

PET - Computação

*Manual de Git*

© 2018 PET - Computação & FACOM - UFMS

Essa publicação está licenciada de acordo com os termos da Creative Commons – Attribution – NonDerivative (CC-BY-ND)



A licença pode ser consultada em <<https://creativecommons.org/licenses/by-nd/4.0/>>  
PET - Computação <[www.petcomputacao.ufms.br](http://www.petcomputacao.ufms.br)>

# Sumário

<b>Introdução</b>	<b>3</b>
Sistema Distribuido de Controle de Versão	3
O que é o Git?	3
Usos do Git	3
Git NÃO é GITHUB!	3
<b>Instalação e Configuração</b>	<b>4</b>
Instalação no Windows	4
Instalação no Linux	4
Compilar a partir do código fonte	4
Configuração Inicial	5
<b>I Git Básico</b>	<b>6</b>
<b>Git trees</b>	<b>7</b>
<b>Trabalhando com repositórios</b>	<b>8</b>
O que é um repositório?	8
Criando um repositório	8
Adicionando um repositório	8
Clonando um repositório	8
<b>Alterando o código</b>	<b>9</b>
O que é um commit?	9
Adicionando e modificando arquivos	9
Desfazendo Mudanças	10
Removendo arquivos	10
Renomeando e movendo arquivos	11
Visualizando o histórico de Mudanças	11
Comandos úteis	11
<b>Trabalhando com Branches</b>	<b>13</b>
O que é uma branch?	13
Criando um branch	13
Destruindo um branch	13
Listando as branches	13
Mesclando branches	14
Fast-forward merge	14
Merge commit	14
Merge conflict	14
<b>Trabalhando com Remotos</b>	<b>16</b>
O que são remotos?	16
Administrando os remotos	16
Configurando um repositório remoto	16
Listando repositórios remotos	16
Removendo repositórios remotos	17
Administrando branches remotas	17
Enviando mudanças para o servidor remoto	17
Obtendo mudanças do servidor remoto	18
<b>Workflows distribuídos</b>	<b>19</b>
Workflow centralizado	19
Workflow por branching	19

Gitflow . . . . .	20
<b>II Git Intermediário</b>	<b>21</b>
<b>Alterando o histórico</b> . . . . .	<b>22</b>
Corrigindo o último commit . . . . .	22
Rebasing . . . . .	22
Rebase simples . . . . .	22
Rebase interativo . . . . .	23
Recuperando um rebase que falhou . . . . .	23
Filtrando o branch . . . . .	24
<b>Assinando seu Trabalho</b> . . . . .	<b>25</b>
 <b>Referências</b> . . . . .	 <b>26</b>

## Introdução

### Sistema Distribuído de Controle de Versão

Um sistema distribuído de controle de versão (DVCS, em inglês) é um sistema que permite a um time de desenvolvimento a 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 repositório local é também um repositório que contém todo o histórico de mudanças.

### O que é o Git?

O Git é um DVCS, ele foi 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 de performance e uso ele criou o git.

### Usos do Git

O git é utilizado para controle de mudanças em uma base de código – um repositório. O funcionamento básico do mesmo é criar ou copiar um repositório, fazer alterações no mesmo, gravar as alterações, receber alguma alteração feita, e enviar as próprias alterações. Ele permite ainda, verificar quem fez alguma alteração, visualizar todo o histórico além de selecionar que mudanças fazem parte do histórico principal.

### Git NÃO é GITHUB!

Vale ressaltar que git é diferente do Github. Git é uma ferramenta (e também um protocolo), já o github, bitbucket, gitlab, etc. são serviços que disponibilizam uma interface sobre o git “normal”, de modo que o uso no dia-a-dia do git é facilitado.

**Eles não são “um git”, são serviços!**

## Instalação e Configuração

### Instalação no Windows

Para o windows, há diversas opções para instalação desde a versão em linha de comando do git até programas GUI e/ou atalhos no menu de contexto. A versão em linha de comando é o git propriamente dito, que 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 SourceTree <<https://www.sourcetreeapp.com/>> entre outros.

