PET - Computação

Manual de Git

FACOM - UFMS

Sumário

Introdução

Sistema Distribuido de Controle de Versão

Um sistema distribuído de controle de versão (DVCS em inglês) é um sistema que permite a um time de desenvolvimento coordenar as mudanças em uma base de código através do tempo. Ao invés de existir somente um repositório fixo com toda a história de desenvolvimento, cada conjunto local também contém todo o histórico de mudanças.

O que é o Git?

O Git é um DVCS criado pelo finlandês Linus Torvalds, também criador do kernel Linux. Ele foi criado a partir da necessidade de um sistema de controle de versão que suportasse a carga de desenvolvimento e mudanças do kernel Linux. Como Torvalds não encontrou nenhum sistema que alcançasse seus requisitos, tanto de performance quanto de uso, ele criou o git.

Usos do Git

O git é utilizado para controle de mudanças em uma base de código – um repositório. Seu funcionamento básico consiste em criar ou copiar um repositório, realizar alterações neste, gravar estas alterações, receber alguma modificação feita, e enviar as próprias atualizações. Ele permite ainda verificar quem fez alguma alteração, visualizar todo o histórico e ainda selecionar quais mudanças fazem parte do histórico principal.

Git NÃO é GITHUB!

Vale ressaltar que git é diferente de Github. Git é uma ferramenta (e também um protocolo), enquanto o github, bitbucket, gitlab, e outros são serviços que disponibilizam uma interface sobre o git "normal", de modo que o uso rotineiro do git seja facilitado.

Eles não são "um git", são serviços!

Instalação e Configuração

Instalação no Windows

Para o windows há diversas opções para instalação, sendo desde a versão em linha de comando do git até programas GUI ou atalhos no menu de contexto. A versão em linha de comando é o git propriamente dito, o qual pode ser obtido no seguinte endereço: <https://git-scm.com/>. Os programas GUI incluem o GitKraken <https://www.gitkraken.com/>, o GitHub Desktop <https://desktop.github.com/>, o SourceTreee <https://www.sourcetreeapp.com/>, entre outros.

Instalação no Linux

O git já é instalado por padrão na maioria das distribuições GNU/Linux. Caso não esteja instalado na sua distribuição, basta instalá-lo a partir do gerenciador de pacotes ou compilar a partir dos fontes a seguir.

Para as distribuições baseadas em Debian:

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

Para as distribuições que utilizam yum/dnf:

```
$ sudo yum install git
$ sudo dnf install git
```

Compilar a partir dos fontes

É possível também compilar o git a partir do código fonte. Para isso é necessário ter os seguintes pacotes instalados: autotools, curl, zlib, openssl, expat, e libiconv.

```
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev gettext
    ↪ libz-dev libssl-dev
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel openssl-
    ↪ devel perl-devel zlib-devel
```

Para a documentação em todos os formatos são requeridas as seguintes depedências adicionais:

```
$ sudo apt-get install asciidoc xmlto docbook2x getopt
$ sudo dnf install asciidoc xmlto docbook2X getopt
```

Para as distribuições baseadas em Fedoral/RHEL o seguinte passo também é necessário:

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

Você deve então obter a versão mais recente dos fontes a partir de <https://github.com/git/git/releases>. Depois é só compilar e rodar!

```
$ tar -zxf git-x.y.z.tar.gz
$ cd git-x.y.z
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Configuração Inicial

A configuração das opções do Git sempre utiliza o comando **git config**. A identificação do usuário é feita utilizando os seguintes comandos:

```
git config --global user.name "Fulano de Tal"
git config --global user.email "fulano.tal@ufms.br"
```

É necessário também configurar o editor de texto padrão para a escrita dos textos dos commits. Isso pode ser feito através do seguinte comando:

```
git config --global core.editor "CaminhoDoEditor"
git config --global core.editor vim #Para o editor vim
git config --global core.editor nano #Para o editor nano
```

Depois disso está tudo pronto para você começar a utilizar o Git!

Parte I

Git Básico

Git trees

Para entender melhor o git é necessário entender como ele mantém os arquivos durante o processo de desenvolvimento. Ele os guarda em três "árvores": o diretório de trabalho (Working Directory – WD), o Index (ou também staging area), e o repositório local. A figura 1 contém uma representação dessas três "árvores".

O diretório de trabalho contém os arquivos que você fez **checkout** e está trabalhando. O index, também conhecido como staging area, é a área em que os arquivos que foram modificados e marcados para mudanças (**add**) são colocados.

Depois de realizar o **commit**, as alterações passam para o repositório local, permitindo então seu envio para o servidor remoto.

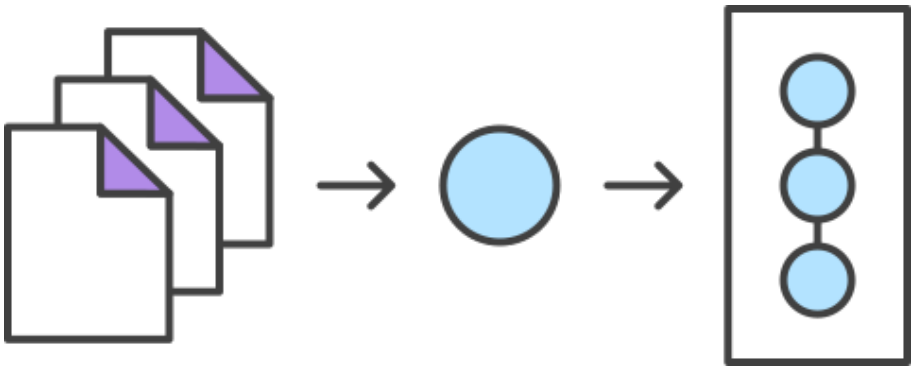


Figura 1 – Git trees

Trabalhando com repositórios

O que é um repositório?

Um repositório nada mais é do que um grupo de arquivos gerenciado pelo git. Um repositório contém, além das versões mais recentes, todas as versões anteriores e ainda informações sobre **branches**, **tags** e **commits**.

Criando um repositório

Pode se obter um projeto Git de duas formas principais. A primeira é fazendo uso de um projeto ou diretório existente e importá-lo para o Git. A segunda é clonar um repositório Git existente a partir de outro servidor.

Adicionando um repositório

Para iniciar um novo repositório a partir de um diretório é necessário, através do terminal, acessar o local escolhido e executar o comando:

```
$ git init
```

Isso cria um subdiretório oculto chamado **.git** que contém todos os arquivos necessários ao versionamento, porém nada ainda é monitorado.

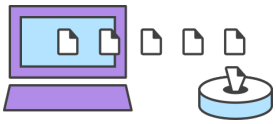
Clonando um repositório

Para copiar um repositório Git já existente o comando necessário é **git clone** [**url**]. Quando se roda esse comando cada versão de cada arquivo do histórico do projeto é obtido. Por exemplo, caso queria clonar a biblioteca Git do Ruby chamada Grit, você pode fazê-lo da seguinte forma:

```
$ git clone git://github.com/schacon/grit.git
```

Isso criará um diretório chamado **grit** e inicializará um diretório **.git** dentro deste, obtendo todas as versões de todos os arquivos além de armazenar no diretório de trabalho os arquivos correspondentes a versão mais recente.

Alterando o código



O que é um commit?

Um **commit** é o objeto básico quando se trabalha com o git. Ele guarda as alterações feitas no diretório de trabalho, associando essa mudança ao autor e às alterações anteriores, construindo assim uma árvore que contém toda a história do repositório.

Um commit contém os arquivos alterados, o autor do commit, um título que é obrigatório e uma mensagem opcional. Certos projetos adicionam no campo da mensagem *Reviewed-by*, *Accepted-by* (ou *Acked-by*), *Tested-by*, *See-also*, *Fix(es)*, entre outros para denotar um workflow específico do projeto ou de alguma ferramenta utilizada por eles. Além disso, outros utilizam *Signed-off-by* como um Certificado de Origem do Desenvolvedor (Developer's Certificate of Origin) – o desenvolvedor ao adicionar *Signed-off-by* está de acordo com as regras de autoria do código.

Adicionando e modificando arquivos

Caso ainda não tenha criado um repositório, inicialize um e crie um arquivo README.md vazio:

```
$ git init
$ touch README.md
```

Ao rodar o comando **git status** você receberá a seguinte mensagem:

```
On branch master

Initial commit

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

nothing added to commit but untracked files present (use "git add" to track)
```

Opcionalmente altere o conteúdo do README.md e insira-o na staging area com o comando:

```
$ git add README.md
```

O comando **git add** também aceita *patterns* de arquivos e pode opcionalmente adicionar todos os arquivos do diretório de trabalho através da opção $--all$. Isso é útil quando inicializamos um repositório em uma pasta não-vazia e queremos adicionar todos os arquivos ao repositório.

Depois de adicionar o arquivo ao index, você pode rodar o comando **git status** para verificar o status do repositório:

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file: README.md
```

Chegou a hora de commitar [sic] as alterações feitas. O comando **git commit** pode ser utilizado de duas formas:

```
$ git commit [-s]
$ git commit [-s] -m "Mensagem"
```

Na primeira versão o git abre o editor de texto configurado para escrever e salvar a mensagem do commit, enquanto na segunda versão a mensagem é o parâmetro entre aspas, que será utilizado como título. A orientação geral é que o título não ultrapasse 50 caracteres e o texto 72 caracteres. O parâmetro opcional -s adiciona a mensagem *Signed-off-by* no final da mensagem.

Quando o commit é feito, o git emite a seguinte mensagem:

```
[master (root-commit) 7a4042f] Primeiro commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
```

Desfazendo Mudanças

Além de salvar mudanças às vezes é necessário desfazer (reverter) um commit. O commit nunca é apagado da história do repositório mas quando se reverte um commit, as alterações promovidas por ele são revertidas e um commit de reversão é adicionado.

