Magia Git

Ben Lynn

Magia Git

Ben Lynn

Tradutor: Leonardo Siqueira Rodrigues
{leonardo.siqueira.rodrigues@gmail.com}

Informações sobre a tradução

Fonte utilizadas

- Anonymous Pro[1]

  ABCDEFGHIJKLMOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz
  012456789

- APHont[2]

  ABCDEFGHIJKLMOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz
  012456789

- **Diavlo Black**[3]

  ABCDEFGHIJKLMOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz
  012456789

Padrões que tentei usar no texto:

- Termos estrangeiros no singular. Assim, quando necessário mudei as frases para que o termo no singular encaixasse melhor;
- Manter o nome Git com o "g" maiúsculo;
- Seguir a convenção abaixo na formatação do texto:
  - *Itálico* para palavras estrangeiras, nome de comandos quando "substantivados". Exemplo: "... no primeiro *commit*...", "... um *backup*...";
  - **Negrito** para comandos, nome de diretórios. Exemplo: "... ajuda do **`git help re-parse`**...".

Toda ajuda para corrigir e/ou melhorar o documento é bem-vinda.

1 http://www.ms-studio.com/FontSales/anonymouspro.html
2 http://www.aph.org/products/aphont.html
3 http://www.exljbris.nl

ÍNDICE

Prefácio

Git[4] é um canivete suíço do controle de versões. Uma ferramente polivalente realmente versátil cuja extraordinária flexibilidade torna-o complicado de aprender, sobre tudo sozinho. Coloquei nestas paginas o pouco que aprendi, pois inicialmente tive dificuldade em compreender o manual do usuário do Git[5].

Como Arthur C. Clarke[6] bem comentou: "Qualquer tecnologia suficientemente avançada é considerada magica". 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 horrendas habilidades.

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

Outras Edições

- Tradução Chinesa[7]: por JunJie, Meng e JiangWei.
- Pagina única[8]: HTML, sem CSS.
- Arquivo PDF[9]: pronto para imprimir.

Agradecimentos!

Agradecimentos a Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan e Derek Mahar pelas sugestões e melhorias. [Se esqueci de você, por favor avise, as vezes esqueço de atualizar esta secção.]

Licença


```
$ git clone git://repo.or.cz/gitmagic.git  # Cria a pasta "gitmagic".
```

Veja a seguir outros locais.

4 http://git-scm.com/
5 http://www.kernel.org/pub/software/scm/git/docs/user-manual.html
6 http://pt.wikipedia.org/wiki/Arthur_C._Clarke
7 http://docs.google.com/View?id=dfwthj68_675gz3bw8kj
8 http://www-cs-students.stanford.edu/~blynn/gitmagic/book.html
9 http://www-cs-students.stanford.edu/~blynn/gitmagic/book.pdf
10 http://www.gnu.org/licenses/gpl-3.0.html

Links

Uma vez listei algumas referências, porém consome muito tempo mante-las. Além disso, qualquer um pode usar um site de busca para encontrar tutoriais, guias e comparações do Git com Subversion[11], Mercurial[12], ou outro sistema de controle de versões.

Repositórios Git grátis

- http://repo.or.cz/ hospeda projetos livres, inclusive este guia.
- http://gitorious.org/ é outro site de hospedagem Git destinado a projetos de código aberto.
- http://github.com/ hospeda projetos de código aberto de graça, inclusive este guia, e projetos privados também.

11 http://subversion.tigris.org/
12 http://www.selenic.com/mercurial/

Introdução

Usarei uma analogia para falar sobre controle de versões. Veja na Wikipédia o verbete sistema de controle de versões[13] para uma melhor explicação .

Trabalhar é Divertido

Me divirto com jogos para computador quase 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 salva-los. Pra 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 revisita-lo outra hora. Ou pior, o ultimo estado salvo é um dificílimo e você terá que recomeçar.

Controle de Versões

Ao editar, você pode "Salvar como ..." num arquivo diferente, ou copiar o arquivo antes de sobrescreve-lo se você quiser manter as versões anteriores. Pode comprimi-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.

