Alterado por: Diogo de Jesus Pina - diogojpina at gmail

Mini-curso de Git

The stupid content tracker

Slides Cedidos pelo Thiago Kenji Okada

O que vamos ver?

- O que são VCSs?
- Histórico
- Por que aprender Git?
- Como o Git funciona?
- Instalação
- Configuração básica
- Uso básico do Git
- Branchs
- Tags

Version Control Systems

- Sistema que guarda as mudanças feitas num arquivo ou num grupo de arquivos no decorrer do tempo…
- ...assim você pode ir para uma versão arbitrária do arquivo…
- ...e caso algo dê errado, você pode voltar e usar uma versão anterior.

Tipos de VCSs

- *Local Version Control Systems (LVCS)*
  - Como o próprio nome diz, é local.
  - Não é possível colaborar com outros desenvolvedores.
  - rcs foi um dos VCS locais mais populares

- *Centralized Version Control Systems (CVCS)*
  - CVS, Subversion, Perforce.
  - Um servidor mantém todos os arquivos.
  - Ponto único de falha (e se o servidor cair?).

- *Distributed Version Control Systems (DVCS)*
  - Git, Mercurial, Bazaar, Darcs.
  - Cada cliente tem uma cópia de todo repositório.

Breve histórico do Git (1)

- Criado pelo Linus Torvalds, o mesmo criador (e principal mantenedor) do kernel Linux.
- Até 2002, o kernel era mantido por troca de arquivos patch e TARs por e-mail.
- Em 2002 o projeto Linux começou a usar o BitKeeper, um DVCS proprietário.
- Em 2005, a empresa que desenvolve o BitKeeper parou de apoiar projetos *open source*.

Breve histórico do Git (2)

- Isso levou a comunidade do kernel Linux a desenvolver sua própria ferramenta a partir das lições aprendidas com o BitKeeper.
- Objetivos:
  - Ser rápido
  - Design simples
  - Bom suporte para desenvolvimento não-linear
  - Completamente distribuído
  - Habilidade de gerenciar projetos grandes, como o Linux, de forma eficiente.

Por que aprender Git?

- De acordo com uma pesquisa feita pela Fundação Eclipse em Maio de 2014, 42,9% dos desenvolvedores profissionais usam Git ou GitHub como VCS principal, frente aos 36% em 2012, 27,6% em 2011 e 12,8% em 2010.
- Na Inglaterra, o site UK IT aproximadamente 20,32% das oportunidades de emprego para desenvolvedores exigem conhecimento em Git.

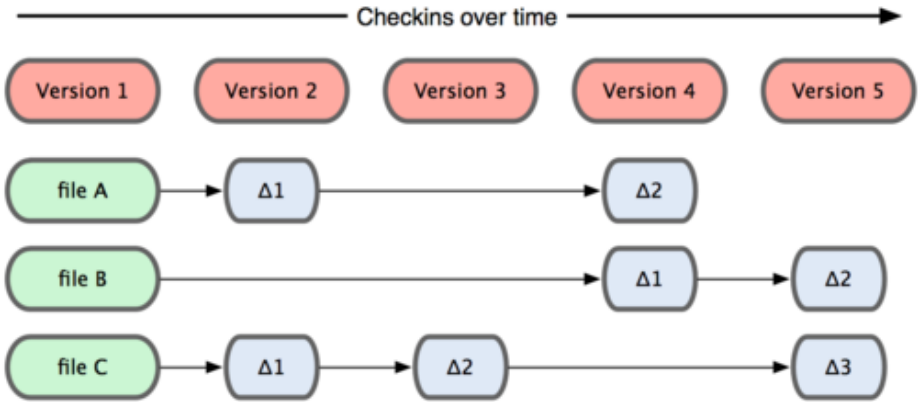
Fonte: http://en.wikipedia.org/wiki/Git_(software)#Adoption

Como o Git funciona?

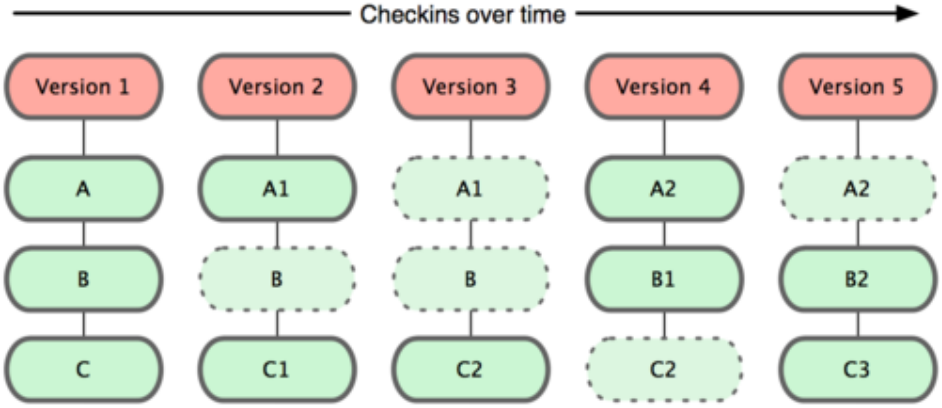
- Ao invés de armazenar as diferenças (*diffs*), o Git armazena uma "cópia" dos dados atuais (*snapshots*).
  - Graças a isso, o Git parece mais um sistema de arquivos que um VCS.
- Quase toda a operação é local.
  - Isso aumenta o desempenho e permite trabalhar *offline*.
- Checagem de integridade (SHA-1).
  - Praticamente impossível fazer uma alteração sem que o Git fique sabendo.
- Git, geralmente, só adiciona dados.

