GIT

Quando iniciamos um novo projeto, uma das primeiras perguntas que surge é: *onde os arquivos desse projeto ficarão armazenados ?* Principalmente quando trabalhamos em equipe com outros desenvolvedores, faz-se necessário que o projeto esteja armazenado em um local que seja acessível a todos os membros da equipe, que permita ainda a atribuição de permissões para que os membros dessa equipe possam alterar os arquivos existentes, ou ainda permitir que novos arquivos sejam criados e que arquivos existentes possam ser excluídos.

Tendo em vista esse cenário, podemos definir que qualquer computador, numa rede onde todos os membros da equipe tenham acesso, é candidato a "local do projeto". Mas, e se ocorrer algum problema com esse computador, como por exemplo, alguma falha em seu hardware?

Então podemos pensar em armazenar os arquivos do projeto em um servidor, com redundância de dispositivos de armazenamento e backups constantes. Uma pasta compartilhada na rede em uma máquina mais segura do que uma estação de trabalho torna-se uma opção melhor. Agora temos um local propício para que o projeto seja armazenado com segurança e acesso para toda a equipe, inclusive com backups constantes que permitem que recuperemos arquivos excluídos acidentalmente. Podemos chamar esse local de **repositório de código**.

Apesar dessa segurança adicional, ficamos um pouco restritos quanto a recuperar arquivos excluídos, não temos como desfazer alterações no nosso projeto seletivamente e também não temos informações suficientes sobre o projeto para saber exatamente *qual versão desse arquivo eu devo recuperar do backup* Podemos criar um histórico (*log*) textual e adicioná-lo como parte do projeto. Agora perceba que ficamos com um ambiente com requisitos demais para um simples "local de armazenamento de projeto".

Pensando nesse problema foram criadas ferramentas de **controle de versão** e esse curso apresentará uma das mais utilizadas hoje em dia, o **Git**. O Git é utilizado em muitos projetos de código aberto no mundo todo. Um dos maiores e mais importantes é o kernel do Linux. É possível, porém, utilizá-lo para projetos privados com todas as restrições e permissões de acesso necessárias.

Uma grande vantagem do Git é a existência de ferramentas comerciais de hospedagem de código na Web. A maior provedora de hospedagem de repositórios Git, abertos ou privados, hoje é o [Github] (<http://www.github.com/>). Ele permite criar gratuitamente repositórios abertos.

Para que seja possível utilizar o Git teremos que instalá-lo. O Git é uma ferramenta baseada em linha de comando, ou seja, realizamos as operações de controle dos arquivos pelo prompt de comando. Se você é adepto de ferramentas gráficas não se preocupe, existem algumas opções para facilitar seu trabalho e vamos conhecê-las no decorrer do curso.

Iniciaremos o curso utilizando o Git no prompt de comando, para que seja possível demonstrar suas funcionalidades que, de certa maneira, são mascaradas em botões e opções de menus nas ferramentas gráficas. O uso do Git

Ciclo Git

O Git é uma ferramenta de controle de versão baseada no sistema de arquivos, ou seja, podemos fazer a associação de uma pasta diretamente a um repositório. Então, vamos criar um novo diretório que conterá os arquivos do nosso projeto:

mkdir curso-git

cd curso-git

Mas, como essa é uma pasta como qualquer outra em nosso computador, será que o git já sabe que ela conterá os arquivos do nosso projeto? Como indicar que essa pasta será o nosso futuro repositório? Isso é feito a partir do seguinte comando:

git init

Será exibida uma mensagem similar a: Initialized empty Git repository in /diretorio/repositorio/do/git E pronto. Já temos um repositório.

# **Adicionando arquivos ao repositório**

Agora começaremos o nosso projeto em si. Faremos um projeto que conterá páginas HTML. Então, vamos criar o primeiro arquivo, uma simples página HTML vazia chamado index.html:

<html>

<head>

</head>

<body>

</body>

</html>

Já temos o nosso primeiro arquivo em nosso projeto. Mas, será que o git já sabe que o arquivo criado pertence ao repositório? Para tal, podemos verificar quais os arquivos que pertencem ao nosso repositório. Isso pode ser feito digitando o comando:

git ls-files

E aí? Retornou algo? Nada? Onde está o nosso arquivo? Como fazemos para adicioná-lo no repositório? Para confirmar, podemos verificar o estado dos arquivos do nosso projeto:

git status

E teremos como saída:

# On branch master

#

# Initial commit

#

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# index.html

Com isso, podemos verificar que o nosso arquivo está na condição de Untracked files, isto é, ele não está na lista de arquivos cujas alterações serão rastreadas, ou controladas. Isso acontece porque o Git não sabe que deve controlar as alterações deste arquivo, ou seja, que ele deva fazer o track. Então, como dizemos ao Git que ele deve realizar o track? Isso é feito a partir do comando git add passando o nome do arquivo do qual o Git deve fazer o track. No nosso caso, queremos adicionar o arquivo index.html ao repositório:

git add index.html

Vamos verificar novamente o estado dos arquivos do nosso repositório com o comando git status:

# On branch master

#

# Initial commit

#

# Changes to be committed:

# (use "git rm --cached <file>..." to unstage)

#

# new file: index.html

#

Vemos que o nosso arquivo passou para condição de Changes to be committed, isto é, na lista de arquivos que estão prontos para o commit. Mas o que é commit? Toda vez que terminamos de realizar as alterações nos arquivos de um projeto, precisamos "entregar" essas alterações, isto é, realizar um commit. E somente as alterações que estiverem sob a condição de "Changes to be commited" é que serão entregues.

