

Hello Git

Um tutorial para usar o Git imediatamente

Francisco de Assis Boldt <boldt.pro.br>

Version 0.1.1 2021-01

Sumário

1. Introdução: O hello world do Git	1
1.1. Por que usar Git?	1
1.2. Qual a diferença entre Git e GitHub?	2
1.3. Como ler este livro?	3
2. Iniciando um repositório local	5
2.1. O comando <code>git status</code>	6
2.2. Criando um arquivo novo	7
2.3. Adicionando um arquivo em <i>stage</i>	8
2.4. Monitorando o arquivo criado	10
2.5. Como desfazer alterações?	12
3. Repositório remoto no GitHub	14
3.1. Fazendo uma cópia local do repositório	16
3.2. O comando <code>git log</code>	18
3.3. Alterando a cópia local	18
3.4. Atualizando a cópia remota	20
3.5. Atualizando o repositório local	22
3.6. Vantagens de ter um repositório remoto	23
4. Criando fotografias novas e acessando fotografias antigas ...	24
4.1. Listando as fotografias do repositório	29
4.2. Mostrando o conteúdo de fotografias	30
4.3. Alterando o estado do sistema	32
4.4. Criando etiquetas para fotografias	35
5. Ramos no projeto	40
5.1. Criando ramos	40
5.2. Uma alteração incompleta para o ramos atual	41
5.3. Terminado a alteração desejada	45
5.4. Mesclando o ramo atual com o ramo principal	47
6. Criando bifurcações no projeto	49

6.1. Criando um ramo comum.....	49
6.2. Criando mais um ramo comum.....	51
6.3. Listando as fotografias em forma de grafo.....	53
6.4. Mesclando o último ramo antes do primeiro.....	54
6.5. Quando não corre tudo bem na mesclagem.....	56
7. Conclusão	60
Bibliografia	61

Capítulo 1. Introdução: O hello world do Git

Este livro é um tutorial que apresenta uma forma muito simples de se usar o Git. Como exemplo, são feitas algumas versões do clássico programa "hello world". Essas versões serão gerenciadas pelo sistema controlador de versões Git. A intenção é mostrar uma forma de se começar a usar o Git dentro de poucos minutos. É apenas um primeiro contato com a ferramenta. Nenhum conteúdo é abordado por completo ou com profundidade. Para isso, são sugeridas obras como [\[hodson2014ry\]](#) e [\[chacon2014pro\]](#).

Mesmo assim, é bom deixar claro que Git é um sistema de gerenciamento de versões de software. Porém, apesar de ter sido idealizado para o desenvolvimento de programas, também pode ser usado para outras finalidades como escrita compartilhada de textos ou qualquer outro tipo de arquivo, até mesmo arquivos de imagens. O Git mantém um histórico das alterações de um repositório permitindo recuperar informações, dividir as tarefas em ramos e mesclar alterações que podem ser feitas independentemente.

1.1. Por que usar Git?

Quem nunca fez uma cópia de um arquivo que desejava alterar (só pra garantir)? Fazemos isso tanto para um código de programa quanto para algum outro tipo documento. O Git mantém essas cópias de forma mais eficiente e organizada. Com o Git, uma equipe consegue trabalhar em diferentes funcionalidades simultaneamente. As cópias só guardam o que foi alterado de uma versão do documento para a outra. Por isso é mais eficiente. O Git te ajuda mesmo que você esteja trabalhando sozinho.

Pessoalmente, várias vezes eu testei alguma alteração em um programa que estava desenvolvendo, e algumas vezes essa alteração saiu muito errada. Não precisei me desesperar. Simplesmente descartei todas as alterações e o programa voltou a funcionar como antes. Mas nem sempre foi assim. Eu mesmo já usei o Git como um simples cliente para enviar e fazer o download do meu código no GitHub como se fosse apenas um backup. Muitas vezes eu precisei excluir o repositório local e baixá-lo novamente. Fiz isso por que eu não conhecia nem os comandos básicos do Git. Espero que você, após ler este livro, não precise cometer os mesmos erros que eu cometi.

1.2. Qual a diferença entre Git e GitHub?

Como já mencionado, Git é um programa que gerencia as versões de arquivos com código fonte de programas e outros documentos. O GitHub é uma plataforma de hospedagem de código que, como o nome já deixa claro, usa o Git como interface de interação. É uma das plataformas mais usadas e famosas. Porém, existem outras como Bitbucket e Gitlab. Neste livro, os exemplos com repositórios remotos serão com o Github. Mas os conceitos são os mesmos para qualquer plataforma de hospedagem que use o Git como interface de interação. O que muda entre as plataformas de hospedagem de código são alguns detalhes da interface web de cada serviço. Também muda o que cada plataforma oferece para seus usuários. Por exemplo, duas ferramentas que eu acho muito úteis no GitHub é o GitHub Pages e o GitHub Actions. Minha página pessoal é hospedada no GitHub pages. Estou usando o GitHub Actions para renderizar este livro toda vez que faço uma atualização no servidor. Acredito que, caso você tenha interesse em usar uma plataforma de hospedagem de código, as informações contidas aqui serão suficientes para você escolher a plataforma que se adequa melhor às suas necessidades.

1.3. Como ler este livro?

Este livro foi pensado para quem nunca teve contato com o Git e nem sabe direito o que ele é. Mas se você já tem uma noção, ou já usou o Git de alguma forma, a figura 1 mostra uma sugestão de como ler este livro.

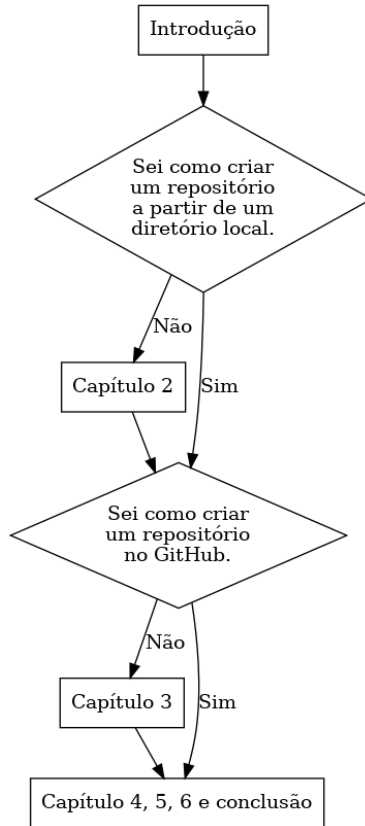


Figura 1. Como ler este livro?

O capítulo 2 ([Iniciando um repositório local](#)) mostra como criar um repositório no seu computador a partir de um diretório. Como criar um repositório no GitHub e utilizá-lo no seu computador é mostrado no capítulo 3 ([Repositório remoto no GitHub](#)). No capítulo 4 ([Criando fotografias novas e acessando fotografias antigas](#)) é mostrado como criar versões e acessar versões anteriores. Veremos como trabalhar simultaneamente

com versões diferentes de um programa no capítulo 5 ([Ramos no projeto](#)). Você verá o mínimo necessário para mesclar bifurcações no capítulo 6 ([Criando bifurcações no projeto](#)). O capítulo 7 ([Conclusão](#)) reforça o que foi mostrado nos demais capítulos e sugere obras para você continuar seu aprendizado.

Neste livro, tudo será apresentado pela linha de comando. O mais importante é entender os conceitos e o motivo de se usar cada comando. Sem estes conceitos uma interface gráfica é inútil. Por isso, acredito que pela linha de comando você entenderá o necessário para usar uma interface gráfica posteriormente. Além disso, os comandos de terminal são os mesmos para Windows, Linux e Mac. Você encontra como instalar o Git em seu computador no endereço [Instalando o Git \(https://git-scm.com/book/pt-br/v2\)](https://git-scm.com/book/pt-br/v2).

Capítulo 2. Iniciando um repositório local

Um repositório Git é apenas um diretório com algumas metainformações sobre as alterações dos arquivos do projeto que está no diretório. A lista de comandos 1 mostra como criar um repositório Git do zero. Note que foi criado um diretório oculto, por que começa com um ponto (.), e por isso só é listado (**ls**) quando se usa o parâmetro **-a**.

Lista de comandos 1. Criando um repositório vazio.

```
~$ mkdir greetings ①
~$ cd greetings ②
~/greetings$ ls -a ③
.  ..
~/greetings$ git init ④
Initialized empty Git repository in
home/fulano/greetings/.git/
~/greetings$ ls -a ⑤
.  ..  .git
```

- ① Cria diretório greetings, onde colocaremos o código do nosso projeto.
- ② Acessa diretório greetings.
- ③ Lista os arquivos do diretório, incluindo arquivos ocultos.
- ④ Cria um repositório a partir do diretório atual.
- ⑤ Lista arquivos do diretório, incluindo o diretório **.git** que é um diretório oculto.

