Conhecendo o Git

Aprenda de forma prática

Eustáquio Rangel

Conhecendo o Git

Eustáquio Rangel de Oliveira Jr.

Esse livro está à venda em http://leanpub.com/conhecendo-o-git

Essa versão foi publicada em 2017-08-24

Esse é um livro Leanpub. A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. Publicação Lean é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

Tweet Sobre Esse Livro!

Por favor ajude Eustáquio Rangel de Oliveira Jr. a divulgar esse livro no Twitter!

O tweet sugerido para esse livro é:

Comprei o ebook ”Conhecendo o Git”, do @taq!

A hashtag sugerida para esse livro é #conhecendo-o-git.

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

https://twitter.com/search?q=#conhecendo-o-git

Conteúdo

Logotipo do Git de autoria de Jason Long[1].

[1] http://twitter.com/jasonlong

Apresentação

O material nesse livro é resultado de alguns workshops que a minha empresa, a Bluefish[2], realizou sobre Git na região de São José do Rio Preto. Esse material é proveniente do meu uso diário de Git e de algumas anotações daqueles recursos menos utilizados, mas que quebram um baita galho, que ficam "no gatilho", quando preciso delas.

O Git é um software com muitos recursos. Eu tento aqui fazer uma introdução ao mundo dos VCSs e mostrar como que podemos utilizar seus recursos no dia-a-dia, com certeza no tópico avançado daria para aprofundar muito mais na escovação de bits, mas preferi não carregar muito para não assustar o marinheiro de primeira viagem. Mesmo nos tópicos abordados, pode ser que algum que seria essencial tenha faltado, nesse caso, me desculpem e me deêm um puxão de orelha no meu Twitter (@taq).

Também gostaria de deixar claro que o conteúdo do livro vai ser mostrado muitas vezes com algum (ou bastante) humor. Não tem jeito, é o meu jeito costumeiro de ser, e a exceção é quando falta o humor, nesse caso, podem crer que eu estava escrevendo em um momento que ou estava cansado, preocupado ou chateado com alguma coisa. Já viu, não dá para ser bobo alegre todo o tempo com todas as responsabilidades da vida adulta. O sarcasmo é que quando podíamos ser bobos alegres geralmente tentávamos ser mais sérios para provar que estávamos crescendo e depois sentimos desesperadamente falta disso. Coisas da vida ...

Como diria um bom amigo, "enfim", espero que o conteúdo aqui seja de bom proveito e que possa ajudá-los ou mesmo instigá-los ao uso do Git e dos VCSs, ferramenta que considero essencial para o desenvolvedor.

[2] http://www.bluefish.com.br

Básico

O que é Git?

Git é um sistema distribuído de controle de versão desenhado para lidar com projetos pequenos até muito grandes, com velocidade e eficiência.

Cada clone do Git é um repositório completo com o histórico completo e capacidades de revisões completas, não dependente de acesso à rede ou de um servidor central. A criação e junção de ramos de desenvolvimento (branches) são rápidas e fáceis de fazer.

Mas o que diabos é controle de versão?

Citando a Wikipedia:

> Um sistema de **controle de versão** (ou **versionamento**), VCS (do inglês version control system) ou ainda SCM (do inglês source code management) na função prática da Ciência da Computação e da Engenharia de Software, é um software com a finalidade de gerenciar diferentes versões no desenvolvimento de um documento qualquer.
>
> Esses sistemas são comumente utilizados no desenvolvimento de software para controlar as diferentes versões histórico e desenvolvimento — dos códigos-fontes e também da documentação.
>
> As principais vantagens de se utilizar um sistema de controle de versão para rastrear as alterações feitas durante o desenvolvimento de software ou o desenvolvimento de um documento de texto qualquer são:
>
> - Controle do histórico: facilidade em desfazer e possibilidade de analisar o histórico do desenvolvimento, como também facilidade no resgate de versões mais antigas e estáveis. A maioria das implementações permitem analisar as alterações com detalhes, desde a primeira versão até a última.
>
> - Ramificação de projeto: a maioria das implementações possibilita a divisão do projeto em várias linhas de desenvolvimento, que podem ser trabalhadas paralelamente, sem que uma interfira na outra.

- Trabalho em equipe: um sistema de controle de versão permite que diversas pessoas trabalhem sobre o mesmo conjunto de documentos ao mesmo tempo e minimiza o desgaste provocado por problemas com conflitos de edições. É possível que a implementação também tenha um controle sofisticado de acesso para cada usuário ou grupo de usuários.

- Marcação e resgate de versões estáveis: a maioria dos sistemas permite marcar onde é que o documento estava com uma versão estável, podendo ser facilmente resgatado no futuro.

Sistemas de controle de versão (abreviados a partir de agora como VCS) são, em minha opinião, item obrigatório nas ferramentas de qualquer desenvolvedor que se considere profissional e sério.

E não é por modinha não: sistemas desse tipo, gratuitos, de código aberto e livres, existem há vários anos, e só não usa quem não quer ou que ainda gosta de fazer gambiarras como backups diários do código-fonte inteiro do projeto em arquivos compactados.

Mas meu compactado me serve bem!

Ok, o backup realmente faz o seu papel de tirar todos os ovos da mesma cesta (até que um dia é verificado que o arquivo compactado está com problemas ou o disco rígido "deu pau" ...) e até de manter um certo estado das alterações (para quem adora o diff - e tem gente que nem isso conhece), mas um backup consistente é a menor e mais básica de nossas necessidades ou obrigações, ou seja, nem vale contar como algo "extra". Inclusive, podemos e devemos fazer backup do repositório do VCS.

Quais são as vantagens de utilizar um VCS?

Vamos detalhar as expostas acima:

Controle do histórico

Quais seriam as respostas para as seguintes questões, levando em conta a não utilização de um VCS e deixando o código em algum diretório local ou da rede:

- Quem alterou o código?
- Que código foi alterado?
- Quando foi alterado?

Não seriam questões fáceis de responder sem um bom esforço e algum tipo de vodu para poder enxergar o passado, pois nessas horas, dependendo do porque essas perguntas estão sendo feitas, existem dois tipos de resposta, com uma alta probabilidade de ficarem sem resposta:

Foi uma bela de uma alteração que trouxe muitas vantagens - todos do time querem assumir as glórias. Fizeram caca - não foi ninguém, acha, quem faria uma nhaca dessas e não falaria, chefe? Enfim, um VCS te livra dessas perguntas, dando efetivamente as respostas: quem fez o que com que e quando. Se em uma equipe pequena ou mesmo para o desenvolvedor individual isso já ajuda, imagem em equipes grandes. Ou nas que tem aquele tipo de espertinho que sempre joga a batata-quente para os outros (e ninguém entende porque diabos ele ainda está no time).

Ramificação do projeto

Quem nunca estava trabalhando em uma feature nova e aparece o chefe desesperado pedindo ASAP para que algo no código de produção seja alterado e feito o deploy em caráter emergencial? Aí você já está trabalhando em código que **não está bom para produção**, como fazer? Pegar o backup de ontem, rezar para estar ok, descompactar em uma outra área, fazer as alterações do chefe que já está enfartando (chefes tem mania de serem meio desesperados assim mesmo - infelizmente), subir para produção e depois dar um jeito de incorporar o código que você acabou de alterar com o seu com a feature nova que você estava trabalhando antes? Pelamor.

Um bom VCS tem que fornecer um conceito otimizado e prático de branching. Uma branch nada mais é do que um "ramo" que o seu código toma, dentro da "árvore" do seu projeto. Para quem assistiu Fringe, é um dos universos paralelos ok? Você pode sair de onde está, ir para outro lugar baseado em um determinado "ramo" ou "universo" que você aponta, aprontar todas lá e se der certo, pedir para fazer um merge, uma fusão, de um universo com outro (só o Peter que ia escapar dessa). Se você ficou bebâdo demais no outro universo e fez um monte de caca, você também pode voltar para onde estava e apagar o dito cujo. É, triste, mata todo mundo, mas serve para isso também.

Enfim, com branches, na situação descrita acima, você poderia sair do ramo onde estava testando a sua feature nova, criando um outro baseado no ramo de produção (você não estava trabalhando nele, estava?), fazer as alterações pedidas pelo chefe (que nessas alturas já está espumando parecendo que enfiou um Sonrisal na boca ou foi mordido por um cachorro louco), voltar para produção, fazer o merge da branch da feature nova, fazer o deploy (que vai salvar o seu chefe e vocẽ da aporrinhação dele), voltar para a branch da feature que você estava trabalhando e fazer o merge com as alterações que foram para produção. Parece complicado, mas não é. E o Git especialmente faz o conceito de branching e merge de maneira muito prática e eficiente, como vamos ver mais à frente.

Trabalho em equipe

Diretório compartihado na rede é muito ... anos 90. Se você tem uma equipe, ou mesmo desenvolvendo sozinho, mandar todo o seu código para um diretório e trabalhar direto nele não é mais produtivo, a não ser que seja algo descartável sendo produzido apenas para rodar algumas poucas

vezes. Mesmo assim, ainda arrisco dizer, que pela facilidade do Git em criar repositórios, até para esses casos serve.

Mas quando temos um projeto mais sério, um VCS é praticamente obrigatório. Se alguém que está lendo isso acha que essa obrigatoriedade é papo-furado, espero que nos próximos capítulos eu consiga mudar essa idéia. Com um VCS, as pessoas do time podem cada uma recuperar uma cópia do código do repositório, fazer suas alterações e enviar de volta para o repositório. Um VCS como o Git permite até que essas cópias locais não dependam de conexão de rede, como alguns outros VCSs requerem, ou seja, a pessoa descarrega o código e só vai precisar de uma conexão de rede quando for enviar de volta para o repositório. Isso permite que você descarregue o código do repositório no seu computador portátil e o leve como você lá para aquele sítio no meio do mato onde não tem conexão nem com sinal de fumaça, trabalhar no projeto enquanto a turma se diverte e quando voltar para casa, mandar de volta para o repositório. Atenção: esse foi um exemplo extremo e infeliz da não-necessidade de conexão, não faça isso no final de semana naquele sítio legal … sua cara metade não vai ficar feliz e você vai ser muito, mais muito, nerd e workaholic.

Outra vantagem de um VCS em uma equipe é a **integração** das alterações individuais, mesmo que em um mesmo arquivo. Em um projeto grande, sem um VCS, coisas malucas podem acontecer se tentarmos comparar e integrar as alterações feitas por todos os desenvolvedores em um sistema desse tipo.

Marcação e resgate de versões estáveis

Essa é mais fácil de explicar. Ao invés de você pegar aquele arquivo chamado `projeto-versao-1.zip` ou coisa do tipo, você requisita ao repositório que te retorne a versão `1` do projeto. Também pode comparar versões etc e tal.

Quais as opções de VCS disponíveis por aí?

O primeiro sistema VCS que utilizei foi o CVS, e o seu primeiro release foi em 1986. Claro que não o utilizei nessa época, onde a minha maior preocupação era aprender alguns riffs para montar uma banda de Metal, mas tão logo conheci esse tipo de software, já passei a utilizá-lo e não larguei mais.

Eu não mudei do CVS para o Subversion pois - me desculpem que é realmente fã do Subversion - apesar das suas vantagens, ele não me apeteceu. Não parecia realmente um motivo fantástico para que eu mudasse de VCS, e como disse o Linus Torvalds um dia, o slogan do Subversion "CVS feito do jeito certo" já demonstrava que apesar de corrigir vários bugs e apresentar várias outras features além do CVS, ainda se baseava demais na "mentalidade" do CVS. Não havia alguma coisa realmente "uau", e por isso eu não fiz a migração de VCS em ordem cronológica, e sim por um fator "cara, isso é bom" que o Git trouxe.

Alguns exemplos de VCS populares de código aberto são:

- Bazaar

- Concurrent Version System (CVS)
- Darcs
- Mercurial
- Subversion (SVN)

E, é claro, o Git.

História do Git

O Git começou quando Linus Torvalds (é, ele mesmo, o cara do kernel Linux, hacker mutcho macho) teve que optar por uma alternativa ao Bitkeeper, que era o VCS que os desenvolvedores (milhares) do kernel utilizavam desde 2002 e que teve sua licença alterada após determinada versão, deixando de fornecer acesso gratuito (que era fornecido sob determinadas condições, como ninguém trabalhar em algum VCS concorrente).

O Bitkeeper não era um software livre, o que levou pessoas como Richard Stallman (sempre ele né) a argumentar contra o seu uso para o desenvolvimento do kernel do Linux, coisa que, como sempre, no começo trataram como mais "lenga-lenga do chato do Software Livre" mas que se mostrou bem válida quando em Julho de 2005 a Bitmover, que é a produtora do Bitkeeper, anunciou que não forneceria mais acesso gratuito. A razão para isso foi que um dos hackers da OSDL, Andrew "Tridge" Tridgell, desenvolveu um modo de exibir os metadados do repositório, item praticamente básico de um bom VCS mas somente disponibilizado para quem possuía uma versão comercial e paga do Bitkeeper. Nunca deixei de imaginar o Stallman fazendo fusquinha para o Linus nessa hora …

Fusquinha

"Fusquinha" é o ato de colocar as duas mãos espalmadas, uma de cada lado do rosto, na altura das bochechas, com os dedões encostando nas bochechas, fazendo um movimento de abrir e fechar os dedos de ambas as mãos enquanto mostra a língua. Tentem fazer isso quando acontecer algo similar ao que aconteceu com o Linus e a Bitmover, é altamente desestressante. E se você não aprender Git após ler esse livro, pelo menos aprendeu a fazer fusquinha.

Como o Linus não ia pagar licença para os milhares de desenvolvedores utilizarem um produto que, apesar de bom, estar ficando meio complicado, começou a analisar algumas opções existentes de VCS.

Bitkeeper exagerando

Nessa questão de complicar as coisas, chegaram ao cúmulo de manter as mesmas regras de não poder trabalhar em softwares similares **mesmo na versão comercial e paga**, inclusive entraram em contato com um cliente exigindo que parasse de contribuir com o Mercurial. Não sei bem porque diabos tinham tanto medo de ter alguma outra ferramenta de nível bom como concorrente a ponto de tentar impedir que pessoas trabalhassem nela.

Linus procurou pelas seguintes características:

- Não se basear no CVS
- Ser distribuído como o Bitkeeper
- Proteger contra corrompimento de arquivos
- Ser rápido, muito rápido, com alta performance.

Como não encontrou nenhum VCS que atendesse à todos esses requisitos, como o Linus é um cara bem mão-na-massa, começou a fazer o seu próprio VCS, o Git, e 4 dias (!!!) depois ele já estava sendo utilizado para armazenar o próprio código. Aí deve ter sido a vez do Linus fazer fusquinha para o pessoal da Bitmover.

Para o Git, o Linus queria e fez um design diferenciado para o Git (lembrem-se da questão CVS x SVN, ele não gastaria tempo fazendo algo que não achasse que fosse realmente uma evolução de VCS em termos de conceitos). Uma vez citou "sou um cara de sistema de arquivos", para explicar como que o Git trabalhava com as várias branches.

Origem do nome

Linus diz que, assim como no caso do Linux, baseou o nome em si mesmo. Nesse caso, no fato da palavra "git" ser uma gíria Inglesa que significa "teimoso, cabeçudo, cabeça-dura, que pensa que está sempre correto". Quem conhece o sujeito, mesmo que só pela fama, entende que ele está sendo sarcástico e ao mesmo tempo honesto. O problema é que grande parte das vezes ele está mesmo correto, e deixa isso claro das maneiras escritas e verbais mais Chuck Norris possíveis.

E dependendo do humor do Linus, o nome também pode significar:

- Apenas uma combinação randômica de letras
- Estúpido, abjeto, desprezível (alguém imaginou o Patolino falando isso?)
- "Global Information Tracker". Uau.
- E finalmente, quando dá algum problema (ei, nada é perfeito): "Goddamn idiotic truckload of shit"

Algumas vantagens sobre outros VCSs

Listando algumas dentre muitas:

- Distribuído, não depende de um repositório central.
- Rápido igual o capeta para lidar com branches. Alguns outros VCS fazem uma cópia completa do conteúdo de uma branch para criar outra branch e necessitam comunicar com o servidor para fazer isso. - O Git faz locamente e leva segundos (quando mais que apenas 1 segundo) enquanto outros levam minutos!

- Podemos fazer registrar nossas mudanças localmente, sem a necessidade de uma conexão de rede, e enviá-las para o repositório remoto quando estivermos conectados.
- Repositórios são bem menores (para o código do Mozilla, 30 vezes menor!)
- As branches carregam o seu histórico completo
- Veloz e escalável, não ficando mais lento conforme o tamanho do projeto aumenta
- E mais algumas que não compensa explicar agora se você ainda não conhece a ferramenta. ;-)

Onde hospedar os repositórios

Pode ser em algum diretório no seu computador pessoal, em um servidor na rede, em um serviço especializado nisso. Vamos ver como criar repositórios locais logo à frente, mas já ficam duas dicas de onde criar seus repositórios na "nuvem":

Github

Para mim, o Github[3] está sendo agora o que o SourceForge[4] foi na década de 90, hospedando muitos projetos de código aberto/livres, inclusive o código do próprio Git[5]. Para projetos de código aberto/livres, o Github fornece hospedagem gratuita, tendo planos pagos para repositórios privados. É a minha escolha atual.

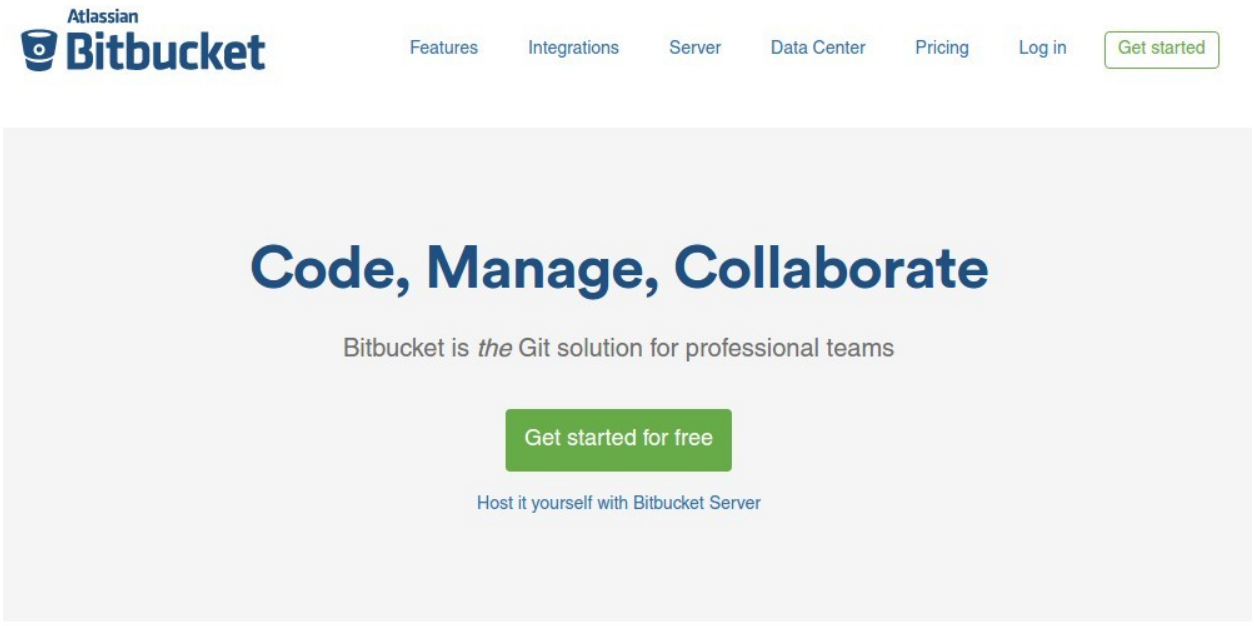
[3] http://github.com
[4] http://sourceforge.net
[5] https://github.com/git/git

Github

Bitbucket

O pessoal do Bitbucket[6] chegou atrasado para a festa, mas estão prestando um ótimo serviço, com o diferencial de repositórios privados e colaboradores públicos **ilimitados**!

[6] http://bitbucket.com


Atlassian
Bitbucket
Features
Integrations
Server
Data Center
Pricing
Log in
Get started
Code, Manage, Collaborate
Bitbucket is the Git solution for professional teams
Get started for free
Host it yourself with Bitbucket Server


Built for professional teams

Bitbucket

Instalando o Git

Linux e Unix

Para várias distribuições, sistemas e situações:

Debian/Ubuntu

$ apt-get install git-core

Fedora

$ yum install git

Gentoo

$ emerge –ask –verbose dev-vcs/git

FreeBSD

$ cd /usr/ports/devel/git

$ make install

Solaris 11 Express

$ pkg install developer/versioning/git

OpenBSD

$ pkg_add git

Código-fonte

https://github.com/git/git/tags

Mac

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

Windows

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

Configurações extras no GNU/Linux

Não, não precisamos de mais do que instalar os pacotes para fazer o Git funcionar no GNU/Linux (ei, não é só o kernel!), mas sim podemos utilizar algumas ferramentas para nos auxilarem a ser mais produtivos com o Git. Uma delas é o recurso de autocompletar. Para ativá-lo, devemos abrir (com o usuário necessário com permissão para tal) o arquivo `/etc/bash.bashrc` e remover os comentários das seguintes linhas (não contando o comentário):

```
1  # enable bash completion in interactive shells
2  if ! shopt -oq posix; then
3    if [ -f /usr/share/bash-completion/bash_completion ]; then
4      . /usr/share/bash-completion/bash_completion
5    elif [ -f /etc/bash_completion ]; then
6      . /etc/bash_completion
7    fi
8  fi
```

Isso vai nos permitir utilizar atalhos dos comandos do git na linha de comando.

