

# Introdução a Git

Vitor Greati<sup>1</sup>   Felipe Fernandes<sup>1</sup>

<sup>1</sup>IEEE Computer Society  
Universidade Federal do Rio Grande do Norte

2019

# Seção 1

## Introdução

# Motivações

Duas necessidades muito comuns na vida de quem trabalha com arquivos:

## Controle de versões

Geralmente é desejável conhecer as alterações realizadas nos arquivos ao longo do tempo, para que estados passados possam ser revisitados caso necessário.

## Desenvolvimento colaborativo

Quando muitas pessoas trabalham sobre o(s) mesmo(s) arquivo(s), as diferentes alterações realizadas por cada uma devem ser integradas.

# Motivações

Duas necessidades muito comuns na vida de quem trabalha com arquivos:

## Controle de versões

Geralmente é desejável conhecer as alterações realizadas nos arquivos ao longo do tempo, para que estados passados possam ser revisitados caso necessário.

## Desenvolvimento colaborativo

Quando muitas pessoas trabalham sobre o(s) mesmo(s) arquivo(s), as diferentes alterações realizadas por cada uma devem ser integradas.

# Motivações

Duas necessidades muito comuns na vida de quem trabalha com arquivos:

## Controle de versões

Geralmente é desejável conhecer as alterações realizadas nos arquivos ao longo do tempo, para que estados passados possam ser revisitados caso necessário.

## Desenvolvimento colaborativo

Quando muitas pessoas trabalham sobre o(s) mesmo(s) arquivo(s), as diferentes alterações realizadas por cada uma devem ser integradas.

# O que é?



Git é um sistema de controle de versão distribuído (**DVCS**<sup>1</sup>) *open source*, gratuito, simples e eficiente. É útil não apenas para programadores, mas para todos que desejem versionar arquivos.

Foi criado por **Linus Torvalds** para controlar as versões do kernel Linux.

---

<sup>1</sup>Em oposição aos centralizados (VCS).

# Instalação

- **Windows**

Acessar <http://git-scm.com> e baixar o instalador da versão mais recente.

- **Mac**

No mesmo site, encontra-se o .dmg.

- **Linux**

Possivelmente já instalado. Do contrário, basta, num terminal, instalar o pacote `git` pelo gerenciador de pacotes disponível.

## Seção 2

### Conceitos fundamentais



# O diretório de trabalho

É um diretório no qual foi inicializado um **repositório Git**. Isso significa que possui, em seu interior, a pasta `.git`, na qual estão todas as informações referentes ao controle de versões e à configuração do Git. Geralmente, corresponde à **pasta do projeto** em que se está trabalhando.

Para criar um repositório, basta executar as seguintes linhas num terminal:

```
mkdir meu_projeto // caso o projeto não exista
cd meu_projeto
git init
```

# O diretório de trabalho

É um diretório no qual foi inicializado um **repositório Git**. Isso significa que possui, em seu interior, a pasta `.git`, na qual estão todas a informações referentes ao controle de versões e à configuração do Git. Geralmente, corresponde à **pasta do projeto** em que se está trabalhando.

Para criar um repositório, basta executar as seguintes linhas num terminal:

```
mkdir meu_projeto // caso o projeto não exista
cd meu_projeto
git init
```

# Ciclo de vida dos arquivos I

O diretório de trabalho conterá os arquivos do projeto, que poderão estar, aos olhos do Git, em um de **quatro estados principais**:

- **Untracked** O arquivo acabou de ser criado ou copiado ao diretório. Nesse instante, o Git não se preocupará em observar suas alterações. Também pode indicar quando se executou o comando `git rm` [11] ou o `git reset HEAD` [11] sobre o arquivo.
- **Unmodified** Ocorre quando se executa o comando `git add` pela primeira vez sobre o arquivo ou o `git commit` [10]. Indica que não há registro de modificação do arquivo.

## Ciclo de vida dos arquivos II

- **Modified** O arquivo recebe este estado quando é editado. Para as alterações serem incluídas na próxima versão, será necessário executar o **git add** sobre ele, levando-o ao estado *staged*.
- **Staged** O arquivo está no **index** ou **stage** do Git, significando que suas alterações serão incluídas na próxima versão ou *commit*.

