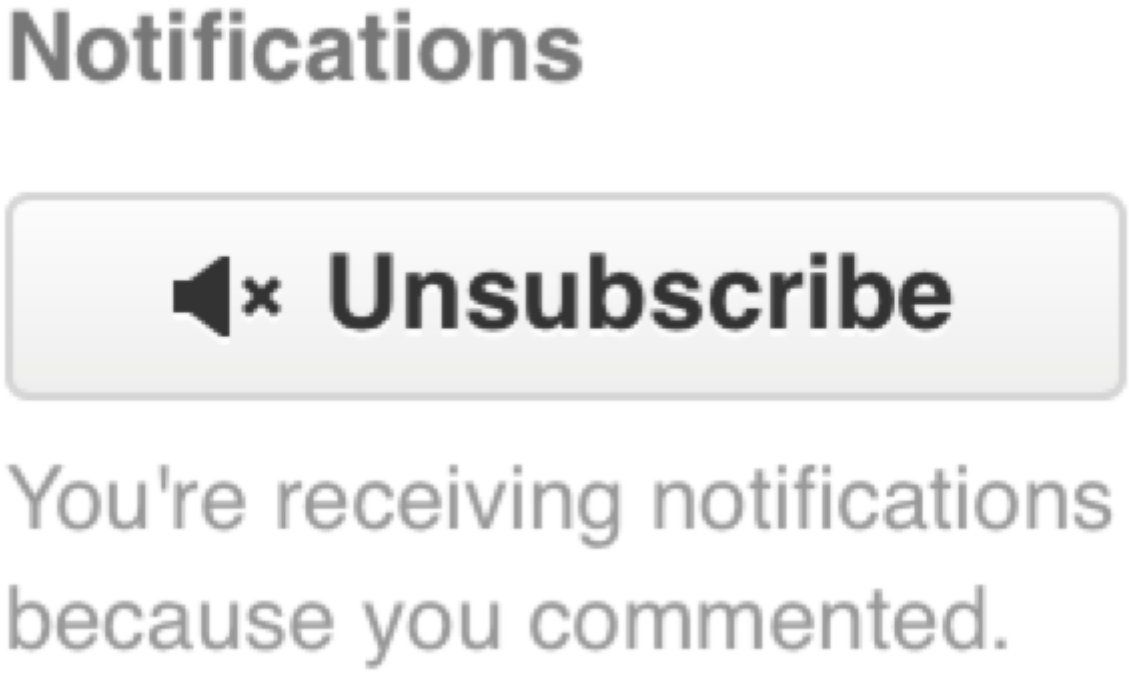


Figura 120. Unsubscribe from an Issue or Pull Request.

The Notifications Page

When we mention “notifications” here with respect to GitHub, we mean a specific way that GitHub tries to get in touch with you when events happen and there are a few different ways you can configure them. If you go to the “Notification center” tab from the settings page, you can see some of the options you have.

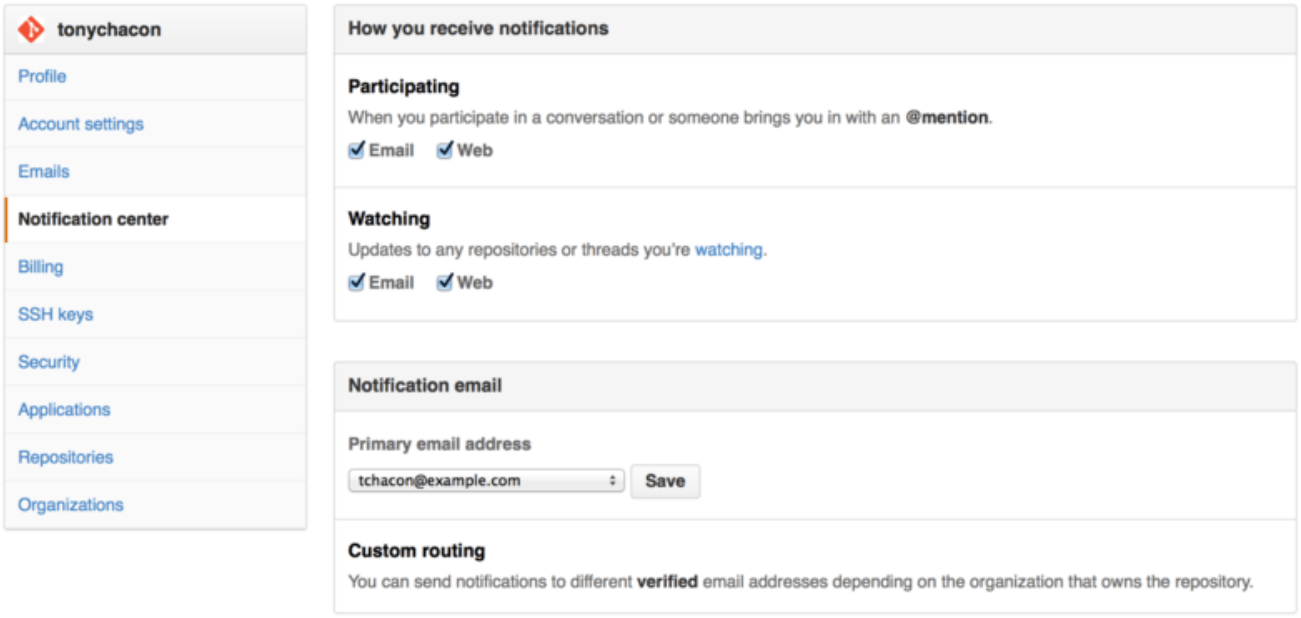


Figura 121. Notification center options.

The two choices are to get notifications over “Email” and over “Web” and you can choose either, neither or both for when you actively participate in things and for activity on repositories you are watching.

Web Notifications

Web notifications only exist on GitHub and you can only check them on GitHub. If you have this option selected in your preferences and a notification is triggered for you, you will see a small blue dot over your notifications icon at the top of your screen as seen in Notification center..

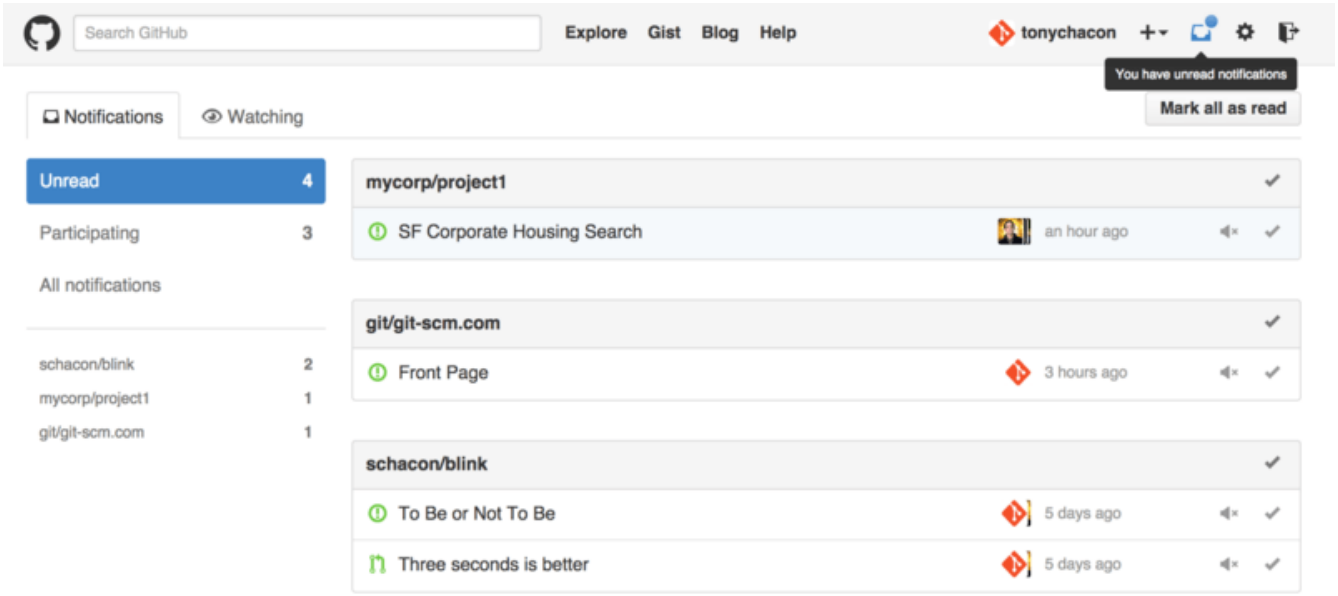


Figura 122. Notification center.

If you click on that, you will see a list of all the items you have been notified about, grouped by project. You can filter to the notifications of a specific project by clicking on its name in the left hand sidebar. You can also acknowledge the notification by clicking the checkmark icon next to any

notification, or acknowledge *all* of the notifications in a project by clicking the checkmark at the top of the group. There is also a mute button next to each checkmark that you can click to not receive any further notifications on that item.

All of these tools are very useful for handling large numbers of notifications. Many GitHub power users will simply turn off email notifications entirely and manage all of their notifications through this screen.

Email Notifications

Email notifications are the other way you can handle notifications through GitHub. If you have this turned on you will get emails for each notification. We saw examples of this in Comentários enviados como notificações pelo email and Email notification of a new Pull Request.. The emails will also be threaded properly, which is nice if you're using a threading email client.

There is also a fair amount of metadata embedded in the headers of the emails that GitHub sends you, which can be really helpful for setting up custom filters and rules.

For instance, if we look at the actual email headers sent to Tony in the email shown in Email notification of a new Pull Request., we will see the following among the information sent:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

There are a couple of interesting things here. If you want to highlight or re-route emails to this particular project or even Pull Request, the information in `Message-ID` gives you all the data in `<user>/<project>/<type>/<id>` format. If this were an issue, for example, the `<type>` field would have been "issues" rather than "pull".

The `List-Post` and `List-Unsubscribe` fields mean that if you have a mail client that understands those, you can easily post to the list or "Unsubscribe" from the thread. That would be essentially the same as clicking the "mute" button on the web version of the notification or "Unsubscribe" on the Issue or Pull Request page itself.

It's also worth noting that if you have both email and web notifications enabled and you read the email version of the notification, the web version will be marked as read as well if you have images allowed in your mail client.

Special Files

There are a couple of special files that GitHub will notice if they are present in your repository.

README

The first is the `README` file, which can be of nearly any format that GitHub recognizes as prose. For example, it could be `README`, `README.md`, `README.asciidoc`, etc. If GitHub sees a README file in your source, it will render it on the landing page of the project.

Many teams use this file to hold all the relevant project information for someone who might be new to the repository or project. This generally includes things like:

- What the project is for
- How to configure and install it
- An example of how to use it or get it running
- The license that the project is offered under
- How to contribute to it

Since GitHub will render this file, you can embed images or links in it for added ease of understanding.

CONTRIBUTING

The other special file that GitHub recognizes is the `CONTRIBUTING` file. If you have a file named `CONTRIBUTING` with any file extension, GitHub will show Opening a Pull Request when a CONTRIBUTING file exists. when anyone starts opening a Pull Request.

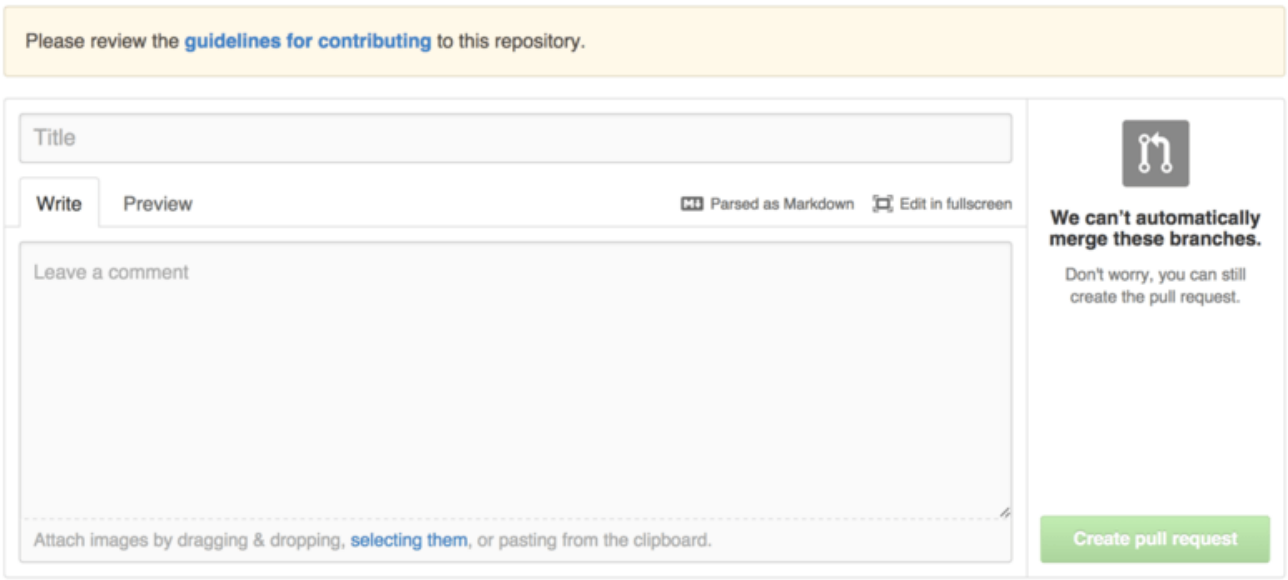


Figura 123. Opening a Pull Request when a CONTRIBUTING file exists.

The idea here is that you can specify specific things you want or don't want in a Pull Request sent to your project. This way people may actually read the guidelines before opening the Pull Request.

Project Administration

Generally there are not a lot of administrative things you can do with a single project, but there are a couple of items that might be of interest.

Changing the Default Branch

If you are using a branch other than “master” as your default branch that you want people to open Pull Requests on or see by default, you can change that in your repository’s settings page under the “Options” tab.

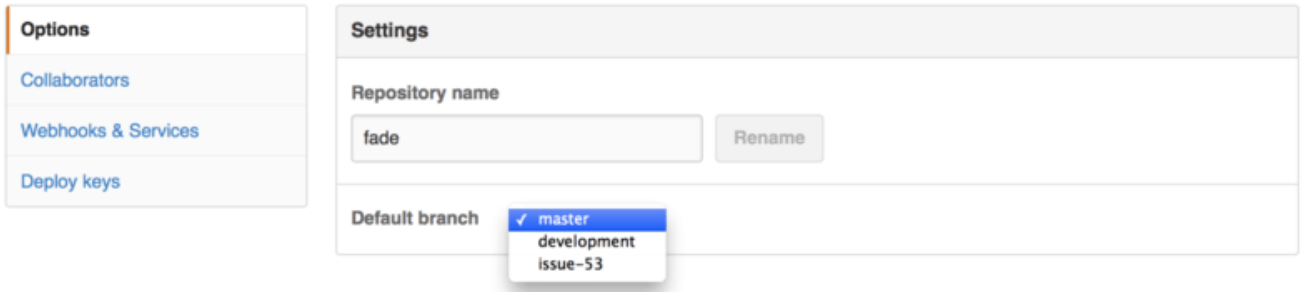


Figura 124. Change the default branch for a project.

Simply change the default branch in the dropdown and that will be the default for all major operations from then on, including which branch is checked out by default when someone clones the repository.

Transferring a Project

If you would like to transfer a project to another user or an organization in GitHub, there is a “Transfer ownership” option at the bottom of the same “Options” tab of your repository settings page that allows you to do this.

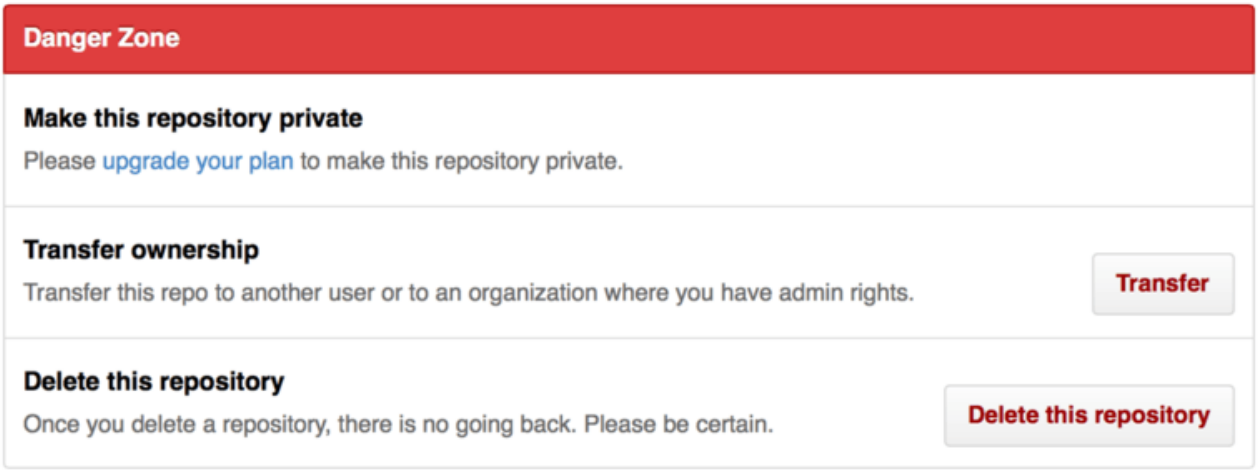


Figura 125. Transfer a project to another GitHub user or Organization.

This is helpful if you are abandoning a project and someone wants to take it over, or if your project is getting bigger and want to move it into an organization.

Not only does this move the repository along with all its watchers and stars to another place, it also sets up a redirect from your URL to the new place. It will also redirect clones and fetches from Git, not just web requests.

Managing an organization

In addition to single-user accounts, GitHub has what are called Organizations. Like personal accounts, Organizational accounts have a namespace where all their projects exist, but many other things are different. These accounts represent a group of people with shared ownership of projects, and there are many tools to manage subgroups of those people. Normally these accounts are used for Open Source groups (such as “perl” or “rails”) or companies (such as “google” or “twitter”).

Organization Basics

An organization is pretty easy to create; just click on the “+” icon at the top-right of any GitHub page, and select “New organization” from the menu.

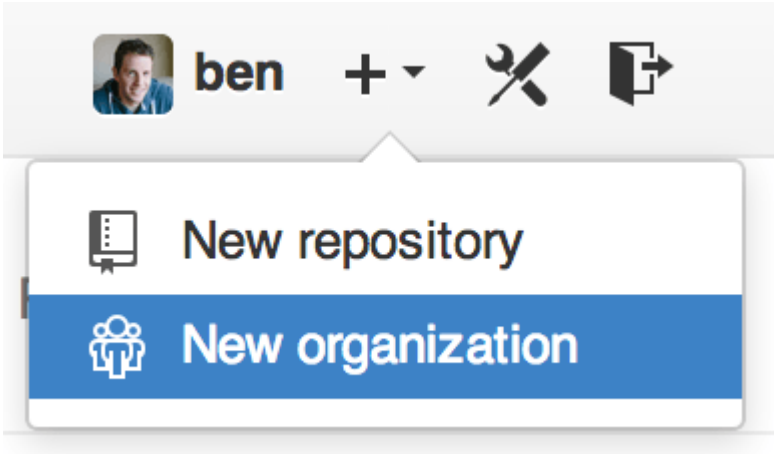


Figura 126. The “New organization” menu item.

First you’ll need to name your organization and provide an email address for a main point of contact for the group. Then you can invite other users to be co-owners of the account if you want to.

Follow these steps and you’ll soon be the owner of a brand-new organization. Like personal accounts, organizations are free if everything you plan to store there will be open source.

As an owner in an organization, when you fork a repository, you’ll have the choice of forking it to your organization’s namespace. When you create new repositories you can create them either under your personal account or under any of the organizations that you are an owner in. You also automatically “watch” any new repository created under these organizations.

Just like in Seu Avatar, you can upload an avatar for your organization to personalize it a bit. Also just like personal accounts, you have a landing page for the organization that lists all of your repositories and can be viewed by other people.

Now let’s cover some of the things that are a bit different with an organizational account.

Teams

Organizations are associated with individual people by way of teams, which are simply a grouping of individual user accounts and repositories within the organization and what kind of access those people have in those repositories.

For example, say your company has three repositories: `frontend`, `backend`, and `deployscripts`. You'd want your HTML/CSS/JavaScript developers to have access to `frontend` and maybe `backend`, and your Operations people to have access to `backend` and `deployscripts`. Teams make this easy, without having to manage the collaborators for every individual repository.

The Organization page shows you a simple dashboard of all the repositories, users and teams that are under this organization.

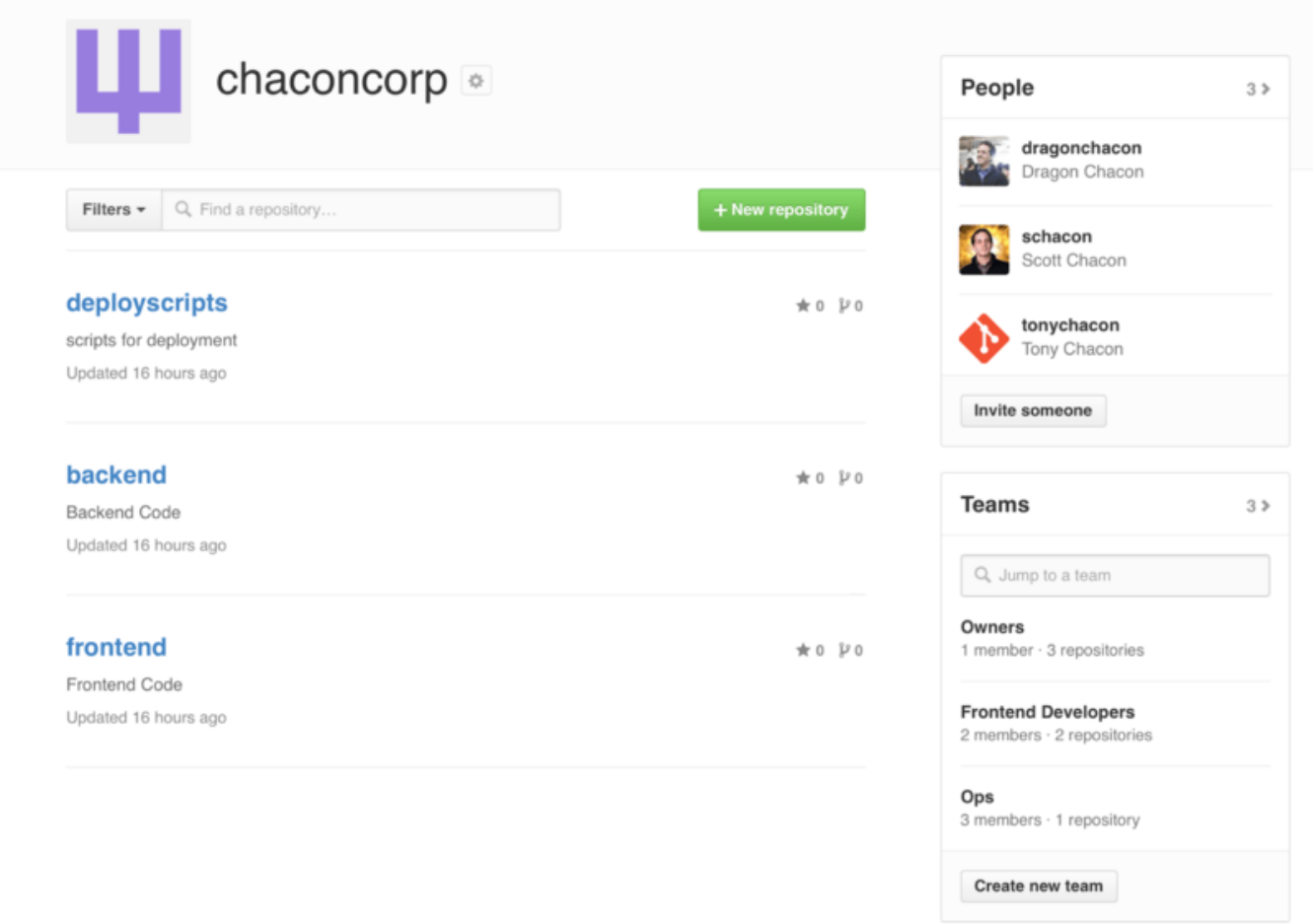


Figura 127. The Organization page.

To manage your Teams, you can click on the Teams sidebar on the right hand side of the page in The Organization page.. This will bring you to a page you can use to add members to the team, add repositories to the team or manage the settings and access control levels for the team. Each team can have read only, read/write or administrative access to the repositories. You can change that level by clicking the "Settings" button in The Team page..

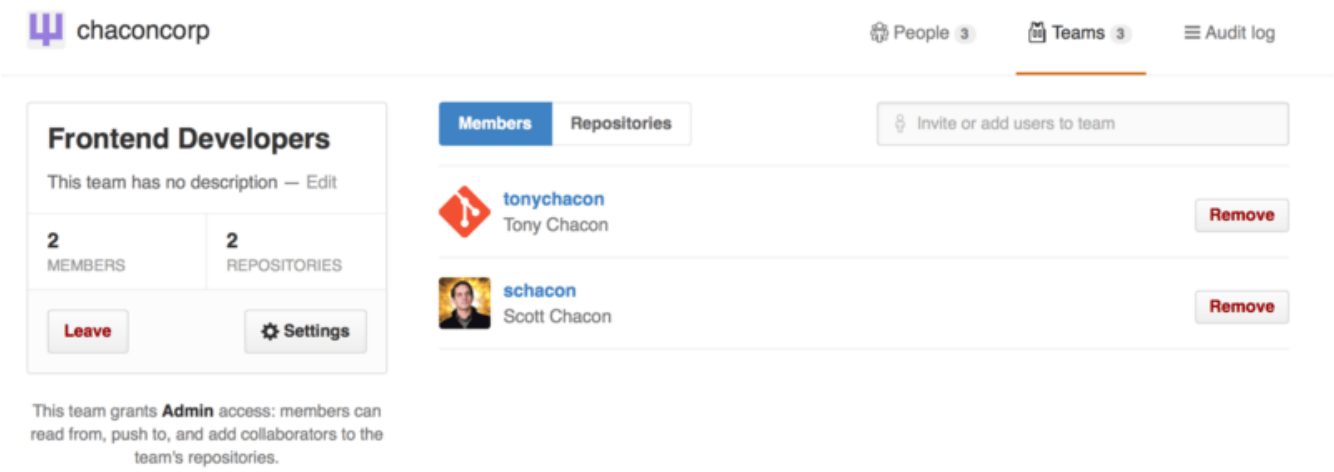


Figura 128. The Team page.

When you invite someone to a team, they will get an email letting them know they've been invited.

Additionally, team `@mentions` (such as `@acmecorp/frontend`) work much the same as they do with individual users, except that **all** members of the team are then subscribed to the thread. This is useful if you want the attention from someone on a team, but you don't know exactly who to ask.

A user can belong to any number of teams, so don't limit yourself to only access-control teams. Special-interest teams like `ux`, `css`, or `refactoring` are useful for certain kinds of questions, and others like `legal` and `colorblind` for an entirely different kind.

Audit Log

Organizations also give owners access to all the information about what went on under the organization. You can go to the *Audit Log* tab and see what events have happened at an organization level, who did them and where in the world they were done.

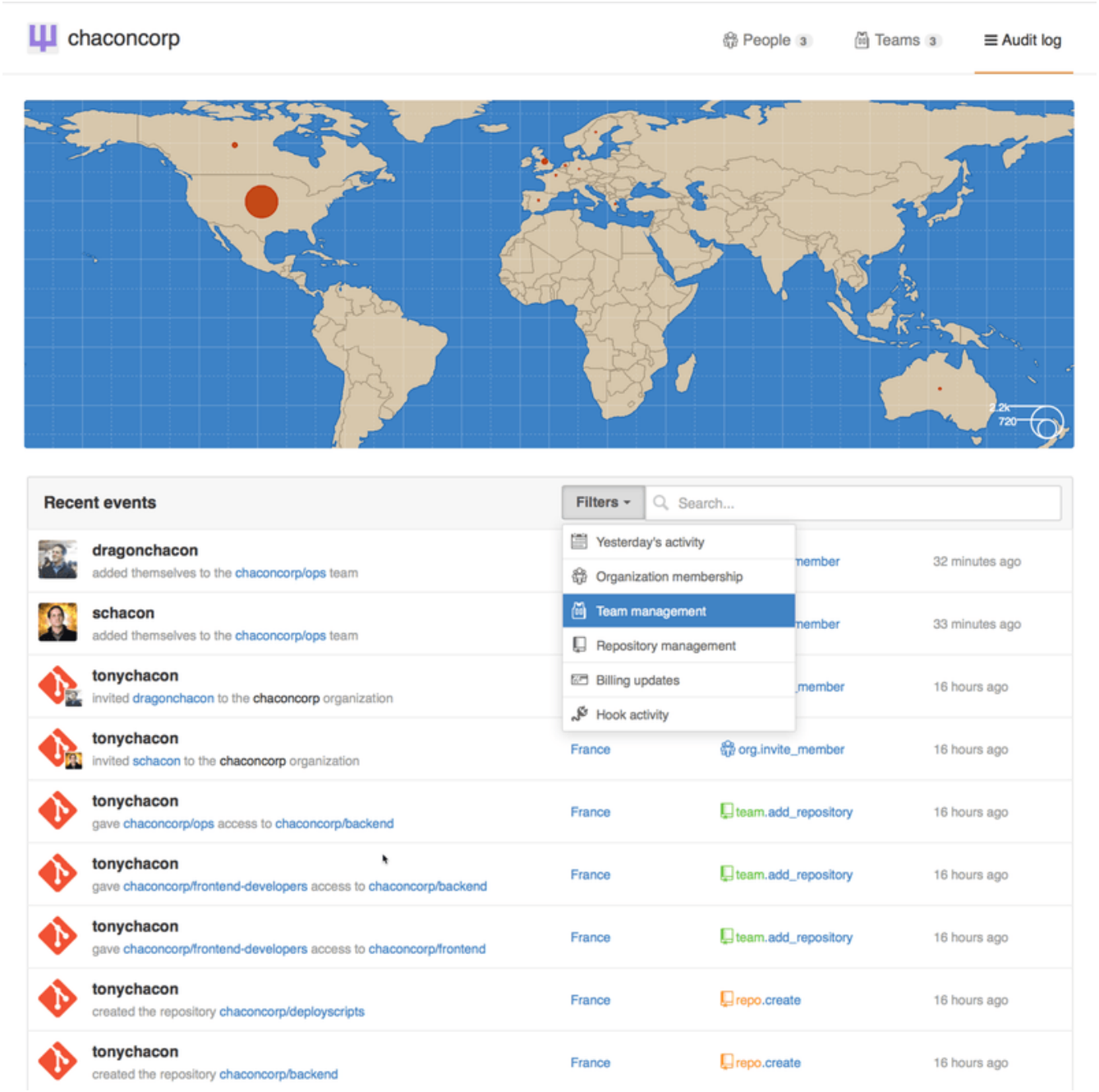


Figura 129. The Audit log.

You can also filter down to specific types of events, specific places or specific people.

Scripting GitHub

So now we've covered all of the major features and workflows of GitHub, but any large group or project will have customizations they may want to make or external services they may want to integrate.

Luckily for us, GitHub is really quite hackable in many ways. In this section we'll cover how to use the GitHub hooks system and its API to make GitHub work how we want it to.

Services and Hooks

The Hooks and Services section of GitHub repository administration is the easiest way to have

GitHub interact with external systems.

Services

First we'll take a look at Services. Both the Hooks and Services integrations can be found in the Settings section of your repository, where we previously looked at adding Collaborators and changing the default branch of your project. Under the "Webhooks and Services" tab you will see something like Services and Hooks configuration section..

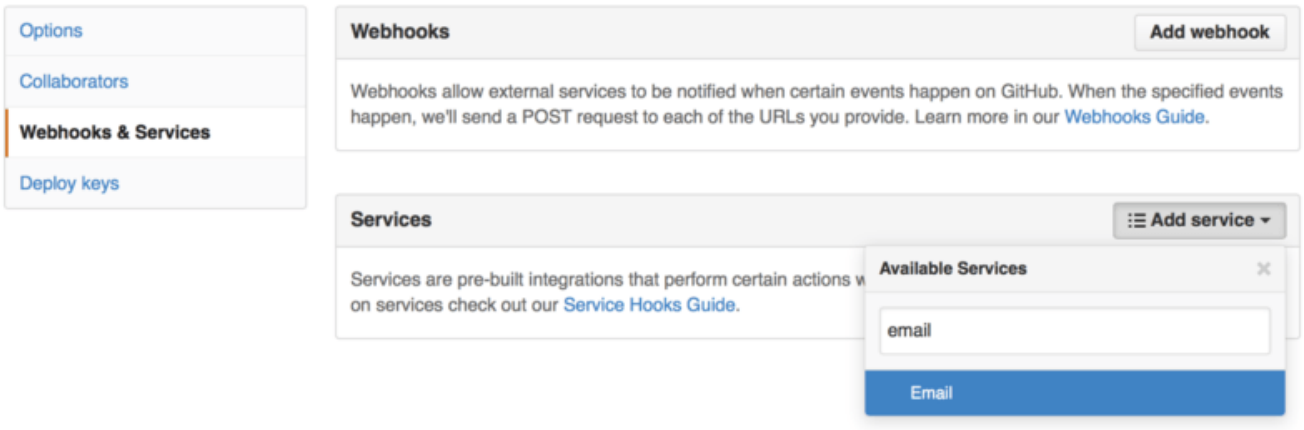


Figura 130. Services and Hooks configuration section.

There are dozens of services you can choose from, most of them integrations into other commercial and open source systems. Most of them are for Continuous Integration services, bug and issue trackers, chat room systems and documentation systems. We'll walk through setting up a very simple one, the Email hook. If you choose "email" from the "Add Service" dropdown, you'll get a configuration screen like Email service configuration..

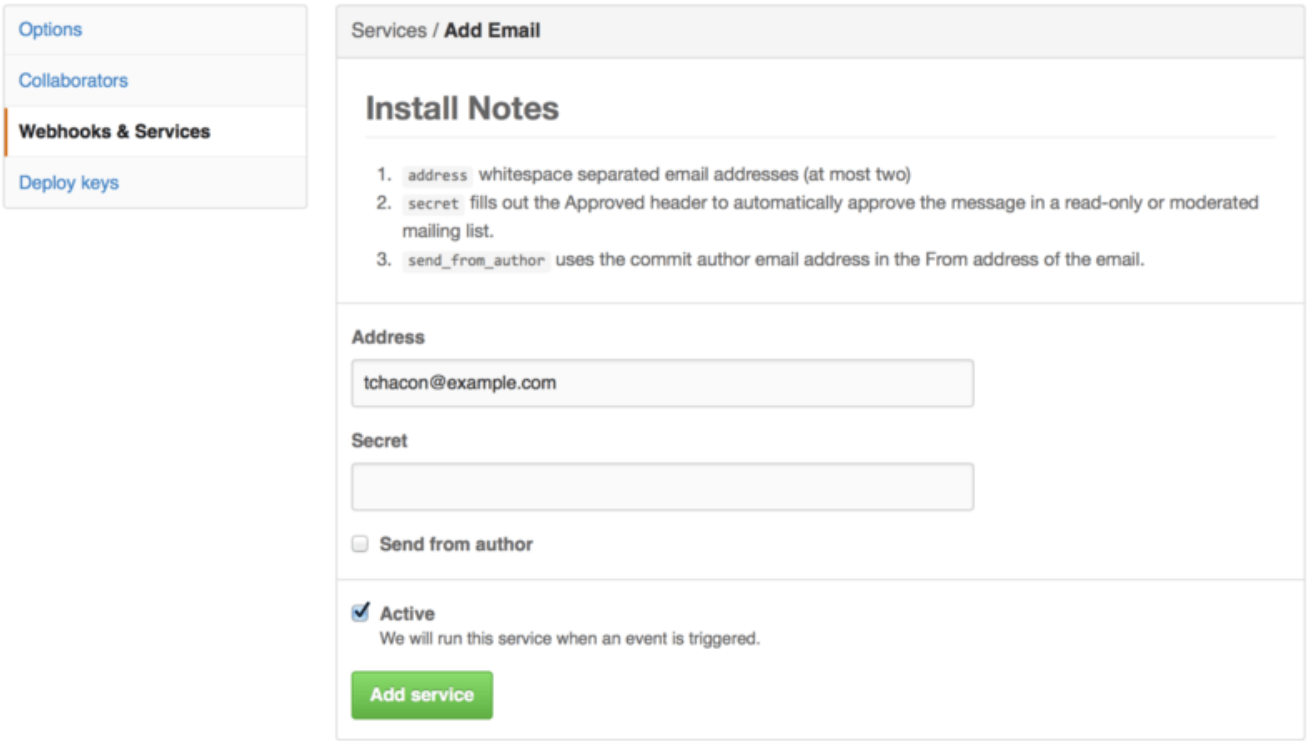


Figura 131. Email service configuration.

In this case, if we hit the “Add service” button, the email address we specified will get an email every time someone pushes to the repository. Services can listen for lots of different types of events, but most only listen for push events and then do something with that data.

If there is a system you are using that you would like to integrate with GitHub, you should check here to see if there is an existing service integration available. For example, if you’re using Jenkins to run tests on your codebase, you can enable the Jenkins builtin service integration to kick off a test run every time someone pushes to your repository.

Hooks

If you need something more specific or you want to integrate with a service or site that is not included in this list, you can instead use the more generic hooks system. GitHub repository hooks are pretty simple. You specify a URL and GitHub will post an HTTP payload to that URL on any event you want.

Generally the way this works is you can setup a small web service to listen for a GitHub hook payload and then do something with the data when it is received.

To enable a hook, you click the “Add webhook” button in Services and Hooks configuration section.. This will bring you to a page that looks like Web hook configuration..

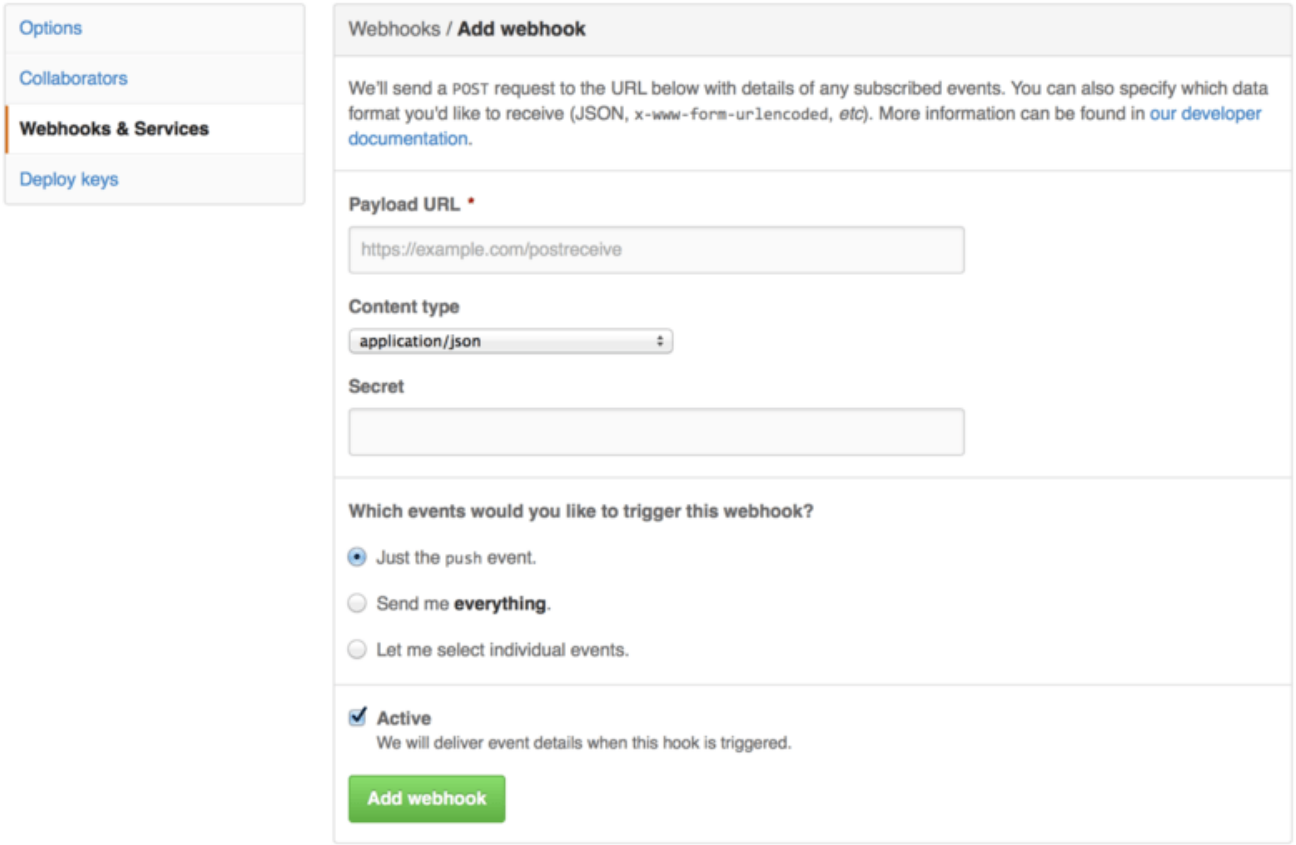


Figura 132. Web hook configuration.

The configuration for a web hook is pretty simple. In most cases you simply enter a URL and a secret key and hit “Add webhook”. There are a few options for which events you want GitHub to send you a payload for — the default is to only get a payload for the `push` event, when someone pushes new code to any branch of your repository.

Let's see a small example of a web service you may set up to handle a web hook. We'll use the Ruby web framework Sinatra since it's fairly concise and you should be able to easily see what we're doing.

Let's say we want to get an email if a specific person pushes to a specific branch of our project modifying a specific file. We could fairly easily do that with code like this:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
     branch == 'ref/heads/special-branch' &&
     files.include?('special-file.txt')

    Mail.deliver do
      from     'tchacon@example.com'
      to       'tchacon@example.com'
      subject  'Scott Changed the File'
      body     "ALARM"
    end
  end
end
```

Here we're taking the JSON payload that GitHub delivers us and looking up who pushed it, what branch they pushed to and what files were touched in all the commits that were pushed. Then we check that against our criteria and send an email if it matches.

In order to develop and test something like this, you have a nice developer console in the same screen where you set the hook up. You can see the last few deliveries that GitHub has tried to make for that webhook. For each hook you can dig down into when it was delivered, if it was successful and the body and headers for both the request and the response. This makes it incredibly easy to test and debug your hooks.

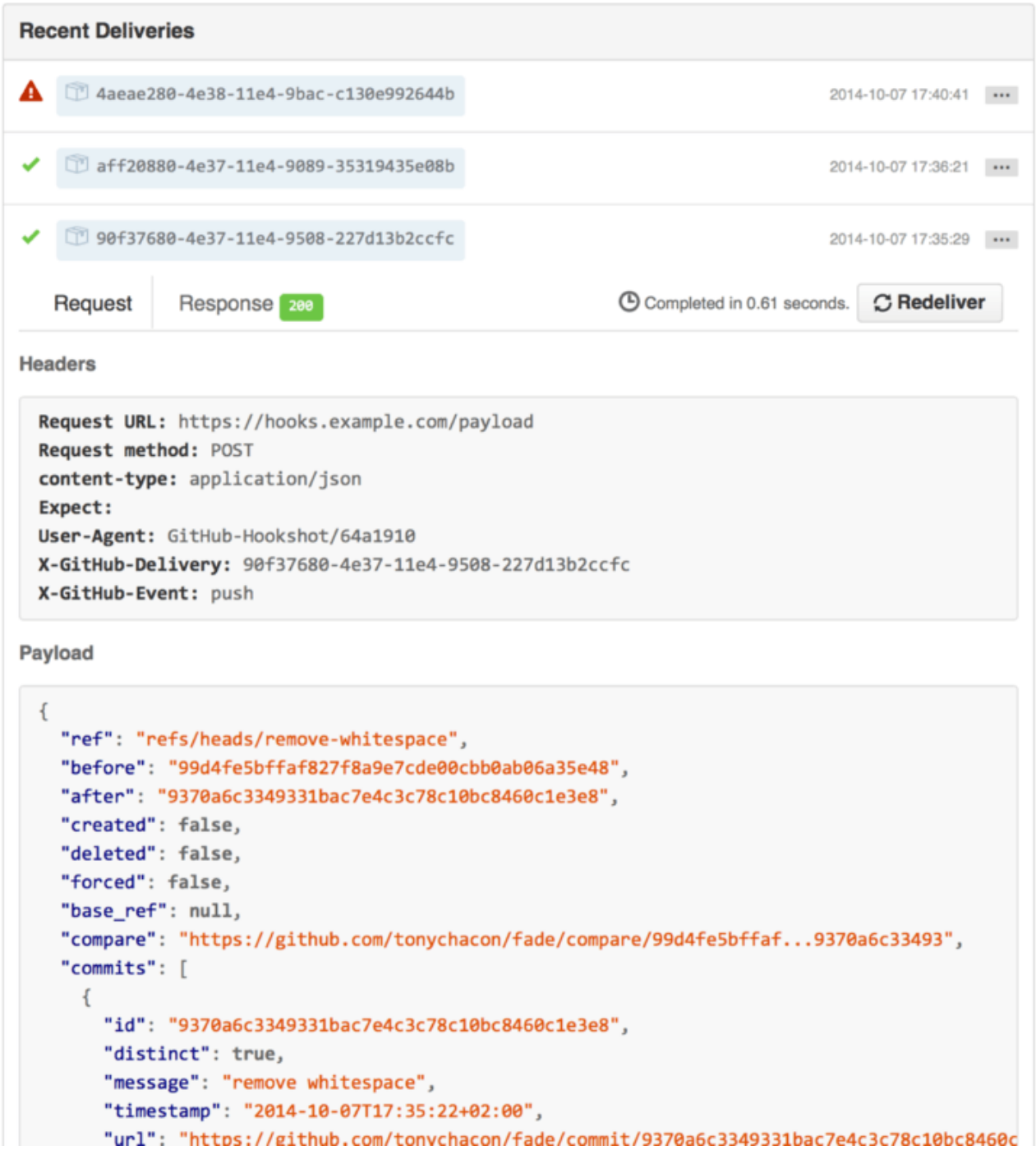


Figura 133. Web hook debugging information.

The other great feature of this is that you can redeliver any of the payloads to test your service easily.

For more information on how to write webhooks and all the different event types you can listen for, go to the GitHub Developer documentation at https://developer.github.com/webhooks/

The GitHub API

Services and hooks give you a way to receive push notifications about events that happen on your repositories, but what if you need more information about these events? What if you need to automate something like adding collaborators or labeling issues?

This is where the GitHub API comes in handy. GitHub has tons of API endpoints for doing nearly

anything you can do on the website in an automated fashion. In this section we'll learn how to authenticate and connect to the API, how to comment on an issue and how to change the status of a Pull Request through the API.

Basic Usage

The most basic thing you can do is a simple GET request on an endpoint that doesn't require authentication. This could be a user or read-only information on an open source project. For example, if we want to know more about a user named "schacon", we can run something like this:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
# …
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

There are tons of endpoints like this to get information about organizations, projects, issues, commits — just about anything you can publicly see on GitHub. You can even use the API to render arbitrary Markdown or find a `.gitignore` template.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Commenting on an Issue

However, if you want to do an action on the website such as comment on an Issue or Pull Request or if you want to view or interact with private content, you'll need to authenticate.

There are several ways to authenticate. You can use basic authentication with just your username and password, but generally it's a better idea to use a personal access token. You can generate this from the "Applications" tab of your settings page.

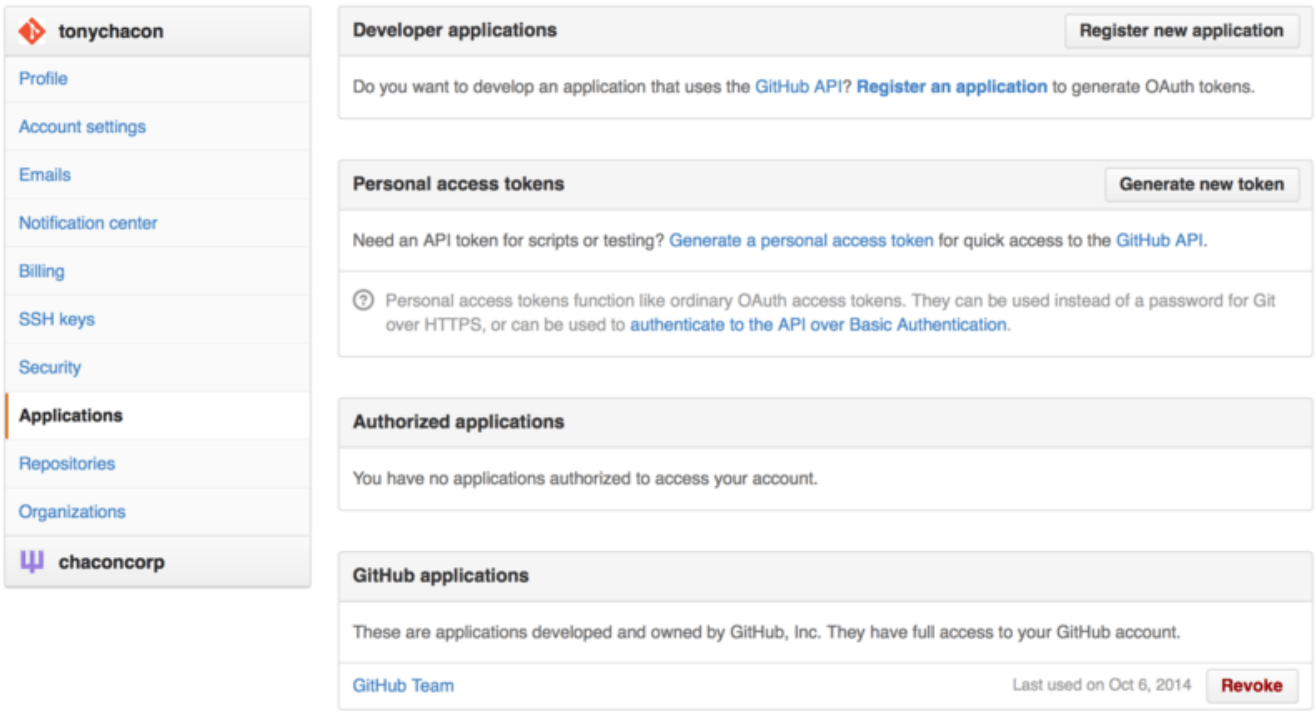


Figura 134. Generate your access token from the "Applications" tab of your settings page.

It will ask you which scopes you want for this token and a description. Make sure to use a good description so you feel comfortable removing the token when your script or application is no longer used.

GitHub will only show you the token once, so be sure to copy it. You can now use this to authenticate in your script instead of using a username and password. This is nice because you can limit the scope of what you want to do and the token is revocable.

This also has the added advantage of increasing your rate limit. Without authenticating, you will be limited to 60 requests per hour. If you authenticate you can make up to 5,000 requests per hour.

So let's use it to make a comment on one of our issues. Let's say we want to leave a comment on a specific issue, Issue #6. To do so we have to do an HTTP POST request to `repos/<user>/<repo>/issues/<num>/comments` with the token we just generated as an Authorization header.

```
$ curl -H "Content-Type: application/json" \
       -H "Authorization: token TOKEN" \
       --data '{"body":"A new comment, :+1:"}' \
       https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

Now if you go to that issue, you can see the comment that we just successfully posted as in A comment posted from the GitHub API..

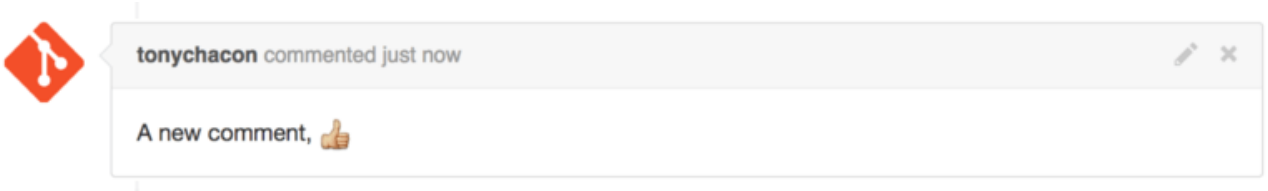


Figura 135. A comment posted from the GitHub API.

You can use the API to do just about anything you can do on the website — creating and setting milestones, assigning people to Issues and Pull Requests, creating and changing labels, accessing commit data, creating new commits and branches, opening, closing or merging Pull Requests, creating and editing teams, commenting on lines of code in a Pull Request, searching the site and on and on.

Changing the Status of a Pull Request

There is one final example we'll look at since it's really useful if you're working with Pull Requests. Each commit can have one or more statuses associated with it and there is an API to add and query that status.

Most of the Continuous Integration and testing services make use of this API to react to pushes by testing the code that was pushed, and then report back if that commit has passed all the tests. You could also use this to check if the commit message is properly formatted, if the submitter followed all your contribution guidelines, if the commit was validly signed — any number of things.

Let's say you set up a webhook on your repository that hits a small web service that checks for a `Signed-off-by` string in the commit message.

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"       => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

Hopefully this is fairly simple to follow. In this web hook handler we look through each commit that was just pushed, we look for the string *Signed-off-by* in the commit message and finally we POST via HTTP to the `/repos/<user>/<repo>/statuses/<commit_sha>` API endpoint with the status.

In this case you can send a state (*success*, *failure*, *error*), a description of what happened, a target URL the user can go to for more information and a “context” in case there are multiple statuses for a single commit. For example, a testing service may provide a status and a validation service like this may also provide a status — the “context” field is how they’re differentiated.

If someone opens a new Pull Request on GitHub and this hook is set up, you may see something like Commit status via the API..

