

# GIT - ITAndroids COMP

Eric Moreira (Precioso, T20)

2017



## 1 Prelúdio

Essa apostila foi escrita por mim no fim do meu primeiro ano de ITA e da ITAndroids, na qual tive bastante experiência usando git. Ela serve para explicar git do zero, não exijo nenhum conhecimento prévio, a não ser alguns comandos básicos de linux e uma conta no site Bitbucket. Recomendo ler esse material uma vez, enquanto acompanha os projetos que estão junto dele. Quando quiserem revisar o tópico, podem usar esse mesmo material como referência. Por fim, divirtam-se! Minha experiência na ITAndroids foi extremamente divertida e eu aprendi muito. Nesse início pode ser meio cansativo ficar apenas em treinamentos, mas quando vocês começam a fazer coisas de engenharia de verdade e sentir sua presença no seu time, tudo vale a pena =) Aproveitem, beijos.

-Precioso, magnífica T20

## 2 GIT básico

Git é um software de controle de versão, ou seja, que permite que você mantenha controle das mudanças feitas a um projeto ao decorrer dele. Git trabalha recordando mudanças feitas a um projeto, armazenando-as e as acessando quando o usuário desejar.

### Mas para que o Git serve?

Simples. O Git facilita o compartilhamento do projeto, mantendo ele reservado num certo local que pode ser acessado online por qualquer participante do projeto a qualquer hora. Duas pessoas podem trabalhar no projeto ao mesmo tempo, enviar as mudanças feitas por cada um e receber as feitas pela outra pessoa de maneira fácil. Além disso, caso seja feito algo de errado no código, o uso de Git pode desfazer essa mudança, retornando o projeto a sua versão anterior ao erro.

### 2.1 Introdução

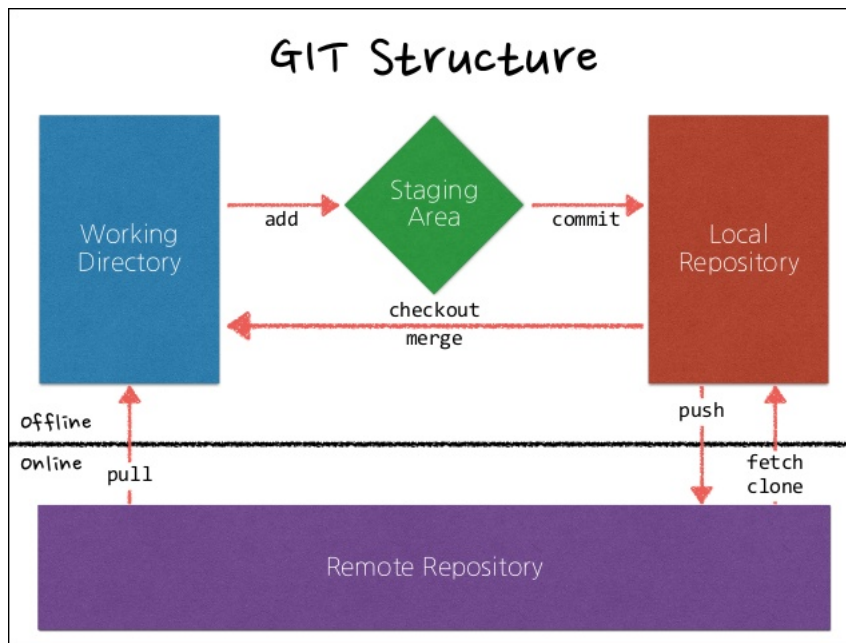
**Repositório Remoto:** Local online onde seu projeto fica armazenado. Ele pode ser acessado por qualquer usuário que receba acesso ao projeto. Ele guarda todo o projeto e lá só são enviadas as mudanças definitivas no código, após serem feitas e testadas.

Os “locais online” onde os diretórios ficam armazenados estão armazenados em um site próprio para esses objetivos. Na ITAndroids, o site utilizado é o *Bitbucket*, mas outra alternativa bem comum é o *GitHub*.

**Repositório Local:** Cópia do Repositório Remoto que fica armazenada no PC do usuário e pode ser acessada a qualquer momento. É por onde o usuário pode acessar o projeto e receber atualizações do repositório remoto.

