Controle de versão com git

Minicurso

Danilo J. S. Bellini
@danilobellini

2024-10-29

SECOMP 2024 @ UFSCar/SP

Breve história dos sistemas de controle de versão e surgimento do `git`

Sistemas de controle de versão:

- CVS (*Concurrent Versions System*), 1986-2008 GPLv1+
- BitKeeper 2000-2018, Apache desde 2016-05-09
- Subversion (`svn`), desde 2000-10-20, Apache
- Mercurial (`hg`), desde 2005-04-19, GPLv2+
- Git, desde 2005-04-07, GPLv2-only

Linux (kernel):

- Um diretório por release até 2002, não havia controle de versão
- Usou o BitKeeper (nonfree) desde 2002, gratuito como "incentivo ao software livre"
- Em 2005, Andrew Tridgell usou engenharia reversa (rede) para criar uma alternativa open source ao BitKeeper, então Larry McVoy retirou o "incentivo"
- Linus Torvalds interrompe o desenvolvimento do Linux e inicia o planejamento e o desenvolvimento do git
- "Thank You, Larry McVoy" por Richard Stallman

Por que o nome "`git`"?

> *"Sou egoísta, e dou nomes relativos a mim em todos os meus projetos." (Linus Torvalds)*

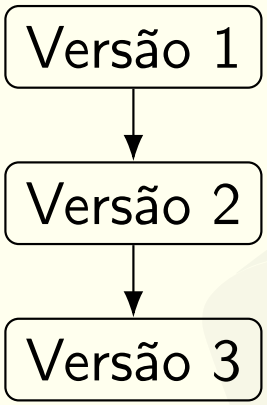
A palavra `git` é uma gíria britânica pejorativa. O `README` do projeto diz que o significado depende de seu humor:

- Arranjo aleatório pronunciável de 3 letras que não é utilizado por nenhum comando UNIX comum. O fato de poder ser um "get" pronunciado erroneamente pode ou não ser relevante.
- Estúpido. Contemptível e desprezível. Simples. Escolha com base em seu dicionário de gírias.
- "*Global information tracker*": Você está de bom humor, e ele funciona para você. Anjos cantam e a luz subitamente preenche a sala.
- "*Goddamn idiotic truckload of sh*t*": quando ele quebra.

Curiosidade: Como o git é descrito em sua *man page* (`man git`)?

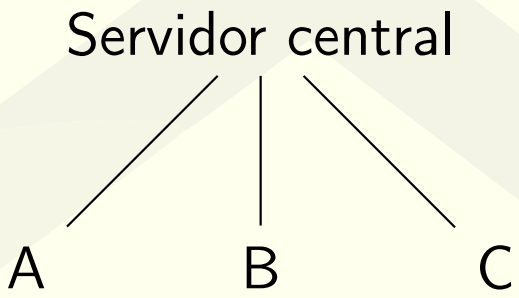
Tipos de sistemas de controle de versão

Local
(e.g. GNU RCS)



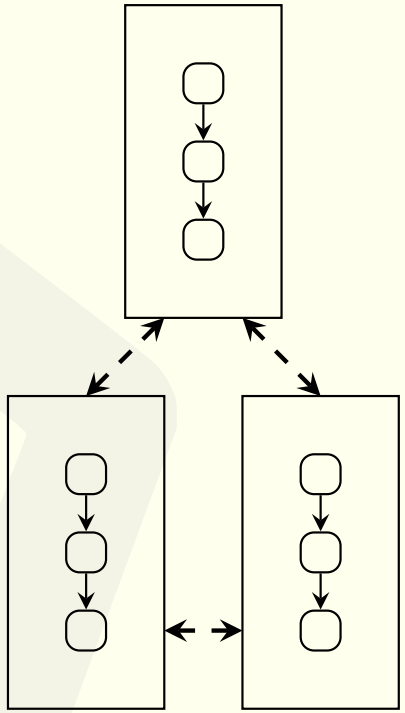
Banco de dados de arquivos (NoSQL) agrupados por versão, sem comunicação remota

Centralizado
(e.g. CVS, SVN)



Um único servidor é a *source of truth*, os demais nós não possuem toda informação; risco de ponto único de falha e interrupção de trabalho por falhas na rede

Distribuído
(e.g. `git`, hg)



Toda informação está (ou pode estar) disponível em todos os nós.

Snapshot VS *delta patching*

Uma das escolhas mais relevantes do *design* do `git` é o fato de que todos os arquivos são armazenados integralmente (*snapshot*). Ou seja, **todos os arquivos** em todas as suas versões **são armazenados por completo**, e as "**mudanças aplicadas**" **são** na realidade **computadas** sob demanda.