O comando a ser utilizado é:

```
$ git revert <hash do commit>
```

O hash do commit pode ser obtido através do comando **git log**, que será explicado na próxima seção.

Removendo arquivos

As vezes é necessário remover um arquivo do repositório pois ele não é mais necessário. Para isso utilizamos o comando git rm que remove arquivos tanto do index quanto do diretório de trabalho.

```
$ git rm <arquivo> [--cached]
```

O parâmetro opcional – – *cached* remove os arquivos somente do index. Isso é útil quando adicionamos erroneamente algum arquivo via git add.

Renomeando e movendo arquivos

Para renomear ou modificar a localização de um arquivo ou diretório, basta realizar essa alteração e adicionar o novo nome com **git add**. O git é inteligente e percebe que o arquivo não é novo e é somente uma mudança de nome e/ou localização.

Existe também o comando **git mv** que move arquivos, diretórios e links simbólicos além de incluir as alterações no index. A sintaxe desse comando é:

```
$ git mv <origem> <destino>
```

Visualizando o histórico de Mudanças

Para checar todo o histórico de mudanças é necessário usar o comando **git log**, que mostra todos os commits da história do repositório. Alternativamente pode ser utilizado o utilitário **gitk** dentro do repositório para uma verificação visual da história.

Um exemplo de histórico pode ser visto a seguir:

```
commit acfa1fafcee46aa6eb8f89f87b644d5409391b80
Author: Fulano de Tal <Fulano.tal@ufms.br>
Date: Wed Mar 7 12:42:53 2018 -0400

    Revert "Primeiro_commit"

    This reverts commit 4b2aabd644747ec53dd4a7dfcfa75aed059b38cf.

commit 4b2aabd644747ec53dd4a7dfcfa75aed059b38cf
Author: Fulano de Tal <Fulano.tal@ufms.br>
Date: Wed Mar 7 12:42:48 2018 -0400

    Primeiro commit
```

Comandos úteis

Em todos os exemplos abaixo o < *hash* > pode ser substituído por HEAD para denotar o commit do topo da história do repositório, ou HEAD~1 para o penúltimo commit, e assim por diante mudando o numeral.

- **git diff**
  Obtém as diferenças entre dois commits.

  ```
  $ git diff <hash> <hash>
  ```

- **git stash**
  Salva as alterações num **stash** que permite serem aplicadas depois, deixando o diretório de trabalho limpo e sem alterações.

  ```
  $ git stash #Para salvar
  $ git stash list #Para listar os stashes
  $ git stash pop <stash> #Para aplicar o stash e retirar da lista de
      ↪ stashes
  ```

- **git tag**
  As vezes é útil marcar um determinado commit para futuras referências, como a versão ou um marco do projeto.

```
$ git tag <nome> #Para uma tag simples, sem mensagem nem dados de
    ↪ criação ou do autor
$ git tag -a [-m "mensagem"] #Para uma tag anotada, com dados de
    ↪ criação e do autor
```

- **git reset**
  Altera o index, o working directory e o commit HEAD. Pode ser utilizado de três principais formas:

  – Remove um ou mais arquivos da **staging area**:

```
$ git reset <arquivo>
```

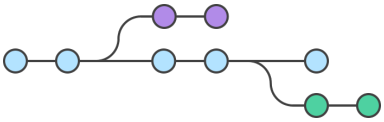
  – Retorna o HEAD para um commit anterior, sem alterar o index ou o working directory, mantendo as alterações como arquivos na staging area:

```
$ git reset --soft <hash>
```

  – Para retornar o HEAD para um commit anterior, alterando o index ou o working directory, descartando qualquer alteração:

```
$ git reset --hard <hash>
```

Trabalhando com Branches



O que é uma branch?

Uma **branch** nada mais é do que um ramo (branch em inglês) de desenvolvimento de um repositório. Mais de uma branch pode existir em cada repositório, mas existe uma padrão em todos repositório que é a **master**.

Novas branches podem ser criadas, mescladas e destruídas livremente para se adequar ao fluxo de desenvolvimento, e é nessas branches que os **commits** e **tags** criados ficarão contidos.

Criando uma Branch

Para criar outro ramo é necessário utilizar o comando **git branch**.

```
$ git branch <nome da branch> <commit inicial>
```

Caso o commit inicial não seja informado é utilizado por padrão o HEAD. Observe que a branch é somente criada, mas o diretório de trabalho não é modificado para ela. Para modificar o diretório de trabalho devemos fazer o **checkout** da branch com o comando **git checkout**.

```
$ git checkout <nome da branch>
```

Você pode também criar uma nova branch a partir do commit atual (provavelmente HEAD) e imediatamente mudar para ela com o comando:

```
$ git checkout -b <nome da branch>
```

Destruindo uma Branch

Para destruir uma branch é necessário que ela tenha sido incorporada a branch master, ou caso um repositório remoto tenha sido configurado, ao branch remoto.

```
$ git branch [-d |--delete] <nome da branch>
```

Para forçar a deleção, independente das limitações, você pode utilizar a opção $-D$ ao invés das outras opções acima.