A lista com todos os arquivos e subdiretórios criado pelo comando **git init** pode ser vista na lista de comandos 2. O comando **ls .git** foi usado para exibir essas informações. O diretório **.git** contém metadados. Ou seja, é um diretório com

dados para o Git controlar os dados do repositórios. É a forma do Git "lembrar" da história do projeto. Um repositório Git nada mais é do que um diretório com um subdiretório `.git` adequadamente estruturado. O subdiretório `.git` não deve ser alterado diretamente. Suas alterações devem ser feitas através do comando `git` acompanhado dos parâmetros correspondentes à ação desejada.

Lista de comandos 2. Listagem de arquivos e diretórios do repositório.

```
~/greetings$ ls .git/  
HEAD branches config description hooks index info  
logs objects packed-refs refs
```

2.1. O comando `git status`

Um comando que é usado o tempo todo em um repositório Git é o comando `git status`, apresentado na lista de comandos 3. Apesar desse comando estar em uma seção dedicada a ele, devido a sua importância, ele não será abordado em profundidade, uma vez que nosso objetivo aqui é mostrar uma utilização do Git com poucos comandos, sem explorá-los por completo.

Lista de comandos 3. Árvore de diretórios do repositório.

```
~/greetings$ git status  
On branch master ①  
  
No commits yet ②  
  
nothing to commit (create/copy files and use "git add"  
to track) ③
```

- ① A resposta desta linha é `On branch master`. *Banches* são ramificações de um projeto Git, que podem tomar rumos diferentes durante seu desenvolvimento. Diferentes ramos

podem ser mesclados ou se tornarem novos projetos. O nome **master** é o nome padrão para o ramo inicial do projeto Git. Não existe nenhuma exigência de que exista um ramo com esse nome no projeto.

- ② Esta linha mostra **No commits yet**. Os *commits* são fotografias do sistema que o Git mantém em seus metadados no diretório **.git**. Como não fizemos nenhum *commit* ainda, não existe nenhuma "fotografia" no Git.
- ③ Aqui aparece **nothing to commit (creat/copy files and use "git add" to track)**. Não há nada para "fotografar" (*commit*) por que o diretório do projeto está vazio. Nenhum arquivo está sendo monitorado. Para monitorar um arquivo temos que usar o comando **git add**, que é apresentado na próxima seção.

2.2. Criando um arquivo novo

Como pode ser traduzido da terceira linha de resposta da lista de comandos 3, o Git monitora (*track*) arquivos criados ou copiados para dentro do diretório do repositório. A lista de comandos 4 mostra o conteúdo que queremos no arquivo **greet.py**. Este arquivo em texto simples pode ser criado dentro do diretório ou copiado para dentro dele. É um arquivo escrito na linguagem de programação Python 3. Porém, não é necessário saber Python para acompanhar este tutorial. Basta notar que o arquivo será alterado e cada alteração será monitorada pelo Git.

Lista de comandos 4. Criando o primeiro arquivo do projeto

```
~/greetings$ cat greet.py / ①
print("hello")
~/greetings$ python greet.py ②
hello
~/greetings$ ls -a ③
.  ..  .git  greet.py
```

- ① Mostra o conteúdo do arquivo `greet.py`.
- ② Executa o arquivo criado com o comando `python greet.py`.
(Não é necessário executar essa linha para acompanhar este tutorial.)
- ③ Só para mostrar que o arquivo `greet.py` foi criado.

Depois de criado o arquivo `greet.py`, o comando `git status` mostrará uma resposta um pouco diferente, como mostra a lista de comandos 5. Apenas as duas primeiras linhas continuam iguais.

Lista de comandos 5. Status com arquivo fora da área de stage

```
~/greetings$ git status
On branch master

No commits yet

Untracked files: ①
  (use "git add <file>..." to include in what will be
  committed)
    greet.py

nothing added to commit but untracked files present
(use "git add" to track) ②
```

- ① Essa linha de resposta diz que o arquivo `greet.py` não está sendo monitorado (*untracked*).
- ② Veja que a resposta do comando já diz como adicionar a modificação no monitoramento (`git add <file>`).

2.3. Adicionando um arquivo em *stage*

Aqui é um ponto que merece uma atenção especial quando se usa o Git. O Git só "fotografa" (*commit*) as alterações que estão em

uma área abstrata de sua organização chamada **stage**. Isso por que, muitas vezes fazemos alterações que não queremos gravar. Testamos algo, não gostamos do resultado, e queremos descartar o que foi feito. Outras vezes, queremos gravar só algumas alterações, mas não todas. Esse processo também nos permite fazer alterações variadas e agrupá-las em fotografias separadas.

A lista de comandos 6 mostra como adicionar um arquivo na área de *stage* do Git, com o comando `git add greet.py`. Note que agora o arquivo está sendo monitorado, mas ainda não foi "fotografado" (*committed*). Se você colocar um arquivo na área de *stage* por engano, pode removê-lo de lá, sem excluí-lo do diretório, com o comando `git rm --cached <file>`, como mostra a resposta do comando `git status`.

Lista de comandos 6. Status com arquivo na da área de stage

```
~/greetings$ git add greet.py
~/greetings$ git status
On branch master

No commits yet

Changes to be committed: ①
  (use "git rm --cached <file>..." to unstage) ②
    new file:   greet.py ③
```

- ① Mostra que existem alterações que podem ser fotografadas.
- ② Mostra como retirar um arquivo da área de stage.
- ③ Indica que `greet.py` é um arquivo novo. Isto é, um arquivo que não estava sendo monitorado, mas poderá ser monitorado se fizer parte do próximo *commit*. É isso que faremos agora.

2.4. Monitorando o arquivo criado

É importante resaltar que até o momento, nenhum arquivo está sendo monitorado pelo Git. O arquivo `greet.py` está pronto para ser "fotografado" e, a partir daí, ser monitorado. Para fotografar as mudanças que estão na área de **stage** deve-se executar o comando da lista de comandos 7.

O comando só será aceito se você estiver com seu nome e email configurado. Para não entrar em detalhes de configuração agora, você pode digitar os comandos `git config user.name 'SEU_PRIMEIRO_NOME SEU_ULTIMO_NOME'` e `git config user.email 'SEU_EMAIL@example.com'`.

Lista de comandos 7. Primeira fotografia do repositório

```
~/greetings$ git commit -m 'primeira fotografia do
sistema'
[master (root-commit) 06cbe0b] primeira fotografia do
sistema
1 file changed, 1 insertion(+)
create mode 100644 greet.py
```

Quando se executa o comando `git commit` sem o parâmetro `-m 'comentário'` o Git abre um editor de texto para que um comentário sobre a fotografia seja escrito. O Git não permite commits sem comentários. Então, foi usado aqui o parametro `-m` para ficar mais resumido e visível através das listas de comandos.

Se você quiser ver a fotografia tirada do sistema pode usar o comando `git show` e terá um resultado parecido com o da lista de comandos 8. Vamos entender essa fotografia.

Lista de comandos 8. Vendo detalhes da fotografia mais recente do sistema

```
~/greetings$ git show ①
commit 06cbe0b360ee871baf55d48aa1914d8b73708b4b (HEAD
-> master) ②
Author: Francisco de Assis Boldt <fboldt@gmail.com> ③
Date: Tue Dec 22 08:21:39 2020 -0300 ④

    primeira fotografia do sistema ⑤

diff --git a/greet.py b/greet.py
new file mode 100644
index 0000000..11b15b1
--- /dev/null ⑥
+++ b/greet.py ⑦
@@ -0,0 +1 @@
+print("hello") ⑧
```

- ① O comando `git show`.
- ② Logo depois da palavra `commit` está o *hash* da fotografia. O *hash* é a assinatura, o identificador, da fotografia. Podemos usar esse identificador para acessar a fotografia posteriormente. Na mesma linha temos a palavra **master**, indicando que o ramo do projeto chamado **master** está atualizado de acordo com esta fotografia. Ainda na mesma linha temos a palavra **HEAD** seguida uma seta (->). Esta seta indica que o estado do sistema que estamos vendo no momento está apontando para o ramo master. Isso ficará mais claro a seguir.
- ③ Quem fez a fotografia (dados inseridos pelos comandos `git config ...`).
- ④ Quando a fotografia foi feita.
- ⑤ O comentário inserido pelo comando `git commit`. No nosso caso, após o parâmetro `-m`.

- ⑥ O arquivo `/dev/null` indica que o arquivo ainda não existia em uma fotografia anterior. Esta é primeira fotografia que registra sua criação. Os três sinais de menos (-) indicam que as linhas seguintes iniciadas por - foram removidas do arquivo. Como o arquivo ainda não existia, nada foi removido.
- ⑦ O nome do arquivo que foi alterado. As linhas que começam com o sinal + são as que foram inseridas no arquivo.
- ⑧ Apenas uma linha foi inserida nesse arquivo recém criado.

