

```
v2.6.0-beta.2 build: release 2.6.0-beta.2
build: build 2.6.0-beta.2
feat: dynamic directive arguments for v-on, v-bind and custom directives (#9371)
  origin/dynamic-directive-arguments | feat: dynamic args for custom directives
perf: improve scoped slots change detection accuracy (#9371)
test: test cases for v-on/v-bind dynamic arguments
refactor: v-bind dynamic arguments use bind helper
test: fix tests, resolve helper conflict
fix: fix middle modifier
feat: handle dynamic argument for v-bind.sync
  origin/slot-optimization | perf: improve scoped slots change detection
feat: dynamic directive arguments for v-bind and v-on
refactor: extend dom-props update skip to more all keys except value
fix: handle event edge case in Firefox
```

# Documentação do Git

Feito por: Sabrina Caldas Berno

```
git help
```

nesse comando, você recebe toda a documentação do git te auxiliando no que você precisar

ex: `git help - - config`

---

```
git config
```

- `git config user.email`

o e-mail é propriedade filho do bloco de configuração do usuário. Isso retorna o endereço de e-mail configurado que o Git associa com commits criados no local.

- `git config --global user.name "meu_nome"`
- `- - local`

Por padrão, o `git config` grava em nível local

Os valores de configuração local são armazenados em um arquivo que pode ser encontrado no diretório .git do repositório: `.git/config`

- - - global

Os valores de configuração global são armazenados em um arquivo localizado no diretório de base do usuário. `~/.gitconfig` em sistemas Unix

- - - system

arquivo de configuração de nível do sistema está no arquivo `gitconfig` localizado fora do caminho raiz do sistema. `$(prefix)/etc/gitconfig` em sistemas Unix.

- gravando valores

```
git config --global user.email "your_email@example.com"
```

- saídas coloridas

```
$ git config --global color.ui false
```

padrão = auto

É possível definir o valor de `color.ui` como `always`, o que aplica a saída do código de cores ao redirecionar o fluxo de saída para arquivos ou pipes. Essa ação pode, sem querer, causar problemas, já que o pipe receptor pode não estar esperando a entrada codificada com cores.

Elas também podem ter valores de cor específicos definidos. Alguns exemplos de valores de cor suportados são: `normal`, `black`, `vermelho`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white`,

— `color.branch`

cor de saída do comando Git branch

— `color.branch.<slot>`

current: a ramificação atual

local: uma ramificação local

remote: uma ramificação remota em refs/remotes

upstream: uma ramificação de rastreamento upstream

plain: qualquer outra ref

```
— color.diff
```

cores na saída do git diff, git log e git show

VERIFICAÇÃO: para saber qual é a configuração do seu git, digite `git config --list`

- Aliases

são atalhos personalizados que definem qual comando é expandido em comandos mais longos ou combinados. O caso de uso comum para aliases do Git é fazer atalhos para o comando de commit. Os aliases do Git são armazenados nos arquivos de configuração.

```
git config --global alias.ci commit
```

Esse exemplo cria o alias ci para o comando `git commit`. Você pode então chamar o git commit

executando o git ci. Os aliases também podem fazer referências a outros aliases para criar combinações poderosas.

---

```
git init
```

Inicializa repositório git. O diretório usado para a criação não precisa necessariamente estar vazio, pode existir arquivos antigos.

- um repositório em um diretório local já existente

você deve ir para esse diretório

```
$ cd /home/user/your_repository
```

e depois você inicia o `git init`

a partir desse ponto, você já pode dar um git add seguido de um commit

- clonando um repositório existente

Caso você queira obter a cópia de um repositório Git existente – por exemplo, um projeto que você queira contribuir – o comando para isso é `git clone`

. você clona um repositório com `git clone [url]`

```
$ git clone https://github.com/libgit2/libgit2
```

sso cria um diretório chamado libgit2, inicializa um diretório .git dentro dele, recebe todos os dados deste repositório e deixa disponível para trabalho a cópia da última versão

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Este comando faz exatamente a mesma coisa que o anterior, mas o diretório de destino será chamado mylibgit (será com o nome diferente)

```
git status
```

Verifica quais os estados que os arquivos estão. Se existe algum arquivo modificado, adiciona ou removido. Além disso, caso esteja rastreado com uma branch no servidor, verifica se a versão está a frente ou atrás da versão do servidor.

