

Pro-Git

Tudo que você precisa saber sobre
a ferramenta Git de controle de
distribuição de código.

Scott Chacon

Pro-Git

Tudo que você precisa saber sobre a ferramenta Git de controle de distribuição de código.

Eric Douglas

Esse livro está à venda em <http://leanpub.com/pro-git>

Essa versão foi publicada em 2014-05-11



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)

Tweet Sobre Esse Livro!

Por favor ajude Eric Douglas a divulgar esse livro no [Twitter!](#)

A hashtag sugerida para esse livro é [#pro-git](#).

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

<https://twitter.com/search?q=#pro-git>

*Todo o livro Pro Git, escrito por **Scott Chacon** e publicado pela Apress, está disponível aqui. Todo conteúdo é licenciado sobre a [Creative Commons Attribution Non Commercial Share Alike 3.0 license](#). Versões impressas do livro estão disponíveis na [Amazon.com](#).*

Você pode conferir a versão online deste livro [aqui](#).

Conteúdo

Capítulo 1. Primeiros passos	2
1.1 Primeiros passos - Sobre Controle de Versão	2
1.2 Primeiros passos - Uma Breve História do Git	5
1.3 Primeiros passos - Noções Básicas de Git	6
1.4 Primeiros passos - Instalando Git	10
1.5 Primeiros passos - Configuração Inicial do Git	12
1.6 Primeiros passos - Obtendo Ajuda	14
1.7 Primeiros passos - Resumo	15
Capítulo 2. Git Essencial	16
2.1 Git Essencial - Obtendo um Repositório Git	16
2.2 Git Essencial - Gravando Alterações no Repositório	18
2.3 Git Essencial - Visualizando o Histórico de Commits	30
2.4 Git Essencial - Desfazendo Coisas	36
2.5 Git Essencial - Trabalhando com Remotos	39
2.6 Git Essencial - Tagging	44
2.7 Git Essencial - Dicas e Truques	50
2.8 Git Essencial - Sumário	52
Capítulo 3. Ramificação (Branching) no Git	53
3.1 Ramificação (Branching) no Git - O que é um Branch	53
3.2 Ramificação (Branching) no Git - Básico de Branch e Merge	59
3.3 Ramificação (Branching) no Git - Gerenciamento de Branches	67
3.4 Ramificação (Branching) no Git - Fluxos de Trabalho com Branches	69
3.5 Ramificação (Branching) no Git - Branches Remotos	72
3.6 Ramificação (Branching) no Git - Rebasing	79
3.7 Ramificação (Branching) no Git - Sumário	87
Capítulo 4 - Git no Servidor	88
4.1 Git no Servidor - Os Protocolos	88
4.2 Git no Servidor - Configurando Git no Servidor	93
4.3 Git no Servidor - Gerando Sua Chave Pública SSH	95
4.4 Git no Servidor - Configurando o Servidor	97
4.5 Git no Servidor - Acesso Público	99

CONTEÚDO

4.6 Git no Servidor - GitWeb	101
4.7 Git no Servidor - Gitis	102
4.8 Git no Servidor - Gitolite	108
4.9 Git no Servidor - Serviço Git	114
4.10 Git no Servidor - Git Hospedado	117
4.11 Git no Servidor - Sumário	124
Capítulo 5 - Git Distribuído	126
5.1 Git Distribuído - Fluxos de Trabalho Distribuídos	126
5.2 Git Distribuído - Contribuindo Para Um Projeto	129
5.3 Git Distribuído - Mantendo Um Projeto	153
5.4 Git Distribuído - Resumo	168
Capítulo 6 - Ferramentas do Git	169
6.1 Ferramentas do Git - Seleção de Revisão	169
6.2 Ferramentas do Git - Área de Seleção Interativa	177
6.3 Ferramentas do Git - Fazendo Stash	182
6.4 Ferramentas do Git - Reescrevendo o Histórico	186
6.5 Ferramentas do Git - Depurando com Git	193
6.6 Ferramentas do Git - Submódulos	196
6.7 Ferramentas do Git - Merge de Sub-árvore (Subtree Merging)	203
6.8 Ferramentas do Git - Sumário	206
Capítulo 7 - Customizando o Git	207
7.1 Customizando o Git - Configuração do Git	207
7.2 Customizando o Git - Atributos Git	217
7.3 Customizando o Git - Hooks do Git	226
7.4 Customizando o Git - Um exemplo de Política Git Forçada	229
7.5 Customizando o Git - Sumário	240
Capítulo 8 - Git e Outros Sistemas	241
8.1 Git e Outros Sistemas - Git e Subversion	241
8.2 Migrando para o Git	253
8.3 Git e Outros Sistemas - Resumo	264
Capítulo 9 - Git Internamente	265
9.1 Git Internamente - Encanamento (Plumbing) e Porcelana (Porcelain)	265
9.2 Git Internamente - Objetos do Git	266
9.3 Git Internamente - Referencias Git	276
9.4 Git Internamente - Packfiles	280
9.5 Git Internamente - O Refspec	283
9.6 Git Internamente - Protocolos de Transferência	286
9.7 Git Internamente - Manutenção e Recuperação de Dados	291
9.8 Git Internamente - Resumo	299

Todo o livro Pro Git, escrito por **Scott Chacon** e publicado pela Apress, está disponível aqui. Todo conteúdo é licenciado sobre a [Creative Commons Attribution Non Commercial Share Alike 3.0 license](https://creativecommons.org/licenses/by-nc-sa/3.0/)¹. Versões impressas do livro estão disponíveis na [Amazon.com](https://www.amazon.com/gp/product/1430218339?ie=UTF8&camp=1789&creative=9325&creativeASIN=1430218339&linkCode=as2&tag=git-sfconservancy-20)².

Você pode conferir a versão online deste livro [aqui](https://git-scm.com/book/pt-br/)³.

¹[http://creativecommons.org/licenses/by-nc-sa/3.0/](https://creativecommons.org/licenses/by-nc-sa/3.0/)

²[http://www.amazon.com/gp/product/1430218339?ie=UTF8&camp=1789&creative=9325&creativeASIN=1430218339&linkCode=as2&tag=git-sfconservancy-20](https://www.amazon.com/gp/product/1430218339?ie=UTF8&camp=1789&creative=9325&creativeASIN=1430218339&linkCode=as2&tag=git-sfconservancy-20)

³[http://git-scm.com/book/pt-br/](https://git-scm.com/book/pt-br/)

Capítulo 1. Primeiros passos

Esse capítulo trata dos primeiros passos usando o Git. Inicialmente explicaremos alguns fundamentos sobre ferramentas de controle de versão, passaremos ao tópico de como instalar o Git no seu sistema e finalmente como configurá-lo para começar a trabalhar. Ao final do capítulo você entenderá porque o Git é muito utilizado, porque usá-lo e como usá-lo.

1.1 Primeiros passos - Sobre Controle de Versão

Sobre Controle de Versão

O que é controle de versão, e por que você deve se importar? O controle de versão é um sistema que registra as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que você possa recuperar versões específicas. Mesmo que os exemplos desse livro mostrem arquivos de código fonte sob controle de versão, você pode usá-lo com praticamente qualquer tipo de arquivo em um computador.

Se você é um designer gráfico ou um web designer e quer manter todas as versões de uma imagem ou layout (o que você certamente gostaria), usar um Sistema de Controle de Versão (Version Control System ou VCS) é uma decisão sábia. Ele permite reverter arquivos para um estado anterior, reverter um projeto inteiro para um estado anterior, comparar mudanças feitas ao decorrer do tempo, ver quem foi o último a modificar algo que pode estar causando problemas, quem introduziu um bug e quando, e muito mais. Usar um VCS normalmente significa que se você estragou algo ou perdeu arquivos, poderá facilmente reavê-los. Além disso, você pode controlar tudo sem maiores esforços.

Sistemas de Controle de Versão Locais

O método preferido de controle de versão por muitas pessoas é copiar arquivos em outro diretório (talvez um diretório com data e hora, se forem espertos). Esta abordagem é muito comum por ser tão simples, mas é também muito suscetível a erros. É fácil esquecer em qual diretório você está e gravar acidentalmente no arquivo errado ou sobrescrever arquivos sem querer.

Para lidar com esse problema, alguns programadores desenvolveram há muito tempo VCSs locais que armazenavam todas as alterações dos arquivos sob controle de revisão (ver Figura 1-1).

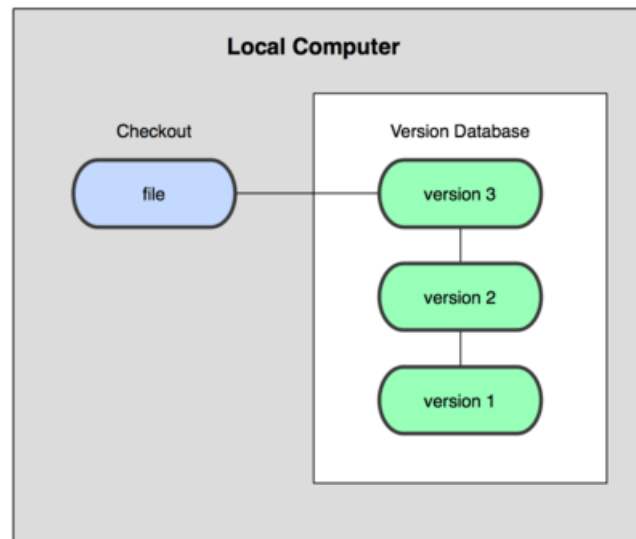


Figura 1-1. Diagrama de controle de versão local.

Uma das ferramentas de VCS mais populares foi um sistema chamado rcs, que ainda é distribuído em muitos computadores até hoje. Até o popular Mac OS X inclui o comando rcs quando se instala o kit de ferramentas para desenvolvedores. Basicamente, essa ferramenta mantém conjuntos de patches (ou seja, as diferenças entre os arquivos) entre cada mudança em um formato especial; a partir daí qualquer arquivo em qualquer ponto na linha do tempo pode ser recriado ao juntar-se todos os patches.

Sistemas de Controle de Versão Centralizados

Outro grande problema que as pessoas encontram estava na necessidade de trabalhar em conjunto com outros desenvolvedores, que usam outros sistemas. Para lidar com isso, foram desenvolvidos Sistemas de Controle de Versão Centralizados (Centralized Version Control System ou CVCS). Esses sistemas, como por exemplo o CVS, Subversion e Perforce, possuem um único servidor central que contém todos os arquivos versionados e vários clientes que podem resgatar (check out) os arquivos do servidor. Por muitos anos, esse foi o modelo padrão para controle de versão.

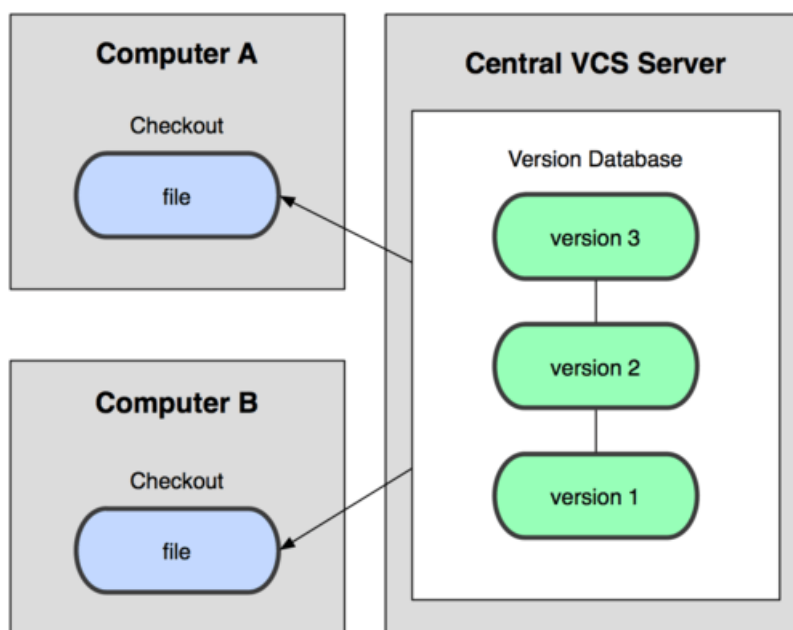


Figura 1-2. Diagrama de Controle de Versão Centralizado.

Tal arranjo oferece muitas vantagens, especialmente sobre VCSs locais. Por exemplo, todo mundo pode ter conhecimento razoável sobre o que os outros desenvolvedores estão fazendo no projeto. Administradores têm controle específico sobre quem faz o quê; sem falar que é bem mais fácil administrar um CVCS do que lidar com bancos de dados locais em cada cliente.

Entretanto, esse arranjo também possui grandes desvantagens. O mais óbvio é que o servidor central é um ponto único de falha. Se o servidor ficar fora do ar por uma hora, ninguém pode trabalhar em conjunto ou salvar novas versões dos arquivos durante esse período. Se o disco do servidor do banco de dados for corrompido e não existir um backup adequado, perde-se tudo — todo o histórico de mudanças no projeto, exceto pelas únicas cópias que os desenvolvedores possuem em suas máquinas locais. VCSs locais também sofrem desse problema — sempre que se tem o histórico em um único local, corre-se o risco de perder tudo.

Sistemas de Controle de Versão Distribuídos

É aí que surgem os Sistemas de Controle de Versão Distribuídos (Distributed Version Control System ou DVCS). Em um DVCS (tais como Git, Mercurial, Bazaar or Darcs), os clientes não apenas fazem cópias das últimas versões dos arquivos: eles são cópias completas do repositório. Assim, se um servidor falha, qualquer um dos repositórios dos clientes pode ser copiado de volta para o servidor para restaurá-lo. Cada checkout (resgate) é na prática um backup completo de todos os dados (veja Figura 1-3).

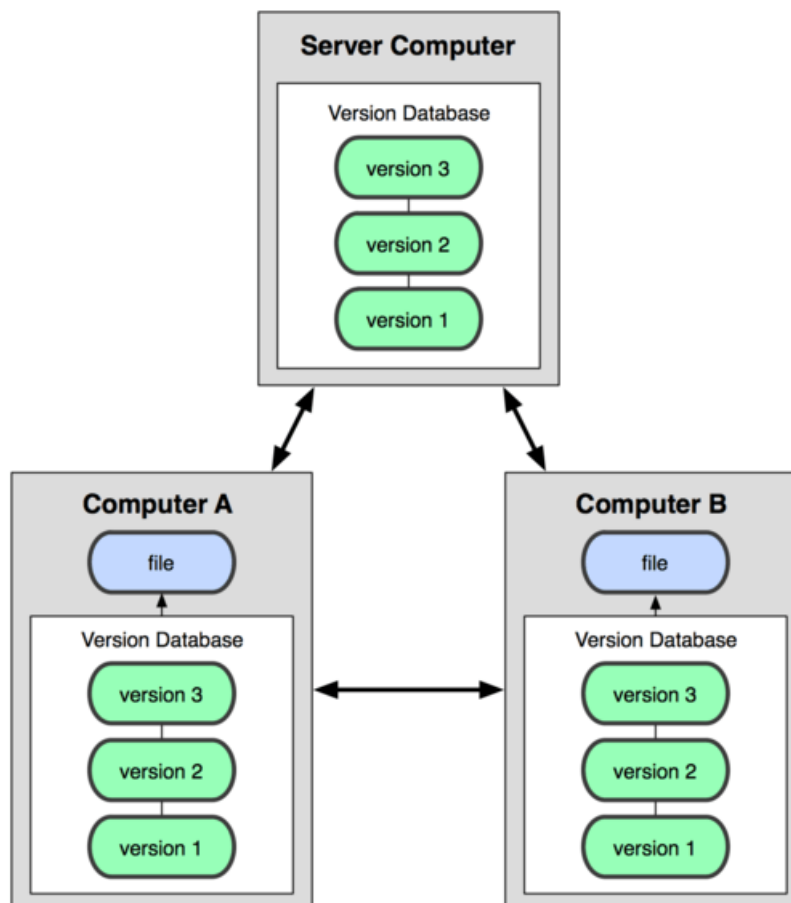


Figura 1-3. Diagrama de Controle de Versão Distribuído.

Além disso, muitos desses sistemas lidam muito bem com o aspecto de ter vários repositórios remotos com os quais eles podem colaborar, permitindo que você trabalhe em conjunto com diferentes grupos de pessoas, de diversas maneiras, simultaneamente no mesmo projeto. Isso permite que você estabeleça diferentes tipos de workflow (fluxo de trabalho) que não são possíveis em sistemas centralizados, como por exemplo o uso de modelos hierárquicos.

1.2 Primeiros passos - Uma Breve História do Git

Uma Breve História do Git

Assim como muitas coisas boas na vida, o Git começou com um tanto de destruição criativa e controvérsia acirrada. O kernel (núcleo) do Linux é um projeto de software de código aberto de escopo razoavelmente grande. Durante a maior parte do período de manutenção do kernel do Linux (1991-2002), as mudanças no software eram repassadas como patches e arquivos compactados. Em 2002, o projeto do kernel do Linux começou a usar um sistema DVCS proprietário chamado BitKeeper.

Em 2005, o relacionamento entre a comunidade que desenvolvia o kernel e a empresa que desenvolvia comercialmente o BitKeeper se desfez, e o status de isento-de-pagamento da ferramenta foi revogado. Isso levou a comunidade de desenvolvedores do Linux (em particular Linus Torvalds, o criador do Linux) a desenvolver sua própria ferramenta baseada nas lições que eles aprenderam ao usar o BitKeeper. Alguns dos objetivos do novo sistema eram:

- Velocidade
- Design simples
- Suporte robusto a desenvolvimento não linear (milhares de branches paralelos)
- Totalmente distribuído
- Capaz de lidar eficientemente com grandes projetos como o kernel do Linux (velocidade e volume de dados)

Desde sua concepção em 2005, o Git evoluiu e amadureceu a ponto de ser um sistema fácil de usar e ainda assim mantém essas qualidades iniciais. É incrivelmente rápido, bastante eficiente com grandes projetos e possui um sistema impressionante de branching para desenvolvimento não-linear (Veja no Capítulo 3).

1.3 Primeiros passos - Noções Básicas de Git

Noções Básicas de Git

Enfim, em poucas palavras, o que é Git? Essa é uma seção importante para assimilar, pois se você entender o que é Git e os fundamentos de como ele funciona, será muito mais fácil utilizá-lo de forma efetiva. À medida que você aprende a usar o Git, tente não pensar no que você já sabe sobre outros VCSs como Subversion e Perforce; assim você consegue escapar de pequenas confusões que podem surgir ao usar a ferramenta. Apesar de possuir uma interface parecida, o Git armazena e pensa sobre informação de uma forma totalmente diferente desses outros sistemas; entender essas diferenças lhe ajudará a não ficar confuso ao utilizá-lo.

Snapshots, E Não Diferenças

A maior diferença entre Git e qualquer outro VCS (Subversion e similares inclusos) está na forma que o Git trata os dados. Conceitualmente, a maior parte dos outros sistemas armazena informação como uma lista de mudanças por arquivo. Esses sistemas (CVS, Subversion, Perforce, Bazaar, etc.) tratam a informação que mantém como um conjunto de arquivos e as mudanças feitas a cada arquivo ao longo do tempo, conforme ilustrado na Figura 1.4.

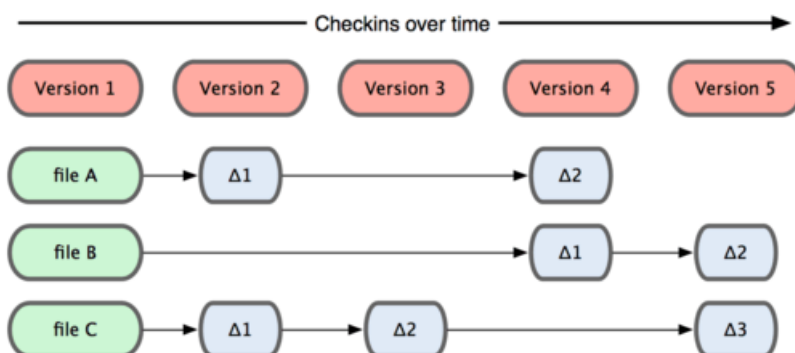


Figura 1-4. Outros sistemas costumam armazenar dados como mudanças em uma versão inicial de cada arquivo.

Git não pensa ou armazena sua informação dessa forma. Ao invés disso, o Git considera que os dados são como um conjunto de snapshots (captura de algo em um determinado instante, como em uma foto) de um mini-sistema de arquivos. Cada vez que você salva ou consolida (commit) o estado do seu projeto no Git, é como se ele tirasse uma foto de todos os seus arquivos naquele momento e armazenasse uma referência para essa captura. Para ser eficiente, se nenhum arquivo foi alterado, a informação não é armazenada novamente - apenas um link para o arquivo idêntico anterior que já foi armazenado. A figura 1-5 mostra melhor como o Git lida com seus dados.

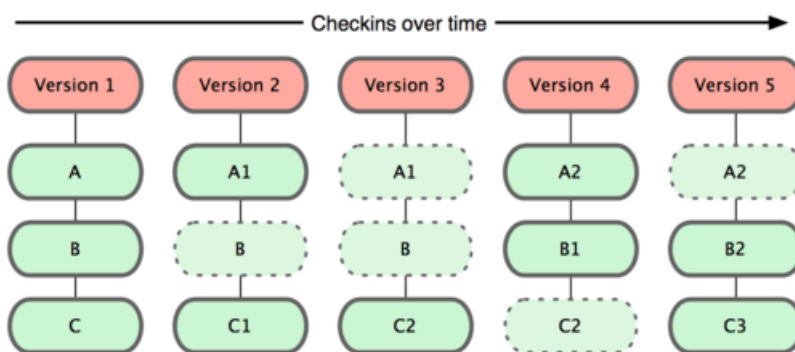


Figura 1-5. Git armazena dados como snapshots do projeto ao longo do tempo.

Essa é uma distinção importante entre Git e quase todos os outros VCSs. Isso leva o Git a reconsiderar quase todos os aspectos de controle de versão que os outros sistemas copiaram da geração anterior. Também faz com que o Git se comporte mais como um mini-sistema de arquivos com algumas poderosas ferramentas construídas em cima dele, ao invés de simplesmente um VCS. Nós vamos explorar alguns dos benefícios que você tem ao lidar com dados dessa forma, quando tratarmos do assunto de branching no Capítulo 3.

Quase Todas Operações São Locais

A maior parte das operações no Git precisam apenas de recursos e arquivos locais para operar — geralmente nenhuma outra informação é necessária de outro computador na sua rede. Se você está acostumado a um CVCS onde a maior parte das operações possui latência por conta de comunicação com a rede, esse aspecto do Git fará com que você pense que os deuses da velocidade abençoaram o

Git com poderes sobrenaturais. Uma vez que você tem todo o histórico do projeto no seu disco local, a maior parte das operações parece ser quase instantânea.

Por exemplo, para navegar no histórico do projeto, o Git não precisa requisitar ao servidor o histórico para que possa apresentar a você — ele simplesmente lê diretamente de seu banco de dados local. Isso significa que você vê o histórico do projeto quase instantaneamente. Se você quiser ver todas as mudanças introduzidas entre a versão atual de um arquivo e a versão de um mês atrás, o Git pode buscar o arquivo de um mês atrás e calcular as diferenças localmente, ao invés de ter que requisitar ao servidor que faça o cálculo, ou puxar uma versão antiga do arquivo no servidor remoto para que o cálculo possa ser feito localmente.

Isso também significa que há poucas coisas que você não possa fazer caso esteja offline ou sem acesso a uma VPN. Se você entrar em um avião ou trem e quiser trabalhar, você pode fazer commits livre de preocupações até ter acesso a rede novamente para fazer upload. Se você estiver indo para casa e seu cliente de VPN não estiver funcionando, você ainda pode trabalhar. Em outros sistemas, fazer isso é impossível ou muito trabalhoso. No Perforce, por exemplo, você não pode fazer muita coisa quando não está conectado ao servidor; e no Subversion e CVS, você pode até editar os arquivos, mas não pode fazer commits das mudanças já que sua base de dados está offline. Pode até parecer que não é grande coisa, mas você pode se surpreender com a grande diferença que pode fazer.

Git Tem Integridade

Tudo no Git tem seu checksum (valor para verificação de integridade) calculado antes que seja armazenado e então passa a ser referenciado pelo checksum. Isso significa que é impossível mudar o conteúdo de qualquer arquivo ou diretório sem que o Git tenha conhecimento. Essa funcionalidade é parte fundamental do Git e é integral à sua filosofia. Você não pode perder informação em trânsito ou ter arquivos corrompidos sem que o Git seja capaz de detectar.

O mecanismo que o Git usa para fazer o checksum é chamado de hash SHA-1, uma string de 40 caracteres composta de caracteres hexadecimais (0-9 e a-f) que é calculado a partir do conteúdo de um arquivo ou estrutura de um diretório no Git. Um hash SHA-1 parece com algo mais ou menos assim:

1 24b9da6552252987aa493b52f8696cd6d3b00373

Você vai encontrar esses hashes em todo canto, uma vez que Git os utiliza tanto. Na verdade, tudo que o Git armazena é identificado não por nome do arquivo mas pelo valor do hash do seu conteúdo.

Git Geralmente Só Adiciona Dados

Dentre as ações que você pode realizar no Git, quase todas apenas acrescentam dados à base do Git. É muito difícil fazer qualquer coisa no sistema que não seja reversível ou remover dados de qualquer forma. Assim como em qualquer VCS, você pode perder ou bagunçar mudanças que ainda

não commitou; mas depois de fazer um commit de um snapshot no Git, é muito difícil que você o perca, especialmente se você frequentemente joga suas mudanças para outro repositório.

Isso faz com que o uso do Git seja uma alegria no sentido de permitir que façamos experiências sem o perigo de causar danos sérios. Para uma análise mais detalhada de como o Git armazena seus dados e de como você pode recuperar dados que parecem ter sido perdidos, veja o Capítulo 9.

Os Três Estados

Agora preste atenção. Essa é a coisa mais importante pra se lembrar sobre Git se você quiser que o resto do seu aprendizado seja tranquilo. Git faz com que seus arquivos sempre estejam em um dos três estados fundamentais: consolidado (committed), modificado (modified) e preparado (staged). Dados são ditos consolidados quando estão seguramente armazenados em sua base de dados local. Modificado trata de um arquivo que sofreu mudanças mas que ainda não foi consolidado na base de dados. Um arquivo é tido como preparado quando você marca um arquivo modificado em sua versão corrente para que ele faça parte do snapshot do próximo commit (consolidação).

Isso nos traz para as três seções principais de um projeto do Git: o diretório do Git (git directory, repository), o diretório de trabalho (working directory), e a área de preparação (staging area).

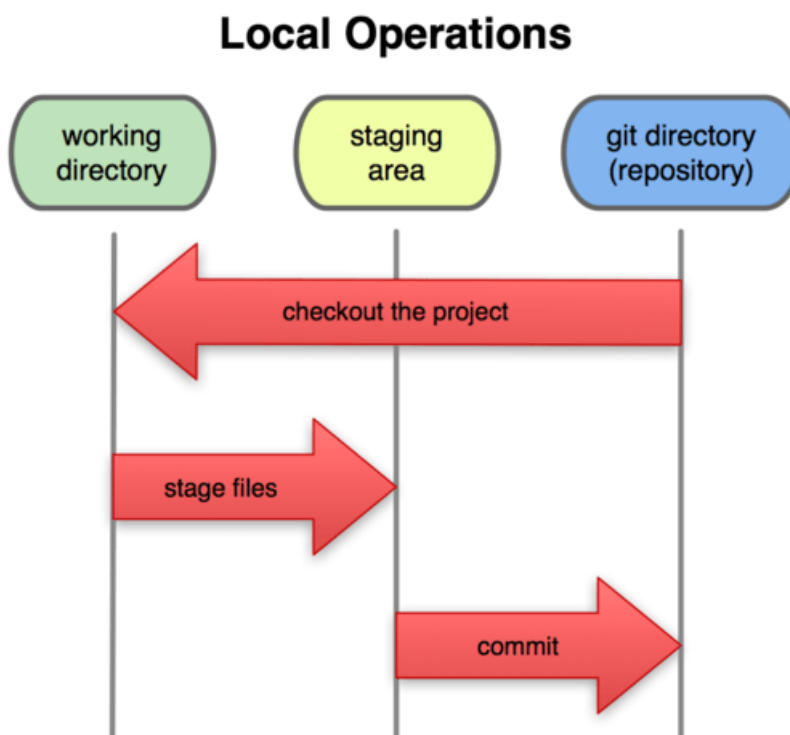


Figura 1-6. Diretório de trabalho, área de preparação, e o diretório do Git.

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

O diretório de trabalho é um único checkout de uma versão do projeto. Estes arquivos são obtidos a partir da base de dados comprimida no diretório do Git e colocados em disco para que você possa utilizar ou modificar.

A área de preparação é um simples arquivo, geralmente contido no seu diretório Git, que armazena informações sobre o que irá em seu próximo commit. É bastante conhecido como índice (index), mas está se tornando padrão chamá-lo de área de preparação.

O workflow básico do Git pode ser descrito assim:

1. Você modifica arquivos no seu diretório de trabalho.
2. Você seleciona os arquivos, adicionando snapshots deles para sua área de preparação.
3. Você faz um commit, que leva os arquivos como eles estão na sua área de preparação e os armazena permanentemente no seu diretório Git.

Se uma versão particular de um arquivo está no diretório Git, é considerada consolidada. Caso seja modificada mas foi adicionada à área de preparação, está preparada. E se foi alterada desde que foi obtida mas não foi preparada, está modificada. No Capítulo 2, você aprenderá mais sobre estes estados e como se aproveitar deles ou pular toda a parte de preparação.

1.4 Primeiros passos - Instalando Git

Instalando Git

Vamos entender como utilizar o Git. Primeiramente você deve instalá-lo. Você pode obtê-lo de diversas formas; as duas mais comuns são instalá-lo a partir do fonte ou instalar um package (pacote) existente para sua plataforma.

Instalando a Partir do Fonte

Caso você possa, é geralmente útil instalar o Git a partir do fonte, porque será obtida a versão mais recente. Cada versão do Git tende a incluir melhoras na UI, sendo assim, obter a última versão é geralmente o melhor caminho caso você sintá-se confortável em compilar o software a partir do fonte. Também acontece que diversas distribuições Linux contêm pacotes muito antigos; sendo assim, a não ser que você tenha uma distro (distribuição) muito atualizada ou está utilizando backports, instalar a partir do fonte pode ser a melhor aposta.

Para instalar o Git, você precisa ter as seguintes bibliotecas que o Git depende: curl, zlib, openssl, expat e libiconv. Por exemplo, se você usa um sistema que tem yum (tal como o Fedora) ou apt-get (tais como os sistemas baseados no Debian), você pode utilizar um desses comandos para instalar todas as dependências:


```
1 $ yum install curl-devel expat-devel gettext-devel \  
2   openssl-devel zlib-devel  
3  
4 $ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \  
5   libz-dev libssl-dev
```

Quando você tiver todas as dependências necessárias, você pode continuar e baixar o snapshot mais recente a partir do web site do Git:

```
1 http://git-scm.com/download
```

Então, compilá-lo e instalá-lo:

```
1 $ tar -zxf git-1.7.2.2.tar.gz  
2 $ cd git-1.7.2.2  
3 $ make prefix=/usr/local all  
4 $ sudo make prefix=/usr/local install
```

Após a conclusão, você também pode obter o Git via o próprio Git para atualizações:

```
1 $ git clone git://git.kernel.org/pub/scm/git/git.git
```

Instalando no Linux

Se você quiser instalar o Git no Linux via um instalador binário, você pode fazê-lo com a ferramenta de gerenciamento de pacotes (packages) disponível na sua distribuição. Caso você esteja no Fedora, você pode usar o yum:

```
1 $ yum install git-core
```

Ou se você estiver em uma distribuição baseada no Debian, como o Ubuntu, use o apt-get:

```
1 $ apt-get install git
```

Instalando no Mac

Existem duas formas fáceis de se instalar Git em um Mac. A mais fácil delas é usar o instalador gráfico do Git, que você pode baixar da página do Google Code (veja Figura 1-7):

1 <http://code.google.com/p/git-osx-installer>

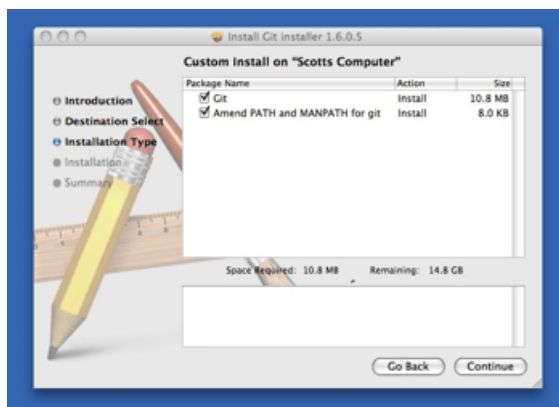


Figura 1-7. Instalador Git OS X.

A outra forma comum é instalar o Git via MacPorts (<http://www.macports.org>). Se você tem o MacPorts instalado, instale o Git via

1 `$ sudo port install git-core +svn +doc +bash_completion +gitweb`

Você não precisa adicionar todos os extras, mas você provavelmente irá querer incluir o +svn caso você tenha que usar o Git com repositórios Subversion (veja Capítulo 8).

Instalando no Windows

Instalar o Git no Windows é muito fácil. O projeto msysgit tem um dos procedimentos mais simples de instalação. Simplesmente baixe o arquivo exe do instalador a partir da página do GitHub e execute-o:

1 <http://msysgit.github.com>

Após concluir a instalação, você terá tanto uma versão command line (linha de comando, incluindo um cliente SSH que será útil depois) e uma GUI padrão.

1.5 Primeiros passos - Configuração Inicial do Git

Configuração Inicial do Git

Agora que você tem o Git em seu sistema, você pode querer fazer algumas coisas para customizar seu ambiente Git. Você só precisa fazer uma vez; as configurações serão mantidas entre atualizações. Você também poderá alterá-las a qualquer momento executando os comandos novamente.

Git vem com uma ferramenta chamada `git config` que permite a você ler e definir variáveis de configuração que controlam todos os aspectos de como o Git parece e opera. Essas variáveis podem ser armazenadas em três lugares diferentes:

- arquivo `/etc/gitconfig`: Contém valores para todos usuários do sistema e todos os seus repositórios. Se você passar a opção `--system` para `git config`, ele lerá e escreverá a partir deste arquivo especificamente.
- arquivo `~/.gitconfig`: É específico para seu usuário. Você pode fazer o Git ler e escrever a partir deste arquivo especificamente passando a opção `--global`.
- arquivo de configuração no diretório `git` (ou seja, `.git/config`) de qualquer repositório que você está utilizando no momento: Específico para aquele único repositório. Cada nível sobrepõem o valor do nível anterior, sendo assim valores em `.git/config` sobrepõem aqueles em `/etc/gitconfig`.

Em sistemas Windows, Git procura pelo arquivo `.gitconfig` no diretório `$HOME` (`C:\Documents and Settings\%USER` para a maioria das pessoas). Também procura por `/etc/gitconfig`, apesar de que é relativo à raiz de MSys, que é o local onde você escolheu instalar o Git no seu sistema Windows quando executou o instalador.

Sua Identidade

A primeira coisa que você deve fazer quando instalar o Git é definir o seu nome de usuário e endereço de e-mail. Isso é importante porque todos os commits no Git utilizam essas informações, e está imutavelmente anexado nos commits que você realiza:

```
1 $ git config --global user.name "John Doe"
2 $ git config --global user.email johndoe@example.com
```

Relembrando, você só precisará fazer isso uma vez caso passe a opção `--global`, pois o Git sempre usará essa informação para qualquer coisa que você faça nesse sistema. Caso você queira sobrepôr estas com um nome ou endereço de e-mail diferentes para projetos específicos, você pode executar o comando sem a opção `--global` quando estiver no próprio projeto.

Seu Editor

Agora que sua identidade está configurada, você pode configurar o editor de texto padrão que será utilizado quando o Git precisar que você digite uma mensagem. Por padrão, Git usa o editor padrão do sistema, que é geralmente Vi ou Vim. Caso você queira utilizar um editor diferente, tal como o Emacs, você pode executar o seguinte:

```
1 $ git config --global core.editor emacs
```

Sua Ferramenta de Diff

Outra opção útil que você pode querer configurar é a ferramenta padrão de diff utilizada para resolver conflitos de merge (fusão). Digamos que você queira utilizar o vimdiff:

```
1 $ git config --global merge.tool vimdiff
```

Git aceita kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge e opendiff como ferramentas válidas para merge. Você também pode configurar uma ferramenta personalizada; veja o Capítulo 7 para maiores informações em como fazê-lo.

Verificando Suas Configurações

Caso você queira verificar suas configurações, você pode utilizar o comando `git config --list` para listar todas as configurações que o Git encontrar naquele momento:

```
1 $ git config --list
2 user.name=Scott Chacon
3 user.email=schacon@gmail.com
4 color.status=auto
5 color.branch=auto
6 color.interactive=auto
7 color.diff=auto
8 ...
```

Você pode ver algumas chaves mais de uma vez, porque o Git lê as mesmas chaves em diferentes arquivos (`/etc/gitconfig` e `~/.gitconfig`, por exemplo). Neste caso, Git usa o último valor para cada chave única que é obtida.

Você também pode verificar qual o valor que uma determinada chave tem para o Git digitando `git config {key}`:

```
1 $ git config user.name
2 Scott Chacon
```

1.6 Primeiros passos - Obtendo Ajuda

Obtendo Ajuda

Caso você precise de ajuda usando o Git, existem três formas de se obter ajuda das páginas de manual (manpage) para quaisquer comandos do Git:

```
1 $ git help <verb>
2 $ git <verb> --help
3 $ man git-<verb>
```

Por exemplo, você pode obter a manpage para o comando config executando

```
1 $ git help config
```

Estes comandos são bons porque você pode acessá-los em qualquer lugar, mesmo offline. Caso as manpages e este livro não sejam suficientes e você precise de ajuda pessoalmente, tente os canais #git ou #github no servidor IRC do Freenode (irc.freenode.net). Esses canais estão regularmente repletos com centenas de pessoas que possuem grande conhecimento sobre Git e geralmente dispostos a ajudar.

1.7 Primeiros passos - Resumo

Resumo

Você deve ter um entendimento básico do que é Git e suas diferenças em relação ao CVCS que você tem utilizado. Além disso, você deve ter uma versão do Git funcionando em seu sistema que está configurada com sua identidade pessoal. Agora é hora de aprender algumas noções básicas do Git.

Capítulo 2. Git Essencial

Se você só puder ler um capítulo para continuar a usar o Git, leia esse. Esse capítulo cobre todos os comandos básicos que você precisa para realizar a maioria das atividades que eventualmente você fará no Git. Ao final desse capítulo você deverá ser capaz de configurar e inicializar um repositório, começar e parar o monitoramento de arquivos, além de selecionar e consolidar (fazer commit) alterações. Também vamos mostrar a você como configurar o Git para ignorar certos tipos de arquivos e padrões de arquivos, como desfazer enganos de forma rápida e fácil, como pesquisar o histórico do seu projeto e visualizar alterações entre commits e como enviar e obter arquivos a partir de repositórios remotos.

2.1 Git Essencial - Obtendo um Repositório Git

Obtendo um Repositório Git

Você pode obter um projeto Git utilizando duas formas principais. A primeira faz uso de um projeto ou diretório existente e o importa para o Git. A segunda clona um repositório Git existente a partir de outro servidor.

Inicializando um Repositório em um Diretório Existente

Caso você esteja iniciando o monitoramento de um projeto existente com Git, você precisa ir para o diretório do projeto e digitar

```
1 $ git init
```

Isso cria um novo subdiretório chamado `.git` que contem todos os arquivos necessários de seu repositório — um esqueleto de repositório Git. Neste ponto, nada em seu projeto é monitorado. (Veja o Capítulo 9 para maiores informações sobre quais arquivos estão contidos no diretório `.git` que foi criado.)

Caso você queira começar a controlar o versionamento dos arquivos existentes (diferente de um diretório vazio), você provavelmente deve começar a monitorar esses arquivos e fazer um commit inicial. Você pode realizar isso com poucos comandos `git add` que especificam quais arquivos você quer monitorar, seguido de um commit:

```
1 $ git add *.c
2 $ git add README
3 $ git commit -m 'initial project version'
```

Bem, nós iremos repassar esses comandos em um momento. Neste ponto, você tem um repositório Git com arquivos monitorados e um commit inicial.

Clonando um Repositório Existente

Caso você queira copiar um repositório Git já existente — por exemplo, um projeto que você queira contribuir — o comando necessário é `git clone`. Caso você esteja familiarizado com outros sistemas VCS, tais como Subversion, você perceberá que o comando é `clone` e não `checkout`. Essa é uma diferença importante — Git recebe uma cópia de quase todos os dados que o servidor possui. Cada versão de cada arquivo no histórico do projeto é obtida quando você roda `git clone`. De fato, se o disco do servidor ficar corrompido, é possível utilizar um dos clones em qualquer cliente para reaver o servidor no estado em que estava quando foi clonado (você pode perder algumas características do servidor, mas todos os dados versionados estarão lá — veja o Capítulo 4 para maiores detalhes).

Você clona um repositório com `git clone [url]`. Por exemplo, caso você quera clonar a biblioteca Git do Ruby chamada Grit, você pode fazê-lo da seguinte forma:

```
1 $ git clone git://github.com/schacon/grit.git
```

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

```
1 $ git clone git://github.com/schacon/grit.git mygrit
```

Este comando faz exatamente a mesma coisa que o anterior, mas o diretório alvo será chamado `mygrit`.

O Git possui diversos protocolos de transferência que você pode utilizar. O exemplo anterior utiliza o protocolo `git://`, mas você também pode ver `http(s)://` ou `user@server:/path.git`, que utilizam o protocolo de transferência SSH. No Capítulo 4, introduziremos 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.

2.2 Git Essencial - Gravando Alterações no Repositório

Gravando Alterações no Repositório

Você tem um repositório Git e um checkout ou cópia funcional dos arquivos para esse projeto. Você precisa fazer algumas mudanças e fazer o commit das partes destas mudanças em seu repositório cada vez que o projeto atinge um estado no qual você queira gravar.

Lembre-se que cada arquivo em seu diretório de trabalho pode estar em um de dois estados: monitorado ou não monitorado. Arquivos monitorados são arquivos que estavam no último snapshot; podendo estar inalterados, modificados ou selecionados. Arquivos não monitorados são todo o restante — qualquer arquivo em seu diretório de trabalho que não estava no último snapshot e também não estão em sua área de seleção. Quando um repositório é inicialmente clonado, todos os seus arquivos estarão monitorados e inalterados porque você simplesmente os obteve e ainda não os editou.

Conforme você edita esses arquivos, o Git passa a vê-los como modificados, porque você os alterou desde seu último commit. Você seleciona esses arquivos modificados e então faz o commit de todas as alterações selecionadas e o ciclo se repete. Este ciclo é apresentado na Figura 2-1.

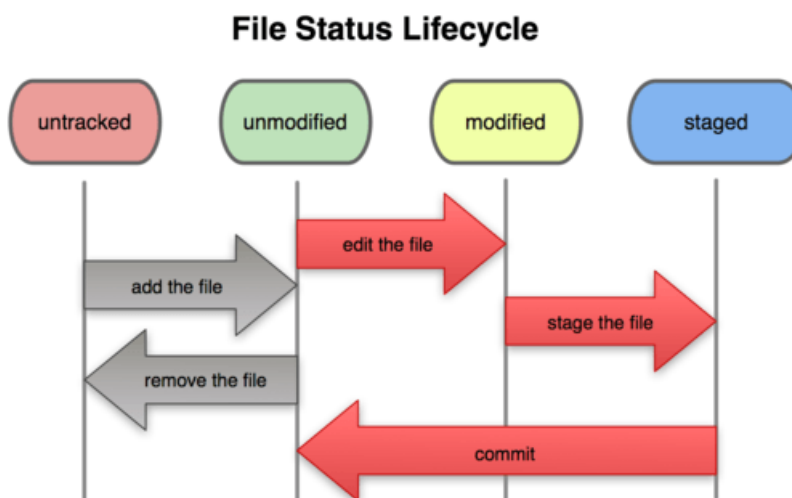


Figura 2-1. O ciclo de vida dos status de seus arquivos.

Verificando o Status de Seus Arquivos

A principal ferramenta utilizada para determinar quais arquivos estão em quais estados é o comando `git status`. Se você executar este comando diretamente após uma clonagem, você deverá ver algo similar a isso:


```
1 $ git status
2 # On branch master
3 nothing to commit (working directory clean)
```

Isso significa que você tem um diretório de trabalho limpo — em outras palavras, não existem arquivos monitorados e modificados. Git também não encontrou qualquer arquivo não monitorado, caso contrário eles seriam listados aqui. Por fim, o comando lhe mostra em qual branch você se encontra. Por enquanto, esse sempre é o master, que é o padrão; você não deve se preocupar com isso. No próximo capítulo nós vamos falar sobre branches e referências em detalhes.

Vamos dizer que você adicione um novo arquivo em seu projeto, um simples arquivo README. Caso o arquivo não exista e você execute `git status`, você verá o arquivo não monitorado dessa forma:

```
1 $ vim README
2 $ git status
3 # On branch master
4 # Untracked files:
5 #   (use "git add <file>..." to include in what will be committed)
6 #
7 #   README
8 nothing added to commit but untracked files present (use "git add" to track)
```

Você pode ver que o seu novo arquivo README não está sendo monitorado, pois está listado sob o cabeçalho “Untracked files” na saída do comando status. Não monitorado significa basicamente que o Git está vendo um arquivo que não existia na última captura (commit); o Git não vai incluí-lo nas suas capturas de commit até que você o diga explicitamente que assim o faça. Ele faz isso para que você não inclua acidentalmente arquivos binários gerados, ou outros arquivos que você não têm a intenção de incluir. Digamos, que você queira incluir o arquivo README, portanto vamos começar a monitorar este arquivo.

Monitorando Novos Arquivos

Para passar a monitorar um novo arquivo, use o comando `git add`. Para monitorar o arquivo README, você pode rodar isso:

```
1 $ git add README
```

Se você rodar o comando status novamente, você pode ver que o seu arquivo README agora está sendo monitorado e está selecionado:

```
1 $ git status
2 # On branch master
3 # Changes to be committed:
4 #   (use "git reset HEAD <file>..." to unstage)
5 #
6 #       new file:   README
7 #
```

Você pode dizer que ele está selecionado pois está sob o cabeçalho “Changes to be committed”. Se você commitar neste ponto, a versão do arquivo no momento em que você rodou o comando `git add` é a que estará na captura (snapshot) do histórico. Você deve se lembrar que quando rodou o comando `git init` anteriormente, logo em seguida rodou o comando `git add` (arquivos) — fez isso para passar a monitorar os arquivos em seu diretório. O comando `git add` recebe um caminho de um arquivo ou diretório; se é de um diretório, o comando adiciona todos os arquivos do diretório recursivamente.

Selecionando Arquivos Modificados

Vamos alterar um arquivo que já está sendo monitorado. Se você alterar um arquivo previamente monitorado chamado `benchmarks.rb` e então rodar o comando `status` novamente, você terá algo semelhante a:

```
1 $ git status
2 # On branch master
3 # Changes to be committed:
4 #   (use "git reset HEAD <file>..." to unstage)
5 #
6 #       new file:   README
7 #
8 # Changes not staged for commit:
9 #   (use "git add <file>..." to update what will be committed)
10 #
11 #       modified:   benchmarks.rb
12 #
```

O arquivo `benchmarks.rb` aparece sob a seção chamada “Changes not staged for commit” — que significa que um arquivo monitorado foi modificado no diretório de trabalho, mas ainda não foi selecionado (staged). Para selecioná-lo, utilize o comando `git add` (é um comando com várias funções — você o utiliza para monitorar novos arquivos, selecionar arquivos, e para fazer outras coisas como marcar como resolvido arquivos com conflito). Agora vamos rodar o comando `git add` para selecionar o arquivo `benchmarks.rb`, e então rodar `git status` novamente:

```
1 $ git add benchmarks.rb
2 $ git status
3 # On branch master
4 # Changes to be committed:
5 #   (use "git reset HEAD <file>..." to unstage)
6 #
7 #       new file:   README
8 #       modified:   benchmarks.rb
9 #
```

Ambos os arquivos estão selecionados e serão consolidados no seu próximo commit. Neste momento, vamos supor que você lembrou de uma mudança que queria fazer no arquivo `benchmarks.rb` antes de commitá-lo. Você o abre novamente e faz a mudança, e então está pronto para commitar. No entanto, vamos rodar `git status` mais uma vez:

```
1 $ vim benchmarks.rb
2 $ git status
3 # On branch master
4 # Changes to be committed:
5 #   (use "git reset HEAD <file>..." to unstage)
6 #
7 #       new file:   README
8 #       modified:   benchmarks.rb
9 #
10 # Changes not staged for commit:
11 #   (use "git add <file>..." to update what will be committed)
12 #
13 #       modified:   benchmarks.rb
14 #
```

Que diabos? Agora o arquivo `benchmarks.rb` aparece listado como selecionado e não selecionado. Como isso é possível? Acontece que o Git seleciona um arquivo exatamente como ele era quando o comando `git add` foi executado. Se você fizer o commit agora, a versão do `benchmarks.rb` como estava na última vez que você rodou o comando `git add` é que será incluída no commit, não a versão do arquivo que estará no seu diretório de trabalho quando rodar o comando `git commit`. Se você modificar um arquivo depois que rodou o comando `git add`, terá de rodar o `git add` de novo para selecionar a última versão do arquivo:

```
1 $ git add benchmarks.rb
2 $ git status
3 # On branch master
4 # Changes to be committed:
5 #   (use "git reset HEAD <file>..." to unstage)
6 #
7 #       new file:   README
8 #       modified:   benchmarks.rb
9 #
```

Ignorando Arquivos

Muitas vezes, você terá uma classe de arquivos que não quer que o Git automaticamente adicione ou mostre como arquivos não monitorados. Normalmente estes arquivos são gerados automaticamente como arquivos de log ou produzidos pelo seu sistema de build. Nestes casos, você pode criar um arquivo contendo uma lista de padrões a serem checados chamado `.gitignore`. Eis um exemplo de arquivo `.gitignore`:

```
1 $ cat .gitignore
2 *.log
3 *~
```

A primeira linha fala para o Git ignorar qualquer arquivo finalizado em `.o` ou `.a` — arquivos objetos e archive (compactados) que devem ter produto da construção (build) de seu código. A segunda linha fala para o Git ignorar todos os arquivos que terminam com um til (~), os quais são utilizados por muitos editores de texto como o Emacs para marcar arquivos temporários. Você também pode incluir um diretório `log`, `tmp` ou `pid`; documentação gerada automaticamente; e assim por diante. Configurar um arquivo `.gitignore` antes de começar a trabalhar, normalmente é uma boa ideia, pois evita que você commite acidentalmente arquivos que não deveriam ir para o seu repositório Git.

As regras para os padrões que você pode pôr no arquivo `.gitignore` são as seguintes:

- Linhas em branco ou iniciando com `#` são ignoradas.
- Padrões glob comuns funcionam.
- Você pode terminar os padrões com uma barra (/) para especificar diretórios.
- Você pode negar um padrão ao iniciá-lo com um ponto de exclamação (!).

Padrões glob são como expressões regulares simples que os shells usam. Um asterisco (*) significa zero ou mais caracteres; `[abc]` condiz com qualquer um dos caracteres de dentro dos colchetes (nesse caso, a, b, ou c); um ponto de interrogação (?) condiz com um único caractere; e os caracteres separados por hífen dentro de colchetes (`[0-9]`) condizem à qualquer um dos caracteres entre eles (neste caso, de 0 à 9).

Segue um outro exemplo de arquivo `.gitignore`:

```
1  # um comentário - isto é ignorado
2  # sem arquivos terminados em .a
3  *.a
4  # mas rastreie lib.a, mesmo que você tenha ignorado arquivos terminados em .a aci\
5  ma
6  !lib.a
7  # apenas ignore o arquivo TODO na raiz, não o subdiretório TODO
8  /TODO
9  # ignore todos os arquivos no diretório build/
10 build/
11 # ignore doc/notes.txt mas, não ignore doc/server/arch.txt
12 doc/*.txt
```

Visualizando Suas Mudanças Seleccionadas e Não Seleccionadas

Se o comando `git status` for muito vago — você quer saber exatamente o que você alterou, não apenas quais arquivos foram alterados — você pode utilizar o comando `git diff`. Nós trataremos o comando `git diff` em mais detalhes posteriormente; mas provavelmente você vai utilizá-lo com frequência para responder estas duas perguntas: O que você alterou, mas ainda não seleccionou (stage)? E o que você seleccionou, que está para ser commitado? Apesar do comando `git status` responder essas duas perguntas de maneira geral, o `git diff` mostra as linhas exatas que foram adicionadas e removidas — o patch, por assim dizer.

Vamos dizer que você edite e selecione o arquivo `README` de novo e então edite o arquivo `benchmarks.rb` sem seleccioná-lo. Se você rodar o comando `status`, você novamente verá algo assim:

```
1  $ git status
2  # On branch master
3  # Changes to be committed:
4  #   (use "git reset HEAD <file>..." to unstage)
5  #
6  #       new file:   README
7  #
8  # Changes not staged for commit:
9  #   (use "git add <file>..." to update what will be committed)
10 #
11 #       modified:   benchmarks.rb
12 #
```

Para ver o que você alterou mas ainda não seleccionou, digite o comando `git diff` sem nenhum argumento:

```

1 $ git diff
2 diff --git a/benchmarks.rb b/benchmarks.rb
3 index 3cb747f..da65585 100644
4 --- a/benchmarks.rb
5 +++ b/benchmarks.rb
6 @@ -36,6 +36,10 @@ def main
7     @commit.parents[0].parents[0].parents[0]
8     end
9
10 +     run_code(x, 'commits 1') do
11 +         git.commits.size
12 +     end
13 +
14     run_code(x, 'commits 2') do
15         log = git.commits('master', 15)
16         log.size

```

Este comando compara o que está no seu diretório de trabalho com o que está na sua área de seleção (staging). O resultado te mostra as mudanças que você fez que ainda não foram selecionadas.

Se você quer ver o que selecionou que irá no seu próximo commit, pode utilizar `git diff --cached`. (Nas versões do Git 1.6.1 e superiores, você também pode utilizar `git diff --staged`, que deve ser mais fácil de lembrar.) Este comando compara as mudanças selecionadas com o seu último commit:

```

1 $ git diff --cached
2 diff --git a/README b/README
3 new file mode 100644
4 index 0000000..03902a1
5 --- /dev/null
6 +++ b/README2
7 @@ -0,0 +1,5 @@
8 +grit
9 + by Tom Preston-Werner, Chris Wanstrath
10 + http://github.com/mojombo/grit
11 +
12 +Grit is a Ruby library for extracting information from a Git repository

```

É importante notar que o `git diff` por si só não mostra todas as mudanças desde o último commit — apenas as mudanças que ainda não foram selecionadas. Isso pode ser confuso, pois se você selecionou todas as suas mudanças, `git diff` não te dará nenhum resultado.

Como um outro exemplo, se você selecionar o arquivo `benchmarks.rb` e então editá-lo, você pode utilizar o `git diff` para ver as mudanças no arquivo que estão selecionadas, e as mudanças que não estão:

```

1 $ git add benchmarks.rb
2 $ echo '# test line' >> benchmarks.rb
3 $ git status
4 # On branch master
5 #
6 # Changes to be committed:
7 #
8 #   modified:   benchmarks.rb
9 #
10 # Changes not staged for commit:
11 #
12 #   modified:   benchmarks.rb
13 #

```

Agora você pode utilizar o `git diff` para ver o que ainda não foi selecionado:

```

1 $ git diff
2 diff --git a/benchmarks.rb b/benchmarks.rb
3 index e445e28..86b2f7c 100644
4 --- a/benchmarks.rb
5 +++ b/benchmarks.rb
6 @@ -127,3 +127,4 @@ end
7   main()
8
9   ##pp Grit::GitRuby.cache_client.stats
10  += test line

```

E executar `git diff --cached` para ver o que você já alterou para o estado staged até o momento:

```

1 $ git diff --cached
2 diff --git a/benchmarks.rb b/benchmarks.rb
3 index 3cb747f..e445e28 100644
4 --- a/benchmarks.rb
5 +++ b/benchmarks.rb
6 @@ -36,6 +36,10 @@ def main
7     @commit.parents[0].parents[0].parents[0]
8     end
9
10  +     run_code(x, 'commits 1') do
11  +         git.commits.size
12  +     end
13  +

```

```

14     run_code(x, 'commits 2') do
15         log = git.commits('master', 15)
16         log.size

```

Fazendo Commit de Suas Mudanças

Agora que a sua área de seleção está do jeito que você quer, você pode fazer o commit de suas mudanças. Lembre-se que tudo aquilo que ainda não foi selecionado — qualquer arquivo que você criou ou modificou que você não tenha rodado o comando `git add` desde que editou — não fará parte deste commit. Estes arquivos permanecerão como arquivos modificados em seu disco. Neste caso, a última vez que você rodou `git status`, viu que tudo estava selecionado, portanto você está pronto para fazer o commit de suas mudanças. O jeito mais simples é digitar `git commit`:

```
1 $ git commit
```

Ao fazer isso, seu editor de escolha é acionado. (Isto é configurado através da variável de ambiente `$EDITOR` de seu shell - normalmente vim ou emacs, apesar de poder ser configurado o que você quiser utilizando o comando `git config --global core.editor` como visto no Capítulo 1).

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

```

1  # Please enter the commit message for your changes. Lines starting
2  # with '#' will be ignored, and an empty message aborts the commit.
3  # On branch master
4  # Changes to be committed:
5  #   (use "git reset HEAD <file>..." to unstage)
6  #
7  #       new file:   README
8  #       modified:  benchmarks.rb
9  ~
10 ~
11 ~
12 ".git/COMMIT_EDITMSG" 10L, 283C

```

Você pode ver que a mensagem default do commit contém a última saída do comando `git status` comentada e uma linha vazia no início. Você pode remover estes comentários e digitar sua mensagem de commit, ou pode deixá-los ai para ajudar a lembrar o que está commitando. (Para um lembrete ainda mais explícito do que foi modificado, você pode passar a opção `-v` para o `git commit`. Ao fazer isso, aparecerá a diferença (diff) da sua mudança no editor para que possa ver exatamente o que foi feito.) Quando você sair do editor, o Git criará o seu commit com a mensagem (com os comentários e o diff retirados).

Alternativamente, você pode digitar sua mensagem de commit junto ao comando `commit` ao especificá-la após a flag `-m`, assim:


```
1 $ git commit -m "Story 182: Fix benchmarks for speed"
2 [master]: created 463dc4f: "Fix benchmarks for speed"
3 2 files changed, 3 insertions(+), 0 deletions(-)
4 create mode 100644 README
```

Agora você acabou de criar o seu primeiro commit! Você pode ver que o commit te mostrou uma saída sobre ele mesmo: qual o branch que recebeu o commit (master), qual o checksum SHA-1 que o commit teve (463dc4f), quantos arquivos foram alterados, e estatísticas a respeito das linhas adicionadas e removidas no commit.

Lembre-se que o commit grava a captura da área de seleção. Qualquer coisa que não foi selecionada ainda permanece lá modificada; você pode fazer um outro commit para adicioná-la ao seu histórico. Toda vez que você faz um commit, está gravando a captura do seu projeto o qual poderá reverter ou comparar posteriormente.

Pulando a Área de Seleção

Embora possa ser extraordinariamente útil para a elaboração de commits exatamente como você deseja, a área de seleção às vezes é um pouco mais complexa do que você precisa no seu fluxo de trabalho. Se você quiser pular a área de seleção, o Git provê um atalho simples. Informar a opção `-a` ao comando `git commit` faz com que o Git selecione automaticamente cada arquivo que está sendo monitorado antes de realizar o commit, permitindo que você pule a parte do `git add`:

```
1 $ git status
2 # On branch master
3 #
4 # Changes not staged for commit:
5 #
6 #   modified:   benchmarks.rb
7 #
8 $ git commit -a -m 'added new benchmarks'
9 [master 83e38c7] added new benchmarks
10 1 files changed, 5 insertions(+), 0 deletions(-)
```

Note que, neste caso, você não precisa rodar o `git add` no arquivo `benchmarks.rb` antes de fazer o commit.

Removendo Arquivos

Para remover um arquivo do Git, você tem que removê-lo dos arquivos que estão sendo monitorados (mais precisamente, removê-lo da sua área de seleção) e então fazer o commit. O comando `git rm` faz isso e também remove o arquivo do seu diretório para você não ver ele como arquivo não monitorado (untracked file) na próxima vez.

Se você simplesmente remover o arquivo do seu diretório, ele aparecerá em “Changes not staged for commit” (isto é, fora da sua área de seleção ou unstaged) na saída do seu `git status`:

```

1 $ rm grit.gemspec
2 $ git status
3 # On branch master
4 #
5 # Changes not staged for commit:
6 #   (use "git add/rm <file>..." to update what will be committed)
7 #
8 #       deleted:    grit.gemspec
9 #

```

Em seguida, se você rodar `git rm`, a remoção do arquivo é colocada na área de seleção:

```

1 $ git rm grit.gemspec
2 rm 'grit.gemspec'
3 $ git status
4 # On branch master
5 #
6 # Changes to be committed:
7 #   (use "git reset HEAD <file>..." to unstage)
8 #
9 #       deleted:    grit.gemspec
10 #

```

Na próxima vez que você fizer o commit, o arquivo sumirá e não será mais monitorado. Se você modificou o arquivo e já o adicionou na área de seleção, você deve forçar a remoção com a opção `-f`. Essa é uma funcionalidade de segurança para prevenir remoções acidentais de dados que ainda não foram gravados em um snapshot e não podem ser recuperados do Git.

Outra coisa útil que você pode querer fazer é manter o arquivo no seu diretório, mas apagá-lo da sua área de seleção. Em outras palavras, você quer manter o arquivo no seu disco rígido mas não quer que o Git o monitore mais. Isso é particularmente útil se você esqueceu de adicionar alguma coisa no seu arquivo `.gitignore` e acidentalmente o adicionou, como um grande arquivo de log ou muitos arquivos `.a` compilados. Para fazer isso, use a opção `--cached`:

```

1 $ git rm --cached readme.txt

```

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

```

1 $ git rm log/\*.log

```

Note a barra invertida (`\`) na frente do `*`. Isso é necessário pois o Git faz sua própria expansão no nome do arquivo além da sua expansão no nome do arquivo no shell. Esse comando remove todos os arquivos que tem a extensão `.log` no diretório `log/`. Ou, você pode fazer algo como isso:

```
1 $ git rm \*~
```

Esse comando remove todos os arquivos que terminam com ~.

Movendo Arquivos

Diferente de muitos sistemas VCS, o Git não monitora explicitamente arquivos movidos. Se você renomeia um arquivo, nenhum metadado é armazenado no Git que identifique que você renomeou o arquivo. No entanto, o Git é inteligente e tenta descobrir isso depois do fato — lidaremos com detecção de arquivos movidos um pouco mais tarde.

É um pouco confuso que o Git tenha um comando `mv`. Se você quiser renomear um arquivo no Git, você pode fazer isso com

```
1 $ git mv arquivo_origem arquivo_destino
```

e funciona. De fato, se você fizer algo desse tipo e consultar o status, você verá que o Git considera que o arquivo foi renomeado:

```
1 $ git mv README.txt README
2 $ git status
3 # On branch master
4 # Your branch is ahead of 'origin/master' by 1 commit.
5 #
6 # Changes to be committed:
7 #   (use "git reset HEAD <file>..." to unstage)
8 #
9 #       renamed:    README.txt -> README
10 #
```

No entanto, isso é equivalente a rodar algo como:

```
1 $ mv README.txt README
2 $ git rm README.txt
3 $ git add README
```

O Git descobre que o arquivo foi renomeado implicitamente, então ele não se importa se você renomeou por este caminho ou com o comando `mv`. A única diferença real é que o comando `mv` é mais conveniente, executa três passos de uma vez. O mais importante, você pode usar qualquer ferramenta para renomear um arquivo, e usar `add/rm` depois, antes de consolidar com o `commit`.

2.3 Git Essencial - Visualizando o Histórico de Commits

Visualizando o Histórico de Commits

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

Estes exemplos usam um projeto muito simples chamado `simplegit`, que eu frequentemente uso para demonstrações. Para pegar o projeto, execute:

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

Quando você executar `git log` neste projeto, você deve ter uma saída como esta:

```
1 $ git log
2 commit ca82a6dff817ec66f44342007202690a93763949
3 Author: Scott Chacon <schacon@gee-mail.com>
4 Date:   Mon Mar 17 21:52:11 2008 -0700
5
6     changed the verison number
7
8 commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
9 Author: Scott Chacon <schacon@gee-mail.com>
10 Date:   Sat Mar 15 16:40:33 2008 -0700
11
12     removed unnecessary test code
13
14 commit a11bef06a3f659402fe7563abf99ad00de2209e6
15 Author: Scott Chacon <schacon@gee-mail.com>
16 Date:   Sat Mar 15 10:31:28 2008 -0700
17
18     first commit
```

Por padrão, sem argumentos, `git log` lista os commits feitos naquele repositório em ordem cronológica reversa. Isto é, os commits mais recentes primeiro. Como você pode ver, este comando lista cada commit com seu checksum SHA-1, o nome e e-mail do autor, a data e a mensagem do commit.

Um grande número e variedade de opções para o comando `git log` estão disponíveis para mostrá-lo exatamente o que você quer ver. Aqui, nós mostraremos algumas das opções mais usadas.

Uma das opções mais úteis é `-p`, que mostra o diff introduzido em cada commit. Você pode ainda usar `-2`, que limita a saída somente às duas últimas entradas.

```

1  $ git log -p -2
2  commit ca82a6dff817ec66f44342007202690a93763949
3  Author: Scott Chacon <schacon@gee-mail.com>
4  Date:   Mon Mar 17 21:52:11 2008 -0700
5
6      changed the verison number
7
8  diff --git a/Rakefile b/Rakefile
9  index a874b73..8f94139 100644
10 --- a/Rakefile
11 +++ b/Rakefile
12 @@ -5,7 +5,7 @@ require 'rake/gempackagetask'
13    spec = Gem::Specification.new do |s|
14      s.version   = "0.1.0"
15      s.version   = "0.1.1"
16      s.author    = "Scott Chacon"
17
18  commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
19  Author: Scott Chacon <schacon@gee-mail.com>
20  Date:   Sat Mar 15 16:40:33 2008 -0700
21
22      removed unnecessary test code
23
24  diff --git a/lib/simplegit.rb b/lib/simplegit.rb
25  index a0a60ae..47c6340 100644
26 --- a/lib/simplegit.rb
27 +++ b/lib/simplegit.rb
28 @@ -18,8 +18,3 @@ class SimpleGit
29     end
30
31   end
32 -
33 -if $0 == __FILE__
34 -  git = SimpleGit.new
35 -  puts git.show
36 -end
37 \ No newline at end of file

```

Esta opção mostra a mesma informação, mas com um diff diretamente seguido de cada entrada. Isso é muito útil para revisão de código ou para navegar rapidamente e saber o que aconteceu durante uma série de commits que um colaborador adicionou. Você pode ainda usar uma série de opções de sumarização com `git log`. Por exemplo, se você quiser ver algumas estatísticas abreviadas para cada commit, você pode usar a opção `--stat`

```

1  $ git log --stat
2  commit ca82a6dff817ec66f44342007202690a93763949
3  Author: Scott Chacon <schacon@gee-mail.com>
4  Date:   Mon Mar 17 21:52:11 2008 -0700
5
6      changed the verison number
7
8  Rakefile |      2 +-
9  1 files changed, 1 insertions(+), 1 deletions(-)
10
11 commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
12 Author: Scott Chacon <schacon@gee-mail.com>
13 Date:   Sat Mar 15 16:40:33 2008 -0700
14
15      removed unnecessary test code
16
17  lib/simplegit.rb |      5 -----
18  1 files changed, 0 insertions(+), 5 deletions(-)
19
20 commit a11bef06a3f659402fe7563abf99ad00de2209e6
21 Author: Scott Chacon <schacon@gee-mail.com>
22 Date:   Sat Mar 15 10:31:28 2008 -0700
23
24      first commit
25
26  README          |      6 ++++++
27  Rakefile         |     23 ++++++
28  lib/simplegit.rb |     25 ++++++
29  3 files changed, 54 insertions(+), 0 deletions(-)

```

Como você pode ver, a opção `--stat` imprime abaixo de cada commit uma lista de arquivos modificados, quantos arquivos foram modificados, e quantas linhas nestes arquivos foram adicionadas e removidas. Ele ainda mostra um resumo destas informações no final. Outra opção realmente útil é `--pretty`. Esta opção muda a saída do log para outro formato que não o padrão. Algumas opções pré-construídas estão disponíveis para você usar. A opção `oneline` mostra cada commit em uma única linha, o que é útil se você está olhando muitos commits. Em adição, as opções `short`, `full` e `fuller` mostram a saída aproximadamente com o mesmo formato, mas com menos ou mais informações, respectivamente:

```

1 $ git log --pretty=oneline
2 ca82a6dff817ec66f44342007202690a93763949 changed the verison number
3 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
4 a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

```

A opção mais interessante é `format`, que permite que você especifique seu próprio formato de saída do log. Isto é especialmente útil quando você está gerando saída para análise automatizada (parsing) — porque você especifica o formato explicitamente, você sabe que ele não vai mudar junto com as atualizações do Git:

```

1 $ git log --pretty=format:"%h - %an, %ar : %s"
2 ca82a6d - Scott Chacon, 11 months ago : changed the verison number
3 085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
4 a11bef0 - Scott Chacon, 11 months ago : first commit

```

Tabela 2-1 lista algumas das opções mais importantes para formatação.

Opção	Descrição de Saída
%H	Hash do commit
%h	Hash do commit abreviado
%T	Árvore hash
%t	Árvore hash abreviada
%P	Hashes pais
%p	Hashes pais abreviados
%an	Nome do autor
%ae	Email do autor
%ad	Data do autor (formato respeita a opção <code>-date=</code>)
%ar	Data do autor, relativa
%cn	Nome do committer
%ce	Email do committer
%cd	Data do committer
%cr	Data do committer, relativa
%s	Assunto

Você deve estar se perguntando qual a diferença entre autor e committer. O autor é a pessoa que originalmente escreveu o trabalho, enquanto o committer é a pessoa que por último aplicou o trabalho. Então, se você envia um patch para um projeto, e algum dos membros do núcleo o aplicam, ambos receberão créditos — você como o autor, e o membro do núcleo como o committer. Nós cobriremos esta distinção um pouco mais no Capítulo 5.

As opções `oneline` e `format` são particularmente úteis com outra opção chamada `--graph`. Esta opção gera um agradável gráfico ASCII mostrando seu branch e histórico de merges, que nós podemos ver em nossa cópia do repositório do projeto Grit:

```

1 $ git log --pretty=format:"%h %s" --graph
2 * 2d3acf9 ignore errors from SIGCHLD on trap
3 * 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
4 | \
5 | * 420eac9 Added a method for getting the current branch.
6 * | 30e367c timeout code and tests
7 * | 5a09431 add timeout protection to grit
8 * | e1193f8 support for heads with slashes in them
9 | /
10 * d6016bc require time for xmlschema
11 * 11d191e Merge branch 'defunkt' into local

```

Estas são apenas algumas opções de formatação de saída do `git log` — há muito mais. A tabela 2-2 lista as opções que nós cobrimos e algumas outras comuns que podem ser úteis, junto com a descrição de como elas mudam a saída do comando `log`.

Opção	Descrição
<code>-p</code>	Mostra o patch introduzido com cada commit.
<code>-stat</code>	Mostra estatísticas de arquivos modificados em cada commit.
<code>-shortstat</code>	Mostra somente as linhas modificadas/inseridas/excluídas do comando <code>-stat</code> .
<code>-name-only</code>	Mostra a lista de arquivos modificados depois das informações do commit.
<code>-name-status</code>	Mostra a lista de arquivos afetados com informações sobre adição/modificação/exclusão dos mesmos.
<code>-abbrev-commit</code>	Mostra somente os primeiros caracteres do checksum SHA-1 em vez de todos os 40.
<code>-relative-date</code>	Mostra a data em um formato relativo (por exemplo, “2 semanas atrás”) em vez de usar o formato de data completo.
<code>-graph</code>	Mostra um gráfico ASCII do branch e histórico de merges ao lado da saída de <code>log</code> .
<code>-pretty</code>	Mostra os commits em um formato alternativo. Opções incluem <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> , e <code>format</code> (onde você especifica seu próprio formato).

Limitando a Saída de Log

Em adição às opções de formatação, `git log` tem inúmeras opções de limitações úteis — que são opções que lhe deixam mostrar somente um subconjunto de commits. Você já viu algumas — a opção `-2`, que mostra apenas os dois últimos commits. De fato, você pode fazer `-<n>`, onde `n` é qualquer inteiro para mostrar os últimos `n` commits. Na verdade, você provavelmente não usará isso frequentemente, porque por padrão o Git enfileira toda a saída em um paginador, e então você vê somente uma página da saída do `log` por vez.

No entanto, as opções de limites de tempo como `--since` e `--until` são muito úteis. Por exemplo,

este comando pega a lista de commits feitos nas últimas duas semanas:

```
1 $ git log --since=2.weeks
```

Este comando funciona com vários formatos — você pode especificar uma data específica (“2008-01-15”) ou uma data relativa como “2 years 1 day 3 minutes ago”.

Você pode ainda filtrar a lista de commits que casam com alguns critérios de busca. A opção `--author` permite que você filtre por algum autor específico, e a opção `--grep` deixa você buscar por palavras chave nas mensagens dos commits. (Note que se você quiser especificar ambas as opções `author` e `grep` simultaneamente, você deve adicionar `--all-match`, ou o comando considerará commits que casam com qualquer um.)

A última opção realmente útil para passar para `git log` como um filtro, é o caminho. Se você especificar um diretório ou um nome de arquivo, você pode limitar a saída a commits que modificaram aqueles arquivos. Essa é sempre a última opção, e geralmente é precedida por dois traços (`--`) para separar caminhos das opções.

Na Tabela 2-3 nós listamos estas e outras opções comuns para sua referência.

Opção	Descrição
<code>-(n)</code>	Mostra somente os últimos <code>n</code> commits.
<code>--since</code> , <code>--after</code>	Limita aos commits feitos depois da data especificada.
<code>--until</code> , <code>--before</code>	Limita aos commits feitos antes da data especificada.
<code>--author</code>	Somente mostra commits que o autor casa com a string especificada.
<code>--committer</code>	Somente mostra os commits em que a entrada do committer bate com a string especificada.

Por exemplo, se você quer ver quais commits modificaram arquivos de teste no histórico do código fonte do Git que foram commitados por Julio Hamano em Outubro de 2008, e não foram merges, você pode executar algo como:

```
1 $ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
2   --before="2008-11-01" --no-merges -- t/
3 5610e3b - Fix testcase failure when extended attribute
4 acd3b9e - Enhance hold_lock_file_for_{update,append}()
5 f563754 - demonstrate breakage of detached checkout wi
6 d1a43f2 - reset --hard/read-tree --reset -u: remove un
7 51a94af - Fix "checkout --track -b newbranch" on detac
8 b0ad11e - pull: allow "git pull origin $something:$cur
```

Dos 20.000 commits mais novos no histórico do código fonte do Git, este comando mostra os 6 que casam com aqueles critérios.

Usando Interface Gráfica para Visualizar o Histórico

Se você quiser usar uma ferramenta gráfica para visualizar seu histórico de commit, você pode querer dar uma olhada em um programa Tcl/Tk chamado gitk que é distribuído com o Git. Gitk é basicamente uma ferramenta visual para git log, e ele aceita aproximadamente todas as opções de filtros que git log aceita. Se você digitar gitk na linha de comando em seu projeto, você deve ver algo como a Figura 2-2.