13 http://pt.wikipedia.org/wiki/Sistema_de_controle_de_versão

Sistemas de controle de versões não são diferentes. Todos tem uma boa interface para gerenciar seu diretório de versões. Você pode salvar o diretório sempre que desejar, e pode rever qualquer um dos estado 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 vence-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 utilizava controle de versões centralizado. Um servidor em algum lugar mantêm os jogos salvos. Ninguém tem todos. Cada jogador mantêm, em sua maioria, 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 antigo jogo salvo por algum motivo? Talvez o atual jogo salvo esteja em um nível impossível de jogar devido alguém ter esquecido um objeto três níveis atrás, e é preciso encontrar o ultimo jogo salvo que está em um nível que pode ser completado com sucesso. Ou talvez queira comparar dois jogos salvo 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 antiga versão. E quanto mais jogos salvos forem necessários, maior é o trafego 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 pegam do servidor, eles recebem todos os jogos salvos, e não apenas o mais recente. É como se estivessem espelhando o servidor.

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

Uma superstição

Um equívoco popular é que sistemas distribuídos estão mal adaptados projetos que exijam um repositório central oficial. Nada poderia estar mais longe da verdade. Fotografar alguém roubara 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 simplesmente mais oneroso que recursos locais. Embora vejamos mais adiante que existem inconvenientes numa abordagem distribuída, é menos provável que faça comparações errôneas com esta regra de ouro.

Um pequeno projeto pode precisar de apenas uma fração dos recursos oferecidos pelo sistema. Mas, você usa algarismos romanos quando calcula 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 tornasse ruim. Em disso, vamos considerar novamente a edição de um documento.

Suponha que Alice insira uma linha no inicio 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. O segundo, 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 nos 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 suas simplicidades, 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 um "fotografia" de todos os arquivos do diretório atual com:

```
$ git init
$ git add .
$ git commit -m "Meu primeiro backup"
```

A sequência de comandos acima devem ser memorizados, ou colocados em um script, pois serão usados com muita frequência.

Assim se algo der errado, você só precisará executar:

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

para voltar para o estado anterior. Para salvar o estado novamente, faça:

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

Adicionar, Remover, Renomear

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

```
$ git add NOVOSARQUIVOS...
```

Do mesmo modo, se você quiser que o Git na verifique certos arquivos, talvez por tê-los apagados, faça:

```
$ git rm ANTIGOSARQUIVOS...
```

Renomear um arquivo é o mesmo que remover 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 ANTIGOARQUIVO NOVOARQUIVO
```

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. Em seguida, digite:

```
$ git reset --hard SHA1_HASH
```

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

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

```
$ git checkout SHA1_HASH
```

Isto levará você de volta no tempo, preservando os novos commit. Entretanto, como nas viagens no tempo do filmes de ficção, se você editar e fizer um commit, você estará nume 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 hora, apenas lembre-se que:

```
$ git checkout master
```

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

Voltemos para a analogia dos jogos de computador :

- `git reset --hard`: carrega um antigo salvamento e apagar todos os salvamento mais novos do que este que foi carregado.
- `git checkout`: carrega um antigo salvamento, mas se jogar a partir dele, os próximos salvamento realizados se desvincularão dos salvamentos já realizados após o que foi carregado. Qualquer salvamento que você fizer será colocado em um *branch* separado representado a realidade alternativa em que entrou. Lidaremos com isso mais a frente.

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

Não gosta de copiar e colar *hash*? Então use:

```
$ git checkout :/"Meu primeiro b"
```

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

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

Revertendo

Como num tribunal, eventos podem ser retirados dos registros. Igualmente, você pode especificar qual *commit* desfazer.

```
$ git commit -a
$ git revert SHA1_HASH
```

irá desfazer apenas o *commit* do *hash* informado. Executando ***git log*** revelará que a regressão é gravada como um novo *commit*.

Download de arquivos

Obtenha a cópia dum projeto gerenciado com GIT digitando:

```
$ git clone git://servidor/caminho/dos/arquivos
```

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

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

A seguir, teremos muito o que dizer sobre o comando ***clone***.

Última Versão

Se você já obteve a copia 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 compartilha-lo. Você poderia simplesmente dizer para pegarem do seu computador, mas se o fizerem enquanto você esta melhorando o *script* ou experimentado 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 esta apresentável.

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

