



Artigo

# Como usar os comandos do Git



80



Veja neste artigo os principais comandos do Git que são utilizados no dia a dia dos desenvolvedores que trabalham com o sistema de controle de versão distribuído mais utilizado do mundo presente nas principais organizações.

[Voltar](#)[Suporte ao aluno](#)[Anotar](#)[Marcar como concluído](#)

Artigos > Banco de Dados > Como usar os comandos do Git

Grandes empresas de desenvolvimento de software trabalham de forma paralela, onde os funcionários se dividem em pequenas equipes e cada uma delas foca em uma determina parte do projeto. Porém, como software desenvolvido é todo integrado, é necessário que as equipes conversem e os seus pedaços de código também.

A partir dessa necessidade o versionamento de um sistema se faz necessário para promover a integração como também cuidar das melhorias constantes, ou muitos outros motivos. É indiscutível manter as cópias do sistema por questões de segurança também. Há um grande do que chamamos de Sistemas de controle de versões, tanto locais como remotos. Dentre as opções encontramos o Git.

O Git é um dos programas mais utilizados para controlar as versões de sistemas em desenvolvimento. Ao longo desse artigo veremos os principais comandos que, com certeza, todo desenvolvedor deve saber para trabalhar em equipe com o versionamento do código.

Para começar vamos clonar um repositório que já está pronto, bastando usar o comando da **Listagem 1**. Assim, pegamos tudo que estava em um repositório remoto e colocamos no nosso repositório local. A partir deste ponto já é possível fazer alterações no projeto.

```
1 | prompt> git clone git://github.com/projetoCMS/meusite.git
2 | Initialized empty Git repository in /work/meusite/.git/
3 | remote: Counting objects: 12, done.
4 | remote: Compressing objects: 100% (8/8), done.
5 | remote: Total 12 (delta 2), reused 0 (delta 0)
6 | Receiving objects: 100% (12/12), done.
7 | Resolving deltas: 100% (2/2), done.
```

**Listagem 1.** Clonagem de repositório

## Comando Add

Os desenvolvedores estão constantemente adicionando novos arquivos e realizando alterações no conteúdo dos seus repositórios. Utilizando o comando `git add` passando o nome do arquivo ou dos arquivos como parâmetro faz com que ele armazene-os no seu `Stage`, estando prontos para serem **comitados**.

As alterações no `Stage` são simplesmente alterações no **working tree** em que se pretende “avisar” ao repositório sobre essas alterações que foram realizadas. O `Stage` também é chamado de `Index` ou `Staging Area` (ou área temporária) no `Git`.

O `Staging Area` é apenas um lugar em que se quer configurar `commits` que serão realizados no repositório local.

De forma geral, tem-se o `working tree` ou `working directory`, que é a visão atual em que estão os arquivos do projeto, o `staging area` ou `index` prepara os arquivos a serem `comitados`, e por fim, após realizar um `commit` tem-se os arquivos no repositório na qual serão versionados, gerenciados e supervisionados pelo `Git`.

Pode parecer que o desenvolvedor tenha que realizar trabalhos duplicados, pois é necessário fazer `git add` para colocar o arquivo no `staging area` e depois um `git commit` para colocá-los no repositório. No entanto, o `staging area` é um local em que se pode preparar os futuros `commits` a serem realizados, sendo algo importante e que deve ser utilizado.

O comando `git add` também possui outras opções para alterar alguns comportamentos. Entre as opções disponíveis pode-se utilizar o comando `-a` para selecionar quais arquivos ou partes de arquivos serão armazenados no `stage`. Para isso utiliza-se a opção `-i`, que abre um `shell` interativo permitindo fazer essas diferentes seleções.

Para exemplificar vamos considerar o arquivo de exemplo da **Listagem 2** chamado `index.html`, que está no `working tree`.

```
1 | <html>
2 |   <head>
3 |     <title>Testando o Git</title>
4 |   </head>
5 |   <body>
6 |     <h1>Página Inicial GIT</h1>
7 |     <ul>
8 |       <li><a href="teste2.html">Teste com Link</a></li>
9 |     </ul>
10|   </body>
11| </html>
```

**Listagem 2.** Arquivo HTML com conteúdo inicial de uma página

Realize um comando `git add` e `git commit` para adicionar ao `stage` e `comitar` o arquivo, respectivamente.

Agora podemos realizar uma alteração no arquivo anterior adicionando um `link` para que o usuário obtenha maiores informações sobre a página. Esse link pode ser adicionado após a `tag` `<h1>`, conforme o exemplo a seguir:

```
1 | <li><a href="informacoes.html">Informações</a></li>
```

Executando o comando `git add -i` tem-se o **prompt** a seguir sendo exibido:

```
1 | prompt> git add -i
2 |       staged untracked path
3 |       1: unchanged +1/-1 index.html
4 |       *** Commands ***
5 |       1: status 2: update 3: revert 4: add untracked
6 |       5: patch 6: diff 7: quit 8: help
7 | What now>
```

Pode-se verificar que o `What now>` está aguardando uma opção a ser digitada (conforme vemos no código a seguir):

- Digitando `1` será gerado o `status` atual no `staging`;
- Digitando `2` será exibida a mensagem a seguir:

```
1 | What now> 2
2 |     staged unstaged path
3 |     1: unchanged +1/-1 index.html
4 | Update>>
```

