Curso de git

Aprendendo na prática

por

Marcelo L. Lotufo (marcel0ll)

Git: Aprendendo na prática

```
          o-o-o (branch-1)
         /     \
o-o-o-o-o-o-o-o-o-o-o (main)
         \         /
          o-o-o-o-o (branch-2)
```

A ideia deste curso é auxiliar o aprendizado de git seguindo três etapas:

- motivação : Por que aprender algo
- teoria : O que é isso
- prática : Como usar + exercícios de fixação

O curso tem como intenção exigir o mínimo de outras tecnologias que não o próprio git. Dessa forma, evita-se
qualquer linguagem de programação específica e são utilizados arquivos de texto .md para teoria e práticas.

Alguns comandos ou argumentos que facilitam o trabalho do dia a dia serão intencionalmente ignorados a fim
de apresentar boas práticas no uso da ferramenta git.

Organização do curso

O curso está organizado nas seguintes partes e seções:

Parte 1: Aula Expositiva

Durante essa parte, um palestrante falará sobre cada uma das seguintes seções
e ao mesmo tempo mostra em um terminal sobre o que está falando. Aconselha-se
a quem está assistindo ao curso que acompanhe conforme conseguir o conteúdo apresentado.

Parte 2: Prática

Durante essa parte, o palestrante e possivelmente outros instrutores auxiliarão
na execução das tarefas listadas.

Índice

1. Aula

1. O que é o git e por que eu deveria me importar? (motivacao.md)
    - As duas ferramentas mais importantes
    - Sistema de controle de versões (vcs)
    - O dia a dia
    - Comandos
        - `git help`
2. Visão geral do git (visao-geral.md)
3. Configurando o git (configurando-o-git.md)
    - Configurações Globais
    - Configurações Locais
4. Diretório de trabalho (diretorio-de-trabalho.md)
5. O repositório git (repositorio.md)
    - Iniciando um repositório
    - Checando o resporitório
    - O diretório git
    - Comandos
        - `git init`
        - `git status`
6. Preparando o **stage** (stage.md)
    - O que é o **stage**?
    - O que é uma modificação?
        - Arquivos
        - Chunks
    - Comandos
        - `git add`
        - `git reset`
        - `git diff`
7. A base de tudo, o **commit** (commit.md)
    - O que é um **commit**?
    - Comandos
        - `git commit`
        - `git log`
    - Boas práticas ao criar um **commit**
8. Dividir para conquistar! (branch.md)
    - O que é um **ref**?

- O que é um **branch**?
- O que é uma **tag**?
- O que é o **HEAD**?
- Comandos
    - `git branch`
    - `git tag`
    - `git checkout`

9. Juntando o trabalho (merge-rebase.md)
    - O que é um **merge**?
    - O que é um **rebase**?
    - Comandos
        - `git merge`
        - `git rebase`
    - O que é um conflito?
    - Resolvendo conflitos
10. Local e remotes (local-remote.md)
    - O que é o **local**?
    - O que é um **remote**?
    - Comandos:
        - `git clone`
        - `git remote`
        - `git fetch`
        - `git push`
        - `git pull`
11. Saiba mais (saber-mais.md)
    - Padrões de commit
        - Convertional commits
    - Nomeando versões
        - Semantic versioning
    - Quem foi que fez isso?!?
        - `git blame`
    - O que é o **reflog**?
        - `git reflog`
    - O que é o **stash**?
        - `git stash`
        - `git stash pop`
    - O que é **cherry-pick**?

        - `git cherry-pick`
    - O que é **bisect**?
        - `git bisect`
    - O que são **hooks**?
12. Fluxos de trabalho para git (fluxos.md)

2. Prática

1. Exercício 1 (exercicio-1.md)

O que é o git e por que eu deveria me importar?

As duas ferramentas mais importantes

Existem duas ferramentas que desenvolvedores usam diariamente, na maioria de seus projetos: um editor de texto e um software de controle de versão.

Sendo assim, é muito importante que se tenha domínio dessas ferramentas.

Sistema de controle de versões (vcs)

O git é um desses softwares de controle de versão. Existem outros atualmente menos usados como mercurial (https://www.mercurial-scm.org/) e subversion (https://subversion.apache.org/).

Se você alguma vez teve um arquivo chamado entrega_final_3b.docx, você teria tido benefícios usando o git, ou qualquer outro software de versionamento.

Um software de controle de versões ajuda você a criar um histórico de todas as modificações que foram feitas em um ou mais arquivos, a partir de um diretório raiz, o diretório de trabalho.

Softwares de controle de versão também são um ótimo instrumento para colaboração no desenvolvimento de projetos, uma vez que cada um pode trabalhar no seu próprio conjunto de modificações e posteriormente usar o software de controle de versão para unificar essas mudanças.

