Chapter 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).

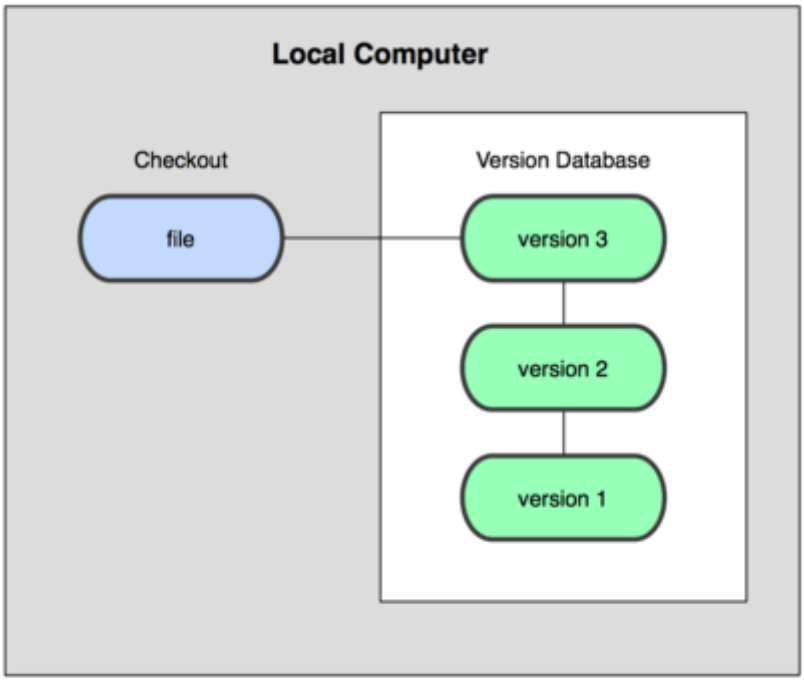

Figura 1-1. Diagrama de controle de versão local.

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.

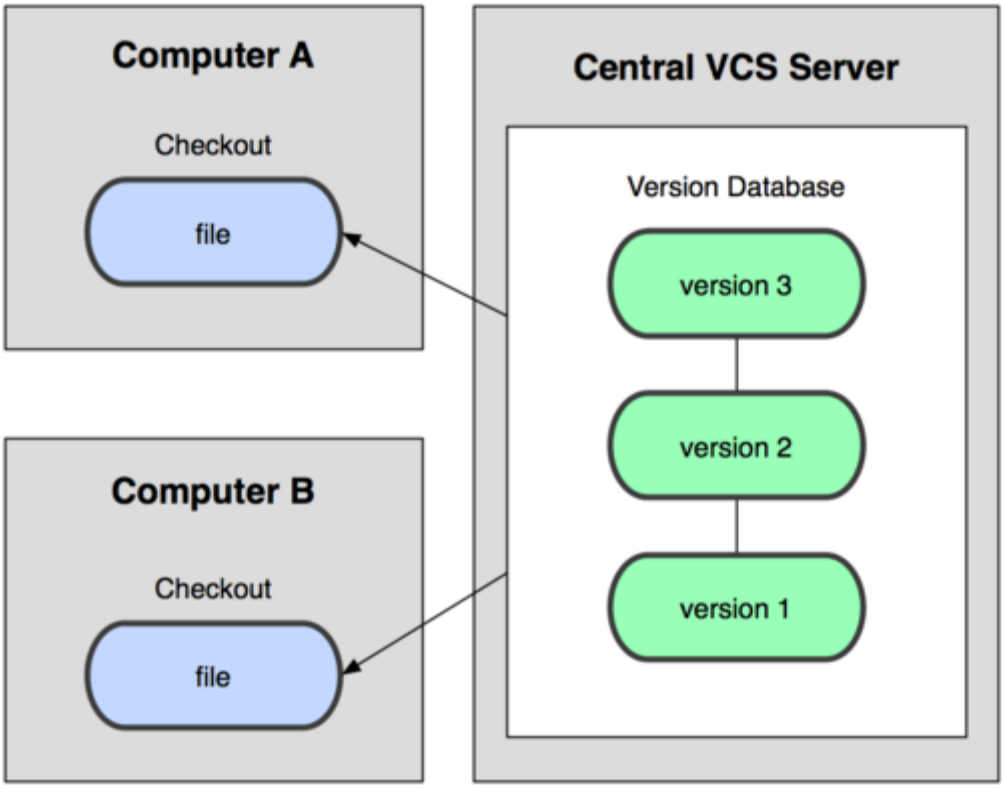

Figura 1-2. Diagrama de Controle de Versão Centralizado.

