Controle de versão utilizando Git

André G. C. Pacheco
http://www.pachecoandre.com.br
Janeiro de 2017

---

1 Introdução

O Git é um sistema de controle de versões distribuído, usado principalmente no desenvolvimento de software, mas pode ser usado para registrar o histórico de edições de qualquer tipo de arquivo. O Controle de Versão permite que você registre as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que se possa recuperar versões específicas. Um Sistema de Controle de Versão também permite que você reverta arquivos para um estado anterior, compare mudanças feitas no decorrer do tempo, veja quem foi o último a modificar algo que pode estar causando problemas, ou quem introduziu um bug e quando, além de muitas outras alternativas. E tem mais, se você estiver usando um controlador de versão você não precisará se preocupar se o computador estragou ou se perdeu algum arquivo, pois com o sistema poderá facilmente reavê-los [1]. O maior uso do Git é para trabalhar em equipe, uma vez que os arquivos podem ser editados em paralelo e depois versionados. Porém, mesmo que não esteja trabalhando em equipe, utilizar o Git é uma excelente de maneira de controlar as versões e evitar problemas como "alterei o código e agora não está funcionando da maneira correta". Esse tipo de problema é facilmente solucionado com uso do Git.

1.1 Funcionamento básico

O Git funciona de maneira descentralizada, isso significa que existe um repositório em algum servidor e as estações de trabalho mantém cópias completas de todos os arquivos versionados. Portanto, todas as versões desenvolvidas ao longo do tempo também estará nas estações locais. A forma que o Git trabalha com cada projeto também é interessante. Cada um deles é dividido em 3 estágios, como ilustrado na Figura 1 1. O primeiro estágio é o diretório em que estaremos trabalhando. Esse diretório é criado quando vamos no repositório e trazemos os arquivos que estão nele para iniciarmos o nosso desenvolvimento. Logo, tudo que for feito neste momento, estará dentro deste diretório. Quando eu utilizo

[1] https://www.oficinadanet.com.br/post/16111-o-que-e-e-como-funciona-o-git-e-github

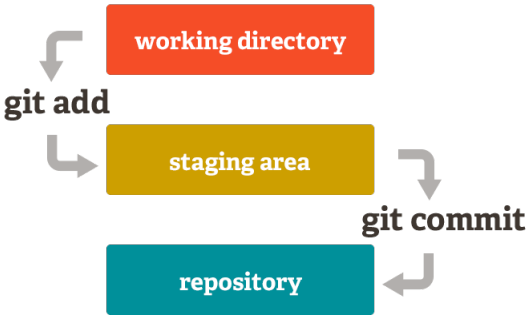

Figura 1: Estágios de um projeto no Git

o comando `git add` todos os arquivos modificados vão para *staging area*, uma espécie de sala de espera na qual os arquivos modificados ficam antes de serem enviados para o repositório. Quando o comando `git commit` for utilizado todos arquivos da *staging area* serão enviados para o repositório principal do projeto. Não se preocupe com esses comandos no momento. Vamos explicá-los ao longo deste documento.

1.2 Instalação

A instalação do Git é muito simples e fácil. Para usuários linux, basta um `sudo apt-get install git`. Para os demais, basta acessar o site https://git-scm.com/ e seguir as instruções oficiais dos desenvolvedores.

1.3 Configuração inicial básica

A configuração inicial básica é informar o nome de usuário e email que serão utilizados no Git. Para isso, abra o seu terminal (quando eu disser terminal, entenda como o prompt de comando do seu sistema operaciona) e utilize os comando:

```
> git config --global user.name "<USER_NAME>"
> git config --global user.email "<USER_EMAIL@provedor.com>"
```

2 O básico para utilização do Git

2.1 Cenário I: repositório local

2.1.1 Criando um repositório

Vamos começar com um primeiro cenário de desenvolvimento na qual um único desenvolvedor trabalha localmente desenvolvendo o seu projeto. No caso, mesmo que exista

apenas um desenvolvedor, o git é excelente para versionar o código. Portanto, para iniciar um repositório Git em um diretório qualquer basta abrir um terminal, navegar até essa pasta e utilizar o seguinte comando:

```
> git init
```

