Comandos Git

Aprenda Git do básico ao avançado

(https://comandosgit.github.io/)

Instalando e Configurando

Instalação

Windows | Mac | Linux

Instalar o Git no Windows é muito fácil. O projeto msysgit tem um dos procedimentos mais simples de instalação. Simplesmente baixe o arquivo exe do instalador a partir da página do GitHub e execute-o:

```
https://msysgit.github.com (https://msysgit.github.com)
```

Após concluir a instalação, você terá tanto uma versão command line (linha de comando, incluindo um cliente SSH que será útil depois) e uma GUI padrão.

git config

Sua Identidade | Ativando Cores

A primeira coisa que você deve fazer quando instalar o Git é definir o seu nome de usuário e endereço de e-mail. Isso é importante porque todos os commits no Git utilizam essas informações, e está imutavelmente anexado nos commits que você realiza:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Relembrando, você só precisará fazer isso uma vez caso passe a opção --global, pois o Git sempre usará essa informação para qualquer coisa que você faça nesse sistema. Caso você queira sobrepor estas com um nome ou endereço de e-mail diferentes para projetos específicos, você pode executar o comando sem a opção --global quando estiver no próprio projeto.

git help

Obter Ajuda

Se você precisar de ajuda ao usar Git, existem três maneiras de obter a ajuda para qualquer um dos comandos Git:

```
git help {comando}
git {comando} --help
man git- {comando}
```

Criando o projeto

git init

Repositório Git

Você pode obter um projeto Git utilizando duas formas principais. A primeira faz uso de um projeto ou diretório existente e o importa para o Git. A segunda clona um repositório Git existente a partir de outro servidor.

Inicializando um Repositório em um Diretório Existente

Caso você esteja iniciando o monitoramento de um projeto existente com Git, você precisa ir para o diretório do projeto e digitar

```
git init
```

Isso cria um novo subdiretório chamado .git que contem todos os arquivos necessários de seu repositório — um esqueleto de repositório Git. Neste ponto, nada em seu projeto é monitorado.

Primeira versão

Caso você queira começar a controlar o versionamento dos arquivos existentes (diferente de um diretório vazio), você provavelmente deve começar a monitorar esses arquivos e fazer um commit inicial. Você pode realizar isso com poucos comandos

```
touch .gitignore
git add .gitignore
git commit -m "Versão inicial do projeto"
```

Bem, nós iremos repassar esses comandos em um momento. Neste ponto, você tem um repositório Git com arquivos monitorados e um commit inicial.

git clone

Clonando um Repositório Existente

Você clona um repositório com **git clone [url]**. Por exemplo, caso você queria clonar a biblioteca Git do Ruby chamada Grit, você pode fazê-lo da seguinte forma:

```
git clone git://github.com/schacon/grit.git
```

Se você entrar no novo diretório grit, você verá todos os arquivos do projeto nele, pronto para serem editados ou utilizados. Caso você queira clonar o repositório em um diretório diferente de grit, é possível especificar esse diretório utilizando a opção abaixo:

```
git clone git://github.com/schacon/grit.git mygrit
```

Este comando faz exatamente a mesma coisa que o anterior, mas o diretório alvo será chamado **mygrit**.

Básico

git add

Gravando Alterações

Quando um repositório é inicialmente clonado, todos os seus arquivos estarão monitorados e inalterados porque você simplesmente os obteve e ainda não os editou. Conforme você edita esses arquivos, o Git passa a vê-los como modificados, porque você os alterou desde seu último commit. Você seleciona esses arquivos modificados e então faz o commit de todas as alterações selecionadas e o ciclo se repete.

Monitorando Novos Arquivos

Para passar a monitorar um novo arquivo, use o comando **git add**. Para monitorar o arquivo **README**, você pode rodar isso:

```
git add README
```

Se você rodar o comando **git status**, você pode ver que o seu arquivo **README** agora está sendo monitorado. Os arquivos monitorados serão os que faram parte do commit.

git status

Verificando o Status

A principal ferramenta utilizada para determinar quais arquivos estão em quais estados é o comando:

```
git status
```

O comando lhe mostra em qual branch você se encontra. Vamos dizer que você adicione um novo arquivo em seu projeto, um simples arquivo **README**. Caso o arquivo não exista e você execute **git status**, você verá o arquivo não monitorado dessa forma:

```
# On branch master
# Untracked files:
# (use "git add {file}..." to include in what will be committed)
#
# README
nothing added to commit but untracked files present (use "git add" to track)
```

Você pode ver que o seu novo arquivo **README** não está sendo monitorado, pois está listado sob o cabeçalho **"Untracked files"** na saída do comando status. Não monitorado significa basicamente que o Git está vendo um arquivo que não existia na última captura (commit); o Git não vai incluí-lo nas suas capturas de commit até que você o diga explicitamente que assim o faça. Ele faz isso para que você não inclua acidentalmente arquivos binários gerados, ou outros arquivos que você não têm a intenção de incluir. Digamos, que você queira incluir o arquivo **README**, portanto vamos começar a monitorar este arquivo.