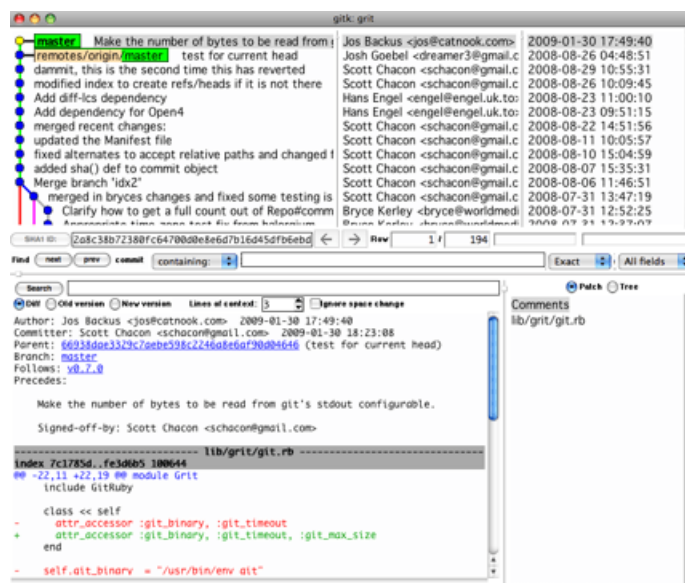


Figura 2-2. O visualizador de histórico gitk.

Você pode ver o histórico de commit na metade de cima da janela juntamente com um agradável gráfico. O visualizador de diff na metade de baixo da janela mostra a você as mudanças introduzidas em qualquer commit que você clicar.

2.4 Git Essencial - Desfazendo Coisas

Desfazendo Coisas

Em qualquer fase, você pode querer desfazer alguma coisa. Aqui, veremos algumas ferramentas básicas para desfazer modificações que você fez. Cuidado, porque você não pode desfazer algumas dessas mudanças. Essa é uma das poucas áreas no Git onde você pode perder algum trabalho se fizer errado.

Modificando Seu Último Commit

Uma das situações mais comuns para desfazer algo, acontece quando você faz o commit muito cedo e possivelmente esqueceu de adicionar alguns arquivos, ou você bagunçou sua mensagem de commit. Se você quiser tentar fazer novamente esse commit, você pode executá-lo com a opção `--amend`:

```
1 $ git commit --amend
```

Esse comando pega sua área de seleção e a utiliza no commit. Se você não fez nenhuma modificação desde seu último commit (por exemplo, você rodou esse comando imediatamente após seu commit anterior), seu snapshot será exatamente o mesmo e tudo que você mudou foi sua mensagem de commit.

O mesmo editor de mensagem de commits abre, mas ele já tem a mensagem do seu commit anterior. Você pode editar a mensagem como sempre, mas ela substituirá seu último commit.

Por exemplo, se você fez um commit e esqueceu de adicionar na área de seleção as modificações de um arquivo que gostaria de ter adicionado nesse commit, você pode fazer algo como isso:

```
1 $ git commit -m 'initial commit'
2 $ git add forgotten_file
3 $ git commit --amend
```

Depois desses três comandos você obterá um único commit — o segundo commit substitui os resultados do primeiro.

Tirando um arquivo da área de seleção

As duas próximas seções mostram como trabalhar nas suas modificações na área de seleção e diretório de trabalho. A parte boa é que o comando que você usa para ver a situação nessas duas áreas também lembra como desfazer suas alterações. Por exemplo, vamos dizer que você alterou dois arquivos e quer fazer o commit deles como duas modificações separadas, mas você acidentalmente digitou `git add *` e colocou os dois na área de seleção. Como você pode retirar um deles? O comando `git status` lembra você:

```
1 $ git add .
2 $ git status
3 # On branch master
4 # Changes to be committed:
5 #   (use "git reset HEAD <file>..." to unstage)
6 #
7 #       modified:   README.txt
8 #       modified:   benchmarks.rb
9 #
```

Logo abaixo do texto “Changes to be committed”, ele diz use `git reset HEAD <file>...` to unstage (“use `git reset HEAD <file>...` para retirá-los do estado unstaged”). Então, vamos usar esse conselho para retirar o arquivo `benchmarks.rb`:

```
1 $ git reset HEAD benchmarks.rb
2 benchmarks.rb: locally modified
3 $ git status
4 # On branch master
5 # Changes to be committed:
6 #   (use "git reset HEAD <file>..." to unstage)
7 #
8 #       modified:   README.txt
9 #
10 # Changes not staged for commit:
11 #   (use "git add <file>..." to update what will be committed)
12 #   (use "git checkout -- <file>..." to discard changes in working directory)
13 #
14 #       modified:   benchmarks.rb
15 #
```

O comando é um pouco estranho, mas funciona. O arquivo `benchmarks.rb` está modificado, mas, novamente fora da área de seleção.

Desfazendo um Arquivo Modificado

E se você perceber que não quer manter suas modificações no arquivo `benchmarks.rb`? Como você pode facilmente desfazer isso — reverter-lo para o que era antes de fazer o último commit (ou do início do clone, ou seja lá como você o conseguiu no seu diretório de trabalho)? Felizmente, `git status` diz a você como fazer isso, também. Na saída do último exemplo, a área de trabalho se parecia com isto:

```
1 # Changes not staged for commit:
2 #   (use "git add <file>..." to update what will be committed)
3 #   (use "git checkout -- <file>..." to discard changes in working directory)
4 #
5 #       modified:   benchmarks.rb
6 #
```

Ele diz explicitamente como descartar as modificações que você fez (pelo menos, as novas versões do Git, 1.6.1 em diante, fazem isso — se você tem uma versão mais antiga, uma atualização é altamente recomendável para ter alguns desses bons recursos de usabilidade). Vamos fazer o que ele diz:

```
1 $ git checkout -- benchmarks.rb
2 $ git status
3 # On branch master
4 # Changes to be committed:
5 #   (use "git reset HEAD <file>..." to unstage)
6 #
7 #       modified:   README.txt
8 #
```

Você pode ver que as alterações foram revertidas. Perceba também que esse comando é perigoso: qualquer alteração que você fez nesse arquivo foi desfeita — você acabou de copiar outro arquivo sobre ele. Nunca use esse comando a menos que você tenha certeza absoluta que não quer o arquivo. Se você só precisa tirá-lo do caminho, vamos falar sobre stash e branch no próximo capítulo; geralmente essas são maneiras melhores de agir.

Lembre-se, qualquer coisa que foi incluída com um commit no Git quase sempre pode ser recuperada. Até mesmo commits que estavam em branches que foram apagados ou commits que foram sobrescritos com um commit --amend podem ser recuperados (consulte o Capítulo 9 para recuperação de dados). No entanto, qualquer coisa que você perder que nunca foi commitada, provavelmente nunca mais será vista novamente.

2.5 Git Essencial - Trabalhando com Remotos

Trabalhando com Remotos

Para ser capaz de colaborar com qualquer projeto no Git, você precisa saber como gerenciar seus repositórios remotos. Repositórios remotos são versões do seu projeto que estão hospedados na Internet ou em uma rede em algum lugar. Você pode ter vários deles, geralmente cada um é somente leitura ou leitura/escrita pra você. Colaborar com outros envolve gerenciar esses repositórios remotos e fazer o push e pull de dados neles quando você precisa compartilhar trabalho. Gerenciar repositórios remotos inclui saber como adicionar repositório remoto, remover remotos que não são mais válidos, gerenciar vários branches remotos e defini-los como monitorados ou não, e mais. Nesta seção, vamos cobrir essas habilidades de gerenciamento remoto.

Exibindo Seus Remotos

Para ver quais servidores remotos você configurou, você pode executar o comando `git remote`. Ele lista o nome de cada remoto que você especificou. Se você tiver clonado seu repositório, você deve pelo menos ver um chamado origin — esse é o nome padrão que o Git dá ao servidor de onde você fez o clone:

```
1 $ git clone git://github.com/schacon/ticgit.git
2 Initialized empty Git repository in /private/tmp/ticgit/.git/
3 remote: Counting objects: 595, done.
4 remote: Compressing objects: 100% (269/269), done.
5 remote: Total 595 (delta 255), reused 589 (delta 253)
6 Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
7 Resolving deltas: 100% (255/255), done.
8 $ cd ticgit
9 $ git remote
10 origin
```

Você também pode especificar `-v`, que mostra a URL que o Git armazenou para o nome do remoto:

```
1 $ git remote -v
2 origin git://github.com/schacon/ticgit.git (fetch)
3 origin git://github.com/schacon/ticgit.git (push)
```

Se você tem mais de um remoto, o comando lista todos. Por exemplo, meu repositório Grit se parece com isso.

```
1 $ cd grit
2 $ git remote -v
3 bakdoor git://github.com/bakdoor/grit.git
4 cho45 git://github.com/cho45/grit.git
5 defunkt git://github.com/defunkt/grit.git
6 koke git://github.com/koke/grit.git
7 origin git@github.com:mojombo/grit.git
```

Isso significa que podemos puxar contribuições de qualquer um desses usuários muito facilmente. Mas note que somente o remoto origin é uma URL SSH, sendo o único pra onde eu posso fazer o push (vamos ver o motivo disso no Capítulo 4).

Adicionando Repositórios Remotos

Eu mencionei e dei algumas demonstrações de adição de repositórios remotos nas seções anteriores, mas aqui está como fazê-lo explicitamente. Para adicionar um novo repositório remoto no Git com um nome curto, para que você possa fazer referência facilmente, execute `git remote add [nomecurto] [url]:`

```
1 $ git remote
2 origin
3 $ git remote add pb git://github.com/paulboone/ticgit.git
4 $ git remote -v
5 origin    git://github.com/schacon/ticgit.git
6 pb       git://github.com/paulboone/ticgit.git
```

Agora você pode usar a string `pb` na linha de comando em lugar da URL completa. Por exemplo, se você quer fazer o `fetch` de todos os dados de Paul que você ainda não tem no seu repositório, você pode executar `git fetch pb`:

```
1 $ git fetch pb
2 remote: Counting objects: 58, done.
3 remote: Compressing objects: 100% (41/41), done.
4 remote: Total 44 (delta 24), reused 1 (delta 0)
5 Unpacking objects: 100% (44/44), done.
6 From git://github.com/paulboone/ticgit
7  * [new branch]      master      -> pb/master
8  * [new branch]      ticgit      -> pb/ticgit
```

O branch `master` de Paul é localmente acessível como `pb/master` — você pode fazer o merge dele em um de seus branches, ou fazer o `check out` de um branch local a partir deste ponto se você quiser inspecioná-lo.

Fazendo o Fetch e Pull de Seus Remotos

Como você acabou de ver, para pegar dados dos seus projetos remotos, você pode executar:

```
1 $ git fetch [nome-remoto]
```

Esse comando vai até o projeto remoto e pega todos os dados que você ainda não tem. Depois de fazer isso, você deve ter referências para todos os branches desse remoto, onde você pode fazer o merge ou inspecionar a qualquer momento. (Vamos ver o que são branches e como usá-los mais detalhadamente no Capítulo 3.)

Se você clonar um repositório, o comando automaticamente adiciona o remoto com o nome `origin`. Então, `git fetch origin` busca qualquer novo trabalho que foi enviado para esse servidor desde que você o clonou (ou fez a última busca). É importante notar que o comando `fetch` traz os dados para o seu repositório local — ele não faz o merge automaticamente com o seus dados ou modifica o que você está trabalhando atualmente. Você terá que fazer o merge manualmente no seu trabalho quando estiver pronto.

Se você tem um branch configurado para acompanhar um branch remoto (veja a próxima seção e o Capítulo 3 para mais informações), você pode usar o comando `git pull` para automaticamente fazer o fetch e o merge de um branch remoto no seu branch atual. Essa pode ser uma maneira mais fácil ou confortável pra você; e por padrão, o comando `git clone` automaticamente configura seu branch local master para acompanhar o branch remoto master do servidor de onde você clonou (desde que o remoto tenha um branch master). Executar `git pull` geralmente busca os dados do servidor de onde você fez o clone originalmente e automaticamente tenta fazer o merge dele no código que você está trabalhando atualmente.

Pushing Para Seus Remotos

Quando o seu projeto estiver pronto para ser compartilhado, você tem que enviá-lo para a fonte. O comando para isso é simples: `git push [nome-remoto] [branch]`. Se você quer enviar o seu branch master para o servidor origin (novamente, clonando normalmente define estes dois nomes para você automaticamente), então você pode rodar o comando abaixo para enviar o seu trabalho para o servidor:

```
1 $ git push origin master
```

Este comando funciona apenas se você clonou de um servidor que você têm permissão para escrita, e se mais ninguém enviou dados no meio tempo. Se você e mais alguém clonarem ao mesmo tempo, e você enviar suas modificações após a pessoa ter enviado as dela, o seu push será rejeitado. Antes, você terá que fazer um pull das modificações deste outro alguém, e incorporá-las às suas para que você tenha permissão para enviá-las. Veja o Capítulo 3 para mais detalhes sobre como enviar suas modificações para servidores remotos.

Inspecionando um Remoto

Se você quer ver mais informação sobre algum remoto em particular, você pode usar o comando `git remote show [nome-remoto]`. Se você rodar este comando com um nome específico, como origin, você verá algo assim:

```
1 $ git remote show origin
2 * remote origin
3   URL: git://github.com/schacon/ticgit.git
4   Remote branch merged with 'git pull' while on branch master
5     master
6   Tracked remote branches
7     master
8     ticgit
```


Ele lista a URL do repositório remoto assim como as branches sendo rastreadas. O resultado deste comando lhe diz que se você está na branch master e rodar `git pull`, ele automaticamente fará um merge na branch master no remoto depois que ele fizer o fetch de todas as referências remotas. Ele também lista todas as referências remotas que foram puxadas.

Este é um simples exemplo que você talvez encontre por aí. Entretanto, quando se usa o Git pra valer, você pode ver muito mais informação vindo de `git remote show`:

```
1 $ git remote show origin
2 * remote origin
3   URL: git@github.com:defunkt/github.git
4   Remote branch merged with 'git pull' while on branch issues
5     issues
6   Remote branch merged with 'git pull' while on branch master
7     master
8   New remote branches (next fetch will store in remotes/origin)
9     caching
10  Stale tracking branches (use 'git remote prune')
11    libwalker
12    walker2
13  Tracked remote branches
14    acl
15    apiv2
16    dashboard2
17    issues
18    master
19    postgres
20  Local branch pushed with 'git push'
21    master:master
```

Este comando mostra qual branch é automaticamente enviado (pushed) quando você roda `git push` em determinados branches. Ele também mostra quais branches remotos que estão no servidor e você não tem, quais branches remotos você tem e que foram removidos do servidor, e múltiplos branches que são automaticamente mesclados (merged) quando você roda `git pull`.

Removendo e Renomeando Remotos

Se você quiser renomear uma referência, em versões novas do Git você pode rodar `git remote rename` para modificar um apelido de um remoto. Por exemplo, se você quiser renomear `pb` para `paul`, você pode com `git remote rename`:

```
1 $ git remote rename pb paul
2 $ git remote
3 origin
4 paul
```

É válido mencionar que isso modifica também os nomes dos branches no servidor remoto. O que costumava ser referenciado como pb/master agora é paul/master.

Se você quiser remover uma referência por qualquer razão — você moveu o servidor ou não está mais usando um mirror específico, ou talvez um contribuidor não está mais contribuindo — você usa `git remote rm`:

```
1 $ git remote rm paul
2 $ git remote
3 origin
```

2.6 Git Essencial - Tagging

Tagging

Assim como a maioria dos VCS's, Git tem a habilidade de criar tags em pontos específicos na história do código como pontos importantes. Geralmente as pessoas usam esta funcionalidade para marcar pontos de release (v1.0, e por aí vai). Nesta seção, você aprenderá como listar as tags disponíveis, como criar novas tags, e quais são os tipos diferentes de tags.

Listando Suas Tags

Listar as tags disponíveis em Git é fácil. Apenas execute o comando `git tag`:

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

Este comando lista as tags em ordem alfabética; a ordem que elas aparecem não tem importância.

Você também pode procurar por tags com uma nomenclatura particular. O repositório de código do Git, por exemplo, contém mais de 240 tags. Se você está interessado em olhar apenas na série 1.4.2, você pode executar o seguinte:

```
1 $ git tag -l 'v1.4.2.*'
2 v1.4.2.1
3 v1.4.2.2
4 v1.4.2.3
5 v1.4.2.4
```

Criando Tags

Git têm dois tipos principais de tags: leve e anotada. Um tag leve é muito similar a uma branch que não muda — é um ponteiro para um commit específico. Tags anotadas, entretanto, são armazenadas como objetos inteiros no banco de dados do Git. Eles possuem uma chave de verificação; o nome da pessoa que criou a tag, email e data; uma mensagem relativa à tag; e podem ser assinadas e verificadas com o GNU Privacy Guard (GPG). É geralmente recomendado que você crie tags anotadas para que você tenha toda essa informação; mas se você quiser uma tag temporária ou por algum motivo você não queira armazenar toda essa informação, tags leves também estão disponíveis.

Tags Anotadas

Criando uma tag anotada em Git é simples. O jeito mais fácil é especificar `-a` quando você executar o comando `tag`:

```
1 $ git tag -a v1.4 -m 'my version 1.4'
2 $ git tag
3 v0.1
4 v1.3
5 v1.4
```

O parâmetro `-m` define uma mensagem, que é armazenada com a tag. Se você não especificar uma mensagem para uma tag anotada, o Git vai rodar seu editor de texto para você digitar alguma coisa.

Você pode ver os dados da tag junto com o commit que foi taggeado usando o comando `git show`:

```
1 $ git show v1.4
2 tag v1.4
3 Tagger: Scott Chacon <schacon@gee-mail.com>
4 Date: Mon Feb 9 14:45:11 2009 -0800
5
6 my version 1.4
7 commit 15027957951b64cf874c3557a0f3547bd83b3ff6
8 Merge: 4a447f7... a6b4c97...
9 Author: Scott Chacon <schacon@gee-mail.com>
10 Date: Sun Feb 8 19:02:46 2009 -0800
11
12 Merge branch 'experiment'
```

O comando mostra a informação da pessoa que criou a tag, a data de quando o commit foi taggeado, e a mensagem antes de mostrar a informação do commit.

Tags Assinadas

Você também pode assinar suas tags com GPG, assumindo que você tenha uma chave privada. Tudo o que você precisa fazer é usar o parâmetro `-s` ao invés de `-a`:

```
1 $ git tag -s v1.5 -m 'my signed 1.5 tag'
2 You need a passphrase to unlock the secret key for
3 user: "Scott Chacon <schacon@gee-mail.com>"
4 1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Se você rodar `git show` na tag, você poderá ver a sua assinatura GPG anexada:

```
1 $ git show v1.5
2 tag v1.5
3 Tagger: Scott Chacon <schacon@gee-mail.com>
4 Date:   Mon Feb 9 15:22:20 2009 -0800
5
6 my signed 1.5 tag
7 -----BEGIN PGP SIGNATURE-----
8 Version: GnuPG v1.4.8 (Darwin)
9
10 iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
11 Ki0An2JeAVUCAiJ70x6ZEtK+NvZAJ82/
12 =WryJ
13 -----END PGP SIGNATURE-----
14 commit 15027957951b64cf874c3557a0f3547bd83b3ff6
15 Merge: 4a447f7... a6b4c97...
16 Author: Scott Chacon <schacon@gee-mail.com>
17 Date:   Sun Feb 8 19:02:46 2009 -0800
18
19     Merge branch 'experiment'
```

Um pouco mais pra frente você aprenderá como verificar tags assinadas.

Tags Leves

Outro jeito para taggear commits é com a tag leve. Esta é basicamente a chave de verificação armazenada num arquivo — nenhuma outra informação é armazenada. Para criar uma tag leve, não informe os parâmetros `-a`, `-s`, ou `-m`:

```

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

```

Desta vez, se você executar `git show` na tag, você não verá nenhuma informação extra. O comando apenas mostra o commit:

```

1 $ git show v1.4-lw
2 commit 15027957951b64cf874c3557a0f3547bd83b3ff6
3 Merge: 4a447f7... a6b4c97...
4 Author: Scott Chacon <schacon@gee-mail.com>
5 Date: Sun Feb 8 19:02:46 2009 -0800
6
7 Merge branch 'experiment'

```

Verificando Tags

Para verificar uma tag assinada, você usa `git tag -v [nome-tag]`. Este comando usa GPG para verificar a sua assinatura. Você precisa da chave pública do assinador no seu chaveiro para este comando funcionar corretamente:

```

1 $ git tag -v v1.4.2.1
2 object 883653babd8ee7ea23e6a5c392bb739348b1eb61
3 type commit
4 tag v1.4.2.1
5 tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700
6
7 GIT 1.4.2.1
8
9 Minor fixes since 1.4.2, including git-mv and git-http with alternates.
10 gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
11 gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
12 gpg: aka "[jpeg image of size 1513]"
13 Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

```

Se você não tiver a chave pública, você receberá algo parecido com a resposta abaixo:

```

1  gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
2  gpg: Can't check signature: public key not found
3  error: could not verify the tag 'v1.4.2.1'

```

Taggeando Mais Tarde

Você também pode taggear commits mais tarde. Vamos assumir que o seu histórico de commits seja assim:

```

1  $ git log --pretty=oneline
2  15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
3  a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
4  0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
5  6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
6  0b7434d86859cc7b8c3d5e1dddfed66ff742fcfc added a commit function
7  4682c3261057305bdd616e23b64b0857d832627b added a todo file
8  166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9  9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
10 964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
11 8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

```

Agora, assuma que você esqueceu de criar uma tag para o seu projeto na versão 1.2 (v1.2), que foi no commit “updated rakefile”. Você pode adicioná-la depois. Para criar a tag no commit, você especifica a chave de verificação (ou parte dela) no final do comando:

```

1  $ git tag -a v1.2 9fceb02

```

Você pode confirmar que você criou uma tag para o seu commit:

```

1  $ git tag
2  v0.1
3  v1.2
4  v1.3
5  v1.4
6  v1.4-lw
7  v1.5
8
9  $ git show v1.2
10 tag v1.2
11 Tagger: Scott Chacon <schacon@gee-mail.com>
12 Date:   Mon Feb 9 15:32:16 2009 -0800

```

```
13
14 version 1.2
15 commit 9fceb02d0ae598e95dc970b74767f19372d61af8
16 Author: Magnus Chacon <mchacon@gee-mail.com>
17 Date: Sun Apr 27 20:43:35 2008 -0700
18
19     updated rakefile
20 ...
```

Compartilhando Tags

Por padrão, o comando `git push` não transfere tags para os servidores remotos. Você deve enviar as tags explicitamente para um servidor compartilhado após tê-las criado. Este processo é igual ao compartilhamento de branches remotos – você executa `git push origin [nome-tag]`.

```
1 $ git push origin v1.5
2 Counting objects: 50, done.
3 Compressing objects: 100% (38/38), done.
4 Writing objects: 100% (44/44), 4.56 KiB, done.
5 Total 44 (delta 18), reused 8 (delta 1)
6 To git@github.com:schacon/simplegit.git
7 * [new tag]          v1.5 -> v1.5
```

Se você tem muitas tags que você deseja enviar ao mesmo tempo, você pode usar a opção `--tags` no comando `git push`. Ele irá transferir todas as suas tags que ainda não estão no servidor remoto.

```
1 $ git push origin --tags
2 Counting objects: 50, done.
3 Compressing objects: 100% (38/38), done.
4 Writing objects: 100% (44/44), 4.56 KiB, done.
5 Total 44 (delta 18), reused 8 (delta 1)
6 To git@github.com:schacon/simplegit.git
7 * [new tag]          v0.1 -> v0.1
8 * [new tag]          v1.2 -> v1.2
9 * [new tag]          v1.4 -> v1.4
10 * [new tag]          v1.4-lw -> v1.4-lw
11 * [new tag]          v1.5 -> v1.5
```

Agora, quando alguém clonar ou fizer um pull do seu repositório, eles irão ter todas as suas tags também.

2.7 Git Essencial - Dicas e Truques

Dicas e Truques

Antes de terminarmos este capítulo em Git Essencial, algumas dicas e truques podem tornar a sua experiência com Git um pouco mais simples, fácil e familiar. Muitas pessoas usam Git sem nenhuma dessas dicas, e não iremos referir à elas ou assumir que você as usou mais tarde no livro; mas você deve ao menos saber como executá-las.

Preenchimento Automático

Se você usa um shell Bash, você pode habilitar um script de preenchimento automático que vem com o Git. Faça download do código fonte, e olhe no diretório `contrib/completion`; lá deve existir um arquivo chamado `git-completion.bash`. Copie este arquivo para o seu diretório home, e adicione a linha abaixo ao seu arquivo `.bashrc`:

```
1 source ~/.git-completion.bash
```

Se você quiser configurar Git para automaticamente ter preenchimento automático para todos os usuários, copie o script para o diretório `/opt/local/etc/bash_completion.d` em Mac ou para o diretório `/etc/bash_completion.d/` em Linux. Este é o diretório de scripts que o Bash automaticamente carregará para prover o preenchimento automático.

Se você estiver usando Windows com Git Bash, que é o padrão quando instalando Git no Windows com msysGit, o preenchimento automático deve estar pré-configurado.

Pressiona a tecla Tab quando estiver escrevendo um comando Git, e ele deve retornar uma lista de sugestões para você escolher:

```
1 $ git co<tab><tab>
2 commit config
```

Neste caso, escrevendo `git co` e pressionando a tecla Tab duas vezes, ele sugere `commit` e `config`. Adicionando `m<tab>` completa `git commit` automaticamente.

Isto também funciona com opções, o que é provavelmente mais útil. Por exemplo, se você estiver executando o comando `git log` e não consegue lembrar uma das opções, você pode começar a escrever e pressionar Tab para ver o que corresponde:

```
1 $ git log --s<tab>
2 --shortstat --since= --src-prefix= --stat --summary
```

Este é um truque bem bacana e irá te poupar tempo e leitura de documentação.

Pseudônimos no Git

O Git não interfere em seu comando se você digitá-lo parcialmente. Se você não quiser digitar o texto todo de cada comando Git, você pode facilmente criar um pseudônimo para cada um usando `git config`. Abaixo alguns exemplos que você pode usar:

```
1 $ git config --global alias.co checkout
2 $ git config --global alias.br branch
3 $ git config --global alias.ci commit
4 $ git config --global alias.st status
```

Isto significa que, por exemplo, ao invés de digitar `git commit`, você só precisa digitar `git ci`. Quanto mais você usar Git, você provavelmente usará outros comandos com frequência também; neste caso, não hesite em criar novos pseudônimos.

Esta técnica também pode ser útil para criar comandos que você acha que devem existir. Por exemplo, para corrigir o problema de usabilidade que você encontrou durante o unstaging de um arquivo, você pode adicionar o seu próprio pseudônimo `unstage` para o Git:

```
1 $ git config --global alias.unstage 'reset HEAD --'
```

Isto faz dos dois comandos abaixo equivalentes:

```
1 $ git unstage fileA
2 $ git reset HEAD fileA
```

Parece mais claro. É também comum adicionar um comando `last`, assim:

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

Desse jeito, você pode ver o último comando mais facilmente:

```
1 $ git last
2 commit 66938dae3329c7aebe598c2246a8e6af90d04646
3 Author: Josh Goebel <dreamer3@example.com>
4 Date: Tue Aug 26 19:48:51 2008 +0800
5
6     test for current head
7
8     Signed-off-by: Scott Chacon <schacon@example.com>
```

Como você pode ver, Git simplesmente substitui o novo comando com o pseudônimo que você deu à ele. Entretanto, talvez você queira rodar um comando externo ao invés de um sub comando do Git. Neste caso, você começa o comando com `!`. Isto é útil se você escreve suas próprias ferramentas que trabalham com um repositório Git. Podemos demonstrar criando o pseudônimo `git visual` para rodar `gitk`:

```
1 $ git config --global alias.visual '!gitk'
```

2.8 Git Essencial - Sumário

Sumário

Neste ponto, você pode executar todas as operações locais básicas do Git — criar ou clonar um repositório, efetuar mudanças, fazer o stage e commit de suas mudanças, e ver o histórico de todas as mudanças do repositório. A seguir, vamos cobrir a melhor característica do Git: o modelo de branching.

Capítulo 3. Ramificação (Branching) no Git

Quase todos os VCS têm alguma forma de suporte a ramificação (branching). Criar um branch significa dizer que você vai divergir da linha principal de desenvolvimento e continuar a trabalhar sem bagunçar essa linha principal. Em muitas ferramentas VCS, este é um processo um pouco caro, muitas vezes exigindo que você crie uma nova cópia do seu diretório de código-fonte, o que pode levar um longo tempo para grandes projetos.

Algumas pessoas se referem ao modelo de ramificação em Git como sua característica “matadora”, e que certamente o destaca na comunidade de VCS. Por que ele é tão especial? A forma como o Git cria branches é inacreditavelmente leve, fazendo com que as operações com branches sejam praticamente instantâneas e a alternância entre os branches seja tão rápida quanto. Ao contrário de muitos outros VCSs, o Git incentiva um fluxo de trabalho no qual se fazem branches e merges com frequência, até mesmo várias vezes ao dia. Compreender e dominar esta característica lhe dará uma ferramenta poderosa e única e poderá literalmente mudar a maneira de como você desenvolve.

3.1 Ramificação (Branching) no Git - O que é um Branch

O que é um Branch

Para compreender realmente a forma como o Git cria branches, precisamos dar um passo atrás e examinar como o Git armazena seus dados. Como você pode se lembrar do capítulo 1, o Git não armazena dados como uma série de mudanças ou deltas, mas sim como uma série de snapshots.

Quando você faz um commit no Git, o Git guarda um objeto commit que contém um ponteiro para o snapshot do conteúdo que você colocou na área de seleção, o autor e os metadados da mensagem, zero ou mais ponteiros para o commit ou commits que são pais deste commit: nenhum pai para o commit inicial, um pai para um commit normal e múltiplos pais para commits que resultem de um merge de dois ou mais branches.

Para visualizar isso, vamos supor que você tenha um diretório contendo três arquivos, e colocou todos eles na área de seleção e fez o commit. Colocar na área de seleção cria o checksum de cada arquivo (o hash SHA-1 que nos referimos no capítulo 1), armazena esta versão do arquivo no repositório Git (o Git se refere a eles como blobs), e acrescenta este checksum à área de seleção (staging area):

```

1 $ git add README test.rb LICENSE
2 $ git commit -m 'commit inicial do meu projeto'

```

Quando você cria um commit executando `git commit`, o Git calcula o checksum de cada subdiretório (neste caso, apenas o diretório raiz do projeto) e armazena os objetos de árvore no repositório Git. O Git em seguida, cria um objeto commit que tem os metadados e um ponteiro para a árvore do projeto raiz, então ele pode recriar este snapshot quando necessário.

Seu repositório Git agora contém cinco objetos: um blob para o conteúdo de cada um dos três arquivos, uma árvore que lista o conteúdo do diretório e especifica quais nomes de arquivos são armazenados em quais blobs, e um commit com o ponteiro para a raiz dessa árvore com todos os metadados do commit. Conceitualmente, os dados em seu repositório Git se parecem como na Figura 3-1.

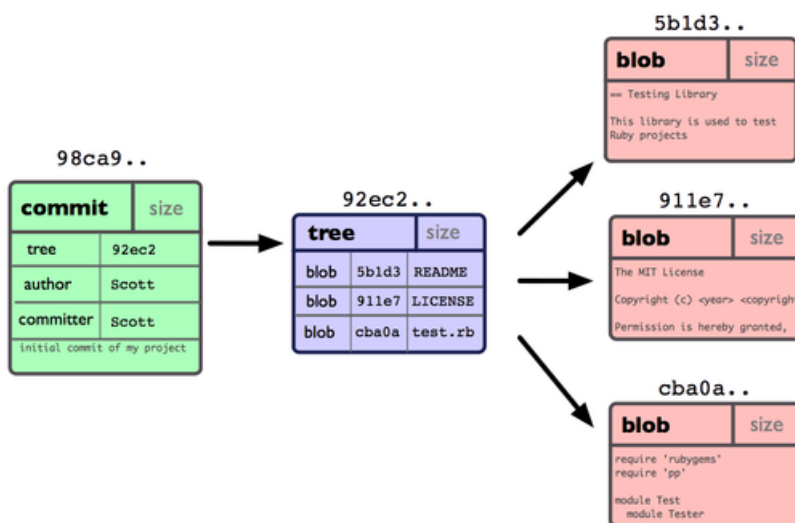


Figura 3-1. Dados de um repositório com um único commit.

Um branch no Git é simplesmente um leve ponteiro móvel para um desses commits. O nome do branch padrão no Git é `master`. Como você inicialmente fez commits, você tem um branch principal (`master branch`) que aponta para o último commit que você fez. Cada vez que você faz um commit ele avança automaticamente.

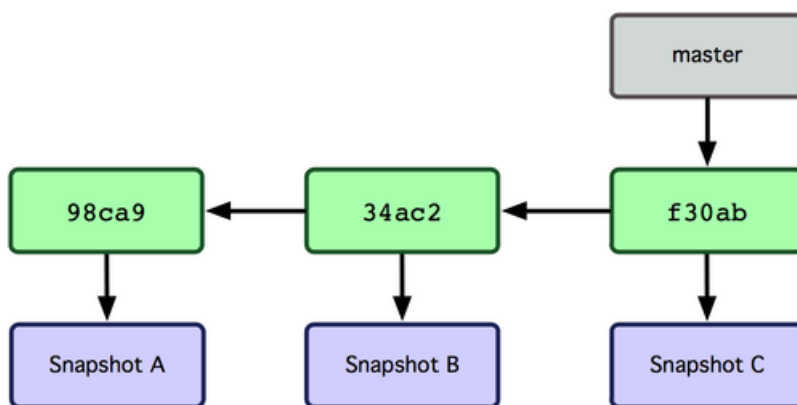


Figura 3-3. Branch apontando para o histórico de commits.

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

```
1 $ git branch testing
```

Isso cria um novo ponteiro para o mesmo commit em que você está no momento (ver a Figura 3-4).

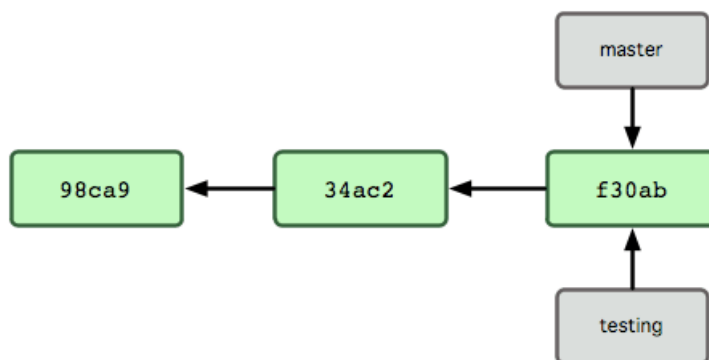


Figura 3-4. Múltiplos branches apontando para o histórico de commits.

Como o Git sabe o branch em que você está atualmente? Ele mantém um ponteiro especial chamado HEAD. Observe que isso é muito diferente do conceito de HEAD em outros VCSs que você possa ter usado, como Subversion e CVS. No Git, este é um ponteiro para o branch local em que você está no momento. Neste caso, você ainda está no master. O comando `git branch` só criou um novo branch — ele não mudou para esse branch (veja Figura 3-5).

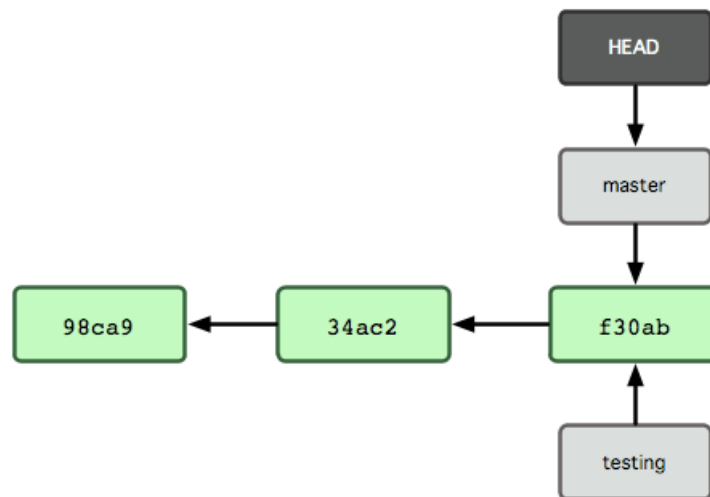


Figura 3-5. HEAD apontando para o branch em que você está.

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

```
1 $ git checkout testing
```

Isto move o HEAD para apontar para o branch de testes (ver Figura 3-6).

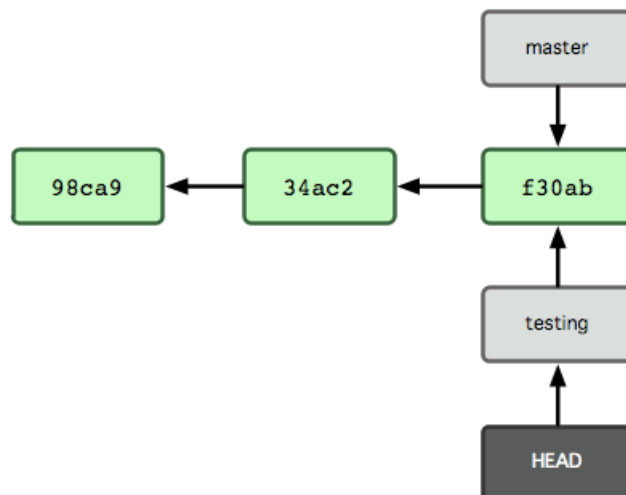


Figura 3-6. O HEAD aponta para outro branch quando você troca de branches.

Qual é o significado disso? Bem, vamos fazer um outro commit:

```
1 $ vim test.rb
2 $ git commit -a -m 'fiz uma alteração'
```

A figura 3-7 ilustra o resultado.

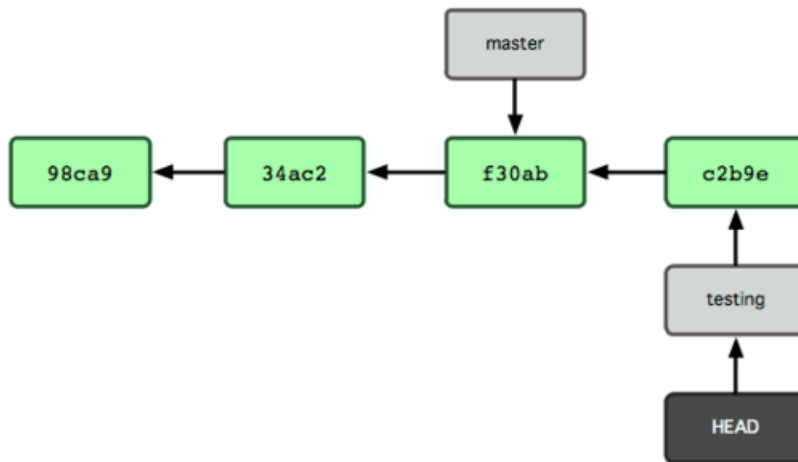


Figura 3-7. O branch para o qual `HEAD` aponta avança com cada commit.

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

```
1 $ git checkout master
```

A figura 3-8 mostra o resultado.

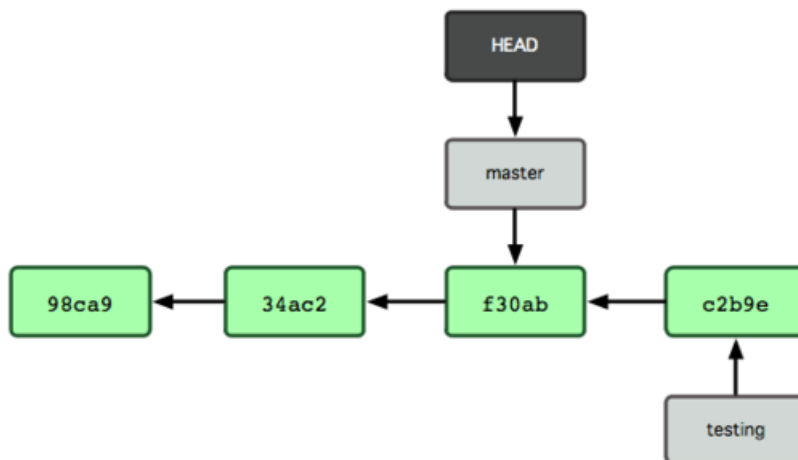


Figura 3-8. O `HEAD` se move para outro branch com um checkout.

Esse comando fez duas coisas. Ele alterou o ponteiro `HEAD` para apontar novamente para o branch `master`, e reverteu os arquivos em seu diretório de trabalho para o estado em que estavam no snapshot para o qual o `master` apontava. Isto significa também que as mudanças feitas a partir deste ponto em diante, irão divergir de uma versão anterior do projeto. Ele essencialmente “volta” o

trabalho que você fez no seu branch `testing`, temporariamente, de modo que você possa ir em uma direção diferente.

Vamos fazer algumas mudanças e fazer o commit novamente:

```
1 $ vim test.rb
2 $ git commit -a -m 'fiz outra alteração'
```

Agora o histórico do seu projeto divergiu (ver Figura 3-9). Você criou e trocou para um branch, trabalhou nele, e então voltou para o seu branch principal e trabalhou mais. Ambas as mudanças são isoladas em branches distintos: você pode alternar entre os branches e fundi-los (merge) quando estiver pronto. E você fez tudo isso simplesmente com os comandos `branch` e `checkout`.

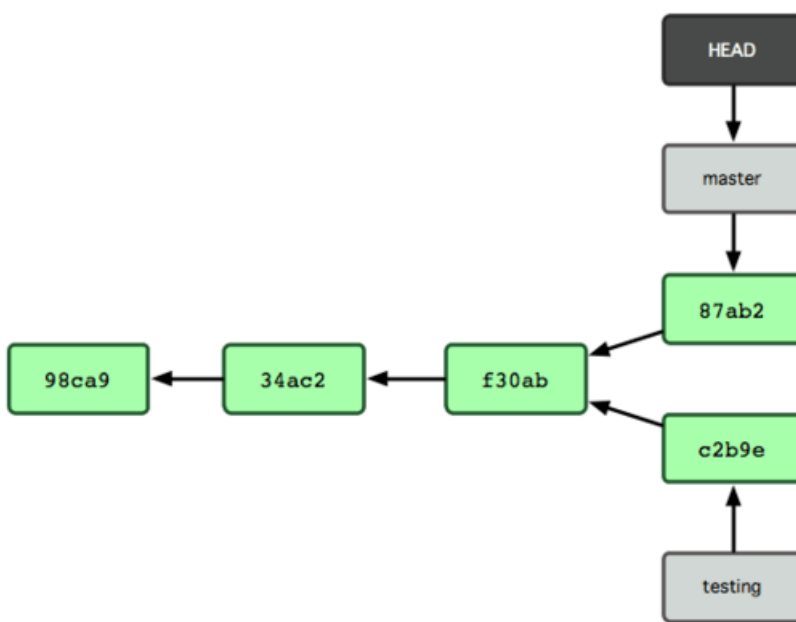


Figura 3-9. O histórico dos branches diverge.

Como um branch em Git é na verdade um arquivo simples que contém os 40 caracteres do checksum SHA-1 do commit para o qual ele aponta, os branches são baratos para criar e destruir. Criar um novo branch é tão rápido e simples como escrever 41 bytes em um arquivo (40 caracteres e uma quebra de linha).

Isto está em nítido contraste com a forma com a qual a maioria das ferramentas VCS gerenciam branches, que envolve a cópia de todos os arquivos do projeto para um segundo diretório. Isso pode demorar vários segundos ou até minutos, dependendo do tamanho do projeto, enquanto que no Git o processo é sempre instantâneo. Também, porque nós estamos gravando os pais dos objetos quando fazemos commits, encontrar uma boa base para fazer o merge é uma tarefa feita automaticamente para nós e geralmente é muito fácil de fazer. Esses recursos ajudam a estimular os desenvolvedores a criar e utilizar branches com frequência.

Vamos ver por que você deve fazê-lo.

3.2 Ramificação (Branching) no Git - Básico de Branch e Merge

Básico de Branch e Merge

Vamos ver um exemplo simples de uso de branch e merge com um fluxo de trabalho que você pode usar no mundo real. Você seguirá esses passos:

1. Trabalhar em um web site.
2. Criar um branch para uma nova história em que está trabalhando.
3. Trabalhar nesse branch.

Nesse etapa, você receberá um telefonema informando que outro problema crítico existe e precisa de correção. Você fará o seguinte:

1. Voltar ao seu branch de produção.
2. Criar um branch para adicionar a correção.
3. Depois de testado, fazer o merge do branch da correção, e enviar para produção.
4. Retornar à sua história anterior e continuar trabalhando.

Branch Básico

Primeiro, digamos que você esteja trabalhando no seu projeto e já tem alguns commits (veja Figura 3-10).

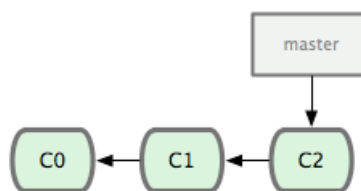


Figura 3-10. Um histórico de commits pequeno e simples.

Você decidiu que irá trabalhar na tarefa (issue) #53 do gerenciador de bugs ou tarefas que sua empresa usa. Para deixar claro, Git não é amarrado a nenhum gerenciador de tarefas em particular; mas já que a tarefa #53 tem um foco diferente, você criará um branch novo para trabalhar nele. Para criar um branch e mudar para ele ao mesmo tempo, você pode executar o comando `git checkout` com a opção `-b`:

```
1 $ git checkout -b iss53
2 Switched to a new branch "iss53"
```

Isso é um atalho para:

```
1 $ git branch iss53
2 $ git checkout iss53
```

Figura 3-11 ilustra o resultado.

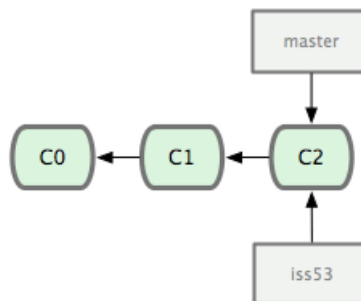


Figura 3-11. Criando um branch novo

Você trabalha no seu web site e faz alguns commits. Ao fazer isso o branch `iss53` avançará, pois você fez o checkout dele (isto é, seu HEAD está apontando para ele; veja a Figura 3-12):

```
1 $ vim index.html
2 $ git commit -a -m 'adicionei um novo rodapé [issue 53]'
```

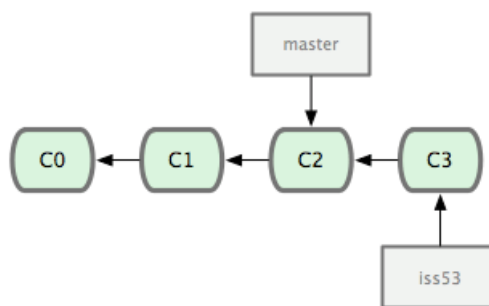


Figura 3-12. O branch `iss53` avançou com suas modificações.

Nesse momento você recebe uma ligação dizendo que existe um problema com o web site e você deve resolvê-lo imediatamente. Com Git, você não precisa fazer o deploy de sua correção junto com as modificações que você fez no `iss53`, e você não precisa se esforçar muito para reverter essas modificações antes que você possa aplicar sua correção em produção. Tudo que você tem a fazer é voltar ao seu branch `master`.

No entanto, antes de fazer isso, note que seu diretório de trabalho ou área de seleção tem modificações que não entraram em commits e que estão gerando conflitos com o branch que você está fazendo o checkout, Git não deixará você mudar de branch. É melhor ter uma área de trabalho limpa quando mudar de branch. Existem maneiras de contornar esta situação (isto é, incluir e fazer o commit) que vamos falar depois. Por enquanto, você fez o commit de todas as suas modificações, então você pode mudar para o seu branch master:

```
1 $ git checkout master
2 Switched to branch "master"
```

Nesse ponto, o diretório do seu projeto está exatamente do jeito que estava antes de você começar a trabalhar na tarefa #53, e você se concentra na correção do erro. É importante lembrar desse ponto: Git restabelece seu diretório de trabalho para ficar igual ao snapshot do commit que o branch que você criou aponta. Ele adiciona, remove, e modifica arquivos automaticamente para garantir que sua cópia é o que o branch parecia no seu último commit nele.

Em seguida, você tem uma correção para fazer. Vamos criar um branch para a correção (hotfix) para trabalhar até a conclusão (veja Figura 3-13):

```
1 $ git checkout -b 'hotfix'
2 Switched to a new branch "hotfix"
3 $ vim index.html
4 $ git commit -a -m 'concordei o endereço de email'
5 [hotfix]: created 3a0874c: "concordei o endereço de email"
6 1 files changed, 0 insertions(+), 1 deletions(-)
```

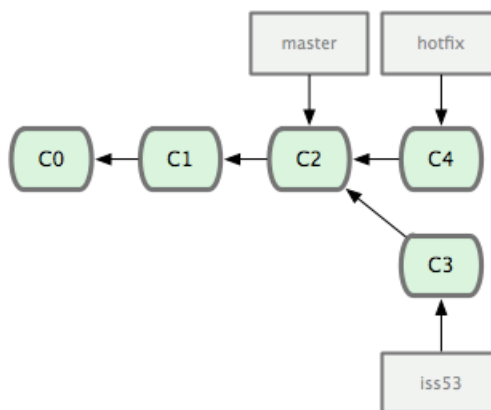


Figura 3-13. branch de correção (hotfix) baseado num ponto de seu branch master.

Você pode rodar seus testes, tenha certeza que a correção é o que você quer, e faça o merge no seu branch master para fazer o deploy em produção. Você faz isso com o comando `git merge`:

```
1 $ git checkout master
2 $ git merge hotfix
3 Updating f42c576..3a0874c
4 Fast forward
5  README |      1 -
6  1 files changed, 0 insertions(+), 1 deletions(-)
```

Você irá notar a frase “Fast forward” no merge. Em razão do branch que você fez o merge apontar para o commit que está diretamente acima do commit que você se encontra, Git avança o ponteiro adiante. Em outras palavras, quando você tenta fazer o merge de um commit com outro que pode ser alcançado seguindo o histórico do primeiro, Git simplifica as coisas movendo o ponteiro adiante porque não existe modificações divergente para fazer o merge — isso é chamado de “fast forward”.

Sua modificação está agora no snapshot do commit apontado pelo branch `master`, e você pode fazer o deploy (veja Figura 3-14).

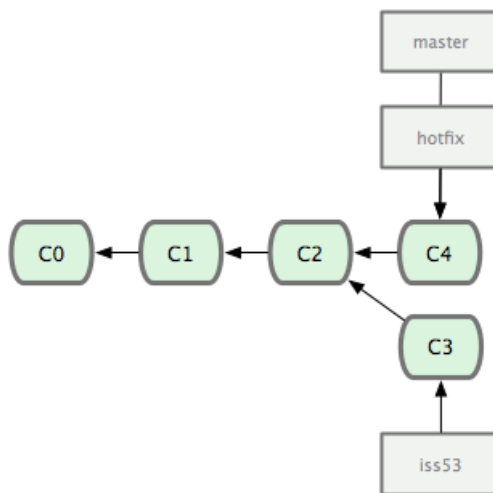


Figura 3-14. Depois do merge seu branch `master` aponta para o mesmo local que o branch `hotfix`.

Depois que a sua correção super importante foi enviada, você está pronto para voltar ao trabalho que estava fazendo antes de ser interrompido. No entanto, primeiro você apagará o branch `hotfix`, pois você não precisa mais dele — o branch `master` aponta para o mesmo local. Você pode excluí-lo com a opção `-d` em `git branch`:

```
1 $ git branch -d hotfix
2 Deleted branch hotfix (3a0874c).
```

Agora você pode voltar para o trabalho incompleto no branch da tafera #53 e continuar a trabalhar nele (veja Figura 3-15):

```

1 $ git checkout iss53
2 Switched to branch "iss53"
3 $ vim index.html
4 $ git commit -a -m 'novo rodapé terminado [issue 53]'
5 [iss53]: created ad82d7a: "novo rodapé terminado [issue 53]"
6 1 files changed, 1 insertions(+), 0 deletions(-)

```

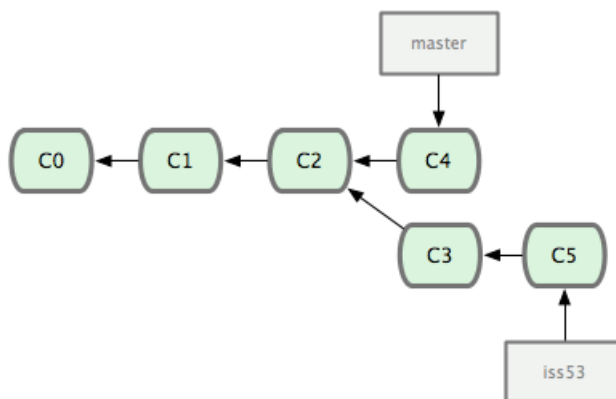


Figura 3-15. Seu branch `iss53` pode avançar de forma independente.

Vale a pena lembrar aqui que o trabalho feito no seu branch `hotfix` não existe nos arquivos do seu branch `iss53`. Se você precisa incluí-lo, você pode fazer o merge do seu branch `master` no seu branch `iss53` executando o comando `git merge master`, ou você pode esperar para integrar essas mudanças até você decidir fazer o pull do branch `iss53` no `master` mais tarde.

Merge Básico

Suponha que você decidiu que o trabalho na tarefa #53 está completo e pronto para ser feito o merge no branch `master`. Para fazer isso, você fará o merge do seu branch `iss53`, bem como o merge do branch `hotfix` de antes. Tudo que você tem a fazer é executar o checkout do branch para onde deseja fazer o merge e então rodar o comando `git merge`:

```

1 $ git checkout master
2 $ git merge iss53
3 Merge made by recursive.
4  README | 1 +
5  1 files changed, 1 insertions(+), 0 deletions(-)

```

Isso parece um pouco diferente do merge de `hotfix` que você fez antes. Neste caso, o histórico do seu desenvolvimento divergiu em algum ponto anterior. Pelo fato do commit no branch em que você está não ser um ancestral direto do branch que você está fazendo o merge, Git tem um trabalho adicional. Neste caso, Git faz um merge simples de três vias, usando os dois snapshots apontados

pelas pontas dos branches e o ancestral comum dos dois. A Figura 3-16 destaca os três snapshots que Git usa para fazer o merge nesse caso.

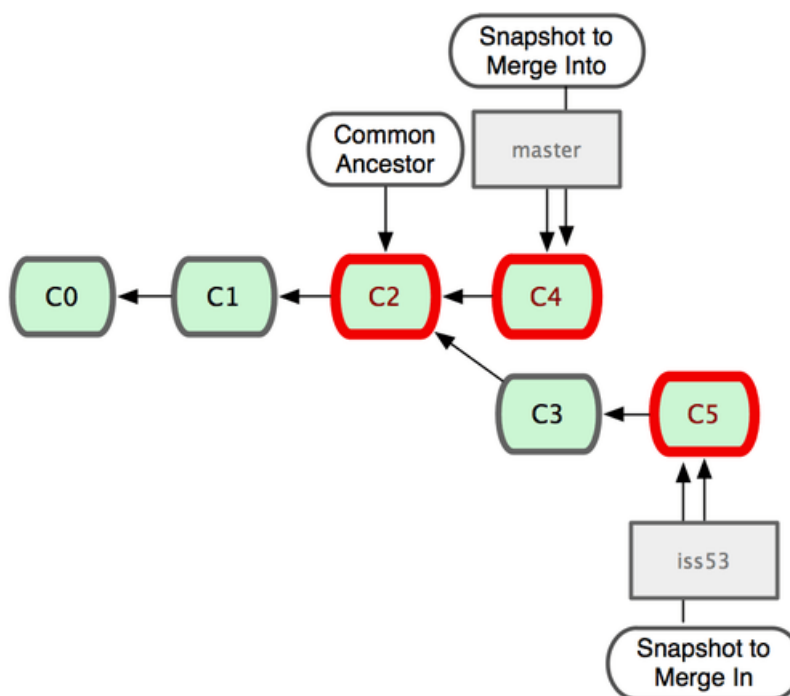


Figura 3-16. Git identifica automaticamente a melhor base ancestral comum para o merge do branch.

Em vez de simplesmente avançar o ponteiro do branch adiante, Git cria um novo snapshot que resulta do merge de três vias e automaticamente cria um novo commit que aponta para ele (veja Figura 3-17). Isso é conhecido como um merge de commits e é especial pois tem mais de um pai.

Vale a pena destacar que o Git determina o melhor ancestral comum para usar como base para o merge; isso é diferente no CVS ou Subversion (antes da versão 1.5), onde o desenvolvedor que está fazendo o merge tem que descobrir a melhor base para o merge por si próprio. Isso faz o merge muito mais fácil no Git do que nesses outros sistemas.

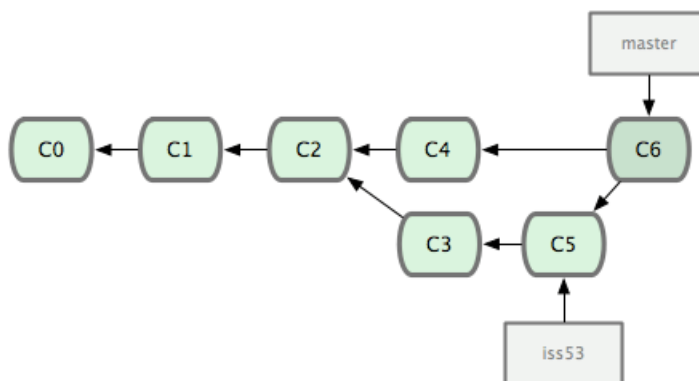


Figura 3-17. Git cria automaticamente um novo objeto commit que contém as modificações do merge.

Agora que foi feito o merge no seu trabalho, você não precisa mais do branch `iss53`. Você pode apagá-lo e fechar manualmente o chamado no seu gerenciador de chamados:

```
1 $ git branch -d iss53
```

Conflitos de Merge Básico

Às vezes, esse processo não funciona sem problemas. Se você alterou a mesma parte do mesmo arquivo de forma diferente nos dois branches que está fazendo o merge, Git não será capaz de executar o merge de forma clara. Se sua correção para o erro #53 alterou a mesma parte de um arquivo que `hotfix`, você terá um conflito de merge parecido com isso:

```
1 $ git merge iss53
2 Auto-merging index.html
3 CONFLICT (content): Merge conflict in index.html
4 Automatic merge failed; fix conflicts and then commit the result.
```

Git não criou automaticamente um novo commit para o merge. Ele fez uma pausa no processo enquanto você resolve o conflito. Se você quer ver em quais arquivos não foi feito o merge, em qualquer momento depois do conflito, você pode executar `git status`:

```
1 [master*]$ git status
2 index.html: needs merge
3 # On branch master
4 # Changes not staged for commit:
5 #   (use "git add <file>..." to update what will be committed)
6 #   (use "git checkout -- <file>..." to discard changes in working directory)
7 #
8 #    unmerged:   index.html
9 #
```

Qualquer coisa que tem conflito no merge e não foi resolvido é listado como “unmerged”. Git adiciona marcadores padrão de resolução de conflitos nos arquivos que têm conflitos, para que você possa abri-los manualmente e resolver esses conflitos. Seu arquivo terá uma seção parecida com isso:

```

1 <<<<<< HEAD:index.html
2 <div id="footer">contato : email.support@github.com</div>
3 =====
4 <div id="footer">
5     por favor nos contate em support@github.com
6 </div>
7 >>>>>> iss53:index.html

```

Isso significa que a versão em HEAD (seu branch master, pois era isso que você tinha quando executou o comando merge) é a parte superior desse bloco (tudo acima de =====), enquanto a versão no seu branch iss53 é toda a parte inferior. Para resolver esse conflito, você tem que optar entre um lado ou outro, ou fazer o merge do conteúdo você mesmo. Por exemplo, você pode resolver esse conflito através da substituição do bloco inteiro por isso:

```

1 <div id="footer">
2 por favor nos contate em email.support@github.com
3 </div>

```

Esta solução tem um pouco de cada seção, e eu removi completamente as linhas <<<<<<, =====, e >>>>>>. Depois que você resolveu cada uma dessas seções em cada arquivo com conflito, rode git add em cada arquivo para marcá-lo como resolvido. Colocar o arquivo na área de seleção o marcar como resolvido no Git. Se você quer usar uma ferramenta gráfica para resolver esses problemas, você pode executar git mergetool, que abre uma ferramenta visual de merge adequada e percorre os conflitos:

```

1 $ git mergetool
2 merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
3 Merging the files: index.html
4
5 Normal merge conflict for 'index.html':
6   {local}: modified
7   {remote}: modified
8 Hit return to start merge resolution tool (opendiff):

```

Se você quer usar uma ferramenta de merge diferente da padrão (Git escolheu opendiff para mim, neste caso, porque eu rodei o comando em um Mac), você pode ver todas as ferramentas disponíveis listadas no topo depois de “merge tool candidates”. Digite o nome da ferramenta que você prefere usar. No *capítulo 7*, discutiremos como você pode alterar esse valor para o seu ambiente.

Depois de sair da ferramenta de merge, Git pergunta se o merge foi concluído com sucesso. Se você disser ao script que foi, ele coloca o arquivo na área de seleção para marcá-lo como resolvido pra você.

Se você rodar git status novamente para verificar que todos os conflitos foram resolvidos:


```
1 $ git status
2 # On branch master
3 # Changes to be committed:
4 #   (use "git reset HEAD <file>..." to unstage)
5 #
6 #       modified:   index.html
7 #
```

Se você está satisfeito com isso, e verificou que tudo que havia conflito foi colocado na área de seleção, você pode digitar `git commit` para concluir o commit do merge. A mensagem de commit padrão é algo semelhante a isso:

```
1 Merge branch 'iss53'
2
3 Conflicts:
4   index.html
5 #
6 # It looks like you may be committing a MERGE.
7 # If this is not correct, please remove the file
8 # .git/MERGE_HEAD
9 # and try again.
10 #
```

Você pode modificar essa mensagem com detalhes sobre como você resolveu o merge se você acha que isso seria útil para outros quando olharem esse merge no futuro — por que você fez o que fez, se não é óbvio.

3.3 Ramificação (Branching) no Git - Gerenciamento de Branches

Gerenciamento de Branches

Agora que você criou, fez merge e apagou alguns branches, vamos ver algumas ferramentas de gerenciamento de branches que serão úteis quando você começar a usá-los o tempo todo.

O comando `git branch` faz mais do que criar e apagar branches. Se você executá-lo sem argumentos, você verá uma lista simples dos seus branches atuais:

```
1 $ git branch
2   iss53
3  * master
4   testing
```

Note o caractere `*` que vem antes do branch `master`: ele indica o branch que você está atualmente (fez o checkout). Isso significa que se você fizer um commit nesse momento, o branch `master` irá se mover adiante com seu novo trabalho. Para ver o último commit em cada branch, você pode executar o comando `git branch -v`:

```
1 $ git branch -v
2   iss53    93b412c concertar problema em javascript
3  * master  7a98805 Merge branch 'iss53'
4   testing  782fd34 adicionar scott para a lista de autores nos readmes
```

Outra opção útil para saber em que estado estão seus branches é filtrar na lista somente branches que você já fez ou não o merge no branch que você está atualmente. As opções `--merged` e `--no-merged` estão disponíveis no Git desde a versão 1.5.6 para esse propósito. Para ver quais branches já foram mesclados no branch que você está, você pode executar `git branch --merged`:

```
1 $ git branch --merged
2   iss53
3  * master
```

Por você já ter feito o merge do branch `iss53` antes, você o verá na sua lista. Os branches nesta lista sem o `*` na frente em geral podem ser apagados com `git branch -d`; você já incorporou o trabalho que existia neles em outro branch, sendo assim você não perderá nada.

Para ver todos os branches que contém trabalho que você ainda não fez o merge, você pode executar `git branch --no-merged`:

```
1 $ git branch --no-merged
2   testing
```

Isso mostra seu outro branch. Por ele conter trabalho que ainda não foi feito o merge, tentar apagá-lo com `git branch -d` irá falhar:

```
1 $ git branch -d testing
2 error: The branch 'testing' is not an ancestor of your current HEAD.
3 If you are sure you want to delete it, run `git branch -D testing`.
```

Se você quer realmente apagar o branch e perder o trabalho que existe nele, você pode forçar com `-D`, como a útil mensagem aponta.

3.4 Ramificação (Branching) no Git - Fluxos de Trabalho com Branches

Fluxos de Trabalho com Branches

Agora que você sabe o básico sobre criação e merge de branches, o que você pode ou deve fazer com eles? Nessa seção, nós vamos abordar alguns fluxos de trabalhos comuns que esse tipo de criação fácil de branches torna possível, então você pode decidir se você quer incorporá-lo no seu próprio ciclo de desenvolvimento.

Branches de Longa Duração Devido ao Git usar um merge de três vias, fazer o merge de um branch em outro várias vezes em um período longo é geralmente fácil de fazer. Isto significa que você pode ter vários branches que ficam sempre abertos e que são usados em diferentes estágios do seu ciclo de desenvolvimento; você pode regularmente fazer o merge de alguns deles em outros.

Muitos desenvolvedores Git tem um fluxo de trabalho que adotam essa abordagem, como ter somente código completamente estável em seus branches master — possivelmente somente código que já foi ou será liberado. Eles têm outro branch paralelo chamado develop ou algo parecido em que eles trabalham ou usam para testar estabilidade — ele não é necessariamente sempre estável, mas quando ele chega a tal estágio, pode ser feito o merge com o branch master. Ele é usado para puxar (pull) branches tópicos (topic, branches de curta duração, como o seu branch `iss53` anteriormente) quando eles estão prontos, para ter certeza que eles passam em todos os testes e não acrescentam erros.

Na realidade, nós estamos falando de ponteiros avançando na linha de commits que você está fazendo. Os branches estáveis estão muito atrás na linha histórica de commits, e os branches de ponta (que estão sendo trabalhados) estão a frente no histórico.

Normalmente é mais fácil pensar neles como um contêiner de trabalho, onde conjuntos de commits são promovidos a um contêiner mais estável quando eles são completamente testados (veja figura 3-19).

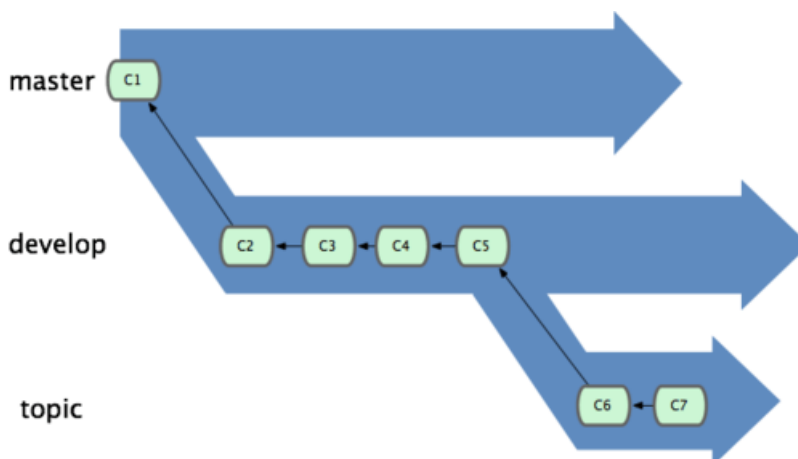


Figura 3-19. Pode ser mais útil pensar em seus branches como contêineres.

Você pode continuar fazendo isso em vários níveis de estabilidade. Alguns projetos grandes podem ter um branch ‘sugerido’ (proposed) ou ‘sugestões atualizadas’ (pu, proposed updates) que contém outros branches integrados que podem não estar prontos para ir para o próximo (next) ou branch master. A ideia é que seus branches estejam em vários níveis de estabilidade; quando eles atingem um nível mais estável, é feito o merge no branch acima deles. Repetindo, ter muitos branches de longa duração não é necessário, mas geralmente é útil, especialmente quando você está lidando com projetos muito grandes ou complexos.

Branches Tópicos (topic)

Branches tópicos, entretanto, são úteis em projetos de qualquer tamanho. Um branch tópico é um branch de curta duração que você cria e usa para uma funcionalidade ou trabalho relacionado. Isso é algo que você provavelmente nunca fez com um controle de versão antes porque é geralmente muito custoso criar e fazer merge de branches. Mas no Git é comum criar, trabalhar, mesclar e apagar branches muitas vezes ao dia.

Você viu isso na seção anterior com os branches `iss53` e `hotfix` que você criou. Você fez commits neles e os apagou depois que fez o merge com seu branch principal. Tecnicamente, isso lhe permite mudar completamente e rapidamente o contexto — em razão de seu trabalho estar separado em contêineres onde todas as modificações naquele branch estarem relacionadas ao tópico, é fácil ver o que aconteceu durante a revisão de código. Você pode manter as mudanças lá por minutos, dias, ou meses, e mesclá-las quando estiverem prontas, não importando a ordem que foram criadas ou trabalhadas.

Considere um exemplo onde você está fazendo um trabalho (no `master`), cria um branch para um erro (`iss91`), trabalha nele um pouco, cria um segundo branch para testar uma nova maneira de resolver o mesmo problema (`iss91v2`), volta ao seu branch principal e trabalha nele por um tempo, e cria um novo branch para trabalhar em algo que você não tem certeza se é uma boa ideia (`dumbidea`). Seu histórico de commits irá se parecer com a Figura 3-20.

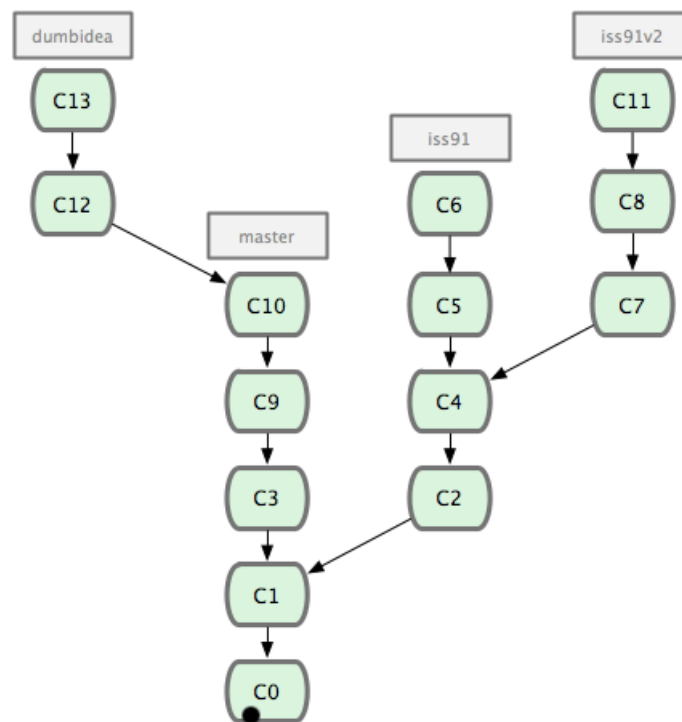


Figura 3-20. Seu histórico de commits com múltiplos branches tópicos.

Agora, vamos dizer que você decidiu que sua segunda solução é a melhor para resolver o erro (iss91v2); e você mostrou seu branch `dumbidea` para seus colegas de trabalho, e ele é genial. Agora você pode jogar fora o branch original `iss91` (perdendo os commits C5 e C6) e fazer o merge dos dois restantes. Seu histórico irá se parecer com a Figura 3-21.

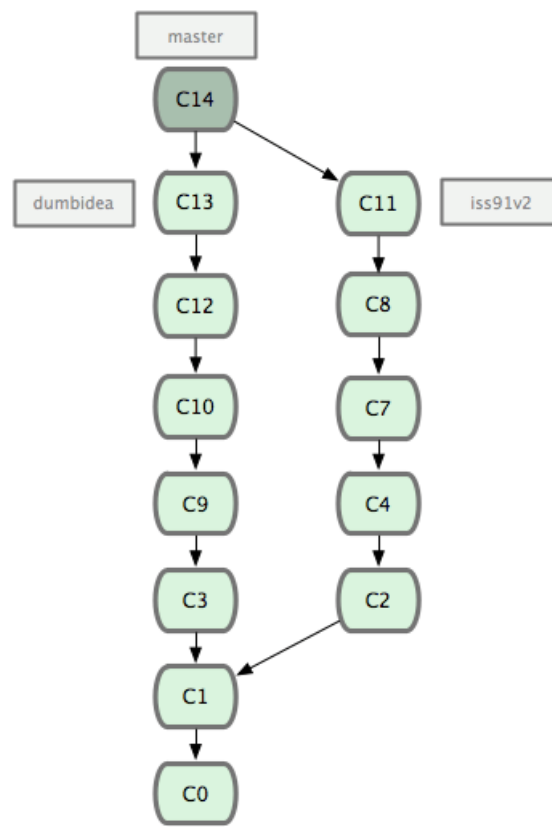


Figura 3-21. Seu histórico depois de fazer o merge de dumbidea e iss91v2.

É importante lembrar que você está fazendo tudo isso com seus branches localmente. Quando você cria e faz o merge de branches, tudo está sendo feito somente no seu repositório Git — nenhuma comunicação com o servidor está sendo feita.

3.5 Ramificação (Branching) no Git - Branches Remotos

Branches Remotos

Branches remotos são referências ao estado de seus branches no seu repositório remoto. São branches locais que você não pode mover, eles se movem automaticamente sempre que você faz alguma comunicação via rede. Branches remotos agem como marcadores para lembrá-lo onde estavam seus branches no seu repositório remoto na última vez que você se conectou a eles.

Eles seguem o padrão (remote)/(branch). Por exemplo, se você quer ver como o branch `master` estava no seu repositório remoto `origin` na última vez que você se comunicou com ele, você deve ver o branch `origin/master`. Se você estivesse trabalhando em um problema com um colega e eles colocassem o branch `iss53` no repositório, você poderia ter seu próprio branch `iss53`; mas o branch no servidor iria fazer referência ao commit em `origin/iss53`.

Isso pode parecer um pouco confuso, então vamos ver um exemplo. Digamos que você tem um servidor Git na sua rede em `git.ourcompany.com`. Se você cloná-lo, Git automaticamente dá o nome `origin` para ele, baixa todo o seu conteúdo, cria uma referência para onde o branch `master` dele está, e dá o nome `origin/master` para ele localmente; e você não pode movê-lo. O Git também dá seu próprio branch `master` como ponto de partida no mesmo local onde o branch `master` remoto está, a partir de onde você pode trabalhar (veja Figura 3-22).

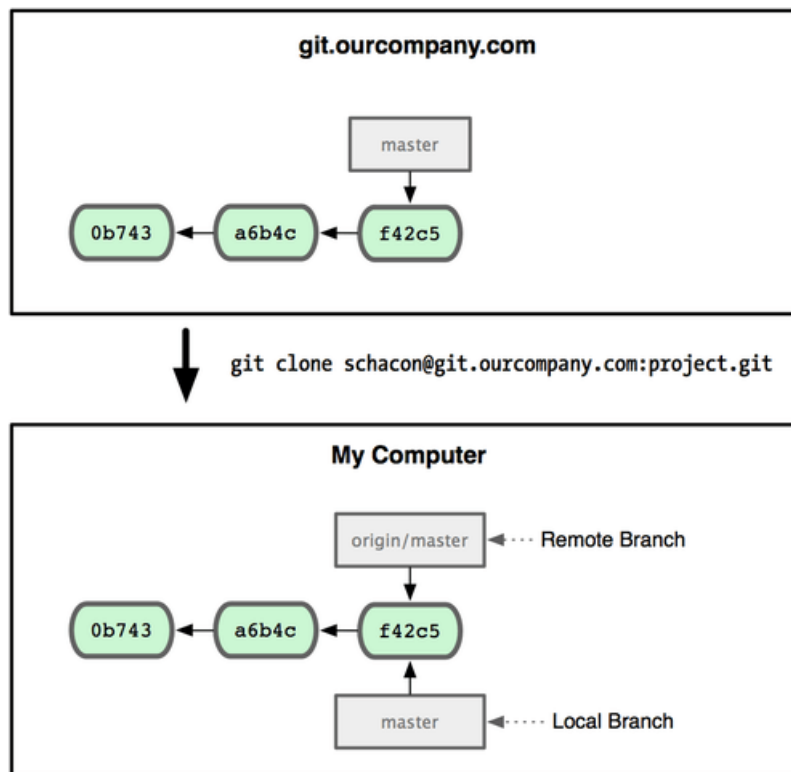


Figura 3-22. Um comando `clone` do Git dá a você seu próprio branch `master` e `origin/master` faz referência ao branch `master` original.

Se você estiver trabalhando no seu branch `master` local, e, ao mesmo tempo, alguém envia algo para `git.ourcompany.com` atualizando o branch `master`, seu histórico avançará de forma diferente. Além disso, enquanto você não fizer contato com seu servidor original, seu `origin/master` não se moverá (veja Figura 3-23).

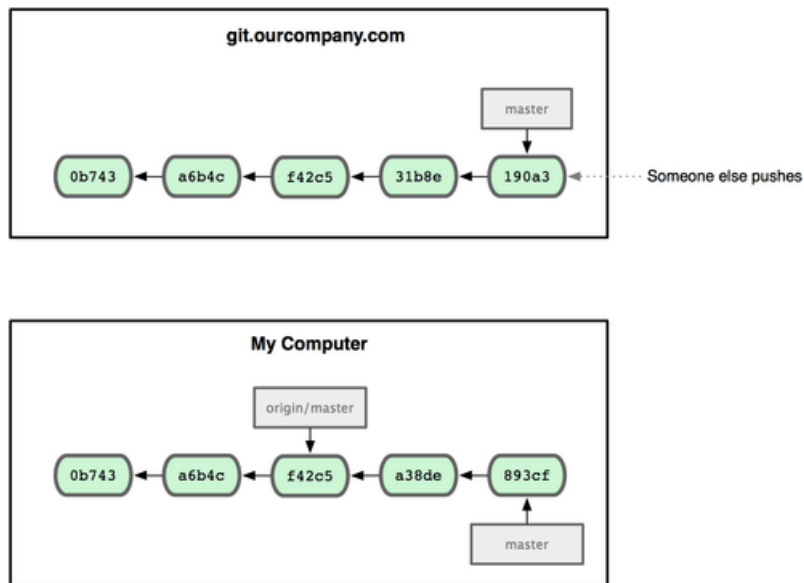


Figura 3-23. Ao trabalhar local e alguém enviar coisas para seu servidor remoto faz cada histórico avançar de forma diferente.

Para sincronizar suas coisas, você executa o comando `git fetch origin`. Esse comando verifica qual servidor “origin” representa (nesse caso, é `git.ourcompany.com`), obtém todos os dados que você ainda não tem e atualiza o seu banco de dados local, movendo o seu `origin/master` para a posição mais recente e atualizada (veja Figura 3-24).

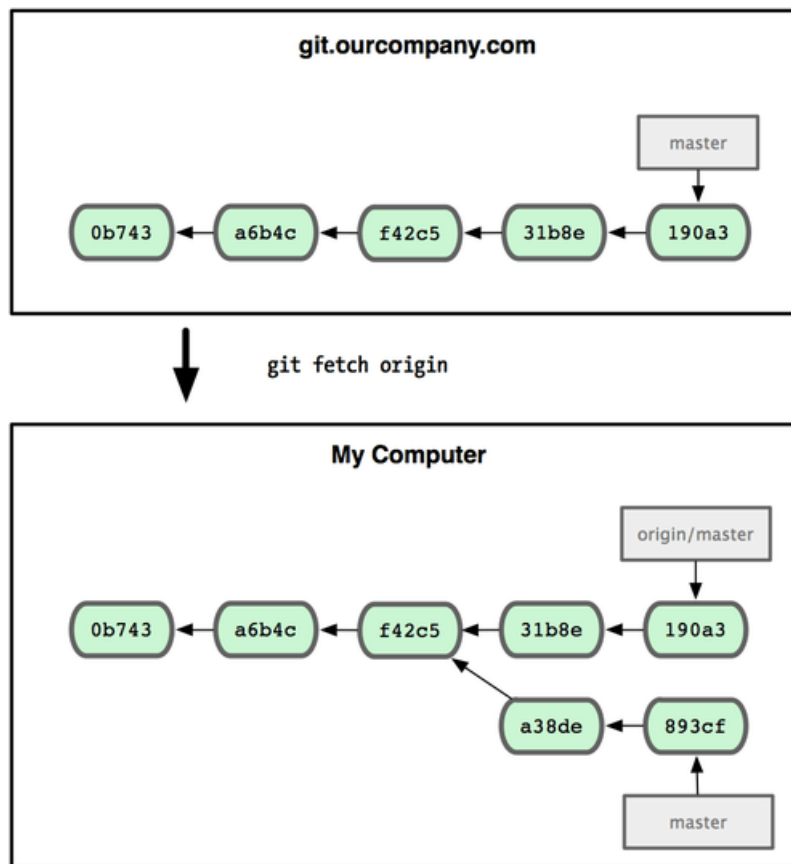


Figura 3-24. O comando `git fetch` atualiza suas referências remotas.

Para demonstrar o uso de múltiplos servidores remotos e como os branches remotos desses projetos remotos parecem, vamos assumir que você tem outro servidor Git interno que é usado somente para desenvolvimento por um de seus times. Este servidor está em `git.team1.ourcompany.com`. Você pode adicioná-lo como uma nova referência remota ao projeto que você está atualmente trabalhando executando o comando `git remote add` como discutimos no capítulo 2. Dê o nome de `teamone`, que será o apelido para aquela URL (veja Figura 3-25).

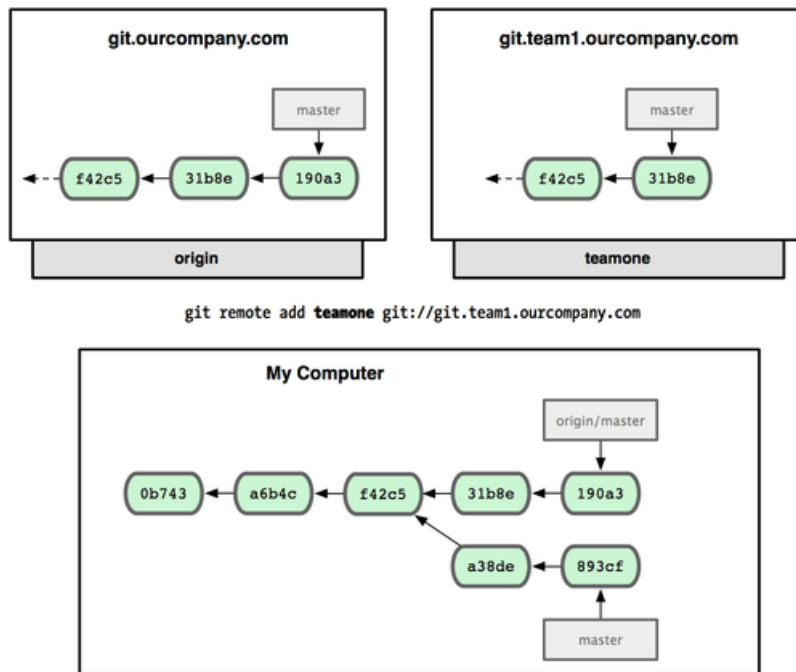


Figura 3-25. Adicionando outro servidor remoto.

Agora, você pode executar o comando `git fetch teamone` para obter tudo que o servidor teamone tem e você ainda não. Por esse servidor ter um subconjunto dos dados que seu servidor origin tem, Git não obtém nenhum dado, somente cria um branch chamado teamone/master que faz referência ao commit que teamone tem no master dele (veja Figura 3-26).

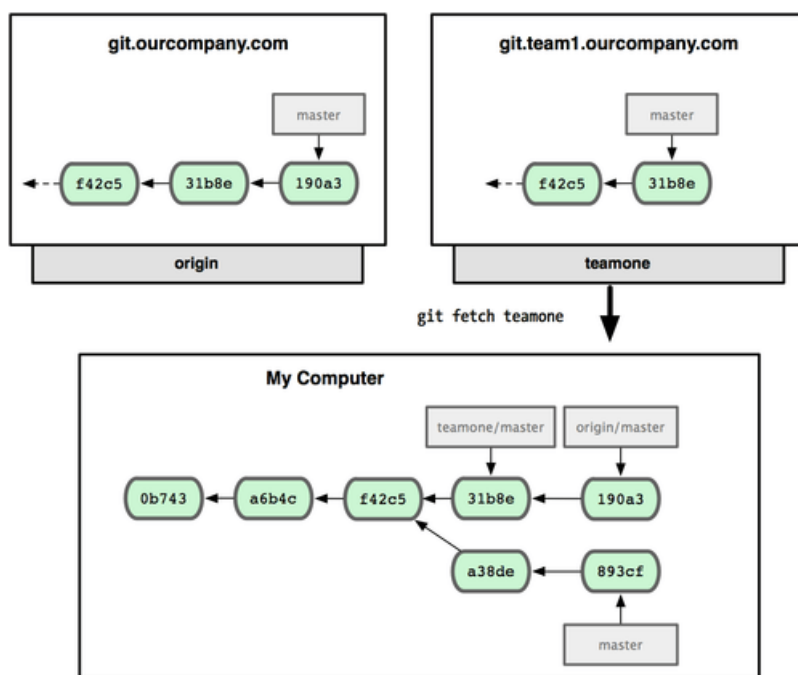


Figura 3-26. Você consegue uma referência local para a posição do branch master do teamone.

Enviando (Pushing)

Quando você quer compartilhar um branch com o mundo, você precisa enviá-lo a um servidor remoto que você tem acesso. Seus branches locais não são automaticamente sincronizados com os remotos — você tem que explicitamente enviar (push) os branches que quer compartilhar. Desta maneira, você pode usar branches privados para o trabalho que não quer compartilhar, e enviar somente os branches tópicos em que quer colaborar.

Se você tem um branch chamado `serverfix` e quer trabalhar com outros, você pode enviá-lo da mesma forma que enviou seu primeiro branch. Execute o comando `git push (remote) (branch)`:

