TUTORIAL PRÁTICO SOBRE Git

por Djalma Oliveira <djalmaoliveira@gmail.com>

Versão 1.1

"Git é um sistema de controle de revisão distribuida, rápido e escalável" (tradução rápida do manual).

Basicamente é um sistema de controle de versionamento de arquivos, muito usado por desenvolvedores para gerenciar versões do software produzidos seja por um desenvolvedor como também por outros participantes do projeto.

Com ele podemos integrar novas funcionalidades efetuadas por outros desenvolvedores, completa, parcial ou não, e tudo isso sendo registrado em um histórico que permite "voltar no tempo" para descobrir por exemplo como funcionava uma determinada versão daquele projeto. Os participantes do projeto podem enviar suas versões, correções, patchs para o projeto principal sem que o projeto principal seja comprometido, permitindo ao dono do projeto a escolha, teste e inclusão dessas alterações no projeto principal se desejar.

O Git foi desenvolvido inicialmente por Linus Torvalds mediante uma necessidade de ter um software robusto para controle de versão do kernel linux.

Outras informações sobre o assunto pode ser acessado através desses links:

Projeto : *http://git.or.cz/*

Vídeo do Linus falando sobre o Git: *http://www.youtube.com/watch?v=4XpnKHJAok8*

Manual do usuário: *http://www.kernel.org/pub/software/scm/git/docs/user-manual.html*

Aqui iniciaremos um rápido e prático tutorial sobre o Git, nada muito profundo, mas bastante útil para aqueles que queiram entender um pouco sobre Git.

TERMOS ANTES DE COMEÇAR

Repositório: Local onde fica armazenado os arquivos do projeto, versões e históricos, local onde os desenvolvedores podem submeter as alterações para o projeto.

Commit: Conjunto de alterações realizadas durante o desenvolvimento, normalmente essas alterações são implementações específicas, seja uma correção, nova versão entre outros. Veja um commit como um ponto histórico no desenvolvimento do projeto, esses pontos podem ser recuperados quando necessário.

INICIANDO UM REPOSITÓRIO LOCAL

Criaremos uma pasta para nosso repositório:

```
mkdir meuprojeto
cd meuprojeto
```

Dentro da pasta iniciaremos um repositório Git:

```
git init
```

Perceba que todos os arquivos dentro da pasta(meuprojeto) fazem parte do repositório, ou seja o seu projeto.

Configuramos agora as informações sobre o desenvolvedor(você), usado para identificar o desenvolvedor em cada commit realizado:

```
git config user.name "Djalma Oliveira"
git config user.email "djalmaoliveira@gmail.com"
```

Perceba que foi criado uma pasta '.git', aonde todas as informações e históricos dos arquivos ficam armazenados e onde acontece a 'mágica'.

Criamos dois arquivos(vazios) para o projeto.

```
touch arq1.txt
touch arq2.txt
```

Agora informamos ao Git que todos os arquivos devem ser incluidos ao próximo commit:

```
git add .
```

Mas se desejar também pode adicionar somente arquivos específicos, basta informar o seu caminho:

```
git add arq2.txt
```

O Git é inteligente o suficiente para detectar se houve alguma alteração nos arquivos que foram indicados para o commit(git add), assim ele não realiza commits sem que pelo menos um arquivo tenha sido alterado, exceto o primeiro commit.

Agora realizamos o commit:

```
git commit –a –m "Meu primeiro commit"
```

* o parâmetro -a informa que deve adicionar todos os arquivos alterados ao commit;
* o parâmetro -m "texto" adiciona uma mensagem informativa para o commit;

Resumindo, para realizar commits seguimos 3 passos:

1 - Efetuar modificações em algum arquivo;

2 - Informar ao Git quais arquivos devem ser adicionados ao próximo commit(git add);

3 - Efetuar o commit(git commit) propriamente dito baseado nos arquivo do passo 2;

No Git todo commit, exceto o primeiro, é descendente(filho) de um outro commit, então você pode verificar diferenças entre commits caso queira.