Agora, podemos realizar o primeiro commit do nosso projeto. Mas como é que o Git sabe quem é o responsável pelo commit? Na seção anterior, vimos o comando git blame, o qual mostrava os responsáveis por cada linha de código. Nós precisamos informar ao Git o nosso nome e e-mail. Isso é feito com os seguintes comandos:

git config user.name "João Carlos Fonseca"

git config user.email "jcfonsecagit@gmail.com"

Com isso, o usuário João Carlos Fonseca será o responsável pelas alterações no repositório atual. Porém, se quisermos fazer isso para outros repositórios, temos que dar o mesmo comando toda vez? Felizmente, o Git nos fornece a opção de definir um nome de usuário e e-mail para todo o sistema, isto é, deixando esta configuração global:

git config --global user.name "João Carlos Fonseca"

git config --global user.email "jcfonsecagit@gmail.com"

Caso essa configuração não seja efetuada, o Git vai determinar o nome de usuário atual do terminal como o autor das alterações. É importante que essa configuração seja feita para que seja mais fácil encontrar suas próprias alterações e facilitar a comunicação entre os membros do time caso haja alguma dúvida sobre um código feito por outra pessoa.

Agora sim, estamos prontos para executar o nosso primeiro commit do projeto. Para que isso aconteça, devemos executar:

git commit -m "Início do projeto"

Com isso, realizamos o primeiro commit do sistema. A flag "-m", indica que o conteúdo a seguir é a mensagem que será utilizada para descrever o que está sendo feito no commit. Verificando os estados dos arquivos do nosso sistema com o comando git status, verificamos que não há nenhuma alteração em nosso projeto.

# On branch master

nothing to commit (working directory clean)

# **Alterando o projeto**

Vamos continuar o nosso projeto modificando o arquivo index.html:

<html>

<head>

</head>

<body>

<h1>Git</h1>

<h2>Trabalhando em Equipe com Controle e Segurança</h2>

<p>Um curso que explora os benefícios de utilizar o Git como ferramenta de controle de versão para projetos em qualquer linguagem, em qualquer plataforma.</p>

<h3>Principais benefícios:</h3>

<ul>

<li>Funciona de maneira distribuída</li>

<li>Permite a edição concorrente de arquivos do projeto</li>

<li>Não depende de uma conexão ativa com um servidor</li>

</ul>

</body>

</html>

Verificando o estado dos arquivos novamente com o comando git status, percebemos que o Git já reconhece que temos arquivos que foram modificados no nosso projeto desde o nosso último commit:

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: index.html

#

no changes added to commit (use "git add" and/or "git commit -a")

Contudo, vemos que essas alterações ainda não fazem parte das alterações que serão adicionadas no próximo commit. Para adicionar esses arquivos para o próximo commit, precisamos rodar novamente o comando git add com o nome do arquivo.

Verificando o estado novamente, vemos que o nosso arquivo index.html está sob a condição "Changes to be committed".

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

Agora as alterações estão prontas para o commit. Ao rodar o comando git commit -m "Conteúdo da página index.html", temos o segundo commit do nosso projeto.

# **Adicionando múltiplos arquivos**

Podemos utilizar o comando git add para adicionar mais de um arquivo ao index ao mesmo tempo.

Você pode adicionar arquivos específicos passando cada um como um parâmetro, separando cada arquivo por um espaço!

git add arquivo1 arquivo2

Você pode adicionar um diretório inteiro passando como parâmetro do comando git add o caminho do diretório!

git add caminhoDeUmDiretorio

O caminho git add . funciona do mesmo modo que passar o caminho de um diretório, pois o atalho . representa o diretório atual. Portanto, serão adicionados todos os arquivos que estiverem dentro do diretório atual!

git add .

# **Recapitulação do processo**

Durante o ciclo básico que demonstramos anteriormente, nós interagimos com 3 estágios diferentes do repositório. O primeiro deles enquanto nós criamos o repositório, mas não indicamos nenhum arquivo para ser rastreado.

Nesse estágio, estamos interagindo com um estado do projeto que chamamos de "Working Directory", ou seja, é o nosso sistema de arquivos atual. Nele estão as alterações que estamos realizando no momento.

O Working Directory pode estar "limpo", quando não há diferença entre os arquivos armazenados no repositório e como estão atualmente. Quando há diferença (por exemplo, alteramos determinado arquivo, mesmo que uma alteração mínima), o Working Directory fica marcado como "sujo".

Em nosso caso o arquivo não existia para se ter uma comparação, pois tínhamos um repositório novo. Após modificarmos nossos arquivos a ponto de definirmos que um "passo" foi concluído, criamos uma visão desse passo, um ponto de controle preliminar com o comando "git add".