Este curso tem a intenção de explicar as vantagens de usar git em seus projetos, explicar quais são as partes mais básicas do git e incentivar que você busque por si saber mais sobre essa ferramenta incrível.

Por mais que eu quisesse, seria impossível passar por todos os comandos e suas opções em poucas horas. Além disso, muitos desses comandos são específicos à maneira como você irá usar a ferramenta. Alguns desses comandos eu nunca cheguei a usar.

O dia a dia

O fluxo padrão de uso do git consiste em adicionar quais arquivos o git deve manter versionados, adicionar as modificações, salvar estágios do software e compartilhar com colegas esses estágios. Para executar essas tarefas básicas, são usados os seguintes comandos:

- git add
- git commit
- git push
- git pull

Mas saber cegamente executar alguns comandos não faz com que você entenda como a ferramenta funciona e, no primeiro momento que ela não responder como desejado, ficaremos perdidos porque não sabemos de fato os fundamentos da ferramenta.

Por essas razões, este curso tem a intenção de ir mais a fundo do que só pincelar o o modo de uso de alguns comandos.

Abaixo segue a saída do comando `git help -a`, com todos os comandos que a ferramenta possui. No dia a dia, são usados alguns poucos, mas é importante saber que a ferramenta tem muito mais a oferecer e o help é o melhor lugar para procurar saber mais.

```
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>]

available git commands in '/usr/lib/git-core'

  add                         merge-ours
  add--interactive            merge-recursive
  am                          merge-resolve
  annotate                    merge-subtree
  apply                       merge-tree
  archive                     mergetool
  bisect                      mktag
  bisect--helper              mktree
  blame                       mv
  branch                      name-rev
  bundle                      notes
  cat-file                    pack-objects
  check-attr                  pack-redundant
  check-ignore                pack-refs
  check-mailmap               patch-id
  check-ref-format            prune
  checkout                    prune-packed
  checkout-index              pull
  cherry                      push
  cherry-pick                 quiltimport
  clean                       read-tree
  clone                       rebase
  column                      rebase--helper
```

```
commit                    receive-pack
commit-tree               reflog
config                    remote
count-objects             remote-ext
credential                remote-fd
credential-cache          remote-ftp
credential-cache--daemon  remote-ftps
credential-store          remote-http
daemon                    remote-https
describe                  remote-testsvn
diff                      repack
diff-files                replace
diff-index                request-pull
diff-tree                 rerere
difftool                  reset
difftool--helper          rev-list
fast-export               rev-parse
fast-import               revert
fetch                     rm
fetch-pack                send-pack
filter-branch             sh-i18n--envsubst
fmt-merge-msg             shell
for-each-ref              shortlog
format-patch              show
fsck                      show-branch
fsck-objects              show-index
gc                        show-ref
get-tar-commit-id         stage
grep                      stash
hash-object               status
help                      stripspace
http-backend              submodule
http-fetch                submodule--helper
http-push                 subtree
imap-send                 symbolic-ref
index-pack                tag
init                      unpack-file
init-db                   unpack-objects
instaweb                  update-index
interpret-trailers        update-ref
log                       update-server-info
ls-files                  upload-archive
ls-remote                 upload-pack
ls-tree                   var
mailinfo                  verify-commit
mailsplit                 verify-pack
merge                     verify-tag
merge-base                web--browse
merge-file                whatchanged
merge-index               worktree
merge-octopus             write-tree
merge-one-file
```

```
'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.
```

Comandos

`git help`

O seu melhor amigo na hora de usar o git. Esta ferramenta contém toda a documentação mais atualizada para a sua versão do git. Não se assute com a interface do terminal.

Visão geral do git

1. Configurar git (user.name, user.email)
2. Clonar repositório git@github.com:marcel0ll/curso-git.git
3. Modificar arquivo README.md
4. Adicionar README ao stage
5. Criar commit
6. Enviar modificação ao github

Editado durante o curso

Configurando o git

O git possui diversas configurações globais que podem ser encontradas em `~/.gitconfig` e também diversas configurações locais que se encontram dentro da pasta `.git/` de cada repositório.

Configurações Globais

Para começarmos a usar o git, é necessário configurar o seu usuário globalmente para que o git possa registrar a cada etapa QUEM foi o AUTOR das modificações.

Para isso, precisaremos configurar o seu `email` e seu nome(`name`) de usuário:

```
git config --global user.email "<seu.endereço.de@email.com>"
git config --global user.name "<seu_nome_de_usuario>"
```

Como dito acima, as configurações globais vão parar em um arquivo na sua pasta de usuário, no arquivo `.gitconfig`.

Nesse arquivo podem estar configurados diversas opções do seu git, como aliases para comandos, qual editor usar para escrever commits, o email e nome de usuário padrão, etc.