2.5. Como desfazer alterações?

Com o que foi mostrado neste capítulo, já dá para usar o Git para desenvolver seus programas. Nossos programas e outros documentos raramente são criados de uma vez. Começamos com uma versão simples e vamos incrementando versão após versão até finalizarmos o que precisamos fazer. Então, ao invés de fazer cópias dos nossos arquivos, podemos simplesmente alterar o arquivo sem medo. Por exemplo, digamos que fizemos uma alteração no arquivo `greet.py` e o programa parou de funcionar. Suponhamos que a alteração foi tão complicada que seria melhor descartar todas as alterações feitas e começar tudo de novo. Se ainda **não** executamos o comando `git add greet.py`, podemos descartar as alterações antes delas entrarem na área de **stage**. Basta executar o comando `git checkout -- greet.py`. Tudo voltará como estava no início.

Mas, se você tinha achado que a alteração seria uma boa idéia ou simplesmente executou o comando `git add .` sem querer, nenhum motivo para desespero. Basta usar o comando `git reset -- greet.py`. O Comando `git add .` adiciona na área de **stage** todas as alterações feitas no repositório. Se você usar o comando `git reset` todas as adições para a área de **stage** serão removidas para a lista de não monitorados. Daí você pode adicionar em **stage** somente os arquivos que você deseja na próxima fotografia.

Em resumo, você já pode usar o Git de maneira eficiente para implementar seus programas ou escrever seus documentos. Nos próximos capítulos você vai conhecer mais alguns comandos do Git que vão lhe ajudar a desenvolver seus trabalhos de forma mais eficiente e segura. Recomendo que você aprenda a usar um repositório remoto, de preferência na nuvem, em uma plataforma como o GitHub. Entretanto, se você já sabe como fazer isso, ou não tem interesse em usar um repositório remoto, você pode pular o capítulo seguinte e ir direto para o capítulo 4.

Capítulo 3. Repositório remoto no GitHub

Para criar uma conta no GitHub basta acessar o site github.com e clicar em um botão escrito **Sign up**, ou colocar o endereço github.com/join direto na barra de endereços de um navegador web. É necessário preencher os campos com

1. um nome de usuário (*Username*) que ainda não exista;
2. um endereço de email válido;
3. uma senha com pelo menos 15 caracteres alfanuméricos ou uma senha com pelo menos 8 caracteres que contenha pelo menos um (1) número e uma (1) letra minúscula.

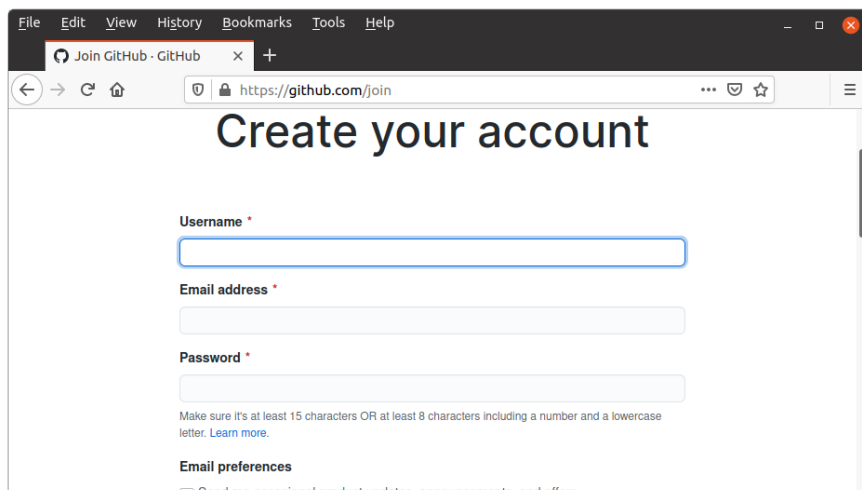


Figura 2. Criar conta no GitHub.

Após ter o cadastro efetuado e confirmado pelo email cadastrado, é possível entrar no sistema pelo endereço github.com/login. Se o login e a senha estiverem corretos, a página aberta terá a divisão apresentada na figura 3. Ao clicar na sua foto ou símbolo que o GitHub cria, é possível acessar seu perfil (*Your profile*), seus repositórios (*Your repositories*) etc.

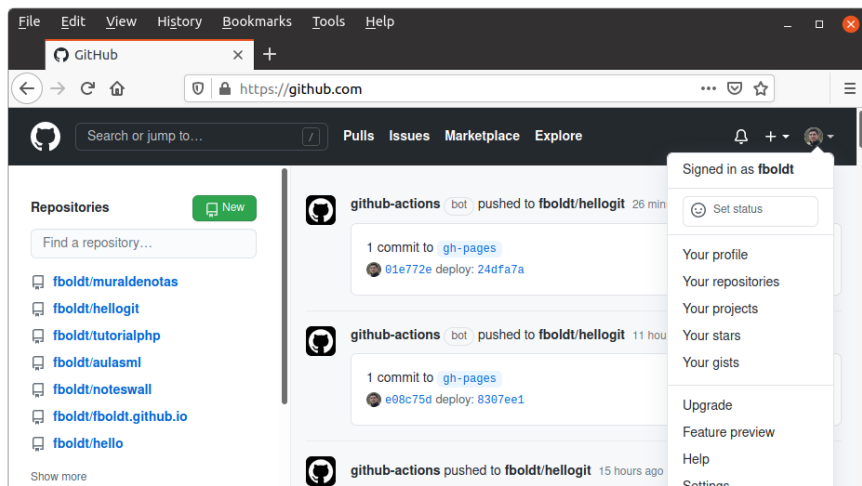


Figura 3. Página inicial do GitHub após login.

Para criar um repositório novo (figura 4) é necessário inserir um nome de repositório que não exista na sua lista. A descrição do repositório é opcional, mas aqui nós colocaremos *O "Hello World" do Git*. Quando o repositório é público (*public*) qualquer pessoa pode visualizá-lo, mas quando é privado (*private*) só você pode visualizá-lo.

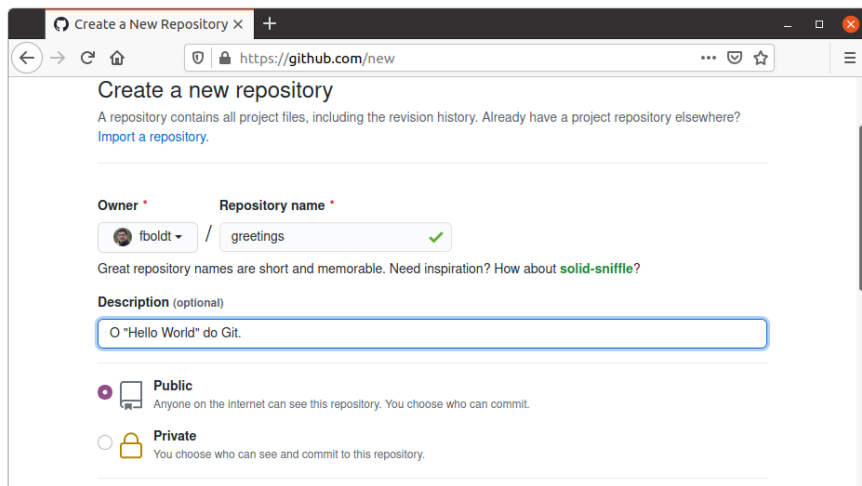


Figura 4. Criando um repositório novo no GitHub.

Para este tutorial, antes de criar o repositório, marque a opção

"Add a README file" (adicionar arquivo README), como mostra a figura 5. Agora é só clicar no botão "Create a repository" (criar repositório).

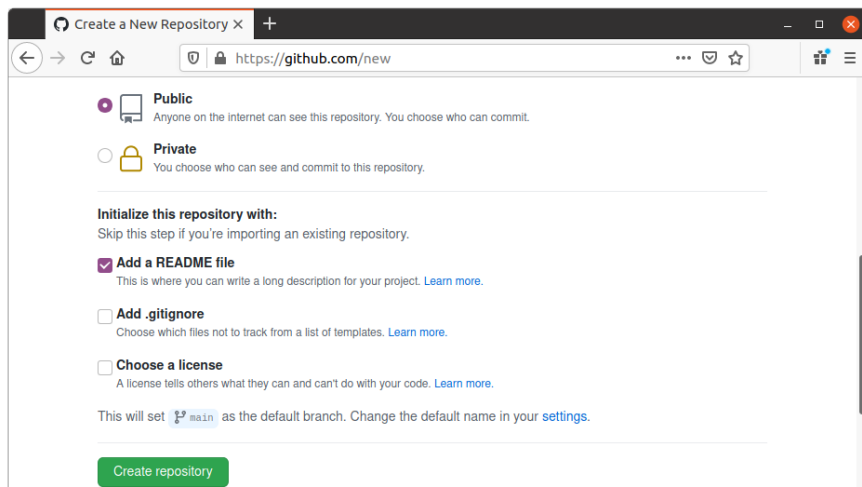


Figura 5. Marcar a opção "Add a README file".