Esse comando cria um novo estágio do repositório, o que chamamos de "Index" ou "Staging Area". Esse estágio é transitório e pode ser alterado ainda antes de se tornar um passo do projeto: podemos adicionar novos arquivos, removê-los ou mesmo alterá-los.

Quando satisfeitos com o conteúdo do "Index", utilizamos o comando "git commit" para persistí-lo, gravar esse passo com todos os arquivos novos e alterações efetuadas. O comando "commit" criou um terceiro estágio do repositório que conhecemos como "HEAD". O HEAD é o último estado que o Git usa como referência, é a visão do último passo do projeto que foi concluído e entregue.

# **Conclusão**

Uma ferramenta como o Git nos permite fazer alterações em nosso projeto com mais segurança pois sabemos que podemos controlar cada alteração, às vezes de maneira bem detalhada, de cada um dos nossos arquivos.

Apesar de poderosa, o Git não é uma ferramenta totalmente automatizada (automatizar esse processo seria impossível, pois o gerenciamento das versões varia muito de acordo com o ambiente do projeto e da equipe). Precisamos interagir com o Git constantemente para podermos extrair o melhor dos benefícios que ele oferece.

Apesar de demonstrarmos o uso do Git desde o início do projeto, podemos utilizar os mesmos passos em um projeto existente para que ele possa, a partir de um momento, ser controlado pelo Git.

Esse capítulo demonstrou o ciclo básico do trabalho de controle de versões de um projeto do ponto de vista de um único desenvolvedor. Com o Git podemos trabalhar em equipe, conforme veremos nos próximos capítulos. Apesar de podermos trabalhar em equipe, não precisamos, por enquanto, nos conectar a um servidor central; fizemos o controle local do nosso projeto. Isso permite que possamos trabalhar em um projeto mesmo sem ter acesso à rede ou à internet pois essas alterações podem ser aplicadas mais tarde ao repositório "central".

# **Quais os estados possíveis?**

Os três estados de um repositório com os quais interagimos ao realizar o fluxo de controle com o Git são:

**Working directory** representa o estado atual dos arquivos no repositório. Podemos utilizar o comando git add para adicioná-los ao **index**, ou staging area, que representa uma visão preliminar das modificações que queremos definir para nosso projeto, comparando essas ao **HEAD**, o último passo completo do nosso projeto, que serve de referência para comparação do trabalho com o working directory.

**Marcando alterações interativamente**

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

Realize uma alteração substancial ao arquivo **index.html** (por exemplo, adicione um novo parágrafo de texto dentro da tag <body> **e** altere um texto existente).

Agora adicione o arquivo utilizando o modo interativo:

Rode o comando git add -i  
$ git add -i

staged unstaged path

1: unchanged +1/-0 index.html

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

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

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

1. What now>

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

staged unstaged path

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

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

staged unstaged path

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

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

updated one path

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

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

1. 5: patch 6: diff 7: quit 8: help

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

staged unstaged path

1: +1/-0 nothing index.html

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

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

1. 5: patch 6: diff 7: quit 8: help

Utilize a opção quit para terminar a interação!  
What now> 7

Bye.

1. $

Agora você pode fazer o commit do arquivo, que já está adicionado ao index. Confira usando o comando git status!

Boas práticas no uso do Git

Como vimos no capítulo de branches, a branch local master é uma branch que mantém referência para a branch master remota (origin/master). O processo de merge, quando acontece, pode confundir muitos desenvolvedores a respeito dessa referência. Outro problema é a geração de mensagens de merge, poluindo o nosso log.

Justamente para evitar tais situações, **é uma boa prática** sempre trabalhar em uma branch local que não seja a master. Então, para isso, um próximo passo é criar uma branch para realizar todo o trabalho e apenas utilizar a master para sincronizar os commits com o repositório remoto (origin).

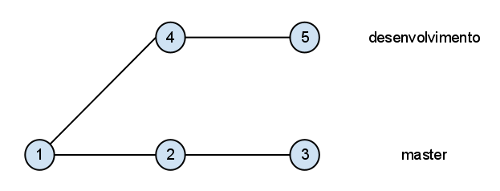
Então, um primeiro passo, que pode ser feito tanto pelo "jcfonsecagit" quanto pela "mmsoaresgit", é criar uma branch local chamada "desenvolvimento" e utilizá-la. Isso pode ser feito com o comando: git checkout -b desenvolvimento. Nesse momento, a branch atual é trocada para a que acabamos de criar, ou seja, não está mais na branch master.

A partir de então, todos os novos commits serão feitos na branch local desenvolvimento. Porém, como fazer para enviar esses commits para a branch master do repositório remoto, se a única que a rastreia é a nossa branch local master? Repare que agora precisamos levar os commits que fizemos na branch desenvolvimento, para a branch master.