Segue abaixo o meu arquivo de configuração que uso normalmente. No momento, só se atente que o que configuramos nos comandos acima estão agora registrados aqui.

```
[user]
    email = marcelo.lopes.lotufo@gmail.com
    name = marceloll

[core]
    editor = vim

; isso é um comentário
[alias]
    st = status
    br = branch
    co = checkout
    last = log -1
    staged = diff --staged
    ;graph = log --graph --oneline --decorate --author-date-order
    amend = commit --amend
    graph = log --all --graph --abbrev-commit --author-date-order --decorate --
format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset)
%C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(auto)%d%C(reset)'
```

Configurações Locais (em cada repositório)

No momento não cabe entrar em muitos detalhes do que é possível configurar no arquivo `.git/config` e nos outros arquivos da pasta `.git`, mas é importante saber que elas existem. Em outras partes do curso, voltaremos a comentar sobre essas configurações locais.

Faça você mesmo:

```
git config --global user.email "<seu.endereço.de@email.com>"
git config --global user.name "<seu_nome_de_usuario>"
```

Diretório de trabalho

Parece até inútil ter essa seção dentro do curso, mas na verdade é importante deixarmos explícitas todas as partes e com o diretório de trabalho não é diferente.

O diretório de trabalho nada mais é que a pasta raiz do repositório, onde se encontram a árvore de arquivos e as pastas. O git irá da sua própria forma espelhar a estrutura de arquivos da pasta de trabalho através das diversas modificações que forem feitas nesses aquivos.

A pasta desse curso, no momento que eu estou escrevendo, está dessa forma:

```
arquivos.md
branch.md
chunks.md
commit.md
configurando-o-git.md
diretorio-de-trabalho.md
fluxos.md
local-remote.md
merge-rebase.md
motivacao.md
README.md
repositorio.md
saber-mais.md
stage.md

.git/
  branches
  COMMIT_EDITMSG
  config
  description
  HEAD
  hooks
  index
  info
  logs
  objects
  packed-refs
  refs
```

O repositório git

```
          o-o-o
         /     \
o-o-o-o-o-o-o-o-o-o-o (main)
         \
          o-o-o-o-o
```

Um repositório git é uma árvore de arquivos rastreados pelo git a partir de uma pasta raiz (diretório de trabalho). Dentro dessa pasta raiz, o git usa a pasta `.git/` para armazenar os dados de que ele necessita para rastrear arquivos e mudanças.

Note que não é porque um arquivo está dentro da pasta de trabalho que ele será automaticamente rastreado. Somente os arquivos adicionados ao repositório são rastreados.

Iniciando um repositório

Para iniciarmos um repositório, precisamos nos movimentar para uma pasta que desejamos versionar com o git e executar o comando:

```
git init
```

Checando o repositório

Para verificarmos se o repositório está certo, o git nos dá o comando `status`, que nos diz qual o estado atual do repositório e de nossas modificações, como no exemplo abaixo:

```
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.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:   repositorio.md
```

Pronto. O git já criou a sua pasta `.git/` e nós já podemos começar a adicionar arquivos a este repositório.

O diretório .git/

```
.git/
  branches
  COMMIT_EDITMSG
  config
  description
  FETCH_HEAD
  HEAD
  hooks
  index
  info
  logs
  objects
  ORIG_HEAD
  packed-refs
  refs
```

Normalmente não olhamos para dentro da pasta `.git/`, pois ela é um diretório escondido que serve de base de dados para o seu repositório e não queremos quebrar nada mexendo em algo que não devemos. Mas para entender de fato como o git funciona é necessário se aventurar um pouco e abrir os arquivos desse diretório.

No momento, deixo aqui explícitos os arquivos e as pastas para posteriormente mencioná-los novamente.

Comandos

`git init`

Comando que inicializa estrutura para o git dentro da pasta `.git/`

`git status`

Comando para checar o estado atual de nosso repositório

Siga junto:

```
mkdir curso-git
cd curso-git
git init
git status
```

Trabalhando com o stage:

O que é o stage?

Como dito antes, é necessário ordenar de forma explícita para o git quais arquivos ele deve rastrear dentro da sua pasta de trabalho e isso acontece basicamente como se fôssemos tirar uma foto para recordação: preparamos o cenário com as pessoas ou objetos e tiramos uma foto desse palco.

Dentro do git o palco é o **stage** e a foto é um **commit**.

O **stage** é um ambiente intermediário entre os arquivos presentes na pasta de trabalho e os arquivos que o git já está ativamente rastreando.

O fluxo comum do git é adicionar uma modificação ao stage e posteriormente armazenar essa mudança no git em um commit.

Pasta de trabalho --`git add`--> **Stage** --`git commit`--> **Repositorio**

O que o git salva? O que é uma modificação?