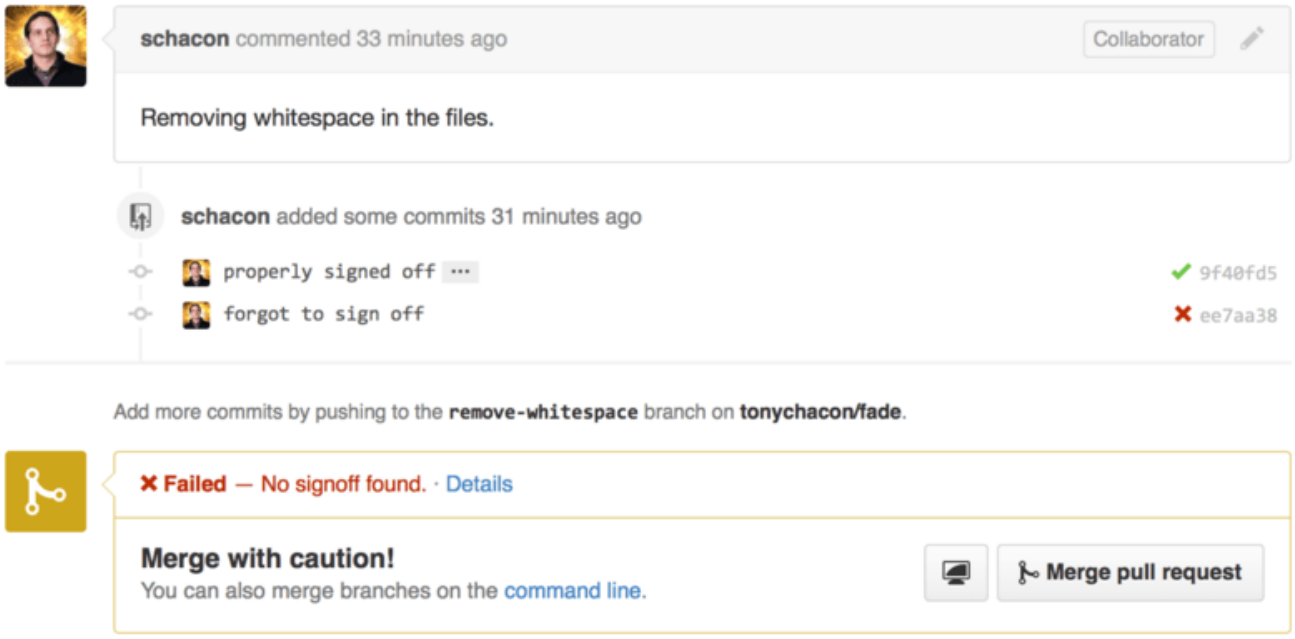


Figura 136. Commit status via the API.

You can now see a little green check mark next to the commit that has a “Signed-off-by” string in the message and a red cross through the one where the author forgot to sign off. You can also see that the Pull Request takes the status of the last commit on the branch and warns you if it is a failure. This is really useful if you’re using this API for test results so you don’t accidentally merge something where the last commit is failing tests.

Octokit

Though we’ve been doing nearly everything through `curl` and simple HTTP requests in these examples, several open-source libraries exist that make this API available in a more idiomatic way. At the time of this writing, the supported languages include Go, Objective-C, Ruby, and .NET. Check out http://github.com/octokit for more information on these, as they handle much of the HTTP for you.

Hopefully these tools can help you customize and modify GitHub to work better for your specific workflows. For complete documentation on the entire API as well as guides for common tasks, check out https://developer.github.com.

Summary

Now you’re a GitHub user. You know how to create an account, manage an organization, create and push to repositories, contribute to other people’s projects and accept contributions from others. In the next chapter, you’ll learn more powerful tools and tips for dealing with complex situations, which will truly make you a Git master.

Git Tools

By now, you've learned most of the day-to-day commands and workflows that you need to manage or maintain a Git repository for your source code control. You've accomplished the basic tasks of tracking and committing files, and you've harnessed the power of the staging area and lightweight topic branching and merging.

Now you'll explore a number of very powerful things that Git can do that you may not necessarily use on a day-to-day basis but that you may need at some point.

Revision Selection

Git allows you to specify specific commits or a range of commits in several ways. They aren't necessarily obvious but are helpful to know.

Single Revisions

You can obviously refer to a commit by the SHA-1 hash that it's given, but there are more human-friendly ways to refer to commits as well. This section outlines the various ways you can refer to a single commit.

Short SHA-1

Git is smart enough to figure out what commit you meant to type if you provide the first few characters, as long as your partial SHA-1 is at least four characters long and unambiguous – that is, only one object in the current repository begins with that partial SHA-1.

For example, to see a specific commit, suppose you run a `git log` command and identify the commit where you added certain functionality:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

In this case, choose `1c002dd...`. If you `git show` that commit, the following commands are equivalent (assuming the shorter versions are unambiguous):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git can figure out a short, unique abbreviation for your SHA-1 values. If you pass `--abbrev-commit` to the `git log` command, the output will use shorter values but keep them unique; it defaults to using seven characters but makes them longer if necessary to keep the SHA-1 unambiguous:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Generally, eight to ten characters are more than enough to be unique within a project.

As an example, the Linux kernel, which is a pretty large project with over 450k commits and 3.6 million objects, has no two objects whose SHA-1s overlap more than the first 11 characters.

A SHORT NOTE ABOUT SHA-1

A lot of people become concerned at some point that they will, by random happenstance, have two objects in their repository that hash to the same SHA-1 value. What then?

If you do happen to commit an object that hashes to the same SHA-1 value as a previous object in your repository, Git will see the previous object already in your Git database and assume it was already written. If you try to check out that object again at some point, you'll always get the data of the first object.

NOTA

However, you should be aware of how ridiculously unlikely this scenario is. The SHA-1 digest is 20 bytes or 160 bits. The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is `p = (n(n-1)/2) * (1/2^160))`. 2^{80} is 1.2 x 10^{24} or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.

Here's an example to give you an idea of what it would take to get a SHA-1 collision. If all 6.5 billion humans on Earth were programming, and every second, each one was producing code that was the equivalent of the entire Linux kernel history (3.6 million Git objects) and pushing it into one enormous Git repository, it would take roughly 2 years until that repository contained enough objects to have a 50% probability of a single SHA-1 object collision. A higher probability exists that every member of your programming team will be attacked and killed by wolves in unrelated incidents on the same night.

Branch References

The most straightforward way to specify a commit requires that it has a branch reference pointed at it. Then, you can use a branch name in any Git command that expects a commit object or SHA-1 value. For instance, if you want to show the last commit object on a branch, the following commands are equivalent, assuming that the `topic1` branch points to `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called `rev-parse`. You can see Funcionamento Interno do Git for more information about plumbing tools; basically, `rev-parse` exists for lower-level operations and isn't designed to be used in day-to-day operations. However, it can be helpful sometimes when you need to see what's really going on. Here you can run `rev-parse` on your branch.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

RefLog Shortnames

One of the things Git does in the background while you're working away is keep a "reflog" – a log of where your HEAD and branch references have been for the last few months.

You can see your reflog by using `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Every time your branch tip is updated for any reason, Git stores that information for you in this temporary history. And you can specify older commits with this data, as well. If you want to see the fifth prior value of the HEAD of your repository, you can use the `@{n}` reference that you see in the reflog output:

```
$ git show HEAD@{5}
```

You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to see where your `master` branch was yesterday, you can type

```
$ git show master@{yesterday}
```

That shows you where the branch tip was yesterday. This technique only works for data that's still in your reflog, so you can't use it to look for commits older than a few months.

To see reflog information formatted like the `git log` output, you can run `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

It's important to note that the reflog information is strictly local – it's a log of what you've done in your repository. The references won't be the same on someone else's copy of the repository; and right after you initially clone a repository, you'll have an empty reflog, as no activity has occurred yet in your repository. Running `git show HEAD@{2.months.ago}` will work only if you cloned the project at least two months ago – if you cloned it five minutes ago, you'll get no results.

Ancestry References

The other main way to specify a commit is via its ancestry. If you place a `^` at the end of a reference, Git resolves it to mean the parent of that commit. Suppose you look at the history of your project:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Then, you can see the previous commit by specifying `HEAD^`, which means "the parent of HEAD":

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

You can also specify a number after the ^ – for example, d921970^2 means "the second parent of d921970." This syntax is only useful for merge commits, which have more than one parent. The first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

The other main ancestry specification is the ~. This also refers to the first parent, so HEAD~ and HEAD^ are equivalent. The difference becomes apparent when you specify a number. HEAD~2 means "the first parent of the first parent," or "the grandparent" – it traverses the first parents the number of times you specify. For example, in the history listed earlier, HEAD~3 would be

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

This can also be written HEAD^^^, which again is the first parent of the first parent of the first parent:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

You can also combine these syntaxes – you can get the second parent of the previous reference (assuming it was a merge commit) by using HEAD~3^2, and so on.

Commit Ranges

Now that you can specify individual commits, let's see how to specify ranges of commits. This is particularly useful for managing your branches – if you have a lot of branches, you can use range

specifications to answer questions such as, “What work is on this branch that I haven’t yet merged into my main branch?”

Double Dot

The most common range specification is the double-dot syntax. This basically asks Git to resolve a range of commits that are reachable from one commit but aren’t reachable from another. For example, say you have a commit history that looks like Example history for range selection..

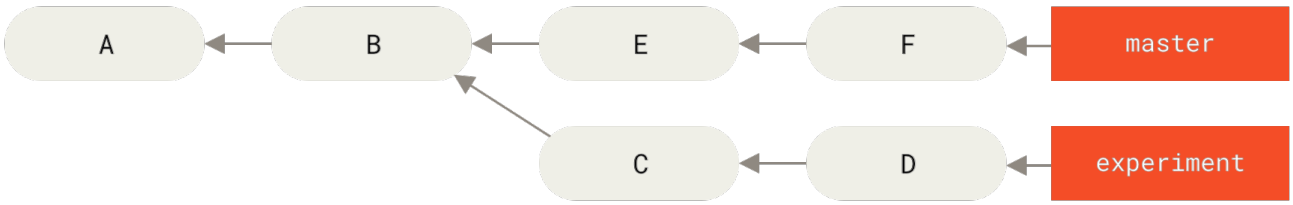


Figura 137. Example history for range selection.

You want to see what is in your experiment branch that hasn’t yet been merged into your master branch. You can ask Git to show you a log of just those commits with `master..experiment` – that means “all commits reachable by experiment that aren’t reachable by master.” For the sake of brevity and clarity in these examples, I’ll use the letters of the commit objects from the diagram in place of the actual log output in the order that they would display:

```
$ git log master..experiment
D
C
```

If, on the other hand, you want to see the opposite – all commits in `master` that aren’t in `experiment` – you can reverse the branch names. `experiment..master` shows you everything in `master` not reachable from `experiment`:

```
$ git log experiment..master
F
E
```

This is useful if you want to keep the `experiment` branch up to date and preview what you’re about to merge in. Another very frequent use of this syntax is to see what you’re about to push to a remote:

```
$ git log origin/master..HEAD
```

This command shows you any commits in your current branch that aren’t in the `master` branch on your `origin` remote. If you run a `git push` and your current branch is tracking `origin/master`, the commits listed by `git log origin/master..HEAD` are the commits that will be transferred to the server. You can also leave off one side of the syntax to have Git assume HEAD. For example, you can get the same results as in the previous example by typing `git log origin/master..` – Git substitutes

HEAD if one side is missing.

Multiple Points

The double-dot syntax is useful as a shorthand; but perhaps you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of several branches that aren't in the branch you're currently on. Git allows you to do this by using either the `^` character or `--not` before any reference from which you don't want to see reachable commits. Thus these three commands are equivalent:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

This is nice because with this syntax you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from `refA` or `refB` but not from `refC`, you can type one of these:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

This makes for a very powerful revision query system that should help you figure out what is in your branches.

Triple Dot

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by either of two references but not by both of them. Look back at the example commit history in Example history for range selection.. If you want to see what is in `master` or `experiment` but not any common references, you can run

```
$ git log master...experiment
F
E
D
C
```

Again, this gives you normal `log` output but shows you only the commit information for those four commits, appearing in the traditional commit date ordering.

A common switch to use with the `log` command in this case is `--left-right`, which shows you which side of the range each commit is in. This helps make the data more useful:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

With these tools, you can much more easily let Git know what commit or commits you want to inspect.

Interactive Staging

Git comes with a couple of scripts that make some command-line tasks easier. Here, you'll look at a few interactive commands that can help you easily craft your commits to include only certain combinations and parts of files. These tools are very helpful if you modify a bunch of files and then decide that you want those changes to be in several focused commits rather than one big messy commit. This way, you can make sure your commits are logically separate changesets and can be easily reviewed by the developers working with you. If you run `git add` with the `-i` or `--interactive` option, Git goes into an interactive shell mode, displaying something like this:

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

You can see that this command shows you a much different view of your staging area – basically the same information you get with `git status` but a bit more succinct and informative. It lists the changes you've staged on the left and unstaged changes on the right.

After this comes a Commands section. Here you can do a number of things, including staging files, unstaging files, staging parts of files, adding untracked files, and seeing diffs of what has been staged.

Staging and Unstaging Files

If you type `2` or `u` at the `What now>` prompt, the script prompts you for which files you want to stage:

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

To stage the TODO and index.html files, you can type the numbers:

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

The * next to each file means the file is selected to be staged. If you press Enter after typing nothing at the Update>> prompt, Git takes anything selected and stages it for you:

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Now you can see that the TODO and index.html files are staged and the simplegit.rb file is still unstaged. If you want to unstage the TODO file at this point, you use the 3 or r (for revert) option:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Looking at your Git status again, you can see that you've unstaged the TODO file:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

To see the diff of what you've staged, you can use the `6` or `d` (for diff) command. It shows you a list of your staged files, and you can select the ones for which you would like to see the staged diff. This is much like specifying `git diff --cached` on the command line:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

With these basic commands, you can use the interactive add mode to deal with your staging area a little more easily.

Staging Patches

It's also possible for Git to stage certain parts of files and not the rest. For example, if you make two changes to your simplegit.rb file and want to stage one of them and not the other, doing so is very easy in Git. From the interactive prompt, type `5` or `p` (for patch). Git will ask you which files you would like to partially stage; then, for each section of the selected files, it will display hunks of the file diff and ask if you would like to stage them, one by one:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

You have a lot of options at this point. Typing `?` shows a list of what you can do:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Generally, you'll type `y` or `n` if you want to stage each hunk, but staging all of them in certain files or skipping a hunk decision until later can be helpful too. If you stage one part of the file and leave another part unstaged, your status output will look like this:

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

The status of the simplegit.rb file is interesting. It shows you that a couple of lines are staged and a couple are unstaged. You've partially staged this file. At this point, you can exit the interactive adding script and run `git commit` to commit the partially staged files.

You also don't need to be in interactive add mode to do the partial-file staging – you can start the same script by using `git add -p` or `git add --patch` on the command line.

Furthermore, you can use patch mode for partially resetting files with the `reset --patch` command, for checking out parts of files with the `checkout --patch` command and for stashing parts of files with the `stash save --patch` command. We'll go into more details on each of these as we get to more advanced usages of these commands.

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Stashing Your Work

To demonstrate, you'll go into your project and start working on a couple of files and possibly stage one of the changes. If you run `git status`, you can see your dirty state:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb
```

Now you want to switch branches, but you don't want to commit what you've been working on yet; so you'll stash the changes. To push a new stash onto your stack, run `git stash` or `git stash save`:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Your working directory is clean:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

At this point, you can easily switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, two stashes were done previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. If you want to apply one of the older stashes, you can specify it by naming it, like this: `git stash apply stash@{2}`. If you don't specify a stash, Git assumes the most recent stash and tries to apply it:

```
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html
    modified:   lib/simplegit.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

You can see that Git re-modifies the files you reverted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from; but having a clean working directory and applying it on the same branch aren't necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash – Git gives you merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but the file you staged before wasn't restaged. To do that, you must run the `git stash apply` command with a `--index` option to tell the command to try to reapply the staged changes. If you had run that instead, you'd have gotten back to your original position:

```
$ git stash apply --index
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb
```

The apply option only tries to apply the stashed work – you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the `--keep-index` option to the `stash save` command. This tells Git to not stash anything that you've already staged with the `git add` command.

This can be really helpful if you've made a number of changes but want to only commit some of them and then come back to the rest of the changes at a later time.

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will only store files that are already in the index. If you specify `--include-untracked` or `-u`, Git will also stash any untracked files you have created.

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Finally, if you specify the `--patch` flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
       end
     end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you've since modified, you'll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run `git stash branch`, which creates a new branch for you, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

```
$ git stash branch testchanges
M   index.html
M   lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

Cleaning your Working Directory

Finally, you may not want to stash some work or files in your working directory, but simply get rid of them. The `git clean` command will do this for you.

Some common reasons for this might be to remove cruft that has been generated by merges or external tools or to remove build artifacts in order to run a clean build.

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with `git clean`. To remove all the untracked files in your working directory, you can run `git clean -f -d`, which removes any files and also any subdirectories that become empty as a result. The `-f` means *force* or "really do this".

If you ever want to see what it would do, you can run the command with the `-n` option, which means "do a dry run and tell me what you *would* have removed".

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

By default, the `git clean` command will only remove untracked files that are not ignored. Any file that matches a pattern in your `.gitignore` or other ignore files will not be removed. If you want to remove those files too, such as to remove all `.o` files generated from a build so you can do a fully clean build, you can add a `-x` to the clean command.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

If you don't know what the `git clean` command is going to do, always run it with a `-n` first to double check before changing the `-n` to a `-f` and doing it for real. The other way you can be careful about the process is to run it with the `-i` or "interactive" flag.

This will run the clean command in an interactive mode.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask
each             5: quit
    6: help
What now>
```

This way you can step through each file individually or specify patterns for deletion interactively.

Signing Your Work

Git is cryptographically secure, but it's not foolproof. If you're taking work from others on the internet and want to verify that commits are actually from a trusted source, Git has a few ways to sign and verify work using GPG.

GPG Introduction

First of all, if you want to sign anything you need to get GPG configured and your personal key installed.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

If you don't have a key installed, you can generate one with `gpg --gen-key`.

```
gpg --gen-key
```

Once you have a private key to sign with, you can configure Git to use it for signing things by setting the `user.signingkey` config setting.

```
git config --global user.signingkey 0A46826A
```

Now Git will use your key by default to sign tags and commits if you want.

Signing Tags

If you have a GPG private key setup, you can now use it to sign new tags. All you have to do is use `-s`

instead of `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

If you run `git show` on that tag, you can see your GPG signature attached to it:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Verifying Tags

To verify a signed tag, you use `git tag -v [tag-name]`. This command uses GPG to verify the signature. You need the signer's public key in your keyring for this to work properly:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

If you don't have the signer's public key, you get something like this instead:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Signing Commits

In more recent versions of Git (v1.7.9 and above), you can now also sign individual commits. If you're interested in signing commits directly instead of just the tags, all you need to do is add a `-S` to your `git commit` command.

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

To see and verify these signatures, there is also a `--show-signature` option to `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    signed commit
```

Additionally, you can configure `git log` to check any signatures it finds and list them in its output with the `%G?` format.

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

Here we can see that only the latest commit is signed and valid and the previous commits are not.

In Git 1.8.3 and later, "git merge" and "git pull" can be told to inspect and reject when merging a commit that does not carry a trusted GPG signature with the `--verify-signatures` command.

If you use this option when merging a branch and it contains commits that are not signed and valid, the merge will not work.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

If the merge contains only valid signed commits, the merge command will show you all the signatures it has checked and then move forward with the merge.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

You can also use the `-S` option with the `git merge` command itself to sign the resulting merge commit itself. The following example both verifies that every commit in the branch to be merged is signed and furthermore signs the resulting merge commit.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Everyone Must Sign

Signing tags and commits is great, but if you decide to use this in your normal workflow, you'll have to make sure that everyone on your team understands how to do so. If you don't, you'll end up spending a lot of time helping people figure out how to rewrite their commits with signed versions. Make sure you understand GPG and the benefits of signing things before adopting this as part of your standard workflow.

Searching

With just about any size codebase, you'll often need to find where a function is called or defined, or find the history of a method. Git provides a couple of useful tools for looking through the code and commits stored in its database quickly and easily. We'll go through a few of them.

Git Grep

Git ships with a command called `grep` that allows you to easily search through any committed tree or the working directory for a string or regular expression. For these examples, we'll look through the Git source code itself.

By default, it will look through the files in your working directory. You can pass `-n` to print out the line numbers where Git has found matches.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:             if (gmtime_r(&now, &now_tm))
date.c:492:             if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

There are a number of interesting options you can provide the `grep` command.

For instance, instead of the previous call, you can have Git summarize the output by just showing you which files matched and how many matches there were in each file with the `--count` option:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

If you want to see what method or function it thinks it has found a match in, you can pass `-p`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date, char
*end, struct tm *tm)
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int
*tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
```

So here we can see that `gmtime_r` is called in the `match_multi_number` and `match_digit` functions in the date.c file.

You can also look for complex combinations of strings with the `--and` flag, which makes sure that multiple matches are in the same line. For instance, let's look for any lines that define a constant with either the strings "LINK" or "BUF_MAX" in them in the Git codebase in an older 1.8.0 version.

Here we'll also use the `--break` and `--heading` options which help split up the output into a more readable format.

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)       (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

The `git grep` command has a few advantages over normal searching commands like `grep` and `ack`. The first is that it's really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

Git Log Searching

Perhaps you're looking not for **where** a term exists, but **when** it existed or was introduced. The `git log` command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If we want to find out for example when the `ZLIB_BUF_MAX` constant was originally introduced, we can tell Git to only show us the commits that either added or removed that string with the `-S` option.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

If we look at the diff of those commits we can see that in `ef49a7a` the constant was introduced and in `e01503b` it was modified.

If you need to be more specific, you can provide a regular expression to search for with the `-G` option.

Line Log Search

Another fairly advanced log search that is insanely useful is the line history search. This is a fairly recent addition and not very well known, but it can be really helpful. It is called with the `-L` option to `git log` and will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function `git_deflate_bound` in the `zlib.c` file, we could run `git log -L :git_deflate_bound:zlib.c`. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

If Git can't figure out how to match a function or method in your programming language, you can also provide it a regex. For example, this would have done the same thing: `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`. You could also give it a range of lines or a single line number and you'll get the same sort of output.

Rewriting History

Many times, when working with Git, you may want to revise your commit history for some reason. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn't mean to be working on something yet with the stash command, and you can rewrite commits that already happened so they look like they happened in a different way. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely – all before you share your work with others.

In this section, you'll cover how to accomplish these very useful tasks so that you can make your commit history look the way you want before you share it with others.

Changing the Last Commit

Changing your last commit is probably the most common rewriting of history that you'll do. You'll often want to do two basic things to your last commit: change the commit message, or change the snapshot you just recorded by adding, changing and removing files.

If you only want to modify your last commit message, it's very simple:

```
$ git commit --amend
```

That drops you into your text editor, which has your last commit message in it, ready for you to modify the message. When you save and close the editor, the editor writes a new commit containing that message and makes it your new last commit.

If you've committed and then you want to change the snapshot you committed by adding or changing files, possibly because you forgot to add a newly created file when you originally committed, the process works basically the same way. You stage the changes you want by editing a file and running `git add` on it or `git rm` to a tracked file, and the subsequent `git commit --amend` takes your current staging area and makes it the snapshot for the new commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It's like a very small rebase – don't amend your last commit if you've already pushed it.

Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the `-i` option to `git rebase`. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages

in that group, you supply as an argument to `git rebase -i` the parent of the last commit you want to edit, which is `HEAD~2^` or `HEAD~3`. It may be easier to remember the `~3` because you're trying to edit the last three commits; but keep in mind that you're actually designating four commits ago, the parent of the last commit you want to edit:

```
$ git rebase -i HEAD~3
```

Remember again that this is a rebasing command – every commit included in the range `HEAD~3..HEAD` will be rewritten, whether you change the message or not. Don't include any commit you've already pushed to a central server – doing so will confuse other developers by providing an alternate version of the same change.

Running this command gives you a list of commits in your text editor that looks something like this:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

It's important to note that these commits are listed in the opposite order than you normally see them using the `log` command. If you run a `log`, you see something like this:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Notice the reverse order. The interactive rebase gives you a script that it's going to run. It will start at the commit you specify on the command line (`HEAD~3`) and replay the changes introduced in each of these commits from top to bottom. It lists the oldest at the top, rather than the newest, because

that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word 'pick' to the word 'edit' for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

These instructions tell you exactly what to do. Type

```
$ git commit --amend
```

Change the commit message, and exit the editor. Then, run

```
$ git rebase --continue
```

This command will apply the other two commits automatically, and then you're done. If you change pick to edit on more lines, you can repeat these steps for each commit you change to edit. Each time, Git will stop, let you amend the commit, and continue when you're finished.

Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the "added cat-file" commit and change the order in which the other two commits are introduced, you can change the rebase script from this

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

to this:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies `310154e` and then `f7f3f6d`, and then stops. You effectively change the order of those commits and remove the “added cat-file” commit completely.

Squashing Commits

It’s also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

If, instead of “pick” or “edit”, you specify “squash”, Git applies both that change and the change directly before it and makes you merge the commit messages together. So, if you want to make a single commit from these three commits, you make the script look like this:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

When you save that, you have a single commit that introduces the changes of all three previous commits.

Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of "updated README formatting and added blame", you want to split it into two commits: "updated README formatting" for the first, and "added blame" for the second. You can do that in the `rebase -i` script by changing the instruction on the commit you want to split to "edit":

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Then, when the script drops you to the command line, you reset that commit, take the changes that have been reset, and create multiple commits out of them. When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (`f7f3f6d`), applies the second (`310154e`), and drops you to the console. There, you can do a mixed reset of that commit with `git reset HEAD^`, which effectively undoes that commit and leaves the modified files unstaged. Now you can stage and commit files until you have several commits, and run `git rebase --continue` when you're done:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git applies the last commit (`a5f4a0d`) in the script, and your history looks like this:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Once again, this changes the SHA-1s of all the commits in your list, so make sure no commit shows up in that list that you've already pushed to a shared repository.

The Nuclear Option: filter-branch

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way – for instance, changing your email address globally or removing a file from every commit. The command is `filter-branch`, and it can rewrite huge swaths of your history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses so you can get an idea of some of the things it's capable of.

Removing a File from Every Commit

This occurs fairly commonly. Someone accidentally commits a huge binary file with a thoughtless `git add .`, and you want to remove it everywhere. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `filter-branch` is the tool you probably want to use to scrub your entire history. To remove a file named `passwords.txt` from your entire history, you can use the `--tree-filter` option to `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called `passwords.txt` from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like `git filter-branch --tree-filter 'rm -f *~' HEAD`.

You'll be able to watch Git rewriting trees and commits and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your master branch after you've determined the outcome is what you really want. To run `filter-branch` on all your branches, you can pass `--all` to the command.

Making a Subdirectory the New Root

Suppose you've done an import from another source control system and have subdirectories that make no sense (`trunk`, `tags`, and so on). If you want to make the `trunk` subdirectory be the new project root for every commit, `filter-branch` can help you do that, too:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory each time. Git will also automatically remove commits that did not affect the subdirectory.

Changing Email Addresses Globally

Another common case is that you forgot to run `git config` to set your name and email address before you started working, or perhaps you want to open-source a project at work and change all your work email addresses to your personal address. In any case, you can change email addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the email addresses that are yours, so you use `--commit-filter`:

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA-1 in your history, not just those that have the matching email address.

Reset Demystified

Before moving on to more specialized tools, let's talk about `reset` and `checkout`. These commands are two of the most confusing parts of Git when you first encounter them. They do so many things, that it seems hopeless to actually understand them and employ them properly. For this, we recommend a simple metaphor.

The Three Trees

An easier way to think about `reset` and `checkout` is through the mental frame of Git being a content manager of three different trees. By "tree" here we really mean "collection of files", not specifically the data structure. (There are a few cases where the index doesn't exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

Tree	Role
HEAD	Last commit snapshot, next parent
Index	Proposed next commit snapshot
Working Directory	Sandbox

The HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It's generally simplest to think of HEAD as the snapshot of **your last commit**.

In fact, it's pretty easy to see what that snapshot looks like. Here is an example of getting the actual directory listing and SHA-1 checksums for each file in the HEAD snapshot:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

The `cat-file` and `ls-tree` commands are "plumbing" commands that are used for lower level things and not really used in day-to-day work, but they help us see what's going on here.

The Index

The Index is your **proposed next commit**. We've also been referring to this concept as Git's "Staging Area" as this is what Git looks at when you run `git commit`.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and `git commit` converts that into the tree for a new commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0   README
100644 8f94139338f9404f26296befa88755fc2598c289 0   Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0   lib/simplegit.rb
```

Again, here we're using `ls-files`, which is more of a behind the scenes command that shows you what your index currently looks like.

The index is not technically a tree structure – it's actually implemented as a flattened manifest – but for our purposes it's close enough.

The Working Directory

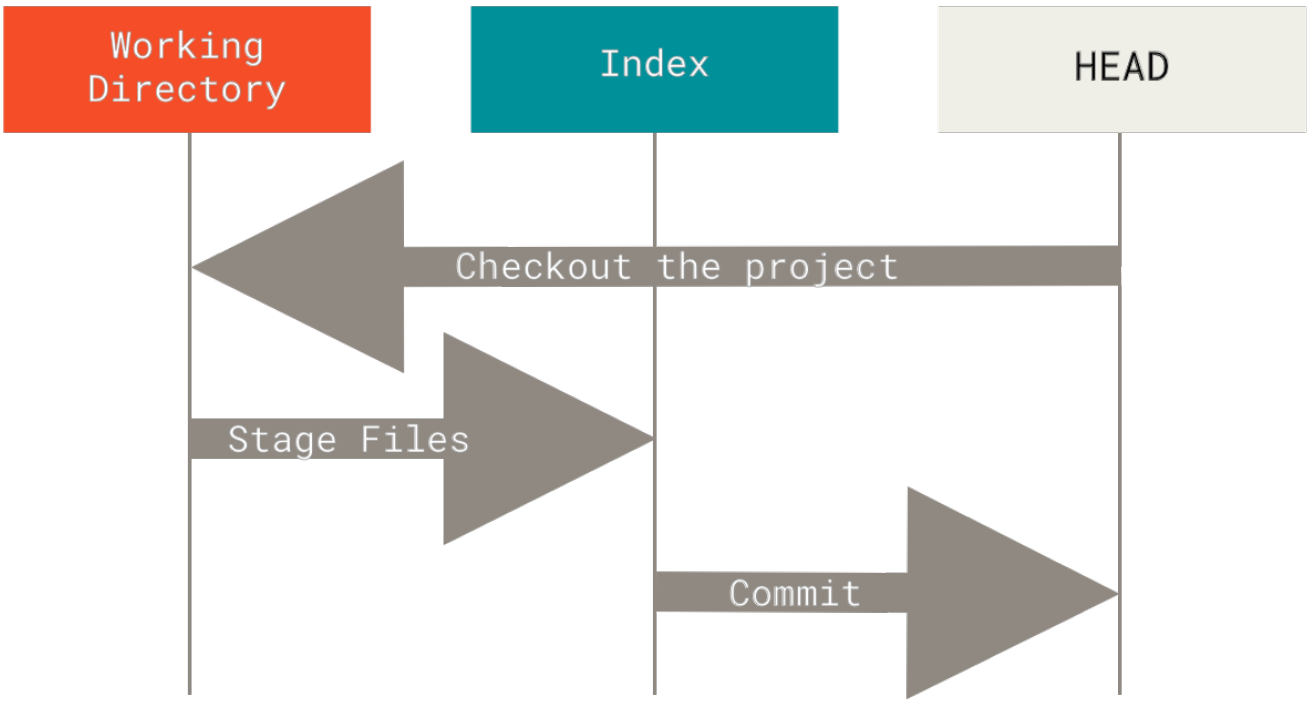
Finally, you have your working directory. The other two trees store their content in an efficient but inconvenient manner, inside the `.git` folder. The Working Directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the Working Directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