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 ou 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).

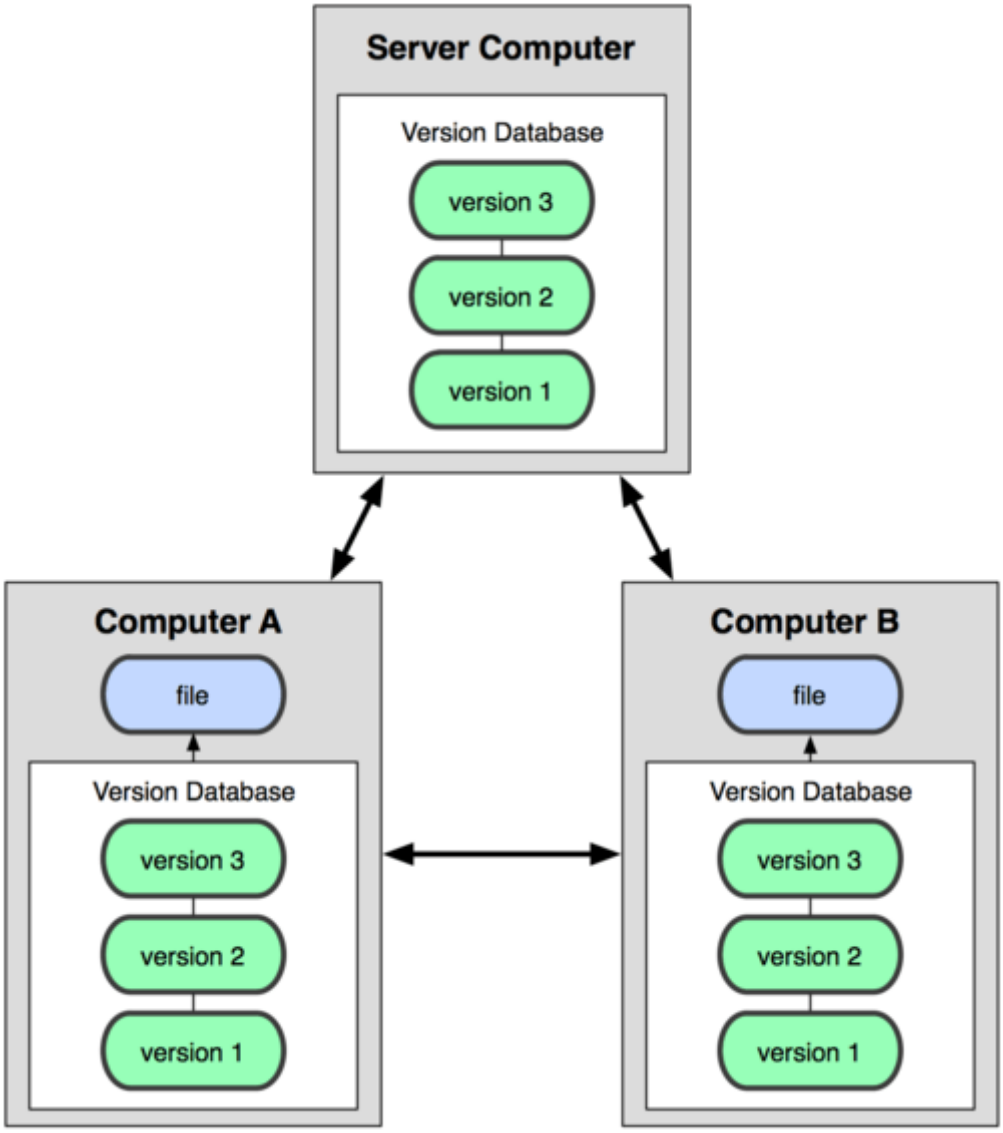

Figura 1-3. Diagrama de Controle de Versão Distribuído.

