Se a instalação correu normalmente, o comando git deve mostrar uma saída como a abaixo, mostrando como usar a ferramenta e quais os comandos mais comuns.

usage: git [--version] [--help] [-C <path>] [-c name=value]

[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]

[-p | --paginate | --no-pager] [--no-replace-objects] [--bare]

[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]

<command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)

clone Clone a repository into a new directory

init Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)

add Add file contents to the index

mv Move or rename a file, a directory, or a symlink

reset Reset current HEAD to the specified state

rm Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)

bisect Use binary search to find the commit that introduced a bug

grep Print lines matching a pattern

log Show commit logs

show Show various types of objects

status Show the working tree status

grow, mark and tweak your common history

branch List, create, or delete branches

checkout Switch branches or restore working tree files

commit Record changes to the repository

diff Show changes between commits, commit and working tree, etc

merge Join two or more development histories together

rebase Reapply commits on top of another base tip

tag Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)

fetch Download objects and refs from another repository

pull Fetch from and integrate with another repository or a local branch

push Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some concept guides. See 'git help <command>' or 'git help <concept>' to read about a specific subcommand or concept.

git blame "arg.xxx": Este comando, seguido pelo nome de um arquivo, mostra a última pessoa que alterou cada linha do mesmo e a data da alteração.

git add pode ser executado de modo interativo com a opção -i. git add -i, o modo interativo inicia um prompt de comando específico, aguardando as decisões do usuário sobre quais as alterações detectadas no working directory devem ser adicionadas ao index.

1. Rode o comando git add -i

$ git add -i

staged unstaged path

1: unchanged +1/-0 index.html

\*\*\* Commands \*\*\*

1: status 2: update 3: revert 4: add untracked

5: patch 6: diff 7: quit 8: help

What now>

1. Para adicionar arquivos alterados, utilize a opção update. Você pode digitar tanto o número, quanto a palavra escrita como também somente a primeira letra dela. No caso, utilizamos aqui a abreviação u e pressione enter.

What now> u

staged unstaged path

1: unchanged +1/-0 index.html

1. Aparecerá uma listagem dos arquivos modificados, com um número atribuído a cada arquivo. Digite o número que representa o arquivo index.html e pressione enter .

Update>> 1

staged unstaged path

\* 1: unchanged +1/-0 index.html

1. Aparecerá um asterisco ao lado do número do arquivo, indicando que ele será adicionado. Pressione enter novamente para confirmar e voltar ao menu principal.

Update>>

updated one path

\*\*\* Commands \*\*\*

1: status 2: update 3: revert 4: add untracked

5: patch 6: diff 7: quit 8: help

1. Se quiser, confira as modificações. Utilize a opção status para conferir que o arquivo index.html foi adicionado corretamente, digitando a opção e pressionando enter.

What now> status

staged unstaged path

1: +1/-0 nothing index.html

\*\*\* Commands \*\*\*

1: status 2: update 3: revert 4: add untracked

5: patch 6: diff 7: quit 8: help

1. Utilize a opção quit para terminar a interação!

What now> 7

Bye.

$

**Configurando o repositório remoto**

git remote add origin https://github.com/[seu\_usuario\_no\_github]/curso-git.git

Com isso, você está indicando que o repositório Git local se conecta com um repositório **remoto** através do **alias** ou atalho origin cujo endereço real é  https://github.com/[seu\_usuario\_no\_github]/curso-git.git.

Agora que configuramos ele, podemos enviar nossos commits para lá através do comando git push origin master.

Esse comando envia os dados da branch local master para a branch remota master, localizada no repositório origin

Outra alternativa é utilizar, no primeiro push, a opção -u ou --set-upstream. Ela atrela a branch remota à local, fazendo com que não seja mais necessário passar como parâmetros a origem e a branch no comando push, que fica então assim: git push.

Ao executarmos o comando git whatchanged -p é possível visualizar quais as linhas que foram modificadas em cada commit do nosso projeto. Obs: também é possível utilizarmos git log -p, que imprime também os commits nos quais não houve modificação.

