Curso de Introdução ao Git

USPGameDev

4 de novembro de 2011

1 Controle de Versão

Um controle de versão é um sistema responsável por gravar as mudanças em um ou mais arquivos. Temos dois tipos de software de controle de versão, os que usão o para digma centralizado e o distribuído. O SVN e o CVS são exemplos de sistemas centralizados. Eles guardão todos os dados de um projeto em um servidor central, online. O git e o mercurial são exemplos de sistemas distribuidos. Eles, além de usarem repositórios online, criam um repositório local no seu computador.

2 O que é o git

Enquanto a maioria dos SCV guardam as informações como um arquivo base e as suas modificações, o git guarda uma "foto"das suas modificações. Toda vez que você modifica ou salva algo no git, ele tira uma "foto"de todo o seu espaço de tra balho.

A maioria do trabalho do git é local. O git salva todas as modificações dos arquivos localmente, e toda vez que você baixar as atualizações do seu projeto, todo o histórico de modificações vem junto. O git tem integridade. Isso quer dizer que é impossível modificar arquivos sem o

git saber que você está fazendo algo. O git usa um mecanismo chamado SHA-1 hash. O git quase sempre adiciona dados. Na maioria das vezes, o git só adiciona dados no repositório. Assim, você pode sempre resgatar arquivos que já foram deletados a muito tempo.

O git usa uma mecânica de 3 estagios para controlar os arquivos na sua área de trabalho. Estes estagios sao: unmodified, modified e staged. Commited significa que o arquivo esta na sua database local. Modificado quer dizer que o arquivo foi modificado. Staged quer dizer que o arquivo foi selecionado para entrar no seu próximo commit.

3 Como iniciar um rep no git

O github tem um bom tutorial para instalar e iniciar o git:

- Windows (Sem o tortoise git. Há um tutorial breve sobre o turtoise git abaixo.)
- Linux
- MacOS

4 Iniciar o git

A primeira coisa a se fazer depois de instalar o git e colocar o seu nome de usuario e e-mail, assim todo commit que voce fizer estara com essas informacoes. Para isso, use os comandos:

```
$git config --global user.name "Seu Nome"
$git config --global user.email seuemail@exemplo.comando
```

Para configurar a sua ferramenta de diff, use o comando:

```
$git config --global merge.tool NomeDaFerramenta
```

Obs: O git aceita kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge e opendiff como ferramentas de merge.

5 Help no git

Para acessar o manual do git (e de seus comandos) voce pode usar qualquer uma das sintaxes abaixo:

```
$git help <comando>
$git <comando> --help
$man git-<comando>
```

6 Como usar o git

Agora que o git está instalado e o repositório já está criado, podemos começar a modificar nosso projeto. O git usa um mecânismo para controlar as modificações nos seus arquivos, e usa um sistema um pouco diferente dos outros controles de versões para mandar essas modificações para o remoto.

6.1 Iniciando um repositório local git

Para iniciar um rep. local em um diretório usamos o comando

```
$ git init
```

ou

```
$ git clone <url>
```

O `git init` inicia um repositório sem qualquer ligação com algum remoto, e o `git clone` pega tudo de um remoto (a `<url>`) copia para um novo repositório local e linka esse local com o remoto de origem.

6.2 Arquivos no Git

Todo arquivo, antes de ser adicionado no seu repositório, é reconhecido no git como **Untracked**. Ao ser adicionado usando o comando:

```
$ git add <Nome-do-Arquivo>
```

ele vira **Tracked** entra num ciclo de 3 fases do git.

Esse ciclo consiste nas fases **modified**, **staged** e **unmodified**. Quando um arquivo é adicionado no git, ele entra como **staged** e será adicionado ao remoto no próximo commit. Ao ser modificado, um arquivo vai para **modified**. Ao ser comitado, o arquivo se torna **unmodified**.

Para verificar quais arquivos estão em qual fase usamos o comando

```
$ git status
```

ele ainda mostra algums comandos para resetar arquivos e outras dicas, falaremos mais sobre isso depois.