Listando as branches

As branches existentes podem ser consultadas com o comando:

```
$ git branch [--list]
```

Esse comando lista todas as branches locais e marca com um asterisco a branch corrente.

```
  feature
* master
```

Mesclando Branches

Para mesclar uma branch a outra utilizamos o comando **git merge**, que mescla os commits do branch especificado para o branch corrente. Observe que os branches devem ter um ancestral comum a partir do qual eles divergiram para que o merge seja possível.

O comando deve ser utilizado da seguinte forma:

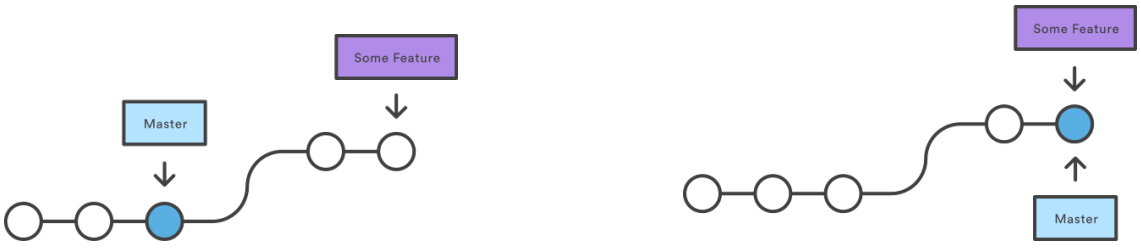
```
$ git merge <branch que deve ser juntado>
```

Cabe dizer que o git escolhe automaticamente diferentes estratégias para realizar o merge. Uma delas é o fast-forward merge e a outra é criando um merge commit, que serão explicados adiante.

Fast-forward merge

Caso as mudanças a serem incorporadas tenham HEAD como ancestral e nenhuma alteração foi feita na branch local, ou seja, a história é linear, não é necessário criar um merge commit. Neste caso os índices são atualizados para o topo da branch que está sendo mesclada, e o branch é **fast-forwarded** para o novo HEAD, daí o nome dessa estratégia. Esse é comportamento padrão do git e ocorre, por exemplo, quando obtemos alterações de um repositório remoto.

No exemplo da figura 2 o branch **Some Feature** foi derivado de **master** que não recebeu nenhuma outra alteração, logo o git realizou um merge **fast-forward**.



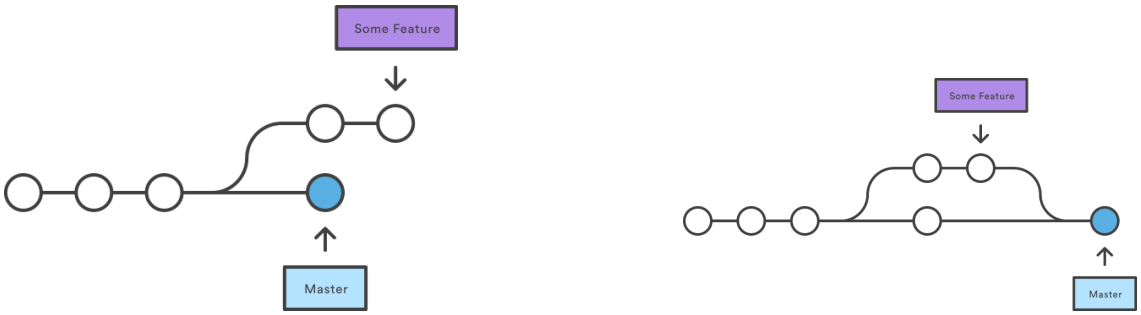
(a) Antes do merge (b) Depois do merge

Figura 2 – Merge Fast Forward

Merge Three-Way

Caso o git determine que um fast-forward não é possível, ele faz o merge através de um **merge commit**. Esse merge também é chamado de **three-way**, que é derivado do fato do git utilizar como referência, além dos HEADs dos branches a serem mesclados, o ponto da história onde os dois branches divergiram.

Essa situação ocorre, por exemplo, quando uma grande feature foi derivada do ramo master e precisa ser integrada de volta. Essa é a situação da figura 3, onde a alteração em **Some Feature** foi extensa e também houveram modificações no branch **master**.



(a) Antes do merge (b) Depois do merge

Figura 3 – Merge Three Way

Um merge commit se parece com o seguinte:

```
commit d3a60b90fcb1aee9dbd525bec4ac81790900c363 (HEAD -> master)
Merge: ff1034a cdda06b
Author: Fulano de Tal <Fulano.tal@ufms.br>
Date: Mon Mar 12 09:58:44 2018 -0400

    Merge branch 'feature'
```

Observe que o campo Merge relaciona os dois commits os quais o merge commit é filho e que a mensagem de commit é padrão "Merge branch '<nome do branch>'".