**Diretório de Trabalho:** Localizado na sua própria máquina, ao contrário do repositório, cada membro possui seu próprio diretório de trabalho. É onde cada um pode fazer sua alteração no projeto. Todas as alterações feitas podem ser revertidas à última versão armazenada no repositório. Todas as alterações são feitas no código localizado na cópia local e, quando essas mudanças chegam ao fim, deve-se enviá-las para o repositório, onde ficarão visíveis para os outros usuário e constituirão uma mudança permanente no projeto.

**Área de Preparação:** Também local, da mesma maneira que o diretório de trabalho. Serve para diferenciar os arquivos modificados que devem ser enviados para o repositório dos arquivos cujas mudanças devem ser revertidas. Depois de modificar seu projeto, o usuário envia alguns arquivos para a área de preparação. Outros, ele decide que devem permanecer na cópia local do projeto. Os que estavam na área de preparação podem então ser enviados para o repositório, enquanto os que estão no diretório de trabalho tem suas mudanças descartadas. Nenhum arquivo ou modificação pode ir diretamente da cópia local para o repositório. Todos passam pelo intermédio da área de preparação.



## 2.2 Comando básicos do GIT

**git init:** Clona o repositório online no computador do usuário, criando um repositório local.

**git status:** Pode ser acionado a qualquer momento dentro do repositório local. Quando usado, o terminal exibe quais arquivos estão na Área de Preparação (Seu nome estará da cor verde) e quais estão no Diretório de trabalho (Estarão vermelhos).

**git add:** Quando decidimos que já terminamos de alterar um documento, podemos usar *git add nome* para passá-lo do Diretório de Trabalho para a Área de Preparação.

Obs: Existe um atalho para o git add quando queremos adicionar um volume grande de documentos: *git add -A*, que adiciona todos os documentos no diretório de trabalho. No entanto, ele não é recomendado e **nunca** é usado na ITAndroids por ser considerado má prática, pois podemos facilmente cometer um erro e enviar algo que não deveria.

**git commit -m "Mensagem":** Esse comando envia as atualizações que estavam na Área de Preparação para o Repositório Local. Commits geralmente são feitos apenas quando temos certeza do que estamos enviando, ou seja, já compilamos e testamos um código. Junto do Commit sempre enviamos uma mensagem descrevendo as alterações contidas no commit, como "Otimizações do movimento de levantar do robô" ou "Resolvendo problema da comunicação".

**git log:** Funciona como um histórico de commits, listando os últimos commits, quando foram feitos, quem fez.

**git push:** Diz para onde enviar nossas alterações quando estivermos prontos. Envia todos os commits feitos e suas respectivas alterações do repositório local para o repositório online. É muito importante que você entenda que esse comando só deve ser executado quando você tiver certeza de que suas alterações estão corretas. Gosto de pensar num git push como se estivesse enviando um lab de CES-10.

**git pull:** Análogo ao comando anterior, git pull atualiza o repositório local. Digamos que um de seus colegas alterou o repositório online, então você precisa atualizar seu repositório local com as alterações que ele fez (Caso não atualize, pode acontecer um *conflito*, o qual falarei depois). Logo, sempre que vocês forem atualizar o código antes de trabalhar, entrem no diretório do repositório local e usem esse comando.

**git diff:** Usado para encontrar as diferenças entre algum dos commits do usuário e o repositório local. Geralmente, é utilizado logo depois do git pull para descobrir a diferença entre o último commit do usuário e o código que ele acabou de receber do repositório online. Assim, ele evidencia as alterações feitas no projeto

**git reset:** Faz exatamente o oposto de git add. Ao usar *git reset nomeDoArquivo*, retiramos um arquivo da Área de Preparação e o retornamos ao Diretório de Trabalho.

**git branch nomeDaBranch:** Esse comando é extremamente importante, pois introduz o conceito de *branch*.

Branch: O conceito de branch é análogo a um galho de uma árvore. Quando voc cria uma branch, é como se as atualizações no seu projeto formassem um tronco e você criasse um galho novo. Um dos galhos será acessado por todos os membros do projeto enquanto na sua branch só quem entrar nela poderá alterar o código. Qualquer alteração feita em uma branch não interfere nas outras branches, e você pode alterá-la como bem entender sem interferir no projeto principal, impedindo que você atrapalhe ou seja atrapalhado pelo trabalho dos outros membros.