```
$ git init
$ git add .
$ git commit -m "Primeira liberação"
```

Então avise aos outros para executarem:

```
$ git clone seu.computador:/caminho/do/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://seu.computador/caminho/do/script
```

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

```
$ git commit -a -m "Nova liberação"
```

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 última *commit* com:

```
$ git diff
```

Ou desde ontem:

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

Ou entre uma versão particular e duas versões atrás:

```
$ git diff SHA1_HASH "master~2"
```

Tente também:

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

As vezes navego pelo histórico com o ***qgit***[14], em razão de sua interface mais fotogênica, ou com o ***tig***[15], uma interface em modo texto ótima para conexões lentas. Alternativamente, instale um servidor *web*, execute `git instaweb` e use um navegador.

14 http://sourceforge.net/projects/qgit
15 http://jonas.nitro.dk/tig/

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 clonasse o 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

Esta foi a razão pela qual usei o Git pela primeira vez. Eu posso aguentar fazer *tarball*[16] ou usar o `rsync` para *backup* e sincronizações básicas. Mas as 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 *commit* seus arquivos em uma das maquinas. Então na outra:

```
$ git clone outro.computador:/caminho/dos/arquivos
```

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

```
$ git commit -a
$ git pull outro.computador:/caminho/dos/arquivos
```

o que deixará os arquivos da maquina 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 "Commit inicial"
```

No servidor principal, inicialize um repositório Git em branco com o mesmo

16 http://pt.wikipedia.org/wiki/TAR

nome, e inicie o *daemon* Git se necessário:

```
$ GIT_DIR=proj.git git init
$ git daemon --detach  # Ele pode já estar sendo executado
```

Algumas hospedagens publicas, tais como o repo.or.cz, terão métodos diferentes para configurar o repositório inicial em branco, como através do preenchimento de um formulário no site deles.

Mande seu projeto para o servidor principal com:

```
$ git push git://servidor.principal/caminho/do/proj.git HEAD
```

Estamos prontos. Para verificar os fontes, um desenvolvedor pode digitar:

```
$ git clone git://servidor.principal/caminho/do/proj.git
```

Após realizar as alterações, o código é enviado para o servidor com:

```
$ git commit -a
$ git push
```

Se o servidor principal tiver sido atualizado enquanto realizava as alterações, será necessário obter a última versão antes de enviar as alterações. Para sincronizar para a última versão:

```
$ git commit -a
$ git pull
```

Fazendo um *Fork*[17] do Projeto

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

```
$ git clone git://servidor.principal/caminho/dos/arquivos
```

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

Qualquer hora depois, você pode mesclar (*merge*) suas mudanças do projeto original no mesmo com:

```
$ git pull
```

Backup Supremos

Gostaria de numerosos arquivos geograficamente 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

17 http://pt.wikipedia.org/wiki/Fork

estado atual, e sim o histórico completo do seu projeto. Graças ao *hash* criptográfico, se cada clonagem for corrompida, ele será identificado assim que tentar se comunicar com os outros.

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

Um paranoico 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, publica-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 . /algum/novo/diretório
```

Git explora, até onde for seguramente possível, *hard links* e compartilhamento de arquivos para criar este clone, assim isto deve ficar pronto em um instante, e você pode agora trabalhar e duas funções independentes simultaneamente. Por exemplo, você pode editar um clone enquanto o outro é compilado.

A qualquer momento, você pode fazer um *commit* e pegar as alterações de outro clone

```
$ git pull /o/outro/clone
```

Controle de Versões de Guerrilha

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

```
$ git init
$ git add .
$ git commit -m "Commit inicial"
```

clone-o, na velocidade da luz:

```
$ git clone . /algum/novo/diretório
```

Agora vai 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 "Sincronizando com os outros"
```

