GIT & GITHUB

Do Básico ao Avançado

Domine Git e GitHub com este material completo. Com ele você aprenderá a gerenciar fluxos de trabalho de maneira simples e dinâmica.

Git Back

Em materias anteriores, nós aprendemos a seguir adiante no processo de versionamento com o Git, ou seja, aprendemos a “andar para frente”.

Mas como será o processo de “andar para trás”? Digo, como eu faço para desfazer certas alterações que já foram comitadas?

É importante ressaltar que atualmente, o git não conta com um sistema tradicional de “desfazer” commits, como um simples [Ctrl] + [Z], ou quem sabe um comando que você usa e: “voilá voltei dois passos atrás”.

Tentando desfazer commits com checkout

Talvez você já tenha imaginado esse tipo de estratégia, aonde podemos retornar para um estado do commit anterior, e começar a trabalhar a partir dele.

Veja um exemplo desse tipo de estratégia em funcionamento:

#1 Supondo que temos atualmente 4 commits diferentes em um determinado projeto.

```
MINGW64:/c/Projeto

SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git log --oneline

b4a1015 (HEAD -> master) 4) Atualização de Arquivo
9e1e293 3) Atualização de Arquivo
80c6253 2) Atualização de Arquivo
1d34372 1) Criação de Arquivo
```

#2 Onde cada commit equivale a uma nova linha de texto:

```
1  [Início] Criação de Arquivo!
2  [Atualização 1]: “Github é uma plataforma bem legal”.
3  [Atualização 2]: “Git não é um sistema de versionamento”.
4  [Atualização 3]: “bla bla bla”.
```

Como podemos perceber, a atualização 2 (linha 3) está com uma informação errada, uma vez que o **Git é sim um sistema de versionamento de código!**

No caso do arquivo acima, houveram algumas atualizações posteriores, como é o caso da linha 7 que também representa um commit.

Podemos dizer, então, que esse “problema” acabou passando batido.

Considerando que queremos voltar para o segundo commit (80c6253) e realizar a correção deste erro a partir daí.

Sendo assim, nós podemos fazer o seguinte:

```
git checkout 80c6253
```

Como já sabemos, automaticamente seguiremos ao estágio de **“detached Head”**.

Se fizermos modificações nesse arquivo, e executarmos um novo commit:

```
git commit -am “3) Atualização do Arquivo”
```

Uma nova hash (commit) será gerada.

```
MINGW64:/c/Projeto

SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git log --oneline

577fd05 (HEAD -> master) 3) Atualização de Arquivo
80c6253 2) Atualização de Arquivo
1d34372 1) Criação de Arquivo
```

O problema é que se voltarmos para **“master”**:

```
git checkout master
```

Todo o progresso que você fez será perdido, de tal forma que se você executar o comando **git log –oneline** você não conseguirá visualizar o commit que acabou de fazer (**577fd05**):

```
MINGW64:/c/Projeto

SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git log --oneline

b4a1015 (HEAD -> master) 4) Atualização de Arquivo
9e1e293 3) Atualização de Arquivo
80c6253 2) Atualização de Arquivo
1d34372 1) Criação de Arquivo
```

Onde só será possível acessar aquele commit novamente, caso você tiver salvo aquela hash em algum outro lugar.

Aí, sim, você consegue visualizar as alterações anteriores:

```
git checkout 577fd05
```

Nesse caso, como queremos trabalhar na ramificação principal (master), essa estratégia de **“desfazer”** acaba não sendo uma das mais úteis.

Tentando desfazer commits com Branchs

A ideia aqui é fazer o uso da mesma estratégia vista anteriormente, só que a diferença é que podemos criar uma nova ramificação por meio de branchs.

Por exemplo, vamos voltar novamente ao estado anterior do projeto (**80c6253**):

```
git checkout 80c6253
```

Realizar alterações em nosso arquivo:

```
1  [Início] Criação de Arquivo!
2  [Atualização 1]: “Github é uma plataforma bem legal”.
3  [Atualização 2]: "Git é um considerado um sistema de versio-
   namento de código".
```

E criar uma nova banch:

```
git checkout -b nova_branch_corrigida
```

A partir de agora, o git entrará em um novo “universo alternativo”, onde os commits **9e1e293** e **b4a1015** não existam mais.

Com isso você pode continuar trabalhando nessa nova branch e considerar aqueles commits como **“desfeitos”**.

Só que, se um dia você precisar da branch master, você verá que essa estratégia de “desfazer” também não é uma das mais apropriadas.

Porque a partir do momento que voltamos para a master, nossas alterações serão perdidas, uma vez que elas só existem dentro da branch **“nova_branch_corrigida”**.

