Magia Git

Ben Lynn

Agosto 2007

Prefácio

Git é um canivete suíço do controle de versões. Uma ferramenta polivalente realmente versátil, cuja extraordinária flexibilidade torna-o complicado de aprender, principalmente sozinho.

Como Arthur C. Clarke observou, "Qualquer tecnologia suficientemente avançada é considerada mágica". Esta é uma ótima forma de abordar o Git: novatos podem ignorar seu funcionamento interno e vê-lo como algo divertido que pode agradar aos amigos e enfurecer os inimigos com suas maravilhosas habilidades.

Ao invés de entrar em detalhes, forneceremos apenas instruções para casos específicos. Após o uso repetido, você gradualmente entenderá como cada truque funciona, e como adaptar as receitas às suas necessidades.

- Simplified Chinese: by JunJie, Meng and JiangWei. Converted to Traditional Chinese via `cconv -f UTF8-CN -t UTF8-TW`.
- French: by Alexandre Garel, Paul Gaborit, and Nicolas Deram. Also hosted at itaapy.
- German: by Benjamin Bellee and Armin Stebich; also hosted on Armin's website.
- Portuguese-Brazilian: by J.I.Serafini and Leonardo Siqueira Rodrigues.
- Russian: by Tikhon Tarnavsky, Mikhail Dymskov, and others.
- Spanish: by Rodrigo Toledo and Ariset Llerena Tapia.
- Ukrainian: by Volodymyr Bodenchuk.
- Vietnamese: by Trần Ngọc Quân; also hosted on his website.
- Single webpage: barebones HTML, with no CSS.
- PDF file: printer-friendly.

- Debian package, Ubuntu package: get a fast and local copy of this site. Handy when this server is offline.
- Physical book [Amazon.com]: 64 pages, 15.24cm x 22.86cm, black and white. Handy when there is no electricity.

Agradecimentos!

É com grande orgulho que vejo o grande número de pessoas que trabalho na tradução deste livro. Eu realmente agradeço pela grande audiência permitida pelos esforços dos tradutores citados a seguir.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin, Tyler Breisacher, Sonia Hamilton, Julian Haagsma, Romain Lespinasse, Sergey Litvinov, Oliver Ferrigni, David Toca, , Joël Thieffry, e Baiju Muthukadan contribuiram com correções e melhorias.

François Marier mantém o pacote Debian criado originalmente por Daniel Baumann.

My gratitude goes to many others for your support and praise. I'm tempted to quote you here, but it might raise expectations to ridiculous heights.

If I've left you out by mistake, please tell me or just send me a patch!

Licença

 Naturalmente, os fontes estão num repositório Git, e podem ser obtido digitando:

```
$ git clone git://repo.or.cz/gitmagic.git  # Cria um diretório "gitmagic".
```

ou a partir de algum desses mirrors:

```
$ git clone git://github.com/blynn/gitmagic.git
$ git clone git://gitorious.org/gitmagic/mainline.git
$ git clone https://code.google.com/p/gitmagic/
$ git clone git://git.assembla.com/gitmagic.git
$ git clone git@bitbucket.org:blynn/gitmagic.git
```

GitHub, Assembla, e Bitbucket suportam repositórios privados, os dois últimos são grátis.

Introdução

Usaremos uma analogia para falar sobre controle de versões. Veja o verbete Sistema de Controle de Versões para uma explicação mais formal.

Trabalhar é Divertido

Divirto-me com jogos para computador quase a minha vida toda. Em contrapartida, só comecei a usar sistemas de controle de versões quando adulto. Suspeito que não fui o único, e comparar os dois pode tornar estes conceitos mais fáceis de explicar e entender.

Pense na edição de seu código, documento, ou qualquer outra coisa, como jogar um jogo. Uma vez que tenha feito muitos progressos, e gostaria de de salvá-los. Para isso, você clica em "Salvar" no seu editor preferido.

Porém, isto vai sobrescrever a versão anterior. É como nos antigos jogos onde você só tinha uma espaço para salvar: você pode salvar, mas nunca mais poderá voltar a um estado salvo anteriormente. O que é um pena, pois o estado anterior era uma parte muito divertida do jogo e você gostaria de poder revisitá-lo outra hora. Ou pior, o último estado salvo é dificílimo e você terá que recomeçar.

Controle de Versões

Ao editar, você pode "Salvar como …" num arquivo diferente, ou copiar o arquivo antes de sobrescrevê-lo se você quiser manter as versões anteriores. Pode também comprimí-los para economizar espaço. Isto é uma forma rudimentar e muito trabalhosa de controle de versões. Jogos de computador aperfeiçoaram este método, muitos deles acrescentam automaticamente a data e a hora aos estados salvos.

Vamos dificultar um pouco. Digamos que são um monte de arquivos juntos, como os fontes do seu projeto, ou arquivos para um website. Agora se quiser manter suas versões anteriores, terá que arquivar todo um diretório ou vários. Manter muitas versões na mão é inconveniente e rapidamente se tornará caro.

Em alguns jogos de computador, um estado salvo consiste de um diretório cheio de arquivos. Estes jogos escondem estes detalhes do jogador e lhe apresentam uma interface conveniente para gerenciar as diferentes versões neste diretório.

Sistemas de controle de versões não são diferentes. Todos têm uma boa interface para gerenciar seu diretório de versões. Você pode salvar o diretório sempre que desejar, e pode rever qualquer um dos estados salvos quando quiser. Ao contrário da maioria dos jogos de computador, eles são geralmente mais espertos na economia de espaço. Normalmente, apenas uns poucos arquivos mudam de

versão para versão, e com poucas diferenças. Armazenar as diferenças, ao invés de todos os arquivos, economiza espaço.

Controle distribuído

Agora imagine um jogo de computador muito difícil. Tão difícil de terminar que vários jogadores experientes pelo mundo decidem formar uma equipe e compartilhar seus estados salvos do jogo para tentar vencê-lo. Speedruns são exemplos reais: jogadores especializados em diferentes níveis do mesmo jogo colaboram na produção de resultados incríveis.

Como você configura um sistema para que todos possam obter facilmente o que os outros salvarem? E salvar novos estados?

Nos velhos tempos, todos os projetos utilizavam um controle de versões centralizado. Um servidor em algum lugar mantinha os jogos salvos. Ninguém mais teria todos os jogos. Cada jogador mantinha apenas alguns jogos salvos em suas maquinas. Quando algum jogador quiser avançar no jogo, ele pega o ultimo jogo salvo do servidor, joga um pouco, salva e manda de volta para o servidor para que os outros possam usar.

E se um jogador quiser pegar um jogo antigo salvo por algum motivo? Talvez o jogo atual salvo esteja em um nível impossível de jogar devido alguém ter esquecido um objeto três níveis atrás, e é preciso encontrar o último jogo salvo que está em um nível que pode ser completado com sucesso. Ou talvez queira comparar dois jogos salvos para saber o quanto um jogador avançou.

Podem existir vários motivos para ver uma versão antiga, mas o modus operandi é o mesmo. Têm que solicitar ao servidor centralizado a versão antiga. E quanto mais jogos salvos forem necessários, maior é o tráfego de informação.

A nova geração de sistemas de controle de versões, dentre eles o Git, é conhecida como sistemas distribuídos, e pode ser pensada como uma generalização dos sistemas centralizados. Quando os jogadores baixam do servidor principal, eles recebem todos os jogos salvos, e não apenas o mais recente. É como se estivessem espelhando o servidor principal.

A primeira operação de clonagem pode ser bem demorada, especialmente se há um longo histórico, mas é compensada no longo prazo. Um benefício imediato é que, se por qualquer razão desejar uma antiga versão, o trafego de informação com o servidor é desnecessário.

Uma superstição

Um equívoco popular é que sistemas distribuídos estão mal adaptados a projetos que exijam um repositório central oficial. Nada poderia estar mais longe da

verdade. Fotografar alguém não irá roubar a sua alma. Igualmente, clonar o repositório master não diminui sua importância.

Uma boa comparação inicial é: qualquer coisa que um sistema centralizado de controle de versões faz, um sistema distribuído de controle de versões bem concebido pode fazer melhor. Recursos de rede são simplesmente mais onerosos que recursos locais. Embora vejamos mais adiante que existem inconvenientes numa abordagem distribuída, é menos provável que alguém faça comparações errôneas com esta regra de ouro.

Um pequeno projeto pode precisar de apenas uma fração dos recursos oferecidos pelo sistema, mas utilizar um sistema que não permite expansão é como utilizar os algarismos romanos quando calculamos com números pequenos.

E mais, seu projeto pode crescer além da suas expectativas. Usando Git, desde o inicio é como ter um canivete suíço, embora o use na maioria das vezes para abrir garrafas. No dia que necessitar, desesperadamente, de uma chave de fenda você agradecerá por ter mais do que um abridor de garrafas.

Conflitos de mesclagem (Merge)

Neste tópico, nossa analogia com jogos de computador torna-se ruim. Em vez disso, vamos considerar novamente a edição de um documento.

Suponha que Alice insira uma linha no início do arquivo, e Bob uma no final. Ambos enviam suas alterações. A maioria dos sistemas irá de maneira automática e reativa deduzir o plano de ação: aceitando e mesclando as mudanças, assim as alterações de Alice e Bob serão aplicadas.

Agora suponha que ambos, Alice e Bob, façam alterações distintas na mesma linha. Tornando impossível resolver o conflito sem intervenção humana. A segunda pessoa, entre Alice e Bob, que enviar suas alterações será informado do conflito, e escolherá se aplica sua alteração sobre a do outro, ou revisa a linha para manter ambas as alterações.

Situações muito mais complexas podem surgir. Sistemas de controle de versões são capazes de resolver os casos mais simples, e deixar os casos mais difíceis para nós resolvermos. Normalmente seu comportamento é configurável.

Truques básicos

Ao invés de se aprofundar no mar de comandos do Git, use estes exemplos elementares para dar os primeiros passos. Apesar de sua simplicidade, cada um deles são muito úteis. Na verdade, no meu primeiro mês com o Git, nunca precisei ir além das informações deste capítulo.

Salvando estados

Pensando em tentar algo mais arriscado? Antes de fazê-lo, tire uma "fotografia" de todos os arquivos do diretório atual com:

```
$ git init
$ git add .
$ git commit -m "My first backup"
```

Assim se algo der errado, você só precisará executar:

```
$ git reset --hard
```

Para salvar o estado novamente, faça:

```
$ git commit -a -m "Another backup"
```

Adicionar, Remover, Renomear

Os comandos acima só irão verificar alterações nos arquivos que estavam presentes quando você executou seu primeiro **git add**. Se você adicionar novos arquivos ou diretórios terá que informar ao Git, com:

```
$ git add readme.txt Documentation
```

Do mesmo modo, se você quiser que o Git não verifique certos arquivos, faça:

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/
```

O Git irá apagar estes arquivos, se você ainda não apagou.

Renomear um arquivo é o mesmo que remover o nome antigo e adicionar um novo nome. Há também o atalho **git mv** que tem a mesma sintaxe do comando **mv**. Por exemplo:

```
$ git mv bug.c feature.c
```

Desfazer/Refazer avançado

Às vezes, você só quer voltar e esquecer todas as mudanças realizadas a partir de um certo ponto, pois estão todos erradas. Então:

```
$ git log
```

mostrará uma lista dos últimos commit e seus hash SHA1:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800
```

```
    Replace printf() with write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Initial commit.
```

Os primeiros caracteres do hash são suficientes para especificar o commit; alternativamente, copie e cole o hash completo. Digite:

```
$ git reset --hard 766f
```

para restaurar ao estado de um dado commit e apagar permanentemente os registros de todos os novos commit a partir deste ponto.