Merge conflict

Um merge conflict ocorre quando as mudanças a serem mescladas batem com as mudanças existentes na branch atual. Quando isso ocorre o git pausa o merge e oferece duas opções: Abortar o merge e desfazer qualquer alteração feita ou resolver o conflito. Existem ferramentas para resolver graficamente esse conflito.

Para abortar o merge, utilizamos o comando de merge com a opção $--abort$. Caso queiramos resolver, o git marca os arquivos com conflito da seguinte maneira:

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

O marcador <<<<<<< denota o começo do trecho com conflito e o marcador >>>>>>>, o fim do conflito. O trecho antes de ======= é o seu arquivo e o trecho depois é o arquivo que está tentando se dar merge.

Daí é só identificar o trecho a ser mantido, retirar os marcadores, realizar o commit e dar continuidade ao merge. Você pode tanto realizar um commit normalmente ou utilizar o comando de merge com a opção $--continue$.

Trabalhando com Remotos

O que são remotos?

Como o git é distribuído e cada usuário tem uma cópia completa do repositório, é necessário que os usuários coordenem o compartilhamento das alterações feitas.

Essa coordenação pode ser realizada através de um repositório central que é escolhido como referência para atualizações dos repositórios locais. Normalmente o repositório central utilizado é um serviço que *hosteia* repositórios git, como o Github, Bitbucket, Gitlab entre outros.

A cada repositório remoto de um repositório local damos o nome de **remoto**, além disso, como no caso das branches, existe um repositório remoto padrão de nome **origin**.

Adicionar um remoto a um repositório local é criar uma ligação entre esses repositórios. Essa ligação não é on-line mas é um modo de referenciar o repositório remoto para os comandos do git.

Observe que apesar do nome remoto, isso não implica que o repositório obrigatoriamente deva residir fora do computador local do usuário, podendo ser um outro repositório na mesma máquina.

Administrando os remotos

Configurando um repositório remoto

Toda a manipulação de remotos utiliza o comando **git remote**. Para adicionar um repositório remoto utilizamos o seguinte comando:

```
$ git remote add <nome> <URL>
```

Onde nome é o nome que queremos dar ao remoto e URL é a URL de acesso. Caso o repositório local tenha sido clonado a partir de um outro repositório, o remoto **origin** já estará configurado.

A URL de acesso a um repositório pode ter duas formas: via SSH e via HTTPS. As URLs tem a seguinte forma:

```
git@<servidor>:<usuário>/<caminho do repositório>.git # URL SSH
https://<servidor>:<usuário>/<caminho do repositório>.git # URL HTTPS
```

Listando repositórios remotos

Para verificar os remotos que estão configurados em um repositório utilizamos o seguinte comando:

```
$ git remote [-v]
```

O parâmetro opcional $-v$ mostra a lista de remotos com suas respectivas URLs.

Removendo repositórios remotos

Para remover um remoto configurado em um repositório utilizamos o seguinte comando:

```
$ git remote rm <nome>
```

Administrando branches remotas

Depois de configurarmos um ou mais remotos para um repositório, devemos configurar cada branch individualmente para fazer o rastreio do branch do remoto.

Isso pode ser feito de duas formas: configurando antes de tentar enviar uma alteração ou durante o envio dessas alterações.

A primeira maneira utiliza o seguinte comando:

```
$ git branch --set-upstream-to=<remoto>/<branch>
```

Caso o remoto não seja especificado, o remoto **origin** é assumido. Nas próximas sessões explicaremos como realizar o envio e recebimento das alterações e tags de um branch remoto.

Enviando mudanças para o servidor remoto

Para fazer o **push** das alterações locais para um remoto, usamos o seguinte comando:

```
$ git push <nome do remoto> <branch local>:<branch remoto> [--all] [--tags]
```

Esse comando é versátil e pode ser utilizado de várias formas. O parâmetro opcional $--all$ faz o push de todos os branches locais para os branches **de mesmo nome** no remoto. Enquanto o parâmetro $--tags$, faz o push de todas as tags para o remoto. No entanto $--tags$ e $--all$ não podem ser utilizados ao mesmo tempo.

Uma das formas de se utilizar o push é quando o remoto já foi configurado mas o branch local ainda não foi configurada para fazer o rastreio de um branch remoto. Para realizar o push, e rastrear aquela branch remota caso tenha sucesso, o comando **git push** é utilizado da seguinte forma:

```
$ git push -u <nome do remoto> <branch local>:<branch remoto>
```

Caso o push seja para um branch remoto de mesmo nome podemos omití-lo. Se omitirmos o nome do remoto, com exceção do comando listado anteriormente, o remoto **origin** é assumido. Similarmente, caso o branch remoto seja omitido é assumido o de mesmo nome. Observe que essas omissões podem ser acumuladas gerando simplesmente a sintaxe **git push**, que faz o push das alterações do branch local para o branch de mesmo nome no remoto **origin**.