TECLAS DE ATALHO GIT BASH

Tab ⇆ : Autocompleta o comando a partir da posição do cursor.

Ctrl+a : Move o cursor para o inicio da linha, equivalente a tecla Home.

Ctrl+b : Move o cursor um caractere atrás.

Ctrl+c : Envia o sinal SIGINT para tarefa atual, então esta é cancelada e fechada.

Ctrl+d : Envia o marcador EOF, este fecha a sessão atual do shell, igual ao exit.

Ctrl+d : Deleta o caracter corrente, se existirem caracteres digitados.

Ctrl+e : Move o cursor para o final da linha, equivalente a tecla End.

Ctrl+f : Move o cursos para o caracter seguinte.

Ctrl+g : Aborta a pesquisa e restaura a linha original.

Ctrl+h : Deleta o caractere anterior, equivalente a tecla Backspace.

Ctrl+i : Equivalente a tecla Tab.

Ctrl+j : Equivalente a tecla Enter.

Ctrl+k : Limpa o conteúdo da linha que está após o cursor e copia

este conteúdo para área de memoria.

Ctrl+l : Limpa o conteudo da tela, equivalente ao comando clear.

Ctrl+n : Recupera o próximo comando (equivalente a tecla ↓).

Ctrl+o : Executa o comando encontrato do histórico.

Ctrl+p : Recupera o comando anterior (equivalente a tecla ↑).

Ctrl+q : Adiciona o próximo caractere digitado à linha textualmente.

Ctrl+r : Executa o ultimo comando digitado.

Ctrl+s : Volta ao próximo comando digitado recentemente.

Ctrl+t : Avançar os dois últimos caracteres.

Ctrl+u : Limpa o conteúdo da linha anterior ao cursor e copia o conteúdo para

área de memoria.

Ctrl+v : Adiciona o próximo carácter à linha textualmente.

Ctrl+w : Apaga a palavra depois do cursor e copia para área de memória.

Ctrl+y : Comando colar.

Ctrl+x Ctrl+e : Edita a linha atual no programa $EDITOR, ou vi.

Ctrl+x Ctrl+r : Ler o conteúdo do arquivo inputrc, e incorporar todas as

ligações ou atribuições de variáveis ​​encontradas lá.

Ctrl+x Ctrl+u : Desfazer, separado por linha.

Ctrl+x Ctrl+v : Exibe informações sobre a versão da instancia corrente do bash.

Ctrl+x Ctrl+x : Alterar a posição do cursor para sua antiga posição.

(C-x, porque x tem uma forma de passagem).

Ctrl+z : Envia o sinal para SIGTSTP a tarefa atual, para suspende-lo. Para

executá-lo em segundo plano pode-se entrar no bg. Para trazer de

volta através de fundo ou suspensão fg ['nome do processo ou

identificação de trabalho "] (primeiro plano) pode ser emitido.

Ctrl+\_ : Desfazer, separado por linha.

Alt+b : move o cursor para palavra anterior.

Alt+c : Capitaliza o caracter sob o cursor e move para o final da palavra.

Alt+d : Corta a palavra depois do cursor.

Alt+f : move o cursor para próxima palavra.

Alt+l : Transforma para caixa baixa todo caractere a partir do cursor

em toda palavra.

Alt+r : Cancela as alterações e colocar de volta a linha de como era na história.

Alt+u : Transforma para caixa alta todo caractere a partir do cursor em toda

palavra.

Alt+. : Insira o último argumento para o comando anterior (a última palavra da

entrada história anterior).

**Trabalhando com branches: primeiros passos**

Já sabemos que a utilização de branches facilita no dia a dia do desenvolvedor. Mas como criar uma nova branch? Para tal, utilizamos o comando git branch, passando como opção o nome da branch que desejamos criar. No nosso caso, criaremos a branch design, onde realizaremos algumas alterações referentes ao design da nossa página html:

git branch design

Ao executarmos o comando, nenhuma saída é mostrada no prompt.