Esta opção gera uma lista de arquivos que podem ser armazenados no `stage`. Nesse caso, existe um arquivo para ser armazenado, assim basta digitar `1` para que ele mude para um asterisco (\*) ao lado de seu nome, significando que ele vai ser armazenado no `stage`, conforme pode ser visto no exemplo a seguir:

```
1 | Update>> 1
2 |     staged unstaged path
3 |     * 1: unchanged +1/-1 index.html
4 | Update>>
```

Agora basta pressionar `Enter` para voltar ao menu principal:

```
1 | What now> 1
2 |     staged unstaged path
3 |     1: +1/-1 nothing index.html
4 |     *** Commands ***
5 |     1: status 2: update 3: revert 4: add untracked
6 |     5: patch 6: diff 7: quit 8: help
7 | What now>
```

Se o `status` for verificado novamente, pode-se verificar que existe uma alteração no `stage` e nada listado no `unstaged changes`.

Outro comando que pode ser utilizado é o `revert` que faz reversão da funcionalidade, ou seja, retira do `stage` a alteração realizada. Segue na **Listagem 3** um exemplo em que a última alteração será revertida.

```
1 | What now> 3
2 |     staged unstaged path
3 |     1: +1/-1 nothing index.html
4 |     Revert>> 1
5 |     staged unstaged path
6 |     * 1: +1/-1 nothing index.html
7 |     Revert>>
8 |     reverted one path
9 |     *** Commands ***
10 |    1: status 2: update 3: revert 4: add untracked
11 |    5: patch 6: diff 7: quit 8: help
12 | What now>
```

**Listagem 3.** Comando revert

Pode-se verificar que a alteração no arquivo `index.html` estava armazenada no `stage`, porém

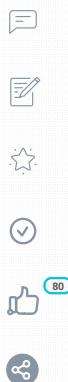
ainda não havia sido executado um `commit` que tiraria a alteração do `stage` e colocaria no repositório local.

Executando novamente o `status` pode-se verificar como está a situação atual:

```
1 | What now> 1
2 |     staged unstaged path
3 |     1: unchanged +1/-1 index.html
4 |     *** Commands ***
5 |     1: status 2: update 3: revert 4: add untracked
6 |     5: patch 6: diff 7: quit 8: help
```

A quarta opção permite colocar no `stage` todos os arquivos que ainda não estão lá. Essa opção é interessante quando foram realizadas diversas modificações e, ao invés de um `add` para cada arquivo ou uma lista de arquivos, adiciona-se todos os arquivos em um único comando.

O `patch` permite que sejam adicionados arquivos através do modo com o mesmo nome, assim, todas as diferenças são exibidas. Ele opera através de um `shell` interativo disponibilizando várias opções, como podemos ver na **Listagem 4**.



```
1 | What now> 5
2 |     staged unstaged path
3 |     1: unchanged +1/-1 index.html
4 |     Patch update>> 1
5 |     staged unstaged path
6 |     * 1: unchanged +1/-1 index.html
7 |     Patch update>>
8 |     diff --git a/index.html b/index.html
9 |     index e812d0a..ca86894 100644
10|     --- a/index.html
11|     +++ b/index.html
12|     @@ -6,7 +6,7 @@
13|     <html>
14|     <head>
15|     <title>Testando o Git</title>
16|     </head>
17|     <body>
18|     <h1>Página Inicial GIT</h1>
19|     <ul>
20|     - <li><a href="teste2.html">Teste com Link</a></li>
21|     + <li><a href="informacoes.html">Informações</a></li>
22|     </ul>
23|     </body>
24|     </html>
25|
26| Stage this hunk [y/n/a/d/e/?]? 
```

**Listagem 4.** Modo Patch

Agora existe uma opção que permite armazenar esse pedaço (ou `hunk`) de código ou negá-lo. Um `hunk` é uma alteração dentro de um arquivo. Cada área diferente em um arquivo é considerada um `hunk`.

Existem algumas opções a serem digitadas:

- Digite `y` e as alterações são aceitas;
- Digite `n` para pula as alterações;
- Digite `a` ou `d` para adicionar ou nega todo o resto das alterações no arquivo;
- Digite `?` para exibir uma ajuda explicando o que cada uma das opções faz.

Se digitar a opção `7"` o `shell` interativo é finalizado, como vemos no código a seguir:

```
1 Stage this hunk [y/n/a/d/e/?]? n
2 *** Commands ***
3 1: status 2: update 3: revert 4: add untracked
4 5: patch 6: diff 7: quit 8: help
5 What now> 7
6 Bye.
```

Agora que os arquivos já foram adicionados ao `stage` podemos `comitá-los`. Como fazer `commits` e as opções disponíveis para o `commit` serão verificadas na próxima seção.

## Comando Commit

`Commit` é um processo simples que adiciona as alterações para o histórico do repositório e atribui um nome ao `commit`.

Um ponto importante é que o `Git` não versiona diretórios. Todo diretório deve possuir pelo menos um arquivo que será então versionado.

O `git` permite que os desenvolvedores criem arquivos que não sejam versionados, ou seja, eles serão ignorados pelo repositório. Para isto basta criar um arquivo com um ponto no inicio do nome como, por exemplo, ".nomearquivo". Todos arquivos que começam com um ponto são ignorados pelo sistema de arquivos. Neste caso, a alteração não é enviada para o repositório central.