```
$ tree
.
├──── README
├──── Rakefile
└──── lib
    └──── simplegit.rb

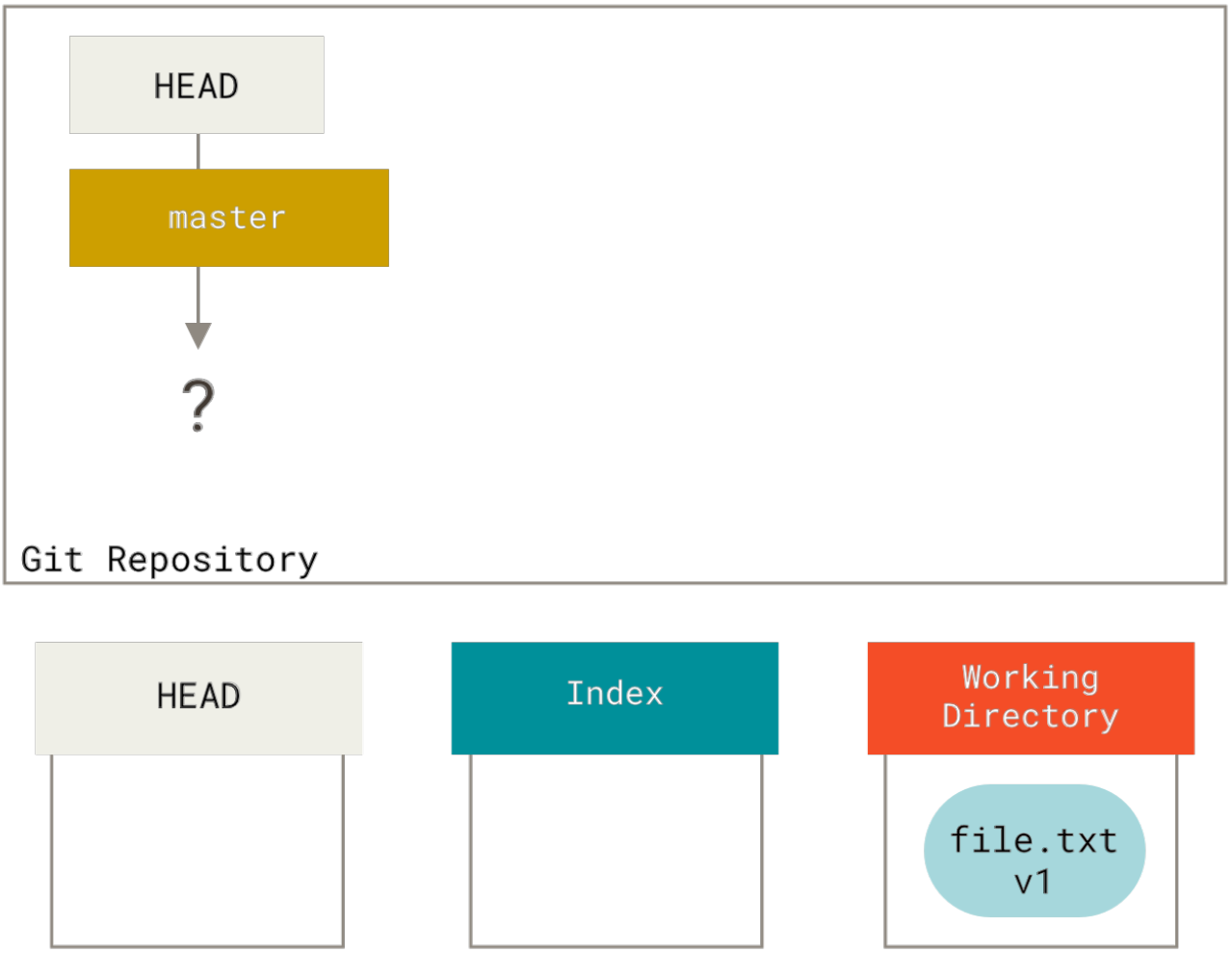
1 directory, 3 files
```

The Workflow

Git's main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.

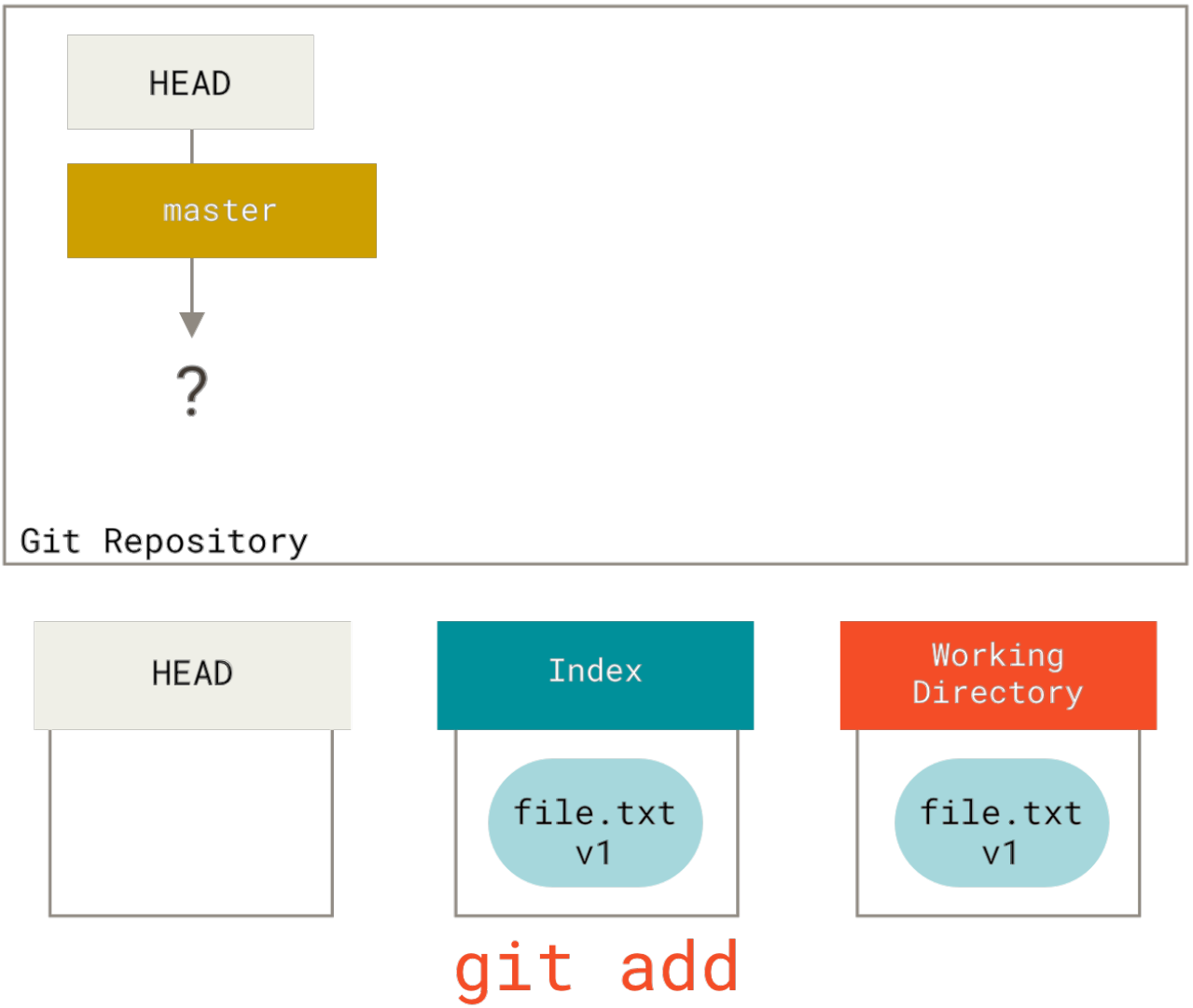


Let's visualize this process: say you go into a new directory with a single file in it. We'll call this **v1** of the file, and we'll indicate it in blue. Now we run `git init`, which will create a Git repository with a HEAD reference which points to an unborn branch (`master` doesn't exist yet).

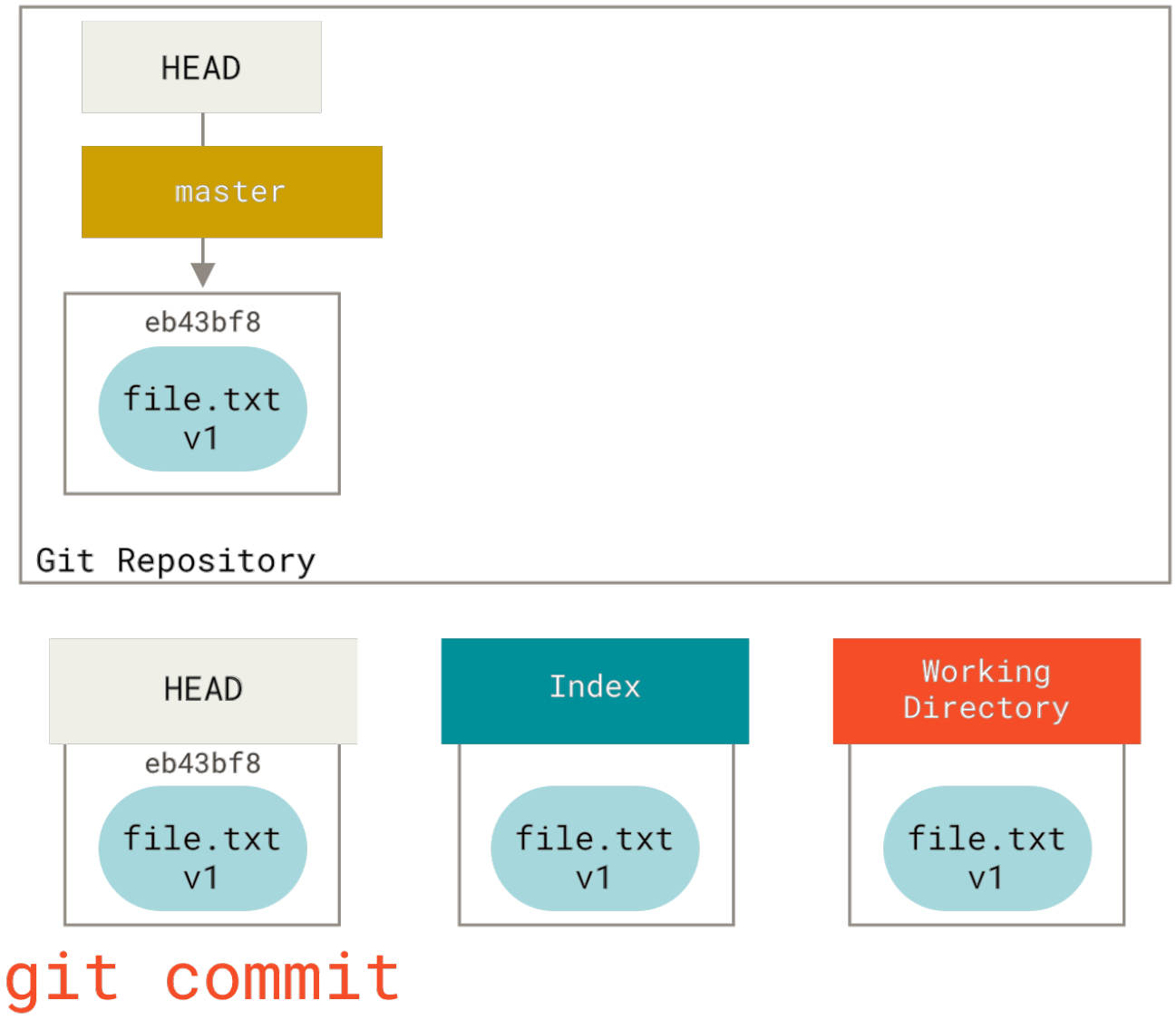


At this point, only the Working Directory tree has any content.

Now we want to commit this file, so we use `git add` to take content in the Working Directory and copy it to the Index.

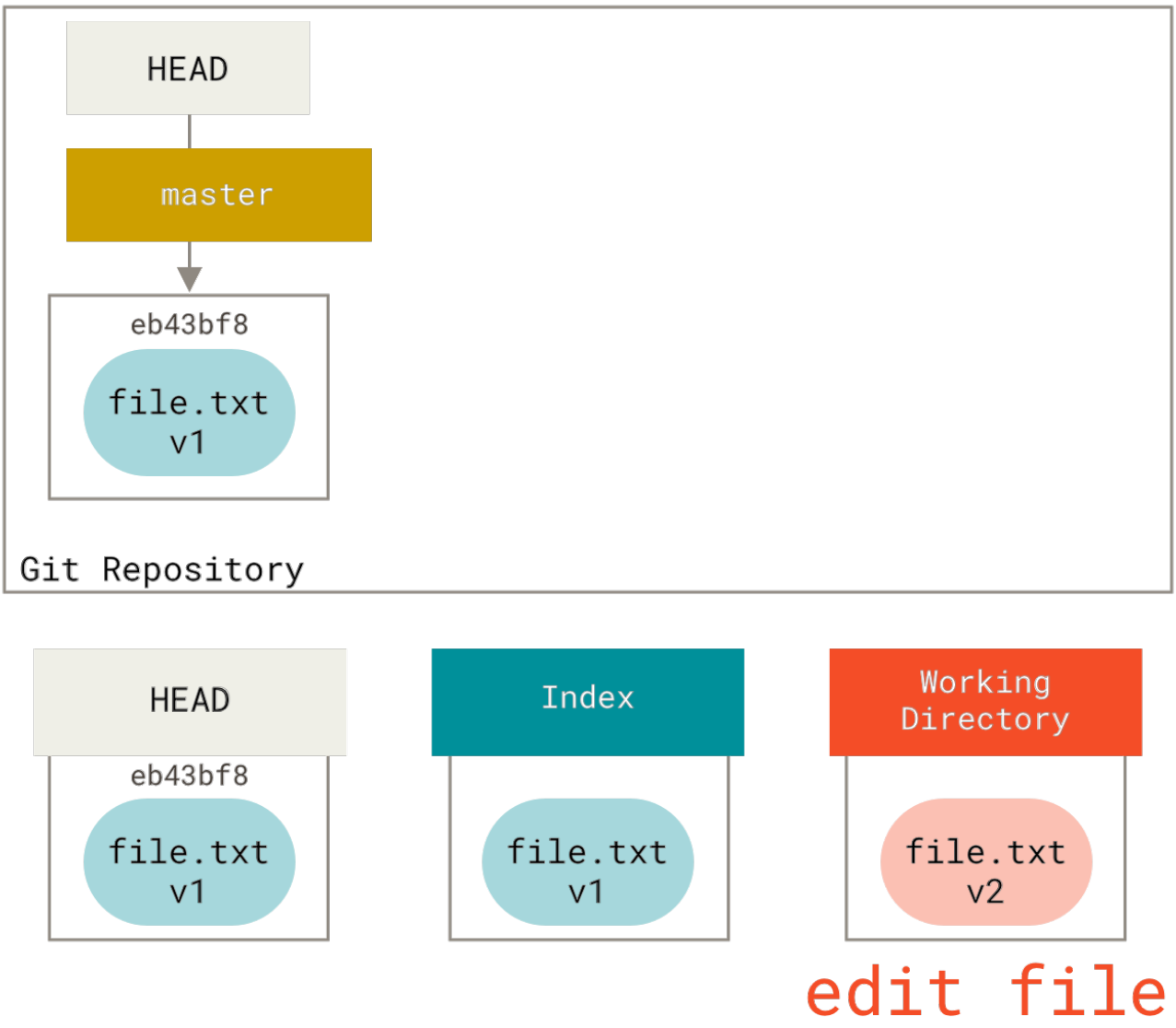


Then we run `git commit`, which takes the contents of the Index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates `master` to point to that commit.

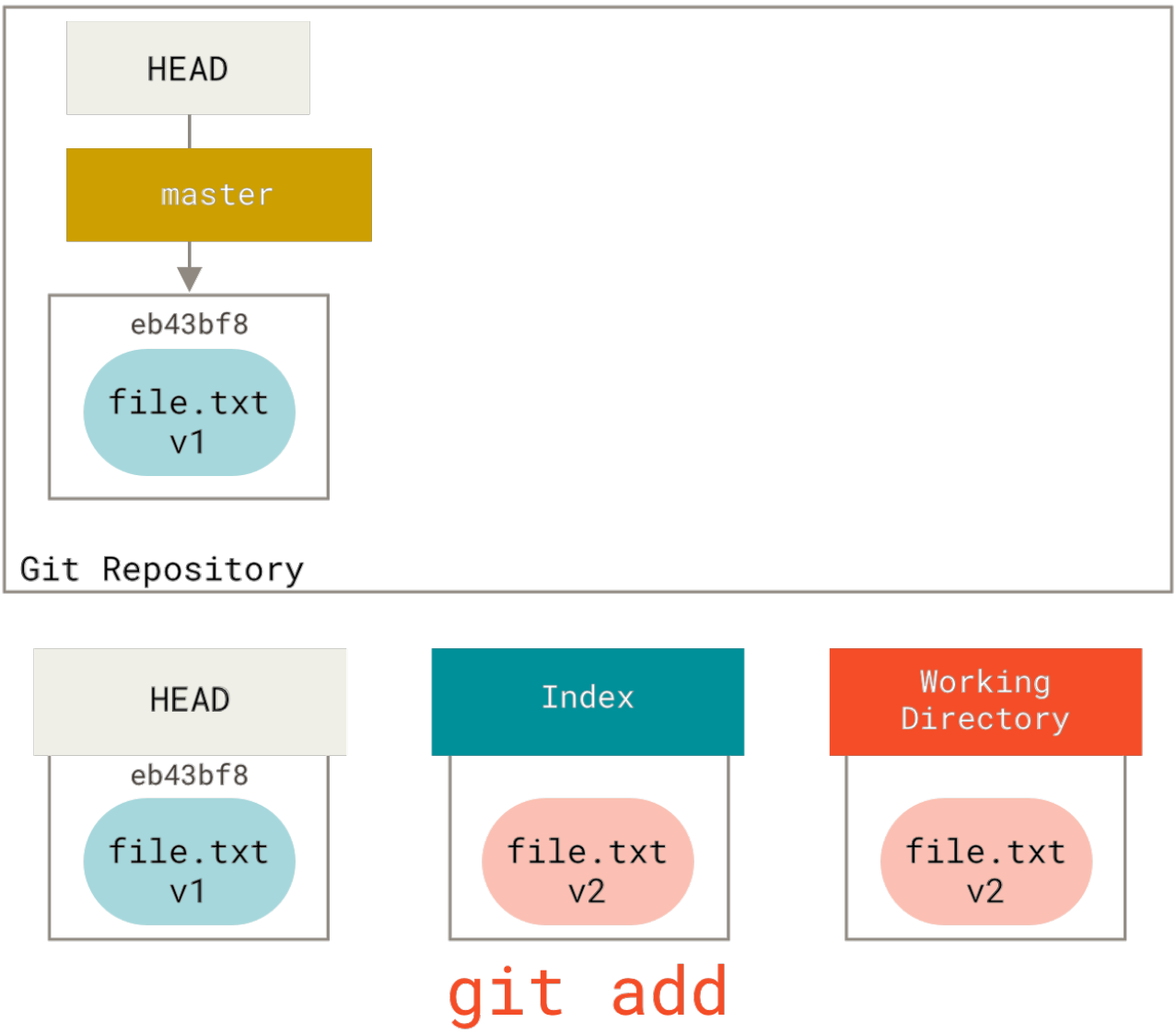


If we run `git status`, we'll see no changes, because all three trees are the same.

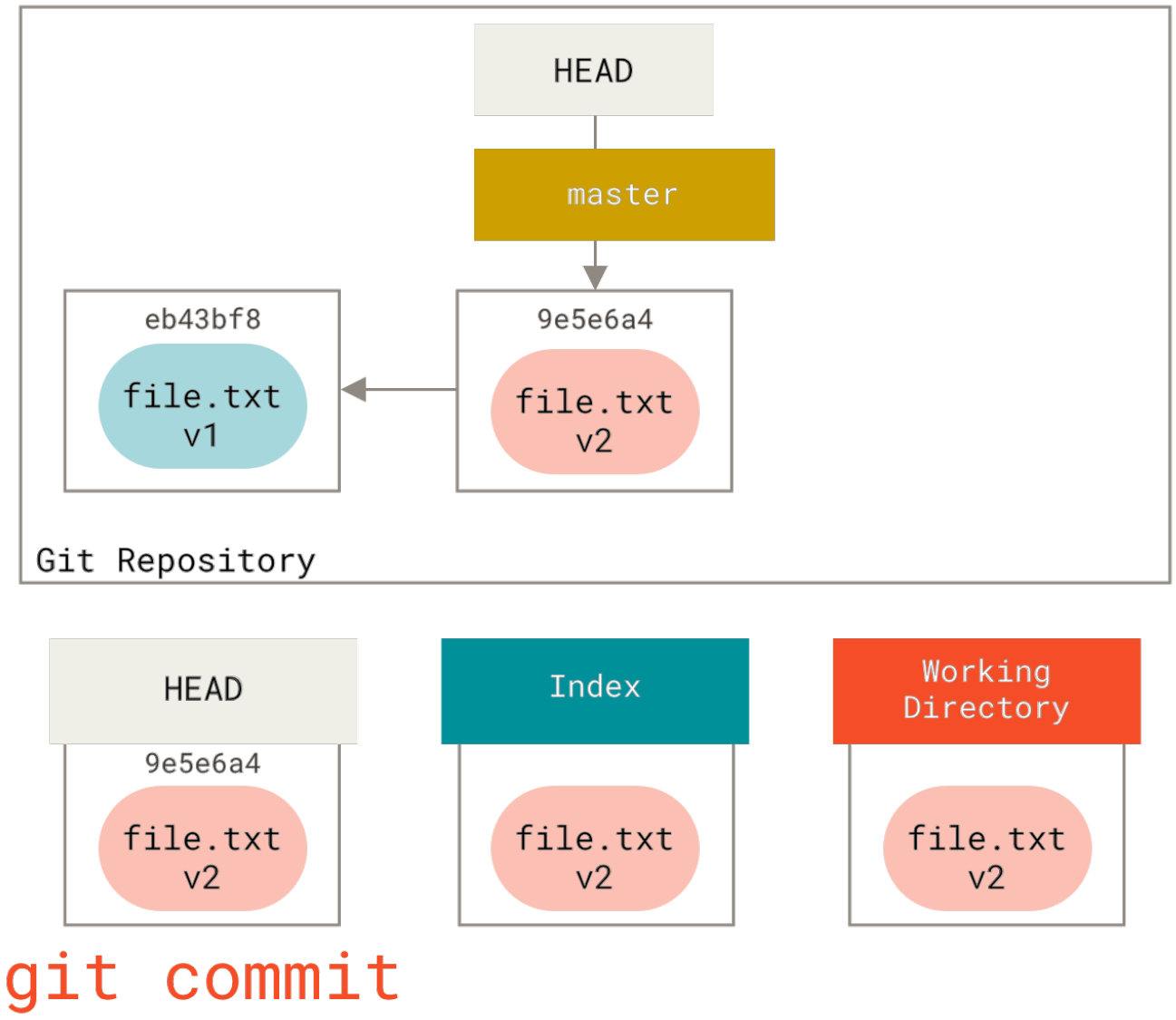
Now we want to make a change to that file and commit it. We'll go through the same process; first we change the file in our working directory. Let's call this **v2** of the file, and indicate it in red.



If we run `git status` right now, we'll see the file in red as "Changes not staged for commit," because that entry differs between the Index and the Working Directory. Next we run `git add` on it to stage it into our Index.



At this point if we run `git status` we will see the file in green under “Changes to be committed” because the Index and HEAD differ – that is, our proposed next commit is now different from our last commit. Finally, we run `git commit` to finalize the commit.



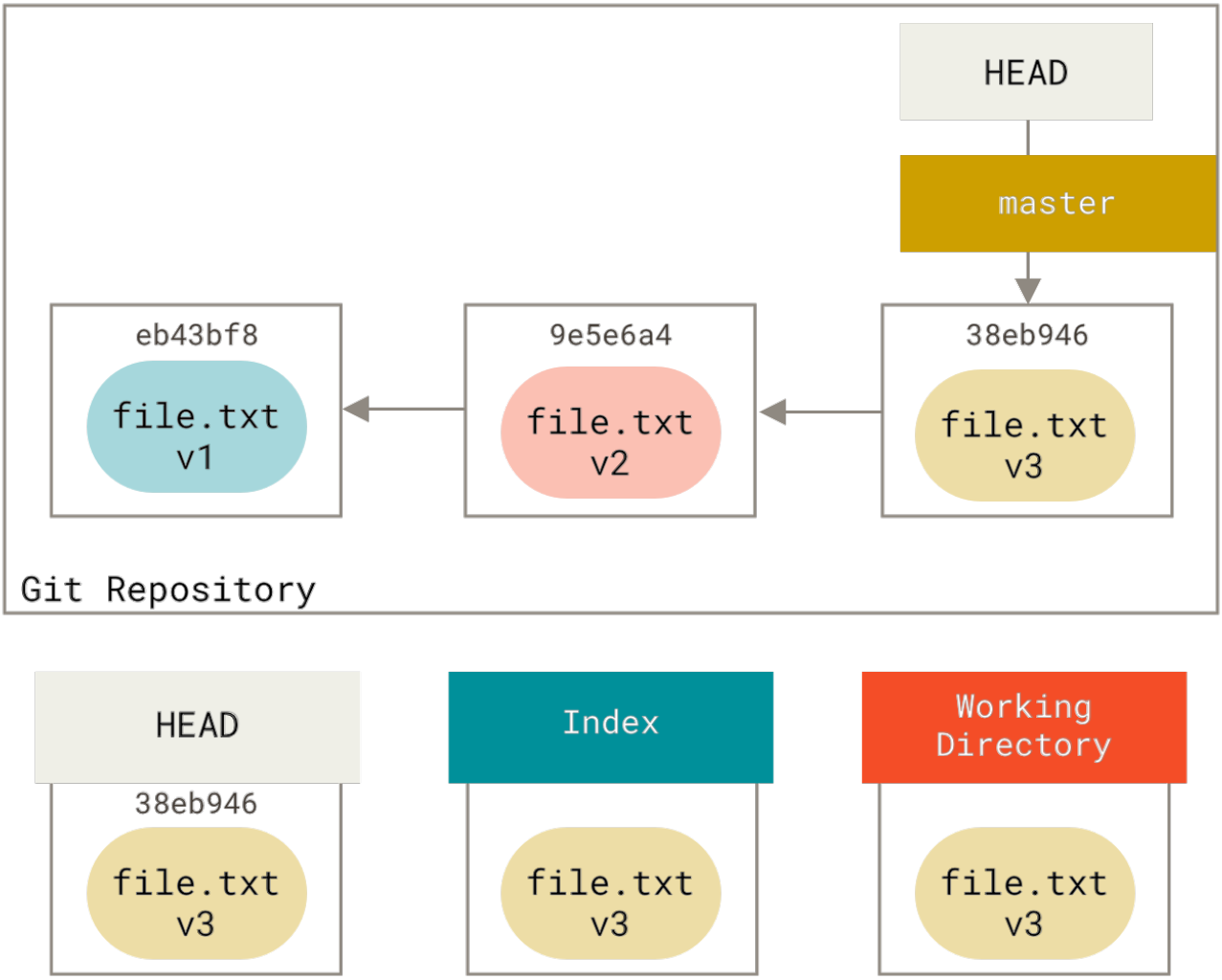
Now `git status` will give us no output, because all three trees are the same again.

Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **Index** with the snapshot of that commit, then copies the contents of the **Index** into your **Working Directory**.

The Role of Reset

The `reset` command makes more sense when viewed in this context.

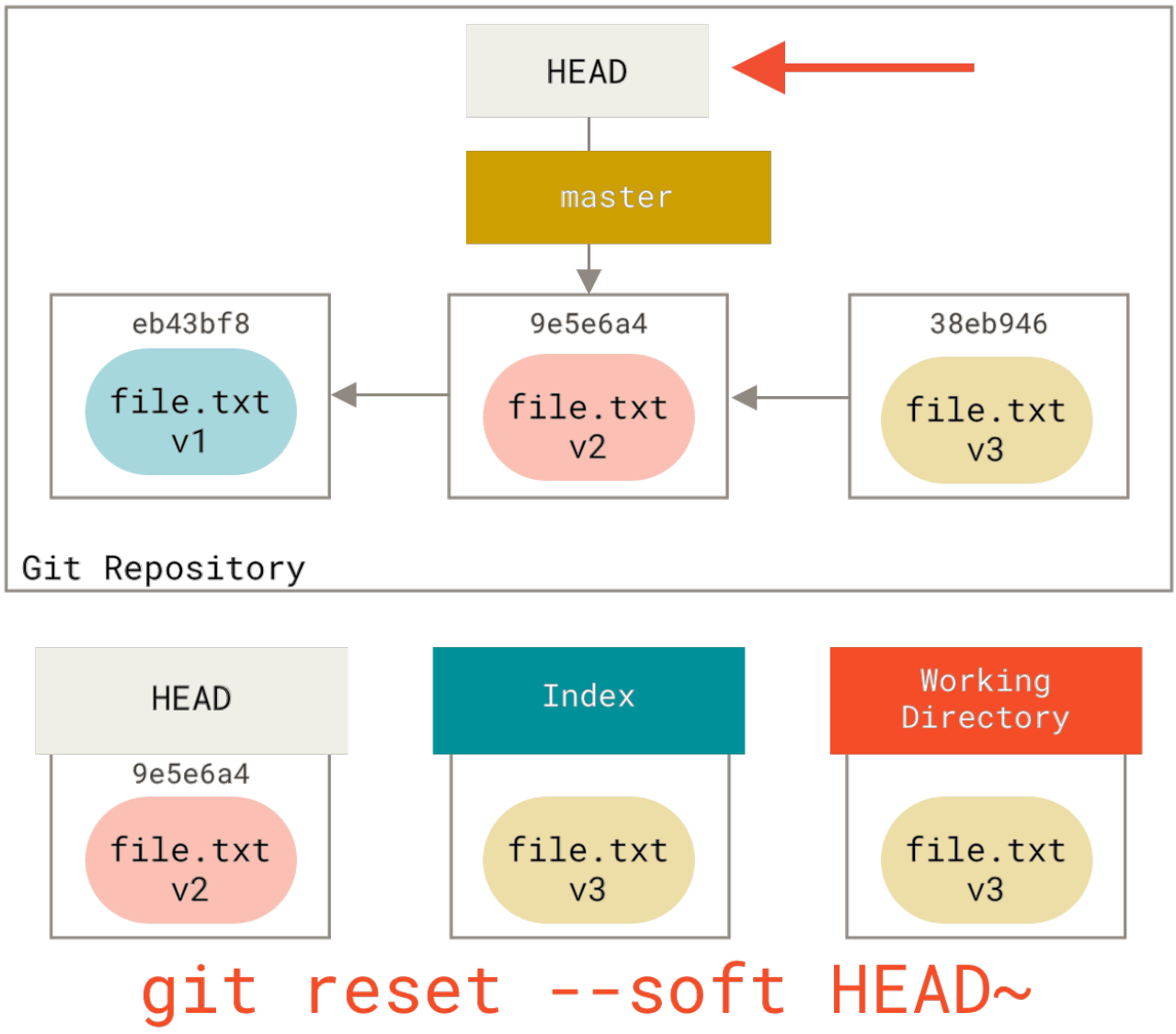
For the purposes of these examples, let's say that we've modified `file.txt` again and committed it a third time. So now our history looks like this:



Let's now walk through exactly what `reset` does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

Step 1: Move HEAD

The first thing `reset` will do is move what HEAD points to. This isn't the same as changing HEAD itself (which is what `checkout` does); `reset` moves the branch that HEAD is pointing to. This means if HEAD is set to the `master` branch (i.e. you're currently on the `master` branch), running `git reset 9e5e6a4` will start by making `master` point to `9e5e6a4`.



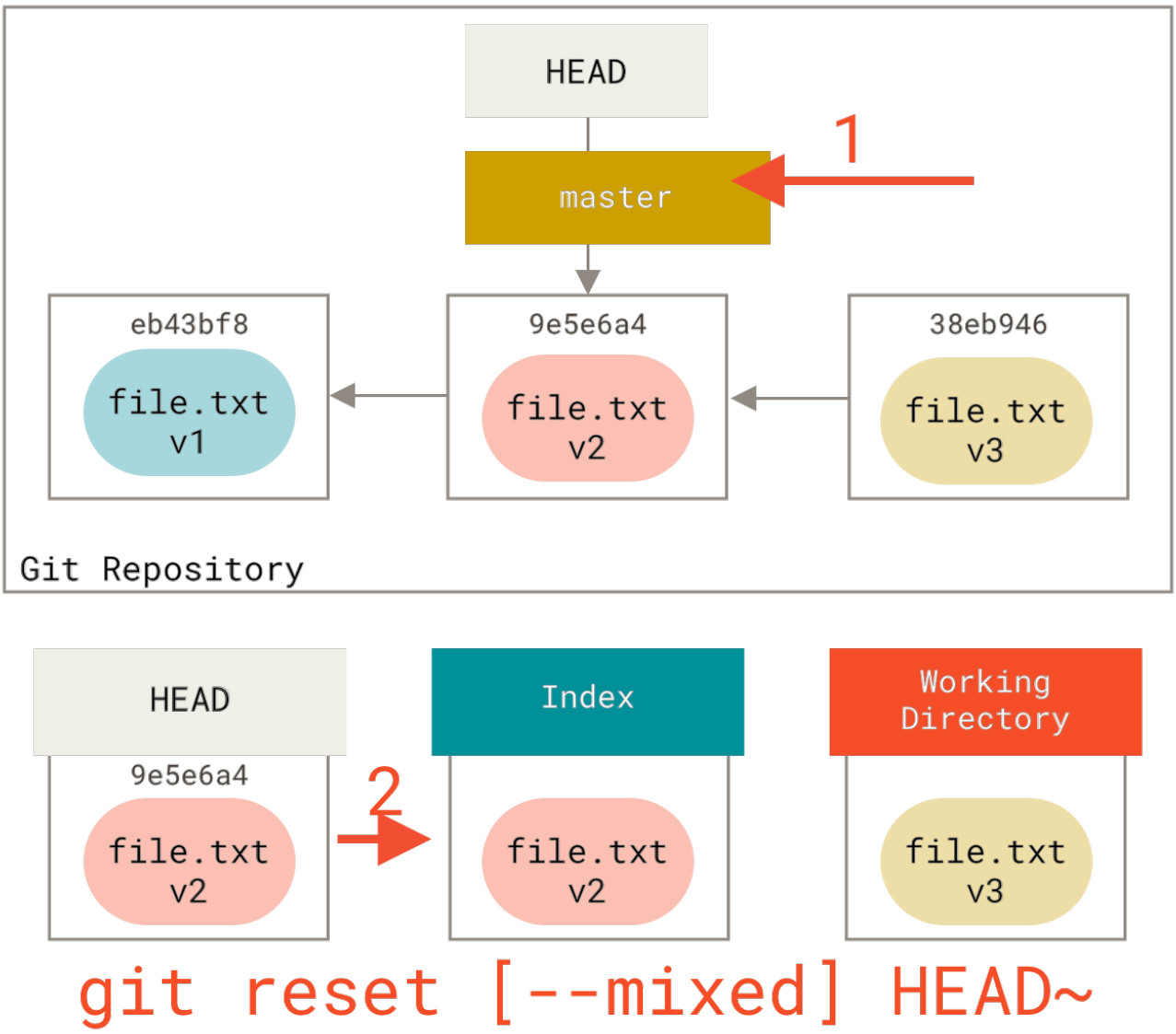
No matter what form of `reset` with a commit you invoke, this is the first thing it will always try to do. With `reset --soft`, it will simply stop there.

Now take a second to look at that diagram and realize what happened: it essentially undid the last `git commit` command. When you run `git commit`, Git creates a new commit and moves the branch that HEAD points to up to it. When you `reset` back to `HEAD~` (the parent of HEAD), you are moving the branch back to where it was, without changing the Index or Working Directory. You could now update the Index and run `git commit` again to accomplish what `git commit --amend` would have done (see Changing the Last Commit).

Step 2: Updating the Index (--mixed)

Note that if you run `git status` now you'll see in green the difference between the Index and what the new HEAD is.

The next thing `reset` will do is to update the Index with the contents of whatever snapshot HEAD now points to.

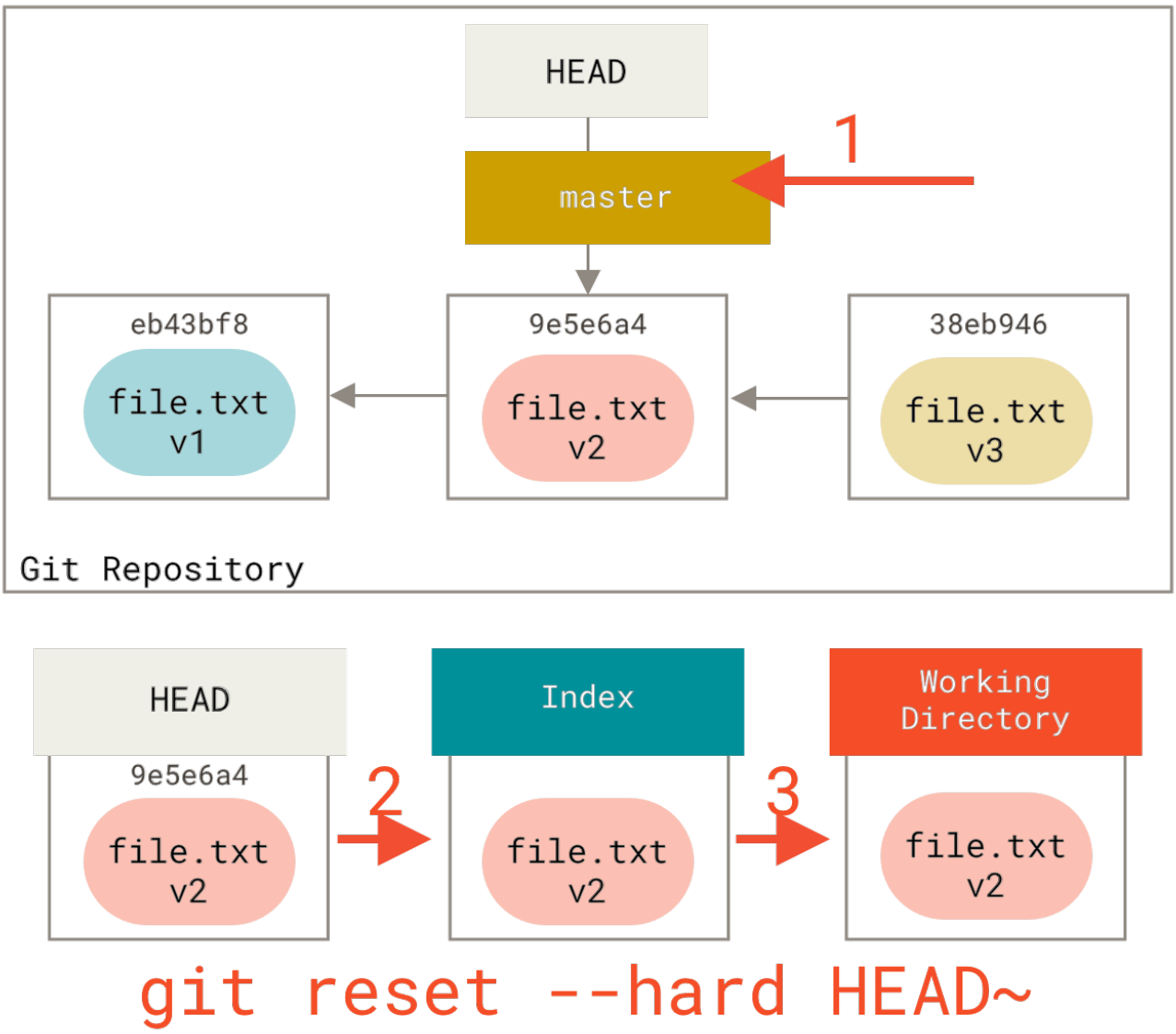


If you specify the `--mixed` option, `reset` will stop at this point. This is also the default, so if you specify no option at all (just `git reset HEAD~` in this case), this is where the command will stop.

Now take another second to look at that diagram and realize what happened: it still undid your last `commit`, but also *unstaged* everything. You rolled back to before you ran all your `git add` and `git commit` commands.

Step 3: Updating the Working Directory (--hard)

The third thing that `reset` will do is to make the Working Directory look like the Index. If you use the `--hard` option, it will continue to this stage.



So let's think about what just happened. You undid your last commit, the `git add` and `git commit` commands, **and** all the work you did in your working directory.

It's important to note that this flag (`--hard`) is the only way to make the `reset` command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of `reset` can be pretty easily undone, but the `--hard` option cannot, since it forcibly overwrites files in the Working Directory. In this particular case, we still have the **v3** version of our file in a commit in our Git DB, and we could get it back by looking at our `reflog`, but if we had not committed it, Git still would have overwritten the file and it would be unrecoverable.

Recap

The `reset` command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch HEAD points to *(stop here if `--soft`)*
2. Make the Index look like HEAD *(stop here unless `--hard`)*
3. Make the Working Directory look like the Index

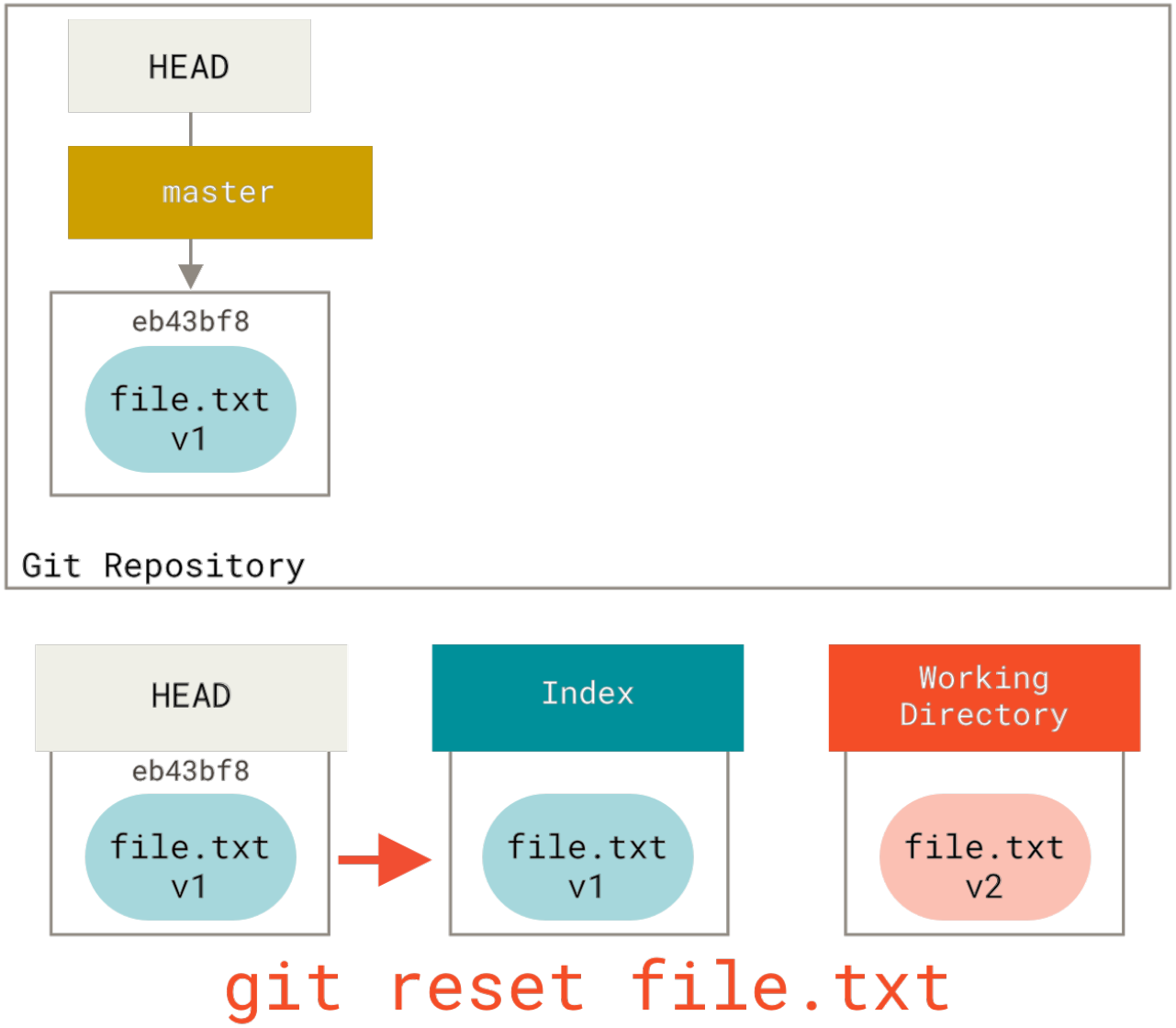
Reset With a Path

That covers the behavior of `reset` in its basic form, but you can also provide it with a path to act upon. If you specify a path, `reset` will skip step 1, and limit the remainder of its actions to a specific file or set of files. This actually sort of makes sense – HEAD is just a pointer, and you can't point to part of one commit and part of another. But the Index and Working directory *can* be partially updated, so reset proceeds with steps 2 and 3.

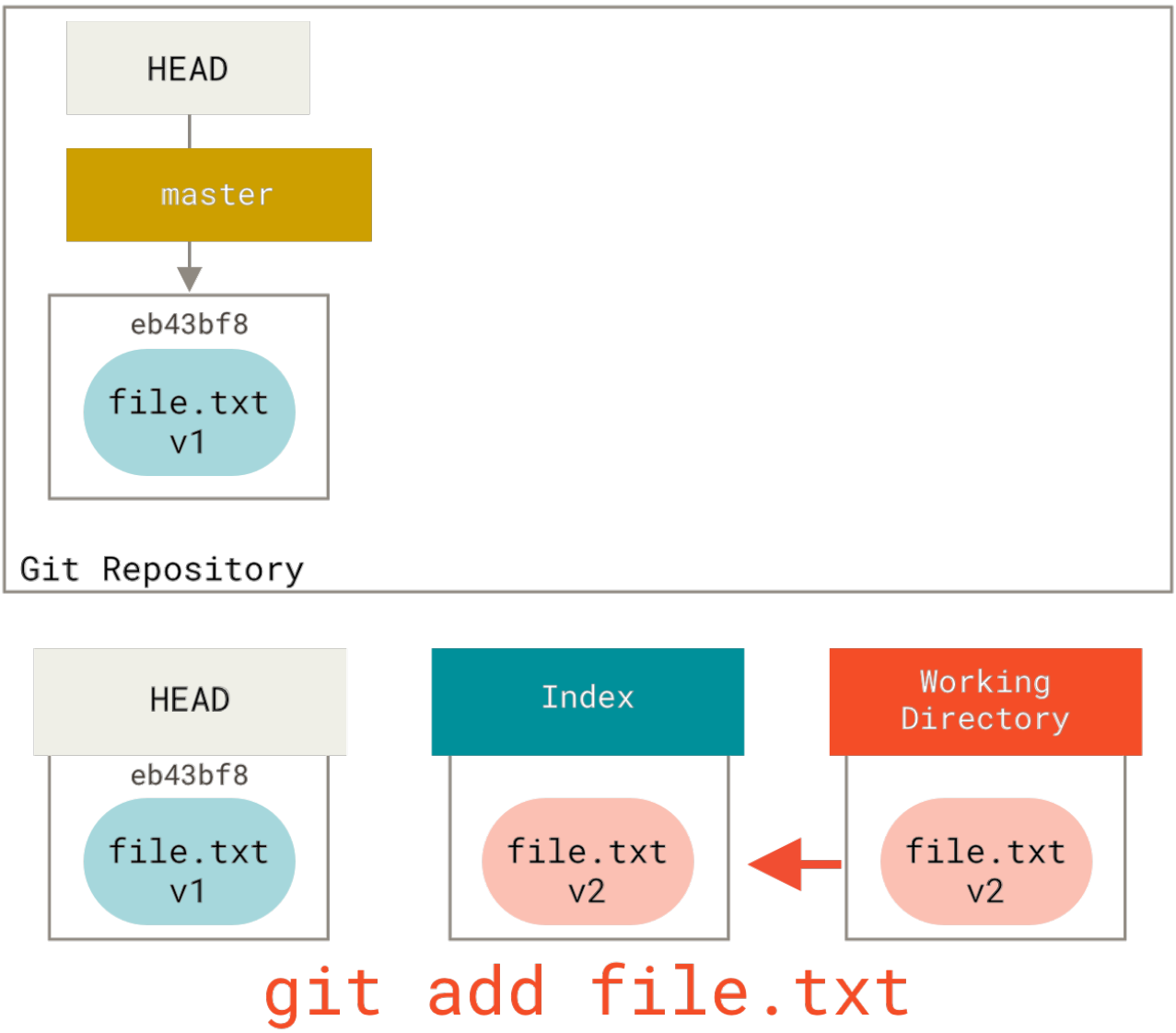
So, assume we run `git reset file.txt`. This form (since you did not specify a commit SHA-1 or branch, and you didn't specify `--soft` or `--hard`) is shorthand for `git reset --mixed HEAD file.txt`, which will:

1. Move the branch HEAD points to *(skipped)*
2. Make the Index look like HEAD *(stop here)*

So it essentially just copies `file.txt` from HEAD to the Index.

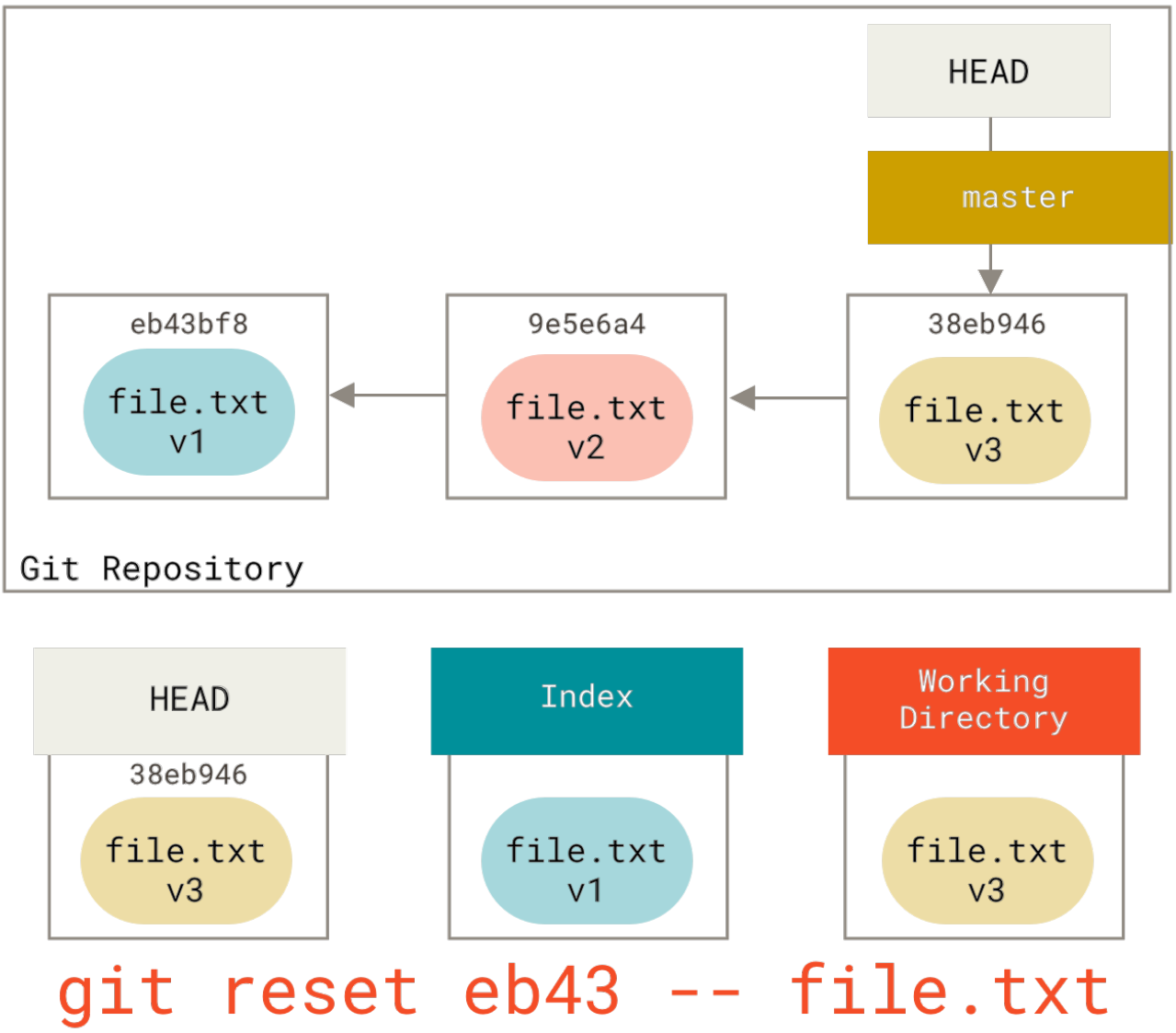


This has the practical effect of *unstaging* the file. If we look at the diagram for that command and think about what `git add` does, they are exact opposites.



This is why the output of the `git status` command suggests that you run this to unstage a file. (See Retirando um arquivo do Stage for more on this.)

We could just as easily not let Git assume we meant "pull the data from HEAD" by specifying a specific commit to pull that file version from. We would just run something like `git reset eb43bf file.txt`.



This effectively does the same thing as if we had reverted the content of the file to **v1** in the Working Directory, ran `git add` on it, then reverted it back to **v3** again (without actually going through all those steps). If we run `git commit` now, it will record a change that reverts that file back to **v1**, even though we never actually had it in our Working Directory again.

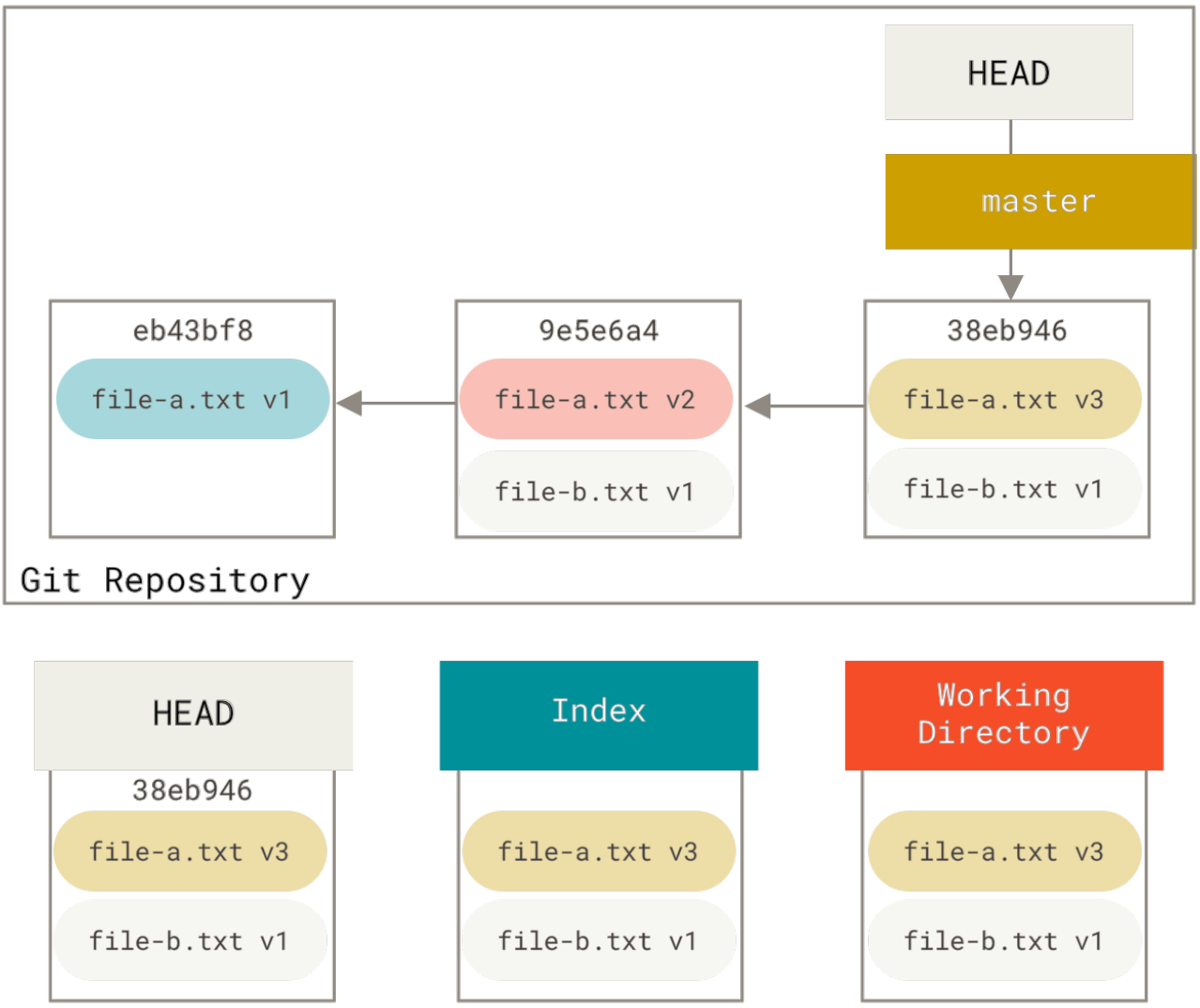
It's also interesting to note that like `git add`, the `reset` command will accept a `--patch` option to unstage content on a hunk-by-hunk basis. So you can selectively unstage or revert content.

Squashing

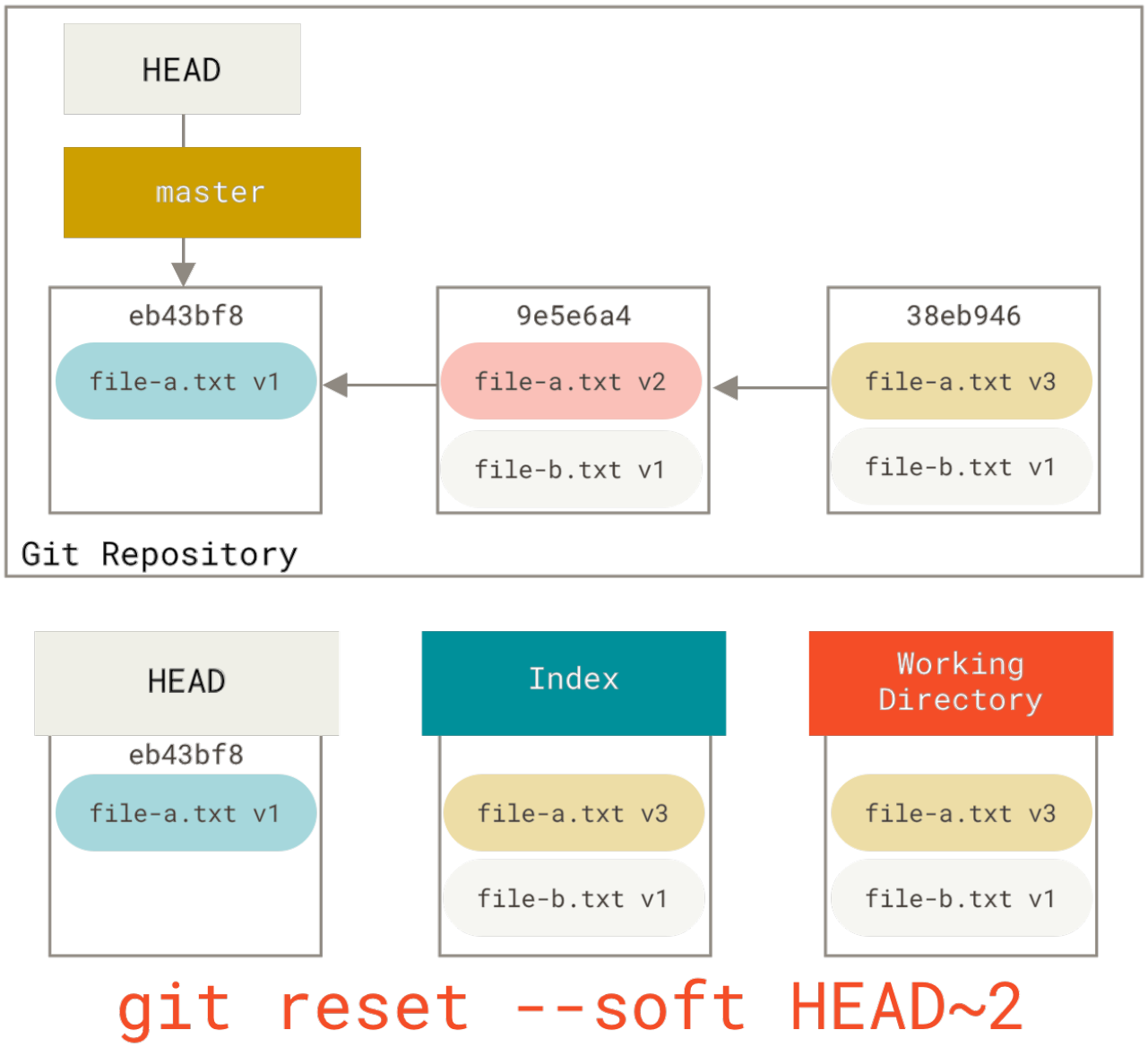
Let's look at how to do something interesting with this newfound power – squashing commits.

Say you have a series of commits with messages like "oops.", "WIP" and "forgot this file". You can use `reset` to quickly and easily squash them into a single commit that makes you look really smart. (Squashing Commits shows another way to do this, but in this example it's simpler to use `reset`.)

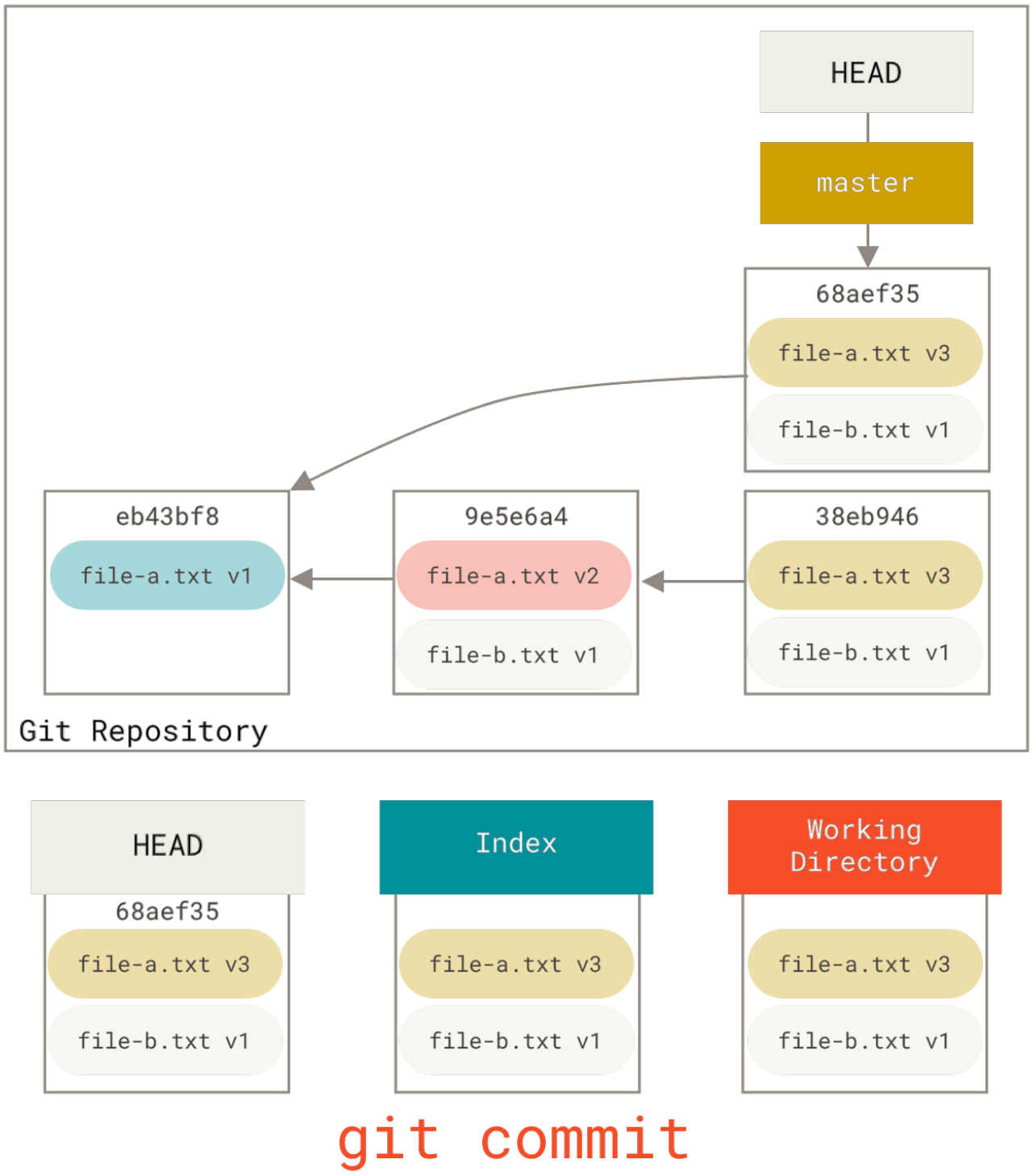
Let's say you have a project where the first commit has one file, the second commit added a new file and changed the first, and the third commit changed the first file again. The second commit was a work in progress and you want to squash it down.



You can run `git reset --soft HEAD~2` to move the HEAD branch back to an older commit (the first commit you want to keep):



And then simply run `git commit` again:



Now you can see that your reachable history, the history you would push, now looks like you had one commit with `file-a.txt` v1, then a second that both modified `file-a.txt` to v3 and added `file-b.txt`. The commit with the v2 version of the file is no longer in the history.

Check It Out

Finally, you may wonder what the difference between `checkout` and `reset` is. Like `reset`, `checkout` manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

Without Paths

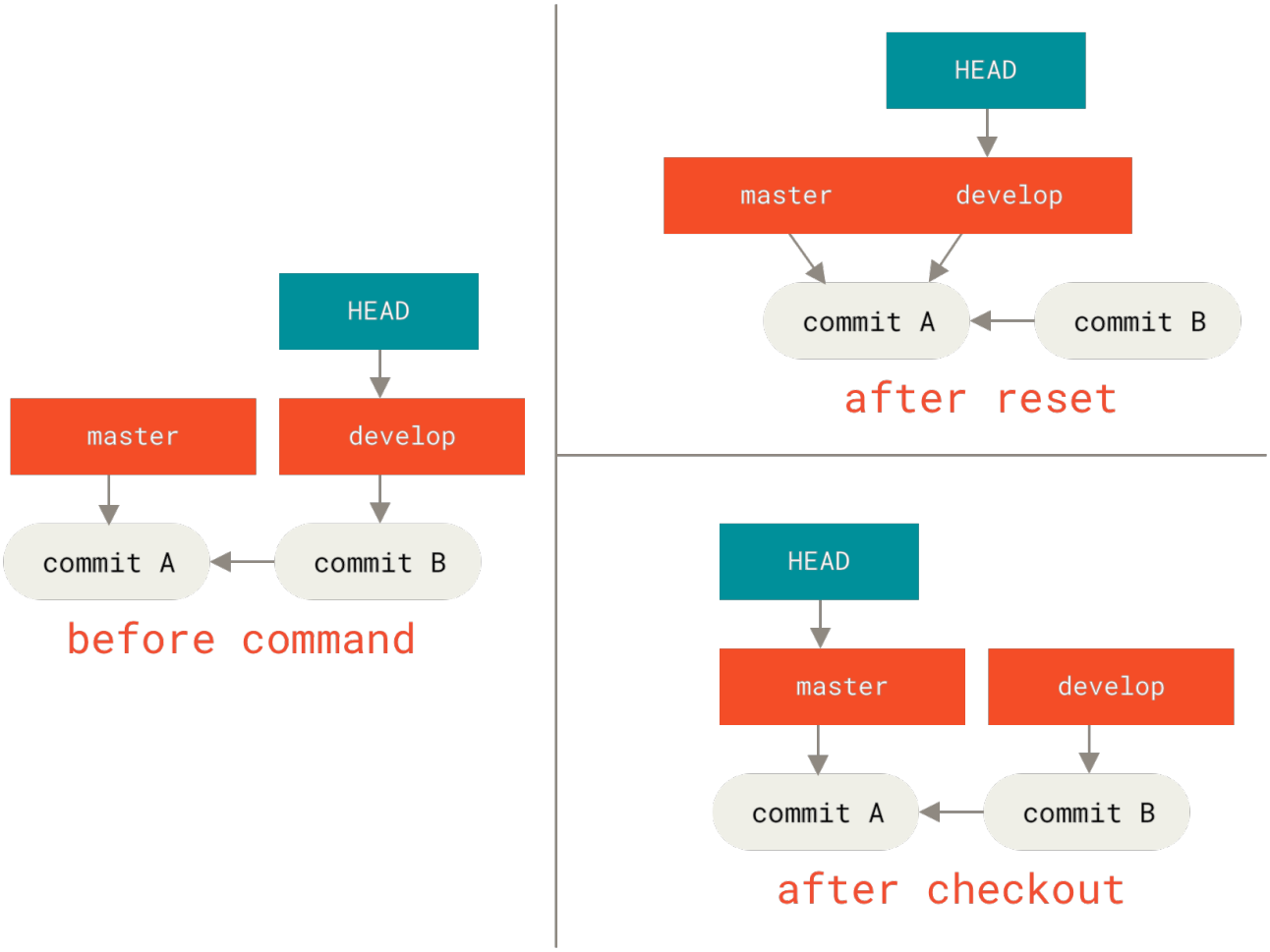
Running `git checkout [branch]` is pretty similar to running `git reset --hard [branch]` in that it updates all three trees for you to look like `[branch]`, but there are two important differences.

First, unlike `reset --hard`, `checkout` is working-directory safe; it will check to make sure it's not blowing away files that have changes to them. Actually, it's a bit smarter than that – it tries to do a trivial merge in the Working Directory, so all of the files you *haven't* changed in will be updated. `reset --hard`, on the other hand, will simply replace everything across the board without checking.

The second important difference is how it updates HEAD. Where `reset` will move the branch that HEAD points to, `checkout` will move HEAD itself to point to another branch.

For instance, say we have `master` and `develop` branches which point at different commits, and we're currently on `develop` (so HEAD points to it). If we run `git reset master`, `develop` itself will now point to the same commit that `master` does. If we instead run `git checkout master`, `develop` does not move, HEAD itself does. HEAD will now point to `master`.

So, in both cases we're moving HEAD to point to commit A, but *how* we do so is very different. `reset` will move the branch HEAD points to, `checkout` moves HEAD itself.



With Paths

The other way to run `checkout` is with a file path, which, like `reset`, does not move HEAD. It is just like `git reset [branch] file` in that it updates the index with that file at that commit, but it also

overwrites the file in the working directory. It would be exactly like `git reset --hard [branch] file` (if `reset` would let you run that) – it's not working-directory safe, and it does not move HEAD.

Also, like `git reset` and `git add`, `checkout` will accept a `--patch` option to allow you to selectively revert file contents on a hunk-by-hunk basis.

Summary

Hopefully now you understand and feel more comfortable with the `reset` command, but are probably still a little confused about how exactly it differs from `checkout` and could not possibly remember all the rules of the different invocations.

Here's a cheat-sheet for which commands affect which trees. The "HEAD" column reads "REF" if that command moves the reference (branch) that HEAD points to, and "HEAD" if it moves HEAD itself. Pay especial attention to the *WD Safe?* column – if it says **NO**, take a second to think before running that command.

	HEAD	**Index**	**Workdir**	**WD Safe?**
Commit Level				
`reset --soft [commit]`	REF	NO	NO	YES
`reset [commit]`	REF	YES	NO	YES
`reset --hard [commit]`	REF	YES	YES	**NO**
`checkout [commit]`	HEAD	YES	YES	YES
File Level				
`reset (commit) [file]`	NO	YES	NO	YES
`checkout (commit) [file]`	NO	YES	YES	**NO**

Advanced Merging

Merging in Git is typically fairly easy. Since Git makes it easy to merge another branch multiple times, it means that you can have a very long lived branch but you can keep it up to date as you go, solving small conflicts often, rather than be surprised by one enormous conflict at the end of the series.

However, sometimes tricky conflicts do occur. Unlike some other version control systems, Git does not try to be overly clever about merge conflict resolution. Git's philosophy is to be smart about determining when a merge resolution is unambiguous, but if there is a conflict, it does not try to be clever about automatically resolving it. Therefore, if you wait too long to merge two branches that diverge quickly, you can run into some issues.

In this section, we'll go over what some of those issues might be and what tools Git gives you to help handle these more tricky situations. We'll also cover some of the different, non-standard types of merges you can do, as well as see how to back out of merges that you've done.

Merge Conflicts

While we covered some basics on resolving merge conflicts in Conflitos Básicos de Merge, for more complex conflicts, Git provides a few tools to help you figure out what's going on and how to better deal with the conflict.

First of all, if at all possible, try to make sure your working directory is clean before doing a merge that may have conflicts. If you have work in progress, either commit it to a temporary branch or stash it. This makes it so that you can undo **anything** you try here. If you have unsaved changes in your working directory when you try a merge, some of these tips may help you lose that work.

Let's walk through a very simple example. We have a super simple Ruby file that prints *hello world*.

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

In our repository, we create a new branch named `whitespace` and proceed to change all the Unix line endings to DOS line endings, essentially changing every line of the file, but just with whitespace. Then we change the line "hello world" to "hello mundo".

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now we switch back to our `master` branch and add some documentation for the function.

```
$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

Now we try to merge in our `whitespace` branch and we'll get conflicts because of the whitespace changes.

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Aborting a Merge

We now have a few options. First, let's cover how to get out of this situation. If you perhaps weren't expecting conflicts and don't want to quite deal with the situation yet, you can simply back out of the merge with `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

The `git merge --abort` option tries to revert back to your state before you ran the merge. The only cases where it may not be able to do this perfectly would be if you had unstashed, uncommitted changes in your working directory when you ran it, otherwise it should work fine.

If for some reason you just want to start over, you can also run `git reset --hard HEAD`, and your repository will be back to the last committed state. Remember that any uncommitted work will be lost, so make sure you don't want any of your changes.

Ignoring Whitespace

In this specific case, the conflicts are whitespace related. We know this because the case is simple, but it's also pretty easy to tell in real cases when looking at the conflict because every line is removed on one side and added again on the other. By default, Git sees all of these lines as being changed, so it can't merge the files.

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with `-Xignore-all-space` or `-Xignore-space-change`. The first option ignores whitespace **completely** when comparing lines, the second treats sequences of one or more whitespace characters as equivalent.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Since in this case, the actual file changes were not conflicting, once we ignore the whitespace changes, everything merges just fine.

This is a lifesaver if you have someone on your team who likes to occasionally reformat everything from spaces to tabs or vice-versa.

Manual File Re-merging

Though Git handles whitespace pre-processing pretty well, there are other types of changes that perhaps Git can't handle automatically, but are scriptable fixes. As an example, let's pretend that Git could not handle the whitespace change and we needed to do it by hand.

What we really need to do is run the file we're trying to merge in through a `dos2unix` program before trying the actual file merge. So how would we do that?

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we're merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under "stages" which each have numbers associated with them. Stage 1 is the common ancestor, stage 2 is your version and stage 3 is from the `MERGE_HEAD`, the version you're merging in ("theirs").