3.1. Fazendo uma cópia local do repositório

Agora que o repositório está criado, é necessário fazer um cópia local dele para poder alterá-lo. O link para fazer essa cópia pode ser encontrado clicando-se no botão "Clone" do repositório.

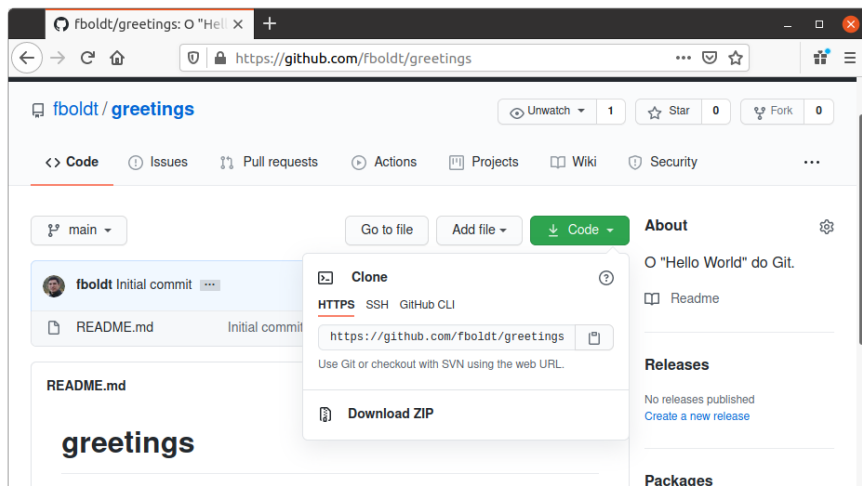


Figura 6. Link para clonar um repositório.

Se você fez os passos do capítulo anterior, antes de continuar o tutorial deste capítulo você precisa excluir ou renomear o repositório anterior.

Depois de copiado o link, usasse o comando `git clone <link>` para fazer o download do repositório no computador local, como mostra a lista de comandos 9.

Lista de comandos 9. Clonar o repositório em seu computador local.

```
~$ git clone https://github.com/fboldt/greetings.git
Cloning into 'greetings'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-
reused 0
Unpacking objects: 100% (3/3), 619 bytes | 619.00
KiB/s, done.
```

3.2. O comando `git log`

Agora, já é possível acessar o diretório criado e verificar que já existe fotografia (*commit*) do repositório. Esta foto foi "tirada" quando inserimos o arquivo "README.md", marcando a opção "Add a README file" ao criar o repositório no GitHub.

Lista de comandos 10. Lista de fotografias do repositório clonado.

```
~$ cd greetings/ ①
~/greetings$ git log ②
commit 58da81bbc897dcdcf877530ae972fd2d4b3cc9c8d (HEAD
-> main) ③
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date: Tue Dec 22 08:29:25 2020 -0300
```

Initial commit

- ① Acessa repositório clonado.
- ② Lista fotografias do repositório.
- ③ Note que o ramo (*branch*) criado pelo GitHub se chama `main` e não `master` como mostrado no capítulo anterior. É apenas uma preferência do GitHub. Mas esse nome pode ser alterado se você quiser.

3.3. Alterando a cópia local

Para continuar, crie (ou copie) o arquivo `greet.py` de forma que ele tenha o conteúdo mostrado na listagem 11.

Lista de comandos 11. Recriando o arquivo em python.

```
~/greetings$ cat greet.py
print("Hello!")
```

O comando `git status` agora traz uma informação a mais, como mostra a listagem 12. Agora aparece a linha `Your branch is up to date with 'origin/main'`, informando que a cópia local do repositório está atualizada com o repositório remoto hospedado no GitHub.

Lista de comandos 12. Status após a criação do arquivo `greet.py`.

```
~/greetings$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

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

Vamos adicionar o arquivo `greet.py` na área de **stage**, como na listagem 13.

Lista de comandos 13. Adiciona alterações de `greet.py` na área Stage.

```
~/greetings$ git add greet.py
~/greetings$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   greet.py
```

O próximo passo é fotografar as alteração da área de **stage**, como na listagem 14.

Lista de comandos 14. Fotografia as alterações.

```
~/greetings$ git commit -m 'cria arquivo greet.py'
[main 6c8951a] cria arquivo greet.py
1 file changed, 1 insertion(+)
create mode 100644 greet.py
```

Note que os comandos e suas respectivas respostas neste capítulo estão muito parecidos com os do capítulo anterior. Por isso, algumas informações contidas lá não estão sendo repetidas aqui.

3.4. Atualizando a cópia remota

Para atualizar o repositório remoto usa-se o comando `git push`, como mostra a listagem 15.

Lista de comandos 15. Faz o upload das alterações.

```
~/greetings$ git push
Username for 'https://github.com': fboldt ①
Password for 'https://fboldt@github.com': ②
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 311 bytes | 311.00 KiB/s,
done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fboldt/greetings.git
58da81b..6c8951a  main -> main
```

- ① Insere nome de usuário com autorização para alterar o repositório.
- ② Insere a senha do usuário.

Ao acessar o repositório no GitHub, pode-se ver as alterações

atualizadas, como mostra a figura 7.

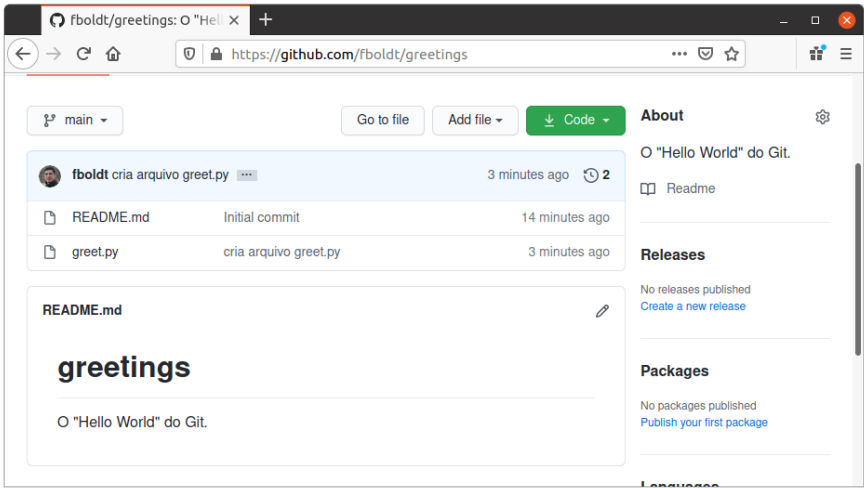


Figura 7. Estado do GitHub de do comando push.

Alterações também podem ser feitas diretamente no site do GitHub. Por exemplo, para alterar o arquivo README.md, basta clicar no lápis da figura 7. Vamos mudar o título de `# greetings` para `# Hello Git`, como na figure 8, e depois clicar em **Commit changes** para fotografar esta alteração.

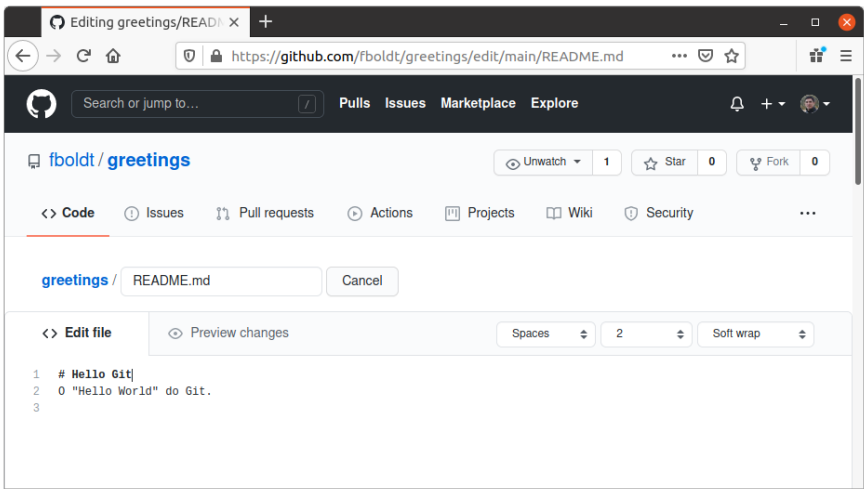


Figura 8. Atualizando o arquivo README.md dentro do GitHub.

3.5. Atualizando o repositório local

Para verificar se o repositório remoto foi alterado, usa-se o comando `git fetch`, como na listagem 16.

Lista de comandos 16. Verifica se existem alterações.

```
~/greetings$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-
reused 0
Unpacking objects: 100% (3/3), 683 bytes | 683.00
KiB/s, done.
From https://github.com/fboldt/greetings
   6c8951a..933fc74  main       -> origin/main
```

O comando `git status` da listagem 17 mostra que nossa cópia local está desatualizada por 1 fotografia (**1 commit**), e pode ser atualizada pelo comando `git pull`.

Lista de comandos 17. Status depois do comando fetch.

```
~/greetings$ git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and
can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Então, podemos executar o comando `git pull`, como na listagem 18.