Hein? Linha de comando? Cruz-credo. Estamos em que ano mesmo?

Epa, peraí. Antes de entrarmos em alguma discussão sobre terminais, IDEs e produtividade, vamos deixar uma coisa clara aqui: vamos aprender os conceitos do Git utilizando o ambiente mínimo para o seu uso, que é um terminal. Nada contra quem queira fazer integração em alguma IDE ou coisa do tipo, mas aqui nesse livro vamos digitar os comandos no terminal mesmo. E arrisco a dizer que algumas pessoas que não acham isso produtivo vão mudar de idéia depois ...

Após configurado o autocompletar, não só o Git mas qualquer software que tenha suporte para esse recurso no GNU/Linux vai estar ativo. Para verificar alguns, é só ver a lista no diretório `/etc/bash_completion.d`:

```
$ ls /etc/bash_completion.d/
total 1,1M
drwxr-xr-x   3 root root 4,0K Mai 24 22:37 .
drwxr-xr-x 149 root root  12K Mai 25 08:31 ..
-rw-r--r--   1 root root 1,4K Mar 30 21:10 abook
-rw-r--r--   1 root root 3,2K Set 18  2009 ack-grep
-rw-r--r--   1 root root 2,0K Mar 30 21:10 ant
-rw-r--r--   1 root root  476 Mar 30 21:10 apache2ctl
-rw-r--r--   1 root root 6,5K Abr 10 10:28 apport_completion
...
```

Se corretamente habilitado, quando digitarmos git re e apertarmos tab, aparecerá algo do tipo:

```
$ git re
rebase         relink         repack         request-pull   revert
reflog         remote         replace        reset
```

Configuração do Git

Não podemos ser estranhos para o Git, e é falta de educação não se apresentar à ele e do jeito que ele vai nos quebrar o galho, devíamos até pedir a benção, mas como não tem jeito, vamos nos apresentar mesmo. No terminal, digite algo do tipo (substitua por seu nome e e-mail) :

```
git config --global user.name "Eustaquio Rangel"
git config --global user.email "eustaquiorangel@gmail.com"
```

Escopos de configuração

Podemos ver que utilizamos `--global` para as configurações de nome e email. Essa opção vai armazenar as configurações no arquivo `~/.gitconfig`, onde ~ é a abreviação de `/home/<user>` em sistemas Unix-like, ou seja, é uma configuração definida para cada usuário e o ponto no ínicio do nome do arquivo indica que ele é um arquivo "escondido" (*hidden*).

Se quisermos uma configuração compartilhada para todos os usuários do computador em questão, podemos utilizar a opção `--system`, que vai armazenar as configurações no arquivo `/etc/gitconfig`.

Ou ainda para uma configuração mais restrita, utilizar `--local`, que armazena as configurações no diretório do repositório corrente.

Formato do arquivo de configurações

Dando uma olhada no arquivo `~/.gitconfig` (no Windows ® fica em `/C/Users/<user>`):

```
$ cat ~/.gitconfig
[user]
name = Eustaquio Rangel
email = eustaquiorangel@gmail.com
```

Onde:

```
[user] uma seção
name uma chave
"Eustaquio Rangel" um valor
```

Se lermos em algum lugar, "altere o **valor** de **name** na seção **user**", já sabemos o que fazer.

Algumas outras configurações:

```
[core]
   editor = vim
[color]
   log = auto
   ui = auto
   branch = auto
   diff = auto
   interactive = auto
   status = auto
```

Configurando atalhos

Além do autocompletar, temos um recurso para digitar comandos do Git de um jeito bem veloz. Podemos configurar atalhos para os comandos que mais usamos (e também para os outros, à vontade), que ficarão armazenados no arquivo de configurações como demonstrado acima.

Para configurar atalhos para, por exemplo, um comando do tipo `git status`, podemos fazer da seguinte maneira:

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

Isso vai produzir as seguintes linhas no arquivo de configuração:

```
[alias]
   st = status
```

Também podemos configurar direto no arquivo, o que eu até acho mais produtivo do que ficar decorando os comandos para adicionar na linha de comando. Por exemplo, para criarmos um atalho para `git init` direto no arquivo de configurações:

```
[alias]
   st = status
   in = init
```

Dica de atalho

Durante o decorrer do livro, vamos ir encontrando várias dicas de comandos customizados do Git que valem a pena serem convertidos em atalhos, fiquem de olho nessas caixas de texto.

Trabalhando com repositórios

Tipos de repositórios

Temos que entender alguns conceitos sobre repositórios no Git: temos repositórios "normais", e os do tipo *bare* (algum fissurado em p0rn já deu uma risadinha aí que eu sei, besteirentos). Os repositórios "normais" contém informações do VCS (em um diretório chamado … ahn … `.git`) e os arquivos do projeto (a *working tree*), enquanto que um repositório bare somente contém as informações.

Como regrinha básica, se você for **trabalhar** em um diretório com controle do Git, você precisa de um diretório "normal", um working directory. Se você quer **compartilhar** alguma coisa, um repositório de onde as pessoas podem enviar (`push`, a partir da versão 1.7.0 do Git) dados, você precisa de um diretório *bare*. Ou seja: sozinho, *working directory*, outros mandando coisas pra dentro, *bare*. Eita analogia sem-vergonha.

Criando um repositório bare

Vamos convencionar que vamos utilizar o diretório `~/git` para executarmos os exemplos aqui do livro. Vamos executar os seguintes comandos e dar uma olhada nos resultados:

```
1   $ cd ~/git
2   $ mkdir repobare
3   $ cd repobare
4   $ git init --bare
5   Initialized empty Git repository in /tmp/repobare/
6   $ ls
7   total 40K
8   drwxrwxr-x 7  taq  taq  4,0K .
9   drwxrwxrwt 17 root root 4,0K ..
10  drwxrwxr-x 2  taq  taq  4,0K branches
11  -rw-rw-r-- 1  taq  taq  66   config
12  -rw-rw-r-- 1  taq  taq  73   description
13  -rw-rw-r-- 1  taq  taq  23   HEAD
14  drwxrwxr-x 2  taq  taq  4,0K hooks
```

O destaque aqui vai para `git init --bare`. Esse é o método de criar um repositório bare. Só isso! Esse comando vai criar no diretório corrente toda a estrutura para o repositório.

Dica de Atalho

Também poderia ser criado a partir do diretório corrente com

```
1    git init --bare repobare
```

Criando um repositório comum

Para criar um repositório do tipo *working directory*, é só executarmos:

```
1   $ cd ~/git
2   $ mkdir repo
3   $ cd repo
4   $ git init
5   Initialized empty Git repository in /tmp/repo/.git/
6
7   $ ls
8   total 12K
9   drwxrwxr-x 3  taq  taq  4,0K .
10  drwxrwxrwt 16 root root 4,0K ..
11  drwxrwxr-x 7  taq  taq  4,0K .git
12
13  $ ls .git
14  total 40K
15  drwxrwxr-x 7 taq taq 4,0K .
16  drwxrwxr-x 3 taq taq 4,0K ..
17  drwxrwxr-x 2 taq taq 4,0K branches
18  ...
```

Aqui pudemos ver que o `git init` cria a estrutura necessária para um *working directory*, onde é criado o diretório `.git`, onde podemos ver o seu conteúdo sem problemas.

Dica de Atalho

Também poderia ser criado a partir do diretório corrente com

```
1    git init repo
```

Adicionando conteúdo no seu repositório

Dentro do nosso *working directory*, primeiro vamos dar uma olhada em que pé que o repositório se encontra utilizando `git status` (ou `git st`, como criado no atalho, ou `git st<tab>` para acionar o autocompletar):

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

Isso mostra que ainda não existe conteúdo no repositório. Podemos criar um arquivo `README` com qualquer tipo de conteúdo para ver como fica, usando `git status`:

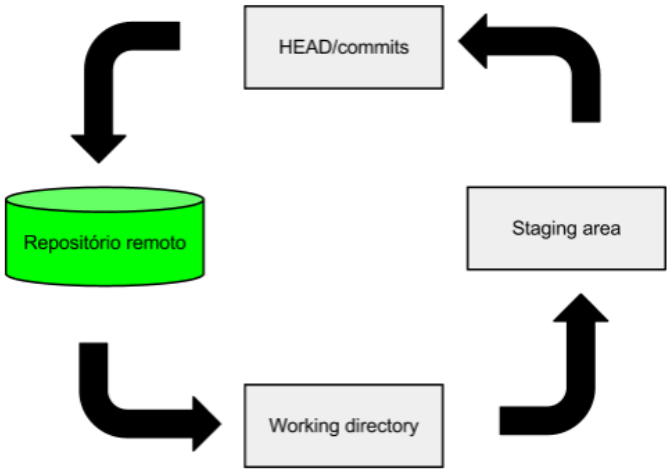
```
$ echo "Projeto do livro de Git" > README
$ ls
total 16K
drwxrwxr-x 3 taq taq 4,0K 19:37 .
drwxr-xr-x 5 taq taq 4,0K 19:37 ..
drwxrwxr-x 7 taq taq 4,0K 19:37 .git
-rw-rw-r-- 1 taq taq   20 19:38 README
$ git st
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  README
nothing added to commit but untracked files present (use "git add" to track)
```

Podemos ver que o arquivo foi efetivamente criado, porém o Git nos dá uma mensagem de `Untracked files`, indicando que o arquivo ainda não está efetivamente sob controle do VCS.

Estados, áreas e árvores

No Git temos três estados (ou áreas, ou árvores, depende de quem define) que são:

1. O *working directory*
2. A *staging area*
3. O repositório (*head/commits*)



Estados, áreas e árvores

Quando estamos trabalhando dentro do nosso código local, como já dito anteriormente, estamos dentro do working directory, onde acabamos de criar um arquivo novo (`README`), que **ainda não foi apresentado ao VCS**.

Quando "*apresentamos*" um arquivo ao VCS, indicamos que queremos que ele esteja sob controle. Nem todos os arquivos que temos no *working directory* nos interessam para estarem sob controle do VCS, como por exemplo, arquivos temporários, bancos de dados descartáveis, etc, por isso que temos que ou ser seletivos na hora de indicarmos quais arquivos que nos interessam ou definirmos que tipos de arquivos não vão ser incluídos no VCS de forma alguma.

Ignorando arquivos

Para ignorarmos determinados arquivos no VCS, criamos e configuramos o arquivo `.gitignore` no diretório do repositório *local* com os paths/máscaras que desejarmos, uma por linha, como por exemplo:

```
1  $ cat .gitignore
2  *.swp
```

Isso vai ignorar todos os arquivos escondidos com a extensão `swp`, geralmente criados pelo editor Vim.

Podemos também criar um comportamento global para ignorar arquivos em todos os repositórios utilizados no computador, criando um arquivo `.gitignore` no nosso diretório `home` (levando em

conta nesse exemplo sistemas derivados de Unix), que pode ser abreviado com `~/.gitignore`, e especificar nas configurações do Git que esse arquivo deve ser utilizado, com:

```
git config --global core.excludesfile ~/.gitignore
```

Nesse caso, serão levados em conta os padrões encontrados em `~/.gitignore` e no `.gitignore` local, se ele existir.

Adicionando conteúdo

Conhecendo os commits

O Linus apesar de dar umas patadas às vezes é bem bonzinho. O próprio Git nos dá dicas de como executarmos determinados comandos relacionados com o estado corrente do repositório. No caso do `git status` que utilizamos acima logo após criar o arquivo `README`, temos:

```
  # Untracked files:
  #   (use "git add <file>..." to include in what will be committed)
```

O Git nos diz "ei, se você quiser apresentar algo para mim, use o `git add`", e é o que vamos fazer agora:

```
  $ git add README
  $ git status
  # On branch master
  # Initial commit
  # Changes to be committed:
  #   (use "git rm --cached <file>..." to unstage)
  #  new file:   README
```

Agora o arquivo `README` saiu do *working directory* e foi parar na **staging area**. Nesse estado, o VCS já sabe que o arquivo está sob seu controle, armazenando o seu histórico de alterações, e pronto para ser registrado como parte de um `commit`, que é o registro das alterações de um ou mais arquivos que foram alterados no *working directory*, deixando o código do repositório local pronto para ir para o repositório remoto.

Removendo arquivos da staging area

No momento da primeira inserção do arquivo no VCS, ele também dá a dica de como remover arquivos que porventura estejam errados ou que inserimos por engano na *staging area*, através de `git rm --cached <file>`. Isso vai retornar o estado do arquivo para o working directory.

Registrando o conteúdo

Agora que temos conteúdo na *staging area*, podemos registrar o que foi alterado e deixar preparado para enviarmos para o repositório remoto. Isso é feito através de um `commit`, que registra os arquivos que foram alterados, criados, quem e quando fez as alterações (respondendo as questões feitas no começo do livro).

Ou seja, um **`commit` é uma unidade de controle de vários arquivos e operações que foram feitas no repositório**, feito para registrar essa unidade de uma forma que o agrupamento das mesmas faça algum sentido, seja pela implementação ou remoção de uma *feature*, por um capítulo feito na documentação etc.

Fica uma dica: pensem no conteúdo de um `commit` como um conjunto de alterações que poderiam ser desfeitas mais tarde ou aplicadas em outro local, de modo a fazer todas as alterações necessárias para um determinado comportamento. Por exemplo, "quero desfazer os testes unitários, funcionais e de integração de determinado recurso" ou "quero aplicar somente os testes unitários, funcionais e de integração de determinado recurso em outra branch, sem necessariamente fazer a fusão completa dela".

Os `commits` do Git diferem de alguns outros VCS's por poderem ser feitos sem a necessidade de conexão de rede. Alguns VCS's exigem a conexão para registrarem um `commit`.

O comando para registrar um commit é ... `git commit`. Para registrarmos um `commit`, precisamos de uma mensagem descritiva do que foi feito.

Apelo e puxada de orelha sobre commits

Pelamor, eu disse *mensagem descritiva*. Já ouvi casos de gente escrevendo "não sei", "tô com sono", "alterei" e barbaridades do tipo. Se você está interessado em utilizar um VCS mostra que você já é bem grandinho para poder utilizar a ferramenta de maneira decente, então faça um favor para os outros desenvolvedores ou até para você mesmo e utilize uma mensagem que preste nessa hora!

Usando templates nas mensagens

Podemos definir um arquivo de *template* para as nossas mensagens, como por exemplo:

```
$ cat ~/.gitmessage
Commit message

Explanation

[Delivers #]
```

Nesse arquivo, deixo na primeira linha uma indicação de que a mensagem do *commit* vai ali, duas linhas abaixo um texto indicando uma explicação do que foi feito, e duas linhas abaixo um texto formatado para o número de uma tarefa para integração com o Pivotal Tracker[7].

Para utilizar esse arquivo (ou qualquer outro) como *template*, configure no seu arquivo `.gitconfig` o *template* do *commit*:

```
[commit]
   template = ~/.gitmessage
```

Configurando um editor padrão

A mensagem pode ser registrada através do editor da sua preferência (que rode no terminal ou com GUI), configurado previamente com o `git config`:

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

$ cat ~/.gitconfig
...
[core]
   editor = vim
...
```

No exemplo acima, o Vim[8] foi configurado como editor padrão.

Temos duas abordagens para mensagens de commit:

[7] https://www.pivotaltracker.com/signin
[8] http://www.vim.org

Mensagens com uma linha

Mensagens com apenas uma linha podem ser digitadas direto na linha de comando usando a opção `-m` do comando `git commit`, como vamos utilizar a seguir:

```
$ git commit -am "Primeiro commit"
[master (root-commit) 724dd31] Primeiro commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
$ git status
# On branch master
nothing to commit (working directory clean)
```

Enviando automaticamente arquivos para a staging area

Vale notar a opção `-a`: ela indica que todos os arquivos que foram modificados ou apagados devem ser automaticamente enviados para a *staging area*, incluindo-os no `commit`.

Mensagens com mais de uma linha

Para mensagens com mais de uma linha, é recomendado abrir o editor configurado acima e digitar a mensagem lá, com um limite de 50 caracteres na linha de "título", seguida por uma linha em branco e 72 caracteres por linha abaixo.

Apesar de muitos dizerem "estamos em 2000 e tantos, eu tenho um baita monitor de trocentas polegadas, para que esse limite?", existem algumas convenções que não vão arrancar pedaço de ninguém em seguir e melhoram a integração com várias ferramentas. Algumas ferramentas, como o Github por exemplo, mostram a primeira linha com uma formatação diferenciada. Mas cada um é cada um, use do jeito que quiser.

Se no exemplo anterior quiséssemos uma mensagem mais descritiva (apesar da puxada de orelha, ei, esse primeiro `commit` não precisa de uma), poderíamos ter utilizado:

```
$ git commit -a "Primeiro commit"
 0 Primeiro commit do projeto.
 1
 2 Esse commit e o primeiro do nosso projeto.
 3 Estamos comecando a conhecer o Git.
 4 # Please enter the commit message for your changes. Lines starting
 5 # with '#' will be ignored, and an empty message aborts the commit.
 6 # On branch master
 7 #
 8 # Initial commit
 9 #
10 # Changes to be committed:
11 #   (use "git rm --cached <file>..." to unstage)
12 #
13 #  new file:   README
14 #
```

Se não utilizarmos a opção `-m`, automaticamente o Git abre o editor para que digitemos a mensagem do `commit`.

Verificando o log

Agora vamos dar uma olhada nas alterações que foram feitas no nosso *working directory*, utilizando `git log`:

```
$ git log
commit 724dd31db62a58ba73d036b438b7e23770f561b0
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Mon Oct 24 21:08:46 2016 -0200
    Primeiro commit
```

Também podemos ver que arquivos foram alterados, se utilizamos a opção `–stat`:

```
$ git log --stat
commit 724dd31db62a58ba73d036b438b7e23770f561b0
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Mon Oct 24 21:08:46 2016 -0200

    Primeiro commit

 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Ali pudemos ver que houveram alterações no arquivo `README`, indicadas como uma linha inserida (`1+`).

Dica de atalho

Podemos ter alguns atalhos para nos mostrar o que foi alterado no último dia, semana e mês:

```
  lastday   = log --since '1 day ago'   --oneline
  lastweek  = log --since '1 week ago'  --oneline
  lastmonth = log --since '1 month ago' --oneline
```

Identificando os commits

Conceito básico e muito importante no Git: todos os `commits` são identificados através do seu `SHA-1`, que é um *hash* do conteúdo do `commit`, como no exemplo acima:

```
$ git log --stat
commit 724dd31db62a58ba73d036b438b7e23770f561b0
```

O `SHA-1` desse `commit` é `724dd31db62a58ba73d036b438b7e23770f561b0`, e vai ser gerado de forma única, nunca se repetindo no repositório, nos permitindo identificar e manipular essa série de alterações mais tarde. Como muito pouca gente tem saco para ficar copiando essa baita sequência de caracteres (mesmo que seja para a área de transferência do computador) e muito menos gente tem memória eidética como o Sheldon Cooper, podemos referenciar o `SHA-1` pelos seus primeiros `7` dígitos, ou seja, nesse caso, `724dd31`.

Adicionando repositórios remotos

Agora que já temos algum conteúdo, vamos adicionar alguns repositórios remotos para onde podemos mandar nossas alterações. Até o momento estão no diretório do nosso *working directory*. O Git nos dá uma facilidade grande para isso, pois aceita muitos repositórios remotos para onde podemos enviar e recuperar alterações.

É importante notar que a partir da versão 1.7.0 do Git, só podemos enviar as alterações para um repositório *bare* (olha ele aí de novo). Como já temos um criado, vamos indicar em nosso working directory o caminho:

```
$ git remote add origin ~/git/repobare/
```

Isso vai gerar o seguinte conteúdo em nosso arquivo `.gitconfig`:

```
$ cat .git/config
[core]
   repositoryformatversion = 0
   filemode = true
   bare = false
   logallrefupdates = true
[remote "origin"]
   url = /home/taq/git/repobare/
   fetch = +refs/heads/*:refs/remotes/origin/*
```

No comando `git remote` acima, indicamos:

- Que queremos adicionar um repositório, através de `add`.
- Que o nome de referência dele vai ser `origin`. A referência `origin` é a referência do repositório default quando não especificamos algum outro.
- Que o caminho do repositório é ~/git/repobare/. Aqui utilizamos um caminho do nosso sistema de arquivos, mas é mais comum utlizarmos um caminho de rede. Em repositórios do Github, por exemplo, seria algo como `git@github.com:<usuário>/<repositório>.git` e no Bitbucket `git@bitbucket.org/<usuário>/<repositório>`.

