Versionamento de código com Git

Sumário

Sistemas de Controle de Versão

O Sistema de Controle de Versão (Version Control System ou VCS) é o responsável por registrar as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo, possibilitando a recuperação de versões específicas.

Mesmo que os exemplos aqui apresentados utilizem arquivos de código fonte, você pode utilizar um VCS com praticamente qualquer tipo de arquivo em um computador. Por exemplo, 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 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 (veja Figura 1).

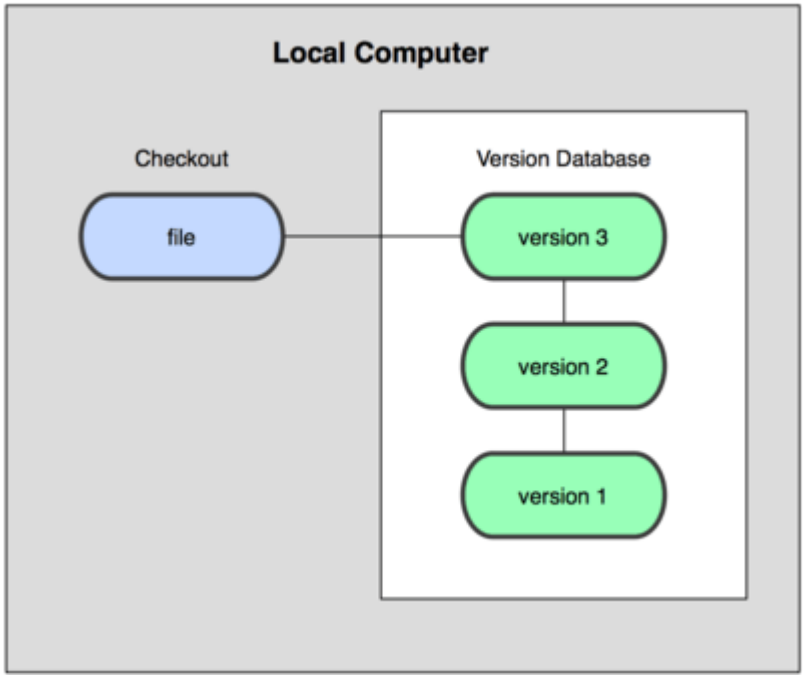

Figura 1 - Diagrama de controle de versão local

Sistemas de Controle de Versão Centralizados (CVCS)

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 (checkout) os arquivos do servidor (veja Figura 2). Por muitos anos, esse foi o modelo padrão para controle de versão.

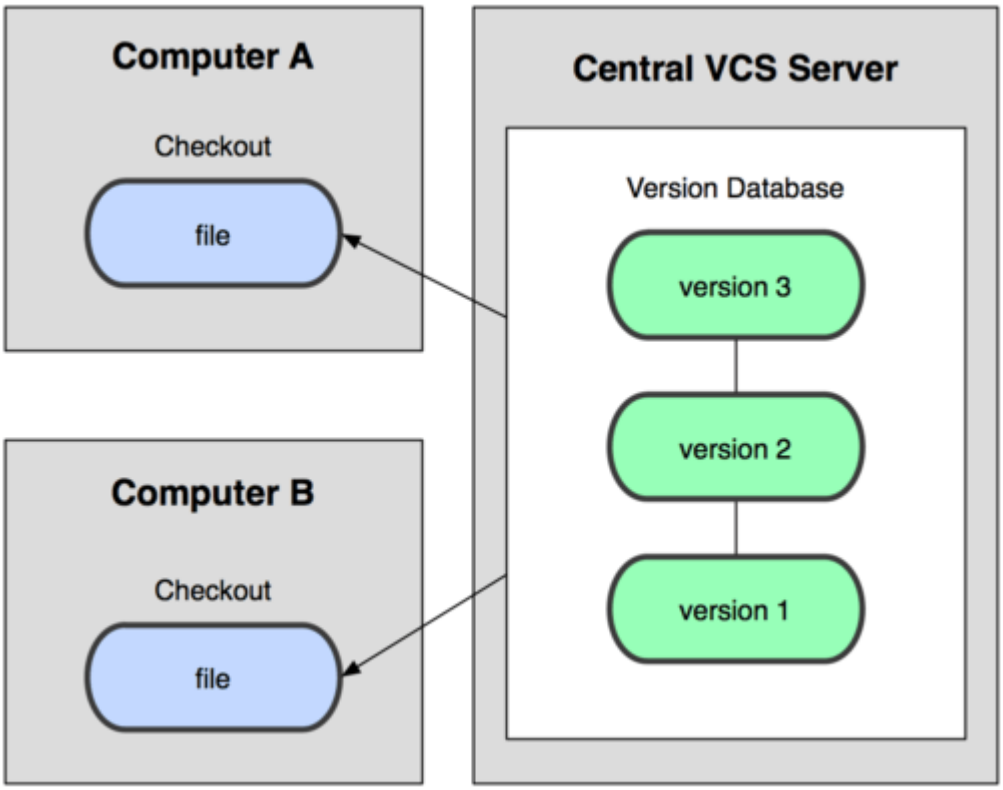

Figura 2 - Diagrama de Controle de Versão Centralizado

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 (DVCS)

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.

