

# Git avançado

boas práticas e como usá-lo com mais eficiência

---

Matheus Tavares


matheus.bernardino@usp.br

MAC0475

# Agenda

1. Como o Git funciona?
2. Como os comandos operam?
  - a. Checkout e Branch
  - b. Merge
  - c. Rebase
3. Boas práticas de commits
4. Trabalhando com *remotes*
5. *Workflows* e modelos de *branching*
6. CI / CD

Aumenta  
o nível de  
abstração



# Zen of Git

```
$ git checkout this
```

Zen of Git

Ugly is better than beautiful.

Explicit is better than implicit.

Complex is better than simple.

Complicated is better than complex.

Flat is better than nested.

Readability is meaningless.

Special cases are everything.

Errors should never be comprehensible.

In the face of ambiguity, copy and paste from Stack Overflow.

There should be no obvious way to do it.

Although there may be endless non obvious ways to do it.

If the documentation is hard to understand, it's a great idea.

If the documentation is easy to understand, it's probably for another tool.

<https://tdhopper.com/blog/zen-of-git>

# Zen of Git

```
$ git checkout this
```

Zen of Git

Ugly is better than beautiful.

Explicit is better than implicit.

Complex is better than simple.

Complicated is better than complex.

Flat is better than nested.

Readability is meaningless.

Special cases are everything.

Errors should never be comprehensible.

In the face of ambiguity, copy and paste from Stack Overflow.

There should be no obvious way to do it.

Although there may be endless non obvious ways to do it.

If the documentation is hard to understand, it's a great idea.

If the documentation is easy to understand, it's probably for another tool.

<https://tdhopper.com/blog/zen-of-git>



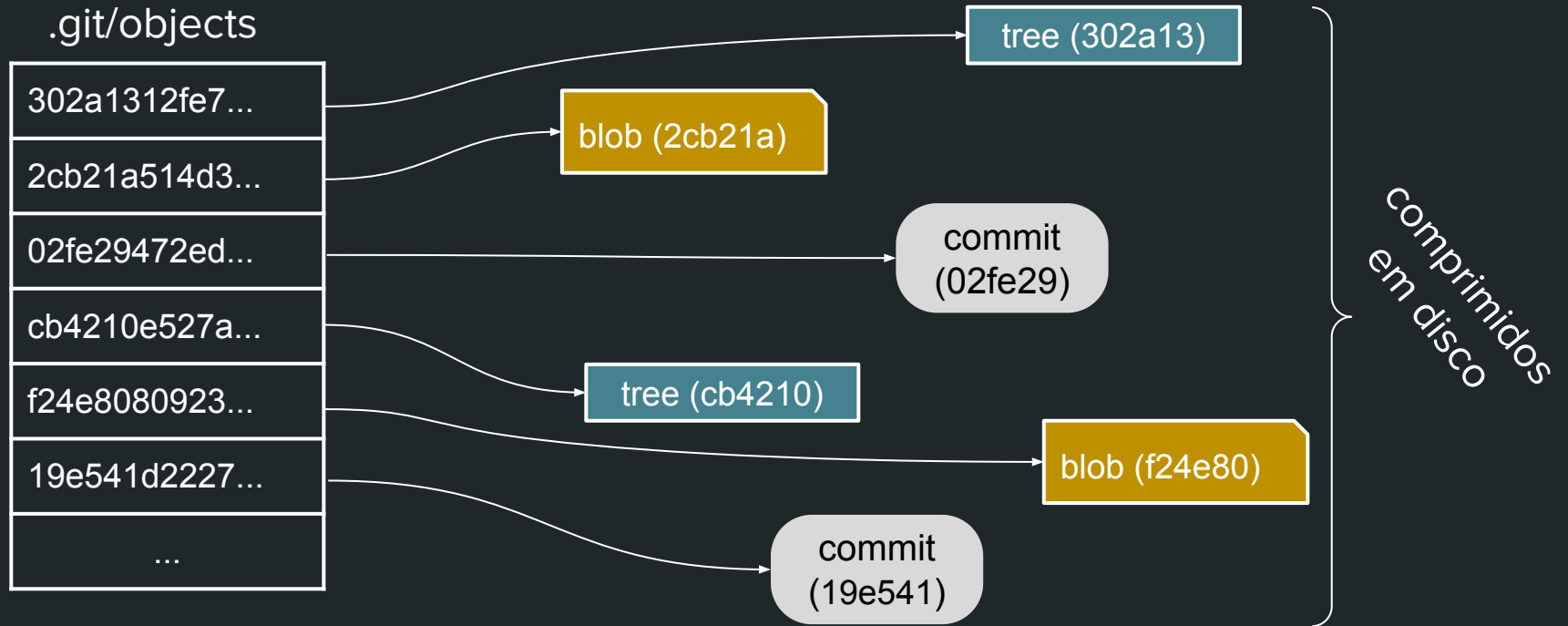
<http://www.quickmeme.com/meme/3t6lr9>



<https://xkcd.com/1597/>

# Um pouco de como o Git funciona

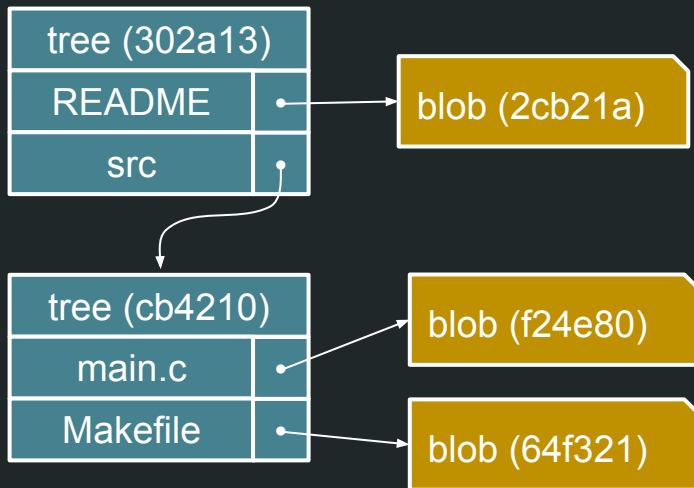
# Uma grande HashTable



# Relações entre os objetos

\$ tree repo

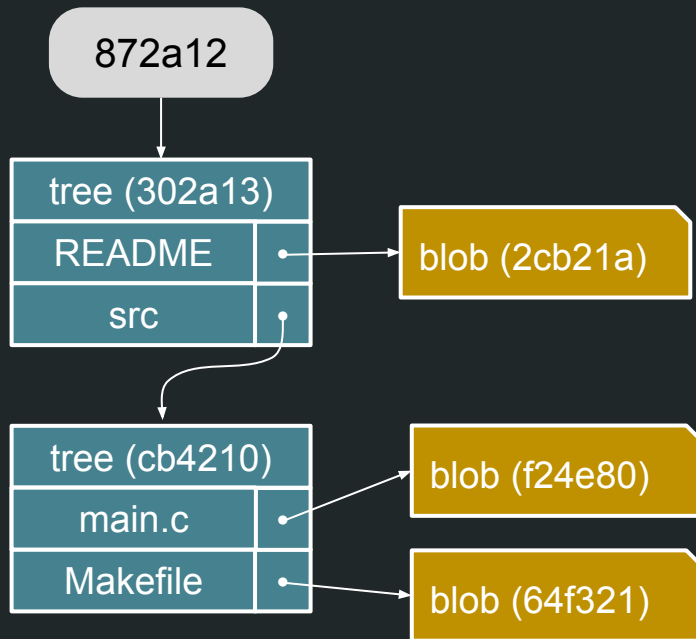
```
repo
├── README
└── src
    ├── main.c
    └── Makefile
```





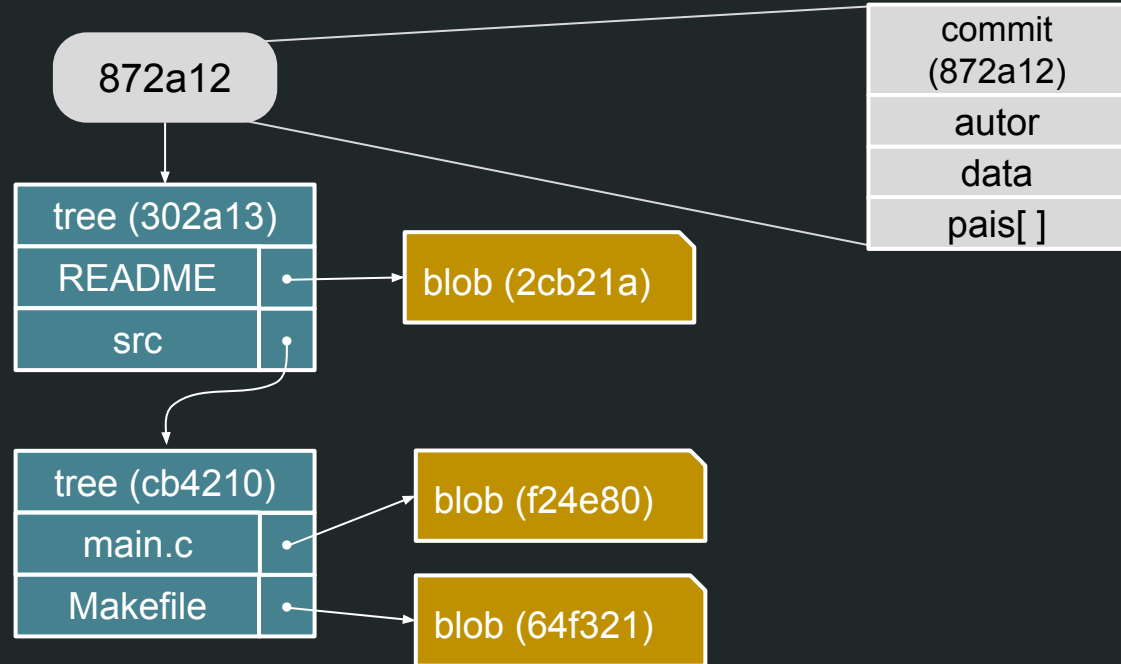
# Relações entre os objetos

- Um *commit* guarda o estado atual do projeto (uma *tree*, não um *diff*).



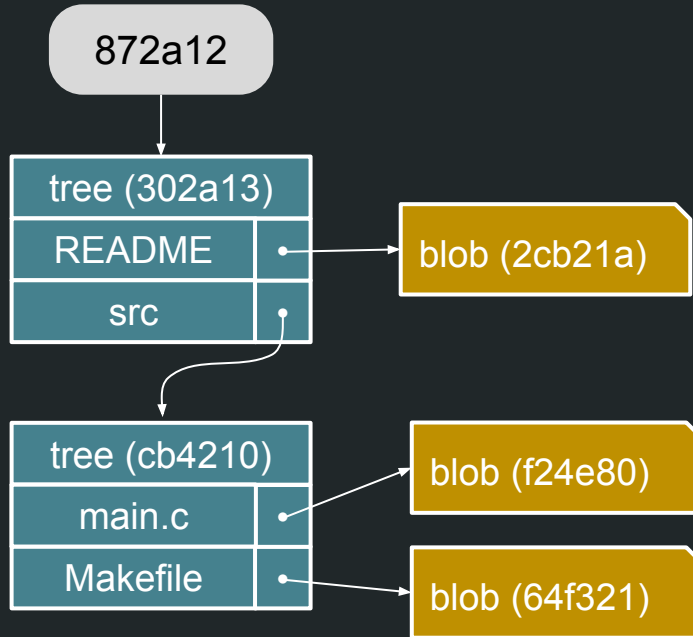
# Relações entre os objetos

- Commits também contém um conjunto de metadados.