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.

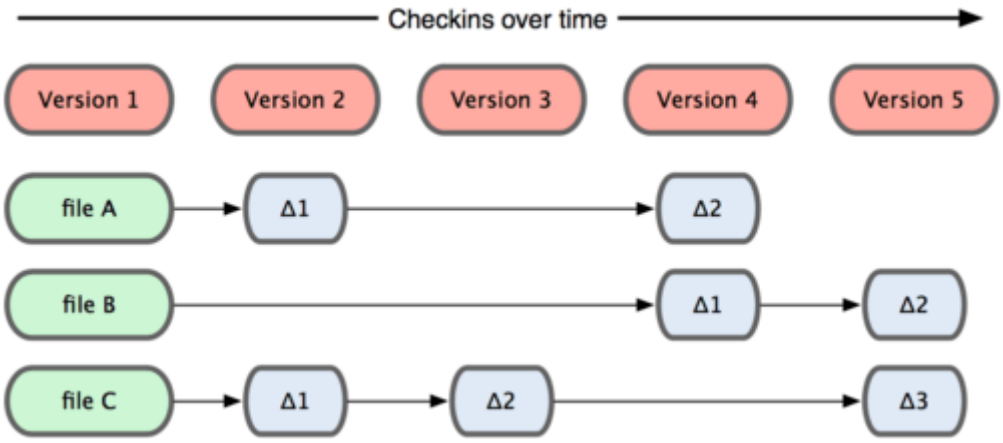

Figura 1-4. Outros sistemas costumam armazenar dados como mudanças em uma versão inicial de cada arquivo.

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.

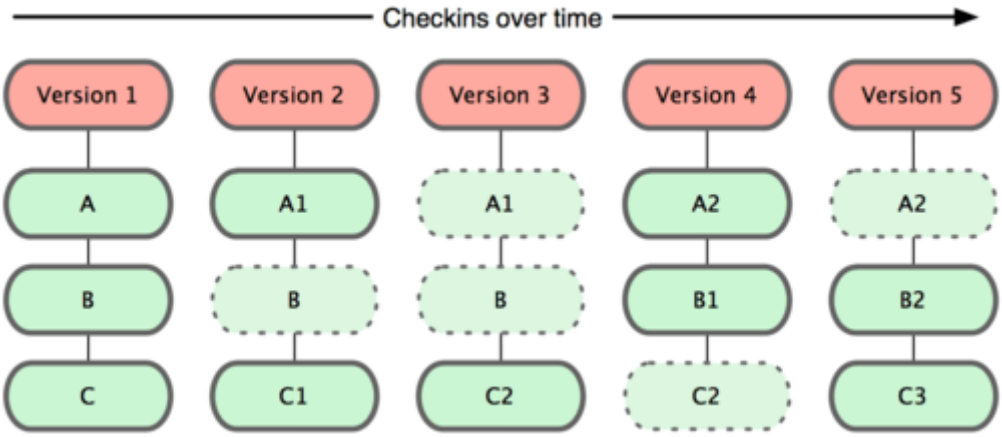

Figura 1-5. Git armazena dados como snapshots do projeto ao longo do tempo.