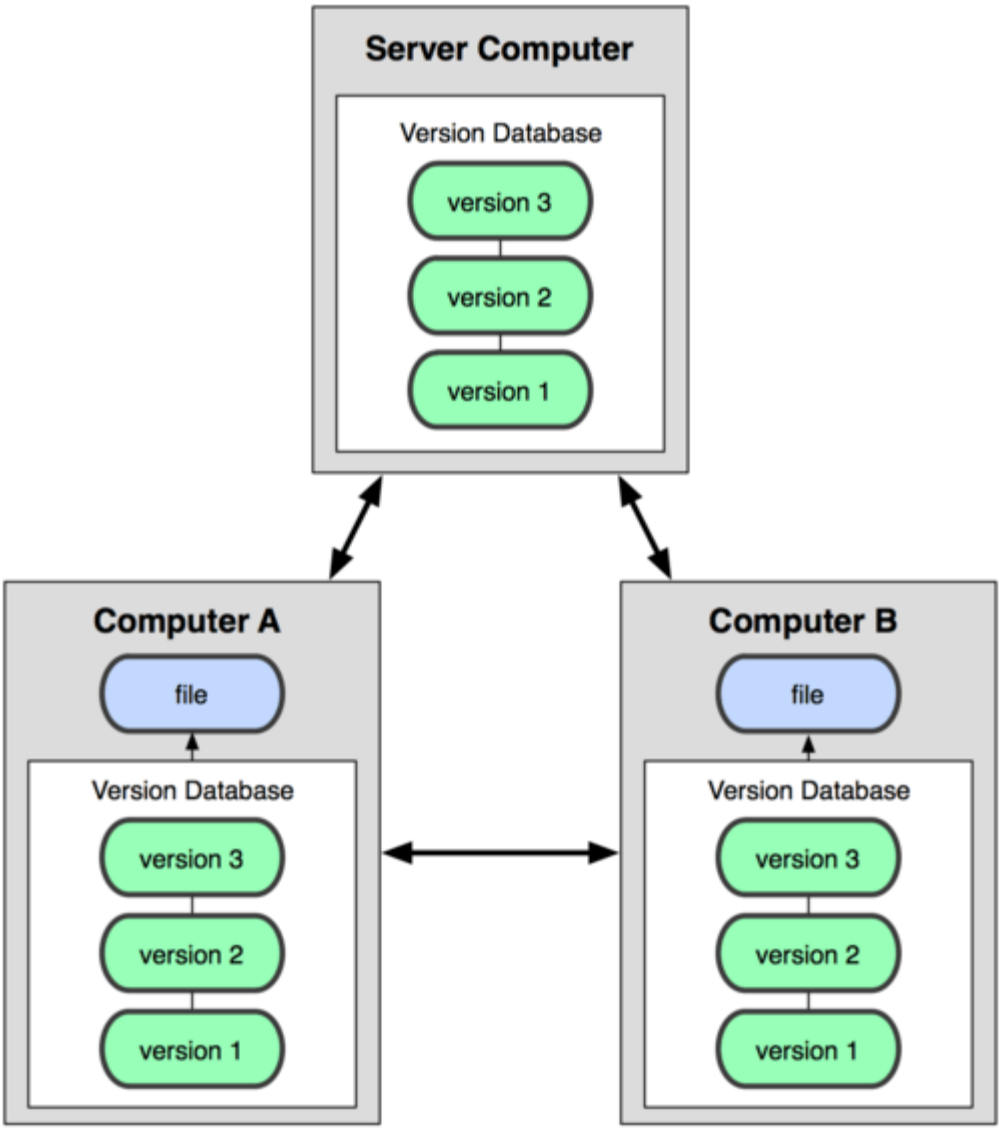

Figura 3 - Diagrama de Controle de Versão Distribuído

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.

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.

Conhecendo o Git – Noções Básicas

Esta seção tem como objetivo introduzir as características principais do GIT. Se você entender bem os fundamentos de como ele funciona, será muito mais fácil utilizá-lo de forma efetiva.

Snapshots

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 (veja Figura 4).

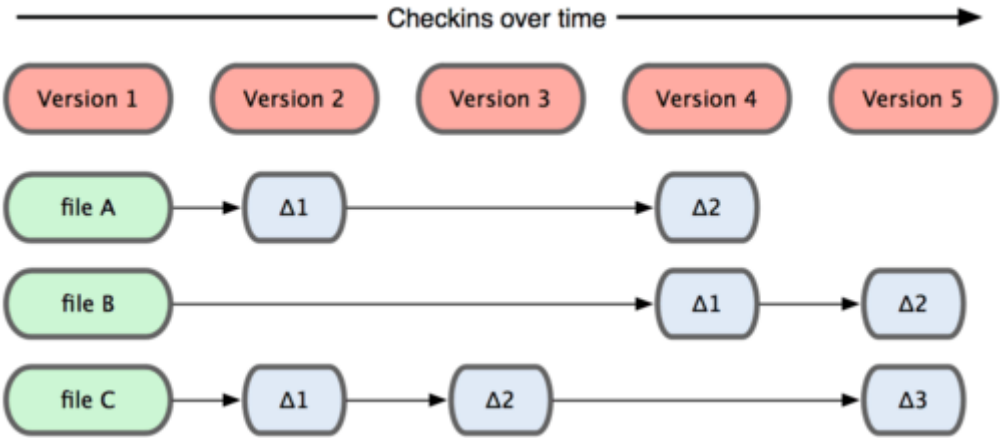

Figura 4 - Organização de mudanças por arquivo em outros sistemas de versionamento

O 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 mostra melhor como o Git lida com seus dados.

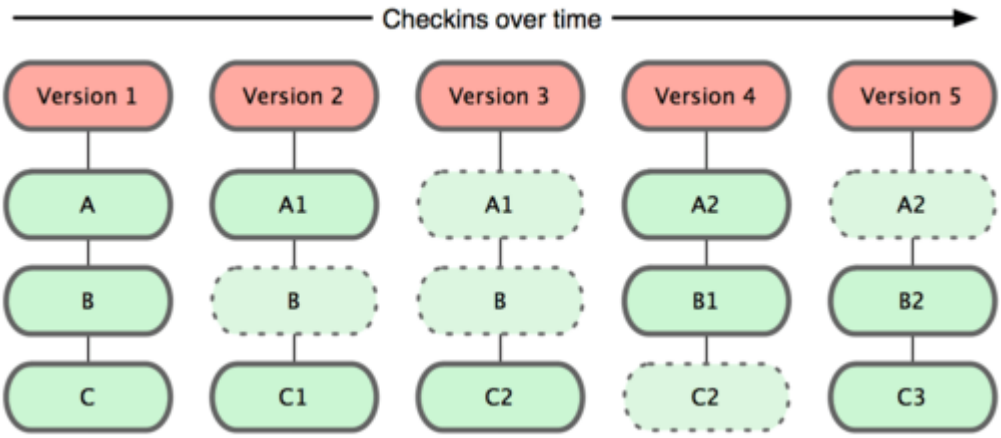

Figura 5 - Organização de mudanças em arquivos no Git

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.

Quase Todas as 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. Isto gera um ganho de velocidade em relação às operações realizadas por CVCS.

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.