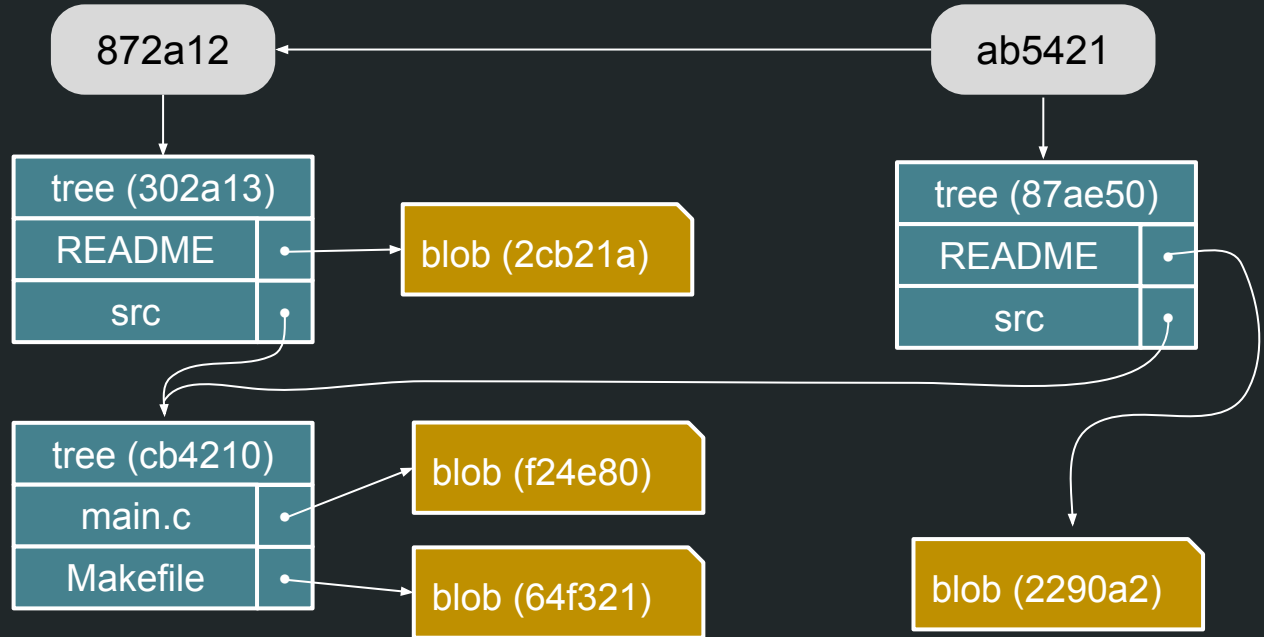
# Relações entre os objetos

\$ vim README  
\$ git commit



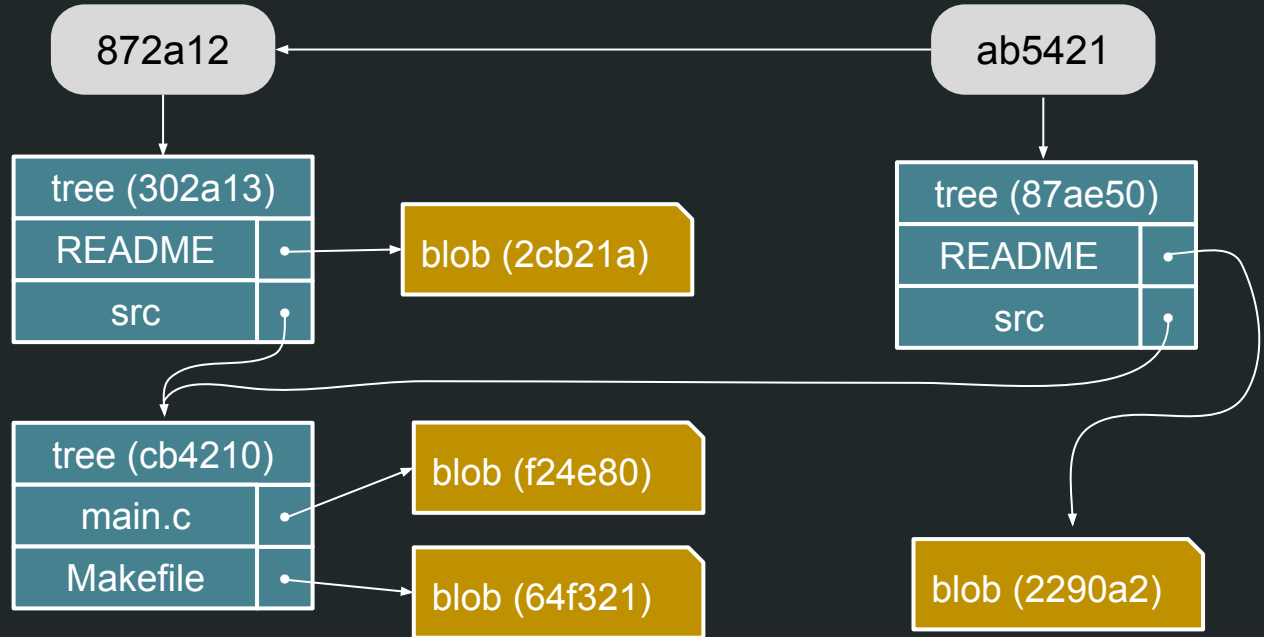
# Relações entre os objetos

\$ vim README  
\$ git commit



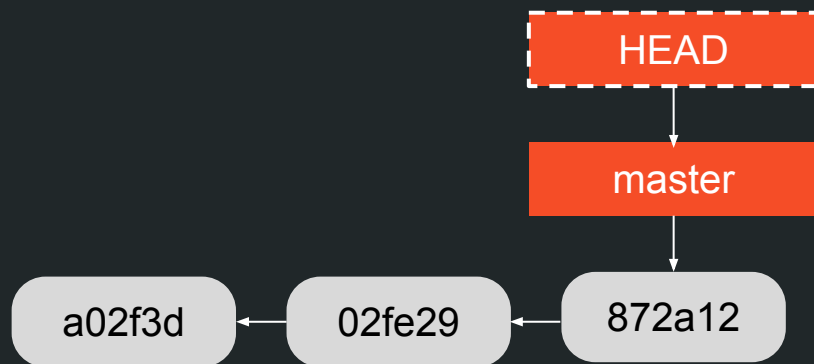
# Grafo Acíclico Dirigido (DAG, pros íntimos)

