



Controle de Versão

Aula 02

<Módulo 02/>

Branches e merge

Sumário



- **Branches e merge**

- Introdução
- O que é uma branch
- Como são usadas as branches
- Git branch
- HEAD
- Git checkout (branches no mesmo commit)
- Git commit
- Mais commits na mesma branch
- Git checkout (branches em commits diferentes)
- Detalhes do git checkout
- Construção da segunda branch
- Git merge (fast-forward)
- Git merge (three-way)
- Quando ocorre merge fast-forward ou three-way ?
- Detalhes do merge three-way
- Merge com conflitos
- Resolvendo conflitos de merge
- Deletando branches

- **Resolução de problemas comuns**

- Introdução
- Git checkout falhou porque Working Tree tem modificações: git stash
- Iniciei um merge, teve conflitos, não quero resolver agora: git merge --abort
- Fiz um merge, mas quero desfazer: git reset --hard
- Esqueci de trocar de branch antes de fazer commit: git reset --hard
- Fiz um commit, quero desfazer sem perder o conteúdo dele: git reset --mixed

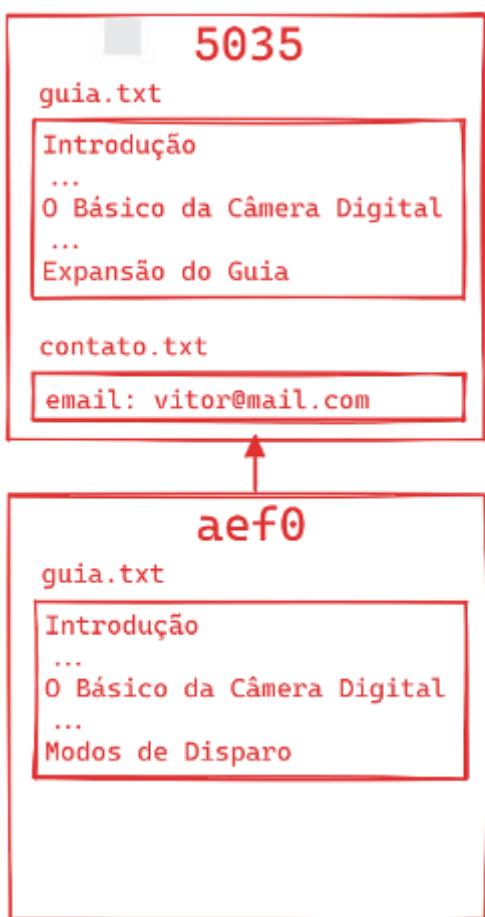
Branches e merge

Introdução



Imagine que você está trabalhando na escrita de um guia completo sobre fotografia digital, que você publica num blog na internet.

Atualmente, seu projeto contém os seguintes dois commits:



* Legenda:
Commits em vermelho (hash e arquivos)

E você tem um programinha que automaticamente publica no seu blog o commit mais recente, toda vez que você faz um commit novo. O commit mais recente é aquele marcado com a palavra master que você vê no GitLens (já já falaremos o que significa esse master aí):

Commit Graph: guia-fotografia X

guia-fotografia > master > Fetch

Search commits (1 for history)

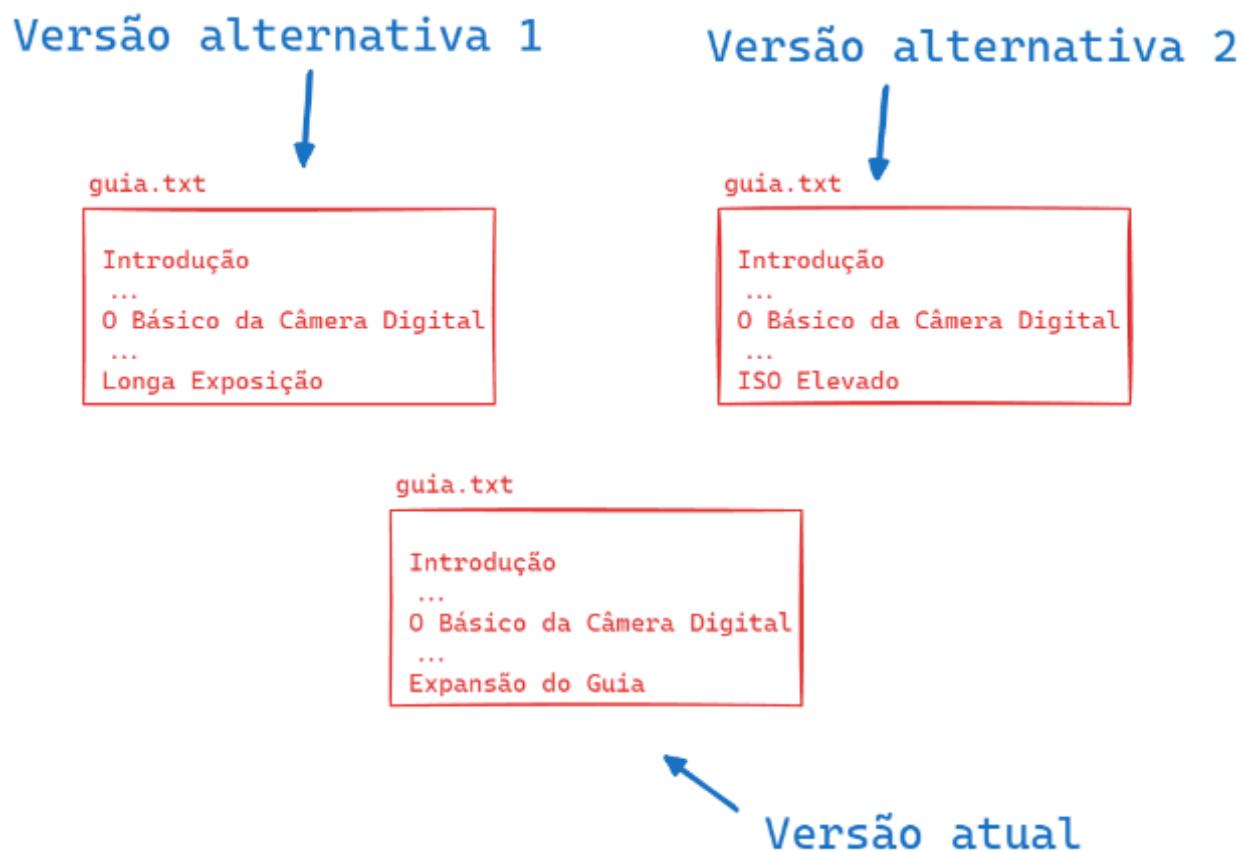
BRANCH / TAG	GRAPH	COMMIT MESSAGE	SHA
✓ master	Initial commit	Add guide expansion section and contact information	503517d
	Work in progress	Initial commit	aef0f34

Branches e merge

Obs: isso é realmente possível de fazer, e é chamado de “entrega contínua” no mercado de software, mas está além do nosso escopo atual. Só imagine que você tem esse programa.

Pensando no que escrever em seguida no guia, você quer testar qual técnica avançada de fotografia as pessoas gostariam mais de ler: longa exposição, ou ISO elevado ?

Você quer escrever duas versões do guia: uma trocando a seção “Expansão do Guia” por “Longa Exposição”, outra trocando “Expansão do Guia” por “ISO Elevado”:



Você quer manter as duas versões separadamente, em vez de escrever sobre os dois temas, senão o guia ficaria muito extenso, e isso não é bom para a popularidade dele!

Aí você pretende passar cada versão do guia para alguns amigos lerem, e assim testar qual é a versão preferida pelo público.

Depois que você souber qual é a melhor versão, pode publicá-la no seu blog.

Mas como fazer isso no Git ? Como manter ao mesmo tempo duas versões de rascunho separadas ?

Para esse fim, vamos introduzir a funcionalidade de “branch” (ramificação) do Git, e o fluxo de trabalho com ela.

Branches e merge

O que é uma branch

Uma das funcionalidades do Git que ainda não mencionamos é chamada "branch" (ramificação).

Uma branch é um marcador que fica em cima de um commit.

Por exemplo, um desses marcadores (branch) é a palavra `master` que você já tem visto aparecer. Neste texto, vamos representar uma branch como um retangulinho que fica em cima de um commit:

"`master`" é o nome de uma branch.
Branch é um marcador que
fica em cima de um commit



Branches e merge

Caso você esteja se perguntando “mas por que tem um marcador com nome `master` ?”, saiba que é porque, quando você criou o repositório com `git init`, o Git automaticamente criou o primeiro marcador com nome `master`.

Depois quando você fez o primeiro commit `aef0`, o marcador (branch) `master` ficou em cima dele marcando.

Ao fazer o segundo commit, a branch `master` avançou e ficou como está agora em cima do segundo commit `5035`.

E assim por diante: quando você fizer um terceiro commit, a branch “`master`” vai sair do segundo commit e avançar ao terceiro commit, vai ficar em cima dele.

Essa característica diferencia uma branch de uma tag, que já vimos em aulas passadas: uma tag fica parada num commit, já uma branch pode avançar de um commit ao próximo.

O desenvolvedor do projeto (você) pode criar outros marcadores (branches) além do `master` e dar nomes quaisquer a eles. Nós não fizemos isso ainda. Veremos por que você poderia querer criar mais marcadores.

Como são usadas as branches

Para o Git não há nada especial no marcador `master` em relação aos outros marcadores que você porventura venha a criar. Todos são igualmente marcadores.

Mas, para a prática de mercado no desenvolvimento de software, existe uma convenção:

- **Branch `master`:**

O commit marcado pela branch `master` é considerado a versão de produção, a ser distribuída para clientes.

Se a branch `master` mudar de commit, o novo commit marcado por ela é a nova versão de produção.

No seu caso, é o commit que seu programinha automaticamente publica no seu blog.

Isso significa que, na indústria de software, a `master` não deve ser colocada num commit de “rascunho”, que não tenha sido testado e verificado que aquela versão do software está funcionando perfeitamente.

E no seu caso, significa que você não quer colocar a `master` em nenhuma das duas versões “Longa Exposição” e “ISO Elevado” do guia, pois elas são versões de rascunho.

Se você colocasse a `master` para marcar uma dessas versões, seu programinha iria automaticamente publicar no blog para todos verem, não é o que você quer.

- **Outras branches:**

Já os outros marcadores são o oposto: eles são usados para marcar commits de “rascunho”.

Versões que ainda não estão finalizadas, e que ainda não estão prontas para serem distribuídas ao público.

Diferente da `master`, que é uma branch automaticamente criada pelo Git ao iniciar o projeto, outras branches são criadas pelo desenvolvedor, e podem ter qualquer nome.

Branches e merge

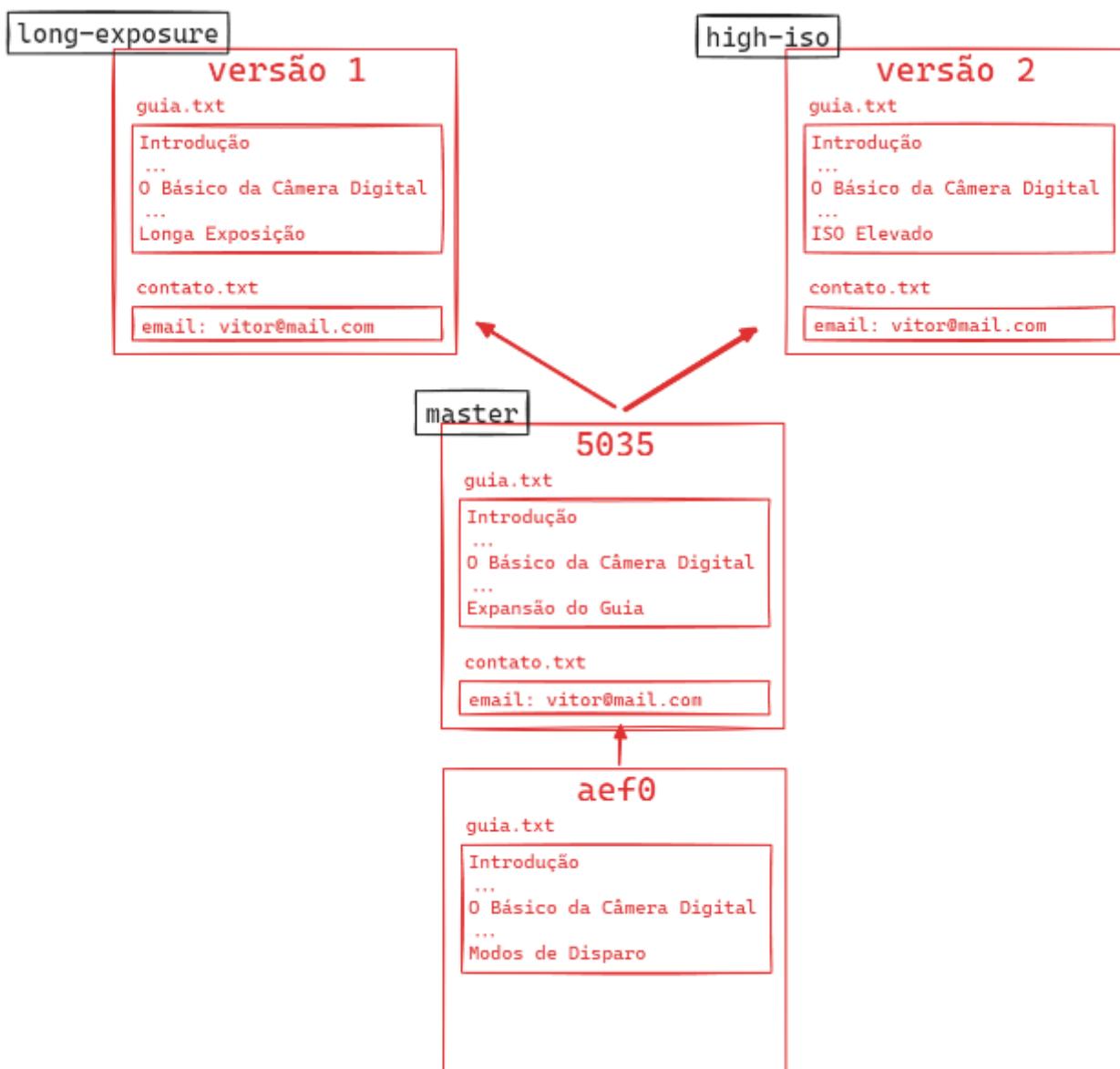
Git branch

O que vamos fazer agora é utilizar o recurso de marcador (branch) para criar as duas novas versões do guia.

Cada versão será armazenada na forma de um novo commit, e vamos marcar cada um desses novos commits com uma branch diferente:

- Branch long-exposure para a versão do guia sobre Longa Exposição
- Branch high-iso para a versão sobre ISO Elevado.
- E a branch master vai ficar onde está agora.

Esses nomes long-exposure e high-iso são arbitrários (o Git não se importa). O objetivo é ficar assim:



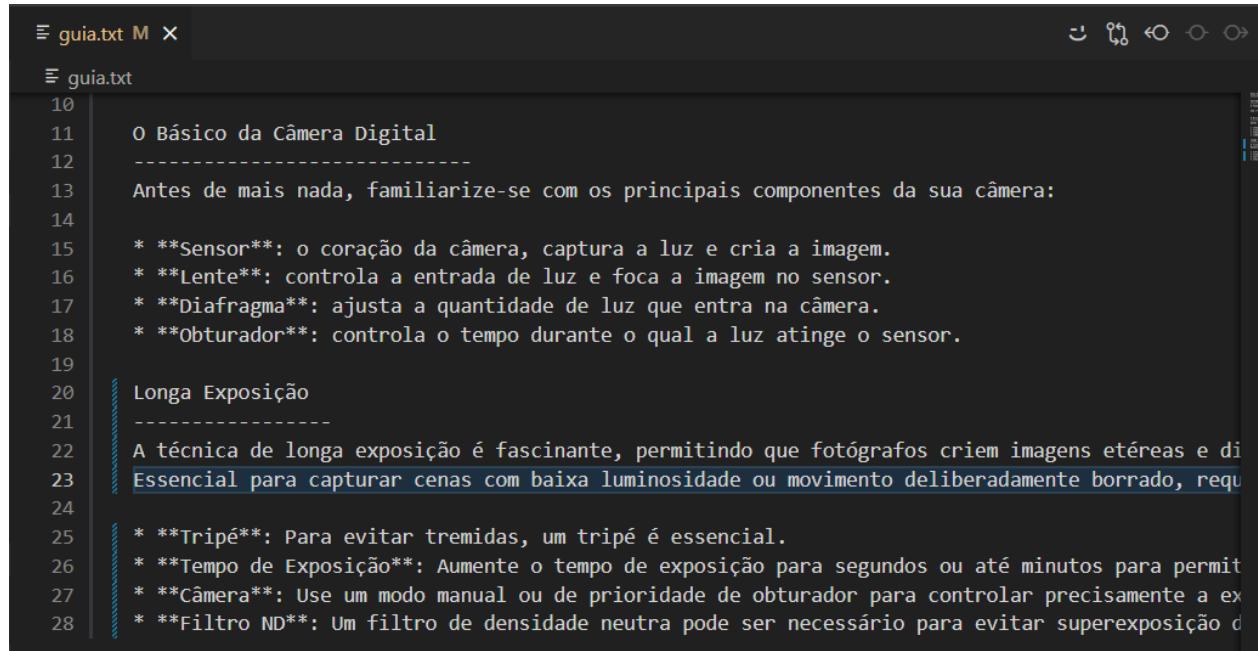
Branches e merge

Vamos começar a implementar as duas novas versões do guia, primeiro a "Longa Exposição".

A princípio, ainda não precisamos criar uma nova branch, podemos fazer como usual:

- Modificar o que queremos
- Executar `git add`

Então modifique o `guia.txt` substituindo a seção "Expansão do Guia" por "Longa Exposição":



```

guia.txt M X
guia.txt
10
11 O Básico da Câmera Digital
12 -----
13 Antes de mais nada, familiarize-se com os principais componentes da sua câmera:
14
15 * **Sensor**: o coração da câmera, captura a luz e cria a imagem.
16 * **Lente**: controla a entrada de luz e foca a imagem no sensor.
17 * **Diafragma**: ajusta a quantidade de luz que entra na câmera.
18 * **Obturador**: controla o tempo durante o qual a luz atinge o sensor.
19
20 Longa Exposição
21 -----
22 A técnica de longa exposição é fascinante, permitindo que fotógrafos criem imagens etéreas e di
23 Essencial para capturar cenas com baixa luminosidade ou movimento deliberadamente borrado, requ
24
25 * **Tripé**: Para evitar tremidas, um tripé é essencial.
26 * **Tempo de Exposição**: Aumente o tempo de exposição para segundos ou até minutos para permit
27 * **Câmera**: Use um modo manual ou de prioridade de obturador para controlar precisamente a ex
28 * **Filtro ND**: Um filtro de densidade neutra pode ser necessário para evitar superexposição d

```

Depois execute `git add` para adicionar a mudança ao Index.

Mas não podemos executar `git commit` ainda, porque sabemos que isso criará um novo commit e fará a master avançar ao novo commit, quando na verdade queremos que a master fique onde está, para que seu programinha de publicação automática *não* publique a nova versão (pois é um rascunho).

Nesse momento precisamos criar uma nova branch, que denominaremos `long-exposure`.

O nome pode ser qualquer um (o Git não se importa), mas é comum nomear a branch segundo a temática dela, que nesse caso é escrever sobre Longa Exposição.

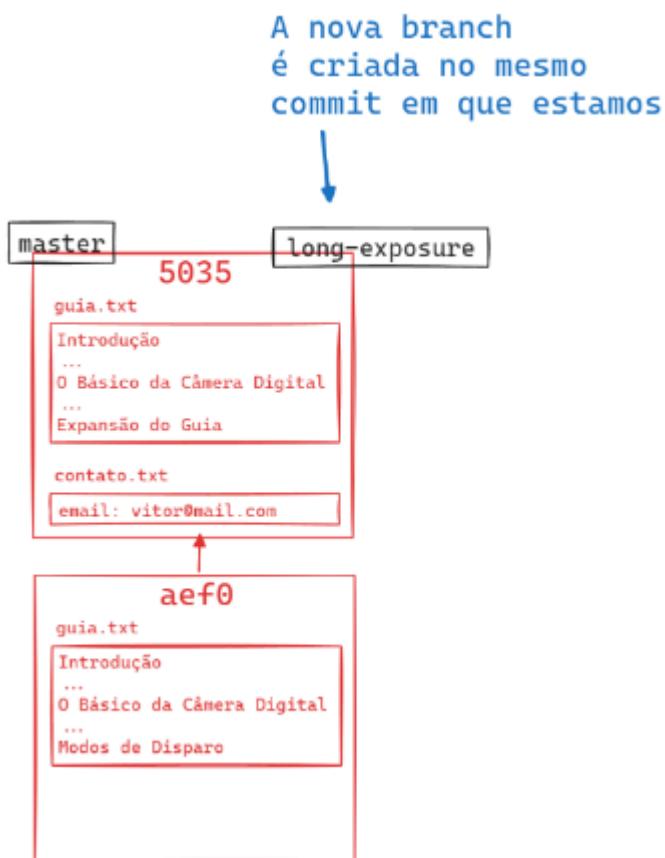
Nomes de branch não podem ter espaço.

Para criar a branch pelo terminal, o comando é `git branch <nome da branch>`, neste caso:

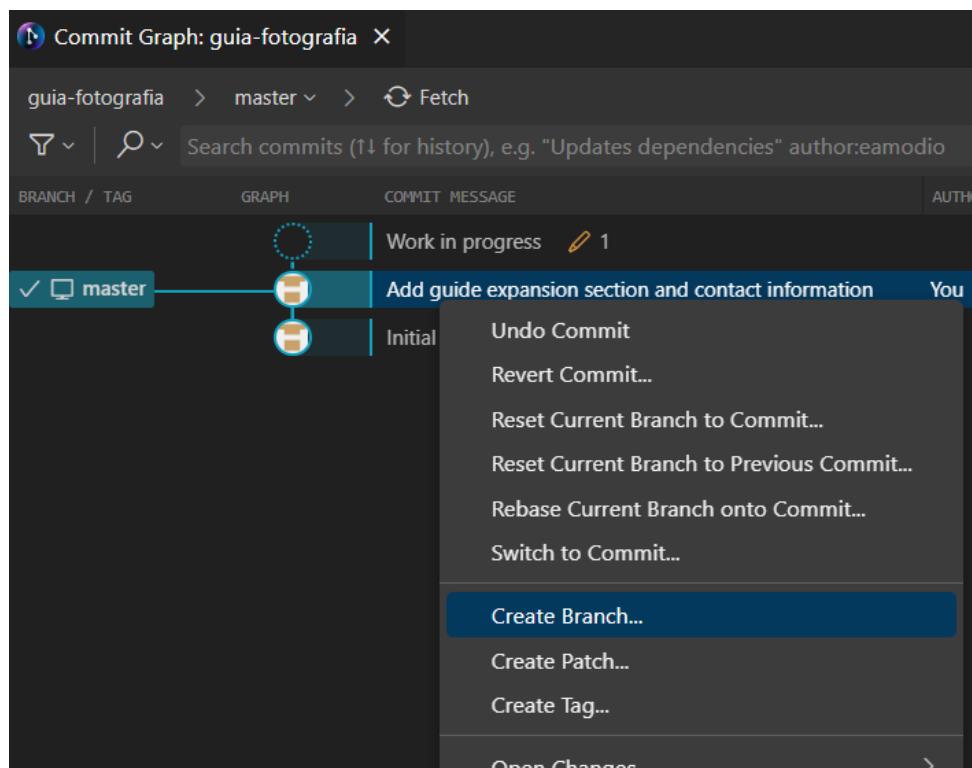
- `git branch long-exposure`

Isso vai criar o marcador sobre o commit em que estamos atualmente (commit 2), ou seja, vai ficar assim:

Branches e merge



Se preferir fazer pelo VSCode, abra a visão de Commit Graph do GitLens, clique com o botão direito do mouse no commit 2, e então em "Create Branch":



Branches e merge

Digite o nome da branch e aperte Enter.