Snapshots, não *diffs*

VCS tradicional

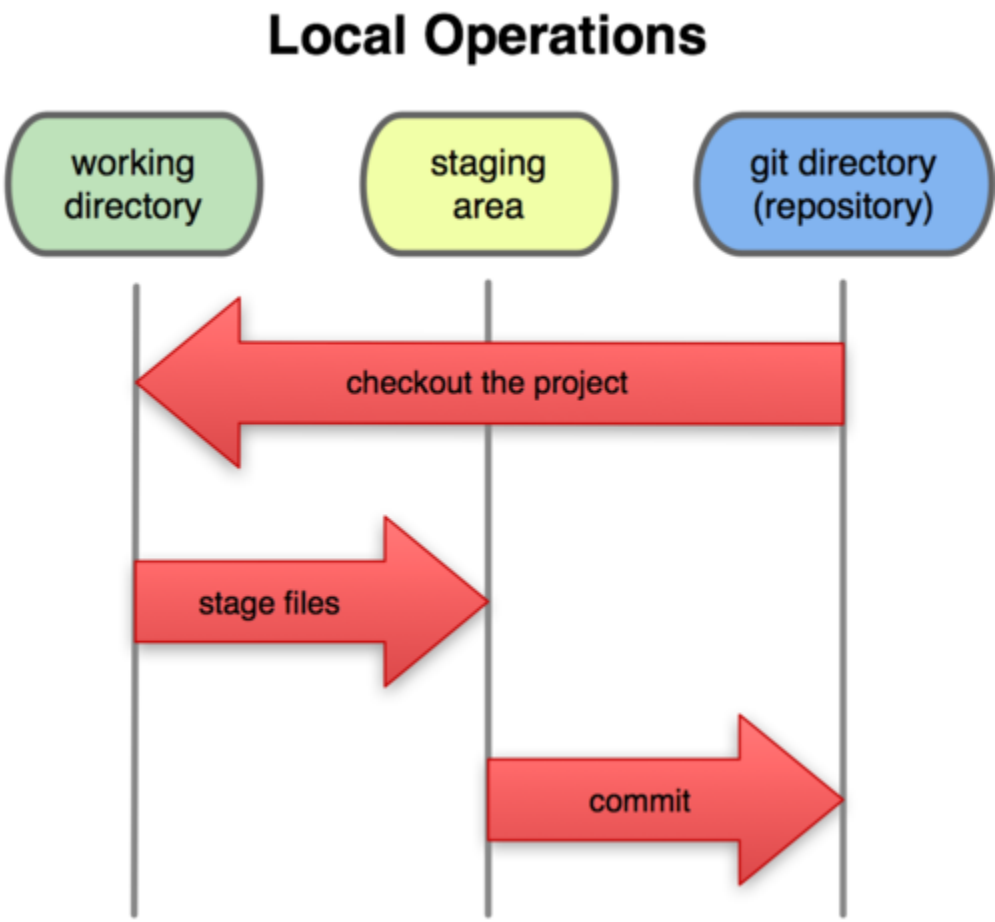


Git



Os três estados

- Um arquivo pode estar em três estados:
  - *unmodified / commited*
  - *modified*
  - *staged*



Instalando o Git (1)

Instalando dependências:

- Fedora

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

- Ubuntu/Debian

```
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Instalando o Git (2)

Faça o download do código-fonte no site:

http://git-scm.com/download

E então compile

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Depois disso, você pode usar o próprio Git para updates:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Instalando o Git (3)

- Você também pode baixar o Git no repositório da sua distro (nem sempre tem a versão mais atual disponível).
- No Mac OS X você pode usar o instalador disponível nesse site:

  http://code.google.com/p/git-osx-installer
- No Windows você pode usar as versões disponíveis nesse site:

  http://msysgit.github.com/

Configurando o Git (1)

- /etc/gitconfig: Configurações do sistema, afetam todos os usuários. Modificado com git config --system (precisa de root).
- ~/.gitconfig: Configurações globais do usuário atual. Modificado com git config --global.
- .git/config: Configurações do diretório Git correspondente. Modificado com git config --local (dentro do diretório).
- Cada nível tem prioridade sobre o anterior, ou seja, configurações locais tem prioridade sobre as configurações do sistema.

Configurando o Git (2)

- Vamos configurar o nome e e-mail, para que o Git mantenha nosso histórico de modificações:

```
$ git config --global user.name "Seu Nome"
$ git config --global user.email seu@email.com
```

- Depois você pode setar algumas preferências, como:

```
$ git config --global core.editor vim
$ git config --global merge.tool vimdiff
```

- Podemos checar as configurações com:

```
$ git config --list
```

Conseguindo ajuda

- Se você precisar de ajuda durante o uso do Git, pode usar as seguintes comandos:

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

- Diferente de outros programas *nix, o Git não tem uma ajuda rápida. Qualquer um dos comandos acima abre uma manpage.

Mãos a obra

- Para usar o Git num projeto já existente, vá até o diretório correspondente e digite:

  ```
  $ git init
  ```

- Isso cria um novo sub-diretório chamado .git que contém todos os arquivos necessários para um repositório - um esqueleto.
- Nenhum arquivo está sendo monitorado ainda.

Adicionando arquivos e o primeiro commit

```
$ git add -A
$ git commit -m 'versão inicial'
```

- Com isso criamos um repositório local e mesmo sem a ajuda de um servidor externo podemos manter o histórico dos nossos arquivos. Ou seja, é possível usar o Git como LVCS também.

Clonando um repositório existente

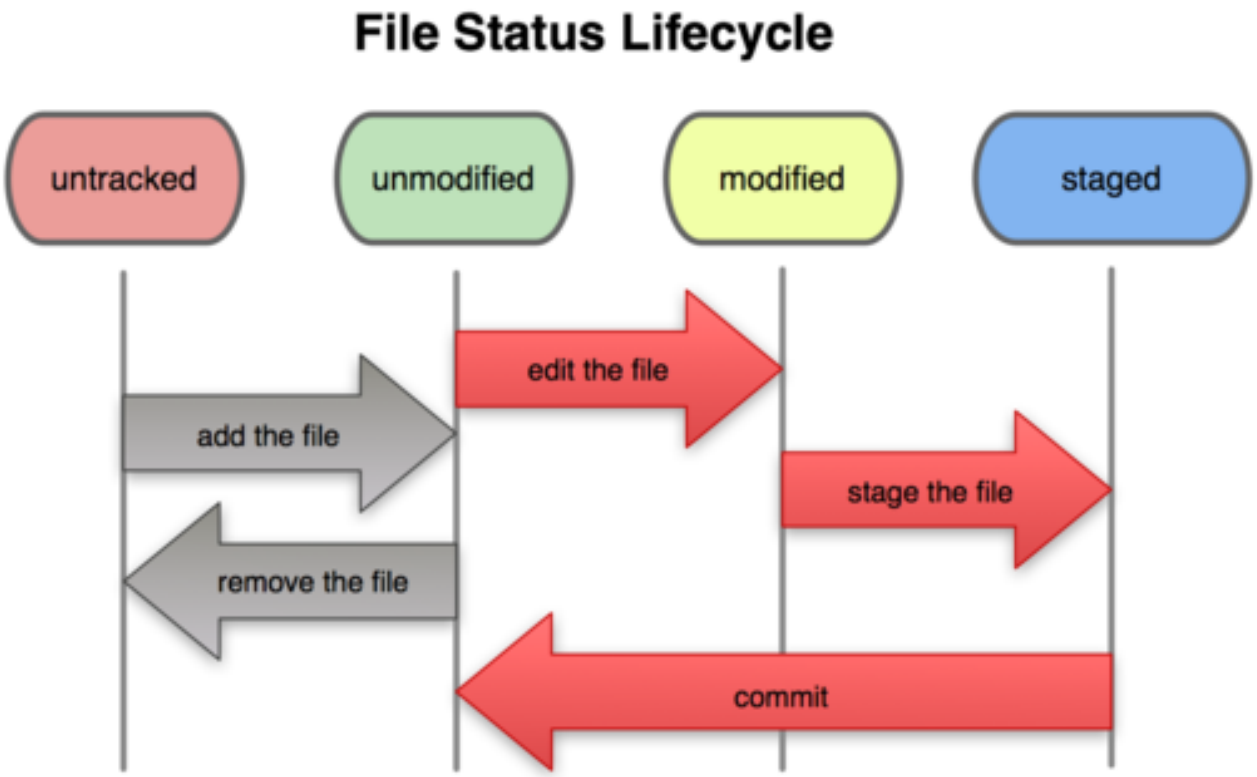
```
$ git clone git://github.com/schacon/grit.git
$ git clone git://github.com/schacon/grit.git mygrit
```

- Reparem que usamos o comando clone ao invés do mais comum checkout. O Git (por ser um DVCS) faz uma cópia completa do repositório ao invés de apenas puxar a última versão.

Fazendo mudanças

- Cada arquivo pode estar em dois estados: *tracked* (monitorado) e *untracked* (não monitorado).
- Arquivos *tracked* são arquivos que estavam no último *snapshot*, eles podem estar em qualquer um dos três estados.
- Arquivos *untracked* são arquivos novos que ainda não estão sendo monitorados.

Ciclo de vida de um arquivo



Checando o estado dos seus arquivos

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

- Como não fizemos nenhuma alteração ainda, o repositório está limpo.

Criando um novo arquivo

```
$ vim README
$ git status
# 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)
```

- Criamos um novo arquivo, ou seja, temos um arquivo *untracked*.

Monitorando um novo arquivo