Obtendo mudanças do servidor remoto

Existem dois modos de se obter as mudanças a partir de um servidor remoto: podemos obter todos os commits e branches do repositório remoto sem automaticamente integrar essas mudanças no repositório local, ou podemos obter essas alterações e automaticamente integrá-las à branch corrente.

A primeira delas envolve utilizar dois comandos do git: **git fetch** e **git merge**.

O comando **git fetch** obtém do servidor as mudanças desde o último fetch até o HEAD do remoto. O modo de se utilizar o comando é:

```
$ git fetch <remote> <branch>
```

Caso o comando seja executado sem argumentos, ele obtém as alterações do remoto para o branch configurados no local. Ele também pode ser executado com a opção $--all$ para obter todas as alterações de todas as branches ou ainda $--tags$ para obter todas as tags. Note que essas opções são mutualmente excludentes.

Depois de obtermos as alterações devemos mesclá-las na branch local utilizando o comando **git merge**. O argumento desse comando será o $<remoto>$ / $<branchremota>$.

Podemos também obter as alterações e incorporá-las utilizando o comando **git pull**. As mesmas opções do comando **git fetch** também valem para o **git pull**.

Workflows distribuídos

A natureza descentralizada do git implica no uso de um workflow de desenvolvimento para evitar os merge conflicts e auxiliar no processo de desenvolvimento. Nessa seção apontamos alguns workflows mais comuns de desenvolvimento.

Workflow centralizado

No worflow centralizado todos os desenvolvedores trabalham a partir do branch **master** do repositório central. Esse worflow pode gerar conflitos de merge, pois cada desenvolvedor pode trabalhar no mesmo arquivo e, mesmo sendo linhas diferentes, os conflitos ocorrem.

Para evitar esse cenário é necessário o uso de **rebasing** para corrigir o histórico de alterações. A utilização desse workflow também implica que o branch **master** conterá código não estável, e ainda que as revisões estáveis deverão ser marcadas através de **tags** ou mensagens de **commit** específicas.

Caso o rebase não seja utilizado e não ocorra conflito entre as mesmas linhas de um arquivo, o git automagicamente resolve o merge e cria merge commits no estilo:

```
commit 4ea03feb2a5ebb09457e62f17222b4907111150e
Merge: 65bcc1c c1fa44d
Author: fulano.tal@ufms.br <fulano.tal@ufms.br>
Date: Sat Sep 3 19:10:36 2016 -0400

    Merge branch 'master' of <repositório>

commit 65bcc1c61e2bdbd35466f96945213bd5a7603188
Merge: 09a751f 387b37b
Author: fulano.tal@ufms.br <fulano.tal@ufms.br>
Date: Sat Sep 3 19:09:53 2016 -0400

    Merge remote-tracking branch 'origin/master'
```

Mas apesar dessas ressalvas, esse é o workflow mais simples de se adotar, além de ser adequado para times extremamente pequenos que podem coordenar as alterações de modo que os conflitos de merge sejam escassos.

Workflow por branching

O workflow por branching também utiliza um repositório centralizado, mas cada desenvolver trabalha nas features que está desenvolvendo em branches separadas. É esperado que essas branches tenham escopo bem definido e também sejam enviadas para o repositório central.

A integração das features no branch **master** pode ser feita de várias maneiras que implicam métodos de aprovação diferentes.

- O **merge** dessas alterações pode depender da aprovação de somente uma pessoa, que tem acesso **read/write** no **master**.
- A aprovação pode ser feita por qualquer outro desenvolvedor após discussão e aprovação, por exemplo, de dois outros desenvolvedores.

No workflow por branching os desenvolvedores são encorajados a criarem **pull requests** como meio de compartilhar, promover discussão e submeter para aprovação superior as alterações que eles fizeram. Com o uso desse workflow, o branch **master** se torna mais estável, podendo ser integrado à ferramentas de integração contínua e *deployment* contínuo.

Gitflow

Gitflow é um famoso workflow publicado por Vincent Driessen no blog nvie. Esse é um workflow adequado para grandes projetos e sugere um formato de branching e de designação de nomes estrito.

Nesse workflow serão utilizados três principais branches: o **master**, que conterá código estável e terá as versões do release estável; o branch **develop**, onde todo o desenvolvimento ocorrerá e o branch **release** que conterá o código candidato a release. Outros branches de suporte também existem como os de features e de hotfix.

Todo o desenvolvimento é feito no branch **develop** e é de lá que os features branches derivam, além de ser o local onde são integrados. Já o branch de **release** é derivado de **develop** e nele só ocorre as últimas preparações para o release, como *bugfixes* e alterações nas informações de versão.

O branch **release** é incorporado de volta em **develop** e no **master**. Usualmente o merge do **release** no **master** é marcado com uma **tag**, para indicar a versão de produção.