```
1 $ git push origin serverfix
2 Counting objects: 20, done.
3 Compressing objects: 100% (14/14), done.
4 Writing objects: 100% (15/15), 1.74 KiB, done.
5 Total 15 (delta 5), reused 0 (delta 0)
6 To git@github.com:schacon/simplegit.git
7 * [new branch]      serverfix -> serverfix
```

Isso é um atalho. O Git automaticamente expande o branch `serverfix` para `refs/heads/serverfix:refs/heads/serverfix`, que quer dizer, “pegue meu branch local `serverfix` e envie para atualizar o branch `serverfix` no servidor remoto”. Nós vamos ver a parte de `refs/heads/` em detalhes no capítulo 9, mas em geral você pode deixar assim. Você pode executar também `git push origin serverfix:serverfix`,

que faz a mesma coisa — é como, “pegue meu serverfix e o transforme no serverfix remoto”. Você pode usar esse formato para enviar (push) um branch local para o branch remoto que tem nome diferente. Se você não quer chamá-lo 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.

Na próxima vez que um dos seus colaboradores obtiver dados do servidor, ele terá uma referência para onde a versão do servidor de `serverfix` está no branch remoto `origin/serverfix`:

```
1 $ git fetch origin
2 remote: Counting objects: 20, done.
3 remote: Compressing objects: 100% (14/14), done.
4 remote: Total 15 (delta 5), reused 0 (delta 0)
5 Unpacking objects: 100% (15/15), done.
6 From git@github.com:schacon/simplegit
7 * [new branch]      serverfix    -> origin/serverfix
```

É importante notar que quando você obtém dados que traz novos branches remotos, você não tem automaticamente cópias locais e editáveis. Em outras palavras, nesse caso, você não tem um novo branch `serverfix` — você tem somente uma referência a `origin/serverfix` que você não pode modificar.

Para fazer o merge desses dados no branch que você está trabalhando, você pode executar o comando `git merge origin/serverfix`. Se você quer seu próprio branch `serverfix` para trabalhar, você pode se basear no seu branch remoto:

```
1 $ git checkout -b serverfix origin/serverfix
2 Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
3 Switched to a new branch "serverfix"
```

Isso dá a você um branch local para trabalhar que começa onde `origin/serverfix` está.

Branches Rastreados (Tracking branches)

Baixar um branch local a partir de um branch remoto cria automaticamente o chamado tracking branch (branches rastreados). Tracking branches são branches locais que tem uma relação direta com um branch remoto. Se você está em um tracking branch e digita `git push`, Git automaticamente sabe para que servidor e branch deve fazer o envio (push). Além disso, ao executar o comando `git pull` em um desses branches, é obtido todos os dados remotos e é automaticamente feito o merge do branch remoto correspondente.

Quando você faz o clone de um repositório, é automaticamente criado um branch `master` que segue `origin/master`. Esse é o motivo pelo qual `git push` e `git pull` funcionam sem argumentos.

Entretanto, você pode criar outros tracking branches se quiser — outros que não seguem branches em origin e não seguem o branch master. Um caso simples é o exemplo que você acabou de ver, executando o comando `git checkout -b [branch] [nomeremoto]/[branch]`. Se você tem a versão do Git 1.6.2 ou mais recente, você pode usar também o atalho `--track`:

```
1 $ git checkout --track origin/serverfix
2 Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
3 Switched to a new branch "serverfix"
```

Para criar um branch local com um nome diferente do branch remoto, você pode facilmente usar a primeira versão com um nome diferente para o branch local:

```
1 $ git checkout -b sf origin/serverfix
2 Branch sf set up to track remote branch refs/remotes/origin/serverfix.
3 Switched to a new branch "sf"
```

Agora, seu branch local `sf` irá automaticamente enviar e obter dados de `origin/serverfix`.

Apagando Branches Remotos

Imagine que você não precise mais de um branch remoto — digamos, você e seus colaboradores acabaram uma funcionalidade e fizeram o merge no branch master remoto (ou qualquer que seja seu branch estável). Você pode apagar um branch remoto usando a sintaxe `git push [nomeremoto] :[branch]`. Se você quer apagar seu branch `serverfix` do servidor, você executa o comando:

```
1 $ git push origin :serverfix
2 To git@github.com:schacon/simplegit.git
3 - [deleted]          serverfix
```

Boom. O branch não existe mais no servidor. Talvez você queira marcar essa página, pois precisará desse comando, e provavelmente esquecerá a sintaxe. Uma maneira de lembrar desse comando é pensar na sintaxe de `git push [nomeremoto] [branchlocal]:[branchremoto]` que nós vimos antes. Se tirar a parte `[branchlocal]`, basicamente está dizendo, “Peque nada do meu lado e torne-o `[branchremoto]`.”

3.6 Ramificação (Branching) no Git - Rebasing

Rebasing

No Git, existem duas maneiras principais de integrar mudanças de um branch em outro: o merge e o rebase. Nessa seção você aprenderá o que é rebase, como fazê-lo, por que é uma ferramenta sensacional, e em quais casos você não deve usá-la.

O Rebase Básico

Se você voltar para o exemplo anterior na seção de merge (veja Figura 3-27), você pode ver que você criou uma divergência no seu trabalho e fez commits em dois branches diferentes.

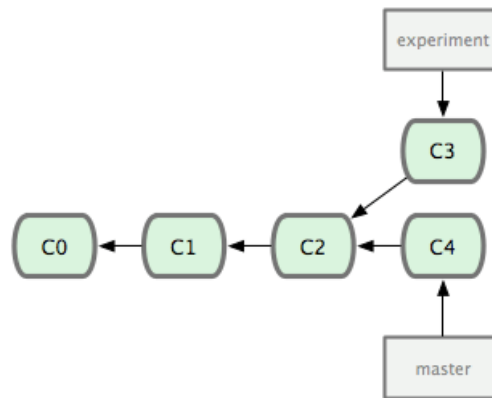


Figura 3-27. Divergência inicial no seu histórico de commits.

A maneira mais fácil de integrar os branches, como já falamos, é o comando `merge`. Ele executa um merge de três vias entre os dois últimos snapshots (cópias em um determinado ponto no tempo) dos branches (C3 e C4) e o mais recente ancestral comum aos dois (C2), criando um novo snapshot (e um commit), como é mostrado na Figura 3-28.

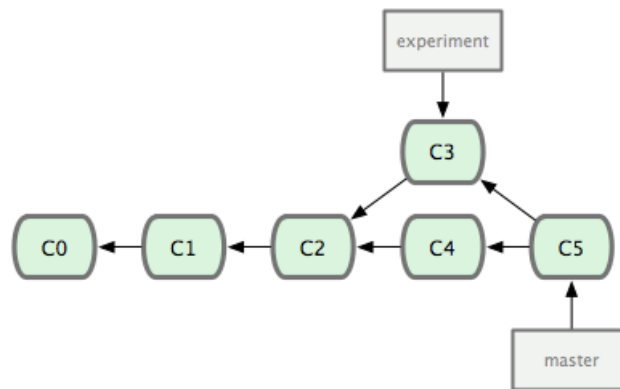


Figura 3-28. Fazendo o merge de um branch para integrar o trabalho divergente.

Porém, existe outro modo: você pode pegar o trecho da mudança que foi introduzido em C3 e reaplicá-lo em cima do C4. No Git, isso é chamado de rebasing. Com o comando `rebase`, você pode pegar todas as mudanças que foram commitadas em um branch e replicá-las em outro.

Nesse exemplo, se você executar o seguinte:

```

1 $ git checkout experiment
2 $ git rebase master
3 First, rewinding head to replay your work on top of it...
4 Applying: added staged command

```

Ele vai ao ancestral comum dos dois branches (no que você está e no qual será feito o rebase), pega a diferença (diff) de cada commit do branch que você está, salva elas em um arquivo temporário, restaura o branch atual para o mesmo commit do branch que está sendo feito o rebase e, finalmente, aplica uma mudança de cada vez. A Figura 3-29 ilustra esse processo.

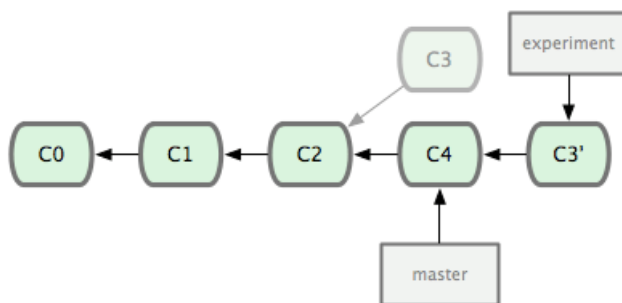


Figura 3-29. Fazendo o rebase em C4 de mudanças feitas em C3.

Nesse ponto, você pode ir ao branch master e fazer um merge fast-forward (Figura 3-30).

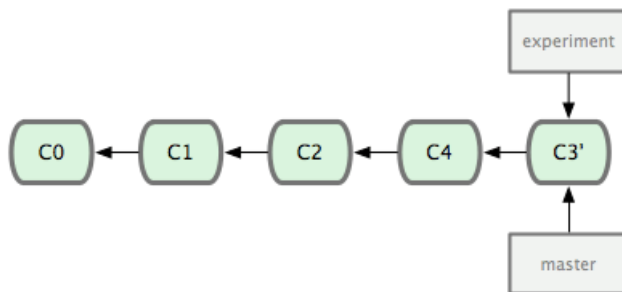


Figura 3-30. Fazendo um fast-forward no branch master.

Agora, o snapshot apontado por C3' é exatamente o mesmo apontado por C5 no exemplo do merge. Não há diferença no produto final dessas integrações, mas o rebase monta um histórico mais limpo. Se você examinar um log de um branch com rebase, ele parece um histórico linear: como se todo o trabalho tivesse sido feito em série, mesmo que originalmente tenha sido feito em paralelo.

Constantemente você fará isso para garantir que seus commits sejam feitos de forma limpa em um branch remoto — talvez em um projeto em que você está tentando contribuir mas não mantém. Nesse caso, você faz seu trabalho em um branch e então faz o rebase em origin/master quando está pronto pra enviar suas correções para o projeto principal. Desta maneira, o mantenedor não precisa fazer nenhum trabalho de integração — somente um merge ou uma inserção limpa.

Note que o snapshot apontado pelo o commit final, o último commit dos que vieram no rebase ou o último commit depois do merge, são o mesmo snapshot — somente o histórico é diferente. Fazer

o rebase reproduz mudanças de uma linha de trabalho para outra na ordem em que foram feitas, já que o merge pega os pontos e os une.

Rebases Mais Interessantes

Você também pode fazer o rebase em um local diferente do branch de rebase. Veja o histórico na Figura 3-31, por exemplo. Você criou um branch tópico (`server`) no seu projeto para adicionar uma funcionalidade no lado servidor e fez o commit. Então, você criou outro branch para fazer mudanças no lado cliente (`client`) e fez alguns commits. Finalmente, você voltou ao ser branch `server` e fez mais alguns commits.

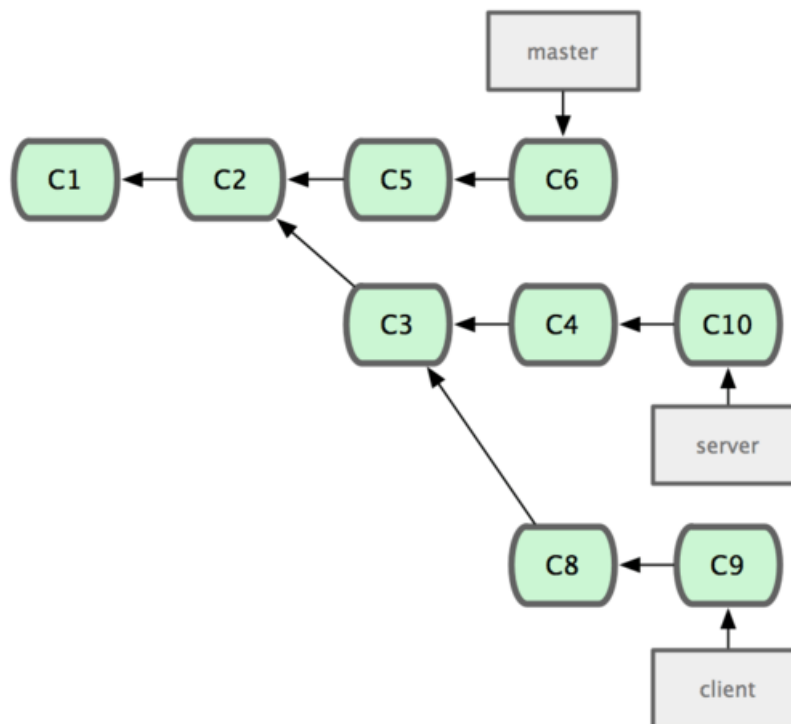


Figura 3-31. Histórico com um branch tópico a partir de outro.

Digamos que você decide fazer um merge das mudanças entre seu branch com mudanças do lado cliente na linha de trabalho principal para lançar uma versão, mas quer segurar as mudanças do lado servidor até que elas sejam testadas melhor. Você pode pegar as mudanças que não estão no servidor (C8 e C9) e incluí-las no seu branch `master` usando a opção `--onto` do `git rebase`:

```
1 $ git rebase --onto master server client
```

Isto basicamente diz, “Faça o checkout do branch `client`, verifique as mudanças a partir do ancestral em comum aos branches `client` e `server`, e coloque-as no `master`.” É um pouco complexo, mas o resultado, mostrado na Figura 3-32, é muito legal:

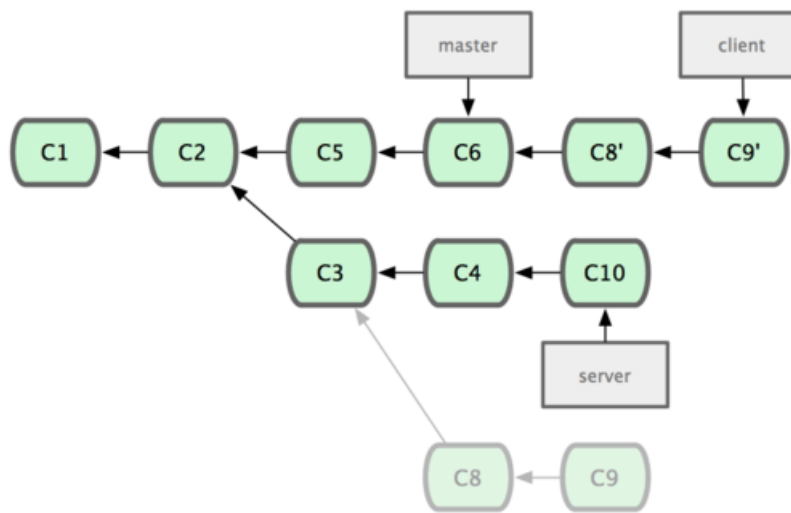


Figura 3-32. Fazendo o rebase de um branch tópico em outro.

Agora você pode avançar (fast-forward) seu branch master (veja Figura 3-33):

- 1 `$ git checkout master`
- 2 `$ git merge client`

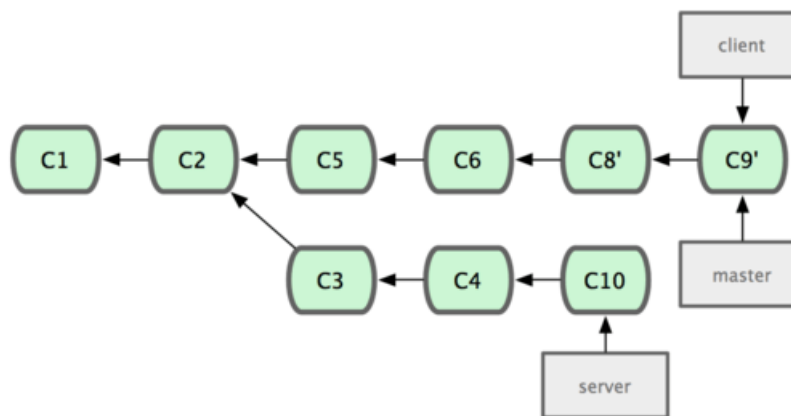


Figura 3-33. Avançando o seu branch master para incluir as mudanças do branch client.

Digamos que você decidiu obter o branch do seu servidor também. Você pode fazer o rebase do branch do servidor no seu branch master sem ter que fazer o checkout primeiro com o comando `git rebase [branchbase] [branchtopico]` — que faz o checkout do branch tópico (nesse caso, `server`) pra você e aplica-o no branch base (`master`):

- 1 `$ git rebase master server`

Isso aplica o seu trabalho em `server` após aquele existente em `master`, como é mostrado na Figura 3-34:

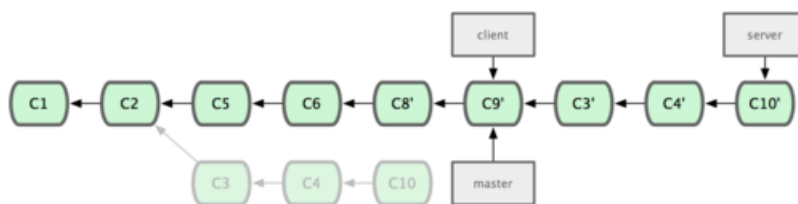


Figura 3-34. Fazendo o rebase do seu branch server após seu branch master.

Em seguida, você pode avançar seu branch base (master):

- 1 `$ git checkout master`
- 2 `$ git merge server`

Você pode apagar os branches `client` e `server` pois todo o trabalho já foi integrado e você não precisa mais deles, deixando seu histórico de todo esse processo parecendo com a Figura 3-35:

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

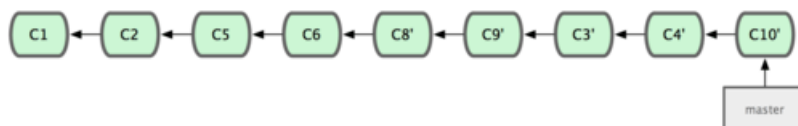


Figura 3-35. Histórico final de commits.

Os Perigos do Rebase

Ahh, mas apesar dos benefícios do rebase existem os inconvenientes, que podem ser resumidos em um linha:

Não faça rebase de commits que você enviou para um repositório público.

Se você seguir essa regra você ficará bem. Se não seguir, as pessoas te odiarão e você será desprezado por amigos e familiares.

Quando você faz o rebase, você está abandonando commits existentes e criando novos que são similares, mas diferentes. Se fizer o push de commits em algum lugar e outros pegarem e fizerem trabalhos baseado neles e você reescrever esses commits com `git rebase` e fizer o push novamente, seus colaboradores terão que fazer o merge de seus trabalhos novamente e as coisas ficarão bagunçadas quando você tentar trazer o trabalho deles de volta para o seu.

Vamos ver um exemplo de como o rebase funciona e dos problemas que podem ser causados quando você torna algo público. Digamos que você faça o clone de um servidor central e faça algum trabalho em cima dele. Seu histórico de commits parece com a Figura 3-36.

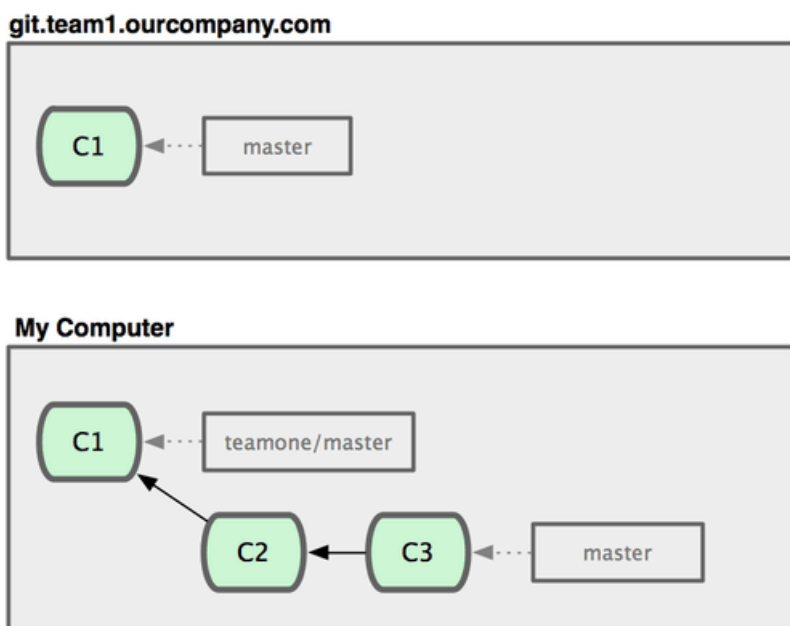


Figura 3-36. Clone de um repositório e trabalho a partir dele.

Agora, outra pessoa faz modificações que inclui um merge e envia (push) esse trabalho para o servidor central. Você o obtêm e faz o merge do novo branch remoto no seu trabalho, fazendo com que seu histórico fique como na Figura 3-37.

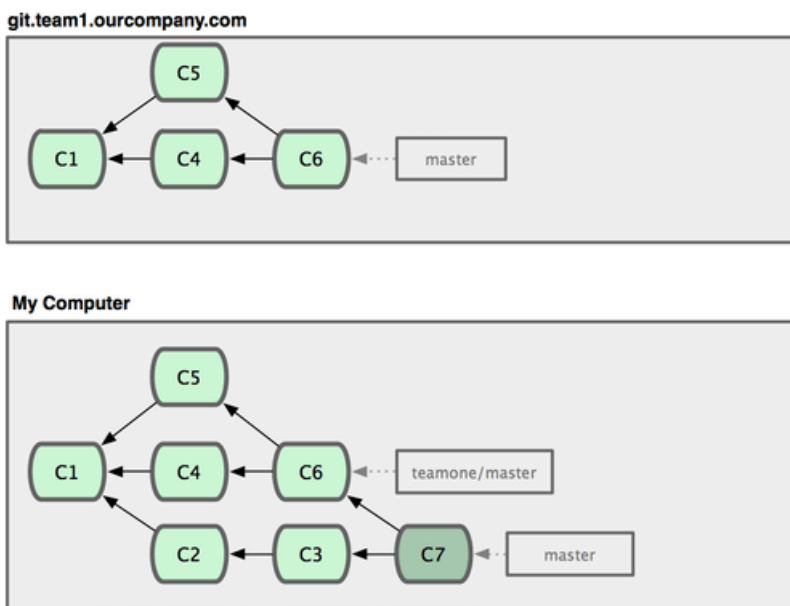


Figura 3-37. Obtêm mais commits e faz o merge deles no seu trabalho.

Em seguida, a pessoa que enviou o merge voltou atrás e fez o rebase do seu trabalho; eles executam `git push --force` para sobrescrever o histórico no servidor. Você então obtêm os dados do servidor, trazendo os novos commits.

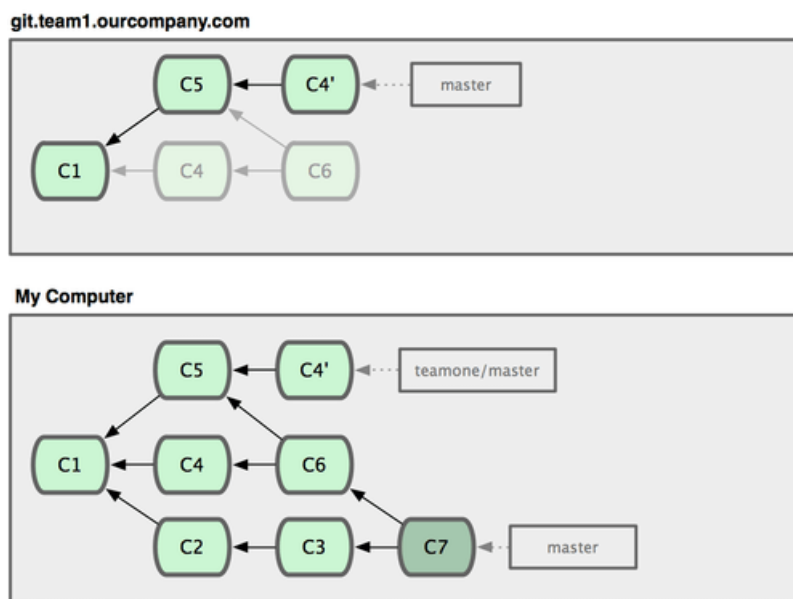


Figura 3-38. Alguém envia commits com rebase, abandonando os commits que você usou como base para o seu trabalho.

Nesse ponto, você tem que fazer o merge dessas modificações novamente, mesmo que você já o tenha feito. Fazer o rebase muda o código hash SHA-1 desses commits, então para o Git eles são commits novos, embora você já tenha as modificações de C4 no seu histórico (veja Figura 3-39).

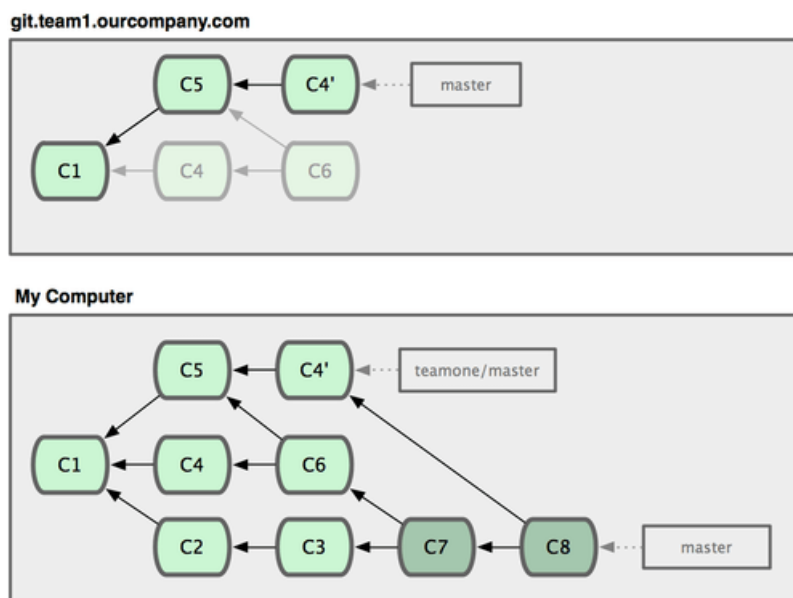


Figura 3-39. Você faz o merge novamente das mesmas coisas em um novo commit.

Você tem que fazer o merge desse trabalho em algum momento para se manter atualizado em relação ao outro desenvolvedor no futuro. Depois de fazer isso, seu histórico de commits terá tanto o commit C4 quanto C4', que tem códigos hash SHA-1 diferentes mas tem as mesmas modificações e a mesma

mensagem de commit. Se você executar `git log` quando seu histórico está dessa forma, você verá dois commits que terão o mesmo autor, data e mensagem, o que será confuso. Além disso, se você enviar (push) esses histórico de volta ao servidor, você irá inserir novamente todos esses commits com rebase no servidor central, o que pode mais tarde confundir as pessoas.

Se você tratar o rebase como uma maneira de manter limpo e trabalhar com commits antes de enviá-los, e se você faz somente rebase de commits que nunca foram disponíveis publicamente, você ficará bem. Se você faz o rebase de commits que já foram enviados publicamente, e as pessoas podem ter se baseado neles para o trabalho delas, então você poderá ter problemas frustrantes.

3.7 Ramificação (Branching) no Git - Sumário

Sumário

Nós abrangemos o básico do branch e merge no Git. Você deve se sentir confortável ao criar e mudar para novos branches, mudar entre branches e fazer o merge de branches locais. Você deve ser capaz de compartilhar seus branches enviando eles a um servidor compartilhado, trabalhar com outros em branches compartilhados e fazer o rebase de seus branches antes de compartilhá-los.

Capítulo 4 - Git no Servidor

Neste ponto, você deve estar apto a fazer a maior parte das tarefas do dia a dia para as quais estará usando o Git. No entanto, para qualquer colaboração no Git, você precisará ter um repositório remoto do Git. Apesar de você poder tecnicamente enviar (push) e receber (pull) mudanças de repositórios de indivíduos, isto é desencorajado pois você pode facilmente confundi-los no que eles estão trabalhando se não for cuidadoso. Além disso, você quer que seus colaboradores possam acessar o repositório mesmo quando seu computador estiver offline — ter um repositório comum mais confiável é útil muitas vezes. Portanto, o método preferido para colaborar com alguém é configurar um repositório intermediário que vocês dois podem acessar, enviar para (push to) e receber de (pull from). Nós iremos nos referir a este repositório como um “Servidor Git”; mas você perceberá que geralmente é necessária uma quantidade ínfima de recursos para hospedar um repositório Git, logo, você raramente precisará de um servidor inteiro para ele.

Rodar um servidor Git é simples. Primeiro, você escolhe quais protocolos seu servidor usará para se comunicar. A primeira seção deste capítulo cobrirá os protocolos disponíveis e os prós e contras de cada um. As próximas seções explicarão algumas configurações típicas usando estes protocolos e como fazer o seu servidor rodar com eles. Por último, passaremos por algumas opções de hospedagem, se você não se importar em hospedar seu código no servidor dos outros e não quiser passar pelo incômodo de configurar e manter seu próprio servidor.

Se você não tiver interesse em rodar seu próprio servidor, você pode pular para a última seção do capítulo para ver algumas opções para configurar uma conta hospedada, e então ir para o próximo capítulo, onde discutiremos os vários altos e baixos de se trabalhar em um ambiente distribuído de controle de versão.

Um repositório remoto é geralmente um *repositório vazio* — um repositório Git que não tem um diretório de trabalho. Uma vez que o repositório é usado apenas como um ponto de colaboração, não há razão para ter cópias anteriores em disco; são apenas dados Git. Em termos simples, um repositório vazio é o conteúdo do diretório `.git` e nada mais.

4.1 Git no Servidor - Os Protocolos

Os Protocolos

O Git pode usar quatro protocolos principais para transferir dados: Local, Secure Shell (SSH), Git e HTTP. Aqui discutiremos o que eles são e em quais circunstâncias básicas você gostaria (ou não) de utilizá-los.

É importante perceber que com exceção dos protocolos HTTP, todos estes requerem que o Git esteja instalado e rodando no servidor.

Protocolo Local

O protocolo mais básico é o *Protocolo local* (Local protocol), em que o repositório remoto está em outro diretório no disco. Isto é frequentemente utilizado se todos no seu time tem acesso a um sistema de arquivos compartilhados como um NFS montado, ou no caso menos provável de que todos acessem o mesmo computador. Este último caso não seria ideal, porque todas as instâncias do seu repositório de código estariam no mesmo computador, fazendo com que uma perda catastrófica seja muito mais provável.

Se você tem um sistema de arquivos compartilhado, então você pode clonar, enviar para e receber de um repositório local baseado em arquivos. Para clonar um repositório desses ou para adicionar um como remoto de um projeto existente, use o caminho para o repositório como a URL. Por exemplo, para clonar um diretório local, você pode rodar algo como:

```
1 $ git clone /opt/git/project.git
```

Ou você pode fazer isso:

```
1 $ git clone file:///opt/git/project.git
```

O Git opera de forma ligeiramente diferente se você explicitar `file://` no começo da URL. Se você apenas especificar o caminho, o Git tenta usar hardlinks ou copiar diretamente os arquivos que necessita. Se você especificar `file://`, o Git aciona os processos que normalmente utiliza para transferir dados através de uma rede, o que é geralmente uma forma de transferência bem menos eficiente. A principal razão para especificar o prefixo `file://` é se você quer uma cópia limpa do repositório com referências e objetos estranhos deixados de lado — geralmente depois de importar de outro sistema de controle de versões ou algo similar (ver Capítulo 9 para tarefas de manutenção). Usaremos o caminho normal aqui pois isto é quase sempre mais rápido.

Para adicionar um repositório local para um projeto Git existente, você pode rodar algo assim:

```
1 $ git remote add local_proj /opt/git/project.git
```

Então você pode enviar para e receber deste remoto como se você estivesse fazendo isto através de uma rede.

Os Prós

Os prós de repositórios baseados em arquivos são que eles são simples e usam permissões de arquivo e acessos de rede existentes. Se você já tem um sistema de arquivos compartilhados ao qual todo o seu time tem acesso, configurar um repositório é muito fácil. Você coloca o repositório vazio em algum lugar onde todos tem acesso compartilhado e configura as permissões de leitura/escrita como você faria para qualquer outro diretório compartilhado. Discutiremos como exportar uma cópia de repositório vazio com este objetivo na próxima seção, “Colocando Git em um Servidor.”

Esta é também uma boa opção para rapidamente pegar trabalhos do diretório em que outra pessoa estiver trabalhando. Se você e seu colega estiverem trabalhando no mesmo projeto e ele quiser que você olhe alguma coisa, rodar um comando como `git pull /home/john/project` é frequentemente mais fácil do que ele enviar para um servidor remoto e você pegar de lá.

Os Contras

Os contras deste método são que o acesso compartilhado é geralmente mais difícil de configurar e acessar de múltiplos lugares do que via conexão básica de rede. Se você quiser enviar do seu laptop quando você está em casa, você tem que montar um disco remoto, o que pode ser difícil e lento comparado com acesso via rede.

É também importante mencionar que isto não é necessariamente a opção mais rápida se você está usando uma montagem compartilhada de algum tipo. Um repositório local é rápido apenas se você tem acesso rápido aos dados. Um repositório em NFS é frequentemente mais lento do que acessar um repositório com SSH no mesmo servidor, permitindo ao Git rodar discos locais em cada sistema.

O Protocolo SSH

Provavelmente o protocolo mais comum de transporte para o Git é o SSH. Isto porque o acesso SSH aos servidores já está configurado na maior parte dos lugares — e se não está, é fácil fazê-lo. O SSH é também o único protocolo para redes em que você pode facilmente ler (do servidor) e escrever (no servidor). Os outros dois protocolos de rede (HTTP e Git) são geralmente somente leitura, então mesmo se você os tiver disponíveis para as massas, você ainda precisa do SSH para seus próprios comandos de escrita. O SSH é também um protocolo de rede autenticado; e já que ele é ubíquo, é geralmente fácil de configurar e usar.

Para clonar um repositório Git através de SSH, você pode especificar uma URL `ssh://` deste jeito:

```
1 $ git clone ssh://user@server/project.git
```

Ou você pode deixar de especificar o protocolo — O Git assume SSH se você não for explícito:

```
1 $ git clone user@server:project.git
```

Você também pode deixar de especificar um usuário, e o Git assume o usuário que você estiver usando atualmente.

Os Prós

Os prós de usar SSH são muitos. Primeiro, você basicamente tem que usá-lo se você quer acesso de escrita autenticado através de uma rede. Segundo, o SSH é relativamente simples de configurar — Serviços (Daemons) SSH são muito comuns, muitos administradores de rede tem experiência com eles, e muitas distribuições de SOs estão configuradas com eles ou tem ferramentas para gerenciá-los. Em seguida, o acesso através de SSH é seguro — toda transferência de dados é criptografada e

autenticada. Por último, como os protocolos Git e Local, o SSH é eficiente, compactando os dados da melhor forma possível antes de transferi-los.

Os Contras

O aspecto negativo do SSH é que você não pode permitir acesso anônimo do seu repositório através dele. As pessoas tem que acessar o seu computador através de SSH para acessá-lo, mesmo que apenas para leitura, o que não faz com que o acesso por SSH seja encorajador para projetos de código aberto. Se você o está usando apenas dentro de sua rede corporativa, o SSH pode ser o único protocolo com o qual você terá que lidar. Se você quiser permitir acesso anônimo somente leitura para seus projetos, você terá que configurar o SSH para envio (push over) mas configurar outra coisa para que as pessoas possam receber (pull over).

O Protocolo Git

O próximo é o protocolo Git. Este é um daemon especial que vem no mesmo pacote que o Git; ele escuta em uma porta dedicada (9418) que provê um serviço similar ao do protocolo SSH, mas absolutamente sem nenhuma autenticação. Para que um repositório seja disponibilizado via protocolo Git, você tem que criar o arquivo `git-daemon-export-ok` — o daemon não disponibilizará um repositório sem este arquivo dentro — mas além disso não há nenhuma segurança. Ou o repositório Git está disponível para todos clonarem ou não. Isto significa que geralmente não existe envio (push) sobre este protocolo. Você pode habilitar o acesso a envio; mas dada a falta de autenticação, se você ativar o acesso de envio, qualquer um na internet que encontre a URL do seu projeto poderia enviar (push) para o seu projeto. É suficiente dizer que isto é raro.

Os Prós

O protocolo Git é o mais rápido entre os disponíveis. Se você está servindo muito tráfego para um projeto público ou servindo um projeto muito grande que não requer autenticação para acesso de leitura, é provável que você vai querer configurar um daemon Git para servir o seu projeto. Ele usa o mesmo mecanismo de transmissão de dados que o protocolo SSH, mas sem o tempo gasto na criptografia e autenticação.

Os Contras

O lado ruim do protocolo Git é a falta de autenticação. É geralmente indesejável que o protocolo Git seja o único acesso ao seu projeto. Geralmente, você o usará em par com um acesso SSH para os poucos desenvolvedores com acesso de envio (push) e todos os outros usariam `git://` para acesso somente leitura. É também provavelmente o protocolo mais difícil de configurar. Ele precisa rodar seu próprio daemon, que é específico — iremos olhar como configurar um na seção “Gitosis” deste capítulo — ele requer a configuração do `xinetd` ou algo similar, o que não é sempre fácil. Ele requer também acesso a porta 9418 via firewall, o que não é uma porta padrão que firewalls corporativos sempre permitem. Por trás de grandes firewalls corporativos, esta porta obscura está comumente bloqueada.

O Protocolo HTTP/S Protocol

Por último temos o protocolo HTTP. A beleza do protocolo HTTP ou HTTPS é a simplicidade em configurar. Basicamente, tudo o que você precisa fazer é colocar o repositório Git do jeito que ele é em uma pasta acessível pelo Servidor HTTP e configurar o gancho (hook) post-update, e estará pronto (veja o *Capítulo 7* para detalhes dos hooks do Git). Neste momento, qualquer um com acesso ao servidor web no qual você colocou o repositório também pode clonar o repositório. Para permitir acesso de leitura ao seu repositório usando HTTP, execute o seguinte:

```
1 $ cd /var/www/htdocs/
2 $ git clone --bare /path/to/git_project gitproject.git
3 $ cd gitproject.git
4 $ mv hooks/post-update.sample hooks/post-update
5 $ chmod a+x hooks/post-update
```

E pronto. O gancho post-update que vem com o Git executa o comando apropriado (`git update-server-info`) para que fetch e clone via HTTP funcionem corretamente. Este comando é executado quando você envia para o repositório usando push via SSH; então, outros podem clonar via algo como

```
1 $ git clone http://example.com/gitproject.git
```

Neste caso particular, estamos usando o caminho `/var/www/htdocs` que é comum para configurações Apache, mas você pode usar qualquer servidor web estático — apenas coloque o caminho do repositório. Os dados no Git são servidos como arquivos estáticos básicos (veja o *Capítulo 9* para mais detalhes sobre como exatamente eles são servidos).

É possível fazer o Git enviar via HTTP também, embora esta técnica não seja muito usada e requer que você configure WebDav com parâmetros complexos. Pelo fato de ser usado raramente, não mostraremos isto neste livro. Se você está interessado em usar os protocolos HTTP-push, você pode ler sobre preparação de um repositório para este propósito em <http://www.kernel.org/pub/software/scm/git/docs/>. Uma coisa legal sobre fazer o Git enviar via HTTP é que você pode usar qualquer servidor WebDAV, sem quaisquer características Git; então, você pode usar esta funcionalidade se o seu provedor web suporta WebDAV com permissão de escrita para o seu web site.

Os Prós

O lado bom de usar protocolo HTTP é que ele é fácil de configurar. Executar o punhado de comandos obrigatórios lhe provê um jeito simples de fornecer ao mundo acesso ao seu repositório Git. Você só precisa de alguns minutos. O protocolo HTTP também não consome muitos recursos no servidor. Pelo fato de usar apenas um servidor HTTP estático para todo o dado, um servidor Apache normal pode servir em média milhares de arquivos por segundo — é difícil sobrecarregar até mesmo um servidor pequeno.

Você também pode servir seus repositórios com apenas acesso de leitura via HTTPS, o que significa que você pode criptografar o conteúdo transferido; ou pode ir até o ponto de fazer seus usuários

usarem certificados SSL assinados. Geralmente, se você está indo até este ponto, é mais fácil usar as chaves públicas SSH; mas pode ser uma solução melhor em casos específicos usar certificados SSL assinados ou outro método de autenticação HTTP para acesso de leitura via HTTPS.

Outra coisa legal é que HTTP é um protocolo tão comumente usado que firewalls corporativos são normalmente configurados para permitir tráfego por esta porta.

Os Contras

O lado ruim de servir seu repositório via HTTP é que ele é relativamente ineficiente para o usuário. Geralmente demora muito mais para clonar ou fazer um fetch do repositório, e você frequentemente tem mais sobrecarga de rede e volume de transferência via HTTP do que com outros protocolos de rede. Pelo fato de não ser inteligente sobre os dados que você precisa — não tem um trabalho dinâmico por parte do servidor nestas transações — o protocolo HTTP é frequentemente referido como o protocolo burro. Para mais informações sobre as diferenças em eficiência entre o protocolo HTTP e outros protocolos, veja o Capítulo 9.

4.2 Git no Servidor - Configurando Git no Servidor

Configurando Git no Servidor

Antes de configurar qualquer servidor Git, você tem que exportar um repositório existente em um novo repositório limpo — um repositório que não contém um diretório sendo trabalhado. Isto é geralmente fácil de fazer. Para clonar seu repositório para criar um novo repositório limpo, você pode executar o comando clone com a opção `--bare`. Por convenção, diretórios de repositórios limpos terminam em `.git`, assim:

```
1 $ git clone --bare my_project my_project.git
2 Initialized empty Git repository in /opt/projects/my_project.git/
```

O resultado deste comando é um pouco confuso. Já que clone é basicamente um `git init` seguido de um `git fetch`, nós vemos um pouco do resultado de `git init`, que cria um diretório vazio. A transferência real de objetos não dá nenhum resultado, mas ocorre. Você deve ter agora uma cópia dos dados do diretório Git no seu diretório `my_project.git`.

Isto é mais ou menos equivalente a algo assim

```
1 $ cp -Rf my_project/.git my_project.git
```

Existem algumas diferenças menores no arquivo de configuração caso você siga este caminho; mas para o propósito, isto é perto da mesma coisa. Ele copia o repositório Git, sem um diretório de trabalho, e cria um diretório especificamente para ele sozinho.

Colocando o Repositório Limpo no Servidor

Agora que você tem uma cópia limpa do seu repositório, tudo o que você precisa fazer é colocá-lo num servidor e configurar os protocolos. Vamos dizer que você configurou um servidor chamado `git.example.com` que você tem acesso via SSH, e você quer armazenar todos os seus repositórios Git no diretório `/opt/git`. Você pode configurar o seu novo repositório apenas copiando o seu repositório limpo:

```
1 $ scp -r my_project.git user@git.example.com:/opt/git
```

Neste ponto, outros usuários com acesso SSH para o mesmo servidor e que possuam acesso de leitura para o diretório `/opt/git` podem clonar o seu repositório executando

```
1 $ git clone user@git.example.com:/opt/git/my_project.git
```

Se um usuário acessar um servidor via SSH e ele tiver acesso de escrita no diretório `/opt/git/my_project.git`, ele também terá acesso para envio (push) automaticamente. Git irá automaticamente adicionar permissões de escrita apropriadas para o grupo se o comando `git init` com a opção `--shared` for executada em um repositório.

```
1 $ ssh user@git.example.com
2 $ cd /opt/git/my_project.git
3 $ git init --bare --shared
```

Você pode ver como é fácil pegar um repositório Git, criar uma versão limpa, e colocar num servidor onde você e seus colaboradores têm acesso SSH. Agora vocês estão prontos para colaborar no mesmo projeto.

É importante notar que isso é literalmente tudo que você precisa fazer para rodar um servidor Git útil no qual várias pessoas possam acessar — apenas adicione as contas com acesso SSH ao servidor, coloque um repositório Git em algum lugar do servidor no qual todos os usuários tenham acesso de leitura e escrita. Você está pronto — nada mais é necessário.

Nas próximas seções, você verá como expandir para configurações mais sofisticadas. Essa discussão irá incluir a característica de não precisar criar contas para cada usuário, adicionar acesso de leitura pública para os seus repositórios, configurar Web UIs, usando a ferramenta Gitis, e mais. Entretanto, mantenha em mente que para colaborar com algumas pessoas em um projeto privado, tudo o que você *precisa* é um servidor SSH e um repositório limpo.

Setups Pequenos

Se você for uma pequena empresa ou está apenas testando Git na sua organização e tem alguns desenvolvedores, as coisas podem ser simples para você. Um dos aspectos mais complicados de configurar um servidor Git é o gerenciamento de usuários. Se você quer que alguns repositórios sejam apenas de leitura para alguns usuários e leitura/escrita para outros, acesso e permissões podem ser um pouco difíceis de arranjar.

Acesso SSH

Se você já tem um servidor ao qual todos os seus desenvolvedores tem acesso SSH, é geralmente mais fácil configurar o seu primeiro repositório lá, pelo fato de você não precisar fazer praticamente nenhum trabalho extra (como mostramos na última seção). Se você quiser um controle de acesso mais complexo nos seus repositórios, você pode gerenciá-los com o sistema de permissão de arquivos do sistema operacional que o seu servidor roda.

Se você quiser colocar seus repositórios num servidor que não possui contas para todos no seu time que você quer dar permissão de acesso, então você deve configurar acesso SSH para eles. Assumimos que se você tem um servidor com o qual fazer isso, você já tem um servidor SSH instalado, e é assim que você está acessando o servidor.

Existem algumas alternativas para dar acesso a todos no seu time. A primeira é configurar contas para todos, o que é simples mas pode se tornar complicado. Você provavelmente não quer executar `adduser` e definir senhas temporárias para cada usuário.

Um segundo método é criar um único usuário ‘git’ na máquina, pedir a cada usuário que deve possuir acesso de escrita para enviar a você uma chave pública SSH, e adicionar estas chaves no arquivo `~/.ssh/authorized_keys` do seu novo usuário ‘git’. Depois disto, todos poderão acessar aquela máquina usando o usuário ‘git’. Isto não afeta os dados de commit de maneira alguma — o usuário SSH que você usa para se conectar não afeta os commits que você gravou previamente.

Outro método é fazer o seu servidor SSH se autenticar a partir de um servidor LDAP ou outro autenticador central que você talvez já tenha previamente configurado. Contanto que cada usuário tenha acesso shell à máquina, qualquer mecanismo de autenticação SSH que você imaginar deve funcionar.

4.3 Git no Servidor - Gerando Sua Chave Pública SSH

Gerando Sua Chave Pública SSH

Vários servidores Git autenticam usando chaves públicas SSH. Para fornecer uma chave pública, cada usuário no seu sistema deve gerar uma se eles ainda não a possuem. Este processo é similar entre os vários sistemas operacionais. Primeiro, você deve checar para ter certeza que você ainda não possui uma chave. Por padrão, as chaves SSH de um usuário são armazenadas no diretório `~/.ssh`. Você pode facilmente verificar se você tem uma chave indo para esse diretório e listando o seu conteúdo:

```

1 $ cd ~/.ssh
2 $ ls
3 authorized_keys2  id_dsa          known_hosts
4 config            id_dsa.pub

```

Você está procurando por um par de arquivos chamados *algo* e *algo.pub*, onde algo é normalmente *id_dsa* ou *id_rsa*. O arquivo *.pub* é a sua chave pública, e o outro arquivo é a sua chave privada. Se você não tem estes arquivos (ou não tem nem mesmo o diretório *.ssh*), você pode criá-los executando um programa chamado *ssh-keygen*, que é fornecido com o pacote SSH em sistemas Linux/Mac e vem com o pacote MSysGit no Windows:

```

1 $ ssh-keygen
2 Generating public/private rsa key pair.
3 Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
4 Enter passphrase (empty for no passphrase):
5 Enter same passphrase again:
6 Your identification has been saved in /Users/schacon/.ssh/id_rsa.
7 Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
8 The key fingerprint is:
9 43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local

```

Primeiro ele confirma onde você quer salvar a chave (*.ssh/id_rsa*), e então pergunta duas vezes por uma senha, que você pode deixar em branco se você não quiser digitar uma senha quando usar a chave.

Agora, cada usuário que executar o comando acima precisa enviar a chave pública para você ou para o administrador do seu servidor Git (assumindo que você está usando um servidor SSH cuja configuração necessita de chaves públicas). Tudo o que eles precisam fazer é copiar o conteúdo do arquivo *.pub* e enviar para você via e-mail. As chaves públicas são parecidas com isso.

```

1 $ cat ~/.ssh/id_rsa.pub
2 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAKl0UpkDHrfHY17SbrmTIpNLTKG9Tjom/BWDSU
3 GP1+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7x1ELEVf4h91FX5QVkbPppSwg0cda3
4 Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvS1VK/7XA
5 t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprX88XypNDvjYNby6vw/Pb0rwert/En
6 mZ+AW40ZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraT1MqVSsbx
7 NrRFi9wrf+M7Q== schacon@agadorlaptop.local

```

Para um tutorial mais detalhado sobre criação de chaves SSH em vários sistemas operacionais, veja o guia do GitHub sobre chaves SSH no endereço <http://github.com/guides/providing-your-ssh-key>.

4.4 Git no Servidor - Configurando o Servidor

Configurando o Servidor

Vamos agora configurar o acesso SSH no lado servidor. Neste exemplo, você irá autenticar seus usuários pelo método das `authorized_keys`. Também assumimos que você esteja rodando uma distribuição padrão do Linux como o Ubuntu. Primeiramente, crie um usuário 'git' e um diretório `.ssh` para ele.

```
1 $ sudo adduser git
2 $ su git
3 $ cd
4 $ mkdir .ssh
```

A seguir, você precisará adicionar uma chave pública de algum desenvolvedor no arquivo `authorized_keys` do usuário 'git'. Vamos assumir que você recebeu algumas chaves por e-mail e as salvou em arquivos temporários. Novamente, as chaves públicas são algo parecido com isso aqui:

```
1 $ cat /tmp/id_rsa.john.pub
2 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQACB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
3 oJG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
4 Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIGS9Ez
5 Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
6 07TCUSBdLQ1gMVOFq1I2uPWQ0kOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
7 dAv8JggJICUvax2T9va5 gsg-keypair
```

Você tem apenas que salvá-las no arquivo `authorized_keys`:

```
1 $ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
2 $ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
3 $ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Agora, você pode configurar um repositório vazio para eles executando o comando `git init` com a opção `--bare`, que inicializa o repositório sem um diretório de trabalho:

```
1 $ cd /opt/git
2 $ mkdir project.git
3 $ cd project.git
4 $ git --bare init
```

Assim, John, Josie ou Jessica podem enviar a primeira versão dos seus projetos para o repositório simplesmente adicionando-o como um remoto e enviando (push) uma branch. Atente que alguém deve acessar o servidor e criar um repositório limpo toda vez que eles queiram adicionar um projeto. Vamos usar gitserver como o nome do servidor no qual você configurou o usuário 'git' e o repositório. Se você estiver rodando ele internamente, e você configurou uma entrada DNS para gitserver apontando para esta máquina, então você pode simplesmente seguir os comandos abaixo:

```
1  # on Johns computer
2  $ cd myproject
3  $ git init
4  $ git add .
5  $ git commit -m 'initial commit'
6  $ git remote add origin git@gitserver:/opt/git/project.git
7  $ git push origin master
```

Neste momento, os outros podem clonar e enviar as mudanças facilmente:

```
1  $ git clone git@gitserver:/opt/git/project.git
2  $ vim README
3  $ git commit -am 'fix for the README file'
4  $ git push origin master
```

Com este método, você pode rapidamente ter um servidor com acesso de leitura e escrita rodando para os desenvolvedores.

Como uma precaução extra, você pode facilmente restringir o usuário 'git' para executar apenas atividades Git com uma shell limitada chamada git-shell que vem com o Git. Se você configurar ela como a shell do seu usuário 'git', o usuário não poderá ter acesso shell normal ao seu servidor. Para usar esta característica, especifique git-shell ao invés de bash ou csh no login shell do usuário. Para fazê-lo, você provavelmente vai ter que editar o arquivo /etc/passwd:

```
1  $ sudo vim /etc/passwd
```

No final, você deve encontrar uma linha parecida com essa:

```
1  git:x:1000:1000::/home/git:/bin/sh
```

Modifique /bin/sh para /usr/bin/git-shell (ou execute `which git-shell` para ver onde ele está instalado). A linha modificada deve se parecer com a de abaixo:


```
1 git:x:1000:1000:~/home/git:/usr/bin/git-shell
```

Agora, o usuário ‘git’ pode apenas usar a conexão SSH para enviar e puxar repositórios Git e não pode se conectar via SSH na máquina. Se você tentar, você verá uma mensagem de rejeição parecida com a seguinte:

```
1 $ ssh git@gitserver
2 fatal: What do you think I am? A shell?
3 Connection to gitserver closed.
```

4.5 Git no Servidor - Acesso Público

Acesso Público

E se você quiser acesso anônimo de leitura ao seu projeto? Talvez ao invés de armazenar um projeto privado interno, você queira armazenar um projeto de código aberto. Ou talvez você tenha alguns servidores de compilação automatizados ou servidores de integração contínua que estão sempre sendo modificados, e você não queira gerar chaves SSH o tempo todo — você simplesmente quer permitir acesso de leitura anônimo.

Provavelmente o jeito mais fácil para pequenas configurações é rodar um servidor web estático com o documento raiz onde os seus repositórios Git estão, e então ativar o gancho (hook) post-update que mencionamos na primeira seção deste capítulo. Vamos trabalhar a partir do exemplo anterior. Vamos dizer que você tenha seus repositórios no diretório /opt/git, e um servidor Apache rodando na máquina. Novamente, você pode usar qualquer servidor web para fazer isso; mas para esse exemplo, vamos demonstrar algumas configurações básicas do Apache para te dar uma ideia do que você vai precisar:

Primeiro você tem que habilitar o gancho:

```
1 $ cd project.git
2 $ mv hooks/post-update.sample hooks/post-update
3 $ chmod a+x hooks/post-update
```

Se você estiver usando uma versão do Git anterior à 1.6, o comando mv não é necessário — o Git começou a nomear os exemplos de gancho com o sufixo .sample apenas recentemente.

O que este gancho post-update faz? Ele se parece basicamente com isso aqui:

```
1 $ cat .git/hooks/post-update
2 #!/bin/sh
3 exec git-update-server-info
```

Isto significa que quando você enviar para o servidor via SSH, o Git irá executar este comando para atualizar os arquivos necessários para fetch via HTTP.

Em seguida, você precisa adicionar uma entrada VirtualHost na sua configuração do Apache com a opção DocumentRoot apontando para o diretório raiz dos seus projetos Git. Aqui, assumimos que você tem uma entrada DNS para enviar *.gitserver para a máquina que você está usando para rodar tudo isso:

```
1 <VirtualHost *:80>
2     ServerName git.gitserver
3     DocumentRoot /opt/git
4     <Directory /opt/git/>
5         Order allow, deny
6         allow from all
7     </Directory>
8 </VirtualHost>
```

Você também precisará configurar o grupo de usuários dos diretórios em /opt/git para www-data para que o seu servidor web possa ler os repositórios, pelo fato do script CGI do Apache rodar (padrão) como este usuário:

```
1 $ chgrp -R www-data /opt/git
```

Quando você reiniciar o Apache, você deve poder clonar os repositórios dentro daquele diretório especificando a URL para o projeto:

```
1 $ git clone http://git.gitserver/project.git
```

Deste jeito, você pode configurar um servidor HTTP com acesso de leitura para os seus projetos para vários usuários em minutos. Outra opção simples para acesso público sem autenticação é iniciar um daemon Git, embora isso necessite que você daemonize o processo - iremos cobrir esta opção na próxima seção, se você preferir esta rota.

4.6 Git no Servidor - GitWeb

GitWeb

Agora que você tem acesso de leitura/escrita e apenas leitura para o seu projeto, você pode querer configurar um visualizador simples baseado em web. Git vem com um script CGI chamado GitWeb que normalmente é usado para isso. Você pode ver o GitWeb em uso em sites como <http://git.kernel.org> (veja a Figura 4-1).

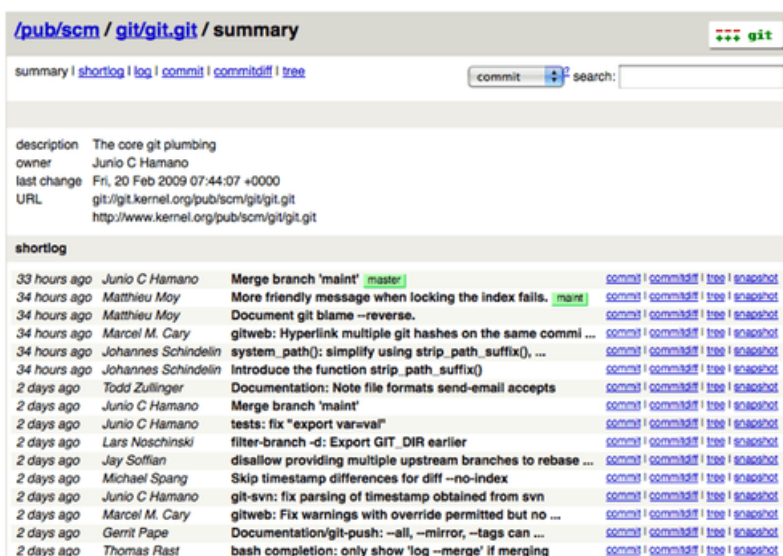


Figure 4-1. A interface de usuário baseada em web GitWeb.

Se você quiser ver como GitWeb aparecerá para o seu projeto, Git vem com um comando para disparar uma instância temporária se você tem um servidor leve no seu sistema como `lighttpd` ou `webrick`. Em máquinas Linux, `lighttpd` normalmente está instalado, então você deve conseguir fazê-lo funcionar digitando `git instaweb` no diretório do seu projeto. Se você está usando um Mac, Leopard vem com Ruby pré-instalado, então `webrick` é sua melhor aposta. Para iniciar `instaweb` com um manipulador diferente de `lighttpd`, você pode rodá-lo com a opção `--httpd`.

```
1 $ git instaweb --httpd=webrick
2 [2009-02-21 10:02:21] INFO WEBrick 1.3.1
3 [2009-02-21 10:02:21] INFO ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Isso inicia um servidor HTTPD na porta 1234 e então automaticamente inicia um navegador web que abre naquela página. Quando você tiver terminado e quiser desligar o servidor, você pode rodar o mesmo comando com a opção `--stop`:

```
1 $ git instaweb --httpd=webrick --stop
```

Se você quer rodar a interface web num servidor o tempo inteiro para a sua equipe ou para um projeto open source que você esteja hospedando, você vai precisar configurar o script CGI para ser servido pelo seu servidor web normal. Algumas distribuições Linux têm um pacote gitweb que você deve ser capaz de instalar via apt ou yum, então você pode tentar isso primeiro. Nós procederemos na instalação do GitWeb manualmente bem rápido. Primeiro, você precisa pegar o código-fonte do Git, o qual o GitWeb acompanha, e gerar o script CGI personalizado:

```
1 $ git clone git://git.kernel.org/pub/scm/git/git.git
2 $ cd git/
3 $ make GITWEB_PROJECTROOT="/opt/git" \
4     prefix=/usr gitweb
5 $ sudo cp -Rf gitweb /var/www/
```

Note que você precisa avisar ao comando onde encontrar os seus repositórios Git com a variável GITWEB_PROJECTROOT. Agora, você precisa fazer o Apache usar CGI para aquele script, para o qual você pode adicionar um VirtualHost:

```
1 <VirtualHost *:80>
2     ServerName gitserver
3     DocumentRoot /var/www/gitweb
4     <Directory /var/www/gitweb>
5         Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
6         AllowOverride All
7         order allow,deny
8         Allow from all
9         AddHandler cgi-script cgi
10        DirectoryIndex gitweb.cgi
11    </Directory>
12 </VirtualHost>
```

Novamente, GitWeb pode ser servido com qualquer servidor CGI. Agora, você poderá visitar `http://gitserver/` para visualizar seus repositórios online, e você pode usar `http://git.gitserver` para efetuar clone e fetch nos seus repositórios via HTTP.

4.7 Git no Servidor - Gitis

Gitis

Manter as chaves públicas de todos os usuários no arquivo `authorized_keys` para acesso funciona bem somente por um tempo. Quando você tem centenas de usuários, gerenciar esse processo se torna

bastante difícil. Você precisa acessar o servidor via shell toda vez, e não existe controle de acesso - todos no arquivo têm acesso de leitura e escrita para cada projeto.

Nessa hora, talvez você queira passar a usar um software largamente utilizado chamado Gitis. Gitis é basicamente um conjunto de scripts que te ajudam a gerenciar o arquivo `authorized_keys`, bem como implementar alguns controles de acesso simples. A parte realmente interessante é que a Interface de Usuário dessa ferramenta utilizada para adicionar pessoas e determinar o controle de acessos não é uma interface web, e sim um repositório Git especial. Você configura a informação naquele projeto; e quando você executa um push nele, Gitis reconfigura o servidor baseado nas configurações que você fez, o que é bem legal.

Instalar o Gitis não é a tarefa mais simples do mundo, mas também não é tão difícil. É mais fácil utilizar um servidor Linux para fazer isso — os exemplos a seguir utilizam um servidor com Ubuntu 8.10.

Gitis requer algumas ferramentas Python, então antes de tudo você precisa instalar o pacote Python `setuptools`, o qual Ubuntu provê sob o nome de `python-setuptools`:

```
1 $ apt-get install python-setuptools
```

Depois, você clona e instala Gitis do site principal do projeto:

```
1 $ git clone https://github.com/tv42/gitis.git
2 $ cd gitis
3 $ sudo python setup.py install
```

Ao fazer isso, você instala alguns executáveis que Gitis vai utilizar. A seguir, Gitis vai querer colocar seus repositórios em `/home/git`, o que não tem nenhum problema. Mas você já configurou os seus repositórios em `/opt/git`, então, ao invés de reconfigurar tudo, você simplesmente cria um link simbólico:

```
1 $ ln -s /opt/git /home/git/repositories
```

Gitis vai gerenciar as suas chaves por você, então você precisa remover o arquivo atual, adicionar as chaves novamente, e deixar Gitis controlar o arquivo `authorized_keys` automaticamente. Por enquanto, tire o arquivo `authorized_keys`:

```
1 $ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

Em seguida, você precisa ativar o seu shell novamente para o usuário 'git', caso você o tenha mudado para o comando `git-shell`. As pessoas ainda não vão conseguir logar no servidor, porém Gitis vai controlar isso para você. Então, vamos alterar essa linha no seu arquivo `/etc/passwd`

```
1 git:x:1000:1000:./home/git:/usr/bin/git-shell
```

de volta para isso:

```
1 git:x:1000:1000:./home/git:/bin/sh
```

Agora é a hora de inicializar o Gitis. Você faz isso executando o comando `gitis-init` com a sua chave pública pessoal. Se a sua chave pública não está no servidor, você vai ter que copiá-la para lá:

```
1 $ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
2 Initialized empty Git repository in /opt/git/gitosis-admin.git/
3 Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

Isso permite ao usuário com aquela chave modificar o repositório Git principal que controla o Gitis. Em seguida, você precisa configurar manualmente o bit de execução no script `post-update` para o seu novo repositório de controle.

```
1 $ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

Sua configuração está pronta. Se as configurações estão todas corretas, você pode tentar acessar o seu servidor via SSH com o usuário cuja chave pública você adicionou para inicializar o Gitis. Você deve ver algo assim:

```
1 $ ssh git@gitserver
2 PTY allocation request failed on channel 0
3 fatal: unrecognized command 'gitosis-serve schacon@quaternion'
4 Connection to gitserver closed.
```

Essa mensagem significa que o Gitis reconhece você mas te desconectou porque você não está tentando fazer nenhum comando Git. Então, vamos fazer um comando do Git — você vai clonar o repositório central do Gitis:

```
1 # on your local computer
2 $ git clone git@gitserver:gitosis-admin.git
```

Agora, você tem um diretório chamado `gitosis-admin`, o qual tem duas grandes partes:

```
1 $ cd gitosis-admin
2 $ find .
3 ./gitosis.conf
4 ./keydir
5 ./keydir/scott.pub
```

O arquivo `gitosis.conf` é o arquivo de controle a ser usado para especificar usuários, repositórios e permissões. O diretório `keydir` é onde você armazena as chaves públicas de todos os usuários que têm algum tipo de acesso aos seus repositórios — um arquivo por usuário. O nome do arquivo em `key_dir` (no exemplo anterior, `scott.pub`) será diferente para você — Gitosis pega o nome da descrição no final da chave pública que foi importada com o script `gitosis-init`.

Se você olhar no arquivo `gitosis.conf`, ele deveria apenas especificar informações sobre o projeto `gitosis-admin` que você acabou de clonar:

```
1 $ cat gitosis.conf
2 [gitosis]
3
4 [group gitosis-admin]
5 writable = gitosis-admin
6 members = scott
```

Ele mostra que o usuário ‘`scott`’ — o usuário cuja chave pública foi usada para inicializar o Gitosis — é o único que tem acesso ao projeto `gitosis-admin`.

Agora, vamos adicionar um novo projeto para você. Você vai adicionar uma nova seção chamada `mobile` onde você vai listar os desenvolvedores na sua equipe de desenvolvimento `mobile` e projetos que esses desenvolvedores precisam ter acesso. Já que ‘`scott`’ é o único usuário no sistema nesse momento, você vai adicioná-lo como o único membro, e você vai criar um novo projeto chamado `iphone_project` para começar:

```
1 [group mobile]
2 writable = iphone_project
3 members = scott
```

Sempre que você fizer alterações no projeto `gitosis-admin`, você vai precisar commitar as mudanças e enviá-las (push) de volta para o servidor, para que elas tenham efeito:

```

1  $ git commit -am 'add iphone_project and mobile group'
2  [master]: created 8962da8: "changed name"
3  1 files changed, 4 insertions(+), 0 deletions(-)
4  $ git push
5  Counting objects: 5, done.
6  Compressing objects: 100% (2/2), done.
7  Writing objects: 100% (3/3), 272 bytes, done.
8  Total 3 (delta 1), reused 0 (delta 0)
9  To git@gitserver:/opt/git/gitosis-admin.git
10     fb27aec..8962da8  master -> master

```

Você pode fazer o seu primeiro push para o novo projeto `iphone_project` adicionando o seu servidor como um repositório remoto da sua versão local do projeto e fazendo um push. Você não precisa mais criar um repositório vazio (bare) manualmente para novos projetos no servidor — Gitosis os cria automaticamente quando ele vê o seu primeiro push:

```

1  $ git remote add origin git@gitserver:iphone_project.git
2  $ git push origin master
3  Initialized empty Git repository in /opt/git/iphone_project.git/
4  Counting objects: 3, done.
5  Writing objects: 100% (3/3), 230 bytes, done.
6  Total 3 (delta 0), reused 0 (delta 0)
7  To git@gitserver:iphone_project.git
8  * [new branch]      master -> master

```

Note que você não precisa especificar o caminho (na verdade, fazer isso não vai funcionar), apenas uma vírgula e o nome do projeto — Gitosis o encontra para você.

Você quer trabalhar nesse projeto com os seus amigos, então você vai ter que adicionar novamente as chaves públicas deles. Mas, ao invés de acrescentá-las manualmente ao arquivo `~/.ssh/authorized_keys` no seu servidor, você vai adicioná-las, uma chave por arquivo, no diretório `keydir`. A forma com que você nomeia as chaves determina como você se refere aos usuários no arquivo `gitosis.conf`. Vamos adicionar novamente as chaves públicas para John, Josie e Jessica:

```

1  $ cp /tmp/id_rsa.john.pub keydir/john.pub
2  $ cp /tmp/id_rsa.josie.pub keydir/josie.pub
3  $ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub

```

Agora você pode adicioná-los à sua equipe ‘mobile’ para que eles possam ter acesso de leitura e escrita no projeto `iphone_project`:


```
1 [group mobile]
2 writable = iphone_project
3 members = scott john josie jessica
```

Depois que você commitar e enviar essa mudança, todos os quatro usuários serão capazes de ler e escrever naquele projeto.