O comando `git commit` pode ser utilizado de várias formas para `comitar` as alterações para o repositório, porém todo `commit` necessita de uma mensagem para logar, já que esta o identificará. Para isso basta utilizar o comando `git commit -m "mensagem de exemplo"`. A mensagem pode ser qualquer `string` válida. Também é possível definir vários parágrafos utilizando várias opções `-m`.

Além dessa, existem outras três formas de realizar um `commit`.

De forma geral, primeiramente o desenvolvedor executa o comando `git add` para adicionar as alterações ao `stage` para então poder comitá-las (bastando executar o comando `commit`). Faremos isso com os comandos a seguir:

```
1 | prompt> git add qualquer-arquivo
2 | prompt> git commit -m "alterações realizadas em qualquer-arquivo"
```

Outra forma é executar o `commit` com o parâmetro `-a`, assim, dizemos ao `Git` para que ele pegue a última versão do `working tree` e realize um `commit` ao repositório. Arquivos novos que não estão versionados não serão adicionados, apenas os que já foram versionados.

Assim, para utilizar o parâmetro `-a` basta executar o comando a seguir:

```
1 | prompt> git commit -m " alterações realizadas em qualquer-arquivo" -a
```

Também é possível `comitar` um arquivo específico através do comando a seguir:

```
1 | prompt> git commit -m " alterações realizadas em qualquer-arquivo" qualquer-arquivo
```

Este comando é útil quando se quer realizar `commit` apenas em alguns arquivos que foram

alterados e não em todos.

Após realizar um `commit` o `stage` é esvaziado.

## Verificando Alterações

Para saber o que foi alterado no `working tree` pode-se utilizar os comandos `git status` e `git diff`.

O comando `git status` verifica todas as alterações que ocorreram no repositório. A saída é gerada baseando-se no `status` dos `commits` e do `working tree`.

Segue na **Listagem 5** um exemplo de como utilizá-lo.

```
1 | prompt> git status
2 | # On branch master
3 | # Changes to be committed:
4 | # (use "git reset HEAD <file>..." to unstage)
5 | #
6 | # modified: index.html
7 | #
```

**Listagem 5.** Comando `git status`

Pode-se notar que existem alterações aguardando para serem `comitadas` conforme informa a linha `"Changes to be committed"`. Porém, vamos adicionar outra alteração no arquivo `index.html` conforme a seguir:

```
1 | <li><a href="contato.html">Contato</a></li>
```

Após salvar as alterações executa-se outro `git status` para verificar agora como ficou o `status` atual:

```
1 | prompt> git status
2 | # On branch master
3 | # Changes to be committed:
4 | # (use "git reset HEAD <file>..." to unstage)
5 | #
6 | # modified: index.html
7 | #
8 | # Changed but not updated:
9 | # (use "git add <file>..." to update what will be committed)
10 |
11 | # modified: index.html
12 | #
```

Pode-se notar que agora o arquivo `index.html` está listado duas vezes. A primeira modificação diz respeito à alteração no `stage` que ainda foi `comitado`. A segunda alteração refere-se à alteração que foi realizada agora no `working tree` e que ainda não foi adicionada no `stage`, por isso a mensagem `Changed but not updated`.

Agora é um bom momento de verificar as modificações realizadas. Para isso pode-se utilizar o comando `diff`. Este comando permite que possamos verificar o que foi alterado no `working tree`, no `stage` e no `repositório`.

O comando da **Listagem 6** mostra as alterações no `working tree` que não foram armazenadas no `stage` ou que não foram `comitadas` ainda.

```

1 | prompt> git diff
2 | diff --git a/index.html b/index.html
3 | index ca86894..5fdc539 100644
4 | --- a/index.html
5 | +++ b/index.html
6 | @@ -7,6 +7,7 @@
7 | <html>
8 | <head>
9 | <title>Testando o Git</title>
10 | </head>
11 | <body>
12 | <h1>Página Inicial GIT</h1>
13 | <ul>
14 | <li><a href="informacoes.html">Informações</a></li>
15 | + <li><a href="contato.html">Contato</a></li>
16 | </ul>
17 | </body>
18 | </html>

```

**Listagem 6.** Comando diff

Veja que tem-se o link que foi adicionado, o sinal `+` no inicio da linha mostra que neste ponto ocorreu uma adição e está faltando no `stage`. Já o link acima "informacoes" mostra que não há qualquer alteração neste ponto.

Executando o comando `diff` sem qualquer parâmetro compara-se as alterações no `working tree` com o `staging area`.

Para visualizar as diferenças entre o `staging area` e o `repositório` basta utilizar o parâmetro `--cached` na chamada conforme a **Listagem 7**.

```

1 | prompt> git diff --cached
2 | diff --git a/index.html b/index.html
3 | index e812d0a..ca86894 100644
4 | --- a/index.html
5 | +++ b/index.html
6 | @@ -6,7 +6,7 @@
7 | <html>
8 | <head>
9 | <title>Testando o Git</title>
10 | </head>
11 | <body>
12 | <h1>Página Inicial GIT</h1>
13 | <ul>
14 | - <li><a href="teste2.html">Teste com Link</a></li>
15 | + <li><a href="informacoes.html">Informações</a></li>
16 | </ul>
17 | </body>
18 | </html>

```

**Listagem 7.** Parâmetro `--cached`

Agora pode-se notar no código que o sinal de `"` indicando que algo foi removido. Mas lembre-se que o que não está no `stage` ainda não é mostrado.