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
```

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.

Os 3 estados

O 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).

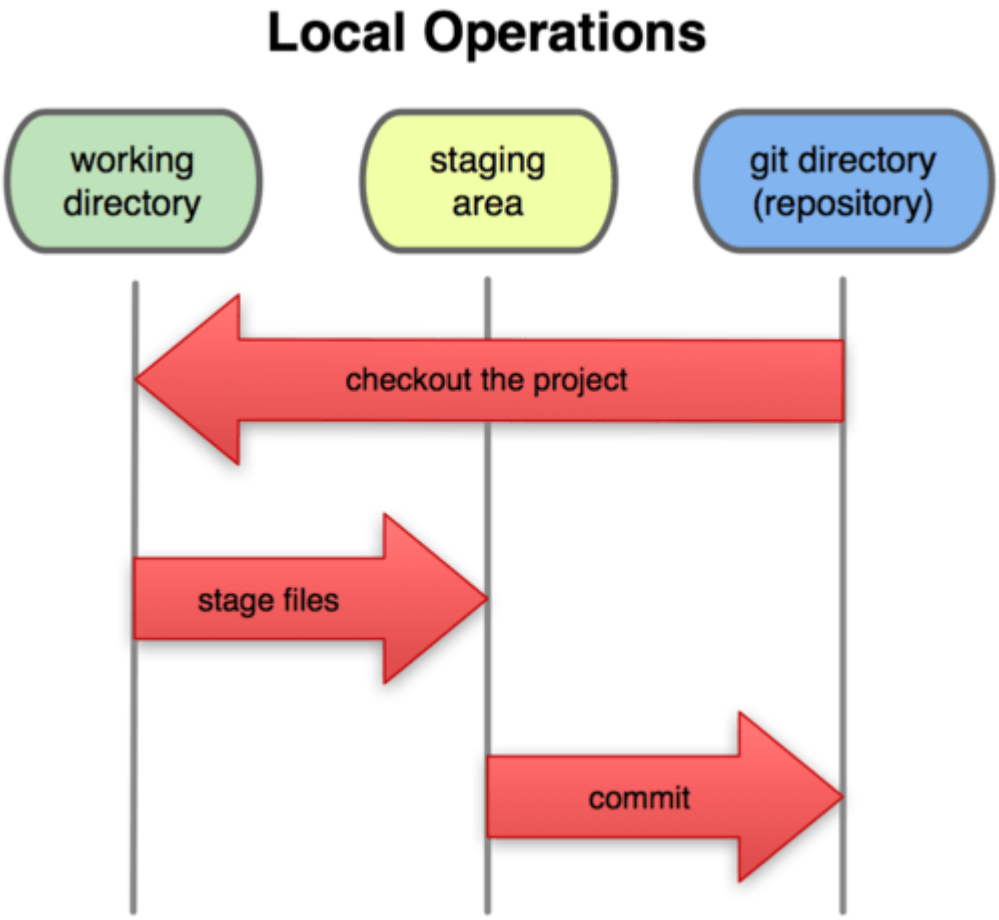

Figura 6 - Diretório de trabalho, área de preparação, e o diretório do Git.

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. Se está modificada, mas foi adicionada à área de preparação, está preparada. E se foi alterada desde que foi obtida, mas não foi preparada, está modificada.

Instalando o Git

Instalação no Windows

Baixe o arquivo exe do instalador a partir da página https://gitforwindows.org/ e execute-o.

Instalação 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
```

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.

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.

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
```

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:

```
$ 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.

Verificar 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
```

Utilizando a 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:

```
$ 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 podem ser acessados em qualquer lugar, mesmo offline.

Obtendo um repositório

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 contém todos os arquivos necessários de seu repositório — um esqueleto de repositório Git. Neste ponto, nada em seu projeto é monitorado.

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'
```

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. Ao executar este comando, o 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.

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.

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 (ver Figura 7).

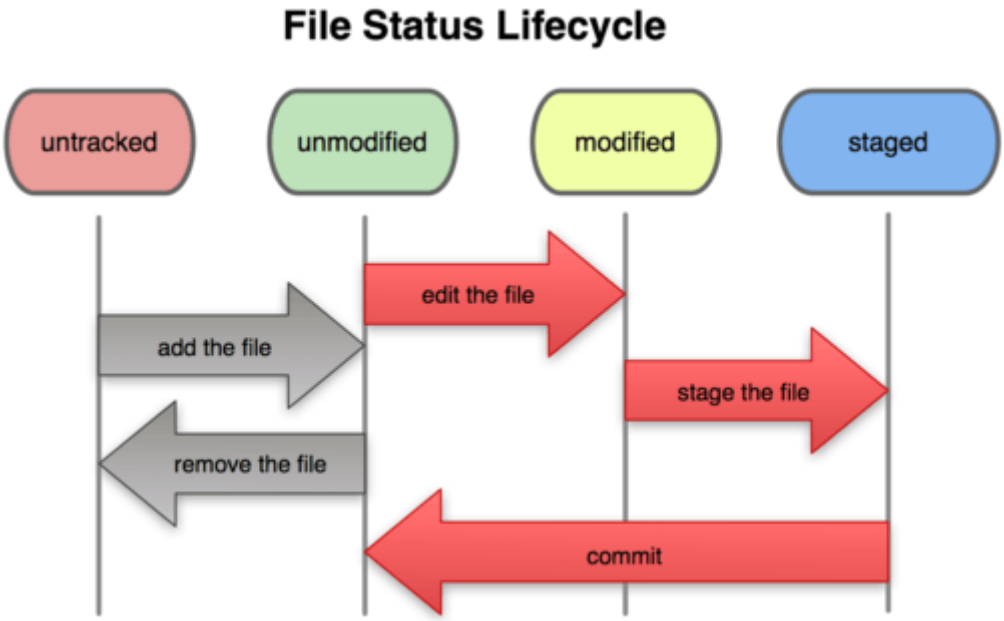

Figura 7 - Ciclo de status de seus arquivos.

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. O 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. Nos próximos capítulos, 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 tem a intenção de incluir.

Monitorando Novos Arquivos

Digamos, que você queira incluir o arquivo README, portanto vamos começar a monitorar este arquivo. Para passar a monitorar um novo arquivo, use o comando “git add”:

```
$ 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 arquivo 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 arquivos com conflito como resolvidos).

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
#
```

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 commitar 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:

```
# 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 repositor
```

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. 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 ali para ajudar a lembrar o que está commitando.

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, da seguinte forma:

```
$ 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 a 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(-)
```

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”:

```
$ 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 o comando “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. 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.

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
```

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.

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”.

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
```

Desfazendo alterações

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

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 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 início do clone, ou outro ponto)? 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
#
```

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.

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. No entanto, qualquer coisa que você perder que nunca foi commitada, provavelmente nunca mais será vista novamente.

Repositórios 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
```

Adicionando Repositórios Remotos

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
```

Fetch e Pull

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-las mais detalhadamente nos próximos capítulos.)

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 os 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 uma branch configurada para acompanhar uma branch remota (falaremos disso em capítulos futuros), você pode usar o comando git pull para automaticamente fazer o fetch e o merge de uma branch remoto na sua branch atual. Essa pode ser uma maneira mais fácil ou confortável pra você; e por padrão, o comando git clone automaticamente configura sua branch local master para acompanhar a branch remota master do servidor de onde você clonou (desde que o remoto tenha uma 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.

Push

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 a sua 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:

```
$ git push origin master
```

Este comando funciona apenas se você clonou de um servidor que você tem 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.

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, v2.0, etc.). 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, e-mail 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 tagueado 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 tagueado, 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'
```

Tags Leves

Outro jeito para taguear 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'
```

Tagueando Mais Tarde

Você também pode taguear 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, suponhamos 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.

Ramificação (Branching) no Git

O que é uma Branch

Para compreender realmente a forma como o Git cria branches, precisamos dar um passo atrás e examinar como o Git armazena seus dados. 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 nos capítulos anteriores), 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):

```
$ git add README test.rb LICENSE
```

```
$ 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 8.