Neste momento, o Git cria uma pasta oculta `.git` na qual será armazenado todas as versões do projeto em questão. Para verificar a situação atual do repositório podemos utilizar:

```
> git status
```

Caso você tenha executado o comando, vai receber uma mensagem dizendo que não existe nada a ser enviado para o *git repository*, em outras palavras, nao existe nada para ser comitado. Agora inclua qualquer arquivo nesta pasta. Neste momento, se verificar o status novamente, você receberá uma mensagem dizendo que existem arquivos que não estão sendo *trackeados* pelo Git. Portanto, precisamos adicioná-los para a *staging area*. Para fazer isso, utilizamos o comando:

```
> git add <NOME_DO_ARQUIVO>
```

Neste momento, o arquivo passa para a *staging area*. Caso tenha mais arquivos, você pode utiliar caracteres coringas para inserir todos ou parte deles. Por exemplo, `git add *.html` para adicionar todos os arquivos html da pasta. Para adicionar todos os arquivos, independente de formato, podemos utilizar `git add *` ou `git add .` (ambos fazem a mesma coisa). Estes são os usos básicos do `git add`.

Como já mencionamos, arquivos na *staging area* precisam ser enviados para o *git repository*. Caso execute o comando `git status` novamente, você vai observar que é necessário realizar um commit, que nada mais é do que uma confirmação das mudanças do projeto. Portanto, agora vamos "comitar"o nosso projeto:

```
> git commit -m "<MENSAGEM DO COMMIT>"
```

Observe que o Git que devemos escrever uma mensagem no commit. Essa mensagem indica as mudanças realizadas nessa versão. Ao executar o commit, será exibido um resumo do que foi realizado pelo Git. E se você verificar novamente o status do repositório, você vai observar que não existe nenhuma alteração pendente para ser enviado para versionamento. Esse é ciclo básico do funcionamento do Git.

Caso você ache chato toda hora ter que adicionar arquivos com o `git add`, podemos economizar um comando fazendo:

```
> git commit -a -m "<MENSAGEM DO COMMIT>"
```

O `-a` informa ao Git para adicionar todos os arquivos modificados e prosseguir com o commit. A desvantagem é que não será possível remover um arquivo da *stagin area* (vamos fazer isso mais para frente) caso necessário.

2.1.2 Evitando que um arquivo seja versionado

Agora imagine a seguinte situação. Você está programando e existe um arquivo de configuração que não deve ser enviado para o repositório. Para isso, existe uma maneira do Git ignorar esse tipo de arquivo. Para fazer isso, basta criar um arquivo na pasta do seu projeto chamada `.gitignore`. Neste arquivo, você insere o *path* de todos os arquivos que devem ser ignorados pelo Git. Por exemplo, caso você queira ignorar um arquivo chamado `config.txt` e todos os arquivos de uma pasta chamada `arquivos`, o arquivo `.gitignore` deve conter:

```
config.txt
arquivos/*
```

A partir deste momento, o Git entende que esses arquivos não devem ser monitorados, e caso você verifique o status do repositório, ele não acusa alterações nestes arquivos.

2.1.3 Visualizando alterações de arquivos e versões

Ao modificar um arquivos e verificar o status do repositório, o Git nos apresenta os arquivos que foram modificados. Caso queira verificar o que foi alterado, você pode utilizar o comando:

```
> git diff
```

Será exibido no terminal todas as alterações de todos os arquivos. Ao adicionar os arquivos modificados na *stating area* e executar o comando `git diff` novamente, o Git não retorna nenhuma diferença. Isso ocorre pois todas as alterações do projeto já foram adicionadas. Neste caso, para saber o que tem de diferença nos arquivos que estão na *staging area*, pode ser utilizado o comando:

```
> git diff --staged
```

O efeito é o mesmo do comando anterior, mas agora considerando os arquivos da *staging area*. Caso você commit as mudanças, e verifique as diferenças na *staging area* nenhuma diferença é retornada, ou seja, mesmo efeito anterior.