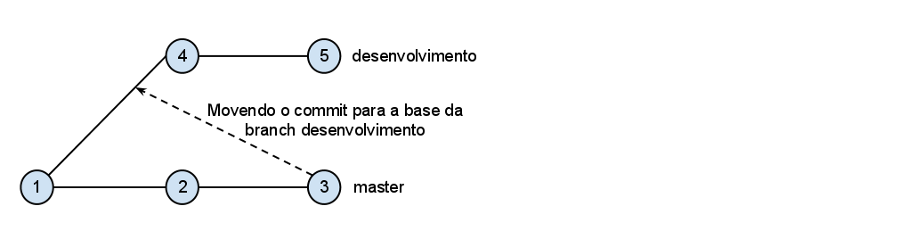
## **Envio dos commits das branches locais para o master remoto**

Considere que o usuário "jcfonsecagit" realizou 2 commits em sua branch local "desenvolvimento" e pretende sincronizar esses commits com a branch remota master. Para realizar essa tarefa, o primeiro passo, é verificar se a branch master não possui nenhuma nova atualização, ou seja, temos que fazer um checkout da master com git checkout master e, em seguida, realizarmos o git pull para buscar as novas alterações feitas por outras pessoas naquele repositório. Considere também que existem alterações para baixar do repositório e, justamente por isso, nesse instante, temos um problema para resolver.

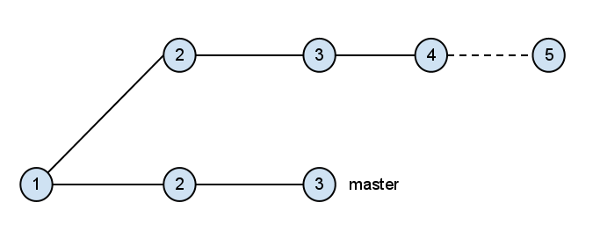
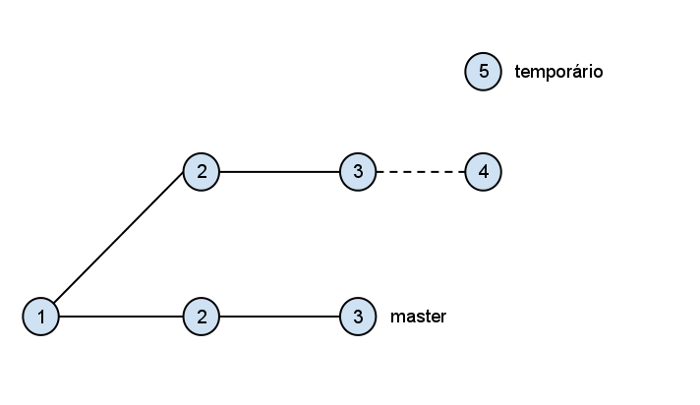
Temos a branch "desenvolvimento", que foi criada no commit 1. Porém, o último commit da master não é mais o commit 1 e sim, o 3.



Se simplesmente jogássemos o conteúdo da branch "desenvolvimento" para a master, poderíamos ter que tratar conflitos de vários commits ao mesmo tempo além do log dos commits ficarem confusos. Justamente para evitar essa situação, o Git possui o comando git rebase, onde podemos indicar qual é a nova base de commits que deve ser utilizada e resolver os conflitos commit por commit. No nosso caso, queremos que a branch "desenvolvimento" utilize como base de commits a "master" que acabamos de atualizar.



Para isso, temos que ir à branch "desenvolvimento" com o comando git checkout desenvolvimento e dizer que a nova base dela é o último commit que está na master, portanto git rebase master. Nesse instante, os commits feitos em "desenvolvimento" são movidos para uma branch temporária e o git atualiza a nova base de commits. Após essa atualização, o próprio Git traz de volta os commits que realizamos e os aplica sobre a nova base, um de cada vez.



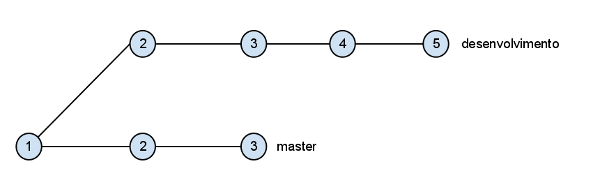
No caso, uma saída possível para a execução do git rebase master é a seguinte:

First, rewinding head to replay your work on top of it...

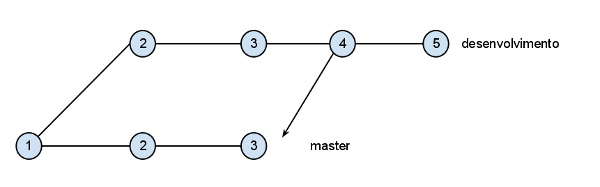
Applying: commit 1 de 2

Applying: commit 2 de 2

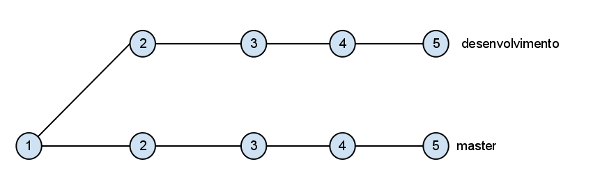
Nesse momento, todos os commits estão aplicados e organizados na branch "desenvolvimento".



Mas agora, é necessário levar esses commits para a branch "master". Repare que é um processo diferente do rebase, onde apenas queríamos trocar a base de commits utilizada. Agora, é necessário colocar apenas os commits novos na branch "master", porém, para isso, existe o comando git merge que move os commits de uma determinada branch para outra branch.