Outras vezes você quer voltar, brevemente, para um estado. Neste caso, digite:

```
$ git checkout 82f5
```

Isto levará você de volta no tempo, preservando os novos commit. Entretanto, como nas viagens no tempo dos filmes de ficção, se você editar e fizer um commit, você estará numa realidade alternativa, pois suas ações são diferentes das realizadas da primeira vez.

Esta realidade alternativa é chamada de branch, nós falaremos mais sobre isso depois. Por ora, apenas lembre-se que:

```
$ git checkout master
```

lhe levará de volta para o presente. Assim faça o Git parar de reclamar, sempre faça um commit ou reset em suas alterações antes de executar um checkout.

Voltemos para a analogia dos jogos de computador :

- `git reset --hard`: carrega um salvamento antigo e apaga todos jogos salvos mais novos do que este que foi carregado.
- `git checkout`: carrega um salvamento antigo, mas se jogar a partir dele, os próximos salvamento realizados se desvincularão dos salvamentos já realizados após o que foi carregado. Qualquer jogo salvo que você fizer será colocado em um branch separado representado uma realidade alternativa em que entrou. Lidaremos com isso mais adiante. We deal with this later.

Você pode escolher restaurar apenas alguns arquivos ou diretórios acrescentando-os ao final do comando:

```
$ git checkout 82f5 some.file another.file
```

Tome cuidado, pois esta forma de **checkout** pode sobrescrever arquivos sem avisar. Para prevenir acidentes, faça um commit antes de qualquer comando checkout, especialmente quando esta aprendendo a utilizar o Git. De modo

geral, sempre que se sentir inseguro sobre alguma operação, seja um comando Git ou não, execute primeiro o comando **git commit -a**.

Não gosta de copiar e colar hash? Então use:

```
$ git checkout :/"My first b"
```

para ir ao commit que começa a frase informada. Você também pode solicitar pelo estado salvo ha 5 commit atrás:

```
$ git checkout master~5
```

Revertendo

Como num tribunal, eventos podem ser retirados dos registros. Da mesma maneira, você pode especificar qual commit desfazer.

```
$ git commit -a
$ git revert 1b6d
```

irá desfazer apenas o commit do hash informado. A regressão é gravada como um novo commit, e que pode ser confirmada executando o comando **git log**.

Geração do Changelog

Alguns projetos precisam de um changelog. Podemos gerar um changelog com o comando:

```
$ git log > ChangeLog
```

Download de arquivos

Para obter uma cópia de um projeto gerenciado com GIT digite:

```
$ git clone git://server/path/to/files
```

Por exemplo, para obter todos os arquivos usados para criar este site:

```
$ git clone git://git.or.cz/gitmagic.git
```

Mais adiante, teremos muito o que dizer sobre o comando **clone**.

A Última Versão

Se você já obteve a cópia de um projeto usando **git clone**, pode agora atualizar para a última versão com:

```
$ git pull
```

Publicação instantânea

Suponha que você tenha escrito um script e gostaria de compartilhá-lo. Você poderia simplesmente dizer para pegarem do seu computador, mas se o fizerem enquanto você está melhorando o script ou experimentando algumas mudanças, eles podem ter problemas. Obviamente, é por isso que existem ciclos de liberação. Desenvolvedores podem trabalhar num projeto com frequência, mas só disponibilizam o código quando sentem que o mesmo está apresentável.

Para fazer isso com Git, no diretório onde está seu script, execute:

```
$ git init
$ git add .
$ git commit -m "First release"
```

Então avise aos outros para executarem:

```
$ git clone your.computer:/path/to/script
```

para obter seu script. Assume-se que eles têm acesso ssh. Se não, execute **git daemon** e avise-os para executar:

```
$ git clone git://your.computer/path/to/script
```

A partir de agora, toda vez que seu script estiver pronto para liberar, execute:

```
$ git commit -a -m "Next release"
```

e seu usuários podem atualizar suas versões, indo para o diretório que contém seu script, e executando:

```
$ git pull
```

Seu usuários nunca ficarão com uma versão do seu script que você não queira. Obviamente este truque serve para tudo, não apenas script.

O que eu fiz?

Saiba quais as mudanças que você fez desde o último commit com:

```
$ git diff
```

Ou desde ontem:

```
$ git diff "@{yesterday}"
```

Ou entre uma versão particular e duas versões atrás:

```
$ git diff 1b6d "master~2"
```

Em cada um dos exemplos, a saída será um patch que pode ser aplicado com o **git apply**. Tente também:

```
$ git whatchanged --since="2 weeks ago"
```

Às vezes navego pelo histórico com o qgit, em razão de sua interface mais fotogênica, ou com o tig, uma interface em modo texto ótima para conexões lentas. Alternativamente, instale um servidor web, e execute git instaweb e use um navegador.

Exercícios

Seja A, B, C D quatro commits sucessivos onde B é idêntico a A exceto por alguns arquivos que foram removidos. Queremos adicionar novamente os arquivos em D. Como podemos fazer isso?

Existem no minimo 3 soluções. Assumindo que estamos em D.

1. A diferença entre A e B são or arquivos removidos. Podemos criar um patch representando esta diferença e aplicá-la:

   ```
   $ git diff B A | git apply
   ```

2. Como salvamos os arquivos em A, podemos recuperá-los:

   ```
   $ git checkout A foo.c bar.h
   ```

3. Podemos visualizar as mudanças de A para B que queremos desfazer:

   ```
   $ git revert B
   ```

Qual a opção é melhor? A que você preferir, É fácil fazer o que você quer com o git, e na maioria das vezes existe mais de uma forma de fazê-lo.

Um pouco de clonagem

Em sistemas de controle de versões mais antigos, checkout é a operação padrão para se obter arquivos. Obtendo assim os arquivos do ponto de salvamento informado.

No Git e em outros sistemas distribuídos de controle de versões, clonagem é a operação padrão. Para obter os arquivos, criamos um clone do repositório inteiro. Em outras palavras, você praticamente faz um espelhamento do servidor central. Tudo o que se pode fazer no repositório principal, você pode fazer no seu repositório local.

Sincronizando Computadores

Eu posso aguentar fazer tarball1 ou usar o **rsync** para backup e sincronizações básicas. Mas, às vezes edito no meu laptop, outras no meu desktop, e os dois podem não ter conversado entre si nesse período.

Inicialize um repositório Git e faça um commit seus arquivos em uma das máquinas. Então faça na outra máquina:

```
$ git clone other.computer:/path/to/files
```

para criar uma segunda cópia dos seus arquivos e do repositório Git. A partir de agora, use:

```
$ git commit -a
$ git pull other.computer:/path/to/files HEAD
```

o que deixará os arquivos da máquina em que você está trabalhando, no mesmo estado que estão no outro computador. Se você recentemente fez alguma alteração conflitante no mesmo arquivo, o Git lhe informará e você poderá fazer um novo commit e então escolher o que fazer para resolvê-lo.

Controle clássico de código

Inicialize um repositório Git para seus arquivos:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

No servidor principal, inicialize um *repositório vazio* do Git em algum diretório:

```
$ mkdir proj.git
$ cd proj.git
$ git --bare init
$ touch proj.git/git-daemon-export-ok
```

E inicie o daemon Git se necessário:

```
$ git daemon --detach  # it may already be running
```

Algumas hospedagens públicas, siga as instruções para configuração de um repositório git vazio. Na maioria das vezes, isso é feito através do preenchimento de um formulário no site deles.

Envie (*Push*) seu projeto para o servidor principal com:

```
$ git push central.server/path/to/proj.git HEAD
```

Para verificar os fontes, um desenvolvedor pode digitar:

```
$ git clone central.server/path/to/proj.git
```

Após realizar as alterações, o desenvolvedor pode salvar as alterações localmente com:

```
$ git commit -a
```

Para atualizar para a ultima versão:

```
$ git pull
```

Qualquer conflito de merge deve ser resolvido e então feito o commit:

```
$ git commit -a
```

Para verificar as mudanças locais no repositório central:

```
$ git push
```

Se o servidor principal possui novas alterações devido a atividades de outros desenvolvedores, o push irá falhar, e o desenvolvedor deverá fazer o pull da ultima versão, resolver qualquer conflito de merge, e então tentar novamente.

Os desenvolvedores devem ter acesso a SSH para utilizar os comandos de push e pull acima. Entretanto qualquer pessoa pode examinar os arquivos-fonte, digitando:

```
$ git clone git://central.server/path/to/proj.git
```

O protocolo nativo do Git é semelhante ao HTTP, não existe nenhum tipo de autenticação, de modo que qualquer um pode baixar o projeto. Da mesma maneira, o push, por default, é proibido pelo protocolo Git.

Codigo-fonte secreto

Para um projeto que não seja de código aberto, omita o comando touch e garanta que você nunca irá criar um arquivo com o nome `git-daemon-export-ok`. O repositório não poderá ser baixado via protocolo git; somente aqueles com acesso SSH poderão visualiza-lo. Se todos os seus repósitórios forem fechados, não é necessária a execução do daemon do git, pois todas as comunicações irão ocorrer por meio do SSH.

Repositorios Vazios

Um repositório é chamado de vazio (bare) porque ele não possui um diretório de trabalho; ele contém somente arquivos que normalmente estão escondidos no subdiretório `.git`. Em outras palavras, ele mantém a história de um projeto, e nunca guarda uma "fotografia" de alguma versão.

Um repositório vazio tem um papel semelhante aquele do servidor principal nos sistemas de controle de versão centralizados: o diretório principal (home) de seu projeto. Os desenvolvedores clonam seu projeto a partir dele, e fazem o push da ultima versão oficial para ele. Geralmente, ele reside em um servidor que somente dissemina os dados. O desenvolvimento ocorre nos clones, de modo que o repositório principal (home) pode funcionar sem um diretório de trabalho.

Muitos comandos git falham em repositórios vazios a não ser que a variável de ambiente `GIT_DIR` seja configurada para o path do repositório, ou a opção `--bare` seja fornecida.

Push versus Pull

Por que nós falamos do comando push, ao invés de basearmos no familar comando pull? Em primeiro lugar, porque o pulling falha em repositórios vazios: ao contrário você deve fazer um fetch, um comando que será discutido mais adiante. Mas mesmo que utilizamos um repositório normal em um servidor centralizado, fazer o pull para ele ainda será problemático. Primeiro, teremos que fazer o login no servidor, e então entrar com o comando pull com o endereço de rede da máquina que estamos fazendo o pull. Os firewall podem interferir, e o que dizer quando não temos acesso a uma janela de shell do servidor?

Entretanto, além desse caso, desencorajamos o push em um repositório , por causa da confusão que pode ocorrer quando o destino possui um diretório de trabalho.

Em resumo, enquanto estiver aprendendo a utilizar o git, somente faça push quando o alvo for um repositório vazio, caso contrário utilize o pull.