Depois vá para o novo diretório e execute:

```
$ git commit -a -m "Descrição das minhas alterações"
$ 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 envia-las para o repositório central.

O comando **`git svn`** automatiza tudo isso para repositórios Subversion, e também pode ser utilizado para exportar um repositório Git para um repositório Subversion[18].

18 http://google-opensource.blogspot.com/2008/05/export-git-project-to-google-code.html

Bruxaria Com *Branch*

Ramificações (*Branch*) e mesclagens (*merge*) instantâneos são as características mas 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 os casos, você abruptamente deixará de lado o que esta fazendo e focará em uma tarefa completamente diferente.

Interromper sua linha de pensamento provavelmente prejudicará sua produtividade, e quanto mais trabalhoso é trocar de contexto, maior é 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 o custo reduzido no espaço usado pelos arquivos que o Git tem com o compartilhamento destes arquivos, os arquivos do projeto em si são recriados na integra no novo diretório.

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 jogue 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 "Sou mais esperto que meu chefe" > meuarquivo.txt
$ git init
$ git add .
$ git commit -m "Commit inicial"
```

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

```
$ git checkout -b chefe  # nada parece ter mudado após isto
$ echo "Meu chefe é mais esperto que eu" > meuarquivo.txt
$ git commit -a -m "Outro commit"
```

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

```
$ git checkout master  # troca para a versão original do arquivo
```

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

```
$ git checkout chefe  # troca para versão adaptada para agradar o chefe
```

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 para uma antiga versão e temporariamente colocar algumas declarações de controle para ver como algo funciona. Então:

```
$ git commit -a
$ git checkout SHA1_HASH
```

Agora você pode adicionar temporariamente código feio em todo canto. 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 sujeira
```

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

```
$ git checkout sujeira
```

Nós já falamos deste comando num capítulo anterior, quando discutimos carregamento de antigos estados 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 a diante.

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 esta fazendo algo quando mandam largar o que quer que seja e corrigir um erro recém descoberto:

```
$ git commit -a
$ git checkout -b correções SHA1_HASH
```

Então assim que tiver corrigido o erro:

```
$ git commit -a -m "Erro corrigido"
$ git push  # envia para o repositório principal
$ git checkout master
```

e volte a trabalhar no que estava fazendo anteriormente.

Fluxo ininterrupto

Alguns projetos requerem que seu código seja revisado antes que o envie. Para tornar a vida fácil daqueles que forem revisar seu código, se você tem um a grande mudança para fazer, você pode quebra em dois ou mais partes, e ter cada parte revisada separadamente.

E se a segunda parte não puder ser escrita até que a primeira seja aprovada e revisada? Em muitos sistemas de controle de versões, você teria que mandar a primeira parte para os revisores, e aguardar até que seja aprovada para então começar a segunda parte.

Atualmente isso não é verdade, mas nestes sistemas de edição da segunda parte antes da primeira ser aprovada envolve muito sofrimento e privações. No Git, os *branch* e *merge* são indolores (termo técnico para rápido e local). Então após fazer o *commit* da primeira parte e enviado para revisão:

```
$ git checkout -b parte2
```

Em seguida, codifique a segunda parte da grande mudança sem esperar que a primeira parte tenha sido aceita. Quando a primeira parte for aprovada e enviada,

```
$ git checkout master
$ git merge parte2
$ git branch -d parte2  # não precisará mais deste branch
```

e a segunda parte das mudanças está pronta para ser revisada.

Mais espere! E se não for tão simples? Digamos que você cometeu em erro na primeira parte, que você tem que corrigir antes de envia-la. Sem problema! Primeiro, volte para *branch master* com:

```
$ git checkout master
```

Corrija o problema da primeira parte das mudanças e aguarde a aprovação. Se não simplesmente repita este passo. Você provavelmente desejará fazer um *merge* da versão corrigida da primeira parte com a segunda, logo:

```
$ git checkout part2e
$ git merge master
```

Agora como anteriormente. Logo que a primeira parte for aprovada e enviada:

```
$ git checkout master
$ git merge parte2
$ git branch -d parte2
```

e, novamente, a segunda parte está pronta para a revisão.

É fácil estender este truque para qualquer número de partes.

Reorganizando uma improvisação

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

```
$ git checkout -b organizado
$ git checkout -b desorganizado
```

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 organizado
$ git cherry-pick SHA1_HASH
```

aplique um dado *commit* ao *branch* "organizado". Com os *cherry-picks* apropriados você pode construir um *branch* que contêm apenas código permanente, e tem *commit* relacionados agrupados juntos.