You can extract a copy of each of these versions of the conflicted file with the `git show` command and a special syntax.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

If you want to get a little more hard core, you can also use the `ls-files -u` plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1   hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2   hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3   hello.rb
```

The `:1:hello.rb` is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known `git merge-file` command which does just that.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

At this point we have nicely merged the file. In fact, this actually works better than the `ignore-space-change` option because this actually fixes the whitespace changes before merge instead of simply ignoring them. In the `ignore-space-change` merge, we actually ended up with a few lines with DOS line endings, making things mixed.

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask `git diff` to compare what is in your working directory that

you're about to commit as the result of the merge to any of these stages. Let's go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

So here we can easily see that what happened in our branch, what we're actually introducing to this file with this merge, is changing that single line.

If we want to see how the result of the merge differed from what was on their side, you can run `git diff --theirs`. In this and the following example, we have to use `-b` to strip out the whitespace because we're comparing it to what is in Git, not our cleaned up `hello.theirs.rb` file.

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

Finally, you can see how the file has changed from both sides with `git diff --base`.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

At this point we can use the `git clean` command to clear out the extra files we created to do the manual merge but no longer need.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Checking Out Conflicts

Perhaps we're not happy with the resolution at this point for some reason, or maybe manually editing one or both sides still didn't work well and we need more context.

Let's change up the example a little. For this example, we have two longer lived branches that each have a few commits in them but create a legitimate content conflict when merged.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

We now have three unique commits that live only on the `master` branch and three others that live on the `mundo` branch. If we try to merge the `mundo` branch in, we get a conflict.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

We would like to see what the merge conflict is. If we open up the file, we'll see something like this:

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

Both sides of the merge added content to this file, but some of the commits modified the file in the same place that caused this conflict.

Let's explore a couple of tools that you now have at your disposal to determine how this conflict came to be. Perhaps it's not obvious how exactly you should fix this conflict. You need more context.

One helpful tool is `git checkout` with the '--conflict' option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass `--conflict` either `diff3` or `merge` (which is the default). If you pass it `diff3`, Git will use a slightly different version of conflict markers, not only giving you the "ours" and "theirs" versions, but also the "base" version inline to give you more context.

```
$ git checkout --conflict=diff3 hello.rb
```

Once we run that, the file will look like this instead:

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

If you like this format, you can set it as the default for future merge conflicts by setting the `merge.conflictstyle` setting to `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

The `git checkout` command can also take `--ours` and `--theirs` options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing.

Merge Log

Another useful tool when resolving merge conflicts is `git log`. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in Triple Dot.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

That’s a nice list of the six total commits involved, as well as which line of development each commit was on.

We can further simplify this though to give us much more specific context. If we add the `--merge`

option to `git log`, it will only show the commits in either side of the merge that touch a file that's currently conflicted.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

If you run that with the `-p` option instead, you get just the diffs to the file that ended up in conflict. This can be **really** helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

Combined Diff Format

Since Git stages any merge results that are successful, when you run `git diff` while in a conflicted merge state, you only get what is currently still in conflict. This can be helpful to see what you still have to resolve.

When you run `git diff` directly after a merge conflict, it will give you information in a rather unique diff output format.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()
```

The format is called "Combined Diff" and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the "ours" branch and the file in your working directory and the second column does the same between the "theirs" branch and your working directory copy.

So in that example you can see that the `<<<<<<<` and `>>>>>>>` lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we're expected to remove them.

If we resolve the conflict and run `git diff` again, we'll see the same thing, but it's a little more

useful.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

This shows us that “hola world” was in our side but not in the working copy, that “hello mundo” was in their side but not in the working copy and finally that “hola mundo” was not in either side but is now in the working copy. This can be useful to review before committing the resolution.

You can also get this from the `git log` for any merge to see how something was resolved after the fact. Git will output this format if you run `git show` on a merge commit, or if you add a `--cc` option to a `git log -p` (which by default only shows patches for non-merge commits).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Undoing Merges

Now that you know how to create a merge commit, you'll probably make some by mistake. One of the great things about working with Git is that it's okay to make mistakes, because it's possible (and in many cases easy) to fix them.

Merge commits are no different. Let's say you started work on a topic branch, accidentally merged it into `master`, and now your commit history looks like this:

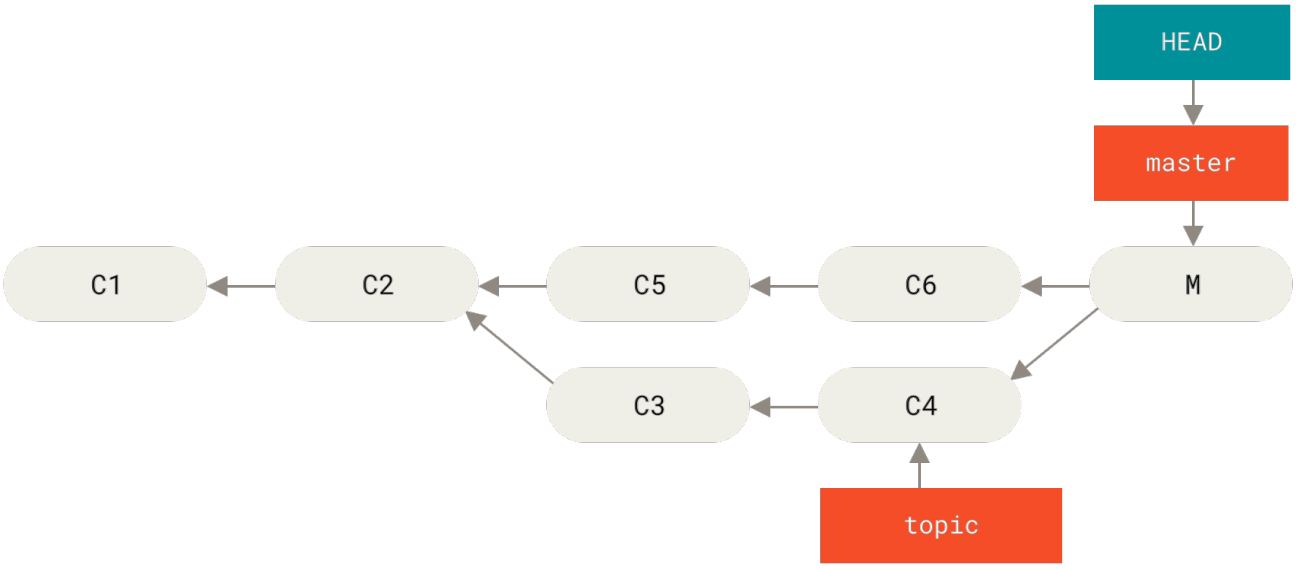


Figura 138. Accidental merge commit

There are two ways to approach this problem, depending on what your desired outcome is.

Fix the references

If the unwanted merge commit only exists on your local repository, the easiest and best solution is to move the branches so that they point where you want them to. In most cases, if you follow the errant `git merge` with `git reset --hard HEAD~`, this will reset the branch pointers so they look like this:

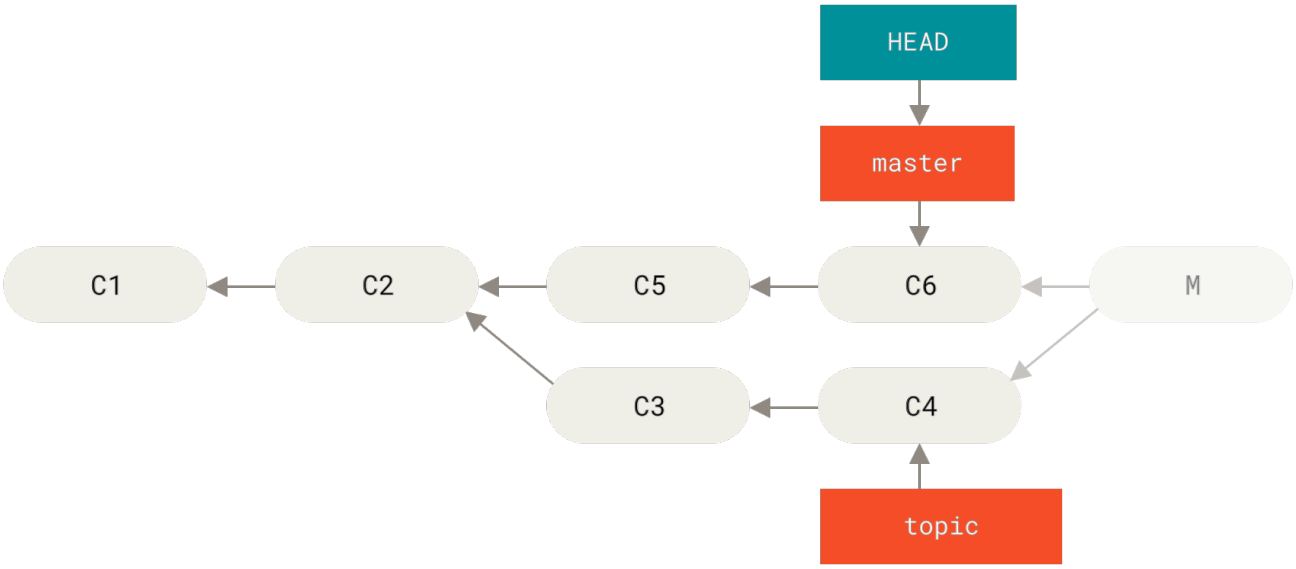


Figura 139. History after `git reset --hard HEAD~`

We covered `reset` back in Reset Demystified, so it shouldn't be too hard to figure out what's going on here. Here's a quick refresher: `reset --hard` usually goes through three steps:

1. Move the branch HEAD points to. In this case, we want to move `master` to where it was before the merge commit (`C6`).
2. Make the index look like HEAD.

3. Make the working directory look like the index.

The downside of this approach is that it's rewriting history, which can be problematic with a shared repository. Check out Os perigos do Rebase for more on what can happen; the short version is that if other people have the commits you're rewriting, you should probably avoid `reset`. This approach also won't work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

Reverse the commit

If moving the branch pointers around isn't going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a "revert", and in this particular scenario, you'd invoke it like this:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

The `-m 1` flag indicates which parent is the "mainline" and should be kept. When you invoke a merge into `HEAD` (`git merge topic`), the new commit has two parents: the first one is `HEAD` (`C6`), and the second is the tip of the branch being merged in (`C4`). In this case, we want to undo all the changes introduced by merging in parent #2 (`C4`), while keeping all the content from parent #1 (`C6`).

The history with the revert commit looks like this:

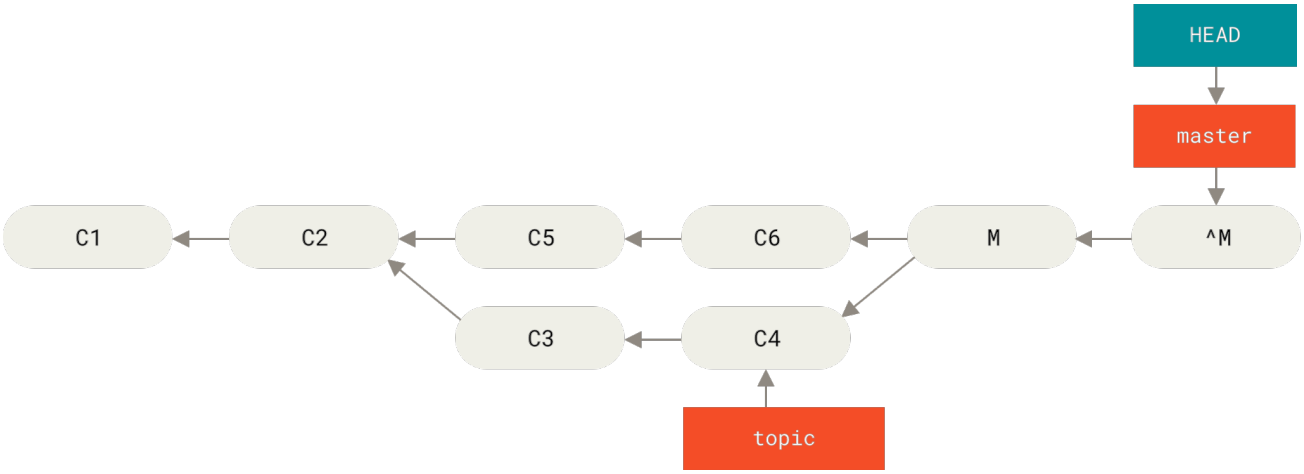


Figura 140. History after `git revert -m 1`

The new commit `^M` has exactly the same contents as `C6`, so starting from here it's as if the merge never happened, except that the now-unmerged commits are still in `HEAD`'s history. Git will get confused if you try to merge `topic` into `master` again:

```
$ git merge topic
Already up-to-date.
```

There's nothing in `topic` that isn't already reachable from `master`. What's worse, if you add work to `topic` and merge again, Git will only bring in the changes *since* the reverted merge:

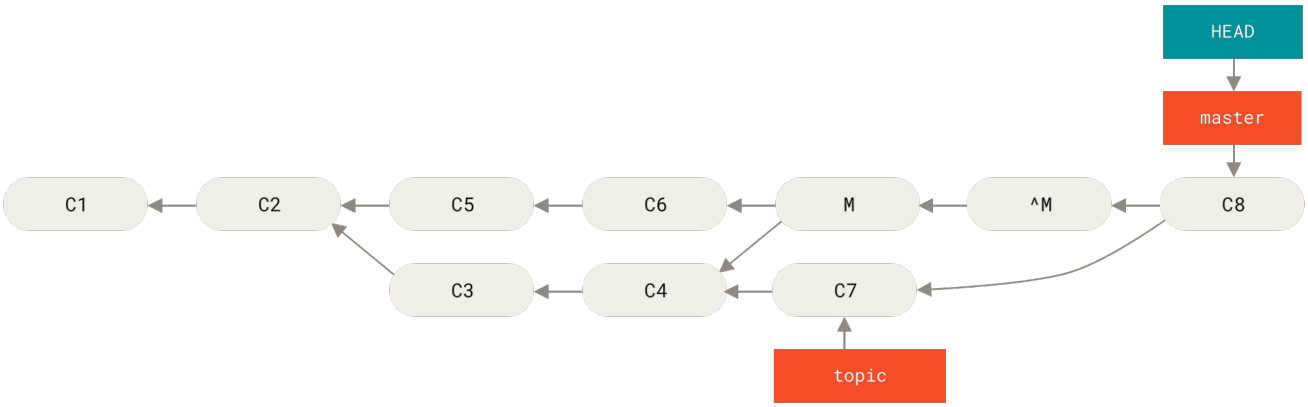


Figura 141. History with a bad merge

The best way around this is to un-revert the original merge, since now you want to bring in the changes that were reverted out, **then** create a new merge commit:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

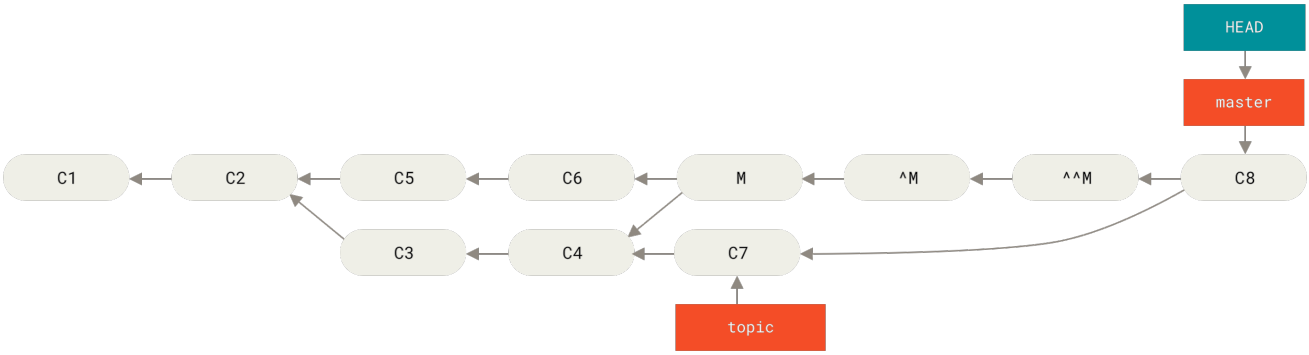


Figura 142. History after re-merging a reverted merge

In this example, `M` and `^M` cancel out. `^^M` effectively merges in the changes from `C3` and `C4`, and `C8` merges in the changes from `C7`, so now `topic` is fully merged.

Other Types of Merges

So far we've covered the normal merge of two branches, normally handled with what is called the "recursive" strategy of merging. There are other ways to merge branches together however. Let's cover a few of them quickly.

Our or Theirs Preference

First of all, there is another useful thing we can do with the normal "recursive" mode of merging. We've already seen the `ignore-all-space` and `ignore-space-change` options which are passed with a `-X` but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually

resolve the conflict, you can pass the `merge` command either a `-Xours` or `-Xtheirs`.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

If we go back to the "hello world" example we were using before, we can see that merging in our branch causes conflicts.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

However if we run it with `-Xours` or `-Xtheirs` it does not.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

In that case, instead of getting conflict markers in the file with "hello mundo" on one side and "hola world" on the other, it will simply pick "hola world". However, all the other non-conflicting changes on that branch are merged successfully in.

This option can also be passed to the `git merge-file` command we saw earlier by running something like `git merge-file --ours` for individual file merges.

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the "ours" merge *strategy*. This is different from the "ours" recursive merge *option*.

This will basically do a fake merge. It will record a new merge commit with both branches as parents, but it will not even look at the branch you're merging in. It will simply record as the result of the merge the exact code in your current branch.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

You can see that there is no difference between the branch we were on and the result of the merge.

This can often be useful to basically trick Git into thinking that a branch is already merged when doing a merge later on. For example, say you branched off a `release` branch and have done some work on it that you will want to merge back into your `master` branch at some point. In the meantime some bugfix on `master` needs to be backported into your `release` branch. You can merge the bugfix branch into the `release` branch and also `merge -s ours` the same branch into your `master` branch (even though the fix is already there) so when you later merge the `release` branch again, there are no conflicts from the bugfix.

Subtree Merging

The idea of the subtree merge is that you have two projects, and one of the projects maps to a subdirectory of the other one. When you specify a subtree merge, Git is often smart enough to figure out that one is a subtree of the other and merge appropriately.

We'll go through an example of adding a separate project into an existing project and then merging the code of the second into a subdirectory of the first.

First, we'll add the Rack application to our project. We'll add the Rack project as a remote reference in our own project and then check it out into its own branch:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Now we have the root of the Rack project in our `rack_branch` branch and our own project in the `master` branch. If you check out one and then the other, you can see that they have different project roots:

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

This is sort of a strange concept. Not all the branches in your repository actually have to be branches of the same project. It's not common, because it's rarely helpful, but it's fairly easy to have branches contain completely different histories.

In this case, we want to pull the Rack project into our `master` project as a subdirectory. We can do that in Git with `git read-tree`. You'll learn more about `read-tree` and its friends in Funcionamento Interno do Git, but for now know that it reads the root tree of one branch into your current staging area and working directory. We just switched back to your `master` branch, and we pull the `rack_branch` branch into the `rack` subdirectory of our `master` branch of our main project:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

When we commit, it looks like we have all the Rack files under that subdirectory – as though we copied them in from a tarball. What gets interesting is that we can fairly easily merge changes from one of the branches to the other. So, if the Rack project updates, we can pull in upstream changes by switching to that branch and pulling:

```
$ git checkout rack_branch
$ git pull
```

Then, we can merge those changes back into our `master` branch. To pull in the changes and prepopulate the commit message, use the `--squash` option, as well as the recursive merge strategy's `-Xsubtree` option. (The recursive strategy is the default here, but we include it for clarity.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

All the changes from the Rack project are merged in and ready to be committed locally. You can also do the opposite – make changes in the `rack` subdirectory of your master branch and then merge them into your `rack_branch` branch later to submit them to the maintainers or push them upstream.

This gives us a way to have a workflow somewhat similar to the submodule workflow without using submodules (which we will cover in Submodules). We can keep branches with other related projects in our repository and subtree merge them into our project occasionally. It is nice in some ways, for example all the code is committed to a single place. However, it has other drawbacks in that it's a bit more complex and easier to make mistakes in reintegrating changes or accidentally pushing a branch into an unrelated repository.

Another slightly weird thing is that to get a diff between what you have in your `rack` subdirectory and the code in your `rack_branch` branch – to see if you need to merge them – you can't use the normal `diff` command. Instead, you must run `git diff-tree` with the branch you want to compare to:

```
$ git diff-tree -p rack_branch
```

Or, to compare what is in your `rack` subdirectory with what the `master` branch on the server was the last time you fetched, you can run

```
$ git diff-tree -p rack_remote/master
```

Rerere

The `git rerere` functionality is a bit of a hidden feature. The name stands for "reuse recorded resolution" and as the name implies, it allows you to ask Git to remember how you've resolved a hunk conflict so that the next time it sees the same conflict, Git can automatically resolve it for you.

There are a number of scenarios in which this functionality might be really handy. One of the examples that is mentioned in the documentation is if you want to make sure a long lived topic branch will merge cleanly but don't want to have a bunch of intermediate merge commits. With `rerere` turned on you can merge occasionally, resolve the conflicts, then back out the merge. If you do this continuously, then the final merge should be easy because `rerere` can just do everything for you automatically.

This same tactic can be used if you want to keep a branch rebased so you don't have to deal with the same rebasing conflicts each time you do it. Or if you want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead - you likely won't have to do all the same conflicts again.

Another situation is where you merge a bunch of evolving topic branches together into a testable head occasionally, as the Git project itself often does. If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

To enable the `rerere` functionality, you simply have to run this config setting:

```
$ git config --global rerere.enabled true
```

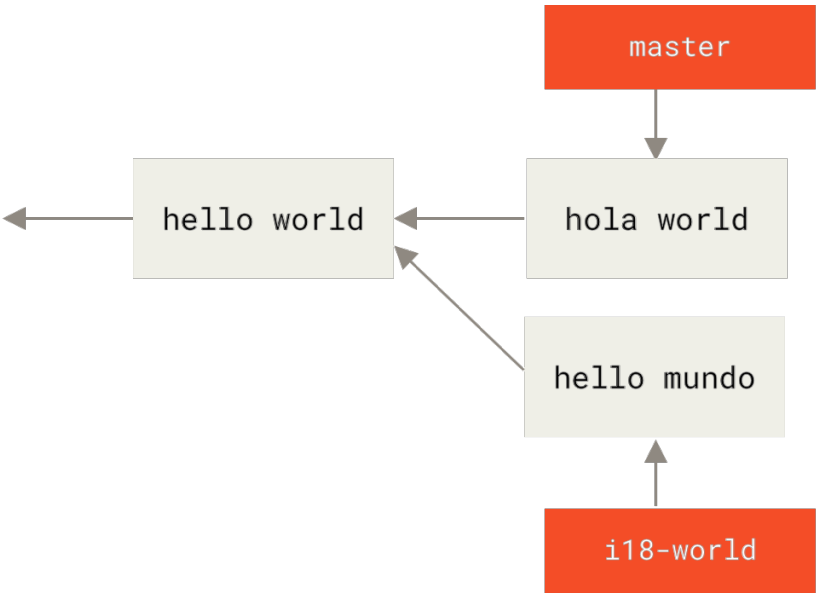
You can also turn it on by creating the `.git/rr-cache` directory in a specific repository, but the config setting is clearer and it can be done globally.

Now let's see a simple example, similar to our previous one. Let's say we have a file named `hello.rb` that looks like this:

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

In one branch we change the word “hello” to “hola”, then in another branch we change the “world” to “mundo”, just like before.



When we merge the two branches together, we’ll get a merge conflict:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

You should notice the new line `Recorded preimage for FILE` in there. Otherwise it should look exactly like a normal merge conflict. At this point, `rerere` can tell us a few things. Normally, you might run `git status` at this point to see what all conflicted:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#   both modified:      hello.rb
#
```

However, `git rerere` will also tell you what it has recorded the pre-merge state for with `git rerere status`:

```
$ git rerere status
hello.rb
```

And `git rerere diff` will show the current state of the resolution - what you started with to resolve and what you've resolved it to.

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
   puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
+>>>>>>> i18n-world
 end
```

Also (and this isn't really related to `rerere`), you can use `ls-files -u` to see the conflicted files and the before, left and right versions:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1   hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2   hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3   hello.rb
```

Now you can resolve it to just be `puts 'hola mundo'` and you can run the `rerere diff` command again to see what rerere will remember:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

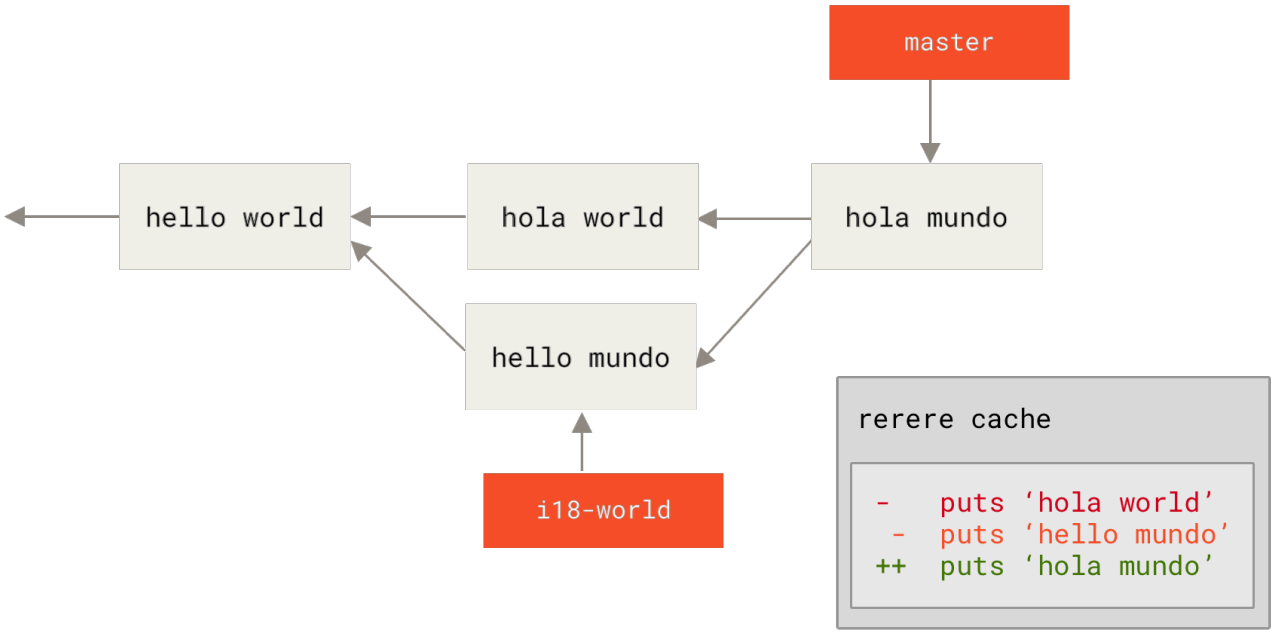
 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

So that basically says, when Git sees a hunk conflict in a `hello.rb` file that has “hello mundo” on one side and “hola world” on the other, it will resolve it to “hola mundo”.

Now we can mark it as resolved and commit it:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

You can see that it "Recorded resolution for FILE".



Now, let’s undo that merge and then rebase it on top of our master branch instead. We can move our branch back by using `reset` as we saw in Reset Demystified.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Our merge is undone. Now let’s rebase the topic branch.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

Now, we got the same merge conflict like we expected, but take a look at the `Resolved FILE using previous resolution` line. If we look at the file, we'll see that it's already been resolved, there are no merge conflict markers in it.

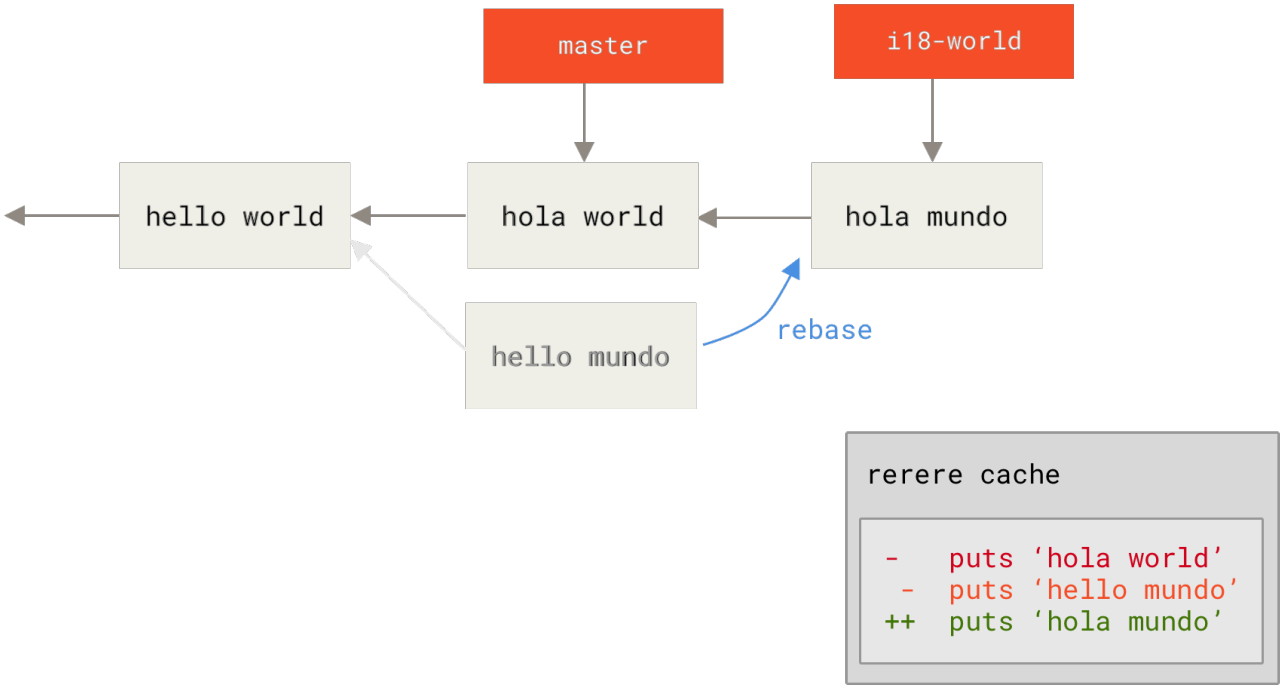
```
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Also, `git diff` will show you how it was automatically re-resolved:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```



You can also recreate the conflicted file state with the `checkout` command:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

We saw an example of this in Advanced Merging. For now though, let's re-resolve it by just running `rerere` again:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

We have re-resolved the file automatically using the `rerere` cached resolution. You can now add and continue the rebase to complete it.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

So, if you do a lot of re-merges, or want to keep a topic branch up to date with your master branch without a ton of merges, or you rebase often, you can turn on `rerere` to help your life out a bit.

Debugging with Git

Git also provides a couple of tools to help you debug issues in your projects. Because Git is designed to work with nearly any type of project, these tools are pretty generic, but they can often help you hunt for a bug or culprit when things go wrong.

File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows you what commit was the last to modify each line of any file. So, if you see that a method in your code is buggy, you can annotate the file with `git blame` to see when each line of the method was last edited and by whom. This example uses the `-L` option to limit the output to lines 12 through 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

Notice that the first field is the partial SHA-1 of the commit that last modified that line. The next two fields are values extracted from that commit–the author name and the authored date of that commit – so you can easily see who modified that line and when. After that come the line number and the content of the file. Also note the `^4832fe2` commit lines, which designate that those lines were in this file's original commit. That commit is when this file was first added to this project, and those lines have been unchanged since. This is a tad confusing, because now you've seen at least three different ways that Git uses the `^` to modify a commit SHA-1, but that is what it means here.

Another cool thing about Git is that it doesn't track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass `-C` to `git blame`, Git analyzes the file you're annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. For example, say you are

refactoring a file named `GITServerHandler.m` into multiple files, one of which is `GITPackUpload.m`. By blaming `GITPackUpload.m` with the `-C` option, you can see where sections of the code originally came from:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

This is really useful. Normally, you get as the original commit the commit where you copied the code over, because that is the first time you touched those lines in this file. Git tells you the original commit where you wrote those lines, even if it was in another file.

Binary Search

Annotating a file helps if you know where the issue is to begin with. If you don't know what is breaking, and there have been dozens or hundreds of commits since the last state where you know the code worked, you'll likely turn to `git bisect` for help. The `bisect` command does a binary search through your commit history to help you identify as quickly as possible which commit introduced an issue.

Let's say you just pushed out a release of your code to a production environment, you're getting bug reports about something that wasn't happening in your development environment, and you can't imagine why the code is doing that. You go back to your code, and it turns out you can reproduce the issue, but you can't figure out what is going wrong. You can bisect the code to find out. First you run `git bisect start` to get things going, and then you use `git bisect bad` to tell the system that the current commit you're on is broken. Then, you must tell bisect when the last known good state was, using `git bisect good [good_commit]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git figured out that about 12 commits came between the commit you marked as the last good commit (v1.0) and the current bad version, and it checked out the middle one for you. At this point,

you can run your test to see if the issue exists as of this commit. If it does, then it was introduced sometime before this middle commit; if it doesn't, then the problem was introduced sometime after the middle commit. It turns out there is no issue here, and you tell Git that by typing `git bisect good` and continue your journey:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Now you're on another commit, halfway between the one you just tested and your bad commit. You run your test again and find that this commit is broken, so you tell Git that with `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

This commit is fine, and now Git has all the information it needs to determine where the issue was introduced. It tells you the SHA-1 of the first bad commit and show some of the commit information and which files were modified in that commit so you can figure out what happened that may have introduced this bug:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

When you're finished, you should run `git bisect reset` to reset your HEAD to where you were before you started, or you'll end up in a weird state:

```
$ git bisect reset
```

This is a powerful tool that can help you check hundreds of commits for an introduced bug in minutes. In fact, if you have a script that will exit 0 if the project is good or non-0 if the project is bad, you can fully automate `git bisect`. First, you again tell it the scope of the bisect by providing the known bad and good commits. You can do this by listing them with the `bisect start` command if you want, listing the known bad commit first and the known good commit second:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Doing so automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. You can also run something like `make` or `make tests` or whatever you have that runs automated tests for you.

Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Here's an example. Suppose you're developing a website and creating Atom feeds. Instead of writing your own Atom-generating code, you decide to use a library. You're likely to have to either include this code from a shared library like a CPAN install or Ruby gem, or copy the source code into your own project tree. The issue with including the library is that it's difficult to customize the library in any way and often more difficult to deploy it, because you need to make sure every client has that library available. The issue with copying the code into your own project is that any custom changes you make are difficult to merge when upstream changes become available.

Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

Starting with Submodules

We'll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let's start by adding an existing Git repository as a submodule of the repository that we're working on. To add a new submodule you use the `git submodule add` command with the absolute or relative URL of the project you would like to start tracking. In this example, we'll add a library called "DbConnector".

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

By default, submodules will add the subproject into a directory named the same as the repository, in this case "DbConnector". You can add a different path at the end of the command if you want it to go elsewhere.

If you run `git status` at this point, you'll notice a few things.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   DbConnector
```

First you should notice the new `.gitmodules` file. This is a configuration file that stores the mapping between the project's URL and the local subdirectory you've pulled it into:

```
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

If you have multiple submodules, you'll have multiple entries in this file. It's important to note that this file is version-controlled with your other files, like your `.gitignore` file. It's pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

NOTA

Since the URL in the .gitmodules file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would to pull from, use the one that others have access to. You can overwrite this value locally with `git config submodule.DbConnector.url PRIVATE_URL` for your own use. When applicable, a relative URL can be helpful.

The other listing in the `git status` output is the project folder entry. If you run `git diff` on that, you see something interesting:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Although `DbConnector` is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

If you want a little nicer diff output, you can pass the `--submodule` option to `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

When you commit, you see something like this:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Notice the `160000` mode for the `DbConnector` entry. That is a special mode in Git that basically means you're recording a commit as a directory entry rather than a subdirectory or a file.

Lastly, push these changes:

```
$ git push origin master
```

Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

The `DbConnector` directory is there, but empty. You must run two commands: `git submodule init` to initialize your local configuration file, and `git submodule update` to fetch all the data from that project and check out the appropriate commit listed in your superproject:

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Now your `DbConnector` subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass `--recursive` to the `git clone` command, it will automatically initialize and update each submodule in the repository.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Working on a Project with Submodules

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

Pulling in Upstream Changes

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let's walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run `git fetch` and `git merge` the upstream branch to update the local code.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Now if you go back into the main project and run `git diff --submodule` you can see that the submodule was updated and get a list of commits that were added to it. If you don't want to type `--submodule` every time you run `git diff`, you can set it as the default format by setting the `diff.submodule` config value to "log".

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run `git submodule update --remote`, Git will go into your submodules and fetch and update for you.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

This command will by default assume that you want to update the checkout to the `master` branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository's "stable" branch, you can set it in either your `.gitmodules` file (so everyone else also tracks it), or just in your local `.git/config` file. Let's set it in the `.gitmodules` file:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

If you leave off the `-f .gitmodules` it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run `git status` at this point, Git will show us that we have "new commits" on the submodule.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

If you set the configuration setting `status.submodulesummary`, Git will also show you a short summary of changes to your submodules:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   .gitmodules
    modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

At this point if you run `git diff` we can see both that we have modified our `.gitmodules` file and also that there are a number of commits that we've pulled down and are ready to commit to our submodule project.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

This is pretty cool as we can actually see the log of commits that we're about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run `git log -p`.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Git will by default try to update **all** of your submodules when you run `git submodule update --remote` so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

Working on a Submodule

It's quite likely that if you're using submodules, you're doing so because you really want to work on the code in the submodule at the same time as you're working on the code in the main project (or across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let's go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we've run the `git submodule update` command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what's called a "detached HEAD" state. This means that there is no local working branch (like "master", for example) tracking changes. With no working branch tracking changes, that means even if you commit changes to the submodule, those changes will quite possibly be lost the next time you run `git submodule update`. You have to do some extra steps if you want changes in a submodule to be tracked.

In order to set up your submodule to be easier to go in and hack on, you need do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then `git submodule update --remote` pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let's go into our submodule directory and check out a branch.

```
$ git checkout stable
Switched to branch 'stable'
```

Let's try it with the "merge" option. To specify it manually, we can just add the `--merge` option to our `update` call. Here we'll see that there was a change on the server for this submodule and it gets merged in.

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

If we go into the DbConnector directory, we have the new changes already merged into our local `stable` branch. Now let's see what happens when we make our own local change to the library and

someone else pushes another change upstream at the same time.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If you forget the `--rebase` or `--merge`, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If this happens, don't worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase `origin/stable` (or whatever remote branch you want) manually.

If you haven't committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
    scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

You can go into the submodule directory and fix the conflict just as you normally would.

Publishing Submodule Changes

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren't available to anyone else yet as we haven't pushed them yet.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn't happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The `git push` command takes the `--recurse-submodules` argument which can be set to either "check" or "on-demand". The "check" option will make `push` simply fail if any of the committed submodule changes haven't been pushed.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

    git push --recurse-submodules=on-demand

or cd to the path and use

    git push

to push them to a remote.
```

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they're externally available and then try this push again. If you want the check behavior to happen for all pushes, you can make this behavior the default by doing `git config push.recurseSubmodules check`.

The other option is to use the "on-demand" value, which will try to do this for you.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail. You can make this behavior the default by doing `git config push.recurseSubmodules on-demand`.

Merging Submodule Changes

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
   9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

So basically what has happened here is that Git has figured out that the two branches record points in the submodule's history that are divergent and need to be merged. It explains it as "merge following commits not found", which is confusing but we'll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn't really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it's simple to figure out. If you run `git diff` you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

So, in this case, `eb41d76` is the commit in our submodule that **we** had and `c771610` is the commit that upstream had. If we go into our submodule directory, it should already be on `eb41d76` as the merge would not have touched it. If for whatever reason it's not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you'll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch based on that second SHA-1 from `git diff` and manually merge.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
 -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes
```

① First we resolve the conflict

② Then we go back to the main project directory

③ We can check the SHA-1s again

④ Resolve the conflicted submodule entry

⑤ Commit our merge

It can be a bit confusing, but it's really not very hard.

Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone merged branches containing these two

commits, so maybe you'll want that one.

This is why the error message from before was "merge following commits not found", because it could not do **this**. It's confusing because who would expect it to **try** to do this?

If it does find a single acceptable merge commit, you'll see something like this:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
"DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

What it's suggesting that you do is to update the index like you had run `git add`, which clears the conflict, then commit. You probably shouldn't do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you're done.

Submodule Tips

There are a few things you can do to make working with submodules a little easier.

Submodule Foreach

There is a `foreach` submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

For example, let's say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from
origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Then we can create a new branch and switch to it in all our submodules.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

      commit_pager_choice();

+     url = url_decode(url_orig);
+
      /* build alias_argv */
      alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
      alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
        return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+       return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
        struct strbuf out = STRBUF_INIT;
```

Here we can see that we're defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

Useful Aliases

You may want to set up some aliases for some of these commands as they can be quite long and you can't set configuration options for most of them to make them defaults. We covered setting up Git aliases in Apelidos Git, but here is an example of what you may want to set up if you plan on working with submodules in Git a lot.

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

This way you can simply run `git supdate` when you want to update your submodules, or `git spush`

to push with submodule dependency checking.

Issues with Submodules

Using submodules isn't without hiccups, however.

For instance switching branches with submodules in them can also be tricky. If you create a new branch, add a submodule there, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)
```

Removing the directory isn't difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run `submodule update --init` to repopulate it.

```
$ git clean -ffdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile    includes    scripts     src
```

Again, not really very difficult, but it can be a little confusing.

The other main caveat that many people run into involves switching from subdirectories to submodules. If you've been tracking files in your project and you want to move them out into a submodule, you must be careful or Git will get angry at you. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. If you delete the subdirectory and then run `submodule add`, Git yells at you:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

You have to unstage the `CryptoLibrary` directory first. Then you can add the submodule:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Now suppose you did that in a branch. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

You can force it to switch with `checkout -f`, but be careful that you don't have unsaved changes in there as they could be overwritten with that command.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Then, when you switch back, you get an empty `CryptoLibrary` directory for some reason and `git submodule update` may not fix it either. You may need to go into your submodule directory and run a `git checkout .` to get all your files back. You could run this in a `submodule foreach` script to run it for multiple submodules.

It's important to note that submodules these days keep all their Git data in the top project's `.git` directory, so unlike much older versions of Git, destroying a submodule directory won't lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.

Bundling

Though we've covered the common ways to transfer Git data over a network (HTTP, SSH, etc), there is actually one more way to do so that is not commonly used but can actually be quite useful.

Git is capable of "bundling" its data into a single file. This can be useful in various scenarios. Maybe your network is down and you want to send changes to your co-workers. Perhaps you're working somewhere offsite and don't have access to the local network for security reasons. Maybe your wireless/ethernet card just broke. Maybe you don't have access to a shared server for the moment, you want to email someone updates and you don't want to transfer 40 commits via `format-patch`.

This is where the `git bundle` command can be helpful. The `bundle` command will package up everything that would normally be pushed over the wire with a `git push` command into a binary file that you can email to someone or put on a flash drive, then unbundle into another repository.

Let's see a simple example. Let's say you have a repository with two commits:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

If you want to send that repository to someone and you don't have access to a repository to push to, or simply don't want to set one up, you can bundle it with `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Now you have a file named `repo.bundle` that has all the data needed to re-create the repository's `master` branch. With the `bundle` command you need to list out every reference or specific range of commits that you want to be included. If you intend for this to be cloned somewhere else, you should add HEAD as a reference as well as we've done here.

You can email this `repo.bundle` file to someone else, or put it on a USB drive and walk it over.

On the other side, say you are sent this `repo.bundle` file and want to work on the project. You can clone from the binary file into a directory, much like you would from a URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

If you don't include HEAD in the references, you have to also specify `-b master` or whatever branch is included because otherwise it won't know what branch to check out.

Now let's say you do three commits on it and want to send the new commits back via a bundle on a USB stick or email.

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

First we need to determine the range of commits we want to include in the bundle. Unlike the network protocols which figure out the minimum set of data to transfer over the network for us, we'll have to figure this out manually. Now, you could just do the same thing and bundle the entire repository, which will work, but it's better to just bundle up the difference - just the three commits we just made locally.

In order to do that, you'll have to calculate the difference. As we described in Commit Ranges, you can specify a range of commits in a number of ways. To get the three commits that we have in our master branch that weren't in the branch we originally cloned, we can use something like `origin/master..master` or `master ^origin/master`. You can test that with the `log` command.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

So now that we have the list of commits we want to include in the bundle, let's bundle them up. We do that with the `git bundle create` command, giving it a filename we want our bundle to be and the range of commits we want to go into it.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

Now we have a `commits.bundle` file in our directory. If we take that and send it to our partner, she can then import it into the original repository, even if more work has been done there in the meantime.

When she gets the bundle, she can inspect it to see what it contains before she imports it into her repository. The first command is the `bundle verify` command that will make sure the file is actually a valid Git bundle and that you have all the necessary ancestors to reconstitute it properly.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

If the bundler had created a bundle of just the last two commits they had done, rather than all three, the original repository would not be able to import it, since it is missing requisite history. The `verify` command would have looked like this instead:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

However, our first bundle is valid, so we can fetch in commits from it. If you want to see what branches are in the bundle that can be imported, there is also a command to just list the heads:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

The `verify` sub-command will tell you the heads as well. The point is to see what can be pulled in, so you can use the `fetch` or `pull` commands to import commits from this bundle. Here we'll fetch the *master* branch of the bundle to a branch named *other-master* in our repository:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

Now we can see that we have the imported commits on the *other-master* branch as well as any commits we've done in the meantime in our own *master* branch.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

So, `git bundle` can be really useful for sharing or doing network-type operations when you don't have the proper network or shared repository to do so.

Replace

Git's objects are unchangeable, but it does provide an interesting way to pretend to replace objects in its database with other objects.

The `replace` command lets you specify an object in Git and say "every time you see this, pretend it's this other thing". This is most commonly useful for replacing one commit in your history with another one.

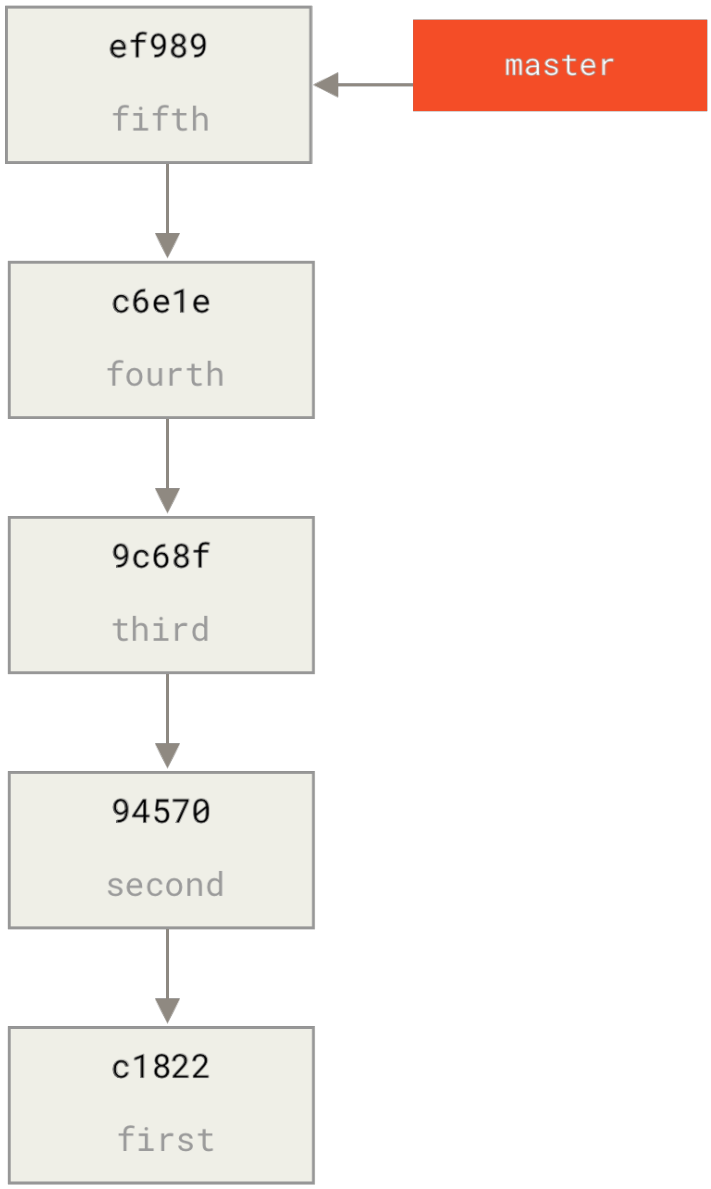
For example, let's say you have a huge code history and want to split your repository into one short history for new developers and one much longer and larger history for people interested in data mining. You can graft one history onto the other by `replace`ing the earliest commit in the new line with the latest commit on the older one. This is nice because it means that you don't actually have to rewrite every commit in the new history, as you would normally have to do to join them together (because the parentage affects the SHA-1s).

Let's try this out. Let's take an existing repository, split it into two repositories, one recent and one historical, and then we'll see how we can recombine them without modifying the recent repositories SHA-1 values via `replace`.

We'll use a simple repository with five simple commits:

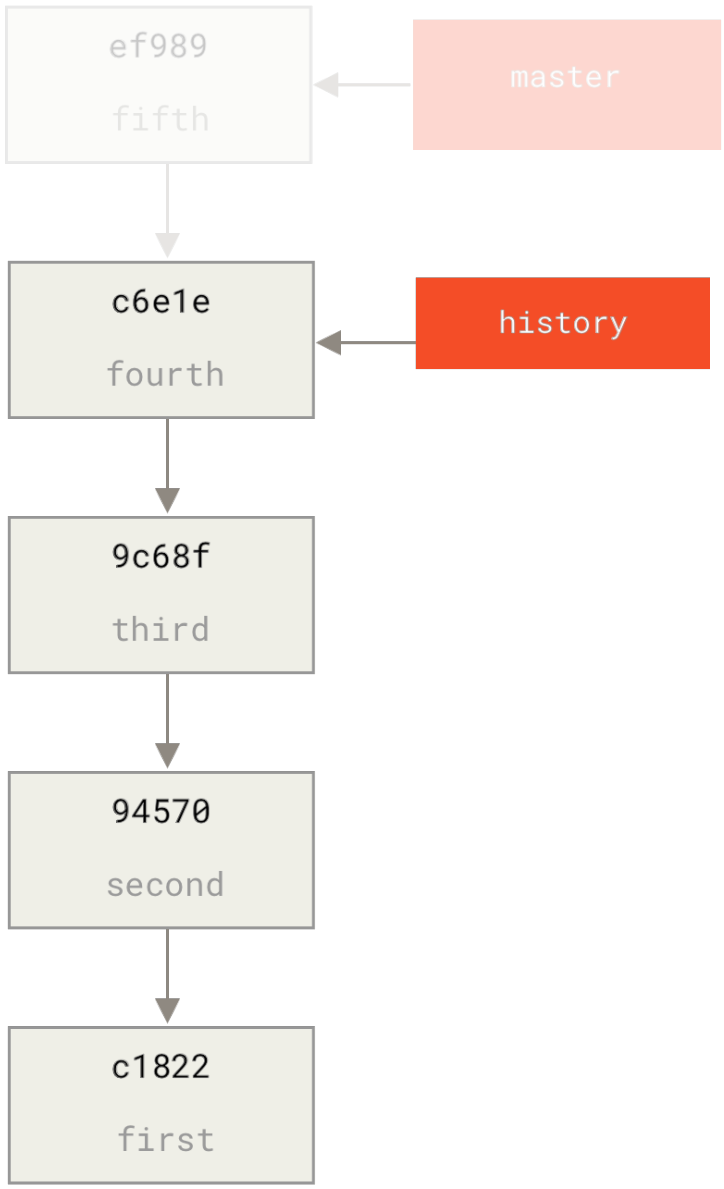
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

We want to break this up into two lines of history. One line goes from commit one to commit four - that will be the historical one. The second line will just be commits four and five - that will be the recent history.



Well, creating the historical history is easy, we can just put a branch in the history and then push that branch to the master branch of a new remote repository.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



Now we can push the new `history` branch to the `master` branch of our new repository:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

OK, so our history is published. Now the harder part is truncating our recent history down so it's

smaller. We need an overlap so we can replace a commit in one with an equivalent commit in the other, so we're going to truncate this to just commits four and five (so commit four overlaps).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

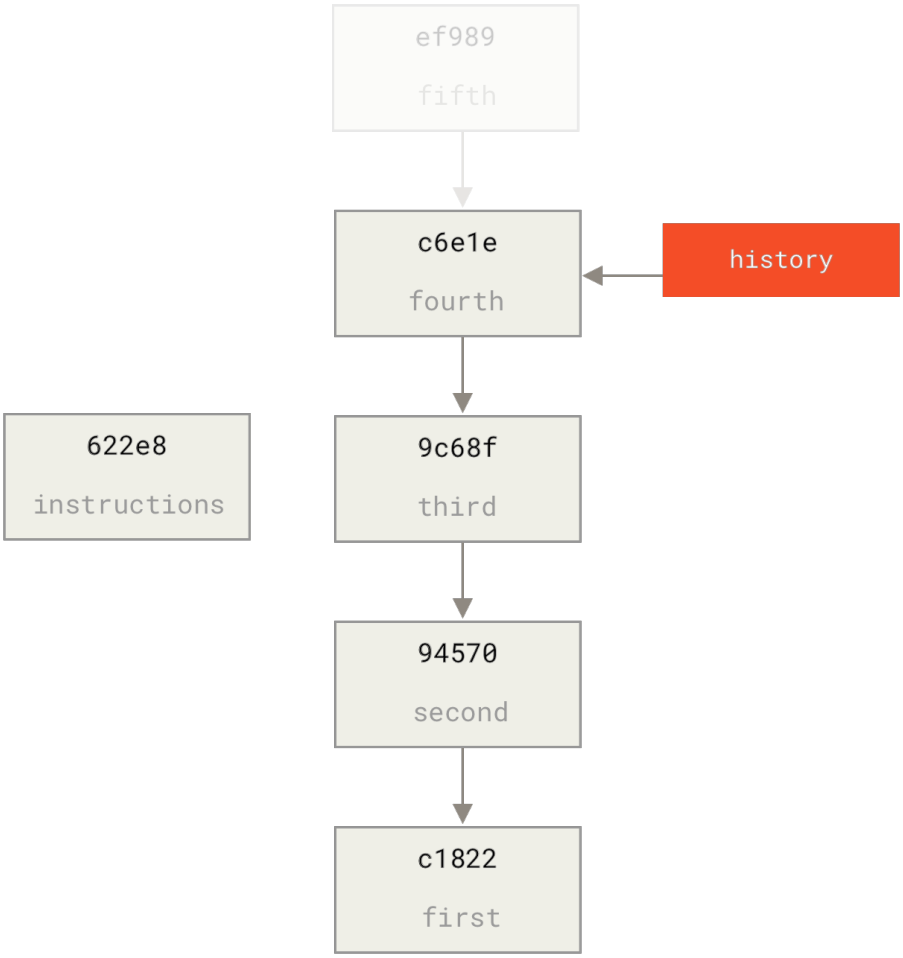
It's useful in this case to create a base commit that has instructions on how to expand the history, so other developers know what to do if they hit the first commit in the truncated history and need more. So, what we're going to do is create an initial commit object as our base point with instructions, then rebase the remaining commits (four and five) on top of it.

