Hello Git

Um tutorial para usar o Git imediatamente

Francisco de Assis Boldt <boldt.pro.br>

Version 0.1 2020-12

Sumário

1. O hello world do Git . . . 1
    1.1. Iniciando um repositório . . . 1
    1.2. O comando `git status` . . . 2
    1.3. Criando e monitorando um arquivo . . . 3
2. Criando fotografias novas e acessando fotografias antigas . . . 6
    2.1. Listando as fotografias do repositório . . . 8
    2.2. Mostrando o conteúdo de fotografias . . . 8
    2.3. Alterando o estado do sistema . . . 9
    2.4. Criando etiquetas para fotografias . . . 11
3. Ramos no projeto . . . 14
    3.1. Criando ramos . . . 14
    3.2. Uma alteração incompleta para o ramos atual . . . 14
    3.3. Terminado a alteração desejada . . . 16
    3.4. Mesclando o ramo atual com o ramo principal . . . 17
4. Criando bifurcações no projeto . . . 19
    4.1. Criando um ramo comum . . . 19
    4.2. Criando mais um ramo comum . . . 20
    4.3. Listando as fotografias em forma de grafo . . . 22
    4.4. Mesclando o último ramo antes do primeiro . . . 23
    4.5. Quando não corre tudo bem na mesclagem . . . 25
Bibliografia . . . 30

Capítulo 1. O hello world do Git

Este tutorial apresenta uma forma muito simples de se usar o Git. Como exemplo, serão 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 Git em 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 (1) e (2). 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 edição 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. Iniciando um repositório

Fisicamente no computador, um repositório Git é apenas um diretório com algumas metainformações sobre as alterações dos arquivos do projeto que está no diretório. A figura 1 mostra como criar um repositório Git do zero. O comando `mkdir greetings` cria o diretório onde colocaremos o código do nosso projeto de exemplo e o comando `cd greetings` acessa o diretório criado. Pode-se ver pelo comando `ls -a` que ainda não existe nenhum arquivo no diretório. Para transformar esse diretório vazio em um repositório Git basta executar o comando `git init`, que o repositório será iniciado, conforme mostra a repetição do comando `ls -a`.

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

Figura 1. Criando um repositório

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`. A árvore com todos os arquivos e subdiretórios criado pelo comando `git init` pode ser vista na figura 2. O comando `tree .git` foi usado para exibir essa árvore. 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.

```
~/greetings$ tree .git
.git
├── branches
├── config
├── description
├── HEAD
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── fsmonitor-watchman.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-merge-commit.sample
│   ├── prepare-commit-msg.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags

9 directories, 16 files
```

Figura 2. Árvore de diretórios do repositório

1.2. O comando `git status`

Um comando que é usado o tempo todo em um repositório Git é o comando `git status` apresentado na figura 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.

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

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Figura 3. Status de um repositório recém criado

A primeira linha de resposta 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. Particularmente, eu nunca vi um projeto Git sem o ramo master, mas não existe nenhuma exigência de que esse ramo exista. É que normalmente não se muda esse nome.

A segunda linha é `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.

A terceira linha é `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.

1.3. Criando e monitorando um arquivo

Como pode ser traduzido da terceira linha de resposta da figura 3, o Git monitora (*track*) arquivos criados ou copiados para dentro do diretório do repositório. A figura 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 em 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. A execução do arquivo com o comando `python greed.py` é opcional. O comando `ls -a` é só para mostrar que o arquivo `greed.py` foi criado.

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

Figura 4. Criando o primeiro arquivo do projeto

Depois de criado o arquivo `greet.py`, o comando `git status` mostrará uma resposta um pouco diferente, como mostra a figura figura 5. As duas primeiras linhas de resposta continuam iguais, mais agora, a terceira linha de resposta lista o arquivo `greet.py` em vermelho, e diz que ele não está sendo monitorado (*untracked*). Veja que a resposta do comando já diz como adicionar a modificação no monitoramento (`git add <file>`). 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.

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

No commits yet

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)
```

Figura 5. Status com arquivo fora da área de stage

A figura 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 listado em verde, 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`.

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

No commits yet

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

Figura 6. Status com arquivo na da área de stage

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 figura 7. Entretanto, 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'`. 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 `-m` para ficar mais resumido e visível através das figuras.