Gerenciando o *Branch*

Digite:

```
$ git branch
```

para listar todos os *branch*. Há sempre um *branch* chamado "*master*", e você começa por aqui, por *default*. Alguns defendem que o *branch* "*master*" deve ser intocável e criar novos *branch* para suas próprias mudanças.

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

Branch Temporários

Depois de um tempo você perceberá que está criando *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 análogo a mudar o canal da TV temporariamente para ver o que esta passando nos outros canais. Mas, ao invés de apertar dois botões, você esta criando, checando e apagando *branch* 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  # Pode ser preciso resolver alguns conflitos.
```

Você pode ter múltiplos *stash*, e manipula-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

Aplicações como Mozilla Firefox[19] permitem que se abra múltiplas abas e múltiplas janelas. Alternando entre as abas temos diferentes conteúdos na mesma janela. *Branch* no Git são como abas para seu diretório de trabalho. Seguindo na analogia, a clonagem no Git é como abrir uma nova janela. Ser capaz de fazer ambos, melhora a experiência do usuário.

Num nível mais alto: vários gerenciadores de janelas no linux[20] permitem a existência de várias áreas de trabalho: e instantaneamente alternar entre elas. O que é similar ao *branch* no Git, enquanto a clonagem seria como anexar outro monitor para ganhar mais uma área de trabalho.

19 http://www.mozilla.com/
20 http://www.linux.org

Outro exemplo é o utilitário ***screen***[21]. Esta preciosidade lhe permite criar, destruir e alternar entre múltiplas sessões de terminais no mesmo terminal. Ao invés de abrir novos terminais (clone), você pode use o mesmo se usar o **`screen`** (*branch*). Na realidade, você pode fazer muito mais com o **`screen`** mas isso é assunto para outro texto.

Clonagem, *branch* e *merge* são rápidos e locais no Git, encorajando-o a usar a combinação que mais lhe convir. Git deixa você trabalhar exatamente como quiser.

21 http://www.gnu.org/software/screen/

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.

Claro, se você controlar todas as outras árvores também, então não há problema, uma vez que pode sobrepor-las.

Alguns desenvolvedores gostam de um histórico imutável, com falhas ou não. Outros, que suas árvores estejam apresentáveis antes de libera-las ao público. O Git contemplá ambos pontos de vista. Tal como clonagem, *branch* e *merge*, rescrever o histórico é simplesmente outro poder que o Git lhe concede. Cabe a você a usá-lo sabiamente.

Estou correto

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

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

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

Quer incluir mais algumas modificações no ultimo *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 fez um monte de *commit*. E você não está muito feliz 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 Adicionado link para repo.or.cz
pick a311a64 Reorganizadas as analogias em "Trabalhe como quiser"
pick 100834f Adicionado origem (push target) ao Makefile
```

Então:

- Remova os *commit* deletando linhas;
- Reorganize os *commit* reorganizando linhas;
- Substitua **pick** por **edit** para modificar a mensagem do *commit*;
- Substitua **pick** por **squash** para unir (*merge*) um *commit* com o anterior.

Se marcar um *commit* para edição, execute:

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

Caso contrário, execute:

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

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

Alterações locais por último

Você esta 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 esta 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.

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 numero do cartão de crédito num arquivo texto e acidentalmente adicione-o ao projeto. Apaga-lo é insuficiente, pois, pode ser acessado pe-

los *commit* anteriores. Temos que remover o arquivo de todos os *commit*:

```
$ git filter-branch --tree-filter 'rm meu/arquivo/secreto' HEAD
```

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

Depois, 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.

Senão, de 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 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 projeto; cd projeto; git init
$ git fast-import < /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**, cujo resultado você pode estudar para escrever seus exportadores, e também para transpor repositórios Git para um formato legível aos humanos. Na verdade, estes comandos podem enviar repositórios de aquivos de texto por canais exclusivamente textuais.