Atente para o parâmetro `-s` no final do atalho criado, essa opção permite um status resumido, resultando em maior dinamismo.

. Se você executar esse comando imediatamente após clonar um repositório, você vai ver algo assim:

```
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
nothing to commit, working directory clean
```

. Digamos que você adiciona um novo arquivo no seu projeto, um simples arquivo README.

Se o arquivo não existia antes, e você executar `git status`, você verá seu arquivo não rastreado da seguinte forma:

```
$ echo 'My Project' > README  
$ git status  
On branch master  
Your branch is up-to-date with 'origin/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)

. "Não rastreado" basicamente significa que o Git vê um arquivo que você não tinha no snapshot (commit) anterior; o Git não vai passar a incluir o arquivo nos seus commits a não ser que você o mande fazer isso explicitamente. Se você quer adicionar o README, vamos rastrear

```
git add
```

\$ git add README

Executando o comando status novamente, você pode ver que seu README agora está sendo rastreado e preparado (staged) para o commit:

```
$ 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: README

Se você fizer um `commit` neste momento, a versão do arquivo que existia no instante em que você executou `git add`, é a que será armazenada no histórico de *snapshots*. Você deve se lembrar que, quando executou `git init` anteriormente, em seguida, você também executou `git add (arquivos)` - isso foi para começar a rastrear os arquivos em seu diretório.

O comando `git add` recebe o caminho de um arquivo ou de um diretório. Se for um

diretório, o comando adiciona todos os arquivos contidos nesse diretório recursivamente.

Vamos modificar um arquivo que já estava sendo rastreado. Se você modificar o arquivo CONTRIBUTING.md, que já era rastreado, e então executar `git status` novamente, você deve ver algo como:

```
$ 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: README

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: CONTRIBUTING.md

O arquivo `CONTRIBUTING.md` aparece sob a seção “Changes not staged for commit” — que indica que um arquivo rastreado foi modificado no diretório de trabalho mas ainda não foi mandado para o *stage* (preparado). Para mandá-lo para o *stage*, você precisa executar o comando `git add`.

Pode ser útil pensar nesse comando mais como “adicone este conteúdo ao próximo *commit*”.

Vamos executar `git add` agora, para mandar o arquivo CONTRIBUTING.md para o *stage*, e então executar `git status` novamente:

```
$ git add CONTRIBUTING.md  
$ 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: README  
modified: CONTRIBUTING.md
```

Ambos os arquivos estão preparados (no *stage*) e entrarão no seu próximo *commit*.

Neste momento, suponha que você se lembre de uma pequena mudança que quer fazer no arquivo `CONTRIBUTING.md` antes de fazer o *commit*. Você abre o arquivo, faz a mudança e está pronto para fazer o *commit*. No entanto, vamos executar `git status` mais uma vez:

```
$ vim CONTRIBUTING.md  
$ 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: README  
modified: CONTRIBUTING.md
```

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: CONTRIBUTING.md
```

Que negócio é esse? Agora o `CONTRIBUTING.md` está listado como preparado (*staged*) e também como não-preparado (*unstaged*). Se você executar `git commit` agora, a versão do `CONTRIBUTING.md` que vai para o repositório é aquela de quando você executou `git add`, não a versão que está no seu diretório de trabalho. Se você modificar um arquivo depois de executar `git add`, você tem que executar `git add` de novo para por sua versão mais recente no *stage*

- Status curto

O Git também tem uma *flag* para status curto, que permite que você veja suas alterações de forma mais compacta. Se você executar `git status -s` ou `git status --short` a saída do comando vai ser bem mais simples:

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

Arquivos novos que não são rastreados têm um ?? do lado, novos arquivos que foram adicionados à área de *stage* têm um A, arquivos modificados têm um M e assim por diante.

- Ignorando arquivos

Frequentemente você terá uma classe de arquivos que não quer que sejam adicionados automaticamente pelo Git e nem mesmo que ele os mostre como não-rastreados.