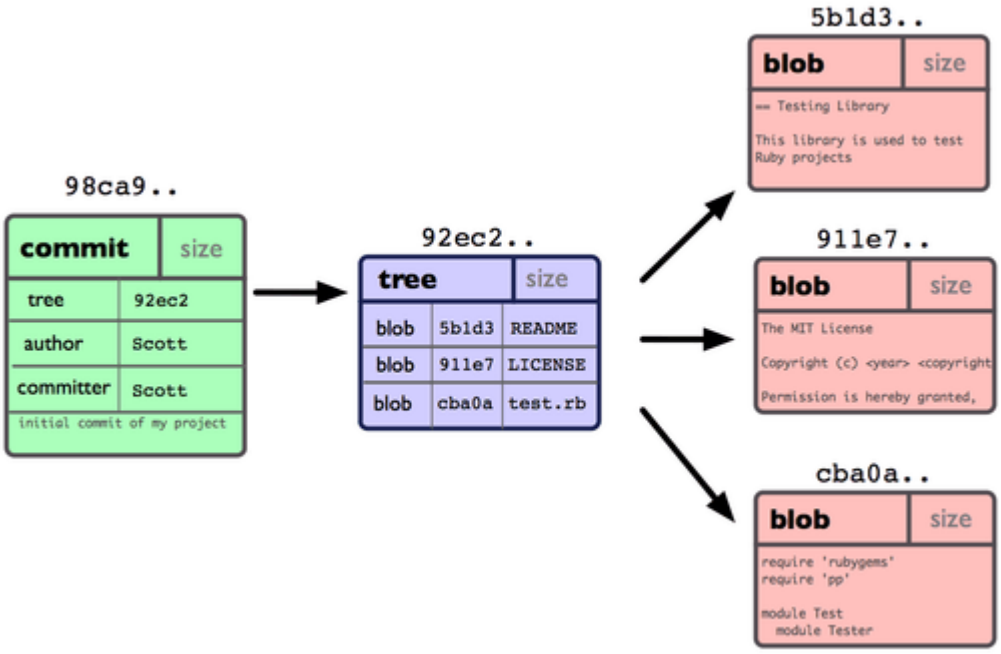

Figura 8 - Dados de um repositório com um único commit

Se você modificar algumas coisas e fizer um commit novamente, o próximo commit armazenará um ponteiro para o commit imediatamente anterior. Depois de mais dois commits, seu histórico poderia ser algo como a Figura 9.

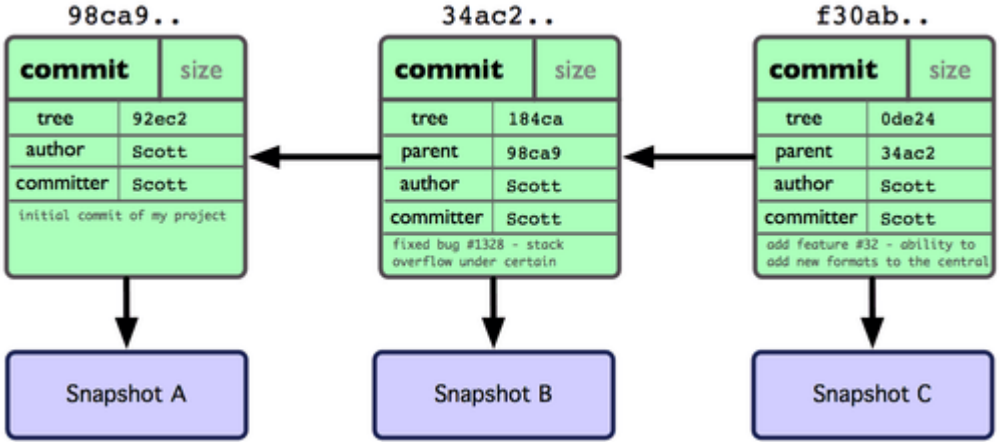

Figura 9 - Dados dos objetos Git para múltiplos commits

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

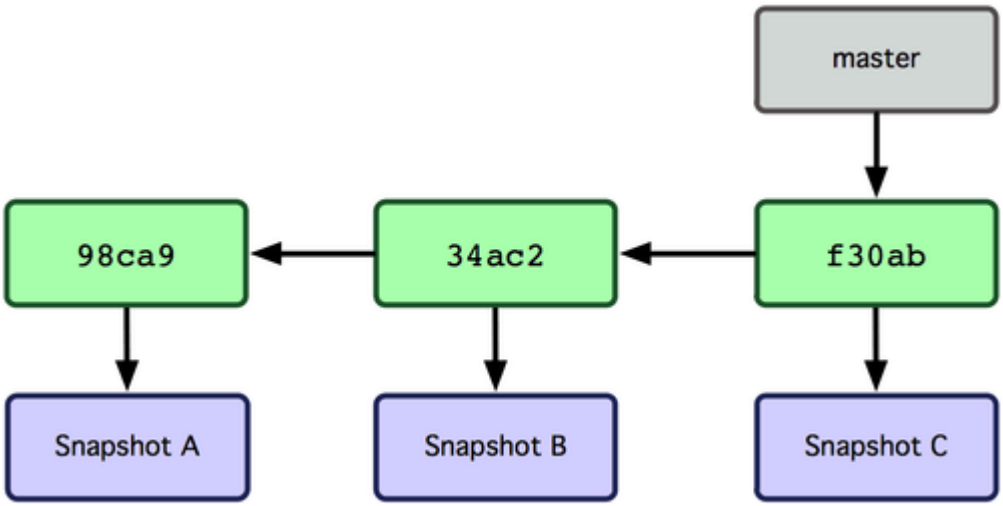

Figura 10 - Branch apontando para o histórico de commits.