\$ vim README  
\$ git commit



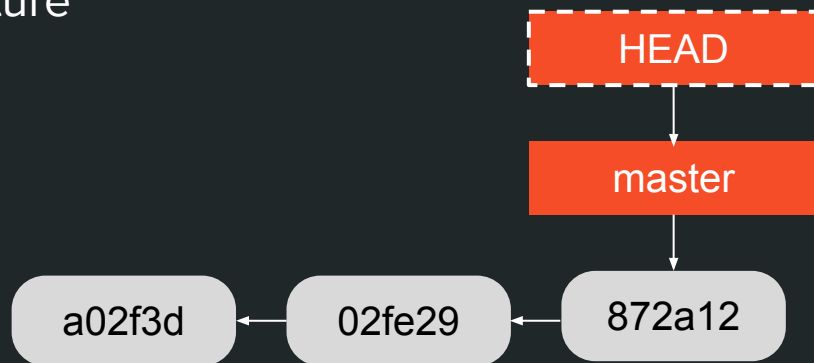
# Referências

- Branches
- Tags
- HEAD



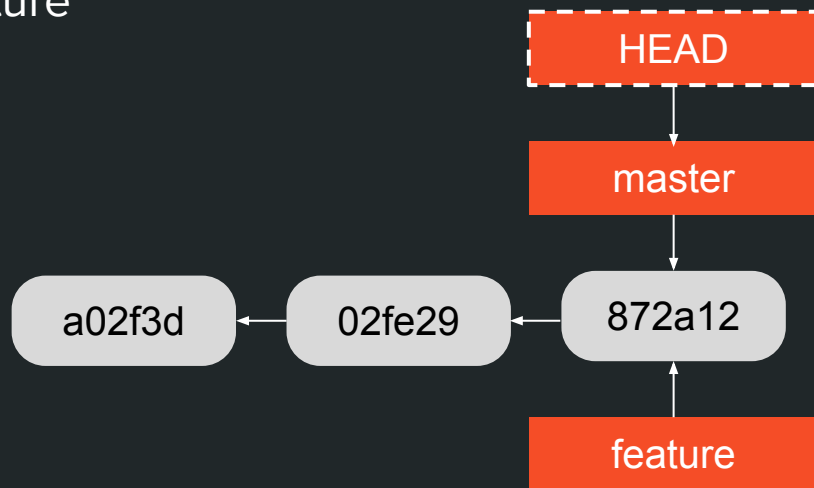
# Referências: branches

\$ git branch feature



# Referências: branches

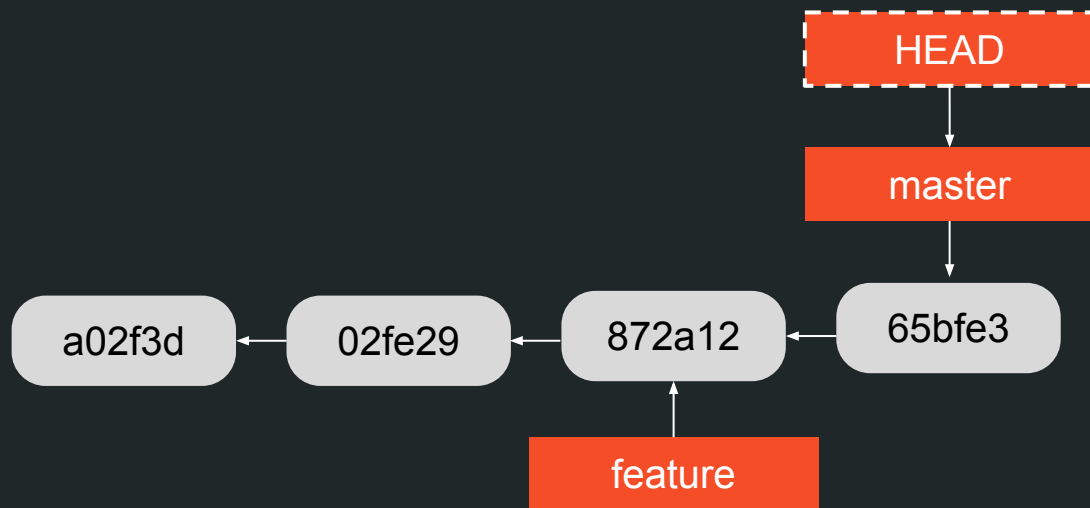
\$ git branch feature





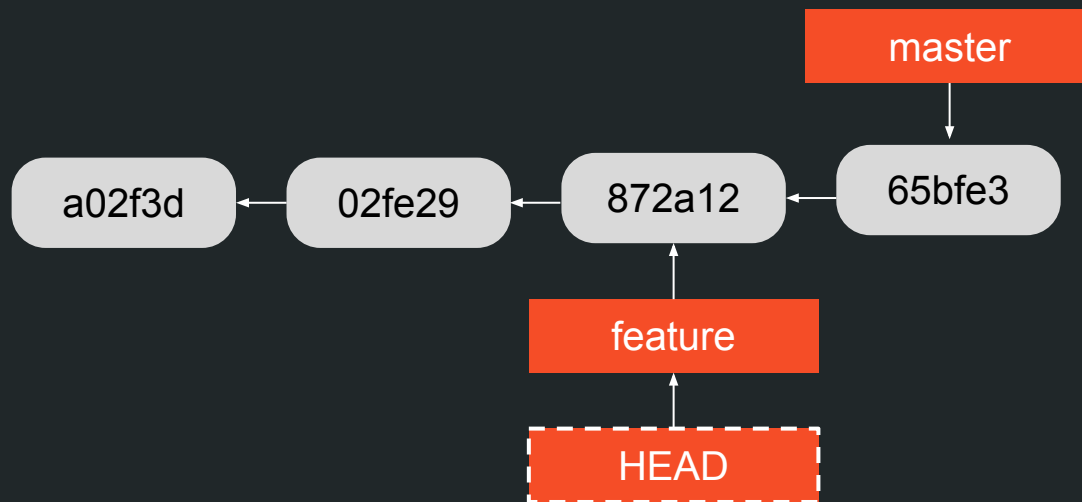
# Referências: branches

\$ git commit



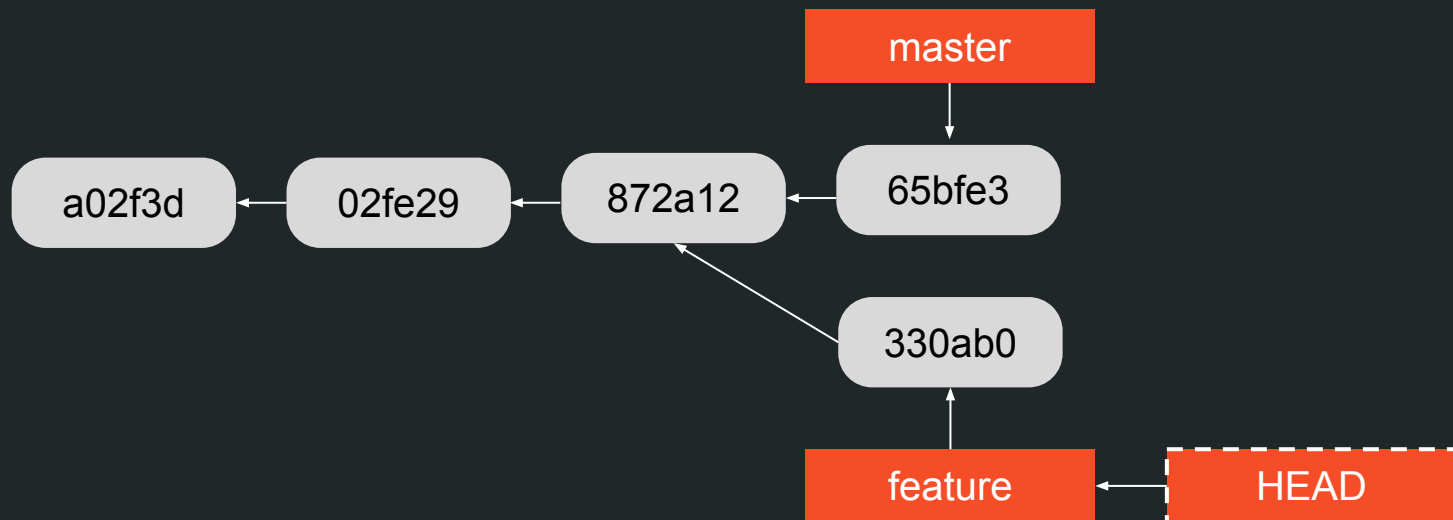
# Referências: branches

\$ git checkout feature



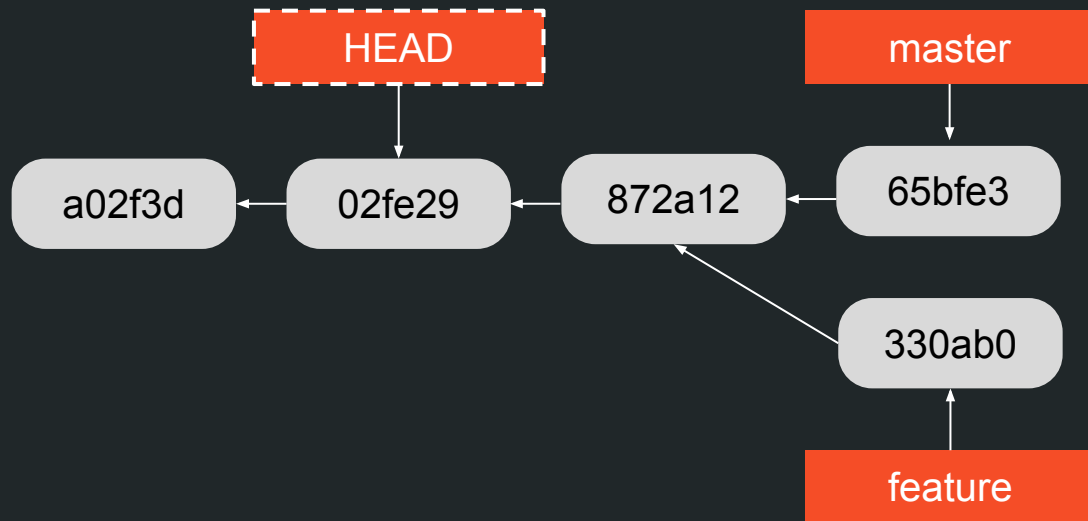
# Referências: branches

\$ git commit



# Referências: branches

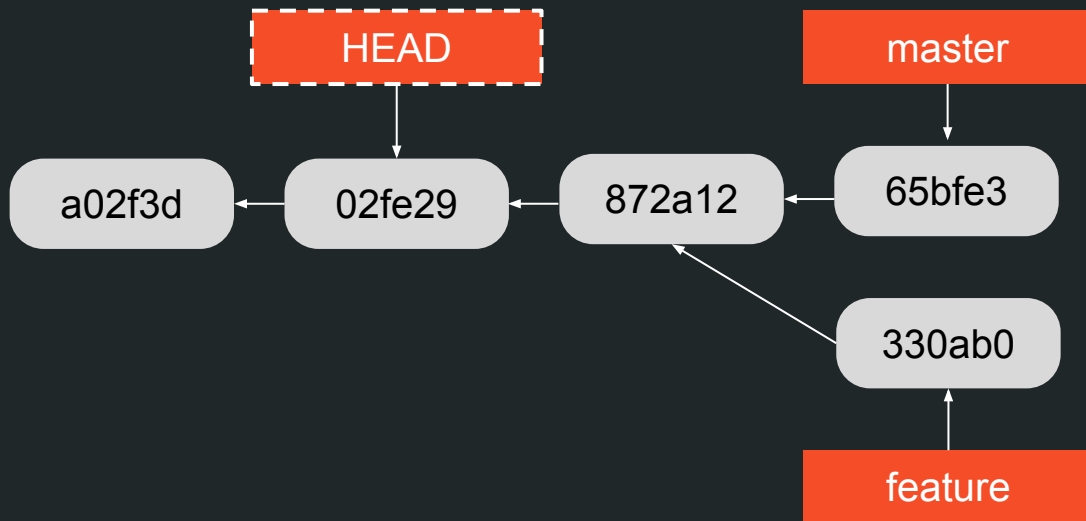
\$ git checkout 02fe29



# Referências: branches

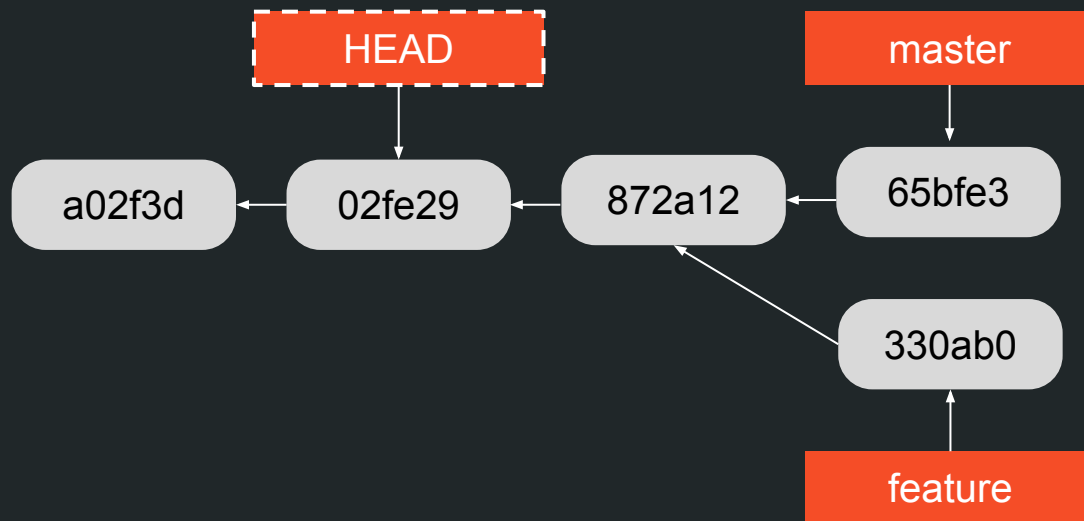
\$ git checkout 02fe29

*You are in “detached HEAD” state.*



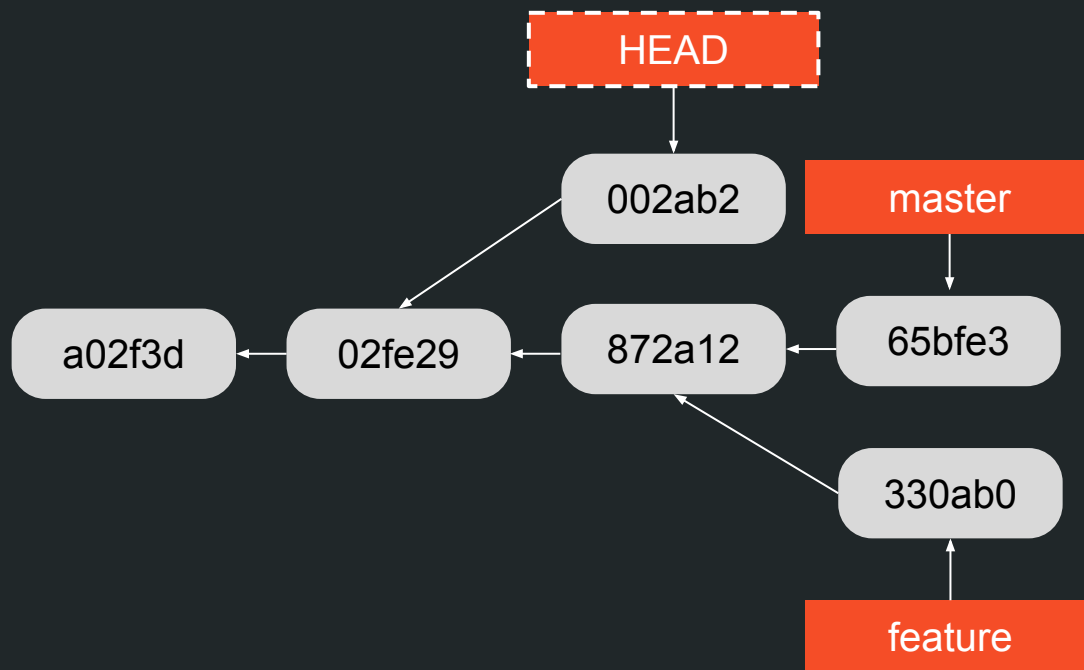
# Referências: branches

\$ git commit



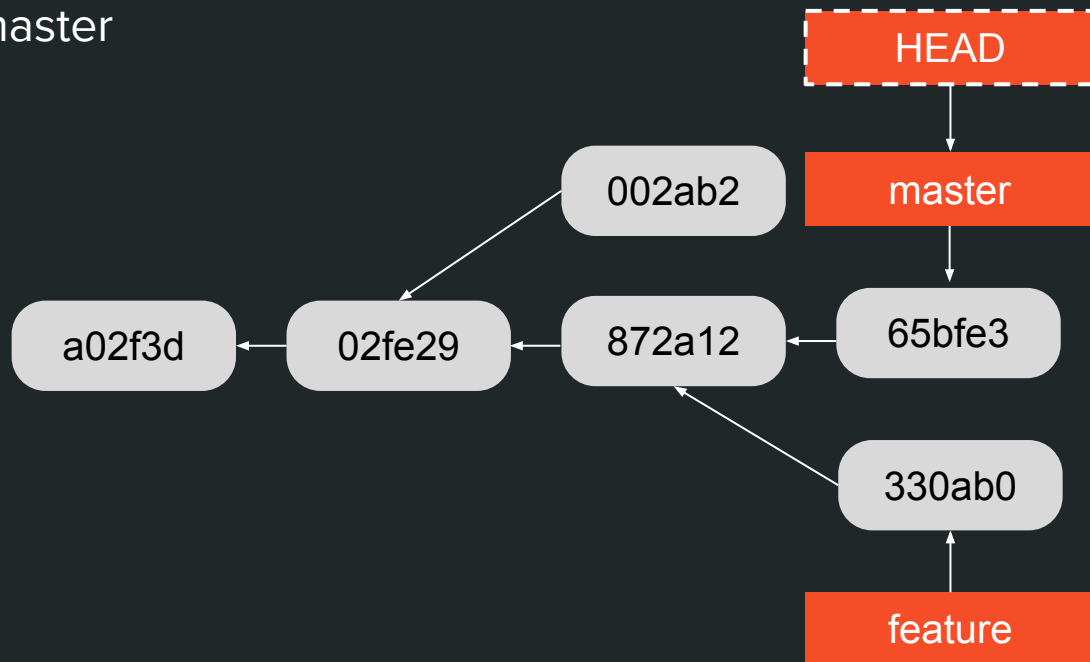
# Referências: branches

\$ git commit



# Referências: branches

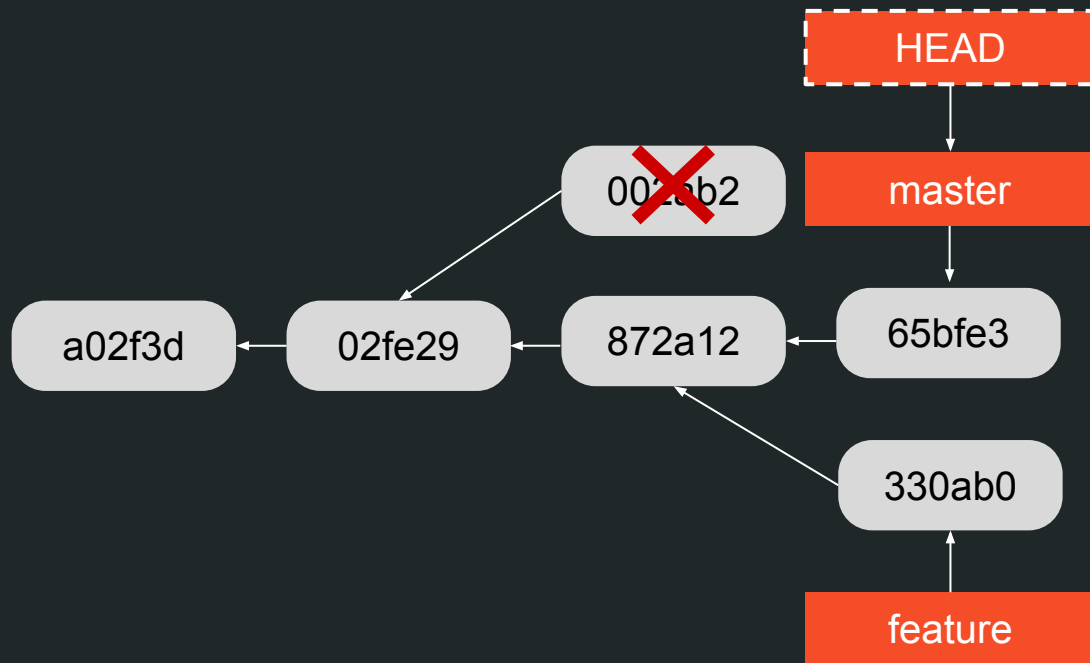
\$ git checkout master





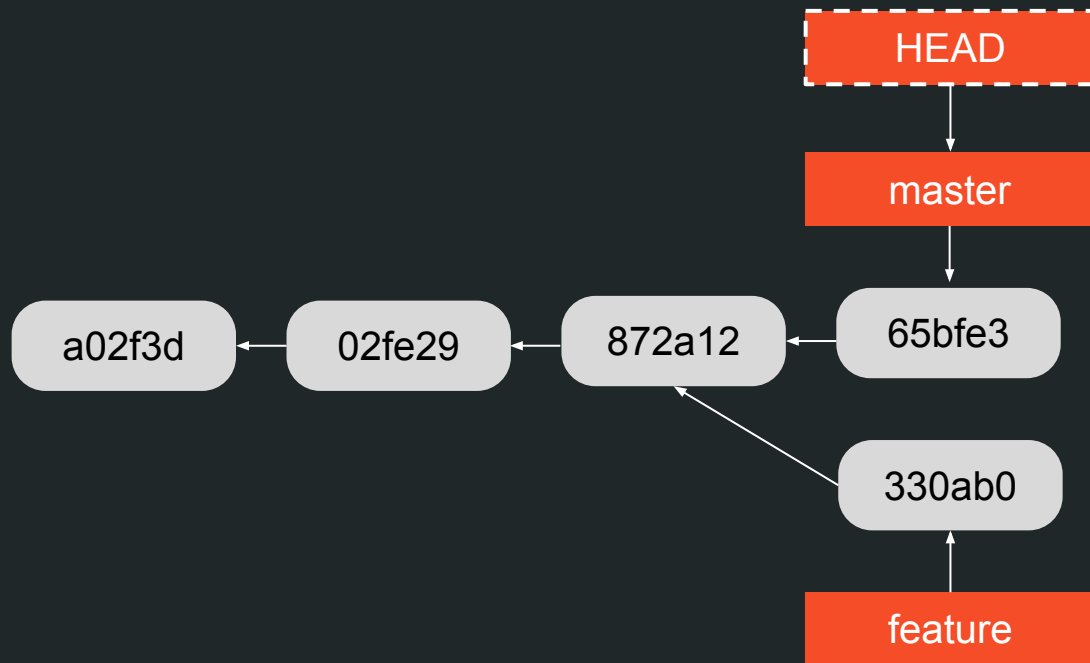
# Referências: branches

\$ git gc --auto



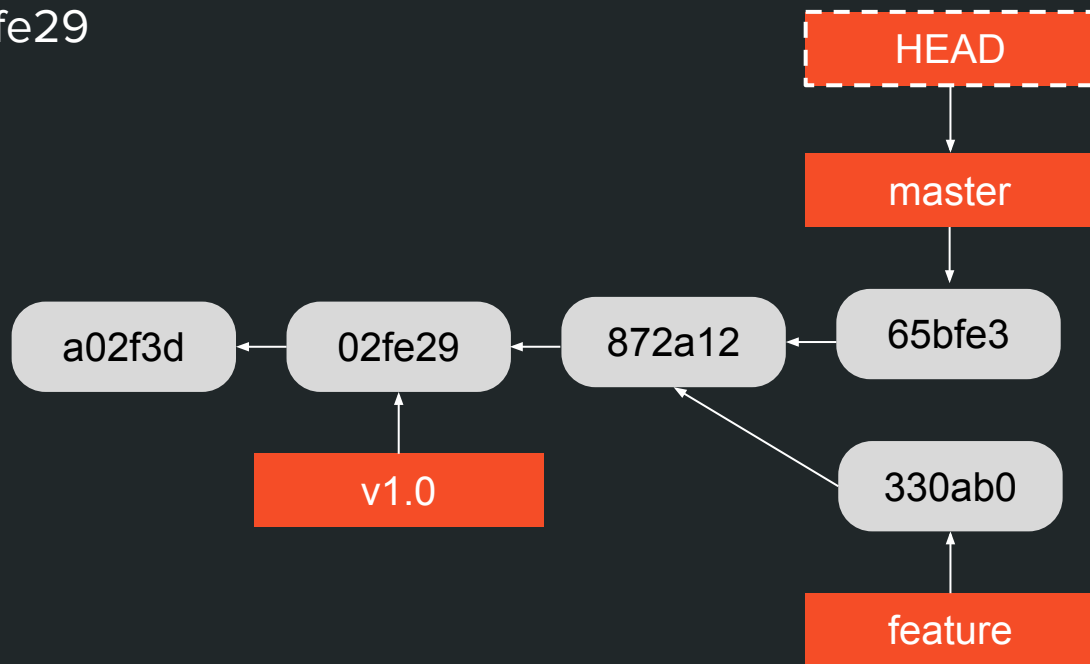
# Referências: branches

\$ git gc --auto



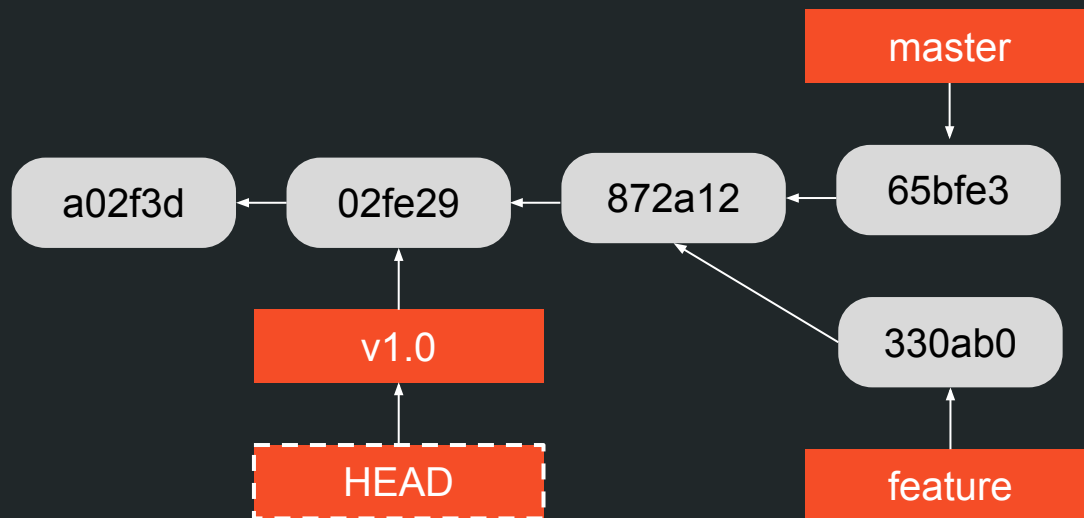
# Referências: tags

```
$ git tag v1.0 02fe29
```