### Instalação no Linux

O git vem instalado por padrão na maioria das distribuições GNU/Linux. Caso não esteja instalado na sua distribuição é só instalá-lo a partir do gerenciador de pacotes da mesma ou compilar a partir dos fontes (abaixo).

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 as seguintes dependências adicionais são requeridas:

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

Para as distribuições baseadas em Federal/RHEL o seguinte passo é também 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 guarda os arquivos durante o processo de desenvolvimento. Ele guarda em três “árvores”, o index (ou também staging area), o diretório de trabalho (Working Directory – WD) e o repositório local.

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. Podendo ser então mandadas para o servidor remoto.

## 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 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 importa-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 diretório 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 você queria clonar a biblioteca Git do Ruby chamada Grit, você pode fazê-lo da seguinte forma:

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

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

## Alterando o código

### O que é um commit?

Um **commit** é o objeto básico quando se trabalha com o git. Ele guarda a história do repositório as alterações feitas no diretório de trabalho associando essa mudança ao autor e as 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. Já 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 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 as 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 pelo mesmo 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 e também já inclui 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 na história do repositório. Alternativamente pode ser utilizado o utilitário **gitk** dentro do repositório para uma verificação visual da história do repositório.

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 4b2aab6d644747ec53dd4a7dfcfa75aed059b38cf.
```

```
commit 4b2aab6d644747ec53dd4a7dfcfa75aed059b38cf
```

```
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 podem ser 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 branch padrão em todos repositório que é a branch **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, por padrão é utilizado o HEAD. Observe que a branch é criada somente, 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 a diante.

### 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, então 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.

### Merge commit

Caso o git determine que um fast-forward não é possível, o git faz então um merge commit, que contém como pais os topos da história a ser mesclada e a história da branch. Isso ocorre, por exemplo, quando uma grande feature foi derivada do ramo master e precisa ser integrada de volta.

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 do qual 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 o conflito.

Para abortar o merge, utilizamos o comando de merge com a opção – *abort*. Caso queiramos resolver o conflito 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 necessitem coordenar 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 um 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 também configurar cada branch individualmente para fazer o rastreo 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 explicamos 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 um comando 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 a branch local ainda não foi configurada para fazer o rastreo de uma branch remota. Para realizar o push, e caso tenha sucesso, rastrear aquela branch remota, o comando **git push** é utilizado da seguinte forma:

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

Caso o push seja para uma branch remota de mesmo nome, podemos omitir o branch remoto. 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 branch remoto 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 e para a branch configurados na branch local. Ele também pode ser executado com a opção `--all` para obter todas as alterações de todas as branches e também `--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 workflow centralizado todos os desenvolvedores trabalham a partir do branch **master** do repositório central. Esse workflow 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 que as revisões estáveis deverão ser marcadas através de **tags** ou mensagens de **commit** específicas.

Caso o rebase não seja utilizado e também não aja conflito entre as mesmas linhas de um arquivo, o git automaticamente 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 e é 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 p.ex. de dois outros desenvolvedores.

No workflow por branching os desenvolvedores são encorajados a 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 a 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 conterà código estável e terá as versões do release estável, o branch **develop**, onde todo o desenvolvimento ocorrerá e o branch **release** que conterà o código candidato a release. Outros branches de suporte também existem: Os branches de features e de hotfix.

Todo o desenvolvimento é feito no branch **develop** e é de lá que os features branches derivam e é pra lá que eles 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, etc.

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 dela é durante o merge de uma outra branch para linearizar o histórico, neste caso utilizamos o **git rebase**.

## Corrigindo o último commit

A necessidade de se alterar o último commit pode ser causada por diversos fatores, 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.

## Rebasing

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

Quando fazemos um rebase, todos os commits do branch corrente são salvos em uma área separada, 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*.

## 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 resolução de um conflito de merge nos capítulos anteriores. Após a resolução desse conflito, 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 os utilizar para alterar um histórico que outras pessoas podem ter estar utilizando.

## **Assinando seu Trabalho**

O git suporta o PGP/GPG para realizar a assinatura de commits e tags

## **Referências**

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