```
$ git add README
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
```

- Agora nosso arquivo está nos estados *tracked* e *stagged*, ou seja, ele será incluído no próximo *commit*.

Vamos fazer alguns testes

1. Modifique um arquivo existente qualquer. Digite git status e veja o resultado.

2. git add no arquivo modificado e git status. Veja o resultado.

3. Modifique o arquivo mais uma vez e git status. O que aconteceu?

O que aconteceu? (1)

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

O que aconteceu? (2)

- O Git faz um *snapshot* do arquivo na hora que você dá um git add.

- Se você commitar agora, você irá commitar a versão antiga do arquivo, do jeito que ele estava no último git add feito.

- É necessário outro git add para que as alterações sejam feitas.

Ignorando arquivos (1)

- É comum você querer ignorar alguns arquivos no seu diretório de trabalho (arquivos .exe, .o, .a, backups, etc.).

- O arquivo .gitignore diz ao Git quais arquivos (ou quais tipos de arquivo) ignorar.

- A sintaxe é relativamente poderosa, suportando *glob patterns* (expressões regulares do *nix).

Ignorando arquivos (2)

- Linhas em branco ou começando com um # são ignoradas.
- *Glob patterns* funcionam.
- Você pode terminar padrões com um / para indicar um diretório.
- Você pode negar o padrão começando o mesmo com uma !.

Ignorando arquivos (3)

```
$ cat .gitignore
                # comentário, isso é ignorado
*~              # ignora os arquivos de backup do vim/emacs
*.[oa]          # ignora arquivos com extensão "o" ou "a"
!lib.a          # mas não ignora "lib.a", apesar da regra acima
/TODO           # só ignora o arquivo TODO do root, não os
                # subdiretórios
build/          # ignora todos os arquivos do sub-diretório build
doc/*.txt       # ignora docs/nota.txt, mas não
                # docs/servidor/nota.txt
```

Vendo o histórico de commits (1)

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Vendo o histórico de commits (2)

- -p exibe os *diffs* entre commits.
- -2 exibe os dois últimos commits.
- --stat abrevia o histórico, útil para ver múltiplos commits.
- --pretty permite você formatar a saída:
  - --pretty=oneline exibe os commits numa única linha.
  - --pretty=short|full|fuller exibe menos/mais informação.
  - --pretty=format:"opção" permite formatar a saída de diversas formas.
  - --graph exibe a árvore de commits em ASCII ;) .
- gitk exibe os commits numa GUI.

Opções do --pretty=format:

Option	Description of Output
%H	Commit hash
%h	Abbreviated commit hash
%T	Tree hash
%t	Abbreviated tree hash
%P	Parent hashes
%p	Abbreviated parent hashes
%an	Author name
%ae	Author e-mail
%ad	Author date (format respects the –date= option)
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cd	Committer date
%cr	Committer date, relative
%s	Subject

Vendo o histórico de commits (3)

```
$ git log --since=2.weeks
```

- Commits feitos nas duas últimas semanas.

```
$ git log --author=thiagoko
```

- Exibe todos os commits feitos pelo usuário thiagoko.

```
$ git log --grep="pao de batata"
```

- Exibe todos os commits que tem a frase "pao de batata".

Modificando seu último commit

- Alterando só a mensagem do commit:

```
$ git commit --amend
```

- Adicionando um arquivo ao último commit:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Removendo um arquivo staged

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#       modified:   benchmarks.rb
#
```

Desfazendo modificações num arquivo

```
$ git checkout -- benchmarks.rb


$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
```

Repositórios remotos

- Nós clonamos um repositório remoto com git clone.

- Se usarmos git remote, veremos quais repositórios remotos essa pasta git está associada (por enquanto só o origin, que representa o repositório de onde clonamos o projeto).

- git remote -v mostra a URL dos repositórios.

Adicionando repositórios remotos

```
$ git remote
origin

$ git remote add pb git://github.com/paulboone/ticgit.git

$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb  git://github.com/paulboone/ticgit.git
```

Puxando informações do novo repositório

- Um git fetch pb pega toda a informação do repositório pb que você ainda não tem, mas não faz nada (ele abre um novo branch, que veremos mais tarde).
- Um git pull pb faz um fetch e depois um merge, incluindo as modificações feitas no repositório pb no seu repositório local.
- Um git clone faz um pull e ainda seta a URL do origin para o branch master como a URL do repositório original.

Enviando suas modificações ao repositório

```
$ git push origin master
```

- Você precisa de acesso a escrita para poder enviar informações num repositório remoto.

Conseguindo informações do repositório remoto

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

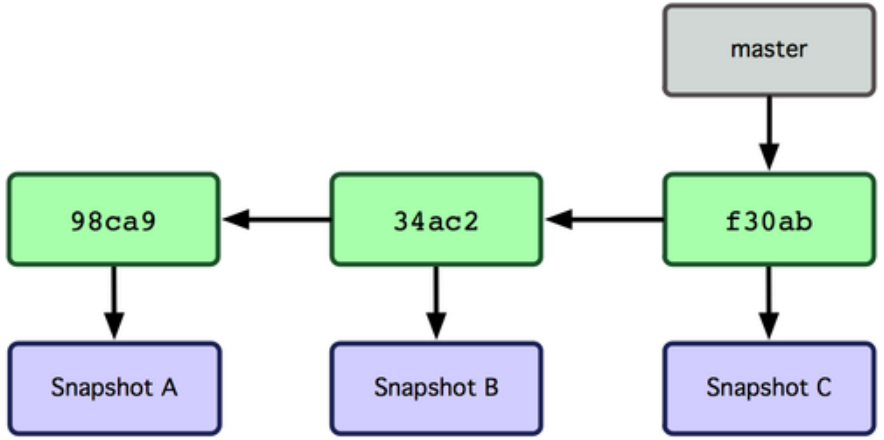
Renomeando e removendo repositórios remotos

```
$ git remote rename pb paul
$ git remote
origin
paul


$ git remote rm paul
$ git remote
origin
```