Claro que se voltarmos para a branch principal (master), executarmos um merge:

```
git merge nova_branch_corrigida
```

E em seguida corrigirmos os conflitos que serão gerados (por meio do seu editor de código favorito), e por fim realizarmos um commit:

```
git commit -am “Correções no Merge”
```

Veremos no **git log --oneline** que um novo commit na ramificação principal (master) foi criado, de modo a “desfazer” os commits anteriores.

```
MINGW64:/c/Projeto

SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git log --oneline

aw3edrt (HEAD -> master) Correções no Merge
b4a1015 4) Atualização de Arquivo
9e1e293 3) Atualização de Arquivo
80c6253 2) Atualização de Arquivo
1d34372 1) Criação de Arquivo
```

Desfazendo commits com o Revert

O git conta com uma função capaz de registrar alguns novos commits para reverter o efeito de alguns commits anteriores. Mas como assim?

Vamos continuar na ideia que ainda temos 4 commits que foram realizados:

```
MINGW64:/c/Projeto

SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git log --oneline

b4a1015 (HEAD -> master) 4) Atualização de Arquivo
9e1e293 3) Atualização de Arquivo
80c6253 2) Atualização de Arquivo
1d34372 1) Criação de Arquivo
```

Sabemos que a partir do commit **9e1e293**, temos uma informação errada, aonde diz que o Git não é um sistema de versionamento de código.

```
git revert hash
```

Com este comando nós podemos criar um novo commit que contenha somente as alterações feitas a partir de um determinado commit anterior.

Ou seja, será por meio do **Revert** que o git executará a seguinte função:

1) Git, volte para o estágio do commit anterior informado pelo usuário.

2) E crie um novo commit acima do commit (HEAD, 'b4a1015'), onde o conteúdo será o mesmo do commit anterior.

Lembrando que após executar o comando **git revert 9e1e293** você ainda precisa decidir quais alterações você deseja manter:

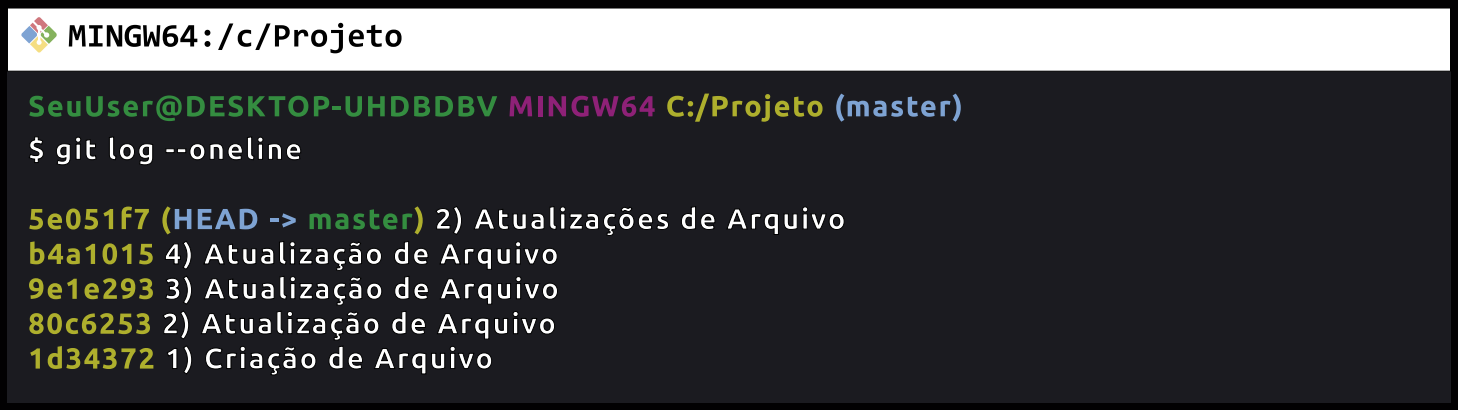
```
[Inicio]: Criação de Arquivo
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
=======

[Atualização 1]: "GitHub é uma plataforma bem legal".
>>>>>>> parent of 9e1e293 (3) Atualização de Arquivo) (Incoming Change)
```

Após fazer sua escolha, basta apenas realizar um novo commit normalmente:

```
git add .
git commit -m “Correções do Commit 3”
```

E pronto, voltando agora para a master e executando o **git log –oneline**, veremos que um novo commit foi realizado:



Se analisarmos bem, veremos que esse tipo de estratégia é muito parecida com aquela de criar uma nova branch, voltar para a principal e fazer um merge, não é verdade?