git diff

Verificando Mudanças

Se o comando **git status** for muito vago — você quer saber exatamente o que você alterou, não apenas quais arquivos foram alterados — você pode utilizar o comando.

```
git diff
```

Apesar do comando **git status** responder essas duas perguntas de maneira geral, o **git diff** mostra as linhas exatas que foram adicionadas e removidas — o patch, por assim dizer.
Se você quer ver o que selecionou que irá no seu próximo commit, pode utilizar:

```
git diff --cached
```

git commit

Commits

Armazena o conteúdo atual do índice em um novo commit, juntamente com uma mensagem de registro do usuário que descreve as mudanças.
Se usa o commit depois de já ter feito o **git add**, para fazer o commit:

```
git commit -m "Mensagem"
```

Para commitar também os arquivos versionados mesmo não estando no Stage basta adicionar o parâmetro -a

```
git commit -a -m "Mensagem"
```

Refazendo commit quando esquecer de adicionar um arquivo no Stage:

```
git commit -m "Mensagem" --amend
```

O amend é destrutivo e só deve ser utilizado antes do commit ter sido enviado ao servidor remoto.

git reset

Desfazendo Coisas

Em qualquer fase, você pode querer desfazer alguma coisa. Aqui, veremos algumas ferramentas básicas para desfazer modificações que você fez. Cuidado, porque você não pode desfazer algumas dessas mudanças. Essa é uma das poucas áreas no Git onde você pode perder algum trabalho se fizer errado.
Para voltar ao último commit:

```
git reset --hard HEAD~1
```

Para voltar ao último commit e mantém os últimos arquivos no Stage:

```
git reset --soft HEAD~1
```

Volta para o commit com a hash XXXXXXXXXX:

```
git reset --hard XXXXXXXXXX
```

Recuperando commit apagado pelo git reset

Para visualizar os hashs

```
git reflog
```

E para aplicar:

```
git merge {hash}
```

git rm

Removendo Arquivos

Para remover um arquivo do Git, você tem que removê-lo dos arquivos que estão sendo monitorados (mais precisamente, removê-lo da sua área de seleção) e então fazer o commit. O comando **git rm** faz isso e também remove o arquivo do seu diretório para você não ver ele como arquivo não monitorado (untracked file) na próxima vez.

```
git rm -f {arquivo}
```

Se você modificou o arquivo e já o adicionou na área de seleção, você deve forçar a remoção com a opção **-f**. Essa é uma funcionalidade de segurança para prevenir remoções acidentais de dados que ainda não foram gravados em um snapshot e não podem ser recuperados do Git.

git mv

Movendo Arquivos

Diferente de muitos sistemas VCS, o Git não monitora explicitamente arquivos movidos. É um pouco confuso que o Git tenha um comando **mv**. Se você quiser renomear um arquivo no Git, você pode fazer isso com

```
git mv arquivo_origem arquivo_destino
```

e funciona. De fato, se você fizer algo desse tipo e consultar o status, você verá que o Git considera que o arquivo foi renomeado.

No entanto, isso é equivalente a rodar algo como:

```
mv README.txt README
git rm README.txt
git add README
```

O Git descobre que o arquivo foi renomeado implicitamente, então ele não se importa se você renomeou por este caminho ou com o comando **mv**. A única diferença real é que o comando **mv** é mais conveniente, executa três passos de uma vez. O mais importante, você pode usar qualquer ferramenta para renomear um arquivo, e usar add/rm depois, antes de consolidar com o commit.

Branch e Merge

git branch

Branch Básico

Um branch no Git é simplesmente um leve ponteiro móvel para um dos commits. O nome do branch padrão no Git é master. Como você inicialmente fez commits, você tem um branch principal (**master branch**) que aponta para o último commit que você fez. Cada vez que você faz um commit ele avança automaticamente.
O que acontece se você criar um novo branch? Bem, isso cria um novo ponteiro para que você possa se mover. Vamos dizer que você crie um novo branch chamado testing. Você faz isso com o comando **git branch**:

```
git branch testing
```

Isso cria um novo ponteiro para o mesmo commit em que você está no momento.
Para mudar para um branch existente, você executa o comando **git checkout**. Vamos mudar para o novo branch testing:

```
git checkout testing
```

Isto move o HEAD para apontar para o branch testing.

