GIT & GITHUB

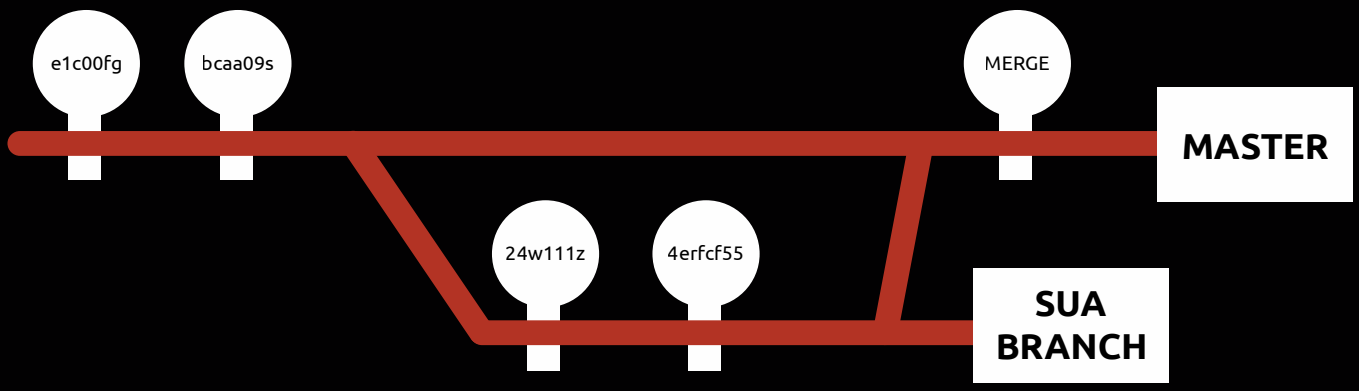
Do Básico ao Avançado

Domine Git e GitHub com este material completo. Com ele você aprenderá a gerenciar fluxos de trabalho de maneira simples e dinâmica.

Git Rebase

Em materias anteriores aprendemos a utilização do comando **merge**, que nada mais é do que a junção dos commits feitos em outras ramificações (branchs) dentro de outras (branch principal ou branchs secundárias).

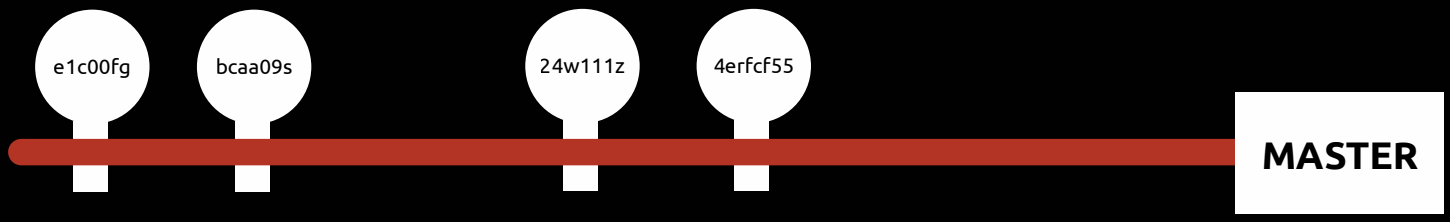
No caso do **merge**, ele cria um único commit que representa todas as modificações realizadas em uma única branch:



Ou seja, de acordo com a ilustração acima, supondo que foram feitos 2 commits na branch master, e em seguida criamos uma outra branch (secundária) aonde realizamos 3 commits.

Quando executarmos um merge (na branch master), o git criará um commit que representa todos os commits que foram realizados na branch secundária.

Isso é, caso a operação não seja do tipo **Fast-Foward**, pois como vimos, em operações como essas, o git simplesmente pega todos os commits da branch secundária e joga cada um deles dentro da branch principal.



Ainda com relação ao comando **merge**, nós vimos que ele não é um comando destruidor de **"universos alternativos"** (branchs).

O que significa dizer, que ele não destrói o histórico de commits do nosso projeto, fazendo com que a gente consiga visualizar todo o histórico e as possíveis ramificações por meio do comando git log e suas flags, onde conseguimos visualizar tudo o que ocorreu antes e depois do merge.