Um conceito importante é que o git não salva ARQUIVOS de fato, ele salva MODIFICAÇÔES em arquivos. Os arquivos e pastas são representados e inferidos a partir dessas modificações feitas.

Essas modificações podem ser representadas por chunks de diff. Temos um exemplo deles mais abaixo.

Esse é o motivo pelo qual o git não armazena diretórios vazios. Dado que eles não possuem conteúdo para ser modificado, o git não tem o que armazenar. Logo, se você adicionar uma pasta ao seu projeto e ela não tiver pelo menos um arquivo dentro, essa pasta não estará dentro do seu repositório.

Arquivos

Um arquivo do próprio sistema de arquivos

Chunks

Um Chunk é um pedaço de um arquivo que representa uma modificação. É representado pelas linhas que foram modificadas no arquivo.

```
index e88cbe0..0ac7c5c 100644
--- a/README.md
+++ b/README.md
@@ -1,7 +1,9 @@
 # Curso git

-A ideia desse curso é auxiliar o aprendizado de git com uma mistura de teoria e prática.
-O curso consiste de uma série de etapas. As estapas iniciais são puramente de fundamentação e depois mistura teoria e prática para fixação dos conceitos.
+A ideia desse curso é auxiliar o aprendizado de git seguindo as seguintes etapas:
+- motivação : O por que de aprender algo
+- teoria    : O que é isso
+- prática   : Como usar e exercícios de fixação
```

Comandos

`git add <arquivos>`

O git add adiciona modificações ao stage

`git reset <arquivos>`

O git reset remove modificações do stage

`git diff`

O git diff mostra as modificações que foram feitas no diretório de trabalho desde o último commit. Ou a diferença entre dois commits.

Faca você Mesmo:

- Crie um novo arquivo
- E adicione no stage

A base de tudo, o commit

O que é um commit?

Um **commit** é como uma FOTO. Mas nela só vemos as diferenças entre a última foto e o momento atual dos arquivos RASTREADOS pelo git.

Cada **commit** tem um id único representado por um hash, seus commits pais, um autor, a data de criação, um título, uma mensagem e também todas as modificações em relação ao seu commit pai.

Um **commit** é de longe a parte mais importante de se entender, porque todo o resto que iremos ver deriva de como esse commit é estruturado.

Aqui podemos ver o output do comando `git log` e as informações mais básicas do commit.

```
commit a96f41474c500941801add83f99b5acb92d0c506
Author: Marcelo Lopes Lotufo <marcelo.lotufo@lotuz.dev>
Date:   Mon Jul 6 12:03:40 2020 -0300

    Criar README.md

    Criar arquivo inicial de README do projeto
```

Logo abaixo podemos ver a saída do comando `git log d579691e8285296f3aa0a1c6d1608cfbb7202473 -1 -c` e podemos ver tanto as informações básicas como 2 chunks de códigos modificados.

Não se preocupe com as flags do comando log no momento, só se atente à saída abaixo.

```
commit d579691e8285296f3aa0a1c6d1608cfbb7202473
Author: marceloll <marcelo.lopes.lotufo@gmail.com>
Date:   Sun Aug 2 14:33:26 2020 -0300

    Adicione links ao README do curso

diff --git a/README.md b/README.md
index e88cbe0..0ac7c5c 100644
--- a/README.md
+++ b/README.md
@@ -1,7 +1,9 @@
 # Curso git

-A ideia desse curso é auxiliar o aprendizado de git com uma mistura de teoria e prática.
-O curso consiste de uma série de etapas. As estapas iniciais são puramente de fundamentação e depois mistura teoria e prática para fixação dos conceitos.
+A ideia desse curso é auxiliar o aprendizado de git seguindo as seguintes etapas:
+- motivação : O por que de aprender algo
+- teoria    : O que é isso
+- prática   : Como usar e exercícios de fixação

 O curso tem como intenção exigir o mínimo de outras tecnologias que não o próprio git, dessa forma são evitadas
 qualquer linguagem de programação específica e são utilizados arquivos de texto .txt genéricos para exemplificação.
@@ -15,11 +17,11 @@ O curso está organizado nas seguintes lições:

 ## Motivação e fundamentação teórica

-1. Motivação para se usar um software de controle de versão
-1. O que é um VCS e quais são as principais diferenças entre git, hg e svn
-1. Quais são os principais elementos e termos do git
+1. [Motivação para se usar um software de controle de versão](parte-1.md)
+1. [Quais são os principais elementos e termos do git](parte-2.md)
     - Repositório
     - Arquivos
+    - Chunks
     - Stage
     - Commit
     - Branch
```

Comandos

git commit

O comando commit cria uma foto das modificações que estão nesse momento em stage. Quando usamos esse comando automaticamente, é aberto o editor configurado globalmente. No meu caso, é aberto o Vim.