O que são branchs? (1)

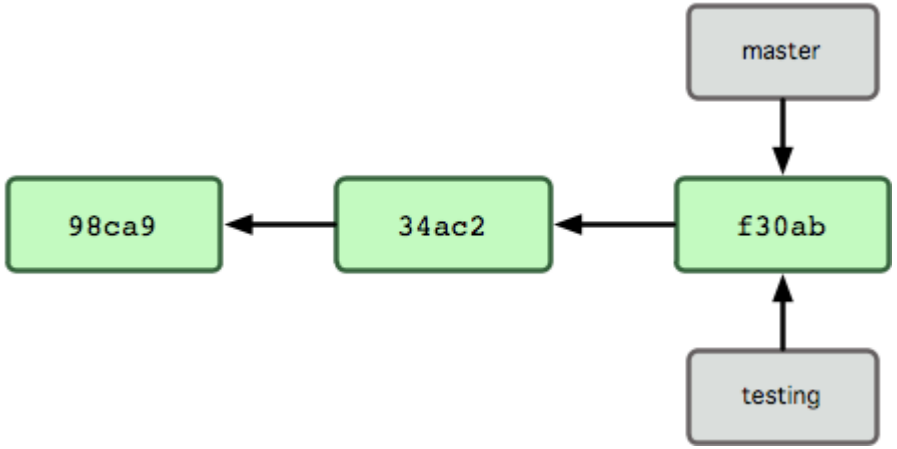
- No Git, cada branch é um ponteiro para um commit.
- A cada commit, o ponteiro avança.



O que são branchs? (2)

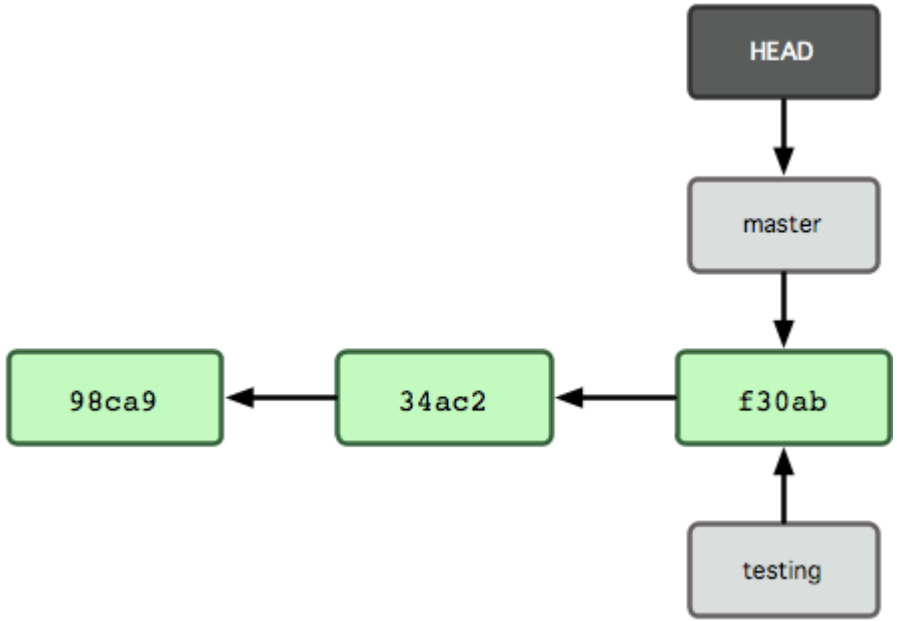
$ git branch testing

- Agora criamos um novo branch, que aponta para a última modificação feita.



O que são branchs? (3)

- Como o Git sabe qual é o branch atual?
- O ponteiro HEAD indica qual o branch que está sendo usado.
- Para mudar o HEAD de lugar, usamos:

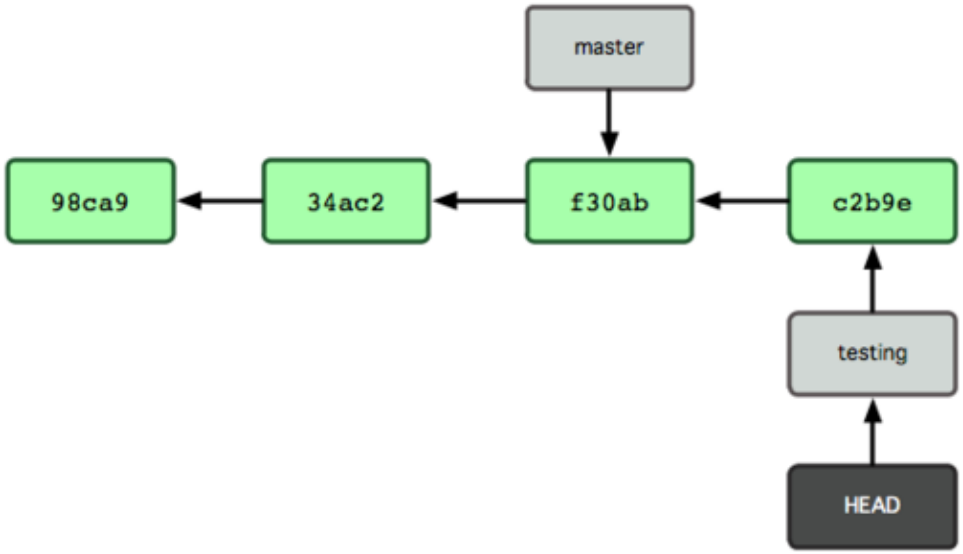


$ git checkout testing

O que são branchs? (4)

- Vamos fazer um commit no novo branch:

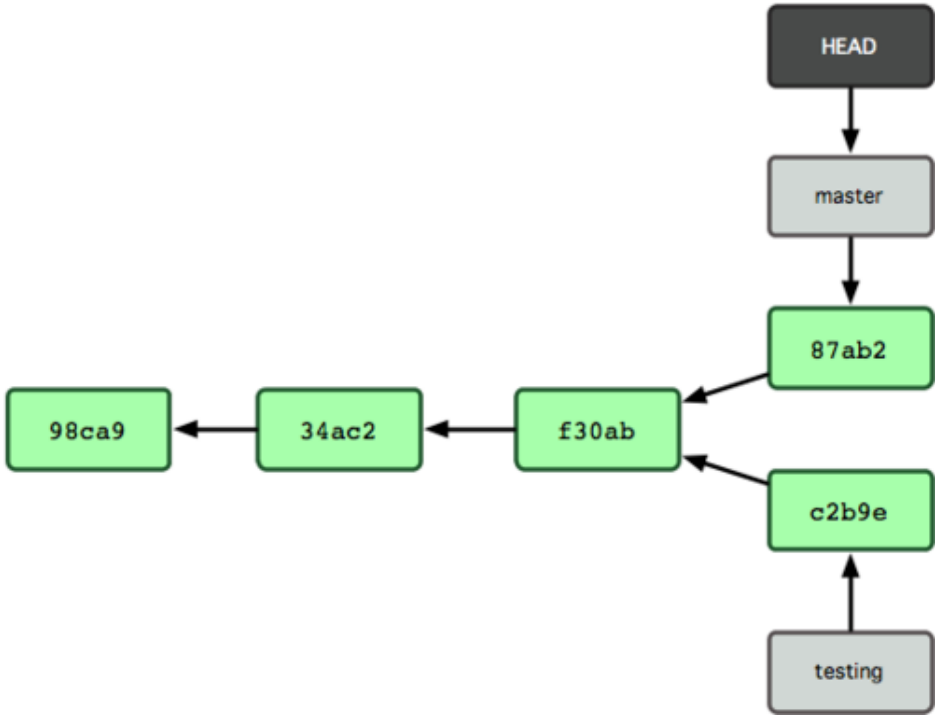
```
$ vim test.rb
$ git commit -a -m 'made a change'
```



O que são branchs? (5)