Após eleger(git add) quais arquivos irão para o próximo commit, você pode ver quais alterações foram realizadas desde o último commit, usando:

```
git diff --cached
```

Essas informações sobre alterações desde o último commit ficam armazenadas em uma estrutura do Git chamada de 'index', ou seja, todas as vezes que você adicionar(git add) arquivos para o próximo commit essas diferenças ficam armazenadas nessa estrutura até o momento em que de fato seja realizado o commit(git commit).

Listando commits realizados:

```
git log
```

Mostrando os arquivos alterados desde o último commit:

```
git status
```

Mostra as alterações realizadas no último commit:

```
git show
```

BRANCHS

Uma branch é uma linha de desenvolvimento do projeto. Você pode ter vários branchs em seu repositório, cada branch representando uma versão específica de seu projeto, por exemplo. Veja um branch como um fork('cópia') do projeto que pode seguir sua própria linha de desenvolvimento.

Todo novo repositório Git (após o primeiro commit) possue um branch chamado por padrão '*master*'.

Por exemplo podemos reprensentar nossos branchs da seguinte forma:

master => projeto principal e a última versão em produção.
working => branch no qual você está trabalhando atualmente;
versao-1.0 => uma das várias versões disponíveis de seu projeto.
versao-1.1 => outra versão...
teste => alguma versão para testes

MANIPULAÇÃO DE BRANCHS LOCAIS

Listando os branchs locais:

```
git branch
```

```
* master
  teste
  working
```

Perceba que o branch que possui um '*' na frente representa o branch corrente.

Criamos um novo branch '*working*' baseado no branch corrente, mantendo-se no branch atual(*master*):

```
git branch working
```

Criando um novo branch 'teste' baseado em um outro branch '*working*' que não é o corrente,ainda mantendo-se no branch atual:

```
git branch teste working
```

Outra forma de criar um branch e ao mesmo tempo tornar o novo como corrente é usando:

```
git checkout –b teste working
```

Até este ponto devemos ter as seguintes branchs:

```
* master
  teste
  working
```

Mudando para o branch working:

```
git checkout working
```

Agora ficará assim:

```
  master
  teste
* working
```

Apagando um branch:

```
git branch –D teste
```

Lembrete: você não pode apagar o branch corrente, alterne para um outro então execute novamente o comando.

MAIS SOBRE OS BRANCHS

Até aqui você já deve ter percebido que não foi necessário sair da pasta do projeto, na realidade a cada mudança de branch (git checkout) os arquivos contidos na pasta do projeto são modificados para refletir exatamente as versões de cada branch, sendo assim o seu projeto sempre vai estar localizado naquela pasta, pois o Git se encarrega de gerenciar as informações relativos a cada branch corrente.

INCORPORANDO(merge) NOVAS MODIFICAÇÕES AO PROJETO

Uma das funcionalidades mais interessantes do Git é a capacidade de incorporar as alterações realizada por programadores ao projeto. Essa 'junção' se dá somente entre branchs e é chamada de merge.

Por exemplo, digamos que o arquivo arq2.txt no branch *working* foi alterado:

```
git checkout working    # mudando para o branch working
echo "opa" > arq2.txt   # alteração propriamente dita
git add .               # marca os arquivos para o próximo commit
git commit -a -m "alterando arq2.txt" # commit
```

Agora a idéia é adicionar as alterações realizadas no branch *working* para o branch *master*(lembre-se que o branch *working* é um fork do *master*!?).