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

```
$ git branch testing
```

Isso cria um novo ponteiro para o mesmo commit em que você está no momento (ver a Figura 11)

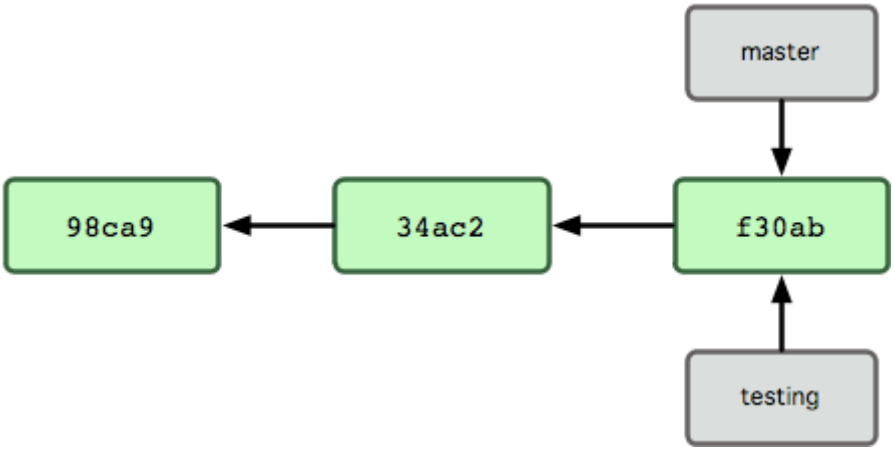

Figura 11 - Múltiplos branches apontando para o histórico de commits.

Como o Git sabe a 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 a branch local em que você está no momento. Neste caso, você ainda está no master. O comando git branch só criou uma nova branch — ele não mudou para essa branch (veja Figura 12).

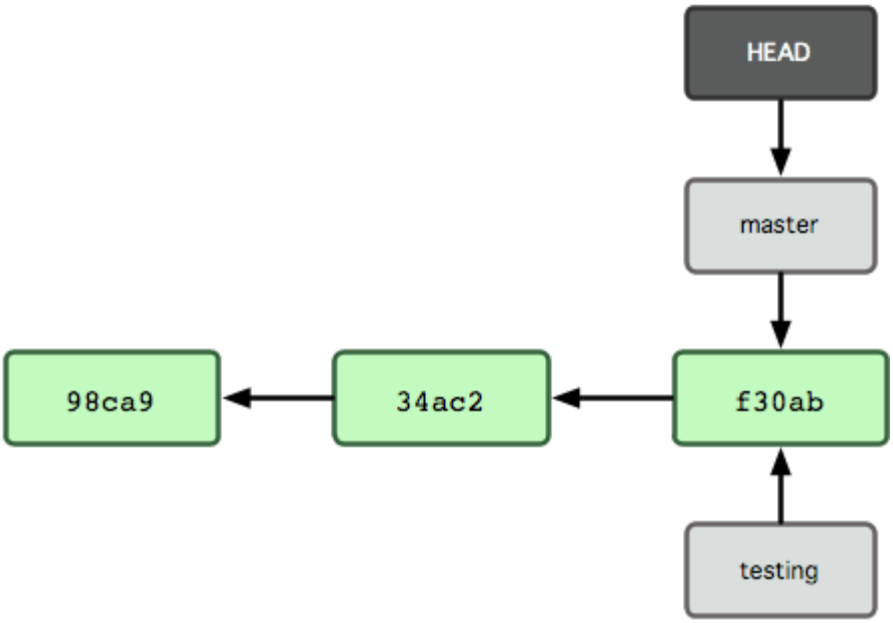

Figura 12 - HEAD apontando para a branch em que você está.

Para mudar para uma branch existente, você executa o comando git checkout. Vamos mudar para a nova branch testing:

```
$ git checkout testing
```

Isto move o HEAD para apontar para a branch testing (ver Figura 13)

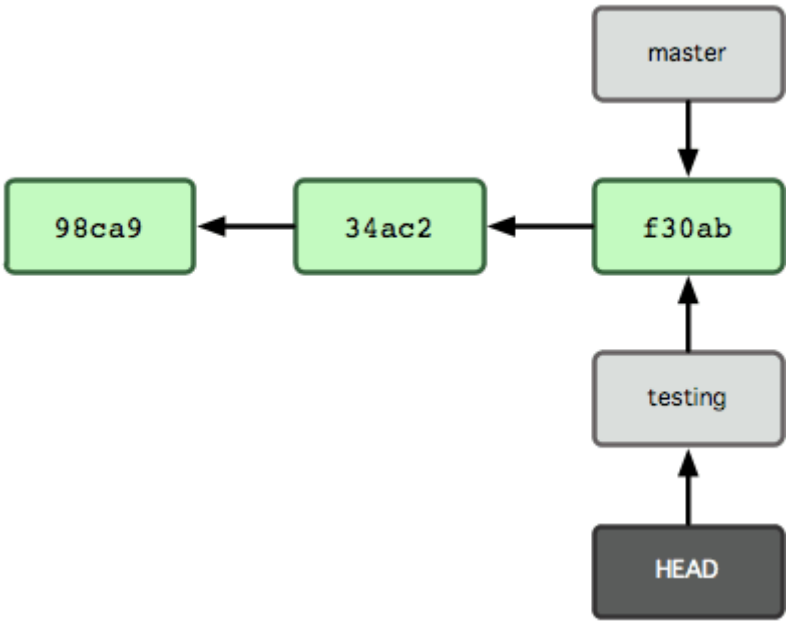

Figura 13 - O HEAD aponta para outro branch quando você troca de branches.

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

```
$ vim test.rb
```

```
$ git commit -a -m 'fiz uma alteração'
```

A Figura 14 ilustra o resultado:

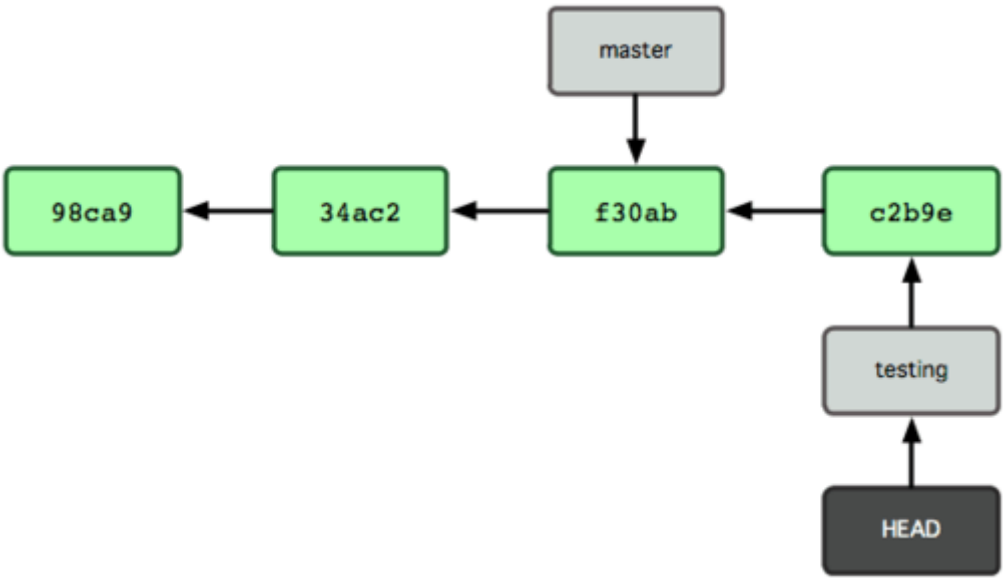

Figura 14 - A branch para o qual HEAD aponta avança com cada commit.