Gitosis também tem um simples controles de acesso. Se você quer que John tenha apenas acesso de leitura para esse projeto, você pode fazer desta forma:

```
1 [group mobile]
2 writable = iphone_project
3 members = scott josie jessica
4
5 [group mobile_ro]
6 readonly = iphone_project
7 members = john
```

Agora John pode clonar o projeto e receber atualizações, porém o Gitosis não vai deixá-lo enviar as suas atualizações ao projeto. Você pode criar quantos desses grupos você queira, cada um contendo usuários e projetos diferentes. Você também pode especificar outro grupo como membro (usando @ como prefixo), para herdar todos os seus membros automaticamente.

```
1 [group mobile_committers]
2 members = scott josie jessica
3
4 [group mobile]
5 writable = iphone_project
6 members = @mobile_committers
7
8 [group mobile_2]
9 writable = another_iphone_project
10 members = @mobile_committers john
```

Se você tiver quaisquer problemas, pode ser útil adicionar `loglevel=DEBUG` abaixo da seção `[gitosis]`. Se você perdeu o acesso de push por enviar uma configuração errada, você pode consertar o arquivo manualmente no servidor em `/home/git/.gitosis.conf` — o arquivo do qual Gitosis lê suas informações. Um push para o projeto pega o arquivo `gitosis.conf` que você acabou de enviar e o coloca lá. Se você editar esse arquivo manualmente, ele permanece dessa forma até o próximo push bem sucedido para o projeto `gitosis.conf`.

4.8 Git no Servidor - Gitolite

Gitolite

Nota: a última cópia desta seção do livro ProGit está sempre disponível na [documentação do gitolite](http://sitaramc.github.com/gitolite/progit.html)⁴. O autor também gostaria de humildemente informar que, embora esta seção seja precisa, e *pode* (e geralmente *tem*) sido usada para instalar gitolite sem necessidade de ler qualquer outra documentação, ela não é completa, e não pode substituir completamente a enorme quantidade de documentação que vem com o gitolite.

Git começou a se tornar muito popular em ambientes corporativos, que tendem a ter alguns requisitos adicionais em termos de controle de acesso. Gitolite foi originalmente criado para ajudar com esses requisitos, mas verifica-se que é igualmente útil no mundo do código aberto: o Projeto Fedora controla o acesso aos seus repositórios de gerenciamento de pacotes (mais de 10.000 deles!) usando gitolite, e essa é provavelmente a maior instalação do gitolite que existe.

Gitolite permite que você especifique as permissões não apenas por repositório (como o Gitis faz), mas também por branch ou nomes de tag dentro de cada repositório. Ou seja, você pode especificar que certas pessoas (ou grupos de pessoas) só podem fazer um push em certas “refs” (branches ou tags), mas não em outros.

Instalando

Instalar o Gitolite é muito fácil, mesmo se você não leu a extensa documentação que vem com ele. Você precisa de uma conta em um servidor Unix de algum tipo; vários tipos de Linux e Solaris 10, foram testados. Você não precisa de acesso root, assumindo que git, perl, e um servidor ssh openssh compatível já estejam instalados. Nos exemplos abaixo, vamos usar uma conta `gitolite` em um host chamado `gitserver`.

Gitolite é um pouco incomum, em relação a software “servidor” – o acesso é via ssh, e por isso cada ID de usuário no servidor é um “host gitolite” em potencial. Como resultado, há uma noção de “instalar” o software em si, e em seguida “criar” um usuário como “host gitolite”.

Gitolite tem 4 métodos de instalação. As pessoas que usam o Fedora ou sistemas Debian podem obter um RPM ou um DEB e instalá-lo. Pessoas com acesso root podem instalá-lo manualmente. Nesses dois métodos, qualquer usuário do sistema pode, então, tornar-se um “host gitolite”.

Pessoas sem acesso root podem instalá-lo dentro de suas próprias userids. E, finalmente, gitolite pode ser instalado executando um script *na estação de trabalho*, a partir de um shell bash. (Mesmo o bash que vem com `msysgit` vai funcionar.)

Vamos descrever este último método neste artigo; para os outros métodos, por favor consulte a documentação.

⁴<http://sitaramc.github.com/gitolite/progit.html>

Você começa pela obtenção de acesso baseado em chave pública para o servidor, de modo que você pode entrar a partir de sua estação de trabalho no servidor sem receber uma solicitação de senha. O método a seguir funciona em Linux; para outros sistemas operacionais, você pode ter que fazer isso manualmente. Nós assumimos que você já tinha um par de chaves gerado usando o `ssh-keygen`.

```
1 $ ssh-copy-id -i ~/.ssh/id_rsa gitolite@gitserver
```

Isso irá pedir a senha para a conta `gitolite`, e depois configurar o acesso à chave pública. Isto é **essencial** para o script de instalação, então verifique para se certificar de que você pode executar um comando sem que seja pedida uma senha:

```
1 $ ssh gitolite@gitserver pwd
2 /home/gitolite
```

Em seguida, você clona o Gitolite do site principal do projeto e executa o script “easy install” (o terceiro argumento é o seu nome como você gostaria que ele aparecesse no repositório `gitolite-admin` resultante):

```
1 $ git clone git://github.com/sitaramc/gitolite
2 $ cd gitolite/src
3 $ ./gl-easy-install -q gitolite gitserver sitaram
```

E pronto! Gitolite já foi instalado no servidor, e agora você tem um repositório novo chamado `gitolite-admin` no diretório `home` de sua estação de trabalho. Você administra seu `gitolite` fazendo mudanças neste repositório e fazendo um `push` (como no `Gitosis`).

Esse último comando produz uma quantidade grande de informações, que pode ser interessante de ler. Além disso, a primeira vez que você executá-lo, um novo par de chaves é criado; você terá que escolher uma senha ou teclar `enter` para deixar em branco. Porque um segundo par de chaves é necessário, e como ele é usado, é explicado no documento “ssh troubleshooting” que vem com o `Gitolite`.

Repositórios chamados `gitolite-admin` e `testing` são criados no servidor por padrão. Se você deseja clonar um desses localmente (a partir de uma conta que tenha acesso SSH para a conta `gitolite` via *authorized_keys*) digite:

```
1 $ git clone gitolite:gitolite-admin
2 $ git clone gitolite:testing
```

Para clonar esses mesmos repositórios de qualquer outra conta:

```
1 $ git clone gitolite@servername:gitolite-admin
2 $ git clone gitolite@servername:testing
```

Customizando a Instalação

Enquanto que o padrão, “quick install” (instalação rápida) funciona para a maioria das pessoas, existem algumas maneiras de personalizar a instalação se você precisar. Se você omitir o argumento `-q`, você entra em modo de instalação “verbose” – mostra informações detalhadas sobre o que a instalação está fazendo em cada etapa. O modo verbose também permite que você altere alguns parâmetros do lado servidor, tais como a localização dos repositórios, através da edição de um arquivo “rc” que o servidor utiliza. Este arquivo “rc” é comentado, assim você deve ser capaz de fazer qualquer alteração que você precisar com bastante facilidade, salvá-lo, e continuar. Este arquivo também contém várias configurações que você pode mudar para ativar ou desativar alguns dos recursos avançados do gitolite.

Arquivo de Configuração e Regras de Controle de Acesso

Uma vez que a instalação seja feita, você alterna para o repositório `gitolite-admin` (colocado no seu diretório HOME) e olha nele para ver o que você conseguiu:

```
1 $ cd ~/gitolite-admin/
2 $ ls
3 conf/  keydir/
4 $ find conf keydir -type f
5 conf/gitolite.conf
6 keydir/sitaram.pub
7 $ cat conf/gitolite.conf
8 #gitolite conf
9 # please see conf/example.conf for details on syntax and features
10
11 repo gitolite-admin
12     RW+                = sitaram
13
14 repo testing
15     RW+                = @all
```

Observe que “sitaram” (o último argumento no comando `gl-easy-install` que você deu anteriormente) tem permissões completas sobre o repositório `gitolite-admin`, bem como um arquivo de chave pública com o mesmo nome.

A sintaxe do arquivo de configuração do Gitolite é muito bem documentada em `conf/example.conf`, por isso vamos apenas mencionar alguns destaques aqui.

Você pode agrupar usuários ou repositórios por conveniência. Os nomes dos grupos são como macros; ao defini-los, não importa nem mesmo se são projetos ou usuários; essa distinção só é feita quando você usa a “macro”.

```

1 @oss_repos      = linux perl rakudo git gitolite
2 @secret_repos   = fenestra pear
3
4 @admins         = scott      # Adams, not Chacon, sorry :)
5 @interns        = ashok      # get the spelling right, Scott!
6 @engineers      = sitaram dilbert wally alice
7 @staff          = @admins @engineers @interns

```

Você pode controlar permissões no nível “ref”. No exemplo a seguir, os estagiários só podem fazer o push do branch “int”. Engenheiros podem fazer push de qualquer branch cujo nome começa com “eng-“, e as tags que começam com “rc” seguido de um dígito. E os administradores podem fazer qualquer coisa (incluindo retroceder (rewind)) para qualquer ref.

```

1 repo @oss_repos
2   RW  int$              = @interns
3   RW  eng-              = @engineers
4   RW  refs/tags/rc[0-9] = @engineers
5   RW+                      = @admins

```

A expressão após o RW ou RW+ é uma expressão regular (regex) que o refname (ref) do push é comparado. Então, nós a chamamos de “refex”! Naturalmente, uma refex pode ser muito mais poderosa do que o mostrado aqui, por isso não exagere, se você não está confortável com expressões regulares perl.

Além disso, como você provavelmente adivinhou, Gitolite prefixa refs/heads/ como uma conveniência sintática se a refex não começar com refs/.

Uma característica importante da sintaxe do arquivo de configuração é que todas as regras para um repositório não precisam estar em um só lugar. Você pode manter todo o material comum em conjunto, como as regras para todos os oss_repos mostrados acima, e em seguida, adicionar regras específicas para casos específicos mais tarde, assim:

```

1 repo gitolite
2   RW+                      = sitaram

```

Esta regra só vai ser adicionada ao conjunto de regras para o repositório gitolite.

Neste ponto, você pode estar se perguntando como as regras de controle de acesso são realmente aplicadas, então vamos ver isso brevemente.

Existem dois níveis de controle de acesso no gitolite. O primeiro é ao nível do repositório; se você tem acesso de leitura (ou escrita) a *qualquer* ref no repositório, então você tem acesso de leitura (ou escrita) ao repositório.

O segundo nível, aplicável apenas a acesso de “gravação”, é por branch ou tag dentro de um repositório. O nome de usuário, o acesso a ser tentado (w ou +), e o refname sendo atualizado são conhecidos. As regras de acesso são verificadas em ordem de aparição no arquivo de configuração, à procura de uma correspondência para esta combinação (mas lembre-se que o refname é combinado com regex, e não meramente string). Se for encontrada uma correspondência, o push é bem-sucedido. Um “fallthrough” resulta em acesso negado.

Controle de Acesso Avançado com Regras “deny”

Até agora, só vimos que as permissões podem ser R, RW, ou RW+. No entanto, gitolite permite outra permissão: -, que significa “negar”. Isso lhe dá muito mais flexibilidade, à custa de alguma complexidade, porque agora o fallthrough não é a única forma do acesso ser negado, então a *ordem das regras agora importa!*

Digamos que, na situação acima, queremos que os engenheiros sejam capazes de retroceder qualquer branch exceto o master e integ. Eis como:

```
1      RW  master integ    = @engineers
2      -   master integ    = @engineers
3      RW+                               = @engineers
```

Mais uma vez, você simplesmente segue as regras do topo para baixo até atingir uma correspondência para o seu modo de acesso, ou uma negação. Um push sem retrocessos (rewind) no master ou integ é permitido pela primeira regra. Um “rewind push” a essas refs não coincide com a primeira regra, cai na segunda, e é, portanto, negada. Qualquer push (rewind ou não) para refs que não sejam integ ou master não coincidirão com as duas primeiras regras de qualquer maneira, e a terceira regra permite isso.

Se isso soar complicado, você pode querer brincar com ele para aumentar a sua compreensão. Além disso, na maioria das vezes você não precisa “negar” regras de qualquer maneira, então você pode optar por apenas evitá-las, se preferir.

Restringindo Pushes Por Arquivos Alterados

Além de restringir a que branches um usuário pode fazer push com alterações, você também pode restringir quais arquivos estão autorizados a alterar. Por exemplo, talvez o Makefile (ou algum outro programa) não deveria ser alterado por qualquer um, porque um monte de coisas que dependem dele iriam parar de funcionar se as mudanças fossem feitas. Você pode dizer ao gitolite:

```

1 repo foo
2     RW              = @junior_devs @senior_devs
3
4     RW NAME/        = @senior_devs
5     -  NAME/Makefile = @junior_devs
6     RW NAME/        = @junior_devs

```

Este poderoso recurso está documentado em `conf/example.conf`.

Branches Pessoais

Gitolite também tem um recurso chamado “personal branches” (ou melhor, “personal branch namespace”) que pode ser muito útil em um ambiente corporativo.

Um monte de troca de código no mundo git acontece por um pedido de pull (pull request). Em um ambiente corporativo, no entanto, o acesso não autenticado é pouco usado, e uma estação de trabalho de desenvolvedor não pode fazer autenticação, por isso você tem que fazer o push para o servidor central e pedir para alguém fazer um pull a partir dali.

Isso normalmente causa uma confusão no branch de mesmo nome, como acontece em VCS centralizados, além de que configurar permissões para isso torna-se uma tarefa árdua para o administrador.

Gitolite permite definir um prefixo de namespace “personal” ou “scratch” para cada desenvolvedor (por exemplo, `refs/personal/<devname>/*`); veja a seção “personal branches” em `doc/3-faq-tips-etc.mkd` para mais detalhes.

Repositórios “Wildcard”

Gitolite permite especificar repositórios com wildcards (regexes realmente Perl), como, por exemplo `assignments/s[0-9][0-9]/a[0-9][0-9]`. Esta é uma característica muito poderosa, que tem que ser ativada pela opção `$GL_WILDREPOS = 1`; no arquivo `rc`. Isso permite que você atribua um modo de permissão novo (“C”), que permite aos usuários criar repositórios baseados em tais coringas, automaticamente atribui a propriedade para o usuário específico que o criou, permite que ele/ela dê permissões de R e RW para outros usuários colaborarem, etc. Este recurso está documentado em `doc/4-wildcard-repositories.mkd`.

Outras Funcionalidades

Vamos terminar essa discussão com exemplos de outros recursos, os quais são descritos em detalhes nas “faqs, tips, etc” e outros documentos.

Log: Gitolite registra todos os acessos com sucesso. Se você foi um pouco preguiçoso ao dar permissões de retrocesso (rewind) (RW+) às pessoas e alguém estragou o branch “master”, o arquivo de log é um salva-vidas, permitindo de forma fácil e rápida encontrar o SHA que foi estragado.

Git fora do PATH normal: Um recurso extremamente conveniente no gitolite é suportar instalações do git fora do \$PATH normal (isso é mais comum do que você pensa; alguns ambientes corporativos ou mesmo alguns provedores de hospedagem se recusam a instalar coisas em todo o sistema e você acaba colocando-os em seus próprios diretórios). Normalmente, você é forçado a fazer com que o git do lado cliente fique ciente de que os binários git estão neste local não-padrão de alguma forma. Com gitolite, basta escolher uma instalação detalhada (verbose) e definir \$GIT_PATH nos arquivos “RC”. Não serão necessárias alterações do lado cliente depois disso.

Reportar direitos de Acesso: Outro recurso conveniente é o que acontece quando você apenas acessa o servidor via ssh. Gitolite mostra que repositórios você tem acesso, e quais permissões de acesso você tem. Aqui está um exemplo:

```
1 hello sitaram, the gitolite version here is v1.5.4-19-ga3397d4
2 the gitolite config gives you the following access:
3   R      anu-wsd
4   R      entrans
5   R W    git-notes
6   R W    gitolite
7   R W    gitolite-admin
8   R      indic_web_input
9   R      shreelipi_converter
```

Delegação: Para instalações realmente grandes, você pode delegar a responsabilidade de grupos de repositórios para várias pessoas e tê-los gerenciando essas peças de forma independente. Isso reduz a carga sobre o administrador principal. Este recurso tem seu próprio arquivo de documentação no diretório doc/.

Suporte a Gitweb: Gitolite suporta gitweb de várias maneiras. Você pode especificar quais repositórios são visíveis através do gitweb. Você pode definir o “dono” e “Descrição” do gitweb a partir do arquivo de configuração do gitolite. Gitweb tem um mecanismo para que você possa implementar o controle de acesso baseado em autenticação HTTP, você pode fazê-lo usar o arquivo de configuração “compilado” que o gitolite produz, o que significa que as mesmas regras de acesso de controle (para acesso de leitura) se aplicam para gitweb e gitolite.

Mirroring: Gitolite pode ajudar a manter múltiplos mirrors (espelhos), e alternar entre eles facilmente se o servidor principal falhar.

4.9 Git no Servidor - Serviço Git

Serviço Git

Para acesso público e não autenticado para leitura de seus projetos, você irá querer utilizar o protocolo Git ao invés do protocolo HTTP. A razão principal é a velocidade. O protocolo Git é muito

mais eficiente e, portanto, mais rápido do que o protocolo HTTP, de modo que usá-lo irá poupar tempo de seus usuários.

Novamente, isso é para acesso não autenticado e somente leitura. Se seu servidor estiver fora da proteção de seu firewall, utilize o protocolo Git apenas para projetos que são publicamente visíveis na internet. Se o servidor estiver dentro de seu firewall, você pode usá-lo para projetos em que um grande número de pessoas ou computadores (integração contínua ou servidores de compilação) têm acesso somente leitura, e você não quer adicionar uma chave SSH para cada pessoa ou computador.

Em todo caso, o protocolo Git é relativamente fácil de configurar. Basicamente, você precisa executar este comando:

```
1 git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

--reuseaddr permite que o servidor reinicie sem esperar que conexões antigas atinjam um tempo limite, a opção --base-path permite que as pessoas clonem projetos sem especificar o caminho inteiro, e o caminho no final (/opt/git/) diz ao serviço Git onde procurar os repositórios para exportar. Se você estiver protegido por um firewall, você também vai precisar liberar a porta 9418 no computador em que estiver rodando o serviço Git.

Você pode iniciar este processo de diversas maneiras, dependendo do sistema operacional que você estiver usando. Em uma máquina Ubuntu, você pode usar um script Upstart. Por exemplo, neste arquivo

```
1 /etc/event.d/local-git-daemon
```

você pode colocar este script:

```
1 start on startup
2 stop on shutdown
3 exec /usr/bin/git daemon \
4     --user=git --group=git \
5     --reuseaddr \
6     --base-path=/opt/git/ \
7     /opt/git/
8 respawn
```

Por razões de segurança, é altamente recomendável ter este serviço executado com um usuário com permissões de somente leitura para os repositórios – você pode fazer isso criando um novo usuário ‘git-ro’ e executar o serviço com ele. Por uma questão de simplicidade, vamos executá-lo com o usuário ‘git’, o mesmo que o Gitis utiliza.

Quando você reiniciar sua máquina, seu serviço Git será iniciado automaticamente e reiniciará automaticamente se ele parar por algum motivo. Para executá-lo sem ter que reiniciar sua máquina, você pode usar este comando:

```
1 initctl start local-git-daemon
```

Em outro Sistema Operacional, talvez você queira usar o `xinetd`, um script em seu sistema `sysvinit`, ou qualquer outra solução — contanto que você tenha o serviço Git rodando e monitorado de alguma forma.

A seguir, você tem que configurar seu servidor Gitis para permitir o acesso não autenticado aos repositórios Git. Se você adicionar uma seção para cada repositório, você pode especificar quais você quer que seu serviço Git tenha permissão de leitura. Se quiser permitir o acesso para o seu projeto `iphone` usando o protocolo Git, acrescente no final do arquivo `gitis.conf`:

```
1 [repo iphone_project]
2 daemon = yes
```

Quando você fizer um commit e um push neste projeto, seu serviço em execução deve começar a servir os pedidos para o projeto a qualquer um que tenha acesso à porta 9418 em seu servidor.

Se você decidir não usar Gitis, mas quer configurar um servidor Git, você terá que executar isso em cada projeto que você deseje que o serviço Git disponibilize:

```
1 $ cd /path/to/project.git
2 $ touch git-daemon-export-ok
```

A presença desse arquivo diz ao Git que ele pode servir esse projeto sem autenticação.

Gitis também pode controlar que projetos o GitWeb irá mostrar. Primeiro, você precisa adicionar algo como o seguinte no arquivo `/etc/gitweb.conf`:

```
1 $projects_list = "/home/git/gitis/projects.list";
2 $projectroot = "/home/git/repositories";
3 $export_ok = "git-daemon-export-ok";
4 @git_base_url_list = ('git://gitserver');
```

Você pode controlar quais projetos GitWeb permite aos usuários navegar, adicionando ou removendo uma configuração gitweb no arquivo de configuração Gitis. Por exemplo, se você deseja que o projeto do iPhone apareça no GitWeb, você pode definir a opção `repo` como abaixo:

```
1 [repo iphone_project]
2 daemon = yes
3 gitweb = yes
```

Agora, se você fizer um commit e um push neste projeto, GitWeb automaticamente começará a mostrar seu projeto `iphone`.

4.10 Git no Servidor - Git Hospedado

Git Hospedado

Se você não quer passar por todo o trabalho envolvido na configuração de seu próprio servidor Git, você tem várias opções para hospedar seus projetos Git em um site externo de hospedagem dedicado. Estes sites oferecem uma série de vantagens: um site de hospedagem geralmente é rápido de configurar e facilita a criação de projetos e não envolve a manutenção do servidor ou monitoramento. Mesmo que você configure e execute seu próprio servidor internamente, você ainda pode querer usar um site público de hospedagem para o seu código fonte aberto — é geralmente mais fácil para a comunidade de código aberto encontrá-lo e ajudá-lo.

Nos dias de hoje, você tem um grande número de opções de hospedagem para escolher, cada um com diferentes vantagens e desvantagens. Para ver uma lista atualizada, veja a seguinte página:

1 <https://git.wiki.kernel.org/index.php/GitHosting>

Como não podemos cobrir todos eles, e porque eu trabalho em um deles, vamos usar esta seção para ensiná-lo a criar uma conta e um novo projeto no GitHub. Isso lhe dará uma ideia do que está envolvido no processo.

GitHub é de longe o maior site open source de hospedagem Git e também é um dos poucos que oferecem hospedagens públicas e privadas para que você possa manter o seu código aberto ou privado no mesmo lugar. Na verdade, nós usamos o GitHub privado para colaborar com esse livro.

GitHub

GitHub é um pouco diferente do que a maioria dos sites de hospedagem de código na maneira que gerencia projetos. Em vez de ser baseado principalmente no projeto, GitHub é centrado no usuário. Isso significa que quando eu hospedar meu projeto `grit` no GitHub, você não vai encontrá-lo em `github.com/grit` mas em `github.com/schacon/grit`. Não há versão canônica de qualquer projeto, o que permite que um projeto possa se deslocar de um usuário para outro se o primeiro autor abandonar o projeto.

GitHub também é uma empresa comercial que cobra para contas que mantêm repositórios privados, mas qualquer um pode rapidamente obter uma conta gratuita para hospedar tantos projetos de código aberto quanto quiser. Nós vamos passar rapidamente sobre como isso é feito.

Criando uma Conta de Usuário

A primeira coisa que você precisa fazer é criar uma conta de usuário gratuita. Se você visitar a página de Preços e Inscrição em <http://github.com/plans> e clicar no botão “Sign Up” na conta gratuita (ver figura 4-2), você é levado à página de inscrição.



Figure 4-2. A página de planos do GitHub.

Aqui você deve escolher um nome de usuário que ainda não foi utilizada no sistema e digitar um endereço de e-mail que será associado com a conta e uma senha (veja a Figura 4-3).

The image shows the GitHub sign-up form. It has a title "Sign up (log in)" and fields for "Username", "Email Address", "Password", and "Confirm Password". Below these is a section for "SSH Public Key" with a link to "explain ssh keys" and a note: "Please enter one key only. You may add more later. This field is not required to sign up." At the bottom, there's a box with text: "You're signing up for the free plan. If you have any questions please email support." and "By signing up, you agree to the Terms of Service, Privacy, and Refund policies." with a button "I agree, sign me up!".

Figure 4-3. O formulário de inscrição do GitHub.

Este é um bom momento para adicionar sua chave pública SSH também. Mostramos como gerar uma nova chave antes, na seção “Gerando Sua Chave Pública SSH”. Copie o conteúdo da chave pública, e cole-o na caixa de texto “SSH Public Key”. Clicando no link “explain ssh keys” irá mostrar instruções detalhadas sobre como fazê-lo em todos os principais sistemas operacionais. Clicando no botão “I agree, sign me up” levará você ao painel principal de seu novo usuário (ver Figura 4-4).

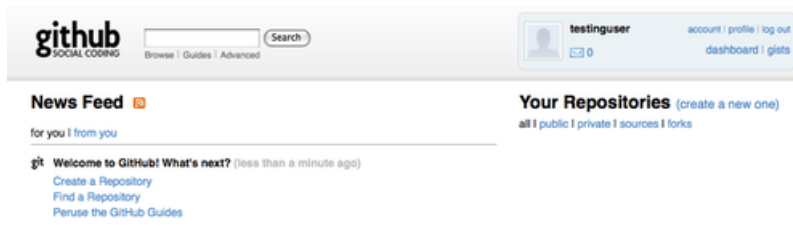


Figure 4-4. O painel principal do usuário do GitHub.

Em seguida, você pode criar um novo repositório.

Criando um Novo Repositório

Comece clicando no link “create a new one” ao lado de seus repositórios no painel do usuário. Você é levado para um formulário para criação de um novo repositório (ver Figura 4-5).

The image shows the 'Create a New Repository' form on GitHub. It has a title 'Create a New Repository' and a subtitle 'Create a new empty repository into which you can push your local git repo.' Below this is a 'NOTE' in red text: 'NOTE: If you intend to push a copy of a repository that is already hosted on GitHub, then you should fork it instead.' The form contains three input fields: 'Project Name' (with 'iphone_project' entered), 'Description' (with 'iphone project for our mobile group' entered), and 'Homepage URL'. Below these fields is a section for 'Who has access to this repository?' with two radio buttons: 'Anyone' (selected) and 'Only people you invite' (with a link to 'Upgrade your plan to create more private repositories!'). At the bottom is a 'Create Repository' button.

Figure 4-5. Criando um novo repositório no GitHub.

Tudo o que você realmente tem que fazer é fornecer um nome de projeto, mas você também pode adicionar uma descrição. Quando terminar, clique no botão “Create Repository”. Agora você tem um novo repositório no GitHub (ver Figura 4-6).

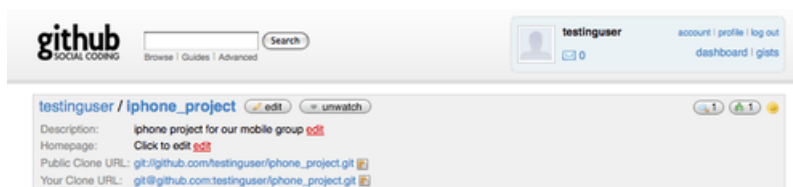


Figure 4-6. Informações de um projeto do GitHub.

Já que você não tem nenhum código ainda, GitHub irá mostrar-lhe instruções de como criar um novo projeto, fazer um push de um projeto Git existente, ou importar um projeto de um repositório Subversion público (ver Figura 4-7).



Figure 4-7. Instrução para novos repositórios.

Estas instruções são semelhantes ao que nós já vimos. Para inicializar um projeto se já não é um projeto Git, você usa

```
1 $ git init
2 $ git add .
3 $ git commit -m 'initial commit'
```

Quando você tem um repositório Git local, adicione GitHub como um remoto e faça um push do branch master:

```
1 $ git remote add origin git@github.com:testinguser/iphone_project.git
2 $ git push origin master
```

Agora seu projeto está hospedado no GitHub, e você pode dar a URL para quem você quiser compartilhar seu projeto. Neste caso, é `http://github.com/testinguser/iphone_project`. Você também pode ver a partir do cabeçalho em cada uma das páginas do seu projeto que você tem duas URLs Git (ver Figura 4-8).

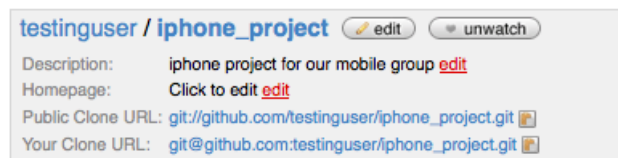


Figure 4-8. Cabeçalho do projeto com uma URL pública e outra privada.

A URL pública é uma URL Git somente leitura sobre a qual qualquer um pode clonar o projeto. Sinta-se a vontade para dar essa URL e postá-la em seu site ou qualquer outro lugar.

A URL privada é uma URL para leitura/gravação baseada em SSH que você pode usar para ler ou escrever apenas se tiver a chave SSH privada associada a chave pública que você carregou para o seu usuário. Quando outros usuários visitarem esta página do projeto, eles não vão ver a URL privada.

Importando do Subversion

Se você tem um projeto Subversion público existente que você deseja importar para o Git, GitHub muitas vezes pode fazer isso por você. Na parte inferior da página de instruções há um link para importação do Subversion. Se você clicar nele, você verá um formulário com informações sobre o processo de importação e uma caixa de texto onde você pode colar a URL do seu projeto Subversion público (ver Figura 4-9).

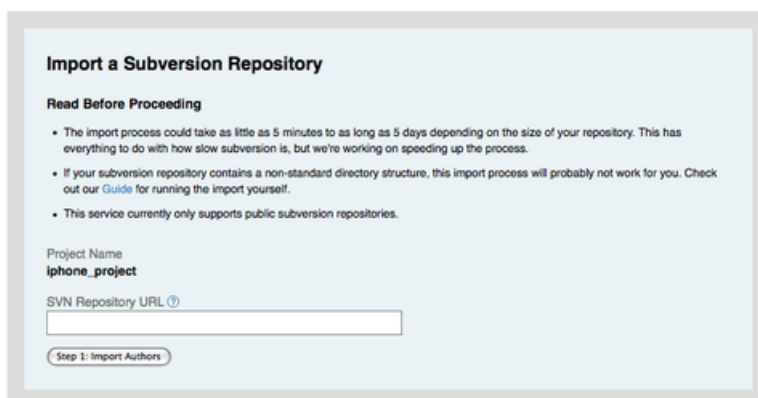
The image shows a web form titled "Import a Subversion Repository". Below the title is a section "Read Before Proceeding" with three bullet points: "The import process could take as little as 5 minutes to as long as 5 days depending on the size of your repository. This has everything to do with how slow subversion is, but we're working on speeding up the process.", "If your subversion repository contains a non-standard directory structure, this import process will probably not work for you. Check out our [Guide](#) for running the import yourself.", and "This service currently only supports public subversion repositories." Below this is a "Project Name" field with the text "iphone_project" entered. Underneath is an "SVN Repository URL" label followed by a text input field. At the bottom of the form is a button labeled "Step 1: Import Authors".

Figure 4-9. Interface de importação do Subversion.

Se o seu projeto é muito grande, fora do padrão, ou privado, esse processo provavelmente não vai funcionar para você. No Capítulo 7, você vai aprender como fazer a importação de projetos mais complicados manualmente.

Adicionando Colaboradores

Vamos adicionar o resto da equipe. Se John, Josie, e Jessica se inscreverem no GitHub, e você quer dar a eles permissão de escrita em seu repositório, você pode adicioná-los ao seu projeto como colaboradores. Isso permitirá que eles façam pushes a partir de suas chaves públicas.

Clique no botão “editar” no cabeçalho do projeto ou na guia Admin no topo do projeto para chegar à página de administração do seu projeto GitHub (ver Figura 4-10).

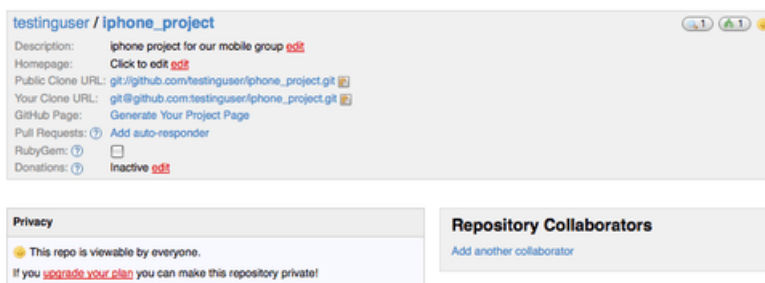


Figure 4-10. Página de administração do GitHub.

Para dar a outro usuário acesso de escrita ao seu projeto, clique no link “Add another collaborator”. Uma nova caixa de texto aparece, no qual você pode digitar um nome de usuário. Conforme você digita, um ajudante aparece, mostrando a você nomes de usuários possíveis. Quando você encontrar o usuário correto, clique no botão “Add” para adicionar o usuário como colaborador em seu projeto (ver Figura 4-11).

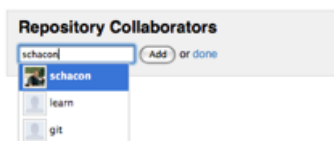


Figure 4-11. Adicionando um colaborador a seu projeto.

Quando você terminar de adicionar colaboradores, você deve ver uma lista deles na caixa de colaboradores do repositório (ver Figura 4-12).

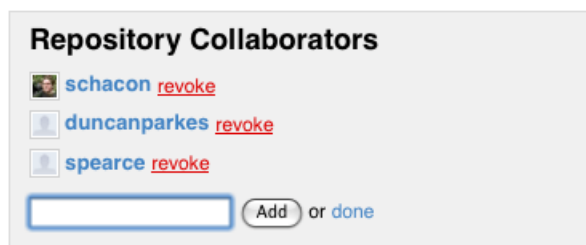


Figure 4-12. Uma lista de colaboradores em seu projeto.

Se você precisar revogar acesso às pessoas, você pode clicar no link “revoke”, e seus acessos de escrita serão removidos. Para projetos futuros, você também pode copiar grupos de colaboradores ao copiar as permissões de um projeto existente.

Seu Projeto

Depois de fazer um push no seu projeto ou tê-lo importado do Subversion, você tem uma página principal do projeto que é algo como Figura 4-13.

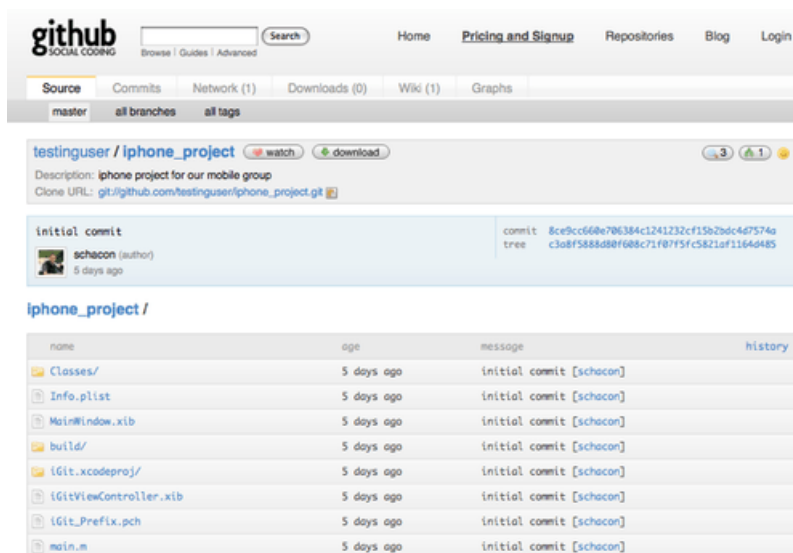


Figure 4-13. A página principal do projeto no GitHub.

Quando as pessoas visitam o seu projeto, elas veem esta página. Ela contém guias para diferentes aspectos de seus projetos. A guia Commits mostra uma lista de commits em ordem cronológica inversa, semelhante à saída do comando `git log`. A guia Network mostra todas as pessoas que criaram um fork do seu projeto e contribuíram para nele. A guia Downloads permite que você faça upload de arquivos binários e crie links para tarballs e versões compactadas de todas as versões de seu projeto. A guia Wiki fornece uma wiki onde você pode escrever documentação ou outras informações sobre o projeto. A guia Graphs tem algumas visualizações e estatísticas de contribuições sobre o seu projeto. A guia Source mostra uma listagem de diretório principal de seu projeto e processa automaticamente o arquivo README abaixo se você tiver um. Essa guia também mostra uma caixa com a informação do commit mais recente.

Criando Forks de Projetos

Se você quiser contribuir para um projeto existente para o qual você não tem permissão de push, GitHub incentiva a utilização de forks do projeto. Quando você acessar uma página de um projeto que parece interessante e você quiser fazer alguma mudança nele, você pode clicar no botão “fork” no cabeçalho do projeto para que o GitHub copie o projeto para o seu usuário para que você possa editá-lo.

Dessa forma, os projetos não têm que se preocupar com a adição de usuários como colaboradores para dar-lhes acesso de escrita. As pessoas podem criar um fork de um projeto e fazer um push nele, e o mantenedor do projeto principal pode fazer um pull dessas mudanças, adicionando-as como remotos e fazendo um merge no seu projeto.

Para fazer um fork de um projeto, visite a página do projeto (neste caso, `mojombo/chronic`) e clique no botão “fork” no cabeçalho (ver Figura 4-14).

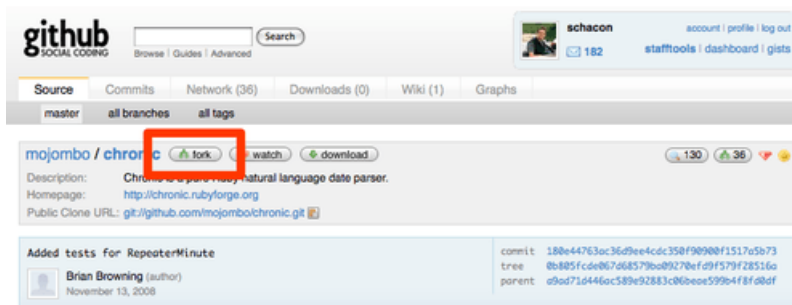


Figure 4-14. Obtenha uma cópia de um projeto, que pode ser modificado, clicando no botão “fork”.

Depois de alguns segundos, você é levado à página do seu novo projeto, o que indica que este projeto é um fork de outro (ver Figura 4-15).

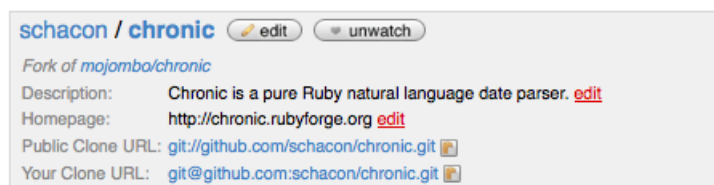


Figure 4-15. Seu fork de um projeto.

Sumário do GitHub

Isso é tudo o que nós vamos falar acerca do GitHub, mas é importante notar o quão rápido você pode fazer tudo isso. Você pode criar uma conta, adicionar um novo projeto, e fazer um push nele em questão de minutos. Se o seu projeto é de código aberto, você também terá uma grande comunidade de desenvolvedores, que agora têm visibilidade de seu projeto e podem fazer forks e ajudar contribuindo. No mínimo, isso pode ser uma maneira de usar o Git e experimentá-lo rapidamente.

4.11 Git no Servidor - Sumário

Sumário

Você tem várias opções para obter um repositório Git remoto instalado e funcionando para que você possa colaborar com outras pessoas ou compartilhar seu trabalho.

Executando o seu próprio servidor lhe dá um grande controle e permite que você execute o servidor dentro do seu próprio firewall, mas tal servidor geralmente requer uma boa quantidade de seu tempo para configurar e manter. Se você colocar seus dados em um servidor hospedado, é fácil de configurar e manter, no entanto, você tem que ser capaz de manter o seu código em servidores de outra pessoa, e algumas organizações não permitem isso.

Deve ser fácil determinar qual a solução ou a combinação de soluções mais adequada para você e sua organização.

Capítulo 5 - Git Distribuído

Agora que você tem um repositório Git remoto configurado como um ponto para todos os desenvolvedores compartilharem seu código, e você está familiarizado com os comandos básicos do Git em um fluxo de trabalho local, você vai ver como utilizar alguns dos fluxos de trabalho distribuídos que o Git lhe proporciona.

Neste capítulo, você verá como trabalhar com Git em um ambiente distribuído como colaborador e integrador. Ou seja, você vai aprender como contribuir código para um projeto da melhor forma possível para você e para os responsáveis do projeto, e também como manter um projeto de sucesso com uma grande quantidade de desenvolvedores.

5.1 Git Distribuído - Fluxos de Trabalho Distribuídos

Fluxos de Trabalho Distribuídos

Ao contrário de Sistemas de Controle de Versão Centralizados (CVCSs), a natureza distribuída do Git lhe permite ser muito mais flexível na forma como os desenvolvedores podem colaborar em projetos. Nos sistemas centralizados, cada desenvolvedor é um nó trabalhando de forma mais ou menos igual em um hub (centralizador). No Git, no entanto, cada desenvolvedor é potencialmente nó e hub — ou seja, cada desenvolvedor pode contribuir com código para outros repositórios e ao mesmo tempo pode manter um repositório público em que outros podem basear seu trabalho e que eles podem contribuir. Isso abre uma vasta gama de possibilidades de fluxo de trabalho para o seu projeto e sua equipe, então eu vou cobrir alguns paradigmas mais comuns que se aproveitam dessa flexibilidade. Vou passar as vantagens e possíveis desvantagens de cada configuração, você pode escolher uma para usar ou combinar características de cada uma.

Fluxo de Trabalho Centralizado

Com sistemas centralizados normalmente há apenas um modelo de colaboração, centralizado. Um hub central, ou repositório, pode aceitar o código, e todos sincronizam o seu trabalho com ele. Vários desenvolvedores são nós — consumidores do hub — e sincronizam em um lugar único (ver Figura 5-1).

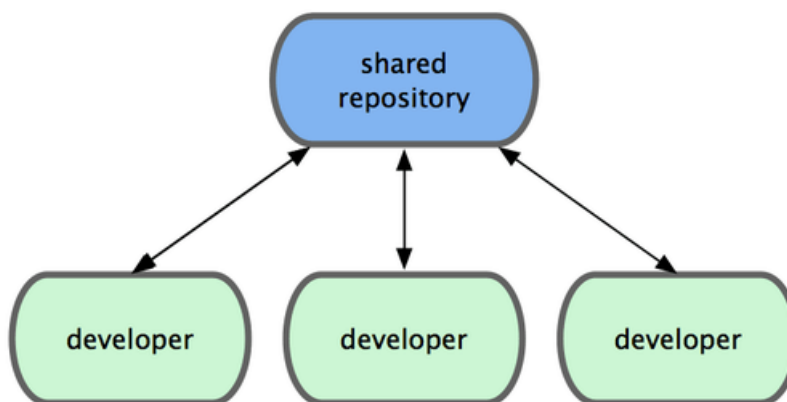


Figura 5-1. Fluxo de Trabalho Centralizado.

Isto significa que se dois desenvolvedores clonam o hub e ambos fazem alterações, o primeiro desenvolvedor a dar push de suas alterações pode fazê-lo sem problemas. O segundo desenvolvedor deve fazer merge do trabalho do primeiro antes de dar push, de modo a não substituir as alterações do primeiro desenvolvedor. Isso vale para o Git assim como o Subversion (ou qualquer CVCS), e este modelo funciona perfeitamente no Git.

Se você tem uma equipe pequena ou se já estão confortáveis com um fluxo de trabalho centralizado em sua empresa ou equipe, você pode facilmente continuar usando esse fluxo de trabalho com o Git. Basta criar um único repositório, e dar a todos em sua equipe acesso para dar push; o Git não deixará os usuários sobrescreverem uns aos outros. Se um desenvolvedor clona, faz alterações, e depois tenta dar push enquanto outro desenvolvedor já deu push com novas alterações nesse meio tempo, o servidor irá rejeitar as novas alterações. Ele será informado que está tentando dar push que não permite fast-forward (avanço rápido) e que não será capaz de fazê-lo até que baixe as últimas alterações e faça merge. Esse fluxo de trabalho é atraente para muita gente porque é um paradigma que muitos estão familiarizados e confortáveis.

Fluxo de Trabalho do Gerente de Integração

Como o Git permite que você tenha múltiplos repositórios remotos, é possível ter um fluxo de trabalho onde cada desenvolvedor tem acesso de escrita a seu próprio repositório público e acesso de leitura a todos os outros. Este cenário, muitas vezes inclui um repositório canônico que representa o projeto “oficial”. Para contribuir com esse projeto, você cria o seu próprio clone público do projeto e guarda suas alterações nele. Depois, você pode enviar uma solicitação para o responsável do projeto principal para puxar as suas alterações. Eles podem adicionar o repositório como um repositório remoto, testar localmente as suas alterações, fazer merge em um branch e propagá-las para o repositório principal. O processo funciona da seguinte maneira (veja Figura 5-2):

1. O mantenedor do projeto propaga as alterações para seu repositório público.
2. O desenvolvedor clona o repositório e faz alterações.
3. O desenvolvedor dá push das alterações para sua própria cópia pública.

4. O desenvolvedor envia um e-mail pedindo para o mantenedor puxar as alterações (pull request).
5. O mantenedor adiciona o repositório do desenvolvedor como um repositório remoto e faz merge das alterações localmente.
6. O mantenedor dá push das alterações mescladas para o repositório principal.

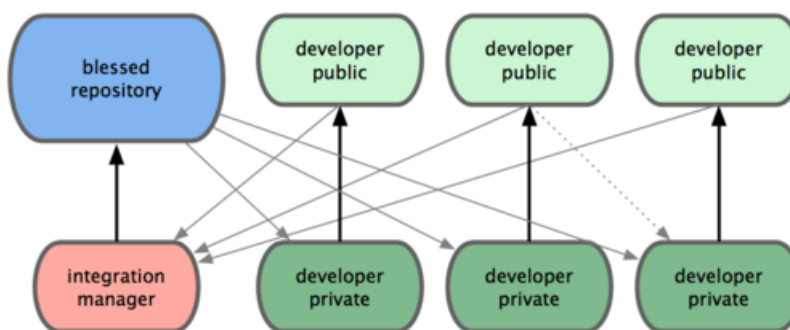


Figura 5-2. Fluxo de trabalho de Gerente de Integração.

Este é um fluxo de trabalho muito comum em sites como GitHub, onde é fácil de fazer uma fork (forquilha ou bifurcação, porque o histórico não-linear é uma árvore) de um projeto e dar push das suas alterações para que todos possam ver. Uma das principais vantagens desta abordagem é que você pode continuar a trabalhar, e o mantenedor do repositório principal pode puxar as alterações a qualquer momento. Desenvolvedores não tem que esperar o projeto incorporar as suas mudanças — cada um pode trabalhar em seu próprio ritmo.

Fluxo de Trabalho de Ditador e Tenentes

Esta é uma variante de um fluxo de trabalho de múltiplos repositórios. É geralmente usado por grandes projetos com centenas de colaboradores. Um exemplo famoso é o kernel do Linux. Vários gerentes de integração são responsáveis por certas partes do repositório, eles são chamados tenentes (liutenants). Todos os tenentes têm um gerente de integração conhecido como o ditador benevolente (benevolent dictator). O repositório do ditador benevolente serve como repositório de referência a partir do qual todos os colaboradores devem se basear. O processo funciona assim (veja Figura 5-3):

1. Desenvolvedores regulares trabalham em seu topic branch e baseiam seu trabalho sobre o master. O branch master é o do ditador.
2. Tenentes fazem merge dos topic branches dos desenvolvedores em seus master.
3. O ditador faz merge dos branches master dos tenentes em seu branch master.
4. O ditador dá push das alterações de seu master para o repositório de referência para que os desenvolvedores possam fazer rebase em cima dele.

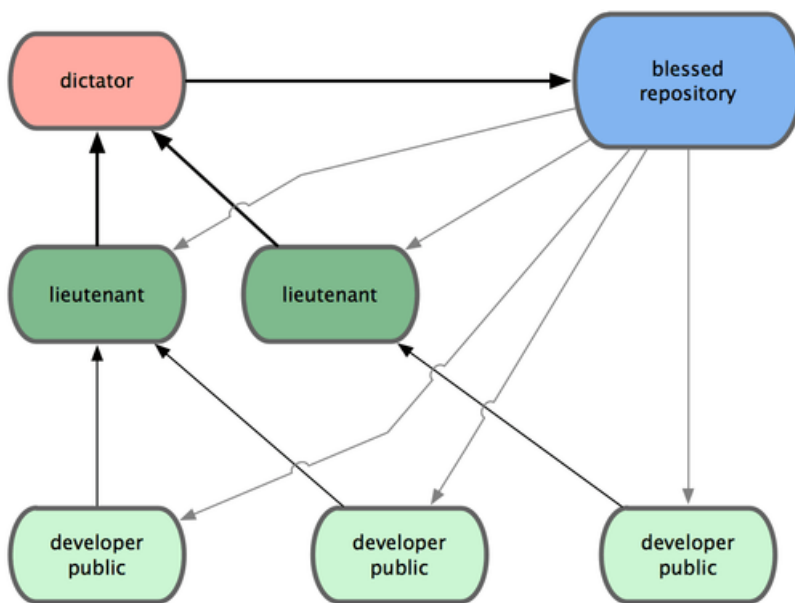


Figura 5-3. Fluxo de Trabalho do Ditador Benevolente.

Este tipo de fluxo de trabalho não é comum, mas pode ser útil em projetos muito grandes ou em ambientes altamente hierárquicos, porque ele permite ao líder do projeto (o ditador) delegar grande parte do trabalho e recolher grandes subconjuntos do código em vários pontos antes de integrar eles.

Estes são alguns fluxos de trabalho comumente utilizados que são possíveis com um sistema distribuído como Git, mas você pode ver que muitas variações são possíveis para se adequar ao seu fluxo de trabalho particular. Agora que você pode (espero eu) determinar qual a combinação de fluxo de trabalho pode funcionar para você, eu vou cobrir alguns exemplos mais específicos de como realizar os principais papéis que compõem os diferentes fluxos.

5.2 Git Distribuído - Contribuindo Para Um Projeto

Contribuindo Para Um Projeto

Você conhece os diferentes fluxos de trabalho que existem, e você deve ter um conhecimento muito bom do uso essencial do Git. Nesta seção, você aprenderá sobre alguns protocolos comuns para contribuir para um projeto.

A principal dificuldade em descrever este processo é o grande número de variações sobre a forma como ele é feito. Porque Git é muito flexível, as pessoas podem e trabalham juntos de muitas maneiras diferentes, é problemático descrever como você deve contribuir com um projeto — cada projeto é um pouco diferente. Algumas das variáveis envolvidas são o tamanho da base de contribuintes ativos, fluxo de trabalho escolhido, permissões e possivelmente o método de contribuição externa.

A primeira variável é o tamanho da base de contribuintes ativos. Quantos usuários estão ativamente contribuindo código para este projeto, e com que frequência? Em muitos casos, você tem dois ou três desenvolvedores com alguns commits por dia ou menos ainda em projetos mais adormecidos. Para as empresas ou projetos realmente grandes, o número de desenvolvedores poderia ser na casa dos milhares, com dezenas ou mesmo centenas de patches chegando todo dia. Isto é importante porque, com mais e mais desenvolvedores, você encontra mais problemas para assegurar que seu código se aplica corretamente ou pode ser facilmente incorporado. As alterações que você faz poderão se tornar obsoletas ou severamente danificadas pelo trabalho que foi incorporado enquanto você estava trabalhando ou quando as alterações estavam esperando para ser aprovadas e aplicadas. Como você pode manter o seu código consistentemente atualizado e suas correções válidas?

A próxima variável é o fluxo de trabalho em uso para o projeto. É centralizado, com cada desenvolvedor tendo acesso igual de escrita ao repositório principal? O projeto tem um mantenedor ou gerente de integração que verifica todos os patches? Todos os patches são revisados e aprovados? Você está envolvido nesse processo? Há um sistema de tenentes, e você tem que enviar o seu trabalho a eles?

A questão seguinte é o seu acesso de commit. O fluxo de trabalho necessário para contribuir para um projeto é muito diferente se você tiver acesso de gravação para o projeto do que se você não tiver. Se você não tem acesso de gravação, como é que o projeto prefere aceitar contribuições? Há uma política sobre essa questão? Quanto trabalho você contribui de cada vez? Com que frequência você contribui?

Todas estas questões podem afetar a forma que você contribui para um projeto e quais fluxos de trabalho são melhores para você. Falarei sobre os aspectos de cada um deles em uma série de casos de uso, movendo-se do mais simples ao mais complexo, você deve ser capaz de construir os fluxos de trabalho específicos que você precisa na prática a partir desses exemplos.

Diretrizes de Commit

Antes de você começar a olhar para os casos de uso específicos, aqui está uma breve nota sobre as mensagens de commit. Ter uma boa orientação para a criação de commits e aderir a ela torna o trabalho com Git e a colaboração com os demais muito mais fácil. O projeto Git fornece um documento que estabelece uma série de boas dicas para a criação de commits para submeter patches — você pode lê-lo no código-fonte do Git no arquivo `Documentation/SubmittingPatches`.

Primeiro, você não quer submeter nenhum problema de espaços em branco (whitespace errors). Git oferece uma maneira fácil de verificar isso — antes de fazer commit, execute `git diff --check` para listar possíveis problemas de espaços em branco. Aqui está um exemplo, onde a cor vermelha no terminal foi substituída por Xs:


```
1 $ git diff --check
2 lib/simplegit.rb:5: trailing whitespace.
3 +   @git_dir = File.expand_path(git_dir)XX
4 lib/simplegit.rb:7: trailing whitespace.
5 + XXXXXXXXXXXXX
6 lib/simplegit.rb:26: trailing whitespace.
7 +   def command(git_cmd)XXXX
```

Se você executar esse comando antes de fazer commit, você pode dizer se você está prestes a fazer commit com problemas de espaços em branco que pode incomodar outros desenvolvedores.

Em seguida, tente fazer de cada commit um conjunto de mudanças (changeset) logicamente separado. Se você puder, tente fazer suas alterações fáceis de lidar — não fique programando um fim de semana inteiro para resolver cinco problemas diferentes e então fazer um commit maciço com todas as alterações na segunda-feira. Mesmo se você não faça commits durante o fim de semana, use a “staging area” na segunda-feira para dividir o seu trabalho em pelo menos um commit por problema, com uma mensagem útil em cada commit. Se algumas das alterações modificarem o mesmo arquivo, tente usar `git add --patch` para preparar partes de arquivos (coberto em detalhes no Capítulo 6). O snapshot do projeto no final do branch é idêntico se você fizer um ou cinco commits, desde que todas as mudanças tenham sido adicionadas em algum momento. Então tente tornar as coisas mais fáceis para seus colegas desenvolvedores quando eles tiverem que revisar as suas alterações. Essa abordagem também torna mais fácil puxar ou reverter algum dos changesets se você precisar mais tarde. O *Capítulo 6* descreve uma série de truques úteis do Git para reescrever o histórico e adicionar as alterações de forma interativa — use essas ferramentas para ajudar a construir um histórico limpo e compreensível.

Uma última coisa que precisamos ter em mente é a mensagem de commit. Adquirir o hábito de criar mensagens de commit de qualidade torna o uso e colaboração com Git muito mais fácil. Como regra geral, as suas mensagens devem começar com uma única linha com não mais que cerca de 50 caracteres e que descreve o changeset de forma concisa, seguido por uma linha em branco e uma explicação mais detalhada. O projeto Git exige que a explicação mais detalhada inclua a sua motivação para a mudança e que contraste a sua implementação com o comportamento anterior — esta é uma boa orientação a seguir. Também é uma boa ideia usar o tempo verbal presente da segunda pessoa nestas mensagens. Em outras palavras, utilizar comandos. Ao invés de “Eu adicionei testes para” ou “Adicionando testes para”, usar “Adiciona testes para”. Esse é um modelo originalmente escrito por Tim Pope em `tpope.net`:

```
1 Breve (50 caracteres ou menos) resumo das mudanças
2
3 Texto explicativo mais detalhado, se necessário. Separe em linhas de
4 72 caracteres ou menos. Em alguns contextos a primeira linha é
5 tratada como assunto do e-mail e o resto como corpo. A linha em branco
6 separando o resumo do corpo é crítica (a não ser que o corpo não seja
7 incluído); ferramentas como rebase podem ficar confusas se você usar
8 os dois colados.
9
10 Parágrafos adicionais vem após linhas em branco.
11
12 - Tópicos também podem ser usados
13
14 - Tipicamente um hífen ou asterisco é utilizado para marcar tópicos,
15 precedidos de um espaço único, com linhas em branco entre eles, mas
16 convenções variam nesse item
```

Se todas as suas mensagens de commit parecerem com essa, as coisas serão bem mais fáceis para você e para os desenvolvedores que trabalham com você. O projeto Git tem mensagens de commit bem formatadas — eu encorajo você a executar `git log --no-merges` lá para ver como um histórico de commits bem formatados se parece.

Nos exemplos a seguir e durante a maior parte desse livro, para ser breve eu não formatarei mensagens dessa forma; em vez disso eu usarei a opção `-m` em `git commit`. Faça como eu digo, não faça como eu faço.

Pequena Equipe Privada

A configuração mais simples que você encontrará é um projeto privado com um ou dois desenvolvedores. Por privado, eu quero dizer código fechado — não acessível para o mundo de fora. Você e os outros desenvolvedores todos têm acesso para dar push de alterações para o repositório.

Nesse ambiente, você pode seguir um fluxo de trabalho similar ao que você usaria com Subversion ou outro sistema centralizado. Você ainda tem as vantagens de coisas como commit offline e facilidade de lidar com branches e merges, mas o fluxo de trabalho pode ser bastante similar; a principal diferença é que merges acontecem no lado do cliente ao invés de no servidor durante o commit. Vamos ver como isso fica quando dois desenvolvedores começam a trabalhar juntos com um repositório compartilhado. O primeiro desenvolvedor, John, clona o repositório, faz alterações e realiza o commit localmente. (Eu estou substituindo as mensagens de protocolo com ... nesses exemplos para diminuí-los um pouco)

```
1 # Máquina do John
2 $ git clone john@github:simplegit.git
3 Initialized empty Git repository in /home/john/simplegit/.git/
4 ...
5 $ cd simplegit/
6 $ vim lib/simplegit.rb
7 $ git commit -am 'removed invalid default value'
8 [master 738ee87] removed invalid default value
9 1 files changed, 1 insertions(+), 1 deletions(-)
```

O segundo desenvolvedor, Jessica, faz a mesma coisa — clona o repositório e faz um commit:

```
1 # Máquina da Jessica
2 $ git clone jessica@github:simplegit.git
3 Initialized empty Git repository in /home/jessica/simplegit/.git/
4 ...
5 $ cd simplegit/
6 $ vim TODO
7 $ git commit -am 'add reset task'
8 [master fbff5bc] add reset task
9 1 files changed, 1 insertions(+), 0 deletions(-)
```

Agora, Jessica dá push do trabalho dela para o servidor:

```
1 # Máquina da Jessica
2 $ git push origin master
3 ...
4 To jessica@github:simplegit.git
5 1edee6b..fbff5bc master -> master
```

John tenta dar push de suas alterações também:

```
1 # Máquina do John
2 $ git push origin master
3 To john@github:simplegit.git
4 ! [rejected] master -> master (non-fast forward)
5 error: failed to push some refs to 'john@github:simplegit.git'
```

John não consegue dar push porque Jessica deu push de outras alterações nesse meio tempo. Isso é especialmente importante de entender se você está acostumado com Subversion porque você irá perceber que dois desenvolvedores não editaram o mesmo arquivo. Enquanto o Subversion faz merge automaticamente no servidor se duas pessoas não editarem o mesmo arquivo, no Git você deve sempre fazer merge dos commits localmente. John tem que baixar as alterações de Jessica e fazer merge antes de poder dar push das alterações:

```
1 $ git fetch origin
2 ...
3 From john@github:simplegit
4 + 049d078...fbff5bc master -> origin/master
```

Nesse ponto, o repositório local de John se parece com a Figura 5-4.

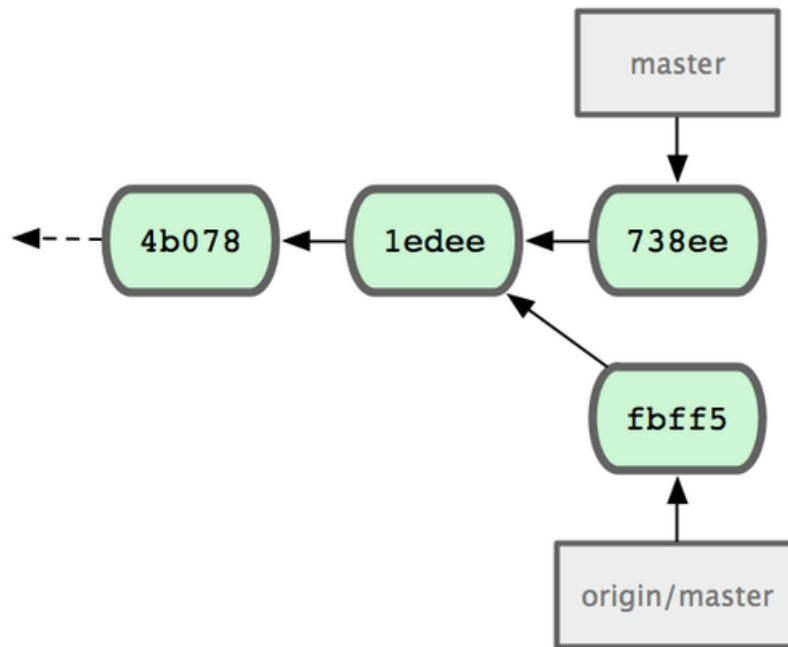


Figura 5-4. Repositório inicial de John.

John tem uma referência para as alterações que Jessica fez, mas ele tem que fazer merge com seu próprio trabalho para poder dar push das suas próprias alterações:

```
1 $ git merge origin/master
2 Merge made by recursive.
3  TODO | 1 +
4  1 files changed, 1 insertions(+), 0 deletions(-)
```

O merge funciona — o histórico de commits do John agora se parece com a Figura 5-5.

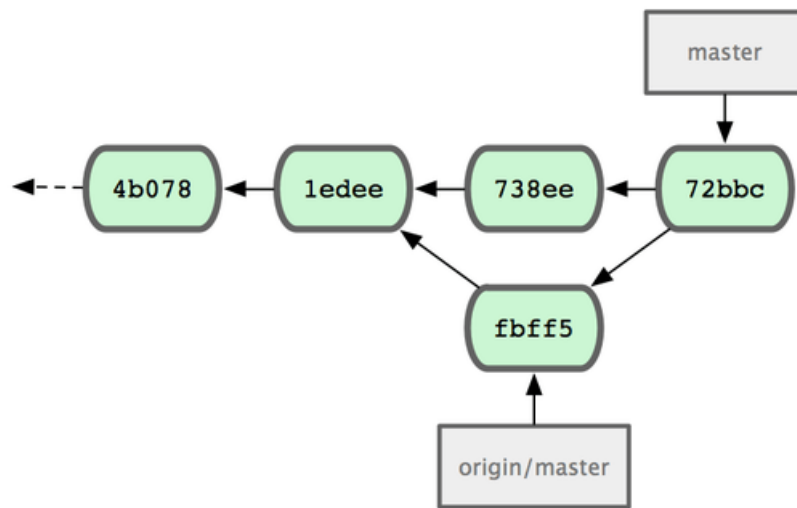


Figura 5-5. Repositório do John depois de fazer merge em origin/master.

Agora John pode testar seu código para ter certeza que ele ainda funciona, e então ele pode dar push de seu novo trabalho mesclado para o servidor:

```
1 $ git push origin master
2 ...
3 To john@github.com:simplegit.git
4    fbff5bc..72bbc59  master -> master
```

Finalmente, o histórico de commits de John se parece com a Figura 5-6.

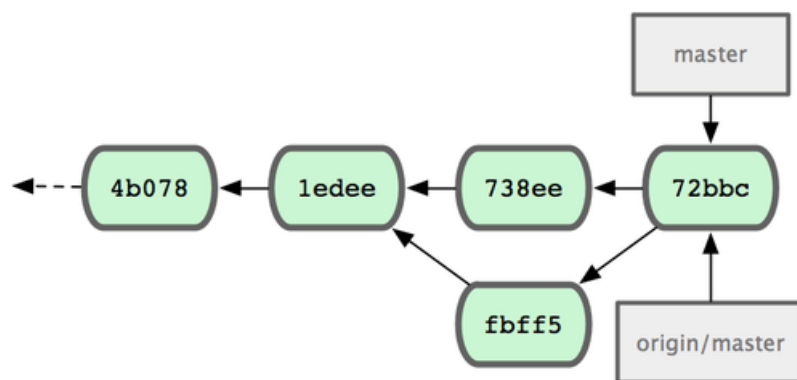


Figura 5-6. O histórico de John depois de ter dado push para o servidor de origem (remote origin).

Nesse meio tempo, Jessica tem trabalhado em um “topic branch”. Ela criou um “topic branch” chamado `issue54` e fez três commits naquele branch. Ela não baixou as alterações de John ainda, então o histórico de commits dela se parece com a Figura 5-7.

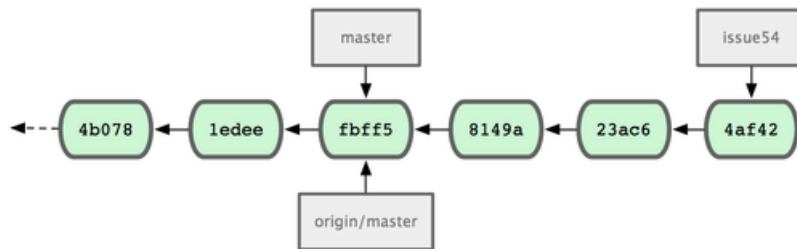


Figura 5-7. Histórico inicial de commits de Jessica.

Jessica quer sincronizar com John, então ela faz fetch:

```

1  # Máquina da Jessica
2  $ git fetch origin
3  ...
4  From jessica@github:implegit
5    fbff5bc..72bbc59 master    -> origin/master

```

Isso baixa o trabalho que John tinha empurrado (push). o histórico de Jessica agora se parece com a Figura 5-8.

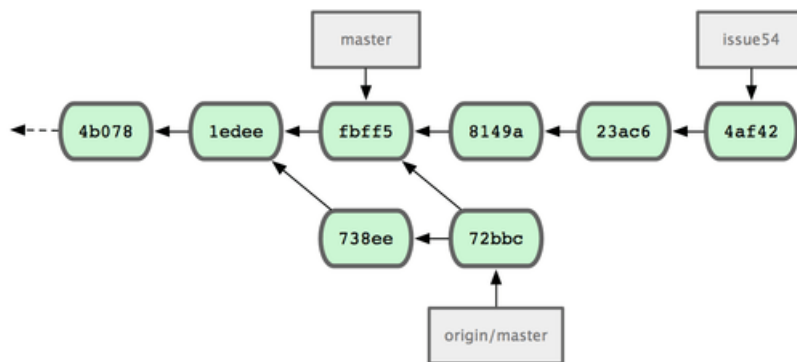


Figura 5-8. O histórico de Jessica depois de baixar as alterações de John.

Jessica pensa que seu “topic branch” está pronto, mas ela quer saber com o que ela precisa fazer merge para poder dar push de seu trabalho. Ela executa `git log` para descobrir:

```

1  $ git log --no-merges origin/master ^issue54
2  commit 738ee872852dfaa9d6634e0dea7a324040193016
3  Author: John Smith <jsmith@example.com>
4  Date:   Fri May 29 16:01:27 2009 -0700
5
6     removed invalid default value

```

Agora, Jessica pode fazer merge de seu topic branch no branch master, fazer merge do trabalho de John (`origin/master`) em seu branch master e finalmente dar push para o servidor. Primeiro, ela troca para o seu branch master para integrar todo esse trabalho:

```

1 $ git checkout master
2 Switched to branch "master"
3 Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.

```

Ela pode fazer merge de origin/master ou issue54 primeiro — ambos são upstream (atualizados), então a ordem não importa. O snapshot final deve ser idêntico, não importa a ordem que ela escolher; apenas o histórico será levemente diferente. Ela escolhe fazer merge de issue54 primeiro:

```

1 $ git merge issue54
2 Updating fbff5bc..4af4298
3 Fast forward
4  README          |      1 +
5  lib/simplegit.rb |      6 +++++-
6  2 files changed, 6 insertions(+), 1 deletions(-)

```

Não acontece nenhum problema; como você pode ver, foi um simples fast-forward. Agora Jessica faz merge do trabalho de John (origin/master):

```

1 $ git merge origin/master
2 Auto-merging lib/simplegit.rb
3 Merge made by recursive.
4  lib/simplegit.rb |      2 +-
5  1 files changed, 1 insertions(+), 1 deletions(-)

```

Tudo mesclou perfeitamente, e o histórico de Jessica agora se parece com a Figura 5-9.

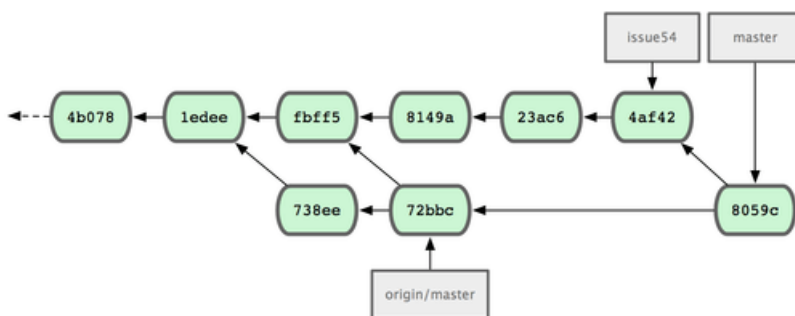


Figura 5-9. O histórico de Jessica depois de mesclar as alterações de John.

Agora origin/master é acessível através do branch master de Jessica, então ela pode perfeitamente dar push (assumindo que John não deu push com novas alterações nesse meio tempo):

```
1 $ git push origin master
2 ...
3 To jessica@github.com:simplegit.git
4    72bbc59..8059c15  master -> master
```

Cada desenvolvedor fez alguns commits e integrou o trabalho do outro com sucesso; veja Figura 5-10.

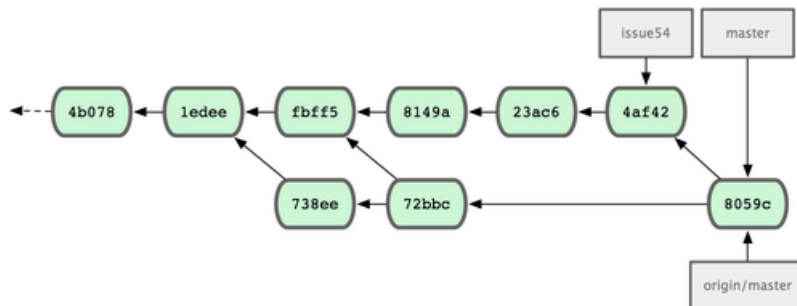


Figura 5-10. O histórico de Jessica depois de dar push para o servidor.

Esse é um dos fluxos de trabalho mais simples. Você trabalha um pouco, geralmente em um topic branch, e faz merge em seu branch `master` quando ele estiver pronto para ser integrado. Quando você quiser compartilhar seu trabalho, você faz merge em seu próprio branch `master`, baixa as últimas alterações do servidor com `fetch` e faz merge de `origin/master` se tiver sido alterado, e então dá push para o branch `master` no servidor. A ordem é semelhante ao mostrado na Figura 5-11.



Figura 5-11. Sequencia geral dos eventos para um fluxo de trabalho simples para Git com múltiplos desenvolvedores.

Equipe Privada Gerenciada

Nesse cenário, você verá os papéis de contribuinte em grupo privado maior. Você aprenderá como trabalhar em um ambiente onde pequenos grupos colaboram em funcionalidades e então essas contribuições por equipe são integradas por outro grupo.

Vamos dizer que John e Jessica estão trabalhando juntos em uma funcionalidade, enquanto Jessica e Josie estão trabalhando em outra. Nesse caso a empresa está usando um fluxo de trabalho com um gerente de integração onde o trabalho de cada grupo é integrado por apenas alguns engenheiros e o branch master do repositório principal pode ser atualizado apenas por esses engenheiros. Nesse cenário, todo o trabalho é feito em equipe e atualizado junto pelos integradores.

Vamos seguir o fluxo de trabalho de Jessica enquanto ela trabalha em suas duas funcionalidades, colaborando em paralelo com dois desenvolvedores diferentes nesse ambiente. Assumindo que ela já clonou seu repositório, ela decide trabalhar no `featureA` primeiro. Ela cria o novo branch para a funcionalidade e faz algum trabalho lá:

```
1 # Máquina da Jessica
2 $ git checkout -b featureA
3 Switched to a new branch "featureA"
4 $ vim lib/simplegit.rb
5 $ git commit -am 'add limit to log function'
6 [featureA 3300904] add limit to log function
7 1 files changed, 1 insertions(+), 1 deletions(-)
```

Nesse ponto, ela precisa compartilhar seu trabalho com John, então ela dá push dos commits de seu branch `featureA` para o servidor. Jessica não tem acesso para dar push no branch `master` — apenas os integradores tem — então ela dá push das alterações para outro branch para poder colaborar com John:

```
1 $ git push origin featureA
2 ...
3 To jessica@github:simplegit.git
4 * [new branch]      featureA -> featureA
```

Jessica avisa John por e-mail que ela deu push de algum trabalho em um branch chamado `featureA` e ele pode olhar ele agora. Enquanto ela espera pelo retorno de John, Jessica decide começar a trabalhar no `featureB` com Josie. Para começar, ela inicia um novo feature branch (branch de funcionalidade), baseando-se no branch `master` do servidor:

```
1 # Máquina da Jessica
2 $ git fetch origin
3 $ git checkout -b featureB origin/master
4 Switched to a new branch "featureB"
```

Agora, Jessica faz dois commits para o branch `featureB`:

```
1 $ vim lib/simplegit.rb
2 $ git commit -am 'made the ls-tree function recursive'
3 [featureB e5b0fdc] made the ls-tree function recursive
4 1 files changed, 1 insertions(+), 1 deletions(-)
5 $ vim lib/simplegit.rb
6 $ git commit -am 'add ls-files'
7 [featureB 8512791] add ls-files
8 1 files changed, 5 insertions(+), 0 deletions(-)
```

O repositório de Jessica se parece com a Figura 5-12.

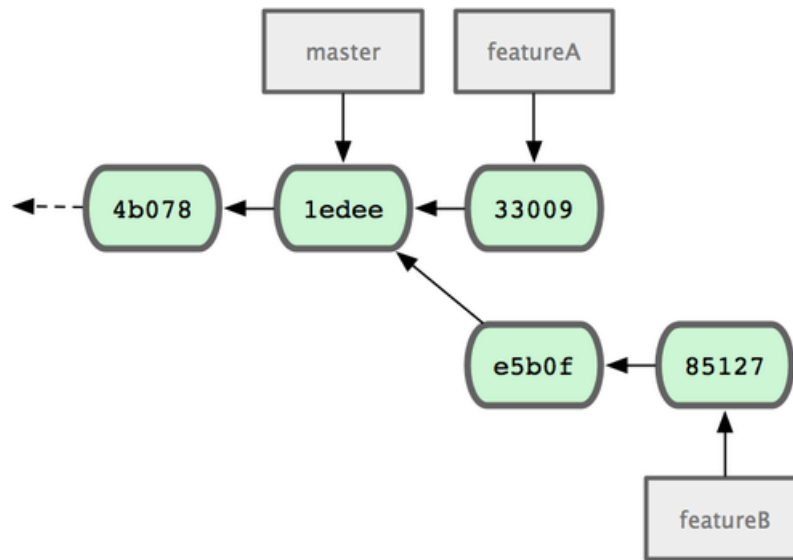


Figura 5-12. O histórico de commits inicial de Jessica.

Ela está pronta para fazer push de seu trabalho, mas recebe um e-mail de Josie avisando que ela já fez um trabalho inicial e que está no servidor no branch `featureBee`. Primeiro Jessica precisa mesclar essas alterações com suas próprias para que ela possa dar push de suas alterações para o servidor. Ela pode então baixar as alterações de Josie com `git fetch`:

```

1 $ git fetch origin
2 ...
3 From jessica@github:simplegit
4 * [new branch]      featureBee -> origin/featureBee

```

Jessica pode agora fazer merge em seu trabalho com `git merge`:

```

1 $ git merge origin/featureBee
2 Auto-merging lib/simplegit.rb
3 Merge made by recursive.
4 lib/simplegit.rb | 4 ++++
5 1 files changed, 4 insertions(+), 0 deletions(-)

```

Há um pequeno problema — ela precisa dar push do trabalho mesclado em seu branch `featureB` para o branch `featureBee` no servidor. Ela pode fazer isso especificando o branch local seguido de dois pontos (:) seguido pelo branch remoto para o comando `git push`:

```
1 $ git push origin featureB:featureBee
2 ...
3 To jessica@github.com:simplegit.git
4    fba9af8..cd685d1  featureB -> featureBee
```

Isso é chamado *refspec*. Veja o *Capítulo 9* para uma discussão mais detalhada dos refsspecs do Git e coisas diferentes que você pode fazer com eles.

Depois, John diz para Jessica que ele deu push de algumas alterações para o branch `featureA` e pede para ela verificá-las. Ela executa um `git fetch` para puxar essas alterações:

```
1 $ git fetch origin
2 ...
3 From jessica@github.com:simplegit
4    3300904..aad881d  featureA -> origin/featureA
```

Então, ela pode ver que essas alterações foram modificadas com `git log`:

```
1 $ git log origin/featureA ^featureA
2 commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
3 Author: John Smith <jsmith@example.com>
4 Date:   Fri May 29 19:57:33 2009 -0700
5
6     changed log output to 30 from 25
```

Finalmente, ela faz merge do trabalho de John em seu próprio branch `featureA`:

```
1 $ git checkout featureA
2 Switched to branch "featureA"
3 $ git merge origin/featureA
4 Updating 3300904..aad881d
5 Fast forward
6  lib/simplegit.rb | 10 +++++++-
7  1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica quer melhorar uma coisa, então ela faz um novo commit e dá push de volta para o servidor:

```

1 $ git commit -am 'small tweak'
2 [featureA ed774b3] small tweak
3 1 files changed, 1 insertions(+), 1 deletions(-)
4 $ git push origin featureA
5 ...
6 To jessica@github:simplegit.git
7 3300904..ed774b3 featureA -> featureA

```

O histórico de commit de Jessica agora parece com a Figura 5-13.