To do that, we need to choose a point to split at, which for us is the third commit, which is `9c68fdc` in SHA-speak. So, our base commit will be based off of that tree. We can create our base commit using the `commit-tree` command, which just takes a tree and will give us a brand new, parentless commit object SHA-1 back.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

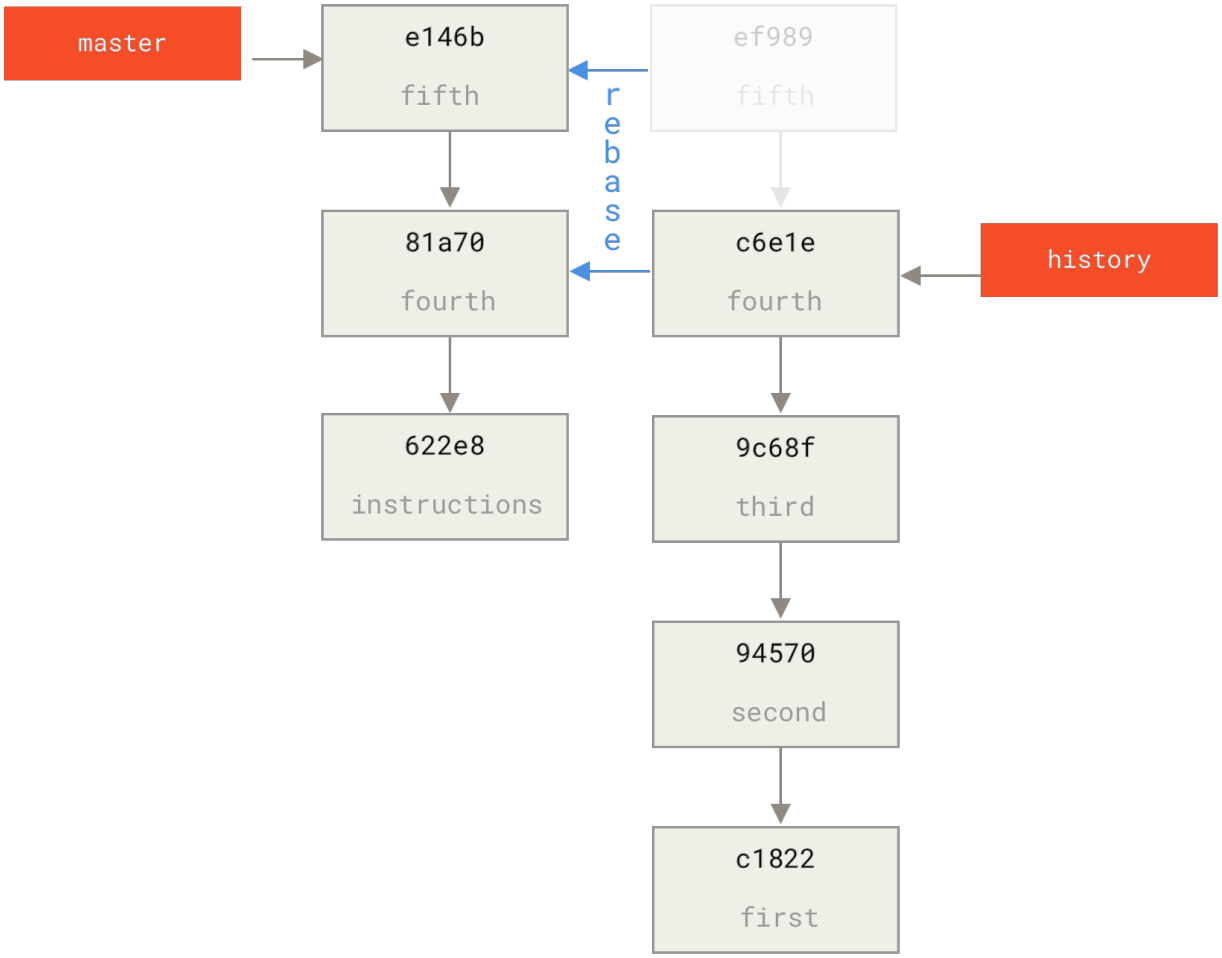
NOTA

The `commit-tree` command is one of a set of commands that are commonly referred to as *plumbing* commands. These are commands that are not generally meant to be used directly, but instead are used by **other** Git commands to do smaller jobs. On occasions when we're doing weirder things like this, they allow us to do really low-level things but are not meant for daily use. You can read more about plumbing commands in Encanamento e Porcelana



OK, so now that we have a base commit, we can rebase the rest of our history on top of that with `git rebase --onto`. The `--onto` argument will be the SHA-1 we just got back from `commit-tree` and the rebase point will be the third commit (the parent of the first commit we want to keep, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



OK, so now we've re-written our recent history on top of a throw away base commit that now has instructions in it on how to reconstitute the entire history if we wanted to. We can push that new history to a new project and now when people clone that repository, they will only see the most recent two commits and then a base commit with instructions.

Let's now switch roles to someone cloning the project for the first time who wants the entire history. To get the history data after cloning this truncated repository, one would have to add a second remote for the historical repository and fetch:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master     -> project-history/master
```

Now the collaborator would have their recent commits in the `master` branch and the historical commits in the `project-history/master` branch.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

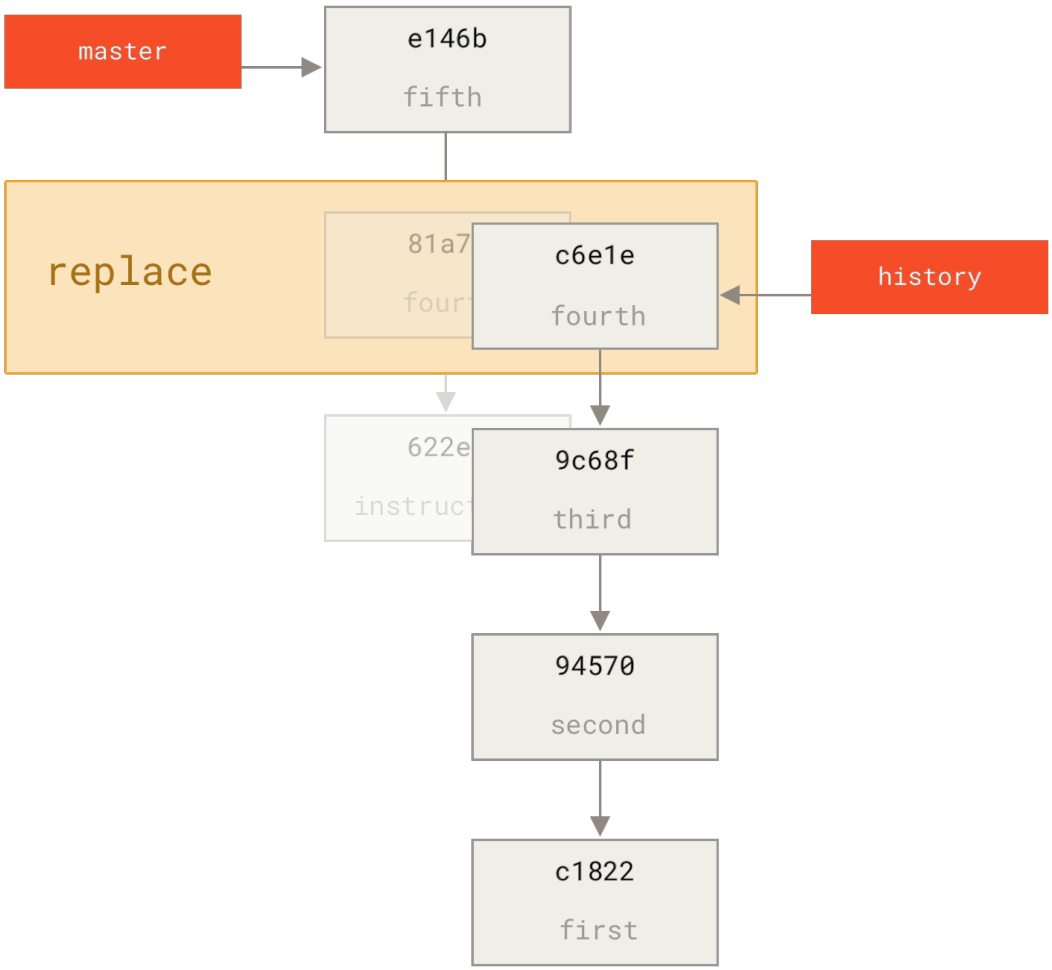
To combine them, you can simply call `git replace` with the commit you want to replace and then the commit you want to replace it with. So we want to replace the "fourth" commit in the master branch with the "fourth" commit in the `project-history/master` branch:

```
$ git replace 81a708d c6e1e95
```

Now, if you look at the history of the `master` branch, it appears to look like this:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Cool, right? Without having to change all the SHA-1s upstream, we were able to replace one commit in our history with an entirely different commit and all the normal tools (`bisect`, `blame`, etc) will work how we would expect them to.



Interestingly, it still shows 81a708d as the SHA-1, even though it's actually using the c6e1e95 commit data that we replaced it with. Even if you run a command like cat-file, it will show you the replaced data:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Remember that the actual parent of 81a708d was our placeholder commit (622e88e), not 9c68fdce as it states here.

Another interesting thing is that this data is kept in our references:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

This means that it's easy to share our replacement with others, because we can push this to our server and other people can easily download it. This is not that helpful in the history grafting scenario we've gone over here (since everyone would be downloading both histories anyhow, so why separate them?) but it can be useful in other circumstances.

Credential Storage

If you use the SSH transport for connecting to remotes, it's possible for you to have a key without a passphrase, which allows you to securely transfer data without typing in your username and password. However, this isn't possible with the HTTP protocols – every connection needs a username and password. This gets even harder for systems with two-factor authentication, where the token you use for a password is randomly generated and unpronounceable.

Fortunately, Git has a credentials system that can help with this. Git has a few options provided in the box:

- The default is not to cache at all. Every connection will prompt you for your username and password.
- The "cache" mode keeps credentials in memory for a certain period of time. None of the passwords are ever stored on disk, and they are purged from the cache after 15 minutes.
- The "store" mode saves the credentials to a plain-text file on disk, and they never expire. This means that until you change your password for the Git host, you won't ever have to type in your credentials again. The downside of this approach is that your passwords are stored in cleartext in a plain file in your home directory.
- If you're using a Mac, Git comes with an "osxkeychain" mode, which caches credentials in the secure keychain that's attached to your system account. This method stores the credentials on disk, and they never expire, but they're encrypted with the same system that stores HTTPS certificates and Safari auto-fills.
- If you're using Windows, you can install a helper called "wincred." This is similar to the "osxkeychain" helper described above, but uses the Windows Credential Store to control sensitive information.

You can choose one of these methods by setting a Git configuration value:

```
$ git config --global credential.helper cache
```

Some of these helpers have options. The "store" helper can take a `--file <path>` argument, which

customizes where the plain-text file is saved (the default is `~/.git-credentials`). The "cache" helper accepts the `--timeout <seconds>` option, which changes the amount of time its daemon is kept running (the default is "900", or 15 minutes). Here's an example of how you'd configure the "store" helper with a custom file name:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git even allows you to configure several helpers. When looking for credentials for a particular host, Git will query them in order, and stop after the first answer is provided. When saving credentials, Git will send the username and password to **all** of the helpers in the list, and they can choose what to do with them. Here's what a `.gitconfig` would look like if you had a credentials file on a thumb drive, but wanted to use the in-memory cache to save some typing if the drive isn't plugged in:

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Under the Hood

How does this all work? Git's root command for the credential-helper system is `git credential`, which takes a command as an argument, and then more input through stdin.

This might be easier to understand with an example. Let's say that a credential helper has been configured, and the helper has stored credentials for `mygithost`. Here's a session that uses the "fill" command, which is invoked when Git is trying to find credentials for a host:

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① This is the command line that initiates the interaction.

② Git-credential is then waiting for input on stdin. We provide it with the things we know: the protocol and hostname.

③ A blank line indicates that the input is complete, and the credential system should answer with what it knows.

④ Git-credential then takes over, and writes to stdout with the bits of information it found.

⑤ If credentials are not found, Git asks the user for the username and password, and provides them back to the invoking stdout (here they're attached to the same console).

The credential system is actually invoking a program that's separate from Git itself; which one and how depends on the `credential.helper` configuration value. There are several forms it can take:

Configuration Value	Behavior
`foo`	Runs `git-credential-foo`
`foo -a --opt=bcd`	Runs `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	Runs `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Code after `!` evaluated in shell

So the helpers described above are actually named `git-credential-cache`, `git-credential-store`, and so on, and we can configure them to take command-line arguments. The general form for this is "git-credential-foo [args] <action>." The stdin/stdout protocol is the same as git-credential, but they use a slightly different set of actions:

- `get` is a request for a username/password pair.
- `store` is a request to save a set of credentials in this helper's memory.
- `erase` purge the credentials for the given properties from this helper's memory.

For the `store` and `erase` actions, no response is required (Git ignores it anyway). For the `get` action, however, Git is very interested in what the helper has to say. If the helper doesn't know anything useful, it can simply exit with no output, but if it does know, it should augment the provided information with the information it has stored. The output is treated like a series of assignment statements; anything provided will replace what Git already knows.

Here's the same example from above, but skipping git-credential and going straight for git-credential-store:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

① Here we tell `git-credential-store` to save some credentials: the username "bob" and the password "s3cre7" are to be used when `https://mygithost` is accessed.

② Now we'll retrieve those credentials. We provide the parts of the connection we already know (`https://mygithost`), and an empty line.

③ `git-credential-store` replies with the username and password we stored above.

Here's what the `~/git.store` file looks like:

```
https://bob:s3cre7@mygithost
```

It's just a series of lines, each of which contains a credential-decorated URL. The `osxkeychain` and `wincred` helpers use the native format of their backing stores, while `cache` uses its own in-memory format (which no other process can read).

A Custom Credential Cache

Given that `git-credential-store` and friends are separate programs from Git, it's not much of a leap to realize that *any* program can be a Git credential helper. The helpers provided by Git cover many common use cases, but not all. For example, let's say your team has some credentials that are shared with the entire team, perhaps for deployment. These are stored in a shared directory, but you don't want to copy them to your own credential store, because they change often. None of the existing helpers cover this case; let's see what it would take to write our own. There are several key features this program needs to have:

1. The only action we need to pay attention to is `get`; `store` and `erase` are write operations, so we'll just exit cleanly when they're received.
2. The file format of the shared-credential file is the same as that used by `git-credential-store`.
3. The location of that file is fairly standard, but we should allow the user to pass a custom path just in case.

Once again, we'll write this extension in Ruby, but any language will work so long as Git can execute the finished product. Here's the full source code of our new credential helper:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline| ④
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] and user ==
known['username'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

① Here we parse the command-line options, allowing the user to specify the input file. The default is `~/.git-credentials`.

② This program only responds if the action is `get` and the backing-store file exists.

③ This loop reads from stdin until the first blank line is reached. The inputs are stored in the `known` hash for later reference.

④ This loop reads the contents of the storage file, looking for matches. If the protocol and host from `known` match this line, the program prints the results to stdout and exits.

We'll save our helper as `git-credential-read-only`, put it somewhere in our `PATH` and mark it executable. Here's what an interactive session looks like:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Since its name starts with “git-”, we can use the simple syntax for the configuration value:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

As you can see, extending this system is pretty straightforward, and can solve some common problems for you and your team.

Summary

You’ve seen a number of advanced tools that allow you to manipulate your commits and staging area more precisely. When you notice issues, you should be able to easily figure out what commit introduced them, when, and by whom. If you want to use subprojects in your project, you’ve learned how to accommodate those needs. At this point, you should be able to do most of the things in Git that you’ll need on the command line day to day and feel comfortable doing so.

Customizing Git

So far, we've covered the basics of how Git works and how to use it, and we've introduced a number of tools that Git provides to help you use it easily and efficiently. In this chapter, we'll see how you can make Git operate in a more customized fashion, by introducing several important configuration settings and the hooks system. With these tools, it's easy to get Git to work exactly the way you, your company, or your group needs it to.

Git Configuration

As you briefly saw in Começando, you can specify Git configuration settings with the `git config` command. One of the first things you did was set up your name and email address:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Now you'll learn a few of the more interesting options that you can set in this manner to customize your Git usage.

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in an `/etc/gitconfig` file, which contains values for every user on the system and all of their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.

The next place Git looks is the `~/.gitconfig` (or `~/.config/git/config`) file, which is specific to each user. You can make Git read and write to this file by passing the `--global` option.

Finally, Git looks for configuration values in the configuration file in the Git directory (`.git/config`) of whatever repository you're currently using. These values are specific to that single repository.

Each of these "levels" (system, global, local) overwrites values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`, for instance.

NOTA	Git's configuration files are plain-text, so you can also set these values by manually editing the file and inserting the correct syntax. It's generally easier to run the `git config` command, though.

Basic Client Configuration

The configuration options recognized by Git fall into two categories: client-side and server-side. The majority of the options are client-side – configuring your personal working preferences. Many, *many* configuration options are supported, but a large fraction of them are only useful in certain edge cases. We'll only be covering the most common and most useful here. If you want to see a list of all the options your version of Git recognizes, you can run

```
$ man git-config
```

This command lists all the available options in quite a bit of detail. You can also find this reference material at http://git-scm.com/docs/git-config.html.

`core.editor`

By default, Git uses whatever you've set as your default text editor (`$VISUAL` or `$EDITOR`) or else falls back to the `vi` editor to create and edit your commit and tag messages. To change that default to something else, you can use the `core.editor` setting:

```
$ git config --global core.editor emacs
```

Now, no matter what is set as your default shell editor, Git will fire up Emacs to edit messages.

`commit.template`

If you set this to the path of a file on your system, Git will use that file as the default message when you commit. For instance, suppose you create a template file at `~/.gitmessage.txt` that looks like this:

```
subject line

what happened

[ticket: X]
```

To tell Git to use it as the default message that appears in your editor when you run `git commit`, set the `commit.template` configuration value:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

Then, your editor will open to something like this for your placeholder commit message when you commit:

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

If your team has a commit-message policy, then putting a template for that policy on your system and configuring Git to use it by default can help increase the chance of that policy being followed regularly.

`core.pager`

This setting determines which pager is used when Git pages output such as `log` and `diff`. You can set it to `more` or to your favorite pager (by default, it's `less`), or you can turn it off by setting it to a blank string:

```
$ git config --global core.pager ''
```

If you run that, Git will page the entire output of all commands, no matter how long they are.

`user.signingkey`

If you're making signed annotated tags (as discussed in Signing Your Work), setting your GPG signing key as a configuration setting makes things easier. Set your key ID like so:

```
$ git config --global user.signingkey <gpg-key-id>
```

Now, you can sign tags without having to specify your key every time with the `git tag` command:

```
$ git tag -s <tag-name>
```

`core.excludesfile`

You can put patterns in your project's `.gitignore` file to have Git not see them as untracked files or try to stage them when you run `git add` on them, as discussed in Ignorando Arquivos.

But sometimes you want to ignore certain files for all repositories that you work with. If your computer is running Mac OS X, you're probably familiar with `.DS_Store` files. If your preferred editor is Emacs or Vim, you know about filenames that end with a `~` or `.swp`.

This setting lets you write a kind of global `.gitignore` file. If you create a `~/.gitignore_global` file with these contents:

```
*~
.*.swp
.DS_Store
```

...and you run `git config --global core.excludesfile ~/.gitignore_global`, Git will never again bother you about those files.

`help.autocorrect`

If you mistype a command, it shows you something like this:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
    checkout
```

Git helpfully tries to figure out what you meant, but it still refuses to do it. If you set `help.autocorrect` to 1, Git will actually run this command for you:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Note that "0.1 seconds" business. `help.autocorrect` is actually an integer which represents tenths of a second. So if you set it to 50, Git will give you 5 seconds to change your mind before executing the autocorrected command.

Colors in Git

Git fully supports colored terminal output, which greatly aids in visually parsing command output quickly and easily. A number of options can help you set the coloring to your preference.

`color.ui`

Git automatically colors most of its output, but there's a master switch if you don't like this behavior. To turn off all Git's colored terminal output, do this:

```
$ git config --global color.ui false
```

The default setting is `auto`, which colors output when it's going straight to a terminal, but omits the color-control codes when the output is redirected to a pipe or a file.

You can also set it to `always` to ignore the difference between terminals and pipes. You'll rarely want this; in most scenarios, if you want color codes in your redirected output, you can instead pass a `--color` flag to the Git command to force it to use color codes. The default setting is almost always what you'll want.

`color.*`

If you want to be more specific about which commands are colored and how, Git provides verb-specific coloring settings. Each of these can be set to `true`, `false`, or `always`:

```
color.branch
color.diff
color.interactive
color.status
```

In addition, each of these has subsettings you can use to set specific colors for parts of the output, if you want to override each color. For example, to set the meta information in your diff output to blue foreground, black background, and bold text, you can run

```
$ git config --global color.diff.meta "blue black bold"
```

You can set the color to any of the following values: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`. If you want an attribute like bold in the previous example, you can choose from `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap foreground and background).

External Merge and Diff Tools

Although Git has an internal implementation of diff, which is what we've been showing in this book, you can set up an external tool instead. You can also set up a graphical merge-conflict-resolution tool instead of having to resolve conflicts manually. We'll demonstrate setting up the Perforce Visual Merge Tool (P4Merge) to do your diffs and merge resolutions, because it's a nice graphical tool and it's free.

If you want to try this out, P4Merge works on all major platforms, so you should be able to do so. We'll use path names in the examples that work on Mac and Linux systems; for Windows, you'll have to change `/usr/local/bin` to an executable path in your environment.

To begin, download P4Merge from Perforce. Next, you'll set up external wrapper scripts to run your commands. We'll use the Mac path for the executable; in other systems, it will be where your `p4merge` binary is installed. Set up a merge wrapper script named `extMerge` that calls your binary with all the arguments provided:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

The diff wrapper checks to make sure seven arguments are provided and passes two of them to your merge script. By default, Git passes the following arguments to the diff program:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Because you only want the `old-file` and `new-file` arguments, you use the wrapper script to pass the ones you need.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

You also need to make sure these tools are executable:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Now you can set up your config file to use your custom merge resolution and diff tools. This takes a number of custom settings: `merge.tool` to tell Git what strategy to use, `mergetool.<tool>.cmd` to specify how to run the command, `mergetool.<tool>.trustExitCode` to tell Git if the exit code of that program indicates a successful merge resolution or not, and `diff.external` to tell Git what command to run for diffs. So, you can either run four config commands

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

or you can edit your `~/.gitconfig` file to add these lines:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

After all this is set, if you run diff commands such as this:

```
$ git diff 32d1776b1^ 32d1776b1
```

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like this:

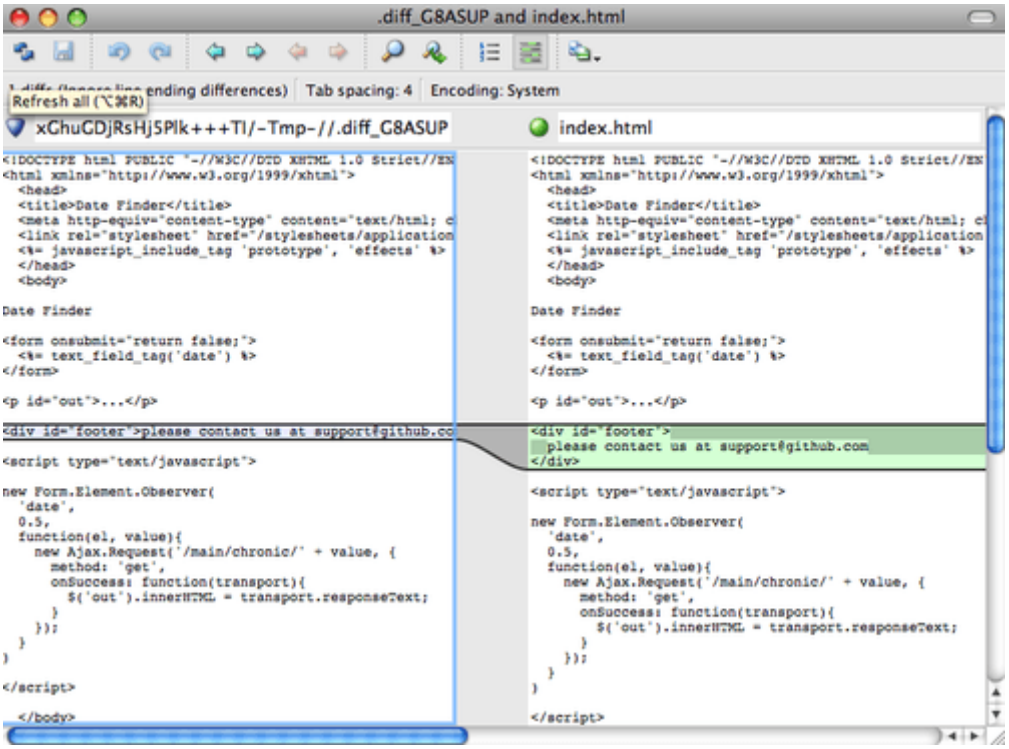


Figura 143. P4Merge.

If you try to merge two branches and subsequently have merge conflicts, you can run the command `git mergetool`; it starts P4Merge to let you resolve the conflicts through that GUI tool.

The nice thing about this wrapper setup is that you can change your diff and merge tools easily. For example, to change your `extDiff` and `extMerge` tools to run the KDiff3 tool instead, all you have to do is edit your `extMerge` file:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Now, Git will use the KDiff3 tool for diff viewing and merge conflict resolution.

Git comes preset to use a number of other merge-resolution tools without your having to set up the cmd configuration. To see a list of the tools it supports, try this:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        emerge
        gvimdiff
        gvimdiff2
        opendiff
        p4merge
        vimdiff
        vimdiff2

The following tools are valid, but not currently available:
        araxis
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        kdiff3
        meld
        tkdiff
        tortoisemerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

If you're not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the kdiff3 command is in your path, then you can run

```
$ git config --global merge.tool kdiff3
```

If you run this instead of setting up the `extMerge` and `extDiff` files, Git will use KDiff3 for merge resolution and the normal Git diff tool for diffs.

Formatting and Whitespace

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git has a few configuration options to help with these issues.

`core.autocrlf`

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems

use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the `core.autocrlf` setting. If you're on a Windows machine, set it to `true` – this converts LF endings into CRLF when you check out code:

```
$ git config --global core.autocrlf true
```

If you're on a Linux or Mac system that uses LF line endings, then you don't want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting `core.autocrlf` to input:

```
$ git config --global core.autocrlf input
```

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to `false`:

```
$ git config --global core.autocrlf false
```

`core.whitespace`

Git comes preset to detect and fix some whitespace issues. It can look for six primary whitespace issues – three are enabled by default and can be turned off, and three are disabled by default but can be activated.

The three that are turned on by default are `blank-at-eol`, which looks for spaces at the end of a line; `blank-at-eof`, which notices blank lines at the end of a file; and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The three that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with spaces instead of tabs (and is controlled by the `tabwidth` option); `tab-in-indent`, which watches for tabs in the indentation portion of a line; and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting `core.whitespace` to the values you want on or off, separated by commas. You can disable an option by prepending a `-` in front of its name, or use the default value by leaving it out of the setting string entirely. For example, if you want all but `space-before-tab` to be set, you can do this (with `trailing-space` being a short-hand to cover both `blank-at-eol` and `blank-at-eof`):

```
$ git config --global core.whitespace \
    trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Or you can specify the customizing part only:

```
$ git config --global core.whitespace \
    -space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Git will detect these issues when you run a `git diff` command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with `git apply`. When you're applying patches, you can ask Git to warn you if it's applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Or you can have Git try to automatically fix the issue before applying the patch:

```
$ git apply --whitespace=fix <patch>
```

These options apply to the `git rebase` command as well. If you've committed whitespace issues but haven't yet pushed upstream, you can run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it's rewriting the patches.

Server Configuration

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

`receive.fsckObjects`

Git is capable of making sure every object received during a push still matches its SHA-1 checksum and points to valid objects. However, it doesn't do this by default; it's a fairly expensive operation, and might slow down the operation, especially on large repositories or pushes. If you want Git to check object consistency on every push, you can force it to do so by setting `receive.fsckObjects` to true:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty (or malicious) clients aren't introducing corrupt data.

`receive.denyNonFastForwards`

If you rebase commits that you've already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn't contain the commit that the remote branch

currently points to, you'll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you're doing and can force-update the remote branch with a `-f` flag to your push command.

To tell Git to refuse force-pushes, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which we'll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

`receive.denyDeletes`

One of the workarounds to the `denyNonFastForwards` policy is for the user to delete the branch and then push it back up with the new reference. To avoid this, set `receive.denyDeletes` to true:

```
$ git config --system receive.denyDeletes true
```

This denies any deletion of branches or tags – no user can do it. To remove remote branches, you must remove the ref files from the server manually. There are also more interesting ways to do this on a per-user basis via ACLs, as you'll learn in An Example Git-Enforced Policy.

Git Attributes

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don't want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you'll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

Binary Files

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed. You'll see how to tell Git which is which.

Identifying Binary Files

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on the Mac contain a file that ends in `.pbxproj`, which is basically a JSON (plain-text JavaScript data format) dataset written out to disk by the IDE, which records your build settings and so on. Although it's technically a text file (because it's all UTF-8), you don't want to treat

it as such because it's really a lightweight database – you can't merge the contents if two people change it, and diffs generally aren't helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all `pbxproj` files as binary data, add the following line to your `.gitattributes` file:

```
*.pbxproj binary
```

Now, Git won't try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project.

Diffing Binary Files

You can also use the Git attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff.

First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific editor around, but oddly, everyone still uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run `git diff` normally, you only see something like this:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your `.gitattributes` file:

```
*.docx diff=word
```

This tells Git that any file that matches this pattern (`.docx`) should use the "word" filter when you try to view a diff that contains changes. What is the "word" filter? You have to set it up. Here you'll configure Git to use the `docx2txt` program to convert Word documents into readable text files, which it will then diff properly.

First, you'll need to install `docx2txt`; you can download it from `http://docx2txt.sourceforge.net`. Follow the instructions in the `INSTALL` file to put it somewhere your shell can find it. Next, you'll write a wrapper script to convert output to the format Git expects. Create a file that's somewhere in your path called `docx2txt`, and add these contents:

```
#!/bin/bash
docx2txt.pl "$1" -
```

Don't forget to `chmod a+x` that file. Finally, you can configure Git to use this script:

```
$ git config diff.word.textconv docx2txt
```

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in `.docx`, it should run those files through the "word" filter, which is defined as the `docx2txt` program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here's an example: Chapter 1 of this book was converted to Word format and committed in a Git repository. Then a new paragraph was added. Here's what `git diff` shows:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning
by explaining some background on version control tools, then move on to how to get Git
running on your system and finally how to get it setup to start working with. At the
end of this chapter you should understand why Git is around, why you should use it and
you should be all setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that
records changes to a file or set of files over time so that you can recall specific
versions later. For the examples in this book you will use software source code as the
files being version controlled, though in reality you can do this with nearly any type
of file on a computer.
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or
layout (which you would most certainly want to), a Version Control System (VCS) is a
very wise thing to use. It allows you to revert files back to a previous state, revert
the entire project back to a previous state, compare changes over time, see who last
modified something that might be causing a problem, who introduced an issue and when,
and more. Using a VCS also generally means that if you screw things up or lose files,
you can easily recover. In addition, you get all this for very little overhead.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another
directory (perhaps a time-stamped directory, if they're clever). This approach is very
common because it is so simple, but it is also incredibly error prone. It is easy to
forget which directory you're in and accidentally write to the wrong file or copy over
files you don't mean to.
```

Git successfully and succinctly tells us that we added the string "Testing: 1, 2, 3.", which is correct. It's not perfect – formatting changes wouldn't show up here – but it certainly works.

Another interesting problem you can solve this way involves diffing image files. One way to do this is to run image files through a filter that extracts their EXIF information – metadata that is recorded

with most image formats. If you download and install the `exiftool` program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened. Put the following line in your `.gitattributes` file:

```
*.png diff=exif
```

Configure Git to use this tool:

```
$ git config diff.exif.textconv exiftool
```

If you replace an image in your project and run `git diff`, you see something like this:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

You can easily see that the file size and image dimensions have both changed.

Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked out and remove it again before it's added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an `$Id$` field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It's important to notice that it isn't the SHA-1 of the commit, but of the blob itself. Put the following line in your `.gitattributes` file:

```
*.txt ident
```

Add an `$Id$` reference to a test file:

```
$ echo '$Id$' > test.txt
```

The next time you check out this file, Git injects the SHA-1 of the blob:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

However, that result is of limited use. If you've used keyword substitution in CVS or Subversion, you can include a datestamp – the SHA-1 isn't all that helpful, because it's fairly random and you can't tell if one SHA-1 is older or newer than another just by looking at them.

It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are called "clean" and "smudge" filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they're checked out ("smudge", see The "smudge" filter is run on checkout.) and just before they're staged ("clean", see The "clean" filter is run when files are staged.). These filters can be set to do all sorts of fun things.

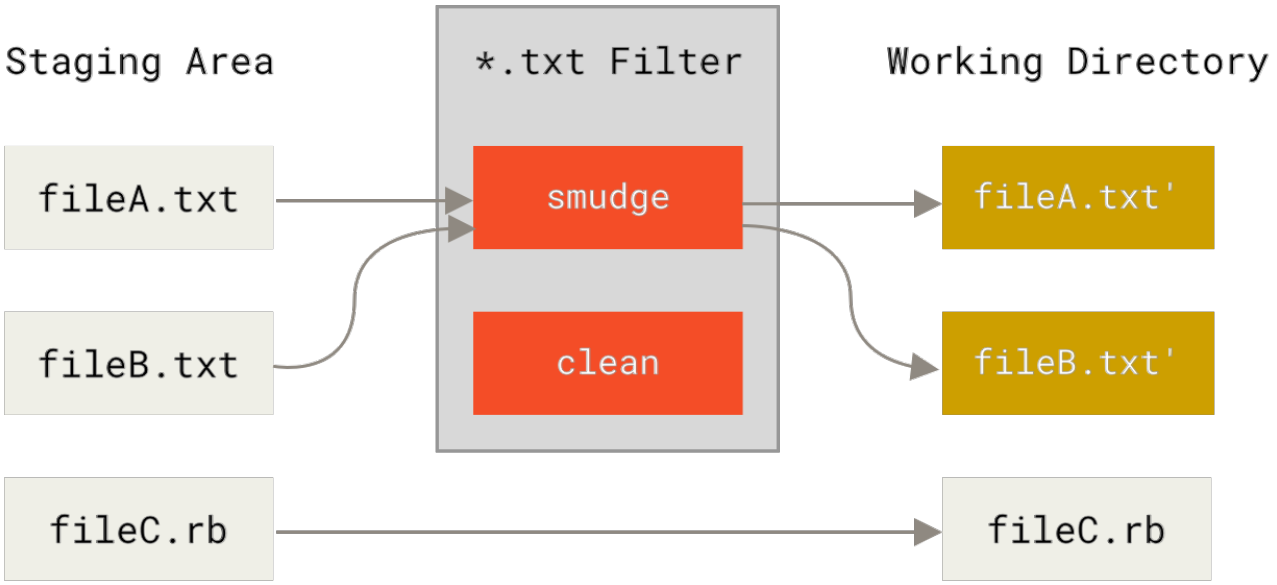


Figura 144. The "smudge" filter is run on checkout.

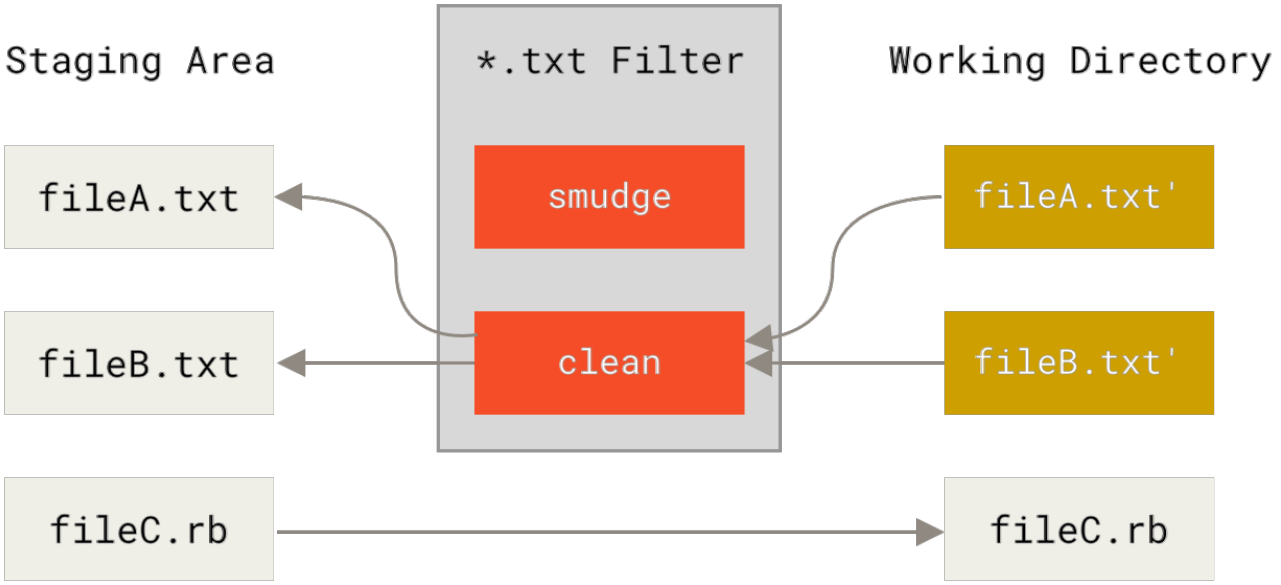


Figura 145. The "clean" filter is run when files are staged.

The original commit message for this feature gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the filter attribute in your `.gitattributes` file to filter `*.c` files with the "indent" filter:

```
*.c filter=indent
```

Then, tell Git what the "indent" filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the indent program before it stages them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program does essentially nothing: it spits out the same data that it comes in. This combination effectively filters all C source code files through `indent` before committing.

Another interesting example gets `$Date$` keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results – it should be simple to do in whatever language you're most comfortable in. You can name this file `expand_date` and put it in your path. Now, you need to

set up a filter in Git (call it `dater`) and tell it to use your `expand_date` filter to smudge the files on checkout. You'll use a Perl expression to clean that up on commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

This Perl snippet strips out anything it sees in a `$Date$` string, to get back to where you started. Now that your filter is ready, you can test it by setting up a Git attribute for that file that engages the new filter and creating a file with your `$Date$` keyword:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

If you commit those changes and check out the file again, you see the keyword properly substituted:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project, but the driver (in this case, `dater`) isn't, so it won't work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

Exporting Your Repository

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

`export-ignore`

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don't want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

```
test/ export-ignore
```

Now, when you run `git archive` to create a tarball of your project, that directory won't be included in the archive.

`export-subst`

When exporting files for deployment you can apply `git log`'s formatting and keyword-expansion processing to selected portions of files marked with the `export-subst` attribute.

For instance, if you want to include a file named `LAST_COMMIT` in your project, and have metadata about the last commit automatically injected into it when `git archive` runs, you can for example set up your `.gitattributes` and `LAST_COMMIT` files like this:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of the archived file will look like this:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

The substitutions can include for example the commit message and any `git notes`, and `git log` can do simple word wrapping:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log'\''s custom formatter

git archive uses git log'\''s `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.
'
$ git archive @ | tar xfO - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
       export-subst uses git log's custom formatter

         git archive uses git log's `pretty=format:` processor directly, and
         strips the surrounding `$Format:` and `$` markup from the output.
```

The resulting archive is suitable for deployment work, but like any exported archive it isn't suitable for further development work.

Merge Strategies

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

```
database.xml merge=ours
```

And then define a dummy `ours` merge strategy with:

```
$ git config --global merge.ours.driver true
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

Installing a Hook

The hooks are all stored in the `hooks` subdirectory of the Git directory. In most projects, that's `.git/hooks`. When you initialize a new repository with `git init`, Git populates the hooks directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine – you can write them in Ruby or Python or whatever language you are familiar with. If you want to use the bundled hook scripts, you'll have to rename them; their file names all end with `.sample`.

To enable a hook script, put a file in the `hooks` subdirectory of your .git directory that is named appropriately (without any extension) and is executable. From that point forward, it should be

called. We'll cover most of the major hook filenames here.

Client-Side Hooks

There are a lot of client-side hooks. This section splits them into committing-workflow hooks, email-workflow scripts, and everything else.

> **NOTA** It's important to note that client-side hooks are **not** copied when you clone a repository. If your intent with these scripts is to enforce a policy, you'll probably want to do that on the server side; see the example in An Example Git-Enforced Policy.

Committing-Workflow Hooks

The first four hooks have to do with the committing process.

The `pre-commit` hook is run first, before you even type in a commit message. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with `git commit --no-verify`. You can do things like check for code style (run `lint` or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

The `prepare-commit-msg` hook is run before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the commit author sees it. This hook takes a few parameters: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn't useful for normal commits; rather, it's good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert information.

The `commit-msg` hook takes one parameter, which again is the path to a temporary file that contains the commit message written by the developer. If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through. In the last section of this chapter, We'll demonstrate using this hook to check that your commit message is conformant to a required pattern.

After the entire commit process is completed, the `post-commit` hook runs. It doesn't take any parameters, but you can easily get the last commit by running `git log -1 HEAD`. Generally, this script is used for notification or something similar.

Email Workflow Hooks

You can set up three client-side hooks for an email-based workflow. They're all invoked by the `git am` command, so if you aren't using that command in your workflow, you can safely skip to the next section. If you're taking patches over email prepared by `git format-patch`, then some of these may be helpful to you.

The first hook that is run is `applypatch-msg`. It takes a single argument: the name of the temporary

file that contains the proposed commit message. Git aborts the patch if this script exits non-zero. You can use this to make sure a commit message is properly formatted, or to normalize the message by having the script edit it in place.

The next hook to run when applying patches via `git am` is `pre-applypatch`. Somewhat confusingly, it is run *after* the patch is applied but before a commit is made, so you can use it to inspect the snapshot before making the commit. You can run tests or otherwise inspect the working tree with this script. If something is missing or the tests don't pass, exiting non-zero aborts the `git am` script without committing the patch.

The last hook to run during a `git am` operation is `post-applypatch`, which runs after the commit is made. You can use it to notify a group or the author of the patch you pulled in that you've done so. You can't stop the patching process with this script.

Other Client Hooks

The `pre-rebase` hook runs before you rebase anything and can halt the process by exiting non-zero. You can use this hook to disallow rebasing any commits that have already been pushed. The example `pre-rebase` hook that Git installs does this, although it makes some assumptions that may not match with your workflow.

The `post-rewrite` hook is run by commands that replace commits, such as `git commit --amend` and `git rebase` (though not by `git filter-branch`). Its single argument is which command triggered the rewrite, and it receives a list of rewrites on `stdin`. This hook has many of the same uses as the `post-checkout` and `post-merge` hooks.

After you run a successful `git checkout`, the `post-checkout` hook runs; you can use it to set up your working directory properly for your project environment. This may mean moving in large binary files that you don't want source controlled, auto-generating documentation, or something along those lines.

The `post-merge` hook runs after a successful `merge` command. You can use it to restore data in the working tree that Git can't track, such as permissions data. This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.

The `pre-push` hook runs during `git push`, after the remote refs have been updated but before any objects have been transferred. It receives the name and location of the remote as parameters, and a list of to-be-updated refs through `stdin`. You can use it to validate a set of ref updates before a push occurs (a non-zero exit code will abort the push).

Git occasionally does garbage collection as part of its normal operation, by invoking `git gc --auto`. The `pre-auto-gc` hook is invoked just before the garbage collection takes place, and can be used to notify you that this is happening, or to abort the collection if now isn't a good time.

Server-Side Hooks

In addition to the client-side hooks, you can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project. These scripts run before and after pushes to the server. The pre hooks can exit non-zero at any time to reject the push as well as

print an error message back to the client; you can set up a push policy that's as complex as you wish.

`pre-receive`

The first script to run when handling a push from a client is `pre-receive`. It takes a list of references that are being pushed from stdin; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they're modifying with the push.

`update`

The `update` script is very similar to the `pre-receive` script, except that it's run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, `pre-receive` runs only once, whereas update runs once per branch they're pushing to. Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push. If the update script exits non-zero, only that reference is rejected; other references can still be updated.

`post-receive`

The `post-receive` hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same stdin data as the `pre-receive` hook. Examples include emailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can't stop the push process, but the client doesn't disconnect until it has completed, so be careful if you try to do anything that may take a long time.

An Example Git-Enforced Policy

In this section, you'll use what you've learned to establish a Git workflow that checks for a custom commit message format, and allows only certain users to modify certain subdirectories in a project. You'll build client scripts that help the developer know if their push will be rejected and server scripts that actually enforce the policies.

The scripts we'll show are written in Ruby; partly because of our intellectual inertia, but also because Ruby is easy to read, even if you can't necessarily write it. However, any language will work – all the sample hook scripts distributed with Git are in either Perl or Bash, so you can also see plenty of examples of hooks in those languages by looking at the samples.

Server-Side Hook

All the server-side work will go into the `update` file in your `hooks` directory. The `update` hook runs once per branch being pushed and takes three arguments:

- The name of the reference being pushed to
- The old revision where that branch was
- The new revision being pushed

You also have access to the user doing the pushing if the push is being run over SSH. If you've allowed everyone to connect with a single user (like "git") via public-key authentication, you may have to give that user a shell wrapper that determines which user is connecting based on the public key, and set an environment variable accordingly. Here we'll assume the connecting user is in the `$USER` environment variable, so your update script begins by gathering all the information you need:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Yes, those are global variables. Don't judge – it's easier to demonstrate this way.

Enforcing a Specific Commit-Message Format

Your first challenge is to enforce that each commit message adheres to a particular format. Just to have a target, assume that each message has to include a string that looks like "ref: 1234" because you want each commit to link to a work item in your ticketing system. You must look at each commit being pushed up, see if that string is in the commit message, and, if the string is absent from any of the commits, exit non-zero so the push is rejected.

You can get a list of the SHA-1 values of all the commits that are being pushed by taking the `$newrev` and `$oldrev` values and passing them to a Git plumbing command called `git rev-list`. This is basically the `git log` command, but by default it prints out only the SHA-1 values and no other information. So, to get a list of all the commit SHA-1s introduced between one commit SHA-1 and another, you can run something like this:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

You can take that output, loop through each of those commit SHA-1s, grab the message for it, and test that message against a regular expression that looks for a pattern.

You have to figure out how to get the commit message from each of these commits to test. To get the raw commit data, you can use another plumbing command called `git cat-file`. We'll go over all these plumbing commands in detail in Funcionamento Interno do Git; but for now, here's what that command gives you:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

A simple way to get the commit message from a commit when you have the SHA-1 value is to go to the first blank line and take everything after that. You can do so with the `sed` command on Unix systems:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

You can use that incantation to grab the commit message from each commit that is trying to be pushed and exit if you see anything that doesn't match. To exit the script and reject the push, exit non-zero. The whole method looks like this:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Putting that in your `update` script will reject updates that contain commits that have messages that don't adhere to your rule.

Enforcing a User-Based ACL System

Suppose you want to add a mechanism that uses an access control list (ACL) that specifies which users are allowed to push changes to which parts of your projects. Some people have full access, and others can only push changes to certain subdirectories or specific files. To enforce this, you'll write those rules to a file named `acl` that lives in your bare Git repository on the server. You'll have the `update` hook look at those rules, see what files are being introduced for all the commits being pushed, and determine whether the user doing the push has access to update all those files.

The first thing you'll do is write your ACL. Here you'll use a format very much like the CVS ACL

mechanism: it uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank meaning open access). All of these fields are delimited by a pipe (`|`) character.

In this case, you have a couple of administrators, some documentation writers with access to the `doc` directory, and one developer who only has access to the `lib` and `tests` directories, and your ACL file looks like this:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

You begin by reading this data into a structure that you can use. In this case, to keep the example simple, you'll only enforce the `avail` directives. Here is a method that gives you an associative array where the key is the user name and the value is an array of paths to which the user has write access:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

On the ACL file you looked at earlier, this `get_acl_access_data` method returns a data structure that looks like this:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, you need to determine what paths the commits

being pushed have modified, so you can make sure the user who's pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in Fundamentos de Git):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

If you use the ACL structure returned from the `get_acl_access_data` method and check it against the listed files in each of the commits, you can determine whether the user has access to push all of their commits:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # user has access to everything
           || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

You get a list of new commits being pushed to your server with `git rev-list`. Then, for each of those commits, you find which files are modified and make sure the user who's pushing has access to all the paths being modified.

Now your users can't push any commits with badly formed messages or with modified files outside of their designated paths.

Testing It Out

If you run `chmod u+x .git/hooks/update`, which is the file into which you should have put all this code, and then try to push a commit with a non-compliant message, you get something like this:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

There are a couple of interesting things here. First, you see this where the hook starts running.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Remember that you printed that out at the very beginning of your update script. Anything your script echoes to `stdout` will be transferred to the client.

The next thing you'll notice is the error message.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

The first line was printed out by you, the other two were Git telling you that the update script exited non-zero and that is what is declining your push. Lastly, you have this:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

You'll see a remote rejected message for each reference that your hook declined, and it tells you that it was declined specifically because of a hook failure.

Furthermore, if someone tries to edit a file they don't have access to and push a commit containing it, they will see something similar. For instance, if a documentation author tries to push a commit

modifying something in the `lib` directory, they see

```
[POLICY] You do not have access to push to lib/test.rb
```

From now on, as long as that `update` script is there and executable, your repository will never have a commit message without your pattern in it, and your users will be sandboxed.

Client-Side Hooks

The downside to this approach is the whining that will inevitably result when your users' commit pushes are rejected. Having their carefully crafted work rejected at the last minute can be extremely frustrating and confusing; and furthermore, they will have to edit their history to correct it, which isn't always for the faint of heart.

The answer to this dilemma is to provide some client-side hooks that users can run to notify them when they're doing something that the server is likely to reject. That way, they can correct any problems before committing and before those issues become more difficult to fix. Because hooks aren't transferred with a clone of a project, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but Git won't set them up automatically.

To begin, you should check your commit message just before each commit is recorded, so you know the server won't reject your changes due to badly formatted commit messages. To do this, you can add the `commit-msg` hook. If you have it read the message from the file passed as the first argument and compare that to the pattern, you can force Git to abort the commit if there is no match:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see this:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

No commit was completed in that instance. However, if your message contains the proper pattern, Git allows you to commit:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Next, you want to make sure you aren't modifying files that are outside your ACL scope. If your project's `.git` directory contains a copy of the ACL file you used previously, then the following `pre-commit` script will enforce those constraints for you:

```
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

This is roughly the same script as the server-side part, but with two important differences. First, the ACL file is in a different place, because this script runs from your working directory, not from your `.git` directory. You have to change the path to the ACL file from this

```
access = get_acl_access_data('acl')
```

to this:

```
access = get_acl_access_data('.git/acl')
```

The other important difference is the way you get a listing of the files that have been changed.

Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Instead of

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

you have to use

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But those are the only two differences – otherwise, the script works the same way. One caveat is that it expects you to be running locally as the same user you push as to the remote machine. If that is different, you must set the `$user` variable manually.

One other thing we can do here is make sure the user doesn't push non-fast-forwarded references. To get a reference that isn't a fast-forward, you either have to rebase past a commit you've already pushed up or try pushing a different local branch up to the same remote branch.

Presumably, the server is already configured with `receive.denyDeletes` and `receive.denyNonFastForwards` to enforce this policy, so the only accidental thing you can try to catch is rebasing commits that have already been pushed.

Here is an example pre-rebase script that checks for that. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that is reachable from one of your remote references, it aborts the rebase.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

This script uses a syntax that wasn't covered in Revision Selection. You get a list of commits that have already been pushed up by running this:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

The `SHA^@` syntax resolves to all the parents of that commit. You're looking for any commit that is reachable from the last commit on the remote and that isn't reachable from any parent of any of the SHA-1s you're trying to push up – meaning it's a fast-forward.