Em algumas situações, pode ser que você queira criar uma branch em outro commit que não é o atual.

Nesse caso, no terminal, o comando é:

- `git branch <nome da branch> <hash do commit>`

E pelo GitLens, o procedimento é como já ilustrado, mas clicando no commit onde você quer colocar a branch.

Para validar que a branch foi criada, pelo terminal veja o git log:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git log --oneline  
503517d (HEAD -> master, long-exposure) Add guide expansion section  
aef0f34 Initial commit
```

Ele mostra a long-exposure no último commit, como queríamos.

Alternativamente, para ver pelo VSCode, abra o Commit Graph do GitLens e coloque o mouse na palavra master. Vão aparecer as duas branches:

The screenshot shows the 'Commit Graph' view in VSCode for the repository 'guia-fotografia'. The top navigation bar shows 'master' as the active branch. Below it, there's a search bar and filter options. The main area displays a graph of commits. Two commits are visible: 'Initial commit' (blue circle) and 'Add guide expansion section and contact information' (orange circle). A tooltip appears over the 'master' branch label, indicating its status: 'Work in progress 1'. The commit graph shows a connection from the 'Initial commit' to the 'Add guide expansion section and contact information' commit, with the 'master' branch pointing to the latter.

Criar uma branch não faz nada além de criar um marcador sobre o commit.

Fica armazenada na pasta .git a informação sobre quais branches existem e em quais commits elas estão.

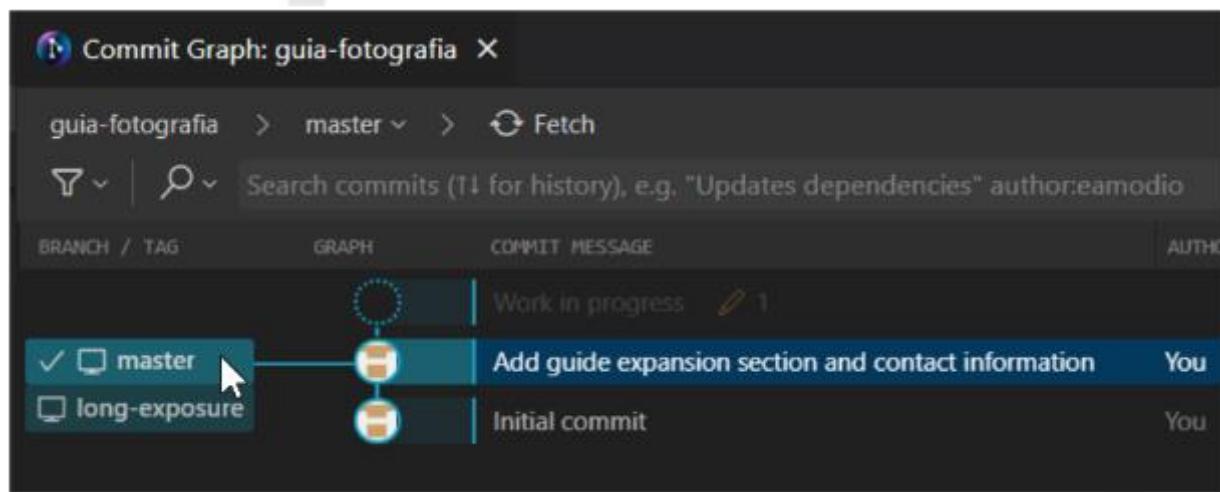
Branches e merge

HEAD

Mesmo tendo criado a nova branch, você ainda não deve fazer `git commit`!

Se fizer, quem vai avançar é o marcador da master, não o da long-exposure, quando o que desejamos é o oposto.

Para entender por quê não aconteceria como queremos, coloque o mouse na palavra master no GitLens:



A branch master tem ao lado um ícone ✓ (sempre teve desde o início do projeto), já a branch long-exposure não tem esse ícone.

O símbolo ✓ indica qual branch está selecionada. No Git, o ✓ é chamado de HEAD, que significa "cabeça" em inglês.

Dizemos que "*a HEAD está em master*". Isso significa que a branch selecionada atualmente é a master. Também é comum dizermos "estamos na master".

Pense na HEAD como a sua cabeça: você está na master / a HEAD está na master.

Ao iniciar o projeto, quando o Git cria a branch master, ele também faz a HEAD selecionar master.

Por isso que a master tem estado selecionada desde sempre.

Ao fazer um commit, a branch que está atualmente selecionada avança ao novo commit. Em outras palavras (mais técnicas), a branch selecionada pela HEAD avança ao novo commit.

Por isso que a branch master sempre avançou de commit em commit: porque nós (a HEAD) estávamos na master.

Branches e merge

Segue uma metáfora para enxergar mais facilmente:

- Os commits são casas marcadas com giz no chão, num jogo de “amarelinha”
- As branches são pedrinhas que ficam nas casas
- A HEAD é você, em pé numa casa da amarelinha
- A HEAD tem alguma branch selecionada.

Ou seja, você está segurando uma pedrinha.

- Ao fazer um novo commit, a HEAD avança ao novo commit levando a branch junto.

Ou seja, você (HEAD) avança para a próxima casa da amarelinha, levando consigo a pedrinha (branch) que estava segurando.

- As outras branches (que a HEAD não está selecionando) ficam paradas no commit onde estão.

Ou seja, ao avançar para a próxima casa da amarelinha, você (HEAD) não leva consigo as pedrinhas (branches) que não está segurando. Elas ficam onde estão.

E por que você precisa saber sobre a HEAD ? Porque lembre que queremos fazer agora um novo commit para salvar a versão do guia sobre Longa Exposição. Mas não queremos que a branch master avance para esse novo commit, mas sim que a branch long-exposure avance.

Portanto precisamos, antes de fazer o commit, trocar a branch selecionada pela HEAD: a HEAD precisa soltar a master e selecionar a long-exposure. Assim ao fazer o novo commit, a branch long-exposure vai avançar, e não a master.

Git checkout (branches no mesmo commit)

Queremos que o próximo commit não faça a branch master avançar, mas sim a branch long-exposure. Então precisamos fazer a HEAD selecionar a long-exposure em vez da master.

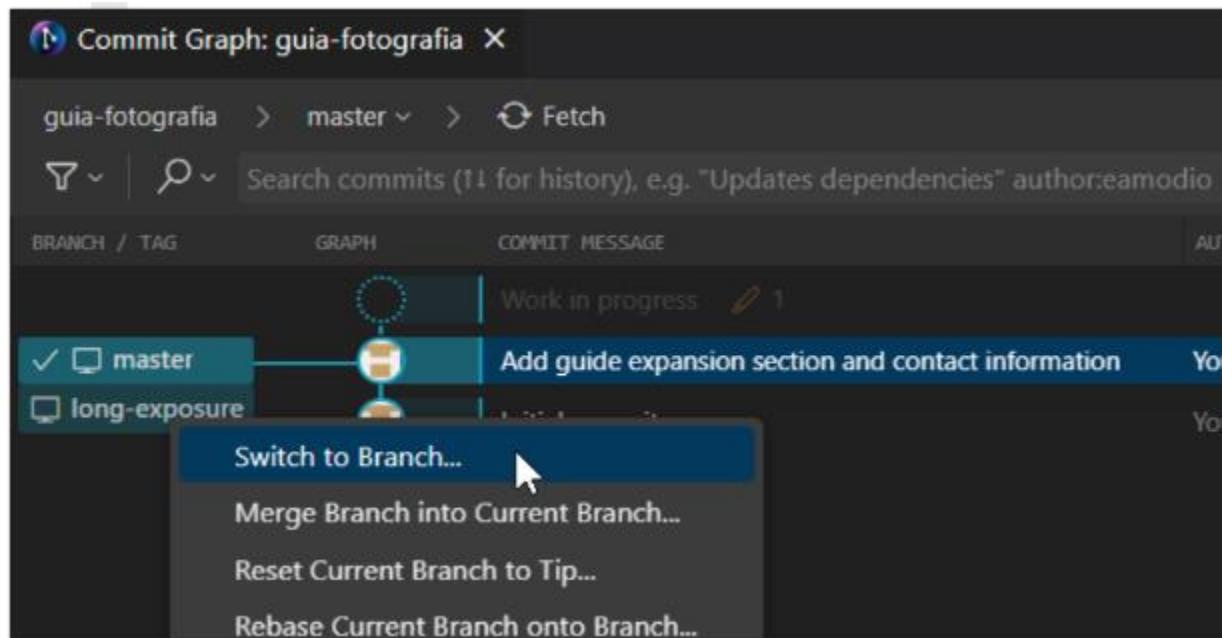
Na metáfora da amarelinha, você (HEAD) precisa soltar a pedrinha da master e segurar a da long-exposure.

Como faz isso ? Para mudar a HEAD para outra branch pelo terminal, o comando é `git checkout <nome da branch>`. Neste caso:

- `git checkout long-exposure`

Se preferir fazer pelo VSCode, abra a visão de Commit Graph do GitLens, clique com o botão direito em long-exposure, e então em “Switch to Branch”:

Branches e merge



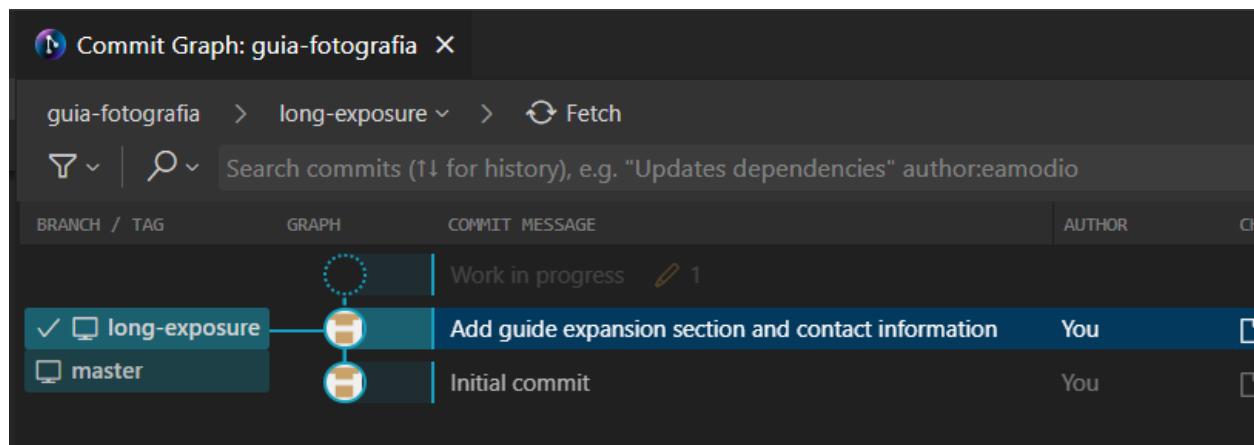
Depois de fazer o checkout, podemos confirmar que ele funcionou.

Pelo terminal, o git log permite ver isso:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git log --oneline
503517d (HEAD -> long-exposure, master) Add guide expansion sec
aef0f34 Initial commit
```

Podemos ver que a HEAD seleciona a long-exposure agora, não mais a master.

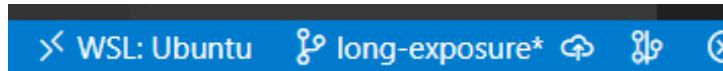
Alternativamente, podemos ver pelo VSCode:



Podemos ver que a long-exposure está selecionada agora (símbolo ✓), em vez da master.

Outra maneira de ver pelo VSCode é a barra azul inferior do programa:

Branches e merge



Essa barra sempre mostra a branch selecionada pela HEAD.

Pela barra também é possível trocar a HEAD de uma branch a outra. Experimente !



Nosso procedimento foi na ordem:

- modificar o que queríamos no guia e dar `git add`
- `git branch` para criar a branch `long-exposure`
- `git checkout` para selecionar a nova branch

Mas não precisa ser nessa ordem.

Você pode criar a branch e trocar para ela antes de fazer as modificações nos arquivos.

Ou pode criar a branch, modificar os arquivos, e depois trocar para a nova branch.

Em outras palavras, as modificações nos arquivos e a criação/troca de branch são duas coisas independentes.

Afinal as branches são somente marcadores de commit (pedrinhas nas casas da amarelinha).



Um atalho para criar uma branch e já mudar a HEAD para ela é o comando `git checkout -b <nome da branch>`.

Isso é equivalente a executar `git branch <nome da branch>` primeiro e depois `git checkout <nome da branch>`.

Branches e merge

Git commit

Agora que HEAD seleciona long-exposure (em vez de master), faça o git commit !

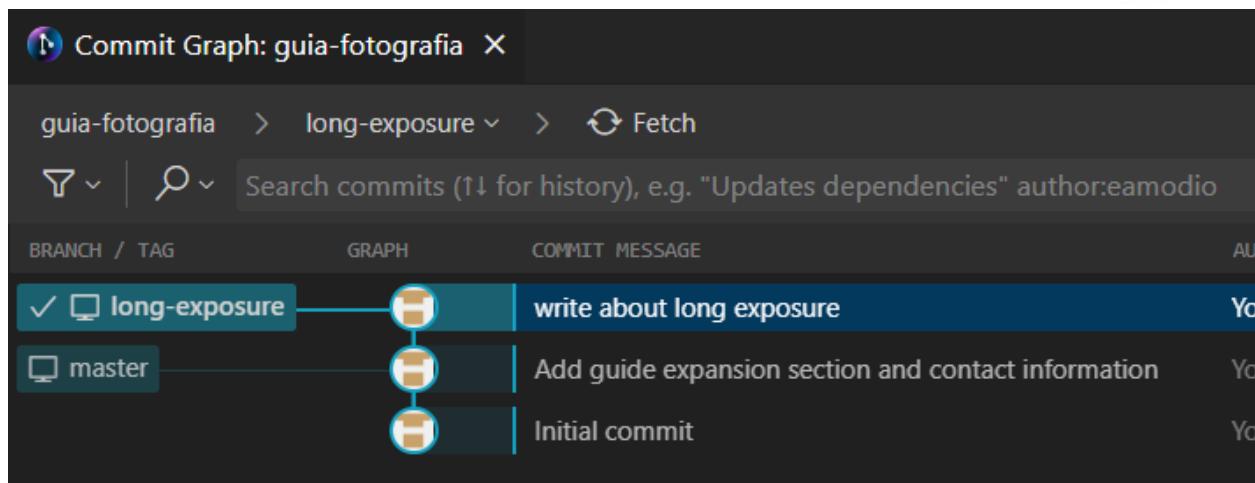
Por exemplo git commit -m "write about long exposure"

O resultado será o seguinte:

Visualizando pelo terminal:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git log --oneline
1eed3e4 (HEAD -> long-exposure) write about long exposure
503517d (master) Add guide expansion section and contact information
aef0f34 Initial commit
```

Visualizando pelo VSCode:



Podemos ver que o novo commit foi feito e a branch long-exposure avançou até ele, mas a master ficou onde estava.

Isso é o que queríamos. Seu programinha de publicação automática ao blog não publicará o novo commit porque a master não está nele.

Mais commits na mesma branch

Você pode fazer mais commits se quiser.

A cada commit, a branch long-exposure vai mudar de lugar, enquanto a master ficará parada.

Por exemplo, na seção sobre Longa Exposição que você acabou de escrever, suponha que tenha sentido falta de dar alguns exemplos onde essa técnica de fotografia é usada.

Então escreva agora esses exemplos no fim do arquivo:

Branches e merge

```

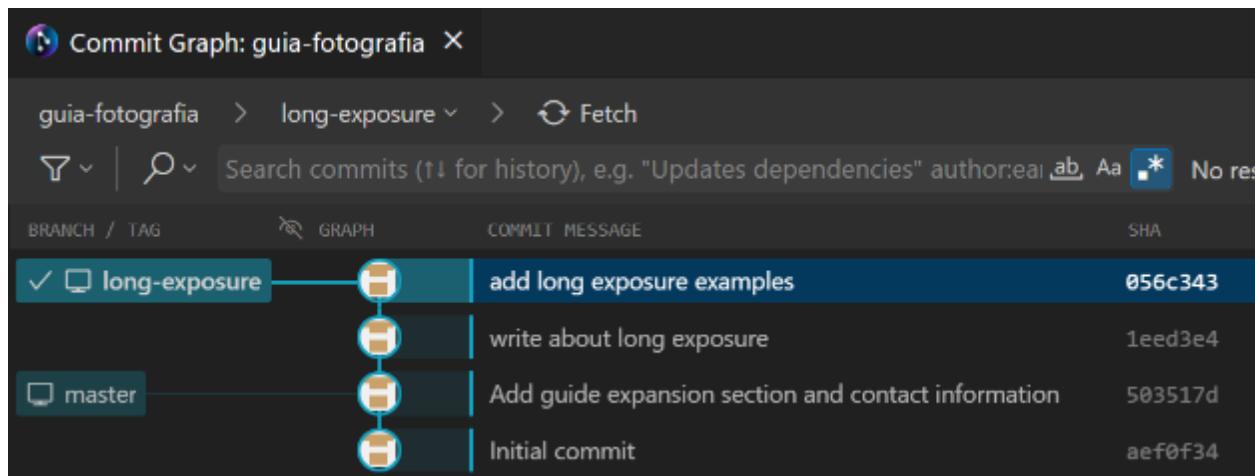
guia.txt M X
guia.txt
10
11 O Básico da Câmera Digital
12 -----
13 Antes de mais nada, familiarize-se com os principais componentes da sua câmera
14
15 * **Sensor**: o coração da câmera, captura a luz e cria a imagem.
16 * **Lente**: controla a entrada de luz e foca a imagem no sensor.
17 * **Diafragma**: ajusta a quantidade de luz que entra na câmera.
18 * **Obturador**: controla o tempo durante o qual a luz atinge o sensor.
19
20 Longa Exposição
21 -----
22 A técnica de longa exposição é fascinante, permitindo que fotógrafos criem ima
23 Essencial para capturar cenas com baixa luminosidade ou movimento deliberadame
24
25 * **Tripé**: Para evitar tremidas, um tripé é essencial.
26 * **Tempo de Exposição**: Aumente o tempo de exposição para segundos ou até mi
27 * **Câmera**: Use um modo manual ou de prioridade de obturador para controlar
28 * **Filtro ND**: Um filtro de densidade neutra pode ser necessário para evitar
29
30 Exemplos práticos incluem fotografar cachoeiras com a água parecendo seda
31 ou ruas da cidade com luzes de carros traçando linhas através do quadro.
32 +

```

Agora siga os passos usuais:

- git add guia.txt
- git commit -m "add long exposure examples"

Com isso, o histórico fica assim:



Branches e merge

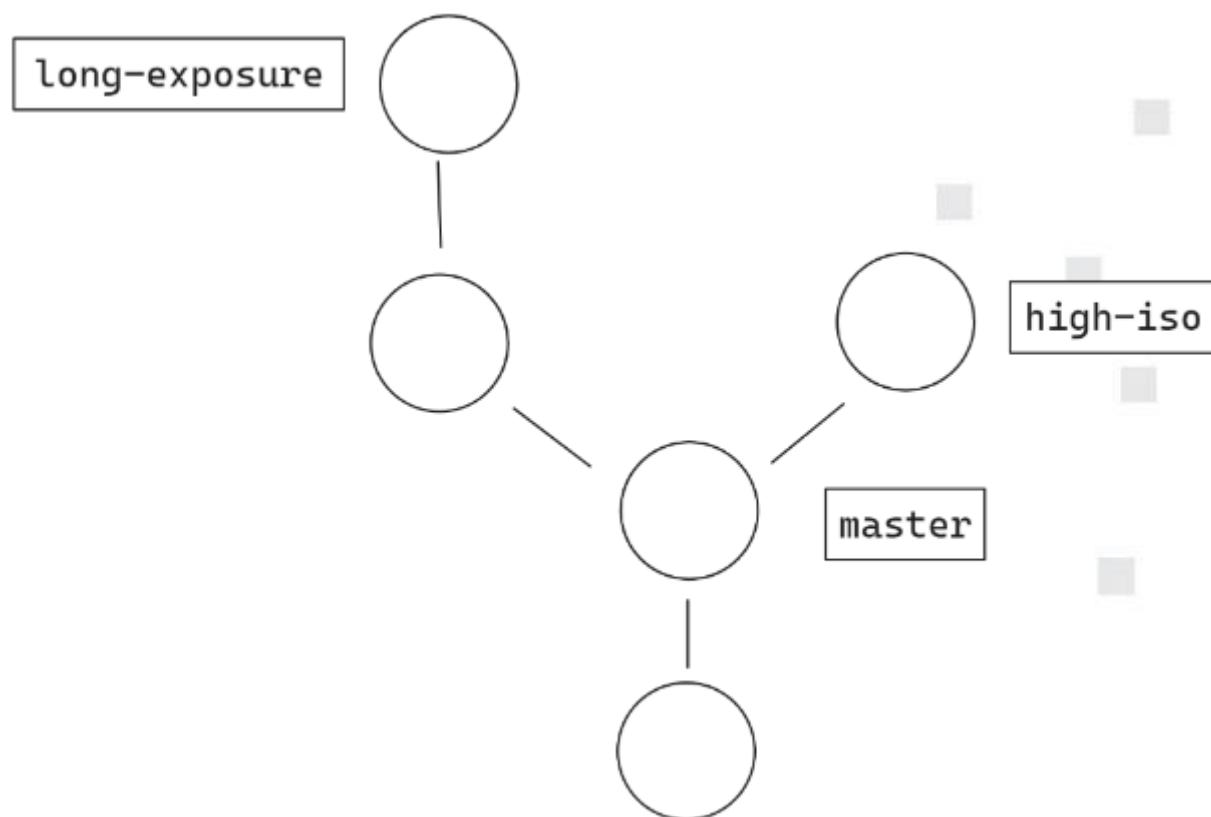
Ou seja, a branch `long-exposure` avançou ao novo commit como esperado.

Se você não tem mais nada para escrever sobre Longa Exposição, vamos fazer agora a outra versão, sobre ISO Elevado.

Git checkout (branches em commits diferentes)

Agora que a versão do guia que fala sobre Longa Exposição está finalizada, você quer escrever a outra versão, sobre ISO Elevado.

Vamos lembrar que o objetivo é ficar com um histórico de commits que se parece com o seguinte no GitLens (bolinhas são commits, retângulos são branches):



Nós já temos o lado esquerdo desse diagrama.

O commit com a branch `long-exposure` é onde estamos (i.e. a HEAD está atualmente).

Queremos criar um outro commit marcado pela branch `high-iso`, que será a versão sobre ISO Elevado.

A primeira coisa que precisamos fazer é levar a HEAD para a branch `master` de volta.

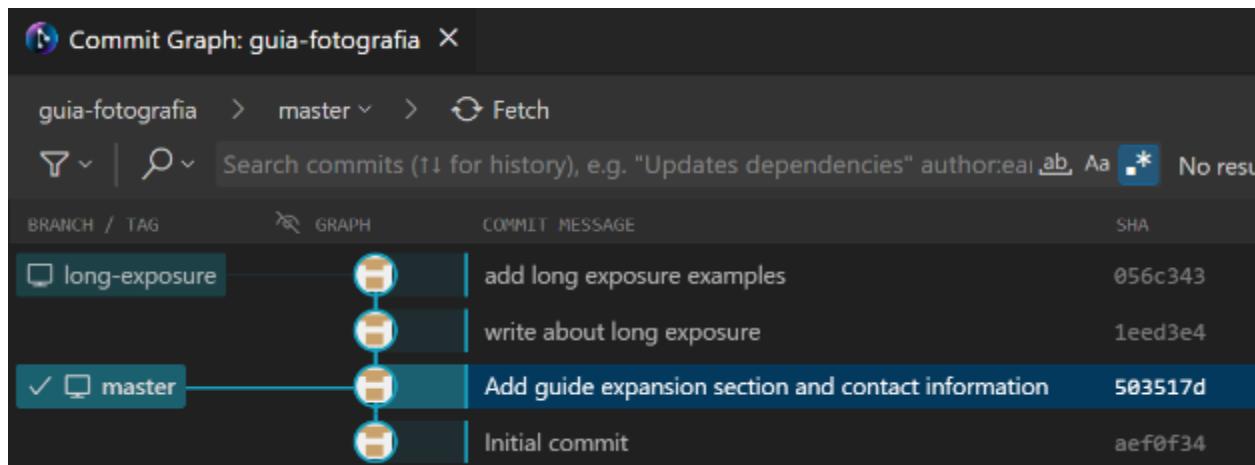
Você já sabe como fazer isso pelo terminal:

- `git checkout master`

Ou use o GitLens, como já explicamos.

Uma vez feito o checkout, o GitLens vai mostrar o seguinte:

Branches e merge



The screenshot shows a GitLens Commit Graph interface for the repository 'guia-fotografia'. The current branch is 'master'. The commit history for 'long-exposure' is shown above 'master', and the commits for 'master' are shown below it. The commits are:

BRANCH / TAG	GRAPH	COMMIT MESSAGE	SHA
long-exposure		add long exposure examples	056c343
long-exposure		write about long exposure	1eed3e4
✓ master		Add guide expansion section and contact information	503517d
✓ master		Initial commit	aef0f34

Vemos que a HEAD (símbolo ✓) está em master como queríamos.

Também podemos checar pelo terminal, mas você vai se surpreender:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git log --oneline
503517d (HEAD -> master) Add guide expansion section and conta
aef0f34 Initial commit
```

Cadê os outros commits ?