Agora podemos enviar o conteúdo do nosso *working directory* para o repositório remoto:

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 236 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /home/taq/git/repobare/
 * [new branch]      master -> master
```

Aqui utilizamos as seguintes opções:

- O comando `push` indica que queremos enviar o conteúdo do nosso *working directory* para o repositório remoto.
- Especificamos que queremos enviar para o repositório `origin`.
- Especificamos que queremos enviar o conteúdo para a *branch* `master`. Vamos ver *branches* mais à frente, mas já fica a dica que `master` é a *branch default* e que na primeira vez que enviamos o conteúdo do *working directory* devemos especificar que queremos enviar para a *branch master*, que nesse caso, por ser a primeira vez que o repositório `bare` vai receber conteúdo, vai ser criada no repositório remoto.

Para provar que o conteúdo foi enviado corretamente para o repositório remoto, vamos apagar o diretório do nosso *working copy*:

```
$ cd ~/git
$ rm -rf repo
$ ls repo
ls: impossível acessar repo: Arquivo ou diretorio nao encontrado
```

Clonando repositórios

Agora vamos recuperar o conteúdo do nosso repositório remoto:

```
$ git clone ~/git/repobare repo
Cloning into repo...
done.
```

Utilizamos `git clone` seguido do caminho onde está o repositório remoto (`~/git/repobare`) e nesse caso o nome do diretório que deve ser criado para o clone, `repo`. Se não for enviado esse último parâmetro, seria clonado em um diretório com o mesmo nome do repositório remoto (`repobare`, que nesse caso já existe).

O resultado é algo como:

```
$ ls repo
total 16K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 15 root root 4,0K ..
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    20 README
```

Olha só o nosso `README` de volta!

Atalhos para repositórios

Podemos configurar atalhos para os nossos repositórios no arquivo `.gitconfig`:

```
[url "git@github.com:taq/"]
        insteadOf = github:

[url "git@bitbucket.org:taq/"]
   insteadOf = bitbucket:
```

O que nesse caso nos permite digitar somente `git clone github:<repositório>` ao invés de `git@github.com:taq/<repositório>` e `git clone bitbucket:<repositório>` ao invés de `git clone git@bitbucket.org:taq/<repositório>`.

Branches

Criando uma branch

Agora vamos utilizar um dos principais recursos de um VCS e que o Git tira de letra: a criação (e também mudança, deleção) de *branches*. *Branches* são ramos de desenvolvimento onde podemos alterar nossos arquivos de modo que fiquem separados uns dos outros, para que mais tarde possamos fundir o seu conteúdo com outra outra branch, de por exemplo, código de produção, ou apagar a *branch* se fizemos alguma coisa que não ficou correta, sem alterar qualquer outro ponto.

Vamos pegar como o exemplo do chefe desesperado mencionado no começo do livro. A estrutura que temos é a seguinte:

- A *branch* `master` é a onde fica o nosso código de produção. Ali fica o código que é enviado para os servidores para rodar o nosso sistema para valer.
- Como somos prevenidos, toda santa vez que clonamos um repositório **não trabalhamos na nossa *branch* `master`**. Sacaram? Não. Trabalhem. Na. Branch. Master. Depois não digam que eu não avisei! Então, logo após clonar o repositório, criamos uma branch chamada `work` (ou `temp`, ou `coding`, whatever, pode ser o nome que quisermos mas é bom utilizarmos um nome **descritivo**).

Nisso chega o chefe desesperado, arrancando os cabelos da cabeça, pois o código de produção precisa ser alterado urgente e enviado para os servidores, por causa de uma nova feature que os clientes precisam (alguma coisa que o governo inventou, por exemplo).

Como você foi prevenido e está trabalhando em uma *branch* separada da `master`, você volta para a `master`, ficando as alterações feitas na `work` mantidas intocadas por lá (dependendo da situação, temos que fazer alguns procedimentos antes de voltar para a `master`, mas isso vai ser dito mais à frente), cria uma nova `branch` a partir da `master` (lembre-se: **não trabalhe na master**, faça disso um mantra), chamada, por exemplo, `novorecurso`, muda para essa **branch criada**, altera o código, faz os testes, e logo após, se, e eu disse se, tudo estiver correto, testado (você utiliza testes no seu sistema, não?), muda para a *branch* `master`, faz a fusão do código da *branch* `novorecurso` e envia para produção nos servidores, apagando a `novorecurso` logo após.

Como o chefe agora está feliz, você pode voltar a trabalhar no que estava trabalhando na *branch* `work`. Assim, você volta para lá e faz a fusão do código que alterou na `master` (senão a sua *branch* que era a mais recente vai ficar desatualizada com o que você acabou de alterar!) e continua a trabalhar por lá. No final do dia, você pode tanto fazer a fusão da `work` na `master` ou enviar a `work` para o repositório remoto para ser recuperada mais tarde para que possa trabalhar nela novamente.

Para criar a *branch* `work`, a partir da `master`, é só utilizar o comando `git checkout`:

```
$ git checkout -b work
Switched to a new branch 'work'
```

Customizando o prompt

Logo acima podemos ver como está o *prompt* do meu terminal, indicando, no final, se eu estiver dentro de um diretório controlado pelo Git, qual é a *branch* corrente. Isso é possível se habilitamos o autocompletar para o Git, conforme visto anteriormente, o que vai nos dar de brinde, entre outros, a variável `__git_ps1`, que indica justamente a *branch* corrente. Ela pode ser utilizada no prompt da seguinte maneira, para o resultado acima:

```
$ export PS1='\u@\h\w $(__git_ps1 "%s"): '
```

Reparem que utilizei um espaço em branco antes e após o nome da *branch*. Resultados similares podem ser testados diretamente na linha de comando:

```
$ __git_ps1               # (work)
$ __git_ps1 " %s "        # work
$ __git_ps1 "branch: [%s]" # branch: [work]
```

Fazendo alterações

Agora que temos algum conteúdo no repositório e criamos uma *branch* de trabalho, `work`, vamos fazer algumas alterações. A primeira vai ser uma errada, na linha 2 do arquivo `README`:

```
$ cat README
Projeto do livro de Git
Primeira alteração - errada! :-p
```

Consultando o log, é indicado que o arquivo `README` foi modificado:

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

Desfazendo alterações

Como modificamos o arquivo de forma errada, agora queremos restaurar ao estado que estava anteriormente às nossas modificações. Atenção que a operação seguinte irá retornar o arquivo até o estado que estava no último commit, não importa quantas vezes ele foi alterado! Para desfazer as alterações, o próprio Git nos dá a dica acima:

```
  # (use "git checkout -- <file>..." to discard changes in working directory)
```

Então usamos `git checkout README` no arquivo:

```
  $ cat README
  Projeto do livro de Git
  Primeira alteração - errada! :-p

  $ git checkout -- README
```

E ele volta a situação anterior:

```
  $ cat README
  Projeto do livro de Git
```

Comparando alterações nas branches

Agora vamos fazer uma alteração correta no `README`, inserindo a segunda linha com o seguinte conteúdo:

```
  $ cat README
  Projeto do livro de Git
  Primeira alteração correta do projeto
```

Podemos comparar agora o conteúdo da *branch* `master` com a *branch* atual, `work`:

```
$ git diff master
diff --git a/README b/README
index cb48e74..15c8ae1 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 Projeto do livro de Git
+Primeira alteração correta do projeto
```

Ali foi indicado que o arquivo que foi alterado (podem ser vários, o Git vai paginar o resultado) foi `README`, onde houve uma inserção (sinal de +) após a linha 1.

Agora é hora novamente de indicar quais foram as alterações feitas, assumindo o mérito (ou a culpa, brrr) por elas. Para isso, vamos novamente utilizar o nosso recém-conhecido amigo, `commit`, verificando o status logo em seguida com ... `status`:

```
$ git commit -am "Primeira alteração com sucesso"
[work 056cd5f] Primeira alteração com sucesso
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git status
# On branch work
nothing to commit (working directory clean)
```

Podemos ver que estamos em um estado limpo, ou seja, nada para inserir na *staging area*.

A partir da versão `2.10`, podemos configurar algumas cores do `diff`, como por exemplo utilizar fontes verdes em itálico para o código que entrou e vermelhas riscadas para o código que saiu:

```
$ git config --global color.diff.new "green italic"
$ git config --global color.diff.old "red strike"
```

Podemos ter um `diff` mas estiloso se instalarmos um pacote do `NPM` chamado `diff-so-fancy`:

```
$ npm install diff-so-fancy
```

Depois é só passar o resultado do `diff` convencional (com a opção `--color`) para o `diff-so-fancy` através de um *pipe*:

```
$ git diff --color | diff-so-fancy
```

Listando branches

Para ver uma lista das branches do *working directory*, podemos utilizar `git branch`:

```
$ git branch
  master
* work
```

O asterisco mostra qual é a *branch* corrente. Para mostrar quais são as *branches* remotas (se houver alguma), utilizamos `git branch -r`:

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
```

E para listarmos todas as branches, utilizamos `git branch -a`:

```
$ git branch -a
  master
* work
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
```

Alternando branches

Para alternar de uma *branch* para outra, utilizamos `git checkout`:

```
$ git branch
  master
* work

$ git checkout master
Switched to branch 'master'

$ git checkout work
Switched to branch 'work'
```

Também temos o atalho `-` para retornar para a *branch* em que estávamos anteriormente:

```
$ git checkout master

$ git checkout work

$ git checkout -

$ git branch
git branch
* master
work
```

Dica de atalho

Podemos fazer um atalho para nos mostrar como as *branches* estão se relacionando. Esse atalho é o `lpo` (ou qualquer nome que você deseje):

lpo = log –graph –all –format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%C(reset)' –abbrev-commit –date=relative

Testando esse atalho, temos um resultado como esse:

```
* 50e2007 - (3 dias atrás) README atualizado - Eustaquio Rangel
*   1683385 - (3 dias atrás) Merge branch 'master' of /home/taq/code/git/conhe\
cendo-o-git/repobare - Eustaquio Rangel
|\
| *   f0a5d61 - (3 dias atrás) Merge branch 'master' into work - Eustaquio Ran\
gel
| |\
| | *   70c12e6 - (3 dias atrás) Merge branch 'ghostbusters' - Eustaquio Rangel
| | |\
| | | * d0b78d0 - (3 dias atrás) Mais I's - Eustaquio Rangel
| * | | 5e9ff2b - (3 dias atrás) Inserida linha no README - Eustaquio Rangel
| |/ /
| * | 7f917fc - (3 dias atrás) Corrigido o título do livro - Eustaquio Rangel
| * | e7e9c16 - (3 dias atrás) Adicionados os K's - Eustaquio Rangel
* | | cde0e98 - (3 dias atrás) Adicionados os 1's - Eustaquio Rangel
* | | ebcf8e5 - (3 dias atrás) Adicionados os L's - Eustaquio Rangel
|/ /
```

Populando o working directory

Adicionando mais arquivos

Agora vamos definir uma convenção didática aqui: um VCS é ótimo para lidar com código, mas não somente isso. Eu costumo escrever alguns livros e tutoriais em LaTeX sob controle de um VCS (caso estejam curiosos, esse aqui não, pelo fato de ter que transformar em ODT para publicação e direto do LaTeX é meio chatinho ...) e utilizar para outras coisas como arquivos de configurações etc.

Para evitar puxar a sardinha para o lado de alguma linguagem (pô, eu ia escolher Ruby, vocês conhecem meu ebook da linguagem[9]?) e prejudicar a didática do ensino do VCS adicionando uma linguagem que o leitor possa não estar familiarizado, vamos utilizar aqui apenas arquivos plain text, os famigerados `.txt's`. O controle do VCS vai ser o mesmo do que utilizando em alguma linguagem e temos uma base universal e clara para testar nossos conhecimentos.

Vamos trabalhar com arquivos-texto utilizando as letras do alfabeto, como `a.txt`, `b.txt` etc. O conteúdo padrão do arquivo vai ser a mesma letra que dá o seu nome, repetida algumas vezes. Vamos criar alguns arquivos com os seguintes conteúdos:

```
$ echo "aaa" > a.txt && cat a.txt

$ echo "bbb" > b.txt && cat b.txt

$ echo "111" > 1.txt && cat 1.txt
```

Ok, eu disse letras e mostrei um arquivo com um número, mas relevem por enquanto. Vamos perguntar para o Git qual o status corrente do repositório:

```
$ git status
# On branch work
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  1.txt
#  a.txt
#  b.txt
```

[9] http://leanpub.com/conhecendo-ruby

Agora precisamos novamente apresentar os arquivos novos, ou seja, os que ainda estão fora do controle do VCS. Para isso utilizamos novamente `add`, onde podemos desde indicar que queremos mover para a *staging area* todos os arquivos do diretório corrente como passar uma máscara de arquivo:

```
$ git add *.txt

$ git status
# On branch work
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   1.txt
#  new file:   a.txt
#  new file:   b.txt
#
```

Arquivos devidamente apresentados para o Git e movidos para a *staging area*. Mas como pudemos perceber, o `1.txt` não faz parte da turma de arquivos que mencionei na declaração da didática, que vão utilizar somente letras, e não números. Ali o Git em toda a sua benevolência (se eu continuar escrevendo assim vou parecer o Linus) já nos dá a dica de como desfazer o que fizemos: utilizar `git reset HEAD <file>` para remover da área de *staging*:

```
$ git reset HEAD *.txt

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

Tudo de volta ao normal, vamos aprender agora como mover arquivos interativamente para a *staging area*. Lógico que poderíamos ter usado algo do tipo (**não façam isso agora**):

```
$ git add a.txt b.txt

$ git status
# On branch work
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   a.txt
#  new file:   b.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  1.txt
```

Funciona, e muito bem, mas vamos pedir uma licença funcional para mostrar como fazer isso interativamente através da opção `-i` do comando `add`.

Adicionando arquivos interativamente

Podemos utilizar `git add -i` para indicar quais arquivos que queremos mover para a área de *staging* (e algumas outras opções):

```
$ git add -i
        staged     unstaged path

*** Commands ***
  1: status     2: update     3: revert     4: add untracked
  5: patch      6: diff       7: quit       8: help
What now> 4
  1: 1.txt
  2: a.txt
  3: b.txt
Add untracked>> 2
  1: 1.txt
* 2: a.txt
  3: b.txt
Add untracked>> 3
  1: 1.txt
* 2: a.txt
* 3: b.txt
```

```
  Add untracked>>
  added 2 paths

  *** Commands ***
    1: status     2: update     3: revert     4: add untracked
    5: patch      6: diff       7: quit       8: help
  What now> 1
             staged     unstaged path
    1:        +1/-0      nothing a.txt
    2:        +1/-0      nothing b.txt

  *** Commands ***
    1: status     2: update     3: revert     4: add untracked
    5: patch      6: diff       7: quit       8: help
  What now> 7
  Bye.

  $ git st
  # On branch work
  # Changes to be committed:
  #   (use "git reset HEAD <file>..." to unstage)
  #
  #  new file:   a.txt
  #  new file:   b.txt
  #
  # Untracked files:
  #   (use "git add <file>..." to include in what will be committed)
  #
  #  1.txt
```

Fica a critério do usuário que abordagem utilizar, seja pela linha de comando seja por modo interativo. Algumas ferramentas fazem mover arquivos dentro e fora da área de *staging* uma tarefa mais fácil e intuitiva, como por exemplo o *plugin* `fugitive` do Vim.

Dica de atalho

Vamos configurar um atalho para remover os arquivos que enviamos para *staging* mas não queremos que fiquem por lá. Para isso, configurem no `.gitconfig` na seção `[alias]`:

```
  unstage = reset HEAD
```

Vamos testar essa dica de atalho, alterando o arquivo `a.txt`, enviando para *staging* com `add`,

removendo de lá com o novo `unstage` e retornando ao conteúdo original:

```
  $ echo "aaa" >> a.txt

  $ git status
  No ramo work
  Changes not staged for commit:
    (utilize "git add <arquivo>..." para atualizar o que será submetido)
    (utilize "git checkout -- <arquivo>..." para descartar mudanças no diretório\
 de trabalho)

          modified:   a.txt

  nenhuma modificação adicionada à submissão (utilize "git add" e/ou "git commit\
 -a")

  $ git add a.txt

  $ git status
  No ramo work
  Mudanças a serem submetidas:
    (use "git reset HEAD <file>..." to unstage)

          modified:   a.txt

  $ git unstage
  Unstaged changes after reset:
  M       a.txt

  $ git status
  No ramo work
  Changes not staged for commit:
    (utilize "git add <arquivo>..." para atualizar o que será submetido)
    (utilize "git checkout -- <arquivo>..." para descartar mudanças no diretório\
 de trabalho)

          modified:   a.txt

  nenhuma modificação adicionada à submissão (utilize "git add" e/ou "git commit\
 -a")

  $ git checkout -- a.txt

```

```
$ git status
No ramo work
nothing to commit, working tree clean
```

Removendo arquivos

Podemos remover arquivos do sistema de arquivos e do controle do VCS utilizando o comando `rm` (do Git, não do sistema!). Lógico que seria bem fácil fazer (não façam!) algo do tipo (também **não executem isso agora!**):

```
$ rm 1.txt
```

Mas vamos supor que enviamos o arquivo `1.txt` para a área de *staging*. Se tentarmos apagar o arquivo com `git rm`, vamos ter uma situação como essa:

```
$ git add 1.txt

$ git status
# On branch work
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   1.txt
#  new file:   a.txt
#  new file:   b.txt
#
$ git rm 1.txt
error: '1.txt' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
```

O Git nos avisa que o arquivo está na área de *staging*, e para remove-lo vamos precisar utilizar a opção `-f`:

```
$ git rm -f 1.txt
rm '1.txt'

$ git status
# On branch work
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   a.txt
#  new file:   b.txt
#

$ ls
total 24K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 16 root root 4,0K ..
-rw-rw-r--  1 taq  taq     4 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    60 README
```

Agora o arquivo foi efetivamente apagado! Vamos fazer um novo `commit` para registrar o que alteramos:

```
$ git commit -am "Iniciados os arquivos do alfabeto"
[work 312dbc0] Iniciados os arquivos do alfabeto
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 a.txt
 create mode 100644 b.txt

$ git log --stat
commit 312dbc096ab3056b0807fa6b9cbe4ca2e3326611
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Tue Oct 25 17:47:50 2016 -0200
    Iniciados os arquivos do alfabeto
 a.txt |    1 +
 b.txt |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
```

Utilizamos no log a opção `--stat`, para mostrar quais arquivos foram alterados em cada `commit`. Vamos comparar novamente a *branch* corrente, `work`, com a *branch* `master`:

```
$ git diff master
diff --git a/README b/README
index 124e2c3..82b27e3 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 Projeto do livro de Git
+Primeira alteração correta do projeto
diff --git a/a.txt b/a.txt
new file mode 100644
index 0000000..72943a1
--- /dev/null
+++ b/a.txt
@@ -0,0 +1 @@
+aaa
diff --git a/b.txt b/b.txt
new file mode 100644
index 0000000..f761ec1
--- /dev/null
+++ b/b.txt
@@ -0,0 +1 @@
+bbb
```

Apagando arquivos e diretórios que ainda não estão em staging

Se por acaso tivermos arquivos e diretórios que ainda não foram adicionados no *staging*, e quisermos apagar (isso, eu disse *apagar*, cuidado!) todos de uma vez, podemos utilizar `clean -fd`:

```
    $ echo "bla" > "bla.txt"

    $ echo "ble" > "ble.txt"

    $ echo "bli" > "bli.txt"

    $ mkdir teste

    $ echo "blo" > teste/blo.txt

    $ echo "blu" > teste/blu.txt

```

```
$ git status
No ramo work
Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)

        bla.txt
        ble.txt
        bli.txt
        teste/

$ git clean -fd
Removing bla.txt
Removing ble.txt
Removing bli.txt
Removing teste/

$ git status
No ramo work
nothing to commit, working tree clean<Paste>
```

Viram que utilizando `clean -fd` tanto os arquivos como os diretórios foram *apagados* sem choro! Se você não queria isso, pode apelidar o comando de `clean -f0d3u`, porque nessa hora não tem como recuperar mais os arquivos.

Dica de atalho

Podemos também fazer do jeito mariquinha, usando o atalho `trash`:

```
  trash = !mkdir -p .trash && git ls-files --others --exclude-standard | xargs m\
v -f -t .trash
```

O que esse atalho faz é criar um diretório chamado `.trash` (em Unix, arquivos ou diretórios começados com ponto (`.`) são escondidos) e vai mover os arquivos que ainda não são monitorados para lá, para que possam ser recuperados depois. A primeira coisa que tem que ser feita para usar esse atalho é incluir o diretório `.trash` no `.gitignore`, para que esse diretório não esteja para controle de versão:

```
$ echo ".trash" >> .gitignore
```

E agora vamos criar os arquivos e o diretório que apagamos acima novamente:

```
$ git status
No ramo master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (utilize "git add <arquivo>..." para atualizar o que será submetido)
  (utilize "git checkout -- <arquivo>..." para descartar mudanças no diretório\
de trabalho)

        modified:   .gitignore

Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)

        bla.txt
        ble.txt
        bli.txt
        teste/
```

Rodar o `trash`:

```
$ git trash

$ git status
No ramo master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

E conferir o conteúdo do diretório `.trash`:

```
$ ls .trash
total 28K
drwxrwxr-x 2 taq taq 4,0K .
drwxrwxr-x 6 taq taq 4,0K ..
-rw-rw-r-- 1 taq taq    4 bla.txt
-rw-rw-r-- 1 taq taq    4 ble.txt
-rw-rw-r-- 1 taq taq    4 bli.txt
-rw-rw-r-- 1 taq taq    4 blo.txt
-rw-rw-r-- 1 taq taq    4 blu.txt
```

Mais algumas operações em branches

Fundindo branches

Vamos aproveitar que estamos na *branch* `master` e presumir que todo o trabalho que fizemos na `work` esteja maduro e pronto para produção. Precisamos fundir o que está na *branch* `work` para a *branch* `master`. Para isso, utilizamos `git merge`:

```
$ ls
total 16K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 16 root root 4,0K ..
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    20 README

$ git merge work
Updating 724dd31..312dbc0
Fast-forward
 README |    1 +
 a.txt  |    1 +
 b.txt  |    1 +
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 a.txt
 create mode 100644 b.txt
```

Agora as duas *branches* vão estar iguais, como mostra o `git diff`:

```
$ git diff work
```

Podemos verificar quais as outras *branches* que estão fundidas com a *branch* atual utilizando:

```
$ git branch --merged
* master
  work
```

Nesse caso, demonstrando que a *branch* `work` foi fundida. Podemos até apagar essas *branches*, já que já foram fundidas:

```
$ git branch --merged | grep -ve ^* | xargs git branch -d
Deleted branch work (was a226419).
```

Cancelando a fusão

Se por acaso fizemos algum deslize e não era para as *branches* serem fundidas, podemos cancelar o `merge`. Vamos voltar para a *branch* `work`, inserir um arquivo que não devia estar lá e fazer o `merge`:

```
$ git checkout -b work
Switched to a new branch 'work'

$ echo "caca" > caca.txt

$ git add caca.txt

$ git commit -am "Feita uma caca"
[work a26bb18] Feita uma caca
 1 file changed, 1 insertion(+)
 create mode 100644 caca.txt
```

Agora vamos fazer um merge com a *branch* `master`:

```
$ git checkout master
Switched to branch 'master'

$ git merge work
Updating 4f65a57..a26bb18
Fast-forward
 caca.txt |    1 +
 1 file changed, 1 insertion(+)
 create mode 100644 caca.txt
```

Mas fizemos caca! Não era para ter feito o `merge`, então vamos desfazê-lo:

```
$ git reset --hard HEAD^
HEAD is now at 312dbc0 Iniciados os arquivos do alfabeto
```

Pronto, desfeito. Também dá para utilizar o `SHA-1` de um determinado `commit` ali no lugar de `HEAD^`.