Agora, como verificamos quais são as branches existentes em um projeto? Isso se resolve com o comando git branch. Ele nos fornece todas as branches criadas na máquina. Ele também possibilita visualizar qual a branch que estamos atualmente através de um "\*" que precede o nome da branch atual.

git branch

design

\* master

Observe que o "\*" precede uma branch chamada master. Mas nós não a criamos agora. De onde ela surgiu? A branch master é criada quando executamos o nosso primeiro commit do projeto. Ela é considerada a branch principal do projeto.

Mas, se quisermos alterar o projeto numa outra branch, como é que fazemos para alterar a branch atual? Isto é feito através do comando git checkout, passando o nome da branch para a qual desejamos mudar. No nosso caso, temos:

git checkout design

Switched to branch 'design'

E pronto. Todas as alterações que realizaremos a partir de agora estarão na branch design.

Vamos adicionar estilo para a nossa página. Para tal, copie o seguinte código num arquivo chamado design.css. Adicione este arquivo na pasta do seu projeto.

body {

background-color: blue;

}

Adicione também a seguinte linha no header do seu arquivo index.html:

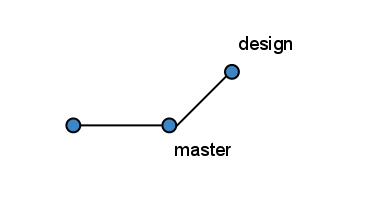
<link rel="stylesheet" type="text/css" href="/design.css"/>

Porém, essas alterações ainda não foram atualizadas no repositório. Para isso, precisamos adicionar os arquivos e commitar as alterações:

git add design.css index.html

git commit -m "Adicionando estilo para a nossa página"

E pronto! As alterações estão salvas na branch design. O esquema do projeto com as branches pode ser vista na figura a seguir:



Se voltarmos para a branch master com o comando git checkout master, vemos que as alterações feitas anteriormente não estão mais presentes.

Por fim, perceba que todas essas alterações foram realizadas sem precisar de conexão com a internet. O Git nos permite trabalhar tanto com o repositório remoto quanto com o nosso próprio repositório local, ao contrário de outros controladores de versão como o SVN e CVS.

**Verificando atualizações do repositório**

git fetch origin

Realizando o comando git fetch origin, podemos verificar todas as atualizações que foram realizadas no repositório referente ao origin.

**Compartilhando branches locais com outros desenvolvedores**

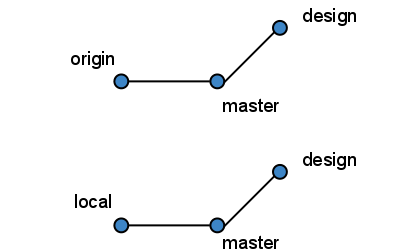
Agora, o que acontece se você começou alguma alteração em um projeto e não terminou? Será que é bom manter apenas localmente? E se alguém quiser continuar as alterações que você iniciou?

E se criamos uma nova ferramenta que quebra a compatibilidade com as versões anteriores? Para resolver esses problemas, é bom manter também essas branches no repositório remoto.

Iniciaremos este trabalho enviando a branch criada localmente para o repositório remoto. Isso é feito utilizando o comando git push passando dois argumentos: o primeiro é o nome do repositório e o segundo, o nome da branch que deseja-se enviar. No nosso caso, temos:

git push origin design

Com isso, o repositório remoto conterá uma cópia fiel da branch design local. Isso pode ser visto na figura a seguir:

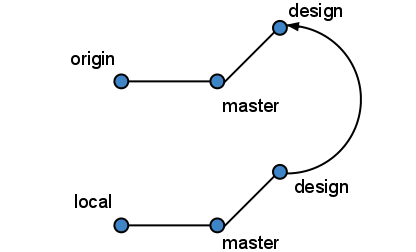


Porém, toda vez que atualizarmos tanto o nosso projeto local quanto o projeto remoto, precisaremos indicar qual o repositório e o nome da branch que a nossa branch local se refere no remoto, isto é, precisaremos digitar git pull origin design e git push origin design para atualizar os repositórios locais e remotos, respectivamente.