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

```
$ git checkout master
```

A Figura 15 mostra o resultado:

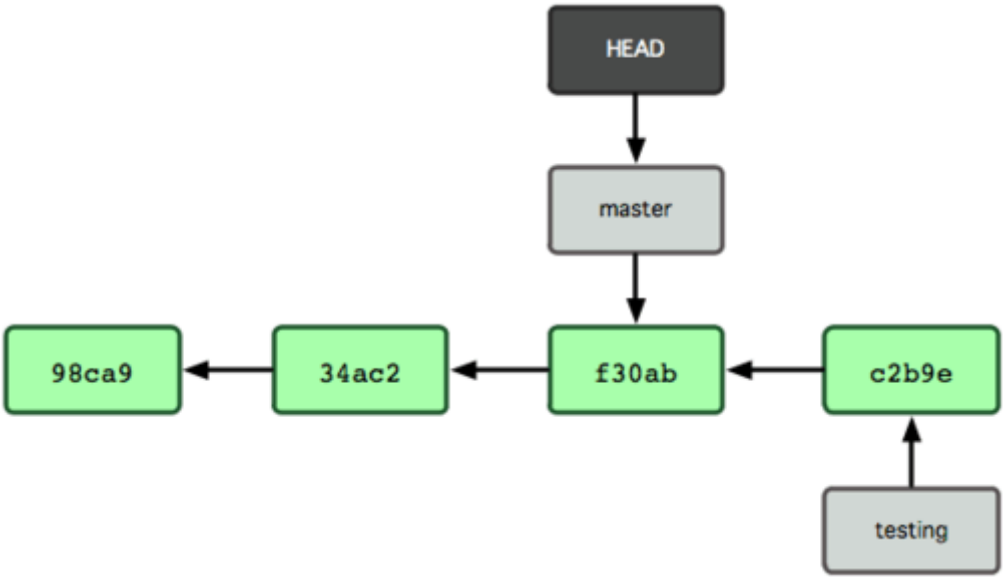

Figura 15 - O HEAD se move para outro branch com um checkout.

Esse comando fez duas coisas. Ele alterou o ponteiro HEAD para apontar novamente para a 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 na sua branch testing, temporariamente, de modo que você possa ir em uma direção diferente.

Vamos fazer algumas mudanças e fazer o commit novamente:

```
$ vim test.rb
```

```
$ git commit -a -m 'fiz outra alteração'
```

Agora o histórico do seu projeto divergiu (ver Figura 16). Você criou e trocou para uma branch, trabalhou nela, e então voltou para a sua branch principal e trabalhou mais. Ambas as mudanças são isoladas em branches distintos: você pode alternar entre as branches e fundi-los (merge) quando estiver pronto. E você fez tudo isso simplesmente com os comandos branch e checkout.

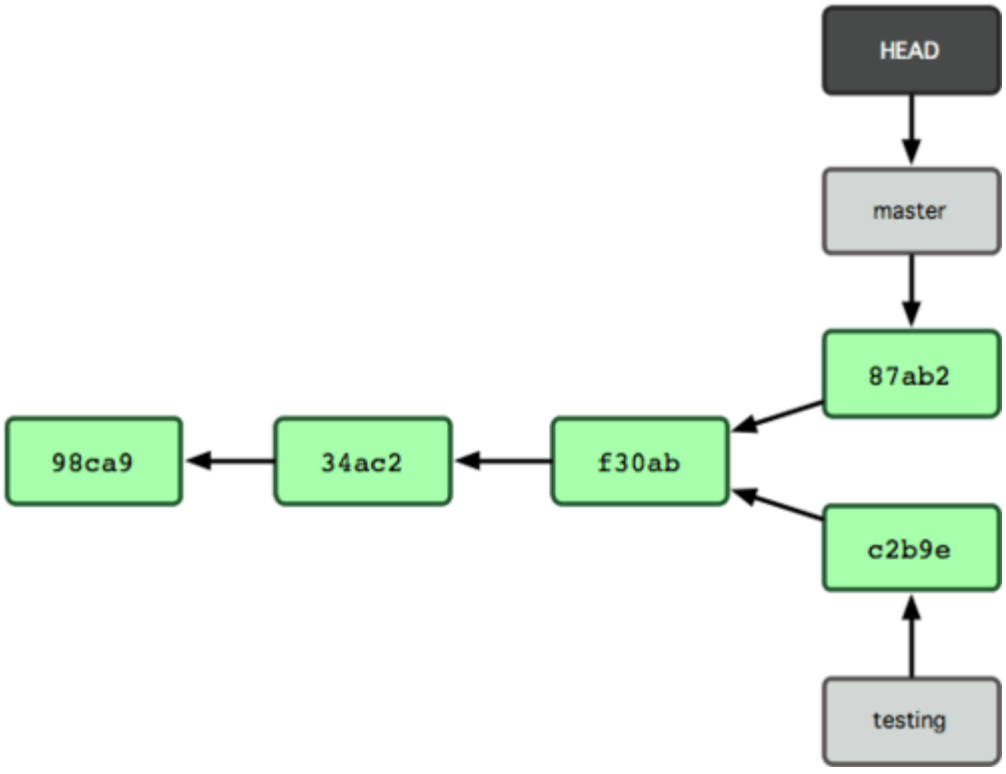

Figura 16 - O histórico das branches diverge.

Como uma 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, as branches são de baixo custo 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.

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.

Nesta 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 17)

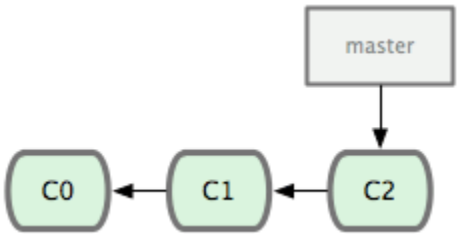

Figura 17 - Um histórico de commits pequeno e simples.

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:

```
$ git checkout -b iss53

Switched to a new branch "iss53"
```

Isso é um atalho para:

```
$ git branch iss53
```

```
$ git checkout iss53
```

A Figura 18 ilustra o resultado.