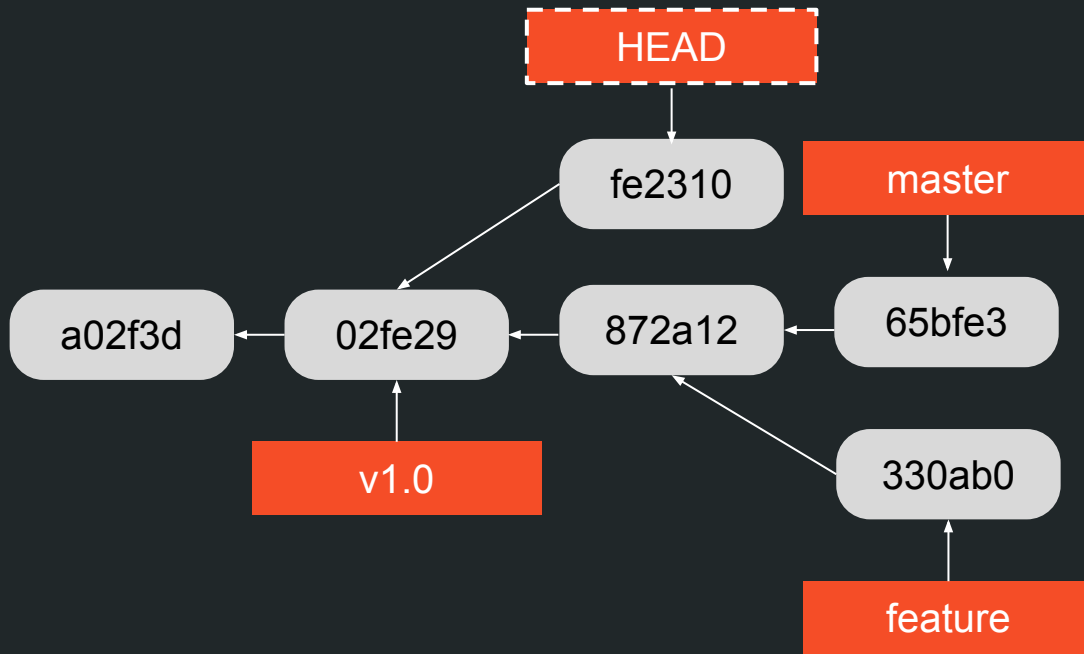
# Referências: tags

\$ git checkout v1.0



## Referências: tags

```
$ git commit
```



# Merging

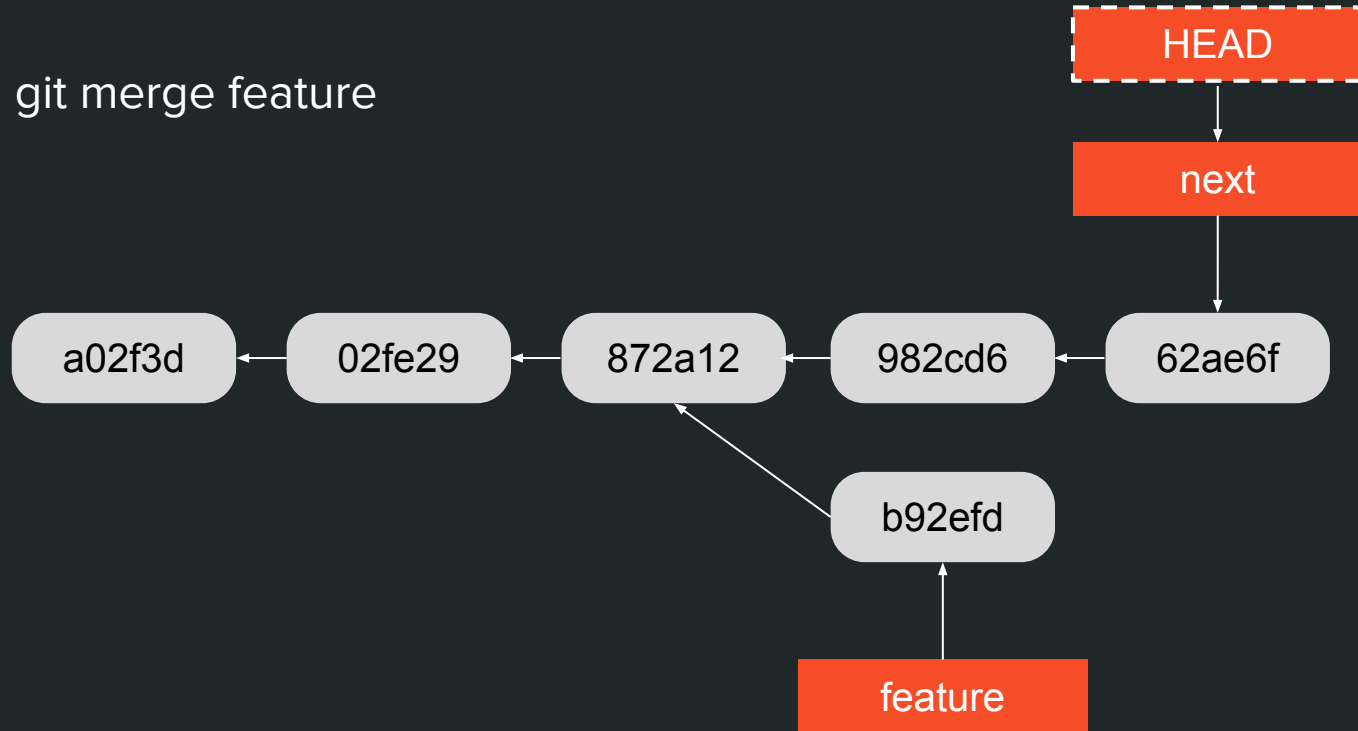
# **GIT MERGE**



<https://giphy.com/gifs/git-merge-cFkiFMDg3iFol>

# Merging

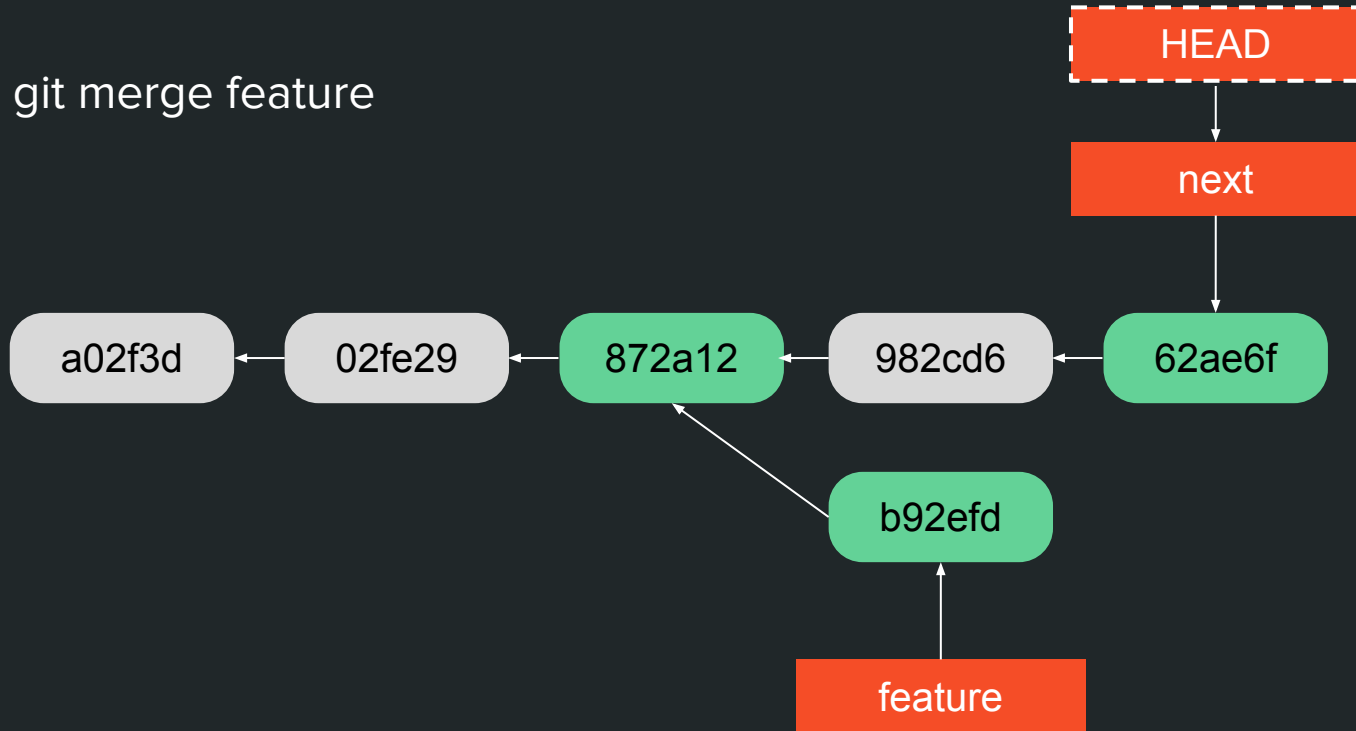
\$ git merge feature





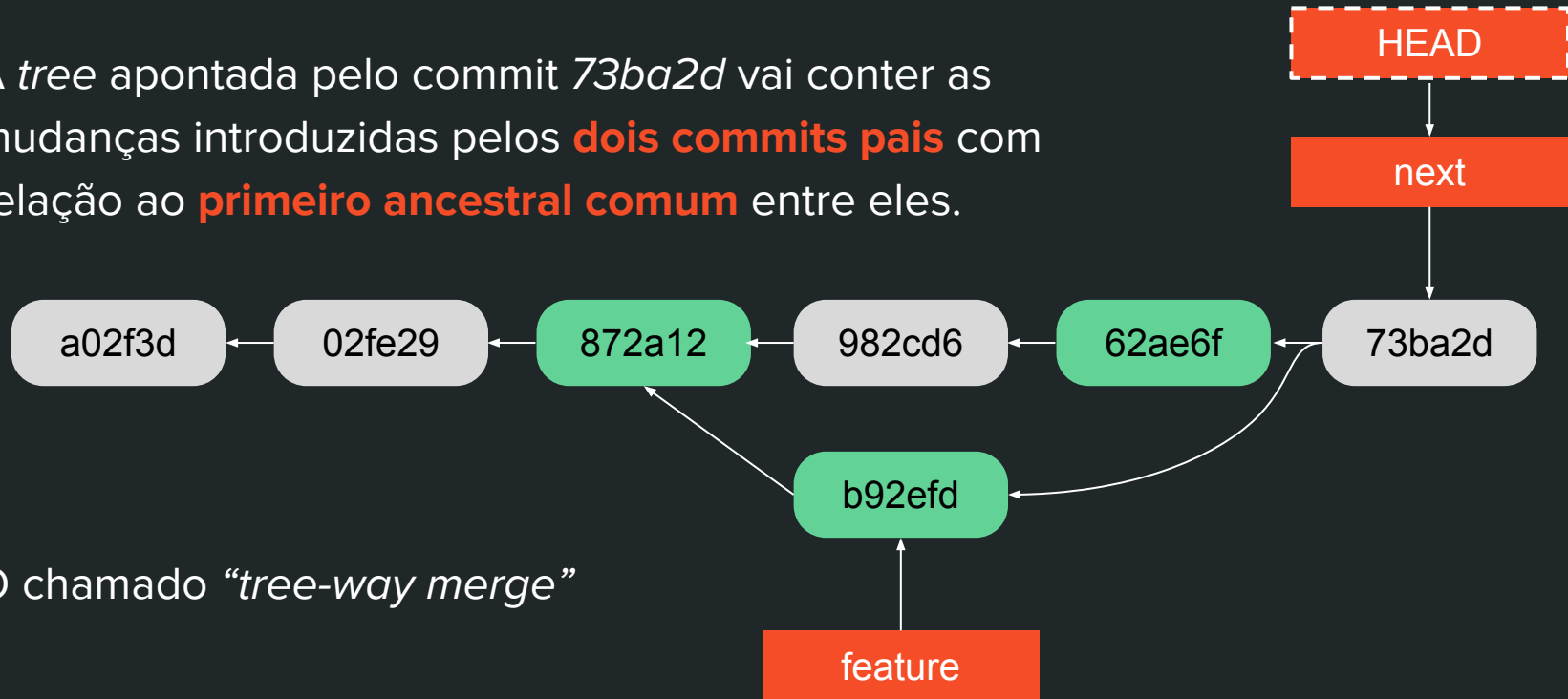
# Merging

\$ git merge feature



# Merging

A *tree* apontada pelo commit `73ba2d` vai conter as mudanças introduzidas pelos **dois commits pais** com relação ao **primeiro ancestral comum** entre eles.



O chamado “*tree-way merge*”

# Three-way merging

## Base

```
...  
if (enemy == "grievous")  
    printf("hello\n");
```

## next

```
...  
if (enemy == "grievous")  
    printf("hello there\n");
```

## feature

```
...  
if (enemy == "grievous" && am_I_obiwan())  
    printf("hello\n");
```

## Result

```
...  
if (enemy == "grievous" && am_I_obiwan())  
    printf("hello there\n");
```

# Three-way merging

## Base

```
...  
if (enemy == "grievous")  
    printf("hello\n");
```

## next

```
...  
if (enemy == "grievous")  
    printf("hello there\n");
```

## feature

```
...  
if (enemy == "grievous" && am_I_obiwan())  
    printf("Olá\n");
```

# Three-way merging

## Base

```
...  
if (enemy == "grievous")  
    printf("hello\n");
```

## next

```
...  
if (enemy == "grievous")  
    printf("hello there\n");
```

## feature

```
...  
if (enemy == "grievous" && am_I_obiwan())  
    printf("Olá\n");
```

## Result

```
Auto-merging greetings.c  
CONFLICT (content): Merge conflict in greetings.c  
Automatic merge failed; fix conflicts and then commit the result.
```

# Three-way merging

## Base

```
...  
if (enemy == "grievous")  
    printf("hello\n");
```

## next

```
...  
if (enemy == "grievous")  
    printf("hello there\n");
```

## feature

```
...  
if (enemy == "grievous" && am_I_obiwan())  
    printf("Olá\n");
```

## Result

```
Auto-merging greetings.c  
CONFLICT (content): Merge conflict in greetings.c  
Automatic merge failed; fix conflicts and then commit the result.
```

# Resolvendo conflitos

```
...
if (enemy == "grievous" && am_I_obiwan())
<<<<<<< HEAD
    printf("hello there\n");
=====
    printf("Olá\n");
>>>>>>> feature
```

- Entenda porque *next* e *feature* modificaram aquela linha.
- Escolha o lado a ser mantido, ou
- Faça uma “mistura” dos dois.

# Resolvendo conflitos

```
...  
if (enemy == "grievous" && am_I_obiwan())  
    printf("Olá a todos\n")
```

- Entenda porque *next* e *feature* modificaram aquela linha.
- Escolha o lado a ser mantido, ou
- Faça uma “mistura” dos dois.