Para visualizar todos os commits realizados no projeto, vamos utilizar o comando:

```
> git log
```

Será retornado no terminal, em ordem cronológica, todos os commits já realizados. VocÊ pode observar que será retornado uma chave, que pode ser utilizado para retornar o projeto para aquela versão específica, o autor do commit, a data e o comentário realizado. Caso queira exibir mais detalhes dos commits, pode ser utilizados alguns comandos extras:

```
> git log -p
```

Além das informações do comando anterior, também vai mostrar o `diff` de cada commit, ou seja, é uma combinação do `git log` com o `git diff`. Observe que esse comando pode retornar um log muito grande. Com isso, para limitar o número de commits retornado no terminal, podemos utilizar outro comando extra:

```
> git log -p -<NUMERO>
```

Neste caso, serão retornados `<NUMERO>` commits no terminal. Uma outra para melhorar a exibição é utilizando:

```
> git log -pretty=oneline
```

Este comando exibe todos os commits utilizando apenas uma linha para exibir a chave e o comentário. Existem outros comando para filtrar commits por usuários, por data etc. Porém, visualizar todas essas informações no terminal é um tanto quanto complicado. Por conta disso, o Git possui um visualizador gráfico para esse tipo de inspenção. Para utilizar:

```
> gitk
```

Uma janela será aberta com todas as informações com uma interface que facilita a visualização e inspeção. Pode ser que seja necessário instalar a ferramenta. Para usuários linux, basta utilizar `sudo apt install gitk`.

2.1.4 Revertendo situações no repositório

O primeiro caso a ser exemplificado é quando realizamos um commit antes da hora (isso certamente vai acontecer em algum momento). Vamos supor que enviamos um commit mas deveríamos incluir uma nova modificação na qual esquecemos. Para editar o commit anterior, fazemos:

```
> git commit --amend -m "<COMENTARIO QUE VAI SUBSTITUIR O ANTERIOR>"
```

Este comando vai alterar o último commit pelo enviado neste momento. Em resumo, não é criado um novo commit, o que ocorre é uma substituição.

Agora imagine que enviamos um novo arquivo no nosso diretório. Mas gostaríamos de deixá-lo de fora do commit (ou seja, não adicioná-lo na *staging area*) pois ele é um arquivo de teste. Mas por algum motivo você deu `git commit .` e ele foi incluido. Para remove-lo da *staging area* basta utilizar:

```
> git reset HEAD <FILE>
```

Ao executar este comando o arquivo informado será retirado da *staging area*.

Agora vamos imaginar que editamos um arquivo do projeto que não deveria ser editado, mas só nos tocamos quando verificamos o status do nosso projeto antes de adicionar as mudanças na *staging area*. Em projetos muito grandes, é comum não lembrar o que e onde foi editado. Neste caso, para reverter apenas esse arquivo para a versão anterior dele, ou seja, antes das modificações indesejadas, fazemos:

```
> git checkout -- <FILE>
```

Após executar esse comando, o arquivo informado volta para versão anterior.

Uma outra situação que pode acontecer é a necessidade de remover arquivos que estão versionados. Se apenas apagarmos esse arquivo do repositório e tentar realizar um commit, o Git vai avisar que algum arquivo foi removido. Para remover esse tipo de arquivo, é simples. Assim como adicionamos, vamos utilizar um comando para remover:

```
> git rm <FILE>
```

Dessa forma o arquivo informado será removido e o commit funcionará de maneira correta.

2.1.5 Tags e branchs

A criação de tags no Git serve para realizar marcações no projeto ao longo do desenvolvimento, por exemplo, versão 1, 2, 3 etc. Utilizando uma tag, você tem um atalho para acessar o seu projeto em uma determinada versão, seja para visualizar ou editar. Para listar todas as tags do projeto utilizamos:

```
> git tag
```

Para criar uma tag fazemos:

```
> git tag -a NOME_TAG -m "<COMENTARIO_DA_TAG>"
```

Neste caso será criado uma tag no commit atual do projeto com o nome e comentário informado. Agora se você listar as tags, vai ser possível ver a que foi adicionada. Adicionar uma tag em um commit antigo também é possível e simples. Basta utilizar o seguinte comando:

```
> git tag -a <NOME_TAG> <CHAVE_DO_COMMIT> -m "<COMENTARIO_DA_TAG>"
```

Lembrando que você pode acessar a chave do commit utilizando o comando `git log`.