```
~/greetings$ git commit -m 'primeira fotografia do sistema'
[master (root-commit) 5c337fc] primeira fotografia do sistema
 1 file changed, 1 insertion(+)
 create mode 100644 greet.py
```

Figura 7. Primeira fotografia do repositório

Se você quiser ver a fotografia tirada do sistema você pode usar o comando `git show` e terá um resultado parecido com o da figura 8. Vamos entender essa fotografia, mas sem seguir a ordem em que os dados aparecem. Vemos o comentário inserido pelo comando `git commit`. Também vemos a data e hora de quando o commit foi executado, que pode ser entendida como o momento da fotografia. A linha que começa com `Author` tem os dados inseridos pelos comandos `git config <etc>`. Em negrito, está indicado que essa é uma fotografia que contém um arquivo novo, e a linha verde que começa com
é o conteúdo do arquivo.

```
~/greetings$ git show
commit 5c337fc459bd0c0be453b551255312f0067ee961 (HEAD -> master)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:14 2020 -0300

    primeira fotografia do sistema

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

Figura 8. Vendo detalhes da fotografia mais recente do sistema

Na primeira linha, em amarelo, 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, em negrito e verde, temos a palavra **master**, indicando que o ramo do projeto chamado **master** está apontando para esta fotografia no momento. Ainda na mesma linha, em azul, temos a palavra **HEAD** seguida dos sinais de $-$ e $>$ representando uma seta (→). Esta seta indica que o estado do sistema que estamos vendo no momento está apontando para o ramos master. Isso ficará mais claro a seguir.

Capítulo 2. 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 figura 9 mostra a alteração sugerida. O resultado do programa continuou quase igual, por isso a palavra "hello" foi colocada toda em maiúsculo para ficar mais clara que uma alteração foi feita.

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

main()
~/greetings$ python greet.py
HELLO
~/greetings$ ls -a
.  ..  .git  greet.py
```

Figura 9. Fazendo uma alteração

Depois da alteração do arquivo, o comando `git status apresenta um retorno diferente, como mostra a figura 10. Novamente o arquivo `greet.py` está em vermelho por não estar na área de stage, mas agora esse arquivo está sendo monitorado. Então temos duas opções. Podemos descartar as alterações com o comando `git restore greet.py` ou podemos adicionar as alterações na área de stage com o comando `git add greet.py`. Adicionaremos as alterações na área de stage, como mostra a figura 11.

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

Figura 10. Status com arquivo modificado fora da area de stage

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

Figura 11. Status com arquivo modificado na da area de stage

O resultado do comando `git status` está muito parecido com o da figura 6. Agora, em verde, não aparece mais "arquivo novo" (new file), mas "modificado" (modified).

A figura 12 mostra o comando `git commit` com o parâmetro `-m` e um comentário relacionado à alteração feita. O comando `git show` mostra como ficou a fotografia. A linha em vermelho que inicia com o sinal $-$ mostra o que foi removido, e as linhas em verde que iniciam com o sinal $+$ mostram o que foi adicionado.

```
~/greetings$ git commit -m 'criação da função main'
[master 76c7a22] criação da função main
 1 file changed, 4 insertions(+), 1 deletion(-)
~/greetings$ git show
commit 76c7a2237d5875db7cc395672e095b0c13650049 (HEAD -> master)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:15 2020 -0300

    criação da função main

diff --git a/greet.py b/greet.py
index 11b15b1..9a5e780 100644
--- a/greet.py
+++ b/greet.py
@@ -1 +1,4 @@
-print("hello")
+def main():
+  print("HELLO")
+
+main()
```

Figura 12. Atualizando o repositório e vendo os detalhes da atualização

Vamos fazer mais uma alteração no sistema, que pode ser vista na figura 13. Novamente, o resultado do programa é virtualmente o mesmo, e para que a alteração seja um pouco mais evidente, a palavra *Hello* foi colocada agora apenas com a primeira letra em maiúsculo.

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

if __name__ == "__main__":
  main()
~/greetings$ python greet.py
Hello
~/greetings$ ls -a
.  ..  .git  greet.py
```

Figura 13. Fazendo mais uma alteração

Depois dessa alteração, o comando `git status` apresentará o mesmo retorno visto na figura 10. 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 figura 11.