Fazendo um Fork do Projeto

Chateado com a rumo que o seu projeto está tomando? Acha que pode fazer um trabalho melhor? Então no seu servidor:

```
$ git clone git://main.server/path/to/files
```

Em seguida avise a todos sobre seu fork do projeto no seu servidor.

A qualquer hora, você poderá mesclar (merge) suas mudanças do projeto original no mesmo com:

```
$ git pull
```

Backup Supremos

Gostaria de ter vários arquivos geográficamente dispersos, redundantes e anti-falsificações? Se seu projeto tem muitos desenvolvedores, não faça nada! Cada clonagem do seu código é um backup efetivo. E não apenas uma cópia do estado atual, e sim o histórico completo do seu projeto. Graças ao hash criptográfico, se alguma clonagem for corrompida, ela será identificada assim que tentar se comunicar com as outras.

Se seu projeto não é tão popular, encontre quantos servidores puder para hospedar seus clones.

Um paranóico verdadeiro sempre anotará os últimos 20 byte do hash SHA1 do cabeçalho (HEAD) em algum lugar seguro. Tem que ser seguro, e não privado. Por exemplo, publicá-lo em um jornal funciona bem, pois é muito difícil para um atacante alterar todas as cópias de um jornal.

Multitarefa na velocidade da luz

Digamos que você queira trabalhar em diversas funções em paralelo. Então, faça um commit do seu projeto executando:

```
$ git clone . /some/new/directory
```

Graças aos hardlinking, os clones locais necessitam de menos tempo e espaço do que os backups comuns.

Agora você pode trabalhar em duas funções independentes de forma simultânea. Por exemplo, pode editar um clone enquanto o outro está sendo compilado. A qualquer momento, você pode fazer um commit e pegar (pull) as alterações de outro clone:

```
$ git pull /the/other/clone HEAD
```

Controle de Versões de Guerrilha

Você está trabalhando em um projeto que utiliza outro sistema de controle de versões, e sente saudade do Git? Inicialize um repositório Git no seu diretório de trabalho:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Faça um clone dele:

```
$ git clone . /some/new/directory
```

Agora vá para o novo diretório e trabalhe nele, não no anterior, usando Git para felicidade geral da nação. De vez em quando você desejará sincronizar com os outros, neste caso, vá para o diretório original, sincronize usando o outro sistema de controle de versões, e então digite:

```
$ git add .
$ git commit -m "Sync with everyone else"
```

Depois vá para o novo diretório e execute:

```
$ git commit -a -m "Description of my changes"
$ git pull
```

O procedimento para enviar suas alterações para os outros depende do outro sistema de controle de versões. O novo diretório contém os arquivos com as suas alterações. Execute qualquer comando do outro sistema de controle de versões necessário para enviá-las para o repositório central.

O subversion, é talvez o melhor sistema de controle de versões centralizado, é utilizado em muitos projetos. O comando **git svn** automatiza tudo isso para repositórios Subversion, e também pode ser utilizado para exportar um repositório Git para um repositório Subversion.

Mercurial

Mercurial é um sistema de controle de versões semelhante, e que pode trabalhar perfeitamente junto com o Git. Com o plugin `hg-git`, um usuário do Mercurial pode realizar push e pulls de um repositorio Git.

Para obter o `hg-git` plugin com o Git:

```
$ git clone git://github.com/schacon/hg-git.git
```

ou no Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

Infelizmente, não conheço de um plugin semelhante para o Git. Por essa razão, aconselho o Git no lugar do Mercurial para o repositório principal, mesmo que você prefira o Mercurial. Com o Mercurial, geralmente um voluntário mantém um repositorio Git paralelo para acomodar os usuários Git, e graças ao plugin `hg-git`, um projeto Git automaticamente acomoda os usuários Mercurial.

Embora o plugin converta um repositorio Mercurial em um repositório Git fazendo o push para um repositório vazio, esse serviço é mais fácil de fazer com o script `hg-fast-export.sh` disponível em:

```
$ git clone git://repo.or.cz/fast-export.git
```

Para fazer a conversão, em um diretório vazio, execute:

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

após adicionar o script ao seu `$PATH`.

Bazaar

Vamos mencionar o Bazaar rapidamente por que é o sistema de controle de versões distribuído mais utilizado, depois do Git e do Mercurial.

Bazaar tem a vantagem de retrospectiva, já que é relativamente recente; seus projetistas puderam aprender com os erros dos projetos anteriores, e evitar as

principais falhas. Além disso, seus desenvolvedores estão preocupados com a portabilidade e interoperação com outros sistemas de controle de versão.

O plugin `bzr-git` permite que os usuários do Bazaar trabalhem com os repositorios Git de alguma forma. O programa `tailor` converte um repositório Bazaar em repositórios Git, e pode fazer isso de forma incremental, enquanto que `bzr-fast-export` é adequada para realizar conversões uma única vez.

Por que uso o Git?

Originalmente, escolhi o Git porque escutei que poderia gerenciar o inimaginável e ingerenciável código fonte do kernel do Linux. E nunca senti necessidade de mudar. O Git tem me servido admiravelmente bem, e já estou acostumado com suas falhas. Como utilizo na maior parte do tempo o Linux, os problemas nas outras plataformas não são importantes.

Também, prefiro programas em C e scripts bash a executáveis tais como scripts Python: existem poucas dependências, e sou viciado em tempos de execução muito rápidos.

Já pensei também como o Git poderia ser melhorado, criando minha própria ferramenta Git, mas somente como um exercício académico. Assim que completei meu projeto, continuei com o git, pois os ganhos seriam mínimos para justificar a utilização de um sistema diferente.

Naturalmente, você pode e precisa pensar diferente, e pode se sentir melhor com outro sistema. No entanto, você não pode estar muito errado com o Git.

Bruxarias com branch

Ramificações (Branch) e mesclagens (merge) instantâneos são as características mais fantásticas do Git.

Problema: Fatores externos inevitavelmente exigem mudanças de contexto. Um erro grave que se manifesta sem aviso, em uma versão já liberada. O prazo final é diminuído. Um desenvolvedor que o ajuda, em uma função chave do seu projeto, precisa sair. Em todos esses casos, você deixará de lado bruscamente o que esta fazendo e focará em uma tarefa completamente diferente.

Interromper sua linha de pensamento provavelmente prejudicará sua produtividade, e quanto mais trabalhoso for trocar de contexto, maior será a perda. Com um controle de versões centralizado precisamos pegar uma nova cópia do servidor central. Sistemas distribuídos fazem melhor, já que podemos clonar o que quisermos localmente.

Mais clonar ainda implica copiar todo o diretório de trabalho, bem como todo o histórico até o ponto determinado. Mesmo que o Git reduza o custo disso com

o compartilhamento de arquivos e hardlinks, os arquivos do projeto devem ser recriados completamente no novo diretório de trabalho.

Solução: O Git tem a melhor ferramenta para estas situações que é muito mais rápida e mais eficiente no uso de espaço do que a clonagem: **git branch**.

Com esta palavra mágica, os arquivos em seu diretório de repente mudam de forma, de uma versão para outra. Esta transformação pode fazer mais do que apenas avançar ou retroceder no histórico. Seus arquivos podem mudar a partir da última liberação para a versão experimental, para a versão atualmente em desenvolvimento, ou para a versão dos seus amigos, etc.

A "tecla" chefe

Sempre joguei um desses jogos que ao apertar de um botão ("a tecla chefe"), a tela instantaneamente mudará para uma planilha ou algo mais sério. Assim se o chefe passar pelo seu escritório enquanto você estiver jogando, poderá rapidamente esconder o jogo.

Em algum diretório:

```
$ echo "I'm smarter than my boss" > myfile.txt
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Criamos um repositório Git que rastreará um arquivo texto contendo uma certa mensagem. Agora digite:

```
$ git checkout -b boss  # nothing seems to change after this
$ echo "My boss is smarter than me" > myfile.txt
$ git commit -a -m "Another commit"
```

ficou parecendo que nós sobrescrevemos nosso arquivo e fizemos um commit. Mas isto é um ilusão. Digite:

```
$ git checkout master  # switch to original version of the file
```

e tcham tcham tcham! O arquivo texto foi restaurado. E se o chefe decidir bisbilhotar este diretório. Digite:

```
$ git checkout boss  # switch to version suitable for boss' eyes
```

Você pode trocar entre as duas versões do arquivo quantas vezes quiser, e fazer commit independentes para cada uma.

Trabalho porco

Digamos que você está trabalhando em alguma função, e por alguma razão, precisa voltar 3 versões, e temporariamente colocar algumas declarações de controle para ver como algo funciona. Então:

```
$ git commit -a
$ git checkout HEAD~3
```

Agora você pode adicionar temporariamente código feio em qualquer lugar. Pode até fazer commit destas mudanças. Quando estiver tudo pronto,

```
$ git checkout master
```

para voltar para o trabalho original. Observe que qualquer mudança sem commit são temporárias.

E se você desejasse salvar as mudanças temporárias depois de tudo? Fácil:

```
$ git checkout -b dirty
```

e faça commit antes de voltar ao branch master. Sempre que quiser voltar à sujeira, simplesmente digite:

```
$ git checkout dirty
```

Nós já falamos deste comando num capítulo anterior, quando discutimos carregamento de estados antigos salvos. Finalmente podemos contar toda a história: os arquivos mudam para o estado requisitado, porém saímos do branch master. Cada commit realizado a partir deste ponto nos seus arquivos o levarão em outra direção, que nomearemos mais adiante.

Em outra palavras, depois de fazer checkout em um estado antigo, o Git automaticamente o colocará em um novo branch não identificado, que pode ser identificado e salvo com **git checkout -b**.

Correções rápidas

Você está fazendo algo quando mandam largar o que quer que seja e corrigir um erro recém-descoberto no commit `1b6d...`:

```
$ git commit -a
$ git checkout -b fixes 1b6d
```

Então assim que tiver corrigido o erro:

```
$ git commit -a -m "Bug fixed"
$ git checkout master
```

e volte a trabalhar no que estava fazendo anteriormente. Você pode até fazer um merge na correção realizada:

```
$ git merge fixes
```

Merging

Com alguns sistemas de controle de versões, a criação de ramos (branching) é fácil mas realizar o merge de volta é dificil. Com o Git, o merge é tão trivial que você pode nem perceber que ele acontece.

Nós, na realidade encontramos o merge há bastante tempo. O comando **pull** na realidade *busca* (*fetches*) um commit e faz um merge no ramo (branch) atual. Se você não tem nenhuma alteração local, então ele faz um merge rápido, que é um caso especial parecido a buscar a ultima versão em um sistema de controle de versões centralizado. Mas se você tem mudanças locais, o Git irá automaticamente fazer o merge, e reportar qualquer conflito.

Comumente, um commit possui exatamente um *commit pai*, que recebe o nome de commit anterior. Fazer o merge de ramos (branches) juntos produz um commit com no mínimo dois pais. Isso levanta a questão: a que commit `HEAD~10` estamos nos referindo? Um commit pode ter vários pais, e qual deles vamos seguir?

Acontece que essa notação escolhe o primeiro pai a cada vez. Isso é o desejável porque o ramo atual se torna o primeiro pai durante o merge; frequentemente você estará preocupado com as alterações que realizou no ramo atual, em oposição às alterações merged de outros ramos.

Podemos referenciar um pai especifico com um circunflexo. Por exemplo, para mostrar os logs do segundo pai:

```
$ git log HEAD^2
```

Podemos omitir o número para o primeiro pai. Por exemplo, para mostrar as diferenças com o primeiro pai:

```
$ git diff HEAD^
```

Podemos combinar essa notação com outras. Por exemplo:

```
$ git checkout 1b6d^^2~10 -b ancient
```

inicia um novo ramo “ancient” representando o estado 10 commit atrás para o segundo pai do primeiro pai do commit iniciando com 1b6d.

Fluxo ininterrupto

Frequentemente em projetos de hardware, o segundo passo de um plano deve esperar que o primeiro passo termine. Um carro que esta esperando uma peça do fabricante deve permanecer na oficina até que a peça chegue. Um protótipo deve esperar até que o chip seja fabricado para que a montagem possa continuar.

Os projetos de software pode ser semelhantes a isso. A segunda parte de uma nova função pode ter que esperar até que a primeira parte seja testada e liberada. Alguns projetos requerem que o código seja revisto antes de seu aceite, de modo que você deve esperar até que a primeira parte seja aprovada antes de iniciar a segunda parte.

Graças à facilidade de realizar branch e merge, podemos "desviar" das regras e trabalhar na parte 2 antes da parte 1 estar oficialmente pronta. Suponha que fizemos o commit da parte 1 e enviamos para a revisão. Vamos dizer que estamos no ramo (branch) `master`. Então podemos ramificar:

```
$ git checkout -b part2
```

Em seguida, trabalhamos na parte 2, fazendo commit das alterações durante o desenvolvimento. Mas errar é humano, e frequentemente você vai querer retornar e corrigir alguma coisa na parte 1. Se você estiver com sorte, ou for realmente bom, pode saltar esses comandos:

```
$ git checkout master  # Go back to Part I.
$ fix_problem
$ git commit -a        # Commit the fixes.
$ git checkout part2   # Go back to Part II.
$ git merge master     # Merge in those fixes.
```

Eventualmente, quando a parte 1 for aprovada:

```
$ git checkout master  # Go back to Part I.
$ submit files         # Release to the world!
$ git merge part2      # Merge in Part II.
$ git branch -d part2  # Delete "part2" branch.
```

Agora, você estará no ramo `master` novamente, com a parte 2 no diretório de trabalho.

É fácil de estender esse truque para qualquer número de partes. É fácil também fazer branching retroativos: suponha que você percebeu tardiamente que deveria ter criado um branch 7 commits atrás. Então digite:

```
$ git branch -m master part2  # Rename "master" branch to "part2".
$ git branch master HEAD~7   # Create new "master", 7 commits upstream.
```

O branch `master` agora contém somente a parte 1, e o branch `part2` contém todo o resto. Estamos no ultimo branch; criamos o `master` sem mudar para ele, porque queremos continuar a trabalhar na `part2`. Isso não é comum. Até agora, nós trocamos de ramos imediatamente após a sua criação, como em:

```
$ git checkout HEAD~7 -b master  # Create a branch, and switch to it.
```

Reorganizando uma Bagunça

Talvez você goste de trabalhar com todos os aspectos de um projeto num mesmo branch. E gostaria de manter seu trabalho para você mesmo e que os outros só vejam seus commit, apenas quando eles estiverem organizados. Inicie um par de branch:

```
$ git branch sanitized    # Create a branch for sanitized commits.
$ git checkout -b medley  # Create and switch to a branch to work in.
```

A seguir, trabalhe em alguma coisa: corrigindo erros, adicionando funções, adicionando código temporário, e assim por diante, faça commit muitas vezes ao longo do caminho. Então:

```
$ git checkout sanitized
$ git cherry-pick medley^^
```

aplique o comit avô ao commit HEAD do ramo "medley" para o ramo "sanitized". Com os cherry-picks apropriados você pode construir um branch que contêm apenas código permanente, e tem os commit relacionados agrupados juntos.

Gerenciando os Branches

Para listar todos os branch, digite:

```
$ git branch
```

Por default, você deve iniciar em um branch chamado "master". Alguns defendem que o branch "master" deve permanecer intocado e que a seja criado novos branchs para suas próprias mudanças.

As opções **-d** e **-m** permitem a você deletar ou mover (renomear) um branch. Veja **git help branch**.

O branch "master" é uma personalização útil. Outros podem assumir que seu repositório possui um branch com esse nome e que ele contém a versão oficial de seu projeto. Embora possamos renomear ou apagar o branch "master", você deve procurar respeitar essa convenção.

Branches Temporários

Depois de um tempo você perceberá que está criando um branch de curta duração, frequentemente e por motivos parecidos: cada novo branch serve apenas para guardar o estado atual, assim você pode rapidamente voltar para estados antigos para corrigir um erro ou algo assim.

É semelhante a mudar o canal da TV temporariamente para ver o que está passando nos outros canais. Mas, ao invés de apertar dois botões, você está

criando, checando e apagando branchs temporários e seus commit. Felizmente, o Git tem um atalho que é tão conveniente como um controle remoto de TV:

```
$ git stash
```

Isto salva o estado atual num local temporário (um *stash*) e restaura o estado anterior. Seu diretório de trabalho parece ter voltado ao estado anteriormente salvo, e você pode corrigir erros, puxar as mudanças mais novas, e assim por diante. Quando quiser retornar ao estado anterior ao uso do stash, digite:

```
$ git stash apply  # You may need to resolve some conflicts.
```

Você pode ter múltiplos stash, e manipulá-los de várias formas. Veja **git help stash**. Como deve ter adivinhado, o Git usa branch por traz dos panos para fazer este truque.

Trabalhe como quiser

Você pode se perguntar se os ramos valem a pena. Afinal, os clones são quase tão rápidos e você pode trocar entre eles com um comando cd ao invés de um comando esotérico do Git.

Considere um navegador web. Por que suportar abas múltiplas bem como janelas múltiplas? É porque ao permitir ambas as características podemos acomodar uma variedade de estilos. Alguns usuários gostam de manter somente uma janela aberta do navegador, e utilizar varias abas para as páginas web. Outros preferem o outro extremo, várias janelas sem nenhuma aba. Outros podem preferir uma mistura dos estilos.

Branching é como as abas para seu diretório de trabalho, e o clone é como uma nova janela do navegador. Essas operações são rápidas e locais, de modo que por que não experimentar para encontrar a combinação que melhor se adequa a você? O Git permite que você trabalhe exatamente do jeito que você desejar.

Lições de historia

Uma consequência da natureza distribuída do Git é que o histórico pode ser editado facilmente. Mas se você adulterar o passado, tenha cuidado: apenas rescreva a parte do histórico que só você possui. Assim como as nações sempre argumentam sobre quem comete atrocidades, se alguém tiver um clone cuja versão do histórico seja diferente do seu, você pode ter problemas para conciliar suas árvores quando interagirem.

Alguns desenvolvedores acreditam que o histórico deva ser imutável, com falhas ou não. Outros, acreditam que suas árvores devem estar apresentáveis antes de liberá-las ao público. O Git contempla ambos pontos de vista. Tal como a

clonagem, branch e merge, rescrever o histórico é simplesmente outro poder que o Git lhe concede. Cabe a você a usá-lo sabiamente.

Eu corrijo

Acabou de fazer um commit, mas queria ter escrito uma mensagem diferente? Então execute:

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

para mudar a última mensagem. Percebeu que esqueceu de adicionar um arquivo? Execute **git add** para adicioná-lo, e então execute o comando acima.

Quer incluir mais algumas modificações no último commit? Faça-as e então execute:

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

... e tem mais

Suponha que o problema anterior é dez vezes pior. Após uma longa sessão onde você fez um monte de commit. E você não está muito satisfeito com a organização deles, e algumas das mensagens dos commit poderiam ser reformuladas. Então execute:

```
$ git rebase -i HEAD~10
```

e os últimos 10 commit aparecerão em seu $EDITOR favorito. Trecho de exemplo:

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Os commit antigos precedem os mais novos nessa lista, diferentemente do comando `log`. Aqui, 5c6eb73 é o commit mais velho e o 100834f é o commit mais novo. Então:

- Remova os commit deletando as linhas. Como o comando revert, mas sem fazer o registro: será como se o commit nunca tenha existido.
- Reorganize os commit reorganizando as linhas.
- Substitua `pick` com:
  - `edit` para modificar a mensagem do commit;
  - `reword` para alterar a mensagem de log;
  - `squash` para fazer o merge de um commit com o commit anterior;

– `fixup` para fazer o merge de um commit com o anterior e descartar a mensagem de log.

Por exemplo, queremos substituir o segundo `pick` por `squash`:

```
pick 5c6eb73 Added repo.or.cz link
squash a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Após salvar e sair, o Git ira fazer o merge a311a64 com o 5c6eb73. Assim **squash** faz o merge no próximo commit: pense como "squash up".

O Git, então combina suas mensagens de logs e apresenta para edição. O comando **fixup** salta essa etapa; a mensagem de log squashed é simplesmente descartada.

Se marcar um commit com **edit**, o Git retorna você ao passado, ao commit mais velho. Você pode emendar o commit velho como descrito na seção anterior, e até mesmo criar um novo commit que pertença a esse. Uma vez que esteja satisfeito com o "retcon", siga adiante executando:

```
$ git rebase --continue
```

O Git reenvia os commits até o próximo **edit**, ou ao presente se não restar nenhum.

Você pode também, abandonar o rebase com:

```
$ git rebase --abort
```

Portanto, faça commit cedo e com frequência: e arrume tudo facilmente mais tarde com um rebase.

Alterações locais por último

Você está trabalhando em um projeto ativo. Faz alguns commit locais ao longo do tempo, e sincroniza com a árvore oficial com merge. Este ciclo se repete algumas vezes até estar tudo pronto para ser enviado à árvore central.

Mas agora o histórico no seu clone local está uma confusão com o emaranhado de modificações locais e oficiais. Você gostaria de ver todas as suas modificações em uma seção contínua e depois todas as modificações oficiais.

Este é um trabalho para **git rebase** conforme descrito acima. Em muitos casos pode-se usar a opção **--onto** e evitar sua interação.

Veja também **git help rebase** com exemplos detalhados deste incrível comando. Você pode dividir commit. Ou até reorganizar branch de uma árvore.

Tome cuidado: o comando rebase é muito poderoso. Para rebases complicados, primeiro faça um backup com **git clone**.

Reescrevendo o histórico

Eventualmente, será necessário que seu controle de código tenha algo equivalente ao modo Stanlinesco de retirada de pessoas das fotos oficiais, apagando-os da história. Por exemplo, suponha que temos a intenção de lançar um projeto, mas este envolve um arquivo que deve ser mantido privado por algum motivo. Talvez eu deixe meu número do cartão de crédito num arquivo texto e acidentalmente adicione-o ao projeto. Apagá-lo é insuficiente, pois, pode ser acessado pelos commit anteriores. Temos que remover o arquivo de todos os commit:

```
$ git filter-branch --tree-filter 'rm top/secret/file' HEAD
```

Veja **git help filter-branch**, que discute este exemplo e mostra um método mais rápido. No geral, **filter-branch** permite que você altere grandes seções do histórico só com um comando.

Depois, o diretório `.git/refs/original` descreve o estado dos casos antes da operação. Verifique se o comando filter-branch faz o que você deseja, e então apague esse diretório se você deseja executar mais comandos filter-branch.