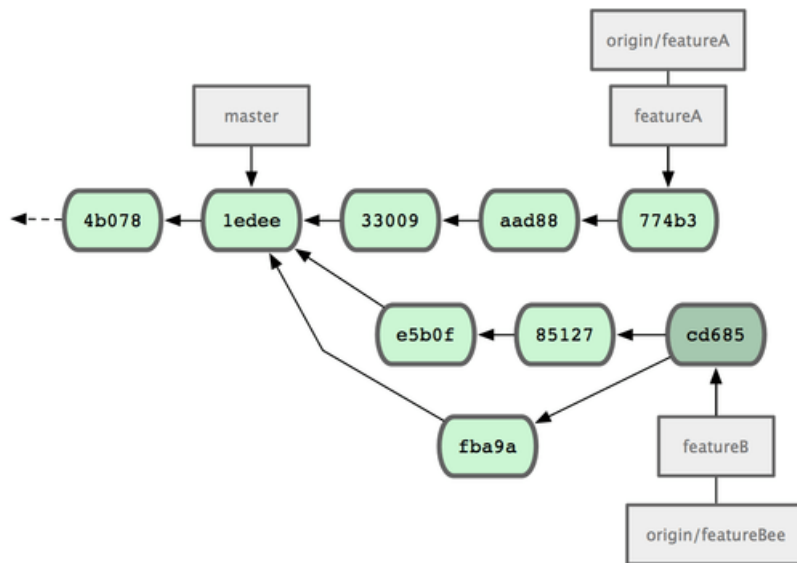


Figura 5-13. O histórico de Jessica depois do commit no feature branch.

Jessica, Josie e John informam os integradores que os branches **featureA** e **featureBee** no servidor estão prontos para integração na linha principal. Depois que eles integrarem esses branches na linha principal, baixar (**fetch**) irá trazer os novos commits mesclados, fazendo o histórico de commit ficar como na Figura 5-14.

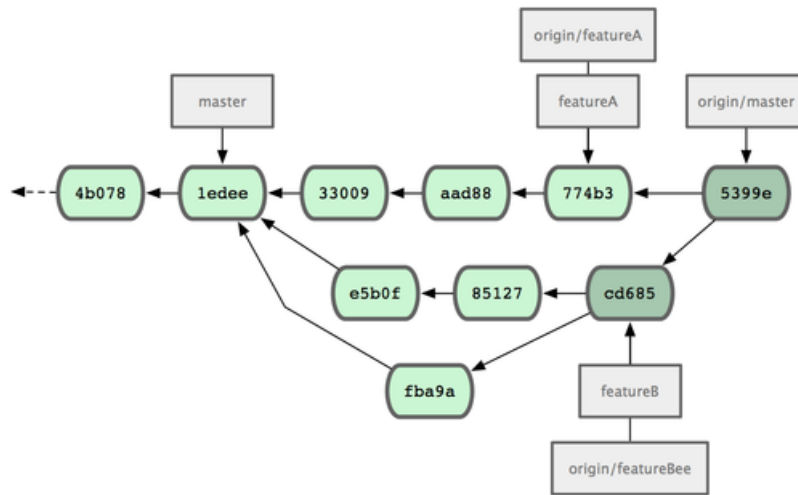


Figura 5-14. O histórico de Jessica depois de mesclar ambos topic branches.

Muitos grupos mudam para Git por causa da habilidade de ter múltiplas equipes trabalhando em paralelo, mesclando diferentes linhas de trabalho mais tarde. A habilidade de partes menores de uma equipe colaborar via branches remotos sem necessariamente ter que envolver ou impedir a equipe inteira é um grande benefício do Git. A sequência para o fluxo de trabalho que você viu aqui é como mostrado na Figura 5-15.

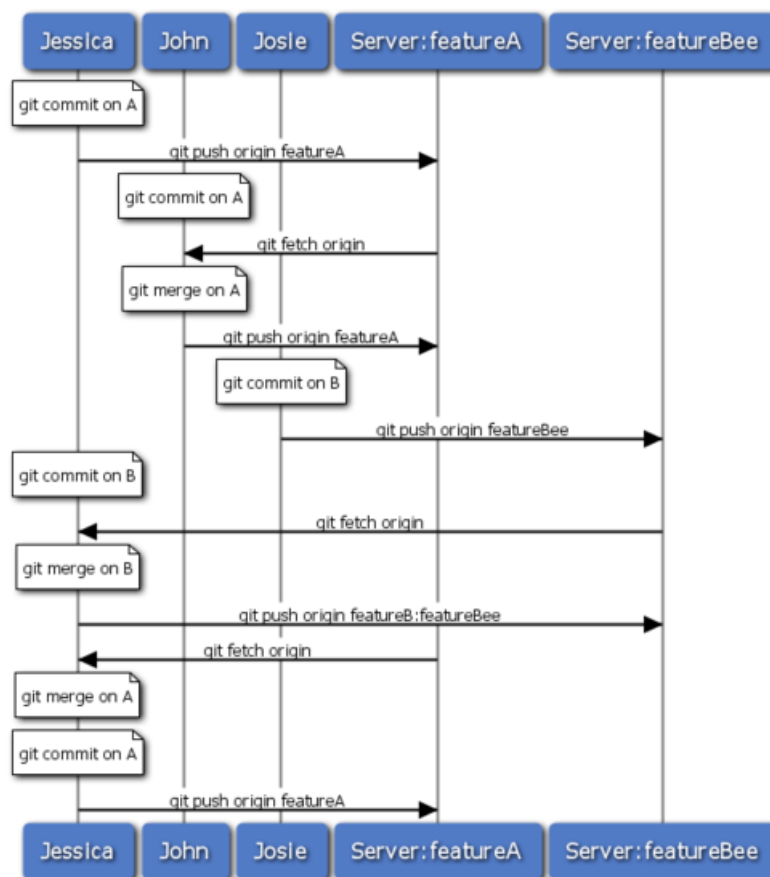


Figure 5-15. Sequencia básica desse fluxo de trabalho de equipe gerenciada.

Pequeno Projeto Público

Contribuindo para projetos públicos é um pouco diferente. Você tem que ter permissões para atualizar diretamente branches no projeto, você tem que passar o trabalho para os mantenedores de uma outra forma. O primeiro exemplo descreve como contribuir via forks em hosts Git que suportam criação simples de forks. Os sites de hospedagem `repo.or.cz` e Github ambos suportam isso, e muitos mantenedores de projetos esperam esse tipo de contribuição. A próxima seção lida com projetos que preferem aceitar patches contribuídos por e-mail.

Primeiro, você provavelmente irá querer clonar o repositório principal, criar um topic branch para o patch ou séries de patch que você planeja contribuir e você fará seu trabalho lá. A sequencia se parece com isso:

```
1 $ git clone (url)
2 $ cd project
3 $ git checkout -b featureA
4 $ (trabalho)
5 $ git commit
6 $ (trabalho)
7 $ git commit
```

Você pode querer usar `rebase -i` para espremer seu trabalho em um único commit, ou reorganizar o trabalho nos commits para fazer o patch mais fácil de revisar para os mantenedores — veja o Capítulo 6 para mais informações sobre rebase interativo.

Quando seu trabalho no branch for finalizado e você está pronto para contribuir para os mantenedores, vá até a página do projeto original e clique no botão “Fork”, criando seu próprio fork gravável do projeto. Você então precisa adicionar a URL desse novo repositório como um segundo remoto (remote), nesse caso chamado `myfork`:

```
1 $ git remote add myfork (url)
```

Você precisa dar push de seu trabalho para ele. É mais fácil dar push do branch remoto que você está trabalhando para seu repositório, ao invés de fazer merge em seu branch `master` e dar push dele. A razão é que se o trabalho não for aceito ou é selecionado em parte, você não tem que rebobinar seu branch `master`. Se os mantenedores mesclam, fazem rebase ou “cherry-pick” (escolhem pequenos pedaços) do seu trabalho, você terá que eventualmente puxar de novo de qualquer forma:

```
1 $ git push myfork featureA
```

Quando seu trabalho tiver na sua fork, você precisa notificar o mantenedor. Isso é geralmente chamado pull request (requisição para ele puxar), e você pode gerar isso pelo website — GitHub tem um “pull request” que notifica automaticamente o mantenedor — ou executar `git request-pull` e enviar por e-mail a saída desse comando para o mantenedor do projeto manualmente.

O comando `request-pull` recebe como argumento um branch base no qual você quer que suas alterações sejam puxadas e a URL do repositório Git que você quer que eles puxem. Por exemplo, se Jessica quer enviar John um pull request e ela fez dois commits no `topic` branch que ela deu push, ela pode executar isso:


```

1 $ git request-pull origin/master myfork
2 The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
3   John Smith (1):
4       added a new function
5
6 are available in the git repository at:
7
8   git://githost/simplegit.git featureA
9
10 Jessica Smith (2):
11     add limit to log function
12     change log output to 30 from 25
13
14 lib/simplegit.rb | 10 ++++++++-
15 1 files changed, 9 insertions(+), 1 deletions(-)

```

O resultado pode ser enviado para o mantenedor - ele fala para eles de onde o trabalho foi baseado, resume os commits e mostra de onde puxar esse trabalho.

Em um projeto que você não é o mantenedor, é geralmente mais fácil ter um branch como master sempre sincronizar com origin/master e fazer seu trabalho em topic branches que você pode facilmente descartar se rejeitados. Tendo temas de trabalho isolados em topic branches também torna mais fácil fazer rebase de seu trabalho se o seu repositório principal tiver sido atualizado nesse meio tempo e seus commits não puderem ser aplicados de forma limpa. Por exemplo, se você quiser submeter um segundo tópico de trabalho para o projeto, não continue trabalhando no topic branch que você deu push por último — inicie de novo a partir do branch master do repositório principal:

```

1 $ git checkout -b featureB origin/master
2 $ (work)
3 $ git commit
4 $ git push myfork featureB
5 $ (email maintainer)
6 $ git fetch origin

```

Agora, cada um de seus tópicos é contido em um silo — similar a uma fila de patches — que você pode reescrever, fazer rebase e modificar sem os tópicos interferirem ou interdepender um do outro como na Figura 5-16.

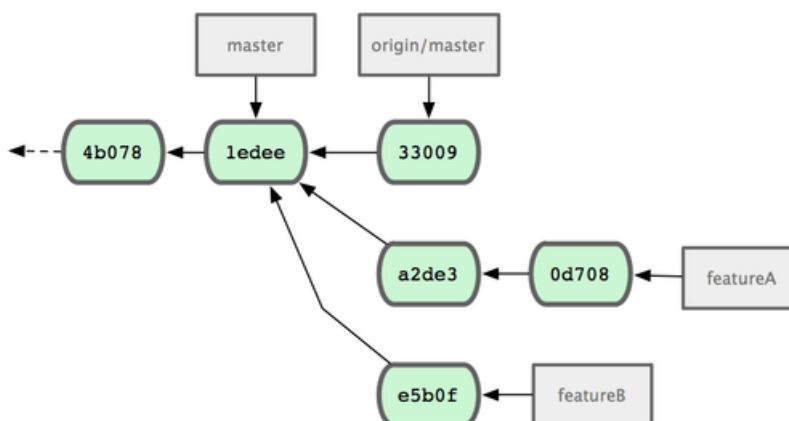


Figura 5-16. Histórico de commits inicial com trabalho do featureB.

Vamos dizer que o mantenedor do projeto tenha puxado um punhado de outros patches e testou seu primeiro branch, mas ele não mescla mais. Nesse caso, você pode tentar fazer rebase daquele branch em cima de `origin/master`, resolver os conflitos para o mantenedor e então submeter novamente suas alterações:

```
1 $ git checkout featureA
2 $ git rebase origin/master
3 $ git push -f myfork featureA
```

Isso sobrescreve seu histórico para parecer com a Figura 5-17.

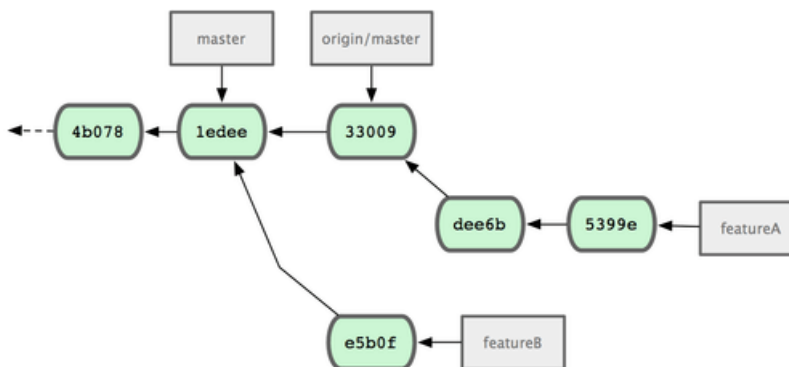


Figura 5-17. Histórico de commits depois do trabalho em featureA.

Já que você fez rebase de seu trabalho, você tem que especificar a opção `-f` para seu comando `push` poder substituir o branch `featureA` no servidor com um commit que não é descendente dele. Uma alternativa seria dar `push` desse novo trabalho para um branch diferente no servidor (talvez chamado `featureAv2`).

Vamos ver mais um cenário possível: o mantenedor olhou o trabalho em seu segundo branch e gostou do conceito, mas gostaria que você alterasse um detalhe de implementação. Você irá também tomar essa oportunidade para mover seu trabalho para ser baseado no branch `master` atual do

projeto. Você inicia um novo branch baseado no branch `origin/master` atual, coloca as mudanças de `featureB` lá, resolve qualquer conflito, faz a alteração na implementação e então dá `push` para o servidor como um novo branch:

```
1 $ git checkout -b featureBv2 origin/master
2 $ git merge --no-commit --squash featureB
3 $ (change implementation)
4 $ git commit
5 $ git push myfork featureBv2
```

A opção `--squash` pega todo o trabalho feito no branch mesclado e espreme ele em um non-merge commit em cima do branch que você está. A opção `--no-commit` fala para o Git não registrar um commit automaticamente. Isso permite a você introduzir as alterações de outro branch e então fazer mais alterações antes de registrar um novo commit.

Agora você pode enviar ao mantenedor uma mensagem informando que você fez as alterações requisitadas e eles podem encontrar essas mudanças em seu branch `featureBv2` (veja Figura 5-18).

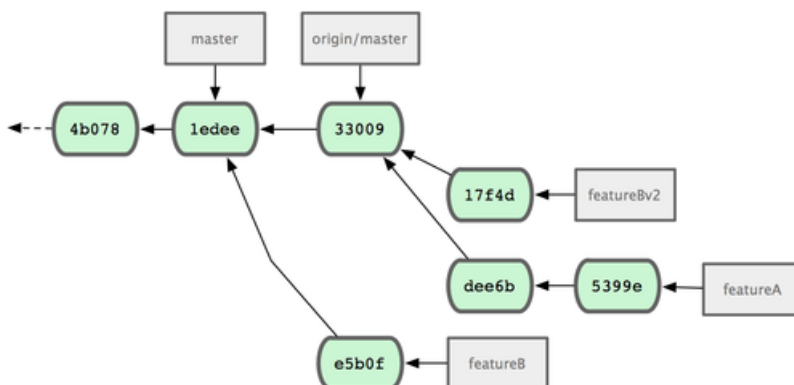


Figura 5-18. Histórico de commit depois do trabalho em `featureBv2`.

Grande Projeto Público

Muitos projetos maiores tem procedimentos estabelecidos para aceitar patches — você irá precisar verificar as regras específicas para cada projeto, porque eles irão diferir. Contudo, muitos projetos maiores aceitam patches via lista de discussão para desenvolvedores, então eu irei falar sobre esse exemplo agora.

O fluxo de trabalho é similar ao caso de uso anterior — você cria topic branches para cada série de patches que você trabalhar. A diferença é como você irá submeter eles para o projeto. Ao invés de fazer um fork do projeto e dar `push` das alterações para sua própria versão gravável, você gera versões em e-mail de cada série de commits e envia para a lista de discussão para desenvolvedores:

```

1 $ git checkout -b topicA
2 $ (work)
3 $ git commit
4 $ (work)
5 $ git commit

```

Agora você tem dois commits que você quer enviar para a lista de discussão. Você usa `git format-patch` para gerar os arquivos em formato mbox que você pode enviar por e-mail para a lista — transforma cada commit em uma mensagem de e-mail com a primeira linha da mensagem do commit como o assunto e o resto da mensagem mais o patch que o commit introduz como corpo. A coisa legal sobre isso é que aplicando um patch de um e-mail gerado com `format-patch` preserva todas as informações do commit, como você irá ver mais nas próximas seções quando você aplica esses commits:

```

1 $ git format-patch -M origin/master
2 0001-add-limit-to-log-function.patch
3 0002-changed-log-output-to-30-from-25.patch

```

O comando `format-patch` imprime o nome dos arquivos patch que ele cria. A opção `-M` fala para o Git verificar se há mudanças de nome. Os arquivos vão acabar parecendo assim:

```

1 $ cat 0001-add-limit-to-log-function.patch
2 From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
3 From: Jessica Smith <jessica@example.com>
4 Date: Sun, 6 Apr 2008 10:17:23 -0700
5 Subject: [PATCH 1/2] add limit to log function
6
7 Limit log functionality to the first 20
8
9 ---
10 lib/simplegit.rb |    2 +-
11 1 files changed, 1 insertions(+), 1 deletions(-)
12
13 diff --git a/lib/simplegit.rb b/lib/simplegit.rb
14 index 76f47bc..f9815f1 100644
15 --- a/lib/simplegit.rb
16 +++ b/lib/simplegit.rb
17 @@ -14,7 +14,7 @@ class SimpleGit
18     end
19
20     def log(treeish = 'master')
21 -     command("git log #{treeish}")

```

```
22 +   command("git log -n 20 #{treeish}")
23   end
24
25   def ls_tree(treeish = 'master')
26   --
27 1.6.2.rc1.20.g8c5b.dirty
```

Você pode também editar esses arquivos de patch para adicionar mais informação para a lista de e-mail que você não quer que apareça na mensagem do commit. Se você adicionar o texto entre a linha com `—` e o início do patch (a linha `lib/simplegit.rb`), então desenvolvedores podem lê-la; mas aplicar o patch a exclui.

Para enviar por e-mail a uma lista de discussão, você pode ou colar o arquivo em seu programa de e-mails ou enviar por um programa em linha de comando. Colando o texto geralmente causa problemas de formatação, especialmente com clientes “expertos” que não preservam linhas em branco e espaços em branco de forma apropriada. Por sorte, Git fornece uma ferramenta que lhe ajuda a enviar um patch via Gmail, que por acaso é o agente de e-mail que eu uso; você pode ler instruções detalhadas para vários programas de e-mail no final do arquivo previamente mencionado `Documentation/SubmittingPatched` no código fonte do Git.

Primeiramente, você precisa configurar a seção `imap` em seu arquivo `~/.gitconfig`. Você pode definir cada valor separadamente com uma série de comandos `git config` ou você pode adicioná-los manualmente; mas no final, seu arquivo de configuração deve parecer mais ou menos assim:

```
1 [imap]
2   folder = "[Gmail]/Drafts"
3   host = imaps://imap.gmail.com
4   user = user@gmail.com
5   pass = p4ssw0rd
6   port = 993
7   sslverify = false
```

Se seu servidor IMAP não usa SSL, as últimas duas linhas provavelmente não serão necessárias e o valor do `host` será `imap://` ao invés de `imaps://`. Quando isso estiver configurado, você pode usar `git send-email` para colocar a série de patches em sua pasta Drafts (Rascunhos) no seu servidor IMAP:

```
1 $ git send-email *.patch
2 0001-added-limit-to-log-function.patch
3 0002-changed-log-output-to-30-from-25.patch
4 Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
5 Emails will be sent from: Jessica Smith <jessica@example.com>
6 Who should the emails be sent to? jessica@example.com
7 Message-ID to be used as In-Reply-To for the first email? y
```

Então, Git imprime um bocado de informação de log parecendo com isso para cada patch que você estiver enviando:

```
1 (mbox) Adding cc: Jessica Smith <jessica@example.com> from
2 \line 'From: Jessica Smith <jessica@example.com>'
3 OK. Log says:
4 Sendmail: /usr/sbin/sendmail -i jessica@example.com
5 From: Jessica Smith <jessica@example.com>
6 To: jessica@example.com
7 Subject: [PATCH 1/2] added limit to log function
8 Date: Sat, 30 May 2009 13:29:15 -0700
9 Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
10 X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
11 In-Reply-To: <y>
12 References: <y>
13
14 Result: OK
```

Nesse ponto, você deve poder ir a sua pasta de rascunhos, modificar o campo “Para” (To) para a lista de discussões que você está enviando o patch, possivelmente o campo CC para o mantenedor ou pessoa responsável por aquela seção, e enviar.

Resumo

Essa seção cobriu uma grande quantidade de fluxos de trabalho comuns para lidar com vários tipos bem diferentes de projetos Git que você provavelmente encontrará e introduziu algumas novas ferramentas para lhe ajudar a gerenciar esse processo. Nas seções seguintes, você irá ver como trabalhar no outro lado da moeda: mantendo um projeto Git. Você irá aprender como ser um ditador benevolente ou gerente de integração.

5.3 Git Distribuído - Mantendo Um Projeto

Mantendo Um Projeto

Além de saber como contribuir efetivamente para um projeto, você pode precisar saber como manter um. Isso pode consistir em aceitar e aplicar patches gerados via `format-patch` e enviados por e-mail para você, ou integrar alterações em branches remotos para repositórios que você adicionou como remotes do seu projeto. Se você mantém um repositório canônico ou quer ajudar verificando ou aprovando patches, você precisa saber como aceitar trabalho de uma forma que é a mais clara para os outros contribuintes e aceitável para você a longo prazo.

Trabalhando em Topic Branches

Quando você estiver pensando em integrar um novo trabalho, é geralmente uma boa ideia testá-lo em um topic branch — um branch temporário criado especificamente para testar o novo trabalho. Dessa forma, é fácil modificar um patch individualmente e deixá-lo se não tiver funcionando até que você tenha tempo de voltar para ele. Se você criar um nome de branch simples baseado no tema do trabalho que você irá testar, como `ruby_client` ou algo similarmente descritivo, você pode facilmente lembrar se você tem que abandoná-la por um tempo e voltar depois. O mantenedor do projeto Git tende a usar namespace nos branches — como `sc/ruby_client`, onde `sc` é uma forma reduzida para o nome da pessoa que contribui com o trabalho. Você deve lembrar que você pode criar um branch baseado no branch `master` assim:

```
1 $ git branch sc/ruby_client master
```

Ou, se você quer também mudar para o branch imediatamente, você pode usar a opção `checkout -b`:

```
1 $ git checkout -b sc/ruby_client master
```

Agora você está pronto para adicionar seu trabalho nesse topic branch e determinar se você quer mesclá-lo em seus branches de longa duração.

Aplicando Patches por E-mail

Se você receber um patch por e-mail que você precisa integrar em seu projeto, você precisa aplicar o patch em seu topic branch para avaliá-lo. Há duas formas de aplicar um patch enviado por e-mail: com `git apply` ou com `git am`.

Aplicando Um Patch Com `apply`

Se você recebeu o patch de alguém que o gerou com `git diff` ou o comando `diff` do Unix, você pode aplicá-lo com o comando `git apply`. Assumindo que você salvou o patch em `/tmp/patch-ruby-client.patch`, você pode aplicar o patch assim:

```
1 $ git apply /tmp/patch-ruby-client.patch
```

Isso modifica os arquivos em seu diretório de trabalho. É quase igual a executar o comando `patch -p1` para aplicar o patch, mas ele aceita menos correspondências nebulosas do que `patch`. Ele também cuida de adições, remoções e mudanças de nome de arquivos se tiverem sido descritos no formato do `git diff`, o que `patch` não faz. Finalmente, `git apply` segue um modelo “aplica tudo ou aborta tudo”, enquanto `patch` pode aplicar arquivos patch parcialmente, deixando seu diretório de trabalho em um estado estranho. `git apply` é no geral muito mais cauteloso que `patch`. Ele não cria um commit para você — depois de executá-lo, você deve adicionar e fazer commit das mudanças introduzidas manualmente.

Você pode também usar `git apply` para ver se um patch aplica corretamente antes de tentar aplicar ele de verdade — você pode executar `git apply --check` com o patch:

```
1 $ git apply --check 0001-seeing-if-this-helps-the-gem.patch
2 error: patch failed: ticgit.gemspec:1
3 error: ticgit.gemspec: patch does not apply
```

Se não tiver nenhuma saída, então o patch deverá ser aplicado corretamente. O comando também sai com um status diferente de zero se a verificação falhar, então você pode usá-lo em scripts se você quiser.

Aplicando um Patch com `am`

Se o contribuinte é um usuário Git e foi bondoso o suficiente para usar o comando `format-patch` para gerar seu patch, então seu trabalho é mais fácil porque o patch contém informação do autor e uma mensagem de commit para você. Se você puder, encoraje seus contribuintes a usar `format-patch` ao invés de `diff` para gerar patches para você. Você só deve ter que usar `git apply` para patches legados e coisas desse tipo.

Para aplicar o patch gerado por `format-patch`, você usa `git am`. Tecnicamente, `git am` foi feito para ler um arquivo mbox, que é um formato simples de texto puro para armazenar uma ou mais mensagens de e-mail em um único arquivo. Ele se parece com isso:

```
1 From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
2 From: Jessica Smith <jessica@example.com>
3 Date: Sun, 6 Apr 2008 10:17:23 -0700
4 Subject: [PATCH 1/2] add limit to log function
5
6 Limit log functionality to the first 20
```

Isso é o início da saída do comando `format-patch` que você viu na seção anterior. Isso é também um formato de e-mail mbox válido. Se alguém lhe enviou um patch por e-mail corretamente usando `git send-email` e você baixou no formato mbox, então você pode apontar aquele arquivo para o `git am`

e ele irá começar a aplicar todos os patches que ele ver. Se você executar um cliente de e-mail que pode salvar vários e-mails em um formato mbox, você pode salvar uma série inteira de patches em um arquivo e então usar o `git am` para aplicar todos de uma vez.

Entretanto, se alguém fez upload de um arquivo patch gerado via `format-patch` para um sistema de chamados ou algo similar, você pode salvar o arquivo localmente e então passar o arquivo salvo no seu disco para `git am` aplicar:

```
1 $ git am 0001-limit-log-function.patch
2 Applying: add limit to log function
```

Você pode ver que ele foi aplicado corretamente e automaticamente criou um novo commit para você. O autor é retirado dos cabeçalhos `From(De)` e `Date(Data)` do e-mail e a mensagem do commit é retirada dos campos `Subject (Assunto)` e `Corpo (body)` (antes do path) do e-mail. Por exemplo, se esse patch foi aplicado do exemplo de mbox que eu acabei de mostrar, o commit gerado irá parecer com isso:

```
1 $ git log --pretty=fuller -1
2 commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
3 Author:      Jessica Smith <jessica@example.com>
4 AuthorDate:  Sun Apr 6 10:17:23 2008 -0700
5 Commit:      Scott Chacon <schacon@gmail.com>
6 CommitDate:  Thu Apr 9 09:19:06 2009 -0700
7
8     add limit to log function
9
10    Limit log functionality to the first 20
```

A informação `Commit` indica a pessoa que aplicou o patch e a hora que foi aplicado. A informação `Author(autor)` é o indivíduo que originalmente criou o patch e quanto ela foi originalmente criada.

Mas é possível que o patch não aplique corretamente. Talvez seu branch principal tenha divergido muito do branch a partir do qual o patch foi feito, ou o patch depende de outro patch que você ainda não aplicou. Nesse caso, o `git am` irá falhar e perguntar o que você quer fazer:

```
1 $ git am 0001-seeing-if-this-helps-the-gem.patch
2 Applying: seeing if this helps the gem
3 error: patch failed: ticgit.gemspec:1
4 error: ticgit.gemspec: patch does not apply
5 Patch failed at 0001.
6 When you have resolved this problem run "git am --resolved".
7 If you would prefer to skip this patch, instead run "git am --skip".
8 To restore the original branch and stop patching run "git am --abort".
```

Esse comando coloca marcadores de conflito em qualquer arquivo que tenha problemas, muito parecido com um conflito das operações de merge ou rebase. Você resolve esse problema, da mesma forma — edita o arquivo e resolve o conflito, adiciona para a “staging area” e então executa `git am --resolved` para continuar com o próximo patch:

```
1 $ (fix the file)
2 $ git add ticgit.gemspec
3 $ git am --resolved
4 Applying: seeing if this helps the gem
```

Se você quer que o Git tente ser um pouco mais inteligente para resolver o conflito, você pode passar a opção `-3` para ele, que faz o Git tentar um three-way merge. Essa opção não é padrão porque ela não funciona se o commit que o patch diz que foi baseado não estiver em seu repositório. Se você tiver o commit — se o patch foi baseado em commit público — então a opção `-3` é geralmente muito melhor para aplicar um patch conflituoso:

```
1 $ git am -3 0001-seeing-if-this-helps-the-gem.patch
2 Applying: seeing if this helps the gem
3 error: patch failed: ticgit.gemspec:1
4 error: ticgit.gemspec: patch does not apply
5 Using index info to reconstruct a base tree...
6 Falling back to patching base and 3-way merge...
7 No changes -- Patch already applied.
```

Nesse caso, eu estava tentando aplicar um patch que eu já tinha aplicado. Sem a opção `-3`, ela parece como um conflito.

Se você estiver aplicando vários patches de uma mbox, você pode também executar o comando `am` com modo interativo, que para a cada patch que ele encontra ele pergunta se você quer aplicá-lo:

```
1 $ git am -3 -i mbox
2 Commit Body is:
3 -----
4 seeing if this helps the gem
5 -----
6 Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Isso é legal se você tem vários patches salvos, porque você pode ver o patch primeiro se você não lembra o que ele é, ou não aplicar o patch se você já tinha feito isso antes.

Quando todos os patches para seu tópico são aplicados e feitos commits em seu branch, você pode escolher se e como integrá-los em um branch de longa duração.

Fazendo Checkout Em Branches Remotos

Se sua contribuição veio de um usuário Git que configurou seu próprio repositório, deu push de várias alterações nele e então enviou uma URL para o repositório e o nome do branch remoto que as alterações estão, você pode adicioná-las como um remote e fazer merges localmente.

Por exemplo, se Jessica envia um e-mail dizendo que ela tem um nova funcionalidade no branch `ruby-client` do repositório dela, você pode testá-la adicionando o remote e fazendo checkout do branch localmente:

```
1 $ git remote add jessica git://github.com/jessica/myproject.git
2 $ git fetch jessica
3 $ git checkout -b rubyclient jessica/ruby-client
```

Se ela manda outro e-mail mais tarde com outro branch contendo outra nova funcionalidade, você pode fazer fetch e checkout porque você já tem o remote configurado.

Isso é mais importante se você estiver trabalhando com uma pessoa regularmente. Se alguém só tem um patch para contribuir de vez em quando, então aceitá-lo por e-mail gastaria menos tempo do que exigir que todo mundo rode seu próprio servidor e tenha que continuamente adicionar e remover remotes para pegar poucos patches. Você também provavelmente não vai querer ter centenas de remotes, cada um para alguém que contribuiu um ou dois patches. Entretanto, scripts e serviços de hospedagem podem tornar isso mais fácil — depende bastante de como você desenvolve e como os contribuintes desenvolvem.

A outra vantagem dessa abordagem é que você terá o histórico dos commits também. Embora você possa ter problemas de merge legítimos, você sabe onde eles são baseados em seu histórico; um `three-way merge` é padrão ao invés de ter que fornecer opção `-3` e esperar que o patch tenha sido gerado de um commit público que você tenha acesso.

Se você estiver trabalhando com a pessoa regularmente, mas ainda quer puxar deles dessa forma, você pode fornecer a URL do repositório remoto para o comando `git pull`. Isso faz um pull uma vez e não salva a URL como um remote:

```
1 $ git pull git://github.com/onetimeguy/project.git
2 From git://github.com/onetimeguy/project
3  * branch                HEAD          -> FETCH_HEAD
4 Merge made by recursive.
```

Determinando O Que É Introduzido

Agora você tem um topic branch que contém trabalho contribuído. Nesse ponto, você pode determinar o que você gostaria de fazer com isso. Essa seção revê alguns comandos para que você possa usá-los para revisar exatamente o que você estará introduzindo se você fizer merge em seu branch principal.

Geralmente ajuda conseguir uma revisão de todos os commits que estão nesse branch, mas que não estão em seu branch principal. Você pode excluir commits em seu branch master adicionando a opção `--not` antes do nome do branch. Por exemplo, se o contribuinte lhe envia dois patches, você cria um branch chamado `contrib` e aplica esses patches lá, você pode executar isso:

```
1 $ git log contrib --not master
2 commit 5b6235bd297351589efc4d73316f0a68d484f118
3 Author: Scott Chacon <schacon@gmail.com>
4 Date:   Fri Oct 24 09:53:59 2008 -0700
5
6     seeing if this helps the gem
7
8 commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
9 Author: Scott Chacon <schacon@gmail.com>
10 Date:   Mon Oct 22 19:38:36 2008 -0700
11
12     updated the gemspec to hopefully work better
```

Para ver quais mudanças cada commit introduz lembre que você pode passar a opção `-p` no `git log` e ele irá adicionar o diff introduzido em cada commit.

Para ver um diff completo do que irá acontecer se você fizer merge desse topic branch com outro branch, você pode ter que usar um truque estranho que consegue os resultados corretos. Você pode pensar em executar isso:

```
1 $ git diff master
```

Esse comando lhe dá um diff, mas é enganoso. Se seu branch master avançou desde que foi criado o topic branch a partir dele, então você terá resultados aparentemente estranhos. Isso acontece porque o Git compara diretamente o snapshot do último commit do topic branch que você está e o snapshot do último commit do branch master. Por exemplo, se você tiver adicionado uma linha em um arquivo

no branch `master`, uma comparação direta dos snapshots irá parecer que o topic branch irá remover aquela linha.

Se `master` é ancestral direto de seu topic branch, isso não é um problema; mas se os dois históricos estiverem divergido, o `diff` irá parecer que está adicionando todas as coisas novas em seu topic branch e removendo tudo único do branch `master`.

O que você realmente quer ver são as mudanças adicionadas no topic branch — o trabalho que você irá introduzir se fizer merge com `master`. Você faz isso mandando o Git comparar o último commit do seu topic branch com o primeiro ancestral comum que ele tem como o branch `master`.

Tecnicamente, você pode fazer isso descobrindo explicitamente qual é o ancestral comum e então executando seu `diff` nele:

```
1 $ git merge-base contrib master
2 36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
3 $ git diff 36c7db
```

Entretanto, isso não é conveniente, então o Git fornece outro atalho para fazer a mesma coisa: a sintaxe dos três pontos. No contexto do comando `diff`, você pode usar três pontos depois de outro branch para fazer um `diff` entre o último commit do branch que você está e seu ancestral comum com outro branch:

```
1 $ git diff master...contrib
```

Esse comando lhe mostra apenas o trabalho que o topic branch introduziu desde seu ancestral em comum com `master`. Essa é uma sintaxe muito útil de se lembrar.

Integrando Trabalho Contribuído

Quando todo o trabalho em seu topic branch estiver pronto para ser integrado em um branch mais importante, a questão é como fazê-lo. Além disso, que fluxo de trabalho você quer usar para manter seu projeto? Você tem várias opções, então eu cobrirei algumas delas.

Fluxos de Trabalho para Merge

Um fluxo de trabalho simples faz merge de seu trabalho em seu branch `master`. Nesse cenário, você tem um branch `master` que contém código estável. Quando você tiver trabalho em um topic branch que você fez ou que alguém contribuiu e você verificou, você faz merge no branch `master`, remove o topic branch e continua o processo. Se você tem um repositório com trabalho em dois branches chamados `ruby_client` e `php_client` que se parecem com a Figura 5-19 e faz primeiro merge de `ruby_client` e então de `php_client`, então seu histórico se parecerá como na Figura 5-20.

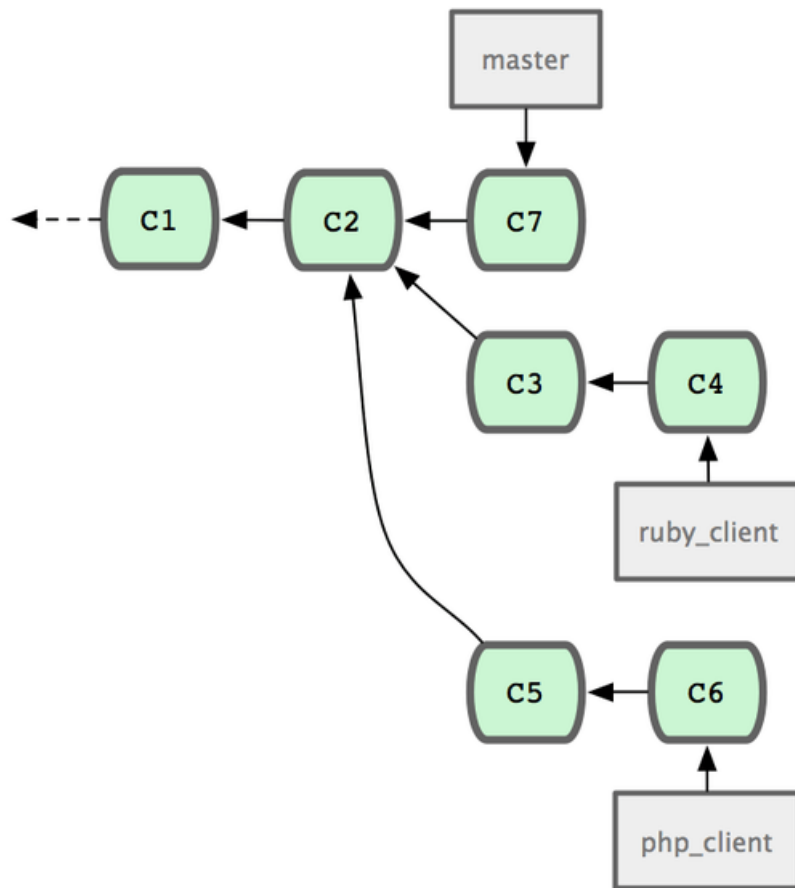


Figura 5-19. histórico com vários topic branches.

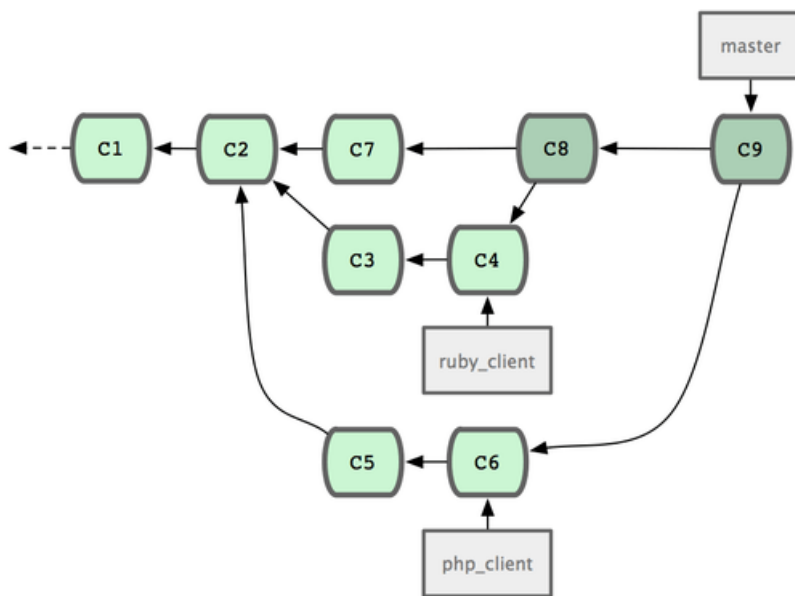


Figura 5-20. Depois de um merge de topic branches.

Isso é provavelmente o fluxo de trabalho mais simples, mas é problemático se você estiver lidando com repositórios ou projetos maiores.

Se você tiver mais desenvolvedores ou um projeto maior, você irá provavelmente querer usar pelo menos um ciclo de merge de duas fases. Nesse cenário você tem dois branches de longa duração, `master` e `develop`, dos quais você determina que `master` é atualizado só quando uma liberação bem estável é atingida e todo novo trabalho é integrado no branch `develop`. Você dá push regularmente de ambos os branches para o repositório público. Cada vez que você tiver um novo topic branch para fazer merge (Figura 5-21), você faz merge em `develop` (Figura 5-22); então, quando você criar uma tag o release, você avança (fast-forward) `master` para onde o agora estável branch `develop` estiver (Figura 5-23).

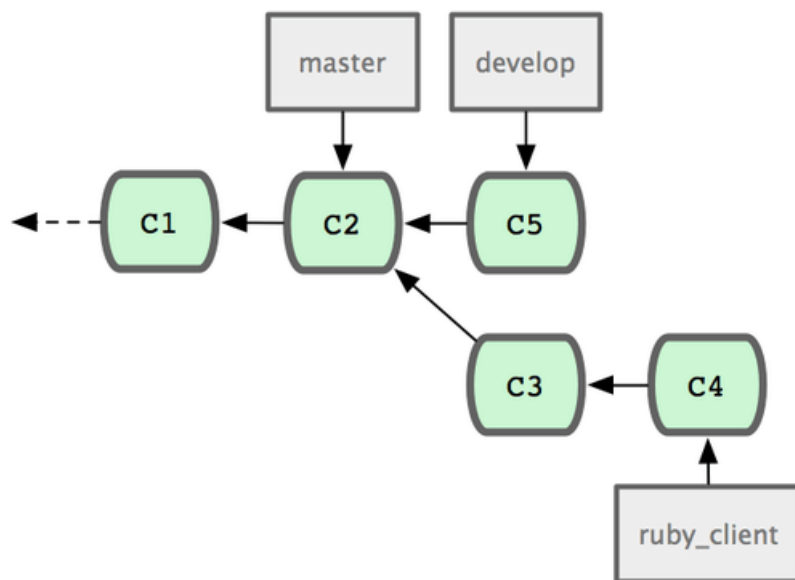


Figura 5-21. Antes do merge do topic branch.

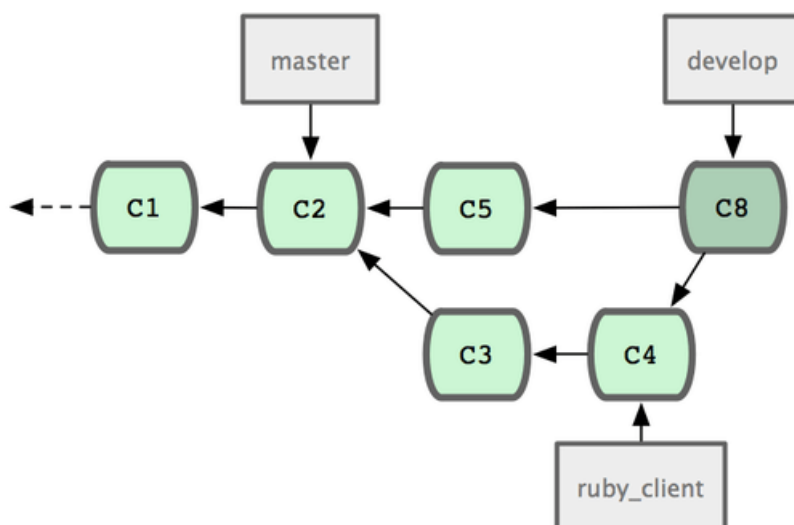


Figura 5-22. Depois do merge do topic branch.

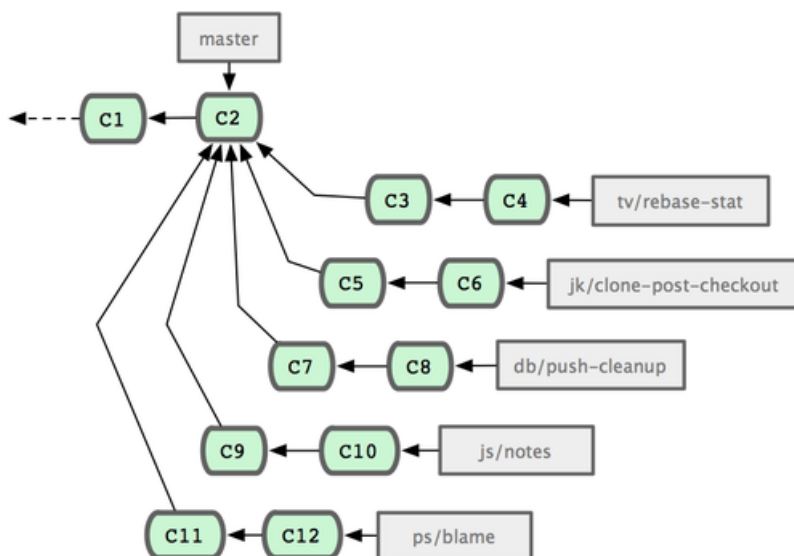


Figura 5-23. Depois da liberação do topic branch.

Dessa forma, quando as pessoas clonarem seu repositório do projeto, eles podem fazer checkout ou do master para fazer build da última versão estável e se manter atualizado facilmente, ou eles pode fazer checkout do develop para conseguir coisas mais de ponta. Você pode também continuar esse conceito, tendo um branch de integração onde é feito merge de todo o trabalho de uma vez. Então, quando a base de código daquele branch for estável e passar nos testes, você faz merge no branch develop; e quando este tiver comprovadamente estável por um tempo, você avança seu branch master.

Fluxo de Trabalho de Merges Grandes

O projeto Git tem quatro branches de longa duração: master, next e pu (proposed updates, atualizações propostas) para trabalho novo e maint para manutenção de versões legadas. Quando trabalho novo é introduzido por contribuintes, ele é coletado em topic branches no repositório do

mantenedor em uma maneira similar ao que já foi descrito (veja Figura 5-24). Nesse ponto, os tópicos são avaliados para determinar se eles são seguros e prontos para consumo ou se eles precisam de mais trabalho. Se eles são seguros, é feito merge em next e é dado push do branch para que todo mundo possa testar os tópicos integrados juntos.

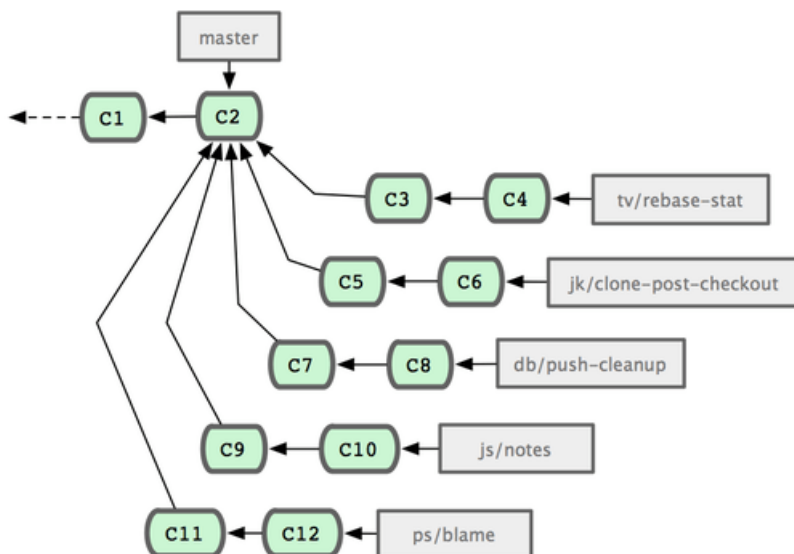


Figura 5-24. Gerenciando uma série complexa de topic branches contribuídos em paralelo.

Se os tópicos ainda precisam de trabalho, é feito merge em pu. Quando é determinado que eles estão totalmente estáveis, é feito novamente merge dos tópicos em master e os branches refeitos com os tópicos que estavam em next, mas não graduaram para master ainda. Isso significa que master quase sempre avança, next passa por rebase de vez em quando e pu ainda mais frequentemente (veja Figura 5-25).

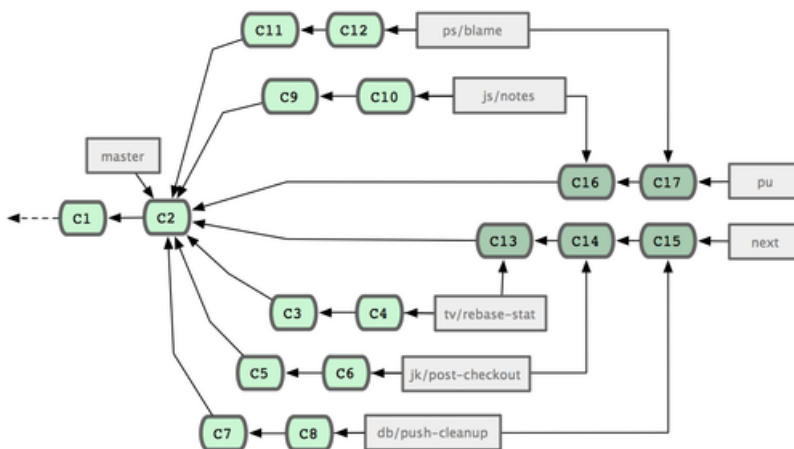


Figura 5-25. Fazendo merge de topic branches contribuídos em branches de integração de longa duração.

Quando finalmente tiver sido feito merge do topic branch em master, ele é removido do repositório. O projeto Git também tem um branch maint que é copiado (fork) da última versãp a fornecer

patches legados no caso de uma versão de manutenção ser requerida. Assim, quando você clona o repositório do Git, você tem quatro branches que você pode fazer checkout para avaliar o projeto em diferentes estágios de desenvolvimento, dependendo em quão atualizado você quer estar ou como você quer contribuir; e o mantenedor tem um fluxo de trabalho estruturado para ajudá-lo a vetar novas contribuições.

Fluxos de Trabalho para Rebase e Cherry Pick

Outros mantenedores preferem fazer rebase ou cherry-pick do trabalho contribuído em cima do branch `master` deles ao invés de fazer merge, para manter um histórico mais linear. Quando você tem trabalho em um topic branch e você determinou que você quer integrá-lo, você muda para aquele branch e executa o comando `rebase` para reconstruir as alterações em cima do branch `master` atual (ou `develop`, e por aí vai). Se isso funcionar bem, você pode avançar seu branch `master` e você irá terminar com um histórico de projeto linear.

A outra forma de mover trabalho introduzido de um branch para outro é `cherry-pick`. Um `cherry-pick` no Git é como um `rebase` para um único commit. Ele pega o patch que foi introduzido em um commit e tenta reaplicar no branch que você está. Isso é útil se você tem vários commits em um topic branch e quer integrar só um deles, ou se você tem um commit em um topic branch que você prefere usar `cherry-pick` ao invés de `rebase`. Por exemplo, vamos supor que você tem um projeto que se parece com a Figura 5-26.

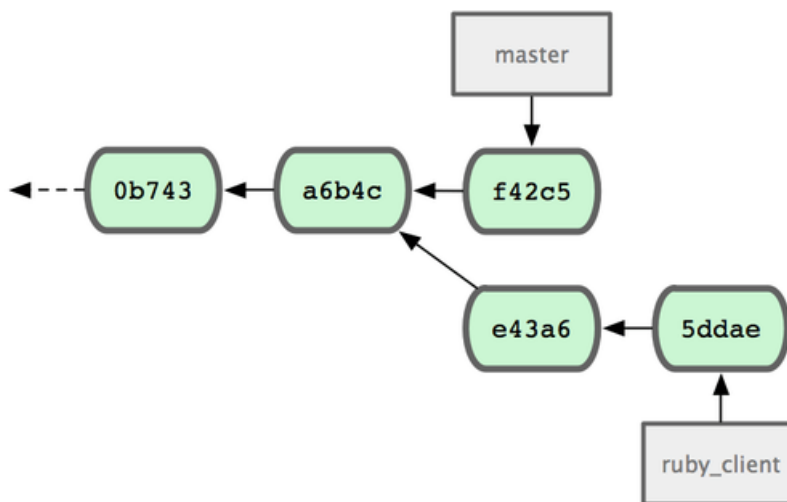


Figura 5-26. Histórico do exemplo antes de um cherry pick.

Se você quer puxar o commit `e43a6` no branch `master`, você pode executar

```

1 $ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcd
2 Finished one cherry-pick.
3 [master]: created a0a41a9: "More friendly message when locking the index fails."
4 3 files changed, 17 insertions(+), 3 deletions(-)

```

Isso puxa as mesmas alterações introduzidas em e43a6, mas o commit tem um novo valor SHA-1, porque a data de aplicação é diferente. Agora seu histórico se parece com a Figura 5-27.

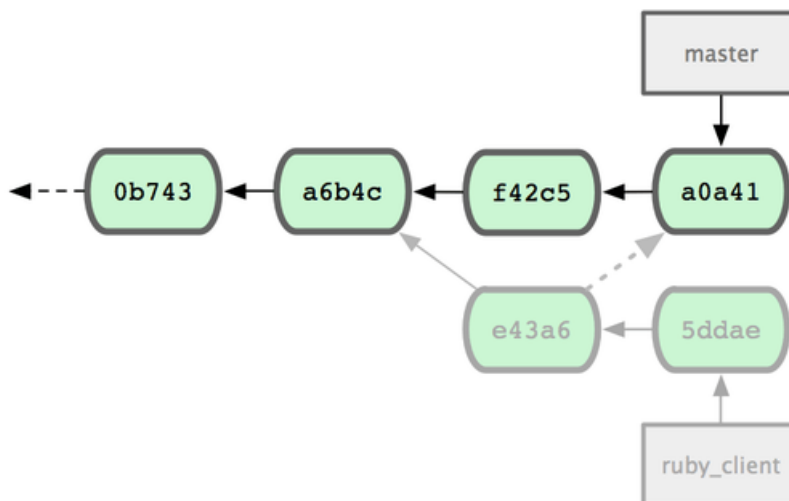


Figura 5-27. Histórico depois de fazer cherry-pick de um commit no topic branch.

Agora você pode remover seu topic branch e se livrar dos commits que você não quer puxar.

Gerando Tag de Suas Liberações (Releases)

Quando você decidir fazer uma release, você provavelmente irá querer fazer uma tag para que você possa recriar aquela liberação em qualquer ponto no futuro. Você pode criar uma nova tag como discutimos no capítulo 2. Se você decidir assinar a tag como mantenedor, o processo parecerá com isso:

```

1 $ git tag -s v1.5 -m 'my signed 1.5 tag'
2 You need a passphrase to unlock the secret key for
3 user: "Scott Chacon <schacon@gmail.com>"
4 1024-bit DSA key, ID F721C45A, created 2009-02-09

```

Se você assinar suas tags, você pode ter o problema de distribuir as chaves PGP públicas usadas para assinar suas tags. O mantenedor do projeto Git tem resolvido esse problema incluindo a chave pública como um blob no repositório e então adicionando a tag que aponta diretamente para o conteúdo. Para fazer isso, você pode descobrir qual a chave que você quer executando `gpg --list-keys`:

```

1 $ gpg --list-keys
2 /Users/schacon/.gnupg/pubring.gpg
3 -----
4 pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
5 uid Scott Chacon <schacon@gmail.com>
6 sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]

```

Então, você pode importar diretamente a chave no banco de dados Git exportando ela passando por pipe com `git hash-object`, que escreve um novo blob com esse conteúdo no Git e lhe devolve o SHA-1 do blob:

```

1 $ gpg -a --export F721C45A | git hash-object -w --stdin
2 659ef797d181633c87ec71ac3f9ba29fe5775b92

```

Agora que você tem os conteúdos da sua chave no Git, você pode criar uma tag que aponta diretamente para ela especificando o novo valor SHA-1 que o comando `hash-object` lhe deu:

```

1 $ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92

```

Se você executar `git push --tags`, a tag `maintainer-pgp-pub` será compartilhada com todo mundo. Se alguém quiser verificar a tag, ele pode importar diretamente sua chave PGP puxando o blob diretamente do banco de dados e importando no GPG:

```

1 $ git show maintainer-pgp-pub | gpg --import

```

Eles podem usar essa chave para verificar todas as tags assinadas. E se você incluir instruções na mensagem da tag, executar `git show <tag>` irá dar ao usuário final instruções mais detalhadas sobre a verificação da tag.

Gerando um Número de Build

Como o Git não tem um número que incrementa monotonicamente como ‘v123’ ou equivalente associado com cada commit, se você quer ter um nome legível que vá com cada commit você pode executar `git describe` naquele commit. Git lhe dá o nome da tag mais próxima com o número de commits em cima da tag e um SHA-1 parcial do commit que você está descrevendo:

```

1 $ git describe master
2 v1.6.2-rc1-20-g8c5b85c

```

Dessa forma, você pode exportar um snapshot ou build e nomeá-lo com algo compreensível para pessoas. De fato, se você compilar Git do código fonte clonado do repositório do Git, `git --version` lhe dará algo que se parece com isso. Se você estiver descrevendo um commit em que você adicionou uma tag, isso lhe dará o nome da tag.

O comando `git describe` favorece annotated tags (tags criadas com as opções `-a` ou `-s`), então tags de liberação devem ser criadas dessa forma se você estiver usando `git describe`, para assegurar que o commit seja nomeado corretamente quando feito o `describe`. Você pode também usar essa string como alvo do checkout, embora ele dependa do SHA-1 abreviado no final, então ele pode não ser válido para sempre. Por exemplo, o kernel do Linux recentemente mudou de 8 para 10 caracteres para assegurar que os SHA-1 sejam únicos, então saídas antigas do `git describe` foram invalidadas.

Preparando Uma Liberação

Agora você quer liberar uma build. Uma das coisas que você irá querer fazer é criar um arquivo do último snapshot de seu código para aquelas almas perdidas que não usam Git. O comando para fazer isso é `git archive`:

```
1 $ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
2 $ ls *.tar.gz
3 v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Se alguém abre o tarball, eles obtêm o último snapshot do projeto dentro de um diretório 'project'. Você pode também criar um arquivo zip quase da mesma forma, mas passando `--format=zip` para `git archive`:

```
1 $ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Você agora tem uma tarball e um arquivo zip da sua liberação que você pode disponibilizar em seu website ou enviar por e-mail para outras pessoas.

O Shortlog

É hora de enviar e-mail para a lista de e-mails das pessoas que querem saber o que está acontecendo no seu projeto. Uma forma legal de conseguir rapidamente um tipo de changelog do que foi adicionado ao seu projeto desde sua última liberação ou e-mail é usar o comando `git shortlog`. Ele resume todos os commits no intervalo que você passar; por exemplo, o seguinte lhe dá um resumo de todos os commits desde a sua última liberação, se sua última liberação foi chamada v1.0.1:

```
1 $ git shortlog --no-merges master --not v1.0.1
2 Chris Wanstrath (8):
3     Add support for annotated tags to Grit::Tag
4     Add packed-refs annotated tag support.
5     Add Grit::Commit#to_patch
6     Update version and History.txt
7     Remove stray `puts`
8     Make ls_tree ignore nils
9
10 Tom Preston-Werner (4):
11     fix dates in history
12     dynamic version method
13     Version bump to 1.0.2
14     Regenerated gemspec for version 1.0.2
```

Você obtém um resumo sucinto de todos os commits desde v1.0.1 agrupados por autor que você pode enviar por e-mail para a sua lista.

5.4 Git Distribuído - Resumo

Resumo

Você deve se sentir bastante confortável contribuindo para um projeto com Git assim como mantendo seu próprio projeto ou integrando contribuições de outros usuários. Parabéns por ser um desenvolvedor eficaz em Git! No próximo capítulo, você irá aprender mais ferramentas poderosas e dicas para lidar com situações complexas, que fará de você verdadeiramente um mestre Git.

Capítulo 6 - Ferramentas do Git

Até aqui, você aprendeu a maioria dos comandos e fluxos de trabalho do dia-a-dia que você precisa para gerenciar ou manter um repositório Git para o controle de seu código fonte. Você concluiu as tarefas básicas de rastreamento e commit de arquivos, e você aproveitou o poder da área de seleção e branches tópicos e merges.

Agora você vai explorar uma série de coisas muito poderosas que o Git pode fazer que você pode necessariamente não usar no dia-a-dia mas pode precisar em algum momento.

6.1 Ferramentas do Git - Seleção de Revisão

Seleção de Revisão

Git permite que você escolha commits específicos ou uma série de commits de várias maneiras. Elas não são necessariamente óbvias mas é útil conhecê-las.

Revisões Únicas

Obviamente você pode se referir a um commit pelo hash SHA-1 que é dado, mas também existem formas mais amigáveis para se referir a commits. Esta seção descreve as várias formas que você pode se referir a um único commit.

SHA Curto

Git é inteligente o suficiente para descobrir qual commit você quis digitar se você fornecer os primeiros caracteres, desde que o SHA-1 parcial tenha pelo menos quatro caracteres e não seja ambíguo — ou seja, somente um objeto no repositório atual comece com esse código SHA-1 parcial.

Por exemplo, para ver um commit específico, digamos que você execute um comando `git log` e identifique o commit que adicionou uma certa funcionalidade:

```

1 $ git log
2 commit 734713bc047d87bf7eac9674765ae793478c50d3
3 Author: Scott Chacon <schacon@gmail.com>
4 Date:   Fri Jan 2 18:32:33 2009 -0800
5
6     fixed refs handling, added gc auto, updated tests
7
8 commit d921970aadf03b3cf0e71becdaab3147ba71cdef
9 Merge: 1c002dd... 35cfb2b...
10 Author: Scott Chacon <schacon@gmail.com>
11 Date:   Thu Dec 11 15:08:43 2008 -0800
12
13     Merge commit 'phedders/rdocs'
14
15 commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
16 Author: Scott Chacon <schacon@gmail.com>
17 Date:   Thu Dec 11 14:58:32 2008 -0800
18
19     added some blame and merge stuff

```

Neste caso, escolho 1c002dd.... Se você executar `git show` nesse commit, os seguintes comandos são equivalentes (assumindo que as versões curtas não são ambíguas):

```

1 $ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
2 $ git show 1c002dd4b536e7479f
3 $ git show 1c002d

```

Git pode descobrir uma abreviação curta, única para seus valores SHA-1. Se você passar a opção `--abbrev-commit` para o comando `git log`, a saída irá usar valores curtos mas os mantém únicos; por padrão ele usa sete caracteres mas, usa mais se necessário para manter o SHA-1 não ambíguo:

```

1 $ git log --abbrev-commit --pretty=oneline
2 ca82a6d changed the verison number
3 085bb3b removed unnecessary test code
4 a11bef0 first commit

```

Geralmente, entre oito e dez caracteres são mais que suficientes para ser único em um projeto. Um dos maiores projetos no Git, o kernel do Linux, está começando a ter a necessidade de 12 caracteres dos 40 possíveis para ser único.

UMA NOTA SOBRE SHA-1

Muitas pessoas ficam preocupadas que em algum momento elas terão, por coincidência aleatória, dois objetos em seus repositórios com hash com o mesmo valor de SHA-1. O que fazer?

Se acontecer de você fazer um commit de um objeto que tem o hash com o mesmo valor de SHA-1 de um objeto existente no seu repositório, Git notará o primeiro objeto existente no seu banco de dados e assumirá que ele já foi gravado. Se você tentar fazer o checkout desse objeto novamente em algum momento, sempre receberá os dados do primeiro objeto.

Porém, você deve estar ciente de quão ridiculamente improvável é esse cenário. O código SHA-1 tem 20 bytes ou 160 bits. O número de objetos com hashes aleatórios necessários para garantir a probabilidade de 50% de uma única colisão é cerca de 2^{80} (a fórmula para determinar a probabilidade de colisão é $p = (n(n-1)/2) * (1/2^{160})$). 2^{80} é 1.2×10^{24} ou 1 milhão de bilhões de bilhões. Isso é 1.200 vezes o número de grãos de areia na Terra.

Aqui está um exemplo para lhe dar uma idéia do que seria necessário para obter uma colisão de SHA-1. Se todos os 6,5 bilhões de humanos na Terra estivessem programando, e a cada segundo, cada um estivesse produzindo código que é equivalente ao histórico inteiro do kernel do Linux (1 milhão de objetos Git) e fazendo o push para um enorme repositório Git, levaria 5 anos até que esse repositório tenha objetos suficientes para ter uma probabilidade de 50% de uma única colisão de objetos SHA-1. Existe uma probabilidade maior de cada membro do seu time de programação ser atacado e morto por lobos na mesma noite em incidentes sem relação.

Referências de Branch

A maneira mais simples de especificar um commit requer que ele tenha uma referência de um branch apontando para ele. Então, você pode usar um nome de branch em qualquer comando no Git que espera um objeto commit ou valor SHA-1. Por exemplo, se você quer mostrar o último objeto commit em um branch, os seguintes comandos são equivalentes, assumindo que o branch `topic1` aponta para `ca82a6d`:

- 1 `$ git show ca82a6dff817ec66f44342007202690a93763949`
- 2 `$ git show topic1`

Se você quer ver para qual SHA específico um branch aponta, ou se você quer ver o que qualquer desses exemplos se resumem em termos de SHAs, você pode usar uma ferramenta de Git plumbing chamada `rev-parse`. Você pode ver o Capítulo 9 para mais informações sobre ferramentas de plumbing (canalização); basicamente, `rev-parse` existe para operações de baixo nível e não é projetada para ser usada em operações do dia-a-dia. Entretanto, ela as vezes pode ser útil quando você precisa ver o que realmente está acontecendo. Aqui você pode executar `rev-parse` no seu branch.

```
1 $ git rev-parse topic1
2 ca82a6dff817ec66f44342007202690a93763949
```

Abreviações do RefLog

Uma das coisas que o Git faz em segundo plano enquanto você está fora é manter um reflog — um log de onde suas referências de HEAD e branches estiveram nos últimos meses.

Você poder ver o reflog usando `git reflog`:

```
1 $ git reflog
2 734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
3 d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
4 1c002dd... HEAD@{2}: commit: added some blame and merge stuff
5 1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
6 95df984... HEAD@{4}: commit: # This is a combination of two commits.
7 1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
8 7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

Cada vez que a extremidade do seu branch é atualizada por qualquer motivo, Git guarda essa informação para você nesse histórico temporário. E você pode especificar commits mais antigos com esses dados, também. Se você quer ver o quinto valor anterior ao HEAD do seu repositório, você pode usar a referência `@{n}` que você vê na saída do reflog:

```
1 $ git show HEAD@{5}
```

Você também pode usar essa sintaxe para ver onde um branch estava há um período de tempo anterior. Por exemplo, para ver onde seu branch `master` estava ontem, você pode digitar

```
1 $ git show master@{yesterday}
```

Isso mostra onde a extremidade do branch estava ontem. Essa técnica funciona somente para dados que ainda estão no seu reflog, você não pode usá-la para procurar commits feitos há muitos meses atrás.

Para ver a informação do reflog formatada como a saída do `git log`, você pode executar `git log -g`:

```

1 $ git log -g master
2 commit 734713bc047d87bf7eac9674765ae793478c50d3
3 Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
4 Reflog message: commit: fixed refs handling, added gc auto, updated
5 Author: Scott Chacon <schacon@gmail.com>
6 Date:   Fri Jan 2 18:32:33 2009 -0800
7
8     fixed refs handling, added gc auto, updated tests
9
10 commit d921970aadf03b3cf0e71becdaab3147ba71cdef
11 Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
12 Reflog message: merge phedders/rdocs: Merge made by recursive.
13 Author: Scott Chacon <schacon@gmail.com>
14 Date:   Thu Dec 11 15:08:43 2008 -0800
15
16     Merge commit 'phedders/rdocs'

```

É importante notar que a informação do reflog é estritamente local — é um log do que você fez no seu repositório. As referências não serão as mesmas na cópia do repositório de outra pessoa; e logo depois que você fez o clone inicial de um repositório, você terá um reflog vazio, pois nenhuma atividade aconteceu no seu repositório. Executar `git show HEAD@{2.months.ago}` funcionará somente se você fez o clone do projeto há pelo menos dois meses atrás — se você fez o clone dele há cinco minutos, você não terá resultados.

Referências Ancestrais

A outra principal maneira de especificar um commit é através de seu ancestral. Se você colocar um `^` no final da referência, Git interpreta isso como sendo o pai do commit. Suponha que você veja o histórico do seu projeto:

```

1 $ git log --pretty=format: '%h %s' --graph
2 * 734713b fixed refs handling, added gc auto, updated tests
3 *   d921970 Merge commit 'phedders/rdocs'
4 | \
5 | * 35cfb2b Some rdoc changes
6 * | 1c002dd added some blame and merge stuff
7 | /
8 * 1c36188 ignore *.gem
9 * 9b29157 add open3_detach to gemspec file list

```

Em seguida, você pode ver o commit anterior especificando `HEAD^`, que significa “o pai do HEAD”:

```

1 $ git show HEAD^
2 commit d921970aadf03b3cf0e71becdaab3147ba71cdef
3 Merge: 1c002dd... 35cfb2b...
4 Author: Scott Chacon <schacon@gmail.com>
5 Date: Thu Dec 11 15:08:43 2008 -0800
6
7 Merge commit 'phedders/rdocs'

```

Você também pode informar um número depois do ^ — por exemplo, d921970^2 significa “o segundo pai de d921970.” Essa sintaxe só é útil para commits com merge, que têm mais de um pai. O primeiro pai é o branch que você estava quando fez o merge, e o segundo é o commit no branch que você fez o merge:

```

1 $ git show d921970^
2 commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
3 Author: Scott Chacon <schacon@gmail.com>
4 Date: Thu Dec 11 14:58:32 2008 -0800
5
6 added some blame and merge stuff
7
8 $ git show d921970^2
9 commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
10 Author: Paul Hedderly <paul+git@mjr.org>
11 Date: Wed Dec 10 22:22:03 2008 +0000
12
13 Some rdoc changes

```

A outra forma de especificar ancestrais é o ~. Isso também faz referência ao primeiro pai, assim HEAD~ e HEAD^ são equivalentes. A diferença se torna aparente quando você informa um número. HEAD~2 significa “o primeiro pai do primeiro pai,” ou “o avô” — passa pelos primeiros pais a quantidade de vezes que você informa. Por exemplo, no histórico listado antes, HEAD~3 seria

```

1 $ git show HEAD~3
2 commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
3 Author: Tom Preston-Werner <tom@mojombo.com>
4 Date: Fri Nov 7 13:47:59 2008 -0500
5
6 ignore *.gem

```

Isso também pode ser escrito HEAD^^^, que novamente, é o primeiro pai do primeiro pai do primeiro pai:

```

1 $ git show HEAD^^^
2 commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
3 Author: Tom Preston-Werner <tom@mojombo.com>
4 Date:   Fri Nov 7 13:47:59 2008 -0500
5
6     ignore *.gem

```

Você também pode combinar essas sintaxes — você pode obter o segundo pai da referência anterior (assumindo que ele era um commit com merge) usando `HEAD~3^2`, e assim por diante.

Intervalos de Commits

Agora que você pode especificar commits individuais, vamos ver como especificar intervalos de commits. Isso é particularmente útil para gerenciar seus branches — se você tem muitos branches, você pode usar especificações de intervalos para responder perguntas como, “Que modificações existem nesse branch que ainda não foram mescladas (merge) no meu branch principal?”.

Ponto Duplo

A especificação de intervalo mais comum é a sintaxe de ponto-duplo. Isso basicamente pede ao Git para encontrar um intervalo de commits que é acessível a partir de um commit, mas não são acessíveis a partir de outro. Por exemplo, digamos que você tem um histórico de commits como a Figure 6-1.

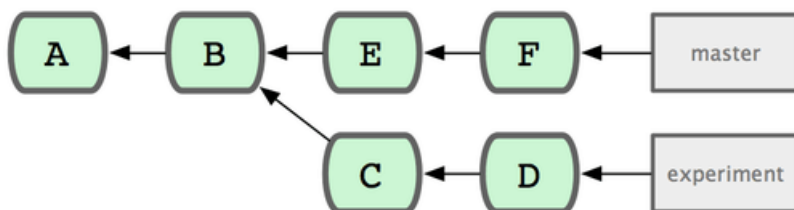


Figura 6-1. Exemplo de histórico de seleção de intervalo.

Você quer ver o que existe no seu branch mas não existe no branch master. Você pede ao Git para mostrar um log apenas desses commits com `master..experiment` — isso significa “todos os commits acessíveis por experiment que não são acessíveis por master.” Para deixar os exemplos mais breves e claros, vou usar as letras dos objetos dos commits do diagrama no lugar da saída real do log na ordem que eles seriam mostrados:

```

1 $ git log master..experiment
2 D
3 C

```

Se, por outro lado, você quer ver o oposto — todos os commits em master que não estão em experiment — você pode inverter os nomes dos branches. `experiment..master` exibe tudo em master que não é acessível em experiment:

```
1 $ git log experiment..master
2 F
3 E
```

Isso é útil se você quer manter o branch `experiment` atualizado e visualizar que merge você está prestes a fazer. Outro uso muito frequente desta sintaxe é para ver o que você está prestes a enviar para um remoto:

```
1 $ git log origin/master..HEAD
```

Esse comando lhe mostra qualquer commit no seu branch atual que não está no branch `master` no seu remoto `origin`. Se você executar um `git push` e seu branch atual está rastreando `origin/master`, os commits listados por `git log origin/master..HEAD` são os commits que serão transferidos para o servidor. Você também pode não informar um lado da sintaxe que o Git assumirá ser `HEAD`. Por exemplo, você pode obter os mesmos resultados que no exemplo anterior digitando `git log origin/master..` — Git substitui por `HEAD` o lado que está faltando.

Múltiplos Pontos

A sintaxe ponto-duplo é útil como um atalho; mas talvez você queira especificar mais de dois branches para indicar sua revisão, como ver quais commits estão em qualquer um dos branches mas não estão no branch que você está atualmente. Git permite que você faça isso usando o caractere `^` ou `--not` antes de qualquer referência a partir do qual você não quer ver commits acessíveis. Assim, estes três comandos são equivalentes:

```
1 $ git log refA..refB
2 $ git log ^refA refB
3 $ git log refB --not refA
```

Isso é bom porque com essa sintaxe você pode especificar mais de duas referências em sua consulta, o que você não pode fazer com a sintaxe ponto-duplo. Por exemplo, se você quer ver todos os commits que são acessíveis de `refA` ou `refB` mas não de `refC`, você pode digitar um desses:

```
1 $ git log refA refB ^refC
2 $ git log refA refB --not refC
```

Este é um sistema de consulta de revisão muito poderoso que deve ajudá-lo a descobrir o que existe nos seus branches.