Esse arquivo vai aparecer nos dois locais. Na realidade, o arquivo que aparece Um arquivo pode ser **staged** e **modified** ao mesmo tempo. Esperimente dar um git add em um arquivo, modificá-lo e dar um git status. na seção **modified** é o arquivo modificado, e o arquivo que está na **staged** é o arquivo sem modifcação.

Para verificar as diferenças entre os arquivos que estão modificados e os suas respectivas cópias no local usamos o comando

```
$ git diff
```

6.3 Commits

Os commits no git, ao contrário dos controles de versão ao estilo SVN, é local. Isso quer dizer que tudo o que é commitado não vai diretamente para o repositório remoto, e sim para o seu repositório local. Os arquivos na área **staged** serão adicionados no próximo commit. Um commit contêm a célula do commit, as informações de quem commitou e uma mensagem sobre o commit. Para commitar suas mudanças, usamos o comando

```
$ git commit
```

ao executar esse comando, um editor de texto vai abrir, e a mensagem do commit deve ser escrita.

O git commit aceita duas opções muito usadas

- `$ git commit -a` ele commita tudo que está na área **modified**

- `$ git commit -m "string"` ele gera um commit com mensagem igual a string passada. Lembre que a string deve ser passada entre áspas.

6.3.1 Como os commits são armazenados

Os commits são armazenados como uma lista ligada, onde cada célula da lista é um arquivo de 41 bits (um HASH para identificar cada commit de maneira única), um ponteiro para um estrutura que aponta para todos os arquivos daquele commit e um ponteiro para o commit pai.

O commit atual é marcado por um apontador chamado HEAD, que é movido toda vez que fazemos um commit. Está estrutura faz com que commitar e andar nos commit seja muito leve, pois só mudamos o valor de um ponteiro, e para carregar os arquivo usamos um arquivo de 41 bits.

6.3.2 Boas práticas de commit

Quando se trabalha com várias pessoas num projeto, é bom seguir algumas práticas pra deixar o trabalho de todos mais fácil:

- Teste o seu código antes de dar git push. Sempre deixe o código que está no repositório funcional.
- Escreva mensagens de commit claras, pequenas e informativas. Elas devem explicar o que você fez nesse commit.
- Faça vários commit pequenos. Assim, você não adiciona muita informação ao mesmo tempo dentro do repositório, facilitando a volta de commits.
- Quebre commits. Ao ínvez de adicionar todas as suas mudanças direto em um grande commit, adicione-ás de uma maneira inteligente, se você modificou dois arquivos muito diferentes, use um commit para cada um.

6.3.3 Mandando commits para o repositório remoto

Os seus commits só são mandados para o repositório remoto quando usamos o comando

```
$ git push
```

Esse comando vai copiar a sua lista ligada à lista ligada de commits do repositório remoto. Ele vai procurar o commit em comum mais antigo entre seu rep. e o remoto e tentar dar cópiar seus commits para o remoto. Se o commit mais antigo não for o último do remoto, ou seja, alguem fez algum push antes do seu, você deve usar o comando

```
$ git pull
```

para pegar as modificações do remoto e aplicar no local, para só depois você poder dar push.

6.4 Retornando a estados anteriores

A grande vantagem de se usar controle de versão é a possibilidade de trazer arquivos de volta para versões antigas deles. No git, podemos voltar arquivos separadamente ou voltar para commits especificos.

6.4.1 Voltando arquivos

Para retornar arquivos para a última versão gravada no git usamos o comando

```
$ git checkout -- <nome-do-arquivo>
```

mas tome cuidado com esse comando, pois ele sobrescreve o seu arquivo, deletando as suas modificações.

Para retornar para uma versão especifica de um arquivo, em um branch especifico, usamos o comando

```
$ git checkout <nome-da-branch>~n <nome-do-arquivo>
```
(n é a diferença do número do commit atual e o alvo).