Ao contrário da estratégia anterior, com **revert** a gente pode continuar usando a mesma ramificação (master), sem precisar guardar o hash de um determinado commit, ou quem sabe criar uma branch, e ter que fazer essa volta toda, não é verdade?

Este é um método ideal para “desfazer” alterações durante o versionamento do nosso projeto, mas se você deseja manter um histórico mais enxuto e limpo, essa estratégia pode não ser satisfatória, uma vez que ela polui um pouco a visualização dos nossos commits.

O Git também conta com outra função mais HARDCORE capaz de desfazer de forma definitiva, limpa e organizada os commits que queremos.

```
git reset hash
```

Com este comando você será capaz de resetar o histórico de commits, sem modificar o conteúdo .

No caso, quando executamos o **git reset hash**, estamos apagando o histórico de commits que foram feitos após o hash informado, sem alterar em nada o conteúdo desses arquivos.

Por exemplo, caso executarmos o comando:

```
git reset 80c6253
```

Veremos que o arquivo de texto que estamos trabalhando não alterou em nada.

Mas se executarmos o **git log –oneline**, veremos que os outros commits não existem mais:

```
MINGW64:/c/Projeto

SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git log --oneline

80c6253 (HEAD -> master) 2) Atualização de Arquivo
1d34372 1) Criação de Arquivo
```

E vamos notar que o arquivo foi tirado da área de Stage e movido para UnStage.

```
git reset --hard hash
```

Com este comando você será capaz de resetar o histórico de commits, voltando o conteúdo dos arquivos para o estado anterior

Ou seja, diferente do comando anterior, que não leva a flag **–hard**, aqui nós não só estamos apagando o histórico dos nossos commits como também, ficando com o conteúdo presente no commit anterior.

Nesse caso, quando usamos a flag **–hard**, estamos desfazendo de vez as alterações que fizemos, como se elas nunca tivesses existido.

Portanto, muito cuidado quando for trabalhar com o reset, ok?

Como vimos, este método de desfazer alterações tem o efeito mais limpo no histórico. No entanto, o reset adiciona complicações ao trabalhar com um repositório remoto, ainda mais quando este é compartilhado.

Por exemplo, supondo que o seu repositório remoto contenha os seguintes commits:

+ jaus777
+ sbag1tt
+ahysu88
+shayz77

E localmente você execute o comando **git reset –hard ahysu88,** a partir do momento que você tentar realizar um **push**.

O Git vai supor que a ramificação (branch) que está sendo enviada não está atualizada devido aos commits ausentes.

Logo você receberá uma mensagem de erro.

Portanto, nesse cenário o **git revert** deve ser o seu método de 'desfazer' escolhido.

Reset com –soft, --mixed, --hard

Até o momento nós vimos dois comandos:

git reset hash: que aponta todo o histórico de commits posteriores para o commit atual, sem modificar nada nos arquivos.

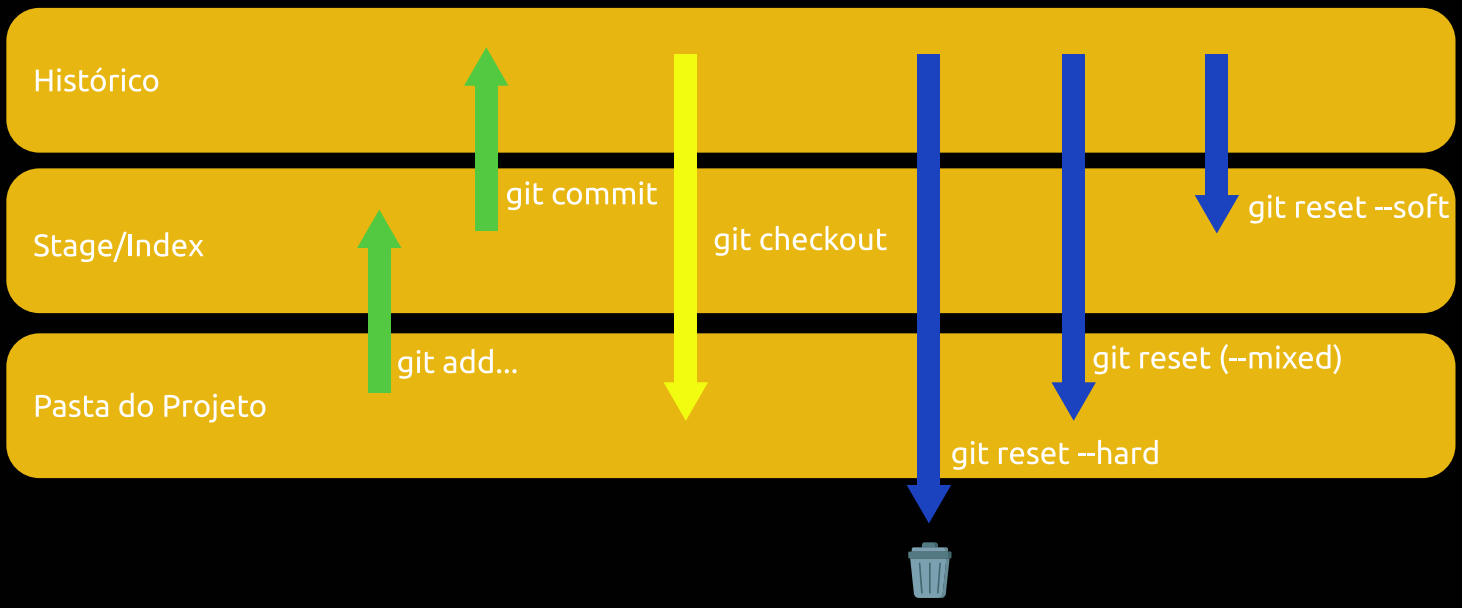
git reset –hard hash: que apaga todo o histórico de commits posteriores modificando também os conteúdos dos arquivos, fazendo com que você fique somente com atualizações realizadas naquele commit.

