



GIT - CURSO BÁSICO

Sérgio Vieira

sergiosvieira@gmail.com

previsão de 8 horas

referência: <http://git-scm.com/book/pt-br/>

AGENDA

- I Primeiros Passos
 - I.1 Sobre Controle de Versão
 - I.2 Uma Breve História do Git
 - I.3 Noções Básicas de Git

AGENDA

- I Primeiros Passos
 - I.4 Instalando Git
 - I.5 Configuração Inicial do Git
 - I.6 Obtendo Ajuda

AGENDA

- 2. Git Essencial
 - 2.1 Obtendo um Repositório Git
 - 2.2 Gravando Alterações no Repositório
 - 2.3 Visualizando o Histórico de Commits
 - 2.4 Desfazendo Coisas

AGENDA

- 2. Git Essencial
 - 2.5 Trabalhando com Remotos
 - 2.6 Tagging
 - 2.7 Dicas e Truques

AGENDA

- 3. Ramificação (Branching) no Git
 - 3.1 O que é um Branch
 - 3.2 Básico de Branch e Merge
 - 3.3 Gerenciamento de Branches
 - 3.4 Fluxos de Trabalho com Branches

AGENDA

- 3. Ramificação (Branching) no Git
 - 3.5 Branches Remotos
 - 3.6 Rebasing

AGENDA

- 4. Git no Servidor
 - 4.1 Os Protocolos
 - 4.2 Configurando Git no Servidor
 - 4.3 Gerando Sua Chave Pública SSH
 - 4.4 Configurando o Servidor
 - 4.5 Acesso Público

AGENDA

- 4. Git no Servidor
- 4.6 GitlabHQ
- 4.7 Serviço Git
- 4.8 Git Hospedado



1.1 SOBRE O CONTROLE DE VERSÃO

I.I SOBRE O CONTROLE DE VERSÃO

- O controle de versão é um sistema que registra as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que você possa recuperar versões específicas.

1.1 SOBRE O CONTROLE DE VERSÃO

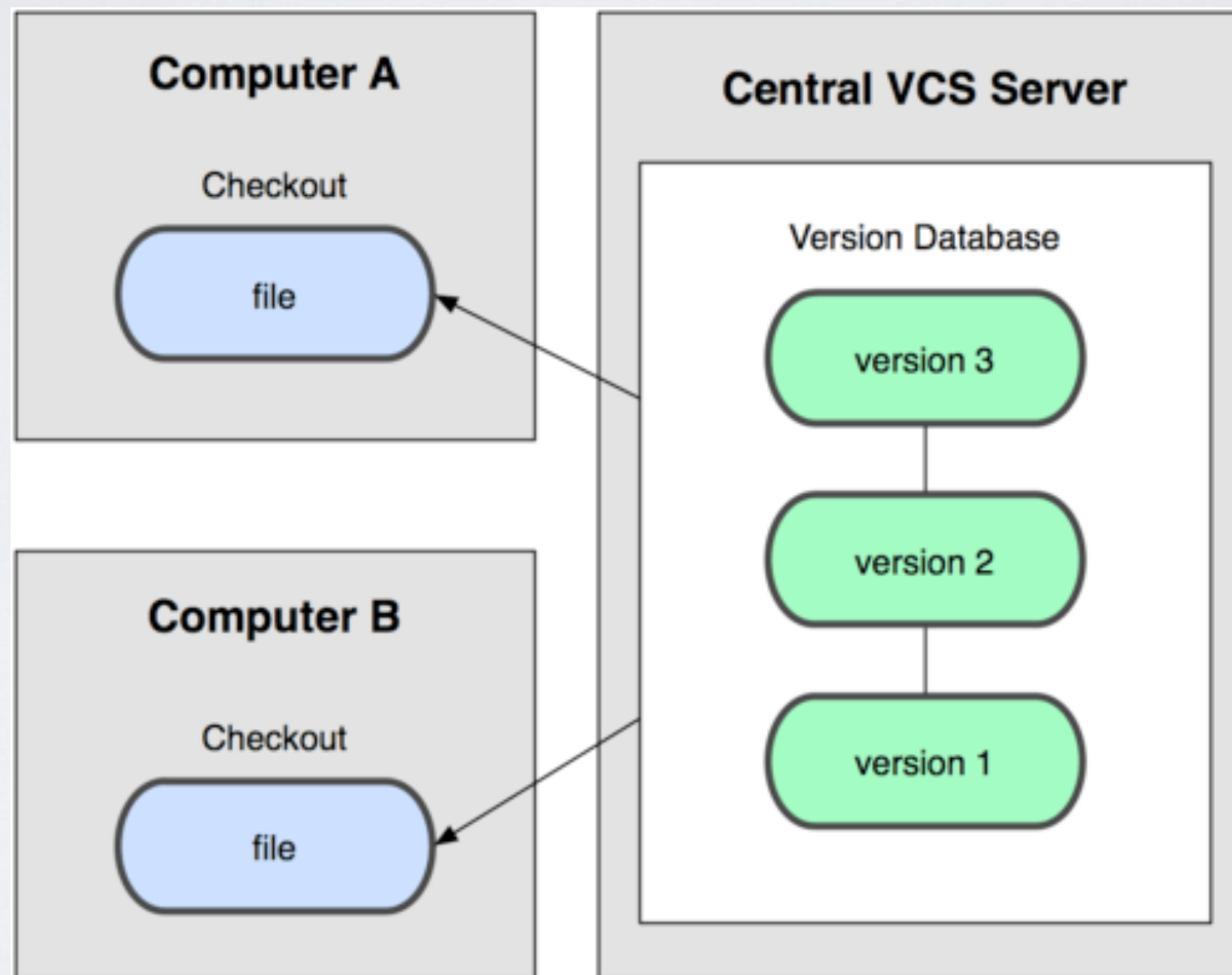


Figura 1-2. Diagrama de Controle de Versão Centralizado.

I.1 SOBRE O CONTROLE DE VERSÃO

Entretanto, esse arranjo também possui grandes desvantagens. O mais óbvio é que o servidor central é um **ponto único de falha**. Se o servidor ficar fora do ar por uma hora, **ninguém pode trabalhar em conjunto ou salvar novas versões dos arquivos durante esse período**. Se o disco do servidor do banco de dados for corrompido e não existir um backup adequado, **perde-se tudo** — todo o histórico de mudanças no projeto, exceto pelas únicas cópias que os desenvolvedores possuem em suas máquinas locais.

1.1 SOBRE O CONTROLE DE VERSÃO

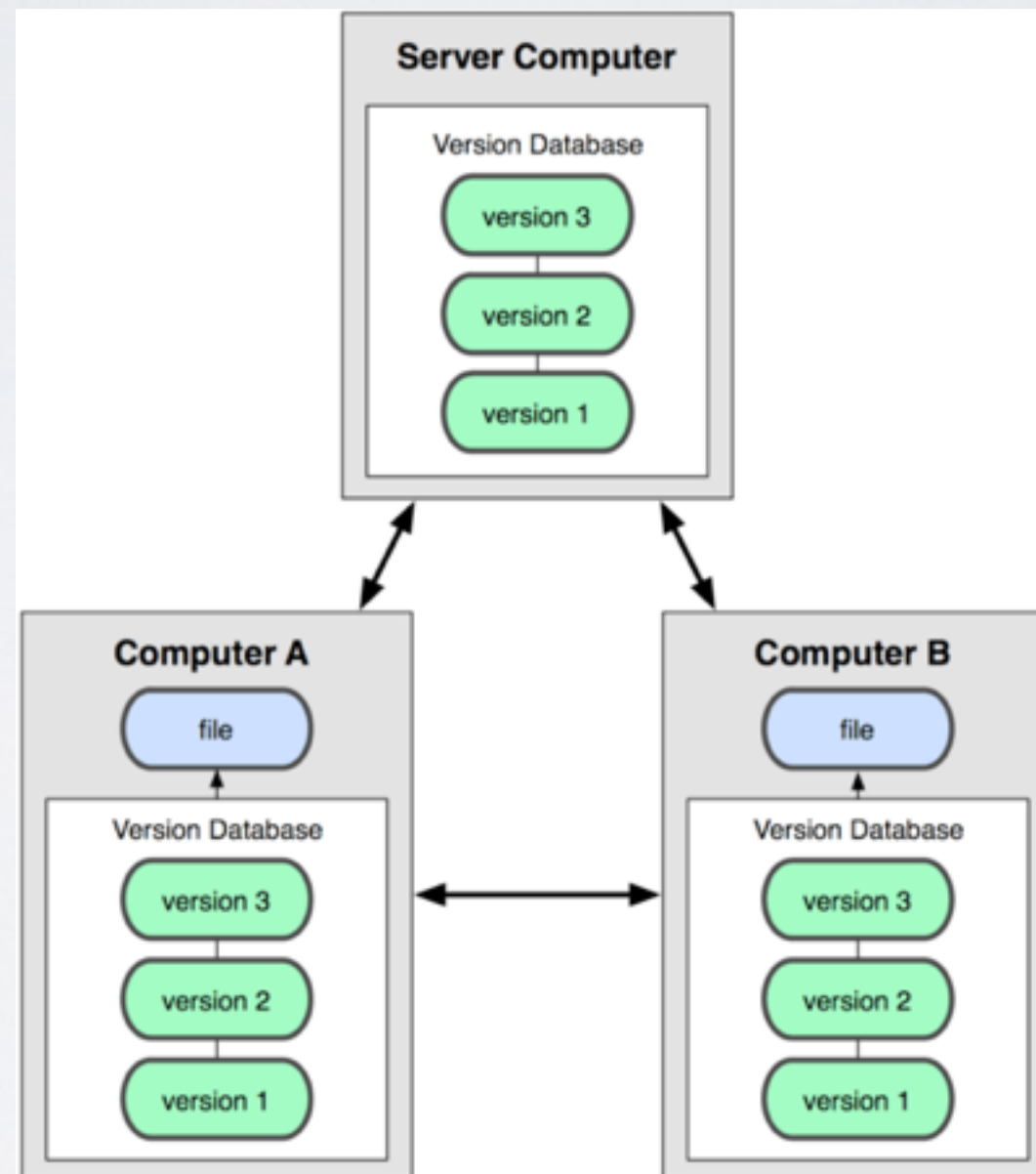


Figura 1-2. Diagrama de Controle de Versão Distribuído.

1.1 SOBRE O CONTROLE DE VERSÃO

- Os clientes não apenas fazem cópias das últimas versões dos arquivos: eles são cópias completas do repositório.
- Assim, se um servidor falha, qualquer um dos repositórios dos clientes pode ser copiado de volta para o servidor para restaurá-lo.
- Além disso, muitos desses sistemas lidam muito bem com o aspecto de ter vários repositórios remotos com os quais eles podem colaborar, permitindo que você trabalhe em conjunto com diferentes grupos de pessoas, de diversas maneiras, simultaneamente no mesmo projeto



1.2 PRIMEIROS PASSOS

UMA BREVE HISTÓRIA DO GIT

- Durante a maior parte do período de manutenção do kernel do Linux (1991-2002), as mudanças no software eram repassadas como patches e arquivos compactados.
- Em 2002, o projeto do kernel do Linux começou a usar um sistema DVCS proprietário chamado BitKeeper.
- Em 2005, o relacionamento entre a comunidade que desenvolvia o kernel e a empresa que desenvolvia comercialmente o BitKeeper se desfez, e o status de isento-de-pagamento da ferramenta foi revogado. Isso levou a comunidade de desenvolvedores do Linux (em particular Linus Torvalds, o criador do Linux) a desenvolver sua própria ferramenta baseada nas lições que eles aprenderam ao usar o BitKeeper.

UMA BREVE HISTÓRIA DO GIT

- Alguns dos objetivos do novo sistema eram:
 - Velocidade
 - Design simples
 - Suporte robusto a desenvolvimento não linear (milhares de branches paralelos)
 - Totalmente distribuído
 - Capaz de lidar eficientemente com grandes projetos como o kernel do Linux (velocidade e volume de dados)

NOÇÕES BÁSICAS DE GIT

- **Atenção**

- À medida que você aprende a usar o Git, tente não pensar no que você já sabe sobre outros VCSs como Subversion e Perforce; assim você consegue escapar de pequenas confusões que podem surgir ao usar a ferramenta.

NOÇÕES BÁSICAS DE GIT

- **Atenção**

- Apesar de possuir uma interface parecida, o Git armazena e pensa sobre informação de uma forma totalmente diferente desses outros sistemas; entender essas diferenças lhe ajudará a não ficar confuso ao utilizá-lo.



SNAPSHOTS, E NÃO DIFERENÇAS

NOÇÕES BÁSICAS DE GIT

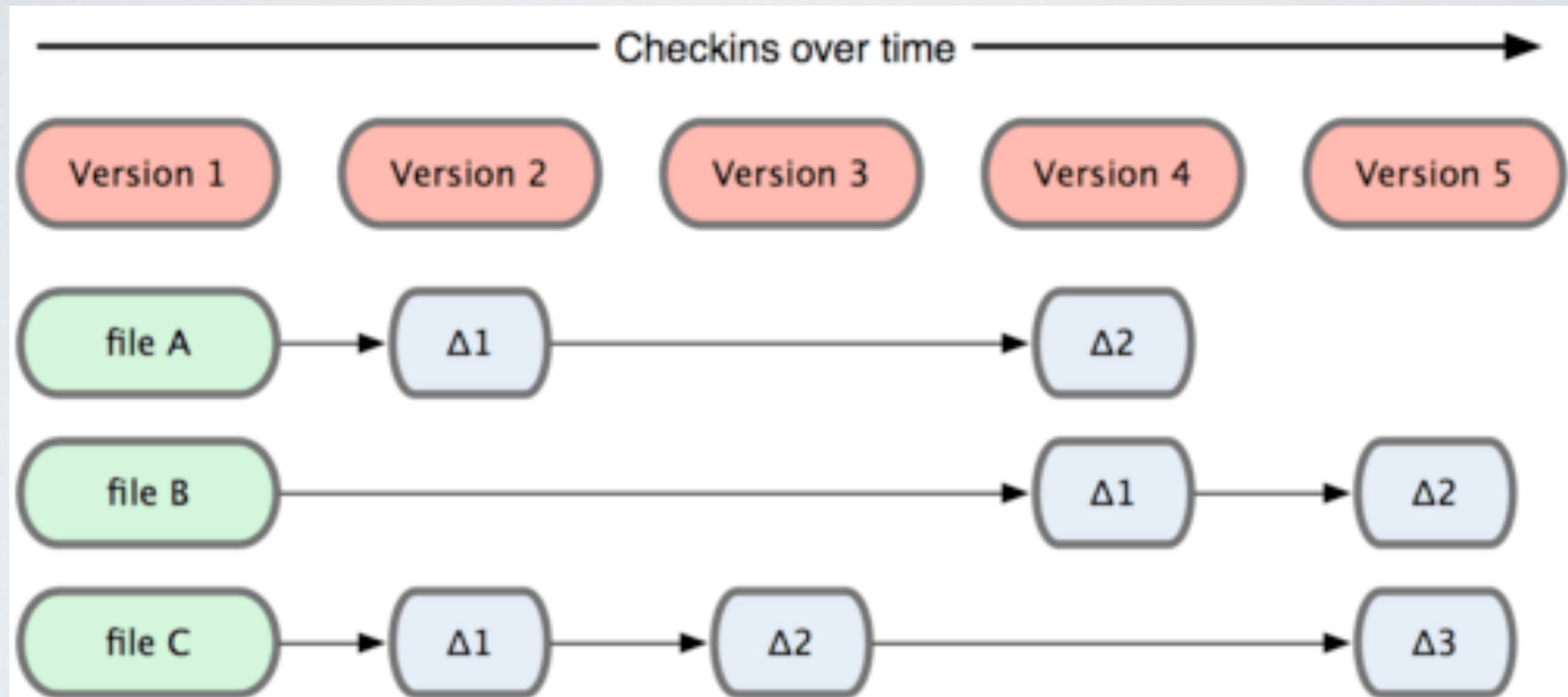


Figura 1-4. Outros sistemas costumam armazenar dados como mudanças em uma versão inicial de cada arquivo.