Para exibir mais informações sobre uma tag com mais detalhes, você pode utilizar o comando:

```
> git show <NOME_TAG>
```

Este comando mostra quem criou a tag, a hora, qual commit etc. Para apagar uma tag também é simples:

```
> git tag -d <NOME_TAG>
```

Agora vamos utilizar a tag de fato. Caso eu queira voltar o meu repositório para uma versão com uma tag, utilizamos:

```
> git checkout <NOME_TAG>
```

Ao executar esse comando, todos os arquivos serão alterados para os mesmos que existiam no momento em que foi criado a tag.

Uma ideia parecida com as tags, mas mais poderosa, são o branchs. Um branch é um ramificação no controle de versão. Com ele é possível trabalhar em diferentes frentes no mesmo projeto. Você pode fazer commits em uma ramificação e eles não alteram as demais. Isso é muito interessante para trabalho em equipe, na qual cada parte dela trabalha em um ponto diferente do projeto. Além disso, uma outro utilizada importante é quanto desejamos adicionar uma feature nova no projeto mas não podemos perder o sistema atual. Para isso, criamos um branch. Por padrão, quando um repositório é criado no Git, ele cria um branch master, que é o principal de trabalho (muitas vezes é único). Para criar um novo branch no repositório, fazemos:

```
> git branch <NOME_BRANCH>
```

Com isso, foi criado um novo branch no repositório. Caso queira listar todos os branchs do sistema, use:

```
> git branch
```

Para mudar o commit atual para o branch, fazemos:

```
> git checkout <NOME_BRANCH>
```

ou podemos utilizar os dois comandos em uma mesma linha adicionando `-b`:

```
> git branch -b <NOME_BRANCH>
```

Agora você pode alterar seus arquivos normalmente e todo commit realizado nele vai alterar apenas o branch selecionado. Para voltar para o branch principal (master) basta utilizar:

```
> git checkout master
```

Os arquivos do projeto são alterados para o da dessa versão e todo commit vai alterar os arquivos deste branch. Portanto, a funcionalidade é muito útil e poderosa, mas devemos ter cuidado para não confundir o branch em que estamos trabalhando e alterar arquivos de maneira errônea.

Agora imagine que você criou um branch para implementar as novas features do sistema e essas features foram implementadas com sistemas. Agora você deseja inserir essas mudanças no branch principal (master). Essa operação é chamada de merge. Para realizar um merge entre dois branch, primeiro entre no merge destino de sua operação utilizando o comando `checkout`. Na sequência, utilize:

```
> git merge <NOME_DO_BRANCH>
```

Este comando vai trazer todas as alterações do branch informado com o `NOME_DO_BRANCH` para o branch que você está no momento. Para apagar o branch que foi mesclado com o atual, você pode utilizar:

```
> git branch -d <NOME_DO_BRANCH>
```

Porém, nem tudo são flores, pode ocorrer conflitos no momento do merge, ou seja, arquivos que possuem dados alterados em diferentes momentos no mesmo local. Com isso, o merge automático vai falhar e será necessário resolver os conflitos manualmente. O Git vai informar os arquivos que estão conflitantes e vai mostrar no branch em que estamos o que está conflitando. Com isso, devemos abrir esse arquivo, solucionar esses conflitos e realizar um commit com essas alterações. Na sequência o merge pode ser realizado novamente e o merge automático vai ocorrer de maneira correta.

2.2 Cenário II: repositório remoto

Até o momento trabalhamos com um repositório local, ou seja, todas as mudanças ficaram versionados na nossa máquina. Todavia, o Git é melhor utilizado com um repositório remoto. Assim, um time de programadores, por exemplo, pode trabalhar no mesmo repositório e compartilhar o mesmo projeto. Neste exemplo vamos utilizar o Github [2] como servidor, mas tenha em mente que o Git pode ser utilizado em uma rede local com um servidor configurado para o seu uso. Se você não conhece ou não possui uma conta no Github, sugiro que a crie e passe utilizar a ferramenta. Após criada a conta no Github, a ideia é que você crie um repositório remoto nele e todos os desenvolvedores utilize o mesmo como repositório. Todos os comandos utilizados anteriormente também serão utilizados aqui.