Temos também outros dois tipos de comandos:

git reset –soft hash: aqui ele pega todos os commits posteriores, e jogam na área de Stage/Index no commit atual.

git reset –mixed hash: funciona da mesma forma que o git reset hash.

Vejamos uma representação gráfica de como isso tudo funciona:



Git Clean

Você sabia que é possível se desfazer de arquivos que não estão comitados?

Sim, isso é possível de ser feito por meio do comando **git clean**.

`git clean` Com este comando você consegue se desfazer de arquivos que estão marcados como não rastreáveis.

Arquivos não rastreáveis são aqueles arquivos que ainda não foram adicionados ao índice de rastreamento do seu repositório local (por meio do comando **git add .**).

Para exemplificar a utilização desse comando, vamos criar uma pasta chamada de “meu-projeto” e dentro dela vamos fazer as configurações inicias do git:

```
git init
```

Em seguida vamos criar um novo arquivo de texto chamado de **file.txt** com o seguinte conteúdo:

```
1  Olá Mundo!
```

Ao executarmos o comando **git status**, veremos que o arquivo de texto que acabamos de criar se encontra marcado como UnTracked.

Se neste momento, se executarmos o comando git clean, automaticamente o git vai nos retornar um erro alegando que o comando git clean não funciona sozinho, dizendo que podemos usa-lo com algumas flags:

git clean -n: Este comando executará um **“dry run”** do git clean, ou seja, ele mostrará quais arquivos serão removidos sem removê-los.

É uma forma de dizer: “Você tem certeza que serão esses arquivos que devem ser removidos?”.

git clean -f: Este comando inicia a exclusão real de arquivos não rastreáveis (UnTracked) existente na pasta do nosso projeto.

A **Flag -f**, significa **“force”**, isso quer dizer que estamos apagando os arquivos não rastreáveis de maneira **“forçada”**.

Como assim forçada?

Lembra quando você tentou executar o comando **git clean**, ele mostrou a seguinte mensagem:

clean.requireForce defaults to true and neither -i, -n, nor -f given; refusing to clean

Quando iniciamos o git em um repositório, de forma padrão, ele deixa desativado a opção de usarmos o **git clean** sem a **flag force (-f)**.

Isso significa que se quisermos utilizar o comando **git clean** sem a **flag -f**, precisamos setar a configuração **“clean.requireForce”** para **true**.

Para ativar essa opção (somente na pasta do projeto), use o comando:

```
git config --local clean.requireForce true
```

Para ativar essa opção (de forma global), use o comando:

```
git config --global clean.requireForce true
```

Para desativar, basta executar o mesmo comando trocando **true** por **false**.

Para removermos um arquivo específico que estava marcado como Untracked podemos usar o comando:

```
git clean -f <path>
```

No local aonde está escrito **<path>** insira o nome ou caminho do arquivo.

Para removermos **diretórios marcados como UnTracked** podemos usar o comando:

```
git clean -df <path>
```

A opção -d informa ao git clean que você também deseja remover todos os diretórios não rastreados; por padrão, ele ignorará os diretórios.

Para removermos **arquivos ignorados** pelo git podemos usar este comando:

```
git clean -xf <path>
```

Lembrando novamente que no local aonde está escrito **<path>** insira o nome ou caminho do arquivo.

END (y/n)

Gostou desse material?

Então não deixe de acessar a nossa seção de Git & GitHub do básico ao avançado.