Lista de comandos 18. O comando `git pull`.

```
~/greetings$ git pull
Updating 6c8951a..933fc74
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Agora o arquivo local está igual ao arquivo remoto, como pode ser visto na listagem [19](#).

Lista de comandos 19. Estado do arquivo `README.md` depois do comando `git pull`.

```
~/greetings$ cat README.md
# Hello Git
O "Hello World" do Git.
```

3.6. Vantagens de ter um repositório remoto

Dentre algumas vantagens de se ter um repositório na nuvem, podemos citar algumas.

1. Pode ser usado como backup.
2. Pode ser usado como repositório central para sincronizar vários computadores.
3. Disponibilizar o código para outras pessoas.
4. Trabalhar em equipe.

Você não precisa necessariamente usar a plataforma GitHub. Existem outras plataformas que também possuem tipos de contas sem custo para o usuário, como por exemplo, Bitbucket e GitLab, para citar algumas.

Capítulo 4. Criando fotografias novas e acessando fotografias antigas

O Git só vai tirar uma nova fotografia do sistema se algo for alterado e colocado na área de stage. A lista de comandos [20](#) mostra a alteração sugerida para esta parte do tutorial. O código foi alterado mas seu resultado do programa continuou igual. É uma boa prática de programação refatorarmos os nossos códigos para prepará-los para alterações de comportamento e adição de funcionalidades. Nesse caso, o programa é extremamente simples. Tal alteração seria desnecessária, mas é uma boa forma de mostrar como utilizar o Git em um processo de refatoração.

Lista de comandos 20. Fazendo uma alteração

```
~/greetings$ cat greet.py
def main():
    print("Hello!")

main()
```

Depois da alteração do arquivo, o comando `git status` apresenta um retorno diferente, como mostra a listagem [21](#). Novamente o arquivo `greet.py` não está na área de stage, mas agora esse arquivo está sendo monitorado. Então, temos duas opções: descartar as alterações com o comando `git restore greet.py` ou adicionar as alterações na área de stage com o comando `git add greet.py`.

Lista de comandos 21. Status com arquivo modificado fora da area de stage

```
~/greetings$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be
  committed) ①
  (use "git restore <file>..." to discard changes in
  working directory) ②
    modified:   greet.py

no changes added to commit (use "git add" and/or "git
commit -a")
```

- ① Mostra como adicionar um arquivo na área de stage.
- ② Mostra como descartar as alterações feitas no arquivo.

Adicionaremos as alterações na área de stage, como mostra a listagem 22.

Lista de comandos 22. Status com arquivo modificado na da area de stage

```
~/greetings$ git add greet.py ①
~/greetings$ git status ②
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   greet.py
```

- ① Adiciona o arquivo alterado na área de stage.
- ② O resultado do comando status mudou novamente.

O resultado do comando `git status` está muito parecido com o da listagem 6. Agora, não aparece mais "arquivo novo" (new file), mas "modificado" (modified). A listagem 23 mostra o comando `git commit` com o parâmetro `-m` e um comentário relacionado à alteração feita.

Lista de comandos 23. Atualizando o repositório e vendo os detalhes da atualização

```
~/greetings$ git commit -m 'cria função main' ①  
[main e0fe2b3] cria função main  
1 file changed, 4 insertions(+), 1 deletion(-) ②
```

- ① Registro da fotografia com um comentário que resume as alterações fotografadas.
- ② Mostra que apenas 1 arquivo foi alterado (*1 file changed*), 4 linhas foram inseridas (*4 insertions(+)*) e 1 linha foi removida (*1 deletion(-)*).

O comando `git show`, na listagem 24, mostra como ficou a fotografia.

Lista de comandos 24. Mostrando como a fotografia gravou as alterações

```
~/greetings$ git show
commit e0fe2b32abaa5b4bcb7a1889a820f155e9ec635e (HEAD
-> main)
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Wed Dec 23 08:24:50 2020 -0300

    cria função main

diff --git a/greet.py b/greet.py
index 693eaec..55bb9ae 100644
--- a/greet.py ①
+++ b/greet.py ②
@@ -1,4 @@
-print("Hello!") ③
+def main(): ④
+    print("Hello!") ④
+ ④
+main() ④
```

- ① Versão anterior do arquivo
- ② Versão atual do arquivo
- ③ Linha removida.
- ④ Linhas inseridas.

Note que qualquer alteração em uma linha considera aquela linha como removida e outra linha com a alteração como inserida. Vamos fazer mais uma alteração no sistema, que pode ser vista na listagem [25](#). Novamente, o código foi alterado sem alterar o comportamento do programa.

Lista de comandos 25. Fazendo mais uma alteração

```
~/greetings$ cat greet.py
def main():
    print("Hello!")

if __name__ == "__main__":
    main()
```

Depois dessa alteração, o comando `git status` apresentará o mesmo retorno visto na listagem 21. Vamos adicionar à área de stage a nova alteração com o comando `git add greet.py`. Após executado esse comando, o status do repositório será igual ao apresentado na listagem 22. Agora estamos prontos para executar o comando `commit` como mostra a lista de comandos 26.

Lista de comandos 26. Atualizando com a terceira alteração

```
~/greetings$ git commit -m 'verifica se é o programa principal'
[main bd61894] verifica se é o programa principal
1 file changed, 2 insertions(+), 1 deletion(-)
```

Novamente, podem ser vistas as alterações feitas, como mostrado na listagem 27.

```
~/greetings$ git show
commit bd618942ddf84ad2ceb062b7ef53c22b12a94dae (HEAD
-> main)
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Wed Dec 23 08:55:49 2020 -0300
```

verifica se é o programa principal

```
diff --git a/greet.py b/greet.py
index 55bb9ae..3062fd5 100644
--- a/greet.py
+++ b/greet.py
@@ -1,4 +1,5 @@
 def main():
     print("Hello!")

-main() ①
+if __name__ == "__main__": ②
+    main() ②
```

① Linha removida.

② Linhas inseridas.

Agora temos cópias seguras das versões anteriores do nosso projeto.

4.1. Listando as fotografias do repositório

A listagem 28 mostra como listar as fotografias do sistema com o comando `git log`. A opção `--oneline` foi usada aqui para que as fotografias sejam vistas de um forma mais compacta. Mas você deve testar sem essa opção também.


```
~/greetings$ git log --oneline
bd61894 (HEAD -> main) verifica se é o programa principal ①
e0fe2b3 cria função main
933fc74 (origin/main, origin/HEAD) Update README.md ②
6c8951a cria arquivo greet.py
58da81b Initial commit
```

- ① Fotografia com o estado atual do repositório local. A palavra **HEAD** está nesta linha.
- ② Fotografia com o estado atual do repositório remoto. As palavras **origin/HEAD** estão nesta linha. Neste caso, origin indica o repositório remoto.

A palavra *HEAD* indica o estado atual do repositório, seja ele local ou remoto. As fotografias do repositório são apresentadas em ordem cronológica reversa. Ou seja, a última fotografia registrada é a primeira a ser apresentada pelo comando **git log**. No início de cada linha vemos uma parte do hash de cada fotografia. Em geral, essa parte do hash é suficiente para visualizar ou acessar uma fotografia. Por exemplo, é possível ver uma fotografia mais antiga (ou mais recente) com o comando **git show <hash>**, onde normalmente, os quatro primeiros caracteres da parte do hash que aparece na listagem 28 é suficiente para identificá-la.

4.2. Mostrando o conteúdo de fotografias

Na listagem 29 o comando **git show** mostra a fotografia anterior usando apenas a parte de seu hash listada na listagem 28. Note que o resultado da listagem 29 é idêntico ao da listagem 24. Ao seguir esses passos, lembre-se de substituir o hash 'e0fe2b3' pelo

hash que aparece na sua lista de logs. Certamente terá um valor diferente.

Lista de comandos 29. Vendo detalhes da fotografia anterior

```
~/greetings$ git show e0fe2b3
commit e0fe2b32abaa5b4bcb7a1889a820f155e9ec635e
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Wed Dec 23 08:24:50 2020 -0300

    cria função main

diff --git a/greet.py b/greet.py
index 693eaec..55bb9ae 100644
--- a/greet.py
+++ b/greet.py
@@ -1,4 @@
-print("Hello!")
+def main():
+    print("Hello!")
+
+main()
```

A listagem [30](#) mostra a primeira fotografia do arquivo `greet.py` no repositório.

Lista de comandos 30. Vendo detalhes da fotografia da primeira fotografia

```
~/greetings$ git show 6c8951a
commit 6c8951a5c2979932ffaed078139616584afd8543
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Tue Dec 22 08:41:06 2020 -0300
```

```
    cria arquivo greet.py
```

```
diff --git a/greet.py b/greet.py
new file mode 100644
index 0000000..693eac
--- /dev/null
+++ b/greet.py
@@ -0,0 +1 @@
+print("Hello!")
```

O comando `git show <hash>` é muito útil para verificarmos se uma determinada fotografia contém as alteração que precisamos. Depois de confirmarmos nossas suspeitas, podemos voltar o repositório para o estado que desejarmos.

4.3. Alterando o estado do sistema

O comando `git checkout` permite colocar o repositório em um estado gravado em alguma fotografia. A listagem 31 mostra como fazer o repositório voltar para o estado em que a função `main` do programa `greet` foi criada. Algumas linhas do resultado foram substituídas por "...". Então, mais informações aparecerão quando você digitar esse comando. Ao seguir esses passos, lembre-se de substituir o hash `'e0fe2b3'` pelo hash que aparece na sua lista de logs. Certamente terá um valor diferente.

Lista de comandos 31. Voltando o sistema para o estado da fotografia anterior

```
~/greetings$ git checkout e0fe2b3
Note: switching to 'e0fe2b3'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

...