NOÇÕES BÁSICAS DE GIT

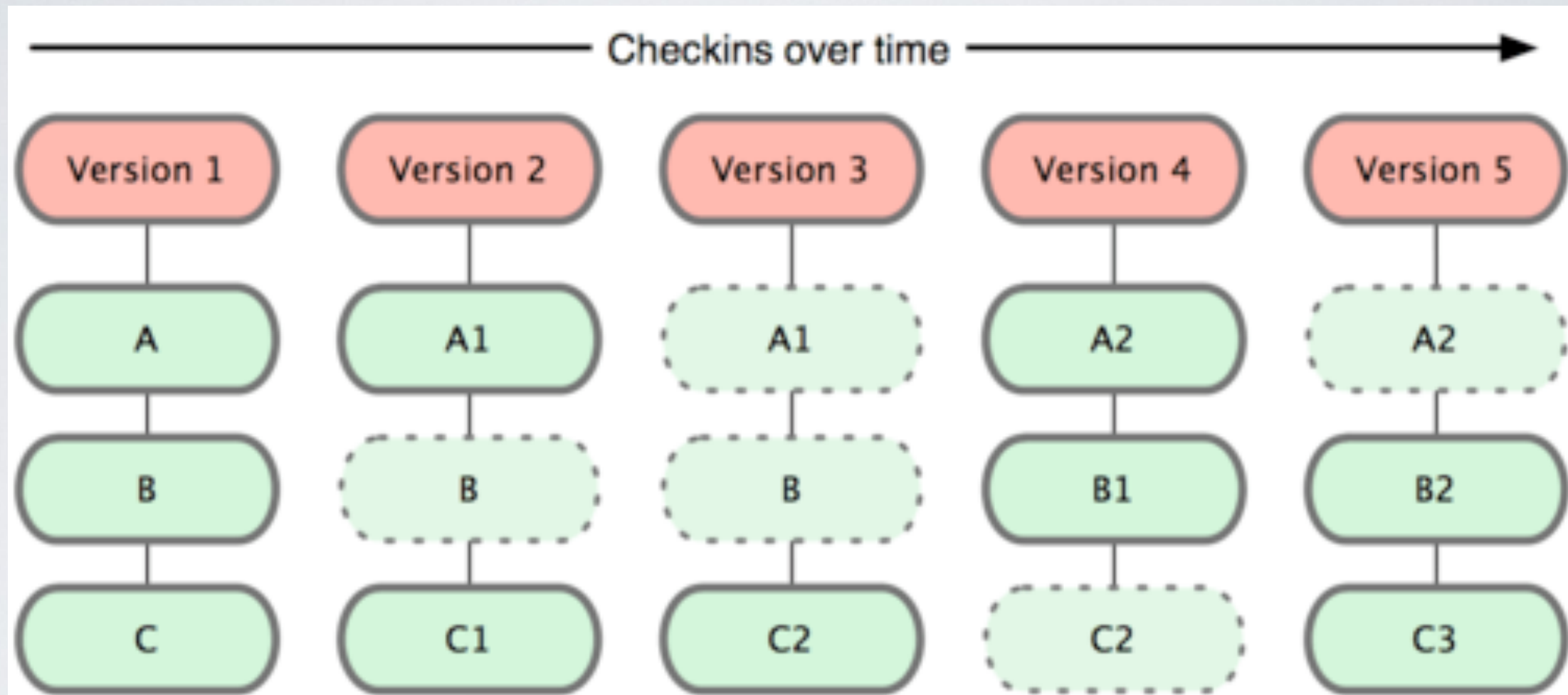


Figura 1-5. Git armazena dados como snapshots do projeto ao longo do tempo.



QUASE TODAS AS
OPERAÇÕES SÃO LOCAIS

NOÇÕES BÁSICAS DE GIT

- A maior parte das operações no Git precisam apenas de recursos e arquivos locais para operar — geralmente nenhuma outra informação é necessária de outro computador na sua rede.
- Para navegar no histórico do projeto, o Git não precisa requisitar ao servidor o histórico para que possa apresentar a você — ele simplesmente lê diretamente de seu banco de dados local. Isso significa que você vê o histórico do projeto quase instantaneamente.



GIT TEM INTEGRIDADE

NOÇÕES BÁSICAS DE GIT

- Tudo no Git tem seu checksum (valor para verificação de integridade) calculado antes que seja armazenado e então passa a ser referenciado pelo checksum.
- O mecanismo que o Git usa para fazer o checksum é chamado de hash SHA-1, uma string de 40 caracteres composta de caracteres hexadecimais (0-9 e a-f) que é calculado a partir do conteúdo de um arquivo ou estrutura de um diretório no Git. Um hash SHA-1 parece com algo mais ou menos assim:

24b9da6552252987aa493b52f8696cd6d3b00373

NOÇÕES BÁSICAS DE GIT

- tudo que o Git armazena é identificado não por nome do arquivo mas pelo valor do hash do seu conteúdo.



GIT GERALMENTE SÓ
ADICIONA DADOS

NOÇÕES BÁSICAS DE GIT

- Dentre as ações que você pode realizar no Git, quase todas apenas acrescentam dados à base do Git.
- Você pode perder ou bagunçar mudanças que ainda não commitou; mas depois de fazer um commit de um snapshot no Git, é muito difícil que você o perca.
- Isso faz com que o uso do Git seja uma alegria no sentido de permitir que façamos experiências sem o perigo de causar danos sérios.



OS TRÊS ESTADOS

Essa é a coisa mais importante pra se lembrar sobre Git se você quiser que o resto do seu aprendizado seja tranquilo.

NOÇÕES BÁSICAS DE GIT

- Git faz com que seus arquivos sempre estejam em um dos três estados fundamentais: consolidado (committed), modificado (modified) e preparado (staged).

NOÇÕES BÁSICAS DE GIT

- Committed
 - Dados são ditos consolidados quando estão seguramente armazenados em sua base de dados local.

NOÇÕES BÁSICAS DE GIT

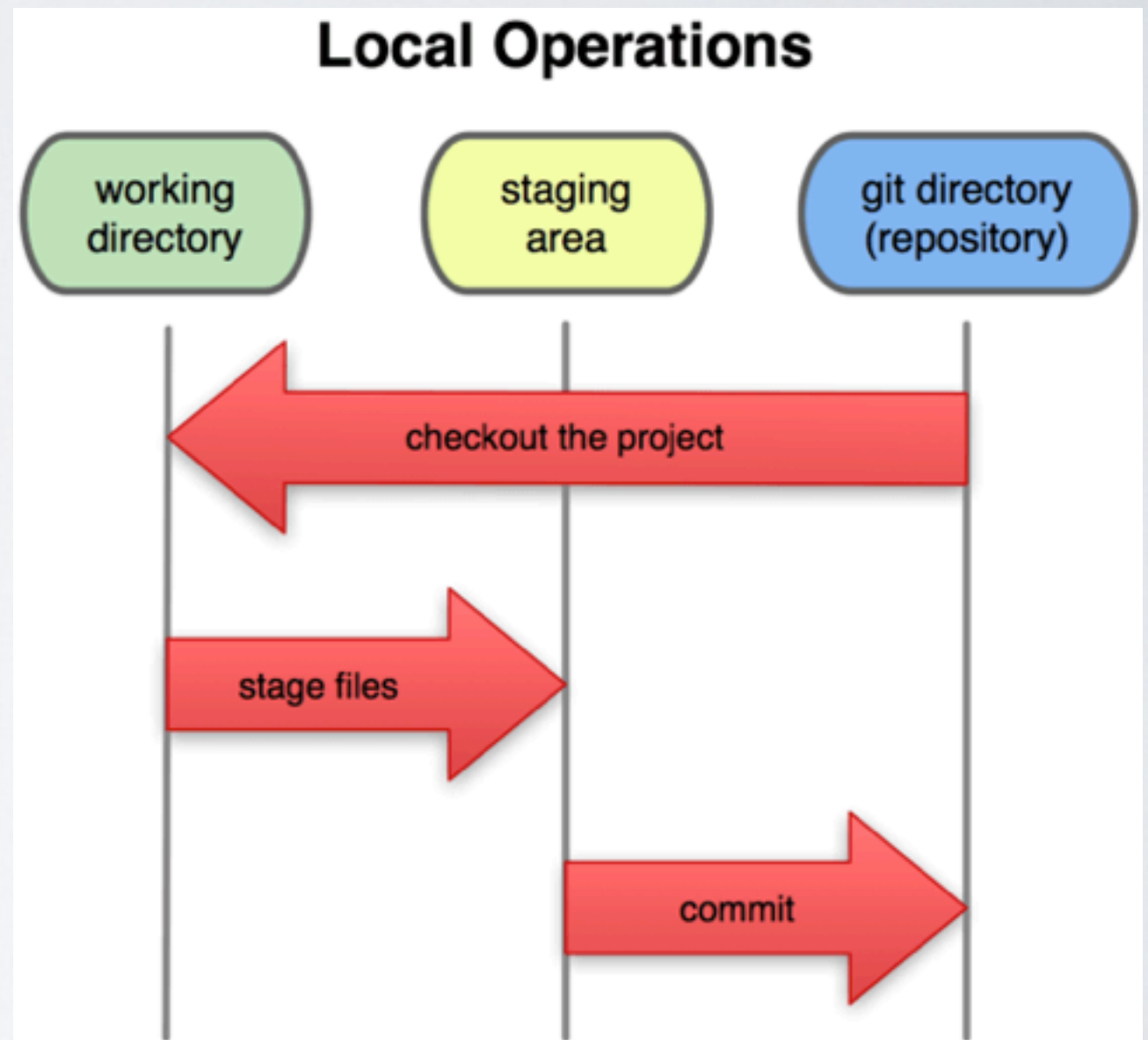
- Modified
 - Modificado trata de um arquivo que sofreu mudanças mas que ainda não foi consolidado na base de dados.

NOÇÕES BÁSICAS DE GIT

- Staged
 - Um arquivo é tido como preparado quando você marca um arquivo modificado em sua versão corrente para que ele faça parte do snapshot do próximo commit.

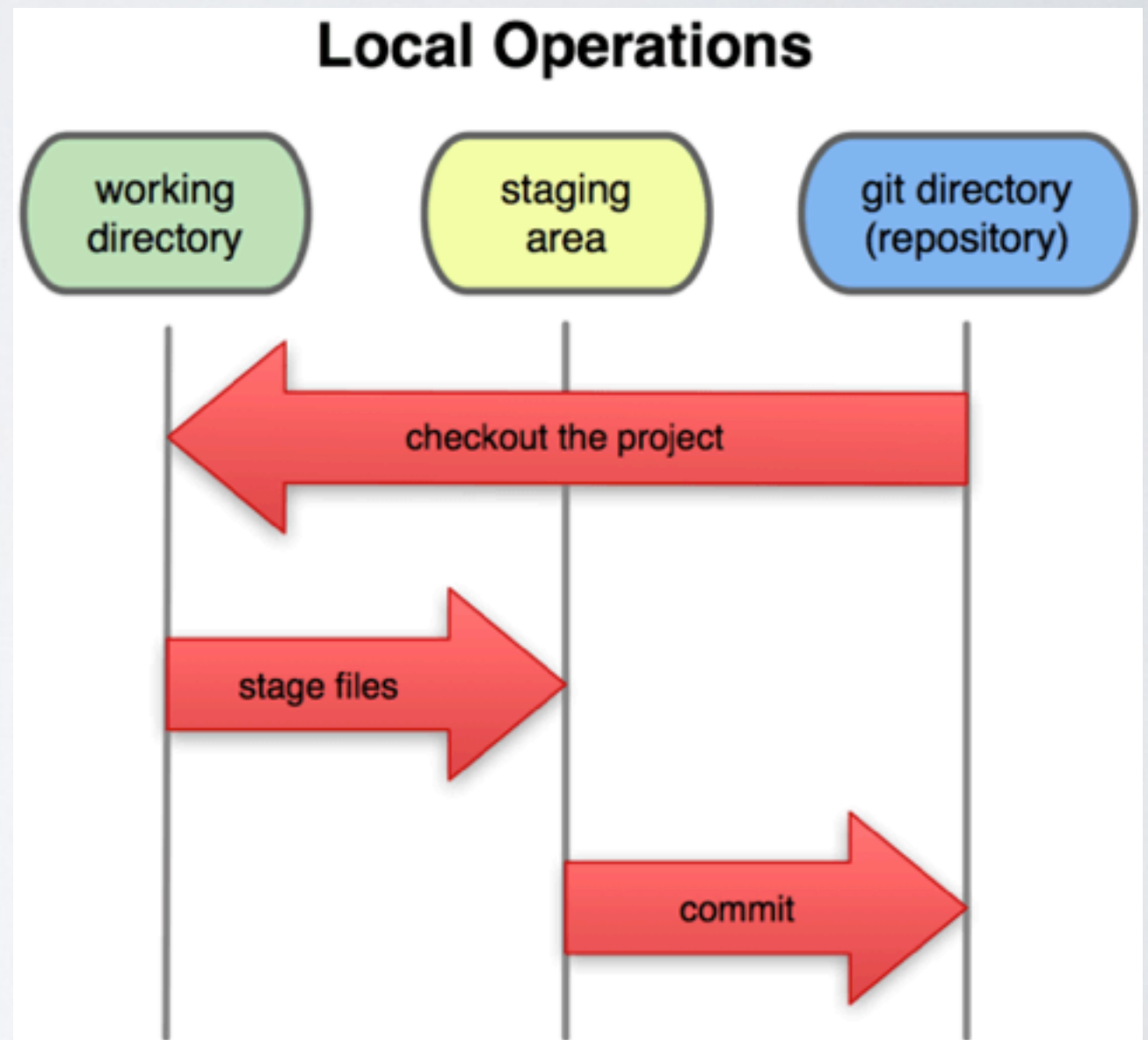
NOÇÕES BÁSICAS DE GIT

- O diretório do Git é o local onde o Git armazena os metadados e o banco de objetos de seu projeto. Esta é a parte mais importante do Git, e é a parte copiada quando você clona um repositório de outro computador.