Com isso em mente já podemos conhecer o comando **Rebase**.

O que é o Git Rebase?

Ele funciona de forma similar ao **Merge**! Com o **Rebase** nos conseguimos modificar o histórico do nosso projeto de modo a:

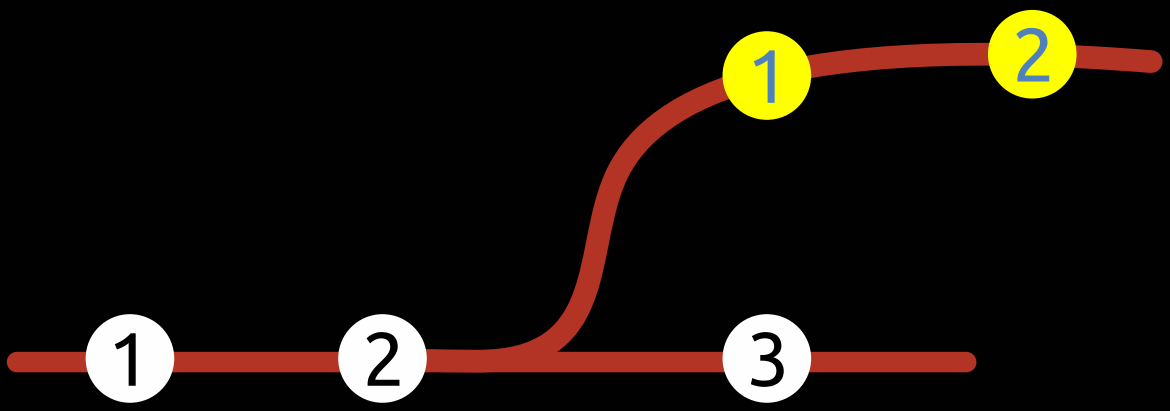
+ Alterar a ordem dos commits.
+ Unir commits (função similar ao "merge").
+ Modificar as mensagens que setamos em cada um de nossos commits.

Ou seja, tudo o que tem a ver com o histórico do nosso projeto, nós conseguimos alterar ou quem sabe fazer melhorias com o comando Rebase.

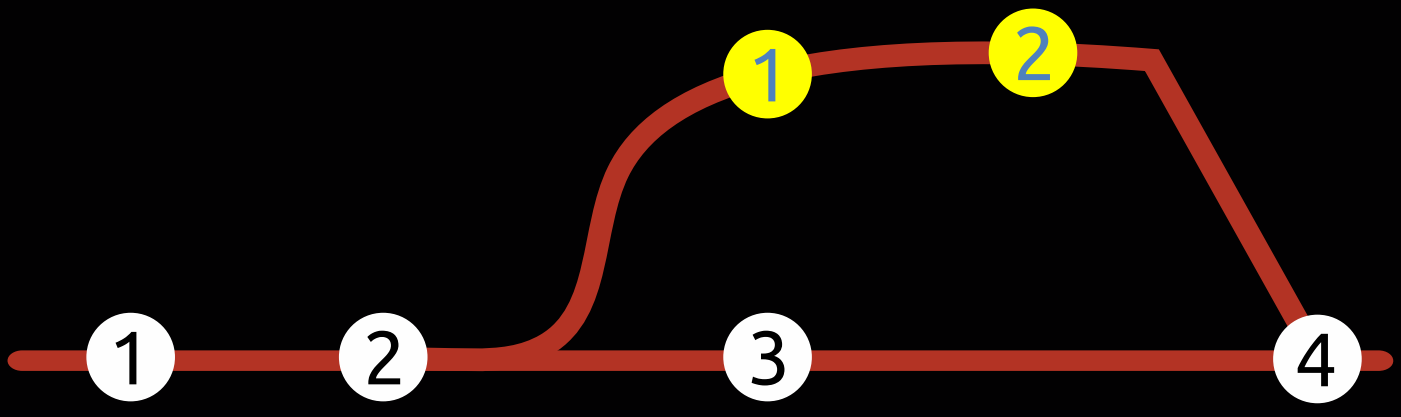
Rebase VS Merge

Com relação ao Merge, vamos imaginar que nos temos um projeto com 3 commits na branch master, e apenas 2 commits na branch secundária.