HEAD is now at e0fe2b3 cria função main

Veja na listagem [32](#) que o programa `greet.py` voltou ao seu estado anterior.

Lista de comandos 32. Estados dos arquivos do sistema depois de voltar uma fotografia

```
~/greetings$ cat greet.py
def main():
    print("Hello!")

main()
```

Ao listar as fotografias do repositório, como mostra a figura [33](#), o comando `git log` não mostra mais o branch main, nem a fotografia da última alteração feita. Além disso, **HEAD** agora está na fotografia **e0fe2b3**.

Lista de comandos 33. Listando fotografias tão ou mais antigas que a atual

```
~/greetings$ git log --oneline
e0fe2b3 (HEAD) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

Você pode estar se perguntando "Git é então um complexo ctrl+z?". Claro que não! A fotografia mais recente continua sendo monitorada e pode ser visualizada com a opção `--all` no comando `git log`, como mostra a figura 34.

Lista de comandos 34. Listando todas fotografias do repositório

```
~/greetings$ git log --oneline --all
bd61894 (main) verifica se é o programa principal
e0fe2b3 (HEAD) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

Na verdade, o Git quase sempre adiciona informação ao repositório. Mesmo sendo possível remover informações de um repositório, isso é raramente recomendado. A figura 35 mostra como colocar o sistema no estado da fotografia mais recente.

Lista de comandos 35. Voltando para versão mais recente do sistema

```
~/greetings$ git checkout main
Previous HEAD position was e0fe2b3 cria função main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)
```

Qualquer alteração nos arquivos monitorados pelo repositório deve ser feita enquanto o estado do repositório (HEAD) está apontando para um ramo (*branch*). Fotografias de alterações feitas fora de ramos não aparecem na resposta do comando `git log`, nem com a opção `--all`.`

4.4. Criando etiquetas para fotografias

Para facilitar o acesso das fotografias pode-se etiquetá-las. O tipo de etiqueta mais comum é mostrado na figura 36, que usa o comando `git tag` com a opção `-a`. Esta opção permite usar a opção `-m` para inserir um comentário na etiqueta.

Lista de comandos 36. Criando etiquetas para a fotografia atual

```
~/greetings$ git tag -a v0.3 -m 'Versão bem complexa
para um programa Hello World'
~/greetings$ git log --oneline --all ①
bd61894 (HEAD -> main, tag: v0.3) verifica se é o
programa principal ②
e0fe2b3 cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

- ① Lista fotografias do repositório.
- ② Mostra a tag "v0.3" na linha correspondente à fotografia.

O comando `git tag` coloca a etiqueta na fotografia atual do sistema, mas é possível etiquetar outras fotografias através de seu hash, como mostra a figura 37.

Lista de comandos 37. Etiketando uma fotografia mais antiga

```
~/greetings$ git tag -a v0.1 -m 'Versão clássica'
6c8951a
~/greetings$ git log --oneline --all
bd61894 (HEAD -> main, tag: v0.3) verifica se é o
programa principal
e0fe2b3 cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a (tag: v0.1) cria arquivo greet.py
58da81b Initial commit
```

Lista de comandos 38. Etiketando uma fotografia sem deixar comentário

```
~/greetings$ git tag v0.2 e0fe2b3
~/greetings$ git log --oneline --all
bd61894 (HEAD -> main, tag: v0.3) verifica se é o
programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a (tag: v0.1) cria arquivo greet.py
58da81b Initial commit
```

A figura 9 mostra como acessar uma fotografia antiga através de sua etiqueta

Lista de comandos 39. Acessando uma fotografia antiga através da sua etiqueta

```
~/greetings$ git checkout v0.2
Note: switching to 'v0.2'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

```
...
HEAD is now at e0fe2b3 cria função main
```

Lista de comandos 40. git log

```
~/greetings$ git log --oneline --all
bd61894 (tag: v0.3, main) verifica se é o programa principal
e0fe2b3 (HEAD, tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a (tag: v0.1) cria arquivo greet.py
58da81b Initial commit
```

O comando **git tag** pode ser usado para listar as etiqueta, como mostra a figura 40.

Lista de comandos 41. Listando todas as etiquetas do repositório

```
~/greetings$ git tag
v0.1
v0.2
v0.3
```

Listagens mais complexas, com caracteres coringa por exemplo,

podem ser feitas com esse comando, mas não serão exploradas aqui.

Quando se executa o comando `git show` com uma etiqueta, ele mostra também os dados da etiqueta, como pode ser visto na figura 41. A informação de quem fez a etiqueta (tagger) e de quando a etiqueta foi criada só é gravada se a opção `-a` for usada na criação dela.

Lista de comandos 42. Mostrando fotografias usando etiquetas

```
~/greetings$ git show v0.2
commit e0fe2b32abaa5b4bcb7a1889a820f155e9ec635e (tag:
v0.2)
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Wed Dec 23 08:24:50 2020 -0300

    cria função main

diff --git a/greet.py b/greet.py
index 693eaec..55bb9ae 100644
--- a/greet.py
+++ b/greet.py
@@ -1,4 @@
-print("Hello!")
+def main():
+    print("Hello!")
+
+main()
```

Lista de comandos 43. Mostrando fotografias usando etiquetas (algumas linhas foram omitidas)

```
~/greetings$ git show v0.3
tag v0.3
Tagger: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Wed Dec 23 09:11:24 2020 -0300

Versão bem complexa para um programa Hello World

commit bd618942ddf84ad2ceb062b7ef53c22b12a94dae (HEAD
-> main, tag: v0.3)
...
```

É importante notar que **HEAD** não aponta para nenhum branch. No caso, não aponta para **main**, que é o único branch do repositório. Para continuar o tutorial execute o comando da figura [\[fig:29\]](#), para que **HEAD** aponte para **main**.

Lista de comandos 44. git checkout main

```
~/greetings$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)
```

Capítulo 5. Ramos no projeto

As etiquetas são fixadas em uma fotografia, mas ramos (braches) são vivos e acompanham novas fotografias que são criadas. Há muitas formas de se usar os braches. Neste capítulo mostraremos uma delas. Também há vários motivos para se usar os branches. Um deles é que você pode inserir uma alteração instável no sistema e querer que essa alteração fique gravada. Ou seja, você fez uma alteração que não está pronta, mas quer que essa alteração seja monitorada pelo Git por algum motivo. Talvez você não tenha certeza que o próximo passo vai funcionar, ou talvez você queira testar o próximo passo de mais do que uma forma. Ou ainda, pode ser que outra pessoa termine essa atualização parcial que você fez. O fato é que você não quer que esta seja a versão usada até que ela esteja terminada.

5.1. Criando ramos

Como ilustração, faremos uma versão brasileira para o nosso programa. Como eu supostamente ainda não sei se isso será fácil ou difícil de terminar, farei um branch como mostra a figura 44. Agora a fotografia mais recente tem dois ramos (na cor verde), **main** e **pt-br**.