Eles não sumiram ! É que o git log assume que, como você (HEAD) está na master, não deve estar interessado no restante dos commits.

Por isso eles não são mostrados.

Mesmo assim é possível fazer todos os commits aparecerem com git log --all:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git log --oneline --all
056c343 (long-exposure) add long exposure examples
1eed3e4 write about long exposure
503517d (HEAD -> master) Add guide expansion section and contact information
aef0f34 Initial commit
```

Agora sim dá para ver que está tudo conforme o que o GitLens mostra.

Mas ainda tem outra coisa que vai te surpreender.

Antes de mudar a HEAD para master, a HEAD estava em long-exposure, certo ?

E o arquivo guia.txt tinha a seção Longa Exposição no final, certo ?

Agora que você (HEAD) está na master, olhe de novo o que está escrito no guia.txt:

Branches e merge

```
guia.txt
```

```
guia.txt
10
11      O Básico da Câmera Digital
12      -----
13 +    Antes de mais nada, familiarize-se com os principais componentes:
14
15      * **Sensor**: o coração da câmera, captura a luz e cria a imagem
16      * **Lente**: controla a entrada de luz e foca a imagem no sensor
17      * **Diafragma**: ajusta a quantidade de luz que entra na câmera
18      * **Obturador**: controla o tempo durante o qual a luz atinge o sensor
19
20      Expansão do Guia
21      -----
22
23      Este guia pode ser expandido para incluir técnicas avançadas como:
24
25      * **Longa Exposição**: captura o movimento suave de elementos como fumaça ou água
26      * **ISO Elevado**: permite fotografar em condições de baixa luminosidade
27
28      Ambas as técnicas requerem compreensão aprofundada da interação entre a luz e o sensor, podendo transformar completamente uma fotografia.
```

Ué ! Não tem mais a seção Longa Exposição no final, no lugar dela está a seção Expansão do Guia de novo. Por quê ?

O que houve foi o seguinte: o comando `checkout`, além de mudar a `HEAD` para `master`, também reverteu a sua Working Tree (i.e. seus arquivos) para a versão do commit da `master`.

E no commit da `master`, o `guia.txt` tinha uma seção Expansão do Guia no final (está lembrado ?)

Em outras palavras, nós voltamos no tempo, para a versão do commit onde está a `master`, que é o commit 5035.

Então o `git checkout` faz duas coisas:

- muda a `HEAD` para a branch desejada
- reseta a Working Tree para que ela coincida com o conteúdo do novo commit onde a `HEAD` foi parar

Com um diagrama é mais fácil ver o que o `git checkout` fez.

Primeiro, antes de executar esse comando, tínhamos o seguinte diagrama:

Branches e merge

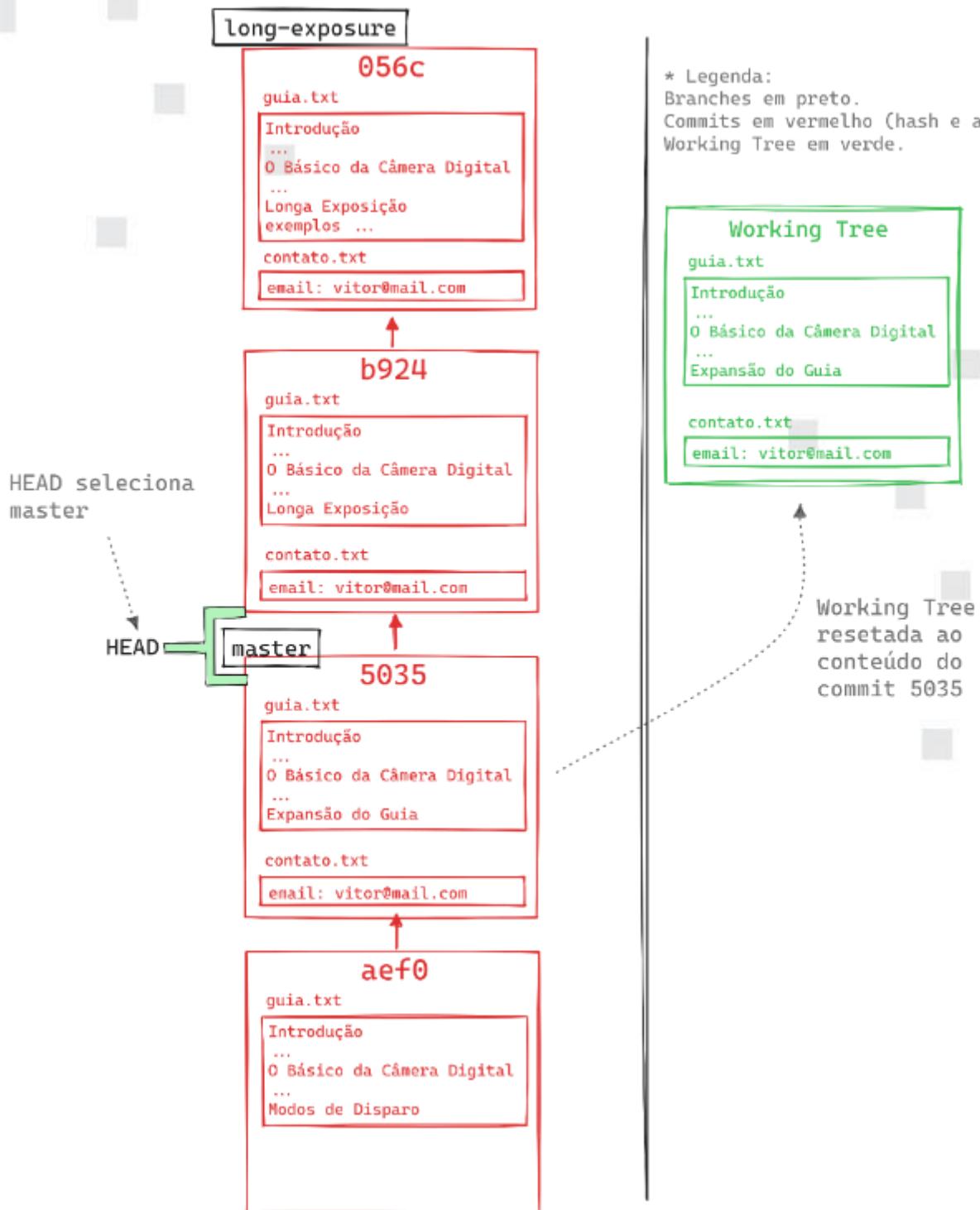


* Legenda:
Branches em preto.
Commits em vermelho (hash e arquivos).
Working Tree em verde.



Branches e merge

O git checkout mudou para o seguinte:



Agora que a HEAD está na `master`, é só proceder da mesma maneira que a outra versão do guia: vamos criar uma nova branch, fazer as alterações que queremos, e dar commit.

Branches e merge

Detalhes do git checkout

Antes de continuar, é importante entender exatamente o que `git checkout` faz para não ter surpresas.

Até agora usamos esse comando duas vezes:

Na seção anterior (segunda vez):

1. A HEAD estava em `long-exposure`, no 4º commit
2. O checkout levou a HEAD para `master`, no 2º commit
3. O checkout reverteu a Working Tree para coincidir com o conteúdo do commit onde a HEAD foi colocada (2º commit)

Já na primeira vez foi um pouco diferente porque as duas branches estavam sobre o mesmo commit:

1. Estávamos no 2º commit com duas branches nele: `master` e `long-exposure`. Ainda não existiam outros commits.
2. A HEAD estava em `master` e tínhamos feito modificações no `guia.txt`, que não estavam commitadas (lembrando de aulas passadas: isso se chama Modificações Locais, são as coisas que aparecem no `git status`)
3. Aí fizemos checkout para trocar a HEAD para `long-exposure` (que está sobre o mesmo commit)
4. Mas não perdemos as Modificações Locais, ou seja, parece que o checkout não reverteu a Working Tree nesse caso ?

Você poderia se perguntar:

Nesse último passo 4, se o `git checkout` reverte a Working Tree para coincidir com o conteúdo do commit de destino (que neste caso é o 2º commit, o mesmo que o commit de origem), por que não perdemos as Modificações Locais ? (alterações feitas no `guia.txt` que não estavam commitadas)

A resposta é que o checkout faz uma reversão *parcial* da Working Tree:

- Arquivos com modificações locais (i.e. tudo que o `git status` mostra) não são tocados, ou seja:
 - Arquivos M (Modified) são mantidos com as modificações
 - Arquivos U (Untracked) não são deletados
 - Arquivos D (Deleted) são mantidos deletados
- Arquivos que não aparecem no `git status` (i.e. todos que você não alterou) são revertidos à versão do commit de destino

Então na maioria das vezes ao fazer `git checkout`, mesmo se a Working Tree estiver “suja” com modificações, você não vai ser atrapalhado pela reversão da Working Tree, porque o Git não vai enconstar nos arquivos que você estava modificando.

Mas há uma minoria das vezes em que o `git checkout` detecta que ele não conseguia manter intactas as suas modificações (não vamos entrar em detalhes do por quê ele não conseguia).

Nesse caso, o `git checkout` vai falhar com uma mensagem (seja pelo terminal ou pelo VSCode):

```
error: Your local changes to the following files would be overwritten by checkout:
      guia.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

Branches e merge

A mensagem diz: o Git percebeu que não conseguiria manter suas modificações intactas, então ele simplesmente não fez o checkout.

Mesmo quando o checkout falha, não há nenhuma chance de você perder o que estava trabalhando: o Git cancela o checkout, nada é alterado, nem a HEAD nem a Working Tree.

Apesar de ser sempre seguro fazer checkout (não há risco de perder dados), quando ele falha você fica com um problema: você queria fazer checkout e não conseguiu.

Se você tiver esse problema, leia a seção desta aula "**Git stash: Git checkout falhou porque Working Tree tem modificações**".



Não custa lembrar que quando o git status mostra que você tem arquivos modificados, ele está comparando o conteúdo da Working Tree com o conteúdo do commit onde está a HEAD.

Se a versão do arquivo na Working Tree difere da versão do mesmo arquivo no commit onde está a HEAD, o arquivo vai aparecer no git status.

- Se ele existe na Working Tree mas não no commit, aparecerá como U (Untracked).
- Se ele existe no commit mas não na Working Tree, aparecerá como D (Deleted).
- Se ele existe em ambos e tem conteúdos diferentes, aparecerá como M (Modified).

Construção da segunda branch

Agora que já escrevemos a versão sobre Longa Exposição, é a vez da versão sobre ISO Elevado.

Recapitulando, nós já movemos a HEAD de volta para master:

BRANCH / TAG	GRAPH	COMMIT MESSAGE	SHA
long-exposure		add long exposure examples	056c343
		write about long exposure	1eed3e4
		Add guide expansion section and contact information	503517d
master		Initial commit	aef0f34

Branches e merge

E não temos modificações locais.

Então comece a escrever sobre ISO Elevado, trocando a seção Expansão do Guia pelo novo conteúdo:

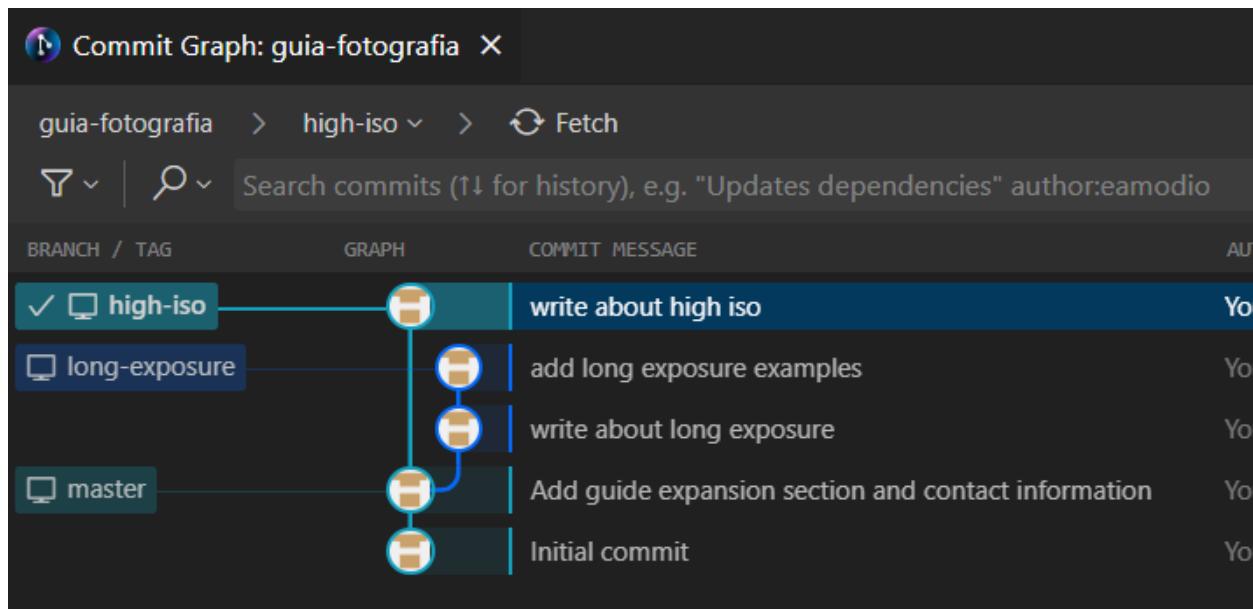
```
guia.txt M X  
guia.txt  
10  
11 O Básico da Câmera Digital  
12 + -----  
13 Antes de mais nada, familiarize-se com os principais componentes  
14  
15 * **Sensor**: o coração da câmera, captura a luz e cria a imagem.  
16 * **Lente**: controla a entrada de luz e foca a imagem no sensor.  
17 * **Diafragma**: ajusta a quantidade de luz que entra na câmera.  
18 * **Obturador**: controla o tempo durante o qual a luz atinge o s  
19  
20 ISO Elevado  
21 -----  
22 O ISO mede a sensibilidade do sensor à luz.  
23 Um ISO elevado pode ser a chave para fotografar em ambientes com  
24 Contudo, ISOs mais altos podem introduzir ruído/grão na imagem.  
25  
26 * **Configuração de ISO**: Aumente o ISO para capturar mais luz c  
27 * **Compensações**: Esteja ciente do compromisso entre luminosida  
28 * **Pós-Processamento**: Programas modernos de edição podem reduz  
29  
30 Esta técnica é útil em eventos internos ou em cenas noturnas onde  
31
```

Aí faça os passos que já sabe:

- git branch high-iso
- git checkout high-iso
- git add guia.txt
- git commit -m "write about high iso"

O resultado é o que desejamos:

Branches e merge



Se quiser ver o histórico no terminal em vez do VSCode, coloque a opção `--graph` senão não vai aparecer a “bifurcação”.