Um exemplo:

Imagine que você quer voltar o arquivo *pessoa.c* para o estado dele 3 commits atrás. Você usa o comando `$ git checkout master~3 pessoa.c`.

6.4.2 Voltando para commits

O comando para se mover na lista ligada de commits é bem parecido com o acima

```
$ git checkout <nome-da-branch>~n
```
(n é a diferença do número do commit atual e o alvo).

A diferença é que como não especificamos um arquivo, voltamos todo o projeto para o seu estado passado.

6.5 Comunicação com o remoto

Como vimos, usamos os seguintes comandos para mandar/receber dados do remoto:

- `$ git pull ->` Recebe as últimas modificações do remoto.
- `$ git push ->` Manda seus commits para o remoto.
- `$ git clone <url> ->` Clona um remoto para dentro da pasta local.
- `$ git remote show ->` Mostra informações sobre o remoto.
- `$ git remote <nome-atual> <novo-nome> ->`Renomeia um remoto.

7 Branches

7.1 O que são branches

Usamos uma branch num controle de versão quando queremos separar o fluxo de desenvolvimento de um projeto. Imagine que temos dois fluxos no projeto, é interesante separarmos eles para que nossa branch principal (normalmente a master) não fique poluída com commits que não acresentão nada diretamente ao projeto.

7.2 Branches no git

Branches sao facilmente criados no git. Eles não passam de outro apontador para commits, tipo o HEAD. O objetivo das branches é criar ambientes de trabalho com o menor contato possível com a branch master (a princípal do projeto) assim não "estragamos"o projeto com features que não estão prontas ainda. Todo trabalho que fazemos numa branch fica somente nela até que juntamos elas com outra branch.

7.3 Criando, deletando e usando Branches

O comando para criar branches é

```
$ git checkout -d <nome-da-branch> ou
$ git branch <nome-da-branch>
```

A diferença é que no primeiro caso a branch é criada e o git vai pra ela. No segundo ela só é criada.

Para se movimentar pelas branches usamos o comando

```
$ git checkout <nome-da-branch>
```

que vai para a branch passada para ele.

Para deletar branches, você precisa sair delas e usar o comando

```
$ git branch -d <nome-da-branch>
```

7.3.1 Merge

O merge é juntar dois arquivos, ou nesse caso, duas branches. Para juntar as branches, use o comando checkout para ir até a branch na qual você quer que o resultado fique e use o comando

```
$ git merge <nome-da-branch>
```

ele vai tentar juntar os arquivos da branch alvo com os da branch atual.

7.3.2 Branches remotas

Vale lembrar que um branch criado por você é local, e ele só é adicionado no repositorio local se usarmos o push desta maneira:

```
$ git push <remote> <branch>
```

Exemplo:

```
$ git push origin teste
```

isso vai fazer o branch teste ser integrado no nosso repositório remoto origin.

7.4 Algumas boas práticas de branches no git

Quando criamos uma branch num projeto onde várias pessoas estão trabalhando, é bom seguir algumas práticas:

- Sempre crie branches com nomes que falem sobre o que o branch é. Se você está fez um branch para arrumar algo sobre e-mail, um bom nome seria e-mailfix.
- Criar uma branch toda vez que você for implementar algo no sistema. Isso impede que o projeto seja desenvolvido direto na master.
- Só de merge com qualquer outra branch quando a feature do branch estiver pronta e testada. Isso impede que pessoas usem coisas da sua branch quando ela ainda não está pronta.

8 Resumo dos comandos no linux

```
$ git clone -> Copia um repositorio remoto para a atual localização.
$ git add <Arquivo> -> Adiciona o arquivo para ser commitado.
$ git commit -> Comita as atuais modificações para o seu repositorio local.
$ git push -> Mandas os atuais commits para o repositorio remoto.
$ git pull -> Puxa os commits do repositorio remoto.
$ git rm -> Remove um arquivo.
$ git mv -> Move um arquivo.
$ git diff -> Diferença entre os arquivos que serão comitados e suas modificações.
$ git status -> Mostra o status de cada arquivo.
$ git branch <Nome-do-Branch> -> Cria um branch com o nome passado.
$ git checkout <Nome-do3-Branch> -> Vai para o branch com o nome passado.
```

9 Extras

9.1 Tags

Assim como várias outras ferramentas de controle de versão, o git permite a criação de tags para identificar commits específicos. A utilização de tags permite que o histórico de commits seja mais limpo e a marcação de pontos importantes no projeto, como releases.