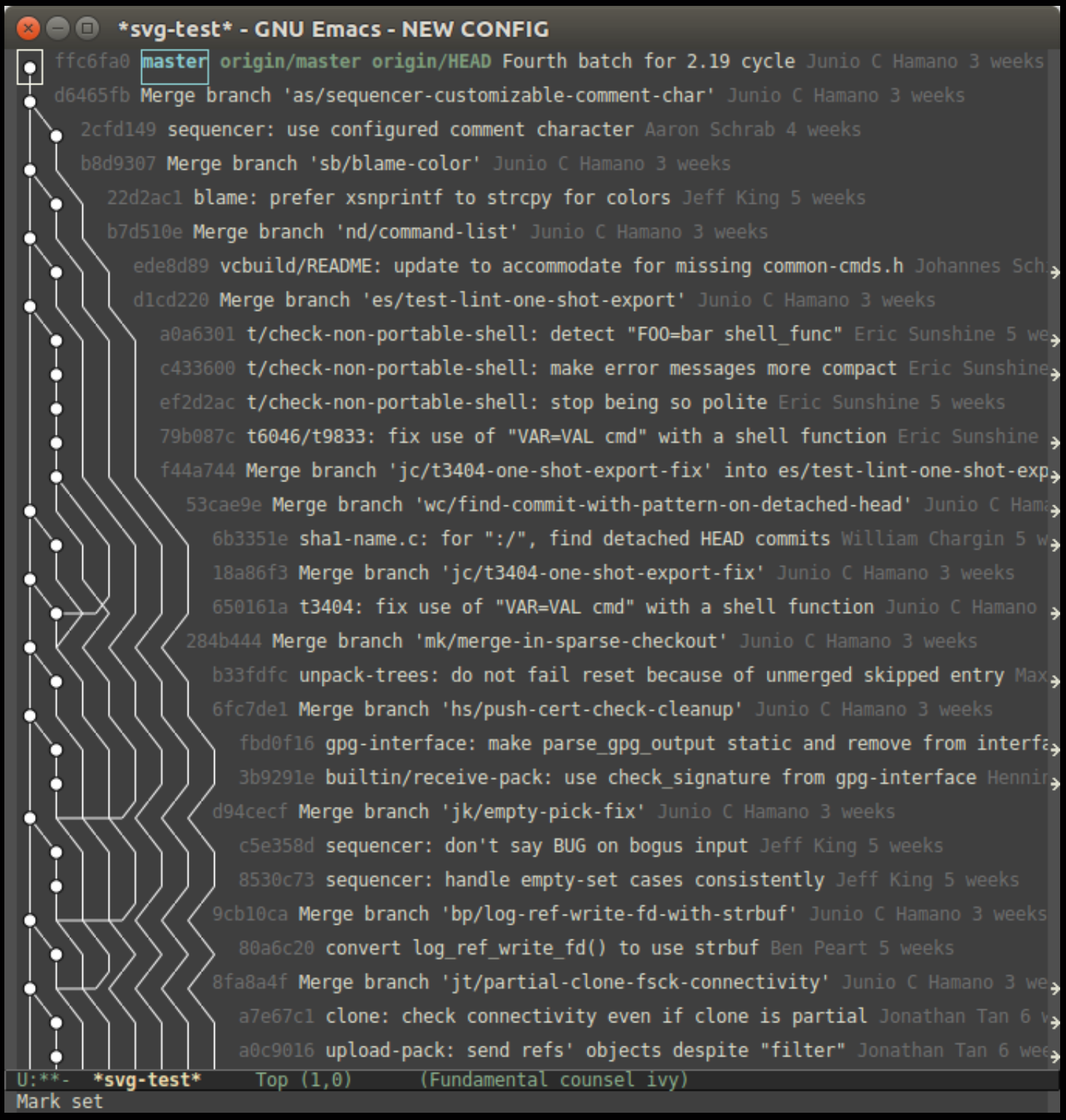
Sendo que a branch secundária, foi criada entre o 2º e o 3º commit da branch master, conforme representado abaixo:



A partir do momento que executarmos a operação **git merge** na branch master, os commits existentes na branch secundária serão mesclados como um 4° commit no master:



Por meio do comando Merge, quando nós temos muitas branchs que realizaram muitos merges, acaba ficando difícil a visualização do histórico do seu projeto, veja um exemplo:



Entre as **vantagens** do comando merge, podemos citar:

+ Os Logs são muito detalhados, o que pode nos ajudar a entender cada fusão que aconteceu durante o histórico do projeto.