O `--graph` desenha um * para simbolizar cada commit, e linhas para mostrar a conexão entre os commits:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git log --oneline --all --graph
* bd7f8c0 (HEAD -> high-iso) write about high iso
| * 056c343 (long-exposure) add long exposure examples
| * 1eed3e4 write about long exposure
|/
* 503517d (master) Add guide expansion section and contact information
* aef0f34 Initial commit
```



O visual do histórico de commits parece o tronco de uma árvore que ramificou em dois galhos.

Daí vem o nome “branch” que o Git utiliza.

Mas não confunda: uma branch *não é* uma sequência de commits que parece um galho de árvore.

Branch é simplesmente um marcador (pedrinha na metáfora da amarelinha) que fica sobre um commit (casa da amarelinha).

Assim, nosso histórico de commits tem 3 branches (3 marcadores).

Branches e merge

Git merge (fast-forward)

Agora que as duas versões do guia estão escritas, você pode imprimir as duas versões e dar para alguns amigos lerem e decidirem qual é a melhor.

A versão que eles preferirem você vai publicar no blog. Vai ser a próxima “versão de produção” do blog. Suponha que venceu a versão Longa Exposição.

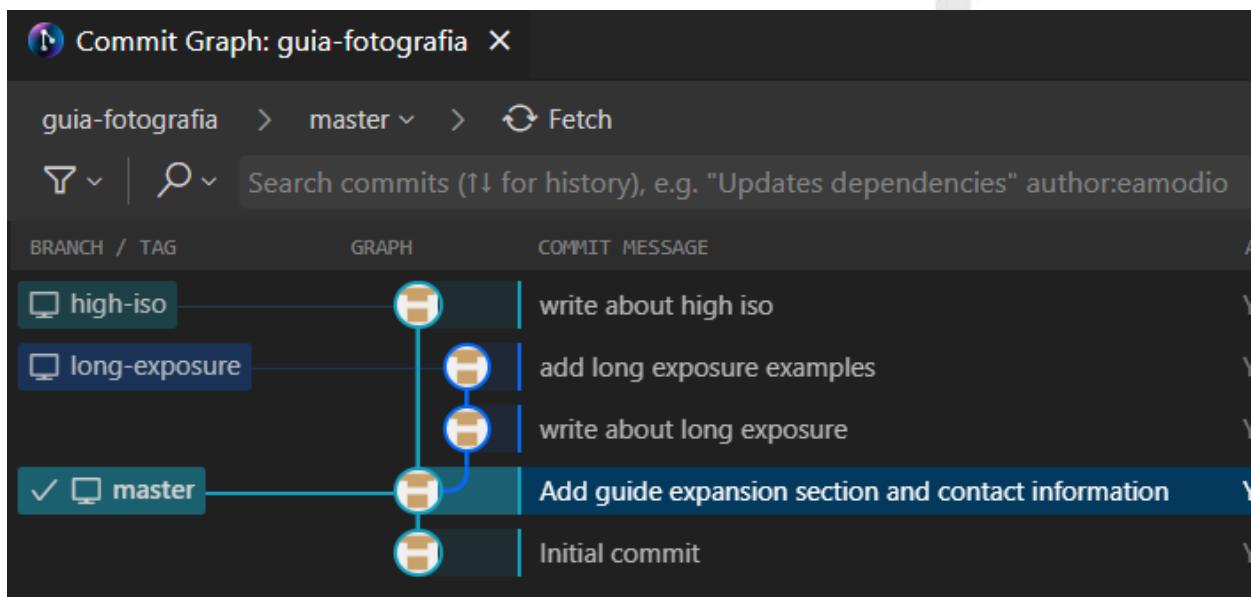
Então precisamos mover o marcador master para colocá-lo no commit marcado pela branch long-exposure (assim o seu programa de publicação automática vai detectar que a master mudou, e vai publicar a nova versão).

No Git, isso é chamado de `merge` (fusão), porque vamos fundir as novidades da long-exposure à master.

Para isso, primeiro mova a HEAD para a branch que vai receber a atualização (master nesse caso):

- `git checkout master`

Com isso ficamos assim:



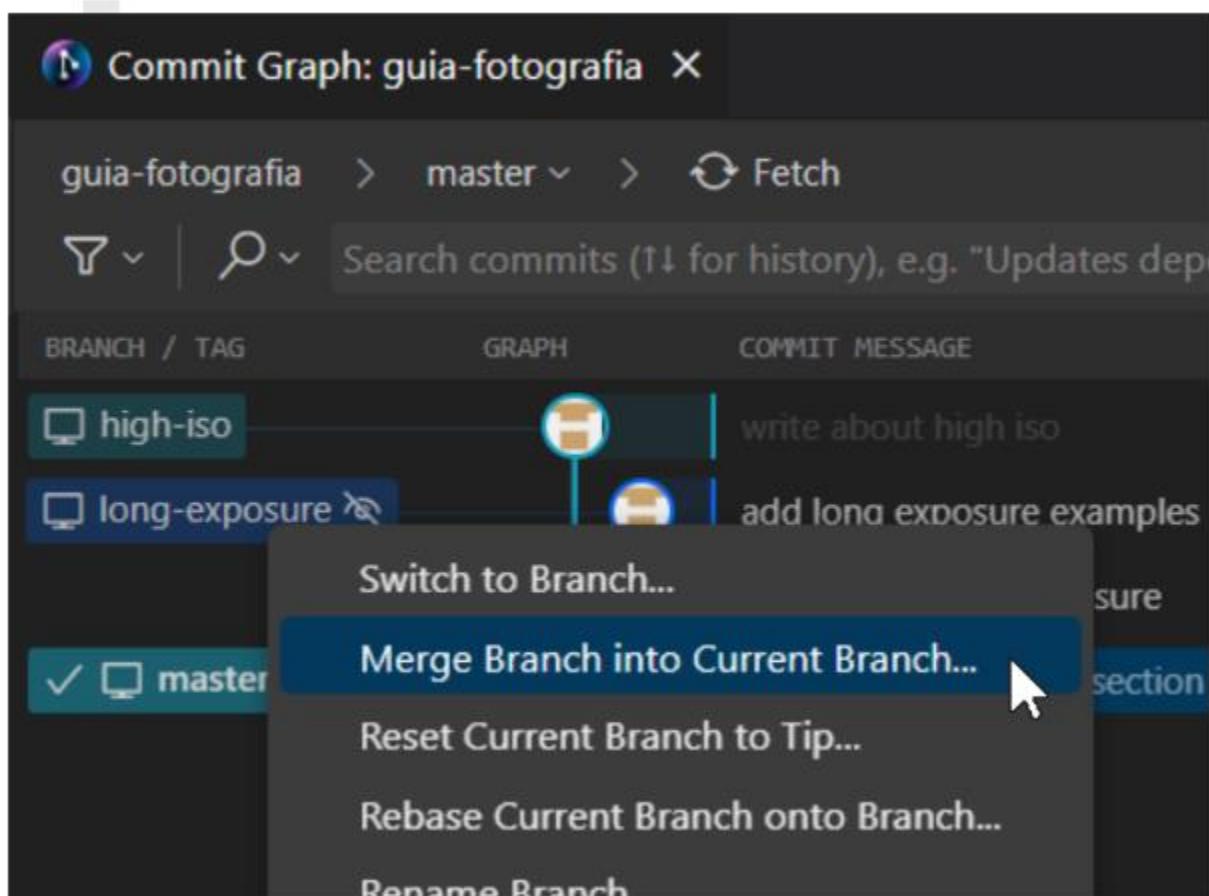
Agora faremos o merge:

Pelo terminal, o comando é `git merge <nome da branch que você quer fundir na branch atual>`. Nesse caso:

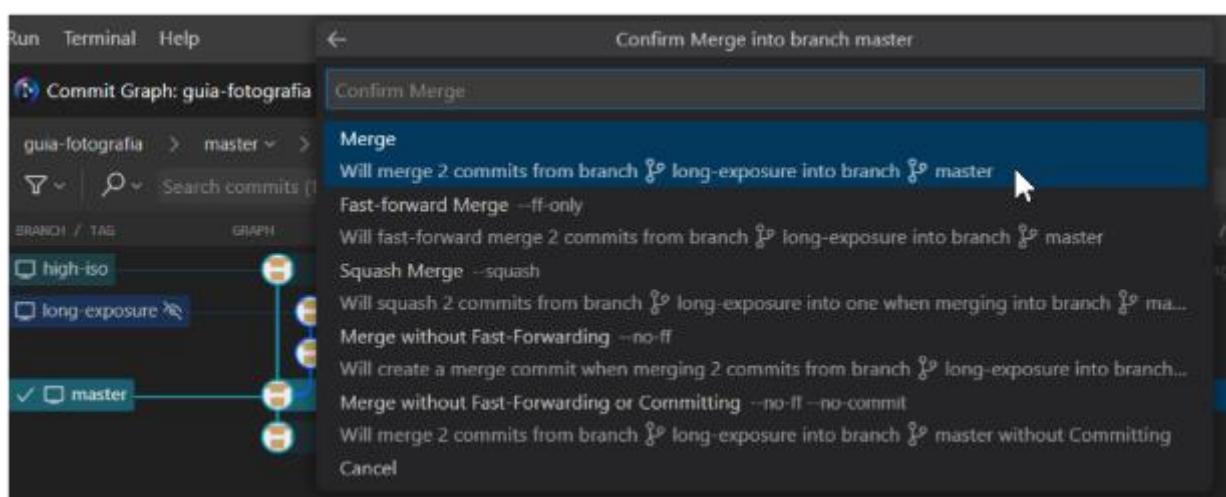
- `git merge long-exposure`

Alternativamente, pelo VSCode clique com o botão direito do mouse em “long-exposure” e escolha a opção “Merge Branch into Current Branch”/“Funda a branch na branch atual”:

Branches e merge

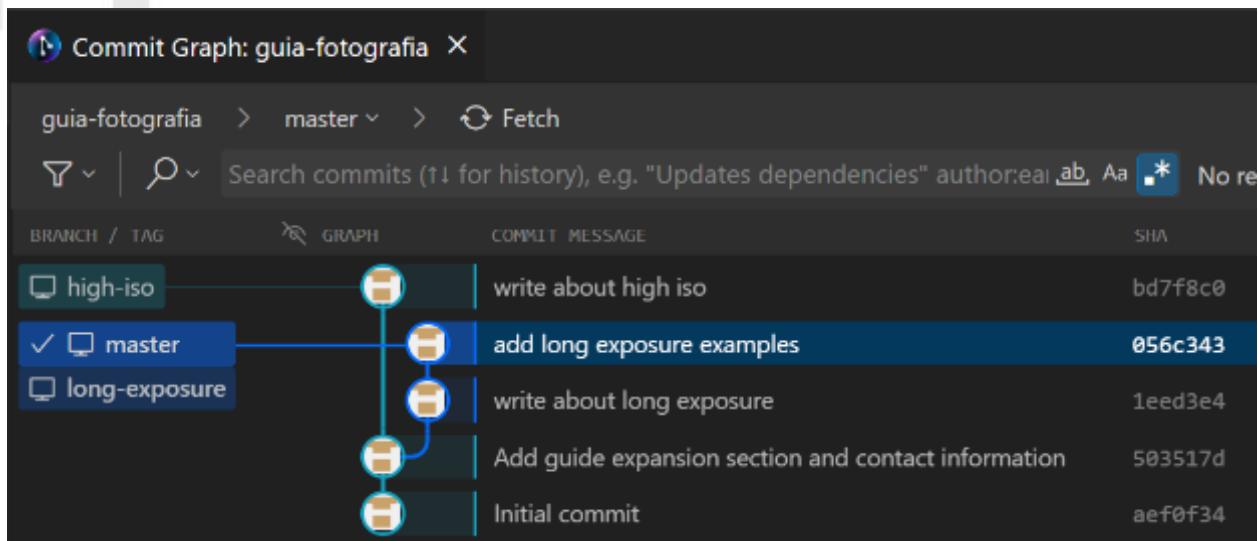


Agora o VSCode vai mostrar algumas opções, escolha a opção padrão (a de cima):



Branches e merge

Está feito o merge. O resultado é o seguinte:



Ou seja, tudo que o Git fez foi mover a master (junto com a HEAD) para o mesmo commit onde está a long-exposure.

Isso é chamado de merge “fast-forward”, porque tudo que aconteceu foi avançar a branch master para frente.

Existe um segundo tipo de merge para casos onde não é possível simplesmente avançar sua branch para frente (veremos em seguida).

Quando você faz um merge (não importa o tipo), a outra branch (long-exposure) não sai do lugar.

Somente se move a sua branch, ou seja, a branch selecionada pela HEAD (no nosso caso, a master).

Apesar disso, dizemos que *“a branch Long-exposure foi mergeada à master”*.



Você pode mergear qualquer branch em qualquer outra.

Basta fazer `git checkout <alguma branch>` e depois `git merge <outra branch>`.



Se você tem mudanças locais, ou seja, arquivos modificados/criados/deletados sem commitar, não tente fazer merge !

Faça commit das mudanças primeiro.

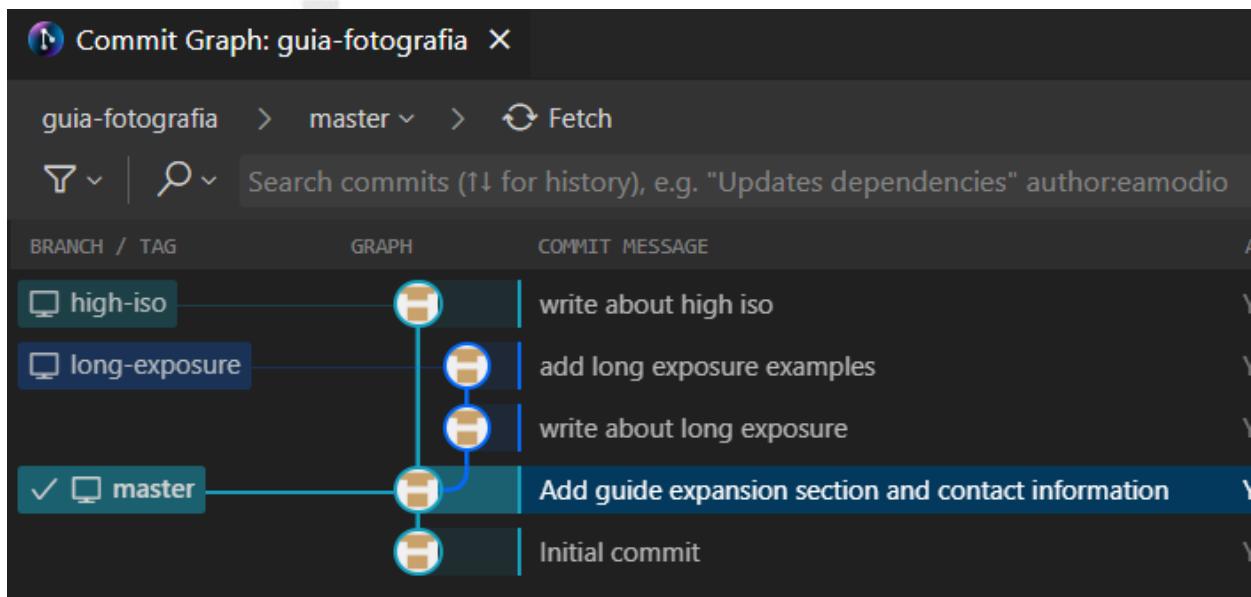
Depois, já com uma Working Tree isenta de modificações, faça o merge que desejar.

Branches e merge

Git merge (three-way)

Vamos voltar um pouco no tempo para ver quando o Git realiza outro tipo de merge que não é fast-forward.

Suponha que você está de novo neste momento:



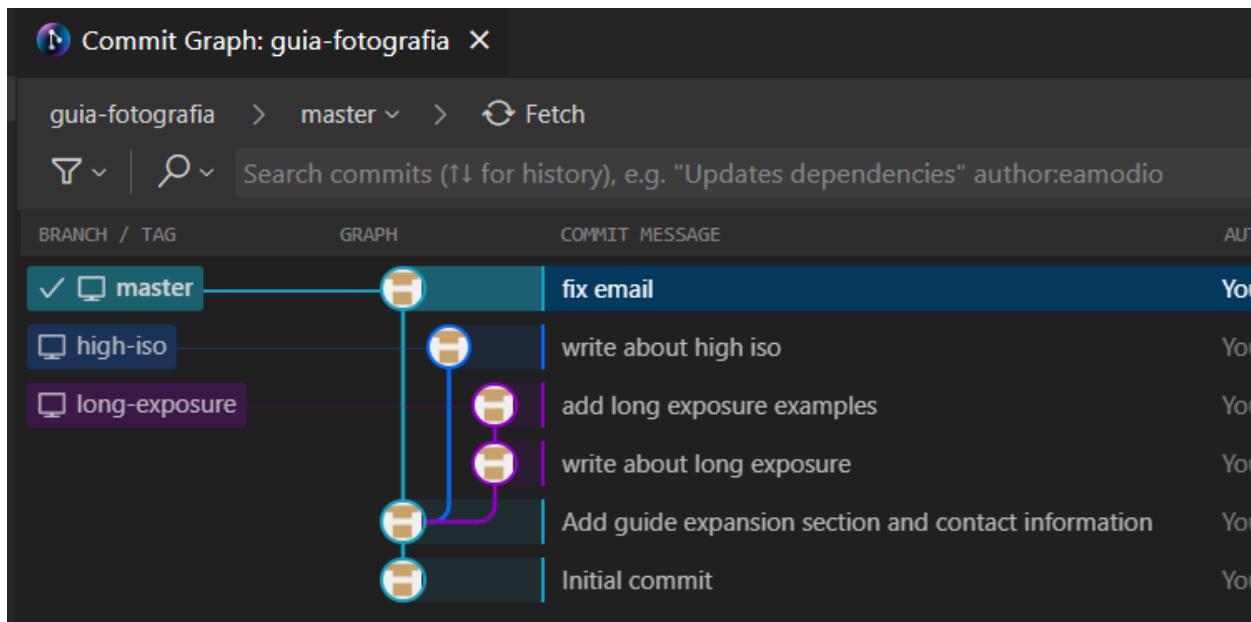
Ou seja, você escreveu as duas versões mas ainda não fez o merge.

E a HEAD está na master (2º commit). Lembramos que o segundo commit tem um arquivo guia.txt e outro contato.txt contendo seu email.

Neste momento, um leitor do seu blog entra em contato para avisar que seu email está escrito errado.

Está vitor@mail.com no arquivo mas deveria ser vitor@gmail.com.

Você sabe que precisa consertar imediatamente o email, então corrige o arquivo contato.txt (na master mesmo) e dá commit, ficando assim:



Branches e merge

Como a master mudou de commit, seu programinha de publicação automática envia a nova versão ao blog imediatamente, que é o que você queria (por isso deu commit na master).

Passada essa urgência, você volta a conversar com seus amigos que estavam lendo as versões high-iso e long-exposure para decidir qual é a melhor.

Eles decidem que long-exposure é melhor, e você decide fazer o merge dessa branch à master para publicar no blog.

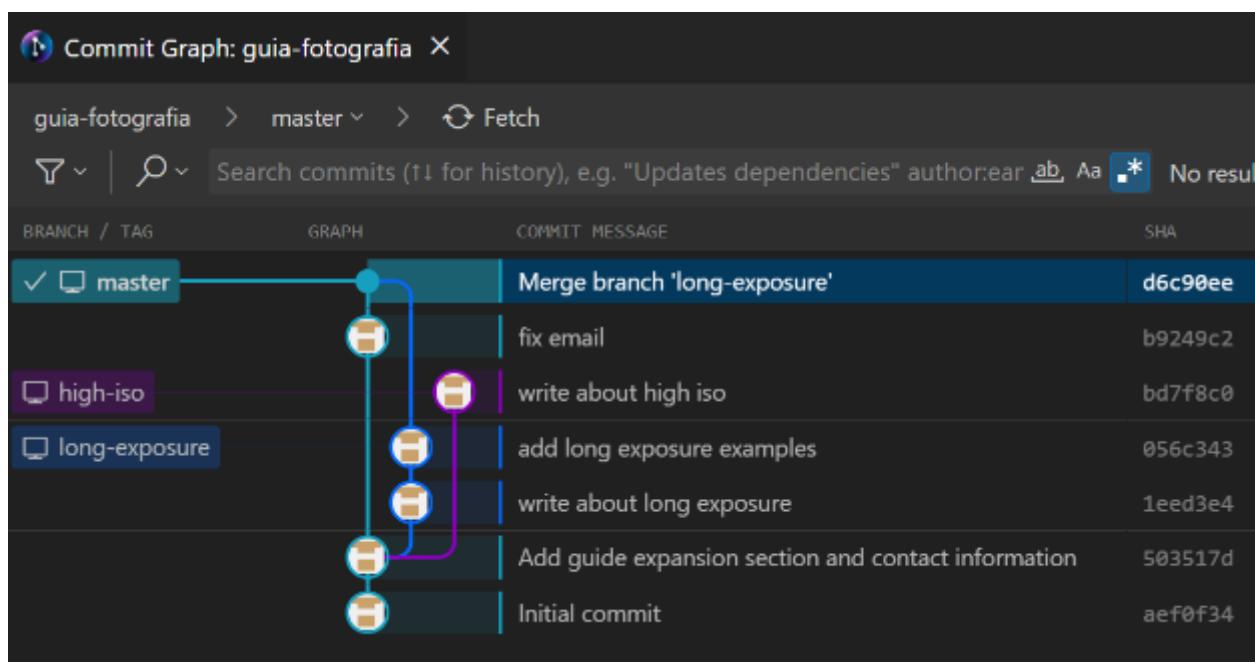
Como a HEAD está na master, você já sabe que basta executar `git merge long-exposure`.

Mas dessa vez o Git não vai fazer um merge fast-forward, ou seja, simplesmente mover a branch master para o mesmo commit da long-exposure.

Se ele fizesse somente isso, estaria jogando fora o conserto do email, que só existe no commit onde a master está agora.

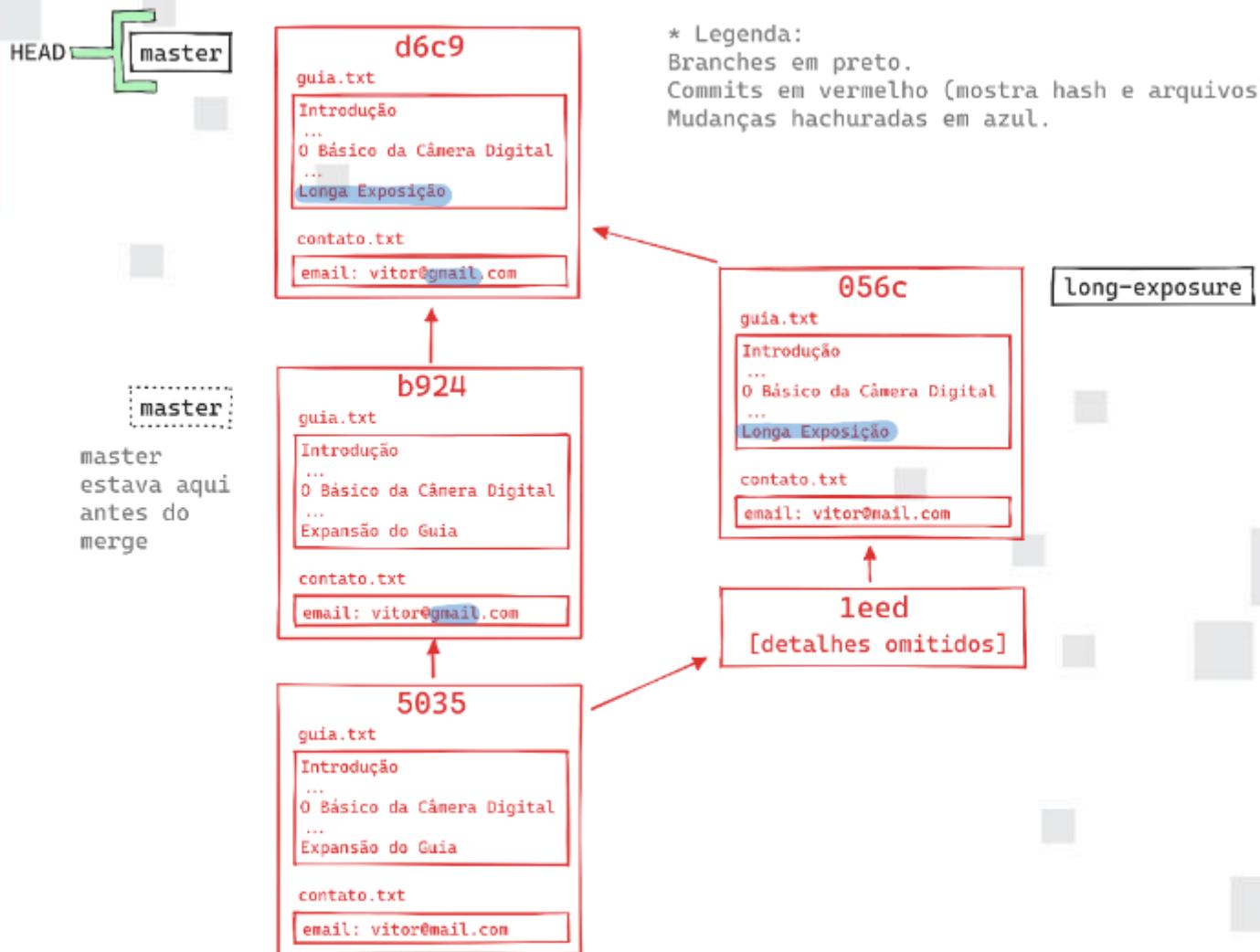
Então o Git vai fazer um merge chamado “three-way”: ele cria um novo commit que funde os dois commits onde estão as branches que estão sendo fundidas.

Vai ficar assim:



Mas é mais fácil visualizar o que aconteceu por meio de um diagrama (compare com a visão do GitLens para se convencer que o diagrama é idêntico):

Branches e merge



O commit d6c9 é o novo, ele é chamado de “commit de merge”. Veja que esse commit possui tanto o conserto do email (vitor@gmail.com no contato.txt) quanto a nova seção sobre Longa Exposição no guia.txt.

A mensagem “Merge branch ‘long-exposure’” vista no GitLens é sugerida automaticamente pelo git ao executar o `git merge`.

Esse novo commit está ligado a dois outros: o b924 onde a master estava, e o 056c onde a long-exposure está (lembremos que o merge só movimenta a branch da HEAD, que é a master, a outra branch long-exposure fica no lugar).

E a HEAD (carregando a master) avançou para o novo commit.

Depois do merge, se você abrir os arquivos guia.txt e contato.txt, verá que:

- o guia.txt tem a seção Longa Exposição (veio do commit 056c)
- o contato.txt tem o email corrigido (veio do commit b924)

Ou seja, o commit de merge funde as mudanças de cada um dos dois commits que foram fundidos.

E isso é o desejado, afinal você quer preservar o email corrigido, e também quer ter a seção sobre Longa Exposição.

Branches e merge



Commits “normais” são feitos com `git commit` enquanto um commit de merge é feito com `git merge`.

Mas depois de feito, um commit de merge é como qualquer outro.

Você pode continuar trabalhando em cima dele fazendo mais commits.

Não tem nada de especial.

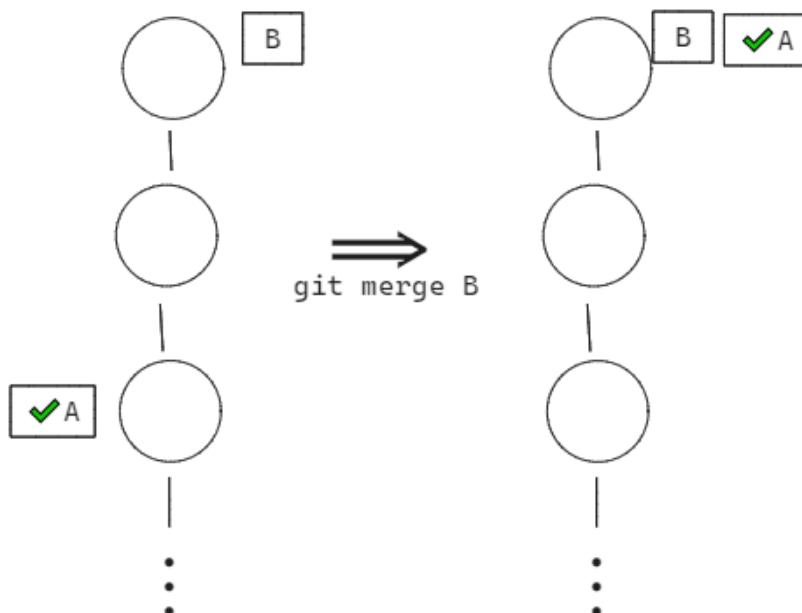
Quando ocorre merge fast-forward ou three-way ?

Suponha que você (HEAD) está numa branch A e faz `git merge B` onde B é outra branch.

Então cada tipo de merge vai ocorrer nas seguintes situações:

Fast-forward:

Ocorre quando existe uma “linha reta” de avanço entre o commit da branch A e o commit da branch B.
O Git vai mover a branch A para frente, colocando-a sobre o commit da branch B.

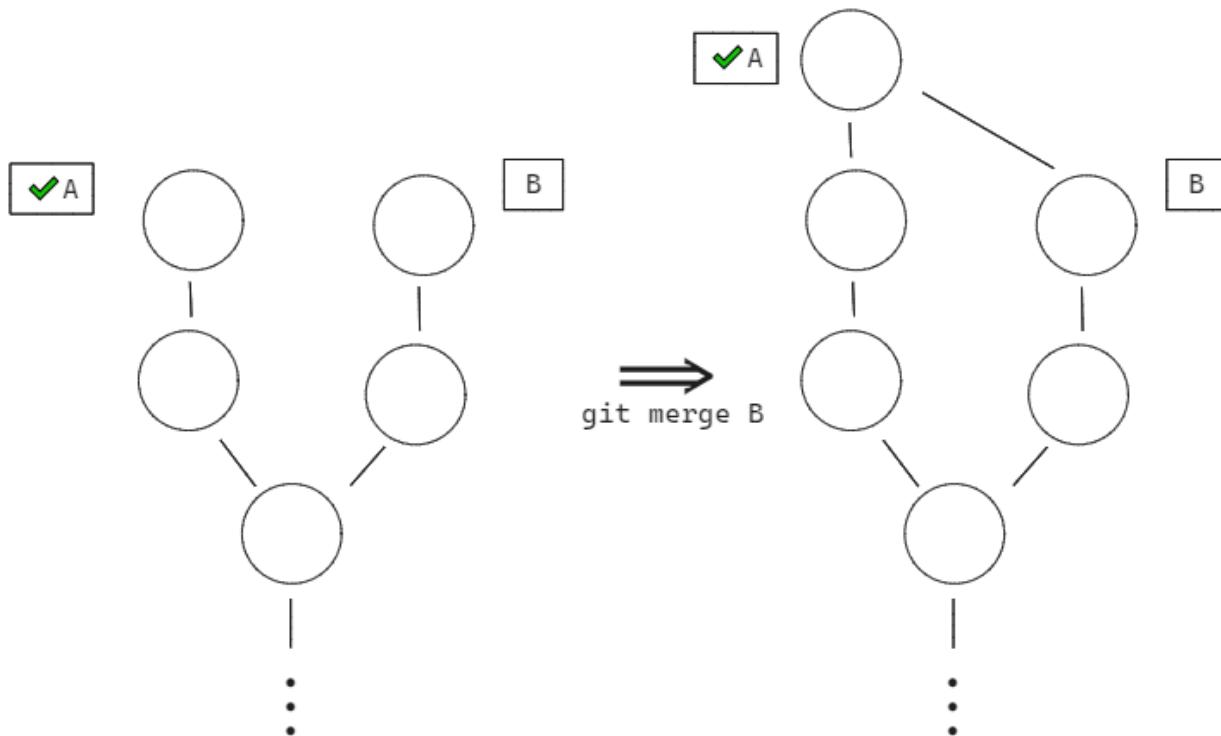


Branches e merge

Three-way:

Ocorre quando não é possível o fast-forward.

O Git vai criar um novo commit que funde os commits das branches A e B, depois vai mover a branch A para esse novo commit.



Branches e merge

Detalhes do merge three-way

O nome three-way vem de um conceito de “três pontos de vista” (three-way view).

No nosso exemplo do projeto de fotografia, esses 3 pontos de vista são os seguintes commits:

- b924, commit onde a master estava
- 056c, commit onde a long-exposure está
- 5035, o último commit comum na história dos outros dois

Ao criar o commit de merge, o Git olha para os dois commits sendo fundidos e precisa decidir quais conteúdos de cada arquivo serão mantidos.

Por exemplo, o guia.txt em b924 tem a seção Expansão do Guia. Já o guia.txt em 056c tem no mesmo lugar a seção Longa Exposição.

Qual seção deve ser mantida no commit de merge ?

Para decidir isso existe o terceiro ponto de vista, que é o commit 5035.

O Git olha para o guia.txt do commit 5035 e percebe que ele já tinha a seção Expansão do Guia, enquanto o 056c mudou para Longa Exposição.

Então ele conclui que o b924 na verdade não mudou nada, foi o 056c que introduziu uma mudança.

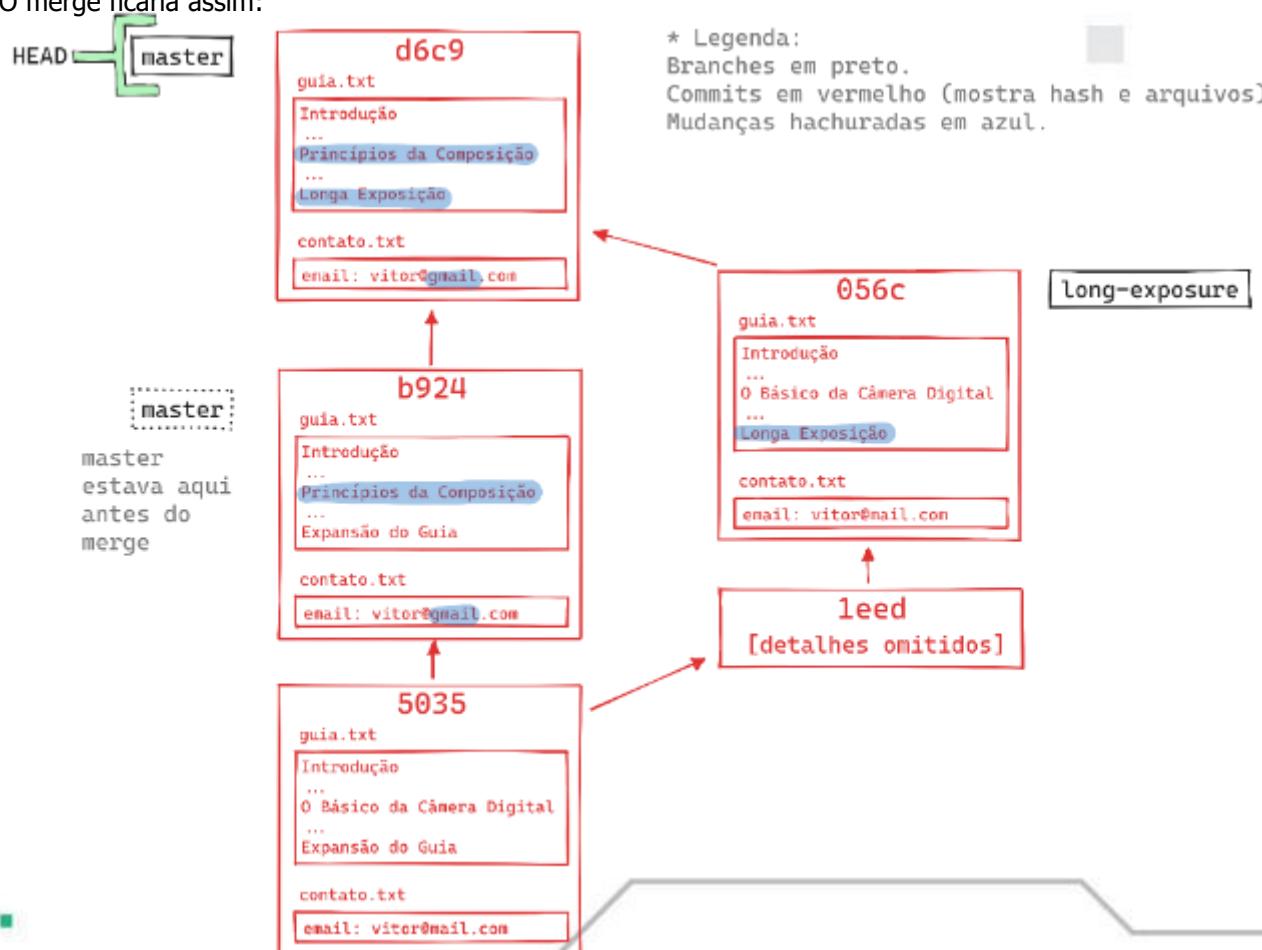
Portanto, para preservar as mudanças, o Git automaticamente deduz que deve ser preservada a seção Longa Exposição.

Para o outro arquivo contato.txt a lógica é semelhante.

Mas é importante observar que um mesmo arquivo no commit de merge pode preservar partes dos dois outros commits.

Por exemplo, suponha hipoteticamente que o commit b924 tivesse trocado a seção “O Básico da Câmera Digital” por “Princípios da Composição”.

O merge ficaria assim:



Branches e merge

Ou seja, o guia.txt no commit de merge preservou a seção “Princípios da Composição” de b924 e “Longa Exposição” de 056c.

A lógica é a mesma: o Git tenta preservar as mudanças.

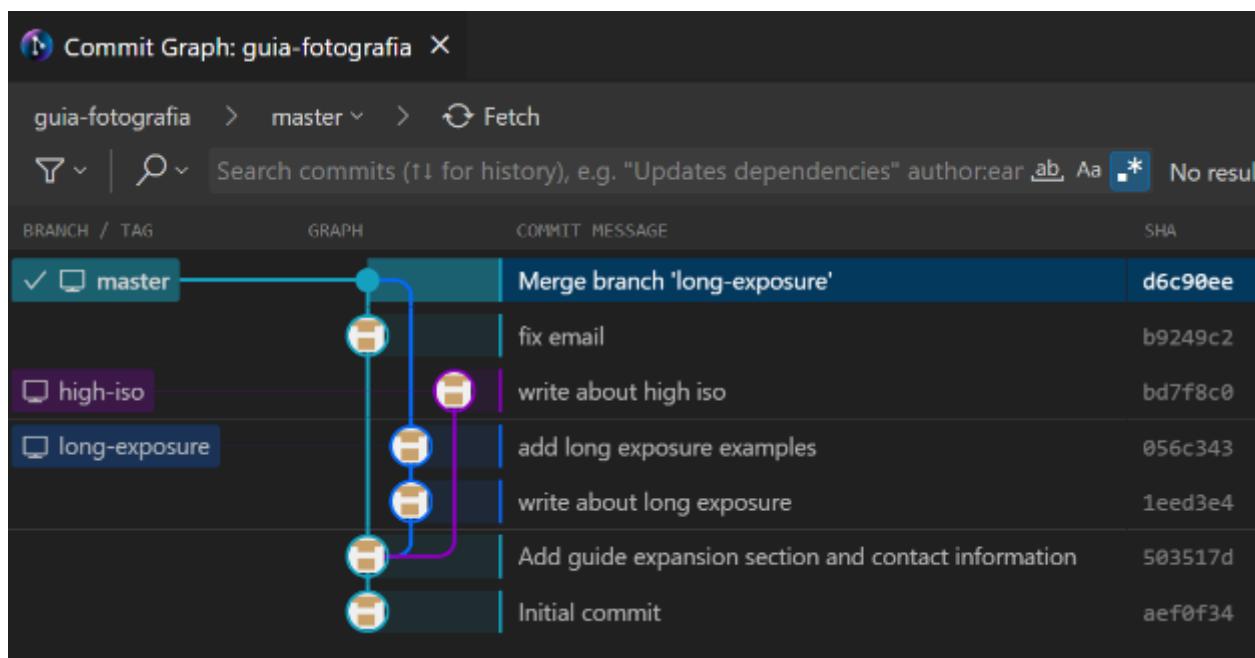
Se ele percebe que um dos commits sendo fundidos tem uma mudança, e o outro não tem mudança nas mesmas linhas do arquivo, ele vai preservar a versão que tem mudanças.

Merge com conflitos

Suponha que você já fez o merge de long-exposure sobre a master.

Vamos assumir que foi o merge three-way, mas poderia ser o fast-forward.

Estamos com a Working Tree limpa, a HEAD está em master, e o histórico está assim:



E o commit atual tem o seguinte conteúdo:

d6c9
guia.txt
Introdução
...
Princípios da Composição
...
Longa Exposição

contato.txt
email: vitor@gmail.com

Branches e merge

O blog está publicado com a nova seção Longa Exposição no final, e tudo parece bem.

A versão high-iso não foi deletada, mas também não foi incorporada ao blog, exatamente de acordo com seu plano inicial de publicar somente uma das duas seções.

Mas, inesperadamente, você está recebendo emails dos leitores do blog. Eles estão pedindo que você escreva sobre ISO Elevado também !

Então você decide que precisa incorporar a seção "ISO Elevado" que está na branch high-iso.

A ideia é que o guia.txt fique com as seções que ele já tem e adicionalmente a seção "ISO Elevado" no final.

Você tenta proceder como antes:

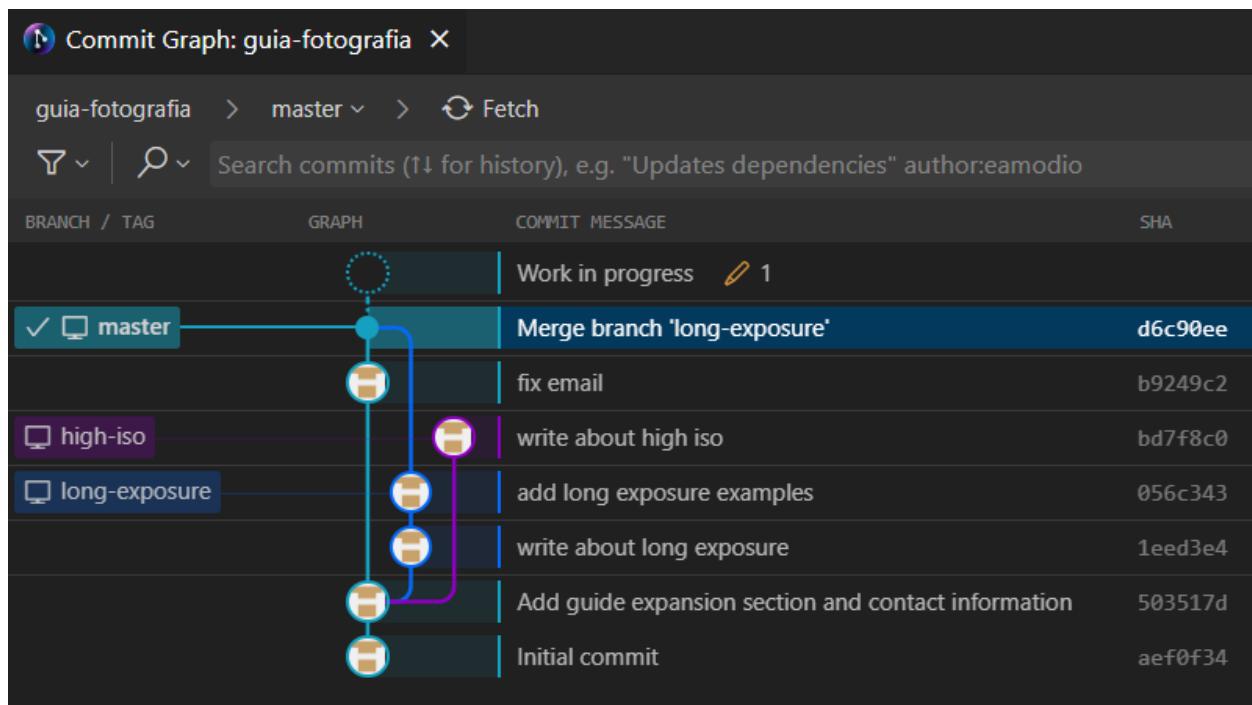
- Olha de novo para confirmar que a HEAD está em master, a branch que vai receber as novidades da high-iso
- Executa git merge high-iso

Mas desta vez acontece algo diferente:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git merge high-iso
Auto-merging guia.txt
CONFLICT (content): Merge conflict in guia.txt
Automatic merge failed; fix conflicts and then commit the result.
```