Receitas de bolo

Durante o livro, vamos encontrar algumas "receitas de bolo" tipo esse `reset` com `HEAD` aí acima. Algumas coisas vão ser explicadas mais adiante, mas convém ir anotando umas "receitas" dessas para efeito prático, mesmo sem entender direito - por enquanto - o que fazem.

Apagando branches

Como já fundimos nosso código da *branch* `work` na `master`, e supostamente não vamos precisar mais dela (chega por hoje, workaholic!), podemos apagá-la utilizando `git branch -d <branch>`. Vamos ver a lista de branches antes e depois de executar o comando:

```
$ git branch
* master
  work

$ git branch -d work
error: The branch 'work' is not fully merged.
If you are sure you want to delete it, run 'git branch -D work'.
```

Só um pequeno detalhe: o arquivo `caca.txt` está na `work` ainda, junto com o seu *commit*. Como o Git é bonzinho, ele nos avisa que a *brach* `work` ainda não foi feito `merge` com nenhuma outra (tinha, mas desfizemos) e já nos dá a dica de que se realmente queremos apagar a *branch* nesse estado, ao invés de `-d` devemos utilizar `-D`:

```
$ git branch -D work
Deleted branch work (was dc90ee9).

$ git branch
* master
```

Agora podemos verificar o status atual e enviar o código do *working directory* para o repositório remoto:

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
nothing to commit (working directory clean)

$ git push
Counting objects: 9, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (7/7), 619 bytes, done.
Total 7 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
To /home/taq/git/repobare
   724dd31..312dbc0  master -> master
```

Quando precisamos mais do que o trivial

Se você quer fazer uso do Git de uma forma bem básica, quando estiver desenvolvendo sozinho e não ter mais colaboradores no seu projeto e como forma de exercitar o conhecimento do VCS, os capítulos anteriores deve dar uma boa base.

Mas quando começamos a mexer um pouquinho mais, começamos a trabalhar em um time, gostaríamos de organizar melhor o histórico, sentimos um gostinho de "quero-mais" ou se alguma coisa der errado (ei, ninguém é infalível, não é mesmo?), vamos precisar saber um pouquinho a mais …

Lidando com conflitos

Os nossos últimos *merges* foram tranquilos: o conteúdo de uma *branch* foi fundido numa boa com o de outra, com o Git lidando com todas as operações necessárias. Vamos inserir um conflito em um arquivo, indicando que o conteúdo de uma *branch* não pode ser fundido de forma automática com o de outra. Para isso, vamos criar a *branch* `work` novamente:

```
$ git checkout -b work
Switched to a new branch 'work'
```

E agora vamos inserir, logo após a primeira linha, um "separador de título", composto de 3 hífens, como no exemplo, já fazendo o `commit` para registrar:

```
$ cat README
Projeto do livro de Git
---
Primeira alteração correta do projeto

$ git commit -am "Separador de titulo"
[work 6bab654] Separador de titulo
 1 file changed, 1 insertion(+)
```

Agora vamos voltar para a *branch* `master`, e criar uma nova *branch* chamada `test` a partir do conteúdo lá (ei, lembrem-se: o Git dá tanta eficiência para criar e destruir *branches* que não é recomendável fazer alterações direto na master) e inserir o separador agora não com hífens, mas com asteriscos:

```
$ git checkout master
Switched to branch 'master'

$ git checkout -b test
Switched to a new branch 'test'

$ cat README
Projeto do livro de Git
***
Primeira alteração correta do projeto
$ git commit -am "Separador de titulo"
[test cdfccc9] Separador de titulo
 1 file changed, 1 insertion(+)
```

Ou seja, agora temos o separador com hífens na *branch* `work` e com asteriscos na *branch* `test`. Vamos imaginar que são *branches* distintas para cada desenvolvedor, qual está errado? O que usou hífens ou asteriscos?

Nessa hora, o Git não tem nada de inteligência artificial para analisar o que está sendo feito (se tivesse, ainda ia ter que ter o "gosto" próprio para escolher entre um ou outro, isso porque são poucos caracteres, imaginem código!), então ele delega a responsabilidade de escolher entre as opções para o usuário. Vamos ir para a *branch* `work` e tentar fazer um *merge* da *branch* `test` para comprovar isso:

```
$ git checkout work
Switched to branch 'work'

$ git merge test
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.

work|MERGING $
```

O Git nos avisa que houve um conflito, nesse caso, no arquivo `README`, e já troca o prompt (se configurado, como anteriormente demonstrado) para indicar que ainda está em processo de fusão, com `MERGING` no final. Vamos dar uma olhada como ficou o arquivo com conflito:

```
$ cat README
Projeto do livro de Git
<<<<<<< HEAD
---
=======
***
>>>>>>> test
Primeira alteração correta do projeto
```

O conteúdo entre `<<<<<<< HEAD` e `=======` é o conteúdo (`---`) da *branch corrente*, `work`, e entre `=======` e `>>>>>>> test` é o conteúdo (`***`) da *branch sendo fundida*, `test`, como demonstrado no final da linha.

Aqui temos que escolher o conteúdo correto (vamos optar por `---`), removendo manualmente as linhas de marcação e o conteúdo que vai ser descartado da outra *branch*. Nesse caso, o arquivo vai ficar assim:

```
$ cat README
Projeto do livro de Git
---
Primeira alteração correta do projeto
```

Agora precisamos enviar o arquivo para a *staging area*, fazer um *commit* indicando que terminamos de resolver o conflito, fundir o código na *branch* `master` e apagar as *branches* `work` e `test`, já que por enquanto não precisamos mais delas:

```
$ git add README

$ git commit -am "Resolvido o conflito do separador"
[work 44f1423] Resolvido o conflito do separador

$ git checkout master
Switched to branch 'master'

$ git merge work
Updating 0c602f8..44f1423
Fast-forward
 README |    1 +
 1 file changed, 1 insertion(+)

$ git branch -d work
Deleted branch work (was 44f1423).

$ git branch -d test
Deleted branch test (was cdfccc9).
```

Reparem que logo após o `commit` o prompt já foi alterado, removendo o `MERGING` que estava presente.

Utilizando stash

No exemplo do chefe desesperado do capítulo anterior, estavámos no melhor dos mundos onde, quando o chefe chamou, trabalhavámos na *branch* `work` que estava em um estado *clean* quando o chefe chamou, ou seja, sem arquivos para adicionar ou enviar para a *staging area*, sem necessidade de fazer algum `commit` ... mas e se não estivéssemos nesse estado? E se precisarmos alternar de *branch* sem a *branch* corrente ainda não esteja em um estado *clean*?

Vamos dar uma olhada, criando novamente a *branch* `work` (tá, eu sei que acabamos de apaga-lá ali em cima, mas serve para ir treinando):

```
$ git checkout -b work
Switched to a new branch 'work'

$ echo "ccc" > c.txt

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

É nesse momento que o chefe chama. E não terminamos de trabalhar com o `c.txt`, ele não está em um estado bom para que possamos fazer um `commit`, e aí? Vamos trocar para a *branch* `master`, que é de onde iremos criar o conteúdo da nova *branch* para trabalhar no recurso que o desesperado quer, mas esperem aí:

```
$ git checkout master
Switched to branch 'master'

$ ls
total 28K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 14 root root 4,0K ..
-rw-rw-r--  1 taq  taq     4 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
-rw-rw-r--  1 taq  taq     4 c.txt
```

```
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    60 README
```

Aquele `c.txt` não era para estar ali! Ele foi criado na *branch* `work`, mas como não foi feito o `commit` lá, ele ficou em um estado meio, digamos, "fantasma intermediário", bem no estilo do Peter Bishop vagando pelos universos paralelos. Mas ainda não estamos prontos para utilizar o `commit` na branch work, o que fazer? Utilizamos o comando `stash`, que vai criar uma pilha de alterações de onde podemos recuperar determinada alteração ou aplicar e remover a mais recente (procedimento que eu acho mais produtivo):

```
$ git checkout work
Switched to branch 'work'

$ git add c.txt

$ git stash
Saved working directory and index state WIP on work: 44f1423
Resolvido o conflito do separador
HEAD is now at 44f1423 Resolvido o conflito do separador
$ git stash list
stash@{0}: WIP on work: 44f1423 Resolvido o conflito do separador

$ git checkout master
Switched to branch 'master'

$ ls
total 24K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 14 root root 4,0K ..
-rw-rw-r--  1 taq  taq     4 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    60 README
```

Podemos ver que dessa vez, o arquivo `c.txt` não foi mostrado na *branch* `master`! Podemos então criar uma nova *branch*, por exemplo, chamada `urgent`, fazer a alteração do chefe desesperado, que nada mais é do que inserir mais duas linhas no arquivo `a.txt`, conferir se está tudo ok e fazer um `merge` na *branch* `master`:

```
$ git checkout -b urgent
Switched to a new branch 'urgent'

$ echo "aaa" >> a.txt

$ echo "aaa" >> a.txt

$ cat a.txt
aaa
aaa
aaa

$ git commit -am "Corrigido o arquivo de A's"
[master 5ce7b06] Corrigido o arquivo de A's
 1 file changed, 2 insertions(+)

$ git checkout master
Switched to branch 'master'

$ git merge urgent
Updating 29a64a0..fe9636f
Fast-forward
 a.txt |    2 ++
 1 file changed, 2 insertions(+)

$ git branch -d urgent
Deleted branch urgent (was fe9636f).
```

Agora que acabamos de fazer a alteração desesperada, podemos retornar para `work` e continuar a trabalhar no arquivo `c.txt`. **Um ponto muito importante aqui é que não queremos perder as alterações que acabamos de incorporar na *branch* `master`**. Por isso que logo após retornarmos para a *branch* `work`, vamos fazer um *merge* da `master`:

```
$ git checkout work
Switched to branch 'work'

$ git merge master
Updating 44f1423..fe9636f
Fast-forward
 a.txt |    2 ++
 1 file changed, 2 insertions(+)

```

```
$ cat a.txt
aaa
aaa
aaa
```

Como vimos, o conteúdo do arquivo `a.txt` está correto. Mas o `c.txt` ainda não existe:

```
$ ls
total 24K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 14 root root 4,0K ..
-rw-rw-r--  1 taq  taq    12 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    60 README
```

Nessa hora que vamos utilizar o comando `stash` juntamente com `pop`, que diz "ei, me devolva o que eu pedi e você guardou por último". Antes vamos dar uma olhada com `stash list` para listar o que tem na pilha do `stash`:

```
$ git stash list
stash@{0}: WIP on work: c219d7d Resolvido o conflito

$ git stash pop
# On branch work
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#       new file:   c.txt
Dropped refs/stash@{0} (34de58ca225d27fab67faf1ed5d57e18219d8bca)

$ ls
total 28K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 14 root root 4,0K ..
-rw-rw-r--  1 taq  taq    12 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
-rw-rw-r--  1 taq  taq     4 c.txt
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    60 README

$ git stash list
```

E olha lá o `c.txt` novamente, igualzinho de como estava quando utilizamos o `stash`! Vamos fazer um `commit`, trocar para a *branch* `master`, fazer `merge` e apagar a `work`:

```
$ git commit -am "Adicionado arquivos de C's"
[work bcdd7c2] Adicionado arquivos de C's
 1 file changed, 1 insertion(+)
 create mode 100644 c.txt

$ git checkout master
Switched to branch 'master'

$ git merge work
Updating fe9636f..bcdd7c2
Fast-forward
 c.txt |    1 +
 1 file changed, 1 insertion(+)
 create mode 100644 c.txt

$ git branch -d work
Deleted branch work (was bcdd7c2).
```

Algumas opções que podem ser utilizadas com `stash`:

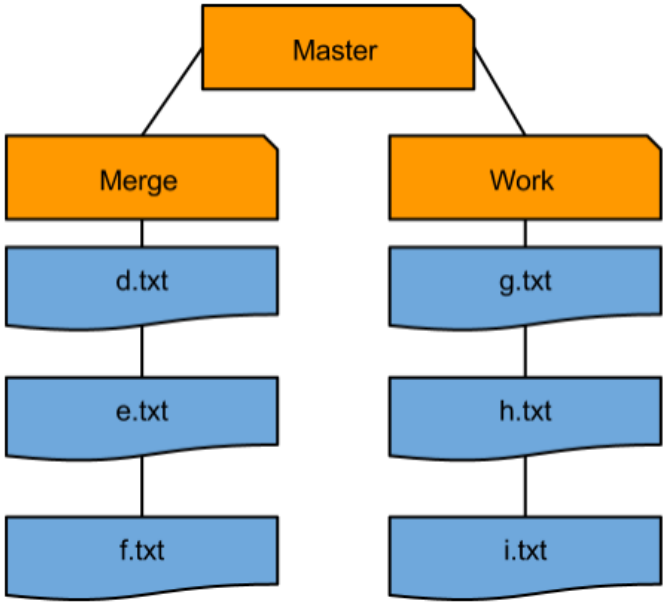
```
git stash
git stash list
git stash apply <stash>
git stash drop <stash>
```

Dá para pedir para fazer `stash` em um `rebase`, ou seja, as alterações atuais vão para a área de `stash` enquanto o `rebase` é aplicado, e retornadas se tudo correr bem. Se prepare para possíveis conflitos quando o `stash` for recuperado.

Utilizando rebase

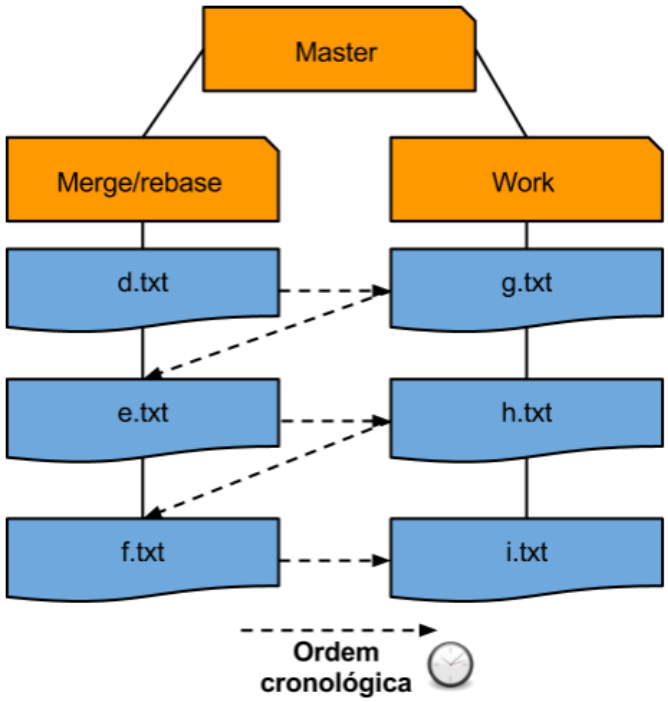
Alguém já deve ter visto por aí uma camiseta escrito "*All your rebase belongs to us*". Ela se refere ao comando `rebase`, que é meio que um primo de primeiro grau do `merge`. Vamos tentar explicar qual seria a diferença entre um e outro.

Para isso vamos criar duas *branches* novas, `merge` e `work`, onde vamos criar alguns arquivos:



Arquivos nas branches merge e work

Ou seja, a *branch* `merge` vai conter os arquivos `d.txt`, `e.txt` e `f.txt`, e a *branch* `work` vai conter os arquivos `g.txt`, `h.txt` e `i.txt`. O detalhe é como vamos criar cada arquivo em cada branch:



Ordem de criação de arquivos nas branches

A *branch* da esquerda vamos usar para testar tanto o `merge` como o `rebase`. Vamos ter que fazer um pouco de exercício agora, pois vamos ficar trocando de uma *branch* para outra para ir criando os arquivos, na seguinte sequência:

1. Ir para **merge** e criar o `d.txt`
2. Ir para **work** e criar o `g.txt`
3. Ir para **merge** e criar o `e.txt`
4. Ir para **work** e criar o `h.txt`
5. Ir para **merge** e criar o `f.txt`
6. E finalmente ir para **work** e criar o `i.txt`

Vamos tentar fazer isso de uma forma rápida, porém não bonita (tá, mais feio que brigar com a mãe por causa da janta, mas funciona):

```
$ git checkout master
$ git checkout -b merge
$ git checkout -b work

$ git checkout merge
$ echo "ddd" > d.txt && git add d.txt && git commit -am "Adicionados os D's"
[merge 3925151] Adicionados os D's
 create mode 100644 d.txt

$ git checkout work
$ echo "ggg" > g.txt && git add g.txt && git commit -am "Adicionados os G's"
[work cfbcf00] Adicionados os G's
 create mode 100644 g.txt

$ git checkout merge
$ echo "eee" > e.txt && git add e.txt && git commit -am "Adicionados os E's"
[merge d7b6084] Adicionados os E's
 create mode 100644 e.txt

$ git checkout work
$ echo "hhh" > h.txt && git add h.txt && git commit -am "Adicionados os H's"
[work 40669b3] Adicionados os H's
 create mode 100644 h.txt

$ git checkout merge
$ echo "fff" > f.txt && git add f.txt && git commit -am "Adicionados os F's"
[merge 1cd0b04] Adicionados os F's
 create mode 100644 f.txt

$ git checkout work
$ echo "iii" > i.txt && git add i.txt && git commit -am "Adicionados os I's"
[work 3059294] Adicionados os I's
 create mode 100644 i.txt
```

Agora que já devemos ter os arquivos corretos em cada *branch*, vamos criar uma outra *branch* chamada **`rebase`** a partir do conteúdo da *branch* `merge`, retornando à `merge` logo depois:

```
$ git checkout merge
$ git checkout -b rebase
$ git checkout merge
```

Ok, parece meio complicado mas não é: o que precisamos é de uma nova *branch* a partir do

conteúdo da `merge` para podemos comparar as diferenças entre `merge` e `rebase`. Dando uma olhada no conteúdo de `merge` (e, consequentemente, de `rebase`, que acabamos de criar):

```
$ ls
total 40K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 14 root root 4,0K ..
-rw-rw-r--  1 taq  taq    12 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
-rw-rw-r--  1 taq  taq     4 c.txt
-rw-rw-r--  1 taq  taq     4 d.txt
-rw-rw-r--  1 taq  taq     4 e.txt
-rw-rw-r--  1 taq  taq     4 f.txt
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq    60 README
```

Os arquivos `d.txt`, `e.txt` e `f.txt` estão corretamente no local.