`git log`

O comando log mostra o histórico de commits e as informações deles.
Como é impresso o log e quanto de informação mostrar podem ser configurados através de diversas opções.

Boas práticas ao criar um commit

De forma geral, a ideia é qun um **commit** agrupe modificações que façam sentido juntas, mas no dia a dia, quando estamos modificando nossos arquivos, nem sempre mexemos em somente um contexto. Vamos criar um formulário de login, mas também editamos a cor do botão.

O ideal é que separemos essas modificações pelo menos em 2 partes:

1. As mudanças do formulário
2. As mudanças de estilo do botão

Para isso, o git nos permite manipular as modificações de forma a serem adicionadas ou removidas do stage de forma interativa com as flags --patch ou --interactive sobre os comandos add e reset.

No momento, iremos focar no --patch, mas fica como curiosidade a flag --interactive.

O comando add/remove --patch (ou -p) nos faz passar por todos os chunks de modificações do diretório de trabalho onde podemos cirurgicamente adicionar só as partes que nos interessam.

Dessa forma, conseguimos criar um commit para cada ideia.

Na hora de dar nome aos commits, tome o cuidado necessário para descrever o que foi feito.
É costume escrever as mensagens de commit de forma imperativa e não no passado:

```
commit 881de6754e9f031a70cc9d06b06a32ad256d7133
Author: marceloll <marcelo.lopes.lotufo@gmail.com>
Date:   Tue Aug 4 15:19:13 2020 -0300

    Melhore a estapa `stage` do curso

    Detalhe os comandos e explique que o git salva modificações e não
    arquivos.
```

Note que além do título da mensagem também temos o corpo da mensagem para detalhar o que foi feito.

É muito importante escrever mensagens descritivas para posteriormente, se necessário, navegarmos pelo
histórico do repositório e sabermos o que e quando fizemos certas modificações.

Padrões de commit

Existem times que adotam padrões específicos para a escrita de seus commits, o que foge do escopo deste curso, mas fica aqui a referência para alguns desses padrões:

- Convetional commits (https://www.conventionalcommits.org/en/v1.0.0-beta.2/)

Faça você mesmo:

```
git commit
```

```
Crie novo arquivo README.me

Iniciando o nosso curso com um commit descritivo
```

Dividir para conquistar:

```
            o (feature-1)
           /
          o-o (dev)
         /
o-o-o-o-o-o (main)
     \
      o       o-o-o
       \     /
        o-o-o
           \
            o
```

Durante nosso trabalho, não queremos ficar presos a só uma possibilidade. Queremos explorar novos caminhos sem o risco de perder o nosso precioso trabalho.

Para isso, o git nos permite dividir nosso histórico em quantas partes quisermos e depois juntarmos as diferenças. No fundo, o que o git permite é que N commits tenham o mesmo commit como pai, efetivamente dividindo o histórico em N ramos.

Mas, quando temos o histórico dividido, começa a ficar difícil navegar pelos commits através dos seus hashes e aí entra as **refs**, que são referências a um commit.

Existem vários tipos de referências, como a referência a um ramo, a uma tag e aos ramos remotos.

O que é de fato uma ref?

Uma **ref** no fundo é um mapeamento de um nome para uma hash de algum commit. O git armazena isso em arquivos nos quais o nome é o nome do arquivo e a hash é armazenada dentro.

Podemos ver isso dentro da pasta `.git/refs/`, por exemplo no arquivo `.git/refs/heads/master`

```
ed6dcbdb79e2c3867f9f52f42bfcbd432c3ef86e
```

O que é uma tag?

Uma tag é uma referência a um commit que não esperamos alterar. Diferente do branch que esperamos que a cada novo commit tenha sua referência atualizada, uma tag é feita para nomear um commit específico e não uma linha de trabalho.

Olhar pasta `.git/refs/tags`

O que é um branch?

Um branch é um ramo da árvore do git. Em outras palavras, é uma linha de trabalho dentro do nosso repositório, como podemos ver no desenho do topo desta página.

Usamos esse ramo para separar uma linha de trabalho de outra. Basicamente, um branch é um nome que damos a uma linha que queremos ver atualizada a cada novo commit.

Olhar pasta `.git/refs/heads`

O que é o HEAD?

O HEAD é uma referência padrão do git que serve para sabermos qual commit estamos olhando e o que o git deve nos mostrar de conteúdo dentro da pasta de trabalho.

Olhar arquivo `.git/HEAD`

Manipulando as refs

Dado que as refs são arquivos, podemos, se quisermos, manipular essas referências manualmente com um editor de textos simples. Por exemplo, se formos à pasta `.git/refs/heads` e criarmos um novo arquivo, com um hash de um commit qualquer de nosso repositório estaremos efetivamente criando um novo ramo.