A mensagem no terminal diz que o merge automático falhou, e que tem algum tipo de "conflito" no guia.txt.

Você olha o histórico de commits e vê que o merge falhou mesmo. O histórico está igual antes, não tem um commit novo:



Branches e merge

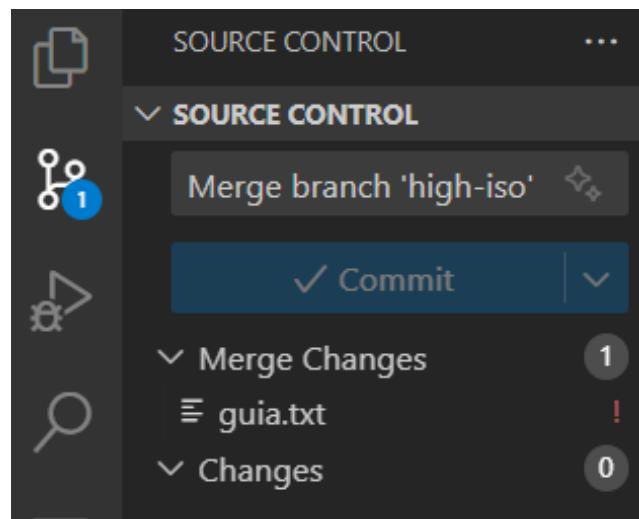
Para ver a situação em que está, você executa `git status` e também vê algo diferente:

Pelo terminal:

```
vitor@DESKTOP-QG84R3G:~/guia-fotografia$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   guia.txt
```

Pelo VSCode:



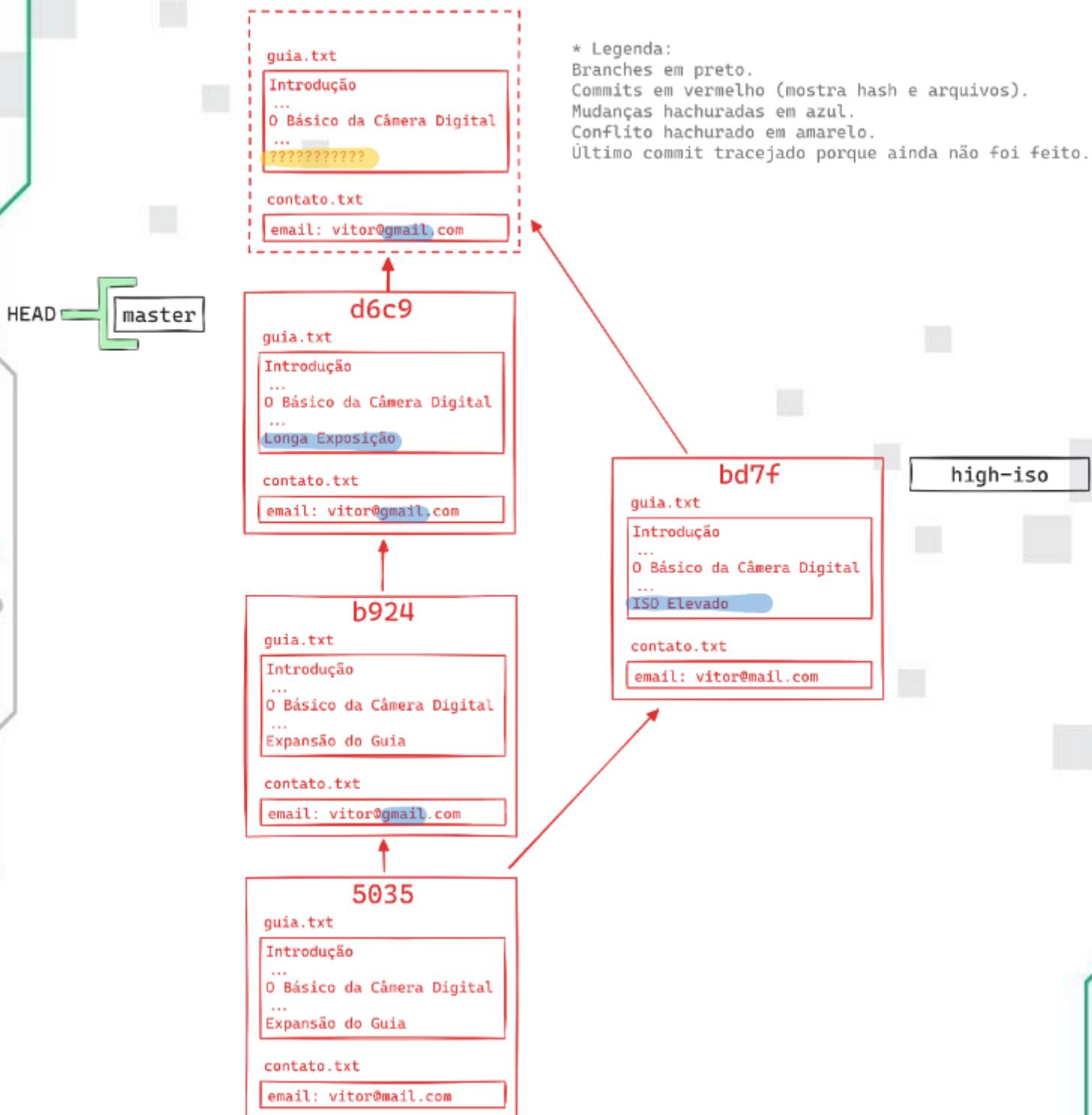
No terminal, aparece "Unmerged paths", que você nunca viu antes.

E no VSCode aparece "Merge Changes", que também você nunca viu antes. Além disso tem uma mensagem de commit pré-preenchida, "Merge branch 'high-iso'".

O que está havendo ?

Para entender, veja um diagrama que ilustra o processo de merge:

Branches e merge



Branches e merge

Ao tentar fundir os commits d6c9 e bd7f, o Git usa a mesma estratégia de preservar as mudanças.

O último commit em comum na história de d6c9 e bd7f é o 5035. Ele é o “terceiro ponto de vista”.

Prosseguindo:

No contato.txt, o Git percebe que d6c9 introduziu uma mudança em relação ao 5035, enquanto o bd7f não.

Logo o git preserva a versão com a mudança. Sem problema.

Mas ao olhar para guia.txt, o Git vê uma situação complicada:

1. d6c9, em relação a 5035, mudou a última seção para “Longa Exposição”
2. bd7f, em relação a 5035, mudou a última seção para “ISO Elevado”
3. As duas mudanças estão localizadas nas mesmas linhas do arquivo

O culpado pela falha do merge automático foi esse terceiro ponto.

Se somente houvesse os dois primeiros pontos, ou seja, ambos os commits tivessem introduzido mudanças no guia.txt, mas elas fossem *em linhas diferentes* do arquivo, o Git conseguiria manter as duas mudanças sem problema (lembre-se que já mostramos um exemplo hipotético disso).

Mas como a mudança de cada commit aconteceu nas mesmas linhas que no outro commit, o Git precisaria escolher uma delas e jogar fora a outra.

Isso seria uma péssima ideia, porque o computador não tem como saber do que se tratam as mudanças.

O Git é um programa que trabalha com textos, ele consegue comparar textos e ver se são iguais ou diferentes, mas não sabe interpretar o que está escrito ali.

Então o Git precisa que o desenvolvedor (você) olhe para os arquivos e decida como preservar as mudanças de um jeito que faça sentido para você.

A existência de mudanças no mesmo arquivo nas mesmas linhas é chamada de “conflito”.

Então o que o git status está tentando mostrar é que o guia.txt tem mudanças conflitantes e você precisa decidir.