Lista de comandos 45. Criando um novo branch

```
~/greetings$ git branch pt-br
~/greetings$ git log --oneline --all
bd61894 (HEAD -> main, tag: v0.3, pt-br) verifica se é
o programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

Para fazer uma alteração no ramo **pt-br**, deve-se mudar **HEAD** para esse ramo, como apresentado na figura 1. Agora **HEAD** aponta para **pt-br**.

Lista de comandos 46. Acessando um branch

```
~/greetings$ git checkout pt-br
Switched to branch 'pt-br'
~/greetings$ git log --oneline --all
bd61894 (HEAD -> pt-br, tag: v0.3, main) verifica se é
o programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
~/greet
```

Quando só existia o ramo **main**, cada comando **commit** movia o ramo **main** para a fotografia mais recente. Agora que **HEAD** aponta para **pt-br**, o comando **commit** vai mover o ramo **pt-br** para as novas fotografias, deixando o ramo **main** na fotografia atual. Assim, fica claro para todos os envolvidos no projeto que o ramo **main** contém uma versão estável do sistema.

5.2. Uma alteração incompleta para o ramo atual

Como ilustração será feita a alteração proposta na figura 45.

Lista de comandos 47. Alterando o sistema no branch atual

```
~/greetings$ cat greet.py
def main():
    print("Olá!")

if __name__ == "__main__":
    main()
```

A resposta do comando `git status` da figura 46 já é conhecida. A única diferença do que já foi visto é a primeira linha que mostra que ramo atual é o **pt-br** (*On branch pt-br*).

Lista de comandos 48. Status do novo branch com arquivo modificado fora da área de stage

```
~/greetings$ git status
On branch pt-br
Changes not staged for commit:
  (use "git add <file>..." to update what will be
  committed)
  (use "git restore <file>..." to discard changes in
  working directory)
    modified:   greet.py

no changes added to commit (use "git add" and/or "git
commit -a")
```

O status após adicionar a alteração na área de stage mostrado na figura 47 também não é muito diferente do que já foi visto.

Lista de comandos 49. Status do novo branch com arquivo modificado na da área de stage

```
~/greetings$ git add greet.py
~/greetings$ git status
On branch pt-br
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   greet.py
```

O resultado dos comandos `git commit` e `git show` apresentados na figura 49 também não apresentam muita novidade.

Lista de comandos 50. Adiciona arquivo modificado na área de stage.

```
~/greetings$ git commit -m 'versão brasileira'
[pt-br 8828ea9] versão brasileira
1 file changed, 1 insertion(+), 1 deletion(-)
```

Lista de comandos 51. Fotografia da versão brasileira do sistema

```
~/greetings$ git show
commit 8828ea9256bc157d561bab306c01f304b3a54821 (HEAD
-> pt-br)
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Wed Dec 23 11:34:03 2020 -0300
```

versão brasileira

```
diff --git a/greet.py b/greet.py
index 3062fd5..66da5e9 100644
--- a/greet.py
+++ b/greet.py
@@ -1,5 +1,5 @@
 def main():
-    print("Hello!")
+    print("Olá!")

if __name__ == "__main__":
    main()
```

Note que o ramo **main** não tem nada de especial. Usar outro nome para um ramo não muda nada no processo de fotografar as versões do sistema.

Lista de comandos 52. Lista de fotografias

```
~/greetings$ git log --oneline --all
8828ea9 (HEAD -> pt-br) versão brasileira
bd61894 (tag: v0.3, main) verifica se é o programa
principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

5.3. Terminado a alteração desejada

Para mostrar como colocar uma alteração no ramo estável do sistema, vamos fazer a alteração proposta na figura 51. Estamos considerando o ramo estável deste repositório o ramo **main**, mas poderia ser qualquer outro nome.

Lista de comandos 53. Parametrizando o sistema

```
~/greetings$ cat greet.py
import sys
def main():
    if len(sys.argv)>1 and sys.argv[1]=='pt-br':
        print("Olá!")
    else:
        print("Hello!")

if __name__ == "__main__":
    main()
```

Lista de comandos 54. Como o programa está executando agora.

```
~/greetings$ python greet.py
Hello!
~/greetings$ python greet.py pt-br
Olá!
```

Depois de colocar a nova alteração na área de stage (**git add greet.py**) e executar o comando **git commit** podemos ver a nova fotografia listada na figura 53.

Lista de comandos 55. Lista das fotografias após a versão brasileira parametrizada

```
~/greetings$ git add greet.py
~/greetings$ git commit -m 'versão brasileira parametrizada'
[pt-br 2aa634b] versão brasileira parametrizada
 1 file changed, 5 insertions(+), 1 deletion(-)
~/greetings$ git log --oneline --all
2aa634b (HEAD -> pt-br) versão brasileira parametrizada
8828ea9 versão brasileira
bd61894 (tag: v0.3, main) verifica se é o programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

A figura 54 mostra como ficou a fotografia mais recente do repositório. Também mostra como executar o programa na versão mais recente, caso ache interessante.

```
~/greetings$ git show
commit 2aa634b3fe78d227bd07482dfb080154e02cc93f (HEAD
-> pt-br)
Author: Francisco de Assis Boldt <fboldt@gmail.com>
Date:   Wed Dec 23 11:49:08 2020 -0300
```

versão brasileira parametrizada

```
diff --git a/greet.py b/greet.py
index 66da5e9..3e2fb6e 100644
--- a/greet.py
+++ b/greet.py
@@ -1,5 +1,9 @@
+import sys
+def main():
+    print("Olá!")
+    if len(sys.argv)>1 and sys.argv[1]=='pt-br':
+        print("Olá!")
+    else:
+        print("Hello!")

if __name__ == "__main__":
    main()
```

5.4. Mesclando o ramo atual com o ramo principal

Agora que a alteração já foi finalizada, é hora de mesclar a atualização no ramo principal. A figura 55 apresenta um procedimento que pode ser executado com essa finalidade. Primeiro, temos que fazer **HEAD** apontar para o ramo principal com o comando **git checkout main**. Depois, usamos o comando **git merge pt-br** para mesclar o ramo **pt-br** com o ramo atual.

Lista de comandos 57. Mesclando a versão brasileira com a versão original

```
~/greetings$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)
~/greetings$ git merge pt-br
Updating bd61894..2aa634b
Fast-forward
 greet.py | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)
```

A figura 56 mostra a lista de fotografias depois da mesclagem de ramos.

Lista de comandos 58. Listando as fotografias do repositório após mesclar versões do sistema

```
~/greetings$ git log --oneline --all
2aa634b (HEAD -> main, pt-br) versão brasileira parametrizada
8828ea9 versão brasileira
bd61894 (tag: v0.3) verifica se é o programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

Na segunda linha da reposta do comando `git merge pt-br` na figura 55 está escrito *Fast-forward*. Isso significa que nenhuma alteração foi feita no ramo **main** enquanto o ramo **pt-br** estava sendo alterado. Assim, não houve nenhum conflito para juntar as versões porque a versão mais recente de **pt-br** era como uma versão futura de **main**. A seguir, veremos um situação que isso não é resolvido tão facilmente.

Capítulo 6. Criando bifurcações no projeto

O capítulo anterior mostrou uma mesclagem do tipo *fast-forward*, que é um tipo sem conflito. Aqui, veremos como resolver conflitos quando ele acontecem.

6.1. Criando um ramo comum

Agora faremos uma versão do sistema em alemão. Para manter uma boa prática de Git vamos criar um novo ramo, como mostra a figura 57.

Lista de comandos 59. Criando um branch para uma versão em alemão

```
~/greetings$ git branch de
~/greetings$ git log --oneline --all
2aa634b (HEAD -> main, pt-br, de) versão brasileira parametrizada
8828ea9 versão brasileira
bd61894 (tag: v0.3) verifica se é o programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
~/greetings$ git checkout de
Switched to branch 'de'
```

Para trabalhar no novo ramo, deve-se usar o comando **git checkout**. O comando **git log** mostra o ramo para o qual **HEAD** aponta.

Depois de fazer a alteração sugerida na figura 58, execute o comando **git commit** para deixar gravada as alterações no

repositório.

Lista de comandos 60. Alteração feita para versão alemã do sistema

```
~/greetings$ cat greet.py
import sys
def main():
    if len(sys.argv)>1 and sys.argv[1]=='pt-br':
        print("Olá!")
    elif len(sys.argv)>1 and sys.argv[1]=='de':
        print("Hallo!")
    else:
        print("Hello!")

if __name__ == "__main__":
    main()
```

Lista de comandos 61. Funcionamento depois da versão alemã

```
~/greetings$ python greet.py
Hello!
~/greetings$ python greet.py pt-br
Olá!
~/greetings$ python greet.py de
Hallo!
```

A fotografia do último *commit* está na figura [60](#).

Lista de comandos 62. Fotografando a versão alemã do sistema

```
~/greetings$ git add greet.py
~/greetings$ git commit -m 'versão alemã parametrizada'
[de 54a47d0] versão alemã parametrizada
1 file changed, 2 insertions(+)
```

A figura [61](#) mostra que o sistema possui seis fotografias até o

momento.