Então vamos iniciar criando um repositório na nossa máquina utilizando os comandos já mencionados. Abre a pasta em que se deseja criá-lo e digite o comando:

```
> git init
```

Agora essa pasta estará trackeada pelo Git. Porém, esse será nosso repositório local, e o que desejamos é ter um repositório remoto para compartilhar código com uma equipe. Sendo assim, entre no sua conta do Github e crie um novo repositório no mesmo. Feito isso, você vai observar que o seu repositório no Github possui um endereço para ser linkado. Logo, vamos linkar o nosso repositório local com remoto executando o seguinte comando na pasta do repositório local:

```
> git remote add origin <URL_REPOSITORIO_REMOTO>
```

A <URL_REPOSITORIO_REMOTO> será fornecida pelo Github e deve será algo parecido com `https://github.com/[SEU_NOME_DE_USUARIO]/[NOME_DO_REPOSITORIO.git`. Neste momento, o seu repositório local está linkado com o remoto. Agora sempre que você adicionar arquivos para sua *staging area* e na sequência realizar um commit, você estará versionando esse código no seu repositório local. Para enviar as mudanças para o repositório remoto será necessário o seguinte comando:

```
> git push origin master
```

Ao executar este comando você estará enviando a versão do seu último commit para o repositório remoto. O nome `origin` é o nome de origem que o Git cria automaticamente e `master` é o branch que estamos enviando os dados (como já discutimos anteriormente). Após feito isso, você pode atualizar a sua página do projeto e verificar que as mudanças

[2] http://www.github.com

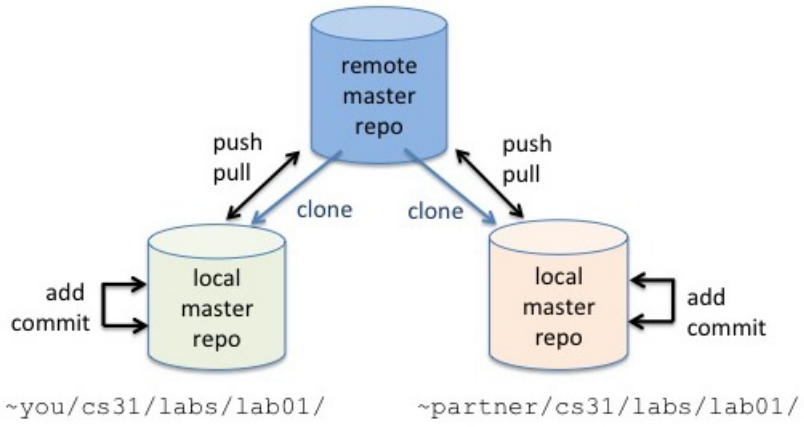

Figura 2: Funcionamento de um repositorio local e remoto

foram enviadas. Lembrando que será necessário incluir seu nome de usuário e senha no Github logo após o comando `push`. Dessa forma somente você e colobadores que você adicionar no Github serão capazes de alterar código neste repositório.

Até o momento você enviou código para o repositório remoto, mas você também tem que consumir dele. Imagine que um colega de trabalho enviou um código para o repositório remoto e agora você precisa trazer essa versão para o seu repositório local. Para isso, utilizamos o seguinte comando:

```
> git pull origin master
```

A ideia é muito similar ao `push`, porém aqui estamos baixando do repositório remoto o último commit enviado para ele. Sendo assim, o nosso repositório local e o remoto ficam sincronizados. Com isso, sempre que você começar a trabalhar no seu código local é importante efetuar um *pull* para sincronizar os repositórios.

Outro comando interessante para repositórios remotos é o `clone`. Como o próprio nome sugere, este comando vai clonar um repositório remoto já criado. Então imagine que um colega já criou o repositório no Github e agora você vai colaborar com ele. Para isso, você precisa trazer esse repositório remoto para sua máquina. Para fazer isso, basta executar:

```
> git clone <URL_REPOSITORIO_REMOTO>
```

Este comando vai criar um repositório local idêntico ao remoto. A partir deste momento você pode começar a trabalhar normalmente no repositório e usufruindo todas as vantangens do Git. Um resumo do funcionamento do repositO local e remoto E apresentado na Figura 2.

Espero que neste momento você tenha entendido a diferença do repositório local e remoto. Caso não tenha, sugiro que leia novamente este documento ou procure por mais

informações online.