+ É muito mais fácil encontrar erros e resolvê-los.

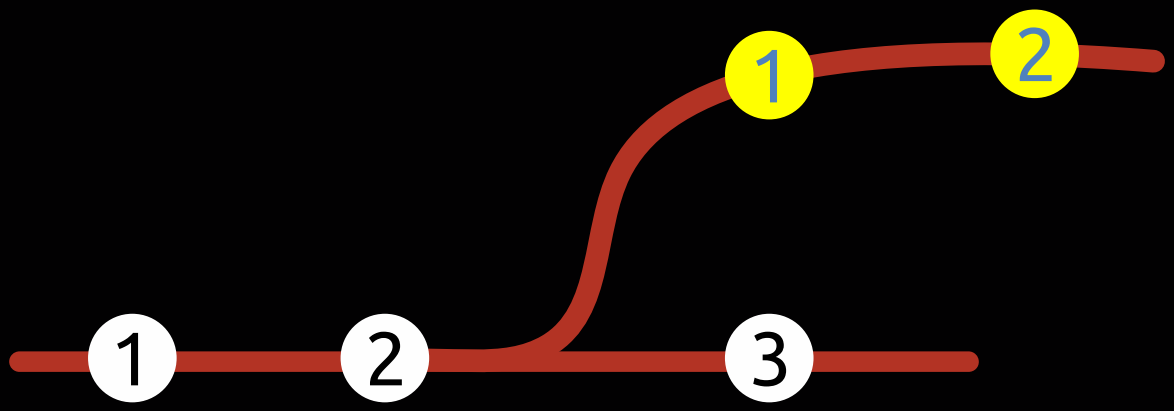
*E*ntre as **desvantagens** do comando merge, podemos citar:

+ Falta de organização e dificuldade para entender o que está acontecendo ali.

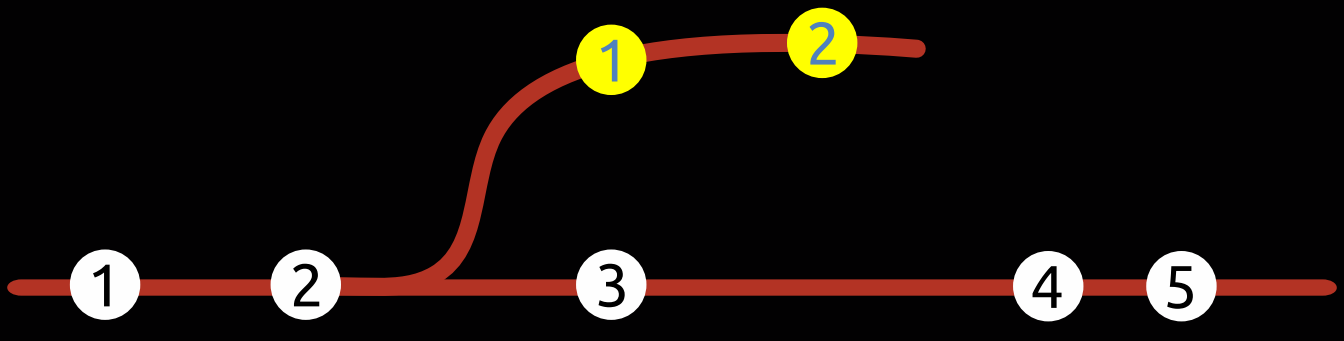
+ Não é muito amigável (pois como vimos, é uma grande confusão).

Se tratando do comando **Rebase**, mais especificamente na função de **"Unir Commits"**, ele permite que a gente limpe esse histórico, de modo a apagar todas as ramificações, fazendo com que nosso projeto fique mais organizado e fácil de entender.

Ainda naquele exemplo anterior:



Se tratando do Rebase, os commits existentes na branch secundária serão inseridos na branch master como 4° e 5° commit, de modo a não existir logs que apontem para a branch sencundária.



