

Curso de Introdução a Git

Pyladies São Carlos

2020-09

List of Figures

1	"Versionamento manual"	1
2	Git é uma das ferramentas de versionamento mais conhecidas . .	1
3	Saída do comando <i>git add -h</i>	4
4	Mensagem de <i>commit</i>	5
5	<i>Commit</i> com trabalho não finalizado	6

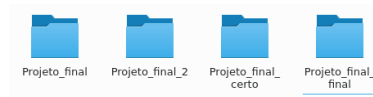
Contents

1	O que é Git?	1
1.1	Como funciona o Git?	1
1.1.1	Os três estados de Git	2
1.1.2	Instalação	2
2	Uso básico Git	4
2.1	Inicializando o repositório em um diretório existente	4

1 O que é Git?

Git é uma ferramenta de controle de versão, ou seja, permite armazenar diferentes mudanças em um arquivo ou em conjunto de arquivos ao longo do tempo, de modo que seja possível recuperar todas as versões salvas. Trata-se de uma poderosa ferramenta para desenvolvimento de softwares colaborativos e de projetos complexos.

Figure 1: "Versionamento manual"



Quem nunca realizou inúmeras cópias do mesmo projeto, com nomes diferentes (e às vezes nada intuitivos), para fazer controle das diversas versões de um mesmo projeto? Esse método informal não é eficaz nem mesmo quando se trata de projeto simples com apenas um contribuidor.

Quando várias pessoas, ao longo do tempo e em diversos computadores, têm acesso ao projeto e modificam diversas partes dele, a complexidade do controle de versionamento recrudesce. Ainda há outros fatores a levar em consideração: e se algum dos contribuidores fizer algo errado e for necessário voltar ao estado anterior? E se duas pessoas trabalharem simultaneamente no projeto e acabarem sobrescrevendo a mesma parte?

Figure 2: Git é uma das ferramentas de versionamento mais conhecidas



1.1 Como funciona o Git?

Git vê os dados como um fluxo de snapshots. Quando salvamos o estado de um projeto, Git tira uma foto do estado de todos os dados naquele momento. Em seguida, ele salva a referência em um snapshot. Caso os arquivos não tenham mudado desde o último ponto de salvamento, Git não os salva novamente, mas cria um link para o último arquivo idêntico armazenado.

A maior parte das operações realizadas pelo Git necessita de arquivos e recursos locais. Visto que o histórico inteiro do projeto é armazenado no disco local, a maior parte das operações são muito rápidas.

Isso implica que, caso o usuário queira visualizar a diferença entre a versão de um projeto atual e uma mais antiga, Git mostra o resultado das diferenças com base na busca local, sem a necessidade de pedir a um servidor remoto ou buscar uma versão antiga armazenada remotamente.

Dessa forma, as limitações para uso offline são poucas. É possível continuar trabalhando no projeto e realizar commits na cópia local. Assim que a conexão com o repositório remoto for re-estabelecida, basta atualizar o repositório remoto.

1.1.1 Os três estados de Git

- Modificado: quando o arquivo é modificado - *modified* -, significa que ele foi alterado mas ainda não foi colocado em *stage*.
- Staged: o arquivo modificado já foi adicionado ao *stage*. Isso significa que o arquivo está marcado para entrar no próximo commit.
- Commitado: Após o commit, o arquivo está armazenado no banco de dados local.

Git funciona, basicamente, da seguinte forma:

1. Os arquivos são modificados na *working tree*.
2. As modificações desejadas para o próximo *commit* são selecionadas para a área de *stage*.
3. Com o *commit*, os arquivos - na versão em que estão na *stage* - são armazenados em formato de *snapshot* no diretório local Git.

Working tree - ou árvore de trabalho - é uma versão do projeto. A área de *stage*, por sua vez, trata-se de um arquivo que armazena informações sobre quais modificações entrarão no próximo *commit*. É uma forma de se certificar que foram selecionados os arquivos corretos para serem commitados. Por último, o diretório Git é o local onde Git armazena metadados. ¹

1.1.2 Instalação

É possível utilizar Git de forma gráfica e por meio da linha de comando. Para este curso, será utilizada a linha de comando.

- Para instalar em distribuições baseadas em Debian, como o Ubuntu:

```
$ sudo apt install git-all
```

- Em distribuições baseadas em RPM, como o Fedora:

¹Metadados são informações associadas a dados que cumprem o objetivo de facilitar a organização. Por exemplo, ao capturar uma foto, metadados são associados a ela, como tamanho e formato do arquivo.

```
$ sudo dnf install git-all
```

O website Git possui outras informações sobre instalação em diversas distribuições Unix: .

Instalação no Windows:

2 Uso básico Git

É possível obter um repositório Git de duas formas: tornando um diretório local em um repositório Git ou clonando um repositório existente.