Para verificar o que está no `working tree` e no `stage` com o que está no repositório pode-se utilizar o comando da **Listagem 8**.

```

1 | prompt> git diff HEAD
2 | diff --git a/index.html b/index.html
3 | index e812d0a..5fdc539 100644
4 | --- a/index.html
5 | +++ b/index.html
6 | @@ -6,7 +6,8 @@
7 | <html>

```

```
8 | <head>
9 | <title>Testando o Git</title>
10 | </head>
11 | <body>
12 | <h1>Página Inicial GIT</h1>
13 | <ul>
14 | - <li><a href="teste2.html">Teste com Link</a></li>
15 | + <li><a href="informacoes.html">Informações</a></li>
16 | + <li><a href=" contato.html ">Contato</a></li>
17 | </ul>
18 | </body>
19 | </html>
```

**Listagem 8.** Verificando o que está no stage

O `HEAD` é uma palavra chave que se refere ao `commit` mais recente no `branch` que estamos atualmente.

Para `comitar` tudo basta utilizar o comando a seguir:

```
1 | prompt> git commit -a -m "Realizando todos os commits" -m "
2 | Adicionado o link sobre informações" -m " Adicionado o link sobre contato"
3 | Created commit 6f1bf6f: Realizando todos os commits
4 | 1 files changed, 2 insertions(+), 1 deletions(-)
```

Algumas vezes é necessário limpar alguma coisa, por isso, o `git` fornece alguns comandos para mover arquivos ou conteúdos, conforme mostra a **Listagem 9**.

```
1 | prompt> git mv index.html testeola.html
2 | prompt> git status
3 | # On branch master
4 | # Your branch is ahead of 'origin/master' by 1 commit.
5 |
6 | # Changes to be committed:
7 | # (use "git reset HEAD <file>..." to unstage)
8 |
9 | # renamed: index.html -> testeola.html
10 | #
```

**Listagem 9.** Limpando arquivos e conteúdos

O `git rm` remove o arquivo, mas mantém o histórico. Para eliminar tudo isso é necessário chamar o `git add` e na sequência `git rm` para remover o arquivo `index.html` que está no repositório. Assim, o `commit` realizado:

```
1 | prompt> git commit -m "testando o arquivo com um novo nome"
2 | Created commit 9a23464: testando o arquivo com um novo nome
3 | 1 files changed, 0 insertions(+), 0 deletions(-)
4 | rename index.html => testeola.html (100%)
```

## Branches

O `Git` pode ser usado com um único `branch`, o `branch master`, similar ao `trunk` do `SVN`, que é outro tipo de repositório. Com isso, o desenvolvedor tem o versionamento do código e histórico e pode colaborar com outros desenvolvedores, não se preocupando com deleções acidentais. O grande problema disso é que o desenvolvedor não estaria usando todo o potencial da ferramenta.

Para começar a exploração dos `branches` pode-se fazer o `clone` de um projeto já existente, conforme o código a seguir:

```
1 | prompt> git clone git://github.com/projetoCMS/meusite.git
2 | Initialized empty Git repository in /work/meusite/.git/
```

Tudo no `Git` é tratado como um `branch`. Mesmo que todo o trabalho feito pelo desenvolvedor esteja no `branch master` é possível alterar o nome para um mais apropriado. Para renomear este `branch` principal para outro nome basta fazer conforme a seguir:

```
1 | prompt> git branch -m master meubranchmaster
2 | prompt> git branch
3 | * meubranchmaster
```

Dessa forma, o `branch master` foi renomeado para `meubranchmaster`. O parâmetro `-m` no primeiro comando solicita ao `Git` a execução da operação de renomeação. Os outros dois parâmetros são: o nome do `branch antigo` seguido pelo nome do `novo branch`.

O segundo comando `git branch` sem nenhum parâmetro tem como saída o nome de todos os `branches locais` do repositório.

Como tudo no `Git` é considerado um `branch`, ele é dito como sendo "barato", diferente de outros sistemas em que os arquivos são copiados em um novo diretório. No `Git`, um `branch` faz o versionamento dos commits que são realizados, mantendo assim o rastreamento do último. Este possui um link para seus pais, por onde o `Git` pode buscar todas as alterações nesse `branch`.

A maior dúvida entre os desenvolvedores e projetistas na maioria das organizações é quando criar um novo `branch` e qual o momento para fazer isso?

Existem basicamente três situações quando se deve criar um novo `branch`:

- 1. Alterações Experimentais:** Quando for preciso tentar reescrever um algoritmo para torná-lo mais rápido ou tentar refatorar uma parte do código para algum padrão pode-se criar um novo `branch`. Assim, é possível trabalhar nessa parte separadamente sem afetar o restante que já está funcional e pronta para `deploy`;
- 2. Novas funcionalidades:** Sempre que é necessário trabalhar em uma nova funcionalidade pode-se criar um novo `branch`. Quando finalizar a implementação da funcionalidade basta fazer um `merge` com o `branch` de `release` da versão;
- 3. Correção de Bugs:** Quando surgirem `bugs` a serem corrigidos no código que ainda não foi lançado ou em uma versão que já foi `taggeada` e lançada cria-se um `branch` para realizar a correção do `bug`. Isso garante uma maior flexibilidade aos desenvolvedores para trabalharem no erro. Após a correção pode-se realizar um `merge`.