The main drawback to this approach is that it can be very slow and is often unnecessary – if you don't try to force the push with `-f`, the server will warn you and not accept the push. However, it's an interesting exercise and can in theory help you avoid a rebase that you might later have to go back and fix.

Summary

We've covered most of the major ways that you can customize your Git client and server to best fit your workflow and projects. You've learned about all sorts of configuration settings, file-based attributes, and event hooks, and you've built an example policy-enforcing server. You should now be able to make Git fit nearly any workflow you can dream up.

Git and Other Systems

The world isn't perfect. Usually, you can't immediately switch every project you come in contact with to Git. Sometimes you're stuck on a project using another VCS, and wish it was Git. We'll spend the first part of this chapter learning about ways to use Git as a client when the project you're working on is hosted in a different system.

At some point, you may want to convert your existing project to Git. The second part of this chapter covers how to migrate your project into Git from several specific systems, as well as a method that will work if no pre-built import tool exists.

Git as a Client

Git provides such a nice experience for developers that many people have figured out how to use it on their workstation, even if the rest of their team is using an entirely different VCS. There are a number of these adapters, called "bridges," available. Here we'll cover the ones you're most likely to run into in the wild.

Git and Subversion

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It's been around for more than a decade, and for most of that time was the *de facto* VCS choice for open-source projects. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on, while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

`git svn`

The base command in Git for all the Subversion bridging commands is `git svn`. It takes quite a few commands, so we'll show the most common while going through a few simple workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you're working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make

your life easier.

Setting Up

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn \
  http://your-svn-server.example.org/svn/
```

This sets up the properties to run the sync. You can then clone the code by running

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .............................[...]
Committed revision 2.
Copied properties for revision 2.
[…]
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository – it's ridiculously inefficient, but it's the only easy way to do this.

Getting Started

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You'll start with the `git svn clone` command, which imports an entire Subversion

repository into a local Git repository. Remember that if you're importing from a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
    A   m4/acx_pthread.m4
    A   m4/stl_hash.m4
    A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
    A   java/src/test/java/com/google/protobuf/WireFormatTest.java
…
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn't that big, but Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Note how this tool manages Subversion tags as remote refs. Let's take a closer look with the Git

plumbing command `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.

Committing Back to Subversion

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r77
    M   README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Notice that the SHA-1 checksum that originally started with `4af61fd` when you committed now begins with `95e0222`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

Pulling in New Changes

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M  README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M  README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M   README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Now, all your work is on top of what is on the Subversion server, so you can successfully `dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r85
    M   README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Note that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict (much like how Subversion works). If someone else pushes a change to one file and then you push a change to another file, your `dcommit` will work fine:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   configure.ac
Committed r87
    M   autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    M   configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ,
using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M  autogen.sh
First, rewinding head to replay your work on top of it...
```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server – in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```
$ git svn rebase
    M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Running `git svn rebase` every once in a while makes sure your code is always up to date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` – otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an `experiment` branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   CHANGES.txt
Committed r89
    M   CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
    M   COPYING.txt
    M   INSTALL.txt
Committed r90
    M   INSTALL.txt
    M   COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the `experiment` branch – instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it, as though you ran `git merge --squash`; they don't see the commit data about where it came from or when it was committed.

Subversion Branching

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using `git svn`.

Creating a New SVN Branch

To create a new branch in Subversion, you run `git svn branch [branchname]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that

branch; if you commit at this point, that commit will go to `trunk` on the server, not `opera`.

Switching Active Branches

Git figures out what branch your dcommits go to by looking for the tip of any of your Subversion branches in your history – you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an `opera` branch that you can work on separately, you can run

```
$ git branch opera remotes/origin/opera
```

Now, if you want to merge your `opera` branch into `trunk` (your `master` branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say "Merge branch opera" instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong – the dcommit makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this situation – Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, `opera`) after you merge it into trunk.

Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't dcommited don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two commands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to stdout the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

Git-Svn Summary

The `git svn` tools are useful if you're stuck with a Subversion server, or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

Git and Mercurial

The DVCS universe is larger than just Git. In fact, there are many other systems in this space, each with their own angle on how to do distributed version control correctly. Apart from Git, the most popular is Mercurial, and the two are very similar in many respects.

The good news, if you prefer Git's client-side behavior but are working with a project whose source code is controlled with Mercurial, is that there's a way to use Git as a client for a Mercurial-hosted repository. Since the way Git talks to server repositories is through remotes, it should come as no surprise that this bridge is implemented as a remote helper. The project's name is git-remote-hg, and it can be found at https://github.com/felipec/git-remote-hg.

git-remote-hg

First, you need to install git-remote-hg. This basically entails dropping its file somewhere in your path, like so:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...assuming `~/bin` is in your `$PATH`. Git-remote-hg has one other dependency: the `mercurial` library for Python. If you have Python installed, this is as simple as:

```
$ pip install mercurial
```

(If you don't have Python installed, visit https://www.python.org/ and get it first.)

The last thing you'll need is the Mercurial client. Go to https://www.mercurial-scm.org/ and install it if you haven't already.

Now you're ready to rock. All you need is a Mercurial repository you can push to. Fortunately, every Mercurial repository can act this way, so we'll just use the "hello world" repository everyone uses to learn Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Getting Started

Now that we have a suitable "server-side" repository, we can go through a typical workflow. As you'll see, these two systems are similar enough that there isn't much friction.

As always with Git, first we clone:

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard "hello, world" program
```

You'll notice that working with a Mercurial repository uses the standard `git clone` command. That's because git-remote-hg is working at a fairly low level, using a similar mechanism to how Git's HTTP/S protocol is implemented (remote helpers). Since Git and Mercurial are both designed for every client to have a full copy of the repository history, this command makes a full clone, including all the project's history, and does it fairly quickly.

The log command shows two commits, the latest of which is pointed to by a whole slew of refs. It turns out some of these aren't actually there. Let's take a look at what's actually in the `.git` directory:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags

9 directories, 5 files
```

Git-remote-hg is trying to make things more idiomatically Git-esque, but under the hood it's managing the conceptual mapping between two slightly different systems. The `refs/hg` directory is where the actual remote refs are stored. For example, the `refs/hg/origin/branches/default` is a Git ref file that contains the SHA-1 starting with "ac7955c", which is the commit that `master` points to. So the `refs/hg` directory is kind of like a fake `refs/remotes/origin`, but it has the added distinction between bookmarks and branches.

The `notes/hg` file is the starting point for how git-remote-hg maps Git commit hashes to Mercurial changeset IDs. Let's explore a bit:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f...  65bb417...
100644 blob 485e178...  ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

So `refs/notes/hg` points to a tree, which in the Git object database is a list of other objects with names. `git ls-tree` outputs the mode, type, object hash, and filename for items inside a tree. Once we dig down to one of the tree items, we find that inside it is a blob named "ac9117f" (the SHA-1 hash of the commit pointed to by `master`), with contents "0a04b98" (which is the ID of the Mercurial changeset at the tip of the `default` branch).

The good news is that we mostly don't have to worry about all of this. The typical workflow won't be very different from working with a Git remote.

There's one more thing we should attend to before we continue: ignores. Mercurial and Git use a very similar mechanism for this, but it's likely you don't want to actually commit a `.gitignore` file into a Mercurial repository. Fortunately, Git has a way to ignore files that's local to an on-disk repository, and the Mercurial format is compatible with Git, so you just have to copy it over:

```
$ cp .hgignore .git/info/exclude
```

The `.git/info/exclude` file acts just like a `.gitignore`, but isn't included in commits.

Workflow

Let's assume we've done some work and made some commits on the `master` branch, and you're ready to push it to the remote repository. Here's what our repository looks like right now:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Our `master` branch is two commits ahead of `origin/master`, but those two commits exist only on our local machine. Let's see if anyone else has been doing important work at the same time:

```
$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master     -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Since we used the `--all` flag, we see the "notes" refs that are used internally by git-remote-hg, but we can ignore them. The rest is what we expected; `origin/master` has advanced by one commit, and our history has now diverged. Unlike the other systems we work with in this chapter, Mercurial is capable of handling merges, so we're not going to do anything fancy.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Perfect. We run the tests and everything passes, so we're ready to share our work with the rest of the team:

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

That's it! If you take a look at the Mercurial repository, you'll see that this did what we'd expect:

```
$ hg log -G --style compact
o    5[tip]:4,2   dc8fa4f932b8   2014-08-14 19:33 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   64f27bcefc35   2014-08-14 19:27 -0700   ben
| |    Update makefile
| |
| o  3:1   4256fc29598f   2014-08-14 19:27 -0700   ben
| |    Goodbye
| |
@ |  2   7db0b4848b3c   2014-08-14 19:30 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

The changeset numbered *2* was made by Mercurial, and the changesets numbered *3* and *4* were made by git-remote-hg, by pushing commits made with Git.

Branches and Bookmarks

Git has only one kind of branch: a reference that moves when commits are made. In Mercurial, this kind of a reference is called a "bookmark," and it behaves in much the same way as a Git branch.

Mercurial's concept of a "branch" is more heavyweight. The branch that a changeset is made on is recorded *with the changeset*, which means it will always be in the repository history. Here's an example of a commit that was made on the `develop` branch:

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

Note the line that begins with "branch". Git can't really replicate this (and doesn't need to; both types of branch can be represented as a Git ref), but git-remote-hg needs to understand the difference, because Mercurial cares.

Creating Mercurial bookmarks is as easy as creating Git branches. On the Git side:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

That's all there is to it. On the Mercurial side, it looks like this:

```
$ hg bookmarks
   featureA                  5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]   8f65e5e02793   2014-08-14 20:06 -0700   ben
|    More documentation
|
o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| |    update makefile
| |
| o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |    goodbye
| |
o |  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

Note the new `[featureA]` tag on revision 5. These act exactly like Git branches on the Git side, with one exception: you can't delete a bookmark from the Git side (this is a limitation of remote helpers).

You can work on a "heavyweight" Mercurial branch also: just put a branch in the `branches` namespace:

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

Here's what that looks like on the Mercurial side:

```
$ hg branches
permanent                      7:a4529d07aad4
develop                        6:8f65e5e02793
default                        5:bd5ac26f11f9 (inactive)
$ hg log -G
o  changeset:   7:a4529d07aad4
|  branch:      permanent
|  tag:         tip
|  parent:      5:bd5ac26f11f9
|  user:        Ben Straub <ben@straub.cc>
|  date:        Thu Aug 14 20:21:09 2014 -0700
|  summary:     A permanent change
|
| @  changeset:   6:8f65e5e02793
|/   branch:      develop
|    user:        Ben Straub <ben@straub.cc>
|    date:        Thu Aug 14 20:06:38 2014 -0700
|    summary:     More documentation
|
o    changeset:   5:bd5ac26f11f9
|\   bookmark:    featureA
| |  parent:      4:0434aaa6b91f
| |  parent:      2:f098c7f45c4f
| |  user:        Ben Straub <ben@straub.cc>
| |  date:        Thu Aug 14 20:02:21 2014 -0700
| |  summary:     Merge remote-tracking branch 'origin/master'
[...]
```

The branch name “permanent” was recorded with the changeset marked *7*.

From the Git side, working with either of these branch styles is the same: just checkout, commit, fetch, merge, pull, and push as you normally would. One thing you should know is that Mercurial doesn’t support rewriting history, only adding to it. Here’s what our Mercurial repository looks like after an interactive rebase and a force-push:

```
$ hg log --style compact -G
o  10[tip]   99611176cbc9   2014-08-14 20:21 -0700   ben
|    A permanent change
|
o  9   f23e12f939c3   2014-08-14 20:01 -0700   ben
|    Add some documentation
|
o  8:1   c16971d33922   2014-08-14 20:00 -0700   ben
|    goodbye
|
| o  7:5   a4529d07aad4   2014-08-14 20:21 -0700   ben
| |    A permanent change
| |
| | @  6   8f65e5e02793   2014-08-14 20:06 -0700   ben
| |/     More documentation
| |
| o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
| |\     Merge remote-tracking branch 'origin/master'
| | |
| | o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| | |    update makefile
| | |
+---o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |      goodbye
| |
| o  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

Changesets *8*, *9*, and *10* have been created and belong to the `permanent` branch, but the old changesets are still there. This can be **very** confusing for your teammates who are using Mercurial, so try to avoid it.

Mercurial Summary

Git and Mercurial are similar enough that working across the boundary is fairly painless. If you avoid changing history that's left your machine (as is generally recommended), you may not even be aware that the other end is Mercurial.

Git and Bazaar

Among the DVCS, another famous one is Bazaar. Bazaar is free and open source, and is part of the GNU Project. It behaves very differently from Git. Sometimes, to do the same thing as with Git, you have to use a different keyword, and some keywords that are common don't have the same

meaning. In particular, the branch management is very different and may cause confusion, especially when someone comes from Git's universe. Nevertheless, it is possible to work on a Bazaar repository from a Git one.

There are many projects that allow you to use Git as a Bazaar client. Here we'll use Felipe Contreras' project that you may find at https://github.com/felipec/git-remote-bzr. To install it, you just have to download the file git-remote-bzr in a folder contained in your `$PATH`:

```
$ wget https://raw.github.com/felipec/git-remote-bzr/master/git-remote-bzr -O
~/bin/git-remote-bzr
$ chmod +x ~/bin/git-remote-bzr
```

You also need to have Bazaar installed. That's all!

Create a Git repository from a Bazaar repository

It is simple to use. It is enough to clone a Bazaar repository prefixing it by `bzr::`. Since Git and Bazaar both do full clones to your machine, it's possible to attach a Git clone to your local Bazaar clone, but it isn't recommended. It's much easier to attach your Git clone directly to the same place your Bazaar clone is attached to – the central repository.

Let's suppose that you worked with a remote repository which is at address `bzr+ssh://developer@mybazaarserver:myproject`. Then you must clone it in the following way:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git
$ cd myProject-Git
```

At this point, your Git repository is created but it is not compacted for optimal disk use. That's why you should also clean and compact your Git repository, especially if it is a big one:

```
$ git gc --aggressive
```

Bazaar branches

Bazaar only allows you to clone branches, but a repository may contain several branches, and `git-remote-bzr` can clone both. For example, to clone a branch:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

And to clone the whole repository:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

The second command clones all the branches contained in the emacs repository; nevertheless, it is

possible to point out some branches:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Some remote repositories don't allow to list their branches, in which case you have to manually specify them, and even though you could specify the configuration in the cloning command, you may find this easier:

```
$ git init emacs
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs
$ git config remote-bzr.branches 'trunk, xwindow'
$ git fetch
```

Ignore what is ignored with .bzrignore

As the format of the `.bzrignore` file is completely compatible with `.gitignore`'s one, and as you shouldn't make a `.gitignore` file in your repository, it is enough to make a symbolic link to `.bzrignore` so that the potential changes of `.bzrignore` are taken into account:

```
$ ln -s .bzrignore .git/info/exclude
```

Fetch the changes of the remote repository

To fetch the changes of the remote, you pull changes as usually, using Git commands. Supposing that your changes are on the `master` branch, you merge or rebase your work on the `origin/master` branch:

```
$ git pull --rebase origin
```

Push your work on the remote repository

Because Bazaar also has the concept of merge commits, there will be no problem if you push a merge commit. So you can work on a branch, merge the changes into `master` and push your work. Then, you create your branches, you test and commit your work as usual. You finally push your work to the Bazaar repository:

```
$ git push origin master
```

Caveats

Git's remote-helpers framework has some limitations that apply. In particular, these commands don't work:

- git push origin :branch-to-delete (Bazaar can't accept ref deletions in this way.)

- git push origin old:new (it will push *old*)
- git push --dry-run origin branch (it will push)

Summary

Since Git's and Bazaar's models are similar, there isn't a lot of resistance when working across the boundary. As long as you watch out for the limitations, and are always aware that the remote repository isn't natively Git, you'll be fine.

Git and Perforce

Perforce is a very popular version-control system in corporate environments. It's been around since 1995, which makes it the oldest system covered in this chapter. As such, it's designed with the constraints of its day; it assumes you're always connected to a single central server, and only one version is kept on the local disk. To be sure, its features and constraints are well-suited to several specific problems, but there are lots of projects using Perforce where Git would actually work better.

There are two options if you'd like to mix your use of Perforce and Git. The first one we'll cover is the "Git Fusion" bridge from the makers of Perforce, which lets you expose subtrees of your Perforce depot as read-write Git repositories. The second is git-p4, a client-side bridge that lets you use Git as a Perforce client, without requiring any reconfiguration of the Perforce server.

Git Fusion

Perforce provides a product called Git Fusion (available at http://www.perforce.com/git-fusion), which synchronizes a Perforce server with Git repositories on the server side.

Setting Up

For our examples, we'll be using the easiest installation method for Git Fusion, which is downloading a virtual machine that runs the Perforce daemon and Git Fusion. You can get the virtual machine image from http://www.perforce.com/downloads/Perforce/20-User, and once it's finished downloading, import it into your favorite virtualization software (we'll use VirtualBox).

Upon first starting the machine, it asks you to customize the password for three Linux users (`root`, `perforce`, and `git`), and provide an instance name, which can be used to distinguish this installation from others on the same network. When that has all completed, you'll see this:

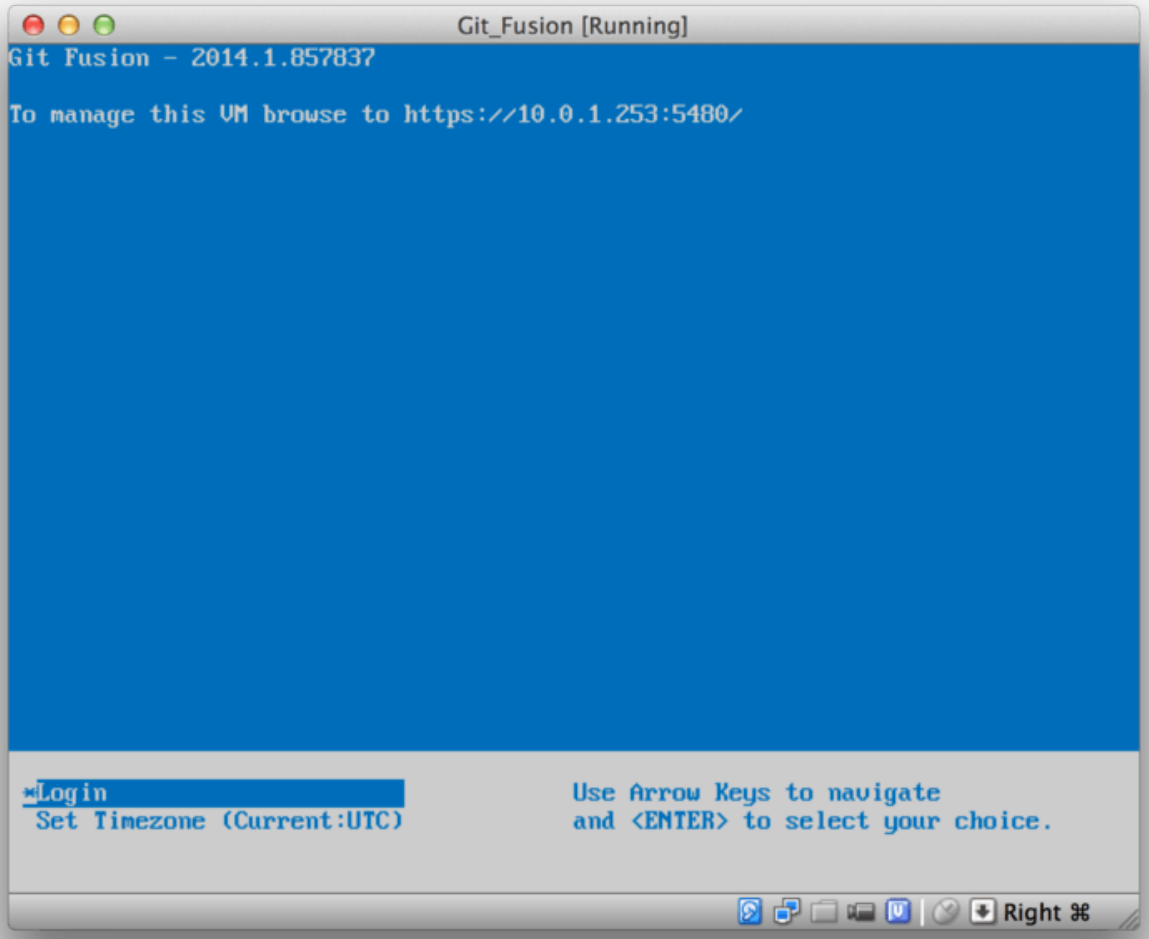


Figura 146. The Git Fusion virtual machine boot screen.

You should take note of the IP address that's shown here, we'll be using it later on. Next, we'll create a Perforce user. Select the "Login" option at the bottom and press enter (or SSH to the machine), and log in as `root`. Then use these commands to create a user:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

The first one will open a VI editor to customize the user, but you can accept the defaults by typing `:wq` and hitting enter. The second one will prompt you to enter a password twice. That's all we need to do with a shell prompt, so exit out of the session.

The next thing you'll need to do to follow along is to tell Git not to verify SSL certificates. The Git Fusion image comes with a certificate, but it's for a domain that won't match your virtual machine's IP address, so Git will reject the HTTPS connection. If this is going to be a permanent installation, consult the Perforce Git Fusion manual to install a different certificate; for our example purposes, this will suffice:

```
$ export GIT_SSL_NO_VERIFY=true
```

Now we can test that everything is working.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

The virtual-machine image comes equipped with a sample project that you can clone. Here we're cloning over HTTPS, with the `john` user that we created above; Git asks for credentials for this connection, but the credential cache will allow us to skip this step for any subsequent requests.

Fusion Configuration

Once you've got Git Fusion installed, you'll want to tweak the configuration. This is actually fairly easy to do using your favorite Perforce client; just map the `//.git-fusion` directory on the Perforce server into your workspace. The file structure looks like this:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
│
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

The `objects` directory is used internally by Git Fusion to map Perforce objects to Git and vice versa, you won't have to mess with anything in there. There's a global `p4gf_config` file in this directory, as well as one for each repository – these are the configuration files that determine how Git Fusion behaves. Let's take a look at the file in the root:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

We won't go into the meanings of these flags here, but note that this is just an INI-formatted text file, much like Git uses for configuration. This file specifies the global options, which can then be overridden by repository-specific configuration files, like `repos/Talkhouse/p4gf_config`. If you open this file, you'll see a `[@repo]` section with some settings that are different from the global defaults. You'll also see sections that look like this:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

This is a mapping between a Perforce branch and a Git branch. The section can be named whatever you like, so long as the name is unique. `git-branch-name` lets you convert a depot path that would be cumbersome under Git to a more friendly name. The `view` setting controls how Perforce files are mapped into the Git repository, using the standard view mapping syntax. More than one mapping can be specified, like in this example:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

This way, if your normal workspace mapping includes changes in the structure of the directories, you can replicate that with a Git repository.

The last file we'll discuss is `users/p4gf_usermap`, which maps Perforce users to Git users, and which you may not even need. When converting from a Perforce changeset to a Git commit, Git Fusion's default behavior is to look up the Perforce user, and use the email address and full name stored there for the author/committer field in Git. When converting the other way, the default is to look up the Perforce user with the email address stored in the Git commit's author field, and submit the changeset as that user (with permissions applying). In most cases, this behavior will do just fine, but consider the following mapping file:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Each line is of the format `<user> <email> "<full name>"`, and creates a single user mapping. The first two lines map two distinct email addresses to the same Perforce user account. This is useful if you've created Git commits under several different email addresses (or change email addresses), but want them to be mapped to the same Perforce user. When creating a Git commit from a Perforce changeset, the first line matching the Perforce user is used for Git authorship information.

The last two lines mask Bob and Joe's actual names and email addresses from the Git commits that are created. This is nice if you want to open-source an internal project, but don't want to publish your employee directory to the entire world. Note that the email addresses and full names should be unique, unless you want all the Git commits to be attributed to a single fictional author.

Workflow

Perforce Git Fusion is a two-way bridge between Perforce and Git version control. Let's have a look at how it feels to work from the Git side. We'll assume we've mapped in the "Jam" project using a configuration file as shown above, which we can clone like this:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on
Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

The first time you do this, it may take some time. What's happening is that Git Fusion is converting all the applicable changesets in the Perforce history into Git commits. This happens locally on the server, so it's relatively fast, but if you have a lot of history, it can still take some time. Subsequent fetches do incremental conversion, so it'll feel more like Git's native speed.

As you can see, our repository looks exactly like any other Git repository you might work with. There are three branches, and Git has helpfully created a local `master` branch that tracks `origin/master`. Let's do a bit of work, and create a couple of new commits:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the
Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

We have two new commits. Now let's check if anyone else has been working:

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master     -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

It looks like someone was! You wouldn't know it from this view, but the `6afeb15` commit was actually created using a Perforce client. It just looks like another commit from Git's point of view, which is exactly the point. Let's see how the Perforce server deals with a merge commit:

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master
```

Git thinks it worked. Let's take a look at the history of the `README` file from Perforce's point of view, using the revision graph feature of `p4v`:

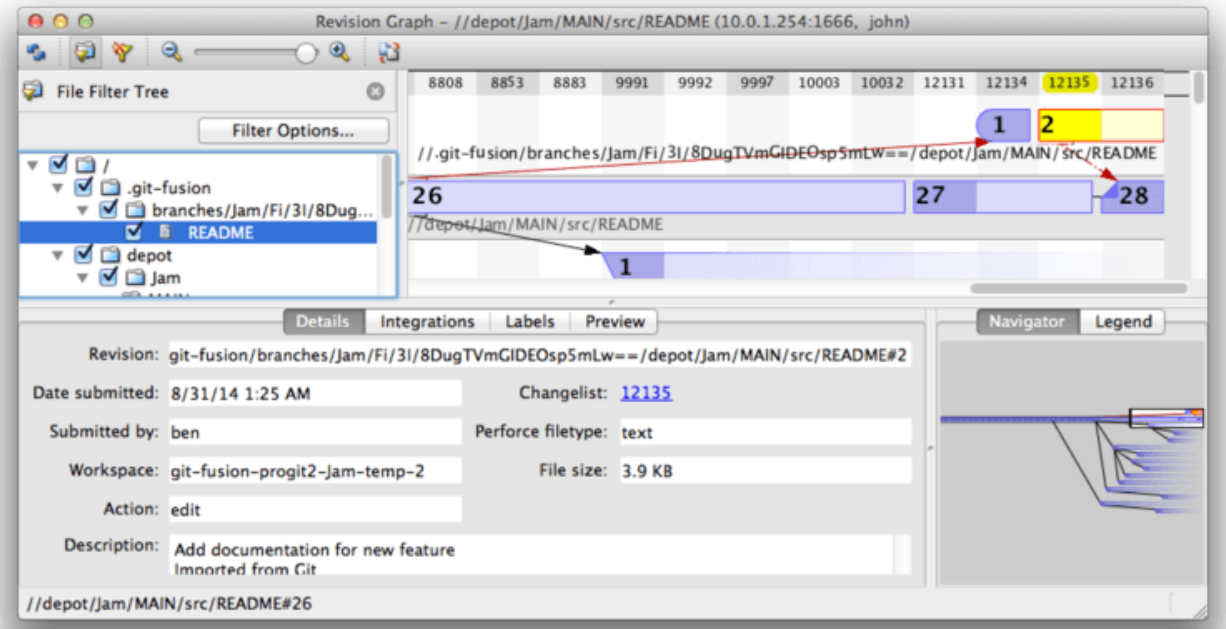


Figura 147. Perforce revision graph resulting from Git push.

If you've never seen this view before, it may seem confusing, but it shows the same concepts as a graphical viewer for Git history. We're looking at the history of the `README` file, so the directory tree at top left only shows that file as it surfaces in various branches. At top right, we have a visual graph of how different revisions of the file are related, and the big-picture view of this graph is at bottom right. The rest of the view is given to the details view for the selected revision (`2` in this case).

One thing to notice is that the graph looks exactly like the one in Git's history. Perforce didn't have a named branch to store the `1` and `2` commits, so it made an "anonymous" branch in the `.git-fusion` directory to hold it. This will also happen for named Git branches that don't correspond to a named Perforce branch (and you can later map them to a Perforce branch using the configuration file).

Most of this happens behind the scenes, but the end result is that one person on a team can be using Git, another can be using Perforce, and neither of them will know about the other's choice.

Git-Fusion Summary

If you have (or can get) access to your Perforce server, Git Fusion is a great way to make Git and Perforce talk to each other. There's a bit of configuration involved, but the learning curve isn't very steep. This is one of the few sections in this chapter where cautions about using Git's full power will not appear. That's not to say that Perforce will be happy with everything you throw at it – if you try to rewrite history that's already been pushed, Git Fusion will reject it – but Git Fusion tries very hard to feel native. You can even use Git submodules (though they'll look strange to Perforce users), and merge branches (this will be recorded as an integration on the Perforce side).

If you can't convince the administrator of your server to set up Git Fusion, there is still a way to use these tools together.

Git-p4

Git-p4 is a two-way bridge between Git and Perforce. It runs entirely inside your Git repository, so you won't need any kind of access to the Perforce server (other than user credentials, of course). Git-p4 isn't as flexible or complete a solution as Git Fusion, but it does allow you to do most of what you'd want to do without being invasive to the server environment.

NOTA	You'll need the `p4` tool somewhere in your `PATH` to work with git-p4. As of this writing, it is freely available at http://www.perforce.com/downloads/Perforce/20-User.

Setting Up

For example purposes, we'll be running the Perforce server from the Git Fusion OVA as shown above, but we'll bypass the Git Fusion server and go directly to the Perforce version control.

In order to use the `p4` command-line client (which git-p4 depends on), you'll need to set a couple of environment variables:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Getting Started

As with anything in Git, the first command is to clone:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into
refs/remotes/p4/master
```

This creates what in Git terms is a "shallow" clone; only the very latest Perforce revision is imported into Git; remember, Perforce isn't designed to give every revision to every user. This is enough to use Git as a Perforce client, but for other purposes it's not enough.

Once it's finished, we have a fully-functional Git repository:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from
the state at revision #head
```

Note how there's a "p4" remote for the Perforce server, but everything else looks like a standard clone. Actually, that's a bit misleading; there isn't actually a remote there.

```
$ git remote -v
```

No remotes exist in this repository at all. Git-p4 has created some refs to represent the state of the server, and they look like remote refs to `git log`, but they're not managed by Git itself, and you can't push to them.

Workflow

Okay, let's do some work. Let's assume you've made some progress on a very important feature, and you're ready to show it to the rest of your team.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at
revision #head
```

We've made two new commits that we're ready to submit to the Perforce server. Let's check if anyone else was working today:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Looks like they were, and `master` and `p4/master` have diverged. Perforce's branching system is *nothing* like Git's, so submitting merge commits doesn't make any sense. Git-p4 recommends that you rebase your commits, and even comes with a shortcut to do so:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

You can probably tell from the output, but `git p4 rebase` is a shortcut for `git p4 sync` followed by `git rebase p4/master`. It's a bit smarter than that, especially when working with multiple branches, but this is a good approximation.

Now our history is linear again, and we're ready to contribute our changes back to Perforce. The `git p4 submit` command will try to create a new Perforce revision for every Git commit between `p4/master` and `master`. Running it drops us into our favorite editor, and the contents of the file look something like this:

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists only.)
#  Files:       What opened files from the default changelist are to be added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:  new

Client:  john_bens-mbp_8487

User: john

Status:  new

Description:
   Update link

Files:
   //depot/www/live/index.html   # edit


######## git author ben@straub.cc does not match your p4 account.
######## Use option --preserve-user to modify authorship.
######## Variable git-p4.skipUserNameCheck hides this message.
######## everything below this line is just the diff #######
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html   2014-
08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
 Jam/MR</a>,
 a software build tool.
 </td>
```

This is mostly the same content you'd see by running `p4 submit`, except the stuff at the end which git-p4 has helpfully included. Git-p4 tries to honor your Git and Perforce settings individually when

it has to provide a name for a commit or changeset, but in some cases you want to override it. For example, if the Git commit you're importing was written by a contributor who doesn't have a Perforce user account, you may still want the resulting changeset to look like they write it (and not you).

Git-p4 has helpfully imported the message from the Git commit as the content for this Perforce changeset, so all we have to do is save and quit, twice (once for each commit). The resulting shell output will look something like this:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The result is as though we just did a `git push`, which is the closest analogy to what actually did happen.

Note that during this process every Git commit is turned into a Perforce changeset; if you want to squash them down into a single changeset, you can do that with an interactive rebase before running `git p4 submit`. Also note that the SHA-1 hashes of all the commits that were submitted as changesets have changed; this is because git-p4 adds a line to the end of each commit it converts:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

    [git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

What happens if you try to submit a merge commit? Let's give it a try. Here's the situation we've gotten ourselves into:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The Git and Perforce history diverge after 775a46f. The Git side has two commits, then a merge commit with the Perforce head, then another commit. We're going to try to submit these on top of a single changeset on the Perforce side. Let's see what would happen if we tried to submit now:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

The `-n` flag is short for `--dry-run`, which tries to report what would happen if the submit command were run for real. In this case, it looks like we'd be creating three Perforce changesets, which correspond to the three non-merge commits that don't yet exist on the Perforce server. That sounds like exactly what we want, let's see how it turns out:

```
$ git p4 submit
[…]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Our history became linear, just as though we had rebased before submitting (which is in fact exactly what happened). This means you can be free to create, work on, throw away, and merge branches on the Git side without fear that your history will somehow become incompatible with Perforce. If you can rebase it, you can contribute it to a Perforce server.

Branching

If your Perforce project has multiple branches, you're not out of luck; git-p4 can handle that in a way that makes it feel like Git. Let's say your Perforce depot is laid out like this:

```
//depot
   └─── project
       ├─── main
       └─── dev
```

And let's say you have a `dev` branch, which has a view spec that looks like this:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 can automatically detect that situation and do the right thing:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

Note the "@all" specifier in the depot path; that tells git-p4 to clone not just the latest changeset for that subtree, but all changesets that have ever touched those paths. This is closer to Git's concept of a clone, but if you're working on a project with a long history, it could take a while.

The `--detect-branches` flag tells git-p4 to use Perforce's branch specs to map the branches to Git refs. If these mappings aren't present on the Perforce server (which is a perfectly valid way to use Perforce), you can tell git-p4 what the branch mappings are, and you get the same result:

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

Setting the `git-p4.branchList` configuration variable to `main:dev` tells git-p4 that "main" and "dev" are both branches, and the second one is a child of the first one.

If we now `git checkout -b dev p4/project/dev` and make some commits, git-p4 is smart enough to target the right branch when we do `git p4 submit`. Unfortunately, git-p4 can't mix shallow clones and multiple branches; if you have a huge project and want to work on more than one branch, you'll have to `git p4 clone` once for each branch you want to submit to.

For creating or integrating branches, you'll have to use a Perforce client. Git-p4 can only sync and submit to existing branches, and it can only do it one linear changeset at a time. If you merge two branches in Git and try to submit the new changeset, all that will be recorded is a bunch of file changes; the metadata about which branches are involved in the integration will be lost.

Git and Perforce Summary

Git-p4 makes it possible to use a Git workflow with a Perforce server, and it's pretty good at it. However, it's important to remember that Perforce is in charge of the source, and you're only using Git to work locally. Just be really careful about sharing Git commits; if you have a remote that other

people use, don't push any commits that haven't already been submitted to the Perforce server.

If you want to freely mix the use of Perforce and Git as clients for source control, and you can convince the server administrator to install it, Git Fusion makes using Git a first-class version-control client for a Perforce server.

Migrating to Git

If you have an existing codebase in another VCS but you've decided to start using Git, you must migrate your project one way or another. This section goes over some importers for common systems, and then demonstrates how to develop your own custom importer. You'll learn how to import data from several of the bigger professionally used SCM systems, because they make up the majority of users who are switching, and because high-quality tools for them are easy to come by.

Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. If you want the history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn't perfect; and because it will take so long, you may as well do it right. The first problem is the author information. In Subversion, each person committing has a user on the system who is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` output and the `git svn log`. If you want to map this to better Git author data, you need a mapping from the Subversion users to the Git authors. Create a file called `users.txt` that has this mapping in a format like this:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that SVN uses, you can run this:

```
$ svn log --xml --quiet | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

That generates the log output in XML format, then keeps only the lines with author information, discards duplicates, strips out the XML tags. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your `users.txt` file so you can add the equivalent Git user data next to each entry.

You can provide this file to `git svn` to help it map the author data more accurately. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command (though if you want to keep the synchronisation-metadata, feel free to omit this parameter). This makes your `import` command look like this:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata --prefix "" -s my_project
$ cd my_project
```

Now you should have a nicer Subversion import in your `my_project` directory. Instead of commits that look like this

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Not only does the Author field look a lot better, but the `git-svn-id` is no longer there, either.

You should also do a bit of post-import cleanup. For one thing, you should clean up the weird references that `git svn` set up. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run:

```
$ for t in $(git for-each-ref --format='%(refname:short)' refs/remotes/tags); do git
tag ${t/tags\//} $t && git branch -D -r $t; done
```

This takes the references that were remote branches that started with `refs/remotes/tags/` and makes them real (lightweight) tags.

Next, move the rest of the references under `refs/remotes` to be local branches:

```
$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); do git branch
$b refs/remotes/$b && git branch -D -r $b; done
```

It may happen that you'll see some extra branches which are suffixed by `@xxx` (where xxx is a number), while in Subversion you only see one branch. This is actually a Subversion feature called

“peg-revisions”, which is something that Git simply has no syntactical counterpart for. Hence, `git svn` simply adds the svn version number to the branch name just in the same way as you would have written it in svn to address the peg-revision of that branch. If you do not care anymore about the peg-revisions, simply remove them:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D
$p; done
```

Now all the old branches are real Git branches and all the old tags are real Git tags.

There’s one last thing to clean up. Unfortunately, `git svn` creates an extra branch named `trunk`, which maps to Subversion’s default branch, but the `trunk` ref points to the same place as `master`. Since `master` is more idiomatically Git, here’s how to remove the extra branch:

```
$ git branch -d trunk
```

The last thing to do is add your new Git server as a remote and push to it. Here is an example of adding your server as a remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Because you want all your branches and tags to go up, you can now run this:

```
$ git push origin --all
$ git push origin --tags
```

All your branches and tags should be on your new Git server in a nice, clean import.

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool called "hg-fast-export", which you’ll need a copy of:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what it will put in the author field for changesets, so this is a good time to clean house. Generating this is a one-line command in a `bash` shell:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

This will take a few seconds, depending on how long your project's history is, and afterwards the `/tmp/authors` file will look something like this:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In this example, the same person (Bob) has created changesets under four different names, one of which actually looks correct, and one of which would be completely invalid for a Git commit. Hg-fast-export lets us fix this by adding `={new name and email address}` at the end of every line we want to change, and removing the lines for any usernames that we want to leave alone. If all the usernames look fine, we won't need this file at all. In this example, we want our file to look like this:

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
```

The next step is to create our new Git repository, and run the export script:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

The `-r` flag tells hg-fast-export where to find the Mercurial repository we want to convert, and the `-A` flag tells it where to find the author-mapping file. The script parses Mercurial changesets and converts them into a script for Git's "fast-import" feature (which we'll discuss in detail a bit later on). This takes a bit (though it's *much* faster than it would be over the network), and the output is fairly verbose:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed
files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                  )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602
attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599
attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

That's pretty much all there is to it. All of the Mercurial tags have been converted to Git tags, and Mercurial branches and bookmarks have been converted to Git branches. Now you're ready to push the repository up to its new server-side home:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Bazaar

Bazaar is a DVCS tool much like Git, and as a result it's pretty straightforward to convert a Bazaar repository into a Git one. To accomplish this, you'll need to import the `bzr-fastimport` plugin.

Getting the bzr-fastimport plugin

The procedure for installing the fastimport plugin is different on UNIX-like operating systems and on Windows. In the first case, the simplest is to install the `bzr-fastimport` package that will install all the required dependencies.

For example, with Debian and derived, you would do the following:

```
$ sudo apt-get install bzr-fastimport
```

With RHEL, you would do the following:

```
$ sudo yum install bzr-fast-import
```

With Fedora, since release 22, the new package manager is dnf:

```
$ sudo dnf install bzr-fastimport
```

If the package is not available, you may install it as a plugin:

```
$ mkdir --parents ~/.bazaar/plugins/bzr     # creates the necessary folders for the
plugins
$ cd ~/.bazaar/plugins/bzr
$ bzr branch lp:bzr-fastimport fastimport   # imports the fastimport plugin
$ cd fastimport
$ sudo python setup.py install --record=files.txt   # installs the plugin
```

For this plugin to work, you'll also need the `fastimport` Python module. You can check whether it is present or not and install it with the following commands:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

If it is not available, you can download it at address https://pypi.python.org/pypi/fastimport/.

In the second case (on Windows), `bzr-fastimport` is automatically installed with the standalone version and the default installation (let all the checkboxes checked). So in this case you have nothing to do.

At this point, the way to import a Bazaar repository differs according to that you have a single branch or you are working with a repository that has several branches.

Project with a single branch

Now `cd` in the directory that contains your Bazaar repository and initialize the Git repository:

```
$ cd /path/to/the/bzr/repository
$ git init
```

Now, you can simply export your Bazaar repository and convert it into a Git repository using the following command:

```
$ bzr fast-export --plain . | git fast-import
```

Depending on the size of the project, your Git repository is built in a lapse from a few seconds to a few minutes.

At this point, the `.git` directory and your working tree are all set up, but the working tree and the index are not synchronized with HEAD:

```
$ git status
On master branch
Changes that will be validated:
  (use "git reset HEAD <fichier>..." to unstage)

        removed :        .bzrignore
        removed :        file.txt
```

This is fixed by typing:

```
$ git reset --hard HEAD
```

Now let us have a look at the files to ignore. As `.bzrignore`'s format is completely compatible with `.gitignore`'s format, the simplest is to rename your `.bzrignore` file:

```
$ git mv .bzrignore .gitignore
```

Then you have to create a commit that contains this change for the migration:

```
$ git commit -am 'Migration from Bazaar to Git'
```

That's all! Now you can push the repository onto its new home server:

```
$ git remote add origin git@my-git-server:mygitrepository.git
$ git push origin --all
$ git push origin --tags
```

Case of a project with a main branch and a working branch

You can also import a Bazaar repository that contains branches. Let us suppose that you have two branches: one represents the main branch (myProject.trunk), the other one is the working branch (myProject.work).

```
$ ls
myProject.trunk myProject.work
```

Create the Git repository and `cd` into it:

```
$ git init git-repo
$ cd git-repo
```

Pull the master branch into git:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject.trunk | \
git fast-import --export-marks=../marks.git
```

Pull the working branch into Git:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

Now `git branch` shows you the `master` branch as well as the `work` branch. Check the logs to make sure they're complete and get rid of the `marks.bzr` and `marks.git` files.

Your working copy is still unsynchronized, so let's reset it:

```
$ git reset --hard HEAD
```

Your Git repository is ready to use.

Perforce

The next system you'll look at importing from is Perforce. As we discussed above, there are two ways to let Git and Perforce talk to each other: git-p4 and Perforce Git Fusion.

Perforce Git Fusion

Git Fusion makes this process fairly painless. Just configure your project settings, user mappings, and branches using a configuration file (as discussed in Git Fusion), and clone the repository. Git Fusion leaves you with what looks like a native Git repository, which is then ready to push to a native Git host if you desire. You could even use Perforce as your Git host if you like.

Git-p4

Git-p4 can also act as an import tool. As an example, we'll import the Jam project from the Perforce Public Depot. To set up your client, you must export the P4PORT environment variable to point to the Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

NOTA In order to follow along, you'll need a Perforce depot to connect with. We'll be using the public depot at public.perforce.com for our examples, but you can use any depot you have access to.

Run the `git p4 clone` command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

This particular project has only one branch, but if you have branches that are configured with branch views (or just a set of directories), you can use the `--detect-branches` flag to `git p4 clone` to import all the project's branches as well. See Branching for a bit more detail on this.

At this point you're almost done. If you go to the `p4import` directory and run `git log`, you can see your imported work:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

You can see that `git-p4` has left an identifier in each commit message. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so – before you start doing work on the new repository. You can use `git filter-branch` to remove the identifier strings en masse:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

If you run `git log`, you can see that all the SHA-1 checksums for the commits have changed, but the `git-p4` strings are no longer in the commit messages:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

Your import is ready to push up to your new Git server.

A Custom Importer

If your system isn't one of the above, you should look for an importer online – quality importers are

available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see Funcionamento Interno do Git for more information). This way, you can write an import script that reads the necessary information out of the system you're importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you'll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As we did in An Example Git-Enforced Policy, we'll write this in Ruby, because it's what we generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with – it just needs to print the appropriate information to `stdout`. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines – `git fast-import` is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. "Mark" is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is:

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as:

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'John Doe <john@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word data, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory – you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, fast-import can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex – you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

NOTA

If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while `git fast-import` expects only LF. To get around this problem and make `git fast-import` happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    puts 'commit refs/heads/master'
    puts "mark :#{mark}"
    puts "committer #{$author} #{date} -0700"
    export_data("imported from #{dir}")
    puts "from :#{last_mark}" if last_mark

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
    end
```

```
    mark
end

# Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
    Dir.glob("*").each do |dir|
        next if File.file?(dir)

        # move into the target directory
        Dir.chdir(dir) do
            last_mark = print_export(dir, last_mark)
        end
    end
end
```

If you run this script, you'll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your script:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of          5
attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4
attempts)
      commits:            4 (         1 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 13 objects total for 4 commits into 1 branch. Now, you can run `git log` to see your new history:

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

There you go – a nice, clean Git repository. It's important to note that nothing is checked out – you don't have any files in your working directory at first. To get them, you must reset your branch to where `master` is now:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

You can do a lot more with the `fast-import` tool – handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code.

Summary

You should feel comfortable using Git as a client for other version-control systems, or importing nearly any existing repository into Git without losing data. In the next chapter, we'll cover the raw internals of Git so you can craft every single byte, if need be.

Funcionamento Interno do Git

Talvez você venha de um capítulo anterior para este, ou talvez você tenha chegado aqui depois de ler o resto do livro. De qualquer forma, aqui iremos para o funcionamento e implementação internos do Git. Nós achávamos que aprender esta informação era de fundamental importância para o entendimento do quão útil e poderoso o Git é, mas outras pessoas alegaram para nós que isso pode ser confuso e desnecessariamente complexo para iniciantes. Desta forma, levamos esta discussão para o último capítulo do livro, de forma que você possa lê-lo tanto no começo quanto mais tarde em seu aprendizado. Deixamos que você decida isso.

Já que você está aqui, vamos começar. Primeiramente, se ainda não está claro, o Git é essencialmente um sistema de arquivos de conteúdo endereçável com uma interface de usuário de um VCS escrita sobre ele. Você logo aprenderá o que isso significa.

Nos primeiros dias do Git (principalmente antes da versão 1.5), a interface de usuário era muito mais complexa por ela enfatizar mais esse sistema de arquivos do que ser um VCS pronto. Nos últimos anos, a interface de usuário foi refinada até se tornar tão clara e fácil quanto qualquer outro VCS; porém, com frequência ainda perdura o esteriótipo da interface antiga do Git que era complexa e difícil de aprender.

O sistema de arquivos de conteúdo endereçável é incrivelmente interessante, então cobriremos isto primeiramente neste capítulo; em seguida, você irá aprender sobre os mecanismos de transporte e as tarefas de manutenção que você eventualmente terá de lidar.

Encanamento e Porcelana

Este livro cobre como usar o Git com mais ou menos 30 verbos como `checkout`, `branch`, `remote`, entre outros. Porém, como o Git a princípio era um conjunto de ferramentas para um VCS em vez de um VCS completo e amigável, ele tem uma porção de verbos que fazem trabalhos de baixo nível e foram feitos para serem encadeados no estilo UNIX ou serem chamados em *scripts*. Esses comandos normalmente são chamados como comandos de “encanamento” (*plumbing*), e os mais amigáveis são chamados de comandos “porcelana” (*porcelain*).

Os primeiros nove capítulos deste livro lidam quase exclusivamente com comandos porcelana. Entretanto, neste capítulo, você irá lidar na maior parte do tempo com os comandos de encanamento, de nível mais baixo, pois eles lhe darão acesso ao funcionamento interno do Git e ajudarão a demonstrar como e porque o Git faz o que ele faz. Muitos desses comandos não foram feitos para serem usados manualmente na linha de comando, mas para serem usados como blocos de construção de novas ferramentas e *scripts* próprios.

Quando você executa `git init` em um diretório novo ou existente, o Git cria um diretório `.git`, que é onde é localizado quase tudo que o Git armazena ou manipula. Se você quer fazer uma cópia ou clonar o seu repositório, copiar apenas este diretório para outro lugar te dá quase tudo que você precisa. Este capítulo inteiro basicamente lida com o conteúdo deste diretório. A aparência dele é essa:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Talvez você veja outros arquivos nesse diretório, porém esse é um repositório logo após um `git init` - isso é o que você vê por padrão. O arquivo `description` só é usado pelo programa GitWeb, então não se preocupe com ele. O arquivo `config` contem opções de configuração específicas do seu projeto, e o diretório `info` mantém um arquivo `exclude` global para padrões ignorados que você não gostaria de colocar em um arquivo `.gitignore`. O diretório `hooks` contém scripts *hooks* tanto para o lado do cliente quanto do servidor, que são discutidos em detalhes em Git Hooks.

Isto nos deixa quatro importantes entradas: o arquivo `HEAD` e o (ainda a ser criado) arquivo `index`, além dos diretórios `objects` e `refs`. Essas são as partes principais do Git. O diretório `objects` guarda todo o conteúdo para o seu banco de dados, o diretório `refs` guarda referências para objetos *commit* nesse banco de dados (*branches*), o arquivo `HEAD` aponta para a *branch* em que você fez um *checkout*, e o arquivo `index` é onde o Git guarda a informação da área de *stage*. Você irá agora ver cada uma das três seções em detalhes para entender como o Git funciona.

Objetos do Git

O Git é um sistema de arquivos de conteúdo endereçável. Ótimo. O que isso significa? Isso significa que o coração do Git é um simples armazenamento chave-valor. Você pode inserir qualquer tipo de conteúdo nele, e ele lhe dará de volta uma chave que você pode usar para recuperar o conteúdo de volta em qualquer momento. Para demonstrar isso, você pode usar o comando de encanamento `hash-object`, que recebe alguns dados, armazena eles em seu diretório `.git`, e lhe devolve de volta a chave com o qual os dados são armazenados. Primeiramente, inicialize um novo repositório Git e verifique que não há nada no diretório `objects`:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

O Git inicializou o diretório `objects` diretamente e criou os subdiretórios `pack` e `info` dentro dele, mas não há nenhum arquivo regular. Agora, guarde algum texto no seu banco de dados do Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

A *flag*`-w` diz ao `hash-object` para armazenar o objeto. Caso contrário, o comando simplesmente lhe diria a chave. `--stdin` diz ao comando para ler o conteúdo do stdin; se você não especificar isto, `hash-object` espera um caminho para um arquivo no fim. A saída do comando é um `checksum hash` de 40 caracteres. Esse é o *hash* SHA-1 - um `checksum` do conteúdo que você está armazendo mais um cabeçalho, que você aprenderá em breve. Agora você pode ver como o Git armazenou seus dados:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Você pode ver um arquivo no diretório `objects`. É assim que o Git armazena o conteúdo inicialmente - como um simples arquivo por porção de conteúdo, nomeado com o *checksum* SHA-1 do conteúdo e seu cabeçalho. O subdiretório é nomeado com os dois primeiros caracteres do SHA-1, e o nome do arquivo são os 38 caracteres restantes.

Você pode recuperar o conteúdo para fora do Git com o comando `cat-file`. Esse comando é um canivete suíço para a inspeção de objetos do Git. Passando `-p` para ele faz com que o `cat-file` descubra o tipo do conteúdo e o mostre gentilmente para você:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Agora, você pode adicionar conteúdo para o Git e recuperá-lo de volta. Você também pode fazer isso para conteúdos em arquivos. Por exemplo, você pode fazer um controle de versão simples em um arquivo. Primeiro, crie um arquivo e salve seus conteúdos em seu banco de dados:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Depois, escreva alguns conteúdos novos nesse arquivo, e salve-o novamente:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Seu banco de dados contém as duas novas versões do arquivo, além do primeiro conteúdo que você gravou:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Agora você pode reverter o arquivo de volta à primeira versão:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

ou à segunda versão:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Lembrando que decorar a chave SHA-1 para cada versão de seu arquivo não é prático; além disso, você não está armazenando o nome do arquivo em seu sitema - apenas o conteúdo. Este tipo de objeto é chamado de *blob*. Você pode pedir para o Git lhe dizer o tipo de objeto de qualquer objeto, dado sua chave SHA-1, com `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Objetos Tree

O próximo tipo que iremos ver é a *tree* (árvore), que resolve o problema de armazenar o nome de arquivo e também permite armazenar de forma conjunta um grupo de arquivos. O Git armazena o conteúdo em uma maneira similar a um sistema de arquivos UNIX, porém um pouco simplificado. Todo o conteúdo é armazenado como objetos *tree* e *blob*, com as *trees* correspondendo a entradas de um diretório UNIX e *blobs* correspondendo mais ou menos a *inodes* ou conteúdos de arquivos. Um único objeto *tree* contém uma ou mais entradas, cada uma contendo uma referência SHA-1 para um *blob* ou *subtree* com seu modo, tipo e nome de arquivo associados. Por exemplo, a *tree* mais recente em um projeto deverá se parecer com algo assim:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

A sintaxe `master^{tree}` especifica o objeto *tree* que é apontado pelo último *commit* em sua *branch* `master` . Note que o subdiretório `lib` não é um *blob*, mas uma referência para outra *tree*:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceitualmente, os dados que são armazenados pelo Git é algo assim:

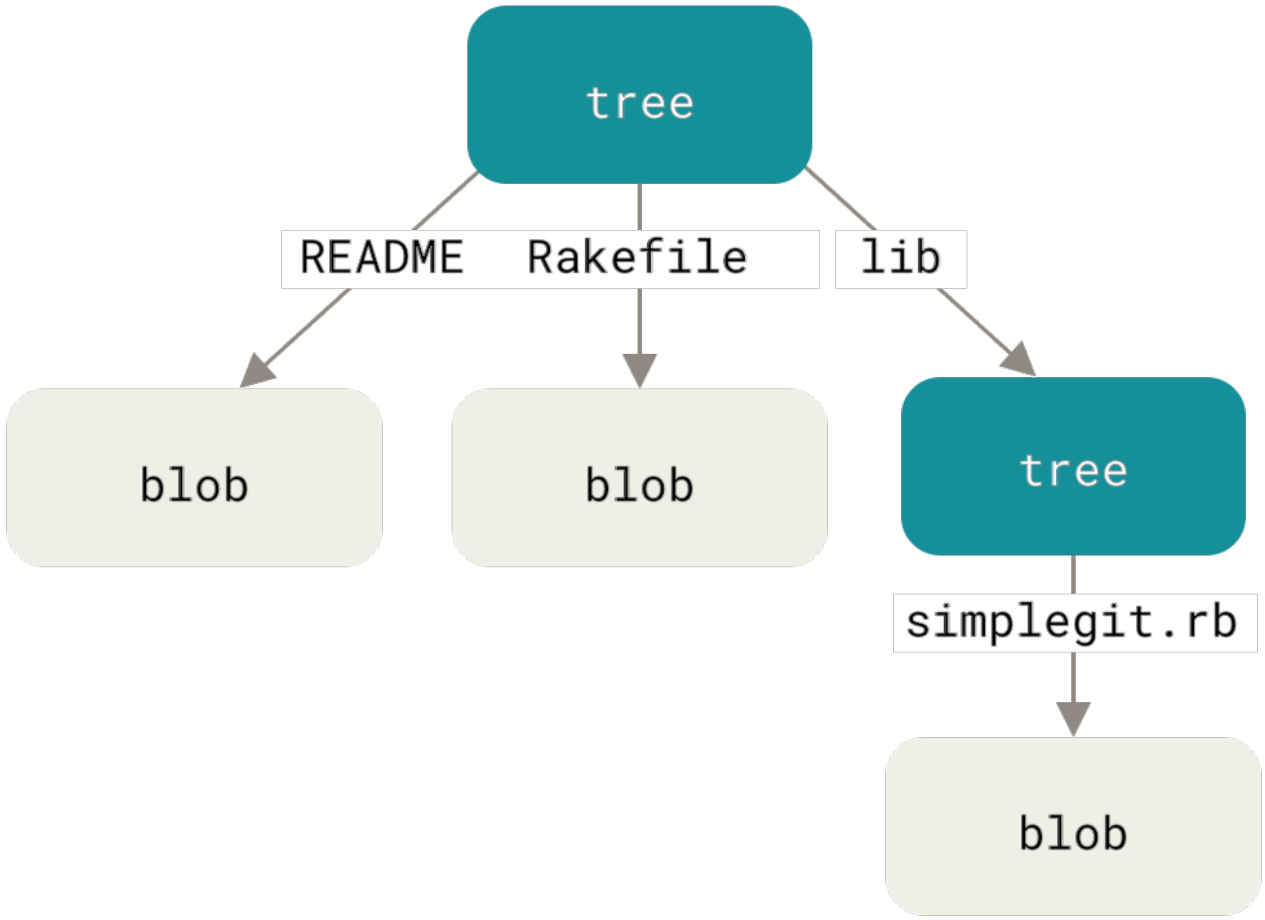


Figura 148. Versão simples do modelo de dados do Git.

Você pode criar facilmente a sua própria *tree*. O Git normalmente cria uma *tree* a partir do estado da sua área de *stage* ou *index* e escrevendo uma série de objetos *tree* a partir dela. Então, para criar um objeto *tree*, você primeiro precisa popular um *index* adicionando alguns arquivos. Para criar um *index* com apenas uma entrada - a primeira versão do do seu arquivo `test.txt` - você pode usar o comando `update-index`. Você usa esse comando para adicionar artificialmente a versão anterior do arquivo `test.txt` à nova área de *stage*. Você precisa passar a ele a opção `--add` porque o arquivo ainda não existe em sua área de *stage* (você nem precisa ter uma área de *stage* ainda) e a opção `--cacheinfo` porque o arquivo que você está adicionando não está em seu diretório mas está no seu banco de dados. Depois você especifica o modo, o SHA-1 e o nome do arquivo:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Neste caso, você está especificando um modo `100644`, o que significa se trata de um arquivo normal. Outras opções são `100755`, o que significa que é um arquivo executável; e `120000`, que especifica um link simbólico. O modo vem dos modos UNIX normais, mas é muito menos flexível - esses três

modos são os únicos que são válidos para arquivos (*blobs*) no Git (ainda que outros modos possam ser usados para diretórios e submódulos).

Agora, você pode usar o comando `write-tree` para escrever a área de *stage* para um objeto *tree*. A opção `-w` não é necessária - chamar `write-tree` automaticamente cria um objeto `tree` a partir do estado do *index* caso a *tree* ainda não exista:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Você também pode verificar que se este é um arquivo *tree*:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Você pode criar um novo arquivo *tree* com a segunda versão de `test.txt`, além de um novo arquivo:

```
$ echo 'new file' > new.txt
$ git update-index --cacheinfo 100644 \
  1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Sua área de *stage* agora tem a nova versão de `test.txt`, bem como o novo arquivo `new.txt`. Escreva essa *tree* (grave o estado da área de *stage* ou *index* para um objeto) e veja como ela se parece:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Note que essa *tree* tem ambas as entradas de arquivo além de que o SHA-1 de `test.txt` é a o SHA-1 da "versão 2" que falamos anteriormente (`1f7a7a`). Apenas por diversão, adicione a primeira *tree* como um subdiretório neste aqui. Você pode ler as *trees* para a área de *stage* chamando `read-tree`. Neste caso, você pode ler uma *tree* existente em sua área de *stage* como uma *subtree* usando a opção `--prefix` em `read-tree`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Se você criasse um diretório de trabalho a partir da nova *tree* que você criou, você teria os dois arquivos no nível mais alto do diretório de trabalho e um subdiretório chamado `bak` que conteria a primeira versão do arquivo `test.txt`. Você pode pensar nos dados que o Git armazena para essas estruturas como sendo algo assim:

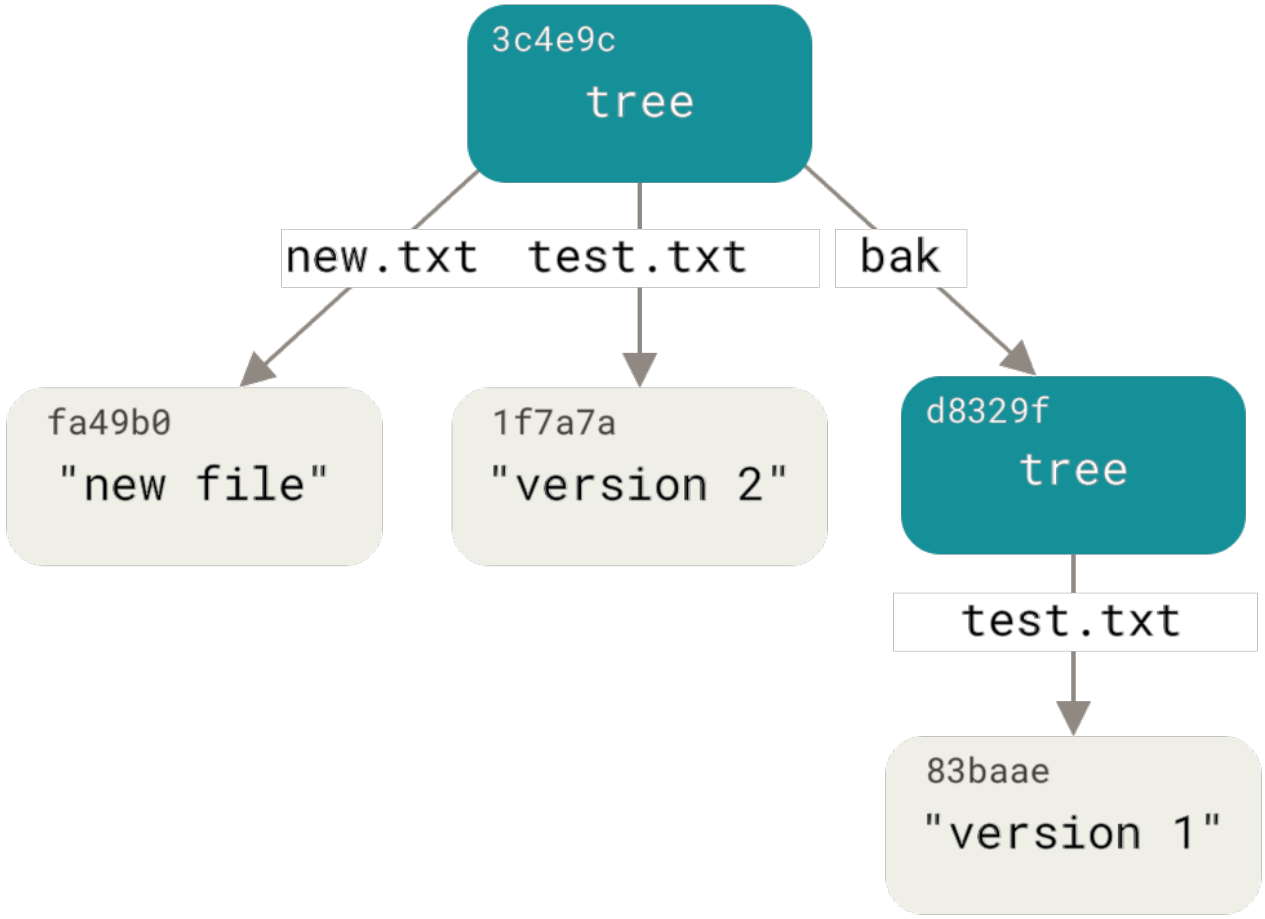


Figura 149. A estrutura atual do conteúdo dos seus dados no Git.

Objetos Commit

Agora você tem três *trees* que especificam os diferentes *snapshots* do seu projeto que você gostaria de rastrear, mas o problema anterior se mantém: você precisa lembrar dos três valores dos SHA-1 para encontrar os *snapshots* Você também não tem nenhuma informação sobre quem salvou os snapshots, quando eles foram salvos, ou porque eles foram salvos. Essas são informações básicas que o objeto *commit* armazena para você.

Para criar um objeto *commit*, você chama `commit-tree` e especifica o SHA-1 de uma única *tree* e

quais objetos *commit* precedem diretamente ele, se houver. Começando com a primeira *tree* que você escreveu:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Você irá obter um valor diferente para o *hash* por causa das diferentes hora de criação e dados do autor. Substitua os *hashes* de *commit* e *tag* pelos seus próprios *checksums* posteriormente neste capítulo. Agora você pode olhar para o seu novo objeto *commit* com `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

O formato para um objeto *commit* é simples: ele especifica a *tree* de nível mais alto para o *snapshot* do projeto neste ponto; a informação do autor/*commiter* (que usa as configurações `user.name` e `user.email`, além de um *timestamp*); uma linha em branco e então a mensagem de *commit*.

A seguir, você irá escrever outros dois objetos *commit*, cada um referenciando o *commit* que veio diretamente antes dele:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Cada um dos três objetos *commit* aponta para uma das três *trees* de *snapshot* que você criou. Curiosamente, você tem agora um histórico do Git real que você pode ver com o comando `git log`, se você executá-lo no SHA-1 do último *commit*:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

Incrível. Você acabou de fazer operações de baixo nível para criar um histórico do Git sem usar nenhum dos comandos de *front-end*. Isso é essencialmente o que o Git faz quando você executa os comandos `git add` e `git commit` - ele armazena *blobs* para os arquivos que mudaram, atualiza o *index*, escreve as *trees* e escreve os objetos *commit* que referenciam as *trees* de mais alto nível e os *commits* que vieram imediatamente antes deles. Esses três principais objetos do Git - o *blob*, a *tree*, e o *commit* - são inicialmente armazenados como arquivos separados em seu diretório `.git/objects`. Estes são todos os objetos no diretório de exemplo, comentados com o que eles armazenam:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Se você seguir as referências internas, você obterá um grafo de objetos mais ou menos como este:

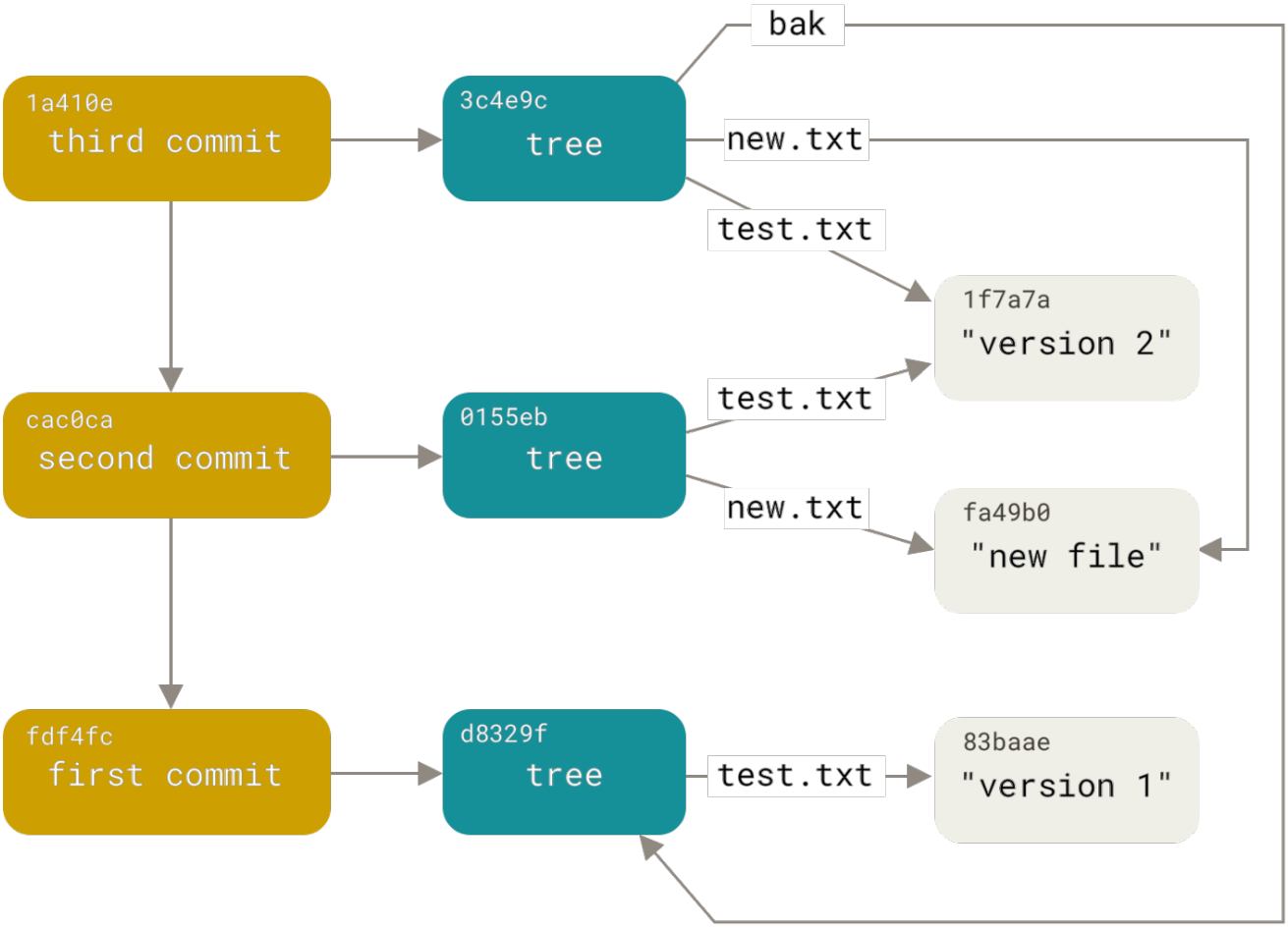


Figura 150. Todos os objetos no seu diretório do Git.

Armazenamento de Objetos

Mencionamos anteriormente que o cabeçalho é armazenado junto com o conteúdo. Vamos tomar um minuto para olhar como o Git armazena seus objetos. Você verá como armazenar um objeto *blob* - neste caso, a *string* "what is up, doc?" - interativamente usando a linguagem de *script* Ruby.

Você pode iniciar o modo interativo do Ruby com o comando `irb`:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

O Git constrói um cabeçalho que começa com o tipo de objeto, neste caso, um *blob*. Depois, ele adiciona um espaço seguido do tamanho do conteúdo e finalmente um *byte* nulo:

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

O Git concatena o cabeçalho e o conteúdo original e então calcula o *checksum* SHA-1 do novo

conteúdo. Você pode calcular o valor SHA-1 de uma *string* em Ruby incluindo a biblioteca SHA1 digest com o comando `require` e então chamando `Digest::SHA1.hexdigest()` com a *string*:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

O Git comprime o novo conteúdo com zlib, o que você pode fazer em Ruby com a biblioteca zlib. Primeiro, você precisa incluir a biblioteca e então executar `Zlib::Deflate.deflate()` no conteúdo:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

Por último, você irá salvar o seu conteúdo comprimido com zlib em um objeto no disco. Você irá determinar o caminho do objeto que você quer escrever (sendo os dois primeiros caracteres do valor do SHA-1 o nome do subdiretório, e os últimos 38 caracteres sendo o nome do arquivo dentro desse diretório). Em Ruby, você pode usar a função `FileUtils.mkdir_p()` para criar o subdiretório se ele não existir. Depois, abra o arquivo com `File.open()` e escreva o conteúdo previamente comprimido com zlib no arquivo chamando `write()` no *file handle* resultante:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

É isso - você criou um objeto *blob* do Git válido. Todos os objetos são armazenados do mesmo jeito, apenas com diferentes tipos - em vez da *string blob*, o cabeçalho começará com *commit* ou *tree*. Além disso, ainda que o conteúdo do *blob* possa ser quase qualquer coisa, o conteúdo do *commit* e da *tree* são especificamente formatados.

Referências do Git

Você pode executar algo como `git log 1a410e` para ver todo o seu histórico, mas você ainda precisa lembrar que `1a410e` é o último *commit* para poder caminhar nesse histórico para encontrar todos esses objetos. Você precisa de um arquivo em que você possa armazenar o valor do SHA-1 com um simples nome para que você possa usar essa referência em vez do valor de um SHA-1 puro.

No Git, chamamos isso de “referências” (*references*) ou “refs”; você pode encontrar os arquivos que contém os valores SHA-1 no diretório `.git/refs`. No projeto atual, este diretório não contém nenhum arquivo, mas contém uma simples estrutura:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Para criar uma nova referência que irá te ajudar a lembrar onde está seu último *commit*, você pode tecnicamente fazer algo tão simples quanto isto:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Agora, você pode usar a referência *head* que você acabou de criar em vez do valor SHA-1 nos seus comandos Git:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Nós não encorajamos você a editar diretamente arquivos de referência. O Git provê um comando mais seguro para fazer isso se você quiser atualizar uma referência, chamado `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Isto é o que uma *branch* é basicamente: uma simples referência para a cabeça de uma linha de trabalho. Para criar uma *branch* no segundo *commit*, você pode fazer isto:

```
$ git update-ref refs/heads/test cac0ca
```

Sua *branch* irá conter apenas o trabalho a partir desse *commit*:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Agora, seu banco de dados do Git conceitualmente aparenta ser algo assim:

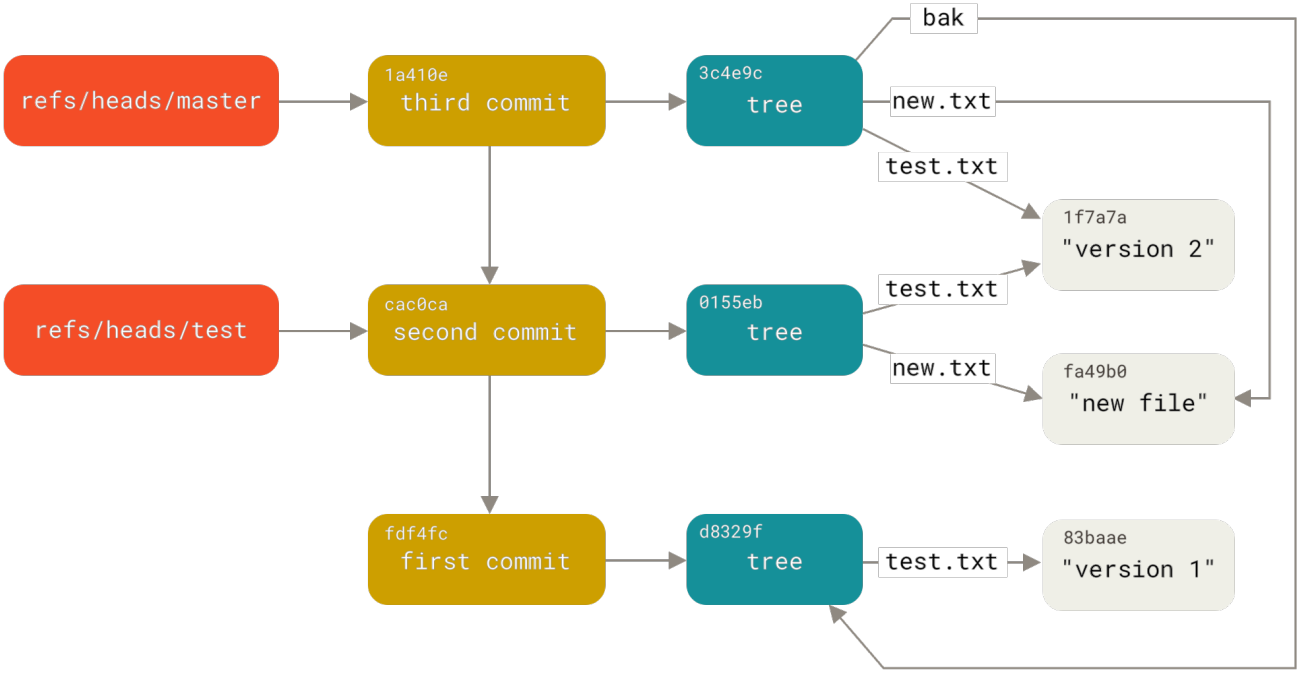


Figura 151. Git directory objects with branch head references included.

Quando você executa comandos como `git branch (nome da branch)`, o Git basicamente executa esse comando `update-ref` para adicionar o SHA-1 do último *commit* da *branch* que você está em qualquer nova referência que você quer criar.

A HEAD

A questão agora é, quando você executa `git branch (nome da branch)`, como o Git sabe o SHA-1 do último *commit*? A resposta é o arquivo HEAD.

O arquivo HEAD é uma referência simbólica para a *branch* que você está no momento. Queremos dizer por referência simbólica que, ao contrário de uma referência normal, em geral ela não contém um valor de um SHA-1, mas um ponteiro para outra referência. Se você olhar para o arquivo, normalmente você verá algo como isto:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Se você executar `git checkout test`, o Git atualizará o arquivo de forma que ele ficará assim:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Quando você executa `git commit`, ele cria um objeto *commit*, especificando como pai desse objeto *commit* o valor do SHA-1 que a referência contida em HEAD aponta.

Você pode alterar manualmente esse arquivo mas, novamente, um comando mais seguro existe para fazer isso: `symbolic-ref`. Você pode ler esse arquivo de sua HEAD através deste comando:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Você também pode definir o valor de HEAD:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Você não pode definir o valor de uma referência simbólica fora do estilo *refs*:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

Nós acabamos de falar sobre os três principais tipos de objetos, mas existe também um quarto. O objeto *tag* é bem parecido com um objeto *commit*, ele contém um *tagger*, a data, a mensagem, e uma referência. A principal diferença é que um objeto *tag* geralmente aponta para um *commit* em vez de uma *tree*. Ele é bem parecido com uma referência do tipo *branch*, mas ele nunca se move - ele sempre aponta para o mesmo *commit* mas dá a ele um nome mais amigável.

Como discutimos em Fundamentos de Git, exitem dois tipos de *tags*: anotada e leve. Você pode criar uma *tag* leve executando algo assim:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Isso é tudo que uma *tag* leve é - uma referência que nunca se move. Entretanto, uma *tag* anotada é mais complexa. Se você criar uma *tag* anotada, o Git cria um objeto *tag* e então escreve uma referência que aponta para ele em vez de apontar diretamente para o *commit*. Você pode ver isso criando uma *tag* anotada (`-a` especifica que é uma *tag* anotada):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Aqui está o valor SHA-1 do objeto que foi criado:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Agora, execute o comando `cat-file` no valor SHA-1:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Note que o item `object` aponta ao SHA-1 do *commit* que você criou a *tag*. Note também que ele não precisa apontar para um *commit*; você pode adicionar qualquer objeto do Git. No código-fonte do Git, por exemplo, o mantenedor adicionou sua chave GPG pública como um objeto *blob* e então criou uma *tag* para ele. Você pode ver a chave pública executando isto em um clone do repositório do Git:

```
$ git cat-file blob junio-gpg-pub
```

O repositório do kernel Linux também tem um objeto *tag* que não aponta para um *commit*: a primeira tag criada aponta para a *tree* inicial da importação do código-fonte.

Remotes

O terceiro tipo de referência que você verá é o *remote* (remoto). Se você adicionar um *remote* e fizer um *push* para ele, o Git armazenará o valor que você fez o *push* para ele para cada *branch* no diretório `refs/remotes`. Por exemplo, você pode adicionar um *remote* chamado `origin` e fazer o *push* da sua `master` para ele:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

Então, você pode ver onde a branch `master` no *remote* `origin` estava na última vez que você se comunicou com o servidor, olhando o arquivo `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remotes se diferenciam de *branches* (referências em `refs/heads`) principalmente pelo fato de normalmente serem somente-leitura. Você pode executar `git checkout` para um *remote*, mas o Git não irá apotar a HEAD para um, então você nunca irá atualizá-la com um commando `commit`. O Git gerencia elas como marcadores para a último estado conhecido de onde essas branches estavam

nos servidores.

Packfiles

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects – 4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresses the contents of these files with zlib, and you're not storing much, so all these files collectively take up only 925 bytes. You'll add some larger content to the repository to demonstrate an interesting feature of Git. To demonstrate, we'll add the `repo.rb` file from the Grit library – this is about a 22K source code file:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ git checkout master
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

If you look at the resulting tree, you can see the SHA-1 value your repo.rb file got for the blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

You can then use `git cat-file` to see how big that object is:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Now, modify that file a little, and see what happens:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Check the tree created by that commit, and you see something interesting:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

The blob is now a different blob, which means that although you added only a single line to the end of a 400-line file, Git stored that new content as a completely new object:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

You have two nearly identical 22K objects on your disk (each compressed to approximately 7K). Wouldn't it be nice if Git could store one of them in full but then the second object only as the delta between it and the first?

It turns out that it can. The initial format in which Git saves objects on disk is called a "loose" object format. However, occasionally Git packs up several of these objects into a single binary file called a "packfile" in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the `git gc` command manually, or if you push to a remote server. To see what happens, you can manually ask Git to pack up the objects by calling the `git gc` command:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

If you look in your objects directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

The objects that remain are the blobs that aren't pointed to by any commit – in this case, the "what is up, doc?" example and the "test content" example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the `gc` were collectively about 15K in size, the new packfile is only 7K. You've cut your disk usage by half by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed up:

```
$ git verify-pack -v .git/objects/pack/pack-
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Here, the `033b4` blob, which if you remember was the first version of your repo.rb file, is referencing the `b042a` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you can see that `b042a` takes up 22K of the file, but that `033b4` only takes up 9 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta – this is because you're most likely to need faster access to the most recent version of the file.

The really nice thing about this is that it can be repacked at any time. Git will occasionally repack your database automatically, always trying to save more space, but you can also manually repack at any time by running `git gc` by hand.

The Refspec

Throughout this book, we've used simple mappings from remote branches to local references, but they can be more complex. Suppose you add a remote like this:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

It adds a section to your `.git/config` file, specifying the name of the remote (`origin`), the URL of the remote repository, and the refspec for fetching:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
```

The format of the refspec is an optional `+`, followed by `<src>:<dst>`, where `<src>` is the pattern for references on the remote side and `<dst>` is where those references will be written locally. The `+` tells Git to update the reference even if it isn't a fast-forward.

In the default case that is automatically written by a `git remote add` command, Git fetches all the references under `refs/heads/` on the server and writes them to `refs/remotes/origin/` locally. So, if there is a `master` branch on the server, you can access the log of that branch locally via

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

They're all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

If you want Git instead to pull down only the `master` branch each time, and not every other branch on the remote server, you can change the fetch line to

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

This is just the default refspec for `git fetch` for that remote. If you want to do something one time, you can specify the refspec on the command line, too. To pull the `master` branch on the remote down to `origin/mymaster` locally, you can run

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

You can also specify multiple refspecs. On the command line, you can pull down several branches like so:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
     topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

In this case, the `master` branch pull was rejected because it wasn't a fast-forward reference. You can override that by specifying the `+` in front of the refspec.

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the `master` and `experiment` branches, add two lines:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

You can't use partial globs in the pattern, so this would be invalid:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

However, you can use namespaces (or directories) to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the `master` branch and any of the QA team's branches but nothing else, you can use a config section like this:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a `qa/` namespace in the first place? You accomplish that by using refspecs to push.

If the QA team wants to push their `master` branch to `qa/master` on the remote server, they can run

```
$ git push origin master:refs/heads/qa/master
```

If they want Git to do that automatically each time they run `git push origin`, they can add a `push` value to their config file:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

Again, this will cause a `git push origin` to push the local `master` branch to the remote `qa/master` branch by default.

Deleting References

You can also use the refspec to delete references from the remote server by running something like this:

```
$ git push origin :topic
```

Because the refspec is `<src>:<dst>`, by leaving off the `<src>` part, this basically says to make the `topic` branch on the remote nothing, which deletes it.

Transfer Protocols

Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

If you’re setting up a repository to be served read-only over HTTP, the dumb protocol is likely what will be used. This protocol is called “dumb” because it requires no Git-specific code on the server side during the transport process; the fetch process is a series of HTTP `GET` requests, where the client can assume the layout of the Git repository on the server.

NOTA	The dumb protocol is fairly rarely used these days. It’s difficult to secure or make private, so most Git hosts (both cloud-based and on-premises) will refuse to use it. It’s generally advised to use the smart protocol, which we describe a bit further on.

Let’s follow the `http-fetch` process for the simplegit library:

```
$ git clone http://server/simplegit-progit.git
```

The first thing this command does is pull down the `info/refs` file. This file is written by the `update-server-info` command, which is why you need to enable that as a `post-receive` hook in order for the HTTP transport to work properly:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

Now you have a list of the remote references and SHA-1s. Next, you look for what the HEAD reference is so you know what to check out when you’re finished:

```
=> GET HEAD
ref: refs/heads/master
```

You need to check out the `master` branch when you’ve completed the process. At this point, you’re

ready to start the walking process. Because your starting point is the `ca82a6` commit object you saw in the `info/refs` file, you start by fetching that:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

You get an object back – that object is in loose format on the server, and you fetched it over a static HTTP GET request. You can zlib-uncompress it, strip off the header, and look at the commit content:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Next, you have two more objects to retrieve – `cfda3b`, which is the tree of content that the commit we just retrieved points to; and `085bb3`, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

That gives you your next commit object. Grab the tree object:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops – it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this – the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there – this is a nice mechanism for projects that are forks of one another to share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the `objects/info/packs` file, which contains a listing of them (also generated by `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it – because the index lists the SHA-1s of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the `master` branch that was pointed to by the HEAD reference you downloaded at the beginning.

The Smart Protocol

The dumb protocol is simple but a bit inefficient, and it can't handle writing of data from the client to the server. The smart protocol is a more common method of transferring data, but it requires a process on the remote end that is intelligent about Git – it can read local data, figure out what the client has and needs, and generate a custom packfile for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

Uploading Data

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote side.

SSH

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which initiates a connection over SSH to your server. It tries to run a command on the remote server via an SSH call that looks something like this:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master□report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

The `git-receive-pack` command immediately responds with one line for each reference it currently

has – in this case, just the `master` branch and its SHA-1. The first line also has a list of the server's capabilities (here, `report-status`, `delete-refs`, and some others, including the client identifier).

Each line starts with a 4-character hex value specifying how long the rest of the line is. Your first line starts with 00a5, which is hexadecimal for 165, meaning that 165 bytes remain on that line. The next line is 0000, meaning the server is done with its references listing.

Now that it knows the server's state, your `send-pack` process determines what commits it has that the server doesn't. For each reference that this push will update, the `send-pack` process tells the `receive-pack` process that information. For instance, if you're updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like this:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
\
    refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
\
    refs/heads/experiment
0000
```

Git sends a line for each reference you're updating with the line's length, the old SHA-1, the new SHA-1, and the reference that is being updated. The first line also has the client's capabilities. The SHA-1 value of all '0's means that nothing was there before – because you're adding the experiment reference. If you were deleting a reference, you would see the opposite: all '0's on the right side.

Next, the client sends a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

```
000eunpack ok
```

HTTP(S)

This process is mostly the same over HTTP, though the handshaking is a bit different. The connection is initiated with this request:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master□report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

That's the end of the first client-server exchange. The client then makes another request, this time a `POST`, with the data that `send-pack` provides.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

The `POST` request includes the `send-pack` output and the packfile as its payload. The server then indicates success or failure with its HTTP response.

Downloading Data

When you download data, the `fetch-pack` and `upload-pack` processes are involved. The client initiates a `fetch-pack` process that connects to an `upload-pack` process on the remote side to negotiate what data will be transferred down.

SSH

If you're doing the fetch over SSH, `fetch-pack` runs something like this:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

After `fetch-pack` connects, `upload-pack` sends back something like this:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

This is very similar to what `receive-pack` responds with, but the capabilities are different. In addition, it sends back what HEAD points to (`symref=HEAD:refs/heads/master`) so the client knows what to check out if this is a clone.

At this point, the `fetch-pack` process looks at what objects it has and responds with the objects that it needs by sending "want" and then the SHA-1 it wants. It sends all the objects it already has with "have" and then the SHA-1. At the end of this list, it writes "done" to initiate the `upload-pack` process to begin sending the packfile of the data it needs:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

The handshake for a fetch operation takes two HTTP requests. The first is a `GET` to the same endpoint used in the dumb protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

This is very similar to invoking `git-upload-pack` over an SSH connection, but the second exchange is performed as a separate request:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Again, this is the same format as above. The response to this request indicates success or failure, and includes the packfile.

Protocols Summary

This section contains a very basic overview of the transfer protocols. The protocol includes many other features, such as `multi_ack` or `side-band` capabilities, but covering them is outside the scope of this book. We've tried to give you a sense of the general back-and-forth between client and server; if you need more knowledge than this, you'll probably want to take a look at the Git source code.

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called "auto gc". Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The "gc" stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

You can run auto gc manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50

packfiles for Git to fire up a real gc command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in Referências do Git. You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

        third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

       modified repo.rb a bit
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Cool – now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the string "dangling commit". You can recover it the same way, by adding a branch that points to that SHA-1.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a `git clone` downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oops – you didn't want to add a huge tarball to your project. Better get rid of it:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Now, `gc` your database and see how much space you're using:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-29…69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in Enforcing a Specific Commit-Message Format. If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

You must rewrite all the commits downstream from `7b30847` to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in Rewriting History:

```
$ git filter-branch --index-filter \
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in Rewriting History, except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` – you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the `7b30847` commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Variáveis de ambiente

O Git sempre é executado no terminal `bash`, e usa um certo conjunto de variáveis de ambiente para determinar como ele se comporta. Ocasionalmente, é útil saber o que são e como podem ser usadas para fazer o Git se comportar da maneira que você deseja. Esta não é uma lista completa de todas as variáveis de ambiente às quais o Git presta atenção, mas cobriremos as mais úteis.

Comportamentos globais

Parte do comportamento geral do Git como um programa de computador depende de variáveis de ambiente.

`GIT_EXEC_PATH` determina onde o Git procura seus subprogramas (como `git-commit`, `git-diff`, e outros). Você pode verificar a configuração atual executando `git --exec-path`.

`HOME` geralmente não é considerado customizável (muitas outras coisas dependem disso), mas é onde o Git procura o arquivo de configuração global. Se você quer uma instalação do Git realmente portátil, que seja completa com configuração global, você pode sobrescrever `HOME` no ambiente do terminal onde o Git portátil será executado.

`PREFIX` é semelhante, mas é voltado para a configuração de sistema. O Git procura por este arquivo em `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, quando definido, desativa o uso do arquivo de configuração de sistema. Isso é útil se a configuração do sistema está interferindo nos comandos, mas você não tem acesso para alterá-la ou removê-la.

`GIT_PAGER` controla o programa usado para exibir a saída de múltiplas páginas na linha de comando. Se não for definido, `PAGER` será usado como um substituto.

`GIT_EDITOR` é o editor que o Git iniciará quando o usuário precisar editar algum texto (uma mensagem de *commit*, por exemplo). Se não for definido, `EDITOR` será usado.

Repository Locations

Git uses several environment variables to determine how it interfaces with the current repository.

`GIT_DIR` is the location of the `.git` folder. If this isn't specified, Git walks up the directory tree until it gets to `~` or `/`, looking for a `.git` directory at every step.

`GIT_CEILING_DIRECTORIES` controls the behavior of searching for a `.git` directory. If you access directories that are slow to load (such as those on a tape drive, or across a slow network connection), you may want to have Git stop trying earlier than it might otherwise, especially if Git is invoked when building your shell prompt.

`GIT_WORK_TREE` is the location of the root of the working directory for a non-bare repository. If not specified, the parent directory of `$GIT_DIR` is used.

`GIT_INDEX_FILE` is the path to the index file (non-bare repositories only).

`GIT_OBJECT_DIRECTORY` can be used to specify the location of the directory that usually resides at `.git/objects`.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` is a colon-separated list (formatted like `/dir/one:/dir/two:…`) which tells Git where to check for objects if they aren't in `GIT_OBJECT_DIRECTORY`. If you happen to have a lot of projects with large files that have the exact same contents, this can be used to avoid storing too many copies of them.

Pathspecss

Um "pathspec" refere-se a como você especifica caminhos para coisas no Git, incluindo o uso de curingas (`wildcards`). Eles são usados no arquivo `.gitignore`, mas também na linha de comando (`git add *.c`).

`GIT_GLOB_PATHSPECS` **e** `GIT_NOGLOB_PATHSPECS` controlam o comportamento padrão dos curingas em pathspecs. Se `GIT_GLOB_PATHSPECS` for definido como 1, os caracteres curinga agem como curingas (que é o padrão); se `GIT_NOGLOB_PATHSPECS` for definido como 1, os caracteres curinga apenas combinam com eles mesmos, o que significa que algo como `**.c**` **iria apenas corresponder a um arquivo *com o nome* "**.c", ao invés de qualquer arquivo cujo nome termine com `.c`. Você pode sobrescrever isso em casos individuais iniciando o pathspec com `:(glob)` ou `:(literal)`, assim como em `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` desativa ambos os comportamentos acima; nenhum caractere curinga funcionará e os prefixos de substituição também ficarão desabilitados.

`GIT_ICASE_PATHSPECS` define todos os pathspecs para funcionarem sem distinção entre maiúsculas e minúsculas.

Fazendo commits

A criação final de um commit, que é um objeto Git, geralmente é feita por `git-commit-tree`, que usa essas variáveis de ambiente como sua fonte primária de informação, usando os valores de configuração apenas se essas não estiverem presentes.

`GIT_AUTHOR_NAME` é um nome legível para o campo "author".

`GIT_AUTHOR_EMAIL` é o email para o campo "author".

`GIT_AUTHOR_DATE` é a data e hora (*timestamp*) para o campo “author”.

`GIT_COMMITTER_NAME` define um nome legível para o campo “committer”.

`GIT_COMMITTER_EMAIL` é o email para o campo “committer”.

`GIT_COMMITTER_DATE` é a data e hora (*timestamp*) para o campo “committer”.

`EMAIL` é o endereço de e-mail substituto caso o valor de configuração `user.email` não esteja definido. Se *esse* não estiver definido, o Git substitui pelo usuário do sistema e nome do computador (hostname).

Rede e Internet

Git usa a biblioteca `curl` para fazer operações de rede sobre HTTP, então **`GIT_CURL_VERBOSE`** diz ao Git para emitir todas as mensagens geradas por aquela biblioteca. Isso é semelhante a fazer `curl -v` na linha de comando.

`GIT_SSL_NO_VERIFY` diz ao Git para não verificar os certificados SSL. Às vezes, isso pode ser necessário se você estiver usando um certificado autoassinado para servir repositórios Git sobre HTTPS ou se estiver no meio da configuração de um servidor Git, mas ainda não instalou um certificado completo.

Se a taxa de dados de uma operação HTTP for inferior a `GIT_HTTP_LOW_SPEED_LIMIT` bytes por segundo por mais de `GIT_HTTP_LOW_SPEED_TIME` segundos, o Git irá abortar a operação. Esses valores substituem os valores de configuração `http.lowSpeedLimit` e `http.lowSpeedTime`.

- `GIT_HTTP_USER_AGENT`* define a string do agente do usuário usada pelo Git ao se comunicar por HTTP. O padrão é um valor como `git/2.0.0`.

Diffing and Merging

`GIT_DIFF_OPTS` is a bit of a misnomer. The only valid values are `-u<n>` or `--unified=<n>`, which controls the number of context lines shown in a `git diff` command.

`GIT_EXTERNAL_DIFF` is used as an override for the `diff.external` configuration value. If it’s set, Git will invoke this program when `git diff` is invoked.

`GIT_DIFF_PATH_COUNTER` and `GIT_DIFF_PATH_TOTAL` are useful from inside the program specified by `GIT_EXTERNAL_DIFF` or `diff.external`. The former represents which file in a series is being diffed (starting with 1), and the latter is the total number of files in the batch.

`GIT_MERGE_VERBOSITY` controls the output for the recursive merge strategy. The allowed values are as follows:

- 0 outputs nothing, except possibly a single error message.
- 1 shows only conflicts.
- 2 also shows file changes.
- 3 shows when files are skipped because they haven’t changed.

- 4 shows all paths as they are processed.
- 5 and above show detailed debugging information.

The default value is 2.

Debugging

Want to *really* know what Git is up to? Git has a fairly complete set of traces embedded, and all you need to do is turn them on. The possible values of these variables are as follows:

- "true", "1", or "2" – the trace category is written to stderr.
- An absolute path starting with / – the trace output will be written to that file.

`GIT_TRACE` controls general traces, which don't fit into any specific category. This includes the expansion of aliases, and delegation to other sub-programs.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554               trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341       trace: run_command: 'git-lga'
20:12:49.879529 git.c:282               trace: alias expansion: lga => 'log' '--graph'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341       trace: run_command: 'less'
20:12:49.899675 run-command.c:192       trace: exec: 'less'
```

`GIT_TRACE_PACK_ACCESS` controls tracing of packfile access. The first field is the packfile being accessed, the second is the offset within that file:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 35175
# […]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
56914983
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

`GIT_TRACE_PACKET` enables packet-level tracing for network operations.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46          packet:          git< # service=git-upload-
pack
20:15:14.867071 pkt-line.c:46          packet:          git< 0000
20:15:14.867079 pkt-line.c:46          packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-band side-
band-64k ofs-delta shallow no-progress include-tag multi_ack_detailed no-done
symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46          packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46          packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# […]
```

`GIT_TRACE_PERFORMANCE` controls logging of performance data. The output shows how long each particular `git` invocation takes.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414            performance: 0.374835000 s: git command: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414            performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414            performance: 3.715349000 s: git command: 'git'
'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--
reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414            performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414            performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414            performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414            performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414            performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414            performance: 6.112217000 s: git command: 'git'
'gc'
```

`GIT_TRACE_SETUP` shows information about what Git is discovering about the repository and environment it's interacting with.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315            setup: git_dir: .git
20:19:47.087184 trace.c:316            setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317            setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318            setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Miscellaneous

`GIT_SSH`, if specified, is a program that is invoked instead of `ssh` when Git tries to connect to an SSH host. It is invoked like `$GIT_SSH [username@]host [-p <port>] <command>`. Note that this isn't the easiest way to customize how `ssh` is invoked; it won't support extra command-line parameters, so you'd have to write a wrapper script and set `GIT_SSH` to point to it. It's probably easier just to use the `~/.ssh/config` file for that.

`GIT_ASKPASS` is an override for the `core.askpass` configuration value. This is the program invoked whenever Git needs to ask the user for credentials, which can expect a text prompt as a command-line argument, and should return the answer on `stdout`. (See Credential Storage for more on this subsystem.)

`GIT_NAMESPACE` controls access to namespaced refs, and is equivalent to the `--namespace` flag. This is mostly useful on the server side, where you may want to store multiple forks of a single repository in one repository, only keeping the refs separate.

`GIT_FLUSH` can be used to force Git to use non-buffered I/O when writing incrementally to stdout. A value of 1 causes Git to flush more often, a value of 0 causes all output to be buffered. The default value (if this variable is not set) is to choose an appropriate buffering scheme depending on the activity and the output mode.

`GIT_REFLOG_ACTION` lets you specify the descriptive text written to the reflog. Here's an example:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

Sumário

Você deve ter um ótimo entendimento do que o Git faz internamente e, de alguma forma, como ele é implementado. Este capítulo cobriu um certo número de comandos de encanamento - comandos que estão em um nível mais baixo e simples do que os que você aprendeu no resto do livro. Entender como o Git funciona a um baixo mais nível torna mais fácil entender o porquê de ele fazer cada coisa e também escrever suas prórprias ferramentas e *scripts* auxiliares para fazer o seu *workflow* trabalhar a seu favor.

O Git como um sistema de arquivos de conteúdo endereçável é uma ferramenta muito poderosa que você poderá usar mais do que apenas um VCS. Esperamos que você use este novo conhecimento do funcionamento interno do Git para implementar incríveis aplicações desta tecnologia e se sentir mais confortável em usar o Git de forma mais avançada.

Apêndice A: Git em Outros Ambientes

Se você leu o livro todo, você aprendeu bastante sobre como usar Git na linha de comando. Você pode trabalhar com arquivos locais, conectar seu repositório a outros através de uma rede, e trabalhar efetivamente com outrem. Mas a história não acaba aí: Git é geralmente utilizado como parte de um ecossistema maior, e o terminal nem sempre é a melhor forma de se trabalhar com esse ecossistema. Agora vamos dar um olhada em alguns dos outros tipos de ambientes onde Git pode ser útil, e como outras aplicações (inclusive a sua) funcionam junto com Git.

Graphical Interfaces

Git's native environment is in the terminal. New features show up there first, and only at the command line is the full power of Git completely at your disposal. But plain text isn't the best choice for all tasks; sometimes a visual representation is what you need, and some users are much more comfortable with a point-and-click interface.

It's important to note that different interfaces are tailored for different workflows. Some clients expose only a carefully curated subset of Git functionality, in order to support a specific way of working that the author considers effective. When viewed in this light, none of these tools can be called "better" than any of the others, they're simply more fit for their intended purpose. Also note that there's nothing these graphical clients can do that the command-line client can't; the command-line is still where you'll have the most power and control when working with your repositories.

`gitk` and `git-gui`

When you install Git, you also get its visual tools, `gitk` and `git-gui`.

`gitk` is a graphical history viewer. Think of it like a powerful GUI shell over `git log` and `git grep`. This is the tool to use when you're trying to find something that happened in the past, or visualize your project's history.

Gitk is easiest to invoke from the command-line. Just `cd` into a Git repository, and type:

```
$ gitk [git log options]
```

Gitk accepts many command-line options, most of which are passed through to the underlying `git log` action. Probably one of the most useful is the `--all` flag, which tells gitk to show commits reachable from *any* ref, not just HEAD. Gitk's interface looks like this:

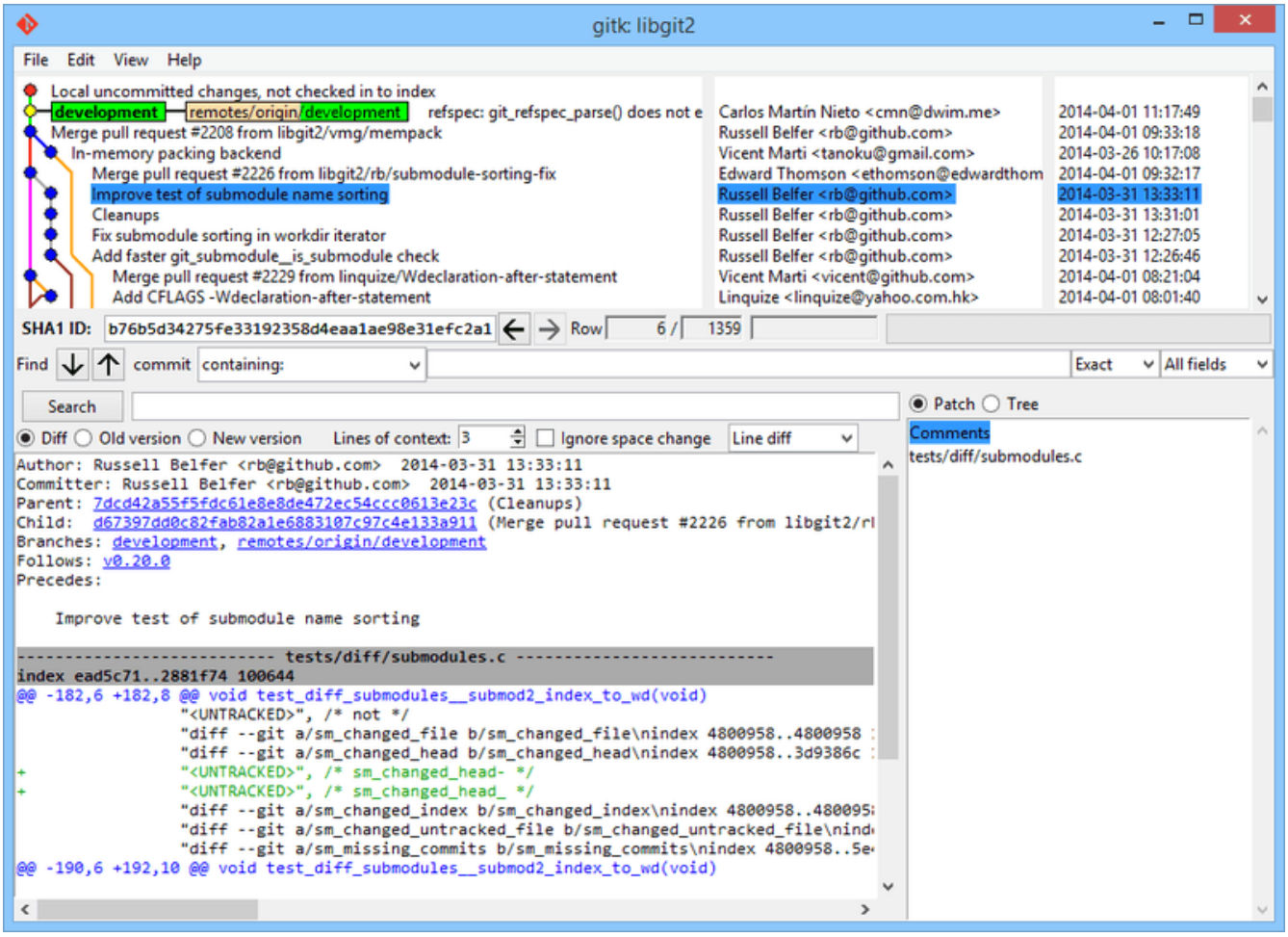


Figura 152. The `gitk` history viewer.

On the top is something that looks a bit like the output of `git log --graph`; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

`git-gui`, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