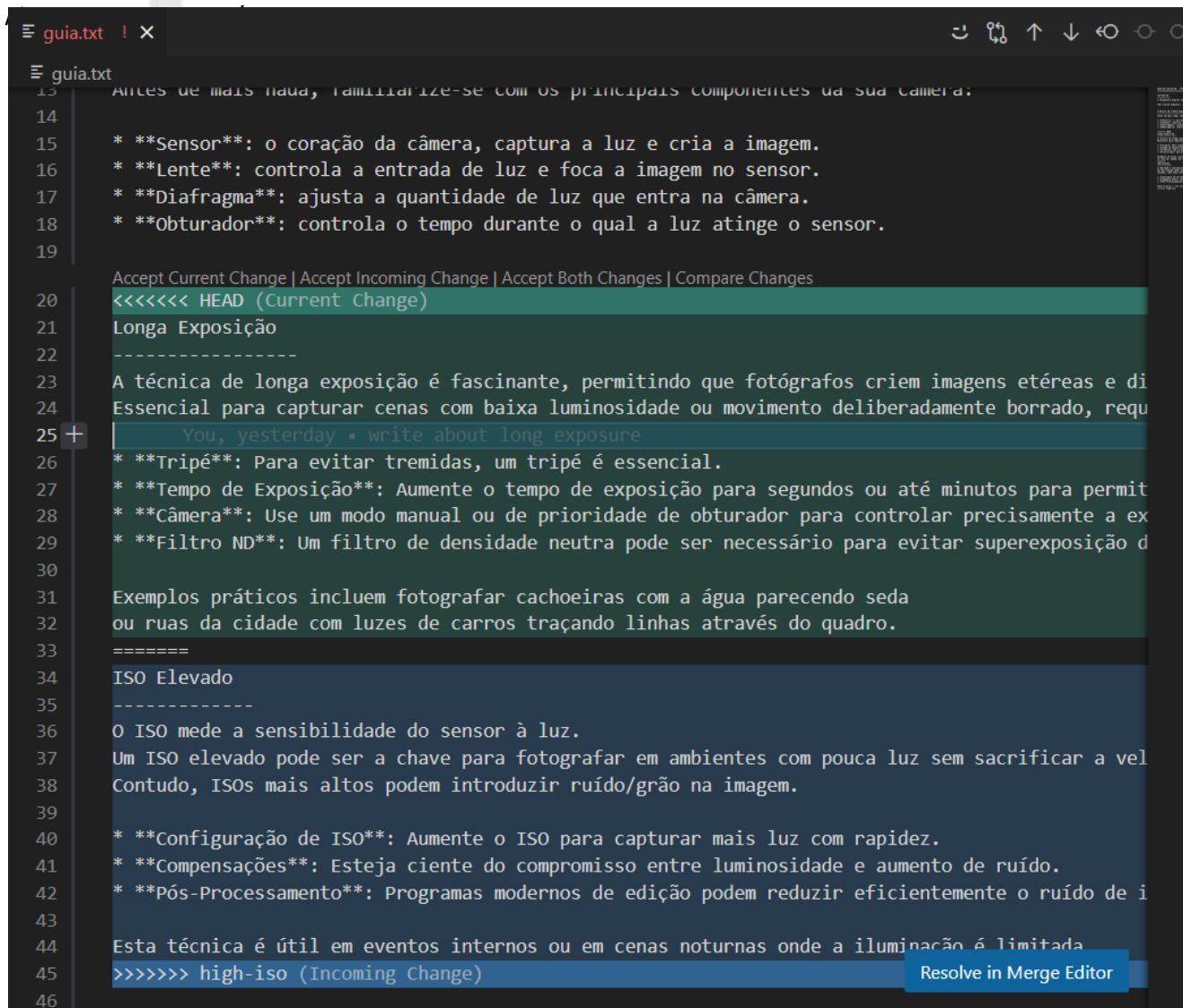


Merge fast-forward nunca tem conflito (por quê ?)

Branches e merge

Resolvendo conflitos de merge

Continuando de onde paramos: para resolver um conflito de merge, você precisa editar o arquivo no editor de texto e resolver as diferenças.



The screenshot shows a code editor with the file 'guia.txt' open. The content of the file is:

```
guia.txt ! X
guia.txt
13 Antes de mais nada, familiarize-se com os principais componentes da sua câmera:
14
15 * **Sensor**: o coração da câmera, captura a luz e cria a imagem.
16 * **Lente**: controla a entrada de luz e foca a imagem no sensor.
17 * **Diafragma**: ajusta a quantidade de luz que entra na câmera.
18 * **Obturador**: controla o tempo durante o qual a luz atinge o sensor.
19
20 Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
21 <<<<< HEAD (Current Change)
22 Longa Exposição
23 -----
24 A técnica de longa exposição é fascinante, permitindo que fotógrafos criem imagens etéreas e di
25 +-----+ Essencial para capturar cenas com baixa luminosidade ou movimento deliberadamente borrado, requ
26 You, yesterday • write about long exposure
27 * **Tripé**: Para evitar tremidas, um tripé é essencial.
28 * **Tempo de Exposição**: Aumente o tempo de exposição para segundos ou até minutos para permit
29 * **Câmera**: Use um modo manual ou de prioridade de obturador para controlar precisamente a ex
30 * **Filtro ND**: Um filtro de densidade neutra pode ser necessário para evitar superexposição d
31 Exemplos práticos incluem fotografar cachoeiras com a água parecendo seda
32 ou ruas da cidade com luzes de carros traçando linhas através do quadro.
33 =====
34 ISO Elevado
35 -----
36 O ISO mede a sensibilidade do sensor à luz.
37 Um ISO elevado pode ser a chave para fotografar em ambientes com pouca luz sem sacrificar a vel
38 Contudo, ISOs mais altos podem introduzir ruído/grão na imagem.
39
40 * **Configuração de ISO**: Aumente o ISO para capturar mais luz com rapidez.
41 * **Compensações**: Esteja ciente do compromisso entre luminosidade e aumento de ruído.
42 * **Pós-Processamento**: Programas modernos de edição podem reduzir eficientemente o ruído de i
43
44 Esta técnica é útil em eventos internos ou em cenas noturnas onde a iluminação é limitada
45 >>>>> high-iso (Incoming Change) Resolve in Merge Editor
46
```

The editor highlights the conflict region between lines 21 and 25 with a green background. The text 'Longa Exposição' is on line 22. Lines 25 through 45 are from a different commit ('high-iso'). The 'Resolve in Merge Editor' button is visible at the bottom right of the conflict area.

A parte superior do arquivo, antes das seções conflitantes, está normal.

Quando chegamos nas linhas do conflito (linha 20 em diante), vemos que o Git inseriu por conta própria alguns textos delimitadores:

1. "<<<<< HEAD" na linha 20
2. "=====+" na linha 33
3. ">>>>> high-iso" na linha 45

Essa é a maneira do Git te mostrar as duas versões, para você editar como quiser.

Entre o delimitador 1 e o 2, está o conteúdo da HEAD (ou seja, do commit d6c9).

Entre o delimitador 2 e o 3, está o conteúdo da branch high-iso (ou seja, do commit bd7f).

Você precisa apagar os delimitadores (o Git inseriu esses textos só para organizar para você), decidir qual versão você quer manter, e depois dar `git add` no arquivo.

No nosso caso, você quer manter as duas seções, uma embaixo da outra, então edite o texto para ficar assim:

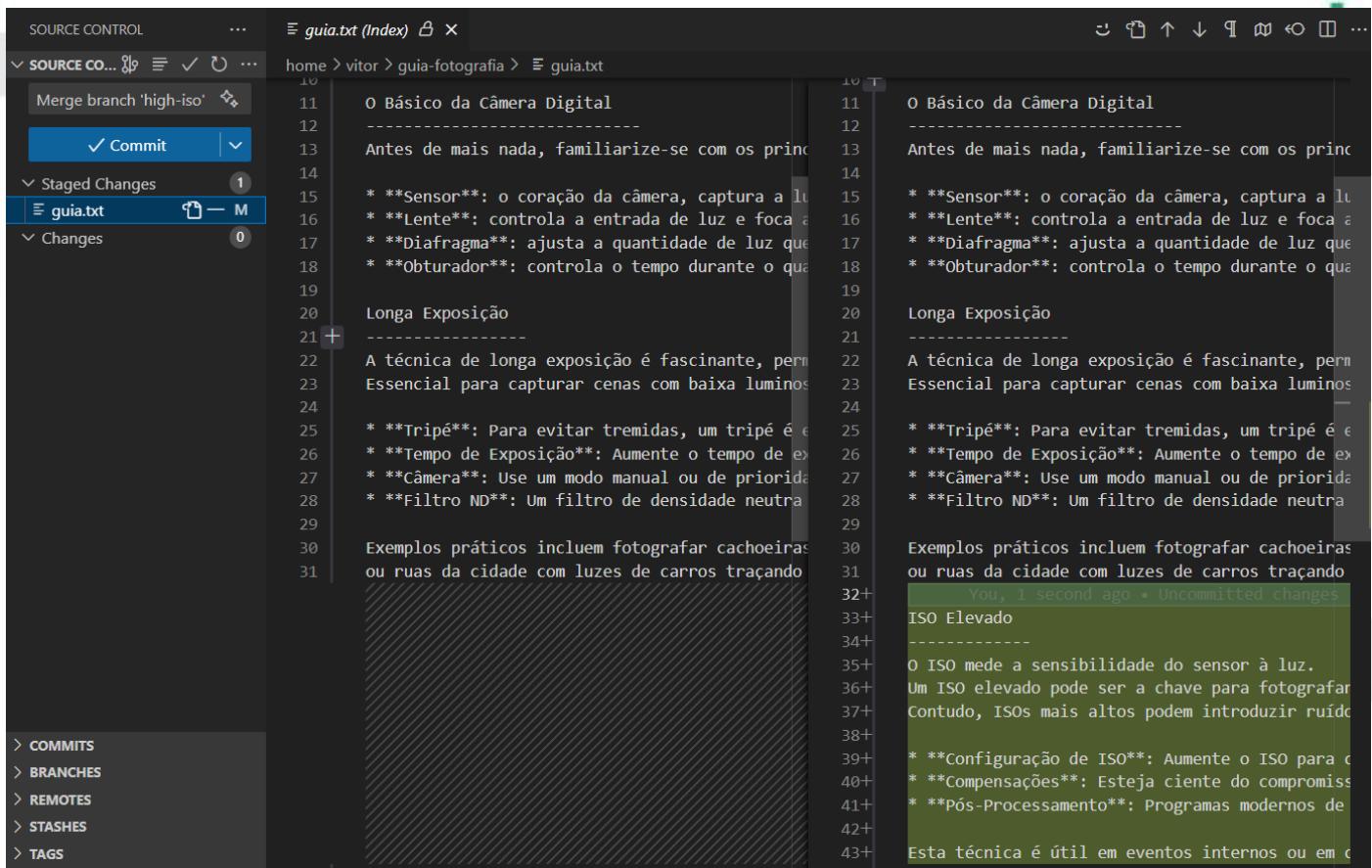
Branches e merge

```
guia.txt ! X
guia.txt
15 * **Sensor**: o coração da câmera, captura a luz e cria a imagem.
16 * **Lente**: controla a entrada de luz e foca a imagem no sensor.
17 * **Diafragma**: ajusta a quantidade de luz que entra na câmera.
18 * **Obturador**: controla o tempo durante o qual a luz atinge o sensor.
19
20 Longa Exposição
21 -----
22 A técnica de longa exposição é fascinante, permitindo que fotógrafos criem imagens etéreas e
23 Essencial para capturar cenas com baixa luminosidade ou movimento deliberadamente borrado, re
24
25 * **Tripé**: Para evitar tremidas, um tripé é essencial.
26 * **Tempo de Exposição**: Aumente o tempo de exposição para segundos ou até minutos para perm
27 * **Câmera**: Use um modo manual ou de prioridade de obturador para controlar precisamente a
28 * **Filtro ND**: Um filtro de densidade neutra pode ser necessário para evitar superexposição
29
30 Exemplos práticos incluem fotografar cachoeiras com a água parecendo seda
31 ou ruas da cidade com luzes de carros traçando linhas através do quadro.
32
33 ISO Elevado
34 -----
35 O ISO mede a sensibilidade do sensor à luz.
36 Um ISO elevado pode ser a chave para fotografar em ambientes com pouca luz sem sacrificar a v
37 Contudo, ISOs mais altos podem introduzir ruído/grão na imagem.
38
39 * **Configuração de ISO**: Aumente o ISO para capturar mais luz com rapidez.
40 * **Compensações**: Esteja ciente do compromisso entre luminosidade e aumento de ruído.
41 * **Pós-Processamento**: Programas modernos de edição podem reduzir eficientemente o ruído de
42
43 Esta técnica é útil em eventos internos ou em cenas noturnas onde a iluminação é limitada.
44
```

Salve o arquivo e dê `git add` nele (pelo terminal ou pelo VSCode mesmo).

Feito isso, o VSCode vai mostrar o arquivo na seção “Staged Changes” e vai mostrar o diff entre como ele está agora versus como ele estava no último commit da master:

Branches e merge



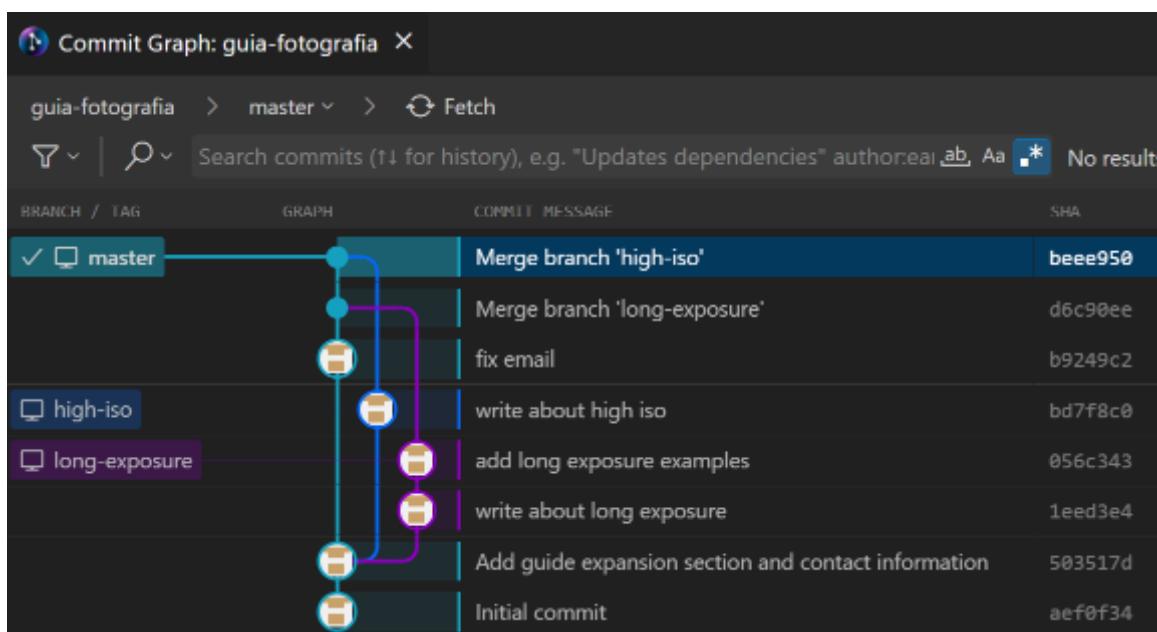
The screenshot shows a VSCode interface with a dark theme. On the left, the Source Control sidebar indicates a merge from 'high-iso' into 'master'. A 'Merge branch 'high-iso'' dialog is open, with a blue 'Commit' button highlighted. The main editor window displays the 'guia.txt' file, which contains text about camera basics and long exposure. The right sidebar shows the commit history for the 'master' branch, listing commits related to sensor, lens, and exposure settings, along with a commit for fixing the email.

Note que o diff é entre o arquivo atual e a versão dele na master (não na high-iso), porque nós (HEAD) estamos na master esse tempo todo desde o git merge.

Agora você pode fazer o commit. Clique no botão azul de "Commit" do VSCode ou faça git commit pelo terminal mesmo.

Com isso, é concretizado o commit de merge (que era o commit tracejado no nosso último diagrama de merge).

E o histórico fica assim:



Branches e merge

Agora que a master mudou de commit, seu programinha de publicação automática envia imediatamente a nova versão para o blog.

Os leitores do blog veem as duas seções Longa Exposição e ISO Elevado no final do guia, como desejado.



Ao abrir o editor de texto para resolver os conflitos de merge, você pode editar como quiser:

- Pode manter somente um dos textos
- Pode manter os dois
- Pode apagar os dois
- Pode manter partes de cada um

O ponto é que são simplesmente textos, você edita como qualquer outro texto com total liberdade.



Se você executou `git merge`, o Git acusou conflitos, mas você não quer resolvê-los agora, execute `git commit --abort` no terminal.

Isso vai voltar ao ponto anterior ao `git merge`.

Ou seja, como se você não tivesse executado esse comando.

Branches e merge

Deletando branches

Opcionalmente, como a branch long-exposure (e high-iso) já foi mergeada na master, não tem necessidade de manter o marcador.

Então você pode excluir o marcador long-exposure com o comando de terminal `git branch -d long-exposure`.

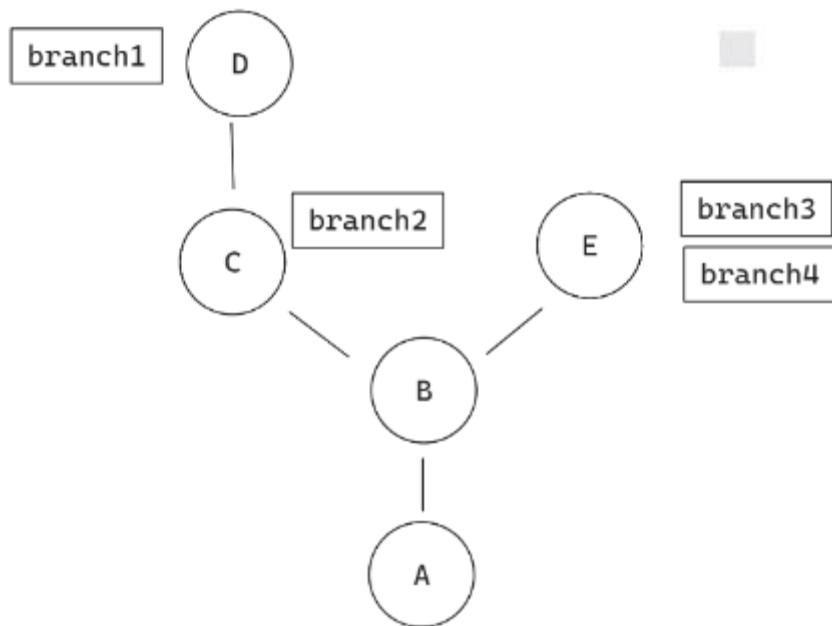
Ou, pelo VSCode, clique com o botão direito do mouse em long-exposure e escolha a opção "Delete Branch"/"Deletar Branch".

Não faça isso se você (HEAD) estiver em long-exposure !

No nosso caso, a HEAD está em master, então não tem problema.

Outro cenário em que não é bom você deletar o marcador é: quando ele está sobre um commit onde não há outro marcador, e esse commit está numa "ponta" do histórico de commits.

Por exemplo, suponha que você tenha o seguinte histórico de commits hipotético:



- Se você deletar a branch1 (somente), o commit D vai sumir, porque ele está na ponta e não tem outra branch nele
- Se você deletar a branch2 (somente), nenhum commit vai sumir, porque o commit C não está na ponta (tem o D depois dele)
- Se você deletar a branch3 (somente), o commit E não vai sumir, porque resta outra branch nele

Pense nos commits como folhas de papel sendo seguradas no lugar por pesos (as branches).

Um commit de ponta é uma folha de papel em cima das outras. Se você retirar o peso dela, o vento leva.

Resolução de problemas comuns

Introdução

Ao usar branches e merge no git, às vezes você vai se encontrar em situações que não se encaixam perfeitamente no roteiro que temos exemplificado até agora.

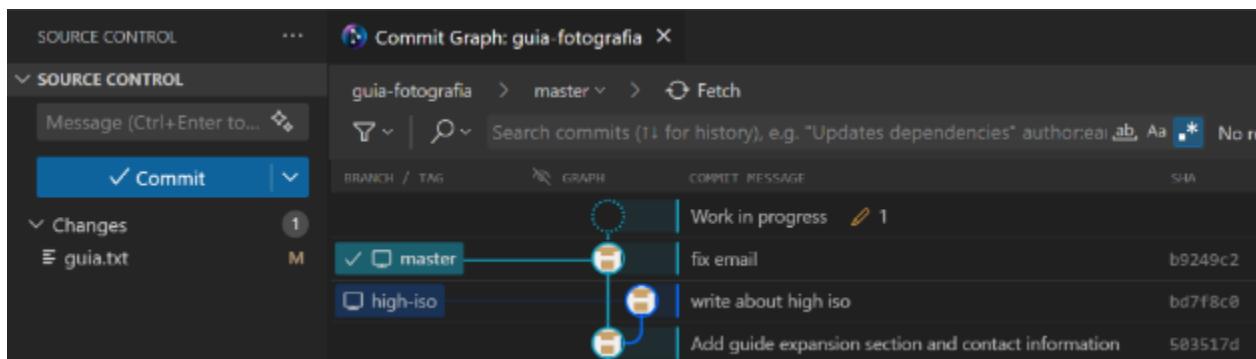
Quando isso acontecer, existem outros comandos do Git que permitem “consertar” o problema.

Nas próximas seções, apresentaremos alguns comandos que podem ser úteis nesses casos.

Não necessariamente você vai precisar imediatamente desses assuntos. Pode usar como um manual de consulta, recorrendo a esta seção quando necessário.

Git checkout falhou porque Working Tree tem modificações: **git stash**

Suponha que você está com o seguinte projeto hipotético:

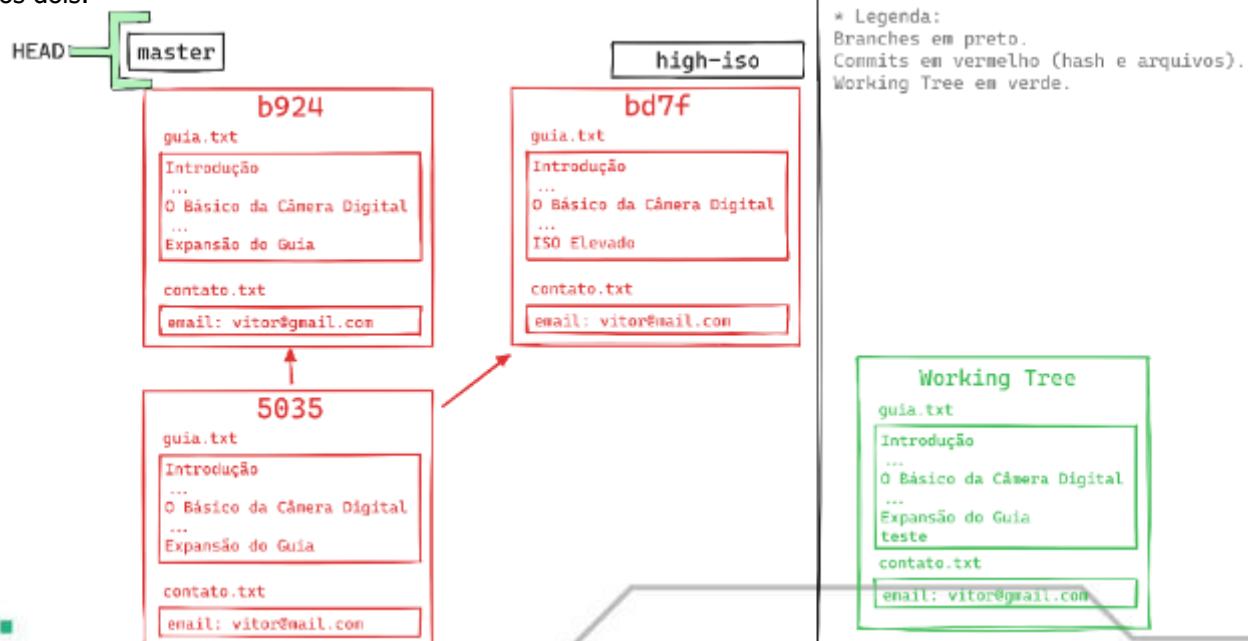


Você quer fazer `git checkout high-iso`.