A criação de um `branch` é simples utilizando o comando `git branch` e adicionando o nome do `branch`:

```
1 | prompt> git branch novobranch
```

Para verificar se o `branch` foi criado basta executar o seguinte comando:

```
1 | prompt> git branch
2 | * master
3 | novobranch
```

Pode-se verificar que existem dois `branches`: o `master` e `novobranch`. O asterisco antes do nome do `branch` indica que este é o `branch` atual que estamos trabalhando no `working tree`.

Agora que o `branch` foi criado é necessário baixar os arquivos do `branch` e realizar as alterações necessárias. Para isso deve-se realizar um `check out` no `branch`:

```
1 | prompt> git checkout novobranch
2 | Switched to branch "novobranch"
```

Executando o comando `git branch` novamente pode-se verificar que agora o `branch` atual foi alterado:

```
1 | prompt> git branch
2 | master
3 | * novobranch
```

O `Git` também permite que um `branch` seja criado e que imediatamente seja realizado um `check out` dele, tudo em apenas um único passo. Para isso será criado um novo `branch` usando `git checkout -b`, lembrando que este precisa ter um nome único no repositório:

```
1 | prompt> git checkout -b alternate master
2 | Switched to a new branch "alternate"
```

Repare que o terceiro parâmetro `master` solicita para o `Git` criar o `branch` a partir do `master`, ao invés do `branch` atual.

Agora é necessário também fazer os `merges` necessários. Para isso existem diferentes formas:

1. **Straight merges**: Este tipo pega o histórico de um `branch` e mescla com o histórico de outro. Este tipo de `merge` é utilizado quando se realiza um `pull`, e para exemplificar este caso basta criar um novo arquivo e realizar um `add` e `commit` no repositório, como mostra a **Listagem 10**.

```
1 | prompt> git add informacoes.html
2 | prompt> git commit -m "Adicionado o esqueleto de uma página de informações"
3 | Created commit 217a88e: Adicionado o esqueleto de uma página de informações
4 | 1 files changed, 15 insertions(+), 0 deletions(-)
5 | create mode 100644 informacoes.html
```

**Listagem 10.** Tipo de merge Straight

Ao realizar esse `commit` em um `branch` pode-se verificar que ele ainda não existe no `branch` `master`. Para isso, basta realizar um `merge` entre os dois. Primeiramente, é preciso trocar para o `branch` que receberá as alterações realizadas em um outro. Nesse caso, é preciso ir para o `branch master`:

```
1 | prompt> git checkout master  
2 | Switched to branch "master"
```

Agora basta usar o `git merge` com o nome do `branch` que será mesclado com o `branch` atual (`master`), como mostra a **Listagem 11**.

```
1 | prompt> git merge alternate  
2 | Updating 9a23464..217a88e  
3 | Fast forward  
4 | informacoes.html | 15 ++++++  
5 | 1 files changed, 15 insertions(+), 0 deletions(-)  
6 | create mode 100644 informacoes.html
```

**Listagem 11.** Git Merge

Dessa forma, as alterações do `branch alternate` foram mescladas com o `branch master`.

2. **Squashed commits:** Este tipo de `merge` pega o histórico de um `branch` e comprime (ou `squash`) em um `commit` no topo de outro `branch`.  
`Branches` criados para adicionar apenas funcionalidades utilizam mais esse tipo de `merge`. Assim, criam-se as funcionalidades, ou talvez se corrija algum bug dentro de um `branch` para então realizar um `commit` `squashed` para mesclar as alterações de volta no `branch`. Esses `commits` são `squashed`, pois o `Git` pega todo o histórico de um `branch` e comprime em um `commit` em outro `branch`. Deve-se ter cuidado com os `commits` `squashed`, pois na maioria das vezes queremos todos os `commits` separados no histórico, porém, se as alterações de um `branch` representam uma alteração individual, então este `branch` é um grande candidato para o `commit squashed`. Assim, ao invés de realizar o `commit` de cada versão da funcionalidade apenas realiza-se o `commit` da versão final.  
Como exemplo, criaremos um `branch` a partir do `master` chamado `contato` e realizaremos o `check out` deste `branch`: Agora se adiciona o arquivo `contato.html` que possui apenas um `e-mail` na página HTML e comitamos no `branch`, como mostra a **Listagem 12**.

```
1 | prompt> git add contato.html  
2 | prompt> git commit -m "Adicionado arquivo HTML contato com o e-mail"  
3 | Created commit 5b4fc7b: Adicionado arquivo HTML contato com o e-mail  
4 | 1 files changed, 15 insertions(+), 0 deletions(-)  
5 | create mode 100644 contato.html
```

**Listagem 12.** Comitando no branch

Alteramos o arquivo colocando um segundo `e-mail` e comitamos novamente:

```
1 | prompt> git commit -m "Adicionado o Segundo e-mail" -a  
2 | Created commit 2f30ccd: Adicionado o Segundo e-mail  
3 | 1 files changed, 4 insertions(+), 0 deletions(-)
```