# Os *commits*

- Um *commit* pode ser visto como um *snapshot*, ou um container de arquivos que será armazenado como uma versão.
- Todo *commit* carrega informações do nome do autor, do e-mail e da data de realização.
- Todo *commit* é identificado unicamente por um hash (código) de 40 caracteres. Isso pode ser observado executando-se `git log`.
- Todo *commit* necessita de uma mensagem explicando o que foi alterado no repositório. É **obrigatória**.
- Vários arquivos podem ser adicionados ao próximo *commit* executando-se `git add -all (-A)` ou `git add *.cpp`, por exemplo, para adicionar todos terminados em `.cpp`.

## Os *commits*

- Um *commit* pode ser visto como um *snapshot*, ou um container de arquivos que será armazenado como uma versão.
- Todo *commit* carrega informações do nome do autor, do e-mail e da data de realização.
- Todo *commit* é identificado unicamente por um hash (código) de 40 caracteres. Isso pode ser observado executando-se `git log`.
- Todo *commit* necessita de uma mensagem explicando o que foi alterado no repositório. É **obrigatória**.
- Vários arquivos podem ser adicionados ao próximo *commit* executando-se `git add -all` (`-A`) ou `git add *.cpp`, por exemplo, para adicionar todos terminados em `.cpp`.

## Os *commits*

- Um *commit* pode ser visto como um *snapshot*, ou um container de arquivos que será armazenado como uma versão.
- Todo *commit* carrega informações do nome do autor, do e-mail e da data de realização.
- Todo *commit* é identificado unicamente por um hash (código) de 40 caracteres. Isso pode ser observado executando-se **git log**.
- Todo *commit* necessita de uma mensagem explicando o que foi alterado no repositório. É **obrigatória**.
- Vários arquivos podem ser adicionados ao próximo *commit* executando-se **git add -all (-A)** ou **git add \*.cpp**, por exemplo, para adicionar todos terminados em `.cpp`.

## Os *commits*

- Um *commit* pode ser visto como um *snapshot*, ou um container de arquivos que será armazenado como uma versão.
- Todo *commit* carrega informações do nome do autor, do e-mail e da data de realização.
- Todo *commit* é identificado unicamente por um hash (código) de 40 caracteres. Isso pode ser observado executando-se **git log**.
- Todo *commit* necessita de uma mensagem explicando o que foi alterado no repositório. É **obrigatória**.
- Vários arquivos podem ser adicionados ao próximo *commit* executando-se **git add -all (-A)** ou **git add \*.cpp**, por exemplo, para adicionar todos terminados em `.cpp`.



## Os *commits*

- Um *commit* pode ser visto como um *snapshot*, ou um container de arquivos que será armazenado como uma versão.
- Todo *commit* carrega informações do nome do autor, do e-mail e da data de realização.
- Todo *commit* é identificado unicamente por um hash (código) de 40 caracteres. Isso pode ser observado executando-se **git log**.
- Todo *commit* necessita de uma mensagem explicando o que foi alterado no repositório. É **obrigatória**.
- Vários arquivos podem ser adicionados ao próximo *commit* executando-se **git add -all (-A)** ou **git add \*.cpp**, por exemplo, para adicionar todos terminados em `.cpp`.

# Dinâmica dos *commits* I

Pode-se enxergar o fluxo de *commits* no Git como uma metáfora temporal, ou seja, sob a ótica do passado, do presente e do futuro:

- **Passado** Aqui se encontram todas as versões terminadas (*commits* executados). Nessa região, está definido o chamado **ponteiro HEAD**, que aponta para o último *commit* realizado.
- **Presente** Aqui se distinguem duas áreas: **a árvore ou diretório de trabalho** e o **INDEX**. No primeiro, estão os arquivos *unstaged* que foram modificados, adicionados ou removidos no presente, ou que estão intocados.

Para que essas mudanças possam ser registradas no próximo *commit*, esses arquivos precisam ser adicionados ao INDEX, por meio do **git add**.

## Dinâmica dos *commits* II

### Como retirar do INDEX?

Se um arquivo adicionado ao INDEX precisa ser removido dele, basta utilizar o comando `git rm --cached arquivo`

Estando todas as alterações desejadas registradas no INDEX, basta executar o `git commit -m "Mensagem"`, e a nova versão será criada, passando a compor o passado, sendo recuperável a partir do ponteiro HEAD.

Após o *commit*, o INDEX torna-se vazio e o diretório de trabalho passa a conter todos os arquivos novamente, agora com as modificações realizadas. Todos os arquivos passam a ser *unmodified*.