Para utilizá-lo, primeiro é preciso ir para a branch para a qual se quer levar os commits, no caso, a "master" (git checkout master). Em seguida, deve-se dizer para o comando git merge de qual branch virão os commits novos, que, no caso, é a branch "desenvolvimento". Para isso, basta executar git merge desenvolvimento.



Nesse instante, tem-se todos os novos commits na branch "master", prontos para serem enviados para o repositório remoto através do comando git push.

Note que agora, por mais que alterações tenham ocorrido no mesmo arquivo, ao visualizar o log, não existe mais aquele commit indicando que houve um merge. Muitos desenvolvedores que utilizam o Git usufruem desse ciclo para trabalhar em seus projetos, usando a branch local "master" apenas como uma transição entre a branch remota "master" e as outras branches locais.

## **Mas o rebase pode falhar**

O nosso último git rebase foi bem sucedido e não tivemos nenhum problema para enviar os commits para o nosso "master" remoto. No entanto, existem situações em que o git rebase pode falhar e, à primeira vista, trabalhar com essa situação pode não ser trivial.

Considere que os dois usuários, "jcfonsecagit" e "mmsoaresgit", estejam com seus repositórios locais totalmente atualizados, ou seja, sem mais nada para baixar. Nesse momento, o arquivo proposta\_1.html, que ambos possuem, está com o seguinte conteúdo:

<html>

<head>

<title>Proposta 1 para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do site</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Rodapé do site</h1>

</body>

</html>

Em seguida, "mmsoaresgit" **faz o git push** de 3 commits. O primeiro commit altera o title para, em vez de usar o número 1, utilizar o número por extenso, ou seja, "um". O segundo commit muda a palavra "site" no cabeçalho para "sistema". E o terceiro commit muda a palavra "site" no rodapé também para "sistema". Por fim, "mmsoaresgit" realiza o git pushde suas alterações, enviando-as para o repositório remoto. Com isso, "mmsoaresgit" gerou o seguinte log de commits:

commit 9ee6a2e5344ff14ba38461ab65f51927bc2d7096

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:33 2011 -0200

troca de site para sistema no rodape

commit 658ed785d5e5c933d6ccead69b5d1801dd52e331

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:12 2011 -0200

troca de site para sistema no cabecalho

commit 12ec2eb6cba5e1021e8ed609ac26188397dc8ed2

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:40:47 2011 -0200

trocando de 1 para um no title

O estado final em que "mmsoaresgit" deixou o arquivo é o seguinte:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Rodapé do sistema</h1>

</body>

</html>

O usuário "jcfonsecagit", trabalhando em seu computador, decide realizar alterações na mesma página numa branch local chamada "desenvolvimento", conforme vimos anteriormente. Contudo, a sua branch master não está sincronizada com a do repositório remoto. Ele altera a mensagem do rodapé para "Copyright - Caelum 2012" e realiza o commit de sua alteração, localmente, na branch "desenvolvimento". Ele realiza um outro commit alterando o "1" do title para "I" em algarismos romanos.

Agora, ele precisa enviar seus commits para o repositório remoto para que todos tenham acesso ao que ele acabou de fazer. Para tanto, antes de fazer o git push, ele vai para a branch "master" e realiza o git pull para trazer os novos commits que "mmsoaresgit" realizou. Para juntar as alterações que a "mmsoaresgit" realizou com as suas, "jcfonsecagit" precisa alterar a base de commits da branch "desenvolvimento", para incluir os novos commits que foram baixados. Para isso, ele vai à branch "desenvolvimento", com o git checkout desenvolvimento, e realiza o rebase, indicando que a base de commits é o que está na branch "master" com o comando git rebase master.

Como explicado anteriormente, os commits que o "jcfonsecagit" realizou serão colocados em um lugar temporário e a base da branch "desenvolvimento" ganhará os 3 novos commits. Em seguida, o rebase faz uma verificação para saber se os commits feitos por "jcfonsecagit" possuem algum conflito com algum dos commits que entraram na nova base, um de cada vez.

Nesse instante, o Git perceberá que a troca do rodapé que o "jcfonsecagit" realizou, adicionando o Copyright, conflitará com a alteração no rodapé que "mmsoaresgit" também fez e mostrará a seguinte mensagem:

First, rewinding head to replay your work on top of it...

Applying: colocando mensagem de copyright no rodapé

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

Auto-merging proposta\_1.html

CONFLICT (content): Merge conflict in proposta\_1.html

Failed to merge in the changes.

Patch failed at 0001 colocando mensagem de copyright no rodapé

When you have resolved this problem run "git rebase --continue".

If you would prefer to skip this patch, instead run "git rebase --skip".

To restore the original branch and stop rebasing run "git rebase --abort".

Novamente, apesar de a mensagem ser grande, fica mais fácil a sua interpretação. O Git primeiro tirou o commit do "jcfonsecagit" para alterar a base de sua branch, utilizando os novos commits da outra usuária, "mmsoaresgit". Após a troca da base de commits, o primeiro commit realizado pelo "jcfonsecagit" é aplicado. Porém, ele conflita com as alterações da outra usuária. Repare que o próprio Git percebe o conflito e tenta fazer o processo de "merge" automaticamente, como aprendemos no começo do capítulo. Mas, dessa vez, ele não consegue fazer, pois houve alteração no mesmo trecho de código.