```
$ git gui
```

And it looks something like this:

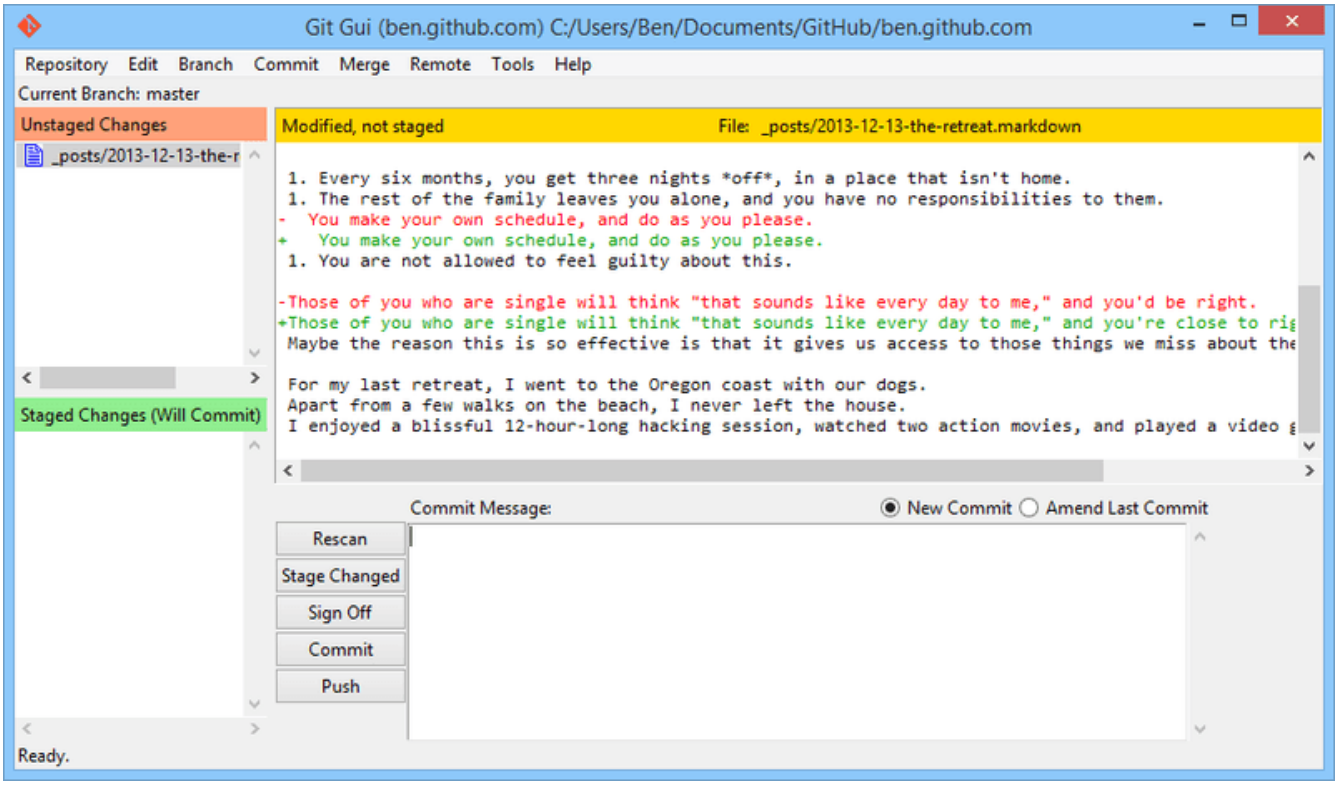


Figura 153. The `git-gui` commit tool.

On the left is the index; unstaged changes are on top, staged changes on the bottom. You can move entire files between the two states by clicking on their icons, or you can select a file for viewing by clicking on its name.

At top right is the diff view, which shows the changes for the currently-selected file. You can stage individual hunks (or individual lines) by right-clicking in this area.

At the bottom right is the message and action area. Type your message into the text box and click “Commit” to do something similar to `git commit`. You can also choose to amend the last commit by choosing the “Amend” radio button, which will update the “Staged Changes” area with the contents of the last commit. Then you can simply stage or unstage some changes, alter the commit message, and click “Commit” again to replace the old commit with a new one.

`gitk` and `git-gui` are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

GitHub for Mac and Windows

GitHub has created two workflow-oriented Git clients: one for Windows, and one for Mac. These clients are a good example of workflow-oriented tools – rather than expose *all* of Git’s functionality, they instead focus on a curated set of commonly-used features that work well together. They look like this:

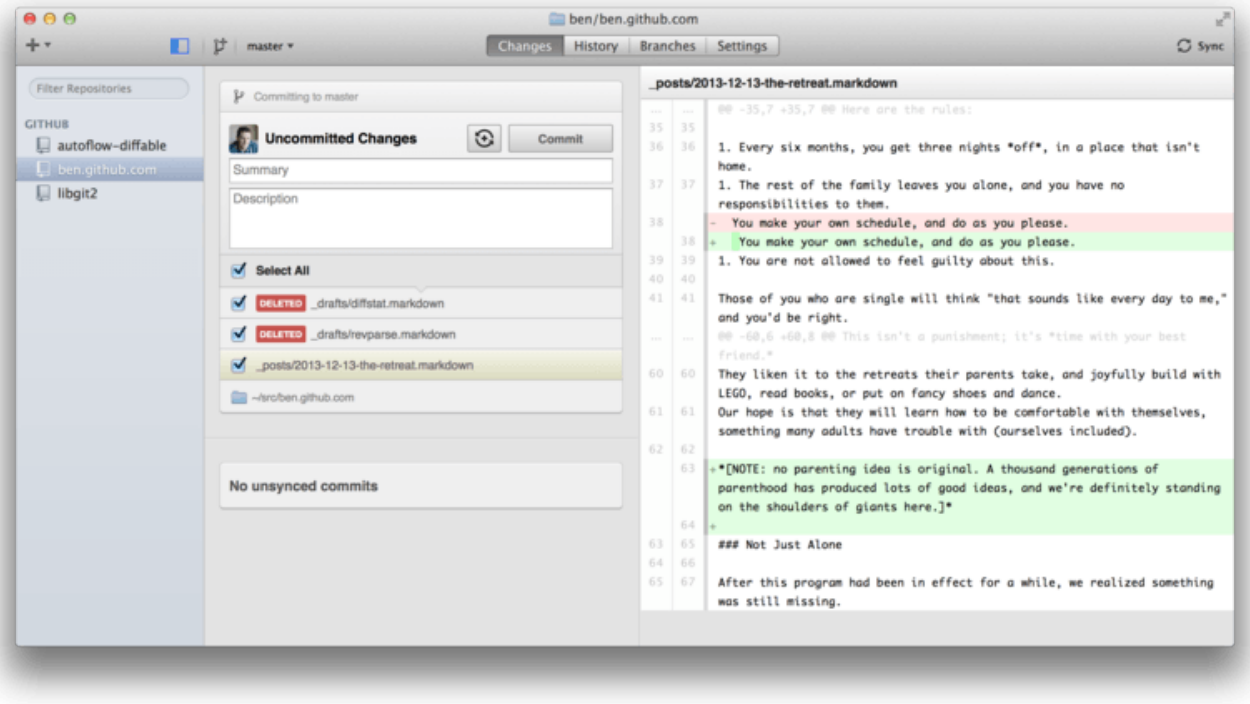


Figura 154. GitHub for Mac.

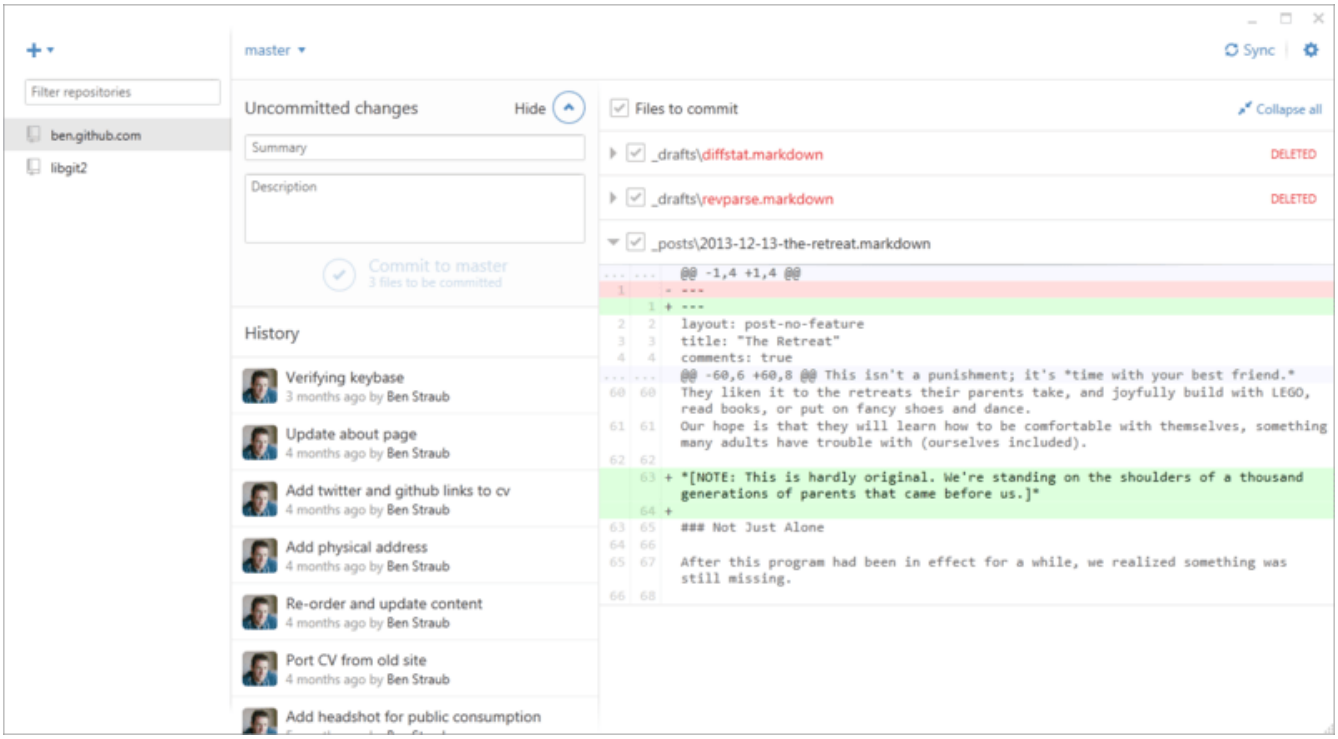


Figura 155. GitHub for Windows.

They are designed to look and work very much alike, so we'll treat them like a single product in this chapter. We won't be doing a detailed rundown of these tools (they have their own documentation), but a quick tour of the "changes" view (which is where you'll spend most of your time) is in order.

- On the left is the list of repositories the client is tracking; you can add a repository (either by cloning or attaching locally) by clicking the "+" icon at the top of this area.
- In the center is a commit-input area, which lets you input a commit message, and select which files should be included. (On Windows, the commit history is displayed directly below this; on

Mac, it's on a separate tab.)

- On the right is a diff view, which shows what's changed in your working directory, or which changes were included in the selected commit.
- The last thing to notice is the "Sync" button at the top-right, which is the primary way you interact over the network.

NOTA You don't need a GitHub account to use these tools. While they're designed to highlight GitHub's service and recommended workflow, they will happily work with any repository, and do network operations with any Git host.

Installation

GitHub for Windows can be downloaded from https://windows.github.com, and GitHub for Mac from https://mac.github.com. When the applications are first run, they walk you through all the first-time Git setup, such as configuring your name and email address, and both set up sane defaults for many common configuration options, such as credential caches and CRLF behavior.

Both are "evergreen" – updates are downloaded and installed in the background while the applications are open. This helpfully includes a bundled version of Git, which means you probably won't have to worry about manually updating it again. On Windows, the client includes a shortcut to launch Powershell with Posh-git, which we'll talk more about later in this chapter.

The next step is to give the tool some repositories to work with. The client shows you a list of the repositories you have access to on GitHub, and can clone them in one step. If you already have a local repository, just drag its directory from the Finder or Windows Explorer into the GitHub client window, and it will be included in the list of repositories on the left.

Recommended Workflow

Once it's installed and configured, you can use the GitHub client for many common Git tasks. The intended workflow for this tool is sometimes called the "GitHub Flow." We cover this in more detail in O fluxo do GitHub, but the general gist is that (a) you'll be committing to a branch, and (b) you'll be syncing up with a remote repository fairly regularly.

Branch management is one of the areas where the two tools diverge. On Mac, there's a button at the top of the window for creating a new branch:

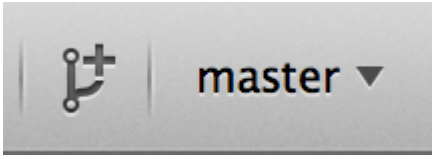


Figura 156. "Create Branch" button on Mac.

On Windows, this is done by typing the new branch's name in the branch-switching widget:

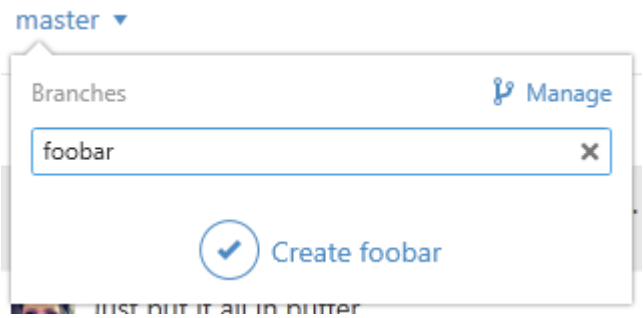


Figura 157. Creating a branch on Windows.

Once your branch is created, making new commits is fairly straightforward. Make some changes in your working directory, and when you switch to the GitHub client window, it will show you which files changed. Enter a commit message, select the files you'd like to include, and click the "Commit" button (ctrl-enter or -enter).

The main way you interact with other repositories over the network is through the "Sync" feature. Git internally has separate operations for pushing, fetching, merging, and rebasing, but the GitHub clients collapse all of these into one multi-step feature. Here's what happens when you click the Sync button:

1. `git pull --rebase`. If this fails because of a merge conflict, fall back to `git pull --no-rebase`.
2. `git push`.

This is the most common sequence of network commands when working in this style, so squashing them into one command saves a lot of time.

Summary

These tools are very well-suited for the workflow they're designed for. Developers and non-developers alike can be collaborating on a project within minutes, and many of the best practices for this kind of workflow are baked into the tools. However, if your workflow is different, or you want more control over how and when network operations are done, we recommend you use another client or the command line.

Other GUIs

There are a number of other graphical Git clients, and they run the gamut from specialized, single-purpose tools all the way to apps that try to expose everything Git can do. The official Git website has a curated list of the most popular clients at http://git-scm.com/downloads/guis. A more comprehensive list is available on the Git wiki site, at https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git in Visual Studio

Starting with Visual Studio 2013 Update 1, Visual Studio users have a Git client built directly into their IDE. Visual Studio has had source-control integration features for quite some time, but they were oriented towards centralized, file-locking systems, and Git was not a good match for this workflow. Visual Studio 2013's Git support has been separated from this older feature, and the

result is a much better fit between Studio and Git.

To locate the feature, open a project that's controlled by Git (or just `git init` an existing project), and select View > Team Explorer from the menu. You'll see the "Connect" view, which looks a bit like this:

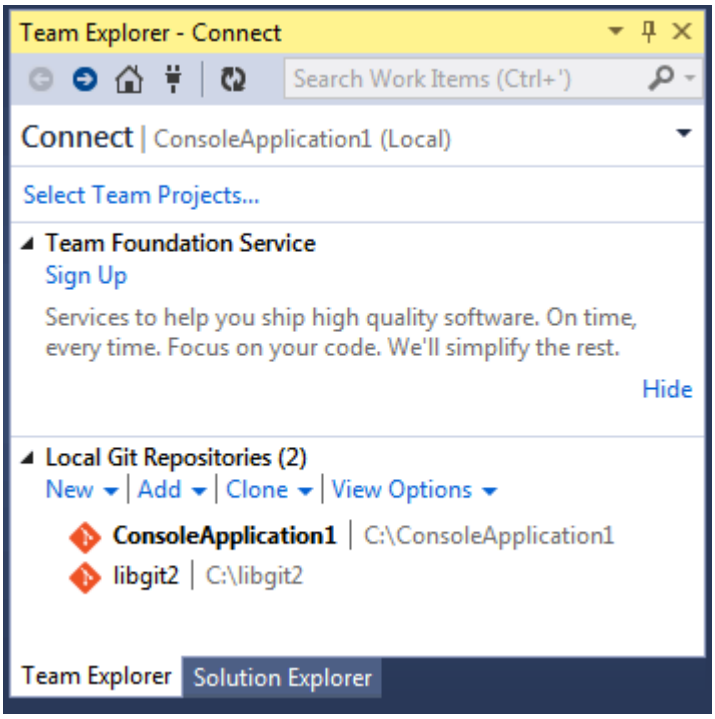


Figura 158. Connecting to a Git repository from Team Explorer.

Visual Studio remembers all of the projects you've opened that are Git-controlled, and they're available in the list at the bottom. If you don't see the one you want there, click the "Add" link and type in the path to the working directory. Double clicking on one of the local Git repositories leads you to the Home view, which looks like The "Home" view for a Git repository in Visual Studio.. This is a hub for performing Git actions; when you're *writing* code, you'll probably spend most of your time in the "Changes" view, but when it comes time to pull down changes made by your teammates, you'll use the "Unsynced Commits" and "Branches" views.

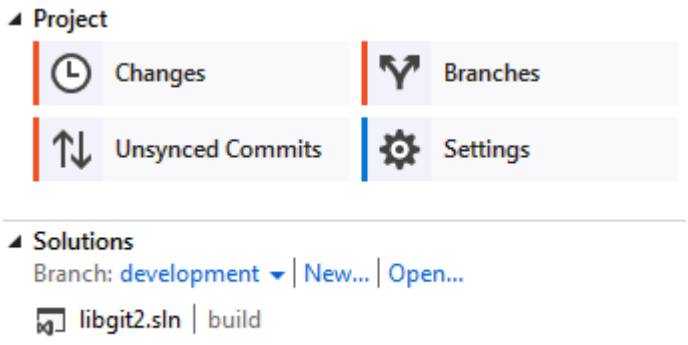


Figura 159. The "Home" view for a Git repository in Visual Studio.

Visual Studio now has a powerful task-focused UI for Git. It includes a linear history view, a diff viewer, remote commands, and many other capabilities. For complete documentation of this feature (which doesn't fit here), go to https://learn.microsoft.com/en-us/visualstudio/version-control/git-with-visual-studio?view=vs-2022.

Git in Eclipse

Eclipse ships with a plugin called Egit, which provides a fairly-complete interface to Git operations. It's accessed by switching to the Git Perspective (Window > Open Perspective > Other..., and select "Git").

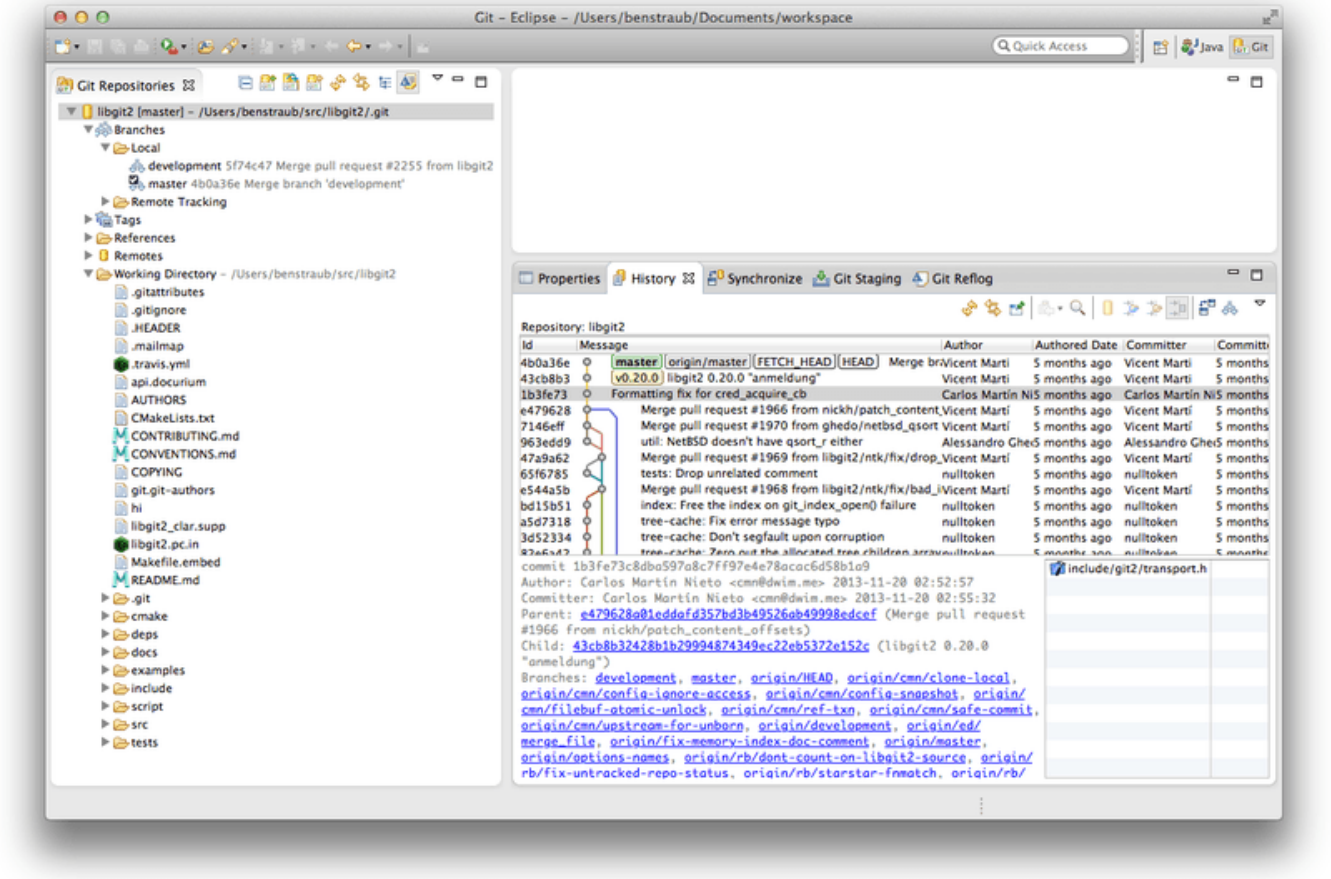


Figura 160. Eclipse's EGit environment.

EGit comes with plenty of great documentation, which you can find by going to Help > Help Contents, and choosing the "EGit Documentation" node from the contents listing.

Git in Bash

If you're a Bash user, you can tap into some of your shell's features to make your experience with Git a lot friendlier. Git actually ships with plugins for several shells, but it's not turned on by default.

First, you need to get a copy of the `contrib/completion/git-completion.bash` file out of the Git source code. Copy it somewhere handy, like your home directory, and add this to your `.bashrc`:

```
. ~/git-completion.bash
```

Once that's done, change your directory to a Git repository, and type:

```
$ git chec<tab>
```

...and Bash will auto-complete to `git checkout`. This works with all of Git's subcommands, command-line parameters, and remotes and ref names where appropriate.

It's also useful to customize your prompt to show information about the current directory's Git repository. This can be as simple or complex as you want, but there are generally a few key pieces of information that most people want, like the current branch, and the status of the working directory. To add these to your prompt, just copy the `contrib/completion/git-prompt.sh` file from Git's source repository to your home directory, add something like this to your `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

The `\w` means print the current working directory, the `\$` prints the `$` part of the prompt, and `__git_ps1 " (%s)"` calls the function provided by `git-prompt.sh` with a formatting argument. Now your bash prompt will look like this when you're anywhere inside a Git-controlled project:

Figura 161. Customized `bash` prompt.

Both of these scripts come with helpful documentation; take a look at the contents of `git-completion.bash` and `git-prompt.sh` for more information.

Git in Zsh

Zsh also ships with a tab-completion library for Git. To use it, simply run `autoload -Uz compinit && compinit` in your `.zshrc`. Zsh's interface is a bit more powerful than Bash's:

```
$ git che<tab>
check-attr        -- display gitattributes information
check-ref-format  -- ensure that a reference name is well formed
checkout          -- checkout branch or paths to working tree
checkout-index    -- copy files from index to working directory
cherry            -- find commits not merged upstream
cherry-pick       -- apply changes introduced by some existing commits
```

Ambiguous tab-completions aren't just listed; they have helpful descriptions, and you can graphically navigate the list by repeatedly hitting tab. This works with Git commands, their arguments, and names of things inside the repository (like refs and remotes), as well as filenames and all the other things Zsh knows how to tab-complete.

Zsh ships with a framework for getting information from version control systems, called `vcs_info`. To include the branch name in the prompt on the right side, add these lines to your `~/.zshrc` file:

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_'%# '
zstyle ':vcs_info:git:*' formats '%b'
```

This results in a display of the current branch on the right-hand side of the terminal window, whenever your shell is inside a Git repository. (The left side is supported as well, of course; just uncomment the assignment to PROMPT.) It looks a bit like this:

Figura 162. Customized `zsh` prompt.

For more information on vcs_info, check out its documentation in the `zshcontrib(1)` manual page, or online at `http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information`.

Instead of vcs_info, you might prefer the prompt customization script that ships with Git, called `git-prompt.sh`; see https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh for

details. `git-prompt.sh` is compatible with both Bash and Zsh.

Zsh is powerful enough that there are entire frameworks dedicated to making it better. One of them is called "oh-my-zsh", and it can be found at https://github.com/robbyrussell/oh-my-zsh. oh-my-zsh's plugin system comes with powerful git tab-completion, and it has a variety of prompt "themes", many of which display version-control data. An example of an oh-my-zsh theme. is just one example of what can be done with this system.

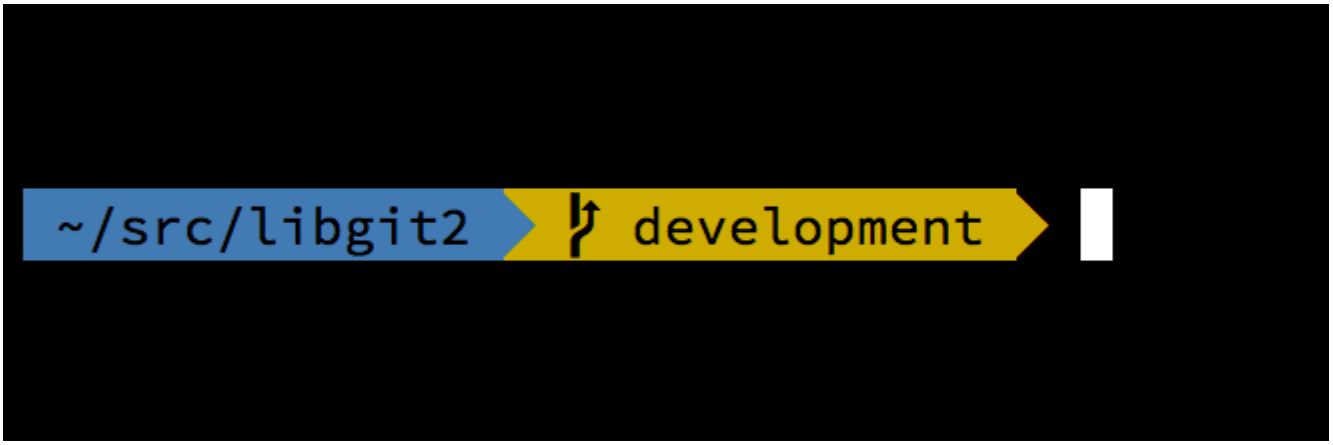


Figura 163. An example of an oh-my-zsh theme.

Git in Powershell

The standard command-line terminal on Windows (`cmd.exe`) isn't really capable of a customized Git experience, but if you're using Powershell, you're in luck. A package called Posh-Git (https://github.com/dahlbyk/posh-git) provides powerful tab-completion facilities, as well as an enhanced prompt to help you stay on top of your repository status. It looks like this:

Figura 164. Powershell with Posh-git.

Just download a Posh-Git release from (https://github.com/dahlbyk/posh-git), and uncompress it to the `WindowsPowershell` directory. Then open a Powershell prompt as the administrator, and do this:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

This will add the proper line to your `profile.ps1` file, and posh-git will be active the next time you open your prompt.

Resumo

Você aprendeu como aproveitar o potencial do Git dentro de ferramentas que você utiliza durante o seu trabalho cotidiando, e também como acessar repositórios Git a partir dos seus próprios programas.

Apêndice B: Embedding Git in your Applications

If your application is for developers, chances are good that it could benefit from integration with source control. Even non-developer applications, such as document editors, could potentially benefit from version-control features, and Git's model works very well for many different scenarios.

If you need to integrate Git with your application, you have essentially three choices: spawning a shell and using the Git command-line tool; Libgit2; and JGit.

Command-line Git

One option is to spawn a shell process and use the Git command-line tool to do the work. This has the benefit of being canonical, and all of Git's features are supported. This also happens to be fairly easy, as most runtime environments have a relatively simple facility for invoking a process with command-line arguments. However, this approach does have some downsides.

One is that all the output is in plain text. This means that you'll have to parse Git's occasionally-changing output format to read progress and result information, which can be inefficient and error-prone.

Another is the lack of error recovery. If a repository is corrupted somehow, or the user has a malformed configuration value, Git will simply refuse to perform many operations.

Yet another is process management. Git requires you to maintain a shell environment on a separate process, which can add unwanted complexity. Trying to coordinate many of these processes (especially when potentially accessing the same repository from several processes) can be quite a challenge.

Libgit2

Another option at your disposal is to use Libgit2. Libgit2 is a dependency-free implementation of Git, with a focus on having a nice API for use within other programs. You can find it at http://libgit2.github.com.

First, let's take a look at what the C API looks like. Here's a whirlwind tour:

```
// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);
```

The first couple of lines open a Git repository. The `git_repository` type represents a handle to a repository with a cache in memory. This is the simplest method, for when you know the exact path to a repository's working directory or `.git` folder. There's also the `git_repository_open_ext` which includes options for searching, `git_clone` and friends for making a local clone of a remote repository, and `git_repository_init` for creating an entirely new repository.

The second chunk of code uses rev-parse syntax (see Branch References for more on this) to get the commit that HEAD eventually points to. The type returned is a `git_object` pointer, which represents something that exists in the Git object database for a repository. `git_object` is actually a "parent" type for several different kinds of objects; the memory layout for each of the "child" types is the same as for `git_object`, so you can safely cast to the right one. In this case, `git_object_type(commit)` would return `GIT_OBJ_COMMIT`, so it's safe to cast to a `git_commit` pointer.

The next chunk shows how to access the commit's properties. The last line here uses a `git_oid` type; this is Libgit2's representation for a SHA-1 hash.

From this sample, a couple of patterns have started to emerge:

- If you declare a pointer and pass a reference to it into a Libgit2 call, that call will probably return an integer error code. A `0` value indicates success; anything less is an error.
- If Libgit2 populates a pointer for you, you're responsible for freeing it.
- If Libgit2 returns a `const` pointer from a call, you don't have to free it, but it will become invalid when the object it belongs to is freed.
- Writing C is a bit painful.

That last one means it isn't very probable that you'll be writing C when using Libgit2. Fortunately, there are a number of language-specific bindings available that make it fairly easy to work with Git repositories from your specific language and environment. Let's take a look at the above example written using the Ruby bindings for Libgit2, which are named Rugged, and can be found at

https://github.com/libgit2/rugged.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

As you can see, the code is much less cluttered. Firstly, Rugged uses exceptions; it can raise things like `ConfigError` or `ObjectError` to signal error conditions. Secondly, there's no explicit freeing of resources, since Ruby is garbage-collected. Let's take a look at a slightly more complicated example: crafting a commit from scratch

```
blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo), ③
    :author => sig,
    :committer => sig, ④
    :message => "Add newfile.txt", ⑤
    :parents => repo.empty? ? [] : [ repo.head.target ].compact, ⑥
    :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧
```

① Create a new blob, which contains the contents of a new file.

② Populate the index with the head commit's tree, and add the new file at the path `newfile.txt`.

③ This creates a new tree in the ODB, and uses it for the new commit.

④ We use the same signature for both the author and committer fields.

⑤ The commit message.

⑥ When creating a commit, you have to specify the new commit's parents. This uses the tip of HEAD for the single parent.

⑦ Rugged (and Libgit2) can optionally update a reference when making a commit.

⑧ The return value is the SHA-1 hash of a new commit object, which you can then use to get a `Commit` object.

The Ruby code is nice and clean, but since Libgit2 is doing the heavy lifting, this code will run pretty fast, too. If you're not a rubyist, we touch on some other bindings in Other Bindings.

Advanced Functionality

Libgit2 has a couple of capabilities that are outside the scope of core Git. One example is pluggability: Libgit2 allows you to provide custom "backends" for several types of operation, so you can store things in a different way than stock Git does. Libgit2 allows custom backends for configuration, ref storage, and the object database, among other things.

Let's take a look at how this works. The code below is borrowed from the set of backend examples provided by the Libgit2 team (which can be found at https://github.com/libgit2/libgit2-backends). Here's how a custom backend for the object database is set up:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*…*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb); ④
```

(Note that errors are captured, but not handled. We hope your code is better than ours.)

① Initialize an empty object database (ODB) "frontend," which will act as a container for the "backends" which are the ones doing the real work.

② Initialize a custom ODB backend.

③ Add the backend to the frontend.

④ Open a repository, and set it to use our ODB to look up objects.

But what is this `git_odb_backend_mine` thing? Well, that's the constructor for your own ODB implementation, and you can do whatever you want in there, so long as you fill in the `git_odb_backend` structure properly. Here's what it *could* look like:

```
typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*…*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = …;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // …

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}
```

The subtlest constraint here is that `my_backend_struct`'s first member must be a `git_odb_backend` structure; this ensures that the memory layout is what the Libgit2 code expects it to be. The rest of it is arbitrary; this structure can be as large or small as you need it to be.

The initialization function allocates some memory for the structure, sets up the custom context, and then fills in the members of the `parent` structure that it supports. Take a look at the `include/git2/sys/odb_backend.h` file in the Libgit2 source for a complete set of call signatures; your particular use case will help determine which of these you'll want to support.

Other Bindings

Libgit2 has bindings for many languages. Here we show a small example using a few of the more complete bindings packages as of this writing; libraries exist for many other languages, including C++, Go, Node.js, Erlang, and the JVM, all in various stages of maturity. The official collection of bindings can be found by browsing the repositories at https://github.com/libgit2. The code we'll write will return the commit message from the commit eventually pointed to by HEAD (sort of like `git log -1`).

LibGit2Sharp

If you're writing a .NET or Mono application, LibGit2Sharp (https://github.com/libgit2/libgit2sharp) is what you're looking for. The bindings are written in C#, and great care has been taken to wrap the raw Libgit2 calls with native-feeling CLR APIs. Here's what our example program looks like:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

For desktop Windows applications, there's even a NuGet package that will help you get started quickly.

objective-git

If your application is running on an Apple platform, you're likely using Objective-C as your implementation language. Objective-Git (https://github.com/libgit2/objective-git) is the name of the Libgit2 bindings for that environment. The example program looks like this:

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]
error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git is fully interoperable with Swift, so don't fear if you've left Objective-C behind.

pygit2

The bindings for Libgit2 in Python are called Pygit2, and can be found at http://www.pygit2.org/. Our example program:

```
pygit2.Repository("/path/to/repo") # open repository
    .head                          # get the current branch
    .peel(pygit2.Commit)           # walk down to the commit
    .message                       # read the message
```

Further Reading

Of course, a full treatment of Libgit2's capabilities is outside the scope of this book. If you want more information on Libgit2 itself, there's API documentation at https://libgit2.github.com/libgit2, and a set of guides at https://libgit2.github.com/docs. For the other bindings, check the bundled README and tests; there are often small tutorials and pointers to further reading there.

JGit

If you want to use Git from within a Java program, there is a fully featured Git library called JGit. JGit is a relatively full-featured implementation of Git written natively in Java, and is widely used in the Java community. The JGit project is under the Eclipse umbrella, and its home can be found at http://www.eclipse.org/jgit.

Getting Set Up

There are a number of ways to connect your project with JGit and start writing code against it. Probably the easiest is to use Maven – the integration is accomplished by adding the following

snippet to the `<dependencies>` tag in your pom.xml file:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

The `version` will most likely have advanced by the time you read this; check http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit for updated repository information. Once this step is done, Maven will automatically acquire and use the JGit libraries that you'll need.

If you would rather manage the binary dependencies yourself, pre-built JGit binaries are available from http://www.eclipse.org/jgit/download. You can build them into your project by running a command like this:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Plumbing

JGit has two basic levels of API: plumbing and porcelain. The terminology for these comes from Git itself, and JGit is divided into roughly the same kinds of areas: porcelain APIs are a friendly front-end for common user-level actions (the sorts of things a normal user would use the Git command-line tool for), while the plumbing APIs are for interacting with low-level repository objects directly.

The starting point for most JGit sessions is the `Repository` class, and the first thing you'll want to do is create an instance of it. For a filesystem-based repository (yes, JGit allows for other storage models), this is accomplished using `FileRepositoryBuilder`:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

The builder has a fluent API for providing all the things it needs to find a Git repository, whether or not your program knows exactly where it's located. It can use environment variables (`.readEnvironment()`), start from a place in the working directory and search (`.setWorkTree(…).findGitDir()`), or just open a known `.git` directory as above.

Once you have a `Repository` instance, you can do all sorts of things with it. Here's a quick sampling:

```
// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

There's quite a bit going on here, so let's go through it one section at a time.

The first line gets a pointer to the `master` reference. JGit automatically grabs the *actual* master ref, which lives at `refs/heads/master`, and returns an object that lets you fetch information about the reference. You can get the name (`.getName()`), and either the target object of a direct reference (`.getObjectId()`) or the reference pointed to by a symbolic ref (`.getTarget()`). Ref objects are also used to represent tag refs and objects, so you can ask if the tag is "peeled," meaning that it points to the final target of a (potentially long) string of tag objects.

The second line gets the target of the `master` reference, which is returned as an ObjectId instance. ObjectId represents the SHA-1 hash of an object, which might or might not exist in Git's object database. The third line is similar, but shows how JGit handles the rev-parse syntax (for more on this, see Branch References); you can pass any object specifier that Git understands, and JGit will return either a valid ObjectId for that object, or `null`.

The next two lines show how to load the raw contents of an object. In this example, we call `ObjectLoader.copyTo()` to stream the contents of the object directly to stdout, but ObjectLoader also has methods to read the type and size of an object, as well as return it as a byte array. For large objects (where `.isLarge()` returns `true`), you can call `.openStream()` to get an InputStream-like object that can read the raw object data without pulling it all into memory at once.

The next few lines show what it takes to create a new branch. We create a RefUpdate instance,

configure some parameters, and call `.update()` to trigger the change. Directly following this is the code to delete that same branch. Note that `.setForceUpdate(true)` is required for this to work; otherwise the `.delete()` call will return `REJECTED`, and nothing will happen.

The last example shows how to fetch the `user.name` value from the Git configuration files. This Config instance uses the repository we opened earlier for local configuration, but will automatically detect the global and system configuration files and read values from them as well.

This is only a small sampling of the full plumbing API; there are many more methods and classes available. Also not shown here is the way JGit handles errors, which is through the use of exceptions. JGit APIs sometimes throw standard Java exceptions (such as `IOException`), but there are a host of JGit-specific exception types that are provided as well (such as `NoRemoteRepositoryException`, `CorruptObjectException`, and `NoMergeBaseException`).

Porcelain

The plumbing APIs are rather complete, but it can be cumbersome to string them together to achieve common goals, like adding a file to the index, or making a new commit. JGit provides a higher-level set of APIs to help out with this, and the entry point to these APIs is the `Git` class:

```
Repository repo;
// construct repo...
Git git = new Git(repo);
```

The Git class has a nice set of high-level *builder*-style methods that can be used to construct some pretty complex behavior. Let's take a look at an example – doing something like `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",
"p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}
```

This is a common pattern with the Git class; the methods return a command object that lets you chain method calls to set parameters, which are executed when you call `.call()`. In this case, we're asking the `origin` remote for tags, but not heads. Also notice the use of a `CredentialsProvider` object for authentication.

Many other commands are available through the Git class, including but not limited to `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, and `reset`.

Further Reading

This is only a small sampling of JGit's full capabilities. If you're interested and want to learn more, here's where to look for information and inspiration:

- The official JGit API documentation can be found at https://www.eclipse.org/jgit/documentation. These are standard Javadoc, so your favorite JVM IDE will be able to install them locally, as well.
- The JGit Cookbook at https://github.com/centic9/jgit-cookbook has many examples of how to do specific tasks with JGit.

Apêndice C: Git Commands

Throughout the book we have introduced dozens of Git commands and have tried hard to introduce them within something of a narrative, adding more commands to the story slowly. However, this leaves us with examples of usage of the commands somewhat scattered throughout the whole book.

In this appendix, we'll go through all the Git commands we addressed throughout the book, grouped roughly by what they're used for. We'll talk about what each command very generally does and then point out where in the book you can find us having used it.

Setup and Config

There are two commands that are used quite a lot, from the first invocations of Git to common every day tweaking and referencing, the `config` and `help` commands.

git config

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories.

The `git config` command has been used in nearly every chapter of the book.

In Configuração Inicial do Git we used it to specify our name, email address and editor preference before we even got started using Git.

In Apelidos Git we showed how you could use it to create shorthand commands that expand to long option sequences so you don't have to type them every time.

In Rebase we used it to make `--rebase` the default when you run `git pull`.

In Credential Storage we used it to set up a default store for your HTTP passwords.

In Keyword Expansion we showed how to set up smudge and clean filters on content coming in and out of Git.

Finally, basically the entirety of Git Configuration is dedicated to the command.

git help

The `git help` command is used to show you all the documentation shipped with Git about any command. While we're giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run `git help <command>`.

We introduced the `git help` command in Pedindo Ajuda and showed you how to use it to find more

information about the `git shell` in Setting Up the Server.

Getting and Creating Projects

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

git init

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run `git init`.

We first introduce this in Obtendo um Repositório Git, where we show creating a brand new repository to start working with.

We talk briefly about how you can change the default branch from "master" in Branches remotos.

We use this command to create an empty bare repository for a server in Putting the Bare Repository on a Server.

Finally, we go through some of the details of what it actually does behind the scenes in Encanamento e Porcelana.

git clone

The `git clone` command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs `git init` to make it an empty Git repository, adds a remote (`git remote add`) to the URL that you pass it (by default named `origin`), runs a `git fetch` from that remote repository and then checks out the latest commit into your working directory with `git checkout`.

The `git clone` command is used in dozens of places throughout the book, but we'll just list a few interesting places.

It's basically introduced and explained in Clonando um Repositório Existente, where we go through a few examples.

In Getting Git on a Server we look at using the `--bare` option to create a copy of a Git repository with no working directory.

In Bundling we use it to unbundle a bundled Git repository.

Finally, in Cloning a Project with Submodules we learn the `--recursive` option to make cloning a repository with submodules a little simpler.

Though it's used in many other places through the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

Basic Snapshotting

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

git add

The `git add` command adds content from the working directory into the staging area (or "index") for the next commit. When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We'll quickly cover some of the unique uses that can be found.

We first introduce and explain `git add` in detail in Rastreando Arquivos Novos.

We mention how to use it to resolve merge conflicts in Conflitos Básicos de Merge.

We go over using it to interactively stage only specific parts of a modified file in Interactive Staging.

Finally, we emulate it at a low level in Objetos Tree, so you can get an idea of what it's doing behind the scenes.

git status

The `git status` command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In its normal form, it also will show you some basic hints on how to move files between these stages.

We first cover `status` in Verificando os Status de Seus Arquivos, both in its basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the `git status` command is covered there.

git diff

The `git diff` command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (`git diff` by itself), between your staging area and your last commit (`git diff --staged`), or between two commits (`git diff master branchB`).

We first look at the basic uses of `git diff` in Visualizando Suas Alterações Dentro e Fora do Stage, where we show how to see what changes are staged and which are not yet staged.

We use it to look for possible whitespace issues before committing with the `--check` option in Diretrizes para Fazer Commites.

We see how to check the differences between branches more effectively with the `git diff A...B` syntax in Determining What Is Introduced.

We use it to filter out whitespace differences with `-b` and how to compare different stages of

conflicted files with `--theirs`, `--ours` and `--base` in Advanced Merging.

Finally, we use it to effectively compare submodule changes with `--submodule` in Starting with Submodules.

git difftool

The `git difftool` command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in `git diff` command.

We only briefly mention this in Visualizando Suas Alterações Dentro e Fora do Stage.

git commit

The `git commit` command takes all the file contents that have been staged with `git add` and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in Fazendo Commit das Suas Alterações. There we also demonstrate how to use the `-a` flag to skip the `git add` step in daily workflows and how to use the `-m` flag to pass a commit message in on the command line instead of firing up an editor.

In Desfazendo coisas we cover using the `--amend` option to redo the most recent commit.

In Branches em poucas palavras, we go into much more detail about what `git commit` does and why it does it like that.

We looked at how to sign commits cryptographically with the `-S` flag in Signing Commits.

Finally, we take a look at what the `git commit` command does in the background and how it's actually implemented in Objetos Commit.

git reset

The `git reset` command is primarily used to undo things, as you can possibly tell by the verb. It moves around the `HEAD` pointer and optionally changes the `index` or staging area and can also optionally change the working directory if you use `--hard`. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of `git reset` in Retirando um arquivo do Stage, where we use it to unstage a file we had run `git add` on.

We then cover it in quite some detail in Reset Demystified, which is entirely devoted to explaining this command.

We use `git reset --hard` to abort a merge in Aborting a Merge, where we also use `git merge --abort`, which is a bit of a wrapper for the `git reset` command.

git rm

The `git rm` command is used to remove files from the staging area and working directory for Git. It is similar to `git add` in that it stages a removal of a file for the next commit.

We cover the `git rm` command in some detail in Removendo Arquivos, including recursively removing files and only removing files from the staging area but leaving them in the working directory with `--cached`.

The only other differing use of `git rm` in the book is in Removing Objects where we briefly use and explain the `--ignore-unmatch` when running `git filter-branch`, which simply makes it not error out when the file we are trying to remove doesn't exist. This can be useful for scripting purposes.

git mv

The `git mv` command is a thin convenience command to move a file and then run `git add` on the new file and `git rm` on the old file.

We only briefly mention this command in Movendo Arquivos.

git clean

The `git clean` command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files.

We cover many of the options and scenarios in which you might used the clean command in Cleaning your Working Directory.

Branching and Merging

There are just a handful of commands that implement most of the branching and merging functionality in Git.

git branch

The `git branch` command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

Most of Branches no Git is dedicated to the `branch` command and it's used throughout the entire chapter. We first introduce it in Criando um Novo Branch and we go through most of its other features (listing and deleting) in Gestão de Branches.

In Rastreando Branches we use the `git branch -u` option to set up a tracking branch.

Finally, we go through some of what it does in the background in Referências do Git.

git checkout

The `git checkout` command is used to switch branches and check content out into your working directory.

We first encounter the command in Alternando entre Branches along with the `git branch` command.

We see how to use it to start tracking branches with the `--track` flag in Rastreando Branches.

We use it to reintroduce file conflicts with `--conflict=diff3` in Checking Out Conflicts.

We go into closer detail on its relationship with `git reset` in Reset Demystified.

Finally, we go into some implementation detail in A HEAD.

git merge

The `git merge` tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

The `git merge` command was first introduced in Ramificação Básica. Though it is used in various places in the book, there are very few variations of the `merge` command — generally just `git merge <branch>` with the name of the single branch you want to merge in.

We covered how to do a squashed merge (where Git merges the work but pretends like it's just a new commit without recording the history of the branch you're merging in) at the very end of Fork de Projeto Público.

We went over a lot about the merge process and command, including the `-Xignore-space-change` command and the `--abort` flag to abort a problem merge in Advanced Merging.

We learned how to verify signatures before merging if your project is using GPG signing in Signing Commits.

Finally, we learned about Subtree merging in Subtree Merging.

git mergetool

The `git mergetool` command simply launches an external merge helper in case you have issues with a merge in Git.

We mention it quickly in Conflitos Básicos de Merge and go into detail on how to implement your own external merge tool in External Merge and Diff Tools.

git log

The `git log` command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you're currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

This command is used in nearly every chapter of the book to demonstrate the history of a project.

We introduce the command and cover it in some depth in Vendo o histórico de Commits. There we look at the `-p` and `--stat` option to get an idea of what was introduced in each commit and the

`--pretty` and `--oneline` options to view the history more concisely, along with some simple date and author filtering options.

In Criando um Novo Branch we use it with the `--decorate` option to easily visualize where our branch pointers are located and we also use the `--graph` option to see what divergent histories look like.

In Time Pequeno Privado and Commit Ranges we cover the `branchA..branchB` syntax to use the `git log` command to see what commits are unique to a branch relative to another branch. In Commit Ranges we go through this fairly extensively.

In Merge Log and Triple Dot we cover using the `branchA...branchB` format and the `--left-right` syntax to see what is in one branch or the other but not in both. In Merge Log we also look at how to use the `--merge` option to help with merge conflict debugging as well as using the `--cc` option to look at merge commit conflicts in your history.

In RefLog Shortnames we use the `-g` option to view the Git reflog through this tool instead of doing branch traversal.

In Searching we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In Signing Commits we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch.

This is basically entirely covered in Stashing and Cleaning.

git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

This command is introduced and covered in detail in Criando Tags and we use it in practice in Tagging Your Releases.

We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in Signing Your Work.

Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

We first look at this command in Buscando e Obtendo de seus Repositórios Remotos and we continue to see examples of it use in Branches remotos.

We also use it in several of the examples in Contribuindo com um Projeto.

We use it to fetch a single specific reference that is outside of the default space in Pull Request Refs and we see how to fetch from a bundle in Bundling.

We set up highly custom refspecs in order to make `git fetch` do something a little different than the default in The Refspec.

git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you're on.

We introduce it quickly in Buscando e Obtendo de seus Repositórios Remotos and show how to see what it will merge if you run it in Inspecionando o Servidor Remoto.

We also see how to use it to help with rebasing difficulties in Rebase quando vocês faz Rebase.

We show how to use it with a URL to pull in changes in a one-off fashion in Checking Out Remote Branches.

Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in Signing Commits.

git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

We first look at the `git push` command in Pushing to Your Remotes. Here we cover the basics of pushing a branch to a remote repository. In Empurrando (Push) we go a little deeper into pushing specific branches and in Rastreando Branches we see how to set up tracking branches to automatically push to. In Removendo Branches remotos we use the `--delete` flag to delete a branch on the server with `git push`.

Throughout Contribuindo com um Projeto we see several examples of using `git push` to share work on branches through multiple remotes.

We see how to use it to share tags that you have made with the `--tags` option in Compartilhando Tags.

In Publishing Submodule Changes we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

In Other Client Hooks we talk briefly about the `pre-push` hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.

Finally, in Pushing Refspecs we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as "origin" so you don't have to type them out all the time. You can have several of these and the `git remote` command is used to add, change and delete them.

This command is covered in detail in Trabalhando de Forma Remota, including listing, adding, removing and renaming them.

It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote add <name> <url>` format.

git archive

The `git archive` command is used to create an archive file of a specific snapshot of the project.

We use `git archive` to create a tarball of a project for sharing in Preparing a Release.

git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The `submodule` command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources.

This command is only mentioned and entirely covered in Submodules.

Inspection and Comparison

git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

We first use it to show annotated tag information in Tags Anotadas.

Later we use it quite a bit in Revision Selection to show the commits that our various revision selections resolve to.

One of the more interesting things we do with `git show` is in Manual File Re-merging to extract

specific file contents of various stages during a merge conflict.

git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

We showed how to use it to create a nice changelog in The Shortlog.

git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It's a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.

We use `git describe` in Generating a Build Number and Preparing a Release to get a string to name our release file after.

Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from figuring out where something was introduced to figuring out who introduced it.

git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

It is fully covered in Binary Search and is only mentioned in that section.

git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

It is covered in File Annotation and is only mentioned in that section.

git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

It is covered in Git Grep and is only mentioned in that section.

Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the

changes they introduce, as though the commit series is a series of patches. These commands help you manage your branches in this manner.

git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you're currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

Cherry picking is described and demonstrated in Rebasing and Cherry-Picking Workflows.

git rebase

The `git rebase` command is basically an automated `cherry-pick`. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

Rebasing is covered in detail in Rebase, including covering the collaborative issues involved with rebasing branches that are already public.

We use it in practice during an example of splitting your history into two separate repositories in Replace, using the `--onto` flag as well.

We go through running into a merge conflict during rebasing in Rerere.

We also use it in an interactive scripting mode with the `-i` option in Changing Multiple Commit Messages.

git revert

The `git revert` command is essentially a reverse `git cherry-pick`. It creates a new commit that applies the exact opposite of the change introduced in the commit you're targeting, essentially undoing or reverting it.

We use this in Reverse the commit to undo a merge commit.

Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

git apply

The `git apply` command applies a patch created with the `git diff` or even GNU diff command. It is similar to what the `patch` command might do with a few small differences.

We demonstrate using it and the circumstances in which you might do so in Applying Patches from Email.

git am

The `git am` command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

We covered usage and workflow around `git am` in Applying a Patch with `am` including using the `--resolved`, `-i` and `-3` options.

There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in Email Workflow Hooks.

We also use it to apply patch formatted GitHub Pull Request changes in Email Notifications.

git format-patch

The `git format-patch` command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted.

We go through an example of contributing to a project using the `git format-patch` tool in Projeto Público através de Email.

git imap-send

The `git imap-send` command uploads a mailbox generated with `git format-patch` into an IMAP drafts folder.

We go through an example of contributing to a project by sending patches with the `git imap-send` tool in Projeto Público através de Email.

git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email.

We go through an example of contributing to a project by sending patches with the `git send-email` tool in Projeto Público através de Email.

git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in Fork de Projeto Público.

External Systems

Git comes with a few commands to integrate with other version control systems.

git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

This command is covered in depth in Git and Subversion.

git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record.

This command is covered in depth in A Custom Importer.

Administration

If you're administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

git gc

The `git gc` command runs "garbage collection" on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in Maintenance.

git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies.

We only quickly use this once in Data Recovery to search for dangling objects.

git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in RefLog Shortnames, where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.

We also go through a practical example of recovering such a lost branch in Data Recovery.

git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

In Removing a File from Every Commit we explain the command and explore several different

options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.

In Git-p4 we use it to fix up imported external repositories.

Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

The first one we encounter is `ls-remote` in Pull Request Refs which we use to look at the raw references on the server.

We use `ls-files` in Manual File Re-merging, Rerere and The Index to take a more raw look at what your staging area looks like.

We also mention `rev-parse` in Branch References to take just about any string and turn it into an object SHA-1.

However, most of the low level plumbing commands we cover are in Funcionamento Interno do Git, which is more or less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.

Index

Notas de Tradução

Depois de copiar este repositório para traduzir o trabalho, este é o arquivo onde devem ficar as as notas para a coordenação do trabalho de tradução. Coisas como padronização de palavras e expressões, de forma a deixar o trabalho mais consistente, ou notas de como o processo de colaboração deve ser tratado.

Como um mantenedor de tradução, também sinta-se livre para modificar ou reescrever completamente o arquivo LEIAME (README) para que contenha instruções específicas para sua tradução.

Situação da Tradução

A medida que o trabalho for sendo traduzido, por favor atualize o arquivo `status.json` para indicar aproximadamente a percentagem de completude de cada arquivo. Isso será mostrado em várias páginas para deixar as pessoas saberem quanto do trabalho já foi feito.