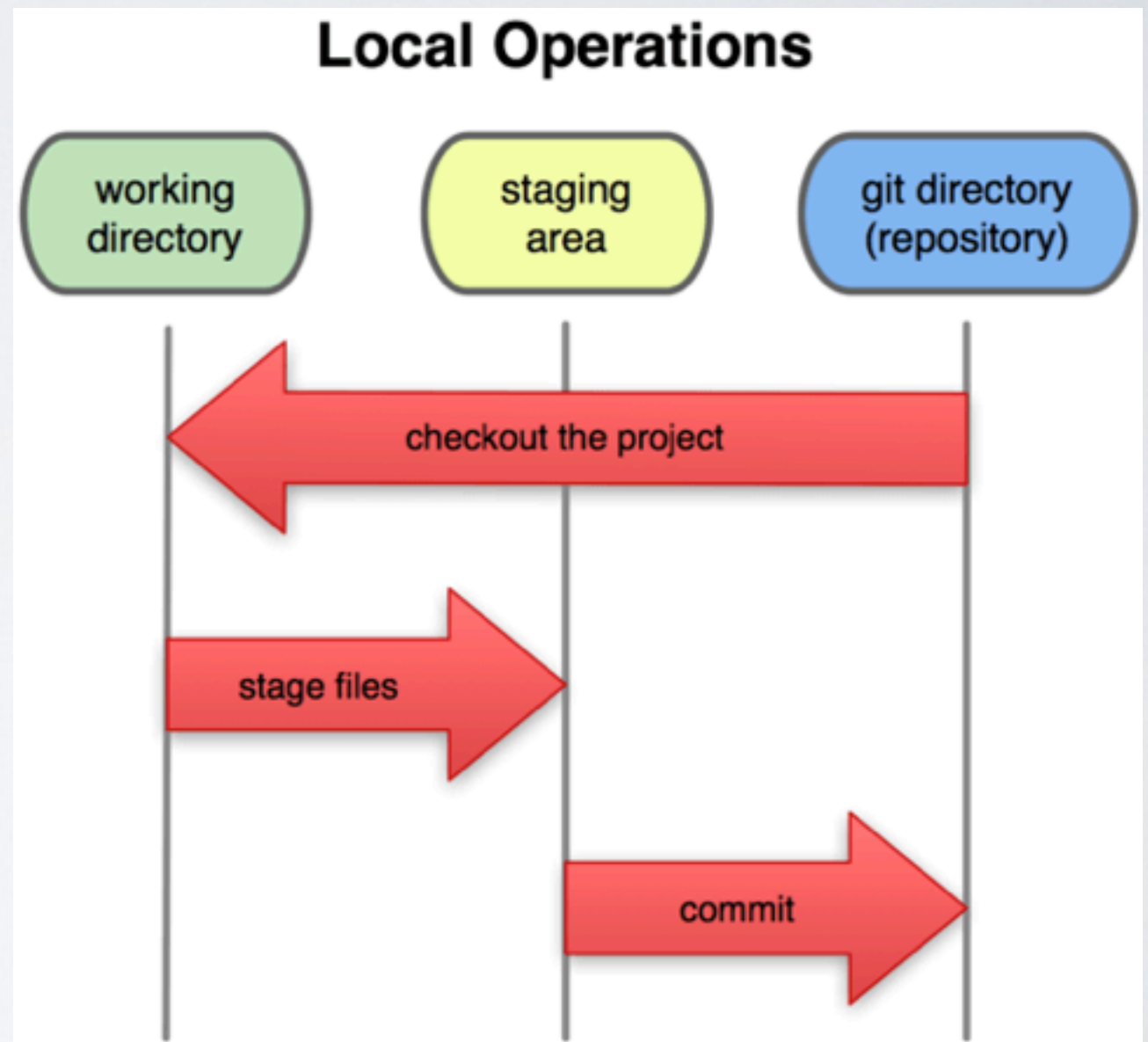
NOÇÕES BÁSICAS DE GIT

- O diretório de trabalho é um único checkout de uma versão do projeto. Estes arquivos são obtidos a partir da base de dados comprimida no diretório do Git e colocados em disco para que você possa utilizar ou modificar.



NOÇÕES BÁSICAS DE GIT

- A área de preparação é um simples arquivo, geralmente contido no seu diretório Git, que armazena informações sobre o que irá em seu próximo commit. É bastante conhecido como índice (index), mas está se tornando padrão chamá-lo de área de preparação.



NOÇÕES BÁSICAS DE GIT

- O workflow básico do Git pode ser descrito assim:
 1. Você modifica arquivos no seu diretório de trabalho.
 2. Você seleciona os arquivos, adicionando snapshots deles para sua área de preparação.
 3. Você faz um commit, que leva os arquivos como eles estão na sua área de preparação e os armazena permanentemente no seu diretório Git.

NOÇÕES BÁSICAS DE GIT

- Se uma versão particular de um arquivo está no diretório Git, é considerada consolidada.
- Caso seja modificada mas foi adicionada à área de preparação, está preparada.
- E se foi alterada desde que foi obtida mas não foi preparada, está modificada.

INSTALANDO GIT





INSTALANDO NO LINUX

NOÇÕES BÁSICAS DE GIT

- Fedora

```
$ sudo yum install git-core
```

- Ubuntu/Debian

```
$ sudo apt-get install git
```



INSTALANDO NO MAC

NOÇÕES BÁSICAS DE GIT

- Instalador gráfico do Git

<http://sourceforge.net/projects/git-osx-installer/>

- MacPorts

```
$ sudo port install git-core +svn +doc  
+bash_completion +gitweb
```



INSTALANDO NO WINDOWS

NOÇÕES BÁSICAS DE GIT

- Instalador

<http://msysgit.github.com>

CONFIGURAÇÃO INICIAL DO GIT



CONFIGURAÇÃO INICIAL DO GIT

- Git vem com uma ferramenta chamada git config que permite a você ler e definir variáveis de configuração que controlam todos os aspectos de como o Git parece e opera.
- arquivo /etc/gitconfig: Contém valores para todos usuários do sistema e todos os seus repositórios. Se você passar a opção --system para git config, ele lerá e escreverá a partir deste arquivo especificamente.

CONFIGURAÇÃO INICIAL DO GIT

- arquivo `~/.gitconfig`: É específico para seu usuário. Você pode fazer o Git ler e escrever a partir deste arquivo especificamente passando a opção `--global`.
- arquivo de configuração no diretório git (ou seja, `.git/config`) de qualquer repositório que você está utilizando no momento: Específico para aquele único repositório. Cada nível sobrepõem o valor do nível anterior, sendo assim valores em `.git/config` sobrepõem aqueles em `/etc/gitconfig`.
- Em sistemas Windows, Git procura pelo arquivo `.gitconfig` no diretório `$HOME` (`C:\Documents and Settings\%USER` para a maioria das pessoas).



SUA IDENTIDADE

CONFIGURAÇÃO INICIAL DO GIT

- 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
```




SEU EDITOR

CONFIGURAÇÃO INICIAL DO GIT

- Agora que sua identidade está configurada, você pode configurar o editor de texto padrão que será utilizado quando o Git precisar que você digite uma mensagem.

```
$ git config --global core.editor emacs
```



SUA FERRAMENTA DIFF

CONFIGURAÇÃO INICIAL DO GIT

- Outra opção útil que você pode querer configurar é a ferramenta padrão de diff utilizada para resolver conflitos de merge (fusão).
- Git aceita kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge e opendiff como ferramentas válidas para merge.

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




VERIFICANDO SUAS
CONFIGURAÇÕES

CONFIGURAÇÃO INICIAL DO GIT

- Caso você queira verificar suas configurações, você pode utilizar o comando `git config --list` para listar todas as configurações que o Git encontrar naquele momento:

```
$ git config --list  
user.name=Scott Chacon  
user.email=schacon@gmail.com
```

...

```
$ git config user.name  
Scott Chacon
```

OBTENDO
AJUDA



CONFIGURAÇÃO INICIAL DO GIT

- Caso você precise de ajuda usando o Git, existem três formas de se obter ajuda das páginas de manual (manpage) para quaisquer comandos do Git:

`$ git help <verb>`

`$ git <verb> --help`

`$ man git-<verb>`

- Por exemplo, você pode obter a manpage para o comando config executando

`$ git help config`



GIT ESSENCIAL

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

INICIALIZANDO UM REPOSITÓRIO EM UM DIRETÓRIO EXISTENTE

- Isso cria um novo subdiretório chamado `.git` que contém todos os arquivos necessários de seu repositório — um esqueleto de repositório Git.

```
$ ls -lah
```

```
drwxr-xr-x  7 sergiosvieira  staff  238B 27 Jul 22:27 .  
drwxr-xr-x 10 sergiosvieira  staff  340B 27 Jul 16:11 ..  
drwxr-xr-x 15 sergiosvieira  staff  510B 27 Jul  
22:29 .git
```

CLONANDO UM REPOSITÓRIO EXISTENTE

- Caso você queira copiar um repositório Git já existente — por exemplo, um projeto que você queira contribuir — o comando necessário é `git clone`.
- Caso você esteja familiarizado com outros sistemas VCS, tais como Subversion, você perceberá que o comando é `clone` e não `checkout`.
- Cada versão de cada arquivo no histórico do projeto é obtida quando você roda `git clone`.

CLONANDO UM REPOSITÓRIO EXISTENTE

- Você clona um repositório com **git clone [url]**:

```
$ git clone https://github.com/sergiosvieira/  
curso-git.git
```

CLONANDO UM REPOSITÓRIO EXISTENTE

- Isso cria um diretório chamado curso-git e inicializa um subdiretório.git dentro.
- Obtém todos os dados do repositório e verifica a cópia atual da última versão.

CLONANDO UM REPOSITÓRIO EXISTENTE

- Caso você queira clonar o repositório em um diretório diferente de curso-git, é possível especificar esse diretório utilizando a opção abaixo:

```
$ git clone https://github.com/sergiosvieira/curso-git.git meucurso-git
```

GRAVANDO ALTERAÇÕES NO REPOSITÓRIO

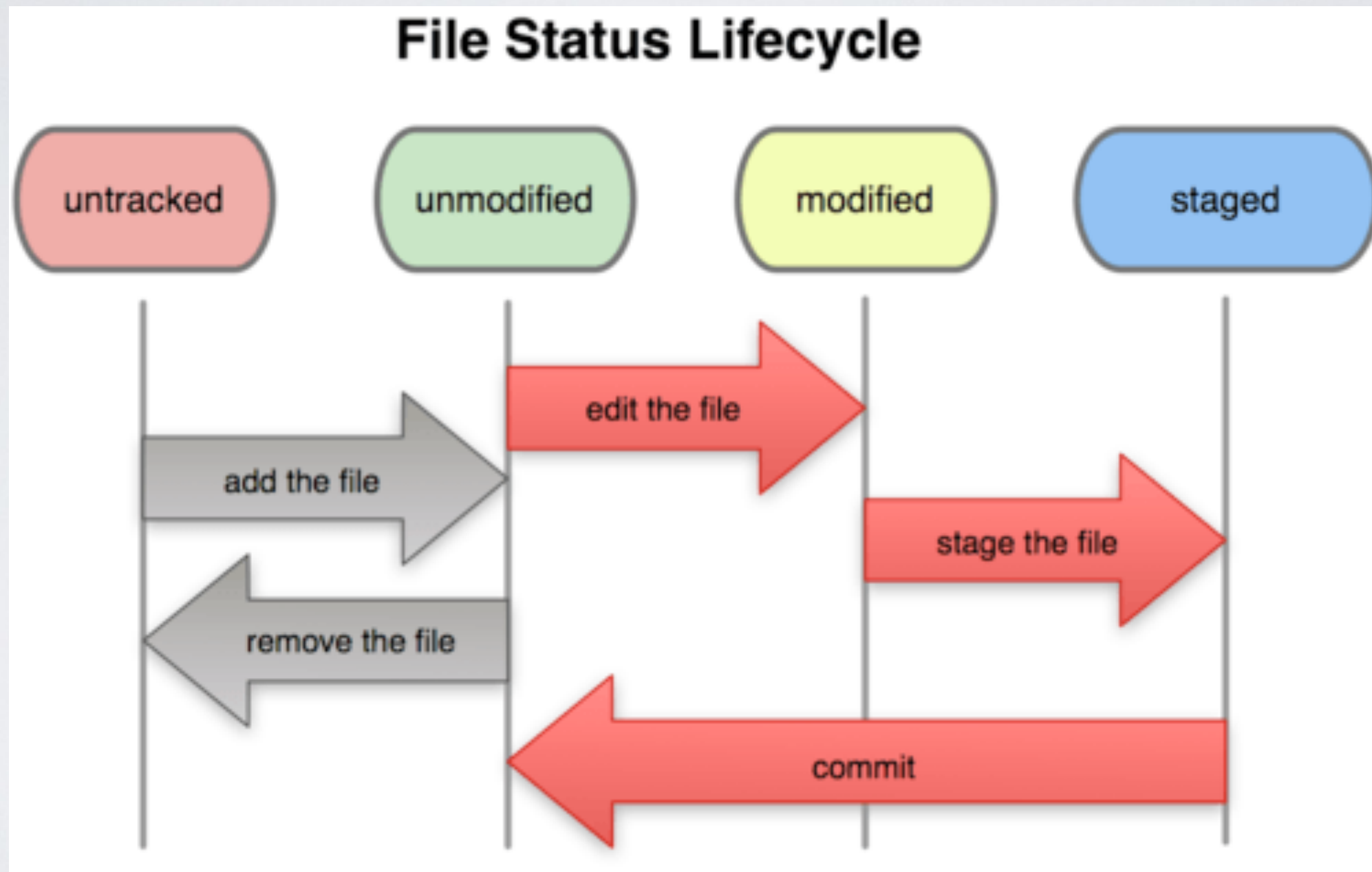


Figura 2-1. O ciclo de vida dos status de seus arquivos.

VERIFICANDO O STATUS DE SEUS ARQUIVOS

- A principal ferramenta utilizada para determinar quais arquivos estão em quais estados é o comando **git status**.
- Se você executar este comando diretamente após uma clonagem, você deverá ver algo similar a isso:

```
$ git status
```

```
# On branch master
```

```
nothing to commit, working directory clean
```

VERIFICANDO O STATUS DE SEUS ARQUIVOS

- Adicione um novo arquivo em seu projeto e execute **git status** novamente.