Agora vamos fazer um `merge` da *branch* `work`, que contém o restante dos arquivos, na nossa *branch corrente*, `merge`:

```
$ git merge work
Merge made by the 'recursive' strategy.
 g.txt     |    1 +
 h.txt     |    1 +
 i.txt     |    1 +
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 g.txt
 create mode 100644 h.txt
 create mode 100644 i.txt
```

Dando uma olhada em como o conteúdo do diretório corrente ficou:

```
$ ls
total 52K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 14 root root 4,0K ..
-rw-rw-r--  1 taq  taq    12 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
-rw-rw-r--  1 taq  taq     4 c.txt
-rw-rw-r--  1 taq  taq     4 d.txt
-rw-rw-r--  1 taq  taq     4 e.txt
```

```
-rw-rw-r--  1 taq  taq     4 f.txt
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq     4 g.txt
-rw-rw-r--  1 taq  taq     4 h.txt
-rw-rw-r--  1 taq  taq     4 i.txt
-rw-rw-r--  1 taq  taq    60 README
```

E agora no log:

```
$ git log
commit e8946058326f2417fa7757723b30858dde468739
    Merge branch 'work' into merge
commit 3059294398e71aab04507bdea02336c1bc5855f9
    Adicionados os I's
commit 1cd0b049ed9ec35d477367f8f4015018758d8390
    Adicionados os F's
commit 40669b37c9f70192c77afad5d6073fe9a076cb81
    Adicionados os H's
commit d7b60841a2c5e64f7b66d985d147cf20978d738e
    Adicionados os E's
commit cfbcf004abadc14f1a0d1683b664fec12593cbb9
    Adicionados os G's
commit 392515141bb2c3dfe8e02d9890cc05b8c2625c6e
    Adicionados os D's
```

Pelo log já vimos que o **`merge`** obedeceu a ordem cronológica de criação dos arquivos, ou seja, as alterações foram fundidas nas *branches* nessa exata ordem, conforme visto na última ilustração apresentada.

Agora vamos trocar para a nossa *branch* `rebase`, anteriormente criada com o mesmo conteúdo de `merge`, e utilizar `rebase` ao invés de `merge` para fazer a fusão:

```
$ git checkout rebase
Switched to branch 'rebase'

$ git rebase work
First, rewinding head to replay your work on top of it...
Applying: Adicionados os D's
Applying: Adicionados os E's
Applying: Adicionados os F's
```

E agora vamos dar uma outra olhada no conteúdo do diretório corrente:

```
$ ls
total 52K
-rw-rw-r--  1 taq  taq    12 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
-rw-rw-r--  1 taq  taq     4 c.txt
-rw-rw-r--  1 taq  taq     4 d.txt
-rw-rw-r--  1 taq  taq     4 e.txt
-rw-rw-r--  1 taq  taq     4 f.txt
drwxrwxr-x  8 taq  taq  4,0K .git
-rw-rw-r--  1 taq  taq     4 g.txt
-rw-rw-r--  1 taq  taq     4 h.txt
-rw-rw-r--  1 taq  taq     4 i.txt
-rw-rw-r--  1 taq  taq    60 README
```

E agora no log:

```
commit 79d11e5e9e78f847461ad1aab413b71460406661
    Adicionados os F's
commit 03379bfd4b3d6f7e8365ab4deefa930eb9a1b214
    Adicionados os E's
commit 4e5794f17052c4dfba6a08dc261ca0c0dfcb8e98
    Adicionados os D's
commit 3059294398e71aab04507bdea02336c1bc5855f9
    Adicionados os I's
commit 40669b37c9f70192c77afad5d6073fe9a076cb81
    Adicionados os H's
commit cfbcf004abadc14f1a0d1683b664fec12593cbb9
    Adicionados os G's
```

Agora a fusão das duas *branches* **não obedeceu a ordem cronológica**, sendo feita uma **reversão** do estado corrente do *working directory*, aplicado o conteúdo da *branch* `work` e depois aplicadas as alterações revertidas em cima de tudo isso. A grosso modo, um `rebase` escreve o histório aplicando o conteúdo de outra *branch* todo junto, e não intercalando como o `merge`.

Vamos fazer um `merge` da *branch* `merge` na `master`, e como não vamos precisar mais das *branches* `work`, `merge` e `rebase`, vamos apagá-las:

```
$ git checkout master
$ git merge merge
$ git branch -d merge
$ git branch -D rebase
$ git branch -D work
```

Lembrem-se, pode ser que o Git reclame se você tentar excluir uma *branch* que teve alterações mas que não foi feito nada com ela. Nesse caso, ele mesmo nos dá a dica:

```
error: The branch 'rebase' is not fully merged.
If you are sure you want to delete it, run 'git branch -D rebase'.
```

É só utilizar o `-D` (d maiúsculo), como utilizado acima e já fizemos anteriormente.

Organizando a casa

Arrumando os commits

Voltemos para a *branch* `work` (é, vamos ter que criá-la novamente, ninguém não vai poder dizer que não aprendeu a fazer isso) para adicionar mais um arquivo e fazer uma pequena barbeiragem intencional na hora de fazer o `commit`:

```
$ git checkout -b work
$ echo "jjj" > j.txt && cat j.txt

$ git add j.txt

$ git commit -am "Adicionados os"
[work 9ba8ded] Adicionados os
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 j.txt

$ git log
commit 9ba8ded4562ead04a6d8d64a2016986e4495ab4f
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Wed Oct 26 15:24:59 2016 -0200
    Adicionados os
```

Esse `commit` ficou feio. "*Adicionados os*" ... "*os*" o que? Sorte que temos a opção `--amend`, que permite alterar a mensagem do último `commit`. Após digitarmos o comando, o editor de texto será aberto com a mensagem corrente, permitindo que a editemos. No caso, vamos alterar para a mensagem correta, "*Adicionados os J's*", e salvar o arquivo (se não salvar, a mensagem não será alterada):

```
$ git commit --amend
[work 84d2b48] Adicionados os J's
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 j.txt

$ git log
commit 84d2b487dc143b49755aa5f095b83ef02373f28e
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Wed Oct 26 15:24:59 2016 -0200
    Adicionados os J's
```

Vamos fazer um `merge` na *branch* `master` e descartar a *branch* `work`:

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

Reorganizando os commits

Eu li (não sei onde, até procurei por aqui mas não encontrei) uma analogia muito boa sobre o *workflow* de um dia costumeiro usando um VCS: temos muitos `commits` (você está fazendo `commits` com frequência, não está?), tantos que talvez sejam demais. Imaginem escrevendo parágrafos para um rascunho de livro: talvez sejam muitos parágrafos onde alguns ficassem melhor se estivessem juntos na edição final.

Vamos imaginar que faz mais sentido ter um `commit` só para os arquivos adicionados do `G` ao `J`. Para fazer isso, vamos criar uma nova *branch* baseada na `master` e utilizar `rebase` **interativamente** da seguinte forma:

```
$ git checkout master
$ git checkout -b rebase
$ git rebase -i HEAD~4
```

Isso significa que queremos editar os `commits` feitos do estado corrente (`HEAD`) até `4` `commits` atrás. O editor de texto será aberto e veremos um conteúdo como esse:

```
pick 0229f05 Adicionados os G's
pick 8b04f2b Adicionados os H's
pick 33ba863 Adicionados os I's
pick 2469cbb Adicionados os J's
```

Seguido dos seguintes comandos, que podem substituir o comando `pick` no início da linha:

```
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
```

Vamos usar o comando `squash`, que vai fundir o `commit` da linha que indicarmos com o anterior (pode ser utilizado somente `s` ao invés de `squash`):

```
pick 0229f05 Adicionados os G's
squash 8b04f2b Adicionados os H's
squash 33ba863 Adicionados os I's
squash 2469cbb Adicionados os J's
```

Alterando a mensagem de `commit` para:

```
Adicionados dos G's aos J's
```

Após salvar o arquivo, os `commits` relacionados serão fundidos e o resultado do log será:

```
$ git log
commit 4087a96c1029ae60164b474e6ba8f8b5db2dfa57
    Adicionados dos G's aos J's
 g.txt    |    1 +
 h.txt    |    1 +
 i.txt    |    1 +
 j.txt    |    1 +
 4 files changed, 4 insertions(+), 0 deletions(-)
```

No `rebase` interativo utilizamos `~4` indicando que queríamos os `4 commits` anteriores à `HEAD`. Podemos utilizar as seguintes opções:

```
HEAD~2     # move dois atrás
HEAD^      # move um atrás
HEAD^^     # move dois atrás
```

Para poucos `commits`, podemos utilizar `^`, depois, `~`. Não vamos fazer `merge` dessa *branch*, utilizamos somente para teste, então podemos apagar agora:

```
$ git checkout master
$ git branch -D rebase
```

Inclusive, **cuidado com esse tipo de reescrita na branch `master`!**. Na `master` convenciona-se que já vai ser assimilado o estado de "produção", e tanto trabalhando sozinho como com outras pessoas, fazer `rebase` desse modo pode levar à algumas situações diferenciadas na hora de fazer o `push`.

Trabalhando com intervalos

Aproveitando a explicação acima, fica a dica que podemos trabalhar com intervalos em vários comandos. Por exemplo, para utilizarmos o `diff` entre o `commit` anterior ao estado atual e 3 `commits` atrás, poderíamos utilizar:

```
$ git diff HEAD^ HEAD~3
```

Notas

Usando notas

Apesar de termos as mensagens dos `commits`, também temos um recurso extra para documentação, o uso de *notas*. Vamos colocar a mensagem "*Esse arquivo de J's é legal.*" (ok, eu avisei para não inserir esse tipo de coisa retardada no repositório, mas aqui é para efeito didático e pode - tirei o meu da reta) no `commit` mais recente, novamente criando a *branch* `work` e utilizando o comando `notes add`, digitando uma mensagem ("Esse arquivo de J's é legal") e verificando logo após com `log`:

```
$ git checkout -b work

$ git notes add

$ git log
commit 2469cbb4924575a33a5b68069e7cebe71908faed
    Adicionados os J's
Notes:
    Esse arquivo de J's é legal.
```

Podemos também, através do `SHA-1`, especificar em qual `commit` a nota vai ser adicionada, por exemplo, no `commit` relativo ao arquivo `i.txt`, inserindo a mensagem criativa "*Ei, o arquivo de I's também é legal*" quando o editor for aberto:

```
$ git log
commit 2469cbb4924575a33a5b68069e7cebe71908faed
    Adicionados os J's
Notes:
    Esse arquivo de J's e legal.

commit 33ba8637a19839223d51292086ff322987650740
    Adicionados os I's

$ git notes add 33ba863

$ git log
commit 2469cbb4924575a33a5b68069e7cebe71908faed
```

```
    Adicionados os J's
Notes:
    Esse arquivo de J's e legal.
commit 33ba8637a19839223d51292086ff322987650740
    Adicionados os I's
Notes:
    Ei, o arquivo de I's também e legal!

$ git checkout master

$ git merge work

$ git branch -d work
```

Um detalhe bem importante, que de maneira similar às `tags` (que vamos ver logo abaixo), quando é utilizando o comando `push` as notas não são enviadas para o repositório. Para enviá-las, vamos precisar do seguinte comando:

```
$ git push origin refs/notes/*

Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 588 bytes | 0 bytes/s, done.
Total 6 (delta 1), reused 0 (delta 0)
To /home/taq/git/repobare/
 * [new branch]      refs/notes/commits -> refs/notes/commits
```

Tags

O conceito de *tags* nos permite marcar determinado ponto do nosso repositório (um `commit`) com algum identificador que reflita o que ocorreu naquele ponto, por exemplo, o lançamento de uma nova versão, como por exemplo `v1`, `v2` etc.

Tags "leves"

Podemos marcar uma *tag* leve, que é o conceito mais simples de *tags*, indicando o valor da *tag* e o `SHA-1` de algum `commit` anterior (verifique qual usando `log`):

```
$ git checkout -b work
$ git tag 1.0.0 e89460
```

Funcionou legal, mas tem uma convenção que recomenda que comecemos o valor da *tag* com uma letra (mas usa-se do jeito que quiser). E agora, que já inserimos uma *tag* naquele commit? É só apagar a *tag*:

```
$ git tag -d 1.0.0
Deleted tag '1.0.0' (was e894605)
```

E agora inserir a *tag* corretamente, já verificando todas as *tags* que estão disponíveis com o próprio comando `tag`:

```
$ git tag v1.0.0 e89460
$ git tag
v1.0.0
```

Enviando as tags para o repositório remoto

Dica **muito importante**: de maneira similar as notas, no momento em que executamos `push` para enviar o conteúdo do repositório local para o repositório remoto, as *tags* **não são enviadas**. Para enviar as *tags*, precisamos de `push --tags`:

```
$ git push --tags
Total 0 (delta 0), reused 0 (delta 0)
To /home/taq/git/repobare
 * [new tag]         v1.0.0 -> v1.0.0
```

Apagando as tags do repositório remoto

Podemos também apagar as *tags* no repositório remoto. Para isso, primeiro apagamos do nosso repositório local e depois no remoto. Vamos testar com uma *tag* chamada `teste`:

```
$ git tag teste

$ git push --tags
Total 0 (delta 0), reused 0 (delta 0)
To /home/taq/~git/repobare
 * [new tag]         teste -> teste

$ git push origin :refs/tags/teste
To /home/taq/git/repobare
 - [deleted]         teste
```

Tags "pesadas"

As *tags* pesadas (heavy `\m/`) fazem a mesma coisa que as *tags* leves, porém abrem o editor de texto do mesmo modo que quando fazemos um `commit`, utilizada ao invés da mensagem do `commit` para onde apontamos a *tag*. Para criar uma *tag* pesada, utilizamos a opção `-a`. Vamos criar uma mensagem "*Primeira tag pesada*":

```
$ git tag v1.0.1 -a
```

Podemos listar as *tags* e suas mensagens, utilizando `-n<número>`, onde `<número>` é o número de linhas que queremos ver na lista:

```
$ git tag
v1.0.0
v1.0.1

$ git tag -n2
v1.0.0          Merge branch 'work' into merge
v1.0.1          Primeira tag "pesada".
```

Tags assinadas

Uma coisa muito importante hoje em dia e que infelizmente é de pouco uso (e até de desconhecimento!) de muitos desenvolvedores é a assinatura digital. Ela permite que assinemos um ou mais documentos de forma que outras pessoas possam verificar a validade da assinatura e assim aumentar a sua confiabilidade no conteúdo disponibilizado, dando força à sua autenticidade. Hoje até utilizamos as nossas chaves públicas para acessar alguns serviços como o Github, por exemplo, mas esquecemos ou desconhecemos que podemos utilizar a nossa chave privada para encriptar ou assinar mensagens e documentos.

Andrew Tanenbaum (vocês sabem que é o cara né?) fez as seguintes premissas para uma assinatura digital:

- O destinatário deve poder verificar a identidade alegada do emissor, indicando a sua **autenticidade**.
- O emissor não pode repudiar mais tarde o conteúdo da mensagem, indicando a sua **integridade**.
- O destinatário não pode ter forjado a mensagem para si mesmo, indicando a sua **irretratabilidade**.

Ou seja: se assinamos documentos com a nossa assinatura digital, quem receber deve ter condições de poder verificar que fomos nós mesmos quem assinamos o conteúdo (que deve "bater" com a assinatura), não podemos, após assinado, dizer que não fomos nós que assinamos a mensagem (ah, isso traria tão menos problemas ...) a não ser que os "espertos" perdessem o arquivo com a chave privada e entregassem a senha de bandeija para alguém, e a pessoa que recebeu a mensagem não teria condições de forjar a mensagem, alegando que fomos nós que a enviamos.

O que talvez muita gente não saiba é que não precisamos pagar por isso (quem tem empresa aposto que adorou ter que pagar uma graninha pelo seu certificado digital, não é mesmo?) e podemos utilizar com ótimas ferramentas como o GPG (GNU Privacy Guard). O ensino de como usar o GPG para gerenciamento de chaves públicas e privadas está fora do escopo desse livro, mas existem vários tutoriais bem legais disponíveis na web, procurem por `gpg+chave+privada`. Um bem legal é esse aqui do Ubuntu-BR[10].

[10] http://wiki.ubuntu-br.org/GnuPG

De posse de nossas chaves públicas e privadas, podemos assinar uma *tag*. Primeiro verificamos o `id` da chave que desejamos utilizar, como no exemplo:

```
$ gpg --list-public-keys
...
pub   1024D/7EF350A0 2008-05-09
uid Eustaquio Rangel de Oliveira Jr. <taq@eustaquiorangel.com>
...
```

Precisamos do `id` que está em destaque ali em cima. Aí verificamos no `~/.gitconfig` se a seguinte configuração reflete o `id`:

```
[user]
signingkey = 7EF350A0
```

Tudo pronto? Agora podemos adicionar mais algum conteúdo, fazer um `commit` e assinar a *tag*, indicando como mensagem "*Primeira tag assinada*":

```
$ echo "kkk" > k.txt && git add k.txt && git commit -am "Adicionados os K's"
[master e000fa2] Adicionados os K's
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 k.txt

$ git tag v1.0.2 -s -m "Primeira tag assinada"
Voce precisa de uma frase secreta para destravar a chave secreta do usuario:
"Eustaquio Rangel de Oliveira Jr. <taq@eustaquiorangel.com>"
```

Agora podemos verificar se a tag assinada corresponde com quem assinou, utilizando a opção `-v`:

```
$ git tag -v v1.0.2
object e000fa2e100667e73018a66525f2fe100dd29a46
type commit
tag v1.0.2
tagger Eustaquio Rangel <taq@eustaquiorangel.com> 1319674189 -0200

Primeira tag assinada!

gpg: Assinatura feita Qua 26 Out 2016 22:09:58 BRST usando DSA chave ID 7EF350\
A0
gpg: Assinatura correta de "Eustaquio Rangel de Oliveira Jr. <taq@eustaquioran\
gel.com>"
```

Podemos visualizar a assinatura da tag assinada utilizando `git show`:

```
$ git show v1.0.2
tag v1.0.2
Tagger: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Wed Oct 26 22:09:49 2016 -0200

Primeira tag assinada!
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.11 (GNU/Linux)

iEYEABECAAYFAk6ooVYACgkQ4bJBin70VaolAwCePYJk60fPKaoFu1+1gBpz/b9r
ifYAoKWYqvbWDM7Qiel4o7Ibu5iM/ack
=VmeU
-----END PGP SIGNATURE-----
```

Para verificarmos para qual *commit* a `tag` aponta, podemos utilizar `show-ref --tags`, o que nos dá um resultado como esse:

```
$ git show-ref --tags
f4382dadae0831515f3e6c30de60c8268d927f03 refs/tags/v1.0.0
7e941d3fddd27c3ad606598a065fd243bf6c622e refs/tags/v1.0.1
ed982f03398a6d739e69a7a2d6611ea5a1589fc9 refs/tags/v1.0.2
```

Commits assinados

Em versões anteriores do Git, não podíamos assinar os *commits*. Ou pelo menos, de acordo com o Linus, não devíamos. Segundo Linus disse[11]:

> "Você não pode fazer isso. Bom, você pode, mas sempre vai ser inferior à assinar uma tag. O lance é: o que você quer proteger? A *tree*, a autoria, a informação de quem fez os commit, o que? E realmente não importa. Pois a assinatura deve ser sobre alguma parte do `commit`, e como o `SHA-1` do `commit` por definição contém tudo, então a coisa mais segura é assinar o próprio `SHA-1`: *ergo*, uma *tag*"

Apesar disso, em versões do Git a partir a `1.7.9`, podemos assinar os *commits*. Para isso utilizamos a opção `-S` junto com `commit`. Vamos alterar o `README` e assinar o commit:

[11] http://git.661346.n2.nabble.com/GPG-signing-for-git-commit-tp2582986p2583316.html

```
  $ cat README

  $ Projeto do livro "Conhecendo o Git"
  ---
  Primeira alteração correta do projeto

  $ git commit -am "Corrigido o título do livro" -S

  Você precisa de uma frase secreta para destravar a chave secreta do usuário: "\
Eustaquio Rangel de Oliveira Jr. <taq@eustaquiorangel.com>"
  1024-bit DSA chave, ID 7EF455AA, criada 2008-05-09

  [work 7f917fc] Corrigido o título do livro
  1 file changed, 1 insertion(+), 1 deletion(-)
```

Podemos ver os *commits* assinados com a opção `--show-signatures` no comando `log`:

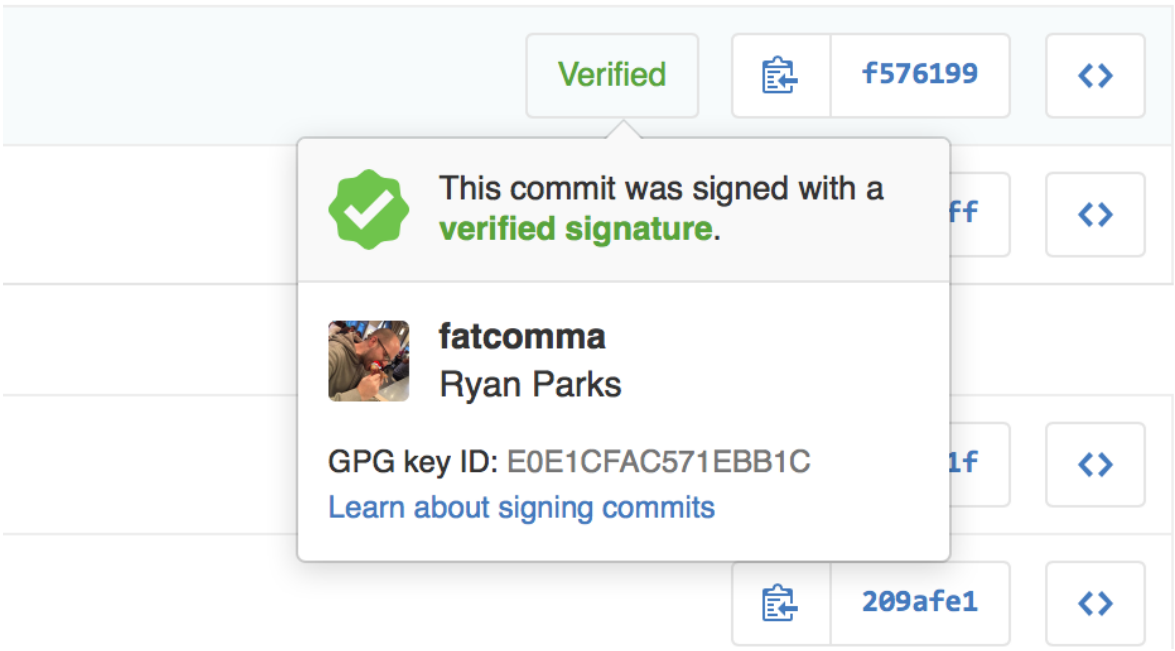
```
  $ git log --show-signature
  commit 7f917fcea5c8adcba6f93f5fa9861c95f9323758
  gpg: Assinatura feita Ter 23 Ago 2016 12:56:34 BRT usando DSA chave ID 7EF455AA
  gpg: Assinatura correta de "Eustaquio Rangel de Oliveira Jr. <taq@eustaquioran\
gel.com>"
  Impressão digital da chave primária: 1CC8 7BD7 B99C 3515 3DCD  35F2 E1B2 418A \
7EF4 55AA
  Author: Eustaquio Rangel <taq@eustaquiorangel.com>
  Date:   Tue Aug 23 12:56:34 2016 -0300

      Corrigido o título do livro
```

Também podemos ver no `log` quais os commits que estão assinados utilizando uma formatação especial com a opção `--pretty` e o formato `%G?`, que vai mostrar `G` como assinado ("*good*") e `N` como não assinado:

```
  git log --pretty="format:%h %G? %aN %s"
  7f917fc G Eustaquio Rangel Corrigido o título do livro
  e7e9c16 N Eustaquio Rangel Adicionados os K's
  6e82c40 N Eustaquio Rangel Adicionados os J's.
```

O Github[12] provê indicações visuais de *commits* assinados, mostrando como na imagem abaixo:

[12] http://github.com



Commits assinados vistos no Github

Se quisermos garantir que os *commits* de uma determinada *branch* estejam todos assinados ao fazer o `merge`, podemos utilizar a opção `--verify-signatures`. Vamos fazer uma pequena alteração no `README` em duas *branches*, uma chamada `nao-assinada`, onde vamos fazer um `commit` normal, e outra `assinada`, onde vamos assinar o *commit*. Primeiro, na `nao-assinada`:

```
$ git checkout work

$ git checkout -b nao-assinada
Switched to a new branch 'nao-assinada'

$ vim README
$ git commit -am "Inserida linha no README"
[nao-assinada e325d32] Inserida linha no README
 1 file changed, 1 insertion(+)

$ git checkout work
Switched to branch 'work'

$ git checkout -b assinada
Switched to a new branch 'assinada'

$ vim README
$ git commit -am "Inserida linha no README" -S
```

```
19
20    Você precisa de uma frase secreta para destravar a chave secreta do usuário: "\
21  Eustaquio Rangel de Oliveira Jr. <taq@eustaquiorangel.com>"
22    1024-bit DSA chave, ID 7EF455AA, criada 2008-05-09
23
24    [assinada 5e9ff2b] Inserida linha no README
25     1 file changed, 1 insertion(+)
```

Agora voltamos para a *branch* `work` e tentamos fazer o `merge` das duas *branches*:

```
1     $ git checkout work
2     Switched to branch 'work'
3
4     $ git merge --verify-signature nao-assinada
5     fatal: Commit e325d32 does not have a GPG signature.
6
7     $ git merge --verify-signature assinada
8     Commit 5e9ff2b has a good GPG signature by Eustaquio Rangel de Oliveira Jr. <t\
9   aq@eustaquiorangel.com>
10    Updating 7f917fc..5e9ff2b
11    Fast-forward
12     README | 1 +
13     1 file changed, 1 insertion(+)
```

Como vimos, no primeiro caso onde o *commit* não foi assinado, o `merge` foi rejeitado. Assinar os *commits* é um meio muito eficiente de garantir a segurança do seu código, mas tenha certeza que as pessoas que você colabora sabem como funciona o esquema de assinatura antes de partir para uma abordagem de `merge` assinado.

Importante notar que também podemos assinar o próprio `merge`, também utilizando a opção `-S` que utilizamos para assinar o *commit*.

Recuperando tags

Após criarmos nossas *tags*, podemos recuperá-las (ou até utilizá-las com um `diff`) utilizando `checkout`:

```
$ git checkout v1.0.0
Note: checking out 'v1.0.0'.
You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches by
performing another checkout.
If you want to create a new branch to retain commits you create,
you may do so (now or later) by using -b with the checkout
command again. Example:

  git checkout -b new_branch_name

HEAD is now at e894605... Merge branch 'work' into merge

$ git log
commit e8946058326f2417fa7757723b30858dde468739
Merge: 1cd0b04 3059294
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Wed Oct 26 14:14:01 2016 -0200

    Merge branch 'work' into merge
```

Agora o estado do nosso *working directory* está de acordo com a *tag* requisitada, mas com um detalhe muito importante: estamos em um *headless state*, um estado sem apontar a `HEAD` para lugar algum.

Como a própria mensagem do Git diz, podemos dar uma olhada nas coisas, fazer algumas alterações e `commits` sem impactar em nenhuma outra *branch*, mas também não podemos fazer `merge` ou `rebase`, por exemplo, do estado atual com alguma outra *branch*, pois não temos nenhuma referência do estado atual, a não ser que criemos, como também instruído na mensagem, uma *branch* nova utilizando `git checkout -b new_branch_name` para acomodar o estado atual, que é o que eu chamo de "*estado mula-sem-cabeça*".

Estado mula-sem-cabeça

Se esquecermos de criar uma *branch* nova com as nossas alterações no estado atual, as quais seriam necessárias para fazer `merge` com alguma outra *branch*, e não prestarmos atenção na mensagem que o Git vai nos dar, indicando o `SHA-1` da *branch*, como no exemplo:

```
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  1cde1bc Teste de burrada

If you want to keep them by creating a new branch, this may be a good time
to do so with:

 git branch new_branch_name 1cde1bcc6be920b471041c1a59eb82de44c5350e
```

Não vai tão fácil retornar (ou mesmo possível) para o local fantasma onde estão as alterações. Aí com certeza vamos nos sentir umas **mulas** e vamos **perder a cabeça** de tanta raiva. Então, prestem atenção e tomem cuidado com esse tipo de coisa. E também com mulheres de padres.

Vamos seguir a recomendação do Git e criar uma *branch* nova para podermos sair do estado mula-sem-cabeça e fazer algumas alterações, inserindo mais 2 linhas no arquivo `i.txt`:

```
$ git checkout -b ghostbusters

$ cat i.txt
iii
iii
iii

$ git commit -am "Mais I's"
[work 6ea3655] Mais I's
 1 files changed, 2 insertions(+), 0 deletions(-)
```

Agora sim, estamos em um estado menos fantasmagórico e podemos até fazer um `merge` da *branch* `work` na *master*:

```
$ git checkout master

$ git merge ghostbusters
Merge made by the 'recursive' strategy.
 i.txt |    2 ++
```

Mais repositórios remotos

Adicionando

Uma das características mais legais do Git é que ele é distribuído, descentralizado. Isso significa que, apesar de geralmente utilizarmos com um repositório central (servidor próprio, Github, Bitbucket), através do `origin`, podemos ver vários outras referências apontando para outros repositórios remotos. Por exemplo, podemos criar uma referência para um colega de trabalho que clonou o mesmo repositório e já fez algumas alterações e `commits`, e comparar nosso conteúdo com o dele.

Vamos testar isso, não esquecendo de dar um `push` na `master` (e se necessário, um `merge` com a `work` antes) para atualizar o repositório remoto:

```
$ cd ~/git
$ git clone ~/git/repobare other
Cloning into other...
done.
```

Ali acima o repositório `bare` foi clonado para o diretório `other`, e agora vamos para lá adicionar algum conteúdo:

```
$ cd other
$ echo "lll" > l.txt && git add l.txt && git commit -am "Adicionados os L's"
[master b9f65c8] Adicionados os L's
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 l.txt

$ echo "111" > 1.txt && git add 1.txt && git commit -am "Adicionados os 1's"
[master a77ef56] Adicionados os 1's
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 1.txt

$ git log
commit a77ef562c75de82d6a24e9901f8384d58296e77d
    Adicionados os 1's
commit b9f65c8c312f5a1a9acaa201558c08104e7d26d5
    Adicionados os L's
```

Ali abrimos outra exceção para a regra de utilizar somente letras, e criamos um arquivo chamado `1.txt`. Podemos ver pelo log acima, que o arquivo foi adicionado juntamente com `1.txt`. Agora vamos voltar no diretório do repositório anterior que estavamos trabalhando até agora há pouco e indicar que temos mais um repositório remoto, chamado `other`:

```
$ cd ~/git/repo
$ pwd
/home/taq/git/repo
$ git remote add other /home/taq/git/other
```

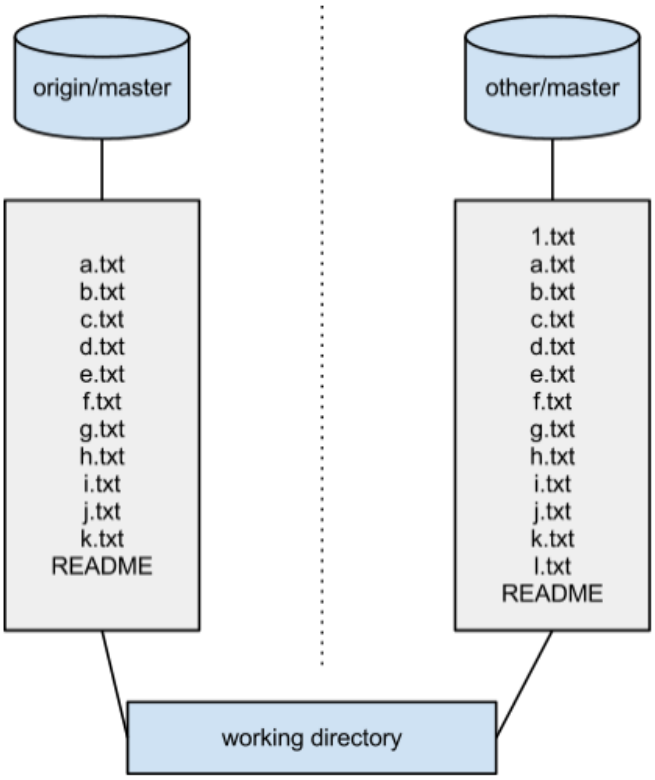
Sincronizando o conteúdo

Agora vamos usar `git fetch` para sincronizar o outro repositório remoto:

```
$ git fetch other
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /home/taq/git/other
 * [new branch]      master     -> other/master
```

Comparando

Temos uma situação como essa agora:



Comparando conteúdo

Isso nos permite alguns recursos como saber, através do nosso *working directory*, quais são as diferenças de qualquer outro repositório remoto, como por exemplo no caso de `other`:

```
$ git diff other/master
diff --git a/1.txt b/1.txt
deleted file mode 100644
index 58c9bdf..0000000
--- a/1.txt
+++ /dev/null
@@ -1 +0,0 @@
-111
diff --git a/l.txt b/l.txt
deleted file mode 100644
index 8a491af..0000000
--- a/l.txt
+++ /dev/null
@@ -1 +0,0 @@
-lll
```

Merging, rebasing ...

Como podemos lidar com qualquer repositório extra da mesma forma que estávamos tratando o `origin` até agora, vamos fazer um `merge` (ou `rebase`, o que for da preferência) com o repositório remoto `other`:

```
$ git checkout -b work
Switched to a new branch 'work'

$ git merge other/master
Updating e000fa2..a77ef56
Fast-forward
 1.txt    |    1 +
 l.txt    |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 1.txt
 create mode 100644 l.txt

$ ls
total 68K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 16 root root 4,0K ..
-rw-rw-r--  1 taq  taq     4 1.txt
-rw-rw-r--  1 taq  taq    12 a.txt
...
-rw-rw-r--  1 taq  taq     4 l.txt
-rw-rw-r--  1 taq  taq    60 README
```

Perfeito! Só que foi uma sujeira em forma de liberdade criativa didática ali: o arquivo `1.txt` não era para ter sido criado.

Pull requests

Uma coisa que está sendo bastante utilizada por times de desenvolvimento hoje em dia é o conceito de *pull requests*, especialmente feitos por ferramentas como o Github. O conceito se baseia no seguinte:

1. Alguém faz um *fork* do seu repositório, ou seja, clona ele para um repositório próprio, e aplica alterações no código desse repositório.
2. A pessoa te envia um link para você clonar o repositório dela, indicando que fez algumas mudanças no código, que gostaria que você incorporasse.

3. Você adiciona o repositório da pessoa como um repositório remoto.
4. Você cria uma *branch* para requisitar o código do repositório remoto.
5. Você pede para atualizar essa nova *branch* com o código do repositório remoto.
6. Você inspeciona as alterações, e se tudo ok, dá um *merge* para a sua *branch* desejada, pegando os *commits* do repositório remoto.

Vamos imaginar que o usuário que está fazendo as alterações criou um repositório em outro local, como por exemplo, `barizon`:

```
$ git clone ~/git/repobare barizon
```

Ele alterou o `README` para isso:

```
Projeto do livro "Conhecendo o Git"
-----------------------------------
Toca Metal aí, pô!
```

Aí podemos adicionar o repositório como remoto, criar a *branch* e examinar as mudanças:

```
$ git remote add barizon ../barizon/

$ git checkout -b barizon
Switched to a new branch 'barizon'

$ git fetch barizon

$ git diff barizon/master
diff --git a/README b/README
index fd83cbd..904cb7e 100644
--- a/README
+++ b/README
@@ -1,3 +1,5 @@
 Projeto do livro "Conhecendo o Git"
  -----------------------------------
  -Toca Metal aí, pô!

$ git merge barizon/master
Updating c5bb064..d058482
Fast-forward
 README | 4 +---
  1 file changed, 1 insertion(+), 3 deletions(-)
```

```
23
24    $ cat README
25    Projeto do livro "Conhecendo o Git"
26    ----------------------------------
27    Toca Metal aí, pô!
```

Nesse ponto, temos o código do outro repositório em uma *branch* local e podemos fazer *merge* ou apenas descartar. Como é muita metaleirice nesse caso, vamos descartar.

No Github existem algumas ferramentas interessantes para esse recurso. Podemos fazer um *fork* do repositório que desejamos contribuir, alterar o código no repositório "forkado" e acionar um botão na interface gráfica para criar o *pull request*, sendo que o dono do repositório original vai ser notificado e o *pull request* vai aparecer na aba correspondente do projeto.

Desfazendo as coisas

No caso acima, o arquivo `1.txt` está em um commit em particular, o que nos ajuda a desfazer uma situação que não queríamos. Para isso vamos usar `git revert`, indicando o `SHA-1` do `commit` que deverá ser revertido e uma mensagem como as de `commit` quando o editor for aberto:

```
$ git log
commit a77ef562c75de82d6a24e9901f8384d58296e77d
    Adicionados os 1's

$ git revert a77ef56
[work 10e6948] Revert "Adicionados os 1's"
 1 files changed, 0 insertions(+), 1 deletions(-)
 delete mode 100644 1.txt
```

Agora vamos fazer o `merge` em outra *branch* baseada na `master`. Lógico que poderíamos fazer direto na `master`, mas não vamos fazer para podermos deixar a `master` testar um outro recurso logo à frente. Lembrem-se: *branches* são recursos baratos e práticos do Git, só tomem cuidado para não criarem muitas e perderem o controle! Organização e bom senso, como sempre, prevalecem.

```
$ git checkout master

$ git checkout -b test

$ git merge work
Updating e000fa2..10e6948
Fast-forward
 l.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 l.txt
```

Como pudemos ver, somente o arquivo `l.txt` é criado. Estão vendo como organizar os `commits` pode ajudar a desfazer alguma possível lambança? Treinem o jeito que organizam o conteúdo de seus `commits` já pensando em uma situação dessas.

Desfazendo com revert

Como vimos anteriormente, podemos utilizar `reset` também para desfazer as alterações, se conseguirmos e pudermos mover o estado do repositório para alguns dos `commits` anteriores. Ou seja, `revert` reverte um `commit`, `reset` reseta o estado atual até um determinado `commit`.

Descobrindo quem fez a arte

Nesse ponto em que estamos precisando desfazer algo, talvez seja interessante saber, além do registro do *commit* mostrando quais arquivos que foram afetados e do *diff* mostrando quais linhas que foram afetadas, também saber quando e quem foi o autor de **cada** linha de determinado arquivo. Para isso, podemos utilizar `git blame`:

```
  git blame README
  ^acfcd41 (Eustaquio Rangel 2016-08-22 19:19:44 -0300 1) Projeto do livro de Git
  919b7b02 (Eustaquio Rangel 2016-08-23 10:03:21 -0300 2) ---
  6332600c (Eustaquio Rangel 2016-08-22 20:06:34 -0300 3) Primeira alteração cor\
reta do projeto
```

Fazendo merge parcial

No `commit` anterior havia uma situação que não era interessante para fazer o `merge` completo da outra *branch*: apenas um dos arquivos estava correto, que é o `1.txt`, estando o `1.txt` errado de acordo com a nossa regra de utilizar somente arquivos com letras. Nesse caso, seria bem melhor que pudéssemos escolher um `commit` separado. Aí que entra o recurso de `cherry pick`.

Cherry pick

Vamos criar novamente a *branch* `work`:

```
  $ git checkout -b work
```

Agora vamos listar o log do outro repositório remoto:

```
  $ git log other/master
  commit a77ef562c75de82d6a24e9901f8384d58296e77d
      Adicionados os 1's
  commit b9f65c8c312f5a1a9acaa201558c08104e7d26d5
      Adicionados os L's
```

Ali o `commit` que nos interessa é o `b9f65c`, que contém o arquivo `1.txt`, e podemos selecionar (mesmo do repositório/*branch* remota, lembrem-se de utilizar o `fetch` antes para deixar tudo correto!) e aplicar o `commit` em nossa *branch* local, utilizando o `cherry pick`:

```
$ git cherry-pick b9f65c
[test 01495a6] Adicionados os L's
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 l.txt

$ ls l.txt
-rw-rw-r-- 1 taq taq 4 l.txt

$ ls 1.txt
ls: impossível acessar 1.txt: Arquivo ou diretorio nao encontrado

$ git checkout master

$ git merge work

$ git branch -d work
```

Pudemos ver que o arquivo `l.txt` foi corretamente criado em nosso *working directory*, porém o `1.txt` não foi criado, o que é o comportamento esperado, afinal, o `commit` em que ele pertence não foi selecionado.

Atualizando o repositório

Durante o dia, podemos estar trabalhando localmente sem precisar ainda enviar nossas alterações com o `push`, porém, em um ambiente com vários repositórios remotos e várias pessoas trabalhando, existem altas chances de que alguém possa ter feito `push`. Nesse caso, o conteúdo do nosso *working directory* está defasado com o que está no repositório remoto, e precisamos de um modo de sincronizá-lo, fazendo `merge` com as alterações feitas por nós no *working directory*.

Simulando uma alteração

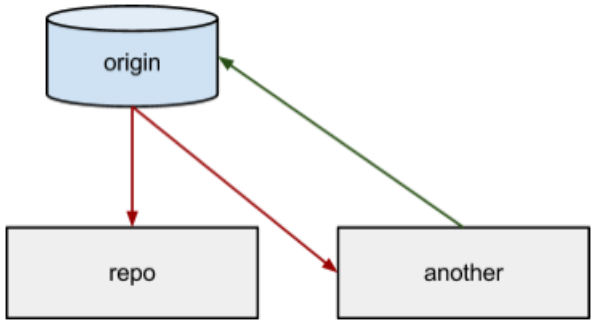
Vamos criar uma nova cópia do repositório remoto:

```
$ git clone /home/taq/git/repobare another
Cloning into 'another'...
done.
$ cd another
```

Agora vamos alterar o arquivo `README` de maneira que fique com o seguinte conteúdo, logo em seguida fazendo um `commit` e um `push`, atualizando o repositório remoto:

```
$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteração correta do projeto

$ git commit -am "README atualizado"
[master 7a2fd06] README atualizado
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 306 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /home/taq/git/repobare
   9d4bf6d..7a2fd06  master -> master
```

Enquanto que em `repo` ainda não foi nada alterado, `another` clonou, fez alterações e enviou novamente para o repositório `origin`, deixando `repo` defasado com o conteúdo mais atual:



Repositórios

Atualizando o working directory com o repositório remoto

Precisamos agora de um jeito de atualizar e sincronizar o nosso *working directory* com o conteúdo mais atualizado de `origin`. Temos duas formas de fazer isso, e fica a critério de cada um cada abordagem utilizar.