Normalmente, todos os membros criam sua própria branch e trabalham independentemente. A branch "padrão" é a *branch master*, e ela é a branch principal, a qual todos tem acesso. Ao fim do trabalho, todos "fundem" (merge) sua branch com a master, passando todas as suas modificações.

O comando git branch nomeDaBranch cria uma branch nova com o nome "nomeDaBranch".

**git checkout novaBranch:** Permite sair da branch atual e trocar para a branch especificada.

## 2.3 Projetos

**Projeto 1 - Áreas da ITAndroids** O atual diretor executivo da ITAndroids, Dono, está muito ocupado resolvendo as listinhas de MAT. Sabendo que você é o bixão do GIT, ele acabou de te mandar um e-mail e uma mensagem no What-sapp te sugando para ajudá-lo a atualizar o `areas.txt`, um arquivo de texto que dá informações básicas sobre as áreas da ITAndroids, e você decidiu ajudá-lo.

Passo 1: Clone o repositório Treinamentos 2017 e acesse o arquivo '`areas.txt`' em `/Treinamento git/Projeto 1`.

Passo 2: Use algum dos comandos acima para verificar se o arquivo está no Diretório de Trabalho ou na Área de Preparação.

Passo 3: Perceba que as áreas Controle e Processamento de Sinais não foram incluída no `.txt`. Seguindo o macaco, adicione informações lá (Não sabe? Fale com mais membros da iniciativa e busque saber um pouco sobre todas as áreas do projeto :D )

Passo 4: Adicione `area.txt` na Área de Preparação.

Passo 5: Use o mesmo comando do passo 2 e veja a diferença!

Passo 6: Faça um commit.

Passo 7: Veja um histórico dos seus commits.

## 3 Backtracking com Git

Quando trabalhamos com Git, muitas vezes realizamos mudanças no código que fazem com que ele não funcione de maneira adequada. Para essas situações, o Git nos oferece ferramentas para desfazer nossos erros. **git show:** Em Git, o commit que você se encontra atualmente é conhecido como o *HEAD* commit. Para verificar as mudanças feitas no commit mais recente, basta digitar: `git show HEAD`.

Outra situação bem comum com a qual nos deparamos quando temos um projeto envolvendo Git é quando fazemos alguma mudança errada no nosso diretório de trabalho e então decidimos apagá-la depois. Uma maneira ingênua seria *Ctrl + Z* repetidos ou então reescrever o documento corretamente. Porém, isso está muito sujeito a erros. Git nos oferece uma solução muito mais prática com o seguinte comando:

**git checkout HEAD filename:** comando o qual restaura o arquivo selecionado, fazendo com que ele fique igual estava no último commit. Qualquer alteração feita desde esse último commit será desconsiderada.

Outra situação interessante que também acontece com certa frequência é quando você adiciona certos arquivos do seu diretório de trabalho para a área de preparação (*Staging Area*) usando um comando **git add**. De repente, você percebe um erro no arquivo que já estava na Área de Preparação (Por exemplo, um bug no seu código). Você pode então retirar um arquivo de lá usando o comando **git reset HEAD filename**. Ele irá retornar o arquivo especificado

para o diretório de trabalho, onde poderemos modificá-lo.

Esse, porém, é apenas um uso mais "simples" do comando `git reset`. Ele tem uma aplicação muito mais interessante em grandes projetos. Para abordá-la, vamos falar antes do seguinte conceito:

**SHA:** SHA é quase como um código (Uma longa sequência numérica) que identifica o seu commit. Cada commit tem seu SHA. Usando o comando **git log**, conseguimos ver os commits mais recentes, assim como seus SHAs.

```
commit 7be7ec672af73cf31ef72c92e3374fd4e29c675a
Author: danasselin <johndoe@example.com>
Date: Tue Nov 3 17:15:05 2015 -0500

    Add first page of scene-7.txt

commit 83f7b3591f4ab7aedb3160388b59e65ee1cd94a2
Author: danasselin <johndoe@example.com>
Date: Tue Nov 3 17:14:48 2015 -0500

    Add first page to scene-5.txt

commit 5d692065cf51a2f50ea8e7b19b5a7ae512f633ba
Author: danasselin <johndoe@example.com>
Date: Tue Nov 3 17:14:30 2015 -0500

    Add first page to scene-3.txt

commit 27a3bfe282808aef69d04f4d111e7a6a0c652dd
Author: danasselin <johndoe@example.com>
Date: Tue Nov 3 17:14:08 2015 -0500

    Add first page to scene-2.txt

commit 96f1625d347d599e6f0f13f23022b3f852d5b116
Author: danasselin <johndoe@example.com>
Date: Tue Nov 3 17:13:38 2015 -0500

    Add .gitignore
$
```

Na figura acima, pode-se observar claramente a estrutura que foi mencionada acima: cada commit possui seu próprio SHA, que funciona como o id do commit.

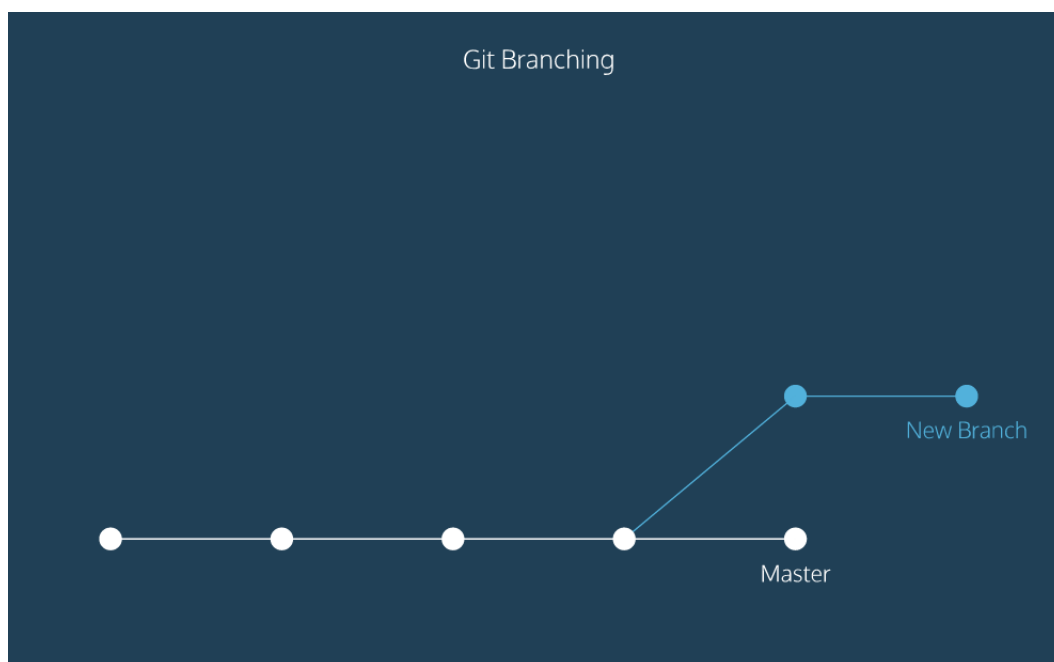
Agora, o comando que queria explicitar é **git reset SHA**. Colocando no lugar de SHA os *7 primeiros dígitos* do SHA de seu commit, é possível retornar seus arquivos para ficarem idênticos ao de commits anteriores que forem mostrados no `git log`. A ideia é que cada commit funciona como back-up. Você pode voltar para ele quando quiser.

## 4 Git Branching

Até agora, nosso estudo de GIT tem acontecido numa única branch, chamada de **master**. Porém, o Git permite que cada projeto possua diversas branches (Galhos). Imagine que um projeto com várias branches é como um jogo com diferentes finais alternativos: algumas vezes o final do jogo é ruim (*bad ending*) e em outras o final é feliz (*happy ending*). Cada final ocorre de forma paralela e independente a outro, mas ambos existem simultaneamente.

Nesse módulo, abordaremos os aspectos teóricos de trabalhar com Git em diferentes branches.