```
$ touch NEWFILE
```

```
$ git status
```

```
# On branch master
```

```
# Untracked files:
```

```
# (use "git add <file>..." to include in what will be committed)
```

```
#
```

```
# NEWFILE
```

```
nothing added to commit but untracked files present (use "git add"  
to track)
```

MONITORANDO NOVOS ARQUIVOS

- Para passar a monitorar um novo arquivo, use o comando git add.

\$ git add NEWFILE

- Rode o comando status novamente:

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# new file: NEWFILE
```

```
#
```

SELECIONANDO ARQUIVOS MODIFICADOS

- Para listar os arquivos e pastas que estão sendo monitorados use:

```
$ git ls-files
```

```
README
```

```
pasta01/arquivo01
```

```
pasta01/arquivo02
```

```
pasta02/arquivo01
```

```
pasta02/arquivo02
```

```
pasta02/arquivo03
```

- Modifique o arquivo README

```
$ echo "Curso de Git" >> README
```


SELECIONANDO ARQUIVOS MODIFICADOS

- Rode novamente `git status` e você verá algo do tipo:

\$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: NEWFILE

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README

SELECIONANDO ARQUIVOS MODIFICADOS

- Rode novamente `git status` e você verá algo do tipo:

\$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

new file: NEWFILE

SELECIONANDO ARQUIVOS MODIFICADOS

- Faça uma nova mudança no arquivo README e rode git status mais uma vez:

```
$ echo "Primeiro dia" >> NEWFILE
```

```
$ git status
```

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

new file: NEWFILE

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: NEWFILE

SELECIONANDO ARQUIVOS MODIFICADOS

- Acontece que o Git seleciona um arquivo exatamente como ele era quando o comando `git add` foi executado.
- Se você fizer o commit agora, a versão do NEWFILE como estava na última vez que você rodou o comando `git add` é que será incluída no commit, não a versão do arquivo que estará no seu diretório de trabalho quando rodar o comando `git commit`.
- Se você modificar um arquivo depois que rodou o comando `git add`, terá de rodar o `git add` de novo para selecionar a última versão do arquivo:

SELECCIONANDO ARCHIVOS MODIFICADOS

```
$ git add README
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# new file:   NEWFILE
```

```
# modified:   README
```

```
#
```

IGNORANDO ARQUIVOS

- Muitas vezes, você terá uma classe de arquivos que não quer que o Git automaticamente adicione ou mostre como arquivos não monitorados.
- Normalmente estes arquivos são gerados automaticamente como arquivos de log ou produzidos pelo seu sistema de build.
- Nestes casos, você pode criar um arquivo contendo uma lista de padrões a serem checados chamado `.gitignore`.

IGNORANDO ARQUIVOS

- Primeiro crie os arquivos de build usando:

```
$ sh make-build-files.sh
```

- Rode git status e verifique a existência de uma nova pasta em untracked files:

```
$ git status
```

- Lista os arquivos criados:

```
$ ls build
```

```
objeto.a  objeto.o
```

IGNORANDO ARQUIVOS

- Crie o arquivo .gitignore e adicione o seguinte padrão:

```
$ echo "build" >> .gitignore
```

- Exemplos de padrões:

- *.o
- *.a
- *.[oa]

IGNORANDO ARQUIVOS

- Existe um site que gera o arquivo gitignore para você.
- Basta especificar o nome da linguagem ou editor para que ele crie a lista de arquivos que deve ser ignorados.

<http://www.gitignore.io/>

VISUALIZANDO SUAS MUDANÇAS SELECIONADAS E NÃO SELECIONADAS

- O que você alterou, mas ainda não selecionou (stage)?
- E o que você selecionou, que está para ser commitado?
- Apesar do comando `git status` responder essas duas perguntas de maneira geral, o `git diff` mostra as linhas exatas que foram adicionadas e removida

VISUALIZANDO SUAS MUDANÇAS SELECIONADAS E NÃO SELECIONADAS

- Para ver o que você alterou mas ainda não selecionou, digite o comando `git diff` sem nenhum argumento:
- E o que você selecionou, que está para ser commitado?
- `git diff` —cached ou `git diff-staged`

VISUALIZANDO SUAS MUDANÇAS SELECIONADAS E NÃO SELECIONADAS

- `diff --git a/textot.txt b/textot.txt`
`index 9c15ed1..574c337 100644`
`--- a/textot.txt`
`+++ b/textot.txt`
`@@ -1 +1,2 @@`
`linha 01`
`+linha2`

CONSOLIDADO AS MODIFICAÇÕES

- Agora que a sua área de seleção está do jeito que você quer, você pode fazer o commit de suas mudanças.
- Lembre-se que tudo aquilo que ainda não foi selecionado — qualquer arquivo que você criou ou modificou que você não tenha rodado o comando git add desde que editou — não fará parte deste commit.

\$ git commit

CONSOLIDANDO AS MODIFICAÇÕES

- Você pode ver que a mensagem default do commit contém a última saída do comando `git status` comentada e uma linha vazia no início.
- Para um lembrete ainda mais explícito do que foi modificado, você pode passar a opção `-v` para o `git commit`.
- Ao fazer isso, aparecerá a diferença (diff) da sua mudança no editor para que possa ver exatamente o que foi feito.

\$ git commit -v

CONSOLIDANDO AS MODIFICAÇÕES

- Alternativamente, você pode digitar sua mensagem de commit junto ao comando commit ao especificá-la após a flag -m, assim:

```
$ git commit -m "Adiciona linha 01 no arquivo  
01"
```

```
[master c9e762f] Adiciona linha 01 no arquivo 01  
1 file changed, 1 insertion(+)
```

PULANDO A ÁREA DE SELEÇÃO

- Se você quiser pular a área de seleção, o Git provê um atalho simples. Informar a opção -a ao comando git commit faz com que o Git selecione automaticamente cada arquivo que está sendo monitorado antes de realizar o commit, permitindo que você pule a parte do git add

```
$ git commit -a -m 'added new benchmarks'
```


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 remova-me
```

REMOVENDO ARQUIVOS

- Outra coisa útil que você pode querer fazer é manter o arquivo no seu diretório, mas apagá-lo da sua área de seleção.
- Em outras palavras, você quer manter o arquivo no seu disco rígido mas não quer que o Git o monitore mais.
- Isso é particularmente útil se você esqueceu de adicionar alguma coisa no seu arquivo `.gitignore` e acidentalmente o adicionou, como um grande arquivo de log ou muitos arquivos `.a` compilados.
- Para fazer isso, use a opção `--cached`:

```
$ git rm --cached remova-me
```

REMOVENDO ARQUIVOS

- Você pode passar arquivos, diretórios, e padrões de nomes de arquivos para o comando `git rm`. Isso significa que você pode fazer coisas como:

```
$ git rm pasta/*.log
```

MOVENDO ARQUIVOS

- Diferente de muitos sistemas VCS, o Git não monitora explicitamente arquivos movidos.
- Se você renomeia um arquivo, nenhum metadado é armazenado no Git que identifique que você renomeou o arquivo.
- No entanto, o Git é inteligente e tenta descobrir isso depois do fato.

MOVENDO ARQUIVOS

- 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 nome_atual novo_nome
```

MOVENDO ARQUIVOS

- 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 nome_atual novo_nome
```

VISUALIZANDO O HISTÓRICO DE COMMITS



VISUALIZANDO O 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 ferramenta mais básica e poderosa para fazer isso é o comando `git log`.

```
$ git clone https://github.com/zedapp/zed.git
```

```
$ git log
```


VISUALIZANDO O 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 ferramenta mais básica e poderosa para fazer isso é o comando `git log`.

```
$ git clone https://github.com/zedapp/zed.git
```

```
$ git log
```

VISUALIZANDO O HISTÓRICO DE COMMITS

- Por padrão, sem argumentos, `git log` lista os commits feitos naquele repositório em ordem cronológica reversa. Isto é, os commits mais recentes primeiro.
- Como você pode ver, este comando lista cada commit com seu checksum SHA-1, o nome e e-mail do autor, a data e a mensagem do commit.

VISUALIZANDO O HISTÓRICO DE COMMITS

commit

e7ee713fcb97d6ff16c168d0e7f6079c334d17cc

Merge: a1e22b9 4c31212

Author: Zef Hemel <zef@zef.me>

Date: Fri Aug 22 18:41:36 2014 +0200

Merge pull request #438 from

TheKiteEatingTree/scrollbars

Toggle Native Scroll Bars

VISUALIZANDO O HISTÓRICO DE COMMITS

- Um grande número e variedade de opções para o comando `git log` estão disponíveis para mostrá-lo exatamente o que você quer ver.
- Uma das opções mais úteis é `-p`, que mostra o diff introduzido em cada commit. Você pode ainda usar `-2`, que limita a saída somente às duas últimas entradas.

```
$ git log -p -2
```


VISUALIZANDO O HISTÓRICO DE COMMITS

- Se você quiser ver algumas estatísticas abreviadas para cada commit, você pode usar a opção **—stat**.
- A opção `--stat` imprime abaixo de cada commit uma lista de arquivos modificados, quantos arquivos foram modificados, e quantas linhas nestes arquivos foram adicionadas e removidas. Ele ainda mostra um resumo destas informações no final.

```
$ git log —stat
```

VISUALIZANDO O HISTÓRICO DE COMMITS

- Se você quiser ver algumas estatísticas abreviadas para cada commit, você pode usar a opção **—stat**.
- A opção `--stat` imprime abaixo de cada commit uma lista de arquivos modificados, quantos arquivos foram modificados, e quantas linhas nestes arquivos foram adicionadas e removidas. Ele ainda mostra um resumo destas informações no final.

```
$ git log —stat
```

VISUALIZANDO O HISTÓRICO DE COMMITS

- Outra opção realmente útil é **--pretty**.
- Esta opção muda a saída do log para outro formato que não o padrão.

\$ git log --stat

VISUALIZANDO O HISTÓRICO DE COMMITS

- Outra opção realmente útil é `--pretty`.
- Esta opção muda a saída do log para outro formato que não o padrão.

```
$ git log --pretty=oneline
```

```
$ git log --pretty=short
```

```
$ git log --pretty=full
```

```
$ git log --pretty=fuller
```


VISUALIZANDO O HISTÓRICO DE COMMITS

- A opção mais interessante é `format`, que permite que você especifique seu próprio formato de saída do log.
- Isto é especialmente útil quando você está gerando saída para análise automatizada (parsing) — porque você especifica o formato explicitamente, você sabe que ele não vai mudar junto com as atualizações do Git.

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

```
ca82a6d - Scott Chacon, 11 months ago : changed the verison number
```

```
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
```

```
a11bef0 - Scott Chacon, 11 months ago : first commit
```

VISUALIZANDO O HISTÓRICO DE COMMITS

%an	Nome do autor
%h	Hash do commit abreviado
%ar	Data do autor, relativa
%s	Assunto
	<u>http://git-scm.com/book/pt-br/Git-Essencial-Visualizando-o-Histórico-de-Commits</u>

LIMITANDO A SAÍDA DE LOG

- Você pode fazer `-<n>`, onde `n` é qualquer inteiro para mostrar os últimos `n` commits.
- Existem opções de limites de tempo como `--since` e `—until`.
- Por exemplo, este comando pega a lista de commits feitos nas últimas duas semanas:

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

```
$ git log --since=14.07.2014
```

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

LIMITANDO A SAÍDA DE LOG

- Existem outras opções como exibido na tabela abaixo:

--since, --after	Limita aos commits feitos depois da data especificada.
--until, --before	Limita aos commits feitos antes da data especificada.
--author	Somente mostra commits que o autor casa com a string especificada.

LIMITANDO A SAÍDA DE LOG

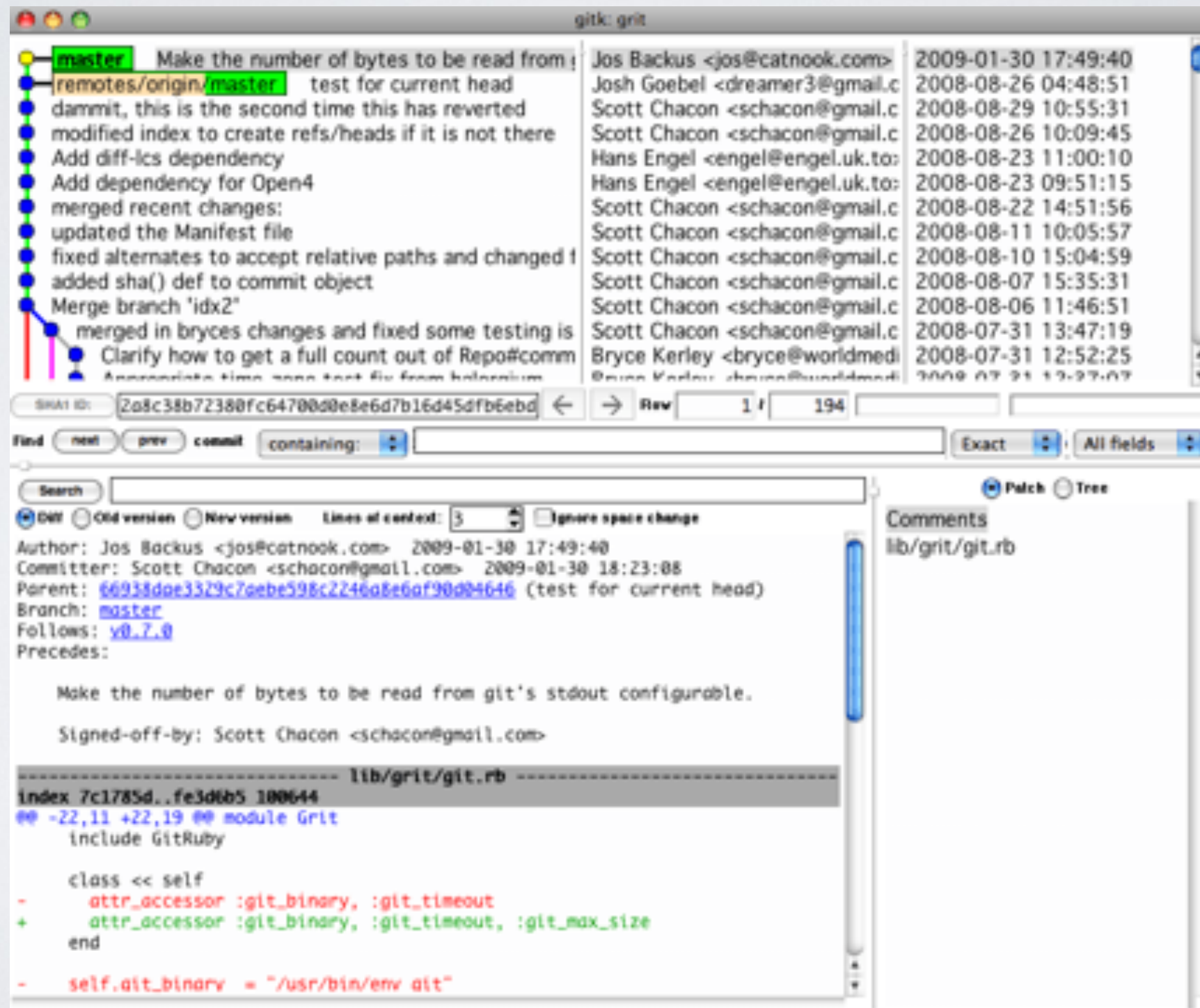
- Se você quer ver quais commits modificaram arquivos de teste no histórico do código fonte do Git que foram commitados por Gist em Outubro de 2008, e não foram merges, você pode executar algo como:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" --  
before="2008-11-01" --no-merges -- test  
5610e3b - Fix testcase failure when extended attribute  
acd3b9e - Enhance hold_lock_file_for_{update,append}()  
f563754 - demonstrate breakage of detached checkout wi  
d1a43f2 - reset --hard/read-tree --reset -u: remove un  
51a94af - Fix "checkout --track -b newbranch" on detac  
b0ad11e - pull: allow "git pull origin $something:$cur
```

USANDO INTERFACE GRÁFICA PARA VISUALIZAR O HISTÓRICO

- Se você quiser usar uma ferramenta gráfica para visualizar seu histórico de commit, você pode querer dar uma olhada em um programa Tcl/Tk chamado gitk que é distribuído com o Git.
- Gitk é basicamente uma ferramenta visual para git log, e ele aceita aproximadamente todas as opções de filtros que git log aceita.