Onde tudo começou a dar 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 pra reclamar. No entanto, se você estiver fazendo *commit*, o Git pode localizar o problema:

```
$ git bisect start
$ git bisect bad SHA1_DA_VERSAO_ERRADA
$ git bisect good SHA1_DA_VERSAO_CERTA
```

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 COMANDO
```

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.

Pode se fazer muito mais: a página de ajuda explica como visualizar bissecções, examinar ou reproduzir o *log* da bissecção, e eliminar mudanças conhecidamente 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 ARQUIVO
```

que marca cada linha do arquivo indicado 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 em 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. Restando as operações básicas: navegar no histórico ou fazer *commit* das mudanças. Em alguns sistemas, é exigido do usuário um 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 numero 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 garantia. 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. Minha, frequentemente ruim, conexão 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 ultima instancia, de conti-

nuar 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. No hora em retorno a tarefa original, o comando já havia finalizado a 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[22]: 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.

22 http://pt.wikipedia.org/wiki/Tragédia_dos_comuns

Grão-Mestre Git

Este capitulo de nome pomposo é minha lixeira para truques do Git não classificados.

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
```

Geração do Registro das Modificações

Uma boa pratica é manter um registro das modificações[23], e alguns projetos o exigem. Se você faz *commit* com frequência, e você deveria fazer, gere um registro das modificações digitando:

```
$ git log > ChangeLog
```

Git por SSH, HTTP[24]

Suponha que você tenha acesso `ssh` ao servidor *web*, mas o Git não está instalado. Embora menos eficiente que o seu protocolo nativo, o Git pode comunicar sobre HTTP.

Baixe, compile e instale o Git na sua conta, e cria um repositório no seu diretório *web*:

```
$ GIT_DIR=proj.git git init
```

No diretório ***proj.git***, execute:

```
$ git --bare update-server-info
$ chmod a+x hooks/post-update
```

Do seu computador, faça um ***push*** via ***ssh***:

```
$ git push servidor.web:/caminho/para/proj.git master
```

os outros pegam seu projeto via:

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

23 http://pt.wikipedia.org/wiki/Changelog
24 http://pt.wikipedia.org/wiki/Http

Git Acima de Tudo

Quer sincronizar repositórios sem usar servidores ou mesmo uma conexão via rede? Precisa improvisar durante uma emergência? Já vimos que com o **`git fast-export`** e o **`git fast-import`** podemos converter repositórios para um único arquivo e vice-versa. Podemos armazenar estes arquivos de qualquer jeito para transportar nossos repositórios sobre qualquer meio, porém a ferramenta mais eficiente é o **`git bundle`**.

Para o remetente cria um *bundle* (pacote):

```
$ git bundle create nomedopacote HEAD
```

e então envia o *bundle*, **`nomdepacote`**, para outro lugar usando o que quiser: *email*, *pendrive*, disquete, uma impressão ***xxd***[25] e um scanner com *OCR*, lendo bits pelo telefone, sinais de fumaça, etc. O destinatário recupera os *commit* do *bundle* digitando:

```
$ git pull nomedopacote
```

O destinatário pode fazer isto até num repositório vazio. Apesar do seu tamanho, **`nomedopacote`** contém todo o repositório Git original.

Em projetos grandes, diminua os resíduos empacotando apenas as mudanças que estão faltando no outro repositório:

```
$ git bundle create nomedopacote HEAD ^COMMON_SHA1
```

Se realizado com frequência, alguém pode esquecer a partir de qual *commit* deve ser enviado. A pagina de ajuda sugere o uso de *tags* para resolver isso. Ou seja depois de enviar um *bundle*, digite:

```
$ git tag -f ultimobundle HEAD
```

e crie um novo *bundle* com as novidades:

```
$ git bundle create novobundle HEAD ^ultimobundle
```

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, automa-

25 http://linux.die.net/man/1/xxd

ticamente. No lugar do segundo comando **add**, execute **git commit -a** se sua intenção é efetuar um *commit* neste momento.

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 Tão 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*, exato, com as modificações aprovadas (s*taged*). 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 arquivos especifico. Alternativamente, execute **git commit --interactive** o que automaticamente efetuará seus *commit* assim que terminar de aprovar.

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 move-la. Por exemplo:

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

irá mover o *HEAD* três *commit* pra 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 pagina do manual do comando ***git reset*** 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 SHA1
```

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.

Na configuração padrão, 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 volares de *hash* em **.git/objects** e por tentativa e erro encontrar o que procura. Mas há um modo mais fácil.