Nesse momento, o Git nos deixa resolver o conflito existente no rebase, de forma manual. Para isso, o próprio Git nos coloca em um "ambiente temporário", no caso, uma branch temporária, só para resolvermos o conflito. Nesse momento, se fosse executado o comando git branch, teríamos o seguinte resultado:

\* (no branch)

desenvolvimento

master

Note a branch chamada "(no branch)" em que estamos. É nessa branch em que deveremos resolver o conflito. Nesse momento, temos 3 opções, que o próprio Git nos avisou anteriormente: continue, skip e abort.

Uma primeira opção é abortar a resolução do conflito e voltar atrás no rebase, ou seja, voltar ao ponto que se estava antes de realizar o rebase. Para isso, bastaria executar o comando git rebase --abort.

Uma segunda opção é descartar o seu commit atual que gera o conflito, o que pode ser realizado com o comando git rebase --skip.

Por fim, caso queiramos resolver o conflito, seja para ficar com nossa alteração, ou seja para fazer alguma outra modificação, é necessário permanecer nessa branch temporária, e corrigir os arquivos conflitados. O primeiro passo para descobrir quais arquivos devem ser mexidos é executar o comando git status:

# Not currently on any branch.

# Unmerged paths:

# (use "git reset HEAD <file>..." to unstage)

# (use "git add/rm <file>..." as appropriate to mark resolution)

#

# both modified: proposta\_1.html

#

no changes added to commit (use "git add" and/or "git commit -a")

Note que a primeira linha já nos diz que não estamos em uma branch válida e que também temos "merge" por fazer. O arquivo indicado pela sentença "both modified", ou seja, que "ambos modificaram", é o arquivo que temos que corrigir o conflito. Abrindo-o, encontraremos o trecho conflitante, demarcado como aprendemos anteriormente:

<<<<<<< HEAD

<h1>Rodapé do sistema</h1>

=======

<h1>Copyright - Caelum 2012</h1>

>>>>>>> mudando o rodapé para utilizar copyright

Agora, basta deixar o código que deve permanecer após o merge. No caso, vamos deixar a linha do Copyright. Assim, o conteúdo final do arquivo será:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Copyright - Caelum 2012</h1>

</body>

</html>

Agora que já sabemos qual é o estado que queremos manter no repositório, basta adicionarmos o nosso arquivo no Index do Git, executando o comando git add proposta\_1.html. Nesse momento, se executar o comando git status, teremos a seguinte saída:

# Not currently on any branch.

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: proposta\_1.html

#

Note que o proposta\_1.html **não** está mais marcado como "both modified". Nesse ponto, quando todos os arquivos conflitantes já estiverem nesse estado, é possível continuar o rebase através do comando git rebase --continue para juntar os outros commits. Novamente, teremos um outro conflito:

Applying: colocando mensagem de copyright no rodapé

Applying: trocando de 1 para I no title

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

Auto-merging proposta\_1.html

CONFLICT (content): Merge conflict in proposta\_1.html

Failed to merge in the changes.

Patch failed at 0002 title

When you have resolved this problem run "git rebase --continue".

If you would prefer to skip this patch, instead run "git rebase --skip".

To restore the original branch and stop rebasing run "git rebase --abort".

Dessa vez, vamos ignorar a alteração feita por "jcfonsecagit" e manter o que a "mmsoaresgit" fez com o comando git rebase --skip, excluindo esse último commit do rebase. Como não há mais nenhum commit que "jcfonsecagit" realizou, o processo de rebase está completo.

Por fim, precisamos atualizar a nossa branch master com os novos commits. Para isso, mudamos para a branch master com o comando git checkout master e juntamos os novos commits com git merge desenvolvimento. Agora, a nossa branch master está pronta enviar as atualizações para o repositório remoto (git push).

**Explicação do git rebase**

Vamos praticar o comando git rebase.

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

**Durante o desenvolvimento em um projeto, é comum realizar alterações no projeto que não são mais necessárias ou que gerem bugs cuja solução não é imediata. E muitas vezes desejamos descartar as alterações do arquivo, retornando para uma versão anterior. Por exemplo, se alteramos o nome de um arquivo junto com todas as suas referências, mas o nome não ficou bom, como podemos deixar o sistema como era antes? Uma solução seria olhar como o arquivo estava numa versão para alterar o nosso arquivo. Para resolver essa tarefa, o Git nos fornece ferramentas que permitem reverter alterações nos arquivos em seus três possíveis estados: Working Directory, Index e Head.**

# **Descartando alterações no Working Directory: git checkout**

Vamos continuar trabalhando com o repositório "propostas\_homepage", onde tratamos os conflitos. A versão atual do nosso arquivo "proposta\_1.html" é:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Copyright - Caelum 2012</h1>

</body>

</html>

O usuário "jcfonsecagit" decide alterar página, porém, por acidente, ele troca a "/" por "" na Tag </div>. O arquivo alterado ficou:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