Mas o comando falhou por causa das modificações que você tem na Working Tree (veja na imagem: `guia.txt` tem modificações).

O diagrama abaixo mostra a visão completa da sua situação.

Podemos ver que o checkout não está funcionando porque sua modificação na Working Tree é a palavra teste no final do `guia.txt`, e o commit `high-iso` tem outro conteúdo, então o Git não saberia conciliar os dois:



Resolução de problemas comuns

Você não quer perder suas modificações, mas mesmo assim quer fazer checkout em high-iso.

Como proceder ?

Uma opção é o comando `git stash`.

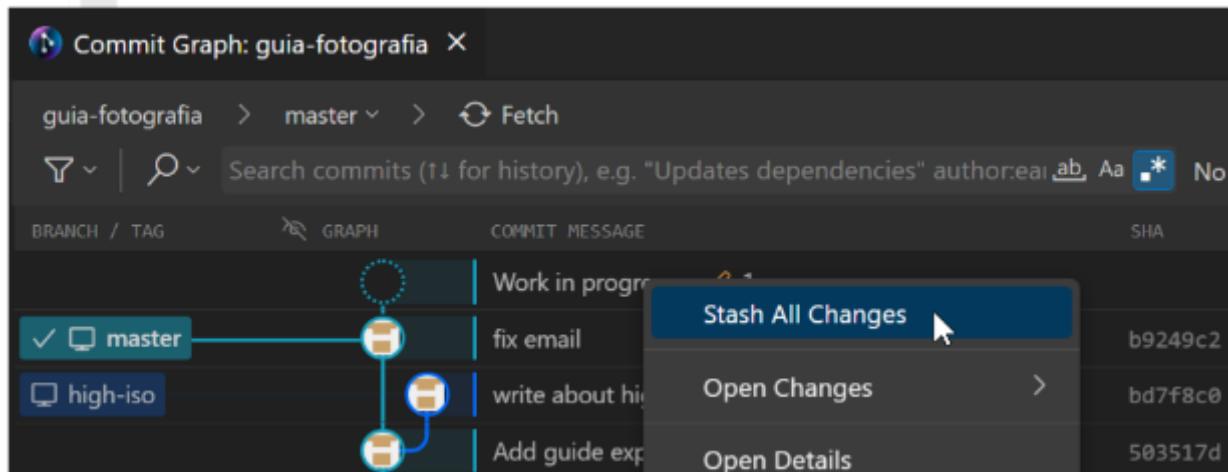
Ele salva suas modificações da Working Tree como se fossem um commit.

E então reverte a Working Tree para o conteúdo do commit onde está a HEAD.

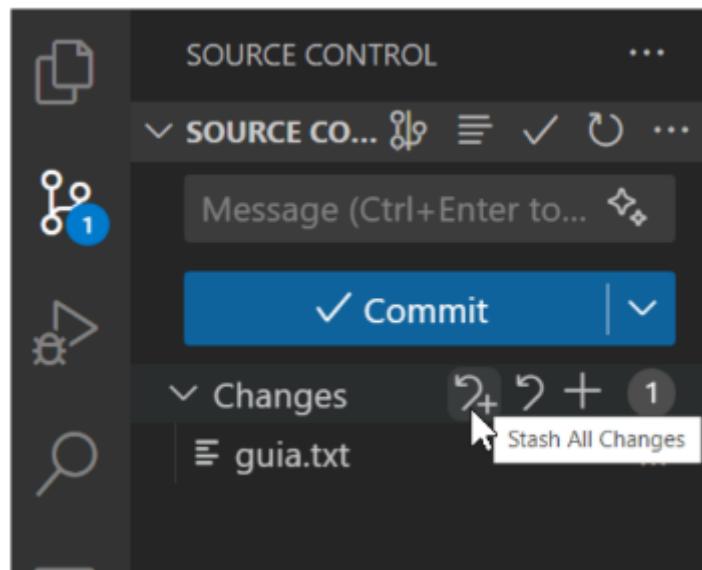
Para usá-lo, digite `git stash` no terminal.

Se preferir pelo VSCode, tem duas maneiras:

Uma é clicar na bolinha tracejada no GitLens, e então em "Stash All Changes":



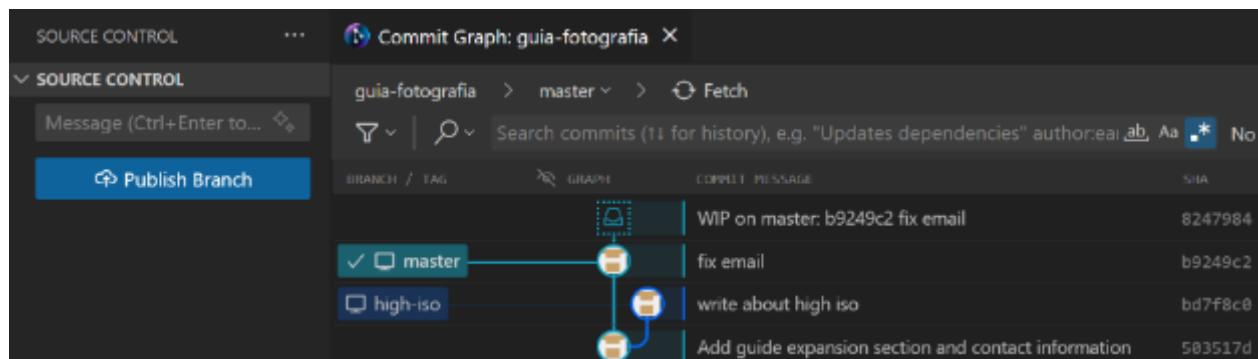
Outra é pela aba de Controle de Versão do VSCode:



Resolução de problemas comuns

O VSCode vai pedir uma mensagem, que você pode deixar vazia.

Seja como for, após o `git stash`, você ficará assim:

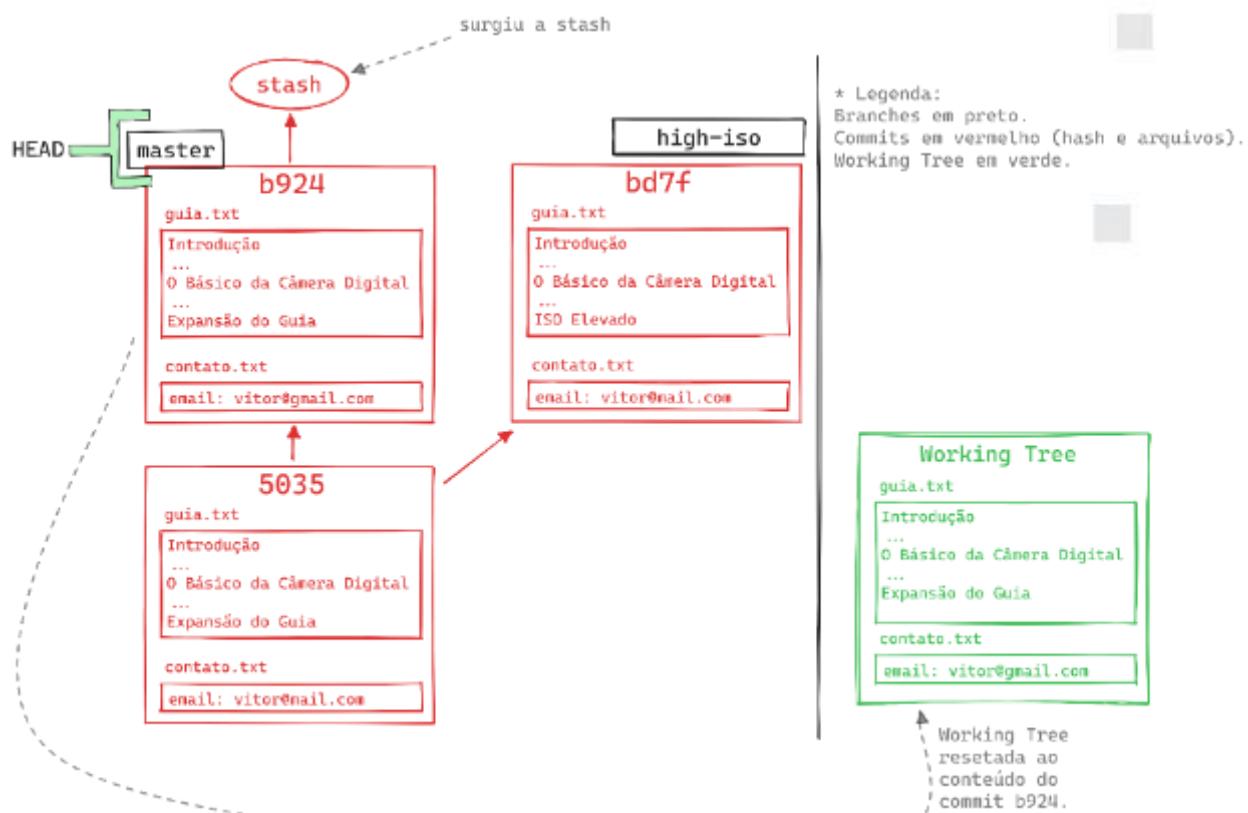


Note que a Working Tree não tem mais modificações.

A HEAD continua no mesmo lugar.

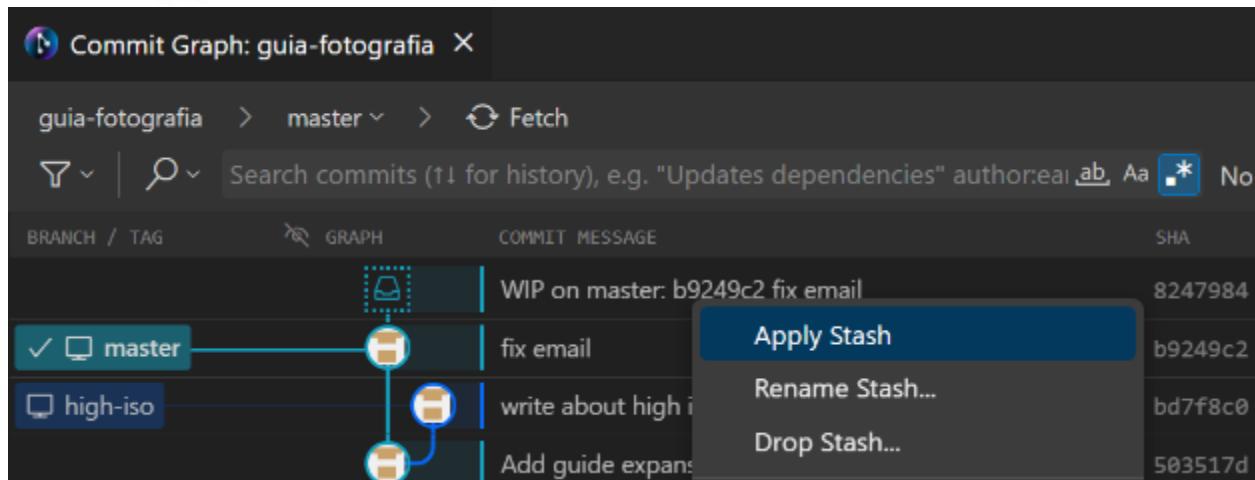
E surgiu a stash em cima do seu commit.

Olhando pelo diagrama, estamos como segue abaixo. Surgiu a stash, e a Working Tree foi resetada (você só não perdeu seu trabalho nela porque ficou salvo na stash):

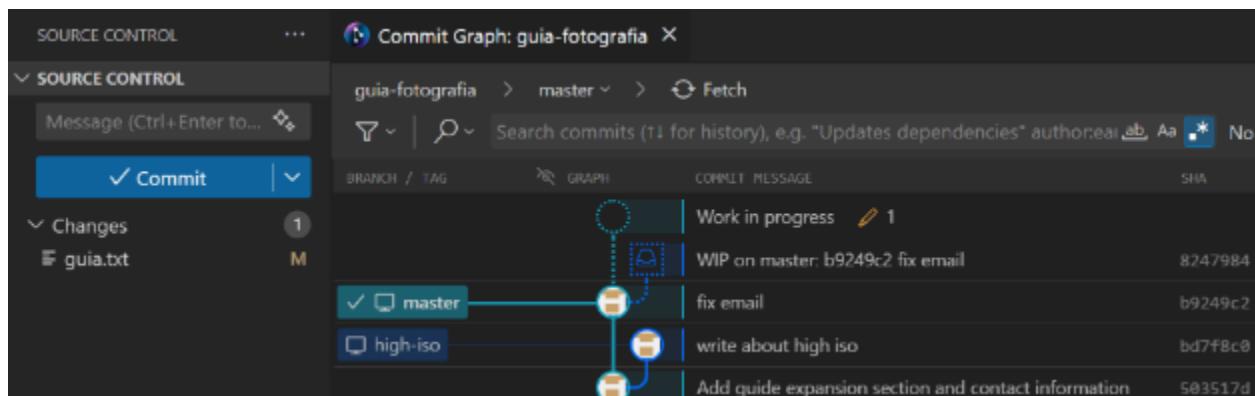


Resolução de problemas comuns

Agora que sua Working Tree está limpa, você pode fazer o `git checkout high-iso` que queria. Mais tarde, quando voltar para a `master` novamente (`git checkout master`), poderá recuperar as mudanças que ficaram salvas na stash. Para isso, basta executar `git stash apply` no terminal. Se preferir pelo VSCode, clique com o botão direito do mouse na stash e então em "Apply Stash":



O Git vai pegar as mudanças que estão salvas na stash e aplicá-las de volta na Working Tree. Você vai ficar assim:



Note que o `git stash apply` não apaga a stash.

Para apagá-la, digite `git stash drop`.

Ou, pelo VSCode, clique com o botão direito do mouse na stash e então em "Drop Stash".

Resolução de problemas comuns



- Para fazer ao mesmo tempo a aplicação e deleção da stash, o comando é `git stash pop`.
- É possível ter mais de uma stash salva. Leia o manual do Git para saber mais:
 - <https://git-scm.com/docs/git-stash>
- Note que `git stash` é útil também para outras situações em que você quer “se livrar” das modificações da Working Tree temporariamente sem perdê-las.
- No momento do `apply`, você (HEAD) pode estar em qualquer commit, não precisa ser o mesmo commit onde criou a stash.

Porém nem sempre o Git consegue fazer o `apply` nesse caso (não vamos entrar em detalhes).

Se ele não conseguir, nada será alterado no projeto, você só verá uma mensagem dizendo que não foi possível fazer o `apply`

Iniciei um merge, teve conflitos, não quero resolver agora: `git merge --abort`

É possível cancelar o merge que deu conflito.

O comando é `git merge --abort`.

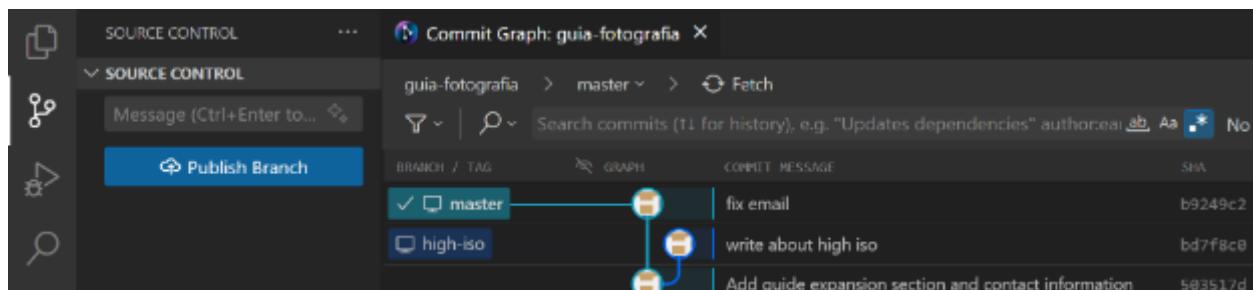
Por outro lado, no caso de um merge que não teve conflitos, o Git vai gerar um commit de merge automaticamente.

Nesse caso, `git merge --abort` não vai fazer nada.

Para uma solução, veja a próxima seção.

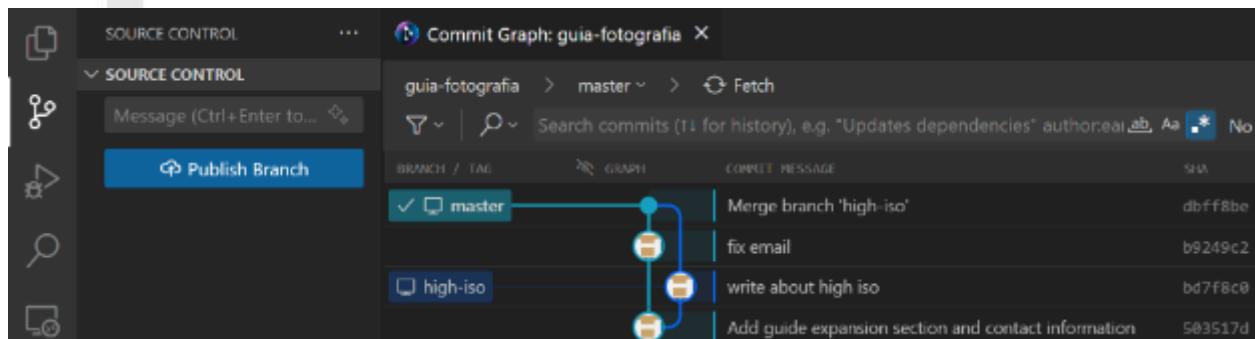
Fiz um merge, mas quero desfazer: `git reset --hard`

Suponha que você está com o seguinte projeto hipotético:



Resolução de problemas comuns

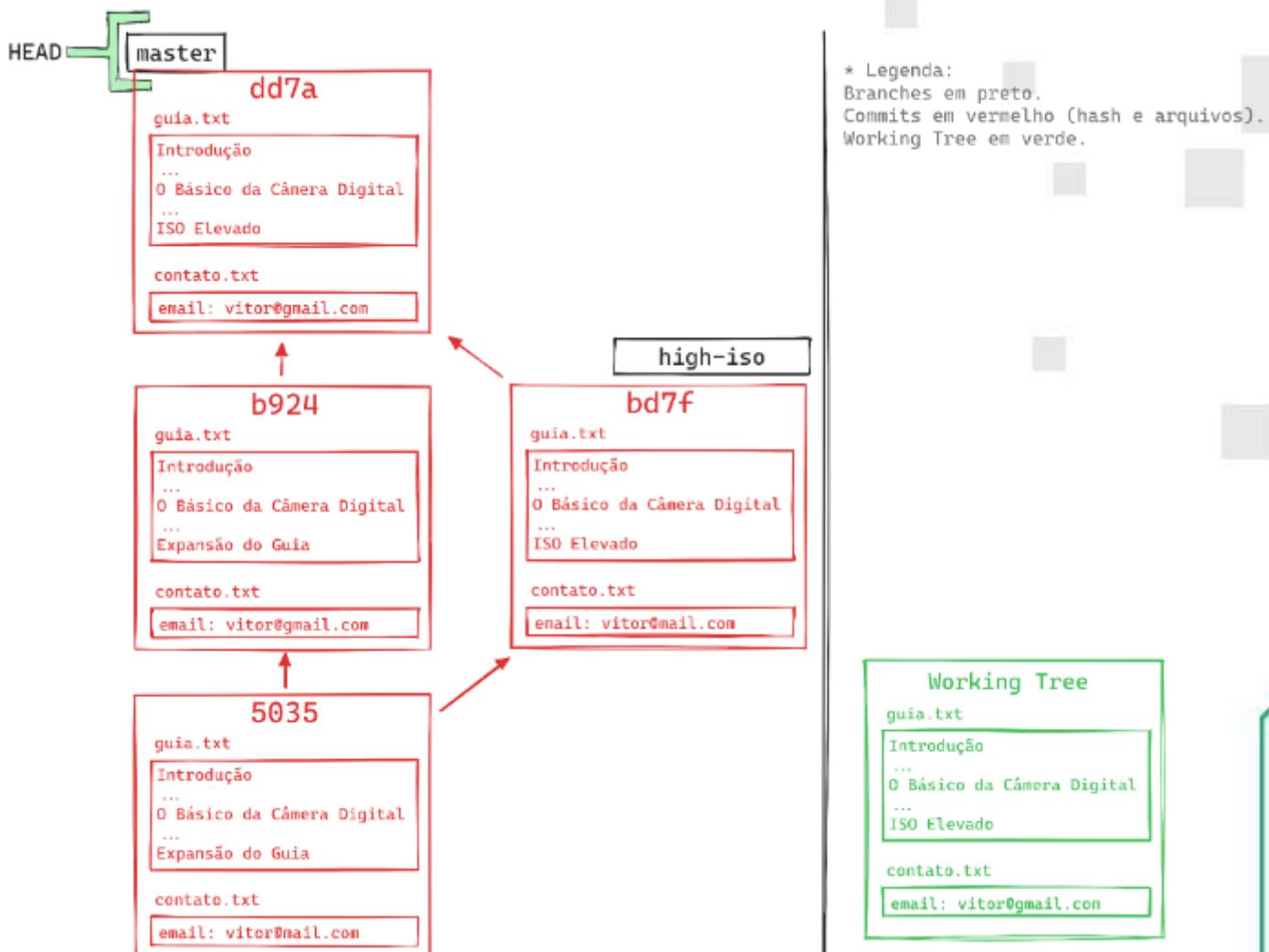
Então, sem querer, você faz `git merge high-iso`, ficando assim:



The screenshot shows the 'Commit Graph' view in VS Code for the 'guia-fotografia' repository. The 'master' branch is the current HEAD. A merge commit is shown where the 'high-iso' branch was merged into 'master'. The commit messages and SHA values are listed:

COMMIT MESSAGE	SHA
Merge branch 'high-iso'	dbff8be
fix email	b9249c2
write about high iso	bd7f8c0
Add guide expansion section and contact information	503517d

Olhando a mesma situação por um diagrama:



Resolução de problemas comuns

Como voltar para o estado anterior? Ou seja, voltar a master para o commit b924?

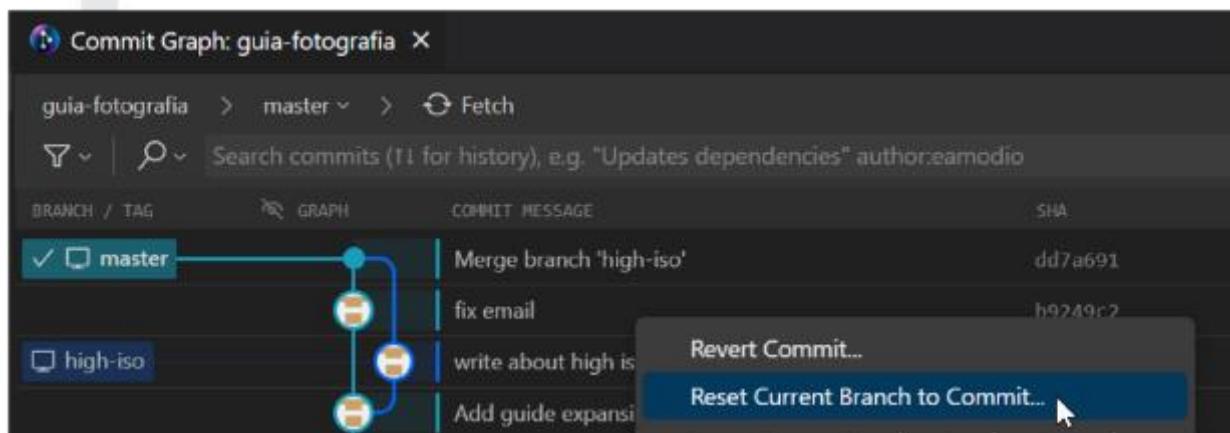
Para isso, a primeira medida é selecionar com a HEAD a branch que você quer mover.