No entanto, não é recomendado fazer esse processo manualmente. Com os comandos abaixo, o git nos dá uma interface melhor do que ficar editando arquivos na mão para criar e manipular essas referências separadamente.

Trocando o HEAD

Para mudarmos o que estamos vendo na nossa pasta de trabalho, precisamos pedir ao git para trocar o commit que estamos olhando. Em outras palavras, precisamos trocar a referência do nosso **HEAD**. Para isso, usamos o comando `git checkout`.

Comandos

`git branch <nome-do-ramo>`

O comando branch cria uma nova referência do tipo branch apontando para o commit atual, ou seja, para o mesmo commit que o nosso **HEAD** está apontando no momento de sua execução.

`git tag`

O comando branch cria uma nova referência do tipo tag apontando para o commit atual, ou seja, para o mesmo commit que o nosso **HEAD** está apontando no momento de sua execução.

`git checkout`

O comando checkout muda o valor da referência do **HEAD**, efetivamente mudando o contexto da nossa pasta de trabalho.

Faça você mesmo:

```
git tag inicio
git tag

git branch dev
git checkout dev
```

- Edite o arquivo README.md
- Adicione modificação ao **stage**
- Crie um novo commit

Juntando o trabalho:

Da mesma forma que antes queriamos dividir nosso trabalho em diversas linhas, agora queremos juntar ou manipular essas linhas com suas diversas modificações.

Para isso, temos dois comandos principais: o `git merge` e o `git rebase`.

O git é muito bom para automaticamente saber como juntar as modificações, mas, em alguns casos em que ele não consegue resolver isso sozinho, são gerados conflitos que devem
ser resolvidos manualmente.

O que é um merge?

```
          o-o-o
         /     \
o-o-o-o-o-o-o-o-o (main)
```

Imagine o caso em que dois editores estão revisando um arquivo de texto em seções diferentes: um está editando o cabeçalho e outro o rodapé. Cada um deles passou horas editando sua parte e agora temos que juntar essas modificações.

Para isso, o git nos dá a opção de fundir duas ou mais linhas de trabalho em uma só.

Isso é um merge e normalmente o executamos sobre dois branchs diferentes.

No nosso exemplo, executaríamos algo como:

```
git merge principal cabecalho
git merge principal rodape
```

E teríamos no final um único arquivo contendo todas as modificações.

O que é um rebase

Imagine o caso em que um desenvolvedor criou uma nova função para um sistema, mas devido a certas urgências o branch dessa nova funcionalidade ficou abandonado por meses. Então um gestor resolveu que era a hora de juntar aquela função, mas antes disso era preciso terminar o desenvolvimento dela.

Em vez de continuar desenvolvendo em um ambiente que não reflete mais a situação do sistema, o desenvolvedor corta o seu galho da árvore e o recoloca no ponto mais atual do projeto.

Árvore do git após desenvolvimento de nova funcionalidade

```
          o-o-o-o (nova_funcionalidade)
         /
o-o-o-o-o-o (main)
```

Árvore do git após mais desenvolvimento no ramo principal

```
          o-o-o-o (nova_funcionalidade)
         /
o-o-o-o-o-o-o-o-o (main)
```

Árvore do git após rebase

```
                  o-o-o-o (nova_funcionalidade)
                 /
o-o-o-o-o-o-o-o-o (main)
```

Comandos

`git merge`

O git merge junta duas ou mais linhas de trabalho e suas modificações em um único ponto.

`git rebase`

O git rebase modifica o histórico dos commits, modificando qual o commit inicial daquela linha de trabalho.

O que é um conflito?

Um conflito é criado quando o git não sabe juntar as modificações de forma apropriada. Dessa forma, ele cria certos separados nos arquivos de texto para que manualmente seja resolvido como os arquivos devem ficar.

```
<<<<<<< HEAD
# Tarefa 1 do curso
=======
# Exercício 1 do curso git
>>>>>>> dev

Escreva um texto abaixo do título como desejar
Editção do corpo do texto
```

Resolvendo conflitos

Para resolver o conflito, basta editar os arquivos da forma que deseja que fiquem, adicioná-los ao stage e continuar o comando que estava sendo executado.

```
# Exercício 1 do curso git

Escreva um texto abaixo do título como desejar
Editção do corpo do texto
```

Faça você mesmo:

```
git checkout master
git merge dev
git log
```

Local e remote

O que é o local?

O `Local` nada mais é que o seu repositório git na sua máquina, ou seja, uma pasta de trabalho com a pasta `.git/`.

O que é um remote?

Um `remote` é um repositório git, mas em uma outra máquina como um servidor.

São muito utilizados serviços como github (https://github.com/), gitlab (https://gitlab.com/) e bitbucket (https://bitbucket.org/) como servidores de git para armazenar seus repositórios remotamente, mas, caso tenha interesse, também é possível instalar e gerenciar o seu próprio servidor de git.