## Dinâmica dos *commits* III

### Esquecendo o presente

É possível esquecer todas as alterações realizadas e fazer com que o presente seja uma cópia do último *commit* realizado. Para isso, executa-se o **git reset --hard HEAD**, o que traz todos os arquivos do *commit* passado para a árvore de trabalho do presente.

- **Futuro** O Git nada tem a ver com isto...

# .gitignore

Muitas vezes, deseja-se ignorar alguns arquivos. Em um projeto C++, por exemplo, se há uma pasta `bin/` que guarda os executáveis, é interessante não os incluir ao controle de versão, pois geralmente devem ser produzidos novamente por compilação.

Para ignorar esses arquivos, cria-se um novo arquivo na raiz do repositório, chamado `.gitignore`<sup>2</sup>. Um exemplo seria:

```
#Um comentário qualquer  
bin/  
.tmp
```

---

<sup>2</sup>Para forçar temporariamente a inclusão: `git add -f arquivo`

## Tags: marcando *commits* especiais

Alguns *commits* determinam metas alcançadas, outros representam um *release* de um software. Em casos assim, em que se deseja destacar uma versão, são usadas tags, criadas da seguinte forma:

```
git tag -a NomeDaTag -m "Descrição da tag"
```

Salienta-se que a mensagem, novamente, é **obrigatória**.

## Seção 3

# Branching

## Cada macaco no seu galho

Uma **branch** é, em termos simplificados, uma sequência de *commits* (podendo ser nula), incluindo o que está para ser realizado no presente. Em termos ainda mais simples, é um caminho que o repositório pode tomar, em termos de *commits*.

### Sempre há uma branch!

Todo repositório tem uma *branch*. Se nenhuma foi criada explicitamente, todos os commits estão sendo adicionados à *branch master*.



# Principais comandos

- Consultar as branches do repositório  
**git branch**
- Criar uma branch  
**git branch** nomeBranch
- Mover-se para outra branch  
**git checkout** nomeBranch

## Por trás das cenas

- O diretório de trabalho é sempre um espelho da branch atual.
- Toda branch possui um ponteiro que aponta para o último *commit* realizado nela, o qual, de fato, representa o estado da branch.

## Mesclando branches I

Ao terminar os trabalhos em uma *branch*, será necessário mesclá-la com a principal, para que as alterações entrem na versão oficial do sistema (por padrão, a *branch* master). Isso é feito pela sequência:

```
git checkout branchReceptora  
git merge outraBranch
```

Com esse procedimento, os ponteiros da `outraBranch`, da `branchReceptora` e o HEAD apontam todos para o mesmo *commit*.

Perceba que não necessariamente a master é a receptora. É possível mesclar *branches* quaisquer.

O merge não é o fim de uma branch. Muitas vezes, recomenda-se fazer o merge mesmo que os objetivos daquela *branch* não tenham sido todos cumpridos.

## Lidando com alterações nas branches

Antes de unir duas *branches*, seria interessante observar quais as diferenças entre elas, para prever possíveis conflitos. Há duas principais ferramentas que a linha de comando do Git oferece:

- **git diff** `branchReceptora..outraBranch`  
Exibe o que há na `branchReceptora` que não está na `outraBranch`.
- **git log** `outraBranch..branchReceptora`  
Exibe como a `outraBranch` difere da `branchReceptora` quanto aos *commits*.
- **git shortlog** `outraBranch..branchReceptora`  
Mesmo funcionamento da anterior, porém exibe resultados mais resumidos.

## Lidando com alterações nas branches

Antes de unir duas *branches*, seria interessante observar quais as diferenças entre elas, para prever possíveis conflitos. Há duas principais ferramentas que a linha de comando do Git oferece:

- **git diff** `branchReceptora..outraBranch`  
Exibe o que há na `branchReceptora` que não está na `outraBranch`.
- **git log** `outraBranch..branchReceptora`  
Exibe como a `outraBranch` difere da `branchReceptora` quanto aos *commits*.
- **git shortlog** `outraBranch..branchReceptora`  
Mesmo funcionamento da anterior, porém exibe resultados mais resumidos.

## Lidando com alterações nas branches

Antes de unir duas *branches*, seria interessante observar quais as diferenças entre elas, para prever possíveis conflitos. Há duas principais ferramentas que a linha de comando do Git oferece:

- **git diff** `branchReceptora..outraBranch`  
Exibe o que há na `branchReceptora` que não está na `outraBranch`.
- **git log** `outraBranch..branchReceptora`  
Exibe como a `outraBranch` difere da `branchReceptora` quanto aos *commits*.
- **git shortlog** `outraBranch..branchReceptora`  
Mesmo funcionamento da anterior, porém exibe resultados mais resumidos.