# Resolvendo conflitos

```
...  
if (enemy == "grievous" && am_I_obiwan())  
    printf("Olá a todos\n")
```

```
$ git add greetings.c
```

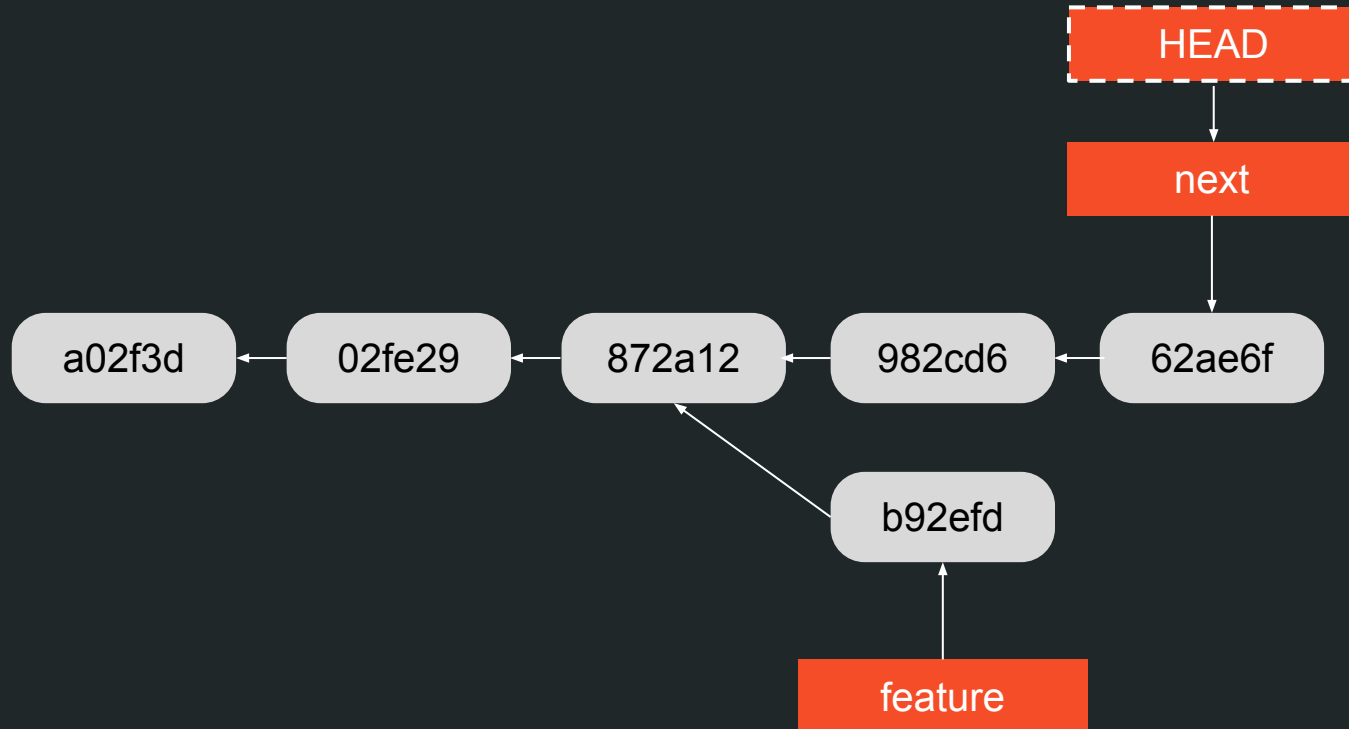
```
$ git merge --continue
```

***Rebase:***  
alterando o  
passado



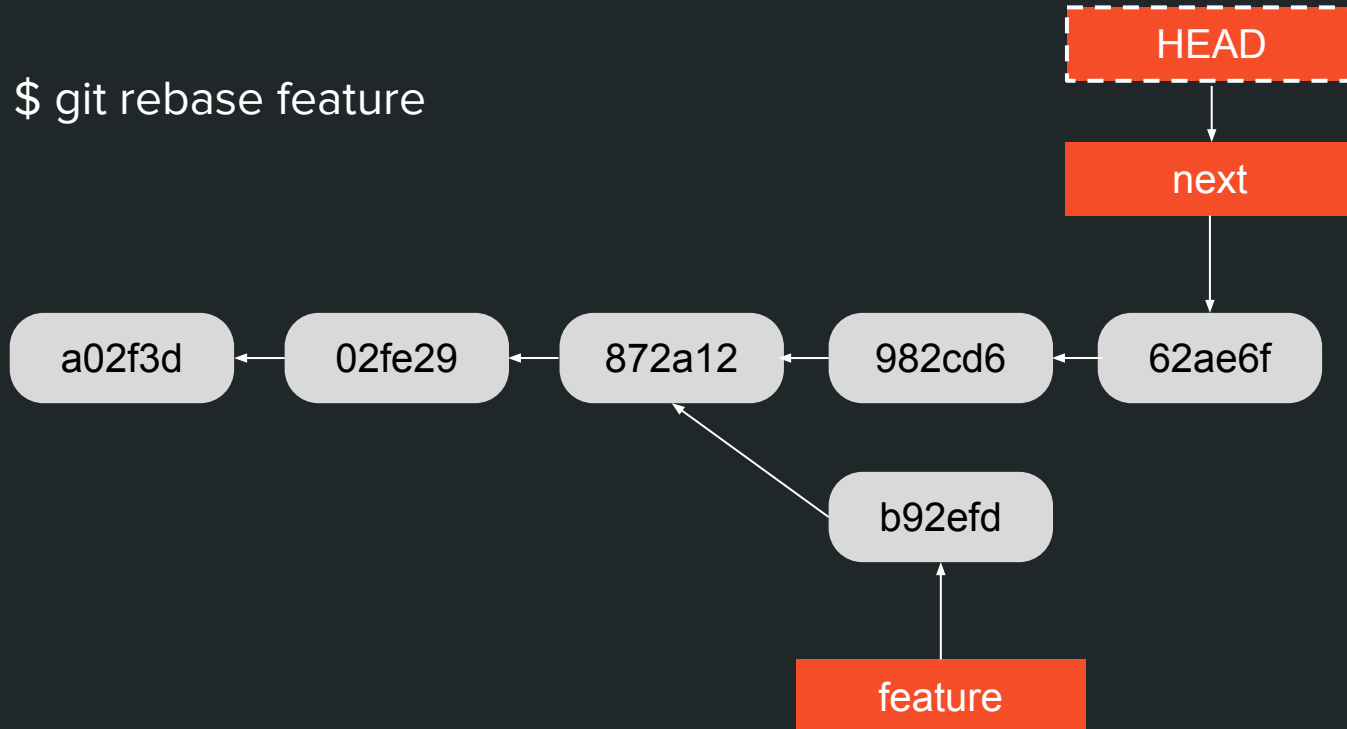
<https://memegenerator.net>

# Rebase



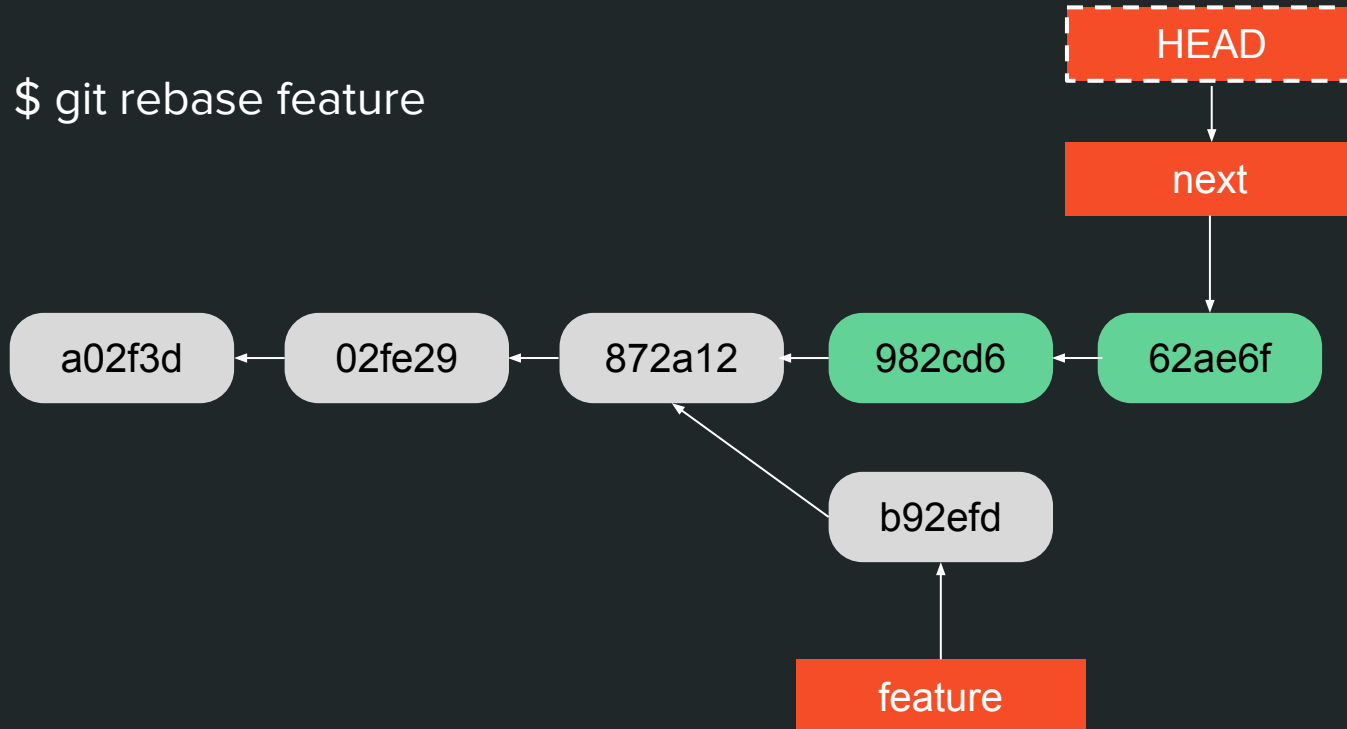
# Rebase

\$ git rebase feature



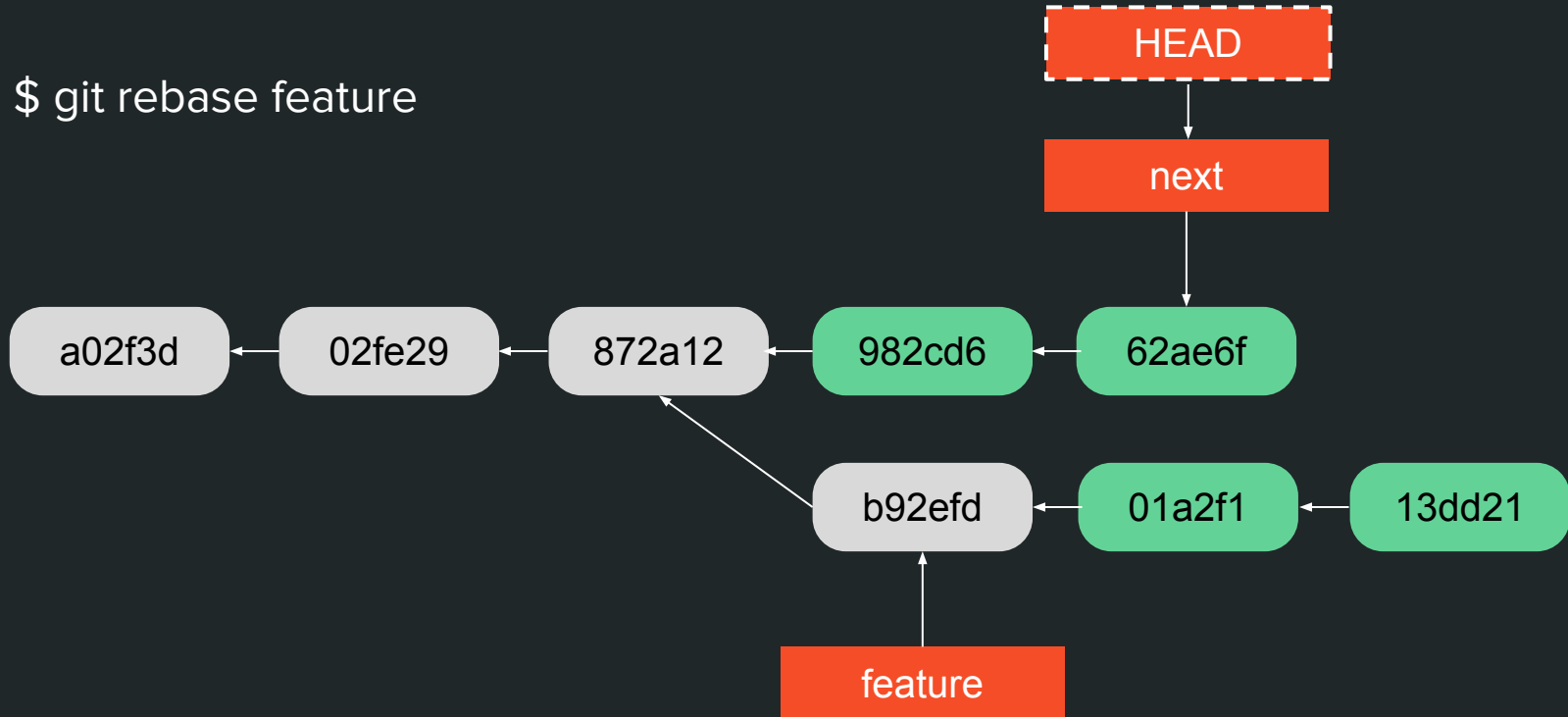