Ponto Triplo

A última grande sintaxe de intervalo de seleção é a sintaxe ponto-triplo, que especifica todos os commits que são acessíveis por qualquer uma das duas referências mas não por ambas. Veja novamente o exemplo de histórico de commits na Figure 6-1. Se você quer ver o que tem em `master` ou `experiment` mas sem referências comuns, você pode executar

```
1 $ git log master...experiment
2 F
3 E
4 D
5 C
```

Novamente, isso lhe dá uma saída de log normal mas mostra somente as informações desses quatro commits, aparecendo na ordem de data de commit tradicional.

Uma opção comum para usar com o comando log nesse caso é `--left-right`, que mostra qual lado do intervalo está cada commit. Isso ajuda a tornar os dados mais úteis:

```
1 $ git log --left-right master...experiment
2 < F
3 < E
4 > D
5 > C
```

Com essas ferramentas, você pode informar ao Git mais facilmente qual ou quais commits você quer inspecionar.

6.2 Ferramentas do Git - Área de Seleção Interativa

Área de Seleção Interativa

Git vem com alguns scripts que facilitam algumas tarefas de linha de comando. Aqui, você verá alguns comandos interativos que podem ajudar você a facilmente escolher combinações ou partes de arquivos para incorporar em um commit. Essas ferramentas são muito úteis se você modificou vários arquivos e decidiu que quer essas modificações em commits separados em vez de um grande e bagunçado commit. Desta maneira, você pode ter certeza que seus commits estão logicamente separados e podem ser facilmente revisados pelos outros desenvolvedores trabalhando com você. Se você executar `git add` com a opção `-i` ou `--interactive`, Git entra em um modo interativo de shell, exibindo algo desse tipo:

```

1 $ git add -i
2         staged      unstaged path
3  1:    unchanged      +0/-1  TODO
4  2:    unchanged      +1/-1  index.html
5  3:    unchanged      +5/-1  lib/simplegit.rb
6
7 *** Commands ***
8  1: status      2: update      3: revert      4: add untracked
9  5: patch       6: diff        7: quit       8: help
10 What now>

```

Você pode ver que esse comando lhe mostra uma visão muito diferente da sua área de seleção — basicamente a mesma informação que você recebe com `git status` mas um pouco mais sucinto e informativo. Ele lista as modificações que você colocou na área de seleção à esquerda e as modificações que estão fora à direita.

Depois disso vem a seção `Commands`. Aqui você pode fazer uma série de coisas, incluindo adicionar arquivos na área de seleção, retirar arquivos, adicionar partes de arquivos, adicionar arquivos não rastreados, e ver diffs de o que já foi adicionado.

Adicionando e Retirando Arquivos

Se você digitar 2 ou u em `What now>`, o script perguntará quais arquivos você quer adicionar:

```

1 What now> 2
2         staged      unstaged path
3  1:    unchanged      +0/-1  TODO
4  2:    unchanged      +1/-1  index.html
5  3:    unchanged      +5/-1  lib/simplegit.rb
6 Update>>

```

Para adicionar os arquivos `TODO` e `index.html`, você pode digitar os números:

```

1 Update>> 1,2
2         staged      unstaged path
3 * 1:    unchanged      +0/-1  TODO
4 * 2:    unchanged      +1/-1  index.html
5  3:    unchanged      +5/-1  lib/simplegit.rb
6 Update>>

```

O * ao lado de cada arquivos significa que o arquivo está selecionado para ser adicionado. Se você pressionar `Enter` sem digitar nada em `Update>>`, Git pega tudo que esta selecionado e adiciona na área de seleção para você:


```

1  Update>>
2  updated 2 paths
3
4  *** Commands ***
5   1: status      2: update      3: revert      4: add untracked
6   5: patch       6: diff       7: quit       8: help
7  What now> 1
8
9   staged      unstaged path
10  1:          +0/-1      nothing TODO
11  2:          +1/-1      nothing index.html
12  3:          unchanged   +5/-1 lib/simplegit.rb

```

Agora você pode ver que os arquivos TODO e index.html estão na área de seleção e o arquivo simplegit.rb ainda está fora. Se você quer retirar o arquivo TODO nesse momento, você usa a opção 3 ou r (para reverter):

```

1  *** Commands ***
2   1: status      2: update      3: revert      4: add untracked
3   5: patch       6: diff       7: quit       8: help
4  What now> 3
5
6   staged      unstaged path
7   1:          +0/-1      nothing TODO
8   2:          +1/-1      nothing index.html
9   3:          unchanged   +5/-1 lib/simplegit.rb
10 Revert>> 1
11
12  * 1:          +0/-1      nothing TODO
13  2:          +1/-1      nothing index.html
14  3:          unchanged   +5/-1 lib/simplegit.rb
15 Revert>> [enter]
16 reverted one path

```

Olhando o seu Git status novamente, você pode ver que você retirou o arquivo TODO:

```

1  *** Commands ***
2   1: status      2: update      3: revert      4: add untracked
3   5: patch       6: diff        7: quit        8: help
4  What now> 1
5           staged      unstaged path
6   1:   unchanged      +0/-1  TODO
7   2:       +1/-1      nothing index.html
8   3:   unchanged      +5/-1  lib/simplegit.rb

```

Para ver o diff do que você adicionou, você pode usar o comando 6 ou d (para diff). Ele exibe uma lista dos seus arquivos adicionados, e você pode selecionar aqueles que você gostaria de ver a diferença. Isso é muito parecido com informar o comando `git diff --cached`:

```

1  *** Commands ***
2   1: status      2: update      3: revert      4: add untracked
3   5: patch       6: diff        7: quit        8: help
4  What now> 6
5           staged      unstaged path
6   1:       +1/-1      nothing index.html
7  Review diff>> 1
8  diff --git a/index.html b/index.html
9  index 4d07108..4335f49 100644
10 --- a/index.html
11 +++ b/index.html
12 @@ -16,7 +16,7 @@ Date Finder
13
14  <p id="out">...</p>
15
16  -<div id="footer">contact : support@github.com</div>
17  +<div id="footer">contact : email.support@github.com</div>
18
19  <script type="text/javascript">

```

Com esses comandos básicos, você pode usar o modo interativo para lidar com sua área de seleção um pouco mais facilmente.

Adicionando Patches

No Git também é possível adicionar certas partes de arquivos e deixar o resto de fora. Por exemplo, se você faz duas mudanças no seu arquivo `simplegit.rb` e quer adicionar uma delas mas não a outra, fazer isso é muito fácil no Git. A partir do prompt interativo, digite 5 ou p (para patch). Git irá perguntar quais arquivos você gostaria de adicionar parcialmente; então, para cada seção dos arquivos selecionados, ele irá exibir partes do diff do arquivo e perguntar se você gostaria de adicioná-los, um por um:

```

1 diff --git a/lib/simplegit.rb b/lib/simplegit.rb
2 index dd5ecc4..57399e0 100644
3 --- a/lib/simplegit.rb
4 +++ b/lib/simplegit.rb
5 @@ -22,7 +22,7 @@ class SimpleGit
6     end
7
8     def log(treeish = 'master')
9 -     command("git log -n 25 #{treeish}")
10 +     command("git log -n 30 #{treeish}")
11     end
12
13     def blame(path)
14 Stage this hunk [y,n,a,d,/,j,J,g,e,]?

```

Você tem várias opções neste momento. Digitando ? exibe uma lista do que você pode fazer:

```

1 Stage this hunk [y,n,a,d,/,j,J,g,e,]? ?
2 y - stage this hunk
3 n - do not stage this hunk
4 a - stage this and all the remaining hunks in the file
5 d - do not stage this hunk nor any of the remaining hunks in the file
6 g - select a hunk to go to
7 / - search for a hunk matching the given regex
8 j - leave this hunk undecided, see next undecided hunk
9 J - leave this hunk undecided, see next hunk
10 k - leave this hunk undecided, see previous undecided hunk
11 K - leave this hunk undecided, see previous hunk
12 s - split the current hunk into smaller hunks
13 e - manually edit the current hunk
14 ? - print help

```

Geralmente, você informará y ou n se você quer adicionar cada parte, mas adicionar todos eles em determinados arquivos ou deixar a decisão de uma parte para depois pode ser útil também. Se você adicionar uma parte do arquivo e deixar outra parte de fora, a saída do status será parecido com isso:

```

1 What now> 1
2           staged      unstaged path
3 1:    unchanged      +0/-1 TODO
4 2:      +1/-1        nothing index.html
5 3:      +1/-1        +4/-0 lib/simplegit.rb

```

O status do arquivo `simplegit.rb` é interessante. Ele lhe mostra que algumas linhas foram adicionadas e algumas estão fora. Você adicionou esse arquivo parcialmente. Neste momento, você pode sair do script de modo interativo e executar `git commit` para fazer o commit parcial dos arquivos adicionados.

Finalmente, você não precisa estar no modo interativo para adicionar um arquivo parcialmente — você pode executar o mesmo script usando `git add -p` ou `git add --patch` na linha de comando.

6.3 Ferramentas do Git - Fazendo Stash

Fazendo Stash

Muitas vezes, quando você está trabalhando em uma parte do seu projeto, as coisas estão em um estado confuso e você quer mudar de branch por um tempo para trabalhar em outra coisa. O problema é, você não quer fazer o commit de um trabalho incompleto somente para voltar a ele mais tarde. A resposta para esse problema é o comando `git stash`.

Fazer Stash é tirar o estado sujo do seu diretório de trabalho — isto é, seus arquivos modificados que estão sendo rastreados e mudanças na área de seleção — e o salva em uma pilha de modificações inacabadas que você pode voltar a qualquer momento.

Fazendo Stash do Seu Trabalho

Para demonstrar, você entra no seu projeto e começa a trabalhar em alguns arquivos e adiciona alguma modificação na área de seleção. Se você executar `git status`, você pode ver seu estado sujo:

```

1 $ git status
2 # On branch master
3 # Changes to be committed:
4 #   (use "git reset HEAD <file>..." to unstage)
5 #
6 #       modified:   index.html
7 #
8 # Changes not staged for commit:
9 #   (use "git add <file>..." to update what will be committed)
10 #
11 #       modified:   lib/simplegit.rb
12 #

```

Agora você quer mudar de branch, mas não quer fazer o commit do que você ainda está trabalhando; você irá fazer o stash das modificações. Para fazer um novo stash na sua pilha, execute `git stash`:

```
1 $ git stash
2 Saved working directory and index state \
3   "WIP on master: 049d078 added the index file"
4 HEAD is now at 049d078 added the index file
5 (To restore them type "git stash apply")
```

Seu diretório de trabalho está limpo:

```
1 $ git status
2 # On branch master
3 nothing to commit (working directory clean)
```

Neste momento, você pode facilmente mudar de branch e trabalhar em outra coisa; suas alterações estão armazenadas na sua pilha. Para ver as stashes que você guardou, você pode usar `git stash list`:

```
1 $ git stash list
2 stash@{0}: WIP on master: 049d078 added the index file
3 stash@{1}: WIP on master: c264051... Revert "added file_size"
4 stash@{2}: WIP on master: 21d80a5... added number to log
```

Nesse caso, duas stashes tinham sido feitas anteriormente, então você tem acesso a três trabalhos stashed diferentes. Você pode aplicar aquele que acabou de fazer o stash com o comando mostrado na saída de ajuda do comando `stash` original: `git stash apply`. Se você quer aplicar um dos stashes mais antigos, você pode especificá-lo, assim: `git stash apply stash@{2}`. Se você não especificar um stash, Git assume que é o stash mais recente e tenta aplicá-lo:

```
1 $ git stash apply
2 # On branch master
3 # Changes not staged for commit:
4 #   (use "git add <file>..." to update what will be committed)
5 #
6 #       modified:   index.html
7 #       modified:   lib/simplegit.rb
8 #
```

Você pode ver que o Git altera novamente os arquivos que você reverteu quando salvou o stash. Neste caso, você tinha um diretório de trabalho limpo quando tentou aplicar o stash, e você tentou aplicá-lo no mesmo branch de onde tinha guardado; mas ter um diretório de trabalho limpo e aplicá-lo no mesmo branch não é necessário para usar um stash com sucesso. Você pode salvar um stash em um branch, depois mudar para outro branch, e tentar reaplicar as alterações. Você também pode ter arquivos alterados e sem commits no seu diretório de trabalho quando aplicar um stash — Git informa conflitos de merge se alguma coisa não aplicar de forma limpa.

As alterações em seus arquivos foram reaplicadas, mas o arquivo que você colocou na área de seleção antes não foi adicionado novamente. Para fazer isso, você deve executar o comando `git stash apply` com a opção `--index` para informar ao comando para tentar reaplicar as modificações da área de seleção. Se você tivesse executado isso, você teria conseguido voltar à sua posição original:

```
1 $ git stash apply --index
2 # On branch master
3 # Changes to be committed:
4 #   (use "git reset HEAD <file>..." to unstage)
5 #
6 #       modified:   index.html
7 #
8 # Changes not staged for commit:
9 #   (use "git add <file>..." to update what will be committed)
10 #
11 #       modified:   lib/simplegit.rb
12 #
```

A opção `apply` somente tenta aplicar o stash armazenado — ele continua na sua pilha. Para removê-lo, você pode executar `git stash drop` com o nome do stash que quer remover:

```
1 $ git stash list
2 stash@{0}: WIP on master: 049d078 added the index file
3 stash@{1}: WIP on master: c264051... Revert "added file_size"
4 stash@{2}: WIP on master: 21d80a5... added number to log
5 $ git stash drop stash@{0}
6 Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Você também pode executar `git stash pop` para aplicar o stash e logo em seguida apagá-lo da sua pilha.

Revertendo um Stash

Em alguns cenários você pode querer aplicar alterações de um stash, trabalhar, mas desfazer essas alterações que originalmente vieram do stash. Git não fornece um comando como `stash unapply`, mas é possível fazer o mesmo simplesmente recuperando o patch associado com um stash e aplicá-lo em sentido inverso:

```
1 $ git stash show -p stash@{0} | git apply -R
```

Novamente, se você não especificar um stash, Git assume que é o stash mais recente:

```
1 $ git stash show -p | git apply -R
```

Você pode querer criar um alias e adicionar explicitamente um comando `stash-unapply` no seu git. Por exemplo:

```
1 $ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
2 $ git stash
3 $ #... work work work
4 $ git stash-unapply
```

Criando um Branch de um Stash

Se você criar um stash, deixá-lo lá por um tempo, e continuar no branch de onde criou o stash, você pode ter problemas em reaplicar o trabalho. Se o `apply` tentar modificar um arquivo que você alterou, você vai ter um conflito de merge e terá que tentar resolvê-lo. Se você quer uma forma mais fácil de testar as modificações do stash novamente, você pode executar `git stash branch`, que cria um novo branch para você, faz o checkout do commit que você estava quando criou o stash, reaplica seu trabalho nele, e então apaga o stash se ele é aplicado com sucesso:

```
1 $ git stash branch testchanges
2 Switched to a new branch "testchanges"
3 # On branch testchanges
4 # Changes to be committed:
5 #   (use "git reset HEAD <file>..." to unstage)
6 #
7 #       modified:   index.html
8 #
9 # Changes not staged for commit:
10 #   (use "git add <file>..." to update what will be committed)
11 #
12 #       modified:   lib/simplegit.rb
13 #
```

Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359) Este é um bom atalho para recuperar facilmente as modificações em um stash e trabalhar nele em um novo branch.

6.4 Ferramentas do Git - Reescrevendo o Histórico

Reescrevendo o Histórico

Muitas vezes, trabalhando com o Git, você pode querer revisar seu histórico de commits por alguma razão. Uma das melhores funcionalidades do Git é que ele permite você tomar decisões no último momento possível. Você pode decidir quais arquivos vai em qual commit um pouco antes de fazer o commit da área de seleção, você pode decidir que não quer trabalhar em alguma coisa ainda com o comando `stash`, e você pode reescrever commits que já aconteceram para que eles pareçam ter acontecido de outra maneira. Isso pode envolver mudar a ordem dos commits, alterar mensagens ou arquivos em um commit, juntar ou separar commits, ou remover commits completamente — tudo isso antes de compartilhar seu trabalho com os outros.

Nesta seção, você verá como realizar essas tarefas muito úteis de modo que você pode fazer seu histórico de commits parecer da forma que você quiser antes de compartilhá-lo com outros.

Alterando o Último Commit

Modificar o último commit é provavelmente a alteração de histórico mais comum que você irá fazer. Muitas vezes você vai querer fazer duas coisas básicas com seu último commit: mudar a mensagem do commit, ou mudar o snapshot que você acabou de salvar, adicionando, alterando e removendo arquivos.

Se você quer somente modificar a mensagem do seu último commit, é muito simples:

```
1 $ git commit --amend
```

Isso abre seu editor de texto, com sua última mensagem de commit nele, pronto para você modificar a mensagem. Quando você salva e fecha o editor, ele salva um novo commit contendo essa mensagem e fazendo esse seu novo commit o mais recente.

Se você fez o commit e quer alterar o snapshot adicionando ou modificando arquivos, possivelmente porque você esqueceu de adicionar um arquivo novo quando fez o commit original, o processo funciona basicamente da mesma maneira. Você adiciona as alterações que deseja na área de seleção editando um arquivo e executando `git add` nele ou `git rm` em um arquivo rastreado, e depois `git commit --amend` pega sua área de seleção atual e faz o snapshot para o novo commit.

Você precisa ter cuidado com essa técnica porque isso muda o SHA-1 do commit. É como um rebase muito pequeno — não altere seu último commit se você já fez o push dele.

Alterando Várias Mensagens de Commit

Para modificar um commit mais antigo em seu histórico, você deve usar ferramentas mais complexas. Git não tem uma ferramenta de modificação de histórico, mas você pode usar o rebase para alterar

uma série de commits no HEAD onde eles estavam originalmente em vez de movê-los para um novo. Com a ferramenta de rebase interativo, você pode parar depois de cada commit que quer modificar e alterar a mensagem, adicionar arquivos, ou fazer o que quiser. Você pode executar o rebase de forma interativa adicionando a opção `-i` em `git rebase`. Você deve indicar quão longe você quer reescrever os commits informando ao comando qual commit quer fazer o rebase.

Por exemplo, se você quer alterar as mensagens dos últimos três commits, ou qualquer mensagem de commit nesse grupo, você informa como argumento para `git rebase -i` o pai do último commit que você quer editar, que é `HEAD~2^` ou `HEAD~3`. Pode ser mais fácil de lembrar o `~3` porque você está tentando editar os últimos três commits; mas lembre-se que você está indicando realmente quatro commits atrás, o pai do último que você deseja editar:

```
1 $ git rebase -i HEAD~3
```

Lembre-se que isso é um comando rebase — todos os commits no intervalo `HEAD~3..HEAD` serão reescritos, quer você mude a mensagem ou não. Não inclua nenhum commit que você já enviou a um servidor central — fazer isso irá confundir outros desenvolvedores fornecendo uma versão alternativa da mesma alteração.

Executando esse comando dará a você uma lista de commits no seu editor de texto que se parece com isso:

```
pick f7f3f6d changed my name a bit pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

```
1 # Rebase 710f0f8..a5f4a0d onto 710f0f8
2 #
3 # Commands:
4 # p, pick = use commit
5 # e, edit = use commit, but stop for amending
6 # s, squash = use commit, but meld into previous commit
7 #
8 # If you remove a line here THAT COMMIT WILL BE LOST.
9 # However, if you remove everything, the rebase will be aborted.
10 #
```

É importante notar que esses commits são listados na ordem inversa do que você normalmente vê usando o comando `log`. Se você executar um `log`, você vê algo como isto:

```
1 $ git log --pretty=format:"%h %s" HEAD~3..HEAD
2 a5f4a0d added cat-file
3 310154e updated README formatting and added blame
4 f7f3f6d changed my name a bit
```

Observe a ordem inversa. O rebase interativo lhe dá um script que ele irá executar. Ele começará no commit que você especifica na linha de comando (HEAD~3) e repete as modificações introduzidas em cada um desses commits de cima para baixo. Ele lista o mais antigo primeiro, em vez do mais recente, porque ele vai ser o primeiro a ser alterado.

Você precisa editar o script para que ele pare no commit que você deseja editar. Para fazer isso, mude a palavra “pick” para a palavra “edit” para cada um dos commits que você deseja que o script pare. Por exemplo, para alterar somente a terceira mensagem de commit, você altera o arquivo para ficar assim:

```
1 edit f7f3f6d changed my name a bit
2 pick 310154e updated README formatting and added blame
3 pick a5f4a0d added cat-file
```

Quando você salva e fecha o editor, Git retorna para o último commit na lista e deixa você na linha de comando com a seguinte mensagem:

```
1 $ git rebase -i HEAD~3
2 Stopped at 7482e0d... updated the gemspec to hopefully work better
3 You can amend the commit now, with
4
5     git commit --amend
6
7 Once you're satisfied with your changes, run
8
9     git rebase --continue
```

Estas instruções lhe dizem exatamente o que fazer. Digite

```
1 $ git commit --amend
```

Altere a mensagem do commit, e saia do editor. Depois execute

```
1 $ git rebase --continue
```

Esse comando irá aplicar os outros dois commits automaticamente, e pronto. Se você alterar “pick” para “edit” em mais linhas, você pode repetir esses passos para cada commit que mudar para “edit”. Cada vez, Git irá parar, permitir que você altere o commit, e continuar quando você tiver terminado.

Reordenando Commits

Você também pode usar rebase interativo para reordenar ou remover commits completamente. Se você quer remover o commit “added cat-file” e mudar a ordem em que os outros dois commits foram feitos, você pode alterar o script do rebase disso

```
1 pick f7f3f6d changed my name a bit
2 pick 310154e updated README formatting and added blame
3 pick a5f4a0d added cat-file
```

para isso:

```
1 pick 310154e updated README formatting and added blame
2 pick f7f3f6d changed my name a bit
```

Quando você salva e sai do editor, Git volta seu branch para o pai desses commits, altera o 310154e e depois o f7f3f6d, e para. Você efetivamente alterou a ordem desses commits e removeu o commit “added cat-file” completamente.

Achatando um Commit

Também é possível pegar uma série de commits e achatá-los em um único commit com a ferramenta de rebase interativo. O script coloca informações importantes na mensagem de rebase:

```
1 #
2 # Commands:
3 # p, pick = use commit
4 # e, edit = use commit, but stop for amending
5 # s, squash = use commit, but meld into previous commit
6 #
7 # If you remove a line here THAT COMMIT WILL BE LOST.
8 # However, if you remove everything, the rebase will be aborted.
9 #
```

Se, em vez de “pick” ou “edit”, você especifica “squash”, Git modifica essa e a alteração imediatamente anterior a ela e faz com que você faça o merge das mensagens de commits. Então, se você quer fazer um único commit a partir desses três commits, você modifica o script para algo como isso:

```
1 pick f7f3f6d changed my name a bit
2 squash 310154e updated README formatting and added blame
3 squash a5f4a0d added cat-file
```

Quando você salva e sai do editor, Git aplica as três modificações e coloca você de volta no editor para fazer o merge das três mensagens de commit:

```
1  # This is a combination of 3 commits.
2  # The first commit's message is:
3  changed my name a bit
4
5  # This is the 2nd commit message:
6
7  updated README formatting and added blame
8
9  # This is the 3rd commit message:
10
11 added cat-file
```

Quando você salvar isso, você terá um único commit com as alterações dos três commits anteriores.

Dividindo um Commit

Dividir um commit significa desfazer um commit e parcialmente adicionar a área de seleção e commits dependendo do número de commits que você quer. Por exemplo, digamos que você quer dividir o commit do meio daqueles seus três commits. Em vez de “updated README formatting and added blame”, você quer dividi-lo em dois commits: “updated README formatting” no primeiro, e “added blame” no segundo. Você pode fazer isso no script rebase `-i` mudando a instrução para “edit” no commit que você quer dividir:

```
1 pick f7f3f6d changed my name a bit
2 edit 310154e updated README formatting and added blame
3 pick a5f4a0d added cat-file
```

Depois, quando o script colocar retornar para a linha de comando, você faz o reset desse commit, pega as alterações desse reset, e cria vários commits delas. Quando você salvar e sai do editor, Git volta ao pai do primeiro commit da sua lista, altera o primeiro commit (f7f3f6d), altera o segundo (310154e), e retorna você ao console. Lá, você pode fazer um reset desse commit com `git reset HEAD^`, que efetivamente desfaz o commit e deixa os arquivos alterados fora da área de seleção. Agora você pode colocar na área de seleção e fazer vários commits, e executar `git rebase --continue` quando estiver pronto:

```
1 $ git reset HEAD^
2 $ git add README
3 $ git commit -m 'updated README formatting'
4 $ git add lib/simplegit.rb
5 $ git commit -m 'added blame'
6 $ git rebase --continue
```

Git altera o último commit (a5f4a0d) no script, e seu histórico fica assim:

```
1 $ git log -4 --pretty=format:"%h %s"
2 1c002dd added cat-file
3 9b29157 added blame
4 35cfb2b updated README formatting
5 f3cc40e changed my name a bit
```

Mais uma vez, isso altera os SHAs de todos os commits na sua lista, então certifique-se que você não fez o push de nenhum commit dessa lista para um repositório compartilhado.

A Opção Nuclear: filter-branch

Existe uma outra opção de reescrita de histórico que você pode usar se precisa reescrever um grande número de commits em forma de script — por exemplo, mudar seu endereço de e-mail globalmente ou remover um arquivo de cada commit. O comando é `filter-branch`, e ele pode reescrever uma grande parte do seu histórico, então você não deve usá-lo a menos que seu projeto ainda não seja público e outras pessoas não se basearam nos commits que você está para reescrever. Porém, ele pode ser muito útil. Você irá aprender alguns usos comuns para ter uma idéia do que ele é capaz.

Removendo um Arquivo de Cada Commit

Isso é bastante comum de acontecer. Alguém acidentalmente faz um commit sem pensar de um arquivo binário gigante com `git add .`, e você quer removê-lo de todos os lugares. Talvez você tenha feito o commit de um arquivo que continha uma senha, e você quer liberar o código fonte do seu projeto. `filter-branch` é a ferramenta que você pode usar para limpar completamente seu histórico. Para remover um arquivo chamado `passwords.txt` completamente do seu histórico, você pode usar a opção `--tree-filter` em `filter-branch`:

```
1 $ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
2 Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
3 Ref 'refs/heads/master' was rewritten
```

A opção `--tree-filter` executa o comando especificado depois de cada checkout do projeto e faz o commit do resultado novamente. Neste caso, você está removendo um arquivo chamado `passwords.txt` de cada snapshot, quer ele exista ou não. Se você quer remover todos os arquivos

de backup do editor que entraram em commits acidentalmente, você pode executar algo como `git filter-branch --tree-filter "find * -type f -name '*~' -delete" HEAD`.

Você irá assistir o Git reescrever árvores e commits e, em seguida, no final, mover a referência do branch. Geralmente é uma boa idéia fazer isso em um branch de teste e depois fazer um hard-reset do seu branch master depois que você viu que isso era realmente o que queria fazer. Para executar `filter-branch` em todos os seus branches, você pode informar `--all` ao comando.

Fazendo um Subdiretório o Novo Raiz

Digamos que você importou arquivos de outro sistema de controle de versão e ele tem subdiretórios que não fazem sentido (trunk, tags, e outros). Se você quer fazer o subdiretório trunk ser a nova raiz do projeto para todos os commits, `filter-branch` pode ajudar a fazer isso, também:

```
1 $ git filter-branch --subdirectory-filter trunk HEAD
2 Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
3 Ref 'refs/heads/master' was rewritten
```

Agora a sua nova raiz do projeto é o que estava no subdiretório trunk. Git também apagará automaticamente os commits que não afetaram o subdiretório.

Alterando o Endereço de E-Mail Globalmente

Outro caso comum é quando você esqueceu de executar `git config` para configurar seu nome e endereço de e-mail antes de começar a trabalhar, ou talvez você queira liberar o código fonte de um projeto do trabalho e quer mudar o endereço de e-mail profissional para seu endereço pessoal. Em todo caso, você também pode alterar o endereço de e-mail em vários commits com um script `filter-branch`. Você precisa ter cuidado para alterar somente o seu endereço de e-mail, use `--commit-filter`:

```
1 $ git filter-branch --commit-filter '
2     if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
3     then
4         GIT_AUTHOR_NAME="Scott Chacon";
5         GIT_AUTHOR_EMAIL="schacon@example.com";
6         git commit-tree "$@";
7     else
8         git commit-tree "$@";
9     fi' HEAD
```

Isso reescreve cada commit com seu novo endereço. Pelo fato dos commits terem os valores SHA-1 dos pais deles, esse comando altera todos os SHAs dos commits no seu histórico, não apenas aqueles que têm o endereço de e-mail correspondente.

6.5 Ferramentas do Git - Depurando com Git

Depurando com Git

Git também fornece algumas ferramentas para lhe ajudar a depurar problemas em seus projetos. Pelo fato do Git ser projetado para funcionar com quase qualquer tipo de projeto, essas ferramentas são bastante genéricas, mas elas muitas vezes podem ajudá-lo a caçar um bug ou encontrar um culpado quando as coisas dão errado.

Anotação de Arquivo Se você encontrar um erro no seu código e deseja saber quando e por quê ele foi inserido, anotação de arquivo é muitas vezes a melhor ferramenta. Ele mostra qual commit foi o último a modificar cada linha de qualquer arquivo. Portanto, se você ver que um método no seu código está com problemas, você pode anotar o arquivo com `git blame` para ver quando cada linha do método foi editada por último e por quem. Esse exemplo usa a opção `-L` para limitar a saída entre as linhas 12 e 22:

```
1 $ git blame -L 12,22 simplegit.rb
2 ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
3 ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show #{tree}\
4 ")
5 ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
6 ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
7 9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
8 79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9 9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
10 9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
11 42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
12 42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path\
13 }")
14 42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Observe que o primeiro campo é o SHA-1 parcial do commit que alterou a linha pela última vez. Os dois campos seguintes são valores extraídos do commit—o nome do autor e a data de autoria do commit — assim você pode ver facilmente quem alterou a linha e quando. Depois disso vem o número da linha e o conteúdo do arquivo. Observe também as linhas de commit com `^4832fe2`, elas dizem que essas linhas estavam no commit original do arquivo. Esse commit foi quando esse arquivo foi adicionado pela primeira vez nesse projeto, e essas linhas não foram alteradas desde então. Isso é um pouco confuso, porque agora você já viu pelo menos três maneiras diferentes de como Git usa o `^` para modificar um SHA de um commit, mas isso é o que ele significa neste caso.

Outra coisa legal sobre Git é que ele não rastreia mudança de nome explicitamente. Ele grava os snapshots e então tenta descobrir o que foi renomeado implicitamente, após o fato. Uma das características interessantes disso é que você também pode pedir que ele descubra qualquer tipo

de mudança de código. Se você informar `-C` para `git blame`, Git analisa o arquivo que você está anotando e tenta descobrir de onde vieram originalmente os trechos de código, se eles foram copiados de outro lugar. Recentemente, eu estava refatorando um arquivo chamado `GITServerHandler.m` em vários arquivos, um deles era `GITPackUpload.m`. Ao usar "blame" `GITPackUpload.m` com a opção `-C`, eu podia ver de onde vinham os trechos de código originalmente:

```

1  $ git blame -C -L 141,153 GITPackUpload.m
2  f344f58d GITServerHandler.m (Scott 2009-01-04 141)
3  f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
4  f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
5  70befddd GITServerHandler.m (Scott 2009-03-22 144)           //NSLog(@"GATHER COMM
6  ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
7  ad11ac80 GITPackUpload.m (Scott 2009-03-24 146)           NSString *parentSha;
8  ad11ac80 GITPackUpload.m (Scott 2009-03-24 147)           GITCommit *commit = [g
9  ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
10 ad11ac80 GITPackUpload.m (Scott 2009-03-24 149)           //NSLog(@"GATHER COMM
11 ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)
12 56ef2caf GITServerHandler.m (Scott 2009-01-05 151)           if(commit) {
13 56ef2caf GITServerHandler.m (Scott 2009-01-05 152)               [refDict setOb
14 56ef2caf GITServerHandler.m (Scott 2009-01-05 153)

```

Isto é realmente útil. Normalmente, você recebe como commit original, o commit de onde o código foi copiado, porque essa foi a primeira vez que você mecheu nessas linhas do arquivo. Git lhe informa o commit original onde você escreveu aquelas linhas, mesmo que seja em outro arquivo.

Pesquisa Binária

Anotar um arquivo ajuda se você sabe onde está o problema. Se você não sabe o que está o problema, e houveram dezenas ou centenas de commits desde a última vez que você sabe que o código estava funcionando, você provavelmente vai usar `git bisect` para ajudá-lo. O comando `bisect` faz uma pesquisa binária em seu histórico de commits para ajudar você a indentificar o mais rápido possível qual commit inseriu o erro.

Digamos que você acabou de enviar seu código para um ambiente de produção, você recebe relatos de erros sobre algo que não estava acontecendo no seu ambiente de desenvolvimento, e você não tem ideia do motivo do código estar fazendo isso. Você volta para seu código e consegue reproduzir o problema, mas não consegue descobrir o que está errado. Você pode usar o “bisect” para descobrir. Primeiro você executa `git bisect start` para começar, e depois você usa `git bisect bad` para informar ao sistema que o commit atual está quebrado. Em seguida, você deve informar ao “bisect” quando foi a última vez que estava correto, usando `git bisect good [good_commit]`:


```

1 $ git bisect start
2 $ git bisect bad
3 $ git bisect good v1.0
4 Bisecting: 6 revisions left to test after this
5 [ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo

```

Git descobre que cerca de 12 commits estão entre o commit que você informou como o commit correto (v1.0) e a versão atual incorreta, e ele faz o um check out do commit do meio para você. Neste momento, você pode executar seus testes para ver se o problema existe neste commit. Se existir, então ele foi inserido em algum momento antes desse commit do meio; se não existir, então o problema foi inserido algum momento após o commit do meio. Acontece que não há nenhum problema aqui, e você informa isso ao Git digitando `git bisect good` e continua sua jornada:

```

1 $ git bisect good
2 Bisecting: 3 revisions left to test after this
3 [b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing

```

Agora você está em outro commit, na metade do caminho entre aquele que você acabou de testar e o commit incorreto. Você executa os testes novamente e descobre que esse commit está quebrado, você informa isso ao Git com `git bisect bad`:

```

1 $ git bisect bad
2 Bisecting: 1 revisions left to test after this
3 [f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table

```

Este commit está correto, e agora Git tem todas as informações que precisa para determinar quando o problema foi inserido. Ele lhe informa o SHA-1 do primeiro commit incorreto e mostra algumas informações do commit e quais arquivos foram alterados nesse commit para que você possa descobrir o que aconteceu que pode ter inserido esse erro:

```

1 $ git bisect good
2 b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
3 commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
4 Author: PJ Hyett <pjhyett@example.com>
5 Date: Tue Jan 27 14:48:32 2009 -0800
6
7     secure this thing
8
9     :040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
10 f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M config

```

Quando você terminar, você deve executar `git bisect reset` para fazer o “reset” do seu HEAD para onde você estava antes de começar, ou você vai acabar em uma situação estranha:

```
1 $ git bisect reset
```

Essa é uma ferramenta poderosa que pode ajudar você a verificar centenas de commits em minutos para encontrar um erro. Na verdade, se você tiver um script que retorna 0 se o projeto está correto ou algo diferente de 0 se o projeto está incorreto, você pode automatizar totalmente `git bisect`. Primeiro, novamente você informa o escopo fornecendo o commit incorreto e o correto. Você pode fazer isso listando eles com o comando `bisect start` se você quiser, primeiro o commit incorreto e o correto em seguida:

```
1 $ git bisect start HEAD v1.0
2 $ git bisect run test-error.sh
```

Ao fazer isso, é executado automaticamente `test-error.sh` em cada commit até o Git encontrar o primeiro commit quebrado. Você também pode executar algo como `make` ou `make tests` ou qualquer coisa que executa testes automatizados para você.

6.6 Ferramentas do Git - Submódulos

Submódulos

Freqüentemente enquanto você está trabalhando em um projeto, você precisa usar um outro projeto dentro dele. Talvez seja uma biblioteca desenvolvida por terceiros ou que você está desenvolvendo separadamente e usando em vários projetos pai. Um problema comum surge nestes cenários: você quer tratar os dois projetos em separado mas ainda ser capaz de usar um dentro do outro.

Aqui vai um exemplo. Digamos que você está desenvolvendo um site e criando Atom feeds. Em vez de criar seu próprio gerador de Atom, você decide usar uma biblioteca. Provavelmente você terá que incluir esse código de uma biblioteca compartilhada, como um instalação CPAN ou Ruby gem, ou copiar o código fonte na árvore do seu projeto. O problema com a inclusão da biblioteca é que é difícil de personalizar livremente e muitas vezes difícil de fazer o deploy dela, porque você precisa ter certeza de que cada cliente tem essa biblioteca disponível. O problema com a inclusão do código no seu projeto é que é difícil de fazer o merge de qualquer alteração que você faz quando existem modificações do desenvolvedor da biblioteca.

Git resolve esses problemas usando submódulos. Submódulos permitem que você mantenha um repositório Git como um subdiretório de outro repositório Git. Isso permite que você faça o clone de outro repositório dentro do seu projeto e mantenha seus commits separados.

Começando com Submódulos

Digamos que você quer adicionar a biblioteca Rack (um servidor de aplicação web em Ruby) ao seu projeto, manter suas próprias alterações nela, mas continuar fazendo o merge do branch principal. A primeira coisa que você deve fazer é fazer o clone do repositório externo dentro do seu subdiretório. Você adiciona projetos externos como submódulos com o comando `git submodule add`:

```
1 $ git submodule add git://github.com/chneukirchen/rack.git rack
2 Initialized empty Git repository in /opt/subtest/rack/.git/
3 remote: Counting objects: 3181, done.
4 remote: Compressing objects: 100% (1534/1534), done.
5 remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
6 Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
7 Resolving deltas: 100% (1951/1951), done.
```

Agora você tem um projeto do Rack no subdiretório `rack` dentro do seu projeto. Você pode ir nesse subdiretório, fazer alterações, adicionar seus próprios repositórios remotos para fazer o push de suas modificações, fazer o fetch e o merge do repositório original, e outras coisas. Se você executar `git status` logo depois de adicionar o submódulo, você verá duas coisas:

```
1 $ git status
2 # On branch master
3 # Changes to be committed:
4 #   (use "git reset HEAD <file>..." to unstage)
5 #
6 #       new file:   .gitmodules
7 #       new file:   rack
8 #
```

Primeiro você percebe o arquivo `.gitmodules`. Esse é um arquivo de configuração que guarda o mapeamento entre a URL do projeto e o subdiretório local que você usou:

```
1 $ cat .gitmodules
2 [submodule "rack"]
3     path = rack
4     url = git://github.com/chneukirchen/rack.git
```

Se você tem vários submódulos, você terá várias entradas nesse arquivo. É importante notar que esse arquivo está no controle de versão como os outros, como o seu arquivo `.gitignore`. É feito o push e pull com o resto do seu projeto. É como as outras pessoas que fazem o clone do projeto sabem onde pegar os projetos dos submódulos.

O outro item na saída do `git status` é sobre o `rack`. Se você executar `git diff` nele, você vê uma coisa interessante:

```

1 $ git diff --cached rack
2 diff --git a/rack b/rack
3 new file mode 160000
4 index 0000000..08d709f
5 --- /dev/null
6 +++ b/rack
7 @@ -0,0 +1 @@
8 +Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

```

Apesar de rack ser um subdiretório no seu diretório de trabalho, Git vê ele como um submódulo e não rastreia seu conteúdo quando você não está no diretório. Em vez disso, Git o grava como um commit especial desse repositório. Quando você altera e faz commit nesse subdiretório, o projeto-pai nota que o HEAD mudou e grava o commit que você está atualmente; dessa forma, quando outros fizerem o clone desse projeto, eles podem recriar o mesmo ambiente.

Esse é um ponto importante sobre submódulos: você os salva como o commit exato onde eles estão. Você não pode salvar um submódulo no master ou em outra referência simbólica.

Quando você faz o commit, você vê algo assim:

```

1 $ git commit -m 'first commit with submodule rack'
2 [master 0550271] first commit with submodule rack
3 2 files changed, 4 insertions(+), 0 deletions(-)
4 create mode 100644 .gitmodules
5 create mode 160000 rack

```

Note o modo 160000 para a entrada do rack. Esse é um modo especial no Git que basicamente significa que você está salvando um commit como um diretório em vez de um subdiretório ou um arquivo.

Você pode tratar o diretório rack como um projeto separado e atualizar seu projeto-pai de vez em quando com uma referência para o último commit nesse subprojeto. Todos os comandos do Git funcionam independente nos dois diretórios:

```

1 $ git log -1
2 commit 0550271328a0038865aad6331e620cd7238601bb
3 Author: Scott Chacon <schacon@gmail.com>
4 Date: Thu Apr 9 09:03:56 2009 -0700
5
6     first commit with submodule rack
7 $ cd rack/
8 $ git log -1
9 commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

```

```
10 Author: Christian Neukirchen <chneukirchen@gmail.com>
11 Date:   Wed Mar 25 14:49:04 2009 +0100
12
13     Document version change
```

Fazendo Clone de um Projeto com Submódulos

Aqui você vai fazer o clone de um projeto com um submódulo dentro. Quando você recebe um projeto como este, você tem os diretórios que contêm os submódulos, mas nenhum dos arquivos ainda:

```
1 $ git clone git://github.com/schacon/myproject.git
2 Initialized empty Git repository in /opt/myproject/.git/
3 remote: Counting objects: 6, done.
4 remote: Compressing objects: 100% (4/4), done.
5 remote: Total 6 (delta 0), reused 0 (delta 0)
6 Receiving objects: 100% (6/6), done.
7 $ cd myproject
8 $ ls -l
9 total 8
10 -rw-r--r--  1 schacon  admin   3 Apr  9 09:11 README
11 drwxr-xr-x  2 schacon  admin  68 Apr  9 09:11 rack
12 $ ls rack/
13 $
```

O diretório rack está lá, mas vazio. Você precisa executar dois comandos: `git submodule init` para inicializar seu arquivo local de configuração, e `git submodule update` para buscar todos os dados do projeto e recuperar o commit apropriado conforme descrito em seu projeto-pai:

```
1 $ git submodule init
2 Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'ra\
3 ck'
4 $ git submodule update
5 Initialized empty Git repository in /opt/myproject/rack/.git/
6 remote: Counting objects: 3181, done.
7 remote: Compressing objects: 100% (1534/1534), done.
8 remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
9 Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
10 Resolving deltas: 100% (1951/1951), done.
11 Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afc433'
```

Agora seu subdiretório rack está na mesma situação que estava quando você fez o commit antes. Se outro desenvolvedor alterar o código de “rack” e fizer o commit, e você faz o pull e o merge, você vê algo um pouco estranho:

```

1 $ git merge origin/master
2 Updating 0550271..85a3eee
3 Fast forward
4  rack |    2 +-
5  1 files changed, 1 insertions(+), 1 deletions(-)
6 [master*]$ git status
7 # On branch master
8 # Changes not staged for commit:
9 #   (use "git add <file>..." to update what will be committed)
10 #   (use "git checkout -- <file>..." to discard changes in working directory)
11 #
12 #       modified:   rack
13 #

```

Você fez o merge do que é basicamente um mudança para a referência do seu submódulo; mas isso não atualiza o código no diretório do submódulo, parece que você tem um estado sujo no seu diretório de trabalho:

```

1 $ git diff
2 diff --git a/rack b/rack
3 index 6c5e70b..08d709f 160000
4 --- a/rack
5 +++ b/rack
6 @@ -1,1 @@
7 -Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
8 +Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afc433

```

A causa disso é que a referência que você tem para o submódulo não é exatamente o que está no diretório do submódulo. Para corrigir isso, você precisa executar `git submodule update` novamente:

```

1 $ git submodule update
2 remote: Counting objects: 5, done.
3 remote: Compressing objects: 100% (3/3), done.
4 remote: Total 3 (delta 1), reused 2 (delta 0)
5 Unpacking objects: 100% (3/3), done.
6 From git@github.com:schacon/rack
7   08d709f..6c5e70b  master    -> origin/master
8 Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'

```

Você tem que fazer isso toda as vezes que pegar uma alteração de um submódulo no projeto principal. É estranho, mas funciona.

Um problema comum acontece quando um desenvolvedor faz uma alteração local em submódulo mas não a envia para um servidor público. Em seguida, ele faz o commit de uma referência para esse estado que não é público e faz o push do projeto-pai. Quando outros desenvolvedores tentam executar `git submodule update`, o sistema do submódulo não consegue achar o commit para essa referência, porque ela só existe no sistema daquele primeiro desenvolvedor. Se isso acontecer, você verá um erro como este:

```
1 $ git submodule update
2 fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
3 Unable to checkout '6c5e70b984a60b3cecd395edd5b48a7575bf58e0' in submodule path 'ra\
4 ck'
```

Você tem que ver quem alterou o submódulo pela última vez:

```
1 $ git log -1 rack
2 commit 85a3eee996800fcfa91e2119372dd4172bf76678
3 Author: Scott Chacon <schacon@gmail.com>
4 Date: Thu Apr 9 09:19:14 2009 -0700
5
6     added a submodule reference I will never make public. hahahahaha!
```

Em seguida, você envia um e-mail para esse cara e grita com ele.

Superprojetos

Às vezes, desenvolvedores querem obter uma combinação de subdiretórios de um grande projeto, dependendo de qual equipe eles estão. Isso é comum se você está vindo do CVS ou Subversion, onde você define um módulo ou uma coleção de subdiretórios, e você quer manter esse tipo de fluxo de trabalho.

Uma boa maneira de fazer isso no Git é fazer cada subpasta um repositório Git separado e em seguida criar um repositório para um projeto-pai que contém vários submódulos. A vantagem desse modo é que você pode definir mais especificamente os relacionamentos entre os projetos com tags e branches no projeto-pai.

Problemas com Submódulos

Usar submódulos tem seus problemas. Primeiro, você tem que ser relativamente cuidadoso quando estiver trabalhando no diretório do submódulo. Quando você executa `git submodule update`, ele faz o checkout de uma versão específica do projeto, mas fora de um branch. Isso é chamado ter uma cabeça separada (detached HEAD) — isso significa que o HEAD aponta diretamente para um commit, não para uma referência simbólica. O problema é que geralmente você não quer trabalhar

em um ambiente com o HEAD separado, porque é fácil perder alterações. Se você executar `submodule update`, fizer o commit no diretório do submódulo sem criar um branch para trabalhar, e em seguida executar `git submodule update` novamente no projeto-pai sem fazer commit nesse meio tempo, Git irá sobrescrever as alterações sem lhe informar. Tecnicamente você não irá perder o trabalho, mas você não terá um branch apontando para ele, por isso vai ser um pouco difícil de recuperá-lo.

Para evitar esse problema, crie um branch quando for trabalhar em um diretório de um submódulo com `git checkout -b work` ou algo equivalente. Quando você atualizar o submódulo pela segunda vez, ele ainda irá reverter seu trabalho, mas pelo menos você terá uma referência para retornar.

Mudar de branches que contêm submódulos também pode ser complicado. Se você criar um novo branch, adicionar um submódulo nele, e mudar para um branch que não tem o submódulo, você ainda terá o diretório do submódulo como um diretório que não está sendo rastreado:

```
1 $ git checkout -b rack
2 Switched to a new branch "rack"
3 $ git submodule add git@github.com:schacon/rack.git rack
4 Initialized empty Git repository in /opt/myproj/rack/.git/
5 ...
6 Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
7 Resolving deltas: 100% (1952/1952), done.
8 $ git commit -am 'added rack submodule'
9 [rack cc49a69] added rack submodule
10 2 files changed, 4 insertions(+), 0 deletions(-)
11 create mode 100644 .gitmodules
12 create mode 160000 rack
13 $ git checkout master
14 Switched to branch "master"
15 $ git status
16 # On branch master
17 # Untracked files:
18 #   (use "git add <file>..." to include in what will be committed)
19 #
20 #       rack/
```

Você tem que tirá-lo de lá ou removê-lo, em todo caso você tem que fazer o clone novamente quando você voltar — e você pode perder alterações ou branches locais que não foram enviados com um push.

O último problema que muitas pessoas encontram envolve mudar de subdiretórios para submódulos. Se você está rastreando arquivos no seu projeto e quer movê-los para um submódulo, você deve ser cuidadoso ou “o Git vai ficar com raiva de você”. Digamos que você tem os arquivos do “rack” em um subdiretório do seu projeto, e você quer transformá-los em um submódulo. Se você apagar o subdiretório e em seguida executar `submodule add`, Git exhibe isto:


```
1 $ rm -Rf rack/
2 $ git submodule add git@github.com:schacon/rack.git rack
3 'rack' already exists in the index
```

Você tem que retirar o diretório rack da área de seleção primeiro. Depois, você pode adicionar o submódulo:

```
1 $ git rm -r rack
2 $ git submodule add git@github.com:schacon/rack.git rack
3 Initialized empty Git repository in /opt/testsub/rack/.git/
4 remote: Counting objects: 3184, done.
5 remote: Compressing objects: 100% (1465/1465), done.
6 remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
7 Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
8 Resolving deltas: 100% (1952/1952), done.
```

Agora digamos que você fez isso em um branch. Se você tentar mudar para um branch onde esses arquivos ainda estão na árvore em vez de um submódulo — você recebe esse erro:

```
1 $ git checkout master
2 error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

Você tem que mover o diretório do submódulo do rack de lá antes de mudar para um branch que não tem ele:

```
1 $ mv rack /tmp/
2 $ git checkout master
3 Switched to branch "master"
4 $ ls
5 README    rack
```

Em seguida, quando você voltar, você terá um diretório rack vazio. Você pode executar `git submodule update` para fazer o clone novamente, ou mover seu diretório `/tmp/rack` de volta para o diretório vazio.

6.7 Ferramentas do Git - Merge de Sub-árvore (Subtree Merging)

Merge de Sub-árvore (Subtree Merging)

Agora que você viu as dificuldades do sistema de submódulos, vamos ver uma maneira alternativa de resolver o mesmo problema. Quando o Git faz o merge, ele olha para as partes que vão sofrer o merge

e escolhe a estratégia adequada de merge para usar. Se você está fazendo o merge de dois branches, Git usa uma estratégia *recursiva* (*recursive strategy*). Se você está fazendo o merge de mais de dois branches, Git usa a estratégia do polvo (*octopus strategy*). Essas estratégias são automaticamente escolhidas para você, porque a estratégia recursiva pode lidar com situações complexas de merge de três vias — por exemplo, mais de um ancestral comum — mas ele só pode lidar com o merge de dois branches. O merge octopus pode lidar com vários branches mas é cauteloso para evitar conflitos difíceis, por isso ele é escolhido como estratégia padrão se você está tentando fazer o merge de mais de dois branches.

Porém, existem também outras estratégias que você pode escolher. Uma delas é o merge de *sub-árvore*, e você pode usá-lo para lidar com o problema do subprojeto. Aqui você vai ver como resolver o problema do “rack” da seção anterior, mas usando merge de sub-árvore.

A ideia do merge de sub-árvore é que você tem dois projetos, e um deles está mapeado para um subdiretório do outro e vice-versa. Quando você escolhe um merge de sub-árvore, Git é inteligente o bastante para descobrir que um é uma sub-árvore do outro e faz o merge adequado — é incrível.

Primeiro você adiciona a aplicação Rack em seu projeto. Você adiciona o projeto Rack como uma referência remota no seu projeto e então faz o checkout dele em um branch:

```
1  $ git remote add rack_remote git@github.com:schacon/rack.git
2  $ git fetch rack_remote
3  warning: no common commits
4  remote: Counting objects: 3184, done.
5  remote: Compressing objects: 100% (1465/1465), done.
6  remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
7  Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
8  Resolving deltas: 100% (1952/1952), done.
9  From git@github.com:schacon/rack
10 * [new branch]      build      -> rack_remote/build
11 * [new branch]      master     -> rack_remote/master
12 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
13 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
14 $ git checkout -b rack_branch rack_remote/master
15 Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
16 Switched to a new branch "rack_branch"
```

Agora você tem a raiz do projeto Rack no seu branch `rack_branch` e o seu projeto no branch `master`. Se você fizer o checkout de um e depois do outro, você pode ver que eles têm raízes de projeto diferentes:

```

1 $ ls
2 AUTHORS          KNOWN-ISSUES  Rakefile      contrib      lib
3 COPYING          README       bin           example      test
4 $ git checkout master
5 Switched to branch "master"
6 $ ls
7 README

```

Você quer colocar o projeto Rack no seu projeto master como um subdiretório. Você pode fazer isso no Git com `git read-tree`. Você irá aprender mais sobre `read-tree` e seus companheiros no Capítulo 9, mas por enquanto saiba que ele escreve a raiz da árvore de um branch na sua área de seleção e diretório de trabalho. Você volta para o branch master, você coloca o branch rack no subdiretório rack no branch master do seu projeto principal:

```

1 $ git read-tree --prefix=rack/ -u rack_branch

```

Quando você faz o commit, parece que você tem todos os arquivos do Rack nesse subdiretório — como se você tivesse copiado de um arquivo. O que é interessante é que você pode facilmente fazer merge de alterações de um branch para o outro. Assim, se o projeto Rack for atualizado, você pode fazer um pull das modificações mudando para o branch e fazendo o pull:

```

1 $ git checkout rack_branch
2 $ git pull

```

Em seguida, você pode fazer o merge dessas alterações no seu branch master. Você pode usar `git merge -s subtree` e ele irá funcionar normalmente; mas o Git também irá fazer o merge do histórico, coisa que você provavelmente não quer. Para trazer as alterações e preencher a mensagem de commit, use as opções `--squash` e `--no-commit` com a opção de estratégia `-s subtree`:

```

1 $ git checkout master
2 $ git merge --squash -s subtree --no-commit rack_branch
3 Squash commit -- not updating HEAD
4 Automatic merge went well; stopped before committing as requested

```

Foi feito o merge de todas suas alterações do projeto Rack e elas estão prontas para o commit local. Você também pode fazer o oposto — alterar o subdiretório rack do seu branch master e depois fazer o merge delas no seu branch rack_branch para enviá-las para os mantenedores do projeto ou para o projeto original.

Para ver o diff entre o que você tem no seu subdiretório rack e o código no seu branch rack_branch — para ver se você precisa fazer o merge deles — você não pode usar o comando `diff`. Em vez disso, você precisa executar `git diff-tree` com o branch que você quer comparar:

```
1 $ git diff-tree -p rack_branch
```

Ou, para comparar o que tem no seu subdiretório `rack` com o que estava no branch `master` no servidor na última vez que você se conectou a ele, você pode executar

```
1 $ git diff-tree -p rack_remote/master
```

6.8 Ferramentas do Git - Sumário

Sumário

Você viu algumas ferramentas avançadas que permitem que você manipule seus commits e área de seleção mais precisamente. Quando você notar problemas, você deve ser capaz de descobrir facilmente qual commit os introduziram, quando, e quem. Se você quer usar subprojetos em seu projeto, você aprendeu algumas maneiras de resolver essas necessidades. Neste momento, você deve ser capaz de fazer a maioria das coisas que você precisa diariamente com o Git na linha de comando e se sentir confortável fazendo isso.

Capítulo 7 - Customizando o Git

Até agora, eu mostrei o básico de como o Git funciona, como usá-lo e apresentei algumas ferramentas que o Git provê para ajudar a usá-lo de forma fácil e eficiente. Neste capítulo, eu mostrarei algumas operações que você pode usar para fazer operações com o Git de uma maneira mais customizada, introduzindo várias configurações importantes e um sistemas de hooks. Com essas ferramentas, será fácil trabalhar com o Git da melhor forma para você, sua empresa ou qualquer grupo.

7.1 Customizando o Git - Configuração do Git

Configuração do Git

Como você viu brevemente no Capítulo 1, você pode configurar o Git com o comando `git config`. Uma das primeiras coisas que você fez foi configurar seu nome e endereço de email:

```
1 $ git config --global user.name "John Doe"
2 $ git config --global user.email johndoe@example.com
```

Agora você vai aprender algumas opções mais interessantes que você pode definir dessa maneira para customizar o uso do Git.

Você viu alguns detalhes simples de configuração do Git no primeiro capítulo, mas vou passar por eles de novo rapidamente. Git usa uma série de arquivos de configuração para determinar comportamentos não-padrão que você pode querer utilizar. O primeiro lugar que o Git procura por estes valores é no arquivo `/etc/gitconfig`, que contém os valores para todos os usuários do sistema e todos os seus repositórios. Se você passar a opção `--system` para `git config`, ele lê e escreve a partir deste arquivo especificamente.

O próximo lugar que o Git olha é no arquivo `~/.gitconfig`, que é específico para cada usuário. Você pode fazer o Git ler e escrever neste arquivo, passando a opção `--global`.

Finalmente, Git procura por valores de configuração no arquivo de configuração no diretório Git (`.git/config`) de qualquer repositório que você esteja usando atualmente. Estes valores são específicos para esse repositório. Cada nível substitui valores no nível anterior, então, valores em `.git/config` sobrepõem valores em `/etc/gitconfig`. Você também pode definir esses valores manualmente, editando o arquivo e inserindo a sintaxe correta mas, é geralmente mais fácil executar o comando `git config`.

Configuração Básica do Cliente

As opções de configuração reconhecidas pelo Git se dividem em duas categorias: lado cliente e lado servidor. A maioria das opções são do lado cliente e utilizadas para configurar suas preferências pessoais de trabalho. Apesar de haverem muitas opções disponíveis, só cobrirei as que são comumente usadas ou podem afetar significativamente o fluxo de trabalho. Muitas opções são úteis apenas em casos extremos que não mostraremos aqui. Se você quiser ver uma lista de todas as opções que a sua versão do Git reconhece, você pode executar

```
1 $ git config --help
```

A página do manual do `git config` lista todas as opções disponíveis com um pouco de detalhe.

core.editor

Por padrão, o Git usa o editor de texto que você definiu como padrão no Shell ou então reverte para o editor Vi para criar e editar suas mensagens de commit e tags. Para alterar esse padrão, você pode usar a opção `core.editor`:

```
1 $ git config --global core.editor emacs
```

Agora, não importa o que esteja definido como seu editor padrão, o Git usará o editor Emacs.

commit.template

Se você ajustar esta opção como um caminho de um arquivo em seu sistema, o Git vai usar esse arquivo como o padrão de mensagem quando você fizer um commit. Por exemplo, suponha que você crie um arquivo de modelo em `$HOME/.gitmessage.txt` que se parece com este:

```
1 subject line
2
3 what happened
4
5 [ticket: X]
```

Para dizer ao Git para usá-lo como a mensagem padrão que aparece em seu editor quando você executar o `git commit`, defina o valor de configuração `commit.template`:

```
1 $ git config --global commit.template $HOME/.gitmessage.txt
2 $ git commit
```

Então, o editor irá abrir com algo parecido com isto quando você fizer um commit:

```

1  subject line
2
3  what happened
4
5  [ticket: X]
6  # Please enter the commit message for your changes. Lines starting
7  # with '#' will be ignored, and an empty message aborts the commit.
8  # On branch master
9  # Changes to be committed:
10 #   (use "git reset HEAD <file>..." to unstage)
11 #
12 # modified:   lib/test.rb
13 #
14 ~
15 ~
16 ".git/COMMIT_EDITMSG" 14L, 297C

```

Se você tiver uma política de mensagens de commit, colocando um modelo para essa política em seu sistema e configurando o Git para usá-lo por padrão pode ajudar a aumentar a chance de que a política seja seguida regularmente.

core.pager

A configuração `core.pager` determina qual pager é usado quando a saída do Git possui várias páginas, como quando são usados os comandos `log` e `diff`. Você pode configurá-lo para `more` ou para o seu pager favorito (por padrão, é `less`), ou você pode desativá-lo, definindo uma string em branco:

```
1 $ git config --global core.pager ''
```

Se você executar isso, Git irá paginar toda a saída de todos os comandos, não importando quão longo eles sejam.

user.signingkey

Se você estiver fazendo annotated tags assinadas (como discutido no Capítulo 2), definir a sua chave de assinatura GPG como uma configuração torna as coisas mais fáceis. Defina o ID da chave assim:

```
1 $ git config --global user.signingkey <gpg-key-id>
```

Agora, você pode assinar tags sem ter de especificar a sua chave toda hora com o comando `git tag`:

```
1 $ git tag -s <tag-name>
```

core.excludesfile

Você pode colocar padrões em seu arquivo de projeto `.gitignore` para que o Git veja-os como arquivos untracked ou tentar coloca-los como staged quando executar o `git add` sobre eles, como discutido no Capítulo 2. No entanto, se você quiser que outro arquivo fora do seu projeto mantenha esses valores ou tenham valores extras, você pode dizer ao Git onde o arquivo com a opção `core.excludesfile` está. Basta configurá-lo para o caminho de um arquivo que tem conteúdo semelhante ao que um arquivo `.gitignore` teria.

help.autocorrect

Esta opção está disponível apenas no Git 1.6.1 e posteriores. Se você digitar um comando no Git 1.6, ele mostrará algo como isto:

```
1 $ git com
2 git: 'com' is not a git-command. See 'git --help'.
3
4 Did you mean this?
5     commit
```

Se você definir `help.autocorrect` para 1, Git automaticamente executará o comando se houver apenas uma possibilidade neste cenário.

Cores no Git

Git pode colorir a sua saída para o terminal, o que pode ajudá-lo visualmente a analisar a saída mais rápido e facilmente. Um número de opções pode ajudar a definir a colorização de sua preferência.

color.ui

Git automaticamente coloriza a maioria de sua saída, se você pedir para ele. Você pode ser muito específico sobre o que você quer e como colorir; mas para ativar a colorização padrão do terminal, defina `color.ui` para `true`:

```
1 $ git config --global color.ui true
```

Quando esse valor é definido, Git coloriza a saída do terminal. Outras configurações possíveis são `false`, que nunca coloriza a saída, e `always`, que coloriza sempre, mesmo que você esteja redirecionando comandos do Git para um arquivo ou através de um pipe para outro comando. Esta configuração foi adicionado na versão 1.5.5 do Git, se você tem uma versão mais antiga, você terá que especificar todas as configurações de cores individualmente.

Você dificilmente vai querer usar `color.ui = always`. Na maioria dos cenários, se você quiser códigos coloridos em sua saída redirecionada, você pode passar a opção `--color` para forçar o comando Git a usar códigos de cores. O `color.ui = true` é o que provavelmente você vai querer usar.

color.*

Se você quiser ser mais específico sobre quais e como os comandos são colorizados, ou se você tem uma versão mais antiga do Git, o Git oferece configurações específicas para colorir. Cada uma destas pode ser ajustada para `true`, `false`, ou `always`:

- 1 `color.branch`
- 2 `color.diff`
- 3 `color.interactive`
- 4 `color.status`

Além disso, cada uma delas tem sub-opções que você pode usar para definir cores específicas para partes da saída, se você quiser substituir cada cor. Por exemplo, para definir a informação meta na sua saída do diff para texto azul, fundo preto e texto em negrito, você pode executar

- 1 `$ git config --global color.diff.meta "blue black bold"`

Você pode definir a cor para qualquer um dos seguintes valores: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, ou `white`. Se você quiser um atributo como negrito no exemplo anterior, você pode escolher entre `bold`, `dim`, `ul`, `blink`, e `reverse`.

Veja a página de manual (manpage) do `git config` para saber todas as sub-opções que você pode configurar.

Ferramenta Externa de Merge e Diff

Embora o Git tenha uma implementação interna do diff, que é o que você estava usando, você pode configurar uma ferramenta externa. Você pode configurar uma ferramenta gráfica de merge para resolução de conflitos, em vez de ter de resolver conflitos manualmente. Vou demonstrar a configuração do Perforce Visual Merge Tool (P4Merge) para fazer suas diffs e fazer merge de resoluções, porque é uma boa ferramenta gráfica e é gratuita.

Se você quiser experimentar, P4Merge funciona em todas as principais plataformas, então você deve ser capaz de usá-lo. Vou usar nomes de caminho nos exemplos que funcionam em sistemas Mac e Linux; para Windows, você vai ter que mudar `/usr/local/bin` para um caminho executável em seu ambiente.

Você pode baixar P4Merge aqui:

- 1 <http://www.perforce.com/perforce/downloads/component.html>

Para começar, você vai configurar um script para executar seus comandos. Vou usar o caminho para o executável Mac; em outros sistemas, este será onde o seu binário do `p4merge` está instalado. Configure um script chamado `extMerge` que chama seu binário com todos os argumentos necessários:

```

1 $ cat /usr/local/bin/extMerge
2 #!/bin/sh/Applications/p4merge.app/Contents/MacOS/p4merge $*

```

Um wrapper diff verifica se sete argumentos são fornecidos e passa dois deles para o seu script de merge. Por padrão, o Git passa os seguintes argumentos para o programa diff:

```

1 path old-file old-hex old-mode new-file new-hex new-mode

```

Já que você só quer os argumentos old-file e new-file, você pode usar o script para passar o que você precisa.

```

1 $ cat /usr/local/bin/extDiff
2 #!/bin/sh
3 [ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"

```

Você também precisa ter certeza de que essas ferramentas são executáveis:

```

1 $ sudo chmod +x /usr/local/bin/extMerge
2 $ sudo chmod +x /usr/local/bin/extDiff

```

Agora você pode configurar o arquivo de configuração para usar a sua ferramenta de diff customizada. Existem algumas configurações personalizadas: `merge.tool` para dizer ao Git qual a estratégia a utilizar, `mergetool.*.cmd` para especificar como executar o comando, `mergetool.trustExitCode` para dizer ao Git se o código de saída do programa indica uma resolução de merge com sucesso ou não, e `diff.external` para dizer ao Git o comando a ser executado para diffs. Assim, você pode executar quatro comandos de configuração

```

1 $ git config --global merge.tool extMerge
2 $ git config --global mergetool.extMerge.cmd \
3     'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
4 $ git config --global mergetool.trustExitCode false
5 $ git config --global diff.external extDiff

```

ou você pode editar o seu arquivo `~/.gitconfig` para adicionar estas linhas.:

```

1  [merge]
2      tool = extMerge
3  [mergetool "extMerge"]
4      cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
5      trustExitCode = false
6  [diff]
7      external = extDiff

```

Depois que tudo isso seja definido, se você executar comandos diff como este:

```
1 $ git diff 32d1776b1^ 32d1776b1
```

Em vez de ter a saída do diff na linha de comando, Git inicia o P4Merge, como mostra a Figura 7-1.

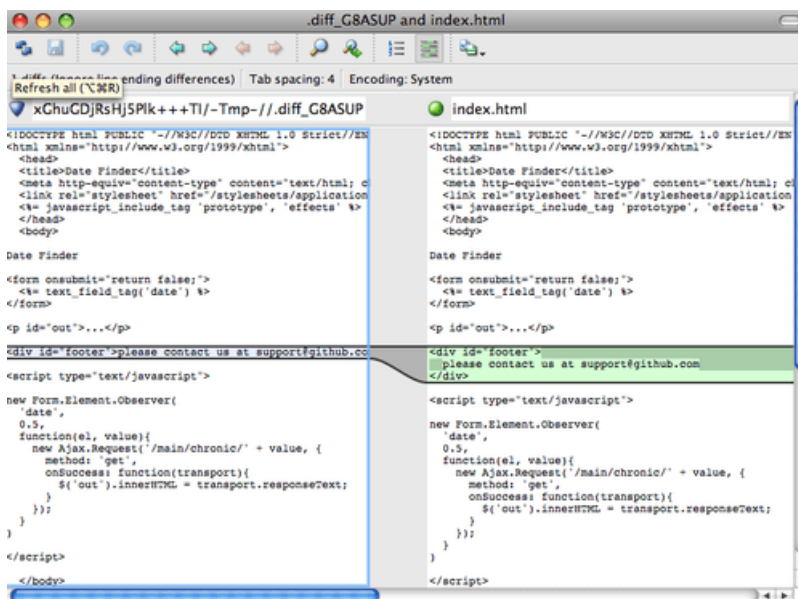


Figura 7-1. P4Merge

Se você tentar mesclar dois branches e, posteriormente, ter conflitos de mesclagem, você pode executar o comando `git mergetool`, que iniciará o P4Merge para deixá-lo resolver os conflitos através dessa ferramenta gráfica.

A coisa boa sobre esta configuração é que você pode mudar o seu diff e ferramentas de merge facilmente. Por exemplo, para mudar suas ferramentas `extdiff` e `extMerge` para executar a ferramenta `KDiff3` no lugar delas, tudo que você tem a fazer é editar seu arquivo `extMerge`:

```

1 $ cat /usr/local/bin/extMerge
2 #!/bin/sh/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*

```

Agora, o Git irá utilizar a ferramenta KDiff3 para visualizar diffs e resolução de conflitos de merge.

O Git vem pré-configurado para usar uma série de outras ferramentas de resolução de merge sem ter que definir a configuração cmd. Você pode definir a sua ferramenta de mesclagem para kdiff3, opendiff, tkdiff, meld, xxdiff, emerge, vimdiff, ou gvimdiff. Se você não estiver interessado em usar o KDiff3 para diff mas quer usá-lo apenas para a resolução de merges, e o comando kdiff3 está no seu path, então você pode executar

```
1 $ git config --global merge.tool kdiff3
```

Se você executar isto ao invés de configurar os arquivos `extMerge` e `extDiff`, Git irá usar o KDiff3 para resolução de merges e a ferramenta diff padrão do Git.

Formatação e Espaços em Branco

Formatação e problemas de espaço em branco são alguns dos problemas mais frustrantes e sutis que muitos desenvolvedores encontram ao colaborar, especialmente em ambientes multi-plataforma. É muito fácil que patches ou outros trabalhos de colabores introduzam mudanças sutis como espaços em branco porque os editores os inserem silenciosamente ou programadores Windows adicionam quebras de linha em projetos multi-plataforma. Git tem algumas opções de configuração para ajudar com estas questões.

core.autocrlf

Se você está programando no Windows ou outro sistema, mas trabalha com pessoas que estão programando em Windows, você provavelmente vai encontrar problemas de quebra de linha em algum momento. Isso porque o Windows usa tanto o caráter carriage-return e um carácter linefeed para novas linhas em seus arquivos, enquanto os sistemas Mac e Linux usam apenas o carácter linefeed. Este é um fato sutil, mas extremamente irritante em trabalhos multi-plataforma.

O Git pode lidar com isso auto-convertendo finais de linha CRLF para LF quando você faz um commit, e vice-versa, quando se faz um checkout de código em seu sistema de arquivos. Você pode ativar esta funcionalidade com a configuração `core.autocrlf`. Se você estiver em uma máquina Windows, defina-o `true` — este converte terminações LF em CRLF quando você faz um checkout do código:

```
1 $ git config --global core.autocrlf true
```

Se você estiver em um sistema Linux ou Mac que usam os finais de linha LF, então você não irá querer que o Git automaticamente converta-os quando você fizer o check-out dos arquivos, no entanto, se um arquivo com terminações CRLF acidentalmente for introduzido, então você pode querer que o Git corrija-o. Você pode dizer ao Git para converter CRLF para LF no commit, mas não o contrário definindo `core.autocrlf` para entrada:

```
1 $ git config --global core.autocrlf input
```

Esta configuração deve deixá-lo com terminações CRLF em checkouts Windows, mas terminações LF em sistemas Mac e Linux e no repositório.

Se você é um programador Windows fazendo um projeto somente para Windows, então você pode desativar essa funcionalidade, registrando os CRLF no repositório, definindo o valor de configuração para false:

```
1 $ git config --global core.autocrlf false
```

core.whitespace

Git vem pré-configurado para detectar e corrigir alguns problemas de espaço em branco. Ele pode olhar por quatro problemas principais relacionados a espaços em branco — duas são ativadas por padrão e podem ser desativadas, e duas não são ativadas por padrão, mas podem ser ativadas.

As duas que são ativadas por padrão são `trailing-space`, que procura por espaços no final de uma linha, e `space-before-tab`, que procura por espaços antes de tabulações no início de uma linha.

As duas que estão desativadas por padrão, mas podem ser ativadas são `indent-with-non-tab`, que procura por linhas que começam com oito ou mais espaços em vez de tabulações, e `cr-at-eol`, que diz ao Git que carriage returns no final das linhas estão OK.

Você pode dizer ao Git quais destes você quer habilitado alterando a opção `core.whitespace` para os valores que deseja on ou off, separados por vírgulas. Você pode desabilitar as configurações, quer deixando-as fora da string de definição ou adicionando um - na frente do valor. Por exemplo, se você quiser tudo, menos `cr-at-eol`, você pode fazer isso:

```
1 $ git config --global core.whitespace \
2   trailing-space,space-before-tab,indent-with-non-tab
```

Git irá detectar esses problemas quando você executar um comando `git diff` e tentar colori-los de modo que você pode, eventualmente, corrigi-los antes de fazer o commit. Ele também irá usar esses valores para ajudar quando você aplicar patches com `git apply`. Quando você estiver aplicando patches, você pode pedir ao Git para avisá-lo se estiver aplicando patches com problemas de espaço em branco:

```
1 $ git apply --whitespace=warn <patch>
```

Ou você pode deixar o Git tentar corrigir automaticamente o problema antes de aplicar o patch:

```
1 $ git apply --whitespace=fix <patch>
```

Essas opções se aplicam ao comando `git rebase` também. Se você commitou problemas de espaço em branco, mas ainda não fez um push, você pode executar um rebase com a opção `--whitespace=fix` para que o Git automaticamente corrija problemas de espaço em branco, como faz com os patches.

Configuração do Servidor

Não existem muitas opções de configuração disponíveis para o lado servidor do Git, mas há algumas interessantes que você pode querer aprender.

`receive.fsckObjects`

Por padrão, o Git não verifica a consistência de todos os objetos que ele recebe durante um push. Embora o Git possa certificar-se de que cada objeto ainda corresponde ao seu SHA-1 checksum e aponta para objetos válidos, ele não faz isso por padrão em cada push. Esta é uma operação relativamente custosa e pode adicionar uma grande quantidade de tempo para cada push, de acordo com o tamanho do repositório ou do push. Se você quiser que o Git verifique a consistência dos objetos em cada push, você pode forçá-lo a fazê-lo definindo `receive.fsckObjects` como `true`:

```
1 $ git config --system receive.fsckObjects true
```

Agora, o Git irá verificar a integridade do seu repositório antes que cada push seja aceito para garantir que clientes defeituosos não estejam introduzindo dados corrompidos.

`receive.denyNonFastForwards`

Se você fizer o rebase de commits já enviados com push e então tentar fazer outro push, ou tentar fazer um push de um commit para um branch remoto que não contenha o commit que o branch remoto atualmente aponta, sua ação será negada. Isso geralmente é uma boa política; mas, no caso do rebase, você pode determinar que você saiba o que está fazendo e pode forçar a atualização do branch remoto com um `-f` no seu comando push.

Para desativar a capacidade de forçar updates em branches remotos para referências não fast-forward, defina `receive.denyNonFastForwards`:

```
1 $ git config --system receive.denyNonFastForwards true
```

A outra forma de fazer isso é através dos hooks em lado servidor, que eu vou falar daqui a pouco. Essa abordagem permite que você faça coisas mais complexas como negar não fast-forwards para um determinado conjunto de usuários.

`receive.denyDeletes`

Uma das soluções para a política `denyNonFastForwards` é o usuário excluir o branch e depois fazer um push de volta com a nova referência. Nas versões mais recentes do Git (a partir da versão 1.6.1), você pode definir `receive.denyDeletes` como `true`:

```
1 $ git config --system receive.denyDeletes true
```

Isto nega exclusão de branches e tags em um push — nenhum usuário pode fazê-lo. Para remover branches remotas, você deve remover os arquivos ref do servidor manualmente. Existem também formas mais interessantes de fazer isso de acordo com o usuário através de ACLs, como você vai aprender no final deste capítulo.

7.2 Customizando o Git - Atributos Git

Atributos Git

Algumas dessas configurações também podem ser especificadas para um path, de modo que o Git aplique essas configurações só para um subdiretório ou conjunto de arquivos. Essas configurações de path específicas são chamadas atributos Git e são definidas em um arquivo `.gitattributes` ou em um de seus diretórios (normalmente a raiz de seu projeto) ou no arquivo `.git/info/attributes` se você não desejar que o arquivo de atributos seja commitado com o seu projeto.