<\div>

<h1>Copyright - Caelum 2012</h1>

</body>

</html>

Ao visualizar a página, o usuário "jcfonsecagit" percebeu que há um bug no sistema, mas ele não consegue encontrar onde está localizado o erro. Ao verificar o estado do sistema com o comando git status, ele verifica que só o arquivo "proposta\_1.html" foi modificado e as alterações ainda estão no "Working Directory". Então, ele decide descartar todas as alterações que foram realizadas desde o último commit, isto é, voltar o seu sistema para o estado em que se encontra no HEAD. Uma opção seria verificar as alterações que foram feitas com o comando git diff e desfazer as alterações manualmente. Mas e se muita coisa foi alterada dentro de um arquivo? Para este problema, o Git nos possibilita descartar todas as alterações que estão no "Working Directory" de um determinado arquivo. Para isso, utilizamos o comando git checkout passando o nome do arquivo cujas alterações serão removidas. No nosso caso, temos:

git checkout proposta\_1.html

A saída não devolve nada mas, ao realizar o comando git status, verificamos que não há mais alterações no arquivo "proposta\_1.html" que estejam no "Working Directory".

Note que já utilizamos o comando git checkout antes para alternar entre as branches do repositório. A diferença entre os dois é o argumento que passamos para o comando: se for o nome de uma branch, trocaremos de branch; mas se for o nome de um arquivo, deixaremos o arquivo conforme ele se encontra no HEAD da branch atual.

Vimos que é uma boa prática desenvolver uma tarefa numa branch diferente da branch master, deixando-a apenas para sincronização com o repositório remoto. Então, o usuário "jcfonsecagit" criou a branch "desenvolvimento" e mudou para ela com o comando git checkout -b desenvolvimento.

Enquanto isso, a usuária "mmsoaresgit" atualizou o repositório remoto modificando o arquivo "proposta\_1.html". Ao realizar o git pull na branch master, o usuário "jcfonsecagit" recebeu a atualização que a "mmsoaresgit" realizou. Agora, ele deseja utilizar a versão apenas do arquivo "proposta\_1.html" que se encontra na branch master, deixando o resto dos arquivos intactos. Uma solução seria olhar o arquivo na branch master e copiar o seu conteúdo para a branch desenvolvimento. Contudo, o Git já nos permite fazer tal operação de uma maneira mais simples: basta passar ao comando git checkout o nome da branch e o do arquivo que se deseja copiar. No nosso caso, temos:

git checkout master proposta\_1.html

Esse comando trará o arquivo "proposta\_1.html" como ele se encontra na branch "master" e o adicionará ao Index do repositório na branch "desenvolvimento", pronto para um commit.

# **Descartando alterações no Index: git reset**

Agora, imagine que o usuário "jcfonsecagit" renomeou o nome do arquivo "proposta\_1.html" e todas as suas referências nos outros arquivos a fim de melhorar a legibilidade do projeto. Em seguida, ele adicionou as alterações ao Index para realizar o commit. Contudo, ele percebe que o novo nome deixou o sistema mais confuso e deseja retornar o projeto como ele se encontra no HEAD. Se a alteração estivesse no Working Directory, poderíamos descartá-la com o comando git checkout. Mas isso não funciona se a alteração estiver no Index. Para isso, precisamos dizer ao Git que desejamos redefinir (reset) o nosso arquivo de acordo com a versão encontrada no HEAD:

git reset HEAD proposta\_1.html

Com isso, obtemos a seguinte saída:

Unstaged changes after reset:

M proposta\_1.html

Essa mensagem está indicando que o estado do arquivo "proposta\_1.html" foi alterado. Verificado-o com o comando git status, vemos que o arquivo está no estado Working Directory. Agora já podemos descartar as alterações do arquivo com o comando git checkout proposta\_1.html.

# **Guardando alterações para mais tarde: git stash**

Considere o caso em que "jcfonsecagit" realizou o seguinte commit alterando o arquivo "proposta\_1.html":

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

<\div>

<h1>Copyright - Caelum 2012</h1>

</body>

</html>

Em seguida, ele começa a modificar o arquivo para resolver uma outra tarefa. Contudo, ele descobre que o seu commit anterior apresentava um bug e deseja resolvê-lo antes de terminar a sua tarefa. Mas as suas alterações que estão no Working Directory e no Index ainda não são suficientes para a realização de um commit. Ele pode descartar as suas modificações com os comandos que vimos anteriormente mas, dessa maneira, ele terá que refazer tudo quando for reiniciar a tarefa. Para resolver isso, o Git nos permite guardar as alterações nesses dois estados em uma área especial, de onde podemos recuperá-los depois. Isso é feito com o comando git stash. Ao realizar este comando, obtemos como saída:

Saved working directory and index state WIP on desenvolvimento: b6c7cc8 trocando de 1 para I no title

HEAD is now at b6c7cc8 trocando de 1 para I no title

A mensagem acima está indicando que as alterações em nosso Working Directory e Index foram salvas em uma área distinta e que o nosso repositório foi restaurado de acordo com o HEAD.

Agora, "jcfonsecagit" consegue corrigir o bug que encontrou. O arquivo corrigido fica:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Copyright - Caelum 2012</h1>

</body>

</html>

Em seguida, ele realiza um commit indicando que corrigiu o bug: git commit -m "Corrigindo bug na tag html". Agora ele já pode retornar para a tarefa que estava executando anteriormente. Porém, como recuperar as alterações que foram salvas anteriormente? Isso é feito com o comando git stash pop. O retorno fica:

# On branch desenvolvimento

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: proposta\_1.html

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (0efeaa547ad59405a3b764334129e08b5846b935)

Assim, todo o trabalho inicial da tarefa que fora salvo retorna ao nosso sistema no estado de Working Directory.