Por ultimo, você deve substituir os clones do seu projeto pela versão revisada se desejar interagir com eles depois.

Fazendo história

Quer migrar um projeto para Git? Se ele for gerenciado por um algum dos sistemas mais conhecidos, então é possível que alguém já tenha escrito um script para exportar todo o histórico para o Git.

Caso contrário, dê uma olhada em **git fast-import**, que lê um texto num formato especifico para criar o histórico Git do zero. Normalmente um script usando este comando é feito as pressas sem muita frescura e é executado apenas uma vez, migrando o projeto de uma só vez.

Por exemplo, cole a listagem a seguir num arquivo temporário, como `/tmp/history`:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
```

```
  printf("Hello, world!\n");
  return 0;
}
EOT


commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Replace printf() with write().
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

Em seguida crie um repositório Git a partir deste arquivo temporário digitando:

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

Faça um checkout da última versão do projeto com:

```
$ git checkout master .
```

O comando **git fast-export** converte qualquer repositório para o formato do **git fast-import** format, cujo resultado você pode estudar para escrever seus exportadores, e também para converter repositórios Git para um formato legível aos humanos. Na verdade, estes comandos podem enviar repositórios de arquivos de texto por canais exclusivamente textuais.

Onde foi que tudo deu errado?

Você acabou de descobrir uma função errada em seu programa, que você sabe com certeza que estava funcionando há alguns meses atrás. Merda! Onde será que este erro começou? Se só você estivesse testando a funcionalidade que desenvolveu.

Agora é tarde para reclamar. No entanto, se você estiver fazendo commit, o Git pode localizar o problema:

```
$ git bisect start
```

```
$ git bisect bad HEAD
$ git bisect good 1b6d
```

O Git verifica um estado intermediário entre as duas versões. Testa a função, e se ainda estiver errada:

```
$ git bisect bad
```

Senão, substitua "bad" por "good". O Git novamente o levará até um estado intermediário entre as versões definidas como good e bad, diminuindo as possibilidades. Após algumas iterações, esta busca binária o guiará até o commit onde começou o problema. Uma vez terminada sua investigação, volte ao estado original digitando:

```
$ git bisect reset
```

Ao invés de testar todas as mudanças manualmente, automatize a busca com:

```
$ git bisect run my_script
```

O Git usa o valor de retorno do comando utilizado, normalmente um único script, para decidir se uma mudança é good ou bad: o comando deve terminar retornando com o código 0 se for good, 125 se a mudança for ignorável e qualquer coisa entre 1 e 127 se for bad. Um valor negativo abortará a bissecção.

Podemos fazer muito mais: a página de ajuda explica como visualizar as bissecções, examinar ou reproduzir o log da bissecção, e eliminar mudanças reconhecidamente inocentes para acelerar a busca.

Quem fez tudo dar errado?

Tal como outros sistema de controle de versões, o Git tem um comando blame (culpado):

```
$ git blame bug.c
```

que marca cada linha do arquivo mostrando quem o modificou por último e quando. Ao contrário de outros sistemas de controle de versões, esta operação ocorre offline, lendo apenas do disco local.

Experiência pessoal

Em um sistema de controle de versões centralizado, modificações no histórico são operações difíceis, e disponíveis apenas para administradores. Clonagem, branch e merge são impossíveis sem uma rede de comunicação. Bem como as operações básicas: navegar no histórico ou fazer commit das mudanças. Em alguns sistemas, é exigido do usuário uma conexão via rede, apenas para visualizar suas próprias modificações ou abrir um arquivo para edição.

Sistemas centralizados impedem o trabalho offline, e exigem um infraestrutura de rede mais cara, especialmente quando o número de desenvolvedores aumenta. Mais importante, todas a operações são mais lentas, até certo ponto, geralmente até o ponto onde os usuários evitam comandos mais avançados até serem absolutamente necessários. Em casos extremos, esta é a regra até para a maioria dos comandos básicos. Quando os usuários devem executar comandos lento, a produtividade sofre por causa de uma interrupção no fluxo de trabalho.

Já experimentei este fenômeno na pele. O Git foi o primeiro sistema de controle de versões que usei. E rapidamente cresci acostumado a ele, tomando muitas de suas características como normais. Simplesmente assumi que os outros sistemas eram semelhante: escolher um sistema de controle de versões deveria ser igual a escolher um novo editor de texto ou navegador para internet.

Fiquei chocado quando, posteriormente, fui forçado a usar um sistema centralizado. Uma conexão ruim com a Internet pouco importa com o Git, mas torna o desenvolvimento insuportável quando precisa ser tão confiável quanto o disco local. Além disso, me condicionava a evitar determinados comandos devido a latência envolvida, o que me impediu, em última instância, de continuar seguindo meu fluxo de trabalho.

Quando executava um comando lento, a interrupção na minha linha de pensamento causava um enorme prejuízo. Enquanto espero a comunicação com o servidor concluir, faço algo para passar o tempo, como checar email ou escrever documentação. Na hora em que retorno a tarefa original, o comando já havia finalizado há muito tempo, e perco mais tempo lembrando o que estava fazendo. Os seres humanos são ruins com trocas de contexto.

Houve também um interessante efeito da tragédia dos comuns: antecipando o congestionamento da rede, os indivíduos consomem mais banda que o necessário em varias operações numa tentativa de reduzir atrasos futuros. Os esforços combinados intensificam o congestionamento, encorajando os indivíduos a consumir cada vez mais banda da próxima vez para evitar os longos atrasos.

Git Multiplayer

Inicialmente, utilizei o Git em um projeto particular onde era o único desenvolvedor. Entre os comandos relacionados a natureza distribuída do Git, eu precisava somente do **pull** e **clone**, de modo a manter o mesmo projeto em vários lugares.

Mais tarde, quis publicar meu código com o Git, e incluir as alterações dos contribuidores. Tive que aprender como gerenciar projetos com vários desenvolvedores de todas as partes do mundo. Felizmente, esse é o forte do Git, e indiscutivelmente sua razão de existir.

Quem sou eu?

Cada commit tem um nome de autor e e-mail, que pode ser mostrado pelo **git log**. O Git utiliza as configurações do sistema para preencher esses campos. Para fazer o preenchimento explícito, digite:

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

Omita o flag global para configurar essas opções somente para o repositório atual.

Git sob SSH, HTTP

Suponha que você tenha acesso SSH a um servidor web, mas que o Git não esteja instalado. Embora não tão eficiente quanto seu protocolo nativo, o Git pode se comunicar via HTTP.

Baixe, compile e instale o Git em sua conta, e crie um repositório no seu diretório web:

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

Para versões mais antigas do Git, o comando copy falha e você deve executar:

```
$ chmod a+x hooks/post-update
```

Agora você pode publicar suas últimas edições via SSH de qualquer clone:

```
$ git push web.server:/path/to/proj.git master
```

E qualquer um pode obter seu projeto com:

```
$ git clone http://web.server/proj.git
```

Git sobre qualquer coisa

Deseja sincronizar os repositórios sem servidores, ou mesmo sem uma conexão de rede? Precisa improvisar durante uma emergência? Vimos que **git fast-export** e **git fast-import** podem converter repositórios para um único arquivo e vice-versa. Podemos enviar esses arquivos para outros lugares fazendo o transporte de repositórios git sob qualquer meio, mas uma ferramenta mais eficiente é o **git bundle**.

O emissor cria um *bundle*:

```
$ git bundle create somefile HEAD
```

E então transporta o pacote, `somefile`, para outro lugar: por e-mail, pen-drive (ou mesmo uma saida impressa **xxd** e um scanner com ocr, sinais binários pelo telefone, sinais de fumaça, etc). O receptor recupera os commits do pacote executando:

```
$ git pull somefile
```

O receptor pode inclusive fazer isso em um diretório vazio. Apesar do seu tamanho, `somefile` contém todo o repositório git original.

Em grandes projetos, podemos eliminar o desperdício fazendo o empacotamento somente das mudanças que o outro repositório necessita. Por exemplo, suponha que o commit "1b6d…" é o commit mais recente compartilhado por ambas as partes:

```
$ git bundle create somefile HEAD ^1b6d
```

Se executado frequentemente, podemos esquecer que commit foi feito por último. A página de ajuda sugere utilizar tags para resolver esse problema. Isto é, após enviar um pacote, digite:

```
$ git tag -f lastbundle HEAD
```

e crie um novo pacote de atualização com:

```
$ git bundle create newbundle HEAD ^lastbundle
```

Patches: A moeda Universal

Patches são representações textuais das suas mudanças que podem ser facilmente entendidas pelos computadores e pelos seres humanos. Isso tem um forte apelo. Você pode mandar um patch por e-mail para os desenvolvedores não importando que sistema de versão eles estão utilizando. Como os desenvolvedores podem ler o e-mail, eles podem ver o que você editou.

Da mesma maneira, do seu lado, tudo o que você precisa é de uma conta de e-mail: não é necessário configurar um repositório online do Git.

Lembrando do primeiro capítulo:

```
$ git diff 1b6d > my.patch
```

produz um patch que pode ser colado em um e-mail para discussão. Em um repositório Git, digite:

```
$ git apply < my.patch
```

para aplicar o patch.

Em configurações mais formais, quando o nome do autor e talvez sua assinatura precisem ser armazenadas, podemos gerar os patches correspondentes a partir de um certo ponto com o comando:

```
$ git format-patch 1b6d
```

Os arquivos resultantes podem ser fornecidos ao **git-send-email**, ou enviados manualmente. Você também pode especificar uma faixa de commits:

```
$ git format-patch 1b6d..HEAD^^
```

No lado do receptor, salve o e-mail como um arquivo, e entre com:

```
$ git am < email.txt
```

Isso aplica o patch de entrada e também cria um commit, incluindo as informações tais como o autor.

Com um navegador cliente de e-mail, pode ser necessário clicar o botão para ver o e-mail em seu formato original antes de salvar o patch para um arquivo.

Existem pequenas diferenças para os clientes de e-mail baseados em mbox, mas se você utiliza algum desses, provavelmente é o tipo de pessoa que pode resolver os problemas sem ler o manual.

Sinto muito, mas mudamos

Após fazer o clone de um repositório, executar o **git push** ou **git pull** irá automaticamente fazer o push ou pull da URL original. Como o Git faz isso? O segredo reside nas opções de configuração criadas com o clone. Vamos dar uma olhada:

```
$ git config --list
```

A opção `remote.origin.url` controla a URL de origem; "origin" é um apelido dados ao repositório fonte. Da mesma maneira que a convenção do branch "master", podemos mudar ou deletar esse apelido mas geralmente não existe um motivo para fazê-lo.

Se o repositório original for movido, podemos atualizar a URL via:

```
$ git config remote.origin.url git://new.url/proj.git
```

A opção `branch.master.merge` especifica o branch remoto default em um **git pull**. Durante a clonagem inicial, ele é configurado para o branch atual do repositório fonte, mesmo que o HEAD do repositório fonte subsequentemente mova para um branch diferente, um pull posterior irá seguir fielmente o branch original.