Geralmente, esses arquivos são gerados automaticamente, tais como arquivos de *log* ou arquivos produzidos pelo seu sistema de compilação (*build*).

```
$ cat .gitignore
*.o
*.a
*~
```

A primeira linha diz ao Git para ignorar todos os arquivos que terminem com “.o” ou “.a” – arquivos objeto ou de arquivamento, que podem ser produtos do processo de compilação. A segunda linha diz ao Git para ignorar todos os arquivos cujo nome termine com um til (~), que é usado por muitos editores de texto, como o Emacs, para marcar arquivos temporários.

- . Linhas em branco ou começando com # são ignoradas.
- . Você pode negar um padrão ao fazê-lo iniciar com um ponto de exclamação (!) exemplos:

```
# ignorar arquivos com extensão .a
*.a
#mas rastrear o arquivo lib.a, mesmo que você esteja ignorando os arquivos .a acima
!lib.a
```

```
# ignorar o arquivo TODO apenas no diretório atual, mas não em subdir/TODO  
/TODO  
  
# ignorar todos os arquivos no diretório build/  
build/  
  
# ignorar doc/notes.txt, mas não doc/server/arch.txt  
doc/*.txt  
  
# ignorar todos os arquivos .pdf no diretório doc/  
doc/**/*.pdf
```

---

```
git diff
```

O que você alterou mas ainda não mandou para o stage (estado preparado)? E o que está no stage, pronto para o commit? Apesar de o git status responder a essas perguntas de forma genérica, listando os nomes dos arquivos, o git diff exibe exatamente as linhas que foram adicionadas e removidas — o patch, como costumava se chamar.

estudar mais

Se você quer ver o que selecionou que irá no seu próximo commit, pode utilizar:

```
git diff --cached
```

---

```
git commit
```

- - m <mensagem>

Tente usar descrições objetivas e simples para que seu time, e até você no futuro, consigam entender o que aquele commit faz

- - a <inclusa todos os arquivos modificados>

você incluirá automaticamente no seu commit todos os arquivos modificados. Ou seja, você pulará o passo de adicionar os arquivos para a área de staging e irá direto para a área de commit. Entretanto, novos arquivos, que ainda não são acompanhados pelo git, não serão incluídos.

```
git commit -am "Aqui vai uma mensagem"
```

- reescreva o último commit

Se você já fez um commit e esqueceu de adicionar algum arquivo ou alteração ou até mesmo quer trocar a mensagem do commit, o amend é a melhor opção. Mas tome cuidado, o amend não só altera o último commit, mas o substitui totalmente pelo que você está criando agora. Tem duas formas de usar o amend. Independentemente de qual você use, se você quiser adicionar alterações de arquivos, faça isso antes de usar esses comandos (você precisa usar o git add para isso):

```
git commit --amend -m "Aqui vai uma mensagem" —> alterando a mensagem
```

```
git commit --amend --no-edit —> sem alterar a mensagem
```

Outro cuidado que você deve tomar, é evitar ao máximo usar o *amend* quando o último commit já está no repositório remoto, ou seja, já está público e pode ter sido usado como referência para outras pessoas.

---

```
desfazendo commits  
git revert  
git reset  
git reflog
```

`git revert`: é a forma mais segura de “desfazer” um commit, pois ele não apaga o commit do histórico. O que ele faz é pegar as alterações do commit que você quer reverter e criar um novo commit com essas alterações desfeitas.

`git reset`: se você realmente precisa apagar um commit, muitas vezes por conter alguma informação mais sensível,

`git reflog`: com esse comando você consegue ter acesso aos logs de referências, também conhecidos como reflogs, de cada um dos commits. A partir disso, você consegue deletar ou realizar outras operações com cada uma das referências.

---

```
git log
```

Para conseguir o projeto, execute

```
$ git clone https://github.com/schacon/simplegit-progit
```

Quando você executa `git log` neste projeto, você deve receber um retorno que se parece com algo assim:

```
$ 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

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

first commit
```

Por padrão, sem argumentos, `git log` lista os commits feitos neste repositório em ordem cronológica inversa; isto é, o commit mais recente aparece primeiro. Como você pode ver, esse comando lista cada commit com o seu checksum SHA-1, o nome e email do autor, data de inserção, e a mensagem do commit.

Uma das opções que mais ajuda é `-p`, que mostra as diferenças introduzidas em cada commit. Você pode também usar `-2`, que lista no retorno apenas os dois últimos itens:

a opção `--stat` apresenta abaixo de cada commit uma lista dos arquivos modificados, quantos arquivos foram modificados, e quantas linhas nestes arquivos foram adicionadas e removidas. Por último ela também colocar um resumo das informações.