Agora estamos prontos para executar o comando `commit` como mostra a figura 14. Novamente podem ser vistas as alterações feitas observando-se as linhas verdes e vermelhas.

```
~/greetings$ git commit -m 'um pouco mais sofisticado'
[master e86b0d6] um pouco mais sofisticado
 1 file changed, 3 insertions(+), 2 deletions(-)
~/greetings$ git show
commit e86b0d6fc58de36e8d4a0baa9777904f7d2d274d (HEAD -> master)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:17 2020 -0300

    um pouco mais sofisticado

diff --git a/greet.py b/greet.py
index 9a5e780..5a90b2d 100644
--- a/greet.py
+++ b/greet.py
@@ -1,4 +1,5 @@
 def main():
-  print("HELLO")
+  print("Hello")

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

Figura 14. Atualizando com a terceira alteração

Agora temos cópias seguras das versões anteriores do nosso projeto.

2.1. Listando as fotografias do repositório

A figura 15 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
e86b0d6 (HEAD -> master) um pouco mais sofisticado
76c7a22 criação da função main
5c337fc primeira fotografia do sistema
```

Figura 15. Listando as fotografias do repositório

As fotografias do repositório são apresentadas em ordem cronológica reversa. Ou seja, a última fotografia é a primeira a ser apresentada e a primeira fotografia é a última. Em amarelo vemos o hash de cada fotografia. Normalmente, essa parte do hash é suficiente para acessar a fotografia. Por exemplo, é possível ver uma fotografia mais antiga (ou mais recente) com o comando `git show <hash>`, onde normalmente a parte do hash que aparece na figura 15 é suficiente para identificá-la.

2.2. Mostrando o conteúdo de fotografias

Na figura 16 o comando `git show mostra a fotografia anterior usando apenas a parte de seu hash listada na figura 15.

```
~/greetings$ git show 76c7a22
commit 76c7a2237d5875db7cc395672e095b0c13650049
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:15 2020 -0300

    criação da função main

diff --git a/greet.py b/greet.py
index 11b15b1..9a5e780 100644
--- a/greet.py
+++ b/greet.py
@@ -1 +1,4 @@
-print("hello")
+def main():
+  print("HELLO")
+
+main()
```

Figura 16. Vendo detalhes da fotografia anterior

A figura 17 mostra a primeira fotografia do repositório.

```
~/greetings$ git show 5c337fc
commit 5c337fc459bd0c0be453b551255312f0067ee961
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:14 2020 -0300

    primeira fotografia do sistema

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

Figura 17. Vendo detalhes da fotografia da primeira fotografia

2.3. Alterando o estado do sistema

O comando `git checkout` permite colocar o repositório em um estado gravado em alguma fotografia. A figura 18 mostra como fazer o repositório voltar para o estado em que a função `main foi criada.

```
~/greetings$ git checkout 76c7a22
Note: switching to '76c7a22'.

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.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 76c7a22 criação da função main
```

Figura 18. Voltando o sistema para o estado da fotografia anterior

Veja na figura 19 que o programa `greet.py voltou ao seu estado anterior.

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

main()
~/greetings$ python greet.py
HELLO
```

Figura 19. Estados dos arquivos do sistema depois de voltar uma fotografia

Ao listar as fotografias do repositório, como mostra a figura 20, o comando `git log` não mostra mais o brach master, nem a fotografia da última alteração feita. Além disso, **HEAD** agora está na fotografia da segunda alteração.

```
~/greetings$ git log --oneline
76c7a22 (HEAD) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 20. Listando fotografias tão ou mais antigas que a atual

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 21.

```
~/greetings$ git log --oneline --all
e86b0d6 (master) um pouco mais sofisticado
76c7a22 (HEAD) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 21. Listando todas fotografias do repositório

Na verdade, o Git sempre adiciona informação ao repositório. Mesmo sendo possível remover informações de um repositório, isso é raramente recomendado.

A figura 22 mostra como colocar o sistema no estado da fotografia mais recente.

```
~/greetings$ git checkout e86b0d6
Previous HEAD position was 76c7a22 criação da função main
HEAD is now at e86b0d6 um pouco mais sofisticado
~/greetings$ git log --oneline
e86b0d6 (HEAD, master) um pouco mais sofisticado
76c7a22 criação da função main
5c337fc primeira fotografia do sistema
```

Figura 22. Voltando para versão mais recente do sistema

Observe que **HEAD** não está mais apontando para **master**. **HEAD** sempre estará no estado atual do repositório. Mas isso não significa que o estado atual é o mais recente.