Essa opção somente aplica ao repositório que fizemos a clonagem em primeiro lugar, que é armazenado na opção `branch.master.remote`. Se fizermos um pull de outro repositório devemos estabelecer explicitamente que branch desejamos:

```
$ git pull git://example.com/other.git master
```

Isso explica por que alguns de nossos exemplos de pull e push não possuem argumentos.

Branches Remotos

Quando clonamos um repositório, clonamos também todos os seus branches. Voce pode não ter notado isso porque o Git sempre esconde os branches: mas podemos solicitá-los especificamente. Isso previne os branches no repositório remoto de interferir com seus branches, e também torna o Git mais fácil para os iniciantes.

Liste os branches remotos com:

```
$ git branch -r
```

Voce deve obter uma saída como:

```
origin/HEAD
origin/master
origin/experimental
```

Eles representam branches e a HEAD de um repositório remoto, e pode ser utilizado em comandos normais do Git. Por exemplo, suponha que você fez vários commits, e gostaria de comparar contra a última versão buscada (fetched). Você pode buscar nos logs pelo hash apropriado SHA1, mas é muito mais fácil digitar:

```
$ git diff origin/HEAD
```

Ou você pode ver o que o branch "experimental" contém:

```
$ git log origin/experimental
```

Remotos Múltiplos

Suponha que dois desenvolvedores estão trabalhando em nosso projeto, e que queremos manter o controle de ambos. Podemos seguir mais de um repositório a qualquer hora com:

```
$ git remote add other git://example.com/some_repo.git
$ git pull other some_branch
```

Agora temos merged em um branch a partir do segundo repositório, e temos fácil acesso a todos os branches de todos os repositórios:

```
$ git diff origin/experimental^ other/some_branch~5
```

Mas e se só queremos comparar as suas alterações sem afetar o trabalho de ambos? Em outras palavras, queremos examinar seus branches sem ter que

suas alterações invadam nosso diretório de trabalho. Então ao invés de pull, execute:

```
$ git fetch        # Fetch from origin, the default.
$ git fetch other  # Fetch from the second programmer.
```

Isso faz com que somente busque os históricos. Embora o diretório de trabalho continue intocado, podemos fazer referencia a qualquer branch de qualquer repositório em um comando Git pois agora possuímos uma copia local.

Lembre-se de nos bastidores, um pull é simplesmente um **fetch** e um **merge**. Geralmente fazemos um **pull** pois queremos fazer um merge do ultimo commit após o fetch; essa situação é uma exceção notável.

Veja **git help remote** para saber como remover repositórios remotos, ignorar certos branches e outras informações.

Minhas preferencias

Para meus projetos, eu gosto que contribuidores para preparar os repositórios a partir dos quais eu possa fazer um pull. Alguns serviços hospedeiros de Git permite que você hospede seu próprio fork de um projeto com o clique de um botão.

Após ter buscado (fetch) uma árvore, eu executo comandos Git para navegar e examinar as alterações, que idealmente estão bem organizadas e bem descritas. Faço merge de minhas próprias alterações, e talvez faça edições posteriores. Uma vez satisfeito, eu faço um push para o repositório principal.

Embora eu receba infrequentes contribuições, eu acredito que essa abordagem funciona bem. Veja esse post do blog do Linus Torvalds.

Continuar no mundo Git é mais conveniente do que fazer patch de arquivos, já que ele me salva de convertê-los para commits Git. Além disso, o Git trata dos detalhes tais como registrar o nome do autor e seu endereço de e-mail, bem como o dia e hora, alem de pedir ao autor para descrever sua própria alteração.

Grão-Mestre Git

Até agora, você deve ser capaz de navegar pelas páginas do **git help** e entender quase tudo. Entretanto, identificar o comando exato para resolver um dados problema pode ser tedioso. Talvez possa economizar algum tempo seu: a seguir estão algumas receitas que precisei no passado.

Disponibilização de Código

Para meus projetos, o Git organiza exatamente os arquivos que quero guardar e disponibilizar para os usuários. Para criar um tarball do código fonte, executo:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

Commit do que Mudou

Mostrar ao Git quando adicionamos, apagamos e/ou renomeamos arquivos pode ser problemático em alguns projetos. Em vez disso, você pode digitar:

```
$ git add .
$ git add -u
```

O Git analisará os arquivos no diretório atual e trabalhar nos detalhes, automaticamente. No lugar do segundo comando add, execute `git commit -a` se sua intenção é efetuar um commit neste momento. Veja **git help ignore** para saber como especificar os arquivos que devem ser ignorados.

Você pode executar isto em apenas um passo com:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

As opções **-z** e **-0** previnem contra os transtornos de arquivos com caracteres estranhos no nome. Note que este comando adiciona os arquivos ignorados. Logo, você pode querer usar as opções `-x` ou `-X`.

Meu Commit é muito Grande!

Você esqueceu de fazer commit por um muito tempo? Ficou codificando furiosamente e esqueceu do controle de versões até agora? Fez uma série de modificações não relacionadas entre si, pois este é seu estilo?

Não se preocupe. Execute:

```
$ git add -p
```

Para cada modificação realizada, o Git mostrará o pedaço do código alterado, e perguntará se ele deve fazer parte do próximo commit. Responda "y" (sim) ou "n" (não). Há outras opções, como o adiamento dessa decisão; digite "?" para aprender como.

Uma vez satisfeito, digite:

```
$ git commit
```

para fazer um commit exatamente com as modificações aprovadas (*staged*). Lembre-se de retirar a opção **-a**, caso contrário o commit conterá todas as modificações.

E se você tiver editado vários arquivos em vários locais? Rever cada modificação uma por uma será frustante e enfadonho. Neste caso, use **git add -i**, cuja interface é menos simples, porém mais flexível. Com algumas poucas teclas, você pode aprovar ou não vários arquivos de uma vez, ou rever e selecionar as modificações em um arquivo especifico. Alternativamente, execute **git commit --interactive** o que automaticamente efetuará seus commit assim que terminar de aprovar.

O Indice: A Área de Atuação do Git

Até agora estivemos evitando o famoso *index* do git, mas agora teremos que enfrentá-lo para explicar o tópico acima. O index é uma área de atuação temporária. O Git frequentemente atualiza os dados diretamente entre seu projeto e sua historia. Preferencialmente, Git primeiro armazena os dados no index, e então copia os dados do index para seu destino final.

Por exemplo, **commit -a** é na realidade um processo em duas etapas. A primeira etapa coloca uma "fotografia" do estado atual de cada arquivo rastreado em um índice. O segundo passo registra permanentemente a "fotografia" localizado no índice. O commit sem a opção **-a** somente executa o segundo passo, e somente faz sentido após a execução de um comando que de alguma maneira altera o índice, tal como o **git add**.

Geralmente podemos ignorar o index e supor que estamos lendo e escrevendo diretamente no histórico. Nessas ocasiões, queremos um controle mais fino, de modo que manipulamos o índice. Colocamos uma "fotografia" de algumas, mas não todas, as nossas alterações no índice, e então registramos permanentemente esta "fotografia" cuidadosamente manipulada.

Não perca a CABEÇA (HEAD)

A etiqueta HEAD (cabeçalho) é como um indicador que normalmente aponta para o último commit, avançando a cada novo commit. Alguns comandos do Git permitem movê-la. Por exemplo:

```
$ git reset HEAD~3
```

irá mover o HEAD três commit para trás. Assim todos os comandos do Git passam a agir como se você não tivesse realizados os últimos três commit, enquanto seus arquivos permanecem no presente. Consulte a página do manual para mais aplicações.

Mas como se faz para voltar para o futuro? Os últimos commit não sabem do futuro.

Se você tem o SHA1 do HEAD original então:

```
$ git reset 1b6d
```

Mas suponha que você não tenha anotado. Não se preocupe, para comandos desse tipo, o Git salva o HEAD original com uma etiqueta chamada de ORIG_HEAD, e você pode retornar são e salvo com:

```
$ git reset ORIG_HEAD
```

Explorando o HEAD

Talvez ORIG_HEAD não seja suficiente. Talvez você só tenha percebido que fez um erro descomunal e precisa voltar para um antigo commit de um branch há muito esquecido.

Por default, o Git mantém um commit por pelo menos duas semanas, mesmo se você mandou o Git destruir o branch que o contêm. O problema é achar o hash certo. Você pode procurar por todos os valores de hash em `.git/objects` e por tentativa e erro encontrar o que procura. Mas há um modo mais fácil.

O Git guarda o hash de todos os commit que ele calcula em `.git/logs`. O sub-diretório refs contêm o histórico de toda atividade em todos os branch, enquanto o arquivo `HEAD` mostra todos os valores de hash que teve. Este último pode ser usado para encontrar o hash de um commit num branch que tenha sido acidentalmente apagado.

O comando reflog fornece uma interface amigável para estes arquivos de log. Experimente

```
$ git reflog
```

Ao invés de copiar e colar o hash do reflog, tente:

```
$ git checkout "@{10 minutes ago}"
```

Ou faça um checkout do quinto último commit com:

```
$ git checkout "@{5}"
```

Leia a seção “Specifying Revisions” da ajuda com **git help rev-parse** para mais informações.

Você pode querer configurar um período mais longo de carência para condenar um commit. Por exemplo:

```
$ git config gc.pruneexpire "30 days"
```

significa que um commit apagado será permanentemente eliminado após passados 30 dias e executado o comando **git gc**.

Você também pode desativar as execuções automáticas do **git gc**:

```
$ git config gc.auto 0
```

assim os commit só serão permanentemente eliminados quando executado o **git gc** manualmente.

Baseando se no Git

Seguindo o jeito UNIX de ser, o Git permite ser facilmente utilizado como um componente de "baixo nível" para outros programas, tais como interfaces GUI, interfaces web, interfaces alternativas de linha de comando, ferramentas de gerenciamento de patchs, ferramentas de importação e conversão e outras. Na realidade, alguns comandos Git são eles mesmos, scripts apoiados em ombros de gigantes. Com pouco trabalho, você pode configurar o Git para suas preferencias.