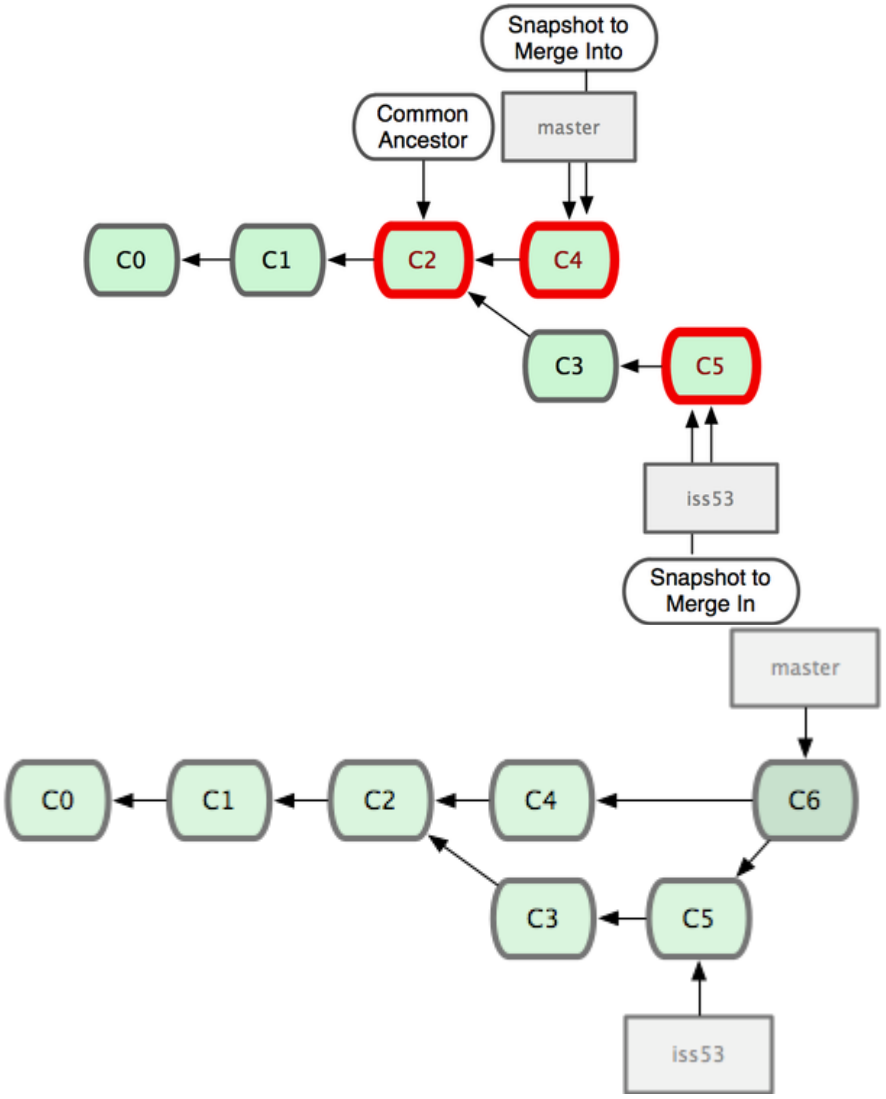
- Vamos voltar ao branch master e fazer outro commit.

```
$ git checkout master
$ vim test.rb
$ git commit -a -m 'made other changes'
```



Merge

- Vamos fazer um merge nas modificações do testing na branch master.




```
$ git checkout master
$ git merge testing
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions
(+), 0 deletions(-)
```

E se o merge não for possível? (1)

```
$ git merge testing
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.


[master*]$ git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#   unmerged:   index.html
#
```

E se o merge não for possível? (2)

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53:index.html


$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   index.html
#
```

E se o merge não for possível? (3)

```
$ git commit


Merge branch 'testing'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

Gerenciando branchs

- git branch lista todos os branchs disponíveis.
- A opção -v mostra o último commit de cada branch.
- A opção --merged mostra quais branchs já sofreram merged no branch atual (geralmente indica um branch seguro para deletar com -d).
- Também existe o --no-merged, que faz o contrário.
- -D deleta um branch --no-merged.

Tags

Existem dois tipos de tags:

- *Lightweight tags* guardam o checksum de um commit num arquivo, basicamente um branch que não muda.

```
$ git tag v1.4-lw
```

- *Annotated tags* são objetos completos, incluindo o nome do criador da tag, e-mail e data. Pode ser assinado com uma chave PGP para garantir a identidade da tag.

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

Stash

- Não é possível mudar de branch enquanto existem modificações pendentes (não commitadas) no branch atual, mas podemos guardar as alterações feitas usando stash.

- git stash guarda as alterações atuais,
- git stash apply aplicam as alterações guardadas,
- git stash list mostram os stashs guardados,
- git stash drop joga o primeiro stash fora.

Branchs remotos (1)

- Se você precisar compartilhar o trabalho de um branch, é necessário enviar seu branch para um servidor:

```
$ git push origin testing
```

- O Git não vai clonar os branchs remotos automaticamente. Precisamos fazer isso explicitamente.

```
$ git checkout -b serverfix origin/serverfix
```

Branchs remotos (2)

- Se você quiser que um branch local monitore algum branch remoto, faça:

```
git checkout -b [branch] [remotename]/[branch]
```

ou

```
$ git checkout --track origin/serverfix
```

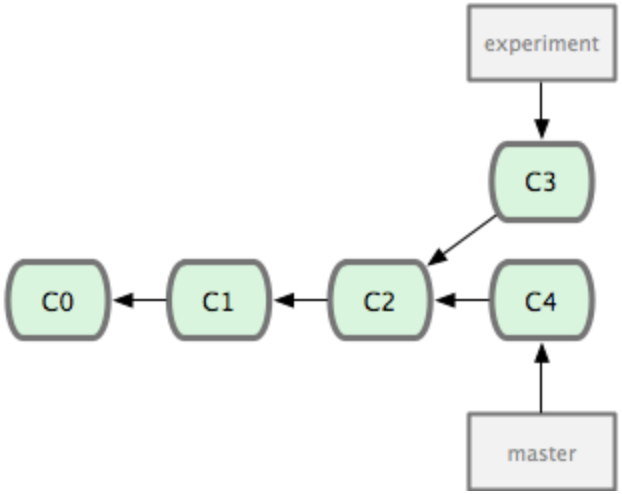
- Deletando branchs remotos:

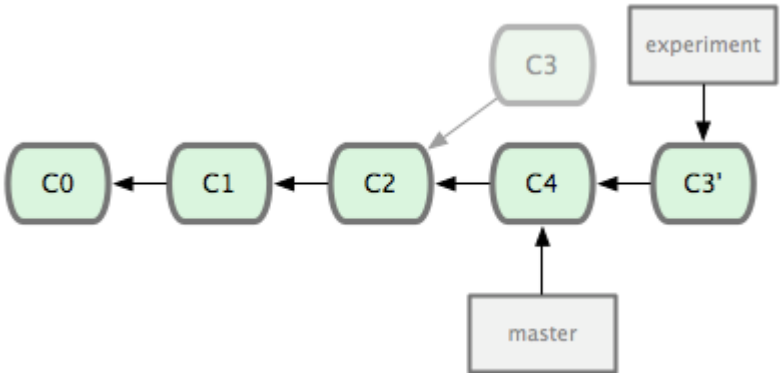
```
$ git push origin :serverfix
```

Rebase (1)

- Outra maneira de juntar seu trabalho a um outro branch.

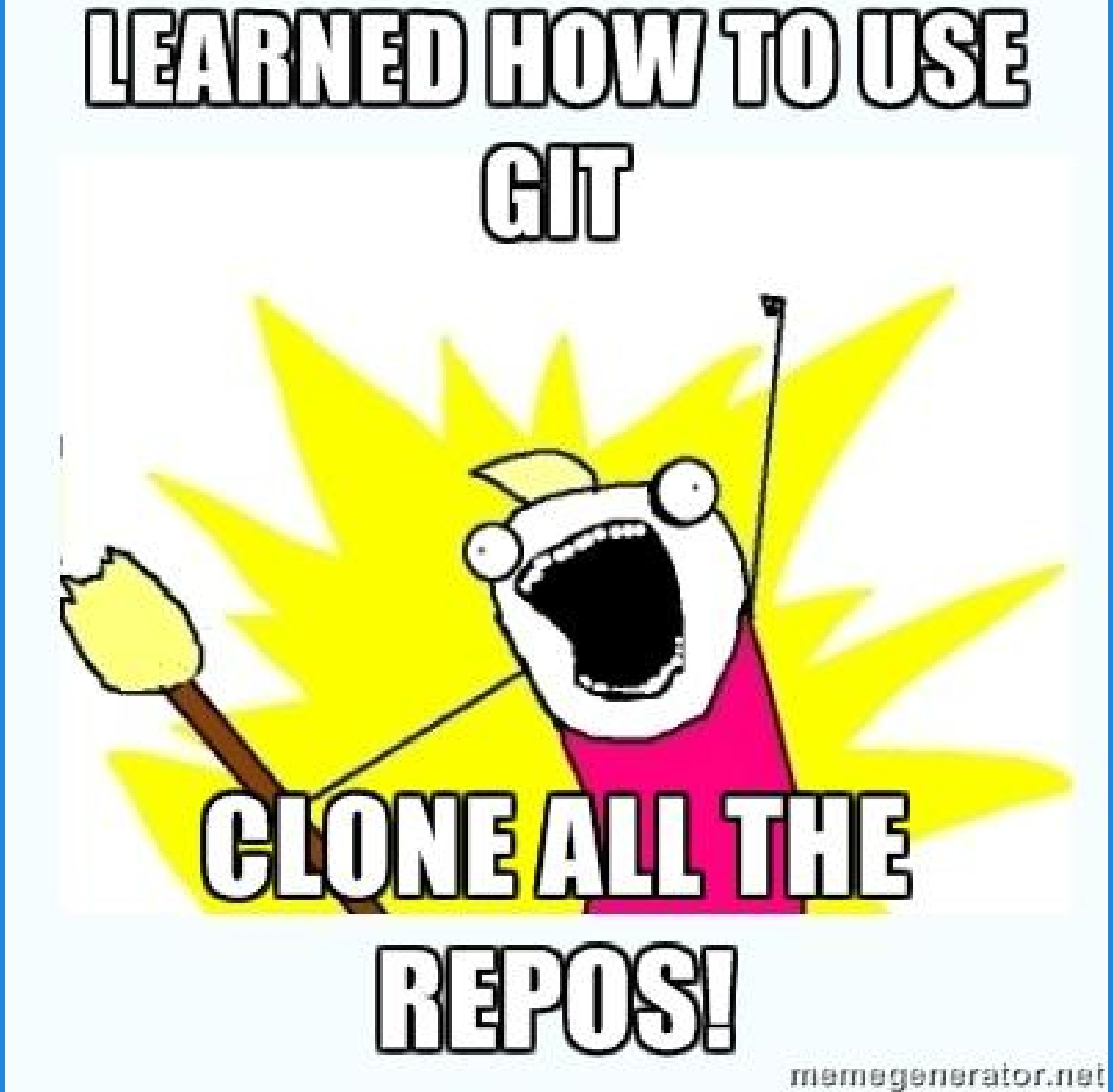
- Não faça rebases em commits que já foram enviados num repositório público.





Rebase (2)

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```


LEARNED HOW TO USE GIT
CLONE ALL THE REPOS!
memegenerator.net

Para saber mais

1. CHACON, Scott. *Pro Git*. Apress, 2009. Disponível gratuitamente em <http://git-scm.com/book/>.
2. [Screencast] Começando com o Git. AkitaOnRails.com. Disponível em <http://bit.ly/SXcohh>.