USANDO INTERFACE GRÁFICA PARA VISUALIZAR O HISTÓRICO



DESFAZENDO
COISAS



DESFAZENDO COISAS

- Uma das situações mais comuns para desfazer algo, acontece quando você faz o commit muito cedo e possivelmente esqueceu de adicionar alguns arquivos, ou você bagunçou sua mensagem de commit.
- Se você quiser tentar fazer novamente esse commit, você pode executá-lo com a opção `--amend`:

\$ git commit --amend

DESFAZENDO COISAS

- Esse comando pega sua área de seleção e a utiliza no commit.
- Se você não fez nenhuma modificação desde seu último commit (por exemplo, você rodou esse comando imediatamente após seu commit anterior), seu snapshot será exatamente o mesmo e tudo que você mudou foi sua mensagem de commit.

\$ git commit -m 'initial commit'

\$ git add forgotten_file

\$ git commit --amend

TRABALHANDO COM REMOTOS

- Para ser capaz de colaborar com qualquer projeto no Git, você precisa saber como gerenciar seus repositórios remotos.
- Repositórios remotos são versões do seu projeto que estão hospedados na Internet ou em uma rede em algum lugar.

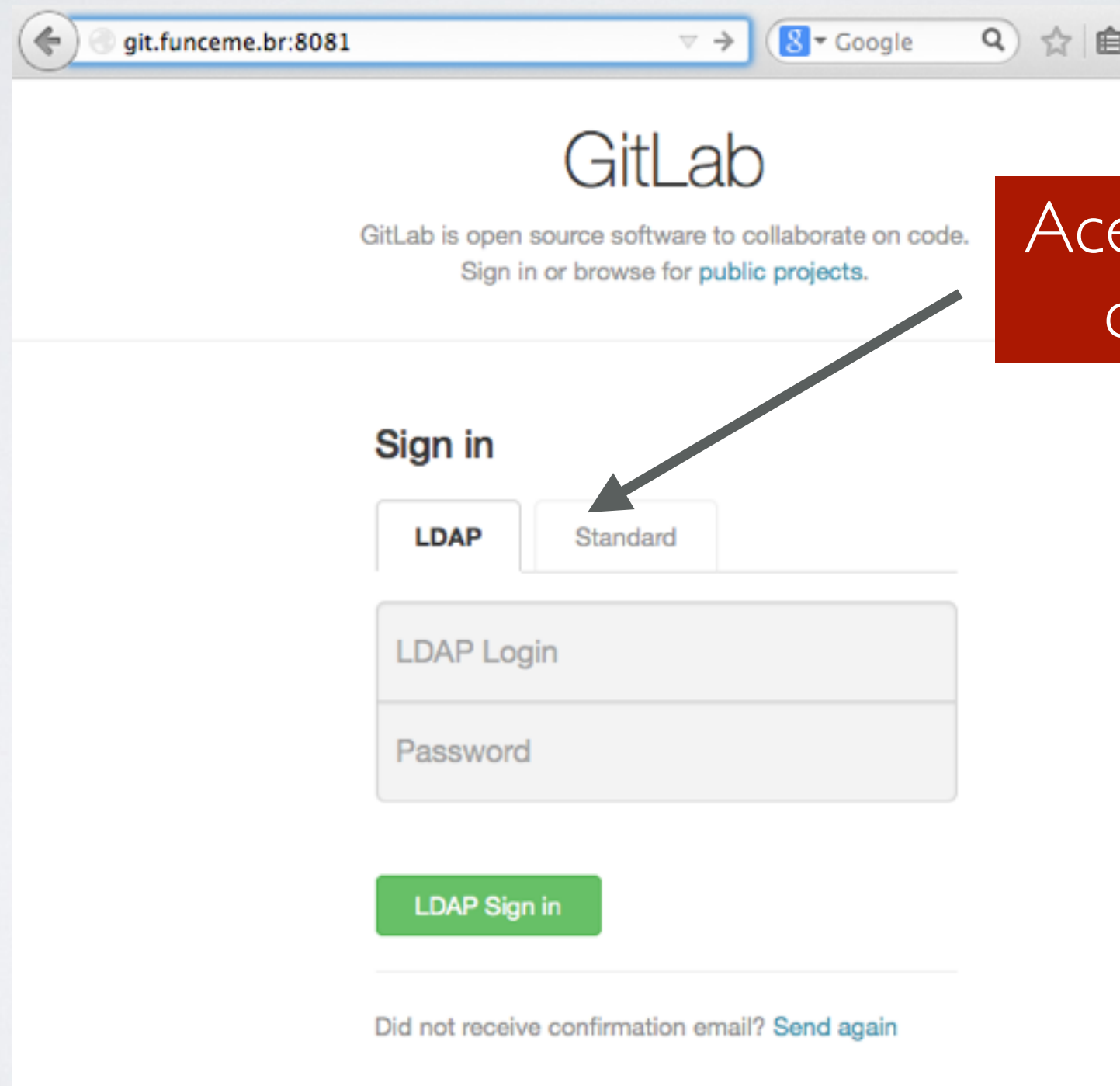
TRABALHANDO COM REMOTOS

- Acesso interno ao GitLab
 - <http://eris.funceme.br:8081>
- Acesso externo ao GitLab
 - <http://git.funceme.br:8081>

SEU PRIMEIRO REPOSITÓRIO REMOTO

1. Acesse sua conta
2. + New Project
3. Preencha as informações do projeto (nome, descrição e visibilidade)
4. Configure seu repositório local com o repositório remoto que você acabou de criar.

SEU PRIMEIRO REPOSITÓRIO REMOTO

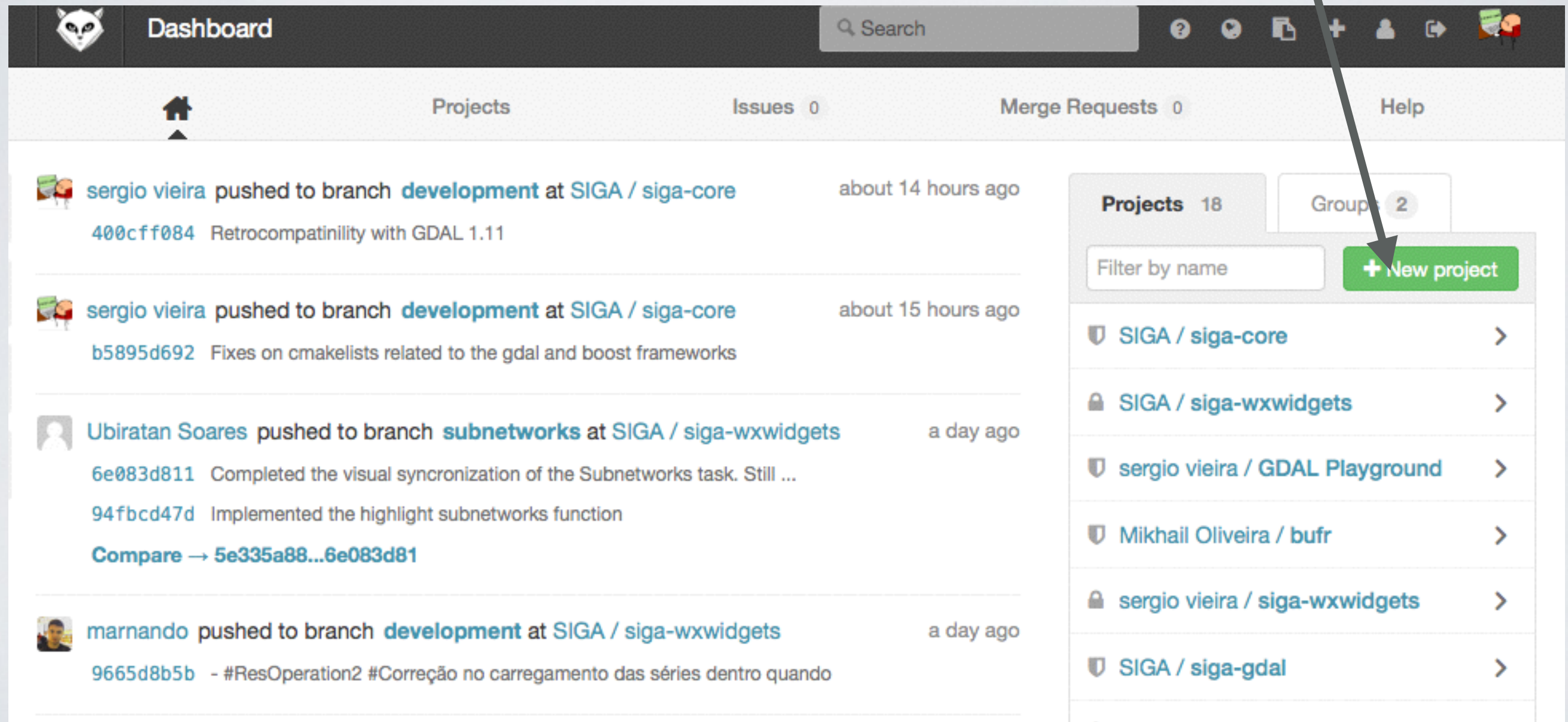


The screenshot shows a web browser window with the address bar displaying 'git.funceme.br:8081'. The page title is 'GitLab'. Below the title, it says 'GitLab is open source software to collaborate on code. Sign in or browse for [public projects](#).' The 'Sign in' section has two tabs: 'LDAP' and 'Standard'. The 'LDAP' tab is selected. Below the tabs, there are two input fields: 'LDAP Login' and 'Password'. A green button labeled 'LDAP Sign in' is below the input fields. At the bottom, there is a link: 'Did not receive confirmation email? [Send again](#)'.

Acesse sua
conta

SEU PRIMEIRO REPOSITÓRIO REMOTO

Crie um Novo Projeto



The screenshot shows the SIGA dashboard interface. At the top, there's a dark header with a cat logo, the word "Dashboard", a search bar, and several icons. Below the header, there's a navigation bar with tabs for "Projects", "Issues 0", "Merge Requests 0", and "Help". The main content area is divided into two sections. On the left, there's a list of recent pushes. On the right, there's a sidebar with a "Projects" tab and a "Groups" tab. The "Projects" tab is active, showing a list of projects. A red box with the text "Crie um Novo Projeto" and an arrow points to the "+ New project" button in the sidebar.

Dashboard Search ? ? ? ? ? ? ? ?

Projects Issues 0 Merge Requests 0 Help

Recent Pushes:

- sergio vieira** pushed to branch **development** at **SIGA / siga-core** about 14 hours ago
400cff084 Retrocompatinility with GDAL 1.11
- sergio vieira** pushed to branch **development** at **SIGA / siga-core** about 15 hours ago
b5895d692 Fixes on cmake lists related to the gdal and boost frameworks
- Ubiratan Soares** pushed to branch **subnetworks** at **SIGA / siga-wxwidgets** a day ago
6e083d811 Completed the visual sincronization of the Subnetworks task. Still ...
94fbcd47d Implemented the highlight subnetworks function
Compare → 5e335a88...6e083d81
- marnando** pushed to branch **development** at **SIGA / siga-wxwidgets** a day ago
9665d8b5b - #ResOperation2 #Correção no carregamento das séries dentro quando