Para evitar tal trabalho, podemos indicar o caminho (track) da branch remota para a nossa branch local. Isso pode ser feito no instante em que criamos a branch remota através da opção "-u". No nosso caso, temos:

git push -u origin design

Com isso, a nossa branch local sabe qual a branch remota que ela se referencia.



E como podemos visualizar as branches já existentes em um repositório remoto? Isso é feito através da opção "-r" passado ao comando git branch.

git branch -r

origin/HEAD -> origin/master

origin/design

Uma vez visto as branches remotas, como copiar uma delas para a máquina local? Isso é feito passando o nome do repositório e da branch remota ao comando git branch, além de indicar o nome da branch que será criada. Mais uma vez, temos o problema de indicar o caminho entre as branches. Para este caso, a opção -t resolve.

git branch -t design origin/design

Ao digitar o comando git push -u origin design, criamos a branch remota design no repositório referente ao origin e criamos uma ligação entre a nossa branch e a branch remota.

O git nos fornece uma opção ao comando git branch para listar tanto as branches locais quanto as remotas.   
git branch -a

Uma sequência de tarefas bem comum referente à cópia de uma branch localizada num repositório remoto é:

## Criação de uma branch local com o mesmo nome da branch remota

## Mudança para essa nova branch criada

## Criação de link entre a branch local e remota

Para evitar toda essa tarefa, o git nos fornece um atalho para evitar esse trabalho todo com uma opção do comando git checkout. Que opção é essa?

Com o comando git checkout -t origin/design, todo esse trabalho é feito automaticamente para a branch chamada design

git push -d origin design Este comando remove a branch remota design.

git branch origin :design Para remover uma branch remota, você deve usar o comando git push, passando alguma opção extra.

git fetch origin Este comando verifica todas as atualizações que foram realizadas no repositório de atalho origin.

git push origin design Este comando vai atualizar o histórico de commits do repositório remoto.

O comando git mergetool --tool-help mostra no console uma lista de programas possíveis de ser utilizados. Dessa lista, pode-se escolher um, instalar no seu computador e utilizar através do comando git mergetool -t nome\_do\_programa.

Algumas das ferramentas gratuitas mais comuns são o [kdiff3](http://kdiff3.sourceforge.net/), [Meld](http://meldmerge.org/" \t "_blank) ou [P4Merge](https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge). Em MacOSX também existe o semi-nativo FileMerge, que vem junto com o XCode.

Vamos praticar o comando git rebase.

1. Da branch master, utilize o comando git checkout -b testeRebase para criar e se mover para uma nova branch
2. Altere alguns arquivos, crie um ou dois commits nesta branch
3. Volte para a master e faça a mesma coisa, se possível evitando conflitos. Crie arquivos com nomes diferentes, por exemplo, e faça um ou dois commits novos
4. Volte para a testeRebase e crie mais um ou dois commits
5. Dê uma olhada na saída do comando git log e repare que os commits que criou na masternão estão listados
6. Execute o comando git rebase master para aplicar os commits de lá que faltam na branch testeRebase
7. Dê uma olhada na saída do comando git log e repare que os commits que criou na masterestão listados, mas não estão por último

git reset --hard

Com este comando, as alterações são removidas do histórico local de commits e também tanto do index quanto do working directory, permanentemente.

git reset --soft

Com este comando, as alterações são removidas do histórico local de commits, mas não são removidas do index.

git reset

Com este comando, as alterações são removidas do histórico local de commits, do index, mas não do working directory`

git reset --hard HEAD~3

Ao digitarmos HEAD~ seguido de um número inteiro n, será feita uma referência ao n-ésimo commitanterior ao HEAD. Por exemplo, HEAD~1, faz referência ao **penúltimo** commit e HEAD~2, ao **antepenúltimo**. No caso específico do penúltimo commit, também pode-se referenciá-lo utilizando o atalho HEAD^

Como descartamos as alterações que estão no último commit?

git reset --hard HEAD~1 : Usando esse comando, descartamos definitivamente as mudanças feitas no último commit.