git checkout

Mudando de Branch

Com o **git checkout** você pode mudar de branch, caso a branch ainda não exista você poderá passar o parâmetro **-b** para criar.

```
git checkout -b {nome_da_branch}
```

git merge

Merge Básico

Suponha que você decidiu que o trabalho na tarefa #53 está completo e pronto para ser feito o merge no branch **master**. Para fazer isso, você fará o merge do seu branch **iss53**, bem como o merge do branch **hotfix** de antes. Tudo que você tem a fazer é executar o checkout do branch para onde deseja fazer o merge e então rodar o comando **git merge**:

```
git checkout master
git merge iss53
```

git mergetool

Resolvendo conflitos

Se você quer usar uma ferramenta gráfica para resolver esses problemas, você pode executar o seguinte comando que abre uma ferramenta visual de merge adequada e percorre os conflitos.

```
git mergetool
```

Arquivos Temporários

git mergetool cria ***.orig** arquivos de backup ao resolver fusões. Estes são seguros para remover uma vez que um arquivo foi fundida e sua **git mergetool** sessão foi concluída.

git log

Histórico de Commits

Depois que você tiver criado vários commits, ou se clonou um repositório com um histórico de commits existente, você provavelmente vai querer ver o que aconteceu. A ferramente mais básica e poderosa para fazer isso é o comando:

```
git log
```

git stash

Fazendo Stash

Muitas vezes, quando você está trabalhando em uma parte do seu projeto, as coisas estão em um estado confuso e você quer mudar de branch por um tempo para trabalhar em outra coisa. O problema é, você não quer fazer o commit de um trabalho incompleto somente para voltar a ele mais tarde. A resposta para esse problema é o comando **git stash**.
Você quer mudar de branch, mas não quer fazer o commit do que você ainda está trabalhando; você irá fazer o stash das modificações. Para fazer um novo stash na sua pilha, execute:

```
git stash
```

Seu diretório de trabalho estará limpo.
Neste momento, você pode facilmente mudar de branch e trabalhar em outra coisa; suas alterações estão armazenadas na sua pilha. Para ver as stashes que você guardou, você pode usar

```
git stash list
```

Você pode aplicar aquele que acabou de fazer o stash com o comando mostrado na saída de ajuda do comando stash original: **git stash apply**. Se você quer aplicar um dos stashes mais antigos, você pode especificá-lo, assim: **git stash apply stash@{2}**. Se você não especificar um stash, Git assume que é o stash mais recente e tenta aplicá-lo.

git tag

Tagging

Git tem a habilidade de criar tags em pontos específicos na história do código como pontos importantes. Geralmente as pessoas usam esta funcionalidade para marcar pontos de release (v1.0, e por aí vai). Nesta seção, você aprenderá como listar as tags disponíveis, como criar novas tags, e quais são os tipos diferentes de tags.
Para listar as **tags** execute:

```
git tag
```

Para criar uma **tag** basta executar o seguinte comando, caso não queira criar a **tag** anotada, somente retire os parâmetros **-a** e **-m**.

```
git tag -a v1.4 -m 'my version 1.4'
```

Compartilhar e Atualizar Projetos

git fetch

Fazendo o Fetch

Para pegar dados dos seus projetos remotos, você pode executar:

```
git fetch origin
```

Esse comando vai até o projeto remoto e pega todos os dados que você ainda não tem. Depois de fazer isso, você deve ter referências para todos os branches desse remoto, onde você pode fazer o merge ou inspecionar a qualquer momento.

git pull

Atualizando local

Incorpora as alterações de um repositório remoto no branch atual. Em seu modo padrão, **git pull** é uma abreviação para **git fetch** seguido de git merge **FETCH_HEAD**. Por exemplo, se eu estiver em uma branch chamada **develop** e quiser atualizar caso haja atualizações remotamente:

```
git pull origin develop
```

git push

Empurrando seus commits

O git push é o comando em que você transfere commits a partir do seu repositório local para um repositório remoto. É a contrapartida do **git fetch**, que busca importações e comprometem as agências locais, utilizando o **git push** as exportações comprometem as filiais remotas. Para fazer isso, você executa **git push [nome_do_repositório_remoto] [nome_da_sua_branch_local]**, que vai tentar fazer que o **[nome_do_repositório_remoto]** receba a sua branch **[nome_da_sua_branch_local]** contendo todos seus commits com alterações. Por exemplo:

```
git push origin develop
```

git remote

git submodule

Inspeção e Comparação

git show

git log

git diff

git shortlog

git describe

Pacotes

git apply

git cherry-pick

git diff

git rebase

git revert

Debugando

git bisect

git blame

git grep

Email

git am

git apply

git format-patch

git send-email

git request-pull

Sistemas Externos

git svn

git fast-import

Administração

git clean

git gc

git fsck

git reflog

git filter-branch

git instaweb

git archive

Servidor

git daemon

git update-server-info

Canalização de Código

git cat-file

git commit-tree

git count-objects

git diff-index

git for-each-ref

git hash-object

git merge-base

git read-tree

git rev-list

git rev-parse

git show-ref

git symbolic-ref

git update-index

git update-ref

git verify-pack

git write-tree