Um truque simples é criar alias (abreviações), internas ao Git para abreviar os comandos utilizados mais frequentemente:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # display current aliases
alias.co checkout
$ git co foo                              # same as 'git checkout foo'
```

Outra é imprimir o branch atual no prompt, ou no título da janela. É só executar:

```
$ git symbolic-ref HEAD
```

que mostra o nome do branch atual. Na prática, muito provavelmente você não quer ver o "refs/heads/" e ignorar os erros:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

O subdiretório `contrib` é um baú do tesouro de ferramentas construídas com o Git. Com o tempo algumas delas devem ser promovidas para comandos oficiais. Nos sistemas Debian e Ubuntu esse diretório esta em `/usr/share/doc/git-core/contrib`.

Um residente popular é `workdir/git-new-workdir`. Por meio de um inteligente link simbólico, este script cria um novo diretório de trabalho cujo histórico é compartilhado com o repositório original:

```
$ git-new-workdir an/existing/repo new/directory
```

O novo diretório e arquivos dentro dele podem ser vistos como um clone, exceto que como o histórico é compartilhado, as duas arvores automaticamente estarão em sincronia. Não é necessário fazer merge, push ou pull.

Manobras Radicais

As versões recentes do Git tornaram mais difícil para o usuário destruir acidentalmente um dado. Mas se você sabe o que esta fazendo, você pode transpor estas salvaguardas utilizadas nos comandos mais comuns.

Checkout: Se há modificações sem commit, um checkout simples falhará. Para destruir estas modificações, e fazer um checkout de um certo commit assim mesmo, use a opção force (-f):

```
$ git checkout -f HEAD^
```

Por outro lado, se for especificado algum endereço em particular para o checkout, então não haverá checagem de segurança. O endereço fornecido será silenciosamente sobrescrito. Tenha cuidado se você usa o checkout desse jeito.

Reset: O Reset também falha na presença de modificações sem commit. Para obrigá-lo, execute:

```
$ git reset --hard 1b6d
```

Branch: Apagar um branch falha se isto levar a perda das modificações. Para forçar, digite:

```
$ git branch -D dead_branch  # instead of -d
```

Analogamente, a tentativa de sobrescrever um branch movendo-o falha for causar a perda de dados. Para forçar a movimentação do branch, use:

```
$ git branch -M source target  # instead of -m
```

Ao contrário do checkout e do reset, estes dois comandos adiarão a destruição dos dados. As modificações ainda serão armazenadas no subdiretório .git, e podem ser resgatados, recuperando o hash apropriado do `.git/logs` (veja a seção "Explorando o HEAD" acima). Por padrão, eles serão mantidos por pelo menos duas semanas.

Clean: Alguns comandos do Git recusam-se a avançar devido o receio de sobrescrever arquivos não "monitorados" (sem commit). Se você tiver certeza de que todos os arquivos e diretórios não monitorados são dispensáveis, então apague-os sem misericórdia com:

```
$ git clean -f -d
```

Da próxima vez, o maldito comando não se recusará a funcionar!

Prevenção a maus Commits

Erros estúpidos poluem meus repositórios. Os mais assustadores são os arquivos perdidos devido a comandos **git add** esquecidos. Transgressões mais leves são

espaço em branco e conflitos de merge não resolvidos: embora sem perigo, eu gostaria que eles nunca aparecessem nos meus registros públicos.

Se eu tivesse comprado um seguro idiota usando *hook* para me alertar sobre esses problemas:

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # Older Git versions: chmod +x pre-commit
```

Agora o Git aborta um commit se espaço em branco sem utilidade ou um conflito de merge não resolvido for detectado.

Para este guia, eu eventualmente adiciono o seguinte ao inicio do hook **pre-commit** para proteção contra as "mentes sem noção".

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

Várias operações do Git suportam o hook: veja **git help hooks**. Nós ativamos o hook de exemplo **post-update** anteriormente quando discutimos o Git sob HTTP. Isso executa sempre que o HEAD se movimenta. O script de exemplo post-update atualiza os arquivos que o Git necessita para a comunicação sob transportes agnósticos do Git, como o HTTP.

Segredos Revelados

Vamos dar uma espiada sob o capô e explicar como o Git realiza seus milagres. Será uma explicação superficial. Para detalhes mais aprofundados consultar o manual do usuário.

Invisibilidade

Como pode o Git ser tão discreto? Fora ocasionais commit e merge, você pode trabalhar como se desconhecesse que existe um controle de versões. Isto é, até que precise dele, e é quando você ficará agradecido ao Git por estar vigiando o que faz o tempo todo.

Outros sistemas de controle de versões não deixam você esquecê-los. As permissões dos arquivos são apenas de leitura, a menos que você diga ao servidor quais arquivos tem a intenção de editar. Os comandos mais básicos podem demorar muito quando o número de usuários aumenta. O trabalho pode ser interrompido quando a rede ou o servidor central para.

Ao contrário, o Git guarda o histórico do seu projeto no diretório `.git` no diretório de trabalho. Essa é a sua cópia do histórico, de modo que você pode

ficar offline até que deseje se comunicar com os outros. Você tem total controle sobre o destino de seus arquivos, pois o Git pode facilmente recriar um estado salvo a partir do `.git` a qualquer hora.

Integridade

A maioria das pessoas associam criptografia com manter informações secretas, mas outra aplicação igualmente importante é manter a integridade da informação. O uso correto das funções criptográficas de hash pode prevenir o corrupção acidental ou intencional dos dados.

Um hash SHA1 pode ser entendido como um número identificador único de 160 bits para cada sequência de bytes que você vai encontrar na vida. Na verdade mais que isso: para cada sequência de bytes que qualquer ser humano jamais usará durante várias vidas.

Como um hash SHA1 é, ele mesmo, uma sequência de bytes, podemos gerar um hash de sequências de bytes formada por outros hash. Essa observação simples é surpreendentemente útil: a procura por *cadeias hash* (*hash chains*). Vamos ver mais adiante como o Git a utiliza para garantir com eficiência a integridade de dados.

A grosso modo, o Git mantém os seus arquivos no subdiretório `.git/objects`, onde ao invés de ter arquivos com nomes "normais", vamos encontrar somente identificadores (Ids). Utilizando os Ids como nomes de arquivos, bem como uns poucos lockfiles e alguns truques com o timestamp, o Git transforma qualquer sistema de arquivos simples em um poderoso banco de dados.

Inteligência

Como o Git sabe que um arquivo foi renomeado, mesmo que você nunca tenha mencionado o fato explicitamente? Com certeza, você executou **git mv**, mas isto é exatamente o mesmo que um **git rm** seguido por um **git add**.

A análise heurística do Git verifica além das ações de renomear e de cópias sucessivas entre versões. De fato, ele pode detectar até pedaços de código sendo movidos ou copiados entre arquivos! Embora não cubra todos os casos, faz um trabalho decente, e esta característica está sendo sempre aprimorada. Caso não funcione com você, tente habilitar opções mais refinadas para detecção de cópias e considere uma atualização.

Indexando

Para cada arquivo monitorado, o Git armazena informações como: tamanho, hora de criação e última modificação, em um arquivo conhecido como *index*.

Para determinar se um arquivo foi modificado, o Git compara seus status atual com o que tem no index. Se coincidem, então ele pode ignorar o arquivo.

Já que verificações de status são imensamente mais baratas que ler o conteúdo do arquivo, se você editar poucos arquivos, o Git vai atualizar seus status quase que instantaneamente.

Falamos anteriormente que o index é uma área de atuação (staging). Por que um monte de status de arquivos é uma area de atuação (staging)? É porque o comando add coloca os arquivos no banco de dados do Git e atualiza esse status, enquanto o comando commit, sem opções, cria um commit baseado somente no status e arquivos já existentes no banco de dados.

Origem do Git

Esta mensagem na lista de discussão do Linux Kernel descreve a sequência de eventos que levaram ao Git. A discussão inteira é um sitio arqueológico fascinante para historiadores do Git.

O Banco de Dados de Objetos

Cada versão de seus dados é mantida em um *bando de dados de objetos*, que reside no subdiretório `.git/objects`: os outros residentes de `.git/` armazenam menos dados: o index, nomes dos branchs, etiquetas (tags), configurações de opções, logs, a localização do commit HEAD, e outros. O bando de dados objeto é elementar e elegante, e a origem do poder do Git.

Cada arquivo dentro de `.git/objects` é um *objeto*. Existem 3 tipos de objetos que nos interessam: objetos *blob*, objetos *árvores* (*tree*), e objetos *commit*.

Blobs

Primeiro, um truque mágico. Peque um nome de arquivo, qualquer arquivo. Em um diretório vazio, execute

```
$ echo sweet > YOUR_FILENAME
$ git init
$ git add .
$ find .git/objects -type f
```

Voce verá `.git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d`.

Como eu posso saber disso, sem saber o nome do arquivo? É por que o hash SHA1 de:

```
"blob" SP "6" NUL "sweet" LF
```

é aa823728ea7d592acc69b36875a482cdf3fd5c8d, onde SP é espaço, NUL é o byte zero e LF é um linefeed. Você pode verificar isso, digitando:

```
$ printf "blob 6\000sweet\n" | sha1sum
```

O Git é *endereçável-por-conteúdo*: os arquivos não são armazenados de acordo com seus nomes de arquivos, e sim pelo hash de seus dados, em um arquivo que chamamos de *objeto blob* (*blob object*). Podemos pensar que o hash é um identificador único para o conteúdo do arquivo, de modo que estamos endereçando os arquivos pelo seu conteúdo. O `blob 6` inicial é meramente um header que consiste do tipo do objeto e seu tamanho em bytes; isso simplifica a organização interna.

Assim eu poderia prever o que você irá ver. O nome do arquivo é irrelevante: somente os dados internos são utilizados para construir o objeto blob.

Você pode estar se perguntando o que acontece com os arquivos idênticos. Tente adicionar cópias de seu arquivo, com qualquer nome de arquivo. O conteúdo do `.git/objects` continua o mesmo não importa quantos você adiciona. O Git somente armazena o dado uma única vez.

A propósito, os arquivos dentro de `.git/objects` são comprimidos com a zlib de modo que você não consegue examiná-los diretamente. Faça uma filtragem por meio do zpipe -d, ou digite:

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

que mostra o objeto em um formato legível.

Árvores

Mas onde estão os nomes de arquivos? Eles precisam ser armazenados em algum lugar. Git fica sabendo o nome do arquivo durante um commit:

```
$ git commit  # Type some message.
$ find .git/objects -type f
```

Você pode ver agora 3 objetos. Dessa vez eu não consigo dizer quais os dois nomes de arquivos, já que eles dependem parcialmente do nome do arquivo que você escolheu. Vamos prosseguir assumindo que você escolheu “rose”. Se não escolheu esse nome, você pode reescrever o histórico para ficar parecido com o que você fez:

```
$ git filter-branch --tree-filter 'mv YOUR_FILENAME rose'
$ find .git/objects -type f
```

Agora você poderá ver o arquivo `.git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9`, porque esse é o hash SHA 1 de seu conteudo:

`"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d`

Verifique que este arquivo contém efetivamente o que falamos acima, digitando:

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

Com o zpipe, é fácil verificar o hash:

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

A verificação do hash é mais complicada via cat-file porque sua saida contem mais do que os dados descomprimidos do arquivo object.

Esse arquivo é um objeto *árvore* (*tree*): uma lista de tuplas que consiste em um tipo de arquivo, um nome de arquivo e um hash. Em nosso exemplo, este tipo de arquivo é 100644, que significa que 'rose' é um arquivo normal, e o hash é o objeto blob que contém o termo 'rose'. Outros tipos possíveis de arquivos são executáveis, symlinks e diretorios. No ultimo exemplo, o hash aponta para um objeto árvore.

Se você executar o filter-branch, você obterá objetos antigos que não precisa mais. Embora sejam eliminados automaticamente quando o período de armazenamento expirar, iremos deletá-los agora para tornar o nosso exemplo mais fácil de seguir:

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

Para projetos reais você deve tipicamente evitar comandos como esses, já que eles destroem os backups. Se você deseja um repositório limpo, é geralmente melhor criar um novo clone. Também, tome cuidado quando manipular diretamente o `.git`: e se um comando Git esta executando ao mesmo tempo, ou uma falha na alimentação eletrica ocorre? Geralmente, as referências podem ser deletadas com **git update-ref -d**, embora seja mais seguro remover manualmente o `refs/original`.