2.4. Criando etiquetas para fotografias

Para facilitar o acesso das fotografias pode-se etiquetá-las. O tipo de etiqueta mais comum é mostrado na figura 23, que usa o comando `git tag` com a opção `-a`. Esta opção permite usar a opção `-m` para inserir um comentário na etiqueta.

```
~/greetings$ git tag -a sofisticada -m 'Hello world mais sofisticado que o n
ecessário'
~/greetings$ git log --oneline
e86b0d6 (HEAD, tag: sofisticada, master) um pouco mais sofisticado
76c7a22 criação da função main
5c337fc primeira fotografia do sistema
```

Figura 23. Criando etiquetas para a fotografia atual

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 24.

```
~/greetings$ git tag funcao 76c7a22
~/greetings$ git log --oneline
e86b0d6 (HEAD, tag: sofisticada, master) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 24. Etiquetando uma fotografia mais antiga

A figura 25 mostra coma acessar uma fotografia antiga através de sua etiqueta

```
~/greetings$ git checkout funcao
Previous HEAD position was e86b0d6 um pouco mais sofisticado
HEAD is now at 76c7a22 criação da função main
~/greetings$ git log --oneline --all
e86b0d6 (tag: sofisticada, master) um pouco mais sofisticado
76c7a22 (HEAD, tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 25. Acessando uma fotografia antiga através da sua etiqueta

O comando `git tag` pode ser usado para listar as etiqueta, como mostra a figura 26.

```
~/greetings$ git tag
funcao
sofisticada
```
Figura 26. Listando todas as etiquetas do repositório

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 27. 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.

```
~/greetings$ git show sofisticada
tag sofisticada
Tagger: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:18 2020 -0300

Hello world mais sofisticado que o necessário

commit e86b0d6fc58de36e8d4a0baa9777904f7d2d274d (HEAD, tag: sofisticada, mas
ter)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:17 2020 -0300

    um pouco mais sofisticado

diff --git a/greet.py b/greet.py
index 9a5e780..5a90b2d 100644
--- a/greet.py
+++ b/greet.py
@@ -1,4 +1,5 @@
 def main():
-  print("HELLO")
+  print("Hello")

-main()
+if __name__ == "__main__":
+  main()
```
Figura 27. Mostrando fotografias usando etiquetas

Para acessar a fotografia mais recente podemos usar o comando mostrada na figura 28.

```
~/greetings$ git checkout sofisticada
Previous HEAD position was 76c7a22 criação da função main
HEAD is now at e86b0d6 um pouco mais sofisticado
~/greetings$ git log --oneline --all
e86b0d6 (HEAD, tag: sofisticada, master) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```
Figura 28. Acessando a fotografia mais recente através da sua etiqueta

É importante notar que **HEAD** não aponta para nenhum branch. No caso, não aponta para **master**, que é o único branch do repositório. Para continuar o tutorial execute o comando da figura 29, para que **HEAD** aponte para **master**.

```
~/greetings$ git checkout master
Switched to branch 'master'
```

Figura 29. Acessando o branch master

Visto que este é um tutorial superficial, não será explicado o motivo deste procedimento.

Capítulo 3. 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.

3.1. 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 30. Agora a fotografia mais recente tem dois ramos (na cor verde), **master** e **pt-br**.

```
~/greetings$ git branch pt-br
~/greetings$ git log --oneline --all
e86b0d6 (HEAD -> master, tag: sofisticada, pt-br) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 30. Criando um novo branch

Para fazer um alteração no ramo **pt-br**, deve-se mudar **HEAD** para esse ramo, como apresentado na figura 31. Agora **HEAD** aponta para **pt-br**.

```
~/greetings$ git checkout pt-br
Switched to branch 'pt-br'
~/greetings$ git log --oneline --all
e86b0d6 (HEAD -> pt-br, tag: sofisticada, master) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 31. Acessando um branch

Quando só existia o ramo **master**, cada comando `commit` movia o ramo **master** 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 **master** na fotografia atual. Assim, fica claro para todos os envolvidos no projeto que o ramo **master** contém uma versão estável do sistema.

3.2. Uma alteração incompleta para o ramos atual

Como ilustração será feita a alteração proposta na figura 32.

```
~/greetings$ cat greet.py
def main():
    print("Oi, tudo bem?")

if __name__ == "__main__":
  main()
~/greetings$ python greet.py
Oi, tudo bem?
~/greetings$ ls -a
.  ..  .git  greet.py
```

Figura 32. Alterando o sistema no branch atual

A resposta do comando `git status` da figura 33 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*).