Um repositório local pode estar conectado a vários remotes, mas é comum que o remote principal seja chamado de `origin`.

Um repositório remoto é usado de espaço para sincronizar o trabalho entre diversos autores, uma vez que todos podem mandar e baixar modificações desse repositório.

Comandos

`git remote`

Esse comando lista os seus remotes e através de certas opções também é possivel criar novos remotos ou mudar as URLs dos remotos já existentes. O comando lembra muito o `git branch`

`git clone`

O clone, como o próprio nome diz, clona um outro repositório para o seu computador. Dessa forma, você não tem que configurar uma nova pasta, um repositório local, linkar com o seu remote. Tudo isso já é feito com um único comando.

`git fetch`

O fetch busca todas as modificações e referências que estão registradas no remote e as traz para o seu repositório local. Permite dessa forma atualizar o seu local em relação ao seu remote.

Demonstrando o git fetch

`git push`

O git push envia os commits no branch atual para um branch remoto.

`git pull`

O git pull pega os commits novos do seu remoto para o seu branch local
No fundo um pull executa um fetch e um merge do branch remoto com o branch local.

Faça você mesmo:

- Crie um repositório novo em um servidor como o github
- Copie a url do repositório

```
git remote add origin <url>
git push --set-upstream origin master
```

- Olhe seu repositório no github.

Saiba mais

Sempre que tiver uma dúvida sobre um comando, lembre-se de olhar a documentação usando:

```
git help <comando>
```

A documentação é sempre a melhor forma de aprender a usar a ferramenta.

Padrões de commit

Existem times que adotam padrões específicos para a escrita de seus commits, o que foge do escopo deste curso, mas fica aqui a referência para alguns desses padrões:

- Convetional commits (https://www.conventionalcommits.org/en/v1.0.0-beta.2/)

Nomeando versões

Existem padrões de como você deve nomear a versão do seu software. Cada projeto pode adotar o seu próprio e seguem aqui alguns deles:

- Semantic Versioning (https://semver.org/)

O que são hooks?

Hooks são gatilhos que o git executa antes ou depois de certos eventos, por exemplo, o evento PRE-COMMIT
é executado um pouco antes de cada `git commit` que você executar. Dessa forma, você pode adicionar funcionalidades
customizadas no seu processo de versionamento. Um uso muito comum disso é usar os hooks para automaticamente reformatar
seu código antes de todo commit.

Os hooks na verdade são arquivos de script que são encontrados em `.git/hooks`

```
.git/hooks/
  applypatch-msg.sample
  commit-msg.sample
  fsmonitor-watchman.sample
  post-update.sample
  pre-applypatch.sample
  pre-commit.sample
  prepare-commit-msg.sample
  pre-push.sample
  pre-rebase.sample
  pre-receive.sample
  update.sample
```

Quem foi que fez isso?!?

Em algum momento você vai querer saber quem raios escreveu essa porcaria de código e o git rapidamente refrescará a
sua memória e mostrará que foi você mesmo quem escreveu isso... no mês passado.

`git blame`

Comando para ver quem, quando e em qual commit foram feitas alterações.

```
git blame visao-geral.md
```

```
2379cc16 (marceloll 2020-08-05 14:20:43 -0300  1) # Visão geral do git
2379cc16 (marceloll 2020-08-05 14:20:43 -0300  2)
c732a318 (marceloll 2020-08-05 15:04:39 -0300  3) 1. Configurar git (user.name,
user.email)
a23a48b2 (marcel0ll 2020-08-05 16:23:39 -0300  4) 1. Clonar repositório
git@github.com:marcel0ll/curso-git.git
a23a48b2 (marcel0ll 2020-08-05 16:23:39 -0300  5) 1. Modificar arquivo
README.md
a23a48b2 (marcel0ll 2020-08-05 16:23:39 -0300  6) 1. Adicionar README ao stage
a23a48b2 (marcel0ll 2020-08-05 16:23:39 -0300  7) 1. Criar commit
a23a48b2 (marcel0ll 2020-08-05 16:23:39 -0300  8) 1. Enviar modificação ao
github
2379cc16 (marceloll 2020-08-05 14:20:43 -0300  9)
2379cc16 (marceloll 2020-08-05 14:20:43 -0300 10) ---
2379cc16 (marceloll 2020-08-05 14:20:43 -0300 11)
2379cc16 (marceloll 2020-08-05 14:20:43 -0300 12)
2379cc16 (marceloll 2020-08-05 14:20:43 -0300 13) [Próximo](configurando-o-
git.md)
```

O que é o reflog?

O reflog é similar ao log, mas em vez de listar commits ele lista as mudanças feitas nas suas referências.