O Git guardar 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 ante-ante-ante-antepenúltimo *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 longo período de carência para condenar um *commit*. Por exemplo:

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

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

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

```
$ git conifg gc.auto 0
```

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

Baseando se no Git

Seguindo o jeito *UNIX*[26] de ser, o Git permite ser facilmente utilizado como um componente de "baixo nível" para outros programas. Existem interfaces *GUI*, interfaces *web*, interfaces alternativas para linha de comando e talvez em breve você crie um *script* ou dois para suas necessidades usando o Git.

Um truque simples é criar `alias`, abreviações, internos ao Git para reduzir os comandos utilizados mais frequentemente:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # exibe os alias criados
alias.co checkout
$ git co foo                              # o mesmo que 'git checkout foo'
```

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

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

mostra o nome do *branch* atual. Na prática, muito provavelmente você não

26 http://pt.wikipedia.org/wiki/Unix

quer ver o **/refs/heads** e ignorar os erros:

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

Veja a página do Git para mais exemplos.

Dublês Duros na Queda

As versões recentes do Git tornaram mais difícil para o usuário destruir acidentalmente um dado. Esta é talvez o principal motivo para uma atualização. No entanto, há vezes em que você realmente quer destruir um dado. Mostraremos maneiras de transpor estas salvaguardas para os comandos mais comuns. Use-as apenas se você sabe o que esta fazendo.

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 COMMIT
```

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 obriga-lo, execute:

```
$ git reset --hard [COMMIT]
```

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

```
$ git branch -D BRANCH  # ao invés de usar -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 [ORIGEM] DESTINO  # ao invés de usar -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 apa-

gue-os sem misericórdia com:

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

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

Segredos Revelados

Vamos dar uma espiada sob o capo e explicar como o Git realiza seus milagres. Será uma explicação superficial. Para detalhes mais aprofundados consultar o manual do usuário[27].

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 esquece-los. As permissões dos arquivos são apenas de leitura, a menos que você diga ao servidor quais arquivos tem a intenção de editar. O servidor central estará monitorando as ações de quem fez o *checkout* e quando. Quando a rede cair, você terá um sofrimento repentino. Desenvolvedores constantemente brigam com a burocracia e a *burrocracia* virtual.

O segredo é o diretório `.git` no seu diretório de trabalho. O Git guarda o histórico do seu projeto nele. O "." no inicio do nome esconde ele de uma listagem `ls` normal. Exceto quando você esta fazendo um *push* ou *pull* das modificações, todas as operações de controle de versões são neste diretório.

Você tem controle total sobre o destino dos seus arquivos pois o Git não se importa com o que faz a eles. 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* podem 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. O *hash* de todo o conteúdo de um arquivo pode ser visto como um identificador único para o mesmo.

27 http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

Uma observação importante é que um *hash* SHA1 é, ele mesmo, uma sequência de *bytes*, assim nós podemos gerar um *hash* de sequências de *bytes* formada por outros *hash*.

A grosso modo, todos os arquivos manipulados pelo Git são referenciados pelo seu identificador único, e não por seu nome. Todos os dados estão em arquivos no subdiretório `.git/objects`, onde não há nenhum arquivo com nomes "normais". O conteúdo são sequências de bytes chamados de *blob* e eles não tem relação com seus nomes.

Seus nomes são registrados em outro lugar. Eles estão nos objetos *tree*, que são listas dos nomes dos arquivos formados com os identificadores de seus conteúdos. Uma vez que o *tree* é uma sequência de *bytes*, ele também tem um identificador único, que é como ele é armazenado no subdiretório `.git/objects`. Objetos *Tree* podem aparecer na lista de outro *tree*, logo, um diretório *tree* e todos os seus arquivos podem representados por objetos *tree* e *blob*.

Finalmente, um *commit* contém um mensagem, alguns identificadores *tree* e a informação de como eles estão relacionados entre si. Um *commit* é também um sequência de *bytes*, logo, possui um identificador único.

Veja você mesmo: pegue qualquer *hash* do diretório `.git/objects`, digite