Caso um *bug* crítico seja descoberto com o código em produção, para corrigí-lo criamos um branch a partir da tag do último release de **master** e a resolução do erro é feita lá. Essa branch é depois incorporada no **master** (gerando outro release), e também em **develop**.

Parte II

Git Intermediário

Alterando o histórico

O histórico de alterações de um repositório é um artefato imutável, mas existem duas situações em que é necessário alterar o histórico de um repositório.

Uma delas é quando temos que reescrever o último commit da história. A outra é durante o merge de uma outra branch para linearizar o histórico, onde neste caso utilizamos o **git rebase**.

Corrigindo o último commit

A necessidade de se alterar o último commit pode ser causada por diversos fatores, como esquecer de fazer o staging de um arquivo para o commit, notar algum erro na mensagem de commit, entre outros.

Para adicionar um arquivo ao commit, realizamos o staging do arquivo como visto anteriormente e utilizamos a opção $--amend$ do comando **git commit**. Ao utilizar esse comando temos a oportunidade de alterar a mensagem de commit utilizada anteriormente e podemos utilizar todas as opções do comando **git commit**.

É importante lembrar que o **amend** de um commit não "edita" o commit antigo, mas sim cria um novo commit com o conteúdo antigo e as modificações incluídas para o amend. Essa situação é exemplificada na Figura 4.

(a) História inicial (b) História depois da alteração

Figura 4 – Commit amend

Rebasing

O **git rebase** é um comando poderoso que permite integrar um branch a outro, editar commits passados e alterar sua ordem. Ele é utilizado para se manter um histórico linear de desenvolvimento que não foi necessariamente linear.

Quando fazemos um rebase, todos os commits do branch corrente são salvos em uma área separada e o branch é retornado para o commit ancestral comum entre ele e o branch contra o qual estamos fazendo o rebase. O git então reaplica os commit do branch no topo desse commit onde as histórias divergem, gerando um histórico linear e possibilitando um merge fast-forward desse branch no outro branch.

Rebase simples

O rebase mais simples é aquele que mescla um feature branch, que foi derivado de **master**, de volta a ele. Neste caso o processo ocorre como explicitado na sessão anterior.

O comando rebase aceita diversos argumentos, mas no exemplo abaixo **master** é o branch contra o qual estamos fazendo o **rebasing** e **feature** é o branch derivado daquele branch.

A sintaxe do comando é:

```
$ git rebase master feature
# ou
$ git checkout feature
$ git rebase master
```

Depois desse comando o branch **feature** consegue ser mesclado ao branch **master** através de um merge *fast-forward*.

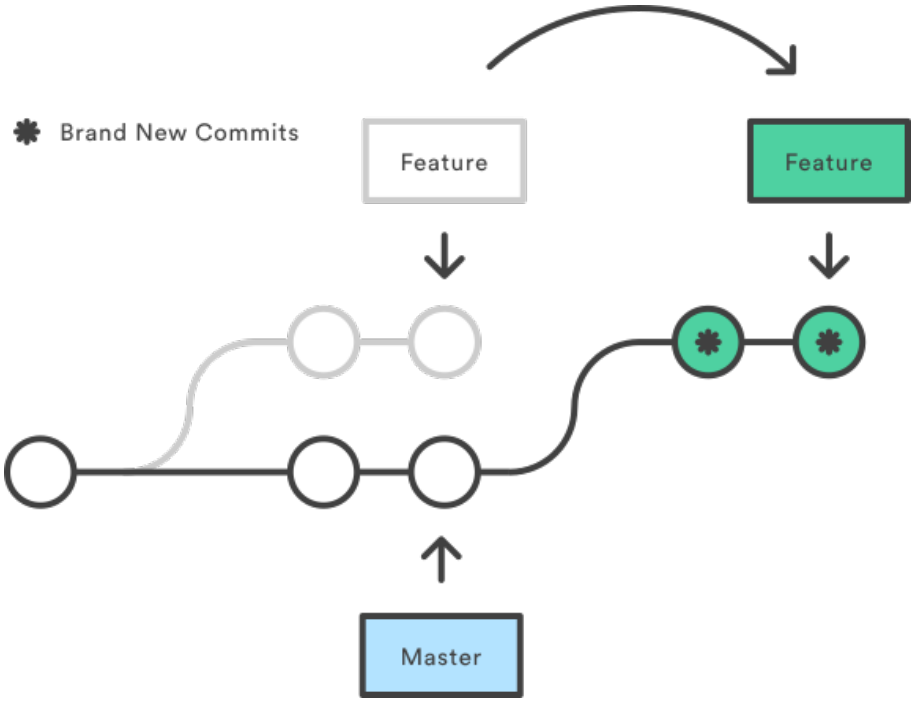


Figura 5 – Rebase simples

Rebase interativo

O rebase interativo é o mesmo rebase simples, mas ele permite você verificar e escolher a ordem e se os commits aparecerão na história final.