`git reflog`

Comando que lista modificações nas suas referências

`git reflog -10`

```
3ee69bc HEAD@{0}: commit: Preencha seção saber-mais.md
ca790df HEAD@{1}: commit: Mude ordem de tag e branch
cb014ca HEAD@{2}: reset: moving to HEAD
cb014ca HEAD@{3}: commit: Mude nome do arquivo para exercicio-1
abcc793 HEAD@{4}: commit: Atualize fluxos de trabalho com git
3b1a7fa HEAD@{5}: reset: moving to HEAD
3b1a7fa HEAD@{6}: commit: Adicione passo para mudança de diretório do projeto
clonado
e4b31b9 HEAD@{7}: commit: Remova arquivo chunks.md
7593f38 HEAD@{8}: commit: Mude exercicio de lista ordenada para lista de pontos
a23a48b HEAD@{9}: commit: Simplifique visao geral
```

O que é o stash?

O stash é um salva-vidas para quando você tem algumas modificações no seu repositório e quer trocar de branchs sem ter
que criar um novo commit. Então você as joga para o stash, faz o que precisa fazer sem maiores
conflitos e depois você as desempilha do seu stash.

Tome cuidado ao usá-lo. Às vezes colocamos modificações no stash para nunca mais lembrarmos delas.

`git stash`

Empilha modificações no `stash`

`git stash pop`

Desempilha modificações do `stash`

O que é cherry-pick?

Cherry-pick é um comando muito pontual. Ele é como se fosse um rebase de um commit só para qualquer lugar do seu
histórico. Você literalmente copia um commit de um lugar e o encaixa em outro commit.

`git cherry-pick`

Recria commit em outra posição do histórico

O que é bisect?

O bisect auxilia você a pesquisar de forma binária uma linha de trabalho entre 2 commits. O git dá um checkout para
certos commits intermediários perguntando se era essa a versão que você estava procurando até que você o encontre.

`git bisect`

Executa uma busca binária interativa à procura de algum commit específico.

Fluxos de trabalho para git:

Para saber mais sobre fluxos de trablho de git, leia [isto.](https://www.atlassian.com/git/tutorials/comparing-workflows)
(https://www.atlassian.com/git/tutorials/comparing-workflows)

No futuro, essa seção será melhor detalhada.

Exercício 1:

Segue abaixo uma série de passos para exercitar os conceitos apresentados:

- Configurar git (user.name, user.email)
- Criar nova pasta `pc1-curso-git-ex-1`
- Iniciar repositório local
- Mostrar Status
- Editar arquivo README.md:

```
# Tarefa do curso
```

- Mostrar Status

- Adicionar README.md ao stage
- Mostrar Status
- Criar commit com mensagem:

```
Crie arquivo de README

Coloque um título no arquivo do projeto
```

- Mostrar log
- Editar arquivo README.md:

```
# Tarefa do curso

Escreva um texto abaixo do título como desejar
```

- Adicionar README.md ao stage
- Criar commit com mensagem:

```
Adicione texto no README

Adicione um texto exemplificando uma edição de arquivo de texto
```

- Mostrar log
- Criar repositório no Github (curso-git-ex-1)
- Configurar no pc1/curso-git-ex-1 o remote origin

```
git remote add origin <url-do-repositório>
```

- Mandar para remote as mudanças local/master

```
git push --set-upstream origin master
```

- Criar branch dev
- Mudar para branch dev
- Editar arquivo README.md:

```
# Exercício 1 do curso git

Escreva um texto abaixo do título como desejar

Escreva um novo parágrafo no seu texto
```

- Adicionar titulo do README.md ao stage
- Criar commit com mensagem:

```
Edite título do README
```

- Adicionar corpo do README.md ao stage
- Criar commit com mensagem:

```
Edite corpo do README
```

- Mostrar log
- Mandar para remote as mudanças local/dev
- Clonar repositório em outra pasta `pc2-curso-git-ex-1`

```
git clone <sua-url> pc2-curso-git-ex-1
```

- Mude para pasta pc2-curso-git-ex1
- Editar arquivo README.md:

```
# Tarefa 1 do curso

Escreva um texto abaixo do título como desejar
```

- Adicionar README.md ao stage
- Criar commit com mensagem:

```
Arrume título do arquivo README

Alguém esqueceu que teremos mais de uma tarefa no curso
```

- Enviar modificações ao github
- Voltar à pasta pc1/curso-git-ex-1
- Pegar atualizações do remote
- Mudar para branch `master`
- Baixar modificações do `origin/master` para `master`
- Dar merge do branch dev
- Resolver conflito no README.md
- Executar commit
- Mandar modificações para o github

Ao finalizar a lista de passos acima, postar o link de seu repositório público no [github (https://github.com/marcel0ll/curso-git/issues/1)](https://github.com/marcel0ll/curso-git/issues/1)