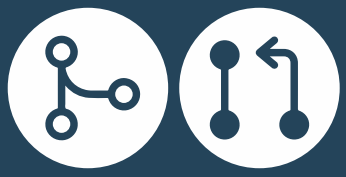
Entre as vantagens do comando **Rebase**, podemos citar:

+ Os logs são lineares.

+ É fácil movê-los no projeto.

Entre as **desvantagens** do comando Rebase, podemos citar apenas uma:

+ Não podemos rastrear quando e como os commits foram mesclados na ramificação de destino.



Rebase ou Merge?

Basicamente o **Merge** é aquele que mescla os commits em uma branch, já o **Rebase** ele organiza os commits para posteriormente - quando você for usar o Merge- tudo aconteça no estilo **"Fast Foward"**.

Com relação ao **Merge** podemos dizer que:

+ Permite mesclar ramificações (branchs) dentro de outras (branchs secundárias ou branch master).

+ Os logs do projeto (git log) mostrarão o histórico completo de todas as modificações feitas.

+ Os commits realizados na branch secundária, podem ser combinados na branch master como um único commit (Recursive Strategy), ou em vários (Fast Foward).

+ O seu uso é recomendado para branchs compartilhadas, que já estão no GitHub (muitos utilizadores).

Com relação ao **Rebase** podemos dizer que:

+ Permite fazer integrações personalizadas de uma branch para outra (posteriormente presamos usar o "merge")

+ Os logs do projeto (git log) não mostrarão o histórico completo, pois são sempre lineares.

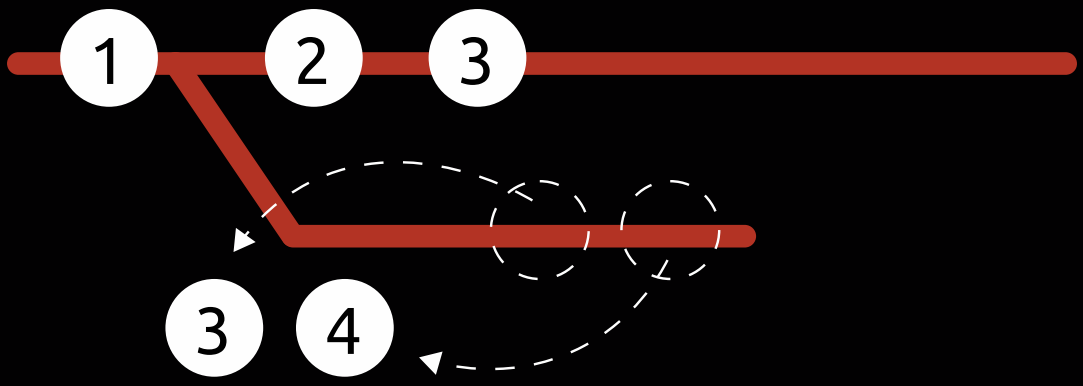
+ Os commits realizados na branch secundária, são combinados na branch master em forma de vários commits (no Rebase é sempre Fast Foward).

+ O seu uso é recomendado para banchs que ainda não foram compartilhadas (só você usa).

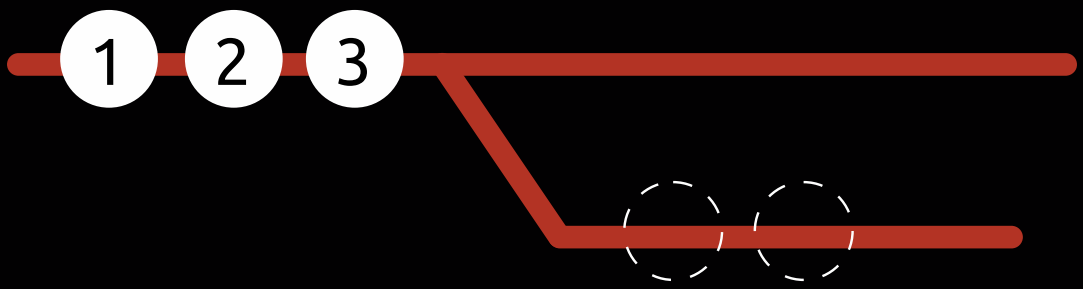
Entendendo a fundo o Rebase

Antes de continuarmos, é importante ressaltar que quando executamos um rebase na branch master para mesclar commits que foram feitos em uma branch secundária.