# Resolvendo conflitos I

Quando se trabalha com múltiplas *branches*, arquivos podem ser alterados mais de uma vez nas mesmas linhas, e, no momento do *merge*, o Git não consegue tratar essa situação sem a ajuda do usuário.

Para evitar ao máximo essas situações, é importante **afastar-se** de duas atitudes quando do uso constante do Git:

- Editar constantemente os mesmos arquivos em *branches* diferentes.
- Demorar muito para mesclar *branches*.

Há dois tipos notáveis de conflito: por **modificação** e por **remoção**.

## Resolvendo conflitos II

- **Por modificação** Ocorre quando as mesmas linhas de um arquivo são modificadas em *branches* distintas. Nessa situação, após o *merge*, o Git escreverá, nos arquivos conflitantes, indicações da localização do conflito. Elas estarão entre <<<<<<< >>>>>>>, e os blocos em conflito serão separados por =====. Para resolver, basta editar o arquivo, substituindo essas indicações pelo conteúdo que de fato deve estar no arquivo, e realizar o *merge* novamente.
- **Por remoção** Ocorre quando um arquivo é editado em uma *branch* e excluído na outra. O Git precisa saber se se deseja excluir ou manter o arquivo editado. Existem duas formas de resolvê-lo:
  - **Mantendo o arquivo** Basta ir à *branch* onde o arquivo fora removido e adicioná-lo. Por fim, o *commit* realizará o *merge* das *branches*.



## Resolvendo conflitos III

- **Excluindo o arquivo** Remover o arquivo da *branch* onde ele fora modificado (`git rm arquivo`), concordando, portanto, com a sua exclusão. No fim, basta um *commit* para marcar a resolução do conflito.

# Stashing

O Git pode não aceitar a mudança de uma *branch* para outra, quando há modificações não comitadas nos arquivos da atual, a menos que se realize o processo de **stashing**. Basicamente, ao se invocar o comando **git stash**, todas as modificações são empacotadas e guardadas para um próximo momento.

Para listar todos os *stashes* criados, basta usar **git stash list**.

Tendo concluído os trabalhos na outra *branch*, retorna-se à inicial e recupera-se o *stash* criado, usando **git stash apply**.

# Stashing

O Git pode não aceitar a mudança de uma *branch* para outra, quando há modificações não comitadas nos arquivos da atual, a menos que se realize o processo de **stashing**. Basicamente, ao se invocar o comando `git stash`, todas as modificações são empacotadas e guardadas para um próximo momento.

Para listar todos os *stashes* criados, basta usar `git stash list`.

Tendo concluído os trabalhos na outra *branch*, retorna-se à inicial e recupera-se o *stash* criado, usando `git stash apply`.

## Stashing

O Git pode não aceitar a mudança de uma *branch* para outra, quando há modificações não comitadas nos arquivos da atual, a menos que se realize o processo de **stashing**. Basicamente, ao se invocar o comando **git stash**, todas as modificações são empacotadas e guardadas para um próximo momento.

Para listar todos os *stashes* criados, basta usar **git stash list**.

Tendo concluído os trabalhos na outra *branch*, retorna-se à inicial e recupera-se o *stash* criado, usando **git stash apply**.

## Seção 4

### Trabalhando remotamente

## Repositório remoto

Um repositório Git remoto é uma cópia do repositório local em um servidor remoto. Por ser um DVCS, o Git permite que se trabalhe com vários repositórios remotos.

Existem diversos servidores remotos para o Git, sendo os mais famosos:

- **GitHub**<sup>3</sup> O mais utilizado atualmente; na versão gratuita, permite a criação de repositórios públicos e privados (aos estudantes, verificar o *GitHub Student Pack*).
- **BitBucket**<sup>4</sup> Interessante para quem precisa trabalhar com repositórios privados.
- **GitLab**<sup>5</sup>: repositórios privados, com gerenciamento de times, registro de containeres, integração com Kubernetes e próprio CI/CD.

---

<sup>3</sup><http://github.com>

<sup>4</sup><http://bitbucket.org>

<sup>5</sup><http://gitlab.com>

# Criando e clonando um repositório remoto

Os dois servidores apresentados possuem uma interface simples para a criação de um novo repositório.

Todo repositório possui uma *url* associada. Uma vez com ela em mãos, uma cópia local pode ser criada (claro, dadas as devidas permissões de acesso ao repositório remoto) usando-se o comando:

```
git clone http://url
```

Feito isso, o repositório estará no diretório atual.