Método para hacker mutcho macho

Esse é o jeito mais fácil, e para quem está seguro do que está fazendo. Não executem ele agora, para que possamos testar o outro mais a frente. Para fazer a sincronização, já executando um `merge`, podemos utilizar apenas:

```
1  $ git pull
```

Isso vai fazer tudo o que precisamos, se não houver algum conflito entre o que temos no working directory e o conteúdo do repositório remoto. Continuem lendo se isso deu medo!

Desfazendo um pull com conflitos

Caso o seu `pull` tenha resultado em vários arquivos com conflitos (que podem ser resolvidos) e você desejar voltar à situação antes do `pull`, utilize

```
1  git reset --merge
```

Método para mariquinhas

Fui apelativo no título para chamar a atenção e fixar nesse método, que realmente não é para mariquinhas, mas para quem quer ter uma idéia do que vai acontecer na atualização/sincronização.

A primeira coisa que vamos fazer é executar `fetch` para sincronizar o conteúdo dos repositórios remotos. Vejam bem: `fetch` vai sincronizar e trazer as alterações remotas sem aplicar nada em alguma das *branches* que estamos trabalhando. Então vamos lá:

```
1  $ git fetch
2  remote: Counting objects: 5, done.
3  remote: Compressing objects: 100% (3/3), done.
4  remote: Total 3 (delta 2), reused 0 (delta 0)
5  Unpacking objects: 100% (3/3), done.
6  From /home/taq/git/repobare
7     9d4bf6d..7a2fd06  master     -> origin/master
```

Com tudo devidamente sincronizado, que tal darmos uma olhada no conteúdo atual do repositório remoto, vendo o que mudou?

```
$ git diff origin/master
diff --git a/README b/README
index 11267f4..b61643a 100644
--- a/README
+++ b/README
@@ -1,4 +1,3 @@
 Projeto do livro "Conhecendo o Git"
 ----------------------------------
 Primeira alteração correta do projeto
-Atualizado!
```

Ah-há, agora já temos uma idéia do que foi alterado por lá! Com `log` podemos inclusive ver o log:

```
$ git log origin/master
commit 92bcd72f8c634a897de5fde7b51ed52538a14a91
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Thu Oct 27 14:59:05 2012 -0200

    README atualizado
```

Como forma de ainda testar o que resultará da atualização do repositório remoto sem fazer `merge` direto na master, vamos criar novamente a *branch* `work` e fazer o `merge` do repositório remoto nela:

```
$ git checkout -b work
Switched to a new branch 'work'

$ git merge origin/master
Updating 9d4bf6d..7a2fd06
Fast-forward
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git diff master
diff --git a/README b/README
index b61643a..11267f4 100644
--- a/README
+++ b/README
@@ -1,3 +1,4 @@
 Projeto do livro de Git
 -----------------------
 Primeira alteração correta do projeto
+Atualizado!
```

Agora que vimos que tudo está ok, podemos fazer o `merge` de `work` com `master` e já enviar para o repositório remoto, que vai nos dizer que está tudo atualizado, afinal, não estamos nesse caso enviando nada de novo para ele:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

$ git merge work
Updating 9d4bf6d..7a2fd06
Fast-forward
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git status
# On branch master
nothing to commit (working directory clean)

$ git push
Everything up-to-date

$ git branch -d work
```

Utilizando patches

Pode ser que por alguma razão, alguém que trabalha com você não consiga acesso ao repositório, ou que tenha pego o código "na unha" e precise das suas últimas alterações. Nesse caso, podemos enviar as alterações para essa pessoa no formato de um `patch`.

Gerando um patch

Vamos aproveitar que ainda temos um repositório criado no diretório `other` e comparar com o nosso *working directory* (não esquecendo de fazer um `fetch` antes para sincronizarmos) após fazer uma alteração no arquivo `README`, escrevendo mais uma das mensagens toscas com liberdade criativa didática, "*Linha para mostrar o patch!*". Vamos aproveitar e redirecionar a saída para um arquivo chamado `test.patch` no diretório `/tmp`:

```
$ cat README
Projeto do livro "Conhecendo o Git"
----------------------------------
Primeira alteração correta do projeto
Atualizado!
Linha para mostrar o patch!

$ git fetch other

$ git diff other/master > /tmp/test.patch

$ cat /tmp/test.patch
diff --git a/README b/README
index d91bb84..0f3d562 100644
--- a/README
+++ b/README
@@ -2,3 +2,4 @@ Projeto do livro de Git
 ---
 Primeira alteração correta do projeto
 Atualizado!
+Linha para mostrar o patch!
```

Só isso, nosso `patch` está criado! Foi só redirecionar a saída do `diff` para um arquivo. Uia!

Aplicando um patch

Legal, agora que temos o arquivo com o `patch`, podemos mandar por e-mail para o nosso amigo. E como que ele vai aplicar o `patch`? Vamos para o diretório `other` fazer uma simulação disso, usando o comando `patch` (disponível na maioria das distribuições GNU/Linux, consulte a sua distribuição/sistema operacional):

```
$ cd ~/git/other

$ patch -p1 < /tmp/test.patch
patching file README

$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteração correta do projeto
Atualizado!
Linha para mostrar o patch!
```

E olhem o patch aplicado corretamente!

Enviando patches por e-mail

Podemos economizar algum tempo já preparando os `patches` para serem enviados por e-mail, utilizando `git format-patch`, criando um novo arquivo e `commit`:

```
$ git checkout -b work
Switched to a new branch 'work'

$ echo "mmm" > m.txt && git add m.txt && git commit -am "Inseridos os M's"
[work 38ba8b2] Inseridos os M's
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 m.txt

$ git format-patch origin/master -o /tmp
/tmp/0001-Inseridos-os-M-s.patch
```

Especificamos acima que queremos os `commits` que **estão** na *branch* corrente (`work`) e **não estão** em `origin/master`, e pedimos para gravar o(s) arquivo(s) resultantes no diretório `/tmp`. Dando uma olhada no arquivo resultante:

```
$ cat /tmp/0001-Inseridos-os-M-s.patch
From 38ba8b27bced0be93cc4142f46a065326770eb18 Mon Sep 17 00:00:00 2001
From: Eustaquio Rangel <taq@eustaquiorangel.com>
Date: Thu, 27 Oct 2016 17:01:25 -0200
Subject: [PATCH] Inseridos os M's
---
 m.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 m.txt
diff --git a/m.txt b/m.txt
new file mode 100644
index 0000000..e1a5159
--- /dev/null
+++ b/m.txt
@@ -0,0 +1 @@
+mmm
--
2.9.3
```

Esse arquivo já está pronto no formato para ser enviado por e-mail, inclusive com todos os cabeçalhos necessários para tal. Podemos enviá-los como arquivos anexos, ou enviar pela linha de comando mesmo.

Pacote adicional para o GNU/Linux

Algumas distribuições, como o Ubuntu, vão precisar de um pacote adicional para o envio do e-mail pela linha de comando, instalado facilmente através de:

```
$ sudo apt-get install git-email
```

Vamos enviar o `patch` (ou os `patches`, onde podemos utilizar os nomes dos arquivos ou uma máscara):

```
$ git send-email --smtp-encryption=tls --smtp-server=smtp.gmail.com
--smtp-user=eustaquiorangel@gmail.com --smtp-server-port=587
--to eustaquiorangel@gmail.com /tmp/0001-Inseridos-os-M-s.patch
```

Esse comando pode ser executado utilizando um comando como a seguir, que pode ser facilmente convertido em um *shell script*, ou utilizar as seguintes configurações (adaptadas, lógico, para os valores corretos):

```
$ git config --global sendemail.smtpencryption tls
$ git config --global sendemail.smtpserver smtp.gmail.com
$ git config --global sendemail.smtpuser eustaquiorangel@gmail.com
$ git config --global sendemail.smtpserverport 587
```

Após executar o comando, será requisitada a senha do e-mail antes de começar a enviar os arquivos.

Podemos agora fazer um *merge* da *branch* `work` com a `master`, para manter o arquivo `m.txt`.

Aplicando os patches recebidos por e-mail

Para os `patches` recebidos por e-mail, temos que salvar a mensagem completa. Como temos muito costume de utilizarmos webmail hoje em dia (o Gmail me salva a pátria!), temos que pedir para ver o conteúdo original da mensagem antes de salvar. No Gmail isso pode ser feito selecionando "*Mostrar original*", no menu pop-up no lado superior direito da mensagem, resultando em algo como:

```
Return-Path: <eustaquiorangel@gmail.com>
Received: from localhost.localdomain ([187.66.70.30])
        by mx.google.com with ESMTPS id r45sm3852877yhg.18.2012.06.06.16.09.53
        (version=TLSv1/SSLv3 cipher=OTHER);
        Wed, 06 Jun 2012 16:09:54 -0700 (PDT)
Sender: =?UTF-8?Q?Eust=C3=A1quio_Rangel?= <eustaquiorangel@gmail.com>
From: Eustaquio Rangel <taq@eustaquiorangel.com>
To: eustaquiorangel@gmail.com
Cc: Eustaquio Rangel <taq@eustaquiorangel.com>
Subject: [PATCH] Inseridos os M's
Date: Thu, 27 Oct 2016 17:10:25 -0200
Message-Id: <1339024072-14772-1-git-send-email-taq@eustaquiorangel.com>
X-Mailer: git-send-email 1.7.10

---
 m.txt |    1 +
 1 file changed, 1 insertion(+)
 create mode 100644 m.txt

diff --git a/m.txt b/m.txt
new file mode 100644
index 0000000..e1a5159
--- /dev/null
+++ b/m.txt
@@ -0,0 +1 @@
+mmm
```

```
--
2.9.3
```

Vamos salvar esse conteúdo em um arquivo chamado `/tmp/email.patch`. Após devidamente salvo o conteúdo do e-mail, vamos aplicar os `patches` no diretório `other`, utilizando `git am`:

```
$ git am /tmp/email.patch
Applying: Inseridos os M's

$ cat m.txt
mmm
```

E olhem lá o arquivo `m.txt`! Não esqueçam de fazer os `commits` necessários após aplicar os `patches` recebidos.

Branches remotas

Como pudemos ver, as *branches* no Git são muito práticas: rápidas, fáceis de criar e de apagar no repositório local. Mas e se estivessemos alterando uma *branch* local, chamada, por exemplo, *development*, e no final do dia ela não estivesse em um estado bom para fazer `merge` com a *branch* de produção? E aí? Formatamos trocentos `patches` para salvar e aplicar depois? Funciona, mas fica meio estranho. É aí que podemos enviar essa *branch* local para o repositório remoto.

Criando branches remotas

Primeiro vamos criar a *branch* e adicionar algum conteúdo:

```
$ git checkout -b development
Switched to a new branch 'development'

$ echo "111" > 1.txt && echo "222" > 2.txt && echo "333" > 3.txt

$ git add *.txt && git commit -am "Iniciando com numeros"
[development 70b491a] Iniciando com numeros
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 1.txt
 create mode 100644 2.txt
 create mode 100644 3.txt

$ git branch -a
* development
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/outro/master
```

E agora vamos enviar o conteúdo dessa *branch* para o repositório remoto:

```
$ git push origin development
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (5/5), 364 bytes, done.
Total 5 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
To /home/taq/git/repobare
 * [new branch]      development -> development
```

Para provar que a *branch* realmente foi salva, vamos apagar o conteúdo do repositório local e clonar o repositório novamente, simulando estar chegando para trabalhar novamente e precisar daquele conteúdo salvo anteriormente:

```
$ pwd
/home/taq/git/repo

$ cd ..

$ rm -rf repo

$ git clone ~/git/repobare repo
Cloning into repo...

$ cd repo

$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/development
  remotes/origin/master
```

A *branch* `development` está lá, e podemos trocar para ela logo após clonar o repositório:

```
$ git checkout development
Branch development set up to track remote branch development from origin.
Switched to a new branch 'development'

$ ls
total 72K
drwxr-xr-x  3 taq  taq  4,0K .
drwxrwxrwt 17 root root 4,0K ..
-rw-rw-r--  1 taq  taq     4 1.txt
-rw-rw-r--  1 taq  taq     4 2.txt
-rw-rw-r--  1 taq  taq     4 3.txt
-rw-rw-r--  1 taq  taq    12 a.txt
-rw-rw-r--  1 taq  taq     4 b.txt
...
```

Inclusive, a *branch* é indicada para apontar para a *branch* correspondente no repositório remoto, ou seja, quaisquer alterações no repositório local nessa *branch*, quando feito o `push`, vão automaticamente para a *branch* remota. Em algumas versões anteriores era necessário fazer isso manualmente.

Se for necessário apontar uma *branch* local para uma *branch* remota, podemos utilizar a opção `-u` (*upstream*):

```
$ git checkout -b teste
Switched to a new branch 'teste'

$ git branch -u origin/master
Branch teste set up to track remote branch master from origin.
```

Nesse exemplo, a *branch* local `teste` está apontando para a remota (em `origin`) `master`.

Clonando e indo direto para a branch desejada

Existe um atalho para ir direto para a branch após clonar o repositório:

$ git clone -b development ~/git/repobare repo

Apagando branches remotas

E depois quando não precisarmos mais da *branch* remota? Para nossa sorte, existe um meio para apagá-la do repositório remoto. É um meio meio estranho. Meio esquisito. Mas é o jeito que inventaram para fazer isso. Para apagar a *branch* remota, é só enviar um `push` para o repositório remoto usando `:` (dois pontos, sim) logo antes do nome da branch remota (sem espaços). Esquisito, mas funciona:

```
$ git checkout master

$ git push origin :development
To /home/taq/git/repobare
 - [deleted]         development

$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
```

Dica

Também temos um *syntax sugar* para apagar uma *branch* remota, levando em conta a *branch* apagada acima:

```
$ git push origin --delete development
```

Atenção!

O Git **não vai perguntar se você deseja apagar a *branch* remota**, então tome **muito cuidado com isso**!

Rodando seu repositório

Até agora falamos de utilizar o sistema de arquivos do computador local para os nossos repositórios. O Git permite isso e também o uso de `SSH`, `HTTP` e até do seu próprio protocolo. Vamos ver como podemos rodar um repositório em um computador (seja ele o local ou remoto) utilizando o protocolo do Git:

```
$ cd /home/taq/git/

$ git daemon --verbose --export-all --base-path=.

$ cd /home/taq/git/

$ git clone git://localhost/repobare newcopy

$ cd newcopy

$ ls
```

Ali fomos para o diretório base (indicado através de `--base-path`, no qual o ponto ali indica o diretório corrente) e disparamos o *daemon*. Desse modo, qualquer diretório abaixo controlado pelo Git pode ser clonado.

Para receber `pushes`, adicionar `--enable=receive-pack` no comando do `daemon`.

Cuidado!

Utilizando dessa forma, não há autenticação e qualquer um poderá fazer `push` no repositório.

Algumas opções de gerenciamento de repositório

Se por acaso não pudermos - ou não quisermos - utilizar alguns dos vários sites que fornecem suporte à repositórios com o Git, podemos utilizar algumas ferramentas em nossa rede para criar e gerenciar os repositórios. A configuração dessas ferramentas está fora do escopo desse livro, mas ficam as dicas para algumas delas:

- Gitlab[13]
- Gitolite[14]

E para o pessoal que precisar hospedar em Windows ® (então né), aqui tem umas dicas boas[15].

[13] http://gitlab.com
[14] http://gitolite.com
[15] http://freshbrewedcode.com/derekgreer/2012/02/19/hosting-a-git-repository-in-windows/

Procurando coisas erradas

A partir daqui vamos aprender alguns recursos do Git que, apesar de não utilizarmos no dia-a-dia, são uma mão-na-roda quando precisamos deles.

Não adianta, por mais que utilizemos ferramentas de ponta, que prestemos atenção, alguma hora alguma coisa pode sair errada. A parte importante é ter recursos para procurar o que está errado de uma forma prática e rápida. E, para variar, o Git nos fornece um recurso muito útil para essas situações, o `bisect`.

Bisect

Lembram-se da nossa convenção de inserir no conteúdo do arquivo a letra do nome dele, 3 vezes, em uma linha? Pois é, lá no comecinho, apenas o `a.txt` foi inserido com 3 linhas, por causa da alteração do chefe desesperado, lembram? Pois é, agora devemos ter **13** arquivos `*.txt` com **15** linhas no total. Vamos verificar isso usando uma linha de comando com alguns *pipes*:

```
$ find *.txt | wc -l && wc -l *.txt | tail -n1 | cut -f1 -d' '
```

Uma explicação rápida do que acontece ali:

1. O `find *.txt` encontra todos os arquivos `*.txt` presentes no diretório.
2. O `wc -l` conta quantas linhas foram passadas para ele, ou seja, quantos arquivos encontrou.
3. O `&&` executa outro comando.
4. Novamente `wc -l`, mas nesse caso, passando a máscara de arquivos, ele soma a quantidade de linhas dentro dos arquivos.
5. O `tail -n1` mostra somente a última linha.
6. O `cut -f1 -d' '` pega o primeiro campo, separando por espaço.

Se por acaso durante o percurso do livro algum arquivo ficou com mais de 1 linha, fora o `a.txt`, corrija para que possamos continuar com esses números de linhas citadas abaixo.

Programação em shell script

Se vocês utilizando um sistema Unix-like e nunca se interessaram por *shell scripting*, por favor, reconsiderem. Quebra um galhão. Uma fonte ótima e recomendadíssima é o livro do Aurélio Jargas[16].

Executando a linha acima, vamos ter o seguinte resultado:

```
13
17
```

Epa! Não seria que ser 13/15 total? O que aconteceu? O arquivo `a.txt` teria que ser o único a ter 3 linhas, onde que está o erro? Qual o sentido da vida? E porque 42 é resposta do Pensador Profundo para isso? Calma, calma, vamos utilizar o `bisect` para nos auxiliar a encontrar em que ponto do nosso repositório que as coisas deram errado.

[16] http://www.shellscript.com.br

Informando para o bisect se as coisas estão boas ou ruins

O `bisect` pede a nossa ajuda para indicar se o estado corrente está bom ou ruim, desse modo, encontrando o `commit` que resultou no problema que estamos tendo, utilizando uma busca binária para procurar o erro.

Antes de mais nada, precisamos indicar que queremos usar o `bisect` e indicar quais os `commits` que estão em um estado ruim e um estado bom. Sabemos que o estado atual está ruim, como indicado na linha de comando executada acima, e o `commit` no qual o arquivo `a.txt` foi adicionado estava bom, então vamos procurar qual era o `SHA-1` dele, utilizando a opção `grep` no `log`:

```
$ git log --grep "A's"
```

que retorna algo os *commits* em que foram encontrados os termos de busca que indicamos.

Também podemos consultar quais `commits` que tem o arquivo:

```
$ git log --all a.txt
```

Quer saber onde que determinado código foi introduzido no repositório? É só utilizar `git log -S <termo>`:

```
$ git log -S iii
commit 38f5e7d150c1b4c910705ae7e20f56845c7c02b1
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Tue Aug 23 10:57:18 2016 -0300

    Adicionados os I's
```

Dica de atalho

Podemos customizar um atalho `find` para fazer as buscas:

find = log –pretty=”format:%Cgreen%H %Cblue%s” –name-status –grep

Aí só pegar o `SHA-1` do `commit` onde o arquivo estava correto, que vai ser o indicado como bom. Vamos iniciar o `bisect` e indicar os *commits* ruim, que é o atual, e o bom, que foi o que encontramos acima:

```
$ git bisect start

$ git bisect bad

$ git bisect good 724dd31d
Bisecting: 9 revisions left to test after this (roughly 3 steps)
[8b04f2b86a36bac3e1afb64968ed1e5c8e0551d8] Adicionados os H's
```

Agora rodamos a linha de comando novamente:

```
$ find *.txt | wc -l && wc -l *.txt | tail -n1 | cut -f1 -d' '
8
10
```

Esse estava legal, 8 arquivos, 10 linhas (lembrem-se: a diferença de linhas são 2), vamos indicar que o estado atual é bom. Importante notar que se der algum outro resultado que seja diferente das 2 linhas que estamos convencionando que são o resultado esperado, ao invés de bom com `good`, marque como ruim com `bad`. Aqui vamos ter um *commit* bom:

```
$ git bisect good
Bisecting: 4 revisions left to test after this (roughly 2 steps)
[6ea3655d6b23f3ab436f710a7509718a60706233] Mais I's
```

Executamos a linha de comando novamente até identificar um estado em que o número de linhas não está legal:

```
$ find *.txt | wc -l && wc -l *.txt | tail -n1 | cut -f1 -d' '
9
13
```

Nesse ponto informamos que o estado atual é ruim:

```
$ git bisect bad
```

E continuamos marcando como ruins e bons até que o `bisect` termine a sua busca binária, baseada em nossas respostas e os `commits` do repositório, nos dedurando qual foi o `commit` que produziu o nosso problema, uia:

```
  6ea3655d6b23f3ab436f710a7509718a60706233 is the first bad commit
  commit 6ea3655d6b23f3ab436f710a7509718a60706233
  Author: Eustaquio Rangel <taq@eustaquiorangel.com>
  Date:   Wed Oct 26 21:18:53 2016 -0200

      Mais I's

  :100644 100644 48519283cb8ffc0c1418ed853f173844f09881e1 9ba5888a28eab75e51111a\
f7114e792d7f23988b
  M       i.txt
```

Nesse momento, temos que informar que queremos parar de usar o `bisect`:

```
  $ git bisect reset
  Previous HEAD position was e894605... Merge branch 'work' into merge
  Switched to branch 'master'
```

E podemos ver o que foi alterado no **commit**, usando o `SHA-1` encontrado, com `show`:

```
  $ git show 6ea365
  commit 6ea3655d6b23f3ab436f710a7509718a60706233
  Author: Eustaquio Rangel <taq@eustaquiorangel.com>
  Date:   Wed Oct 26 21:18:53 2016 -0200

      Mais I's

  diff --git a/i.txt b/i.txt
  index 4851928..9ba5888 100644
  --- a/i.txt
  +++ b/i.txt
  @@ -1 +1,3 @@
   iii
  +iii
  +iii
```

Ou pedir para ver a diferença com `diff` entre uma versão do *commit* para trás (utlizando `^`) e ele mesmo:

```
$ git diff 6ea365^ 6ea365
diff --git a/i.txt b/i.txt
index 4851928..9ba5888 100644
--- a/i.txt
+++ b/i.txt
@@ -1 +1,3 @@
 iii
+iii
+iii
```

Automatizando o bisect

Podemos melhorar mais ainda o uso do `bisect`, fazendo um *shell script* para rodar a cada iteração, o qual retornará 0 para o caso do estado atual for bom, e resultado diferente de 0 (nesse caso, 1) se o estado atual for ruim:

```
$ cat checker.sh
#!/bin/bash

files=$(find -iname '*.txt' | wc -l)
rows=$(wc -l *.txt | tail -n1 | cut -b1-2 | tr -d ' ')
ok=$(($files+2))

echo verificando $files arquivos e $rows linhas, linhas tem que ser $ok

if [ "$rows" -ne "$ok" ]; then
   echo diferentes!
   exit 1
fi
echo iguais!
exit 0
```

Explicando o script:

1. Verifica quantos arquivos `*.txt` se encontram no diretório
2. Verifica quantas linhas esses arquivos tem, selecionando somente o valor numérico
3. Calcula qual o valor esperado correto
4. Mostra uma mensagem indicando quais são os valores atuais
5. Se os resultados das linhas forem diferentes do esperado, retorna 1
6. Se forem iguais, retorna 0

Não esquecendo de ativar o flag de executável no *script*:

```
$ chmod +x checker.sh
```

Agora podemos executar o `bisect`, indicando direto quais são os commits ruim e bom (nessa ordem, hein!):

```
$ git bisect start HEAD 724dd3
```

E agora executar `bisect run`, indicando o nosso *script*:

```
$ git bisect run ./checker.sh
running ./checker.sh
verificando 8 arquivos e 10 linhas, linhas tem que ser 10
iguais!

...

Bisecting: 0 revisions left to test after this (roughly 1 step)
[e8946058326f2417fa7757723b30858dde468739] Merge branch 'work' into merge
running ./checker.sh
verificando 9 arquivos e 11 linhas, linhas tem que ser 11
iguais!

6ea3655d6b23f3ab436f710a7509718a60706233 is the first bad commit
commit 6ea3655d6b23f3ab436f710a7509718a60706233
Author: Eustaquio Rangel <taq@eustaquiorangel.com>
Date:   Wed Oct 26 21:18:53 2016 -0200
    Mais I's

:100644 100644 48519283cb8ffc0c1418ed853f173844f09881e1 9ba5888a28eab75e51111a\
f7114e792d7f23988b
M       i.txt
bisect run success
```

No final, foi indicado o `commit` que introduziu o problema e retornou o status de `success`. Yay! Agora só precisamos indicar que encontramos o que queríamos e vamos parar de usar o `bisect`:

```
$ git bisect reset
Previous HEAD position was e894605... Merge branch 'work' into merge
Switched to branch 'master'
```

Submódulos

Imaginem que precisemos, dentro do nosso repositório, de um diretório com outro repositório, controlado separadamente também pelo Git. Esse é o conceito de **submódulos**.

Para entender esse conceito, vamos criar outro repositório `bare`, chamado `sub`:

```
$ cd ~/git
$ git init --bare sub
Initialized empty Git repository in /home/taq/git/sub/
```

Agora vamos clonar esse novo repositório, adicionar algum conteúdo e fazer um `push` para popular o diretório `bare` recém criado:

```
$ git clone ~/git/sub newsub
Cloning into 'newsub'...
warning: You appear to have cloned an empty repository.
done.

$ cd newsub

$ echo "sub1" >> sub1.txt

$ git add .

$ git commit -am "Primeiro commit"
[master (root-commit) 528c5ee] Primeiro commit
 1 file changed, 1 insertion(+)
 create mode 100644 sub1.txt

$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 218 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /home/taq/git/sub
 * [new branch]      master -> master
```

Agora podemos retornar para o nosso repositório padrão e adicionar o repositório como um submódulo:

```
$ cd ~/git/repo
$ git submodule add ~/git/sub
Cloning into 'sub'...
done.
```

Vamos dar uma olhada como está o arquivo `.gitmodules` que foi criado:

```
[submodule "sub"]
   path = sub
   url = /home/taq/git/sub
```

E agora inicializar e atualizar os submódulos (na primeira vez em que o submódulo é criado, nem é necessário fazer os dois, mas é bom para garantir alguma mudança):

```
$ git submodule init
Submodule 'sub' () registered for path 'sub'

$ git submodule update

$ ls sub/
total 16K
drwxr-xr-x 2 taq taq 4,0K .
drwxr-xr-x 4 taq taq 4,0K ..
-rw-rw-r-- 1 taq taq   45 .git
-rw-rw-r-- 1 taq taq    5 sub1.txt
```

Atualizando o submódulo

E agora o que acontece se ocorrerem atualizações no repositório do submódulo? Como fica o diretório local em que clonamos o seu conteúdo? Vamos dar uma olhada, primeiro voltando para o diretório onde clonamos somente o submódulo, fazendo algumas alterações (adicionando mais algumas linhas no arquivo `sub1.txt`) e um `push` para atualizar o repositório remoto:

```
$ cd ~/git/newsub/

$ echo "sub1" >> sub1.txt

$ echo "sub1" >> sub1.txt

$ cat sub1.txt
sub1
sub1
sub1

$ git commit -am "Adicionadas mais algumas linhas no arquivo"
[master 498aafe] Adicionadas mais algumas linhas no arquivo
 1 file changed, 2 insertions(+)

$ git push
Counting objects: 5, done.
Writing objects: 100% (3/3), 273 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /home/taq/git/sub
   528c5ee..498aafe  master -> master
```

Voltando para o nosso repositório padrão e dando uma olhada no diretório do submódulo:

```
$ cd ~/git/repo

$ cat sub/sub1.txt
sub1
```

Oh-oh, parece que tudo continua na mesma. Isso tem até um bom motivo: enquanto estamos fazendo alterações em nosso *working directory*, talvez não seja uma boa idéia atualizar os submódulos. Imaginem que na primeira vez que criamos o submódulo ele veio com a versão `1.0.0` de alguma *lib* que precisamos para o nosso projeto, mas a versão atual `1.1.0`, que está no repositório remoto do submódulo no momento atual introduz uma quebra de compatibilidade com a versão utilizada anteriormente. Nosso projeto com certeza não vai funcionar enquanto não o adaptarmos para a nova versão. Por isso que fica a critério de cada um se as atualizações do submódulo são necessárias, como nesse caso, onde queremos atualizar e dar uma olhada no conteúdo. Vamos dar uma de machos e usar `pull` direto:

```
$ git pull
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/taq/git/sub
   528c5ee..498aafe  master     -> origin/master
Updating 528c5ee..498aafe
Fast-forward
 sub1.txt |    2 ++
 1 file changed, 2 insertions(+)

$ cat sub1.txt
sub1
sub1
sub1
```

Atualizando o submódulo de dentro do nosso repositório

E se quisermos atualizar, de dentro do repositório onde criamos o submódulo, o próprio submódulo?

```
$ cd ~/git/repo

$ cd sub

$ echo "sub2" > sub2.txt

$ git add sub2.txt

$ git commit -am "Adicionado o sub2"
[master a82ea1c] Adicionado o sub2
 1 file changed, 1 insertion(+)
 create mode 100644 sub2.txt

$ git push
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 282 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
```

```
   To /home/taq/git/sub
      498aafe..a82ea1c  master -> master
```

Esse processo foi bem melhorado nas versões mais atuais do Git. Anteriormente, ir para o diretório do submódulo nos levava para um estado mula-sem-cabeça, onde precisávamos fazer um `checkout` para ir para a *branch* `master`. Se por acaso você for para um estado mula-sem-cabeça, façam o `checkout` em alguma *branch*, provavelmente na `master`.

Agora vamos dar uma olhada se o conteúdo foi realmente atualizado, fazendo a sequência mariquinha no outro diretório onde o repositório do submódulo foi clonado:

```
$ cd ~/git/newsub/

$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/taq/git/sub
   a82ea1c..19aaf31  master     -> origin/master

$ git diff origin/master
diff --git a/sub2.txt b/sub2.txt
deleted file mode 100644
index 48df0cb..0000000
--- a/sub2.txt
+++ /dev/null
@@ -1 +0,0 @@

$ git merge origin/master
Updating a82ea1c..19aaf31
Fast-forward
 sub2.txt |    1 +
 1 file changed, 2 insertions(+)
```

Podemos clonar um repositório já inicializando os submódulos com `--recursive`:

```
  git clone ~/git/repobare repo --recursive
  Cloning into 'repo'...
  done.
  Submodule 'sub' (/home/taq/code/git/conhecendo-o-git/sub) registered for path \
'sub'
  Cloning into '/home/taq/code/git/conhecendo-o-git/repo/sub'...
  done.
  Caminho do sub-módulo 'sub': confirmado '699caee0802c18ea033be3ac8d43096a4f35f\
3a3'
```

Ganchos

Temos um recurso bem interessante que se chamam "ganchos" (*hooks*, em Inglês, não é Hulk não, hein). São procedimentos que são disparados a partir de eventos aceitos pelo Git e configurados pelo usuário, divididos entre os de servidor e de cliente.

Ganchos no servidor

Alguns dos ganchos que podemos configurar no servidor são:

- `pre-receive` - acionado antes de receber um push
- `post-receive` - acionado depois de receber um push
- `update` - similar ao `pre-receive`, que roda apenas uma vez, o update roda para cada *branch* recebida.

Os ganchos tem que ser configurados em nosso diretório *bare* (`~/git/repobare`). Vamos imaginar a seguinte situação: sempre que o repositório remoto receber um `push`, deve copiar, exportando, o conteúdo do repositório para um determinado diretório. Vamos utilizar o diretório `/tmp/docs` nesse caso.

Antes de mais nada, vamos para o diretório do repositório `bare`:

```
$ cd ~/git/repobare/
```

E agora criar o arquivo `hooks/post-receive` com o seguinte conteúdo:

```
$ cat hooks/post-receive
#!/bin/sh
if [ -d /tmp/docs ]; then
   rm -rf /tmp/docs
fi
mkdir /tmp/docs
GIT_WORK_TREE=/tmp/docs git checkout -f
```

Alterar o *flag* de executável dele:

```
$ chmod +x hooks/post-receive
```

E para comprovar que ainda não existe esse diretório:

```
  $ ls /tmp/docs
  ls: impossivel acessar /tmp/docs: Arquivo ou diretorio nao encontrado
```

Agora podemos retornar para o outro diretório que contém o nosso repositório padrão, aproveitar que não adicionamos o `checker.sh` na staging area, adicionar e fazer um `push`:

```
  $ git add checker.sh

  $ git commit -am "Adicionado o verificador para o bisect"
  [master 2860860] Adicionado o verificador para o bisect
   1 files changed, 11 insertions(+), 0 deletions(-)
   create mode 100755 checker.sh

  $ git push
  Counting objects: 4, done.
  Delta compression using up to 2 threads.
  Compressing objects: 100% (3/3), done.
  Writing objects: 100% (3/3), 485 bytes, done.
  Total 3 (delta 1), reused 0 (delta 0)
  Unpacking objects: 100% (3/3), done.
  To /home/taq/git/repobare
     e000fa2..2860860  master -> master
```

E agora dando uma olhada em /tmp/docs:

```
  $ ls /tmp/site
  total 60K
  drwxrwxr-x  2 taq  taq  4,0K 2016-10-27 19:34 .
  drwxrwxrwt 18 root root 4,0K 2016-10-27 19:34 ..
  -rw-rw-r--  1 taq  taq    12 2016-10-27 19:34 a.txt
  -rw-rw-r--  1 taq  taq     4 2016-10-27 19:34 b.txt
  -rwxrwxr-x  1 taq  taq   277 2016-10-27 19:34 checker.sh
  ...
```

Yay! O gancho foi acionado corretamente.

Ganchos no cliente

Alguns dos ganchos que podemos utilizar no cliente são:

- `pre-commit` - Roda antes de digitar a mensagem de `commit`
- `prepare-commit-msg` - Roda antes do editor, mas depois de gerar a mensagem *default* de `commit`
- `commit-msg` - Recebe como parâmetro o path do arquivo temporário com a mensagem, podendo terminar o processo de `commit` se o script retornar não-zero.
- `post-commit` - Roda após o `commit`
- `pre-rebase` - Roda antes do `rebase`
- `post-checkout` - Roda após o `checkout`
- `post-merge` - Roda após o `merge`

Esses ganchos são definidos no diretório `.git/hooks` (abaixo do diretório do repositório corrente) e com um detalhe muito importante: **eles não são salvos ali naquele diretório quando enviamos o conteúdo para o repositório remoto**! Ou seja, depois que apagamos o repositório/diretório local, já era. Mas que diabos, porque fazem isso?

Vamos imaginar a seguinte situação (não dando sugestões para ninguém fazer meleca, por favor), se os ganchos de cliente fossem salvos no repositório remto: um programador que tem acesso ao repositório vai ser despedido no dia seguinte, porque ele é muito encrenqueiro, rock star e arrogante - os piores tipos, o que vale dizer que é baita desperdício quando o sujeito é realmente um programador bom - e resolve fazer uma sacanagem: configurar o `post-commit` para exibir uma mensagem mandando todo mundo para ponte que caiu e apagando o conteúdo do `home` do usuário. Feio né? Após fazer a mudança do gancho e mandar para o repositório (novamente: se os ganchos de cliente fossem salvos assim), o primeiro "sortudo" que clonasse o repositório e fizesse um `commit` ia ficar muito bravo.

Por isso que os ganchos de clientes não são salvos automaticamente. Uma forma de salvá-los é configurar um diretório dentro do repositório com os ganchos, e após clonar o repositório, mover os ganchos (dando uma olhadinha neles em situações meio delicadas como a representada acima) para o diretório `.git/hooks`.

Alerta e explicação feitos, vamos dar uma olhada como configurar o `post-commit`, inserido o seguinte conteúdo em `.git/hooks/post-commit`, mostrado aqui por `cat`:

```
1  $ cat .git/hooks/post-commit
2  #!/bin/sh
3  echo "Obrigado pelo seu commit."
4  echo "Se voce fez coisa errada, vamos te pegar, safado!"
```

Agora vamos ativar o *flag* de executável do arquivo:

```
$ chmod +x .git/hooks/post-commit
```

E fazer mais um `commit` para testar, alterando o `README`:

```
$ cat README
Projeto do livro de Git
---
Primeira alteração correta do projeto
Atualizado!
Quase acabando o livro!

$ git commit -am "Testando o post-commit"
Obrigado pelo seu commit.
Se voce fez coisa errada, vamos te pegar, safado!
[master 7e1e9ec] Testando o post-commit
 1 file changed, 1 insertion(+)
```

Rerere

O que é o rerere

Não, sério, não estou sacaneando não (rerere!), o nome do negócio é esse mesmo! É uma abreviação de "*reuse recorded resolution*". Esse recurso permite que "ensinemos" ao Git alguns jeitos de resolver conflitos automaticamente.

Utilizando o rerere

Vamos pegar como exemplo o conflito do separador do `README` que utilizamos bem no começo do livro, inserindo novamente um conflito no final do arquivo:

```
$ git checkout master

$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteração correta do projeto
Atualizado!
Quase acabando o livro!

$ git branch -D work

$ git branch -D test

$ git checkout -b work
Switched to a new branch 'work'

$ echo "***" >> README

$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteração correta do projeto
Atualizado!
Quase acabando o livro!
```

```
***

$ git commit -am "Terminador"
[work 8effb63] Terminador
 1 file changed, 1 insertion(+)

$ git checkout master
Switched to branch 'master'

$ git checkout -b test
Switched to a new branch 'test'

$ echo "---" >> README

$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteraçãoo correta do projeto
Atualizado!
Quase acabando o livro!
---

$ git commit -am "Terminador"
[test c209b79] Terminador
 1 file changed, 1 insertion(+)
```

E agora vamos habilitar o uso do `rerere`;

```
$ git config --global rerere.enabled true
```

E fazer o `merge` de `test` em `work`, o que vai gerar o conflito:

```
$ git checkout work

$ git merge test
Auto-merging README
CONFLICT (content): Merge conflict in README
Recorded preimage for 'README'
Automatic merge failed; fix conflicts and then commit the result.
```

Como habilitamos o `rerere`, podemos perguntar qual o seu status atual:

```
$ git rerere status
README
```

Se dermos uma olhada no diretório `.git/rr-cache` (que, se criado, é outra forma de habilitar o `rerere`), temos algo como:

```
$ cat .git/rr-cache/ea22efd4151554610acece42030c273e98bed0f7/preimage
Projeto do livro "Conhecendo o Git"
----------------------------------
Primeira alteração correta do projeto
Atualizado!
Quase acabando o livro!
<<<<<<<
***
=======
---
>>>>>>>
```

Agora, vamos resolver o conflito, editando o arquivo `README` e deixando o separador `---`, afinal, se parece mais com uma linha do que *** que parecem três piolhos achatados e dando uma olhada no `rerere diff` para ver do que ele vai se lembrar:

```
$ git rerere diff
--- a/README
+++ b/README
@@ -3,8 +3,4 @@
 Primeira alteração correta do projeto
 Atualizado!
 Quase acabando o livro!
-<<<<<<<
-***
-=======
 ---
->>>>>>>
```

Ali basicamente diz: "*quando encontrar três asteriscos no início da linha, pode trocar para três hífens*". Vamos terminar a resolução do conflito adicionando o arquivo na *staging area*, fazendo um `commit` e `merge` da `work` com a `master`:

```
$ git add README

$ git commit
Recorded resolution for 'README'.
[work 4e48042] Merge branch 'test' into work
```

Agora vamos testar se o `rerere` realmente aprendeu alguma coisa ou se está nos sacaneando com esse nome engraçado. Vamos repetir os procedimentos com os três asteriscos e os três hífens no final do `README`:

```
$ git checkout master

$ git branch -D work

$ git branch -D test

$ git checkout -b work

$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteração correta do projeto
Atualizado!
***

$ git commit -am "Terminador"
[work c0f305a] Terminador
 1 file changed, 1 insertion(+)

$ git checkout master

$ git checkout -b test

$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteração correta do projeto
Atualizado!
---

$ git commit -am "Terminador"
[test 944714d] Terminador
```

```
 1 file changed, 1 insertion(+)

$ git checkout work
Switched to branch 'work'

$ git diff test
diff --git a/README b/README
index b54fdc3..a537ec3 100644
--- a/README
+++ b/README
@@ -3,4 +3,4 @@ Projeto do livro de Git
 Primeira alteração correta do projeto
 Atualizado!
 ----
+***
```

Agora estamos no estado que vai gerar o conflito novamente. Vamos para `work` fazer o `merge` de `test` e verificar o que acontece:

```
$ git merge test
Auto-merging README
CONFLICT (content): Merge conflict in README
Resolved 'README' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

A mensagem do conflito foi mostrada novamente, mas olhem a mensagem "`Resolved 'README' using previous resolution.`" e o conteúdo do `README`:

```
$ cat README
Projeto do livro "Conhecendo o Git"
-----------------------------------
Primeira alteração correta do projeto
Atualizado!
---
```

Yeah! O `rerere` realmente funciona e não é nenhuma sacanagem. Rerere.

Cursos e treinamentos

Se você gostou desse livro, a Bluefish[17] realiza consultoria e vários treinamentos e workshops *inhouse/incompany* sobre Git, Ruby/Rails e outras tecnologias.

Entre em contato conosco para mais detalhes através de contato@bluefish.com.br[18].

Bluefish

[17] http://www.bluefish.com.br
[18] contato@bluefish.com.br