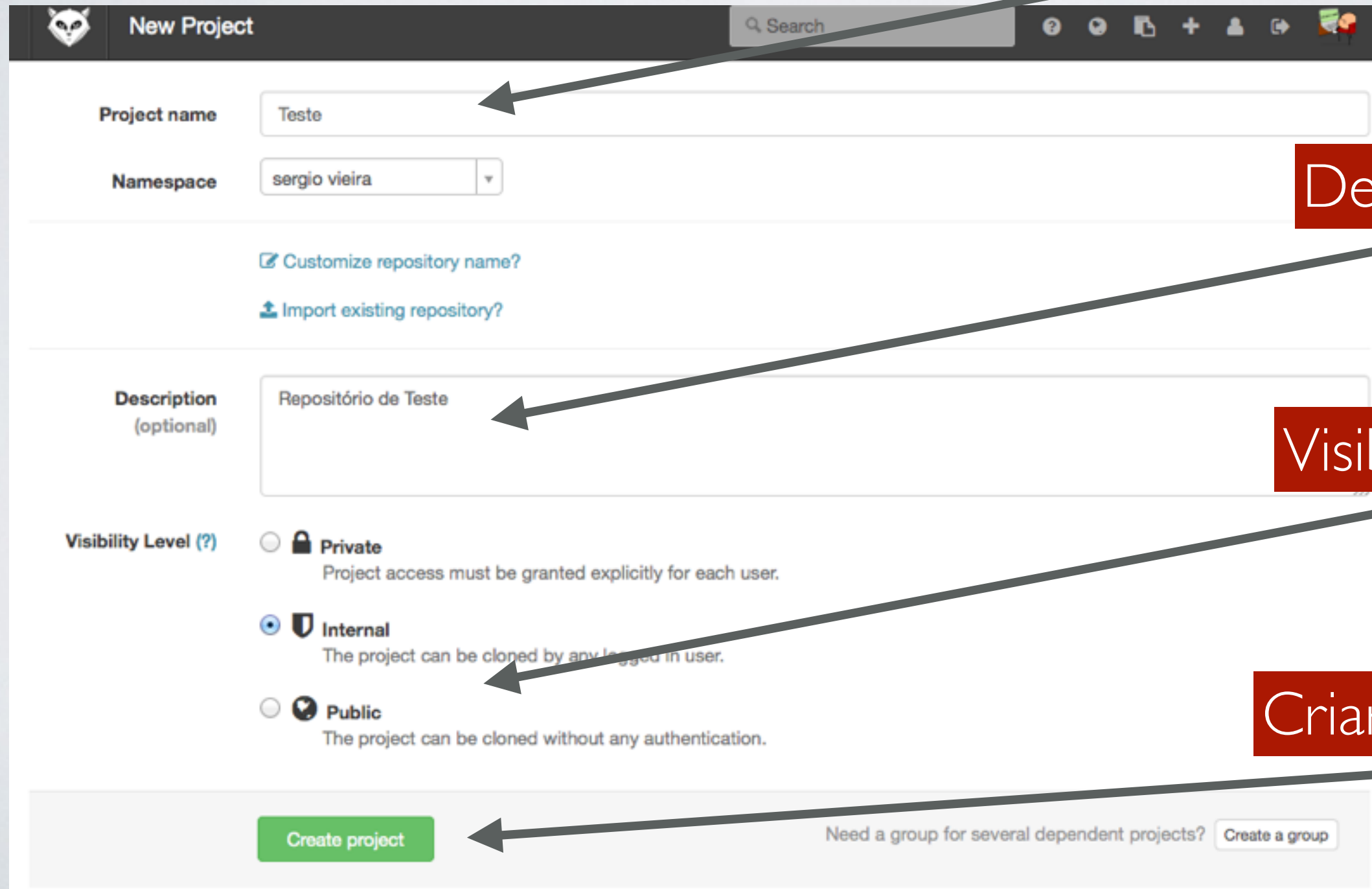
Projects 18 **Groups** 2

Filter by name **+ New project**

- SIGA / siga-core** >
- SIGA / siga-wxwidgets** >
- sergio vieira / GDAL Playground** >
- Mikhail Oliveira / bufr** >
- sergio vieira / siga-wxwidgets** >
- SIGA / siga-gdal** >

SEU PRIMEIRO REPOSITÓRIO REMOTO

Nome do Projeto



The screenshot shows the GitHub 'New Project' form. It includes a header with the GitHub logo, 'New Project' text, and a search bar. The form fields are: 'Project name' (containing 'Teste'), 'Namespace' (containing 'sergio vieira'), 'Description (optional)' (containing 'Repositório de Teste'), and 'Visibility Level' (with radio buttons for 'Private', 'Internal' (selected), and 'Public'). At the bottom is a green 'Create project' button. Annotations with arrows point from text boxes to these specific fields: 'Nome do Projeto' to the Project name field, 'Descrição' to the Description field, 'Visibilidade' to the Internal visibility option, and 'Criar Projeto' to the Create project button.

Project name Teste

Namespace sergio vieira

☒ Customize repository name?

☐ Import existing repository?

Description (optional) Repositório de Teste

Visibility Level (?)

- ☐ **Private**
Project access must be granted explicitly for each user.
- ☒ **Internal**
The project can be cloned by any logged in user.
- ☐ **Public**
The project can be cloned without any authentication.

Create project

Need a group for several dependent projects? [Create a group](#)

Descrição

Visibilidade

Criar Projeto

SEU PRIMEIRO REPOSITÓRIO REMOTO

Suas Configurações Globais de Identificação

sergio vieira / Teste

Repositório de Teste – [Edit](#)

Git global setup:

```
git config --global user.name "sergio vieira"
git config --global user.email "sergiosvieira@gmail.com"
```

Create Repository

```
mkdir teste
cd teste
git init
touch README
git add README
git commit -m 'first commit'
git remote add origin git@eris.funceme.br:sergiosvieira/teste.git
git push -u origin master
```

Existing Git Repo?

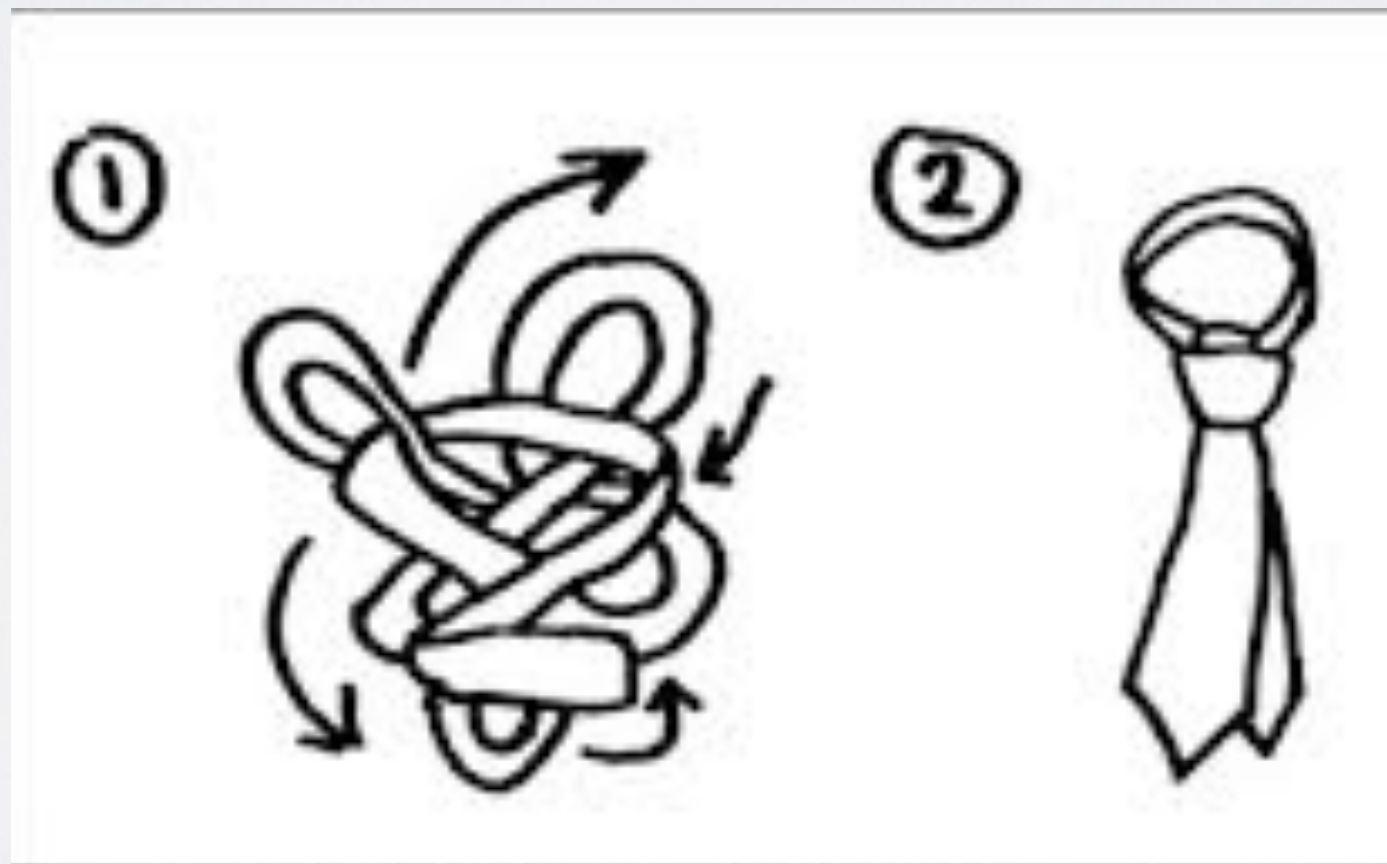
```
cd existing_git_repo
git remote add origin git@eris.funceme.br:sergiosvieira/teste.git
git push -u origin master
```

Criar novo rep. local e configurá-lo com o rep. remoto

Configurar rep. local existente com rep. remoto

TRABALHANDO COM REMOTOS

- Como associar seu repositório local ao remoto?



TRABALHANDO COM REMOTOS

- Como associar seu repositório local ao remoto?
- Acesso Interno

```
$ git remote add origin http://eris.funceme.br:8081/  
seu_nome/teste.git
```

- Acesso Externo

```
$ git remote add origin http://git.funceme.br:8081/  
seu_nome/teste.git
```

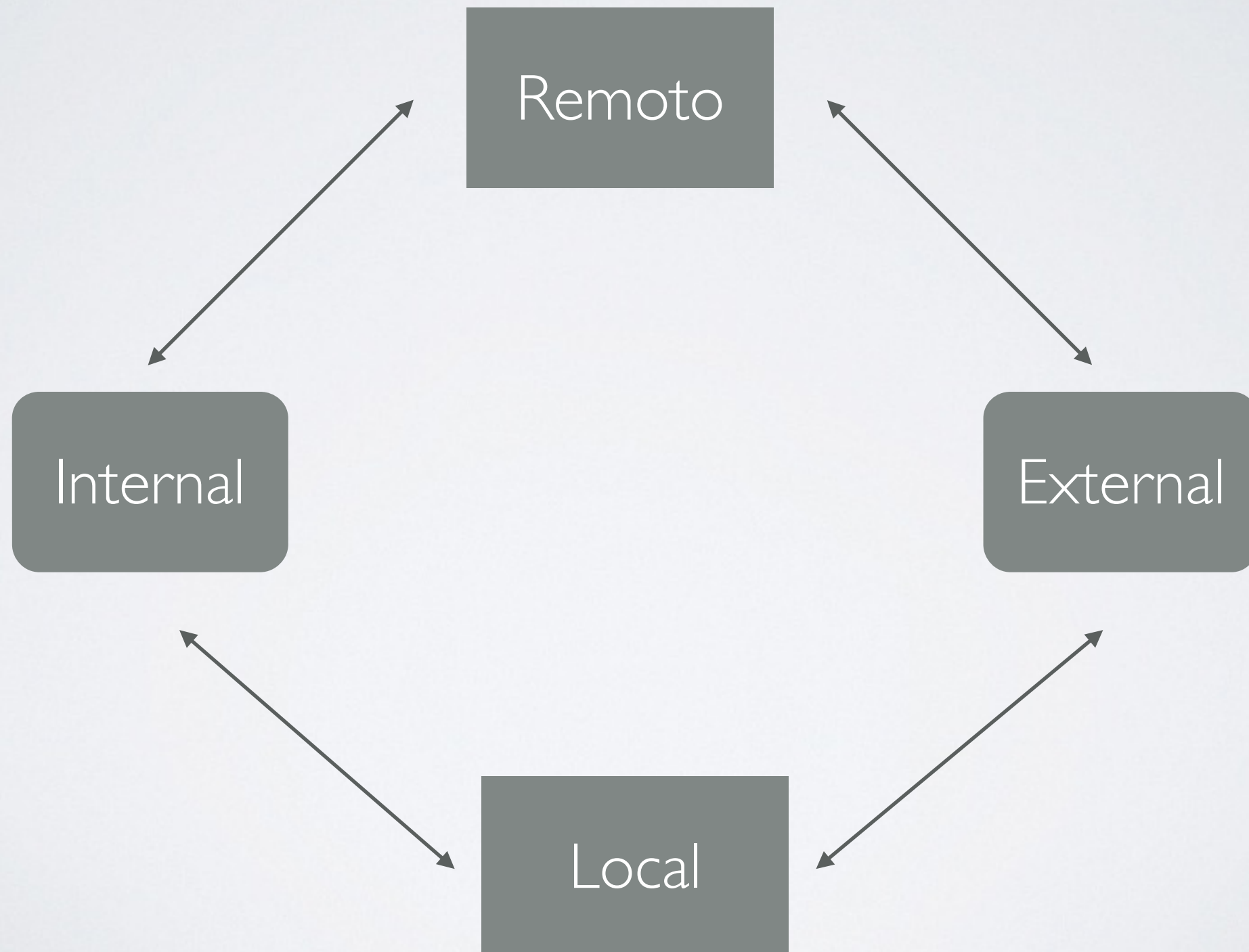
TRABALHANDO COM REMOTOS

- Como associar seu repositório local ao remoto?
- Acesso Interno e Externo

```
$ git remote add internal http://eris.funceme.br:  
8081/seu_nome/teste.git
```

```
$ git remote add external http://git.funceme.br:  
8081/seu_nome/teste.git
```


TRABALHANDO COM REMOTOS



TRABALHANDO COM REMOTOS

Remoto



Acesso



Local

TRABALHANDO COM REMOTOS

- A quais repositórios remotos o meu local está associado?

```
$ git remote -v
```

```
origin https://github.com/sergiosvieira/curso-  
git.git (fetch)
```

```
origin https://github.com/sergiosvieira/curso-  
git.git (push)
```

ADICIONANDO USUÁRIOS AO SEU PROJETO

sergio vieira / Teste

Issues 0 Merge Requests 0 Wiki Settings

SSH HTTP git@eris.funceme.br:sergiosvieira/teste.git

Repositório de Teste – [Edit](#)

Git global setup:

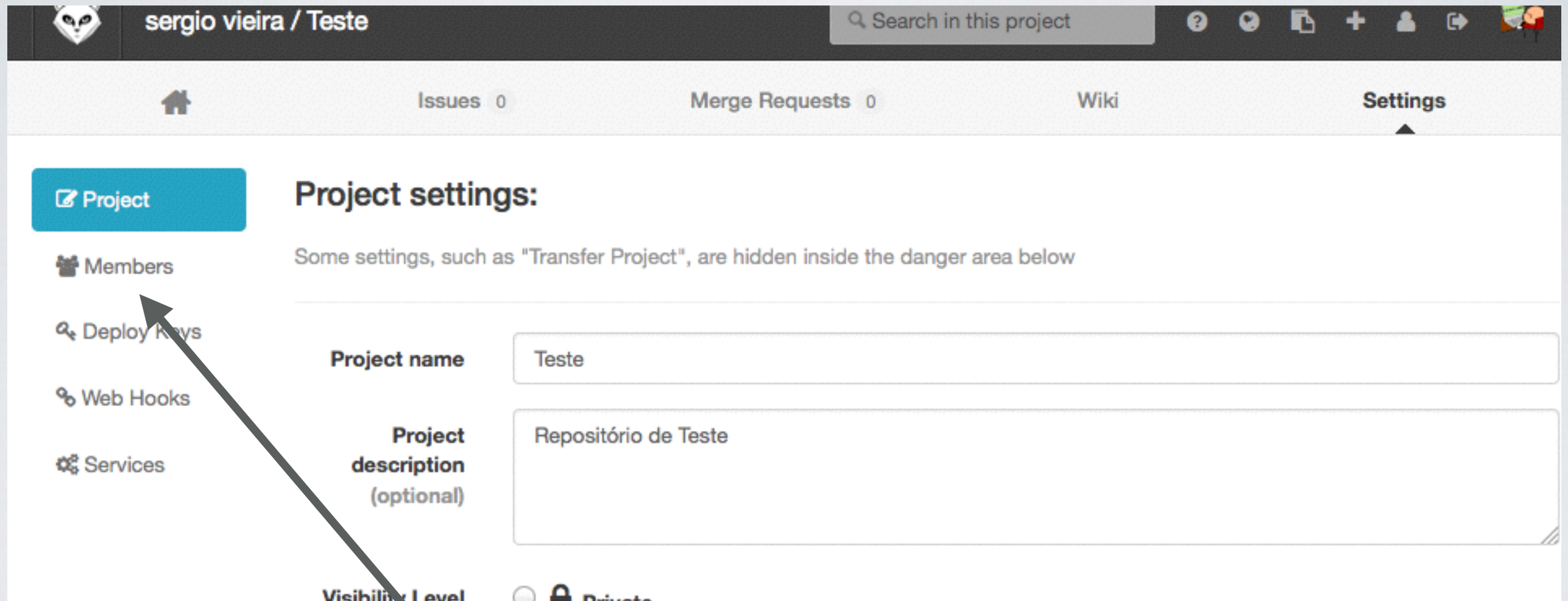
```
git config --global user.name "sergio vieira"
git config --global user.email "sergiosvieira@gmail.com"
```