# Associando um repositório local a um remoto

Dado que se tenha um repositório local não associado a um remoto, pode-se proceder da seguinte maneira para associá-lo:

1. Criar um repositório remoto no servidor desejado.
2. Criar o nome do repositório remoto localmente, via **git remote add** [nome remoto] [url]
3. Realizar um *commit* e, em seguida, um *push* usando o nome remoto criado.



# Aplicando as alterações locais no repositório remoto I

“Já deu push aí?”

Utiliza-se o comando **push**. A forma abaixo garante também que *branches* criadas e modificadas localmente apareçam também no servidor remoto:

```
git push [nome remoto] [branch]
```

O nome remoto padrão do repositório é **origin**, e, como se sabe, o da *branch* é **master**.

# Aplicando as alterações locais no repositório remoto II

“Já deu push aí?”

Caso se deseje comitar em outro servidor remoto (por exemplo, ter o repositório no GitHub e no BitBucket), será preciso registrar um novo nome de repositório remoto, pelo comando

```
git remote add [nome remoto] [url]
```

Para ver os nomes remotos disponíveis, basta executar

```
git remote -v
```

Para checar as configurações remotas do repositório remoto, basta

```
git remote show [nome remoto]
```

# Aplicando as alterações remotas ao repositório local

“Eu fiz sim! Você ainda não deu pull!”

Primeiro, é interessante trazer as informações sobre as diferenças entre os dois repositórios. Faz-se isso pelo comando:

```
git fetch
```

Após isso, **git status** mostrará tais diferenças em termos de *commits*. O próximo passo será efetivamente mesclar os *commits* do repositório remoto com os do local, através do comando

```
git pull [nome remoto] [branch]
```

Um *pull* é, de fato, uma operação de *fetch* e *merge*. Assim, a etapa de *fetch* pode ser saltada se a intenção é logo após realizar um *pull*.

# Seção 5

## Boas práticas

# Boas práticas

## Comite logo e com frequência

O git não consegue gerenciar as versões do seu trabalho se você não comitar! Não deixe que as possibilidades de um futuro melhor te impeçam de comitar agora. Encontre *checkpoints* no seu trabalho e transforme-os em *commits*.

## Mensagens significantes para commits

Faça as pessoas rapidamente entenderem o que os seus commits **provocarão caso sejam integrados ao repositório delas**.

Algumas boas ideias para commits curtos [3]:

- Para commits curtos, 50 caracteres ou menos.
- Comece com letra maiúscula.
- Sem ponto final.
- Escreva no imperativo: “Fix bug”, e não “Fixed bug”, ou “Fixes bug”.

# Boas práticas

## Comite logo e com frequência

O git não consegue gerenciar as versões do seu trabalho se você não comitar! Não deixe que as possibilidades de um futuro melhor te impeçam de comitar agora. Encontre *checkpoints* no seu trabalho e transforme-os em *commits*.

## Mensagens significantes para commits

Faça as pessoas rapidamente entenderem o que os seus commits **provocarão caso sejam integrados ao repositório delas**.

Algumas boas ideias para commits curtos [3]:

- Para commits curtos, 50 caracteres ou menos.
- Comece com letra maiúscula.
- Sem ponto final.
- Escreva no imperativo: “Fix bug”, e não “Fixed bug”, ou “Fixes bug”.

# Boas práticas

## Use commits para consertar o que está público

Uma vez que você deu *push*, alguém pode estar utilizando suas alterações. O ideal é commitar consertos, ao invés de tentar desfazer as alterações diretamente pelo git.

## Divida o trabalho em repositórios

- Um repositório por conceito.
- Repositórios para arquivos compartilhados entre múltiplos projetos.
- Se necessário ter arquivos binários, incluí-los em repositórios separados.

# Boas práticas

## Use commits para consertar o que está público

Uma vez que você deu *push*, alguém pode estar utilizando suas alterações. O ideal é commitar consertos, ao invés de tentar desfazer as alterações diretamente pelo git.

## Divida o trabalho em repositórios

- Um repositório por conceito.
- Repositórios para arquivos compartilhados entre múltiplos projetos.
- Se necessário ter arquivos binários, incluí-los em repositórios separados.



# Referências



F. Santacroce

*Git Essentials.*



Seth Robertson

*Commit often, Perfect later, Publish once*



Tim Pope

*A Note About Git Commit Messages*