```
~/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")
```

Figura 33. Status do novo branch com arquivo modificado fora da área de stage

O status após adicionar a alteração na área de stage mostrado na figura 34 também não é muito diferente do que já foi visto.

```
~/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
```

Figura 34. Status do novo branch com arquivo modificado na da área de stage

O resultado dos comandos `git commit` e `git show` apresentados na figura 35 também não apresentam muita novidade.

```
~/greetings$ git commit -m 'versão brasileira'
[pt-br 3767329] versão brasileira
 1 file changed, 1 insertion(+), 1 deletion(-)
~/greetings$ git show
commit 37673294806a0a4a8e0ce3110e38abfb0a7a6a96 (HEAD -> pt-br)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:18 2020 -0300

    versão brasileira

diff --git a/greet.py b/greet.py
index 5a90b2d..f9d73d3 100644
--- a/greet.py
+++ b/greet.py
@@ -1,5 +1,5 @@
 def main():
-  print("Hello")
+    print("Oi, tudo bem?")

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

Figura 35. Fotografia da versão brasileira do sistema

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

3.3. Terminado a alteração desejada

Para mostrar como colocar uma alteração no ramo estável do sistema, vamos fazer a alteração proposta na figura 36. Estamos considerando o ramo estável deste repositório o ramo **master**, mas poderia ser qualquer outro nome.

```
~/greetings$ cat greet.py
import sys

def main():
  if len(sys.argv) > 1 and sys.argv[1] == 'pt-br':
    print("Oi, tudo bem?")
  else:
    print("Hello!")

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

Figura 36. Parametrizando o sistema

Depois de colocar a nova alteração na área de stage e executar o comando `commit` podemos ver a nova fotografia listada na figura 37.

```
~/greetings$ git log --oneline --all
2ffbed1 (HEAD -> pt-br) versão brasileira parametrizada
3767329 versão brasileira
e86b0d6 (tag: sofisticada, master) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 37. Lista das fotografias após a versão brasileira parametrizada

A figura 38 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 2ffbed1645ee4dee7dfe6239c7377a5192371779 (HEAD -> pt-br)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:19 2020 -0300

    versão brasileira parametrizada

diff --git a/greet.py b/greet.py
index f9d73d3..c16b5ad 100644
--- a/greet.py
+++ b/greet.py
@@ -1,5 +1,10 @@
+import sys
+
 def main():
+  if len(sys.argv) > 1 and sys.argv[1] == 'pt-br':
     print("Oi, tudo bem?")
+  else:
+    print("Hello!")

 if __name__ == "__main__":
   main()
~/greetings$ python greet.py pt-br
Oi, tudo bem?
```

Figura 38. Fotografia da versão brasileira atualizada

3.4. 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 39 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 master`. Depois, usamos o comando `git merge pt-br` para mesclar o ramo **pt-br** com o ramo atual.

```
~/greetings$ git checkout master
Switched to branch 'master'
~/greetings$ git merge pt-br
Updating e86b0d6..2ffbed1
Fast-forward
 greet.py | 7 ++++++-
 1 file changed, 6 insertions(+), 1 deletion(-)
```

Figura 39. Mesclando a versão brasileira com a versão original

A figura 40 mostra a lista de fotografias depois da mesclagem de ramos.

```
~/greetings$ git log --oneline --all
2ffbed1 (HEAD -> master, pt-br) versão brasileira parametrizada
3767329 versão brasileira
e86b0d6 (tag: sofisticada) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 40. Listando as fotografias do repositório após mesclar versões do sistema

Na segunda linha da reposta do comando `git merge pt-br` na figura 39 está escrito *Fast-forward*. Isso significa que nenhuma alteração foi feita no ramo **master** 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 **master**. A seguir, veremos um situação que isso não é resolvido tão facilmente.

Capítulo 4. 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.

4.1. 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 41.

```
~/greetings$ git branch de
~/greetings$ git log --oneline --all
2ffbed1 (HEAD -> master, pt-br, de) versão brasileira parametrizada
3767329 versão brasileira
e86b0d6 (tag: sofisticada) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 41. Criando um branch para uma versão em alemão

Para trabalhar no novo ramo, deve-se usar o comando `git checkout`. O comando `git log` mostra o ramo para o qual **HEAD** aponta. O resoltado pode ser visto na figura 42.