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:

```
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).

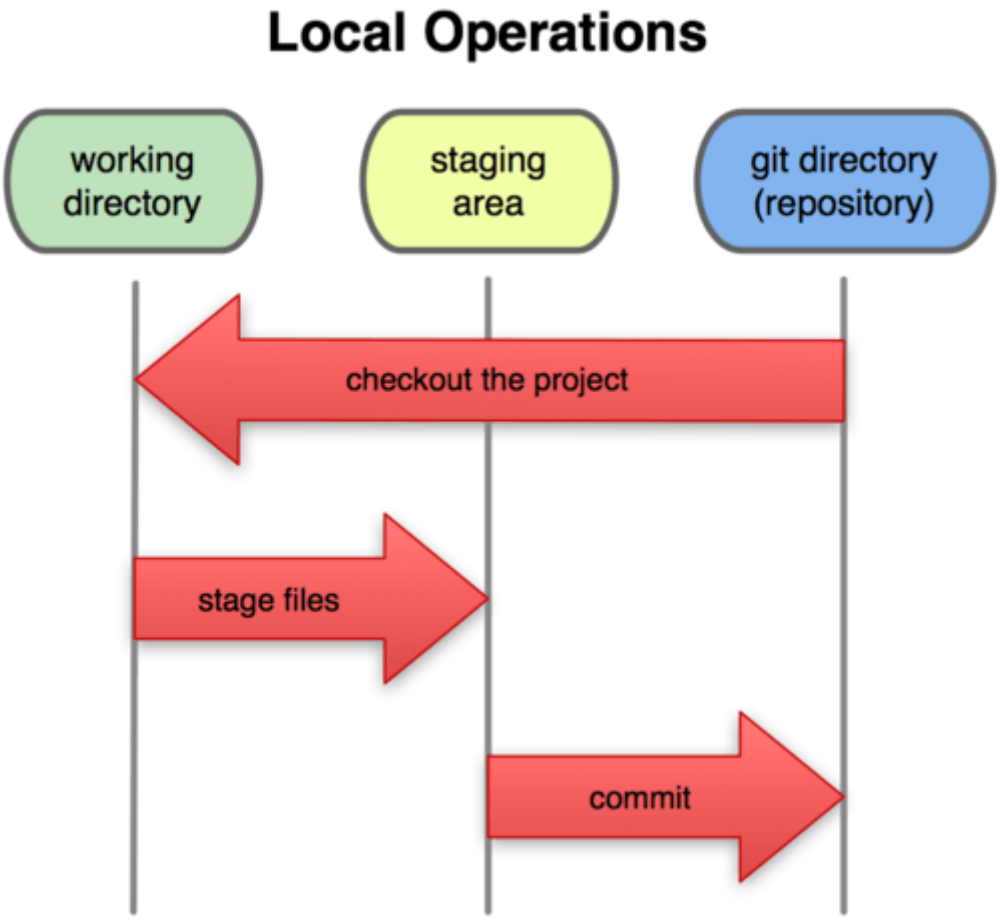

Figura 1-6. Diretório de trabalho, área de preparação, e o diretório do Git.

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 sinta-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:

```
$ yum install curl-devel expat-devel gettext \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  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:

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

Então, compilá-lo e instalá-lo:

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

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

```
$ 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:

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

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

```
$ 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 SourceForge (veja Figura 1-7):

```
http://sourceforge.net/projects/git-osx-installer/
```

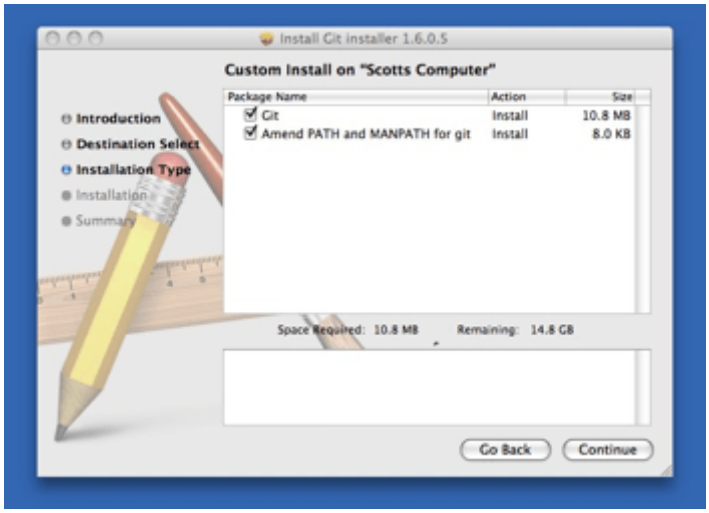

Figura 1-7. Instalador Git OS X.