Assim, existem dois `commits` no `branch contato`. Podemos comprimir estes em apenas um no `branch master`, mas para isso é necessário primeiramente ir para o `branch master` e realizar o `merge` passando `--squash` como parâmetro, como mostra a **Listagem 13**.

```
1 | prompt> git checkout master  
2 | Switched to branch "master"  
3 | prompt> git merge --squash contato
```

```
4 | Updating 217a88e..2f30ccd
5 | Fast forward
6 | Squash commit -- not updating HEAD
7 | contato.html | 19 ++++++
8 | 1 files changed, 19 insertions(+), 0 deletions(-)
9 | create mode 100644 contato.html
```

**Listagem 13.** Comprimindo a **Listagem 12** em um branch

Agora os `commits` do `branch contato` foram aplicados no `working tree` e no `stage`, porém ainda não foram `comitados`. Para isso basta executar o comando `git status` para verificar, presente na **Listagem 14**.

```
1 | prompt> git status
2 | # On branch master
3 | # Your branch is ahead of 'origin/master' by 1 commit.
4 | #
5 | # Changes to be committed:
6 | # (use "git reset HEAD <file>..." to unstage)
7 | #
8 | # new file: contato.html
9 | #
```

**Listagem 14.** Comando Git Status

Agora então precisamos executar o comando a seguir para realizar o `commit`, como mostra o código a seguir:

```
1 | prompt> git commit -m "Adicionada a página contato" -m "possuindo o e-mail"
2 | Created commit 18f822e: Adicionada a página contato
3 | 1 files changed, 19 insertions(+), 0 deletions(-)
4 | create mode 100644 contato.html
```

Com esses passos o merge já foi realizado.

3. **Cherry-picking:** Este `merge` é necessário quando é preciso realizar o `merge` de apenas um `commit` entre `branches`, e assim, não queremos realizar um `commit` de tudo. Para realizar esses `commits` individuais utiliza-se o `cherry-pick`. Exemplificando este caso primeiramente realizamos um `check out` no `branch contato` novamente com os comandos a seguir:

```
1 | prompt> git checkout contato
2 | Switched to branch "contato"
```

Agora podemos adicionar outro contato no arquivo `contato.html` como, por exemplo, o `link` do `Twitter`, e `comitamos` o arquivo no `branch` com os comandos a seguir:

```
1 | prompt> git commit -m "Adicionado link do twitter" -a
2 | Created commit 321d76f: Adicionado link do twitter
3 | 1 files changed, 4 insertions(+), 0 deletions(-)
```

Devemos guardar o nome do `commit: 321d76f` para fazer o `cherry-pick`, já que esses nomes são únicos, portanto não tem problema de dar algum conflito.

Agora podemos fazer um `check out` no `branch` que receberá as alterações, nesse caso o `branch master`, e por fim realizar o `cherry-pick` com os comandos da **Listagem 15**.

```
1 | prompt> git checkout master
2 | Switched to branch "master"
3 | prompt> git cherry-pick 321d76f
4 | Finished one cherry-pick.
5 | Created commit 294655e: Adicionado link do twitter
6 | 1 files changed, 4 insertions(+), 0 deletions(-)
```

**Listagem 15.** Comandos para o cherry-pick

Para realizar o `cherry-pick` de múltiplos `commits` basta passar apenas o parâmetro `-n` junto.

Outro problema bastante comum que ocorrem nos `merges` são os conflitos, que ocorrem quando um mesmo arquivo é editado de diferentes formas em diferentes `branches`, assim, ao tentar realizar um merge o `Git` avisa que existe um conflito.

Portanto, o `Git` acusa um conflito quando ele não consegue realizar automaticamente o `merge`. Um exemplo de um conflito é usarmos um nome de variável diferente em cada `branch` nas mesmas partes de um arquivo. Para um exemplo de como funciona um conflito segue o comando a seguir em que se cria um `branch`:

```
1 | prompt> git checkout -b informacoes master
2 | Switched to branch "informacoes"
```

Neste momento pode-se criar um arquivo `informacoes.html` com os nomes de algumas linguagens de programação. Após isso basta executar os comandos a seguir:

```
1 | prompt> git add informacoes.html
2 | prompt> git commit -m "Lista com linguagens de programação."
3 | Created commit 01fe684: Lista com linguagens de programação.
4 | 1 files changed, 7 insertions(+), 0 deletions(-)
```

Crie um segundo `branch` chamado `informacoes2`, mas sem alterar para este `branch`:

```
1 | prompt> git branch informacoes2 informacoes
```

E antes de mudar para o `branch informacoes2` é necessário preencher o arquivo `informacoes.html` com outra linguagem de programação. Após salvar a página basta realizar um `commit`:

```
1 | prompt> git commit -m "Adicionado Javascript para a lista" -a
2 | Created commit 9e114ac: Adicionado Javascript para a lista
3 | 1 files changed, 1 insertions(+), 0 deletions(-)
```

Portanto, agora existem dois `informacoes.html` diferentes no `branch informacoes` e no `branch informacoes2`.