# Rebase

\$ git rebase feature



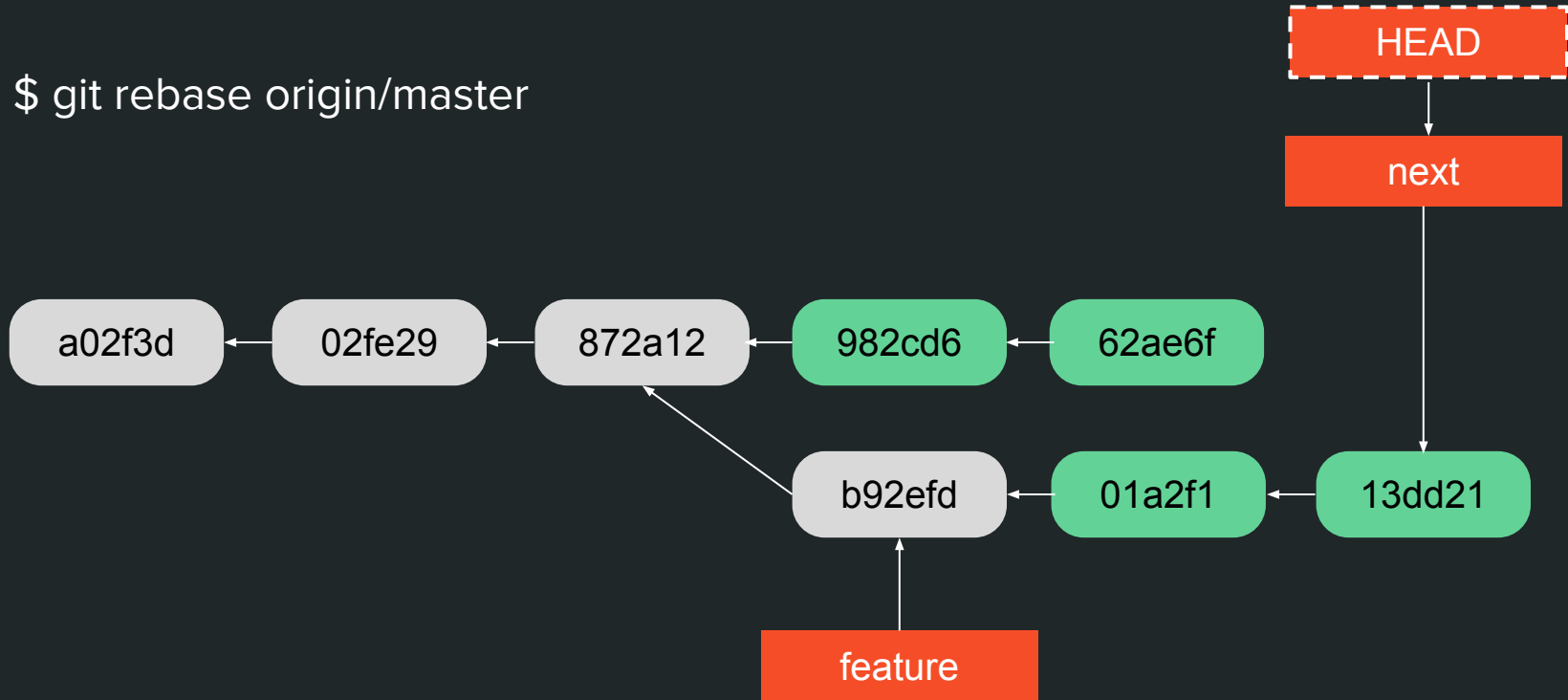
# Rebase

\$ git rebase feature



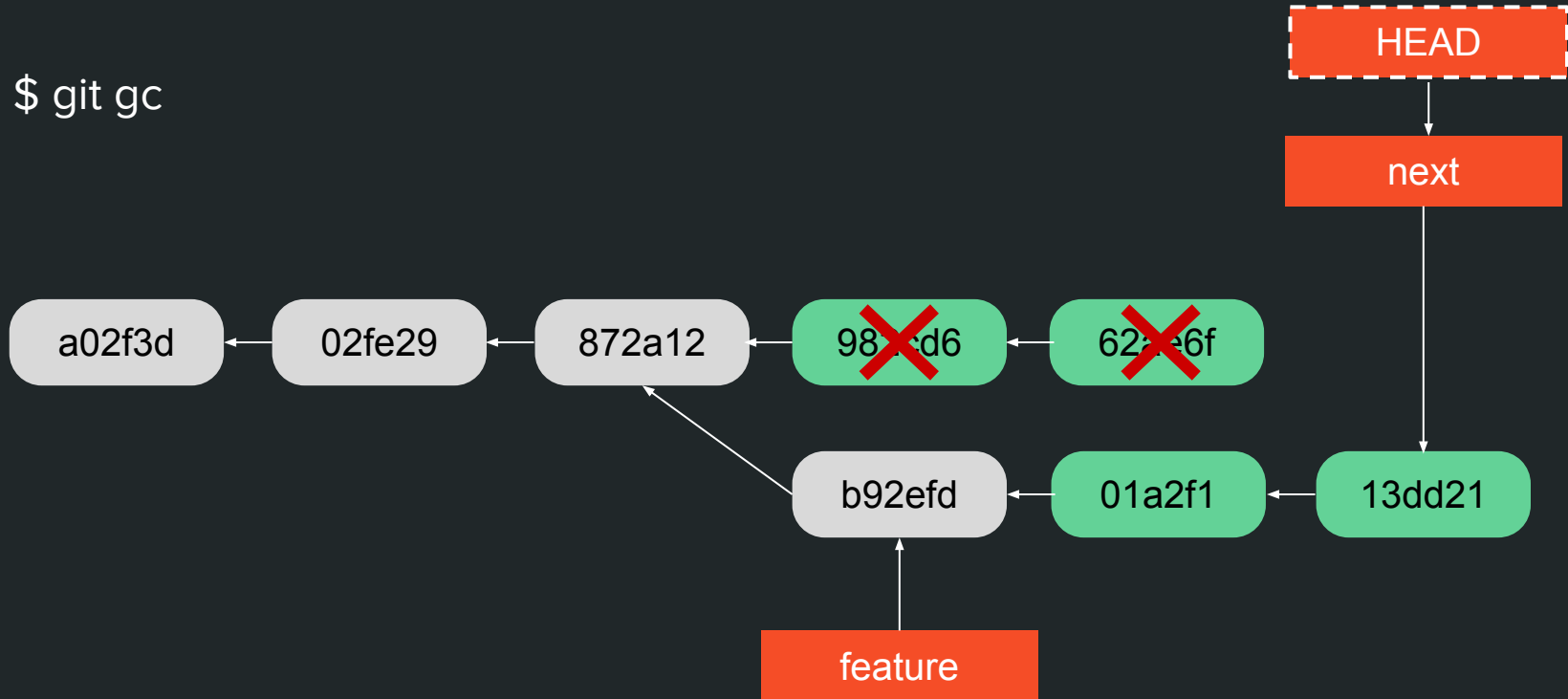
# Rebase

\$ git rebase origin/master



# Rebase

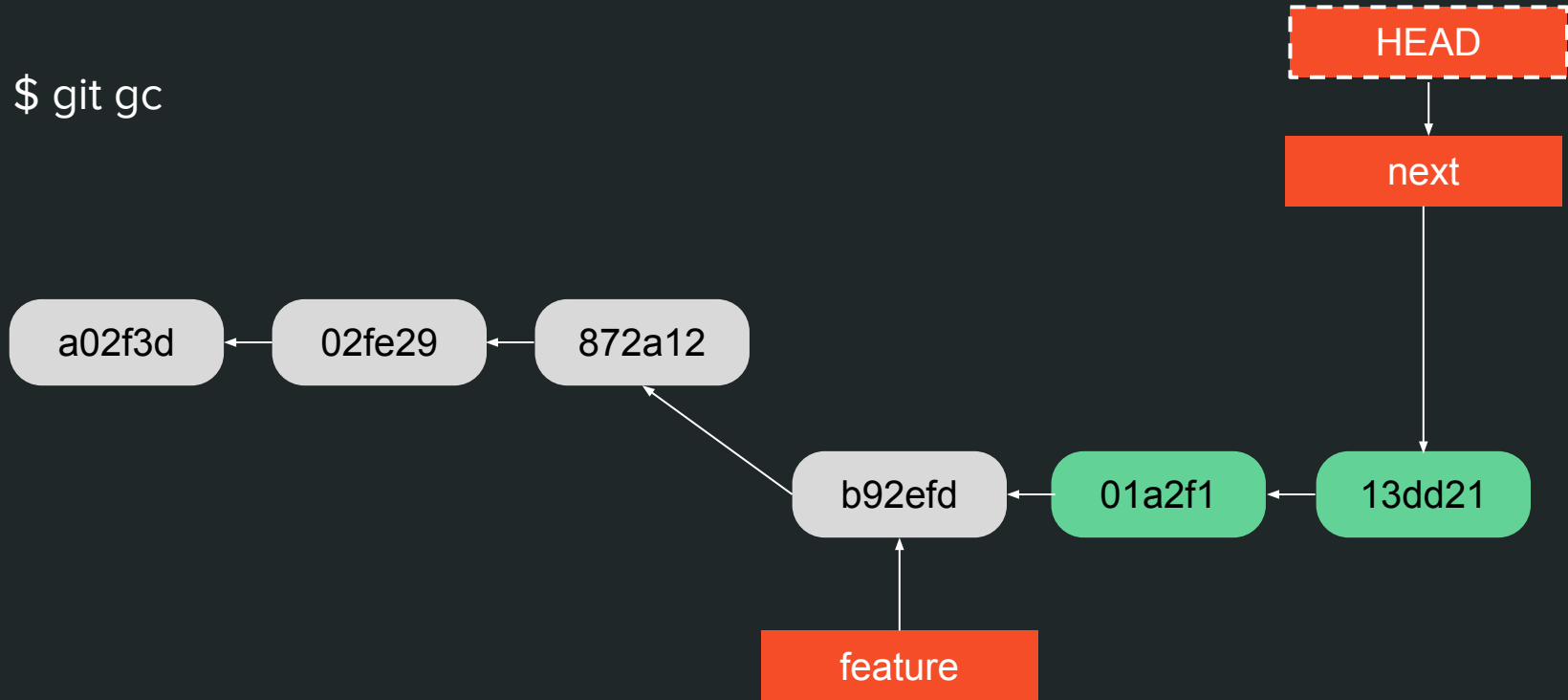
\$ git gc





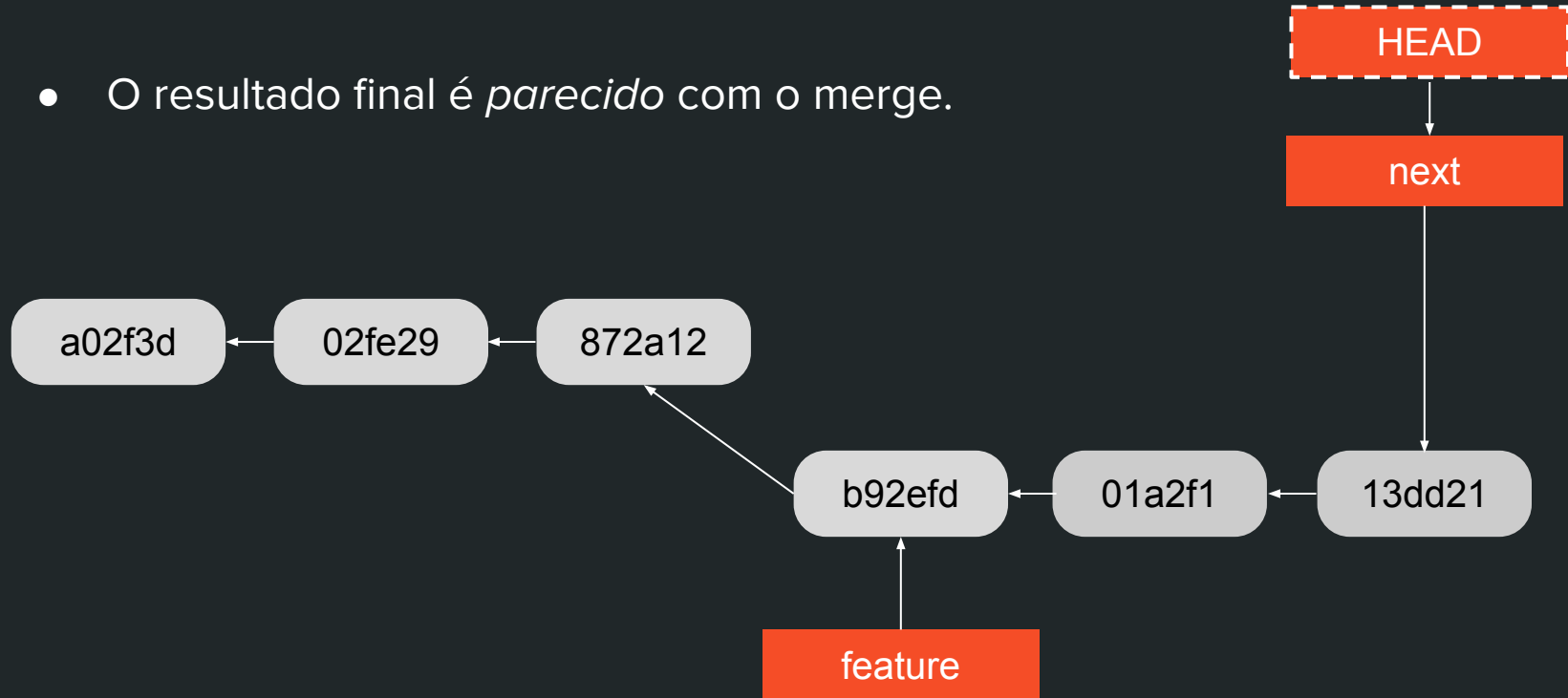
# Rebase

\$ git gc



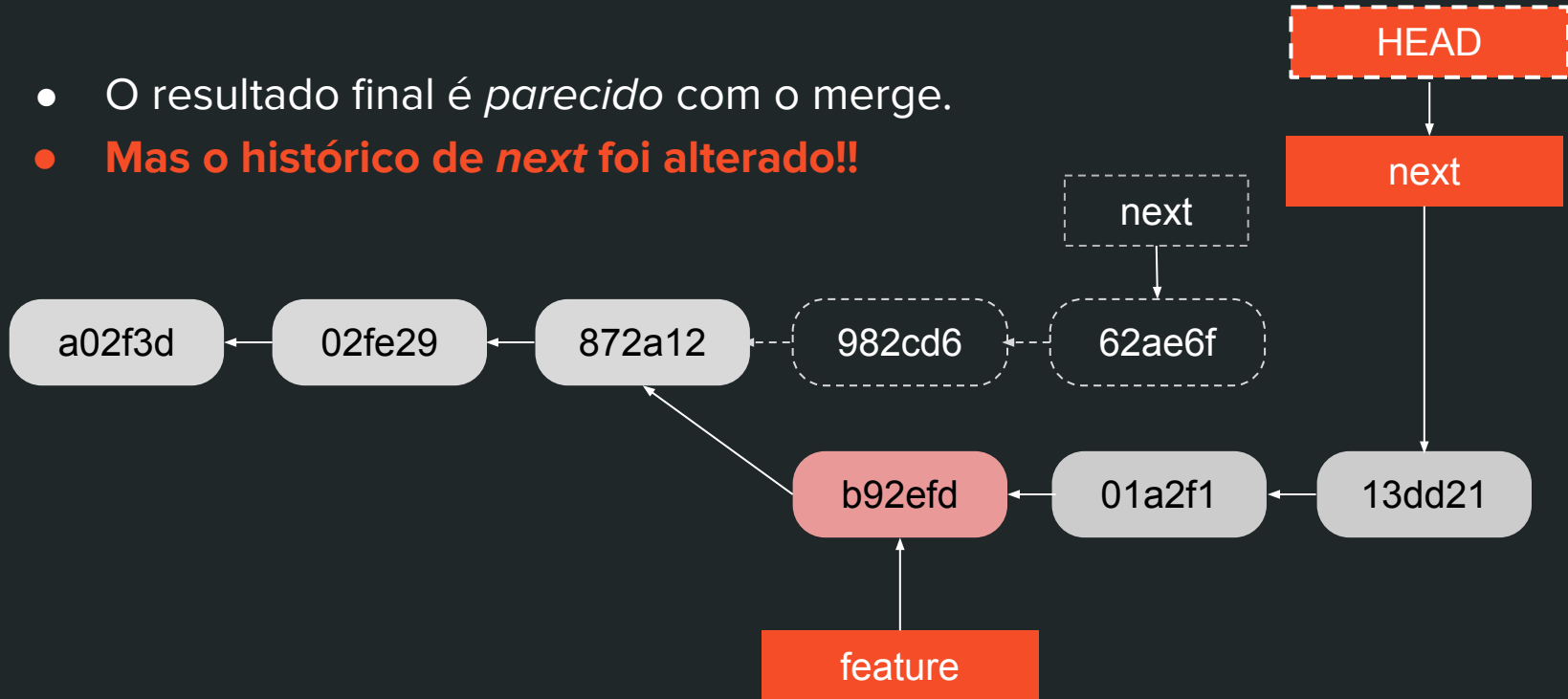
# Rebase

- O resultado final é *parecido* com o merge.