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

```
$ 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:

```
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 Settins\$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:

```
$ git config --global user.name "John Doe"

$ 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 sobrepor 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:

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

Sua Ferramenta de Diff

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

```
$ 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:

```
$ git config --list

user.name=Scott Chacon
```

```
user.email=schacon@gmail.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...
```

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}`:

```
$ git config user.name

Scott Chacon
```

Obtendo Ajuda

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

```
$ git help <verb>

$ git <verb> --help

$ man git-<verb>
```

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

```
$ 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.

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.

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

```
$ 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:

```
$ git add *.c

$ git add README

$ 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 queria clonar a biblioteca Git do Ruby chamada Grit, você pode fazê-lo da seguinte forma:

```
$ 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:

```
$ 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.

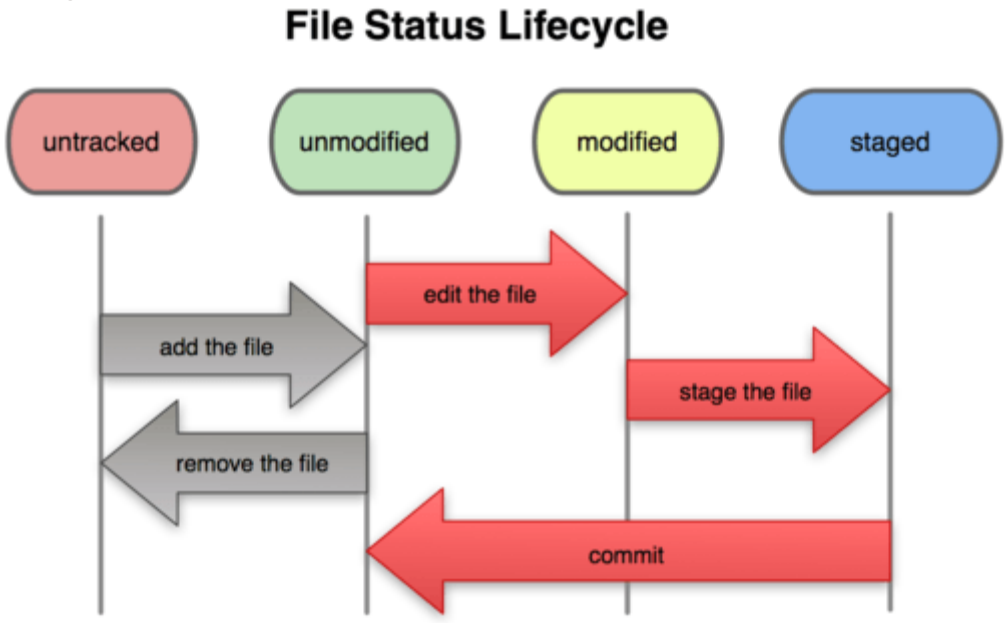

Figura 2-1. O ciclo de vida dos status de seus arquivos.

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:

```
$ git status
# On branch master
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:

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

Você pode ver que o seu novo arquivo `README` 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:

```
$ 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:

```
$ git status
# On branch master
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
#
#    new file:   README
#
```

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 aquivo previamente monitorado chamado `benchmarks.rb` e então rodar o comando `status` novamente, você terá algo semelhante a:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#    modified:   benchmarks.rb
#
```

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 aquivos com conflito). Agora vamos rodar o comando `git add` para selecionar o arquivo `benchmarks.rb`, e então rodar `git status` novamente:

```
$ git add benchmarks.rb
$ git status
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    new file:   README
#    modified:   benchmarks.rb
#
```

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:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    new file:   README
#    modified:   benchmarks.rb
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#    modified:   benchmarks.rb
#
```

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:

```
$ git add benchmarks.rb
```

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#    new file:   README

#    modified:   benchmarks.rb

#
```

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`:

```
$ cat .gitignore

*.[oa]

*~
```

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 asterísco (`*`) 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`:

```
# um comentário - isto é ignorado

# sem arquivos terminados em .a

*.a
```

```
# mas rastreie lib.a, mesmo que você tenha ignorado arquivos terminados em .a acima
!lib.a
# apenas ignore o arquivo TODO na raiz, não o subdiretório TODO
/TODO
# ignore todos os arquivos no diretório build/
build/
# ignore doc/notes.txt mas, não ignore doc/server/arch.txt
doc/*.txt
```

Visualizando Suas Mudanças Selecionadas e Não Selecionadas

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 selecionou (stage)? E o que você selecionou, 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 selecioná-lo. Se você rodar o comando `status`, você novamente verá algo assim:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#    modified:   benchmarks.rb
#
```

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

```
$ git diff
```

```
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           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:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
```