```
$ git cat-file -p SHA1_HASH
```

Agora suponha que alguém tenta reescrever o histórico e tenta modificar o conteúdo de um arquivo de uma versão antiga. Então o identificador único do arquivo sofrerá modificações já que agora ele é uma sequência de *bytes* diferente. Isto modifica o identificador de qualquer objeto *tree* referente a este arquivo, que por sua vez modifica o identificador de todos os objetos *commit* envolvendo este *tree*. A corrupção do repositório ruim é exposta quando todos recebem todos os *commit* já que o arquivo manipulado tem o identificador errado.

Ignorei detalhes como as permissões e assinaturas do arquivo. Mas, em suma, enquanto os 20 *bytes* representando o último *commit* forem seguro, é impossível de enganar um repositório Git.

Inteligência

Como o Git sabe que um arquivo foi renomeado, mesmo se você nunca mencionou 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 analise heurística do Git verifica além das ações de renomear e de copias sucessivas entre versões. De fato, ele pode detectar pedaços de código sendo movidos ou copiados entre arquivos! Embora não cubra todos os casos, faz um traba-

lho 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.

Repositórios Crus

Você pode estar imaginando que formato o repositório online do Git usa. Eles são repositórios Git simples, iguais ao seu diretório `.git`, exceto que eles tem nomes como `proj.git`, e não tem nenhum diretório de trabalho associado a eles.

A maioria dos comandos do Git esperam que o *index* do Git esteja no `.git`, e falharão nos repositórios crus. Corrija isto configurando a variável de ambiente `GIT_DIR` com o caminho do seu repositório cru, ou executando o Git a partir deste diretório com a opção `--bare`.

Origem do Git

Esta mensagem na lista de discução do Linux Kernel[28] descreve a sequência de eventos que levaram ao Git. A discussão inteira é um sitio arqueológico fascinante para historiadores do Git.

28 http://lkml.org/lkml/2005/4/6/121

Atalhos 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 espera por uma solução. Ou melhor ainda, solucione-as e ajude a todos!

Estive brincando com algumas ideias para sistemas de controle de versões, e escrevi um sistema experimental baseado no Git[29], que aborda algumas desses questões.

Microsoft Windows

O Git no *Microsoft Windows* pode ser trabalhoso:

- Cygwin[30], um ambiente que deixa o *Windows* parecido com o *Linux*, tem uma versão do Git para Windows[31].
- Git no MSys[32] é um 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 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 força-lo a explicitamente marcar um arquivo de alguma maneira antes de edita-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;

29 http://www-cs-students.stanford.edu/~blynn/gg/
30 http://cygwin.com/
31 http://cygwin.com/packages/git/
32 http://code.google.com/p/msysgit/

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

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

A penalidade é usualmente rápida, e vale a pena devido a eficiência que dá às 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 trabalhoso 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 oco 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 versão pra versão. 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 pra ajudar, mas os usuários padrões 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 possa ser trocado. 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 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 ocos, que adiciona pouco ou quase nada ao histórico do projeto. É claro, que muitas ferramentas do Git se tronaram inadequadas, 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 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 cria e distribui um *script* que usa o Git para clonar o código e o ***rsync*** ou um clone oco 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 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, ou talvez uma marca, 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 atual implementação do Git, e não seu o *design*, é a razão deste inconveniente. Com sorte, uma vez que o Git ganhe mais tração, 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.

Há benefícios ao Git por definir o *commit* zero: assim que um repositório é construído, o *HEAD* será definido para uma sequência constituída de 20 *bytes* zero. Este *commit* especial representa um *tree* vazio, sem predecessor, num momento anterior a todos os repositórios Git.

Se em seguida for executado por exemplo, o `git log`, será informado ao usuário que ainda não foi realizado nenhum *commit*, ao invés de terminar devido um erro fatal. Similar a outras ferramentas.

Todos *commit* inicial é implicitamente um descendente deste *commit* zero. Por exemplo, fazendo um *rebase* num *branch* não monitorado pode ocasionar um enxerto de todo o *branch* no destino. Atualmente, todos, inclusive o *commit* inicial, serão enxertados, resultando num conflito de *merge*. Uma solução é usar `git checkout` seguido de `git commit -C` no *commit* inicial, e um *rebase* no restante.

Infelizmente há casos piores. Se vários *branch* com diferentes *commit* iniciais forem mesclados (*merge*), então um *rebase* do resultado vai requer uma substancial intervenção manual.