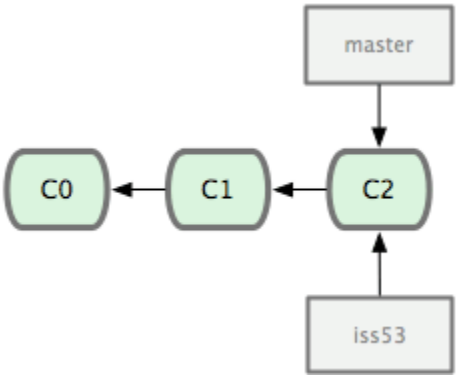

Figura 18 - Criando uma nova branch

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 19):

```
$ vim index.html
```

```
$ git commit -a -m 'adicionei um novo rodapé [issue 53]'
```

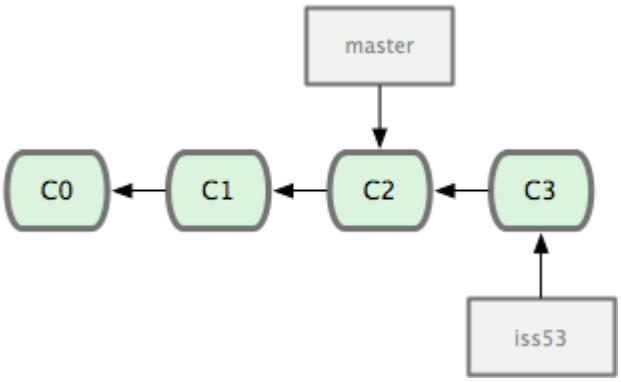

Figura 19 - A branch iss53 avançou com suas modificações.

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 a 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. Por enquanto, você fez o commit de todas as suas modificações, então você pode mudar para a sua branch master:

```
$ git checkout master

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 a branch parecia no seu último commit nele.

Em seguida, você tem uma correção para fazer. Vamos criar uma branch para a correção (hotfix) para trabalhar até a conclusão (veja a Figura 20).

```
$ git checkout -b 'hotfix'

Switched to a new branch "hotfix"
```

```
$ vim index.html
```

```
$ git commit -a -m 'consertei o endereço de e-mail'

[hotfix]: created 3a0874c: "consertei o endereço de e-mail"
 1 files changed, 0 insertions(+), 1 deletions(-)
```

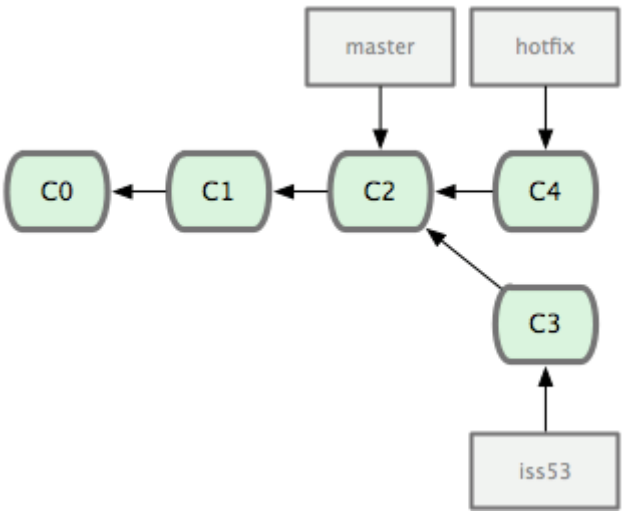

Figura 20 - branch de correção (hotfix) baseada num ponto de seu branch master.

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":

```
$ git checkout master
```

```
$ git merge hotfix

Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Você irá notar a frase "Fast forward" no merge. Em razão da 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 pela branch master, e você pode fazer o deploy (veja a Figura 21).

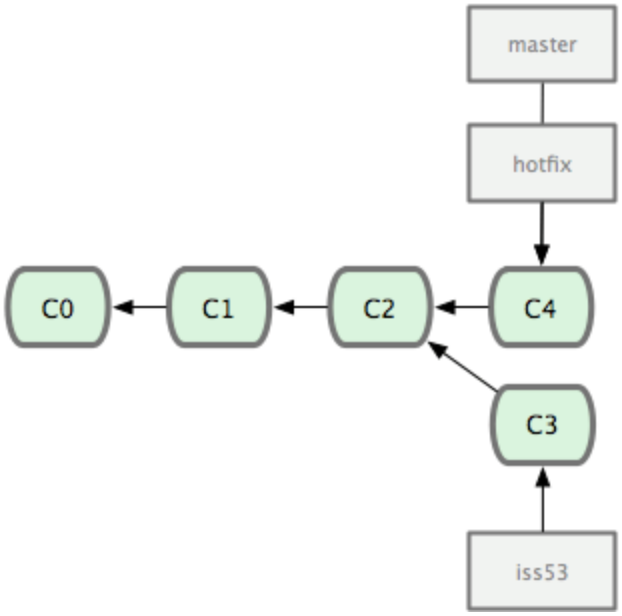

Figura 21 - Depois do merge, sua branch master aponta para o mesmo local que a branch hotfix.

Depois que a sua correção foi enviada, você está pronto para voltar ao trabalho que estava fazendo antes de ser interrompido. No entanto, primeiro você apagará a branch hotfix, pois você não precisa mais dele — a branch master aponta para o mesmo local.

Você pode excluí-lo com a opção -d em git branch:

```
$ git branch -d hotfix

Deleted branch hotfix (3a0874c).
```

Agora você pode voltar para o trabalho incompleto na branch da tarefa #53 e continuar a trabalhar nele (veja a Figura 22):

```
$ git checkout iss53

Switched to branch "iss53"
```

```
$ vim index.html
```

```
$ git commit -a -m 'novo rodapé terminado [issue 53]'

[iss53]: created ad82d7a: "novo rodapé terminado [issue 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

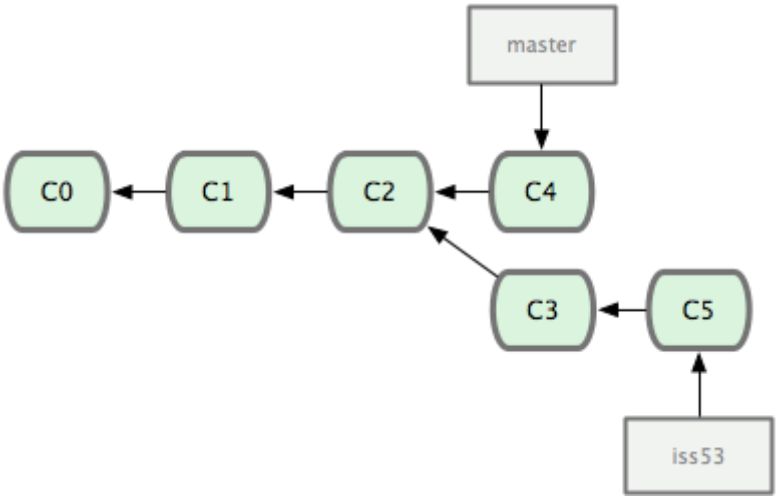

Figura 22 - Sua branch iss53 pode avançar de forma independente.

Vale a pena lembrar aqui que o trabalho feito na sua branch hotfix não existe nos arquivos da sua branch iss53. Se você precisa incluí-lo, você pode fazer o merge da sua branch master na sua 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 na master mais tarde.

Merge Básico

Suponha que você decidiu que o trabalho na tarefa #53 está completo e pronto para ser feito o merge na branch master. Para fazer isso, você fará o merge da sua branch iss53, bem como o merge da branch hotfix de antes. Tudo que você tem a fazer é executar o checkout da branch para onde deseja fazer o merge e então rodar o comando git merge:

```
$ git checkout master
```

```
$ git merge iss53

Merge made by recursive.
 README | 1 +
 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 na branch em que você está não ser um ancestral direto da 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 das branches e o ancestral comum dos dois. A Figura 23 destaca os três snapshots que Git usa para fazer o merge nesse caso.