```
+ http://github.com/mojombo/grit

+

+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:

```
$ git add benchmarks.rb

$ echo '# test line' >> benchmarks.rb

$ git status

# On branch master

#

# Changes to be committed:

#

#    modified:   benchmarks.rb

#

# Changes not staged for commit:

#

#    modified:   benchmarks.rb

#
```

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

```
$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index e445e28..86b2f7c 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -127,3 +127,4 @@ end

 main()
```

```
 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

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

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           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`:

```
$ 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):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".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 comanto `commit` ao especificá-la após a flag `-m`, assim:

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

```
$ git status

# On branch master

#

# Changes not staged for commit:

#

#    modified:   benchmarks.rb

#

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

 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 saida do seu `git status`:

```
$ rm grit.gemspec

$ git status

# On branch master

#

# Changes not staged for commit:

#   (use "git add/rm <file>..." to update what will be committed)

#

#       deleted:    grit.gemspec

#
```

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

```
$ git rm grit.gemspec

rm 'grit.gemspec'
```

```
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

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`:

```
$ 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:

```
$ 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:

```
$ 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

```
$ 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:

```
$ git mv README.txt README

$ git status

# On branch master

# Your branch is ahead of 'origin/master' by 1 commit.

#

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       renamed:    README.txt -> README

#
```

No entanto, isso é equivalente a rodar algo como:

```
$ mv README.txt README

$ git rm README.txt

$ 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 ferramente 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:

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

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

```
$ git log

commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>

Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the verison number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Por padrão, sem argumentos, `git log` lista os commits feitos 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.

```
$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the verison number

diff --git a/Rakefile b/Rakefile
```

```
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"

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

    removed unnecessary test code

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

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
```

```
\ No newline at end of file
```

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`

```
$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the verison number

 Rakefile |    2 +-

 1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

 lib/simplegit.rb |    5 -----

 1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

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

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the verison number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
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:

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

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

Opção	**Descrição da saída**
`%H`	Hash do commit
`%h`	Hash do commit abreviado
`%T`	Árvore hash
`%t`	Árvore hash abreviada
`%P`	Hashes pais

Opção	Descrição da saída
`%p`	Hashes pais abreviados
`%an`	Nome do autor
`%ae`	Email do autor
`%ad`	Data do autor (formato respeita a opção -date=)
`%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 *commiter* 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 commiter. 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:

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

```
*   11d191e Merge branch 'defunkt' into local
```

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

```
$ 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 simultâneamente, 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
`-(n)`	Mostra somente os últimos n commits.
`--since, --after`	Limita aos commits feitos depois da data especificada.
`--until, --before`	Limita aos commits feitos antes da data especificada.
`--author`	Somente mostra commits que o autor casa com a string especificada.
`--committer`	Somente mostra os commits em que a entrada do commiter 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:

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

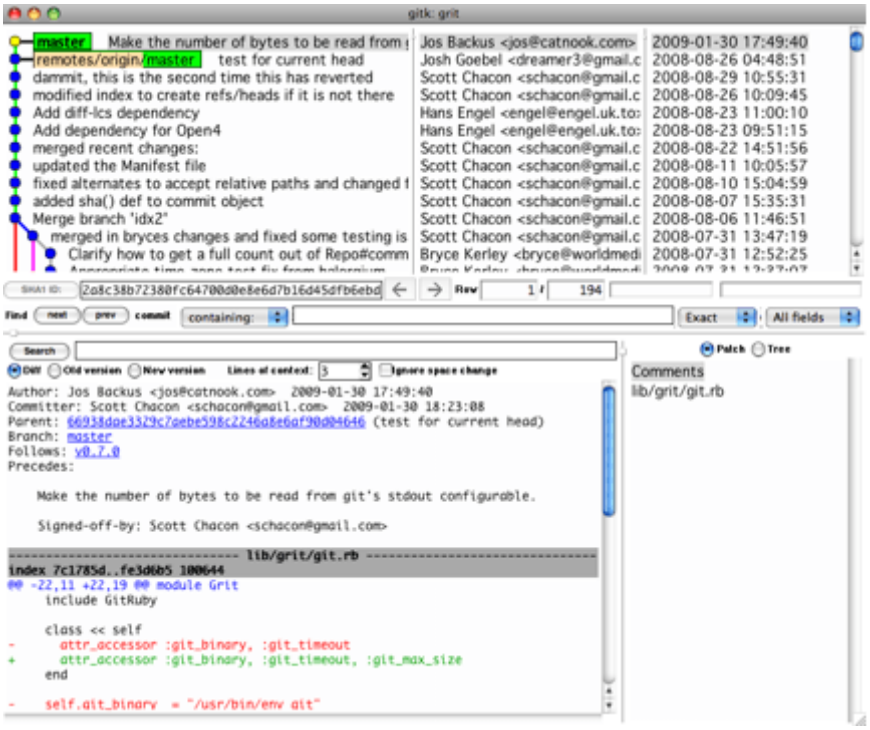

Figura 2-2. O visualizador de histórico gitk.

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`:

```
$ 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:

```
$ git commit -m 'initial commit'

$ git add forgotten_file

$ 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:

```
$ git add .

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       modified:   README.txt

#       modified:   benchmarks.rb

#
```

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`:

```
$ git reset HEAD benchmarks.rb

benchmarks.rb: locally modified

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       modified:   README.txt
```

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

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 — revertê-lo para o que era antes de fazer o último commit (ou do inicio do clone, ou seja la 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:

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

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:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
```

```
#
```

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:

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

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

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

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
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]`:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin    git://github.com/schacon/ticgit.git
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:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
```

```
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [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:

```
$ 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 sevidor:

```
$ 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:

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    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`:

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
```

```
    master

    postgres

  Local branch pushed with 'git push'

    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`:

```
$ git remote rename pb paul

$ git remote

origin

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`:

```
$ git remote rm paul

$ git remote

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`:

```
$ git tag

v0.1
```

```
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:

```
$ git tag -l 'v1.4.2.*'

v1.4.2.1

v1.4.2.2

v1.4.2.3

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`:

```
$ git tag -a v1.4 -m 'my version 1.4'

$ git tag

v0.1

v1.3

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`:

```
$ git show v1.4

tag v1.4

Tagger: Scott Chacon <schacon@gee-mail.com>

Date:   Mon Feb 9 14:45:11 2009 -0800
```

```
my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Sun Feb 8 19:02:46 2009 -0800


    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`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for

user: "Scott Chacon <schacon@gee-mail.com>"

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:

```
$ git show v1.5

tag v1.5

Tagger: Scott Chacon <schacon@gee-mail.com>

Date:   Mon Feb 9 15:22:20 2009 -0800


my signed 1.5 tag

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.8 (Darwin)


iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN

Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/

=WryJ
```

```
-----END PGP SIGNATURE-----

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Sun Feb 8 19:02:46 2009 -0800


    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`:

```
$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5
```

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

```
$ git show v1.4-lw

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date:   Sun Feb 8 19:02:46 2009 -0800


    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:

```
$ git tag -v v1.4.2.1

object 883653babd8ee7ea23e6a5c392bb739348b1eb61

type commit

tag v1.4.2.1

tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700


GIT 1.4.2.1


Minor fixes since 1.4.2, including git-mv and git-http with alternates.

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Good signature from "Junio C Hamano <junkio@cox.net>"

gpg:                 aka "[jpeg image of size 1513]"

Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

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

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Can't check signature: public key not found

error: could not verify the tag 'v1.4.2.1'
```

Taggeando Mais Tarde

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

```
$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file
```

```
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Agora, 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:

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

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

```
$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5


$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date:   Mon Feb 9 15:32:16 2009 -0800


version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date:   Sun Apr 27 20:43:35 2008 -0700


    updated rakefile
```

```
...
```

Compartilhando Tags

Por padrão, o comando `git push` não 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]`.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [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.

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

```
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:

```
$ git co<tab><tab>

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:

```
$ git log --s<tab>

--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:

```
$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status
```

Isto significa que, por exemplo, ao invés de digitar `git commit`, você só precisa digitar `git ci`. 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 unstanging de um arquivo, você pode adicionar o seu próprio pseudônimo unstage para o Git:

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

Isto faz dos dois comandos abaixo equivalentes:

```
$ git unstage fileA

$ git reset HEAD fileA
```

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

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

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

```
$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date:   Tue Aug 26 19:48:51 2008 +0800


    test for current head


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

Como você pode ver, 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`:

```
$ 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.