# Rebase

- O resultado final é *parecido* com o merge.
- **Mas o histórico de *next* foi alterado!!**



# Regra de ouro do Rebase

- Alterar o histórico de **uma branch local**, antes de um *push*, para organizar seus commits, **é uma ótima prática!**
- Alterar o histórico de **uma branch pública**, já utilizada por outros como base para seus trabalhos, **é uma má prática!**



<http://jdduarte.com/git-training/#!/>

# \$ git rebase -i

Te permite as seguintes operações sobre os commits passados:

- Adicionar
- Remover
- Reordenar
- Editar
- Juntar dois ou mais commits em um
- Executar um comando para cada commit  
(muito útil para rodar os testes para cada commit em uma branch)

# Boas práticas de *Commits*

# Parece uma boa ideia...

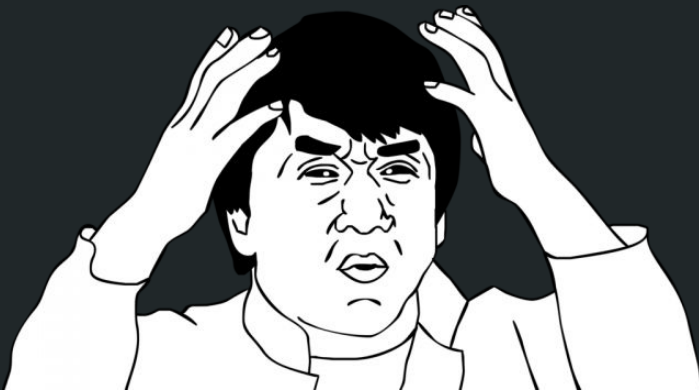
```
$ vim file.c
```

```
$ git add .
```

```
$ git commit -m "Update file.c"
```

```
$ git push origin master
```

... até que você precise  
relembrar o que fez.



<https://blog.tiagopariz.com/wp-content/uploads/2018/10/jackie-chan-meme.png>

```
commit 5e2faaea31dceaad23a3331cd9ad20afad6c719b
Author: joh123 <>
```

Minor changes

```
commit ae95825114ccac3a996251bd1de4da5e83c95172
Author: joh123 <>
```

Fixed some bugs

```
commit 6e639b014f82521138025091fac660cd2474b7a0
Author: joh123 <>
```

Now it compiles!

```
commit f717b52ddd1e95dbbdd5a1476051b842468cb195
Author: jonh123 <>
```

Actually it didn't. NOW IT DOES

```
commit 90384aec06d44a5d40d9c299f082334503943b32
Author: john123 <>
```

Update README



# Dissecando um bom commit

1. Mudanças **não correlacionadas** pertencem a **commits separados**.
2. **Não *commitar*** blocos de trabalho **incompletos**.
3. Invista na escrita de mensagens de commit **informativas**.

# 1. Mudanças **não correlacionadas** pertencem a **commits separados**.

- Mais fácil de **revisar** (→ melhores revisões → melhor código)
- Mais fácil de **reverter**
- Mais fácil de **integrar** com outros commits e branches. (i.e. é mais fácil de resolver conflitos de merge)

1. Mudanças **não correlacionadas** pertencem a **commits separados**.

```
commit 157c64679f49c4be16c08ba683d0e79652c6cb70  
Author: A U Thor <author@example.com>
```



```
Use 0 as default exit code and rename test files
```

## 2. **Não *commitar*** blocos de trabalho **incompletos**.

- Commits devem ser justificáveis por si só. (Embora um commit possa depender de outro)
- É legal garantir que **cada commit** seja compilável e passe os testes.  
(→ melhor usabilidade do *git-bisect*)

## 2. **Não *commitar*** blocos de trabalho **incompletos.**

```
commit c6444bb0d21d625311d8f06935acd6ea2cf3a8bd
Author: A U Thor <author@example.com>
```

Iniciando implementação do método de euler

```
diff --git a/a b/a
new file mode 100644
index 0000000..d8ab891
--- /dev/null
+++ b/a
@@ -0,0 +1,6 @@
+
+def euler(x0, v0, a, dt, N):
+    x, v = x0, v0
+    for i in range(N):
+        x = x + v*dt
+        v =
```

```
commit 1a3f223c6341684ee78d04760a9188563397b028
Author: A U Thor <author@example.com>
```

Finalizando implementação do método de euler

```
diff --git a/a b/a
index d8ab891..96be0ca 100644
--- a/a
+++ b/a
@@ -3,4 +3,5 @@ def euler(x0, v0, a, dt, N):
     x, v = x0, v0
     for i in range(N):
         x = x + v*dt
-        v =
+        v = v + a*dt
+    return x
```

Esses commits podem ser unidos em um.

### 3. Invista na escrita de mensagens de commit **informativas**.

“a well-crafted Git commit message is the best way to communicate context about a change to fellow developers (and indeed to [your future self]). A diff will tell you **what changed**, but only the commit message can **properly tell you why**.”

- Chris Beams (<https://chris.beams.io/posts/git-commit/>)

### 3. Invista na escrita de mensagens de commit **informativas**.

- **Descreva o problema:** o que não está legal no código atual?
- **Justifique como as mudanças resolvem o problema:** porque o estado do projeto após este commit é melhor do que o atual?
- **Alternativas descartadas [opcional]:** existem outros modos de implementar a mudança? Se sim, porque este foi escolhido?

#### **Dica:**

*git commit -v ou  
git config --global commit.verbose true*

# O formato da mensagem também é importante

Título resumindo as mudanças em até 50 chars

Corpo, separado do título por uma linha em branco, e justificado em 72 colunas. Explica a mudança em mais detalhes, como descrito no slide anterior. (Para mudanças *triviais*, pode ser omitido.)

O corpo pode ter múltiplos parágrafos e também:

- Bullet points
- Tabelas ou outros

Normalmente:

- No imperativo
- Não pontuado
- Primeira letra em maiúsculo

*Trailers*



# O formato da mensagem também é importante

## Trailers:

- Closes #21
- Fix #53
- Signed-off-by: A U Thor <author@example.com>
- Co-authored-by: A U Thor <author@example.com>
- Reviewed-by: A U Thor <author@example.com>
- Reported-by: A U Thor <author@example.com>
- Acked-by: A U Thor <author@example.com>

# Dissecando um bom commit

## Exemplo 1)

```
commit 7655b4119d49844e6ebc62da571e5f18528dbde8
```

```
Author: René Scharfe <l.s.r@web.de>
```

```
Date: Tue Mar 3 21:55:34 2020 +0100
```

```
remote-curl: show progress for fetches over dumb HTTP
```

Fetching over dumb HTTP transport doesn't show any progress, even with the option `--progress`. If the connection is slow or there is a lot of data to get then this can take a long time while the user is left to wonder if git got stuck.

We don't know the number of objects to fetch at the outset, but we can count the ones we got. Show an open-ended progress indicator based on that number if the user asked for it.

```
Signed-off-by: René Scharfe <l.s.r@web.de>
```

```
Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

# Porque isso é importante mesmo?

Vamos para um Exemplo!

(entendendo uma linha de código com `git-blame` + `git-show`)

# Trabalhando com *Remotes*

# Remote

- Um repositório *remoto* relativo ao mesmo projeto. (e.g. “origin”)
  - \$ git push origin master → “envie p/ a branch ‘master’ do repositório remoto ‘origin’”
- Você pode ter **vários** remotes (e.g. o repositório *upstream*, o seu *fork*, o *fork* de um colega, ...)
  - **\$ git remote add** john https://github.com/john/repo.git

# Remote Branches

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
    (use "git push" to publish your local commits)

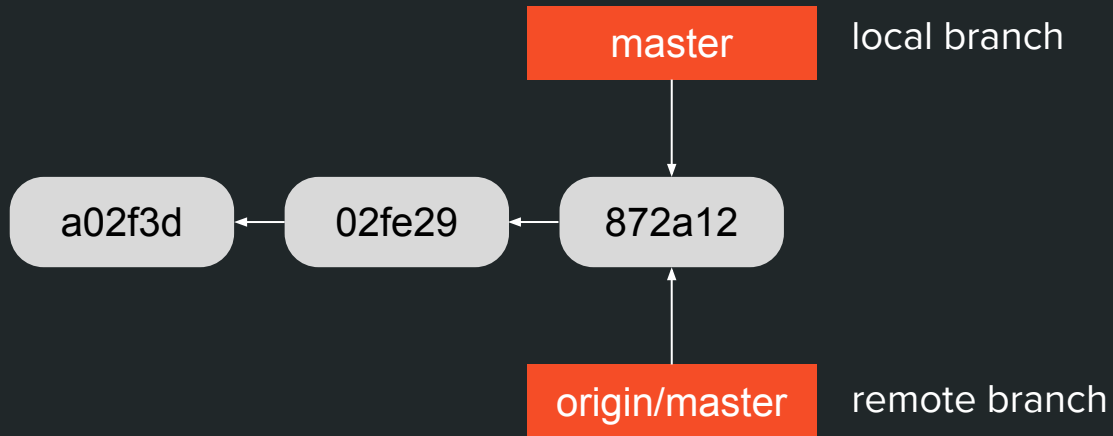
nothing to commit, working tree clean
```

- `.git/refs/heads/*` (local branches)
- `.git/refs/remotes/*` (remote branches)
  - e.g. `.git/refs/remotes/origin/master`

Nota: pode estar também em `.git/packed-refs` (formato mais eficiente)

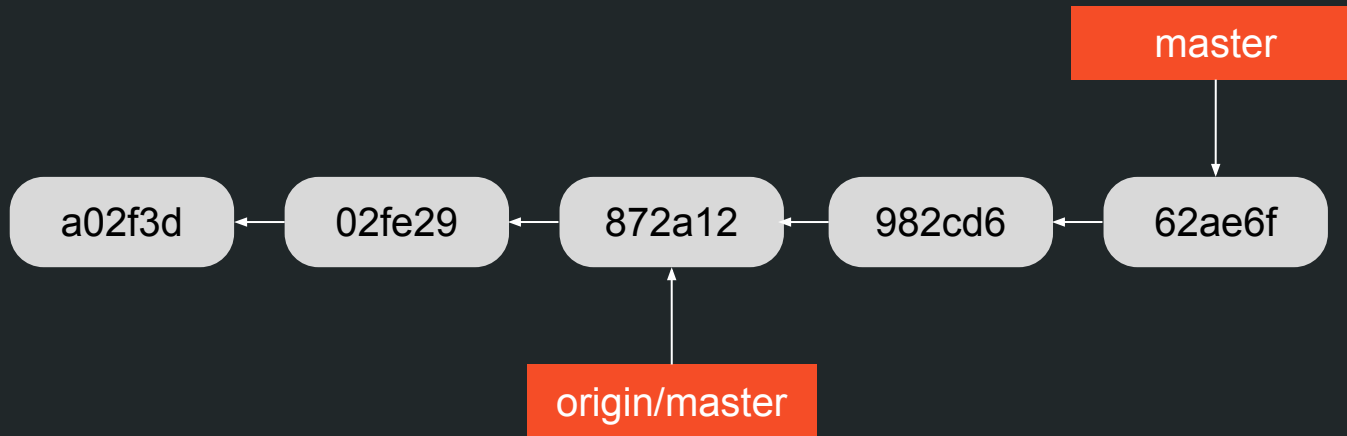
# Remote Branches

\$ git clone



# Remote Branches

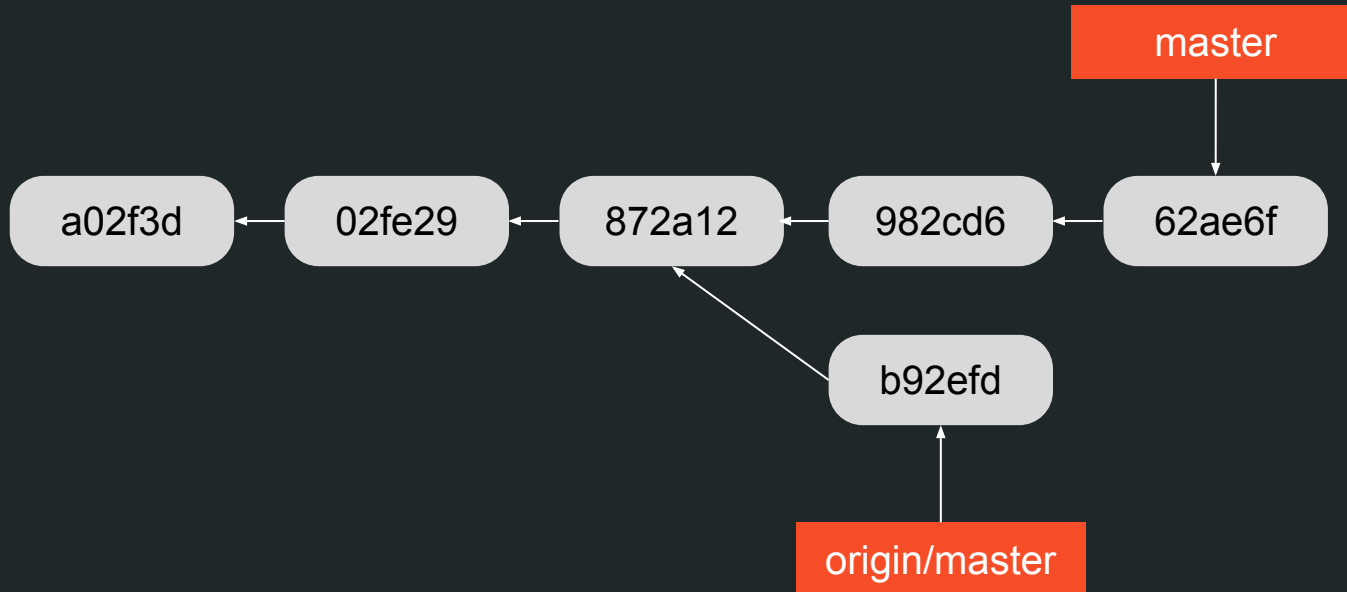
\$ git commit (2x)