Create Repository

```
mkdir teste
cd teste
git init
touch README
git add README
git commit -m 'first commit'
git remote add origin git@eris.funceme.br:sergiosvieira/teste.git
git push -u origin master
```

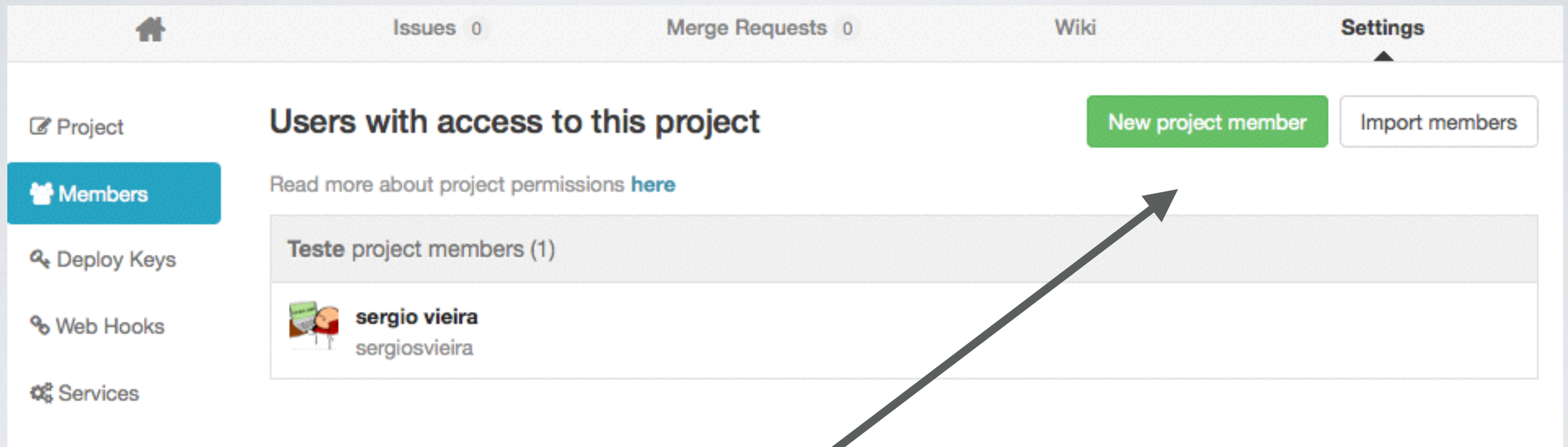
Existing Git Repo?

ADICIONANDO USUÁRIOS AO SEU PROJETO



Membros

ADICIONANDO USUÁRIOS AO SEU PROJETO



Adicionar novo
usuário ao projeto

ADICIONANDO USUÁRIOS AO SEU PROJETO

Sergio Vieira / teste

Search in this project

Issues 0 Merge Requests 0 Wiki Settings

Project

Members

Deploy Keys

Web Hooks

Services

New project member(s)

1. Choose people you want in the project

People

2. Set access level for them

Project Access

- Guest
- Reporter
- Developer
- Master

Cancel

TRABALHO EM EQUIPE



TRABALHO EM EQUIPE - FÁCIL

- Prática
 - Crie e configure um repositório local. (remoto <http://eris.funcene.br:8081/sergiosvieira/teste.git>)
 - Crie um arquivo com seu nome e dentro dele digite seu email.
 - Consolide e mande para o repositório remoto.

BRANCHS E
MERGES

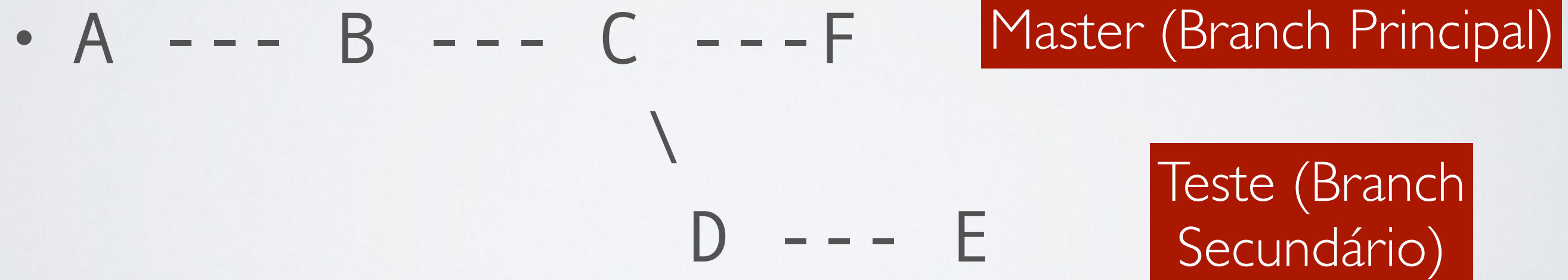


BRANCHS E MERGE

- Geralmente, branches são descritos como sendo uma “linha de desenvolvimento”.
- Melhor, um branch é um "grafo dirigido acíclico de desenvolvimento" ao invés de uma linha.
- Entenda-os como sendo uma espécie de objetos “baratos”, computacionalmente falando.

BRANCHS E MERGE

- Eles nomeiam um determinado commit e seus commits ancestrais.



BRANCHS

- Para criar um branch use:

\$ git branch <nome do branch>

- Para listar os branches criados use:

\$ git branch

- Para trocar de branch use:

\$ git checkout <nome do branch>

- Para criar e trocar de branch com um único comando use:

\$ git checkout -b <nome do branch>

BRANCHS E MERGE

- Suponha que você tenha dois branches.

• A - - - C - - - E

master

\

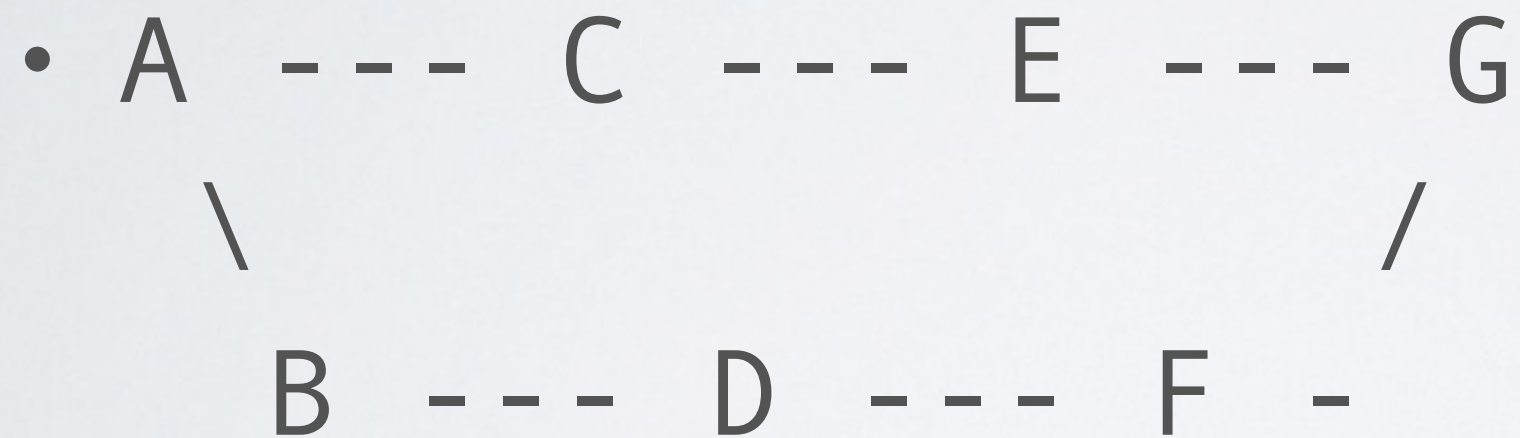
B - - - D - - - F

new-idea

BRANCHS E MERGE

- E queria juntá-los.
- Então use o comando `merge`. Faça:
- `$ git checkout master`
`$ git merge new-idea`

BRANCHS E MERGE

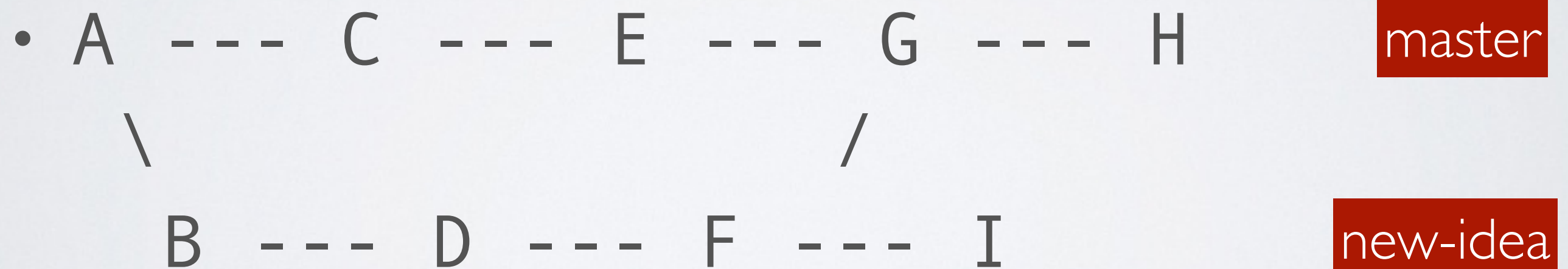


master

new-idea

BRANCHS E MERGE

- Caso você continue consolidando no master e em new-idea.



BRANCHS E MERGE

- Observações:
 - Em branchs diferentes, o git considera apenas as diferenças consolidadas.

BRANCHS E MERGE

- Exemplo

```
$ git branch novo_branch
```

```
$ touch file
```

```
$ git status
```

```
$ git checkout novo_branch
```

```
$ git status
```

Arquivo continua em untracked

BRANCHS E MERGE

- Resolvendo conflitos:

Normal merge conflict for 'novo_branch'

{local}: modified file

{remote}: modified file

**Hit return to start merge resolution tool
(opendiff):**

BRANCHS E MERGE

- Para resolver conflitos utilize:
- **\$ git mergetool**

BRANCHS E MERGE

- Para resolver conflitos utilize:
- **\$ git mergetool**
- Depois de resolvido os conflitos você deve comitar.

BRANCHS E MERGE

- Caso queira enviar um branch local para o remoto utilize:

\$ git pull origin novo_branch

- Caso queira criar um criar um branch local a partir de um remoto use:

\$ git fetch

\$ git checkout —track origin/nome_do_branch

BRANCHS E MERGE

- Para atualizar um branch específico faça:

\$ git checkout nome_do_branch

\$ git pull origin nome_do_branch

BRANCHS E MERGE

- Para apagar um branch remoto faça:
\$ git push origin :nome_do_branch

TAGGING



TAGGING

- Assim como a maioria dos VCS's, 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, por exemplo).

TAGGING

- Git têm dois tipos principais de tags: leve e anotada.
- Um tag leve é muito similar a uma branch que não muda — é um ponteiro para um commit específico.
- Tags anotadas, entretanto, são armazenadas como objetos inteiros no banco de dados do Git.
- Eles possuem uma chave de verificação; o nome da pessoa que criou a tag, email e data; uma mensagem relativa à tag; e podem ser assinadas e verificadas com o GNU Privacy Guard (GPG).

TAGGING

- Para criar uma tag anotada basta fazer:

```
$ git tag -a v1.0 -m “Versão 1.0”
```

- Para listar suas tags basta fazer:

```
$ git tag
```

- Para ver informações sobre a tag:

```
$ git show v1.0
```

TAGGING

- Para remover tags use:

```
$ git tag -d v1.0
```

- Para mudar o seu trabalho para a versão 1.0 use:

```
$ git checkout v1.0 -b versão 1.0
```

TAGGING

- Caso você queira criar um tag a partir de commit específico faça:

```
$ git tag -a v1.0.1
```

```
f157d90db48469e0fb2d4e0018681a634c0974
```

```
69
```

TAGGING

- Por padrão, o comando `git push` não transfere tags para os servidores remotos.
- Você deve enviar as tags explicitamente para um servidor compartilhado após tê-las criado.