Mudamos para o branch que receberá as alterações:

```
git checkout master
```

Realizamos agora o merge das alterações do branch *working* para o corrente(*master*):

```
git merge working
```

Após o comando acima será mostrado um resumo do que foi realizado:

```
Updating 751bee8..625fa43

Fast forward
 arq2.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Confirmando a alteração no arquivo:

```
cat arq2.txt
```

Peceba que neste momento toda e qualquer alteração realizada no último commit do branch *working* agora está refletida do branch *master*, inclusive o commit realizado na branch working, que agora está na branch *master*.

Verifique o último commit:

```
git log
```

COMPLICANDO MAIS

Perceba que antes de efetuarmos o merge, somente a branch *working* possuia alterações, a *master* não, mas e se a branch *master* fosse alterada antes de dar-mos um merge a branch *working*?

Essa situação é bastante comum em projetos grandes que alterações são adicinadas frequentemente ao projeto principal, não refletindo mais a versão atual do *working*.

"Lembrando que todo *branch* possui um *branch* pai que é a sua *base* a partir de então."

Nessa situação a melhor forma de realizar o *merge* seria realizar um 'rebase', ou seja, pegar o último commit do *master,* que é a base do working (por isso o nome rebase), trazer para o *working* e aplicar todos os seus commits(na mesma ordem de criação) nele, agora sim a sua versão do *working* estará sincronizada com a última versão do *master* mais os seus commits.

Realizar um merge nessa situação pode até funcionar(o Git é inteligente em algumas situações), mas não é recomendado pois isso poderia causar conflitos que seriam mais trabalhosos de resolver do que se efetuasse um 'rebase'.

Então antes de aplicar-mos um merge ao *master*, faremos um rebase a branch *working* antes para refletir a última versão do *master*:

```
git checkout working
git rebase master
```

Agora podemos aplicar um merge ao *master*:

```
git checkout master
git merge working
```

REPOSITÓRIOS REMOTOS

A idéia é a mesma de um repositório local, mas os remotos são hospedados em servidores na internet ou outra máquina que não a sua (mas pode ser a sua também).
Para trabalharmos com repositórios remotos usamos alguns comandos extras além dos já estudados.

Para criar um clone (cópia de todo o projeto, incluindo todos commits) do projeto podemos usar um dos seguintes comandos (dependendo a configuração do servidor):

```
git clone git://sitehospedado.com.br/projeto
                        ou
git clone http://sitehospedado.com.br/projeto
```

Logo após, você verá que será criado um diretório com o nome do projeto, e dentro a cópia(clone) de todo o projeto.

Veja o repositório remotos disponível agora:

```
cd projeto
git branch -r

origin/HEAD
origin/master
origin/working
```

Perceba que a palavra origin é o nome padrão (mas você pode criar um com nome diferente desse se quiser) para repositórios remotos, representando os branchs que existem no projeto, mas que são de origem remotas para você agora.
Veja que você deve criar um branch local baseado em algum branch remoto antes de começar a efetuar suas alterações.

```
git checkout -b working origin/working
```

Agora digamos que você efetuou várias alterações neste repositório e durante esse tempo o projeto principal também foi alterado não correspondendo mais a base que você possui agora e então deseja sincronizar com a última versão disponível do projeto.

Primeiramente recuperamos a versão recente do projeto:

```
git fetch
```

Agora efetua o merge com o branch atual:

```
git merge origin/master
```

Uma outra forma de realizar poderia ser assim:

```
git pull origin master
```

Ou até mesmo:

```
git pull
```

Neste último caso considerando que você tem um branch master ele fará o merge automaticamente.

SINCRONIZANDO SUAS ALTERAÇÕES COM O REPOSITÓRIO REMOTO

Após efetuar suas modificações no branch, você pode enviá-lo para o servidor remoto através deste simples comando:

```
git push
```

INFORMAÇÕES FINAIS

Bom, finalizo por aqui mas existem ainda muitos comandos sobre o Git que podem ser abordados em versões futuras deste manual, então aguardem atualizações.

Espero ter abordado de forma clara, algumas características do Git e comentem se encontrarem algum erro, pois algumas partes foram traduzidos(minha versão) do manual do usuário.