```
~/greetings$ git checkout de
Switched to branch 'de'
~/greetings$ git log --oneline --all
2ffbed1 (HEAD -> de, pt-br, master) versão brasileira parametrizada
3767329 versão brasileira
e86b0d6 (tag: sofisticada) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 42. Acessando o branch onde será implementada a versão alemã do sistema

Depois de fazer a alteração sugerida na figura 43, execute o comando `git commit` para deixar gravada as alterações no repositório.

```
~/greetings$ cat greet.py
import sys

def main():
  if len(sys.argv) > 1 and sys.argv[1] == 'pt-br':
    print("Oi, tudo bem?")
  elif len(sys.argv) > 1 and sys.argv[1] == 'de':
    print("Hallo!")
  else:
    print("Hello!")

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

Figura 43. Alteração feita para versão alemã do sistema

A fotografia do último *commit* está na figura 44.

```
~/greetings$ git commit -m 'versão alemã parametrizada'
[de 73a4e4b] versão alemã parametrizada
 1 file changed, 2 insertions(+)
~/greetings$ git show
commit 73a4e4bfba61d6b79a4de56d67fbbcead7c0e673 (HEAD -> de)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:21 2020 -0300

    versão alemã parametrizada

diff --git a/greet.py b/greet.py
index c16b5ad..c01e31b 100644
--- a/greet.py
+++ b/greet.py
@@ -3,6 +3,8 @@ import sys
 def main():
   if len(sys.argv) > 1 and sys.argv[1] == 'pt-br':
     print("Oi, tudo bem?")
+  elif len(sys.argv) > 1 and sys.argv[1] == 'de':
+    print("Hallo!")
   else:
     print("Hello!")
```

Figura 44. Fotografando a versão alemã do sistema

A figura 45 mostra que o sistema possui seis fotografias até o momento.

```
~/greetings$ git log --oneline --all
73a4e4b (HEAD -> de) versão alemã parametrizada
2ffbed1 (pt-br, master) versão brasileira parametrizada
3767329 versão brasileira
e86b0d6 (tag: sofisticada) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 45. Listando as fotografias após a inclusão da versão alemã

Na condição atual, no exemplo co 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 me nossa situações hipotética que não temos certeza que a resposta correta em alemão é *hallo*. Por isso, vamos adiar a mesclagem com o ramo principal.

4.2. Criando mais um ramo comum

Normalmente, conflitos de mesclagem não são criados intencionalemente. 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 **master**. Após fotografarmos a alteração do sistema com a versão italiana, teremos dois ramos que nasceram a partir de **master**. Um deles é facilmente mesclável. O outro, nem tanto.

Uma forma de se criar um ramo a partir de **master** é estando com **HEAD** apontando para **master**. Em seguida, usa-se o comando `git branch` para criar um novo branch, como na figura 46.

```
~/greetings$ git checkout master
Switched to branch 'master'
~/greetings$ git branch it
~/greetings$ git log --oneline --all
73a4e4b (de) versão alemã parametrizada
2ffbed1 (HEAD -> master, pt-br, it) versão brasileira parametrizada
3767329 versão brasileira
e86b0d6 (tag: sofisticada) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 46. Criando um branch para implementar uma versão italiana a partir da versão brasileira

Note que **HEAD** continua apontando para **master**. Portanto é necessário mover **HEAD** para o novo ramo, como figura 47.

```
~/greetings$ git checkout it
Switched to branch 'it'
~/greetings$ git log --oneline --all
73a4e4b (de) versão alemã parametrizada
2ffbed1 (HEAD -> it, pt-br, master) versão brasileira parametrizada
3767329 versão brasileira
e86b0d6 (tag: sofisticada) um pouco mais sofisticado
76c7a22 (tag: funcao) criação da função main
5c337fc primeira fotografia do sistema
```

Figura 47. Acessando o branch da versão italiana

Implemente a alteração sugerida na figura 48.

```
~/greetings$ cat greet.py
import sys

def main():
  if len(sys.argv) > 1 and sys.argv[1] == 'pt-br':
    print("Oi, tudo bem?")
  elif len(sys.argv) > 1 and  sys.argv[1] == 'it':
    print("Ciao!")
  else:
    print("Hello!")