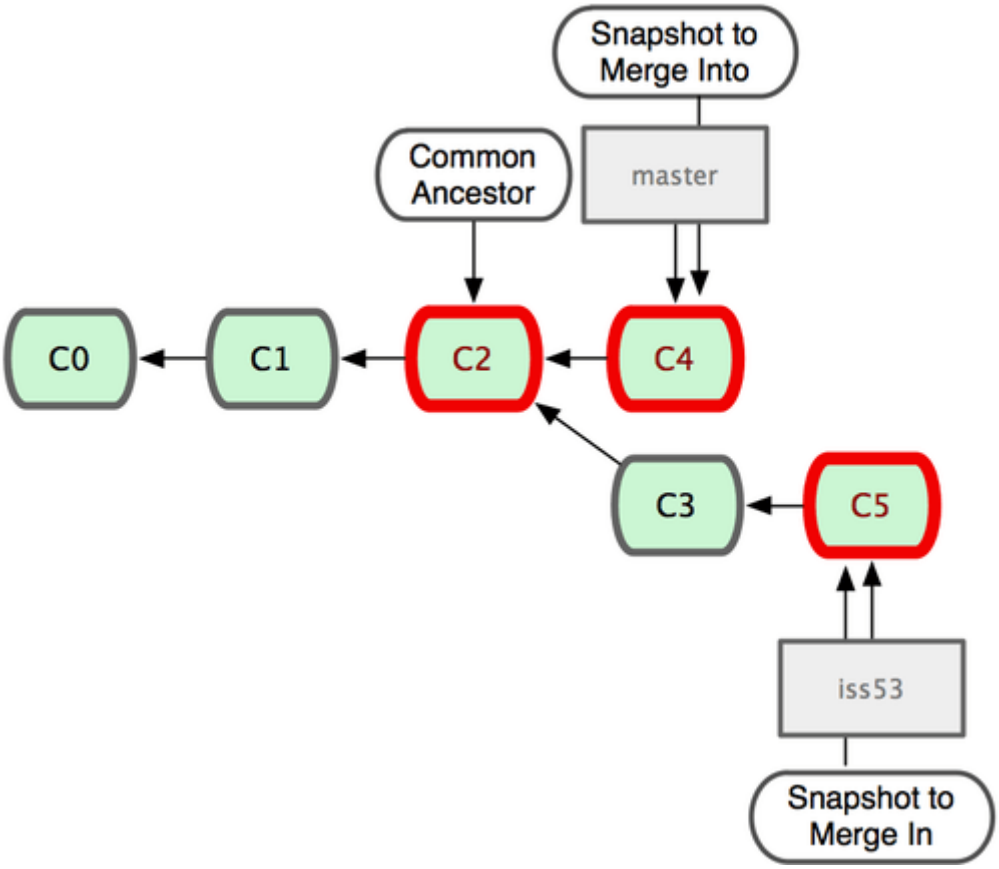

Figura 23 - Git identifica automaticamente a melhor base ancestral comum para o merge da branch.

Em vez de simplesmente avançar o ponteiro da 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 a Figura 24). 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.

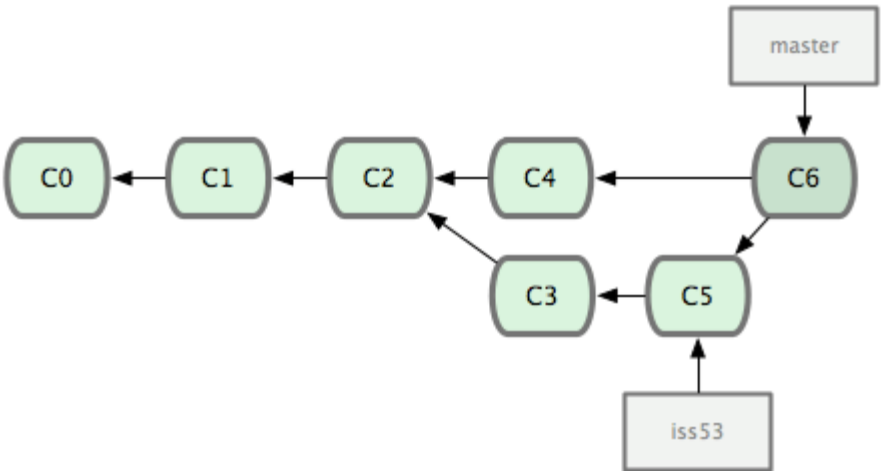

Figura 24 - Git cria automaticamente um novo objeto commit que contém as modificações do merge.

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

```
$ 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 nas duas 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:

```
$ git merge iss53

Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
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”:

```
[master*]$ git status

index.html: needs merge
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    unmerged:   index.html
#
```

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:

```
<<<<<<< HEAD:index.html
<div id="footer">contato : email.support@github.com</div>
=======
<div id="footer">
  por favor nos contate em support@github.com
</div>
>>>>>>> iss53:index.html
```

Isso significa que a versão em HEAD (sua 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:

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

Esta solução tem um pouco de cada seção. Removemos 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.

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

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   index.html
#
```

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:

```
Merge branch 'iss53'

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

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.

Licença

scm.com/book/pt-br/v1, que está sob a licença Creative Commons Attribution Non Commercial Share Alike 3.0.

Bibliografia

- https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Versão
- https://git-scm.com/book/pt-br/v1/Git-Essencial-Gravando-Altera%C3%A7%C3%B5es-no-Reposit%C3%B3rio
- https://git-scm.com/book/pt-br/v1/Ramifica%C3%A7%C3%A3o-Branching-no-Git-O-que-%C3%A9-um-Branch
- https://git-scm.com/book/pt-br/v1/Ramifica%C3%A7%C3%A3o-Branching-no-Git-B%C3%A1sico-de-Branch-e-Merge