Uma outra opção realmente muito útil é `--pretty`. Essa opção modifica os registros retornados para formar outro formato diferente do padrão. Algumas opções pré-definidas estão disponíveis para você usar. A opção `oneline` mostra cada commit em uma única linha, esta é de muita ajuda se você está olhando para muitos commits. Em adição, as opções `short`, `full`, e `fuller` apresentam o retorno quase no mesmo formato porem com menos ou mais informações

```
git remote
```

`-v` mostra que as URLs que o Git tem armazenado pelo nome abreviado a ser usado para ler ou gravar naquele repositório remoto

Se você clonou seu repositório, você deve pelo menos ver `origin(origem)` – que é o nome padrão dado pelo Git ao servidor que você clonou:

Isto significa que nós podemos obter (pull) contribuições de qualquer um desses usuários muito facilmente.

- adicionando repositórios remotos

Para adicionar um novo repositório Git remoto como um nome curto que você pode referenciar facilmente, execute `git remote add <shortname> <url>`

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin    https://github.com/schacon/ticgit (fetch)
origin    https://github.com/schacon/ticgit (push)
pb      https://github.com/paulboone/ticgit (fetch)
pb      https://github.com/paulboone/ticgit (push)
```

Agora você pode usar a string `pb` na linha de comando no lugar de uma URL completa. Por exemplo, se você quiser buscar toda a informação que Paul tem, mas você ainda não tem em seu repositório, você pode executar `git fetch pb`

O comando `git fetch` vai até aquele projeto remoto e extrai todos os dados daquele projeto que você ainda não tem. Depois que você faz isso, você deve ter como referência todos as ramificações(branches) daquele repositório remoto, que você pode mesclar(merge) com o atual ou inspecionar a qualquer momento.

Se o branch atual é configurando para rastrear um branch remoto, você pode usar o comando `git pull` para buscar(fetch) e então mesclar(merge) automaticamente aquele branch remoto dentro do seu branch atual. Este pode ser um fluxo de trabalho mais fácil e mais confortável para você, e por padrão, o comando `git clone` automaticamente configura a sua master branch local para rastrear a master branch remota ou qualquer que seja o nome do branch padrão no servidor de onde você o clonou. Executar `git pull` comumente busca os dados do servidor de onde você originalmente clonou e automaticamente tenta mesclá-lo dentro do código que você está atualmente trabalhando.

- inspecionando arquivo remoto

Se você quiser ver mais informações sobre um servidor remoto em particular, você pode usar o comando `git remote show [nome-remoto]`

Ele lista a URL para o repositório remoto, bem como as informações de rastreamento do branch. O comando, de forma útil, comunica que se você estiver no branch master e executar `git pull`, ele irá mesclar (merge) automaticamente no branch master do servidor após buscar (fetch) todas as referências remotas. Ele também lista todas as referências remotas recebidas.

- renomeando e removendo remotamente

Você pode utilizar o `git remote rename` para alterar o nome curto de servidores remotos. Por exemplo, se você deseja renomear `pb` para `paul`, você pode fazer isso com `git remote rename`:

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

Vale a pena mencionar que isso muda todos os nomes de ramificações de rastreamento remoto também. O que costumava ser referenciado em `pb/master` agora está em `paul/master`.

Se você quiser remover um servidor remoto por algum motivo - e você anteriormente moveu o servidor ou não está mais usando um em particular, ou talvez um contribuidor não esteja mais contribuindo - você pode usar `git remote remove` ou `git remote rm`:

```
$ git remote remove paul  
$ git remote  
origin
```

---

```
git push
```

comando `git push` é usado para enviar o conteúdo do repositório local para um repositório remoto. O comando `push` transfere commits do repositório local a um repositório remoto. É o oposto do comando `git fetch`, que importa commits para branches locais, enquanto o comando `push` exporta commits para branches remotos.

Quando você tem seu projeto em um ponto que deseja compartilhar, é necessário enviá-lo para o servidor remoto. O comando para isso é simples: `git push [remote-name] [branch-name]`

```
git push <remote> --all —> manda tudo local para o remoto
```