Usando atributos, você pode fazer coisas como especificar estratégias de merge separadas para arquivos individuais ou pastas no seu projeto, dizer ao Git como fazer diff de arquivos não textuais, ou mandar o Git filtrar conteúdos antes de fazer o checkout para dentro ou fora do Git. Nesta seção, você vai aprender sobre alguns dos atributos que podem ser configurados em seus paths de seu projeto Git e ver alguns exemplos de como usar esse recurso na prática.

Arquivos Binários

Um truque legal para o qual você pode usar atributos Git é dizendo ao Git quais arquivos são binários (em casos que de outra forma ele não pode ser capaz de descobrir) e dando ao Git instruções especiais sobre como lidar com esses arquivos. Por exemplo, alguns arquivos de texto podem ser gerados por máquina e não é possível usar diff neles, enquanto que em alguns arquivos binários pode ser usado o diff — você verá como dizer ao Git qual é qual.

Identificando Arquivos Binários

Alguns arquivos parecem com arquivos de texto, mas para todos os efeitos devem ser tratados como dados binários. Por exemplo, projetos Xcode no Mac contém um arquivo que termina em `.pbxproj`, que é basicamente um conjunto de dados de JSON (formato de dados em texto simples JavaScript), escrito no disco pela IDE que registra as configurações de builds e assim por diante. Embora seja tecnicamente um arquivo de texto, porque é tudo ASCII, você não quer tratá-lo como tal, porque ele é na verdade um banco de dados leve — você não pode fazer um merge do conteúdo, se duas pessoas o mudaram, e diffs geralmente não são úteis. O arquivo é para ser lido pelo computador. Em essência, você quer tratá-lo como um arquivo binário.

Para dizer ao Git para tratar todos os arquivos `pbxproj` como dados binários, adicione a seguinte linha ao seu arquivo `.gitattributes`:

```
1 *.pbxproj -crlf -diff
```

Agora, o Git não vai tentar converter ou corrigir problemas CRLF; nem vai tentar calcular ou imprimir um diff para mudanças nesse arquivo quando você executar `show` ou `git diff` em seu projeto. Na série 1.6 do Git, você também pode usar uma macro que significa `-crlf -diff`:

```
1 *.pbxproj binary
```

Diff de Arquivos Binários

Na série 1.6 do Git, você pode usar a funcionalidade de atributos do Git para fazer diff de arquivos binários. Você faz isso dizendo ao Git como converter os dados binários em um formato de texto que pode ser comparado através do diff normal.

Arquivos do MS Word

Como este é um recurso muito legal e não muito conhecido, eu vou mostrar alguns exemplos. Primeiro, você vai usar esta técnica para resolver um dos problemas mais irritantes conhecidos pela humanidade: controlar a versão de documentos Word. Todo mundo sabe que o Word é o editor mais horrível que existe, mas, estranhamente, todo mundo o usa. Se você quiser controlar a versão de documentos do Word, você pode colocá-los em um repositório Git e fazer um commit de vez em quando; mas o que de bom tem isso? Se você executar `git diff` normalmente, você só verá algo como isto:

```
1 $ git diff
2 diff --git a/chapter1.doc b/chapter1.doc
3 index 88839c4..4afcb7c 100644
4 Binary files a/chapter1.doc and b/chapter1.doc differ
```

Você não pode comparar diretamente duas versões, a menos que você verifique-as manualmente, certo? Acontece que você pode fazer isso muito bem usando atributos Git. Coloque a seguinte linha no seu arquivo `.gitattributes`:

```
1 *.doc diff=word
```

Isto diz ao Git que qualquer arquivo que corresponde a esse padrão (`.doc`) deve usar o filtro “word” quando você tentar ver um diff que contém alterações. O que é o filtro “word”? Você tem que configurá-lo. Aqui você vai configurar o Git para usar o programa `strings` para converter documentos do Word em arquivos de texto legível, o que poderá ser visto corretamente no diff:

```
1 $ git config diff.word.textconv strings
```


Este comando adiciona uma seção no seu `.git/config` que se parece com isto: `[diff "word"] textconv = strings`

Nota: Há diferentes tipos de arquivos `.doc`, alguns usam uma codificação UTF-16 ou outras “páginas de código” e `strings` não vão encontrar nada de útil lá. Seu resultado pode variar.

Agora o Git sabe que se tentar fazer uma comparação entre os dois snapshots, e qualquer um dos arquivos terminam em `.doc`, ele deve executar esses arquivos através do filtro “word”, que é definido como o programa `strings`. Isso cria versões em texto de arquivos do Word antes de tentar o diff.

Aqui está um exemplo. Eu coloquei um capítulo deste livro em Git, acrescentei algum texto a um parágrafo, e salvei o documento. Então, eu executei `git diff` para ver o que mudou:

```
1 $ git diff
2 diff --git a/chapter1.doc b/chapter1.doc
3 index c1c8a0a..b93c9e4 100644
4 --- a/chapter1.doc
5 +++ b/chapter1.doc
6 @@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
7  re going to cover how to get it and set it up for the first time if you don
8  t already have it on your system.
9  In Chapter Two we will go over basic Git usage - how to use Git for the 80%
10 -s going on, modify stuff and contribute changes. If the book spontaneously
11 +s going on, modify stuff and contribute changes. If the book spontaneously
12 +Let's see if this works.
```

Git com sucesso e de forma sucinta me diz que eu adicionei a string “Let’s see if this works”, o que é correto. Não é perfeito — ele acrescenta um monte de coisas aleatórias no final — mas certamente funciona. Se você pode encontrar ou escrever um conversor de Word em texto simples que funciona bem o suficiente, esta solução provavelmente será incrivelmente eficaz. No entanto, `strings` está disponível na maioria dos sistemas Mac e Linux, por isso pode ser uma primeira boa tentativa para fazer isso com muitos formatos binários.

Documentos de Texto OpenDocument

A mesma abordagem que usamos para arquivos do MS Word (`*.doc`) pode ser usada para arquivos de texto OpenDocument (`*.odt`) criados pelo OpenOffice.org.

Adicione a seguinte linha ao seu arquivo `.gitattributes`:

```
1 *.odt diff=odt
```

Agora configure o filtro `diff odt` em `.git/config`:

```

1  [diff "odt"]
2      binary = true
3      textconv = /usr/local/bin/odt-to-txt

```

Arquivos OpenDocument são na verdade diretórios zipados contendo vários arquivos (o conteúdo em um formato XML, folhas de estilo, imagens, etc.) Vamos precisar escrever um script para extrair o conteúdo e devolvê-lo como texto simples. Crie o arquivo `/usr/local/bin/odt-to-txt` (você é pode colocá-lo em um diretório diferente) com o seguinte conteúdo:

```

1  #!/usr/bin/env perl
2  # Simplistic OpenDocument Text (.odt) to plain text converter.
3  # Author: Philipp Kempgen
4
5  if (! defined($ARGV[0])) {
6      print STDERR "No filename given!\n";
7      print STDERR "Usage: $0 filename\n";
8      exit 1;
9  }
10
11 my $content = '';
12 open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;
13 {
14     local $/ = undef; # slurp mode
15     $content = <$fh>;
16 }
17 close $fh;
18 $_ = $content;
19 s/<text:span\b[^>]*>//g; # remove spans
20 s/<text:h\b[^>]*>/\n\n*****/g; # headers
21 s/<text:list-item\b[^>]*>\s*<text:p\b[^>]*>/\n -- /g; # list items
22 s/<text:list\b[^>]*>/\n\n/g; # lists
23 s/<text:p\b[^>]*>/\n /g; # paragraphs
24 s/<[^>]+>//g; # remove all XML tags
25 s/\n{2,}/\n\n/g; # remove multiple blank lines
26 s/\A\n+//; # remove leading blank lines
27 print "\n", $_, "\n\n";

```

E torne-o executável

```

1  chmod +x /usr/local/bin/odt-to-txt

```

Agora `git diff` será capaz de dizer o que mudou em arquivos `.odt`.

Outro problema interessante que você pode resolver desta forma envolve o `diff` de arquivos de imagem. Uma maneira de fazer isso é passar arquivos PNG através de um filtro que extrai suas informações EXIF — metadados que são gravados com a maioria dos formatos de imagem. Se você baixar e instalar o programa `exiftool`, você pode usá-lo para converter suas imagens em texto sobre os metadados, assim pelo menos o `diff` vai mostrar uma representação textual de todas as mudanças que aconteceram:

```
1 $ echo '*.png diff=exif' >> .gitattributes
2 $ git config diff.exif.textconv exiftool
```

Se você substituir uma imagem em seu projeto e executar o `git diff`, você verá algo como isto:

```
1 diff --git a/image.png b/image.png
2 index 88839c4..4afcb7c 100644
3 --- a/image.png
4 +++ b/image.png
5 @@ -1,12 +1,12 @@
6  ExifTool Version Number      : 7.74
7  -File Size                   : 70 kB
8  -File Modification Date/Time : 2009:04:21 07:02:45-07:00
9  +File Size                   : 94 kB
10 +File Modification Date/Time  : 2009:04:21 07:02:43-07:00
11  File Type                   : PNG
12  MIME Type                   : image/png
13  -Image Width                 : 1058
14  -Image Height                : 889
15  +Image Width                 : 1056
16  +Image Height                : 827
17  Bit Depth                   : 8
18  Color Type                   : RGB with Alpha
```

Você pode facilmente ver que o tamanho do arquivo e as dimensões da imagem sofreram alterações.

Expansão de Palavra-chave

Expansão de Palavra-chave no estilo SVN ou CVS são frequentemente solicitados pelos desenvolvedores acostumados com estes sistemas. O principal problema disso no Git é que você não pode modificar um arquivo com informações sobre o commit depois que você já fez o commit, porque o Git cria os checksums dos arquivos primeiro. No entanto, você pode injetar texto em um arquivo quando é feito o checkout dele e removê-lo novamente antes de ser adicionado a um commit. Atributos Git oferecem duas maneiras de fazer isso.

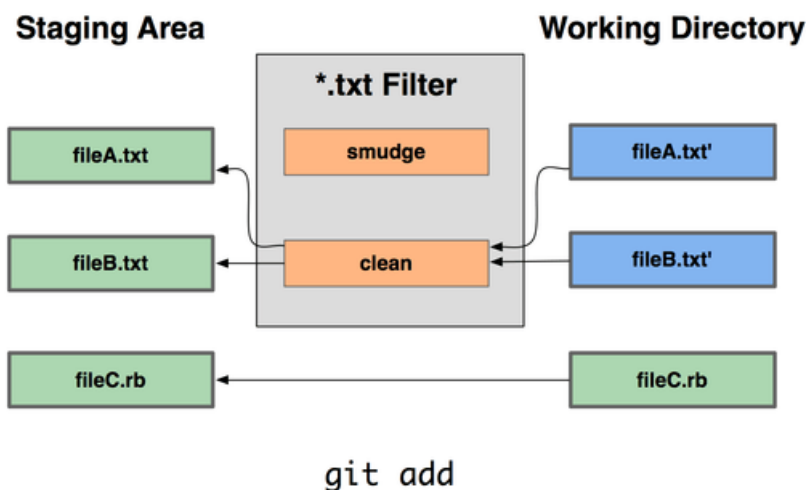


Figura 7-3. O filtro “clean” é rodado quando arquivos passam para o estado staged.

A mensagem original do commit para esta funcionalidade dá um exemplo simples de como passar todo o seu código fonte C através do programa `indent` antes de fazer o commit. Você pode configurá-lo, definindo o atributo de filtro no arquivo `.gitattributes` para filtrar arquivos `*.c` com o filtro “indent”:

```
1 *.c      filter=indent
```

Então, diga ao Git o que o filtro “indent” faz em `smudge` e `clean`:

```
1 $ git config --global filter.indent.clean indent
2 $ git config --global filter.indent.smudge cat
```

Neste caso, quando você commitar os arquivos que correspondem a `*.c`, Git irá passá-los através do programa `indent` antes de commitá-los e depois passá-los através do programa `cat` antes de fazer o checkout de volta para o disco. O programa `cat` é basicamente um no-op: ele mostra os mesmos dados que ele recebe. Esta combinação efetivamente filtra todos os arquivos de código fonte C através do `indent` antes de fazer o commit.

Outro exemplo interessante é a expansão da palavra-chave `$Date$`, estilo RCS. Para fazer isso corretamente, você precisa de um pequeno script que recebe um nome de arquivo, descobre a última data de commit deste projeto, e insere a data no arquivo. Aqui há um pequeno script Ruby que faz isso:

```

1  #!/usr/bin/env ruby
2  data = STDIN.read
3  last_date = `git log --pretty=format:"%ad" -1`
4  puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')

```

Tudo o que o script faz é obter a última data de commit do comando `git log`, coloca ele em qualquer string `$Date$` que vê no stdin, e imprime os resultados — deve ser simples de fazer em qualquer linguagem que você esteja confortável. Você pode nomear este arquivo `expand_date` e colocá-lo em seu path. Agora, você precisa configurar um filtro no Git (chamaremos de `dater`) e diremos para usar o seu filtro `expand_date` para o smudge dos arquivos no checkout. Você vai usar uma expressão Perl para o clean no commit:

```

1  $ git config filter.dater.smudge expand_date
2  $ git config filter.dater.clean 'perl -pe "s/\\\\$Date[^\$]*\\\\$/\\\\$Date\\\\$/'"

```

Este trecho Perl retira qualquer coisa que vê em uma string `$Date$`, para voltar para onde você começou. Agora que o seu filtro está pronto, você pode testá-lo através da criação de um arquivo com a sua palavra-chave `$Date$` e então criar um atributo Git para esse arquivo que envolve o novo filtro:

```

1  $ echo '# $Date$' > date_test.txt
2  $ echo 'date*.txt filter=dater' >> .gitattributes

```

Se você fizer o commit dessas alterações e fizer o checkout do arquivo novamente, você verá a palavra-chave corretamente substituída:

```

1  $ git add date_test.txt .gitattributes
2  $ git commit -m "Testing date expansion in Git"
3  $ rm date_test.txt
4  $ git checkout date_test.txt
5  $ cat date_test.txt
6  # $Date: Tue Apr 21 07:26:52 2009 -0700$

```

Você pode ver o quão poderosa esta técnica pode ser para aplicações customizadas. Você tem que ter cuidado, porém, porque o arquivo `.gitattributes` está sendo commitado e mantido no projeto, mas o `dater` não é; assim, ele não vai funcionar em todos os lugares. Ao projetar esses filtros, eles devem ser capazes de falhar e ainda assim manter o projeto funcionando corretamente.

Exportando Seu Repositório

Dados de atributo Git também permitem que você faça algumas coisas interessantes ao exportar um arquivo do seu projeto.

export-ignore

Você pode dizer para o Git não exportar determinados arquivos ou diretórios ao gerar um arquivo. Se existe uma subpasta ou arquivo que você não deseja incluir em seu arquivo, mas que você quer dentro de seu projeto, você pode determinar estes arquivos através do atributo `export-ignore`.

Por exemplo, digamos que você tenha alguns arquivos de teste em um subdiretório `test/`, e não faz sentido incluí-los na exportação do tarball do seu projeto. Você pode adicionar a seguinte linha ao seu arquivo de atributos Git:

```
1 test/ export-ignore
```

Agora, quando você executar `git archive` para criar um arquivo tar do seu projeto, aquele diretório não será incluído no arquivo.

export-subst

Outra coisa que você pode fazer para seus arquivos é uma simples substituição de palavra. Git permite colocar a string `$Format:$` em qualquer arquivo com qualquer um dos códigos de formatação `--pretty=format`, muitos dos quais você viu no Capítulo 2. Por exemplo, se você quiser incluir um arquivo chamado `LAST_COMMIT` em seu projeto, e a última data de commit foi injetada automaticamente quando `git archive` foi executado, você pode configurar o arquivo como este:

```
1 $ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
2 $ echo "LAST_COMMIT export-subst" >> .gitattributes
3 $ git add LAST_COMMIT .gitattributes
4 $ git commit -am 'adding LAST_COMMIT file for archives'
```

Quando você executar `git archive`, o conteúdo do arquivo quando aberto será parecido com este:

```
1 $ cat LAST_COMMIT
2 Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

Estratégias de Merge

Você também pode usar atributos Git para dizer ao Git para utilizar estratégias diferentes para mesclar arquivos específicos em seu projeto. Uma opção muito útil é dizer ao Git para não tentar mesclar arquivos específicos quando eles têm conflitos, mas sim para usar o seu lado do merge ao invés do da outra pessoa.

Isso é útil se um branch em seu projeto divergiu ou é especializado, mas você quer ser capaz de fazer o merge de alterações de volta a partir dele, e você deseja ignorar determinados arquivos. Digamos que você tenha um arquivo de configurações de banco de dados chamado `database.xml` que é diferente em dois branches, e você deseja mesclar em seu outro branch sem bagunçar o arquivo de banco de dados. Você pode configurar um atributo como este:

```
1 database.xml merge=ours
```

Se você fizer o merge em outro branch, em vez de ter conflitos de merge com o arquivo `database.xml`, você verá algo como isto:

```
1 $ git merge topic
2 Auto-merging database.xml
3 Merge made by recursive.
```

Neste caso, `database.xml` fica em qualquer versão que você tinha originalmente.

7.3 Customizando o Git - Hooks do Git

Hooks do Git

Como muitos outros sistemas de controle de versão, Git tem uma maneira para disparar scripts personalizados quando certas ações importantes ocorrerem. Existem dois grupos desses hooks: lado cliente e lado servidor. Os hooks do lado cliente são para operações do cliente, tais como commit e merge. Os hooks do lado servidor são para operações de servidor, como recebimento de um push. Você pode usar estes hooks para todo tipo de coisa, e você vai aprender sobre alguns deles aqui.

Instalando um Hook

Os hooks são todos armazenados no subdiretório `hooks` do diretório Git. Na maioria dos projetos, é em `.git/hooks`. Por padrão, o Git preenche este diretório com um monte de scripts de exemplo, muitos dos quais são úteis por si só, mas eles também documentam os valores de entrada de cada script. Todos os exemplos são escritos como shell scripts, com um pouco de Perl, mas todos os scripts executáveis devidamente nomeados irão funcionar bem — você pode escrevê-los em Ruby ou Python ou em que você quiser. Para as versões do Git superiores a 1.6, esses hooks de exemplo terminam com `.sample`; você precisa renomeá-los. Para versões anteriores a 1.6 do Git, os arquivos de exemplo são nomeados corretamente, mas não são executáveis.

Para ativar um script de hook, coloque um arquivo no subdiretório `hooks` do seu diretório Git que é nomeado de forma adequada e é executável. A partir desse ponto, ele deve ser chamado. Eu vou cobrir a maior parte dos nomes dos arquivos de hook importantes aqui.

Hooks do Lado Cliente

Há um monte de hooks do lado do cliente. Esta seção divide eles em committing-workflow hooks, e-mail-workflow scripts, e o resto dos scripts do lado cliente.

Committing-Workflow Hooks Os primeiros quatro hooks têm a ver com o processo de commit. O hook `pre-commit` é executado primeiro, antes mesmo de digitar uma mensagem de confirmação. É usado para inspecionar o snapshot que está prestes a ser commitado, para ver se você se esqueceu de alguma coisa, para ter certeza que os testes rodem, ou para analisar o que você precisa inspecionar no código. Retornando um valor diferente de zero a partir deste hook aborta o commit, mas você pode ignorá-lo com `git commit --no-verify`. Você pode fazer coisas como checar o estilo do código (executar lint ou algo equivalente), verificar o espaço em branco (o hook padrão faz exatamente isso), ou verificar a documentação apropriada sobre novos métodos.

O hook `prepare-commit-msg` é executado antes que o editor de mensagem de commit seja iniciado, mas depois que a mensagem padrão seja criada. Ele permite que você edite a mensagem padrão antes que autor do commit a veja. Este hook tem algumas opções: o caminho para o arquivo que contém a mensagem de confirmação até agora, o tipo de commit, e o SHA-1 do commit se este é um commit amended. Este hook geralmente não é útil para o commit normal, mas sim, para commits onde a mensagem padrão é gerada automaticamente, tal como um template de mensagem de commit, commits de merge, squashed commits, e amended commits. Você pode usá-lo em conjunto com um modelo de commit para inserir informações programaticamente.

O hook `commit-msg` tem um parâmetro, que novamente, é o caminho para um arquivo temporário que contém a mensagem atual de commit. Se este script não retornar zero, Git aborta o processo de commit, de modo que você pode usá-lo para validar o seu estado de projeto ou mensagem de commit antes de permitir que um commit prossiga. Na última seção deste capítulo, vou demonstrar usando este hook como verificar se a sua mensagem de commit está em conformidade com um padrão desejado.

Depois que todo o processo de commit esteja concluído, o hook `post-commit` é executado. Ele não recebe nenhum parâmetro, mas você pode facilmente obter o último commit executando `git log -1 HEAD`. Geralmente, esse script é usado para notificação ou algo similar.

Os scripts committing-workflow do lado cliente podem ser usados em praticamente qualquer fluxo de trabalho. Eles são muitas vezes utilizados para reforçar certas políticas, embora seja importante notar que estes scripts não são transferidos durante um clone. Você pode aplicar a política do lado servidor para rejeitar um push de um commit que não corresponda a alguma política, mas é inteiramente de responsabilidade do desenvolvedor usar esses scripts no lado cliente. Portanto, estes são scripts para ajudar os desenvolvedores, e eles devem ser criados e mantidos por eles, embora eles possam ser substituídos ou modificados por eles a qualquer momento.

E-mail Workflow Hooks

Você pode configurar três hooks do lado cliente para um fluxo de trabalho baseado em e-mail. Eles são todos invocados pelo comando `git am`, por isso, se você não está usando este comando em seu

fluxo de trabalho, você pode pular para a próxima seção. Se você estiver recebendo patches por e-mail preparados por `git format-patch`, então alguns deles podem ser úteis para você.

O primeiro hook que é executado é `applypatch_msg`. Ele recebe um único argumento: o nome do arquivo temporário que contém a mensagem de commit. Git aborta o patch se este script retornar valor diferente de zero. Você pode usar isso para se certificar de que uma mensagem de commit está formatada corretamente ou para normalizar a mensagem através do script.

O próximo hook a ser executado durante a aplicação de patches via `git am` é `pre-applypatch`. Ele não tem argumentos e é executado após a aplicação do patch, então, você pode usá-lo para inspecionar o snapshot antes de fazer o commit. Você pode executar testes ou inspecionar a árvore de trabalho com esse script. Se algo estiver faltando ou os testes não passarem, retornando um valor diferente de zero também aborta o script `git am` sem `commit` o patch.

O último hook a ser executado durante um `git am` é `post-applypatch`. Você pode usá-lo para notificar um grupo ou o autor do patch que você aplicou em relação ao que você fez. Você não pode parar o processo de patch com esse script.

Outros Hooks de Cliente

O hook `pre-rebase` é executado antes de um rebase e pode interromper o processo terminando com valor diferente de zero. Você pode usar esse hook para não permitir rebasing de commits que já foram atualizados com um push. O hook `pre-rebase` de exemplo que o Git instala faz isso, embora ele assuma que o próximo é o nome do branch que você publicará. É provável que você precise mudar isso para seu branch estável ou publicado.

Depois de executar um `git checkout` com sucesso, o hook `post-checkout` é executado, você pode usá-lo para configurar o diretório de trabalho adequadamente para o seu ambiente de projeto. Isso pode significar mover arquivos binários grandes que você não quer controlar a versão, documentação auto-gerada, ou algo parecido.

Finalmente, o hook `post-merge` roda depois de um merge executado com sucesso. Você pode usá-lo para restaurar dados na árvore de trabalho que o GIT não pode rastrear, como dados de permissões. Este hook pode igualmente validar a presença de arquivos externos ao controle do Git que você pode querer copiado quando a árvore de trabalho mudar.

Hooks do Lado Servidor

Além dos Hooks do lado do cliente, você pode usar alguns hooks importantes do lado servidor como administrador do sistema para aplicar quase qualquer tipo de política para o seu projeto. Esses scripts são executados antes e depois um push para o servidor. Os “pre hooks” podem retornar valor diferente de zero em qualquer momento para rejeitar um push, assim como imprimir uma mensagem de erro para o cliente, você pode configurar uma política de push tão complexa quanto você queira.

pre-receive e post-receive

O primeiro script a ser executado ao tratar um push de um cliente é o `pre-receive`. É preciso uma lista de referências que estão no push a partir do `stdin`; se ele não retornar zero, nenhum deles

são aceitos. Você pode usar esse hook para fazer coisas como verificar se nenhuma das referências atualizadas não são fast-forwards; ou para verificar se o usuário que está fazendo o push tem acesso para criar, apagar, ou fazer push de atualizações para todos os arquivos que ele está modificando com o push.

O hook `post-receive` roda depois que todo o processo esteja concluído e pode ser usado para atualizar outros serviços ou notificar os usuários. Ele recebe os mesmos dados do `stdin` que o hook `pre-receive`. Exemplos incluem envio de e-mails, notificar um servidor de integração contínua, ou atualização de um sistema de ticket-tracking — você pode até analisar as mensagens de confirmação para ver se algum ticket precisa ser aberto, modificado ou fechado. Este script não pode parar o processo de push, mas o cliente não se desconecta até que tenha concluído; por isso, tenha cuidado quando você tentar fazer algo que possa levar muito tempo.

update

O script `update` é muito semelhante ao script `pre-receive`, exceto que ele é executado uma vez para cada branch que o usuário está tentando atualizar. Se o usuário está tentando fazer um push para vários branches, `pre-receive` é executado apenas uma vez, enquanto que `update` é executado uma vez por branch do push. Em vez de ler do `stdin`, este script recebe três argumentos: o nome da referência (branch), o SHA-1, que apontava para a referência antes do push, e o SHA-1 do push que o usuário está tentando fazer. Se o script `update` retornar um valor diferente de zero, apenas a referência é rejeitada; outras referências ainda podem ser atualizadas.

7.4 Customizando o Git - Um exemplo de Política Git Forçada

Um exemplo de Política Git Forçada

Nesta seção, você vai usar o que aprendeu para estabelecer um fluxo de trabalho Git que verifica um formato de mensagem personalizado para commit, e força o uso apenas de push fast-forward, e permite que apenas alguns usuários possam modificar determinados subdiretórios em um projeto. Você vai construir scripts cliente que ajudam ao desenvolvedor saber se seu push será rejeitado e scripts de servidor que fazem valer as políticas.

Eu usei Ruby para escrever estes, isso porque é a minha linguagem de script preferida e porque eu sinto que é a linguagem de script que mais parece com pseudocódigo; assim você deve ser capaz de seguir o código, mesmo que você não use Ruby. No entanto, qualquer linguagem funcionará bem. Todos os exemplos de scripts de hooks distribuídos com o Git são feitos em Perl ou Bash, então você também pode ver vários exemplos de hooks nessas linguagens olhando os exemplos.

Hook do Lado Servidor

Todo o trabalho do lado servidor irá para o arquivo `update` no seu diretório de hooks. O arquivo `update` é executado uma vez por branch de cada push e leva a referência do push para a revisão

antiga onde o branch estava, e a nova revisão do push. Você também terá acesso ao usuário que está realizando o push, se o push está sendo executado através de SSH. Se você permitiu que todos se conectem com um único usuário (como “git”), através de autenticação de chave pública, você pode ter que dar ao usuário um “shell wrapper” que determina qual usuário está se conectando com base na chave pública, e definir uma variável de ambiente especificando o usuário. Aqui eu assumo que o usuário de conexão está na variável de ambiente \$USER, então, seu script de atualização começa reunindo todas as informações que você precisa:

```

1  #!/usr/bin/env ruby
2
3  $refname = ARGV[0]
4  $oldrev  = ARGV[1]
5  $newrev  = ARGV[2]
6  $user    = ENV['USER']
7
8  puts "Enforcing Policies... \n(*{$refname}) (*{$oldrev[0,6]}) (*{$newrev[0,6]})"
```

Sim, eu estou usando variáveis globais. Não me julgue — é mais fácil para demonstrar desta maneira.

Impondo um Formato Específico de Mensagens de Commit

Seu primeiro desafio é impor que cada mensagem de commit deve aderir a um formato específico. Só para se ter uma meta, vamos supor que cada mensagem tem de incluir uma string que parece com “ref: 1234” porque você quer que cada commit tenha um link para um item de trabalho no seu sistema de chamados. Você deve olhar para cada commit do push, ver se essa sequência está na mensagem de commit, e, se a string estiver ausente de qualquer um dos commits, retornar zero para que o push seja rejeitado.

Você pode obter uma lista dos valores SHA-1 de todos os commits de um push, através dos valores \$newrev e \$oldrev e passando-os para um comando Git plumbing chamado `git rev-list`. Este é basicamente o comando `git log`, mas por padrão ele mostra apenas os valores SHA-1 e nenhuma outra informação. Assim, para obter uma lista de todos os SHAs de commits introduzidos entre um commit SHA e outro, você pode executar algo como abaixo:

```

1  $ git rev-list 538c33..d14fc7
2  d14fc7c847ab946ec39590d87783c69b031bdfb7
3  9f585da4401b0a3999e84113824d15245c13f0be
4  234071a1be950e2a8d078e6141f5cd20c1e61ad3
5  dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
6  17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Você pode pegar essa saída, percorrer cada um dos SHAs dos commits, pegar a mensagem para ele, e testar a mensagem contra uma expressão regular que procura um padrão.

Você tem que descobrir como pegar a mensagem de confirmação de cada um dos commits para testar. Para obter os dados brutos do commit, você pode usar um outro comando plumbing chamado `git cat-file`. Eu vou falar de todos estes comandos plumbing em detalhes no Capítulo 9; mas, por agora, aqui está o resultado do comando:

```
1 $ git cat-file commit ca82a6
2 tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
3 parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
4 author Scott Chacon <schacon@gmail.com> 1205815931 -0700
5 committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
6
7 changed the verison number
```

Uma maneira simples de obter a mensagem de confirmação de um commit quando você tem o valor SHA-1 é ir para a primeira linha em branco e retirar tudo depois dela. Você pode fazer isso com o comando `sed` em sistemas Unix:

```
1 $ git cat-file commit ca82a6 | sed '1,/^\$/d'
2 changed the verison number
```

Você pode usar isso para pegar a mensagem de cada commit do push e sair se você ver algo que não corresponde. Para sair do script e rejeitar o push, retorne um valor diferente de zero. Todo o método se parece com este:

```
1 $regex = /\[ref: (\d+)\]/
2
3 # enforced custom commit message format
4 def check_message_format
5   missed_revs = `git rev-list #{oldrev}..#{newrev}`.split("\n")
6   missed_revs.each do |rev|
7     message = `git cat-file commit #{rev} | sed '1,/^\$/d'`
8     if !$regex.match(message)
9       puts "[POLICY] Your message is not formatted correctly"
10      exit 1
11    end
12  end
13 end
14 check_message_format
```

Colocar isso no seu script `update` rejeitará atualizações que contenham commits que tem mensagens que não aderem à sua regra.

Impondo um Sistema ACL Baseado em Usuário

Suponha que você queira adicionar um mecanismo que utiliza uma lista de controle de acesso (ACL) que especifica quais usuários têm permissão para fazer push com mudanças para partes de seus projetos. Algumas pessoas têm acesso total, e outras só têm acesso a alterar determinados subdiretórios ou arquivos específicos. Para impor isso, você vai escrever essas regras em um arquivo chamado `acl` que ficará em seu repositório Git no servidor. O hook `update` verificará essas regras, verá quais arquivos estão sendo introduzidos nos commits do push, e determinará se o usuário que está fazendo o push tem acesso para atualizar todos os arquivos.

A primeira coisa que você deve fazer é escrever o seu ACL. Aqui você vai usar um formato muito parecido com o mecanismo de ACL CVS: ele usa uma série de linhas, onde o primeiro campo é `avail` ou `unavail`, o próximo campo é uma lista delimitada por vírgula dos usuários para que a regra se aplica, e o último campo é o caminho para o qual a regra se aplica (branco significando acesso em aberto). Todos esses campos são delimitados por um caractere pipe (`|`).

Neste caso, você tem alguns administradores, alguns escritores de documentação com acesso ao diretório `doc`, e um desenvolvedor que só tem acesso aos diretórios `lib` e `tests`, seu arquivo ACL fica assim:

```
1 avail|nickh,pjhyett,defunkt,tpw
2 avail|usinclair,cdickens,ebronte|doc
3 avail|schacon|lib
4 avail|schacon|tests
```

Você começa lendo esses dados em uma estrutura que você pode usar. Neste caso, para manter o exemplo simples, você só vai cumprir as diretrizes do `avail`. Aqui está um método que lhe dá um array associativo onde a chave é o nome do usuário e o valor é um conjunto de paths que o usuário tem acesso de escrita:

```
1 def get_acl_access_data(acl_file)
2   # read in ACL data
3   acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
4   access = {}
5   acl_file.each do |line|
6     avail, users, path = line.split('|')
7     next unless avail == 'avail'
8     users.split(',').each do |user|
9       access[user] ||= []
10      access[user] << path
11    end
12  end
13  access
14 end
```

No arquivo ACL que você viu anteriormente, o método `get_acl_access_data` retorna uma estrutura de dados que se parece com esta:

```
1 { "defunkt" => [nil],
2   "tpw" => [nil],
3   "nickh" => [nil],
4   "pjhyett" => [nil],
5   "schacon" => ["lib", "tests"],
6   "cdickens" => ["doc"],
7   "usinclair" => ["doc"],
8   "ebronte" => ["doc"] }
```

Agora que você tem as permissões organizadas, é preciso determinar quais os paths que os commits do push modificam, de modo que você pode ter certeza que o usuário que está fazendo o push tem acesso a todos eles.

Você pode muito facilmente ver quais arquivos foram modificados em um único commit com a opção `--name-only` do comando `git log` (mencionado brevemente no Capítulo 2):

```
1 $ git log -1 --name-only --pretty=format:'' 9f585d
2
3 README
4 lib/test.rb
```

Se você usar a estrutura ACL retornada pelo método `get_acl_access_data` e verificar a relação dos arquivos listados em cada um dos commits, você pode determinar se o usuário tem acesso ao push de todos os seus commits:

```
1 # only allows certain users to modify certain subdirectories in a project
2 def check_directory_perms
3   access = get_acl_access_data('acl')
4
5
6   # see if anyone is trying to push something they can't
7   new_commits = `git rev-list #{oldrev}..#{newrev}`.split("\n")
8   new_commits.each do |rev|
9     files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n\
10 ")
11     files_modified.each do |path|
12       next if path.size == 0
13       has_file_access = false
14       access[$user].each do |access_path|
```

```
15         if !access_path || # user has access to everything
16             (path.index(access_path) == 0) # access to this path
17             has_file_access = true
18         end
19     end
20     if !has_file_access
21         puts "[POLICY] You do not have access to push to #{path}"
22         exit 1
23     end
24 end
25 end
26 end
27
28 check_directory_perms
```

A maior parte do código deve ser fácil de acompanhar. Você receberá uma lista de novos commits do push com `git rev-list`. Então, para cada um desses, você acha quais arquivos são modificados e verifica se o usuário que está fazendo o push tem acesso a todos os paths sendo modificados. Um Rubyism que pode não ser claro é `path.index(access_path) == 0`, que é verdadeiro se o caminho começa com `access_path` — isso garante que `access_path` não está apenas em um dos caminhos permitidos, mas um path permitido começa com cada path acessado.

Agora seus usuários não podem fazer o push de qualquer commit com mensagens mal formadas ou com arquivos modificados fora de seus paths designados.

Impondo Apenas Fast-Forward Pushes

A única coisa que resta é impor apenas fast-forward pushes. Nas versões Git 1.6 ou mais recentes, você pode definir as configurações `receive.denyDeletes` e `receive.denyNonFastForwards`. Mas utilizar um hook irá funcionar em versões mais antigas do Git, e você pode modificá-lo para impor a diretiva apenas para determinados usuários ou fazer qualquer outra coisa que você queira.

A lógica para verificar isso é ver se algum commit é acessível a partir da revisão mais antiga que não é acessível a partir da versão mais recente. Se não houver nenhum, então foi um push Fast-Forward; caso contrário, você nega ele:


```

1  # enforces fast-forward only pushes
2  def check_fast_forward
3      missed_refs = `git rev-list #{newrev}..#{oldrev}`
4      missed_ref_count = missed_refs.split("\n").size
5      if missed_ref_count > 0
6          puts "[POLICY] Cannot push a non fast-forward reference"
7          exit 1
8      end
9  end
10
11 check_fast_forward

```

Tudo está configurado. Se você executar `chmod u+x .git/hooks/update`, que é o arquivo no qual você deve ter colocado todo este código, e então tentar fazer um push de uma referência não fast-forwarded, você verá algo como isto:

```

1  $ git push -f origin master
2  Counting objects: 5, done.
3  Compressing objects: 100% (3/3), done.
4  Writing objects: 100% (3/3), 323 bytes, done.
5  Total 3 (delta 1), reused 0 (delta 0)
6  Unpacking objects: 100% (3/3), done.
7  Enforcing Policies...
8  (refs/heads/master) (8338c5) (c5b616)
9  [POLICY] Cannot push a non fast-forward reference
10 error: hooks/update exited with error code 1
11 error: hook declined to update refs/heads/master
12 To git@gitserver:project.git
13 ! [remote rejected] master -> master (hook declined)
14 error: failed to push some refs to 'git@gitserver:project.git'

```

Há algumas coisas interessantes aqui. Primeiro, você vê quando o hook começa a funcionar.

```

1  Enforcing Policies...
2  (refs/heads/master) (8338c5) (c5b616)

```

Observe que você imprimiu aquilo no stdout no início do seu script de atualização. É importante notar que qualquer coisa que seu script imprima no stdout será transferido para o cliente.

A próxima coisa que você vai notar é a mensagem de erro.

```
1 [POLICY] Cannot push a non fast-forward reference
2 error: hooks/update exited with error code 1
3 error: hook declined to update refs/heads/master
```

A primeira linha foi impressa por você, as outras duas foram pelo Git dizendo que o script de atualização não retornou zero e é isso que está impedindo seu push. Por último, você verá isso:

```
1 To git@gitserver:project.git
2 ! [remote rejected] master -> master (hook declined)
3 error: failed to push some refs to 'git@gitserver:project.git'
```

Você verá uma mensagem de rejeição remota para cada referência que o seu hook impediu, e ele diz que ele foi recusado especificamente por causa de uma falha de hook.

Além disso, se o marcador ref não existir em nenhum dos seus commits, você verá a mensagem de erro que você está imprimindo para ele.

```
1 [POLICY] Your message is not formatted correctly
```

Ou se alguém tentar editar um arquivo que não têm acesso e fazer um push de um commit que o contém, ele verá algo semelhante. Por exemplo, se um autor de documentação tenta fazer um push de um commit modificando algo no diretório `lib`, ele verá

```
1 [POLICY] You do not have access to push to lib/test.rb
```

Isto é tudo. A partir de agora, desde que o script `update` esteja lá e seja executável, seu repositório nunca será rebobinado e nunca terá uma mensagem de commit sem o seu padrão nela, e os usuários terão restrições.

Hooks do Lado Cliente

A desvantagem desta abordagem é a choringa que resultará inevitavelmente quando os pushes de commits de seus usuários forem rejeitados. Tendo seu trabalho cuidadosamente elaborada rejeitado no último minuto pode ser extremamente frustrante e confuso; e, além disso, eles vão ter que editar seu histórico para corrigi-lo, o que nem sempre é para os fracos de coração.

A resposta para este dilema é fornecer alguns hooks do lado cliente que os usuários possam usar para notificá-los quando eles estão fazendo algo que o servidor provavelmente rejeitará. Dessa forma, eles podem corrigir quaisquer problemas antes de fazer o commit e antes desses problemas se tornarem mais difíceis de corrigir. Já que hooks não são transferidos com um clone de um projeto, você deve distribuir esses scripts de alguma outra forma e, então, usuários devem copiá-los para seu diretório

.git/hooks e torná-los executáveis. Você pode distribuir esses hooks dentro do projeto ou em um projeto separado, mas não há maneiras de configurá-los automaticamente.

Para começar, você deve verificar a sua mensagem de confirmação antes que cada commit seja gravado, então você saberá que o servidor não irá rejeitar as alterações devido a mensagens de commit mal formatadas. Para fazer isso, você pode adicionar o hook `commit-msg`. Se fizer ele ler as mensagens do arquivo passado como o primeiro argumento e comparar ele com o padrão, você pode forçar o Git a abortar o commit se eles não corresponderem:

```
1  #!/usr/bin/env ruby
2  message_file = ARGV[0]
3  message = File.read(message_file)
4
5  $regex = /\[ref: (\d+)\]/
6
7  if !$regex.match(message)
8    puts "[POLICY] Your message is not formatted correctly"
9    exit 1
10 end
```

Se esse script está no lugar correto (em `.git/hooks/commit-msg`) e é executável, e você fizer um commit com uma mensagem que não está formatada corretamente, você verá isso:

```
1  $ git commit -am 'test'
2  [POLICY] Your message is not formatted correctly
```

Nenhum commit foi concluído nessa instância. No entanto, se a mensagem conter o padrão adequado, o Git permite o commit:

```
1  $ git commit -am 'test [ref: 132]'
2  [master e05c914] test [ref: 132]
3  1 files changed, 1 insertions(+), 0 deletions(-)
```

Em seguida, você quer ter certeza de que você não está modificando os arquivos que estão fora do seu escopo ACL. Se o seu diretório de projeto `.git` contém uma cópia do arquivo ACL que você usou anteriormente, então o seguinte script pre-commit irá impor essas restrições para você:

```

1  #!/usr/bin/env ruby
2
3  $user = ENV['USER']
4
5  # [ insert acl_access_data method from above ]
6
7  # only allows certain users to modify certain subdirectories in a project
8  def check_directory_perms
9      access = get_acl_access_data('.git/acl')
10
11     files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
12     files_modified.each do |path|
13         next if path.size == 0
14         has_file_access = false
15         access[$user].each do |access_path|
16             if !access_path || (path.index(access_path) == 0)
17                 has_file_access = true
18             end
19             if !has_file_access
20                 puts "[POLICY] You do not have access to push to #{path}"
21                 exit 1
22             end
23         end
24     end
25
26     check_directory_perms

```

Este é aproximadamente o mesmo script da parte do lado servidor, mas com duas diferenças importantes. Primeiro, o arquivo ACL está em um lugar diferente, porque este script é executado a partir do seu diretório de trabalho, e não de seu diretório Git. Você tem que mudar o path para o arquivo ACL, disso

```

1  access = get_acl_access_data('acl')

```

para isso:

```

1  access = get_acl_access_data('.git/acl')

```

A outra diferença importante é a forma como você obtém uma lista dos arquivos que foram alterados. Como o método do lado servidor olha no log de commits e, neste momento, o commit não foi gravado ainda, você deve pegar sua lista de arquivos da área staging. Em vez de

```
1 files_modified = `git log -1 --name-only --pretty=format:' ' #{ref}`
```

you should use

```
1 files_modified = `git diff-index --cached --name-only HEAD`
```

Mas essas são as duas únicas diferenças — caso contrário, o script funciona da mesma maneira. Uma ressalva é que ele espera que você esteja executando localmente como o mesmo usuário que você fez o push para a máquina remota. Se ele for diferente, você deve definir a variável `$user` manualmente.

A última coisa que você tem que fazer é verificar se você não está tentando fazer o push de referências não fast-forwarded, mas isso é um pouco menos comum. Para obter uma referência que não é um fast-forward, você tem que fazer um rebase depois de um commit que já foi enviado por um push ou tentar fazer o push de um branch local diferente até o mesmo branch remoto.

Como o servidor vai dizer que você não pode fazer um push não fast-forward de qualquer maneira, e o hook impede pushes forçados, a única coisa acidental que você pode tentar deter são commits de rebase que já foram enviados por um push.

Aqui está um exemplo de script pré-rebase que verifica isso. Ele recebe uma lista de todos os commits que você está prestes a reescrever e verifica se eles existem em qualquer uma das suas referências remotas. Se ele vê um que é acessível a partir de uma de suas referências remotas, ele aborta o rebase:

```
1  #!/usr/bin/env ruby
2
3  base_branch = ARGV[0]
4  if ARGV[1]
5    topic_branch = ARGV[1]
6  else
7    topic_branch = "HEAD"
8  end
9
10 target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
11 remote_refs = `git branch -r`.split("\n").map { |r| r.strip }
12
13 target_shas.each do |sha|
14   remote_refs.each do |remote_ref|
15     shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
16     if shas_pushed.split("\n").include?(sha)
17       puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
18       exit 1
19     end
20   end
21 end
```

Este script utiliza uma sintaxe que não foi coberta na seção Seleção de Revisão do Capítulo 6. Você obterá uma lista de commits que já foram enviados em um push executando isto:

```
1 git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

A sintaxe `SHA^@` resolve para todos os pais daquele commit. Você está à procura de qualquer commit que é acessível a partir do último commit no remoto e que não é acessível a partir de qualquer pai de qualquer um dos SHAs que você está tentando fazer o push — o que significa que é um fast-forward.

A principal desvantagem desta abordagem é que ela pode ser muito lenta e muitas vezes é desnecessária — se você não tentar forçar o push com `-f`, o servidor irá avisá-lo e não aceitará o push. No entanto, é um exercício interessante e pode, em teoria, ajudar a evitar um rebase que você possa mais tarde ter que voltar atrás e corrigir.

7.5 Customizando o Git - Sumário

Sumário

Você viu a maior parte das principais formas que você pode usar para personalizar o seu cliente e servidor Git para melhor atender a seu fluxo de trabalho e projetos. Você aprendeu sobre todos os tipos de configurações, atributos baseados em arquivos, e hooks de eventos, e você construiu um exemplo de política aplicada ao servidor. Agora você deve ser capaz de usar o Git em quase qualquer fluxo de trabalho que você possa sonhar.

Capítulo 8 - Git e Outros Sistemas

O mundo não é perfeito. Normalmente, você não pode migrar cada projeto que você tem para o Git. Às vezes, você está preso em um projeto usando outro VCS, e geralmente ele é o Subversion. Você vai passar a primeira parte deste capítulo aprendendo sobre `git svn`, a ferramenta de gateway bidirecional entre Subversion e Git.

Em algum momento, você pode querer converter seu projeto existente para o Git. A segunda parte deste capítulo aborda como migrar seu projeto para o Git: primeiro do Subversion, depois a partir do Perforce e, finalmente, através de um script de importação customizado para um caso atípico de importação.

8.1 Git e Outros Sistemas - Git e Subversion

Git e Subversion

Atualmente, a maioria dos projetos de desenvolvimento de código aberto e um grande número de projetos corporativos usam o Subversion para gerenciar seu código fonte. É o VCS open source mais popular e tem sido assim por quase uma década. É também muito similar em muitos aspectos ao CVS, que foi muito utilizado antes disso.

Uma das grandes características do Git é uma ponte bidirecional para Subversion chamada `git svn`. Esta ferramenta permite que você use Git como um cliente válido para um servidor Subversion, então você pode usar todos os recursos locais do Git e fazer um push para um servidor Subversion, como se estivesse usando o Subversion localmente. Isto significa que você pode fazer ramificação (branching) local e fusão (merge), usar a área de teste (staging area), cherry-picking, e assim por diante, enquanto os seus colaboradores continuam a trabalhar em seus caminhos escuros e antigos. É uma boa maneira de utilizar o Git em ambiente corporativo e ajudar os seus colegas desenvolvedores a se tornarem mais eficientes enquanto você luta para obter a infra-estrutura para suportar Git completamente.

git svn

O comando base no Git para todos os comandos de ponte do Subversion é `git svn`. Você inicia tudo com isso. São precisos alguns comandos, para que você aprenda sobre os mais comuns, indo através de um fluxo de trabalho pequeno.

É importante notar que quando você está usando `git svn`, você está interagindo com o Subversion, que é um sistema muito menos sofisticado do que Git. Embora você possa facilmente fazer ramificação local e fusão, é geralmente melhor manter seu histórico tão linear quanto possível

usando rebasing no seu trabalho e evitar fazer coisas como, simultaneamente, interagir com um repositório Git remoto.

Não reescreva seu histórico e tente fazer um push de novo, e não faça um push para um repositório Git paralelo para colaborar com desenvolvedores Git ao mesmo tempo. Subversion pode ter apenas um histórico linear simples, e confundi-lo é muito fácil. Se você está trabalhando com uma equipe, e alguns estão usando SVN e outros estão usando Git, certifique-se que todo mundo está usando o servidor SVN para colaborar — isso vai facilitar a sua vida.

Configurando

Para demonstrar essa funcionalidade, você precisa de um repositório SVN típico que você tenha acesso de gravação. Se você deseja copiar esses exemplos, você vai ter que fazer uma cópia gravável do meu repositório de teste. A fim de fazer isso facilmente, você pode usar uma ferramenta chamada svnsync que vem com as versões mais recentes do Subversion — ele deve ser distribuído a partir da versão 1.4. Para estes testes, eu criei um novo repositório Subversion no Google code que era uma cópia parcial do projeto protobuf, que é uma ferramenta que codifica dados estruturados para transmissão de rede.

Para acompanhar, primeiro você precisa criar um novo repositório Subversion local:

```
1 $ mkdir /tmp/test-svn
2 $ svnadmin create /tmp/test-svn
```

Então, permitir que todos os usuários possam alterar revprops — o caminho mais fácil é adicionar um script pre-revprop-change que sempre retorna 0:

```
1 $ cat /tmp/test-svn/hooks/pre-revprop-change
2 #!/bin/sh
3 exit 0;
4 $ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Agora você pode sincronizar este projeto na sua máquina local chamando svnsync init com os repositórios to e from.

```
1 $ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Isso define as propriedades para executar a sincronização. Você pode, então, clonar o código executando


```

1 $ svnsync sync file:///tmp/test-svn
2 Committed revision 1.
3 Copied properties for revision 1.
4 Committed revision 2.
5 Copied properties for revision 2.
6 Committed revision 3.
7 ...

```

Embora essa operação possa demorar apenas alguns minutos, se você tentar copiar o repositório original para outro repositório remoto em vez de um local, o processo vai demorar quase uma hora, mesmo que haja menos de 100 commits. Subversion tem que clonar uma revisão em um momento e em seguida, fazer um push de volta para outro repositório — é ridiculamente ineficientes, mas é a única maneira fácil de fazer isso.

Primeiros Passos

Agora que você tem um repositório Subversion que você tem acesso de gravação, você pode usar um fluxo de trabalho típico. Você vai começar com o comando `git svn clone`, que importa um repositório Subversion inteiro em um repositório Git local. Lembre-se de que se você está importando de um repositório Subversion hospedado, você deve substituir o `file:///tmp/test-svn` aqui com a URL do seu repositório Subversion:

```

1 $ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
2 Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
3 r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
4     A    m4/acx_pthread.m4
5     A    m4/stl_hash.m4
6 ...
7 r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
8 Found possible branch point: file:///tmp/test-svn/trunk => \
9     file:///tmp/test-svn /branches/my-calc-branch, 75
10 Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
11 Following parent with do_switch
12 Successfully followed parent
13 r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
14 Checked out HEAD:
15     file:///tmp/test-svn/branches/my-calc-branch r76

```

Isso executa o equivalente a dois comandos — `git svn init` seguido por `git svn fetch` — na URL que você fornecer. Isso pode demorar um pouco. O projeto de teste tem apenas cerca de 75 commits e a base de código não é tão grande, por isso leva apenas alguns minutos. No entanto, Git

tem de verificar cada versão, uma de cada vez, e commitá-las individualmente. Para um projeto com centenas ou milhares de commits, isso pode literalmente levar horas ou até dias para terminar.

A parte `-T trunk -b branches -t tags` diz ao Git que este repositório Subversion segue a ramificação (branching) básica e convenções de tag. Se você nomeou seu trunk, branches, ou tags de maneira diferente, você pode alterar estas opções. Já que isso é muito comum, você pode substituir esta parte inteira com `-s`, o que significa layout padrão e implica todas essas opções. O comando a seguir é equivalente:

```
1 $ git svn clone file:///tmp/test-svn -s
```

Neste ponto, você deve ter um repositório Git válido que importou seus branches e tags:

```
1 $ git branch -a
2 * master
3   my-calc-branch
4   tags/2.0.2
5   tags/release-2.0.1
6   tags/release-2.0.2
7   tags/release-2.0.2rc1
8   trunk
```

É importante observar como esta ferramenta nomeia (namespaces) suas referências remotas de forma diferente. Quando você está clonando de um repositório Git normal, você recebe todos os branches que estão disponíveis no servidor remoto localmente, algo como `origin/[branch]` — nomeados com o nome do remoto. No entanto, `git svn` assume que você não vai ter vários remotos e salva todas as suas referências em pontos no servidor remoto sem “namespacing”. Você pode usar o comando `Git plumbing show-ref` para ver os seus nomes de referência completa:

```
1 $ git show-ref
2 1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
3 aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
4 03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
5 50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
6 4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
7 1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
8 1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

Um repositório Git normal se parece com isto:

```

1 $ git show-ref
2 83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3 3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
4 0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
5 25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing

```

Você tem dois servidores remotos: um chamado `gitserver` com um branch `master`, e outro chamado `origin` com dois branches, `master` e `testing`.

Observe como no exemplo de referências remotas importados com `git svn`, tags são adicionadas como branches remotos, não como tags Git reais. Sua importação do Subversion parece que tem um remoto chamado `tags` branches nele.

Commitando de Volta no Subversion

Agora que você tem um repositório de trabalho, você pode fazer algum trabalho no projeto e fazer um push de seus commits de volta ao upstream, usando Git efetivamente como um cliente SVN. Se você editar um dos arquivos e fazer o commit, você tem um commit que existe no Git local que não existe no servidor Subversion:

```

1 $ git commit -am 'Adding git-svn instructions to the README'
2 [master 97031e5] Adding git-svn instructions to the README
3 1 files changed, 1 insertions(+), 1 deletions(-)

```

Em seguida, você precisa fazer o push de suas mudanças ao upstream. Observe como isso muda a maneira de trabalhar com o Subversion — Você pode fazer vários commits offline e depois fazer um push com todos de uma vez para o servidor Subversion. Para fazer um push a um servidor Subversion, você executa o comando `git svn dcommit`:

```

1 $ git svn dcommit
2 Committing to file:///tmp/test-svn/trunk ...
3      M      README.txt
4 Committed r79
5      M      README.txt
6 r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
7 No changes between current HEAD and refs/remotes/trunk
8 Resetting to the latest refs/remotes/trunk

```

Isso leva todos os commits que você fez em cima do código do servidor Subversion, faz um commit Subversion para cada um, e então reescreve seu commit Git local para incluir um identificador único. Isto é importante porque significa que todas as somas de verificação (checksums) SHA-1 dos seus commits mudam. Em parte por esta razão, trabalhar com Git em versões remotas de seus projetos ao mesmo tempo com um servidor Subversion não é uma boa ideia. Se você olhar para o último commit, você pode ver o novo `git-svn-id` que foi adicionado:

```
1 $ git log -1
2 commit 938b1a547c2cc92033b74d32030e86468294a5c8
3 Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
4 Date: Sat May 2 22:06:44 2009 +0000
5
6 Adding git-svn instructions to the README
7
8 git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

Observe que a soma de verificação SHA que inicialmente começou com 97031e5 quando você commitou agora começa com 938b1a5. Se você quer fazer um push para tanto um servidor Git como um servidor Subversion, você tem que fazer um push (dcommit) para o servidor Subversion primeiro, porque essa ação altera os dados de commit.

Fazendo Pull de Novas Mudanças

Se você está trabalhando com outros desenvolvedores, então em algum ponto um de vocês vai fazer um push, e depois o outro vai tentar fazer um push de uma mudança que conflita. Essa mudança será rejeitada até você mesclá-la em seu trabalho. No `git svn`, é parecido com isto:

```
1 $ git svn dcommit
2 Committing to file:///tmp/test-svn/trunk ...
3 Merge conflict during commit: Your file or directory 'README.txt' is probably \
4 out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
5 core/git-svn line 482
```

Para resolver essa situação, você pode executar `git svn rebase`, que puxa quaisquer alterações no servidor que você não tem ainda e faz um rebase de qualquer trabalho que você tem em cima do que está no servidor:

```
1 $ git svn rebase
2      M      README.txt
3 r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
4 First, rewinding head to replay your work on top of it...
5 Applying: first user change
```

Agora, todo o seu trabalho está em cima do que está no servidor Subversion, para que você possa com sucesso usar `dcommit`:

```

1 $ git svn dcommit
2 Committing to file:///tmp/test-svn/trunk ...
3     M      README.txt
4 Committed r81
5     M      README.txt
6 r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
7 No changes between current HEAD and refs/remotes/trunk
8 Resetting to the latest refs/remotes/trunk

```

É importante lembrar que, ao contrário do Git, que exige que você mescle trabalhos do upstream que você ainda não tem localmente antes que você possa fazer um push, `git svn` faz você fazer isso somente se as alterações conflitarem. Se alguém fizer um push de uma alteração em um arquivo e então você fizer um push de uma mudança de outro arquivo, o seu `dcommit` vai funcionar:

```

1 $ git svn dcommit
2 Committing to file:///tmp/test-svn/trunk ...
3     M      configure.ac
4 Committed r84
5     M      autogen.sh
6 r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
7     M      configure.ac
8 r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
9 W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
10 using rebase:
11 :100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
12 015e4c98c482f0fa71e4d5434338014530b37fa6 M autogen.sh
13 First, rewinding head to replay your work on top of it...
14 Nothing to do.

```

É importante lembrar disto, porque o resultado é um estado de projeto que não existia em nenhum dos seus computadores quando você fez o push. Se as alterações forem incompatíveis, mas não entram em conflito, você pode ter problemas que são difíceis de diagnosticar. Isso é diferente de usar um servidor Git — no Git, você pode testar completamente o estado do sistema cliente antes de publicá-lo, enquanto que no SVN, você não pode nunca estar certo de que os estados imediatamente antes e depois do commit são idênticos.

Você também deve executar este comando para fazer o pull das alterações do servidor Subversion, mesmo que você não esteja pronto para commitar. Você pode executar `git svn fetch` para pegar os novos dados, mas `git svn rebase` faz a busca e atualiza seus commits locais.

```

1 $ git svn rebase
2     M      generate_descriptor_proto.sh
3 r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
4 First, rewinding head to replay your work on top of it...
5 Fast-forwarded master to refs/remotes/trunk.

```

Executando `git svn rebase` de vez em quando irá manter seu código sempre atualizado. Você precisa ter certeza de que seu diretório de trabalho está limpo quando você executar isso. Se você tiver alterações locais, você deve guardar o seu trabalho (stash) ou temporariamente fazer o commit dele antes de executar `git svn rebase` — caso contrário, o comando irá parar se ver que o rebase irá resultar em um conflito de mesclagem.

Problemas de Branching no Git

Quando você se sentir confortável com um fluxo de trabalho Git, é provável que você crie branches tópicos, trabalhe neles, e em seguida, faça um merge deles. Se você está fazendo um push para um servidor Subversion via `git svn`, você pode querer fazer o rebase de seu trabalho em um único branch de cada vez, em vez de fundir (merge) branches juntos. A razão para preferir rebasing é que o Subversion tem um histórico linear e não lida com fusões (merges), como Git faz, assim `git svn` segue apenas o primeiro pai ao converter os snapshots em commits Subversion.

Suponha que seu histórico se parece com o seguinte: você criou um branch `experiment`, fez dois commits, e depois fundiu-os de volta em `master`. Quando você executar `dcommit`, você verá uma saída como esta:

```

1 $ git svn dcommit
2 Committing to file:///tmp/test-svn/trunk ...
3     M      CHANGES.txt
4 Committed r85
5     M      CHANGES.txt
6 r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
7 No changes between current HEAD and refs/remotes/trunk
8 Resetting to the latest refs/remotes/trunk
9 COPYING.txt: locally modified
10 INSTALL.txt: locally modified
11     M      COPYING.txt
12     M      INSTALL.txt
13 Committed r86
14     M      INSTALL.txt
15     M      COPYING.txt
16 r86 = 2647f6b86ccfaad4ec58c520e369ec81f7c283c (trunk)
17 No changes between current HEAD and refs/remotes/trunk
18 Resetting to the latest refs/remotes/trunk

```

Executando `dcommit` em um branch com histórico mesclado funcionará bem, exceto que quando você olhar no seu histórico de projeto Git, ele não reescreveu nenhum dos commits que você fez no branch `experiment` — em vez disso, todas essas alterações aparecem na versão SVN do único commit do merge.

Quando alguém clona esse trabalho, tudo o que vêem é o commit do merge com todo o trabalho comprimido nele; eles não veem os dados de commit sobre de onde veio ou quando ele foi commitado.

Branching no Subversion

Ramificação no Subversion não é o mesmo que ramificação no Git; se você puder evitar usá-lo muito, é provavelmente melhor. No entanto, você pode criar e commitar em branches no Subversion usando `svn git`.

Criando um Novo Branch SVN Para criar um novo branch no Subversion, você executa `git svn branch [branchname]`:

```
1 $ git svn branch opera
2 Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera.\
3 ..
4 Found possible branch point: file:///tmp/test-svn/trunk => \
5   file:///tmp/test-svn/branches/opera, 87
6 Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
7 Following parent with do_switch
8 Successfully followed parent
9 r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

Isso faz o equivalente ao comando `svn copy trunk branches/opera` no Subversion e funciona no servidor Subversion. É importante notar que ele não faz um checkout nesse branch; se você commitar neste momento, este commit irá para `trunk` no servidor, em vez de `opera`.

Mudar Branches Ativos

Git descobre para que branch seus `dcommits` irão olhando para a extremidade de qualquer branch Subversion no seu histórico — você deve ter apenas um, e ele deve ser o último com um `git-svn-id` em seu histórico atual de branch.

Se você quiser trabalhar em mais de um branch ao mesmo tempo, você pode criar branches locais para `dcommit` para branches Subversion específicos iniciando-os no commit Subversion importado para esse branch. Se você quiser um branch `opera` em que você possa trabalhar em separado, você pode executar

```
1 $ git branch opera remotes/opera
```

Agora, se você deseja mesclar seu branch `opera` em `trunk` (seu branch `master`), você pode fazer isso com `git merge`. Mas você precisa fornecer uma mensagem descritiva do commit (via `-m`), ou o merge vai dizer “Merge branch `opera`” em vez de algo útil.

Lembre-se que, apesar de você estar usando `git merge` para fazer esta operação, e provavelmente o merge será muito mais fácil do que seria no Subversion (porque Git irá detectar automaticamente a base de mesclagem apropriada para você), este não é um commit `git merge` normal. Você tem que fazer o push desses dados para um servidor Subversion que não pode lidar com um commit que rastreia mais de um pai; por isso, depois de fazer o push dele, ele vai parecer como um único commit que contém todo o trabalho de outro branch em um único commit. Depois que você mesclar um branch em outro, você não pode facilmente voltar e continuar a trabalhar nesse branch, como você normalmente faz no Git. O comando `dcommit` que você executa apaga qualquer informação que diz qual branch foi incorporado, então cálculos merge-base posteriores estarão errados — o `dcommit` faz os resultados do seu `git merge` parecerem que você executou `git merge --squash`. Infelizmente, não há nenhuma boa maneira de evitar esta situação — Subversion não pode armazenar essa informação, assim, você vai ser sempre prejudicado por essas limitações enquanto você estiver usando-o como seu servidor. Para evitar problemas, você deve excluir o branch local (neste caso, `opera`), depois de mesclá-lo em `trunk`.

Comandos do Subversion

O conjunto de ferramentas `git svn` fornece um número de comandos para ajudar a facilitar a transição para o Git, fornecendo uma funcionalidade que é semelhante ao que você tinha no Subversion. Aqui estão alguns comandos parecidos com o Subversion.

Estilo de Histórico do SVN

Se você está acostumado a usar o Subversion e quer ver seu histórico no estilo do SVN, você pode executar `git svn log` para ver o seu histórico de commits na formatação SVN:

```
1 $ git svn log
2 -----
3 r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines
4
5 autogen change
6
7 -----
8 r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines
9
10 Merge branch 'experiment'
11
12 -----
```



```

13 r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines
14
15 updated the changelog

```

Você deve saber duas coisas importantes sobre `git svn log`. Primeiro, ele funciona offline, ao contrário do comando `svn log` verdadeiro, que pede os dados ao servidor Subversion. Segundo, ele só mostra commits que foram commitados ao servidor Subversion. Commits Git locais que você não tenha `dcommited` não aparecem; nem commits que as pessoas fizeram no servidor Subversion neste meio tempo. É mais como o último estado conhecido dos commits no servidor Subversion.

SVN Annotation

Assim como o comando `git svn log` simula o comando `svn log` off-line, você pode obter o equivalente a `svn annotate` executando `git svn blame [FILE]`. A saída se parece com isto:

```

1  $ git svn blame README.txt
2  2    temporal Protocol Buffers - Google's data interchange format
3  2    temporal Copyright 2008 Google Inc.
4  2    temporal http://code.google.com/apis/protocolbuffers/
5  2    temporal
6  22   temporal C++ Installation - Unix
7  22   temporal =====
8  2    temporal
9  79   schacon Committing in git-svn.
10 78   schacon
11 2    temporal To build and install the C++ Protocol Buffer runtime and the Protoc\
12 ol
13 2    temporal Buffer compiler (protoc) execute the following:
14 2    temporal

```

Novamente, ele não mostra commits que você fez localmente no Git ou que foram adicionados no Subversion neste meio tempo.

Informações do Servidor SVN

Você também pode obter o mesmo tipo de informação que `svn info` lhe dá executando `git svn info`:

```
1 $ git svn info
2 Path: .
3 URL: https://schacon-test.googlecode.com/svn/trunk
4 Repository Root: https://schacon-test.googlecode.com/svn
5 Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
6 Revision: 87
7 Node Kind: directory
8 Schedule: normal
9 Last Changed Author: schacon
10 Last Changed Rev: 87
11 Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Ele é paracido com `blame` e `log` eles são executados offline e é atualizado só a partir da última vez que você se comunicou com o servidor Subversion.

Ignorando o Que o Subversion Ignora

Se você clonar um repositório Subversion que tem propriedades `svn:ignore` definidas em qualquer lugar, é provável que você deseje definir arquivos `.gitignore` correspondentes para que você não possa fazer o commit de arquivos acidentalmente. `git svn` tem dois comandos para ajudar com este problema. O primeiro é `git svn create-ignore`, que cria automaticamente arquivos `.gitignore` correspondentes para você, assim seu próximo commit pode incluí-los.

O segundo comando é `git svn show-ignore`, que imprime em `stdout` as linhas que você precisa para colocar em um arquivo `.gitignore` para que você possa redirecionar a saída do arquivo de exclusão de seu projeto:

```
1 $ git svn show-ignore > .git/info/exclude
```

Dessa forma, você não suja o projeto com arquivos `.gitignore`. Esta é uma boa opção se você é o único usuário Git em uma equipe Subversion, e seus companheiros de equipe não querem arquivos `.gitignore` no projeto.

Resumo do Git-Svn

As ferramentas do `git svn` são úteis se você está preso com um servidor Subversion por agora ou está em um ambiente de desenvolvimento que necessita executar um servidor Subversion. No entanto, você deve considerá-lo um Git “aleijado”, ou você vai encontrar problemas na tradução que pode confundir você e seus colaboradores. Para ficar fora de problemas, tente seguir estas orientações:

- Mantenha um histórico Git linear que não contém merge de commits realizados por `git merge`. Rebase qualquer trabalho que você fizer fora de seu branch principal (mainline) de volta para ele; não faça merge dele

- Não colabore em um servidor Git separado. Eventualmente, tenha um para acelerar clones para novos desenvolvedores, mas não faça push de nada para ele que não tenha uma entrada `git-svn-id`. Você pode até querer adicionar um hook `pre-receive` que verifica cada mensagem de confirmação para encontrar um `git-svn-id` e rejeitar pushes que contenham commits sem ele. Se você seguir essas orientações, trabalhar com um servidor Subversion pode ser mais fácil. No entanto, se for possível migrar para um servidor Git real, isso pode melhorar muito o ciclo de trabalho de sua equipe.

8.2 Migrando para o Git

Migrando para o Git

Se você tem uma base de código existente em outro VCS mas você decidiu começar a usar o Git, você deve migrar seu projeto de um jeito ou outro. Esta seção vai mostrar alguns importadores que estão incluídos no Git para sistemas comuns e demonstra como desenvolver o seu importador personalizado.

Importando

Você vai aprender como importar dados de dois dos maiores sistemas SCM utilizados profissionalmente — Subversion e Perforce — isso porque eles são usados pela maioria dos usuários que ouço falar, e porque ferramentas de alta qualidade para ambos os sistemas são distribuídos com Git.

Subversion

Se você ler a seção anterior sobre o uso do `git svn`, você pode facilmente usar essas instruções para `git svn clone` um repositório; então, parar de usar o servidor Subversion, fazer um push para um servidor Git novo, e começar a usar ele. Se precisar do histórico, você pode conseguir isso tão rapidamente como extrair os dados do servidor Subversion (que pode demorar um pouco).

No entanto, a importação não é perfeita; e já que vai demorar tanto tempo, você pode fazer isso direito. O primeiro problema é a informação do autor. No Subversion, cada pessoa commitando tem um usuário no sistema que está registrado nas informações de commit. Os exemplos nas seções anteriores mostram `schacon` em alguns lugares, como a saída do `blame` e do `git svn log`. Se você deseja mapear isso para obter melhores dados de autor no Git, você precisa fazer um mapeamento dos usuários do Subversion para os autores Git. Crie um arquivo chamado `users.txt` que tem esse mapeamento em um formato como este:

```
1 schacon = Scott Chacon <schacon@geemail.com>
2 selse = Someo Nelse <selse@geemail.com>
```

Para obter uma lista dos nomes de autores que o SVN usa, você pode executar isto:

```

1 $ svn log ^/ --xml | grep -P "^<author" | sort -u | \
2     perl -pe 's/<author>(.*?)</author>/$1 = /' > users.txt

```

Isso te dá a saída do log em formato XML — você pode pesquisar pelos autores, criar uma lista única, e depois tirar o XML. (Obviamente isso só funciona em uma máquina com `grep`, `sort`, e `perl` instalados.) Em seguida, redirecione a saída em seu arquivo `users.txt` assim, você pode adicionar os dados de usuários do Git equivalentes ao lado de cada entrada.

Você pode fornecer esse arquivo para `git svn` para ajudar a mapear os dados do autor com mais precisão. Você também pode dizer ao `git svn` para não incluir os metadados que o Subversion normalmente importa, passando `--no-metadata` para o comando `clone` ou `init`. Isso faz com que o seu comando `import` fique parecido com este:

```

1 $ git-svn clone http://my-project.googlecode.com/svn/ \
2     --authors-file=users.txt --no-metadata -s my_project

```

Agora você deve ter uma importação Subversion mais agradável no seu diretório `my_project`. Em vez de commits ele se parece com isso

```

1 commit 37efa680e8473b615de980fa935944215428a35a
2 Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
3 Date:   Sun May 3 00:12:22 2009 +0000
4
5     fixed install - go to trunk
6
7     git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
8     be05-5f7a86268029

```

they look like this:

```

1 commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
2 Author: Scott Chacon <schacon@gmail.com>
3 Date:   Sun May 3 00:12:22 2009 +0000
4
5     fixed install - go to trunk

```

Não só o campo `Author` parece muito melhor, mas o `git-svn-id` não está mais lá também.

Você precisa fazer um pouco de limpeza `post-import`. Por um lado, você deve limpar as referências estranhas que `git svn` configura. Primeiro você vai migrar as tags para que sejam tags reais, em vez de estranhos branches remotos, e então você vai migrar o resto dos branches de modo que eles sejam locais.

Para migrar as tags para que sejam tags Git adequadas, execute

```
1 $ cp -Rf .git/refs/remotes/tags/* .git/refs/tags/
2 $ rm -Rf .git/refs/remotes/tags
```

Isso leva as referências que eram branches remotos que começavam com tag/ e torna-os tags (leves) reais.

Em seguida, importamos o resto das referências em refs/remotes para serem branches locais:

```
1 $ cp -Rf .git/refs/remotes/* .git/refs/heads/
2 $ rm -Rf .git/refs/remotes
```

Agora todos os branches velhos são branches Git reais e todas as tags antigas são tags Git reais. A última coisa a fazer é adicionar seu servidor Git novo como um remoto e fazer um push nele. Aqui está um exemplo de como adicionar o servidor como um remoto:

```
1 $ git remote add origin git@my-git-server:myrepository.git
```

Já que você quer que todos os seus branches e tags sejam enviados, você pode executar isto:

```
1 $ git push origin --all
```

Todos os seus branches e tags devem estar em seu servidor Git novo em uma agradável importação limpa.

Perforce

O sistema seguinte de que importaremos é o Perforce. Um importador Perforce também é distribuído com Git, mas apenas na seção contrib do código fonte — que não está disponível por padrão como git svn. Para executá-lo, você deve obter o código fonte Git, que você pode baixar a partir de git.kernel.org:

```
1 $ git clone git://git.kernel.org/pub/scm/git/git.git
2 $ cd git/contrib/fast-import
```

Neste diretório fast-import, você deve encontrar um script Python executável chamado git-p4. Você deve ter o Python e a ferramenta p4 instalados em sua máquina para esta importação funcionar. Por exemplo, você vai importar o projeto Jam do depósito público Perforce. Para configurar o seu cliente, você deve exportar a variável de ambiente P4PORT para apontar para o depósito Perforce:

```
1 $ export P4PORT=public.perforce.com:1666
```

Execute o comando `git-p4 clone` para importar o projeto Jam do servidor Perforce, fornecendo o caminho do depósito e do projeto e o caminho no qual você deseja importar o projeto:

```
1 $ git-p4 clone //public/jam/src@all /opt/p4import
2 Importing from //public/jam/src@all into /opt/p4import
3 Reinitialized existing Git repository in /opt/p4import/.git/
4 Import destination: refs/remotes/p4/master
5 Importing revision 4409 (100%)
```

Se você for para o diretório `/opt/p4import` e executar `git log`, você pode ver o seu trabalho importado:

```
1 $ git log -2
2 commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
3 Author: Perforce staff <support@perforce.com>
4 Date: Thu Aug 19 10:18:45 2004 -0800
5
6 Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into
7 the main part of the document. Built new tar/zip balls.
8
9 Only 16 months later.
10
11 [git-p4: depot-paths = "//public/jam/src/": change = 4409]
12
13 commit ca8870db541a23ed867f38847eda65bf4363371d
14 Author: Richard Geiger <rmg@perforce.com>
15 Date: Tue Apr 22 20:51:34 2003 -0800
16
17 Update derived jamgram.c
18
19 [git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

Você pode ver o identificador do `git-p4` em cada commit. É bom manter esse identificador lá, caso haja necessidade de referenciar o número de mudança Perforce mais tarde. No entanto, se você gostaria de remover o identificador, agora é a hora de fazê-lo — antes de você começar a trabalhar no novo repositório. Você pode usar `git filter-branch` para remover as strings identificadoras em massa:

```
1 $ git filter-branch --msg-filter '  
2     sed -e "/^\[git-p4:/d"  
3 '  
4 Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)  
5 Ref 'refs/heads/master' was rewritten
```

Se você executar o `git log`, você pode ver que todas as checksums SHA-1 dos commits mudaram, mas as strings do `git-p4` não estão mais nas mensagens de commit:

```
1 $ git log -2  
2 commit 10a16d60cffca14d454a15c6164378f4082bc5b0  
3 Author: Perforce staff <support@perforce.com>  
4 Date: Thu Aug 19 10:18:45 2004 -0800  
5  
6 Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into  
7 the main part of the document. Built new tar/zip balls.  
8  
9 Only 16 months later.  
10  
11 commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2  
12 Author: Richard Geiger <rmg@perforce.com>  
13 Date: Tue Apr 22 20:51:34 2003 -0800  
14  
15 Update derived jamgram.c
```

Sua importação está pronta para ser enviada (push) para o seu novo servidor Git.

Um Importador Customizado

Se o sistema não é o Subversion ou Perforce, você deve procurar um importador online — importadores de qualidade estão disponíveis para CVS, Clear Case, Visual Source Safe, e até mesmo um diretório de arquivos. Se nenhuma destas ferramentas funcionar para você, você tem uma ferramenta mais rara, ou você precisa de um processo de importação personalizado, você deve usar `git fast-import`. Este comando lê instruções simples da `stdin` para gravar dados Git específicos. É muito mais fácil criar objetos Git desta forma do que executar os comandos crús do Git ou tentar escrever os objetos crús (ver Capítulo 9 para mais informações). Dessa forma, você pode escrever um script de importação que lê as informações necessárias do sistema de que está importando e imprimir instruções simples no `stdout`. Você pode então executar este programa e redirecionar sua saída através do `git fast-import`.