	Commit1	Commit2	Commit3	Commit4	Commit5
`README`	R1	...R1...	...R1...	...R1...	R2
`a.py`	N/A	m1	m2	...m2...	m3
`b.py`	N/A	N/A	N/A	p1	...p1...

Isso permite que todo arquivo seja acessado diretamente, sem a necessidade de "reconstruir" cada arquivo a partir de suas mudanças individuais (contraste com CVS e `darcs`).

Commits no `git`

Podemos interpretar o `git` como um banco de dados NoSQL de "*snapshots*" imutáveis, as versões ou *commits*. *Commits*:

- Possuem um *hash* determinístico (*git* utiliza SHA-1)
- Consistem em um conjunto de arquivos (conteúdo, *path* e *bit* de execução), além de metadados: mensagem (título e descrição), autor, *timestamp* de autoria, *committer*, *timestamp* do *commit*, lista ordenada de *parents* (ancestrais imediatos) referenciados pelos seus *hashes*
- São referenciáveis, por exemplo através de seu *hash* ou seus primeiros 7 ou mais *nibbles* (dígitos hexadecimais)
- Formam um DAG (grafo direcional acíclico)

Show me the co... commmand lines

```
mkdir primeiro_repositorio
cd primeiro_repositorio
git init  # Cria o diretório .git/ sem nenhum commit

echo Hello World > hello_world.txt
git add hello_world.txt  # Coloca a alteração em "staging"
                         # (marcado para entrar no commit)

git commit  # Chama o editor para uma mensagem

git log  # Mostra a história de commits
git show  # Mostra o commit HEAD (referência padrão)
gitk  # Visualização em GUI (opcional, requer Tcl/Tk)
```

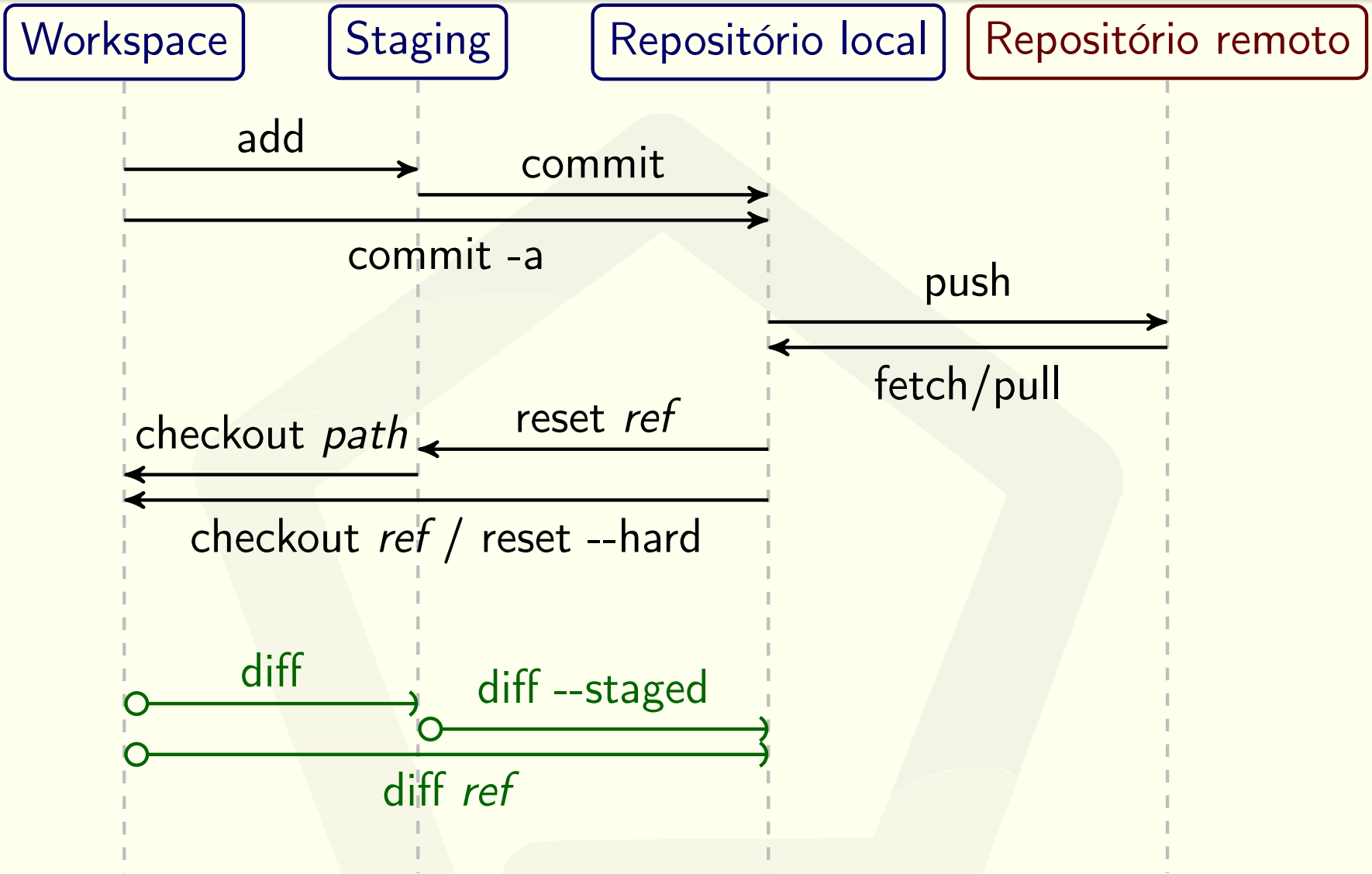
Para o primeiro uso, é necessário configurar o nome e e-mail:

```
git config --global user.name 'Fulano de Tal'
git config --global user.email 'fulano@exemplo.com.br'
git config --global core.editor vim  # Opcional

# --global é a configuração do usuário (~/.gitconfig)
# --local  é a configuração do repositório (.git/config)

git config -l --show-scope # Mostra a configuração
man git config  # A seção "Variables" pode ser útil
```