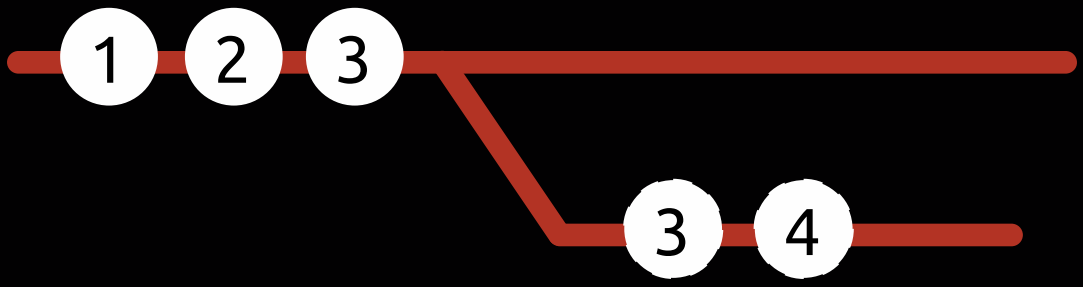
Tudo o que git faz é resguardar os commits realizados na branch secundária em um local temporário.



Em seguida o git move a branch pra frente, como se ela tivesse sido criada a partir do último commit feito na branch principal:



Depois o git re-insere os commits dessa branch:



E por fim, ele espera que você faça uma mesclagem (merge) na branch principal:

Por esse motivo que o comando **Rebase** ele sempre realiza um merge do tipo Fast Foward.

Porque como vimos, é como se tudo estivesse de maneira linear, por esse motivo que não temos um commit unificado quando realizamos um rebase.

É importante ressaltar também que quando o rebase move os commits para um local temporário, esses commits são refeitos de modo a receberem novos hashs.

E esse é um dos motivos principais para não realizarmos rebase em projetos compartilhados (aqueles que já estão no GitHub, por exemplo), uma vez que destruiremos o histórico do projeto.

Portanto, nunca faça **rebase** quando:

** Seu projeto está compartilhado.*
** Mais de uma pessoa está trabalhando na mesma branch.*

IMPORTATE: O comando Rebase apenas organiza e não faz o Merge, portanto, ainda precisamos do Merge.

Trabalhando com Rebase

```
git rebase minha-branch
```

Com este comando podemos fazer o rebase do histórico do nosso projeto.

O **Rebase** é usado em branchs secundárias, uma vez que ele vai recuperar todas as alterações da branch master (principal) e outras branchs existentes, e vai mesclar com a branch secundária de modo a empurrar os commits para frente.

Para que quando realizarmos um merge no master desse tipo:

```
git merge minha-branch
```

Tudo aconteça de forma mais **"Fast Foward"** possível.

Durante o Rebase, pode acontecer que existam alterações nos mesmos arquivos, ocasionando um conflito. Em casos como esses, nós precisamos resolver o conflito usando nosso editor de código favorito.

Em seguida adicionar a Stage com o comando **git add .**, e finalizar com o comando **git rebase minha-branch --continue**.

Caso você queria abortar um Rebase caso der um conflito, você pode usar o comando **git rebase minha-branch --abort**.

Caso você queria pular todos os conflitos e aceitar somente as alterações feitas na branch master, use o comando **git rebase minha-branch --skip**.

Rebase Interativo (-i)

Mudar nome dos commits? Juntar commits em um só? Sim, isso é possível graças ao **rebase interativo**.

Para começar a usar, primeiro você tem que se certificar que há pelo menos um único commit feito em todo o seu projeto.