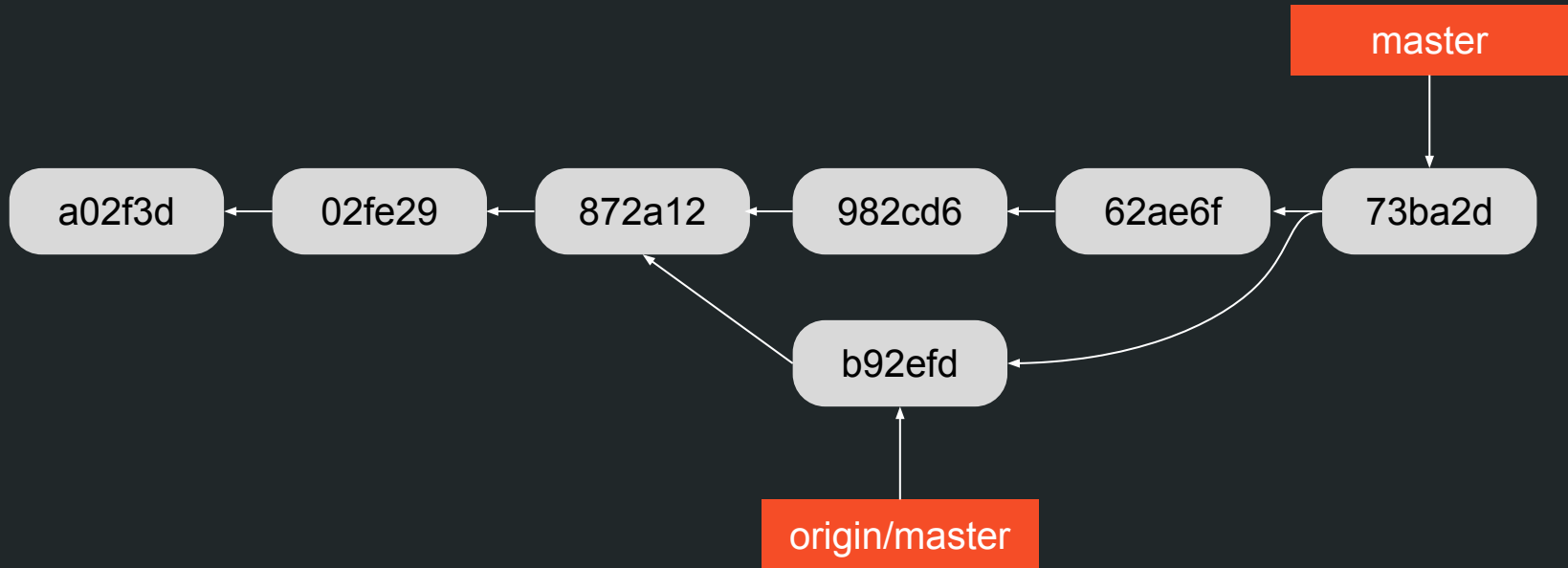
# Remote Branches

\$ git fetch origin



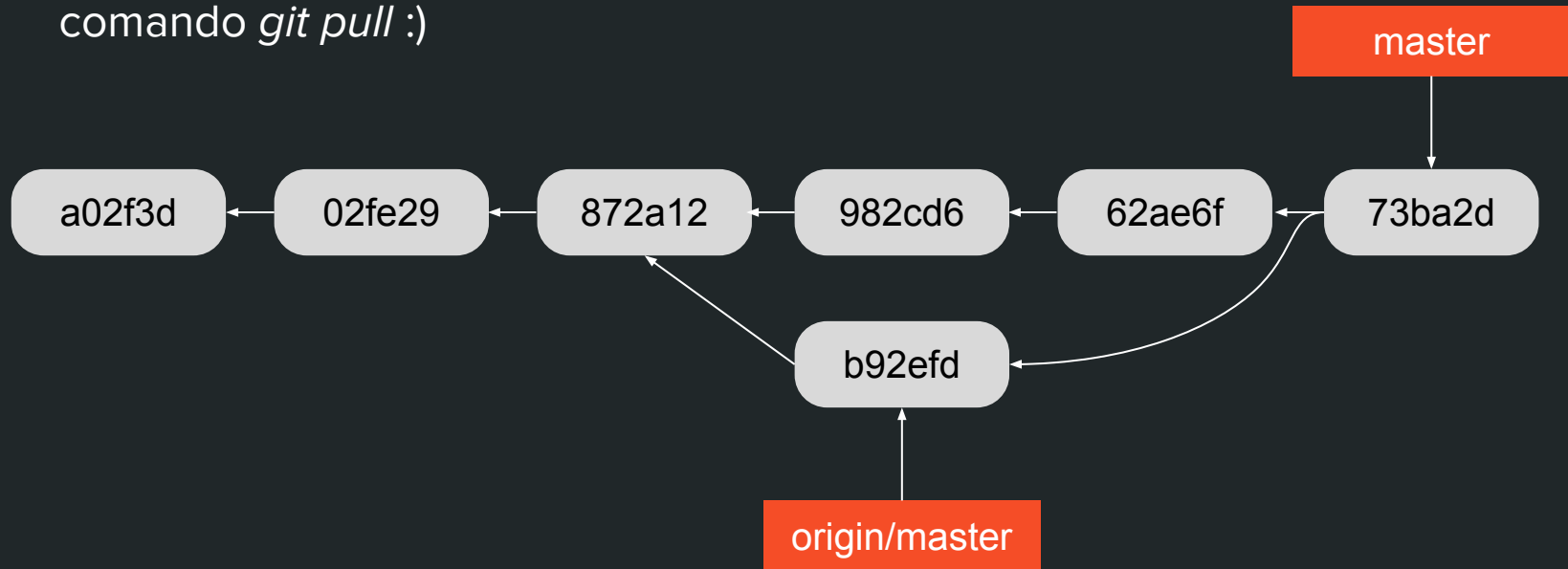
# Remote Branches

\$ git merge origin/master



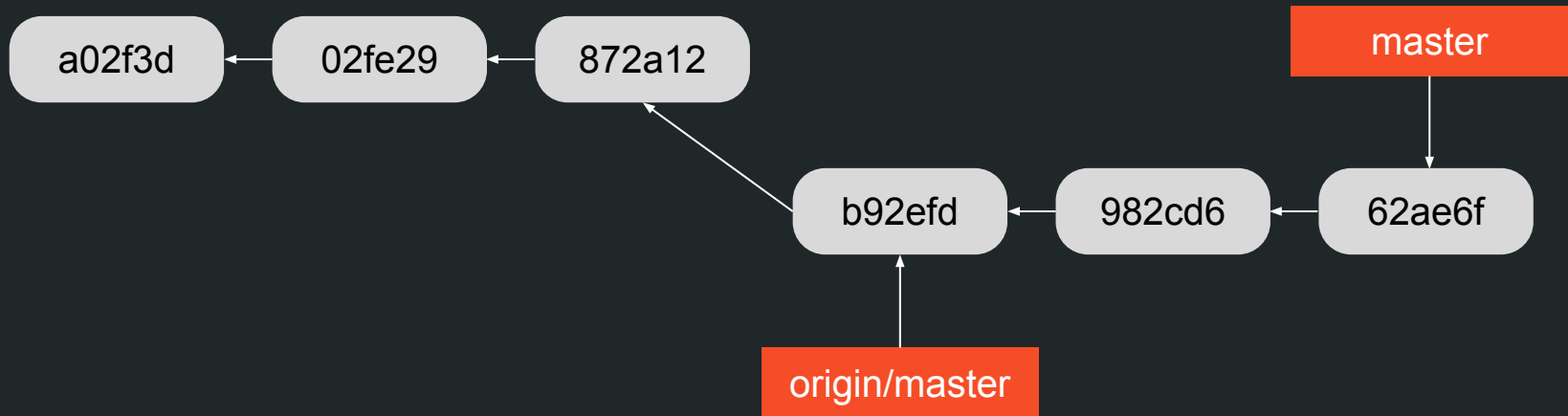
# Remote Branches

- O que acabamos de fazer é exatamente o que faz o comando *git pull* :)



# Remote Branches

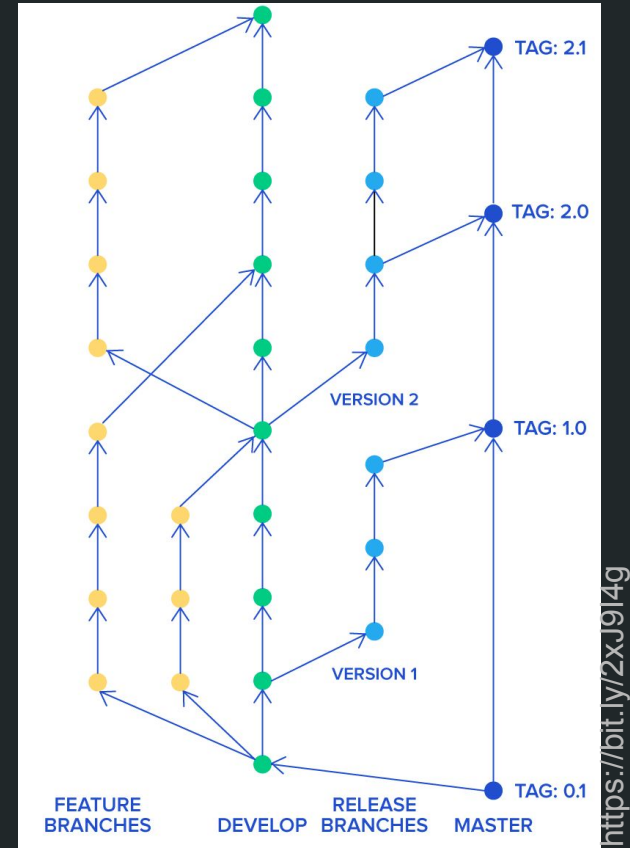
- Alternativa: `$ git pull --rebase`



# ***Workflows & Modelos de Branching***

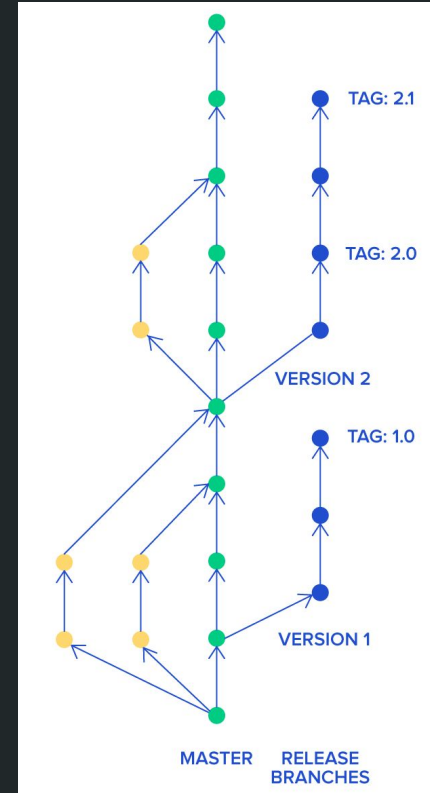
# Git Flow

- Mecanismo de “graduação” de branches. Possivelmente, com branches de acesso restrito.
- Topic branches de longa-duração.
- Custo de integração é consideravelmente alto.
- A master só contém código já validado e lançado.



# Trunk Based Development (TBD)

- Desenvolvedores trabalham mais próximo à branch principal.
- Topic branches de curta-duração.
- Integração rápida e contínua → garante integrações mais fluidas e menos *merge conflicts*.
- A master frequentemente contém features incompletas, geralmente escondidas com *feature flags*.



<https://bit.ly/2xJ9l4g>

# TBD: algumas ressalvas

- Alguns defendem *pushs* direto na *origin/master*. Se for usar este modelo:
  - **Pair programming** é fundamental.
  - Também um **bom CI / bateria de testes**.
  - É interessante usar ***git pull --rebase***
- Alternativamente, use topic branches de curta duração, com Pull/Merge Request e revisões
  - Use ***git rebase -i*** para corrigir problemas em sua branch.
  - Use ***git push -f*** para a respectiva branch remota, para atualizar o P/MR.



# CI / CD no GitLab

(similar p/ GitHub e outros)

# Continuous Integration, Delivery e Deployment

- **Continuous Integration:** integrar as mudanças dos desenvolvedores à base de código de forma contínua e rápida (até mesmo múltiplas vezes ao dia).
  - Objetiva evitar problemas de integração tardia, quando os códigos já divergiram muito.
  - Viabilizada pelo uso de testes (unitários e de integração) automatizados. Geralmente com uma infraestrutura própria para testar cada commit e informar se ele é *integrável*.

# Continuous Integration, Delivery e Deployment

- **Continuous Delivery:** “manter a base de código *entregável (deliverable)* a qualquer momento”.
  - Nesta fase, o servidor roda processos automatizados de *build* e *packaging*, permitindo que a aplicação esteja pronta para ser lançada, quando desejado.
- **Continuous Deployment:** Um passo além: não só constrói o *deliverable* como também faz o deploy da aplicação de forma automática e contínua.

# Continuous Integration, Delivery e Deployment

- Automatizar fases do processo de desenvolvimento.
  - Mitiga possibilidades de erros e bugs
  - Acelera o desenvolvimento
  - Integração muito boa com Métodos Ágeis

# CI / CD no GitLab

- <https://docs.gitlab.com/ee/ci/introduction/index.html#introduction-to-cicd-methodologies>
- Arquivo YAML “.gitlab-ci.yml”:

```
before_script: <install dependencies>
```

```
job1:  
  script: <run tests>
```

```
job2:  
  script: <build application>
```



“pipeline”

## Exemplo 1) CI p/ teste de programas em lua

```
1 image: pipelinecomponents/luacheck
2
3 before_script:
4 - apk update && apk upgrade && apk add make
5
6 test:
7   stage: test
8   script: make check
9   only:
10  - merge_requests
11  - master
```

## Exemplo 2) CI / CD p/ GitLab Pages c/ Jekyll

```
1 image: ruby:2.6
2
3 variables:
4   JEKYL_ENV: production
5
6 cache:
7   paths:
8     - vendor/
9
10 before_script:
11 - gem install bundler -v '2.0.1'
12 - bundle install --jobs $(nproc) --path vendor
13
```

```
14 test:
15   stage: test
16   script:
17     - bundle exec jekyll build -d test
18   artifacts:
19     paths:
20       - test
21   except:
22     - master
23
24 pages:
25   stage: deploy
26   script:
27     - bundle exec jekyll build -d public
28   artifacts:
29     paths:
30       - public
31   only:
32     - master
```

# Referências

1. **Pro Git**, Scott Chacon and Ben Straub:  
<https://git-scm.com/book/en/v2>
2. **Git Docs**: <https://git-scm.com/docs/>
3. **How to Write a Git Commit Message**,  
Criss Beans:  
<https://chris.beams.io/posts/git-commit/>
4. **Developer Tip: Keep Your Commits  
“Atomic”**, Sean Patterson:  
<https://www.freshconsulting.com/atomic-commits/>
5. **The Zen of Git**, Tianyu Pu:  
<https://speakerdeck.com/tianyupu/the-zen-of-git>





# Referências II

6. **Git For Computer Scientists**, Tommi Virtanen:  
<https://eagain.net/articles/git-for-computer-scientists/>
7. <https://trunkbaseddevelopment.com/>
8. **Trunk-based Development vs. Git Flow**, Konrad Gadzinowski:  
<https://www.toptal.com/software/trunk-based-development-git-flow>
9. **What is Continuous Integration**, Max Rehkopf:  
<https://www.atlassian.com/continuous-delivery/continuous-integration>



# Obrigado!

<https://matheustavares.gitlab.io>

# Some Extra Git Tips

- `$ git help glossary` contém definições sobre nomenclaturas usadas no Git.
- `$ git help workflows` contém dicas de workflows no estilo Git Flow.
- Sempre rode *status* antes de um *reset*
- Use o *git-config* para definir configurações default (e.g. `commit.verbose`)
- Use o *git-reflog*. Muito útil para recuperar o que aparentemente foi perdido!
- Quando tiver problemas, leia com calma a mensagem de erro. Entendendo as nomenclaturas e estruturas do Git, as mensagens são intuitivas.