Modelo do git para commits e respectivos comandos



O `git diff` não realiza alterações; "*staging*" também é chamado de *índice* ou *cache*; *ref* é uma referência a um *commit* (versão).

Referências a *commits*, *branches*, *detached HEAD*

Uma referência especial que merece destaque é a HEAD, um nome para o commit "atual" tomado como base de comparação ao avaliar o que mudou no *workspace*, e que sempre acompanha novos *commits*. A HEAD pode apontar para:

- Uma *branch* (ramificação) local, através de seu *nome*. Nesse caso, novos *commits* modificam a *branch*, continuado-a sequencialmente, e a HEAD naturalmente acompanha tais modificações, mas continua a apontar para a *branch*.
- Um *hash*, referenciando um único *commit*. Nesse caso, dizemos que a referência HEAD está `detached` (desprendida). Cada novo *commit* altera a HEAD, mas isso não possui nenhum outro efeito.

Esse comportamento é característico das *branches* locais; todos os outros tipos de referências após um *checkout* resultarão em *detached HEAD*, o que significa que apenas as *branches* acompanham naturalmente novos *commits*.

Primeiros experimentos com *branches*, *tags* e outras *refs*

```
cria_commit_um_arquivo() {
  echo "# Arquivo vazio $1" > $1
  git add $1
  git commit -m "Insere $1"
}

mkdir segundo_repositorio && cd segundo_repositorio
git init

# Commit inicial na branch principal (master ou main)
cria_commit_um_arquivo raiz.py
git branch  # Permite ver a branch atual

# Nova branch "be-um" terá um commit acima de main
git checkout -b be-um  # "-b" cria uma nova branch, similar a
                       # git branch b1 && git checkout b1
cria_commit_um_arquivo b1.py

# Nova branch "be-dois" terá 2 commits acima de main
git checkout HEAD^  # O "^" representa "parent"
cria_commit_um_arquivo b2.py  # Ainda detached!
git tag estava-detached  # Cria uma tag (outra ref)
git checkout -b be-dois
cria_commit_um_arquivo b2_child.py

# Visualização
git log --oneline --all --graph
```

Exercício de visualização, clone/bisect/grep e Quatérnions!

Exercício: Quatérnions são números "4D" não comutativos com propriedades particularmente relevantes para modelar rotações em 3D de maneira eficiente e eficaz, por exemplo em computação gráfica. Encontrar o autor, a data do primeiro commit a partir do qual o Sympy passou a ter a palavra `Quaternion`.

```
git clone https://github.com/sympy/sympy
cd sympy

# Inicia a busca por um commit
git bisect start  # Inicia a busca
git bisect new    # O commit atual é novo (posterior)
git checkout REF  # Alguma ref que seja antiga (qual?)
git bisect old    # Marca o commit como anterior
# ... em cada novo commit, é preciso verificar manualmente
# se ele é posterior ou anterior ao que se busca,
# e então chamar o git bisect new ou old para achar
# o próximo commit
git bisect reset  # Finaliza a busca
```

Clonando via SSH ou HTTPS faz diferença? Qual o resultado de `git remote -v`? As datas de autoria e commit batem?

FIM!

https://github.com/danilobellini/slides-latex

Referências:

- https://www.gnu.org/philosophy/mcvoy.en.html (Richard Stallman, "Thank You, Larry McVoy")
- https://www.linuxjournal.com/content/git-origin-story (Zack Brown)
- https://pt.slideshare.net/slideshow/20150314-grupysp-projetos-open-source-como-colaborar/45854503#2
- https://git-scm.com/book
- https://github.com/git/git
- https://github.com/danilobellini/git-tutorial-br/