Commits

Explicamos 2 dos 3 objetos. O terceiro é o objeto *commit*. Seu conteúdo depende da mensagem de commit bem como da data e hora em que foi criado. Para combinar com o que temos aqui, vamos ter que fazer um pequeno truque:

```
$ git commit --amend -m Shakespeare  # Change the commit message.
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"' # Rig timestamps and authors.
```

```
$ find .git/objects -type f
```

Agora você pode ver `.git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187` que é o hash SHA 1 de seu conteúdo:

```
"commit 158" NUL
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

Como anteriormente, você pode executar o zpipe ou cat-file para ver você mesmo.

Esse é o primeiro commit, de modo que não existe nenhum commit pai, mas commit posteriores irão sempre conter no mínimo uma linha identificando o seu commit pai.

Indistinguível da Magia

O segredo do Git parece ser tão simples. Parece que você pode misturar um pouco de script shell e adicionar uma pitada de código C para cozinha-lo em questão de horas: uma mistura de operações básicas do sistema de arquivos e hash SHA 1, guarnecido com arquivos lock e fsyncs para robustez. De fato, isso descreve acuradamente as primeiras versões do Git. No entanto, além de truques engenhosos de empacotamento para economizar espaço, e truques engenhosos de indexação para economizar espaço, agora sabemos como o Git habilmente transforma um sistema de arquivos em um banco de dados perfeito para o controle de versões.

Por exemplo, se algum arquivo dentro do banco de dados de objetos é corrompido por um erro de disco, então o seu hash não irá corresponder mais, alertando-nos sobre o problema. Fazendo hash de hash de outros objetos, mantemos a integridade em todos os níveis. Os commits são atomicos, isto é, um commit nunca pode armazenar parcialmente as mudanças: só podemos calcular o hash de um commit e armazenar ele no banco de dados após ter armazenado todas as árvores relevantes, blobs e os commits pais. O banco de dados de objetos é imune a interrupções inesperadas tais como falha de alimentação eletrica.

Nós derrotamos até mesmo os adversários mais tortuosos. Suponha que alguem tente de maneira escondida, modificar o conteudo de um arquivo em uma versão antiga do projeto. Para manter o banco de dados dos objetos com uma aparência saudável, ele deve também alterar o hash do objeto blob correspondente, já que ele contem agora uma cadeia de bytes diferente. Isso significa que ele terá que alterar o hash de qualquer objeto árvore que referencia o arquivo, e em seguida alterar o hash de todos os objetos commit que estão envolvidos com esses objetos árvores, além dos hash de todos os descendentes desses commits.

Isso significa que o hash do Head oficial será diferente do hash do repositório alterado. Seguindo a trilha dos hash não correspondentes podemos apontar o arquivo alterado, bem como o commit onde ele foi corrompido.

Resumindo, graças aos 20 bytes que representam o ultimo commit seguro, é impossível adulterar um repositório Git.

É sobre as famosas características do Git? Branching, Merging? Tags? Meros detalhes. O cabeçalho atual é mantido em um arquivo `.git/HEAD`, que contém um hash de um objeto commit. O hash é atualizado durante um commit bem como com muitos outros comandos. Branch são quase a mesma coisa: eles são arquivos em `.git/refs/heads`. Tags também: elas estão em `.git/refs/tags` mas são atualizadas por um conjunto diferente de comandos.

Apêndice A: Deficiências do Git

Há algumas questões sobre o Git que joguei pra debaixo do tapete. Algumas são facilmente tratadas com script e gambiarras, algumas requerem uma reorganização ou redefinição do projeto, e para as poucas chateações remanescentes, só resta esperar por uma solução. Ou melhor ainda, solucione-as e ajude a todos!

Pontos fracos do SHA1

A medida que o tempo passa, os criptógrafos descobrem mais e mais os pontos fracos do SHA1. Atualmente, encontrar colisões de hash é factível para algumas organizações bem equipadas. Dentro de alguns anos, talvez até um PC comum terá capacidade computacional suficiente para corromper silenciosamente um repositório Git.

Esperamos que o Git irá migrar para uma função hash melhor antes que mais pesquisas destruam o SHA1.

Microsoft Windows

O Git no Microsoft Windows pode ser trabalhoso:

- Cygwin, é um ambiente que deixa o Windows parecido com o Linux, tem uma versão do Git para Windows.
- Git para Windows é uma alternativa que requer suporte minimo para execução, embora alguns poucos comandos precisem ser mais trabalhados.

Arquivos Independentes

Se seu projeto é muito grande e tem muitos arquivos independentes que estão sendo constantemente modificados, o Git pode ser prejudicado mais do que os outros sistemas, pois os arquivos não são monitorados isoladamente. O Git monitora modificações no projeto como um todo, o que geralmente é benéfico.

Uma solução é dividir seu projeto em pedaços, cada um composto de arquivos relacionados. Use **git submodule** se ainda quiser manter tudo num repositório só.

Quem Está Editando O Que?

Alguns sistemas de controle de versões irão forçá-lo a marcar explicitamente um arquivo de alguma maneira antes de editá-lo. Embora seja especialmente irritante quando isso envolve usar um servidor centralizado, isto tem dois benefícios:

1. Diff são rápido pois apenas os arquivos marcados são examinados;
2. Outros podem saber quem está trabalhando no arquivo perguntando ao servidor central quem marcou o arquivo para edição.

Com o script certo, você pode fazer o mesmo com o Git. Isto requer apenas a cooperação dos programadores, que devem executar o script em particular quando estiver editando um arquivo.

Arquivo do Histórico

Como o Git armazena modificações muito amplas no projeto, reconstruir o histórico de um único arquivo requer mais trabalho do que em sistemas de controle de versões que monitoram arquivos individualmente.

A penalidade é usualmente leve, e vale a pena devido à eficiência que dá as outras operações. Por exemplo, `git checkout` é tão rápido quanto `cp -a`, e os deltas que abrangem grandes partes do projeto tem uma compressão melhor do que os deltas de agrupamentos de arquivos.

Clone Inicial

A criação de um clone é mais trabalhosa do que fazer checkout em outros sistemas de controle de versões quando há um histórico grande.

O custo inicial se paga a longo prazo, pois as futuras operações serão mais rápidas e offline. Entretanto, em algumas situações, é preferível criar um clone

vazio com a opção `--depth`. Isto é muito mais rápido, porém resulta em um clone com funcionalidades reduzidas.

Projetos Voláteis

O Git foi feito para ser rápido no que diz respeito ao tamanho das mudanças. Humanos fazem poucas edições de uma versão para outra. É a correção de uma falha numa linha, uma nova característica do sistema, inclusão de comentário e assim por diante. Mas se seus arquivos diferem muito de uma versão para outra, em cada commit, seu histórico irá crescer acompanhando o tamanho do seu projeto todo.

Não há nada que qualquer sistema de controle de versões possa fazer para ajudar, mas os usuários comuns do Git devem sofrer mais quando estiverem clonando históricos.

As razões pelas quais as mudanças são tão grandes, devem ser analisadas. Talvez os formatos dos arquivos possam ser trocados. Edições menores só devem causar pequenas modificações em poucos arquivos.

Ou talvez um banco de dados ou uma solução de backup/arquivamento seja o que você realmente precisa, e não um sistema de controle de versões. Por exemplo, um controle de versões pode não ser adequado para gerenciar fotos feitas periodicamente de uma webcam.

Se os arquivos estão, realmente, mudando constantemente e precisam ser versionados, uma possibilidade é usar o Git de uma maneira centralizada. Pode-se criar clones vazios, que adiciona pouco ou quase nada ao histórico do projeto. É claro, que muitas ferramentas do Git não estarão disponíveis, e as correções devem ser enviadas como patch. Isto deve ser razoavelmente útil, para alguém que deseja manter um histórico de arquivos demasiadamente instáveis.

Outro exemplo é um projeto dependente de firmware, o qual provavelmente estará em um grande arquivo binário. O histórico de arquivos de firmware é irrelevante para os usuários, e as atualizações têm uma péssima compressão, assim revisões de firmware estourarão o tamanho do repositório sem necessidade.

Neste caso, o código fonte deve ser armazenado num repositório Git, e os arquivos binários mantidos separados do mesmo. Para facilitar o trabalho, alguém pode criar e distribuir um script que usa o Git para clonar o código e o faz um rsync ou um cria um clone vazio do Git para o firmware.

Contador Global

Alguns sistemas centralizados de controle de versões mantém um número inteiro positivo que é incrementado quando um novo commit é aceito. O Git referencia as modificações por seus hash, o que é o melhor na maioria das circunstâncias.

Mas algumas pessoas gostariam de ter este número por perto. Felizmente, é fácil criar um script que faça isso a cada atualização, o repositório central do Git incrementa o número, talvez em uma marca (tag), e associa a mesma com o hash do último commit.

Cada clone poderia gerenciar este contador, porém isto provavelmente seja desnecessário, já que apenas o contador do repositório central é que importará para todos.

Subdiretórios Vazios

Subdiretórios vazios não são monitorados. Crie arquivos vazios para resolver esse problema.

A implementação atual do Git, e não seu o design, é a razão deste inconveniente. Com sorte, uma vez que o Git ganhe mais utilização, mais usuários devem clamar por esse recurso e ele poderá ser implementado.

Commit Inicial

Um cientista da computação tipico inicia uma contagem do 0, ao invés do 1. Entretanto, no que diz respeito a commit, o Git não segue esta convenção. Muito comandos são confusos antes do commit inicial. Além disso existem algumas arestas que precisam aparadas manualmente, seja com um rebase de um branch com um commit inicial diferente.

O Git iria se beneficiar por definir o commit zero: assim que um repositório é construído, o HEAD deve ser definido para uma cadeia de 20 bytes zero. Esse commit especial representaria uma arvore vazia, sem pai, em um momento anterior a todos os repositórios Git

Então executando um git log, por exemplo, deveria informar ao usuário que nenhum commit foi feito até agora, ao invés de terminar com um erro fatal. Da mesma maneira que as outras ferramentas.

Cada commit inicial é implicitamente um decendente desse commit zero.

Infelizmente existem alguns casos problemáticos. Se vários branch com commit iniciais diferentes forem merged juntos, então um rebase do resultado vai requerer uma intervenção manual substancial.

Peculiaridades da Interface

Para commit A e B, o significado da expressão "A..B" e "A...B" depende de onde o comando espera os dois pontos ou uma faixa. Veja **git help diff** e **git help rev-parse**.

Apendice B: Traduzindo esse guia

Recomendo os seguintes passos para a tradução desse guia, de modo que meus scripts produzam a versão HTML e PDF, e que todas as traduções possam conviver em um mesmo repositório.

Faça um clone do fonte, e então crie um diretório correspondente a tag IETF da idioma alvo: veja o artigo da W3C sobre internacionalização. Por exemplo, Ingles (English) é ”en” e Japonês é ”ja”. No novo diretório, traduza os arquivos .txt a partir do subdiretorio ”en”.

Por exemplo, para traduzir este guia para a língua Klingon, você deve digitar:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" is the IETF language code for Klingon.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # Translate the file.
```

e assim por diante, para cada arquivo .txt

Edite o Makefile e adicione o código do idioma na variável `TRANSLATIONS`. Você poderá então rever seu trabalho de forma incremental:

```
$ make tlh
$ firefox book-tlh/index.html
```

Faça frequentes commits de suas alterações, e me avise quando o trabalho estiver pronto. O GitHub tem uma interface que facilita isso: faça um fork do projeto ”gitmagic”, faça um push de suas alterações, e então me peça para fazer um merge.