Qual o comando que devemos executar para guardar as alterações que estão no working directory e no index e retornar ao estado do último commit?

git stash:  faz com que as alterações que estão no working directory e no index sejam salvas, retornando o estado para o último commit.

Como fazemos para recuperar o último estado salvo com o comando git stash?

git stash pop: Este comando faz com que o último estado salvo seja recuperado e, caso não haja conflitos, deletado da pilha de dados armazenados.

Como fazemos para recuperar o último estado salvo com o comando git stash sem removê-lo da pilha de alterações?

git stash apply: Usando a opção apply, recuperamos as últimas alterações da pilha sem removê-las.

Às vezes, salvamos um estado utilizando o git stash mas não desejamos mais usá-lo. Como fazemos para apagar o último estado salvo?

git stash drop: faz com que o último estado salvo seja apagado. Também podemos utilizar o nome de cada elemento do stash para remover algum estado que não seja o último. Por fim, se quisermos excluir todos os estados, podemos utilizar o comando git stash clear

**Redução dos comandos com Alias**

Com o passar de algum tempo trabalhando com Git, muitos desenvolvedores sentem que digitar os mesmos comando a todo o momento é uma tarefa extremamente improdutiva e repetitiva. Imagine, sempre que você quiser realizar um git push, ter que trocar de branch, fazer um pull do repositório remoto, realizar o rebase, trocar novamente de branch, fazer o git merge para, aí sim, poder executar o git push. Nesse ínterim, foram 5 comandos que tiveram que ser digitados e executados, excluindo o commit que deve ter sido feito anteriormente. Nessa situação, é muito comum encontrar desenvolvedores que perdem bastante tempo digitando todos esses comandos em sequência e, consequentemente, perdendo o foco em algo que estavam fazendo e até mesmo produtividade.

Justamente para solucionar esse problema, o Git nos permite criar alias para os comandos. Um exemplo de alias simples, que é bastante utilizado no mercado ao trabalhar com o Git, é usar a abreviação st para o comando status. Dessa maneira, ao executar git st, é executado o comando git status. Para criar os alias dos comandos, basta editar o arquivo .gitconfig existente na pasta do seu usuário, adicionando uma seção chamada [alias] contendo um conjunto de chave e valor onde a chave indica o nome do novo comando (o alias) e o valor indica o comando original que será executado. Nesse caso, podemos ter um .gitconfig com o seguinte conteúdo:

[alias]

st = status

Muitos desenvolvedores que já utilizaram Subversion (SVN) em linha de comando também se identificam com o comando ci, que realiza um commit no SVN. Nesse caso, agora, podemos introduzi-lo ao Git através de um alias para o comando commit:

[alias]

st = status

ci = commit

Outros alias comuns são br para branch, co para checkout e df para diff.

Um outro truque muito comum de ser usado com os alias do Git é a execução de vários comandos em apenas um. Com isso, podemos criar um comando chamado envia que sincroniza nossa branch desenvolvimento com o repositório remoto, faz o rebase e o merge e depois realiza o push, ou seja, o fluxo comum de ser utilizado com o Git.

No caso desse alias, como vamos querer que ele execute vários comandos, teremos a seguinte entrada no .gitconfig:

[alias]

envia = git checkout master && git pull && git checkout desenvolvimento && git rebase master && git checkout master && git merge desenvolvimento && git push

Note que o nosso alias, além de ter ficado grande, pois queremos resumir vários comandos em apenas um, também utiliza os comandos git e o && para realizar a sequência de comandos. Os alias, por padrão, não suportam a execução de outros comandos, como estamos fazendo, porém, é algo possível de ser habilitado. Para isso, basta adicionar uma ! ao começo do comando:

[alias]

envia = !git checkout master && git pull && git checkout desenvolvimento && git rebase master && git checkout master && git merge desenvolvimento && git push

Pronto. Agora, se executarmos o comando git envia, todo esse processo será automaticamente feito para nós.