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

Figura 48. Implementando a versão italiana

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 49.

```
~/greetings$ git show
commit 52a35b1d794b6efc2ece7d4364975184d60179e9 (HEAD -> it)
Author: Fulano de Tal <fulano@provedor.com>
Date:   Wed Jul 29 21:07:22 2020 -0300

    versão italiana parametrizada

diff --git a/greet.py b/greet.py
index c16b5ad..e3a9d75 100644
--- a/greet.py
+++ b/greet.py
@@ -3,6 +3,8 @@ import sys
 def main():
   if len(sys.argv) > 1 and sys.argv[1] == 'pt-br':
     print("Oi, tudo bem?")
+  elif len(sys.argv) > 1 and  sys.argv[1] == 'it':
+    print("Ciao!")
   else:
     print("Hello!")
```

Figura 49. Visualizando a fotografia da versão italiana

4.3. 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 50.

```
~/greetings$ git log --all --graph
* commit 52a35b1d794b6efc2ece7d4364975184d60179e9 (HEAD -> it)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:22 2020 -0300
|
|     versão italiana parametrizada
|
| * commit 73a4e4bfba61d6b79a4de56d67fbbcead7c0e673 (de)
|/  Author: Fulano de Tal <fulano@provedor.com>
|   Date:   Wed Jul 29 21:07:21 2020 -0300
|
|       versão alemã parametrizada
|
* commit 2ffbed1645ee4dee7dfe6239c7377a5192371779 (pt-br, master)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:19 2020 -0300
|
|     versão brasileira parametrizada
|
* commit 37673294806a0a4a8e0ce3110e38abfb0a7a6a96
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:18 2020 -0300
|
|     versão brasileira
|
* commit e86b0d6fc58de36e8d4a0baa9777904f7d2d274d (tag: sofisticada)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:17 2020 -0300
|
|     um pouco mais sofisticado
|
* commit 76c7a2237d5875db7cc395672e095b0c13650049 (tag: funcao)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:15 2020 -0300
|
|     criação da função main
|
* commit 5c337fc459bd0c0be453b551255312f0067ee961
  Author: Fulano de Tal <fulano@provedor.com>
  Date:   Wed Jul 29 21:07:14 2020 -0300

      primeira fotografia do sistema
```

Figura 50. Listando todas as fotografias do repositório em forma de grafo

Note que acima do ramo **master** 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.

4.4. Mesclando o último ramo antes do primeiro

O último ramo criado foi o ramo **it**, mas aqui vasmos mesclá-lo ao ramo principal antes do ramo mais antigo, que é o ramo **de**. A figura 51 mostra uma forma de como isso pode ser feito. Ocorreu uma mesclagem do tipo *fast-forward* sem nenhum problema.

```
~/greetings$ git checkout master
Switched to branch 'master'
~/greetings$ git merge it
Updating 2ffbed1..52a35b1
Fast-forward
 greet.py | 2 ++
 1 file changed, 2 insertions(+)
```

Figura 51. Mesclando a versão italiana com a principal

Na figura 52 podemos ver que o grafo não foi alterado, mas agora **HEAD** e **master** estão na fotografia mais recente.

```
~/greetings$ git log --all --graph
* commit 52a35b1d794b6efc2ece7d4364975184d60179e9 (HEAD -> master, it)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:22 2020 -0300
|
|     versão italiana parametrizada
|
| * commit 73a4e4bfba61d6b79a4de56d67fbbcead7c0e673 (de)
|/  Author: Fulano de Tal <fulano@provedor.com>
|   Date:   Wed Jul 29 21:07:21 2020 -0300
|
|       versão alemã parametrizada
|
* commit 2ffbed1645ee4dee7dfe6239c7377a5192371779 (pt-br)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:19 2020 -0300
|
|     versão brasileira parametrizada
|
* commit 37673294806a0a4a8e0ce3110e38abfb0a7a6a96
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:18 2020 -0300
|
|     versão brasileira
|
* commit e86b0d6fc58de36e8d4a0baa9777904f7d2d274d (tag: sofisticada)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:17 2020 -0300
|
|     um pouco mais sofisticado
|
* commit 76c7a2237d5875db7cc395672e095b0c13650049 (tag: funcao)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:15 2020 -0300
|
|     criação da função main
|
* commit 5c337fc459bd0c0be453b551255312f0067ee961
  Author: Fulano de Tal <fulano@provedor.com>
  Date:   Wed Jul 29 21:07:14 2020 -0300

      primeira fotografia do sistema
```