Lista de comandos 63. Listando as fotografias após a inclusão da versão alemã

```
~/greetings$ git log --oneline --all
54a47d0 (HEAD -> de) versão alemã parametrizada
2aa634b (pt-br, main) versão brasileira parametrizada
8828ea9 versão brasileira
bd61894 (tag: v0.3) verifica se é o programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

Na condição atual, no exemplo do capítulo anterior, o ramo foi mesclado com o ramo estável. Mas para exemplificar um conflito, não vamos mesclá-lo agora. Um motivo para não mesclar é não ter feito todos os testes no seu ramo. Ou o ramo ainda não está terminado. Vamos supor as nossas situações hipotéticas que não temos certeza que a resposta correta em alemão é *hallo*. Por isso, vamos adiar a mesclagem com o ramo principal.

6.2. Criando mais um ramo comum

Normalmente, conflitos de mesclagem não são criados intencionalmente. Mas para ilustrar a resolução de conflitos que inevitavelmente acontecerão, vamos fazer uma versão em italiano do nosso sistema para forçar um conflito. Para isso, vamos começar criando um novo ramo a partir de **main**. Após fotografarmos a alteração do sistema com a versão italiana, teremos dois ramos que nasceram a partir de **main**. Um deles é facilmente mesclável. O outro, nem tanto.

Uma forma de se criar um ramo a partir de **main** é estando com **HEAD** apontando para **main**. Em seguida, usa-se o comando **git**

branch para criar um novo branch, como na figura 62.

Lista de comandos 64. Criando um branch para implementar uma versão italiana a partir da versão brasileira

```
~/greetings$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)
~/greetings$ git branch it
~/greetings$ git checkout it
Switched to branch 'it'
```

Implemente a alteração sugerida na figura 63.

Lista de comandos 65. Implementando a versão italiana

```
~/greetings$ cat greet.py
import sys
def main():
    if len(sys.argv)>1 and sys.argv[1]=='pt-br':
        print("Olá!")
    elif len(sys.argv)>1 and sys.argv[1]=='it':
        print("Ciao!")
    else:
        print("Hello!")

if __name__ == "__main__":
    main()
```

Adicione as alterações na área de stage e execute o comando **git commit** para fazer a nova fotografia.

Após adicionar as alterações na área de stage e executar o comando **git commit** a fotografia mais atual deverá estar parecida com a da figura 64.

```
~/greetings$ git add greet.py
~/greetings$ git commit -m 'versão italiana parametrizada'
[it 46244be] versão italiana parametrizada
 1 file changed, 2 insertions(+)
~/greetings$ git log --oneline --all
46244be (HEAD -> it) versão italiana parametrizada
54a47d0 (de) versão alemã parametrizada
2aa634b (pt-br, main) versão brasileira parametrizada
8828ea9 versão brasileira
bd61894 (tag: v0.3) verifica se é o programa principal
e0fe2b3 (tag: v0.2) cria função main
933fc74 (origin/main, origin/HEAD) Update README.md
6c8951a cria arquivo greet.py
58da81b Initial commit
```

6.3. Listando as fotografias em forma de grafo

A opção `--graph` do comando `git log` lista as fotografias do repositório em forma de grafo, como na figura [65](#).

Lista de comandos 67. Listando todas as fotografias do repositório em forma de grafo

```
~/greetings$ git log --oneline --all --graph
* 46244be (HEAD -> it) versão italiana parametrizada
| * 54a47d0 (de) versão alemã parametrizada
|/
* 2aa634b (pt-br, main) versão brasileira parametrizada
* 8828ea9 versão brasileira
* bd61894 (tag: v0.3) verifica se é o programa
principal
* e0fe2b3 (tag: v0.2) cria função main
* 933fc74 (origin/main, origin/HEAD) Update README.md
* 6c8951a cria arquivo greet.py
* 58da81b Initial commit
```

Note que acima do ramo **main** as linhas estão vermelhas, indicando um possível conflito. Observe que o ramo **de** (alemão), que é mais antigo que o ramo **it** (italiano), se mostra como um ramo que está saindo de um galho.

6.4. Mesclando o último ramo antes do primeiro

O último ramo criado foi o ramo **it**, mas aqui vamos mesclá-lo ao ramo principal antes do ramo mais antigo, que é o ramo **de**. A figura 66 mostra uma forma de como isso pode ser feito. Ocorreu uma mesclagem do tipo *fast-forward* sem nenhum problema.

Lista de comandos 68. Mesclando a versão italiana com a principal

```
~/greetings$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)
~/greetings$ git merge it
Updating 2aa634b..46244be
Fast-forward
 greet.py | 2 ++
 1 file changed, 2 insertions(+)
```

Na figura [67](#) podemos ver que o grafo não foi alterado, mas agora **HEAD** e **main** estão na fotografia mais recente.

Lista de comandos 69. Listando todas as fotografias do repositório em forma de grafo após mesclar a versão italiana

```
~/greetings$ git log --oneline --all --graph
* 46244be (HEAD -> main, it) versão italiana
parametrizada
| * 54a47d0 (de) versão alemã parametrizada
|/
* 2aa634b (pt-br) versão brasileira parametrizada
* 8828ea9 versão brasileira
* bd61894 (tag: v0.3) verifica se é o programa
principal
* e0fe2b3 (tag: v0.2) cria função main
* 933fc74 (origin/main, origin/HEAD) Update README.md
* 6c8951a cria arquivo greet.py
* 58da81b Initial commit
```

6.5. Quando não corre tudo bem na mesclagem

Agora, veja figura 68 o que ocorre quando tentamos mesclar o ramo alemão com o ramo principal.

Lista de comandos 70. Mesclando a versão alemã com a principal

```
~/greetings$ git merge de
Auto-merging greet.py
CONFLICT (content): Merge conflict in greet.py
Automatic merge failed; fix conflicts and then commit
the result.
```

Lista de comandos 71. Como ficou o arquivo

```
~/greetings$ cat greet.py
import sys
def main():
    if len(sys.argv)>1 and sys.argv[1]=='pt-br':
        print("Olá!")
<<<<<< HEAD
    elif len(sys.argv)>1 and sys.argv[1]=='it':
        print("Ciao!")
=====
    elif len(sys.argv)>1 and sys.argv[1]=='de':
        print("Hallo!")
>>>>>> de
    else:
        print("Hello!")

if __name__ == "__main__":
    main()
```

Conforme a figura 69, a ferramenta **vimdiff** pode ser selecionada como na figura 70.

Lista de comandos 72. status

```
~/greetings$ git status
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   greet.py

no changes added to commit (use "git add" and/or "git
commit -a")
```

Lista de comandos 73. Como ficou o arquivo

```
~/greetings$ cat greet.py
import sys
def main():
    if len(sys.argv)>1 and sys.argv[1]=='pt-br':
        print("Olá!")
    elif len(sys.argv)>1 and sys.argv[1]=='it':
        print("Ciao!")
    elif len(sys.argv)>1 and sys.argv[1]=='de':
        print("Hallo!")
    else:
        print("Hello!")

if __name__ == "__main__":
    main()
```

Lista de comandos 74. Resultado

```
~/greetings$ git add greet.py
~/greetings$ git commit -m 'merge com versão alemã'
[main 4377f73] merge com versão alemã
~/greetings$ git log --oneline --all --graph
*   4377f73 (HEAD -> main) merge com versão alemã
| \
|  * 54a47d0 (de) versão alemã parametrizada
* | 46244be (it) versão italiana parametrizada
|/
* 2aa634b (pt-br) versão brasileira parametrizada
* 8828ea9 versão brasileira
* bd61894 (tag: v0.3) verifica se é o programa principal
* e0fe2b3 (tag: v0.2) cria função main
* 933fc74 (origin/main, origin/HEAD) Update README.md
* 6c8951a cria arquivo greet.py
* 58da81b Initial commit
```

Lista de comandos 75. Programa

```
~/greetings$ python greet.py
Hello!
~/greetings$ python greet.py pt-br
Olá!
~/greetings$ python greet.py it
Ciao!
~/greetings$ python greet.py de
Hallo!
```

Lista de comandos 76. Push

```
~/greetings$ git push
Username for 'https://github.com': fboldt
Password for 'https://fboldt@github.com':
Enumerating objects: 23, done.
Counting objects: 100% (23/23), done.
Delta compression using up to 8 threads
Compressing objects: 100% (20/20), done.
Writing objects: 100% (21/21), 2.14 KiB | 2.14 MiB/s,
done.
Total 21 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/fboldt/greetings.git
933fc74..4377f73  main -> main
```

Lista de comandos 77. Lista final

```
~/greetings$ git log --oneline --all --graph
* 4377f73 (HEAD -> main, origin/main, origin/HEAD)
merge com versão alemã
| \
| * 54a47d0 (de) versão alemã parametrizada
* | 46244be (it) versão italiana parametrizada
| /
* 2aa634b (pt-br) versão brasileira parametrizada
* 8828ea9 versão brasileira
* bd61894 (tag: v0.3) verifica se é o programa
principal
* e0fe2b3 (tag: v0.2) cria função main
* 933fc74 Update README.md
* 6c8951a cria arquivo greet.py
* 58da81b Initial commit
```

Capítulo 7. Conclusão

Agora, você já sabe uma forma de se usar o Git. A forma apresentada aqui não é a única, nem a melhor. É um exemplo para ser aplicado imediatamente. É claro que um projeto real, que necessite de um gerenciador de versões, possivelmente terá mais arquivos no que o exemplo `hello world` apresentado aqui. Porém, trabalhar com mais arquivos pode facilitar o gerenciamento das versões. Conflitos geralmente ocorrem quando o mesmo arquivo sofre alterações em ramos diferentes.

Bibliografia

1. Ryan Hodson. *Ry's Git Tutorial*. RyPress. 2014.
2. Scott Chacon & Ben Straub. *Pro Git*. Spring Nature. 2014.