A sintaxe do comando é:

```
$ git rebase -i master feature
# ou
$ git checkout feature
$ git rebase -i master
```

Depois de rodar esse comando o git mostra no editor padrão uma lista com os commits, um shortlog da mensagem e uma letra/comando para fazer ou não a escolha.

```
#
# 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
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

As opções são auto-explicativas, mas uma coisa importante para se ter em mente é que caso um commit seja apagado ele será ignorado e não será incluído na história final. Ainda, se outros commits dependerem da alteração desse commit que foi omitido, o rebase irá falhar no meio do caminho.

Recuperando um rebase que falhou

As vezes um rebase pode falhar no meio do processo por causa de conflitos. A recuperação se dá de duas maneiras: podemos abortar o rebase ou corrigir o erro.

Para abortar utilizamos o comando **git rebase** com a opção $--abort$. Essa opção altera o diretório de trabalho e retorna-o para o estado anterior ao rebase.

Ao encontrar um erro, o **git rebase** pausa o rebase e aguarda a resolução. Essa resolução é feita como a de um conflito de merge nos capítulos anteriores. Após, o rebase pode ser retomado com **git rebase** $--continue$. Caso se deseje omitir o commit que causou o conflito, o comando **git rebase** é utilizado com a opção $--skip$.

Filtrando o branch

Outro modo de se alterar o histórico é reescreve-lo de modo a remover algum arquivo que não foi utilizado mas que está presente em todo commit. Para isso utilizamos o comando **git filter-branch**. Esse comando é bastante poderoso, podendo alterar mensagens de commit, autores, entre outros, mas o uso que mostraremos aqui é o caso mais simples.

```
$git filter-branch --index-filter 'git rm --cached --ignore-unmatch <file>'
```

No final desse comando é possível também declarar o intervalo que o **filter-branch** deve trabalhar. Isso é feito utilizando a seguinte notação HASH_INITIAL..HASH_FINAL. Caso nada seja passado o comando assume todo o histórico. É importante dizer que o comando não inclui o HASH_INITIAL mas inclui o HASH_FINAL.

Esse comando trabalha sobre o index e altera cada commit, removendo o arquivo e refazendo o commit. Isso gera uma história paralela a original mas que não contém o arquivo.

É importante ressaltar que, como esses comandos alteram a história de um repositório, não é recomendado utilizá-los para alterar um histórico que outras pessoas estejam utilizando.

Assinando seu Trabalho

O Git suporta o PGP/GPG para realizar a assinatura de commits e tags. Com a utilização de assinatura, podemos garantir que as alterações realmente provém dos usuários autorizados a realizá-las. É importante estabelecer um **workflow** claro do que será assinado e quem assinará para evitar responsabilizações indevidas.

Na maioria dos serviços git é possível cadastrar a chave pública dos autores autorizados e a interface web exibirá um simbolo de verificado quando o commit e/ou tag for assinado com uma das chaves autorizado, trazendo para o usuário mais confiança no repositório.

Criando chaves

Para realizar a assinatura, o Git utiliza a ferramenta de linha de comando GPG e é necessário tê-la instalado previamente.

Para verificar se existe alguma chave criada, utilize o seguinte comando:

```
$ gpg --list-secret-keys --keyid-format LONG
```

Caso não exista nenhuma chave, podemos criar uma nova chave com o comando:

```
$ gpg --gen-key
```

O programa perguntará os dados pessoais para criar a chave. É importante cadastrar uma senha forte e difícil de ser adivinhada, além de colocar o endereço de e-mail associado a conta do serviço.

Quando o primeiro comando for executado novamente ele mostrará o **ID** da chave criada. Guarde-o pois precisaremos dele adiante.

Configurando a assinatura

Antes de assinar qualquer coisa é necessário configurar o git para que ele saiba qual chave deve utilizar para assinatura. Fazemos isso com o comando:

```
$ git config --global user.signingkey ID
```

Além disso é necessário exportar a chave pública para que o serviço Git saiba quais as chaves autorizadas. A chave pode ser exportada com o comando:

```
$ gpg --armor --export ID
```

Os **IDs** nos dois últimos comandos se refere ao ID da chave obtido anteriormente.

Assinando e verificando as alterações

Com tudo configurado podemos começar a assinar nosso trabalho. Para assinar um commit ou uma tag, adicionamos o parâmetro $-S$ ao respectivo comando. A ferramenta GPG irá solicitar a senha da chave e assinará o trabalho.

Para verificar uma tag assinada, utilizamos o comando a seguir substituindo TAG pela tag que queremos verificar:

```
$ git tag -v TAG
```

Já para verificar um commit utilizamos a opção $--show-signature$ do comando **git log**.

Assinando um merge commit

É possível assinar um *merge commit* passando a mesma opção $-S$ para o comando **git merge**. Além disso é possível realizar a verificação de todos os commits **branch** antes de mesclá-lo através da opção $--verify-signatures$. Caso o **branch** tenha assinaturas inválidas o merge não continuará.

Referências

Este texto foi composto em Utopia, de Robert Slimbach, através do pacote fournier.