Agora executa-se o comando a seguir para ir ao `branch informacoes2`:

```
1 | prompt> git checkout informacoes2  
2 | Switched to branch "informacoes2"
```

O arquivo `informacoes.html` não possui a última linguagem de programação adicionada anteriormente, pois aquela alteração está no `branch informacoes`. Então, para após adicionar, salve e `comite` as alterações conforme a seguir:

```
1 | prompt> git commit -m "Adicionado EMCAScript para a lista" -a  
2 | Created commit b84ffdc: Adicionado EMCAScript para a lista  
3 | 1 files changed, 1 insertions(+), 0 deletions(-)
```

Mas ai existe um conflito! Agora voltamos ao `branch informacoes` e tentamos fazer um `merge` com `informacoes2` com os comandos da **Listagem 16**.

```
1 | prompt> git checkout informacoes  
2 | Switched to branch "informacoes"  
3 | prompt> git merge informacoes2  
4 | Auto-merged informacoes.html  
5 | CONFLICT (content): Merge conflict in informacoes.html  
6 | Automatic merge failed; fix conflicts and then commit the result.
```

**Listagem 16.** Merge com informacoes2

Pode-se verificar que a linha `CONFLICT` indica que foi detectado um conflito no arquivo `informacoes.html`, conforme a **Listagem 17**.

```
1 | <ul>  
2 | <li>Erlang</li>  
3 | <li>Python</li>  
4 | <li>Objective C</li>  
5 | <<<<< HEAD:informacoes.html  
6 | <li>Javascript</li>  
7 | =====  
8 | <li>EMCAScript</li>  
9 | >>>>> informacoes2:informacoes.html  
10 | </ul>
```

**Listagem 17.** Conflito identificado

Pode-se verificar que o `Git` encontrou dois nomes diferentes no último elemento `<li>`. A primeira linha é o `commit` realizado anteriormente e o segundo é a linha que foi adicionada no `informacoes2`.

O conflito começa com a linha "`<<<<< HEAD:informacoes.html`". O código conflitado com o outro `branch` é exibido após o separador `"=====`" e finaliza com a linha `">>>>> informacoes2:informacoes.html`".

O que estas linhas dizem é que qualquer código precedido de `<<<<<` é o código do `branch` atual, e qualquer código após `>>>>>` é do outro `branch`. Além disso, o nome que está tentando realizar o `merge` vem antes do nome do arquivo. Neste caso, `HEAD` que é o nome do `commit` mais recente no `branch` atual que conflitou com `informacoes2` que é o outro `branch`.

Para resolver o conflito deve-se realizar o procedimento manualmente, abrindo-o em um editor e realizando a alteração. Neste caso, pode-se deixar a linha que deu conflito igual à do outro `branch` e adiciona-se abaixo desta mais um `<li>` com o nome da outra linguagem de

programação.

Após realizar os `merges` e criar a `tag` que lança a versão efetivamente pode-se apagar todos os `branches`, visto que eles já serviram aos seus propósitos. Assim, basta realizar o comando a seguir:

```
1 | prompt> git branch -d informacoes2
2 | Deleted branch informacoes2.
```

Para isto funcionar, o `branch` que será deletado já deve ter sido mesclado com o `branch` atual que estamos usando atualmente. Por exemplo, se voltar ao `branch master`, em que não foi realizado nenhum `merge`, e se for realizada uma tentativa de exclui-lo não será possível como vemos a seguir:

```
1 | prompt> git checkout master
2 | Switched to branch "master"
3 | prompt> git branch -d informacoes
4 | error: The branch 'informacoes' is not an ancestor of your current HEAD.
5 | If you are sure you want to delete it, run 'git branch -D informacoes'.
```

No entanto, se for necessário remover o `branch` sem realizar `merge`, basta forçar a deleção com o parâmetro `-D` ao invés de `-d`.

Outra situação que às vezes se faz necessária é renomear um `branch`, e para isto basta executar o comando da **Listagem 18**.

```
1 | prompt> git branch -m contato contatos
2 | prompt> git branch
3 |   informacoes
4 |   alternate
5 |   contatos
6 | * master
7 |   novo
```

**Listagem 18.** Renomeando o branch

Outro comando bastante útil para executar em qualquer `branch` é verificar o histórico dele. Qualquer arquivo adicionado ou alterações realizadas cria um novo `commit` que mantém um histórico no repositório.

O comando `git log` mostra todo histórico do repositório e o `log` é exibido em ordem cronológica reversa, como um `blog`. Para exibi-lo basta realizar o comando da **Listagem 19**.

```
1 | prompt> git log
2 | commit 0bb3dfb752fa3c890ffc781fd6bd5dc5d34cd3be
3 | Author: Higor Medeiros <contato@devmedia.com>
4 | Date: Sat Nov 4 11:06:47 2015 -0500
5 | Adicionado link para o twitter
6 | commit 18f822eb1044761f59aebaf7739261042ae10392
7 | Author: Higor Medeiros <contato@devmedia.com>
8 | Date: Sat Nov 4 10:34:51 2015 -0500
9 | Adicionada página de contato
```

**Listagem 19.** Comando log

O `Git` mostra algumas informações do `commit` como o `nome`, o `autor`, a `data` e uma `mensagem` utilizada ao realizá-lo.

Uma opção que pode ajudar é o `-p`, que mostra a diferença que a revisão criou. Também é possível visualizar apenas um número limitado de `commits`. Por exemplo, utilizando o `-1` o `log` é limitado a apenas um `commit`, `-2` limita a dois `commits`, e assim por diante.

Se for necessário apenas verificar o `log` de uma dada revisão basta passar a revisão como na

**Listagem 20.**

```
1 | prompt> git log 7b1558c
2 | commit 7b1558c92a7f755d8343352d5051384d98f104e4
3 | Author: Higor Medeiros <contato@devmedia.com>
4 | Date: Sun Nov 21 14:20:21 2015 -0500
5 | Adicionada mensagem de teste ao arquivo HTML
```

**Listagem 20.** Log por revisão

Pode-se verificar que não é necessário passar o nome todo e sim os primeiros caracteres, como os quatro ou cinco primeiros caracteres, porém os sete primeiros são mais garantidos.