2.1 Inicializando o repositório em um diretório existente

Para inicializar o repositório, é necessário ir até o diretório do projeto. Uma vez dentro do diretório, basta digitar:

```
$ git init
```

Esse comando é responsável por criar um subdiretório chamado *.git* que contém os arquivos essenciais para o repositório. Nenhum arquivo está sendo rastreado ainda. Arquivos podem ser adicionados através do comando *git add*:

```
$ git add .
```

Ao inserir *.* após o comando, você está dizendo que Git deve adicionar à área de *stage* todos os arquivos não rastreados. Outros exemplos de uso do comando *git add*:

```
$ git add *.c
$ git add my_first_file.txt
```

Ao utilizar os dois comandos acima, você está dizendo ao Git para adicionar à área de *stage* todos os arquivos que tenham a extensão *.c* e para adicionar o arquivo *my_first_file.txt*, respectivamente.

Para visualizar mais opções de uso do comando *git add* (ou de qualquer outro comando), você pode acrescentar o argumento *-h* ou *-help*:

```
$ git add -h
```

A saída para o comando acima é a descrição de uso do comando, como mostrado na figura 3:

Figure 3: Saída do comando *git add -h*

```
usage: git add [<options>] [--] <pathspec>...

-n, --dry-run          dry run
-v, --verbose          be verbose

-i, --interactive      interactive picking
-p, --patch            select hunks interactively
-e, --edit             edit current diff and apply
-f, --force           allow adding otherwise ignored files
-u, --update          update tracked files
--renormalize          renormalize EOL of tracked files (implies -u)
-N, --intent-to-add    record only the fact that the path will be added later
-A, --all              add changes from all tracked and untracked files
--ignore-removal       ignore paths removed in the working tree (same as --no-all)
--refresh             don't add, only refresh the index
--ignore-errors        just skip files which cannot be added because of errors
--ignore-missing       check if - even missing - files are ignored in dry run
--chmod (+|-)x        override the executable bit of the listed files
--pathspec-from-file <file>
                        read pathspec from file
--pathspec-file-nul    with --pathspec-from-file, pathspec elements are separated with NUL character
```

Em seguida, após selecionar os arquivos que deverão ser adicionados à área de *stage*, podemos dar *commit* às modificações:

```
$ git commit -m "Initial project version"
```

Ao digitar o comando com o parâmetro *-m*, você está dizendo ao Git que o *commit* está acompanhado da mensagem de *commit* entre aspas (que pode ser simples ou dupla).

É muito importante prestar atenção à maneira como você escreve a mensagem de *commit*. A intenção é que ela seja assertiva, sucinta e padronizada. Normalmente, é escrita em inglês, iniciada por um verbo que indica a modificação (*Add*, *Fix*, *Remove*, *Update*, *Create*) e provê uma descrição a respeito das modificações contidas no *commit*.

Já sabemos que um dos principais motivos para utilizar uma ferramenta de versionamento é a possibilidade de organizar e controlar eficazmente um projeto complexo e desenvolvido por várias pessoas. Nesse contexto, é de suma importância que tanto o código quanto as mensagens de *commit* sejam padronizadas e, acima de tudo, sejam facilmente compreendidas por outras pessoas.

Um exemplo ruim de mensagem de *commit*:

```
$ git commit -m "Fixed the problems with main function"
```

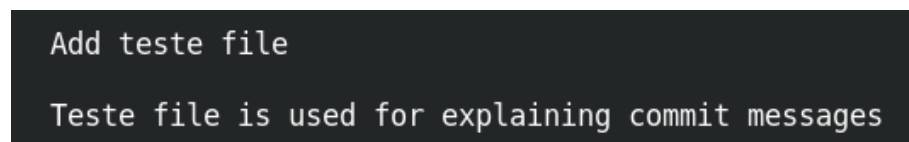
Um bom exemplo de mensagem de *commit*:

```
$ git commit -m "Fix seg fault on main function"
```

O objetivo é explicar qual a necessidade do *commit*. Ela deve ser sucinta - você deve buscar resumir as mudanças em 50 caracteres ou menos.

Também é possível dividir a mensagem de *commit* em tema e corpo. Nesse caso, é possível fornecer uma explicação mais detalhada dentro do corpo, como mostrado na figura 4:

Figure 4: Mensagem de *commit*



```
Add teste file

Teste file is used for explaining commit messages
```


Para isso, basta inserir o comando da seguinte forma:

```
$ git commit -m "Add teste file" -m "Teste file is used for explaining"
```

Às vezes, você quer salvar o progresso em um projeto dentro de um *commit* para poder fazer upload no Github e deixar as modificações seguras, caso algo dê errado com seu computador ou com sua cópia local. No entanto, você não quer que os outros contribuidores vejam o novo *commit* e pensem que a tarefa

está terminada. Uma boa prática é adicionar a *tag* '[WIP]' - *Work in Progress* - no início do *commit*, como ilustrado na figura 5:

Figure 5: *Commit* com trabalho não finalizado



```
[WIP] Add formatting file to book
```