9.1.1 Tags no git

Para listar as tags no git, use o comando

```
$ git tag
```

O git tem dois tipos de tags, as anotadas (annotated) e as leves (lightweight). As leves são como branches fixas, elas só apontam para um commit. As anotadas são mais completas. Elas tem um nome, e-mail, data de criação e uma mensagem. Para criar tags usamos o comando:

`$ git tag -a <nome-da-tag> -m "mensagem" <hash-do-commit>` ou
`$ git tag <nome-da-tag>`.

O primeiro comando cria uma tag anotada e o segundo cria uma leve. O hash do commit não é necessário para criar uma tag no commit atual.

9.1.2 Tags no remoto

O git não adiciona automâticamente as tags no servidor remoto, precisamos colocá-las na mão no remoto. Para fazer isso, usamos:

`$ git push origin <nome-da-tag>` ou
`$ git push origin --tags` para mandar todas as tags.

9.2 Alias no git

O git permite a criação de alias. Usamos o comando *git config* para isso:

```
$ git config --global alias.co checkout
$ git config --global alias.st status
$ git config --global alias.unstage 'reset HEAD --'
```

O último exemplo faz com que o comando:

```
$ git unstage pessoa.c
```

fique igual ao

```
$ git reset --HEAD pessoa.c
```

9.3 Stashing

O git não deixa você trocar de branch se houver trabalho não commitado na branch atual, mas nem sempre queremos commitar o trabalho atual, seja por que ele está meio feito ou por que não é algo relevante. Para esses casos, existe o comando `$ git stash`. Esse comando pega todas as suas modificações e adiciona numa pilha de "stashes"limpando a branch atual.

9.3.1 Aplicando Stash

Podemos aplicar os arquivos de uma stash usando o comando:

`$ git stash apply` ou
`$ git stash apply stash@{n}` para aplicar o n-ésimo stash da pilha.

A stash vai ser aplicada na branch atual e vai funcionar como um "pull". O git vai avisar se algum problema de merge ocorrer.

9.3.2 Criando branches de stashs

Para criar uma branch a partir de um stash, usamos o comando:

```
$ git stash branch <nome-da-branch>
```

10 Git no Windows usando o tortoise git

Primeiro precisamos instalar os seguintes arquivos:

Tortoise Git	http://code.google.com/p/tortoisegit/
msysgit	http://code.google.com/p/msysgit/
PuTTY	http://www.chiark.greenend.org.uk/ sgtatham/putty/download.html

Baixe as versões mais novas, e instale tudo normalmente, sem mudar nenhuma opcao, a nao ser que seja de sua preferencia. Agora, precisamos configurar uma chave de ssh para que se computador possa con versar com o repositorio remoto. Va ate a pasta em que o PuTTY foi instalado e abra o executavel puttygen. Clique em Generate, depois digite uma senha no campo Key passphrase (essa senha sera pedida toda vez que voce fizer um push ou um pull. Nao a esqueca!). Clique em save private key, e a salve num lugar onde voce possa achar facilmente. Abra o site do github e na secao Account settings, va em SSH Public Keys e adicione a chave de ssh que voce acabou de gerar. Quando voce for clonar um repositorio, o tortoise vai pedir o seu nome e seu email. Isso e para que cada commit foito por voce tenha as suas informacoes, assim o grupo pode saber quem fez o que. Depois, voce tera que fornecer o link da chave de ssh para o tortoise, sem isso ele nao pode clonar um repositorio.