**git branch:** Esse comando responde à pergunta: *"Em qual branch estou?"*



No esquema acima, cada círculo em branco é um commit. Observe que podemos criar uma branch nova a qualquer momento. Ela receberá todos os commits da branch Master até aquele momento. Porém, em seguida, ela seguirá seu próprio trajeto com seus próprios commits.

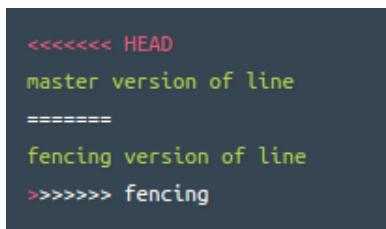
**git branch new-branch:** Esse comando serve para criar uma branch nova. No caso, new-branch é o nome da nova branch criada. Uma boa prática é dar nomes que façam sentido!

*Obs.:* Nomes de branches não podem conter espaços em branco. Por exemplo, "new-branch" é nome válido, mas "new branch" não é.

**git checkout nome-da-branch:** Troca da branch em que você está para a branch *nome-da-branch*.

Até agora, vimos como separar o projeto em diversas branches. Isso é extremamente útil quando temos diversas pessoas trabalhando num projeto, pois dessa forma o trabalho de cada uma é mais independente. Porém, quando entregamos a versão final de um projeto, temos que ter uma única versão final. Ou seja, uma única branch. Para garantir isso, usamos o seguinte comando:

**git merge nome-da-branch:** Como o próprio nome diz, o comando tem a intenção de "fundir" a branch e a master de maneira inteligente. Ele substitui as partes do código que existem em uma, mas não na outra. As partes que existem em ambos os códigos (Chamadas de partes conflitantes) são sinalizadas e devem ser resolvidas pelo usuário.

A imagem mostra um exemplo de conflito de código no Git. O texto é colorido para indicar o status das linhas: vermelho para linhas adicionadas pelo HEAD, verde para linhas adicionadas pela branch 'fencing', e amarelo para linhas conflitantes. O texto é o seguinte:  
<<<<<< HEAD  
master version of line  
=====  
fencing version of line  
>>>>>> fencing

A imagem acima demonstra um exemplo claro de conflito ao usar o git merge. Para resolvê-lo, devemos apagar toda a estrutura da imagem, exceto as partes identificadas por *master version of the line* ou *fencing version of the line*, dependendo se queremos manter a versão da master ou da branch a ser fundida.

**git branch -b nome-da-branch** Esse comando apaga a branch "nome-da-branch". Geralmente usamos esse comando depois do git merge. Afinal, como passamos as modificações da branch para a master, não precisamos mais dela em nosso projeto.

## 5 Cooperação com Git

Nesse último módulo, abordamos como começar a usar GIT com múltiplos usuários. Até agora, nosso uso de GIT foi totalmente focado em um usuário em si, num modelo onde cada um trabalha em sua branch e o projeto flui.

Entretanto, muitas vezes teremos que cooperar com outros programadores numa mesma branch, trabalhando simultaneamente. Para esse tipo de trabalho com múltiplos programadores, usamos um **Remote**, ou seja, um repositório compartilhado online que pode ser acessado e modificado por várias pessoas: no caso da ITAndroids, usamos o Bitbucket. Outro serviço bem comum é o Github.



**git clone remote-location clone-name:** Esse comando serve para clonar um repositório online, para que o usuário possa ter uma cópia idêntica dele localmente em seu computador. Você deve modificar essa cópia, enviar as modificações para o repositório online e atualizar o seu próprio repositório caso alguma outra pessoa tenha feito alguma alteração. *remote-location* diz ao Git onde achar esse repositório remoto. Já *clone-name* é o nome que você dará ao diretório no qual o GIT colocará o repositório clonado.

Agora, digamos que você fez modificações no código e foi dormir. Quando você acorda, outro programador alterou o código. Se você tentar atualizar o código com as modificações dele, as suas serão sobrescrevidas! Uma maneira de verificar o que foi alterado num projeto com GIT e trazer as mudanças para o seu repositório local (clonado) é usando o comando **git fetch**. Porém, ele não é a mesma coisa que **git merge**. Ele deixa as alterações numa *branch remota*, chamada de "origin/master". Para fundir essa branch ao seu código atual e selecionar quais mudanças serão sobrescritas, basta usar **git merge origin/master**.