No nosso caso, a HEAD já está na master, então não precisamos fazer git checkout master.

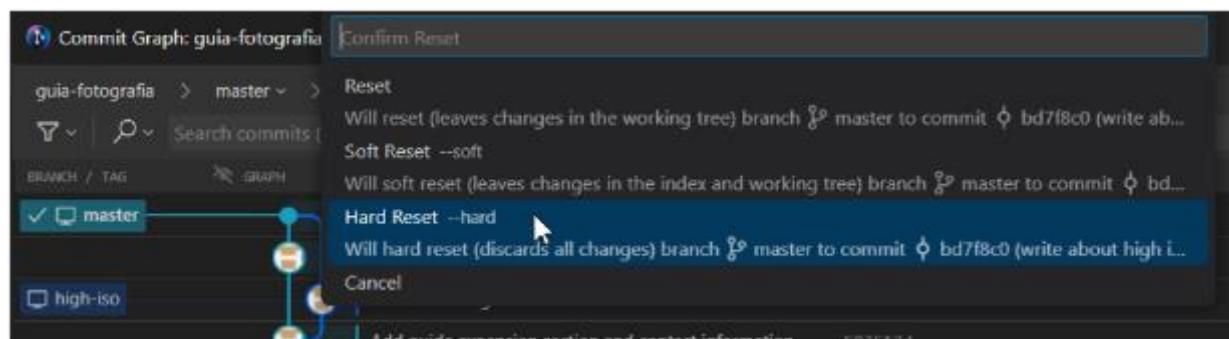
Uma vez na branch a ser movida, você pode usar o comando git reset --hard <nome do commit de destino>. No nosso caso:

- git reset --hard b924

Se preferir fazer pelo VSCode, clique com o botão direito no commit de destino (b924) e então em "Reset Current Branch to Commit":



No menu que vai aparecer, escolha a opção "Hard Reset":



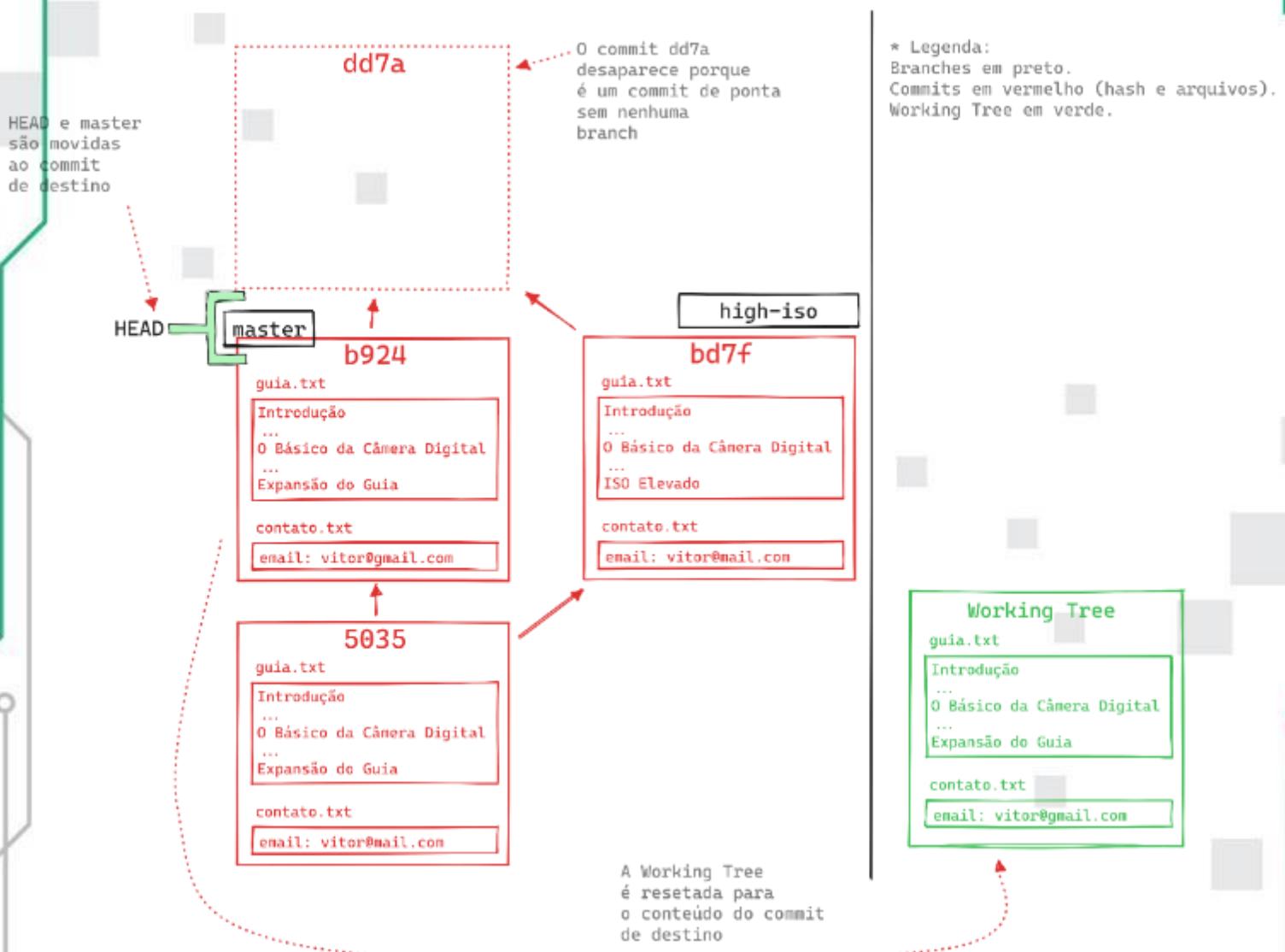
Esse comando é parecido com o checkout, porque o reset --hard também move a HEAD para o commit de destino e reverte a Working Tree para coincidir com o commit de destino.

Mas o reset --hard move também a branch que a HEAD tem selecionada. No nosso caso, a master. Enquanto o checkout não move branches.

E é isso que justamente nos interessa: nós queríamos mover a branch de volta para onde ela estava antes do merge.

Um diagrama ilustra o que o reset --hard faz:

Resolução de problemas comuns



Depois de executar o comando, o projeto voltará a ser como estava no início antes do merge, ou seja:

- A HEAD, com a `master`, estará de volta sobre `b924`
- A Working Tree estará com o conteúdo deste commit
- E o commit de merge vai desaparecer, porque (como já explicamos) ele é um commit de ponta que ficou sem nenhuma branch.

Resolução de problemas comuns



- Cuidado com o uso do `git reset --hard`, porque ele reseta o conteúdo da Working Tree.

Portanto, sua Working Tree como estava antes do reset será perdida.

Se você estava trabalhando em modificações não-commitadas (portanto não estão salvas no histórico de commits), elas serão *totalmente perdidas*.

O `git reset --hard` é um dos poucos comandos que podem te fazer perder dados.

Em contraste, como já vimos, o `checkout` não pode causar perda de dados não-commitados da Working Tree porque ele aborta caso perceba que isso aconteceria.

- Se você quer que o `reset` atue como o `checkout`, ou seja:

- preserva as mudanças não-commitadas da Working Tree
- reseta o restante da Working Tree
- aborta caso não seja possível fazer os dois

Então use `git reset --keep`

- O nome `reset` pode passar a impressão de que ele só serve para fazer uma branch retroceder a um commit mais antigo.

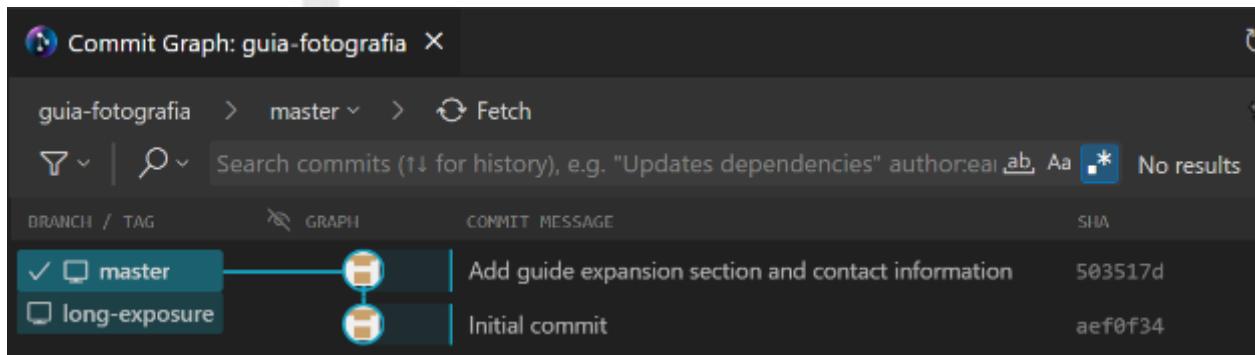
Isso não é verdade.

Apesar do nome, o `reset` serve para mover uma branch para qualquer outro commit.

Resolução de problemas comuns

Esqueci de trocar de branch antes de fazer commit: git reset --hard

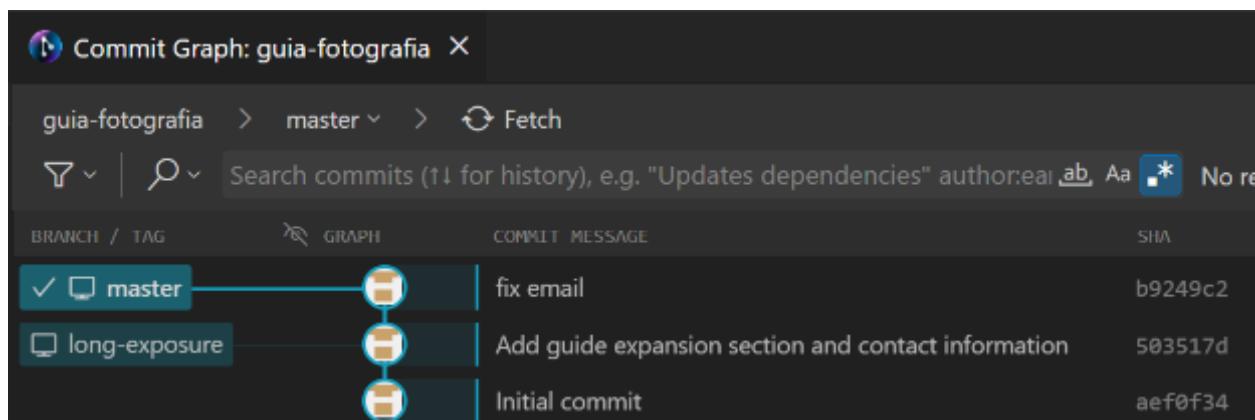
Para mostrar outro uso do `git reset --hard` (já apresentado na seção anterior), suponha que você está com o seguinte projeto hipotético:



Você já criou a branch `long-exposure` e sua intenção é fazer o próximo commit nela.

Você dá `git add` nas suas mudanças e `git commit`, sem perceber que a HEAD estava em `master` em vez de `long-exposure` (veja de novo a figura).

Então, após o commit, a `master` avança e a `long-exposure` fica parada (você queria o contrário):



Como corrigir isso ?

Ou seja, como colocar a `master` no commit 5035 e a `long-exposure` no b924 ?

Na verdade você já tem as ferramentas para resolver isso (com um pouco de criatividade).

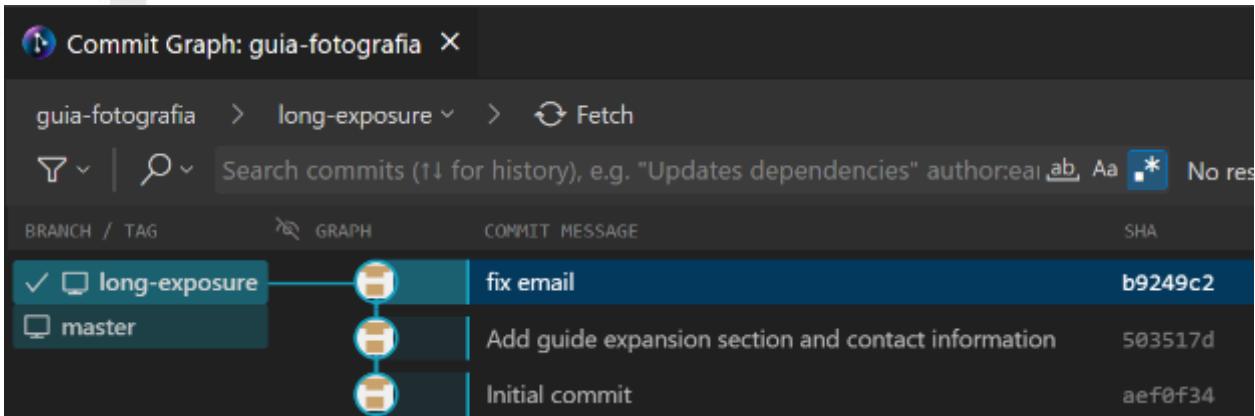
Primeiro podemos avançar a `long-exposure`. Basta fazer:

- `git checkout long-exposure`
- `git merge master`

O merge vai ser fast-forward (por quê ?)

Resolução de problemas comuns

E o histórico vai ficar assim:



BRANCH / TAG	GRAPH	COMMIT MESSAGE	SHA
<input checked="" type="checkbox"/> long-exposure		fix email	b9249c2
<input type="checkbox"/> master		Add guide expansion section and contact information	503517d
		Initial commit	aef0f34

Agora podemos voltar a master para 5035, com:

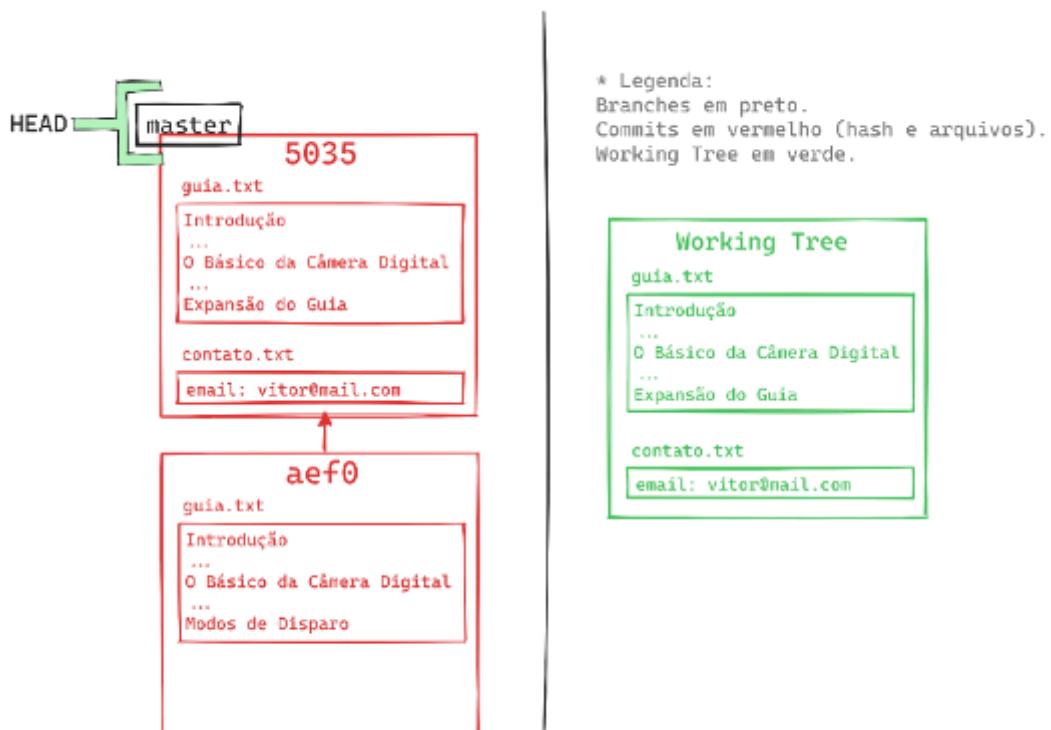
- git checkout master
- git reset --hard 5035

Se você não conhece o git reset --hard, leia a seção anterior.

Depois do segundo passo, o histórico ficará como você queria, a HEAD estará na master em 5035, e a Working Tree terá o conteúdo desse mesmo commit.

Fiz um commit, quero desfazer sem perder o conteúdo dele: **git reset --mixed**

Suponha que você acaba de fazer um commit, não fez nenhuma outra modificações depois dele, e seu projeto está conforme o seguinte diagrama:



Resolução de problemas comuns

No último commit, note que você alterou o `guia.txt` e criou o `contato.txt`.

Pensando melhor, você acha que o histórico ficaria mais claro se tivesse feito essas duas mudanças em commits separados:

- primeiro alterar o `guia.txt`
- depois criar o `contato.txt`

Mas agora o commit já está feito ! Como voltar atrás ?

Para isso, execute `git reset --mixed <hash do commit de destino>`. Nesse caso:

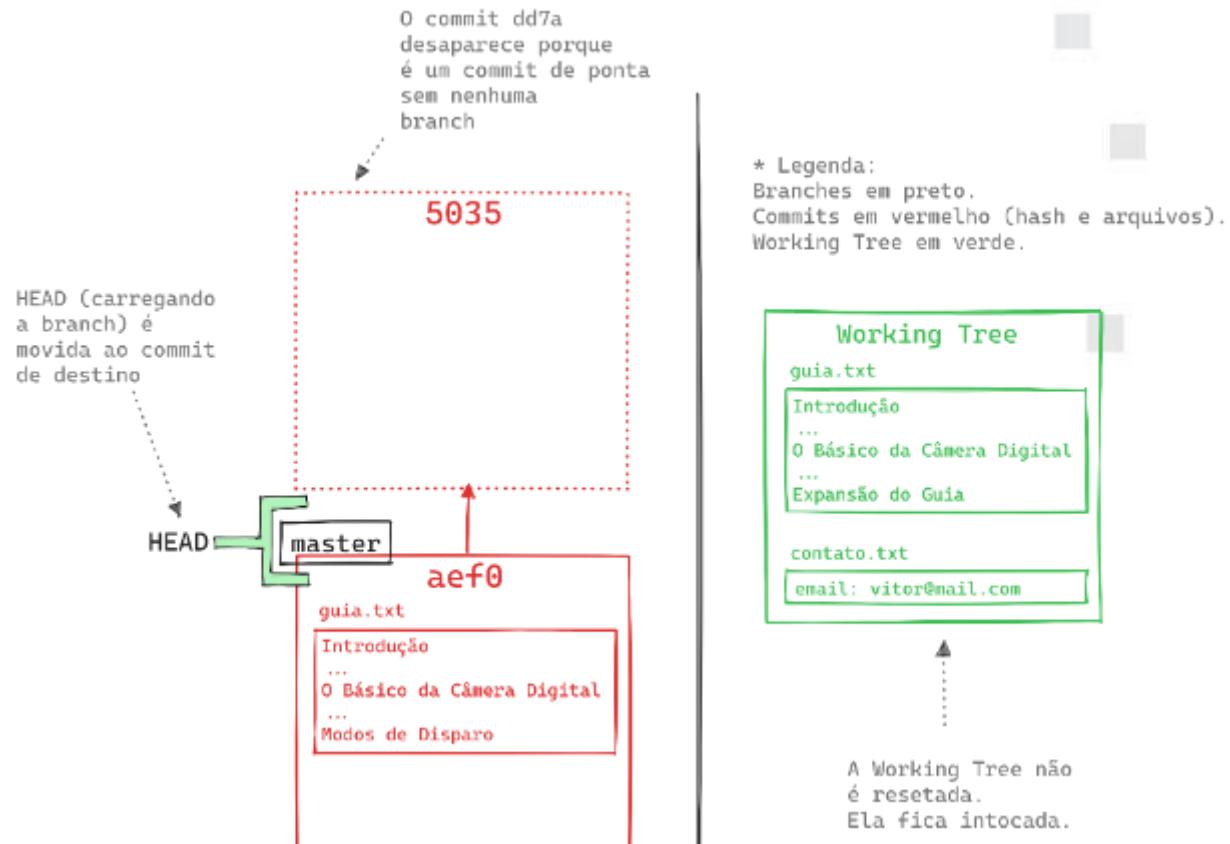
- `git reset --mixed aef0`

O `git reset --mixed` é a opção padrão do `git reset`, ou seja, se você digitar somente `git reset aef0`, o Git já vai implicitamente entender que é `--mixed`.

Se preferir fazer pelo VSCode:

- Clique no commit `aef0` com o botão direito do mouse.
- Depois clique em “Reset Current Branch to Commit”.
- Vai aparecer o menu de opções de reset, escolha a primeira, onde está escrito somente “Reset”.

Feito o reset `--mixed`, o resultado é o seguinte:



Resolução de problemas comuns

Ou seja:

- A HEAD (carregando a branch) é movida ao commit de destino.
Igual ao `git reset --hard`.
- O commit 5035 desaparece porque se torna um commit da ponta sem nenhuma branch.
- A Working Tree permanece como estava.
Isso é diferente do `git reset --hard`!
Como já vimos, a opção `--hard` reseta a Working Tree, mas a opção `--mixed` a preserva sem alterações.

Então agora você está de volta no commit `aef0`.

E o `git status` vai mostrar que existem modificações porque (como você já sabe) ele compara a Working Tree contra o commit onde está a HEAD.

O `git status` mostra:

- Existem modificações no `guia.txt`: a última seção é “Expansão do Guia” na Working Tree mas “Modos de Disparo” no commit onde está a HEAD
- Foi criado o `contato.txt`: porque ele existe na Working Tree mas não no commit onde está a HEAD

Agora é só proceder como você quer.

Dê `git add guia.txt` e faça commit.

Depois dê `git add contato.txt` e faça outro commit.

Com isso o histórico ficará como desejado:

- O primeiro commit `aef0` conterá o `guia.txt` com última seção “Modos de Disparo” (não mexemos nesse commit)
- O segundo commit conterá o `guia.txt` com última seção “Expansão do Guia”. Não terá o `contato.txt`.
- O terceiro commit terá a mesma versão do `guia.txt`, e terá o `contato.txt`.



O `git reset` tem várias opções:

- `--hard`
- `--mixed`
- `--keep`
- `--soft`
- `--merge`

Tratamos aqui sobre as 3 primeiras.

Além disso, existe a forma `git reset HEAD <arquivo>`, que já vimos em aulas passadas.

Como vimos lá, essa forma não tem o propósito de mover branches, mas sim é o “antônimo” do `git add`.

E essa forma não admite as opções `--hard` e as outras.

Consulte o manual do Git para saber mais sobre todas as utilidades e opções do `git reset`.