TAGGING

- Para enviar uma tag específica use:
\$ git push origin v1.0
- Para enviar todas várias tags ao mesmo tempo use:
\$ git push origin --tags

REBASING

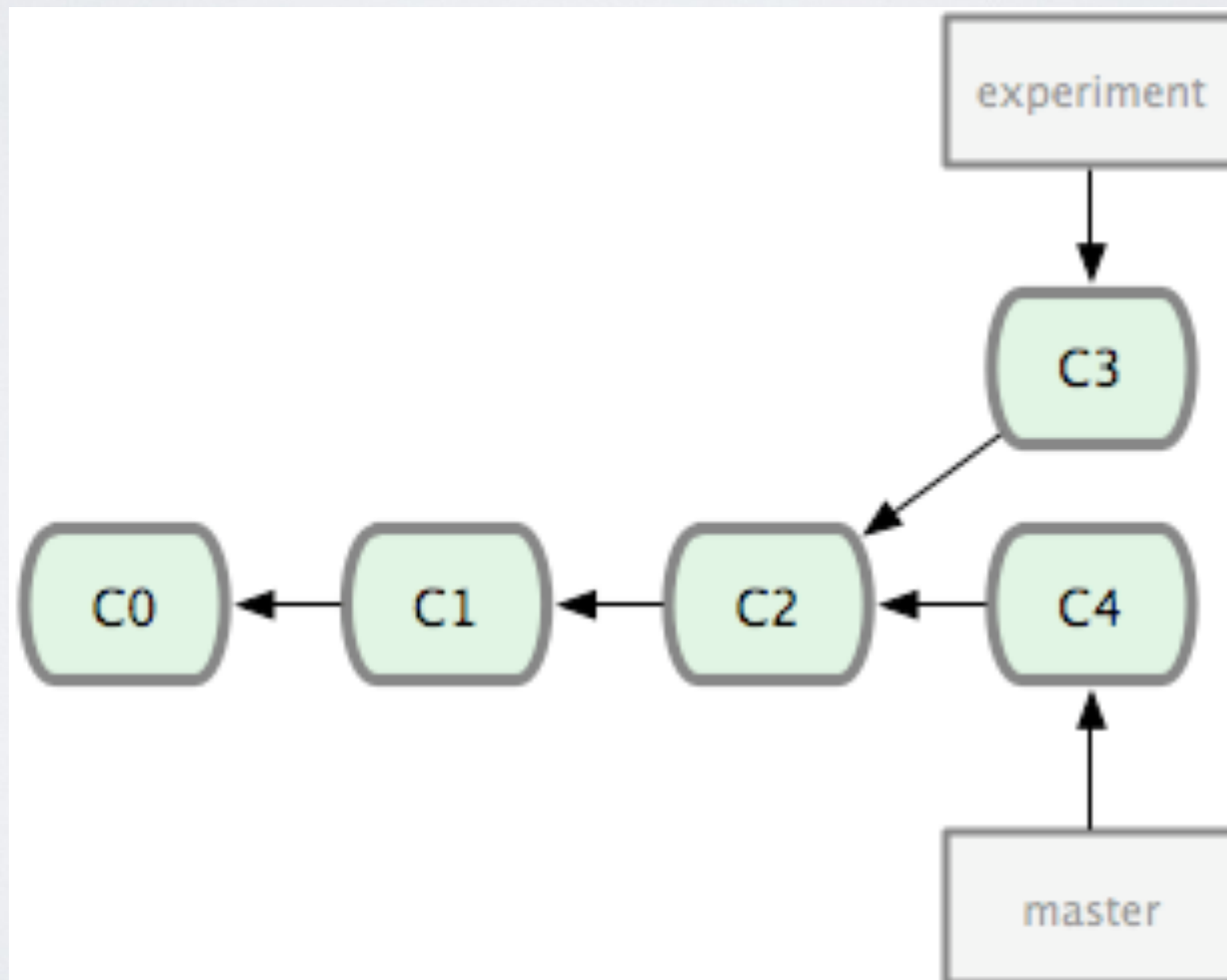


REBASING

- No Git, existem duas maneiras principais de integrar mudanças de um branch em outro: o merge e o rebase.

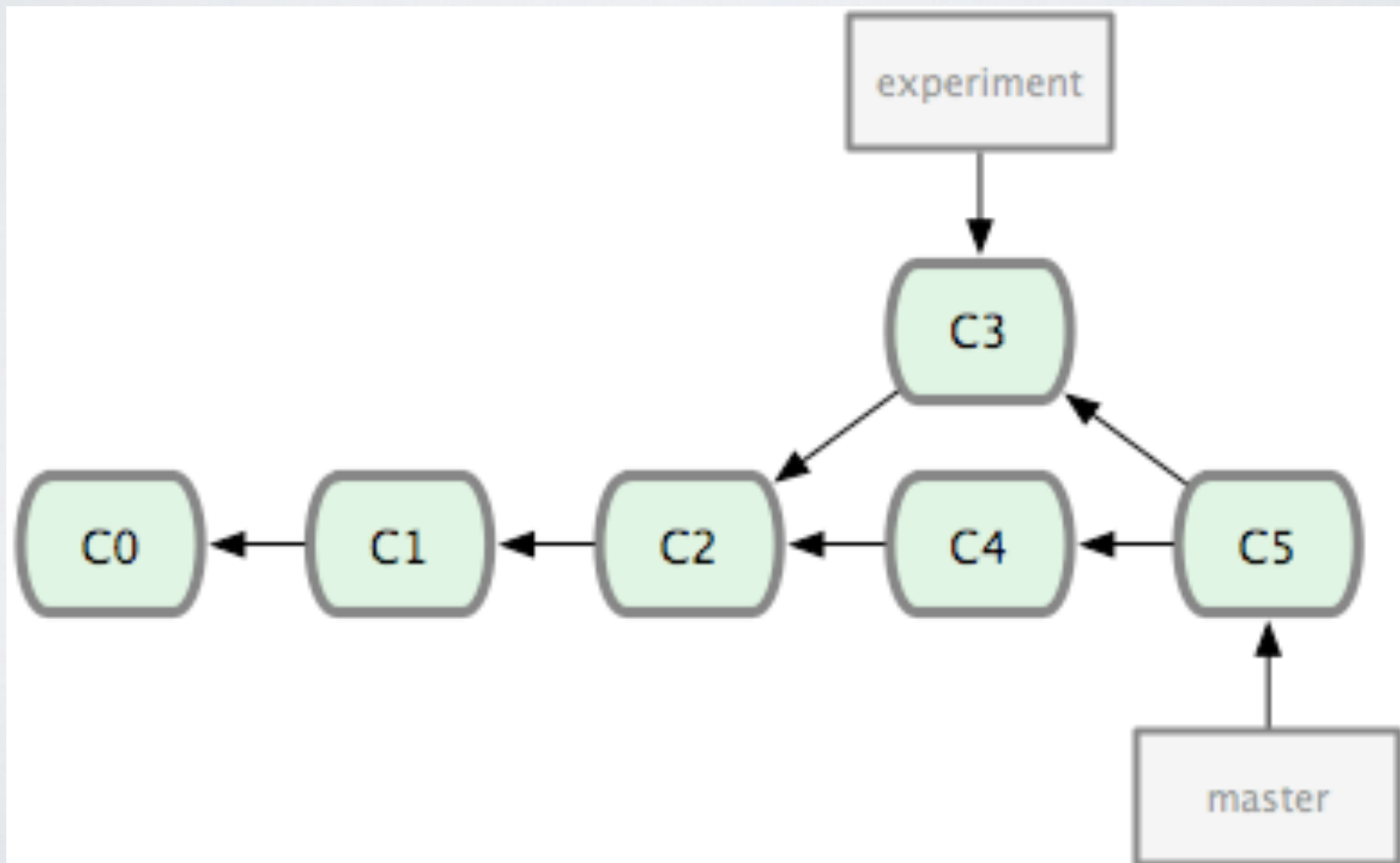
REBASING

- Merge



REBASING

- Merge



REBASING

- Porém, existe outro modo: você pode pegar o trecho da mudança que foi introduzido em C3 e reaplicá-lo em cima do C4. No Git, isso é chamado de rebasing.
- Com o comando rebase, você pode pegar todas as mudanças que foram commitadas em um branch e replicá-las em outro.

REBASING

- Ele vai ao ancestral comum dos dois branches (no que você está e no qual será feito o rebase).
- Pega a diferença (diff) de cada commit do branch que você está.
- Salva elas em um arquivo temporário, restaura o branch atual para o mesmo commit do branch que está sendo feito o rebase e, finalmente, aplica uma mudança de cada vez.

REBASING

- Para usar o rebase faça:

\$ git checkout experiment

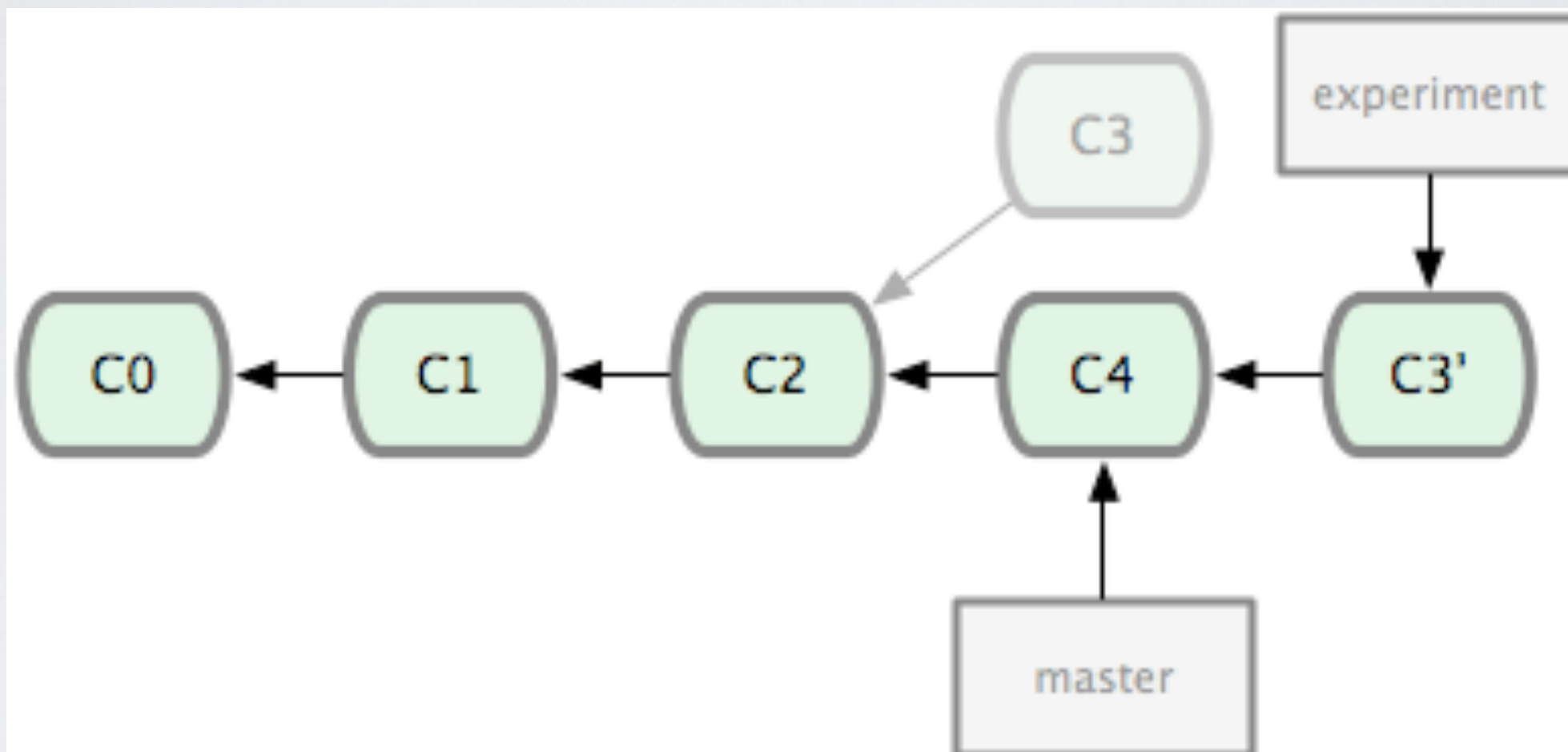
\$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

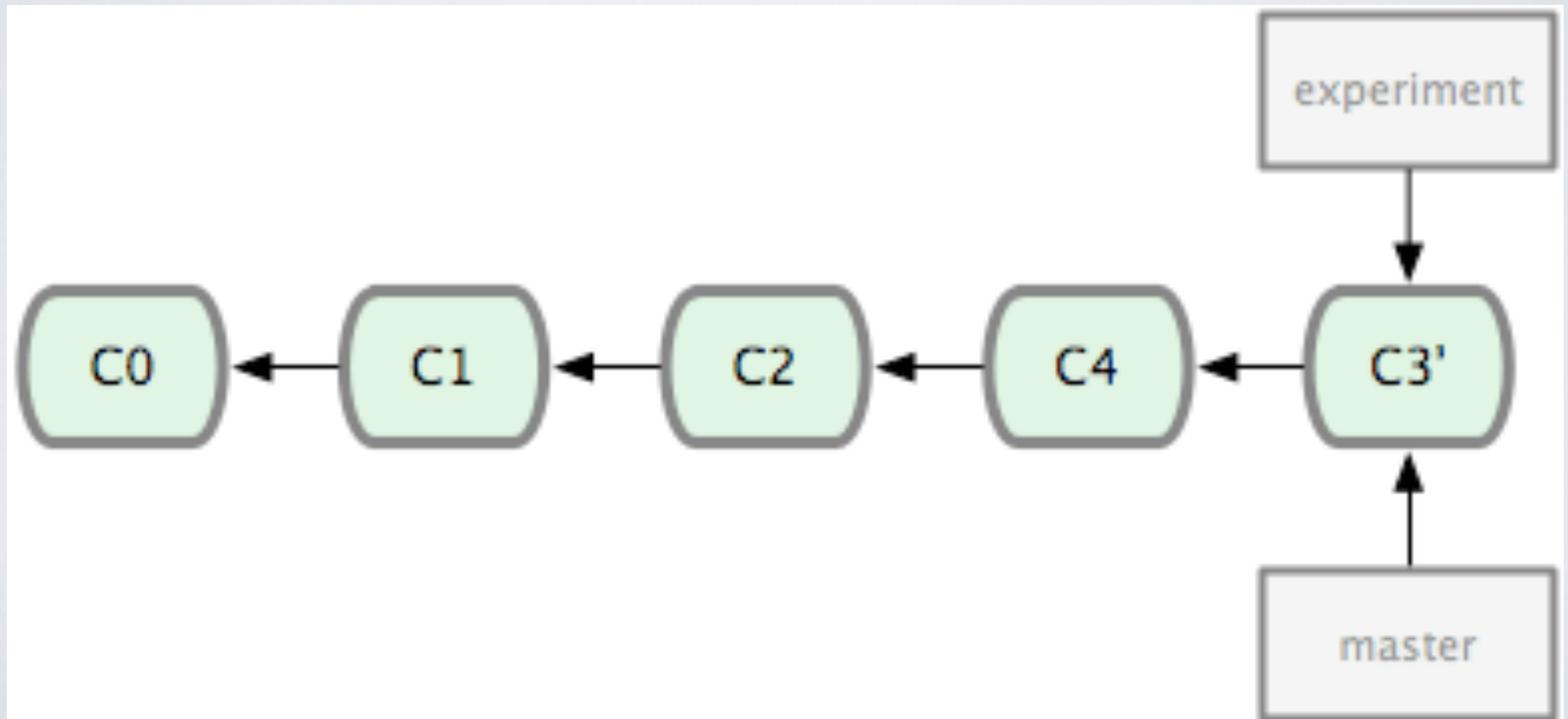
REBASING

- Rebase



REBASING

- Rebase



REBASING

- Agora, o snapshot apontado por C3' é exatamente o mesmo apontado por C5 no exemplo do merge.
- Não há diferença no produto final dessas integrações, mas o rebase monta um histórico mais limpo.
- Se você examinar um log de um branch com rebase, ele parece um histórico linear: como se todo o trabalho tivesse sido feito em série, mesmo que originalmente tenha sido feito em paralelo.