Como é o caso do commit abaixo referente a **"Adição do Arquivo_1.txt"**, que tem o seguinte hash: **C5FZTR1.**

```
MINGW64:/c/Projeto

SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git log --oneline

C5FZTR1 Adição Arquivo_1.txt
```

Considerando que queremos mudar a descrição desse commit, podemos usar o seguinte comando:

```
git rebase -i --root
```

Após a execução desse comando, o git vai abrir o nosso editor de texto padrão – que nós definimos lá atrás durante a instalação, lembra? -, que vai conter o seguinte conteúdo:

```
pick c52941f Adição do Aquivo_1.txt
pick 8d60ff5 Adição do Arquivo_2.txt

# Rebase 8d60ff5 onto e875bcd (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref> is
#                       updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

Tudo o que está marcado com **#**, significa que é um comentário que está ali para te auxiliar, logo não devemos apagar.

Já os comandos que começa com o termo **"pick"** representa cada um dos commits, por exemplo:

pick: é um comando que você deverá substituir por reword, edit, squash, fixup, exec, break, drop, label, reset, merge, update… que são os comandos existentes no comentário.

A partir do momento que trocamos a palavra pick por reword, por exemplo, estamos dizendo ao Git que queremos trocar a descrição daquele commit.

Mas calma, porque ainda não é o momento de trocar a descrição por ali.

Uma vez que, assim que você trocar **"pick"** por **"reword"**, salvar o arquivo, e fechar, o git vai abrir novamente o editor de texto padrão para que você possa trocar exclusivamente a mensagem daquele commit:

```
Adição do Arquivo_2.txt

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Fri Mar 31 08:46:55 2023 -0300
#
# interactive rebase in progress; onto e875bcd
# Last commands done (2 commands done):
#    pick c52941f Adição do Aquivo_1.txt
#    reword 8d60ff5 Adição do Arquivo_2.txt
# No commands remaining.
# You are currently editing a commit while rebasing branch 'master' on 'e875bcd'.
#
# Changes to be committed:
#       new file:   Arquivo_2.txt
#
```

Ali, sim, você poderá alterar a mensagem, salvar o arquivo e depois fechar, para que assim que você realize o **git log –oneline**, você veja que a descrição daquele determinado commit que foi alterado.

É importante ressaltar que no exemplo acima, você viu terem dois commits, nesse caso se você optar por utilizar o termo **“reword”** nos dois, o Git irá abrir e fechar duas vezes o seu editor de texto padrão até que todas as alterações sejam realizadas.

Você também pode dizer ao **Rebase** que gostaria de receber somente a lista de commits, de um determinado commit para cima, por exemplo:

```
git rebase -i hash
```

No, más, é só entender os outros comandos existentes naquele arquivo de texto e seguir os procedimentos do Git.

Rebase (sem conflito)

Para colocar nossos conhecimentos em prática, vamos começar criando uma pasta chamada **“meu-projeto”**, e dentro dela, vamos adicionar o nosso primeiro arquivo chamado de **arquivo1.txt**, que contém o seguinte conteúdo:

```
1  #Arquivo_1
```

Após isso, vamos fazer os procedimentos do commit:

```
git init
git add .
git commit -m “Arquivo 1”
```

Em seguida vamos criar uma nova branch:

```
git checkout -b minha-branch
```

Dentro dela, vamos adicionar um novo arquivo chamado de **“arquivo_2.txt”**, aqui não precisamos criar um conteúdo para ele, basta apenas que você crie um arquivo de texto em branco e o salve.

Após isso, vamos fazer os procedimentos do commit:

```
git add .
git commit -m “Arquivo 2”
```

Voltando agora para nossa branch master:

```
git checkout master
```

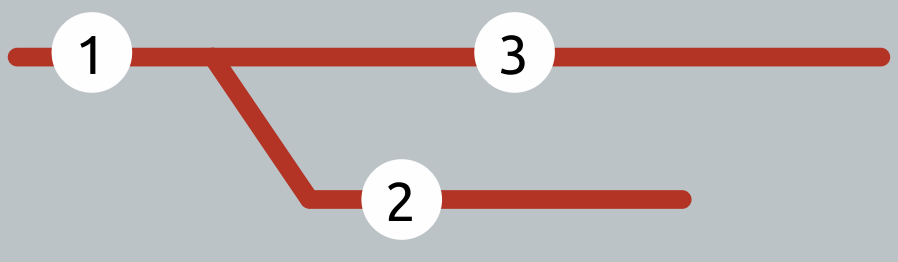
Vamos fazer mais uma modificação no Arquivo_1.txt:

```
1  #Arquivo_1
2  #Modificação: 1
```

E em seguida fazer os mesmos procedimentos do commit:

```
git add .
git commit -m “Modificação 1”
```

Colocando tudo isso numa representação gráfica, nós temos:



Para testarmos o **Rebase**, precisamos voltar para a nossa branch, e executar o rebase:

```
git checkout minha-branch
git rebase master
```

Automaticamente o **arquivo_1.txt**, sofrerá as alterações que fizemos no master.

Ou seja, aqui nos estamos atualizando o histórico da nossa branch baseada na branch master.

Realizando o **git log --oneline**, veremos os 3 commits que foram trabalhados.

Por fim, basta voltarmos para nossa branch master e realizar um merge, que tudo ficará atualizado pelo modo **"Fast Foward"**:

```
git checkout master
git merge minha-branch
```

Executando o **git log --oneline**, veremos que não existirá um histórico da branch **"minha-branch"**, é como se o merge nunca tivesse existido, e só tivéssemos trabalhado diretamente no master.

Rebase (conflito)

Dessa vez nós vamos ver como o Rebase se comporta quando temos um **conflito** na aplicação.

Para isso criei uma nova pasta chamada **"meu-projeto"**, com um arquivo chamado **"arquivo_1.txt"**, com o seguinte conteúdo:

```
1  #Arquivo_1
```

Após isso, vamos fazer os procedimentos do commit:

```
git init
git add .
git commit -m “Arquivo 1”
```

Em seguida vamos criar uma nova branch:

```
git checkout -b minha-branch
```

Fiz uma pequena edição no arquivo_1.txt, para:

```
1  #Arquivo_1
2  #Modificação: 1
```

Após isso, voltei para a branch master

```
git checkout master
```

E fiz uma alteração no arquivo_1.txt:

```
1  #Arquivo_1
2  #Alteração: 1
```

Por fim, realizei o commit:

```
git add .
git commit -m “Alteração 1”
```

Agora chegou o tão esperado momento de fazermos com que o **git dê conflito usando o rebase**, para isso vamos voltar para a nossa branch e tentar executar o rebase por lá:

```
git checkout minha-branch
git rebase master
```

E da mesma forma que aconteceu no nosso merge, a gente precisa resolver o conflito, e salvar o arquivo (Use o seu editor de código/texto para resolver isso no **“arquivo_1.txt”**).

Note que apareceu uma mensagem ao lado da sua branch indicando **“Rebase 1/1”**, isso significa que o **Rebase** foi pausado, e que nesse momento precisamos adicionar o arquivo novamente a Stage e realizar o continue:

```
git add .
git rebase --continue
```

E para finalizar, voltamos para a Master, e realizamos um **Merge (Fast Foward)**:

```
git checkout master
git merge minha-branch
```

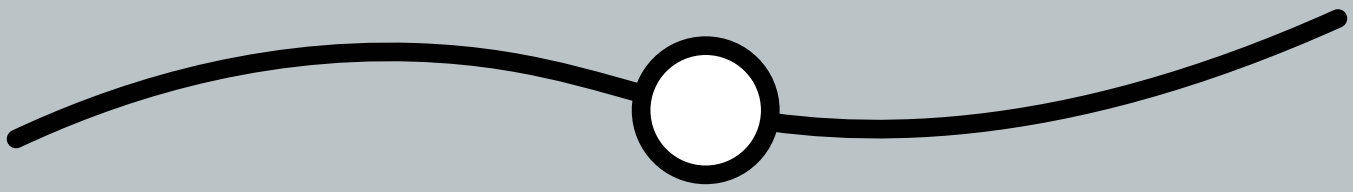
E pronto, mais um merge do tipo **“Fast Foward”** sem nenhum conflito aparente!

E agora, o que fazer?

Agora que você já sabe utilizar o rebase, e suas principais diferenças perante ao merge, que tal treinar mais um pouco antes de prosseguir para a próxima fase?

Neste material você aprendeu um pouco mais sobre:

Rebase *
Conceitos profundos sobre Rebase e Merge *
Diferenças entre Rebase e Merge *



END (y/n)

Gostou desse material?

Então não deixe de acessar a nossa seção de Git & GitHub do básico ao avançado.