Figura 52. Listando todas as fotografias do repositório em forma de grafo após mesclar a versão italiana

4.5. Quando não corre tudo bem na mesclagem

Agora, veja figura 53 o que ocorre quando tentamos mesclar o ramo alemão com o ramo princial.

```
~/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.
```

Figura 53. Mesclando a versão alemã com a principal

O comando `git mergetool --tool-help` lista as disponíveis no seu sistema operacional (figura 54).

```
~/greetings$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following
                araxis
                vimdiff
                vimdiff2
                vimdiff3

The following tools are valid, but not currently available:
                bc
                bc3
                codecompare
                deltawalker
                diffmerge
                diffuse
                ecmerge
                emerge
                examdiff
                guiffy
                gvimdiff
                gvimdiff2
                gvimdiff3
                kdiff3
                meld
                opendiff
                p4merge
                smerge
                tkdiff
                tortoisemerge
                winmerge
                xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

Figura 54. Buscando ajuda com mergetool

Conforme a figura 54, a ferramenta `vimdiff` pode ser selecionada como na figura 55.

```
~/greetings$ git mergetool --tool=vimdiff
Merging:
greet.py

Normal merge conflict for 'greet.py':
  {local}: modified file
  {remote}: modified file
4 files to edit
```

Figura 55. Escolhendo programa para usar com mergetool

A figura 56 mostra como deve estar o arquivo `greet.py` depois de resolvidos os conflitos. A solução apresentada é apenas uma sugestão da resolução do conflito que ocorreu. Outras soluções poderiam ter resolvido o conflito também.

```
~/greetings$ cat greet.py
import sys

def main():
  if len(sys.argv) > 1 and sys.argv[1] == 'pt-br':
    print("Oi, tudo bem?")
  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()
```

Figura 56. Programa após a resolução de conflitos

A utilização do `vimdiff` excede o escopo deste tutorial, mas você pode ver como é a apresentação dessa ferramenta na figura 57.

Figura 57. vimdiff

Depois de usar uma ferremanta de solução de conflitos para solucioná-los, o comando `git commit deve ser executado para fotografar a forma como os conflitos foram solucionados.

A figura 58 mostra como os ramos separados se juntam depois da solução de conflitos.

```
~/greetings$ git commit -m 'versão alemã sem conflitos';
[master d301f55] versão alemã sem conflitos
~/greetings$ git log --all --graph
*   commit d301f552c58012124165ff438d3c09c53391a1b2 (HEAD -> master)
|\  Merge: 52a35b1 73a4e4b
| | Author: Fulano de Tal <fulano@provedor.com>
| | Date:   Wed Jul 29 21:07:23 2020 -0300
| |
| |     versão alemã sem conflitos
| |
| * commit 73a4e4bfba61d6b79a4de56d67fbbcead7c0e673 (de)
| | Author: Fulano de Tal <fulano@provedor.com>
| | Date:   Wed Jul 29 21:07:21 2020 -0300
| |
| |     versão alemã parametrizada
| |
* | commit 52a35b1d794b6efc2ece7d4364975184d60179e9 (it)
|/  Author: Fulano de Tal <fulano@provedor.com>
|   Date:   Wed Jul 29 21:07:22 2020 -0300
|
|       versão italiana parametrizada
|
* commit 2ffbed1645ee4dee7dfe6239c7377a5192371779 (pt-br)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:19 2020 -0300
|
|     versão brasileira parametrizada
|
* commit 37673294806a0a4a8e0ce3110e38abfb0a7a6a96
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:18 2020 -0300
|
|     versão brasileira
|
* commit e86b0d6fc58de36e8d4a0baa9777904f7d2d274d (tag: sofisticada)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:17 2020 -0300
|
|     um pouco mais sofisticado
|
* commit 76c7a2237d5875db7cc395672e095b0c13650049 (tag: funcao)
| Author: Fulano de Tal <fulano@provedor.com>
| Date:   Wed Jul 29 21:07:15 2020 -0300
|
|     criação da função main
|
* commit 5c337fc459bd0c0be453b551255312f0067ee961
  Author: Fulano de Tal <fulano@provedor.com>
  Date:   Wed Jul 29 21:07:14 2020 -0300

      primeira fotografia do sistema
~/greetings$
```

Figura 58. Listando todas as fotografias do repositório em forma de grafo após mesclar a versão alemã

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, possívelmente 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.