Existem algumas opções mais úteis como filtrar as informações. Por exemplo, para recuperar os `commits` das últimas cinco horas pode-se executar o comando da **Listagem 21**.

```
1 | prompt> git log --after='5 hours'
2 | commit 0bb3dfb752fa3c890fffc781fd6bd5dc5d34cd3be
3 | Author: Higor Medeiros <contato@devmedia.com>
4 | Date: Sat Nov 4 11:06:47 2015 -0500
5 | Adicionado link para o twitter
```

**Listagem 21.** Recuperando commits

Também é possível pular as últimas cinco horas e visualizar apenas os mais antigos com o código da **Listagem 22**.

```
1 | prompt> git log --before="5 hours" -1
2 | commit 18f822eb1044761f59aebaf7739261042ae10392
3 | Author: Higor Medeiros <contato@devmedia.com>
4 | Date: Sat Nov 4 10:34:51 2015 -0500
5 | Adicionada página de contato.
```

**Listagem 22.** Logs antigos

Outras opções que o `Git` também aceita são: `--since="24 hours"`, `--since="1 minute"` e `--before="2008-10-01"`.

Além disso, também é possível passar uma faixa de `commits` com o código da **Listagem 23**.

```
1 | prompt> git log 18f822e..0bb3dfb
2 | commit 0bb3dfb752fa3c890fffc781fd6bd5dc5d34cd3be
3 | Author: Higor Medeiros <contato@devmedia.com>
4 | Date: Sat Nov 4 11:06:47 2015 -0500
5 | Adicionado link para o twitter
```

**Listagem 23.** Passando faixa de commits

Também é possível utilizar o `HEAD`, que é o sinônimo de última versão do `branch` atual, como

podemos ver na **Listagem 24**.

```
1 | prompt> git log 18f822e..HEAD
2 | commit 0bb3dfb752fa3c890ffc781fd6bd5dc5d34cd3be
3 | Author: Higor Medeiros <contato@devmedia.com>
4 | Date: Sat Nov 4 11:06:47 2015 -0500
5 | Adicionado link para o twitter
```

**Listagem 24.** Utilizando o HEAD

O comando `diff` também é bastante utilizado quando é preciso visualizar o histórico dentro do repositório, como mostra a **Listagem 25**.

```
1 | prompt> git diff 18f822e
2 | diff --git a/contato.html b/contato.html
3 | index 64135cb..63617c2 100644
4 | --- a/contato.html
5 | +++ b/contato.html
6 | @@ -13,6 +13,10 @@
7 | <p>
8 | <a href="mailto:contato@devmedia.com">E-mail</a>
9 | </p>
10 | +
11 | + <p>
12 | + <a href="http://twitter.com/devmedia">Twitter</a>
13 | + </p>
14 | </body>
15 | </html>
```

**Listagem 25.** Comando diff

O `diff` também permite visualizar algumas estatísticas sobre as alterações que tem sido realizadas, como mostra a **Listagem 26**.

```
1 | prompt> git diff --stat 1.0
2 | informacoes.html | 15 ++++++-----
3 | contato.html | 23 ++++++-----
4 | testeola.html | 13 ++++++-----
5 | index.html | 9 -----
6 | 4 files changed, 51 insertions(+), 9 deletions(-)
```

**Listagem 26.** Utilizando diff

Por fim, pode-se também reverter um `commit` devido alguma alteração realizada equivocadamente ou por algum outro fator que invalide o `commit`, conforme mostra o exemplo da **Listagem 27**.

```
1 | prompt> git revert -n HEAD
2 | Finished one revert.
3 | prompt> git revert -n 540ecb7
4 | Removed copy.txt
5 | Finished one revert.
6 | prompt> git commit -m "revert 45eaf98 and 540ecb7"
7 | Created commit 2b3c1de: revert 45eaf98 and 540ecb7
8 | 2 files changed, 0 insertions(+), 10 deletions(-)
9 | delete mode 100644 copy.txt
```

**Listagem 27.** Revertendo Commit





Veja que código reverteu o `commit` de nome `540ecb7` e o `HEAD`. E após a reversão o `commit` foi realizado, pois o Git mantém as alterações no `stage`, assim o desenvolvedor deve realizar um `commit` para efetivar as alterações.

Veja que não é difícil usar o Git para garantir o versionamento do seu código. Vimos pela sequência de códigos que erros cometidos durante o armazenamento podem ser revertidos. E qualquer alteração é registrada com os logs.

Espero que tenham gostado do artigo. Até a próxima.

## Bibliografia

[1] Travis Swicegood. Pragmatic Version Control Using Git. Bookshelf, 2008.

[2] **Git Reference**

<https://git-scm.com/>

[3] **GitHub**

<https://github.com/>

Voltar

Anotar

Marcar como concluído



Por Higor

Em 2015



## Menu

[Quem Somos](#)

[Planos de estudo](#)

[Fale conosco](#)

[Plano para Instituição de ensino](#)

[Assinatura para empresas](#)

[Assine agora](#)

Hospedagem web por Porta 80 Web Hosting