Para demonstrar rapidamente, você vai escrever um importador simples. Suponha que você faz backup do seu projeto de vez em quando copiando o diretório em um diretório de backup nomeado por data `back_YYYY_MM_DD`, e você deseja importar ele no Git. Sua estrutura de diretórios fica assim:

```
1 $ ls /opt/import_from
2 back_2009_01_02
3 back_2009_01_04
4 back_2009_01_14
5 back_2009_02_03
6 current
```

Para importar um diretório Git, você precisa rever como o Git armazena seus dados. Como você pode lembrar, Git é fundamentalmente uma lista encadeada de objetos commit que apontam para um snapshot de um conteúdo. Tudo que você tem a fazer é dizer ao `fast-import` quais são os snapshots de conteúdo, que dados de commit apontam para eles, e a ordem em que estão dispostos. Sua estratégia será a de passar pelos snapshots um de cada vez e criar commits com o conteúdo de cada diretório, ligando cada commit de volta para a anterior.

Como você fez na seção “Um exemplo de Política Git Forçada” do *Capítulo 7*, vamos escrever isso em Ruby, porque é o que eu geralmente uso e tende a ser fácil de ler. Você pode escrever este exemplo muito facilmente em qualquer linguagem que você esteja familiarizado — ele só precisa imprimir a informação apropriada para o `stdout`. E, se você estiver rodando no Windows, isso significa que você terá que tomar cuidados especiais para não introduzir carriage returns no final de suas linhas — git `fast-import` só aceita line feeds (LF) e não aceita carriage return line feeds (CRLF) que o Windows usa.

Para começar, você vai mudar para o diretório de destino e identificar cada subdiretório, cada um dos quais é um instantâneo que você deseja importar como um commit. Você vai entrar em cada subdiretório e imprimir os comandos necessários para exportá-los. Seu loop básico principal fica assim:

```
1 last_mark = nil
2
3 # loop through the directories
4 Dir.chdir(ARGV[0]) do
5   Dir.glob("*").each do |dir|
6     next if File.file?(dir)
7
8     # move into the target directory
9     Dir.chdir(dir) do
10      last_mark = print_export(dir, last_mark)
11    end
12  end
13 end
```

Você executa o `print_export` dentro de cada diretório, o que pega o manifest e marca do snapshot anterior e retorna o manifest e marca deste; dessa forma, você pode ligá-los corretamente. “Mark”

é o termo `fast-import` para um identificador que você dá a um commit; como você cria commits, você dá a cada um uma marca que você pode usar para ligar ele a outros commits. Então, a primeira coisa a fazer em seu método `print_export` é gerar uma mark do nome do diretório:

```
1 mark = convert_dir_to_mark(dir)
```

Você vai fazer isso criando uma matriz de diretórios e usar o valor do índice como a marca, porque uma marca deve ser um inteiro. Seu método deve se parecer com este:

```
1 $marks = []
2 def convert_dir_to_mark(dir)
3   if !$marks.include?(dir)
4     $marks << dir
5   end
6   ($marks.index(dir) + 1).to_s
7 end
```

Agora que você tem uma representação usando um número inteiro de seu commit, você precisa de uma data para os metadados do commit. Como a data está expressa no nome do diretório, você vai utilizá-la. A próxima linha em seu arquivo `print_export` é

```
1 date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
1 def convert_dir_to_date(dir)
2   if dir == 'current'
3     return Time.now().to_i
4   else
5     dir = dir.gsub('back_', '')
6     (year, month, day) = dir.split('_')
7     return Time.local(year, month, day).to_i
8   end
9 end
```

Que retorna um valor inteiro para a data de cada diretório. A última peça de meta-dado que você precisa para cada commit são os dados do committer (autor do commit), que você codificar em uma variável global:

```
1 $author = 'Scott Chacon <schacon@example.com>'
```

Agora você está pronto para começar a imprimir os dados de commit para o seu importador. As primeiras informações indicam que você está definindo um objeto commit e que branch ele está, seguido pela mark que você gerou, a informação do committer e mensagem de commit, e o commit anterior, se houver. O código fica assim:

```
1 # print the import information
2 puts 'commit refs/heads/master'
3 puts 'mark :' + mark
4 puts "committer #{ $author } #{date} -0700"
5 export_data('imported from ' + dir)
6 puts 'from :' + last_mark if last_mark
```

Você coloca um valor estático (hardcode) do fuso horário (-0700), porque é mais fácil. Se estiver importando de outro sistema, você deve especificar o fuso horário como um offset. A mensagem de confirmação deve ser expressa em um formato especial:

```
1 data (size)\n(contents)
```

O formato consiste dos dados de texto, o tamanho dos dados a serem lidos, uma quebra de linha, e finalmente os dados. Como você precisa utilizar o mesmo formato para especificar o conteúdo de arquivos mais tarde, você pode criar um método auxiliar, `export_data`:

```
1 def export_data(string)
2   print "data #{string.size}\n#{string}"
3 end
```

Tudo o que resta é especificar o conteúdo do arquivo para cada snapshot. Isso é fácil, porque você tem cada um em um diretório — você pode imprimir o comando `deleteall` seguido do conteúdo de cada arquivo no diretório. Git, então, grava cada instantâneo apropriadamente:

```
1 puts 'deleteall'
2 Dir.glob("**/*").each do |file|
3   next if !File.file?(file)
4   inline_data(file)
5 end
```

Nota: Como muitos sistemas tratam suas revisões, como mudanças de um commit para outro, fast-import também pode receber comandos com cada commit para especificar quais arquivos foram adicionados, removidos ou modificados e o que os novos conteúdos são. Você poderia calcular as diferenças entre os snapshots e fornecer apenas estes dados, mas isso é mais complexo — assim como você pode dar ao Git todos os dados e deixá-lo descobrir. Se isto for mais adequado aos seus dados, verifique a man page (manual) do fast-import para obter detalhes sobre como fornecer seus dados desta forma.

O formato para listar o conteúdo de arquivos novos ou especificar um arquivo modificado com o novo conteúdo é o seguinte:

```
1 M 644 inline path/to/file
2 data (size)
3 (file contents)
```

Aqui, 644 é o modo (se você tiver arquivos executáveis, é preciso detectá-los e especificar 755), e inline diz que você vai listar o conteúdo imediatamente após esta linha. O seu método `inline_data` deve se parecer com este:

```
1 def inline_data(file, code = 'M', mode = '644')
2   content = File.read(file)
3   puts "#{code} #{mode} inline #{file}"
4   export_data(content)
5 end
```

Você reutiliza o método `export_data` definido anteriormente, porque é da mesma forma que você especificou os seus dados de mensagem de commit.

A última coisa que você precisa fazer é retornar a mark atual para que possa ser passada para a próxima iteração:

```
1 return mark
```

NOTA: Se você estiver usando Windows, você vai precisar se certificar de que você adiciona um passo extra. Como já relatado anteriormente, o Windows usa CRLF para quebras de linha enquanto git fast-import aceita apenas LF. Para contornar este problema e usar git fast-import, você precisa dizer ao ruby para usar LF em vez de CRLF:

```
1 $stdout.binmode
```

Isso é tudo. Se você executar este script, você vai ter um conteúdo parecido com isto:

```

1  $ ruby import.rb /opt/import_from
2  commit refs/heads/master
3  mark :1
4  committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
5  data 29
6  imported from back_2009_01_02deleteall
7  M 644 inline file.rb
8  data 12
9  version two
10 commit refs/heads/master
11 mark :2
12 committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
13 data 29
14 imported from back_2009_01_04from :1
15 deleteall
16 M 644 inline file.rb
17 data 14
18 version three
19 M 644 inline new.rb
20 data 16
21 new version one
22 (...)

```

Para executar o importador, redirecione a saída através do `git fast-import` enquanto estiver no diretório Git que você quer importar. Você pode criar um novo diretório e executar `git init` nele para iniciar, e depois executar o script:

```

1  $ git init
2  Initialized empty Git repository in /opt/import_to/.git/
3  $ ruby import.rb /opt/import_from | git fast-import
4  git-fast-import statistics:
5  -----
6  Alloc'd objects:      5000
7  Total objects:        18 (      1 duplicates      )
8     blobs   :          7 (      1 duplicates      0 deltas)
9     trees   :          6 (      0 duplicates      1 deltas)
10    commits:          5 (      0 duplicates      0 deltas)
11     tags    :          0 (      0 duplicates      0 deltas)
12 Total branches:        1 (      1 loads      )
13     marks:    1024 (      5 unique      )
14     atoms:         3
15 Memory total:        2255 KiB

```

```

16      pools:          2098 KiB
17      objects:         156 KiB
18 -----
19 pack_report: getpagesize()      =      4096
20 pack_report: core.packedGitWindowSize =    33554432
21 pack_report: core.packedGitLimit   =   268435456
22 pack_report: pack_used_ctr        =          9
23 pack_report: pack_mmap_calls      =          5
24 pack_report: pack_open_windows    =         1 /         1
25 pack_report: pack_mapped          =       1356 /       1356
26 -----

```

Como você pode ver, quando for concluído com êxito, ele mostra um monte de estatísticas sobre o que ele realizou. Neste caso, você importou um total de 18 objetos para 5 commits em 1 branch. Agora, você pode executar `git log` para ver seu novo histórico:

```

1 $ git log -2
2 commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
3 Author: Scott Chacon <schacon@example.com>
4 Date:   Sun May 3 12:57:39 2009 -0700
5
6     imported from current
7
8 commit 7e519590de754d079dd73b44d695a42c9d2df452
9 Author: Scott Chacon <schacon@example.com>
10 Date:   Tue Feb 3 01:00:00 2009 -0700
11
12     imported from back_2009_02_03

```

Ai está — um repositório Git limpo. É importante notar que não é feito check-out de nada — você não tem arquivos em seu diretório de trabalho no início. Para obtê-los, você deve redefinir o seu branch para master:

```

1 $ ls
2 $ git reset --hard master
3 HEAD is now at 10bfe7d imported from current
4 $ ls
5 file.rb  lib

```

Você pode fazer muito mais com a ferramenta `fast-import` — lidar com diferentes modos, dados binários, múltiplos branches e mesclagem (merging), tags, indicadores de progresso, e muito mais. Uma série de exemplos de cenários mais complexos estão disponíveis no diretório `contrib/fast-import` do código-fonte Git, um dos melhores é o script `git-p4` que acabei de mostrar.

8.3 Git e Outros Sistemas - Resumo

Resumo

Você deve se sentir confortável usando Git com o Subversion ou importar quase qualquer repositório existente em um novo repositório Git sem perder dados. O próximo capítulo irá cobrir o funcionamento interno do Git para que você possa criar cada byte, se for necessário.

Capítulo 9 - Git Internamente

Você pode ter pulado para este capítulo a partir de um capítulo anterior, ou você pode ter chegado aqui depois de ler o resto do livro — em ambos os casos, este é o lugar onde você vai conhecer o funcionamento interno e implementação do Git. Descobri que aprender esta informação era de fundamental importância para a compreensão de quanto o Git é útil e poderoso, mas outros argumentaram que pode ser confuso e desnecessariamente complexo para iniciantes. Assim, eu fiz essa discussão o último capítulo do livro para que você possa lê-lo mais cedo ou mais tarde, em seu processo de aprendizagem. Deixo isso para você decidir.

Agora que você está aqui, vamos começar. Primeiro, se ainda não for claro, o Git é fundamentalmente um sistema de arquivos de conteúdo endereçável com uma interface de usuário VCS escrito em cima dele. Você vai aprender mais sobre o que isto significa daqui a pouco.

Nos primeiros dias do Git (principalmente antes da versão 1.5), a interface de usuário era muito mais complexa, pois enfatizou este sistema de arquivos, em vez de um VCS. Nos últimos anos, a interface do usuário tem sido aperfeiçoada até que ela se torne tão limpa e fácil de usar como qualquer outro sistema; mas, muitas vezes, o estereótipo persiste sobre a UI antiga do Git que era complexa e difícil de aprender.

A camada de sistema de arquivos de conteúdo endereçável é incrivelmente interessante, então eu vou falar dela primeiro neste capítulo; então, você vai aprender sobre os mecanismos de transporte e as tarefas de manutenção do repositório.

9.1 Git Internamente - Encanamento (Plumbing) e Porcelana (Porcelain)

Encanamento (Plumbing) e Porcelana (Porcelain)

Este livro aborda como usar o Git com 30 ou mais verbos como `checkout`, `branch`, `remote`, e assim por diante. Mas como o Git foi pensado inicialmente um conjunto de ferramentas para um VCS, em vez de apenas um VCS amigável, ele tem um grupo de verbos que fazem o trabalho de baixo nível e foram projetados para serem encadeados (usando pipe) no estilo UNIX ou chamados a partir de scripts. Estes comandos são geralmente referidos como comandos de “encanamento” (plumbing), e os comandos mais amigáveis são chamados de comandos “porcelana” (porcelain).

Os oito primeiros capítulos do livro tratam quase que exclusivamente com comandos porcelana. Mas, neste capítulo, você estará lidando principalmente com os comandos de nível inferior de encanamento, porque eles te dão acesso aos trabalhos internos do Git e ajudam a demonstrar como e por que o Git faz o que faz. Estes comandos não são destinados a ser usados manualmente na

linha de comando, mas sim para serem usados como blocos de construção para novas ferramentas e scripts personalizados.

Quando você executa `git init` em um diretório novo ou existente, Git cria o diretório `.git`, que é onde quase tudo que o Git armazena e manipula está localizado. Se você deseja fazer o backup ou clonar seu repositório, copiar este diretório para outro lugar lhe dará quase tudo o que você precisa. Este capítulo inteiro trata basicamente do conteúdo deste diretório. Eis o que ele contém:

```
1 $ ls
2 HEAD
3 branches/
4 config
5 description
6 hooks/
7 index
8 info/
9 objects/
10 refs/
```

Você pode ver alguns outros arquivos lá, mas este é um novo repositório `git init` — é o que você vê por padrão. O diretório `branches` não é usado por versões mais recentes do Git, e o arquivo `description` só é usado pelo programa `gitweb`, então não se preocupe com eles. O arquivo `config` contém as opções de configuração específicas do projeto, e o diretório `info` contém um arquivo de exclusão global com padrões ignorados que você não deseja rastrear em um arquivo `.gitignore`. O diretório `hooks` contém os “hook scripts” de cliente ou servidor, que são discutidos em detalhes no *Capítulo 7*.

Isso deixa quatro entradas importantes: os arquivos `HEAD` e `index` e diretórios `objects` e `refs`. Estas são as peças centrais do Git. O diretório `objects` armazena todo o conteúdo do seu banco de dados, o diretório `refs` armazena os ponteiros para objetos de commit (`branches`), o arquivo `HEAD` aponta para o branch atual, e o arquivo `index` é onde Git armazena suas informações da área de preparação (`staging area`). Você vai agora ver cada uma dessas seções em detalhes para saber como o Git opera.

9.2 Git Internamente - Objetos do Git

Objetos do Git

Git é um sistema de arquivos de conteúdo endereçável. O que significa isso? Isso significa que o núcleo do Git armazena dados usando um simples mecanismo chave-valor. Você pode inserir qualquer tipo de conteúdo nele, e ele vai retornar uma chave que você pode usar para recuperar o conteúdo a qualquer momento. Para demonstrar, você pode usar o comando de encanamento `hash-object`, pega alguns dados, armazena eles em seu diretório `.git`, e retorna a chave dos dados armazenados. Primeiro, você inicializa um novo repositório Git e verifica se não há nada no diretório `objects`:


```
1 $ mkdir test
2 $ cd test
3 $ git init
4 Initialized empty Git repository in /tmp/test/.git/
5 $ find .git/objects
6 .git/objects
7 .git/objects/info
8 .git/objects/pack
9 $ find .git/objects -type f
10 $
```

Git inicializou o diretório `objects` e criou os subdiretórios `pack` e `info`, mas não existem arquivos comuns neles. Agora, armazene um texto em seu banco de dados Git:

```
1 $ echo 'test content' | git hash-object -w --stdin
2 d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

O `-w` diz ao `hash-object` para armazenar o objeto; caso contrário, o comando simplesmente diz qual seria a chave. `--stdin` indica ao comando para ler o conteúdo da entrada padrão (`stdin`); se você não especificar isso, `hash-object` espera um caminho (`path`) para um arquivo. A saída do comando é uma soma hash de 40 caracteres. Este é o hash SHA-1 — uma soma de verificação do conteúdo que você está armazenando mais um cabeçalho, que você vai entender mais daqui a pouco. Agora você pode ver como o Git armazenou seus dados:

```
1 $ find .git/objects -type f
2 .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Você pode ver um arquivo no diretório `objects`. Isto é como o Git armazena o conteúdo inicialmente — como um único arquivo por parte de conteúdo, nomeado com o checksum SHA-1 do conteúdo e seu cabeçalho. O subdiretório é nomeado com os 2 primeiros caracteres do SHA, e o arquivo é nomeado com os 38 caracteres restantes.

Você pode fazer um pull do conteúdo com o comando `cat-file`. Este comando é uma espécie de canivete suíço para inspecionar objetos Git. Passando `-p` para ele instrui o comando `cat-file` a descobrir o tipo de conteúdo e exibi-lo para você:

```
1 $ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
2 test content
```

Agora, você pode adicionar conteúdo no Git e fazer pull dele. Você também pode fazer isso com o conteúdo de arquivos. Por exemplo, você pode fazer algum controle de versão simples em um arquivo. Primeiro, crie um novo arquivo e salve seu conteúdo em seu banco de dados:

```
1 $ echo 'version 1' > test.txt
2 $ git hash-object -w test.txt
3 83baae61804e65cc73a7201a7252750c76066a30
```

Então, escreva algum conteúdo novo no arquivo e salve-o novamente:

```
1 $ echo 'version 2' > test.txt
2 $ git hash-object -w test.txt
3 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Seu banco de dados contém as duas novas versões do arquivo, assim como o primeiro conteúdo que você armazenou lá:

```
1 $ find .git/objects -type f
2 .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
3 .git/objects/83/baae61804e65cc73a7201a7252750c76066a30
4 .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Agora você pode reverter o arquivo de volta para a primeira versão

```
1 $ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
2 $ cat test.txt
3 version 1
```

ou a segunda versão:

```
1 $ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
2 $ cat test.txt
3 version 2
```

Mas, lembrar a chave SHA-1 para cada versão de seu arquivo não é prático; mais ainda, você não está armazenando o nome do arquivo em seu sistema — apenas o conteúdo. Esse tipo de objeto é chamado de blob. Você pode fazer o Git informar o tipo de objeto de qualquer objeto no Git, dada a sua chave SHA-1, com `cat-file -t`:

```
1 $ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
2 blob
```

Objetos Árvore

O próximo tipo que você verá é o objeto árvore, que resolve o problema de armazenar o nome do arquivo e também permite que você armazene um grupo de arquivos juntos. Git armazena o conteúdo de uma maneira semelhante a um sistema de arquivos UNIX, mas de forma um pouco simplificada. Todo o conteúdo é armazenado como árvore e objetos blob, com árvores correspondendo às entradas de diretório do UNIX e os blobs correspondendo mais ou menos a inodes ou conteúdo de arquivos. Um único objeto árvore contém uma ou mais entradas de árvores, cada uma das quais contém um ponteiro SHA-1 para um blob ou subárvore com seu modo associado, tipo e o nome do arquivo. Por exemplo, a árvore mais recente no projeto simplegit pode parecer algo como isto:

```
1 $ git cat-file -p master^{tree}
2 100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
3 100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
4 040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

A sintaxe `master^{tree}` especifica o objeto árvore que é apontado pelo último commit em seu branch `master`. Observe que o subdiretório `lib` não é um blob, mas sim, um ponteiro para outra árvore:

```
1 $ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
2 100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceitualmente, os dados que o Git está armazenando são algo como mostra a Figura 9-1.

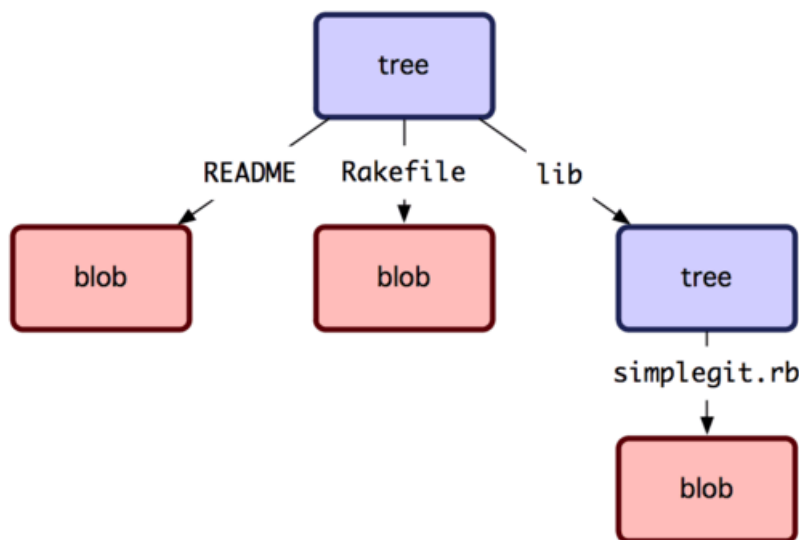


Figura 9-1. Versão simples do modelo de dados Git.

Você pode criar sua própria árvore. Git normalmente cria uma árvore, a partir do estado de sua área de seleção ou índice e escreve um objeto árvore a partir dele. Assim, para criar um objeto árvore, primeiro você tem que criar um índice colocando alguns arquivos na área de seleção (staging area). Para criar um índice com uma única entrada — a primeira versão do seu arquivo `test.txt` — você pode usar o comando `plumbing update-index`. Você pode usar este comando para adicionar artificialmente a versão anterior do arquivo `test.txt` em uma nova área de seleção. Você deve passar a opção `--add` porque o arquivo ainda não existe na sua área de seleção (você não tem sequer uma área de seleção criada ainda) e `--cacheinfo` porque o arquivo que você está adicionando não está em seu diretório, mas está em seu banco de dados. Então, você especifica o modo, o SHA-1, e o nome do arquivo:

```
1 $ git update-index --add --cacheinfo 100644 \  
2 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Neste caso, você está especificando um modo `100644`, que significa que é um arquivo normal. Outras opções são `100755`, que significa que é um arquivo executável, e `120000`, que especifica um link simbólico. O modo é obtido a partir de modos normais do Unix, mas é muito menos flexível — estes três modos são os únicos que são válidas para arquivos (blobs) no Git (embora outros modos sejam usados para diretórios e submódulos).

Agora, você pode usar o comando `write-tree` para escrever a área de seleção em um objeto árvore. Nenhuma opção `-w` é necessária — chamando `write-tree` cria automaticamente um objeto árvore a partir do estado do índice se a árvore ainda não existe:

```
1 $ git write-tree  
2 d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
3 $ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
4 100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Você também pode verificar que este é um objeto árvore:

```
1 $ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
2 tree
```

Você vai agora criar uma nova árvore com a segunda versão do `test.txt` e um novo arquivo também:

```
1 $ echo 'new file' > new.txt  
2 $ git update-index test.txt  
3 $ git update-index --add new.txt
```

Sua área de seleção agora tem a nova versão do `test.txt` bem como o novo arquivo `new.txt`. Escreva aquela árvore (grave o estado da área de seleção ou índice em um objeto árvore) e veja o que aparece:

```

1 $ git write-tree
2 0155eb4229851634a0f03eb265b69f5a2d56f341
3 $ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
4 100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
5 100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt

```

Note que esta árvore tem entradas de arquivos e também que o SHA de test.txt é a “versão 2” do SHA de antes (1f7a7a). Apenas por diversão, você vai adicionar a primeira árvore como um subdiretório nesta árvore. Você pode ler as árvores em sua área de seleção chamando read-tree. Neste caso, você pode ler uma árvore existente em sua área de seleção como uma subárvore usando a opção --prefix em read-tree:

```

1 $ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
2 $ git write-tree
3 3c4e9cd789d88d8d89c1073707c3585e41b0e614
4 $ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
5 040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
6 100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
7 100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt

```

Se você criou um diretório de trabalho da nova árvore que acabou de escrever, você teria os dois arquivos no nível mais alto do diretório de trabalho e um subdiretório chamado bak, que continha a primeira versão do arquivo teste.txt. Você pode pensar nos dados que o Git contém para estas estruturas como sendo parecidas com a Figura 9-2.

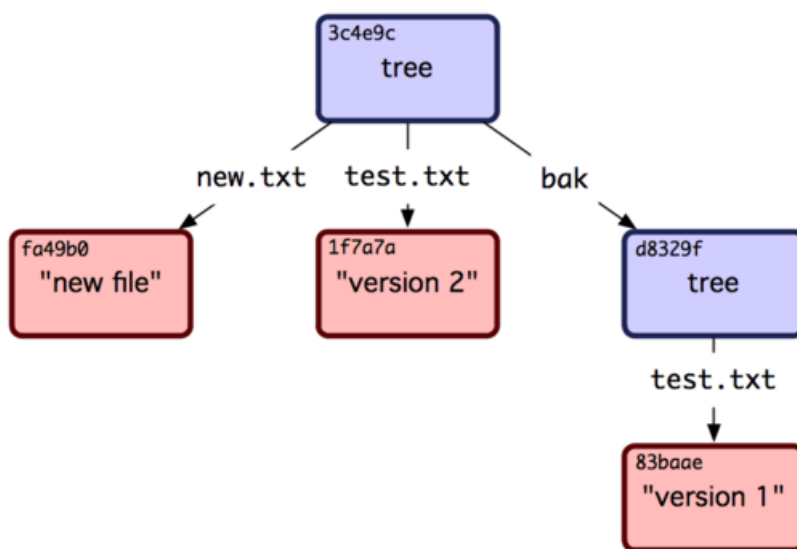


Figura 9-2. A estrutura de conteúdo de seus dados Git atuais.

Objetos de Commit

Você tem três árvores que especificam os diferentes snapshots de seu projeto que você deseja acompanhar, mas o problema anterior mantém-se: você deve se lembrar de todos os três valores SHA-1, a fim de recuperar os snapshots. Você também não tem qualquer informação sobre quem salvou os snapshots, quando eles foram salvos, ou por que eles foram salvos. Esta é a informação básica que os objetos commit armazenam para você.

Para criar um objeto commit, você chama `commit-tree` e especifica uma única árvore SHA-1 e quais objetos commit, se houverem, diretamente precederam ele. Comece com a primeira árvore que você escreveu:

```
1 $ echo 'first commit' | git commit-tree d8329f
2 fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Agora você pode ver o seu novo objeto commit com `cat-file`:

```
1 $ git cat-file -p fdf4fc3
2 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
3 author Scott Chacon <schacon@gmail.com> 1243040974 -0700
4 committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
5
6 first commit
```

O formato de um objeto commit é simples: ele especifica a árvore de nível superior para o snapshot do projeto nesse momento; a informação do author/committer (autor do commit) obtido de suas opções de configuração `user.name` e `user.email`, com a timestamp atual; uma linha em branco, e em seguida, a mensagem de commit.

Em seguida, você vai escrever os outros dois objetos commit, cada um referenciando o commit que veio diretamente antes dele:

```
1 $ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
2 cac0cab538b970a37ea1e769cbbde608743bc96d
3 $ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
4 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Cada um dos três objetos commit apontam para uma das três árvores de snapshot criadas. Curiosamente, agora você tem um histórico Git real que você pode ver com o comando `git log`, se você executá-lo no último commit SHA-1:

```
1 $ git log --stat 1a410e
2 commit 1a410efbd13591db07496601ebc7a059dd55cfe9
3 Author: Scott Chacon <schacon@gmail.com>
4 Date:   Fri May 22 18:15:24 2009 -0700
5
6     third commit
7
8     bak/test.txt |    1 +
9     1 files changed, 1 insertions(+), 0 deletions(-)
10
11 commit cac0cab538b970a37ea1e769cbbde608743bc96d
12 Author: Scott Chacon <schacon@gmail.com>
13 Date:   Fri May 22 18:14:29 2009 -0700
14
15     second commit
16
17     new.txt |    1 +
18     test.txt |    2 +
19     2 files changed, 2 insertions(+), 1 deletions(-)
20
21 commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
22 Author: Scott Chacon <schacon@gmail.com>
23 Date:   Fri May 22 18:09:34 2009 -0700
24
25     first commit
26
27     test.txt |    1 +
28     1 files changed, 1 insertions(+), 0 deletions(-)
```

Incrível. Você acabou de fazer as operações de baixo nível para construir um histórico Git sem usar qualquer um dos front ends. Isso é essencialmente o que o Git faz quando você executa os comandos `git add` e `git commit` — ele armazena blobs dos arquivos que foram alterados, atualiza o índice, escreve árvores, e escreve objetos commit que fazem referência às árvores de nível superior e os commits que vieram imediatamente antes deles. Esses três objetos Git principais — o blob, a árvore, e o commit — são inicialmente armazenados como arquivos separados no seu diretório `.git/objects`. Aqui estão todos os objetos no diretório de exemplo agora, comentado com o que eles armazenam:

```

1 $ find .git/objects -type f
2 .git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
3 .git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
4 .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
5 .git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
6 .git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
7 .git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
8 .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
9 .git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
10 .git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
11 .git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1

```

Se você seguir todos os ponteiros internos, você tem um gráfico como o da Figura 9-3.

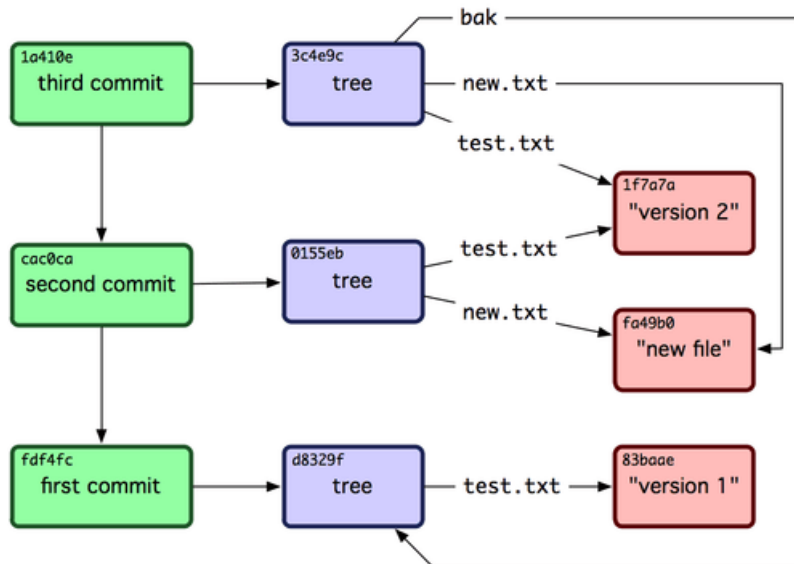


Figura 9-3. Todos os objetos em seu diretório Git.

Armazenamento de Objetos

Eu mencionei anteriormente que um cabeçalho é armazenado com o conteúdo. Vamos ver como Git armazena seus objetos. Você vai ver como armazenar um objeto blob — neste caso, a string “what is up, doc?” — interativamente na linguagem de script Ruby. Você pode iniciar o modo interativo Ruby com o comando `irb`:

```

1 $ irb
2 >> content = "what is up, doc?"
3 => "what is up, doc?"

```

Git constrói um cabeçalho que começa com o tipo do objeto, neste caso um blob. Em seguida, é adicionado um espaço seguindo-se a dimensão do conteúdo e, finalmente, um byte nulo:


```

1 >> header = "blob #{content.length}\0"
2 => "blob 16\000"

```

Git concatena o cabeçalho e o conteúdo original e, em seguida, calcula o checksum SHA-1 do novo conteúdo. Você pode calcular o valor SHA-1 em Ruby, incluindo a biblioteca SHA1 digest com o comando `require` e então chamar `Digest::SHA1.hexdigest()` passando a string:

```

1 >> store = header + content
2 => "blob 16\000what is up, doc?"
3 >> require 'digest/sha1'
4 => true
5 >> sha1 = Digest::SHA1.hexdigest(store)
6 => "bd9dbf5aae1a3862dd1526723246b20206e5fc37"

```

Git comprime o conteúdo novo com `zlib`, o que você pode fazer em Ruby com a biblioteca `zlib`. Primeiro, você precisa incluir a biblioteca e, em seguida, executar `Zlib::Deflate.deflate()` no conteúdo:

```

1 >> require 'zlib'
2 => true
3 >> zlib_content = Zlib::Deflate.deflate(store)
4 => "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\3110\266\a\000_\034\a\235"

```

Finalmente, você vai escrever o seu conteúdo “zlib-deflated” em um objeto no disco. Você vai determinar o caminho do objeto que você deseja escrever (os dois primeiros caracteres do valor SHA-1 serão o nome do subdiretório, e os últimos 38 caracteres serão 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. Em seguida, abra o arquivo com `File.open()` e escreva o conteúdo anteriormente comprimido com `zlib` no arquivo com uma chamada a `write()` no identificador de arquivo resultante:

```

1 >> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
2 => ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
3 >> require 'fileutils'
4 => true
5 >> FileUtils.mkdir_p(File.dirname(path))
6 => ".git/objects/bd"
7 >> File.open(path, 'w') { |f| f.write zlib_content }
8 => 32

```

Isso é tudo — você criou um objeto Git blob válido. Todos os objetos do Git são armazenados da mesma maneira, só que com diferentes tipos — em vez da string blob, o cabeçalho vai começar com `commit` ou `tree`. Além disso, embora o conteúdo do blob possa ser pequeno, o `commit` e conteúdo da árvore são formatados muito especificamente.

9.3 Git Internamente - Referencias Git

Referencias Git

Você pode executar algo como `git log 1a410e` para ver seu histórico inteiro, mas você ainda tem que lembrar que `1a410e` é o último commit, a fim de que o você possa navegar no histórico para encontrar todos os objetos. Você precisa de um arquivo no qual você possa armazenar o valor SHA-1 em um nome simples para que você possa usar esse ponteiro em vez do valor SHA-1.

No Git, eles são chamados de “referências” (references) ou “refs”; você pode encontrar os arquivos que contêm os valores SHA-1 no diretório `.git/refs`. No projeto atual, esse diretório não contém arquivos, mas contém uma estrutura simples:

```
1 $ find .git/refs
2 .git/refs
3 .git/refs/heads
4 .git/refs/tags
5 $ find .git/refs -type f
6 $
```

Para criar uma nova referência que irá ajudá-lo a se lembrar onde seu último commit está, você pode tecnicamente fazer algo tão simples como isto:

```
1 $ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Agora, você pode usar a referência head que você acabou de criar em vez do valor SHA-1 em seus comandos do Git:

```
1 $ git log --pretty=oneline master
2 1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
3 cac0cab538b970a37ea1e769cbbde608743bc96d second commit
4 fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Você não deve editar diretamente os arquivos de referência. Git oferece um comando mais seguro para fazer isso, se você deseja atualizar uma referência chamada `update-ref`:

```
1 $ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Isso é basicamente o que um branch em Git: um simples ponteiro ou referência para o head de uma linha de trabalho. Para criar um branch de volta ao segundo commit, você pode fazer isso:

```
1 $ git update-ref refs/heads/test cac0ca
```

Seu branch irá conter apenas o trabalho do commit abaixo:

```
1 $ git log --pretty=oneline test
2 cac0cab538b970a37ea1e769cbbde608743bc96d second commit
3 fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Agora, seu banco de dados Git conceitualmente é algo como a Figura 9-4.

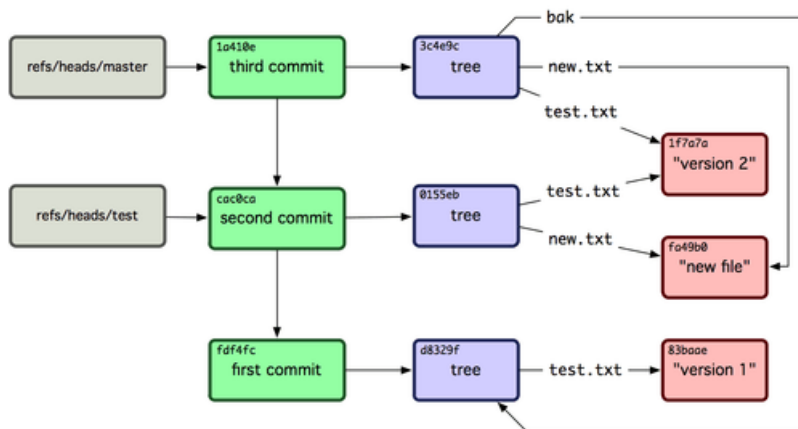


Figura 9-4. Objetos de diretório Git com referências ao branch head incluídas.

Quando você executar comandos como `git branch (branchname)`, Git basicamente executa o comando `update-ref` para adicionar o SHA-1 do último commit do branch em que você está em qualquer nova referência que deseje criar.

O HEAD

A questão agora é, quando você executar `git branch (branchname)`, como é que o Git sabe o SHA-1 do último commit? A resposta é o arquivo HEAD. O arquivo HEAD é uma referência simbólica para o branch em que você está no momento. Por referência simbólica, quer dizer que, ao contrário de uma referência normal, ele geralmente não contém um valor SHA-1 mas sim um apontador para uma outra referência. Se você olhar no arquivo, você normalmente verá algo como isto:

```
1 $ cat .git/HEAD
2 ref: refs/heads/master
```

Se você executar `git checkout test`, Git atualiza o arquivo para ficar assim:

```
1 $ cat .git/HEAD
2 ref: refs/heads/test
```

Quando você executar `git commit`, ele criará o objeto commit, especificando o pai desse objeto commit para ser o valor SHA-1 de referência apontada por HEAD.

Você também pode editar manualmente esse arquivo, mas um comando mais seguro existe para fazer isso: `symbolic-ref`. Você pode ler o valor de seu HEAD através deste comando:

```
1 $ git symbolic-ref HEAD
2 refs/heads/master
```

Você também pode definir o valor de HEAD:

```
1 $ git symbolic-ref HEAD refs/heads/test
2 $ cat .git/HEAD
3 ref: refs/heads/test
```

Você não pode definir uma referência simbólica fora do estilo refs:

```
1 $ git symbolic-ref HEAD test
2 fatal: Refusing to point HEAD outside of refs/
```

Tags

Você acabou de ver os três tipos de objetos principais do Git, mas há um quarto. O objeto tag é muito parecido com um objeto commit — contém um tagger (pessoa que cria a tag), uma data, uma mensagem e um ponteiro. A principal diferença é que um objeto tag aponta para um commit em vez de uma árvore. É como uma referência de branch, mas nunca se move — ele sempre aponta para o mesmo commit, mas te dá um nome mais amigável para ele.

Como discutido no Capítulo 2, existem dois tipos de tags: anotadas (annotated) e leves (lightweight). Você pode fazer uma tag leve executando algo como isto:

```
1 $ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Isso é tudo que uma tag leve é — um branch que nunca se move. Uma tag anotada é mais complexa. Se você criar uma tag anotada, Git cria um objeto tag e depois escreve uma referência para apontar para ela em vez de diretamente para um commit. Você pode ver isso através da criação de uma tag anotada (-a especifica que é uma tag anotada):

```
1 $ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Aqui está o valor SHA-1 do objeto que ele criou:

```
1 $ cat .git/refs/tags/v1.1
2 9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Agora, execute o comando `cat-file` com este valor SHA-1:

```
1 $ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
2 object 1a410efbd13591db07496601ebc7a059dd55cfe9
3 type commit
4 tag v1.1
5 tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700
6
7 test tag
```

Observe que a entrada do objeto aponta para o valor SHA-1 do commit que você taggeou. Também observe que ele não precisa apontar para um commit; você pode taggear qualquer objeto Git. No código-fonte Git, por exemplo, o mantenedor adicionou sua chave pública GPG como um objeto blob e depois taggeou ele. Você pode ver a chave pública, executando

```
1 $ git cat-file blob junio-gpg-pub
```

no repositório de código-fonte Git. O repositório do kernel Linux também tem um objeto tag que não aponta para um commit — a primeira tag criada aponta para a árvore inicial da importação do código fonte.

Remotos

O terceiro tipo de referência que você vai ver é uma referência remota. Se você adicionar um remoto e fazer um push para ele, Git armazena o valor de seu último push para esse remoto para cada branch no diretório `refs/remotes`. Por exemplo, você pode adicionar um remoto chamado `origin` e fazer um push do seu branch `master` nele:

```

1 $ git remote add origin git@github.com:schacon/simplegit-progit.git
2 $ git push origin master
3 Counting objects: 11, done.
4 Compressing objects: 100% (5/5), done.
5 Writing objects: 100% (7/7), 716 bytes, done.
6 Total 7 (delta 2), reused 4 (delta 1)
7 To git@github.com:schacon/simplegit-progit.git
8    a11bef0..ca82a6d  master -> master

```

Então, você pode ver como era o branch master no remoto origin da última vez que você se comunicou com o servidor, verificando o arquivo refs/remotes/origin/master:

```

1 $ cat .git/refs/remotes/origin/master
2 ca82a6dff817ec66f44342007202690a93763949

```

Referências remotas diferem dos branches (referências refs/heads), principalmente no sentido de que não pode ser feito o checkout delas. Git move elas como indicadores para o último estado conhecido de onde os branches estavam nesses servidores.

9.4 Git Internamente - Packfiles

Packfiles

Vamos voltar para o banco de dados de objetos do seu repositório de testes Git. Neste momento, você tem 11 objetos — 4 blobs, 3 árvores, 3 commits, e 1 tag:

```

1 $ find .git/objects -type f
2 .git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
3 .git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
4 .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
5 .git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
6 .git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
7 .git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
8 .git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
9 .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
10 .git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
11 .git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
12 .git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1

```

Git comprime o conteúdo desses arquivos com zlib, então todos estes arquivos ocupam coletivamente apenas 925 bytes. Você vai adicionar um conteúdo um pouco maior no repositório para demonstrar uma característica interessante do Git. Adicione o arquivo repo.rb da biblioteca Grit em que você trabalhou anteriormente — trata-se de um arquivo de código fonte de 12K:

```

1 $ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb
2 $ git add repo.rb
3 $ git commit -m 'added repo.rb'
4 [master 484a592] added repo.rb
5 3 files changed, 459 insertions(+), 2 deletions(-)
6 delete mode 100644 bak/test.txt
7 create mode 100644 repo.rb
8 rewrite test.txt (100%)

```

Se você olhar para a árvore resultante, você pode ver o valor SHA-1 que o arquivo `repo.rb` tem no objeto blob:

```

1 $ git cat-file -p master^{tree}
2 100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
3 100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e      repo.rb
4 100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt

```

Você pode então usar `git cat-file` para ver o quão grande esse objeto é:

```

1 $ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
2 12898

```

Agora, modifique o arquivo um pouco, e veja o que acontece:

```

1 $ echo '# testing' >> repo.rb
2 $ git commit -am 'modified repo a bit'
3 [master ab1afef] modified repo a bit
4 1 files changed, 1 insertions(+), 0 deletions(-)

```

Verifique a árvore criada por este commit, e você verá algo interessante:

```

1 $ git cat-file -p master^{tree}
2 100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
3 100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
4 100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt

```

O blob agora é um blob diferente, o que significa que, embora você tenha adicionado apenas uma única linha ao final de um arquivo de 400 linhas, Git armazenou esse novo conteúdo como um objeto completamente novo:

```
1 $ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
2 12908
```

Você tem dois objetos de 12K quase idênticos em seu disco. Não seria bom se o Git pudesse armazenar um deles na íntegra, mas, o segundo objeto apenas como o delta entre ele e o primeiro?

Acontece que ele pode. O formato inicial em que Git salva objetos em disco é chamado de formato de objeto solto (loose object format). No entanto, ocasionalmente Git empacota vários desses objetos em um único arquivo binário chamado de packfile, a fim de economizar espaço e ser mais eficiente. Git faz isso, se você tem muitos objetos soltos, se você executar o comando `git gc` manualmente, ou se você fizer push para um servidor remoto. Para ver o que acontece, você pode manualmente pedir ao Git para arrumar os objetos chamando o comando `git gc`:

```
1 $ git gc
2 Counting objects: 17, done.
3 Delta compression using 2 threads.
4 Compressing objects: 100% (13/13), done.
5 Writing objects: 100% (17/17), done.
6 Total 17 (delta 1), reused 10 (delta 0)
```

Se você olhar em seu diretório de objetos, você vai descobrir que a maioria de seus objetos sumiram, e um novo par de arquivos apareceu:

```
1 $ find .git/objects -type f
2 .git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
3 .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
4 .git/objects/info/packs
5 .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
6 .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Os objetos que permanecem são os blobs que não são apontados por qualquer commit — neste caso, os blobs de exemplo “what is up, doc?” e o exemplo “test content” que você criou anteriormente. Como você nunca adicionou eles a qualquer commit, eles são considerados pendentes e não são embalados em sua nova packfile.

Os outros arquivos são o seu novo packfile e um índice. O packfile é um arquivo único contendo o conteúdo de todos os objetos que foram removidos do seu sistema de arquivos. O índice é um arquivo que contém offsets deste packfile assim você pode rapidamente buscar um objeto específico. O que é legal é que, embora os objetos em disco antes de executar o `gc` tinham coletivamente cerca de 12K de tamanho, o packfile novo tem apenas 6K. Você reduziu a utilização do disco pela metade empacotando seus objetos.

Como o Git faz isso? Quando Git empacota objetos, ele procura por arquivos que são nomeados e dimensionados de forma semelhante, e armazena apenas os deltas de uma versão do arquivo para a próxima. Você pode olhar dentro do packfile e ver o que o Git fez para economizar espaço. O comando `plumbing git verify-pack` permite que você veja o que foi empacotado:


```

1 $ git verify-pack -v \
2   .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
3 0155eb4229851634a0f03eb265b69f5a2d56f341 tree    71 76 5400
4 05408d195263d853f09dca71d55116663690c27c blob    12908 3478 874
5 09f01cea547666f58d6a8d809583841a7c6f0130 tree    106 107 5086
6 1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
7 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob    10 19 5381
8 3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree    101 105 5211
9 484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
10 83baae61804e65cc73a7201a7252750c76066a30 blob    10 19 5362
11 9585191f37f7b0fb9444f35a9bf50de191beadc2 tag      136 127 5476
12 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob    7 18 5193 1 \
13   05408d195263d853f09dca71d55116663690c27c
14 ab1afe80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
15 cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
16 d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree    36 46 5316
17 e3f094f522629ae358806b17daf78246c27c007b blob    1486 734 4352
18 f8f51d7d8a1760462eca26eebafde32087499533 tree    106 107 749
19 fa49b077972391ad58037050f2a75f74e3671e92 blob    9 18 856
20 fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
21 chain length = 1: 1 object
22 pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok

```

Aqui, o blob 9bc1d, que se você se lembrar foi a primeira versão do seu arquivo `repo.rb`, faz referência ao blob 05408, que foi a segunda versão do arquivo. A terceira coluna da saída é o tamanho do objeto no pacote, assim você pode ver que 05408 ocupa 12K do arquivo, mas que 9bc1d ocupa apenas 7 bytes. O que também é interessante é que a segunda versão do arquivo é a que é armazenada intacta, enquanto que a versão original é armazenada como um delta — isso porque é mais provável a necessidade de acesso mais rápido para a versão mais recente do arquivo.

A coisa realmente interessante sobre isso é que ele pode ser reembalado a qualquer momento. Git irá ocasionalmente reembalar seu banco de dados automaticamente, sempre tentando economizar mais espaço. Você pode também manualmente reembalar a qualquer momento executando `git gc`.

9.5 Git Internamente - O Refspec

O Refspec

Ao longo deste livro, você usou um simples mapeamento de branches remotos para referências locais; mas eles podem ser mais complexos. Suponha que você adicione um remoto como este:

```
1 $ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Ele adiciona uma seção em seu arquivo `.git/config`, especificando o nome do remoto (`origin`), a URL do repositório remoto, e o refspec a ser buscado:

```
1 [remote "origin"]
2     url = git@github.com:schacon/simplegit-progit.git
3     fetch = +refs/heads/*:refs/remotes/origin/*
```

O formato do refspec é um `+` opcional, seguido por `<src>:<dst>`, onde `<src>` é o padrão para referências no lado remoto e `<dst>` é onde essas referências serão escritas localmente. O `+` diz ao Git para atualizar a referência, mesmo que não seja um fast-forward.

No caso padrão que é automaticamente escrito por um comando `git remote add`, Git busca todas as referências em `refs/heads/` no servidor e grava-os em `refs/remotes/origin/` localmente. Então, se há um branch `master` no servidor, você pode acessar o log desse branch localmente através de

```
1 $ git log origin/master
2 $ git log remotes/origin/master
3 $ git log refs/remotes/origin/master
```

Eles são todos equivalentes, porque Git expande cada um deles em `refs/remotes/origin/master`.

Se você quiser que o Git só faça o pull do branch `master` toda vez, e não qualquer outro branch do servidor remoto, você pode alterar a linha `fetch` para

```
1 fetch = +refs/heads/master:refs/remotes/origin/master
```

Este é apenas o refspec padrão do `git fetch` para esse remoto. Se você quiser fazer algo apenas uma vez, você pode especificar o refspec na linha de comando também. Para fazer o pull do branch `master` do remoto até `origin/mymaster` localmente, você pode executar

```
1 $ git fetch origin master:refs/remotes/origin/mymaster
```

Você também pode especificar múltiplos refspecs. Na linha de comando, você pode fazer pull de vários branches assim:

```

1 $ git fetch origin master:refs/remotes/origin/mymaster \
2   topic:refs/remotes/origin/topic
3 From git@github.com:schacon/simplegit
4 ! [rejected]      master    -> origin/mymaster  (non fast forward)
5 * [new branch]   topic     -> origin/topic

```

Neste caso, o pull do branch master foi rejeitado, porque não era uma referência fast-forward. Você pode evitar isso especificando o + na frente do refspec.

Você também pode especificar múltiplos refsspecs em seu arquivo de configuração. Se você quer sempre buscar os branches master e experiment, adicione duas linhas:

```

1 [remote "origin"]
2   url = git@github.com:schacon/simplegit-progit.git
3   fetch = +refs/heads/master:refs/remotes/origin/master
4   fetch = +refs/heads/experiment:refs/remotes/origin/experiment

```

Você não pode usar globs parciais no padrão, então isto seria inválido:

```

1 fetch = +refs/heads/qa*:refs/remotes/origin/qa*

```

No entanto, você pode usar namespacing para realizar algo assim. Se você tem uma equipe de QA que faz push de uma série de branches, e você deseja obter o branch master e qualquer um dos branches da equipe de QA, mas nada mais, você pode usar uma seção de configuração como esta:

```

1 [remote "origin"]
2   url = git@github.com:schacon/simplegit-progit.git
3   fetch = +refs/heads/master:refs/remotes/origin/master
4   fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*

```

Se você tem um fluxo de trabalho complexo que tem uma equipe de QA fazendo push de branches, desenvolvedores fazendo push de branches, e equipes de integração fazendo push e colaborando em branches remotos, você pode nomeá-los (namespace) facilmente desta forma.

Fazendo Push de Refspecs

É legal que você possa buscar referências nomeadas dessa maneira, mas como é que a equipe de QA obtêm os seus branches em um namespace qa/? Você consegue fazer isso utilizando refsspecs para fazer o push.

Se a equipe de QA quer fazer push de seu branch master em qa/master no servidor remoto, eles podem executar

```
1 $ git push origin master:refs/heads/qa/master
```

Se eles querem que o Git faça isso automaticamente toda vez que executar `git push origin`, eles podem adicionar o valor `push` ao seu arquivo de configuração:

```
[remote "origin"] url = git@github.com:schacon/simplegit-progit.git fetch = +refs/heads/:refs/remotes/origin/  
push = refs/heads/master:refs/heads/qa/master
```

Novamente, isso vai fazer com que `git push origin` faça um `push` do branch `master` local para o branch remoto `qa/master` por padrão.

Deletando Referencias

Você também pode usar o `refspec` para apagar referências do servidor remoto executando algo como isto:

```
1 $ git push origin :topic
```

Já que o `refspec` é `<src>: <dst>`, ao remover `<src>`, basicamente diz para enviar nada para o branch tópico no remoto, o que o exclui.

9.6 Git Internamente - Protocolos de Transferência

Protocolos de Transferência

Git pode transferir dados entre dois repositórios de duas maneiras principais: através de HTTP e através dos protocolos chamados inteligentes: `file://`, `ssh://` e `git://`. Esta seção irá mostrar rapidamente como esses dois principais protocolos operam.

O Protocolo Burro

HTTP é muitas vezes referido como o protocolo burro, porque não requer código Git específico do lado servidor durante o processo de transporte. O processo de `fetch` é uma série de solicitações GET, onde o cliente pode assumir o layout do repositório Git no servidor. Vamos ver o processo `http-fetch` para obter a biblioteca `simplegit`:

```
1 $ git clone http://github.com/schacon/simplegit-progit.git
```

A primeira coisa que este comando faz é obter o arquivo `info/refs`. Este arquivo é escrito pelo comando `update-server-info`, é por isso que você precisa ativar ele como um `hook post-receive` para que o transporte HTTP funcione corretamente:

```
1 => GET info/refs
2 ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Agora você tem uma lista de referências remotas e os SHAs. Em seguida, você verifica qual é a referência HEAD para que você saiba o que deve ser obtido (check out) quando você terminar:

```
1 => GET HEAD
2 ref: refs/heads/master
```

Você precisa fazer o check out do branch master quando você tiver concluído o processo. Neste momento, você está pronto para iniciar o processo de “caminhada”. Como o seu ponto de partida é o objeto commit ca82a6 que você viu no arquivo info/refs, você começa obtendo ele:

```
1 => GET objects/ca/82a6dff817ec66f44342007202690a93763949
2 (179 bytes of binary data)
```

Você obtém um objeto — este objeto é um “loose format” no servidor, e você o obteve a partir de uma conexão HTTP GET estática. Você pode descompactá-lo, retirar o cabeçalho, e ver o conteúdo do commit:

```
1 $ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
2 tree cfd3bfb379e4f8dba8717dee55aab78aef7f4daf
3 parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
4 author Scott Chacon <schacon@gmail.com> 1205815931 -0700
5 committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
6
7 changed the version number
```

A seguir, você tem mais dois objetos para obter — cfd3b, que a árvore de conteúdo do commit que acabamos de obter referencia, e 085bb3, que é o commit pai:

```
1 => GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
2 (179 bytes of data)
```

Isso lhe dá o seu próximo objeto commit. Obtenha o objeto árvore:

```
1 => GET objects/cf/da3bfb379e4f8dba8717dee55aab78aef7f4daf
2 (404 - Not Found)
```

Oops — parece que esse objeto árvore não está em “loose format” no servidor, então você recebe um erro 404. Há algumas razões para isso — o objeto pode estar em um repositório alternativo, ou poderia estar em um packfile neste repositório. Git verifica a existência de quaisquer alternativas listadas primeiro:

```
1 => GET objects/info/http-alternates
2 (empty file)
```

Se isso retornar uma lista de URLs alternativas, Git verifica a existência de “loose files” e packfiles lá — este é um mecanismo legal para projetos que são forks de um outro para compartilhar objetos no disco. No entanto, como não há substitutos listados neste caso, o objeto deve estar em um packfile. Para saber quais packfiles estão disponíveis neste servidor, você precisa obter o arquivo `objects/info/packs`, que contém uma lista deles (também gerado pelo `update-server-info`):

```
1 => GET objects/info/packs
2 P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Há apenas um packfile no servidor, então o seu objeto deve estar lá, mas você vai verificar o arquivo de índice para ter certeza. Isso também é útil se você tem vários packfiles no servidor, assim você pode ver qual packfile contém o objeto que você precisa:

```
1 => GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
2 (4k of binary data)
```

Agora que você tem o índice do packfile, você pode ver se o seu objeto está nele — porque o índice lista os SHAs dos objetos contidos no packfile e os deslocamentos (offsets) desses objetos. Seu objeto está lá, então vá em frente e obtenha todo o packfile:

```
1 => GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
2 (13k of binary data)
```

Você tem o seu objeto árvore, então você continua navegando em seus commits. Eles todos estão dentro do packfile que você acabou de baixar, então você não precisa fazer mais solicitações para o servidor. Git faz o check out de uma cópia de trabalho do branch `master`, que foi apontada pela referência `HEAD` que você baixou no início.

Toda a saída deste processo é parecida com isto:

```

1 $ git clone http://github.com/schacon/simplegit-progit.git
2 Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
3 got ca82a6dff817ec66f44342007202690a93763949
4 walk ca82a6dff817ec66f44342007202690a93763949
5 got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
6 Getting alternates list for http://github.com/schacon/simplegit-progit.git
7 Getting pack list for http://github.com/schacon/simplegit-progit.git
8 Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
9 Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
10 which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
11 walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
12 walk a11bef06a3f659402fe7563abf99ad00de2209e6

```

O Protocolo Inteligente

O método HTTP é simples, mas um pouco ineficiente. Usar protocolos inteligentes é um método mais comum de transferência de dados. Estes protocolos têm um processo no sistema remoto que é inteligente em relação ao Git — ele pode ler dados locais e descobrir o que o cliente tem ou precisa e gera dados personalizados para ele. Existem dois conjuntos de processos de transferência de dados: um par para enviar dados (upload) e um par para download de dados.

Fazendo Upload de Dados

Para fazer upload de dados para um processo remoto, Git usa os processos `send-pack` e `receive-pack`. O processo `send-pack` é executado no cliente e se conecta a um processo `receive-pack` no lado remoto.

Por exemplo, digamos que você execute `git push origin master` em seu projeto, e `origin` é definido como uma URL que usa o protocolo SSH. Git executa o processo `send-pack`, que inicia uma conexão SSH ao seu servidor. Ele tenta executar um comando no servidor remoto através de uma chamada SSH que é parecida com isto:

```

1 $ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
2 005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-r\
3 efs
4 003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
5 0000

```

O comando `git-receive-pack` imediatamente responde com uma linha para cada referência que tem atualmente — neste caso, apenas o branch `master` e seu SHA. A primeira linha também tem uma lista de recursos do servidor (aqui, `report-status` e `delete-refs`).

Cada linha se inicia com um valor hexadecimal de 4 bytes especificando quão longo o resto da linha é. Sua primeira linha começa com `005B`, que é 91 em hexadecimal, o que significa que 91 bytes permanecem nessa linha. A próxima linha começa com `003e`, que é 62, então você leu os 62 bytes restantes. A próxima linha é `0000`, ou seja, o servidor terminou com a sua lista de referências.

Agora que ele sabe o estado do servidor, o seu processo `send-pack` determina que commits ele tem mas que o servidor não tem. Para cada referência que este push irá atualizar, o processo `send-pack` diz essa informação ao processo `receive-pack`. Por exemplo, se você está atualizando o branch `master` e está adicionando um branch `experiment`, a resposta do `send-pack` pode ser parecida com esta:

```
1 0085ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83\
2 b3ff6 refs/heads/master report-status
3 0067000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf6\
4 3e8d refs/heads/experiment
5 0000
```

O valor SHA-1 de todos os '0's significa que nada estava lá antes — porque você está adicionando a referência de `experiment`. Se você estivesse apagando uma referência, você veria o oposto: tudo '0' no lado direito.

Git envia uma linha para cada referência que você está atualizando com o SHA antigo, o SHA novo, e a referência que está sendo atualizada. A primeira linha também tem as capacidades do cliente. Em seguida, o cliente faz upload de um packfile com todos os objetos que o servidor não tem ainda. Finalmente, o servidor responde com uma indicação de sucesso (ou falha):

```
1 000Aunpack ok
```

Fazendo Download de Dados

Quando você baixa os dados, os processos `fetch-pack` e `upload-pack` são usados. O cliente inicia um processo `fetch-pack` que se conecta a um processo `upload-pack` no lado remoto para negociar os dados que serão transferidos para a máquina local.

Existem diferentes maneiras de iniciar o processo `upload-pack` no repositório remoto. Você pode fazê-lo via SSH da mesma forma que o processo `receive-pack`. Você também pode iniciar o processo, através do daemon Git, que escuta em um servidor na porta 9418 por padrão. O processo `fetch-pack` envia dados que se parecem com isso para o servidor após se conectar:

```
1 003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

Ele começa com os 4 bytes que especificam a quantidade de dados enviados, seguido pelo comando a ser executado, seguido por um byte nulo e, em seguida, o hostname do servidor seguido por um ultimo byte nulo. O daemon Git verifica que o comando pode ser executado e que o repositório existe e tem permissões públicas. Se tudo estiver certo, ele aciona o processo `upload-pack` e envia o pedido para ele.

Se você estiver fazendo o fetch através de SSH, `fetch-pack` executa algo como isto:


```
1 $ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

Em todo caso, depois que fetch-pack conectar, upload-pack devolve algo como isto:

```
1 0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
2   side-band side-band-64k ofs-delta shallow no-progress include-tag
3 003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
4 003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
5 0000
```

Isto é muito semelhante a resposta de receive-pack, mas as funcionalidades são diferentes. Além disso, ele envia de volta a referência de HEAD para que o cliente saiba o que deve ser verificado (check out) se este for um clone.

Neste momento, o processo fetch-pack verifica quais objetos ele possui e responde com os objetos de que necessita através do envio de “want” e do SHA que quer. Ele envia todos os objetos que ele já tem com “have” e também o SHA. No final da lista, ele escreve “done” para iniciar o processo upload-pack para começar a enviar o packfile dos dados que ele precisa:

```
1 0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
2 0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
3 0000
4 0009done
```

Isto é um caso muito simples dos protocolos de transferência. Em casos mais complexos, o cliente suporta funcionalidades multi_ack ou side-band; mas este exemplo mostra o funcionamento de upload e download básico utilizado pelos processos de protocolo inteligentes.

9.7 Git Internamente - Manutenção e Recuperação de Dados

Manutenção e Recuperação de Dados

Ocasionalmente, você pode ter que fazer alguma limpeza — compactar um repositório, limpar um repositório importado, ou recuperar trabalhos perdidos. Esta seção irá mostrar alguns desses cenários.

Manutenção

Ocasionalmente, Git automaticamente executa um comando chamado “auto gc”. Na maioria das vezes, este comando não faz nada. No entanto, se houverem muitos objetos soltos (loose objects) (objetos que não estejam em um packfile) ou muitos packfiles, Git executa um verdadeiro comando `git gc`. O `gc` significa garbage collect (coleta de lixo), e o comando faz uma série de coisas: ele reúne todos os objetos soltos e os coloca em packfiles, consolida packfiles em um packfile maior, e remove objetos que não estejam ao alcance de qualquer commit e tem poucos meses de idade.

Você pode executar `auto gc` manualmente da seguinte forma:

```
1 $ git gc --auto
```

Mais uma vez, isso geralmente não faz nada. Você deve ter cerca de 7.000 objetos soltos ou mais de 50 packfiles para que o Git execute um comando `gc` real. Você pode modificar esses limites com as opções de configuração `gc.auto` e `gc.autopacklimit`, respectivamente.

A outra coisa que `gc` irá fazer é arrumar suas referências em um único arquivo. Suponha que seu repositório contém os seguintes branches e tags:

```
1 $ find .git/refs -type f
2 .git/refs/heads/experiment
3 .git/refs/heads/master
4 .git/refs/tags/v1.0
5 .git/refs/tags/v1.1
```

Se você executar `git gc`, você não terá mais esses arquivos no diretório `refs`. Git irá movê-los para um arquivo chamado `.git/packed-refs` que se parece com isto:

```
1 $ cat .git/packed-refs
2 # pack-refs with: peeled
3 cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
4 ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
5 cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
6 9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
7 ^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Se você atualizar uma referência, Git não edita esse arquivo, mas em vez disso, escreve um novo arquivo em `refs/heads`. Para obter o SHA apropriado para uma dada referência, Git checa a referência no diretório `refs` e então verifica o arquivo `packed-refs` como um último recurso. No entanto, se você não conseguir encontrar uma referência no diretório `refs`, ela está provavelmente em seu arquivo `packed-refs`.

Observe a última linha do arquivo, que começa com um `^`. Isto significa que a tag diretamente acima é uma tag anotada (annotated tag) e que a linha é o commit que a tag anotada aponta.

Recuperação de Dados

Em algum ponto de sua jornada Git, você pode acidentalmente perder um commit. Geralmente, isso acontece porque você forçou a remoção (force-delete) de um branch que tinha informações nele, e depois se deu conta de que precisava do branch; ou você resetou (hard-reset) um branch, abandonando commits com informações importantes. Assumindo que isso aconteceu, como você pode obter o seu commit de volta?

Aqui está um exemplo que reseta (hard-resets) o branch master no seu repositório de teste para um commit antigo e depois recupera os commits perdidos. Primeiro, vamos rever onde seu repositório está neste momento:

```
1 $ git log --pretty=oneline
2 ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
3 484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
4 1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
5 cac0cab538b970a37ea1e769cbbde608743bc96d second commit
6 fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Agora, mova o branch master de volta para o commit do meio:

```
1 $ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
2 HEAD is now at 1a410ef third commit
3 $ git log --pretty=oneline
4 1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
5 cac0cab538b970a37ea1e769cbbde608743bc96d second commit
6 fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Você efetivamente perdeu os dois primeiros commits — você não tem um branch de onde os commits são alcançáveis. Você precisa encontrar o SHA do último commit e em seguida, adicionar um branch que aponta para ele. O truque é encontrar o SHA do commit mais recente — você não o memorizou, certo?

Muitas vezes, a maneira mais rápida é usar uma ferramenta chamada git reflog. Quando você está trabalhando, Git silenciosamente registra onde está o HEAD cada vez que você mudá-lo. Cada vez que você fizer um commit ou alterar branches, o reflog é atualizado. O reflog também é atualizado pelo comando git update-ref, o que é mais um motivo para usá-lo em vez de apenas escrever o valor SHA em seus arquivos ref, como abordado anteriormente na seção “Referências Git” deste capítulo. Você pode ver onde você está, a qualquer momento, executando git reflog:

```

1 $ git reflog
2 1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
3 ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD

```

Aqui podemos ver os dois commits que obtemos com check out, no entanto, não há muita informação aqui. Para ver a mesma informação de uma forma muito mais útil, podemos executar `git log -g`, que vai lhe dar uma saída de log normal do seu reflog.

```

1 $ git log -g
2 commit 1a410efbd13591db07496601ebc7a059dd55cfe9
3 Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
4 Reflog message: updating HEAD
5 Author: Scott Chacon <schacon@gmail.com>
6 Date:   Fri May 22 18:22:37 2009 -0700
7
8     third commit
9
10 commit ab1afef80fac8e34258ff41fc1b867c702daa24b
11 Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
12 Reflog message: updating HEAD
13 Author: Scott Chacon <schacon@gmail.com>
14 Date:   Fri May 22 18:15:24 2009 -0700
15
16     modified repo a bit

```

Parece que o commit de baixo é o que você perdeu, então você pode recuperá-lo através da criação de um novo branch neste commit. Por exemplo, você pode criar um branch chamado `recover-branch` naquele commit (`ab1afef`):

```

1 $ git branch recover-branch ab1afef
2 $ git log --pretty=oneline recover-branch
3 ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
4 484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
5 1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
6 cac0cab538b970a37ea1e769cbbde608743bc96d second commit
7 fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit

```

Agora você tem um branch chamado `recover-branch` que é onde o seu branch `master` costumava estar, fazendo os dois primeiros commits acessíveis novamente. Em seguida, suponha que a sua perda por algum motivo não está no reflog — você pode simular isso ao remover `recover-branch` e apagar o reflog. Agora os dois primeiros commits não são acessíveis por qualquer coisa:

```
1 $ git branch -D recover-branch
2 $ rm -Rf .git/logs/
```

Como os dados do reflog são mantidos no diretório `.git/logs/`, você efetivamente não tem reflog. Como você pode recuperar aquele commit agora? Uma maneira é usar o utilitário `git fsck`, que verifica a integridade de seu banco de dados. Se você executá-lo com a opção `--full`, ele mostra todos os objetos que não são apontados por outro objeto:

```
1 $ git fsck --full
2 dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
3 dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
4 dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
5 dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

Neste caso, você pode ver o seu commit desaparecido após o commit pendente (dangling). Você pode recuperá-lo da mesma forma, adicionando um branch que aponta para seu SHA.

Removendo Objetos

Há um monte de coisas boas em relação ao Git, mas um recurso que pode causar problemas é o fato de que o `git clone` baixa todo o histórico do projeto, incluindo todas as versões de cada arquivo. Isso é bom se a coisa toda for código fonte, porque Git é altamente otimizado para comprimir os dados de forma eficiente. No entanto, se alguém, em algum momento adicionou um arquivo enorme, todo clone será forçado a baixar o arquivo grande, mesmo que ele tenha sido retirado do projeto no commit seguinte. Como ele é acessível a partir do histórico, ele sempre estará lá.

Isso pode ser um grande problema quando você está convertendo repositórios do Subversion ou Perforce para Git. Como você não baixa todo o histórico nesses sistemas, este tipo de adição traz poucas consequências. Se você fez uma importação de outro sistema ou descobriu que seu repositório é muito maior do que deveria ser, eis aqui como você pode encontrar e remover objetos grandes.

Esteja avisado: esta técnica é destrutiva para o seu histórico de commits. Ele reescreve a cada objeto commit da primeira árvore que você tem que modificar para remover uma referência de arquivo grande. Se você fizer isso, imediatamente após uma importação, antes que alguém tenha começado usar o commit, você ficará bem — caso contrário, você tem que notificar todos os contribuidores para que eles façam um rebase do trabalho deles em seus novos commits.

Para demonstrar, você vai adicionar um arquivo grande em seu repositório, removê-lo no próximo commit, encontrá-lo e removê-lo permanentemente a partir do repositório. Primeiro, adicione um objeto grande no seu histórico:

```
1 $ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
2 $ git add git.tbz2
3 $ git commit -am 'added git tarball'
4 [master 6df7640] added git tarball
5 1 files changed, 0 insertions(+), 0 deletions(-)
6 create mode 100644 git.tbz2
```

Oops - você não queria adicionar um tarball enorme no seu projeto. Melhor se livrar dele:

```
1 $ git rm git.tbz2
2 rm 'git.tbz2'
3 $ git commit -m 'oops - removed large tarball'
4 [master da3f30d] oops - removed large tarball
5 1 files changed, 0 insertions(+), 0 deletions(-)
6 delete mode 100644 git.tbz2
```

Agora, use gc no seu banco de dados e veja quanto espaço você está usando:

```
1 $ git gc
2 Counting objects: 21, done.
3 Delta compression using 2 threads.
4 Compressing objects: 100% (16/16), done.
5 Writing objects: 100% (21/21), done.
6 Total 21 (delta 3), reused 15 (delta 1)
```

Você pode executar o comando count-objects para ver rapidamente quanto espaço você está usando:

```
1 $ git count-objects -v
2 count: 4
3 size: 16
4 in-pack: 21
5 packs: 1
6 size-pack: 2016
7 prune-packable: 0
8 garbage: 0
```

A entrada size-pack é do tamanho de seus packfiles em kilobytes, então você está usando 2MB. Antes do último commit, você estava usando quase 2K — claramente, removendo o arquivo do commit anterior não remove-o de seu histórico. Toda vez que alguém clonar este repositório, eles

vão ter que clonar os 2MB para obter este projeto, porque você acidentalmente acrescentou um arquivo grande. Vamos nos livrar dele.

Primeiro você tem que encontrá-lo. Neste caso, você já sabe qual é o arquivo. Mas suponha que você não saiba; como você identifica o arquivo ou arquivos que estão ocupando tanto espaço? Se você executar `git gc`, todos os objetos estarão em um packfile; você pode identificar os objetos grandes, executando outro comando encanamento (plumbing) chamado `git verify-pack` e classificar pelo terceiro campo da saída, que é o tamanho do arquivo. Você também pode direcionar a saída (pipe) através do comando `tail` porque você está interessado apenas nos últimos poucos arquivos maiores:

```
1 $ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail \
2 -3
3 e3f094f522629ae358806b17daf78246c27c007b blob      1486 734 4667
4 05408d195263d853f09dca71d55116663690c27c blob      12908 3478 1189
5 7a9eb2fba2b1811321254ac360970fc169ba2330 blob      2056716 2056872 5401
```

O objeto grande está na parte inferior: 2MB. Para saber qual é o arquivo, você vai usar o comando `rev-list`, que você usou brevemente no Capítulo 7. Se você passar `--objects` para `rev-list`, ele lista todos os SHAs dos commits e também os SHAs dos blob com os caminhos de arquivos (paths) associados a eles. Você pode usar isso para encontrar o nome do blob:

```
1 $ git rev-list --objects --all | grep 7a9eb2fb
2 7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Agora, você precisa remover o arquivo de todas as árvores em que ele estiver. Você pode facilmente ver quais commits modificaram este arquivo:

```
1 $ git log --pretty=oneline --branches -- git.tbz2
2 da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
3 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Você deve reescrever todos os commits desde 6df76 para remover completamente este arquivo do seu histórico Git. Para fazer isso, você usa `filter-branch`, que você já usou no capítulo 6:

```
1 $ git filter-branch --index-filter \
2     'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
3 Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
4 Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
5 Ref 'refs/heads/master' was rewritten
```

A opção `--index-filter` é semelhante a opção `--tree-filter` utilizada no *Capítulo 6*, exceto que em vez de passar um comando que modifica os arquivos que você fez check-out no disco, você está modificando sua área de seleção (staging area) ou índice. Em vez de remover um arquivo específico com algo como `rm file`, você tem que removê-lo com `git rm --cached` — você deve removê-lo do índice, não do disco. A razão para fazê-lo desta maneira é a velocidade — porque o Git não precisa fazer o check out de cada revisão no disco antes de executar o seu filtro, o processo pode ser muito mais rápido. Você pode realizar a mesma tarefa com `--tree-filter` se você quiser. A opção `--ignore-unmatch` do `git rm` diz a ele para não mostrar erros se o padrão que você está tentando remover não estiver lá. Finalmente, você pede a `filter-branch` para reescrever seu histórico apenas a partir do commit `6df7640`, porque você sabe que é onde o problema começou. Caso contrário, ele vai começar desde o início e vai demorar mais tempo desnecessariamente.

Seu histórico já não contém uma referência para o arquivo. No entanto, seu reflog e um novo conjunto de refs que o git adicionou quando você fez o `filter-branch` em `.git/refs/original` ainda não, então você tem que removê-los e, em seguida, fazer um repack do banco de dados. Você precisa se livrar de qualquer coisa que tenha um ponteiro para aqueles commits antigos antes de fazer o repack:

```
1 $ rm -Rf .git/refs/original
2 $ rm -Rf .git/logs/
3 $ git gc
4 Counting objects: 19, done.
5 Delta compression using 2 threads.
6 Compressing objects: 100% (14/14), done.
7 Writing objects: 100% (19/19), done.
8 Total 19 (delta 3), reused 16 (delta 1)
```

Vamos ver quanto espaço você economizou.

```
1 $ git count-objects -v
2 count: 8
3 size: 2040
4 in-pack: 19
5 packs: 1
6 size-pack: 7
7 prune-packable: 0
8 garbage: 0
```

O tamanho do repositório compactado reduziu para 7K, que é muito melhor do que 2MB. Pelo tamanho você pode ver que o grande objeto ainda está em seus objetos soltos, portanto não foi eliminado; mas ele não será transferido em um clone ou push posterior, e isso é o que importa. Se você realmente quiser, você pode remover o objeto completamente executando `git prune --expire`.

9.8 Git Internamente - Resumo

Resumo

Você deve ter uma compreensão muito boa do que Git faz de verdade e, até certo ponto, como ele é implementado. Este capítulo mostrou uma série de comandos de encanamento (plumbing) — comandos que são de nível inferior e mais simples do que os comandos de porcelana (porcelain) que você aprendeu no resto do livro. Compreender como Git funciona em um nível inferior deve torná-lo mais fácil de entender porque ele está fazendo o que está fazendo e também para escrever suas próprias ferramentas e scripts para criar seu próprio fluxo de trabalho.

Git como um sistema de arquivos de conteúdo endereçável é uma ferramenta muito poderosa que você pode usar facilmente como mais do que apenas um VCS. Eu espero que você possa usar seu novo conhecimento do Git para implementar a sua própria aplicação a partir desta tecnologia e se sentir mais confortável usando Git de formas mais avançadas.