E se a gente quiser saber se há algum estado salvo via git stash? Isso pode ser feito listando todos os stashes salvos no momento com o comando git stash list. Cada um dos estados salvo é nomeado da seguinte maneira: "WIP on [nome\_do\_branch]: [hash] [mensagem\_do\_commit\_head]". Também há uma referência ao stash com stash@{0} por exemplo, para o último stash criado. Caso houvessem outros, as referências seriam stash@{1}, stash@{2} e assim por diante.

O comando git stash pop utiliza como padrão o último stash criado. Para aplicar alterações de um stash mais antigo, usamos sua referência:

git stash pop stash@{1}

# **Descartando commits indesejados**

Enquanto as modificações indesejadas ainda estão em nosso Working Directory ou no Index, a operação de descartá-las ou desfazê-las é simples. Mas o que acontece se já foi realizado o commit com as alterações que desejamos remover? Vamos ver um exemplo em que isso acontece.

Imagine novamente que "jcfonsecagit" alterou o arquivo "proposta\_1.html" trocando "/" por "\ " na Tag div:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

<\div>

<h1>Copyright - Caelum 2012</h1>

</body>

</html>

Em seguida, ele realizou o commit: git commit -m "Alterando a tag div". Mas, ao ver a página, "jcfonsecagit" percebeu que a página contém um erro e deseja desfazer as alterações do seu último commit.

Semelhante ao caso em que as alterações estavam no Index, precisamos indicar ao Git que desejamos redefenir o arquivo. Contudo, não podemos mais usar o HEAD como referência. No lugar dele, devemos usar o penúltimo HEAD como referência. Para isso, podemos utilizar o hash correspondente ao penúltimo commit. Para encontrar o hash, usamos o comando git log:

commit 23923a7a8059bc37c15fe4331af862eb15ceee89

Author: João Carlos Fonseca <jcfonseca@gmail.com>

Date: Thu Jan 5 15:49:30 2012 -0200

alterando a div

commit b6c7cc8e3fea9b255b5845e1114588206679f609

Author: João Carlos Fonseca <jcfonseca@gmail.com>

Date: Thu Jan 5 15:48:38 2012 -0200

trocando de 1 para I no title

commit fe69c05c59e9775b19ecb02256c2ad1b50278037

Author: João Carlos Fonseca <jcfonseca@gmail.com>

Date: Thu Jan 5 15:48:13 2012 -0200

colocando mensagem de copyright no rodapé

commit 9ee6a2e5344ff14ba38461ab65f51927bc2d7096

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:33 2011 -0200

troca de site para sistema no rodape

commit 658ed785d5e5c933d6ccead69b5d1801dd52e331

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:12 2011 -0200

troca de site para sistema no cabecalho

commit 12ec2eb6cba5e1021e8ed609ac26188397dc8ed2

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:40:47 2011 -0200

trocando de 1 para um no title

O hash é a sequência de caracteres localizada à direita da palavra commit. No nosso caso, o hash do penúltimo commit é b6c7cc8e3fea9b255b5845e1114588206679f609. Portanto, se digitarmos o comando git reset b6c7cc8e3fea9b255b5845e1114588206679f609, o último commit será descartado, direcionando nosso HEAD para o penúltimo commit. Dessa maneira, as alterações encontradas no último commit são revertidas e aplicadas aos arquivos em nosso Working Directory. Isso pode ser visto a partir do log do nosso projeto:

commit b6c7cc8e3fea9b255b5845e1114588206679f609

Author: João Carlos Fonseca <jcfonseca@gmail.com>

Date: Thu Jan 5 15:48:38 2012 -0200

trocando de 1 para I no title

commit fe69c05c59e9775b19ecb02256c2ad1b50278037

Author: João Carlos Fonseca <jcfonseca@gmail.com>

Date: Thu Jan 5 15:48:13 2012 -0200

colocando mensagem de copyright no rodapé

commit 9ee6a2e5344ff14ba38461ab65f51927bc2d7096

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:33 2011 -0200

troca de site para sistema no rodape

commit 658ed785d5e5c933d6ccead69b5d1801dd52e331

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:12 2011 -0200

troca de site para sistema no cabecalho

commit 12ec2eb6cba5e1021e8ed609ac26188397dc8ed2

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:40:47 2011 -0200

trocando de 1 para um no title

# **Desfazendo commits antigos**

Nem sempre bugs são encontrados logo após que foram criados, tendo vários commits realizados desde que o bug foi introduzido. O comando git resetpermite desfazer qualquer número de commits, bastando utilizar o hash do commit que queremos manter como HEAD. Contudo, todos os commits que foram realizados após ele serão descartados, perdendo todas as novas funcionalidades. Por isso, o comando git reset só é recomendado quando desejamos desfazer poucos commits e, principalmente, se esses ainda não tiverem sido enviados a um repositório remoto. Se os commits já foram enviados, há alguma chance dos commits já terem sido adquiridos pelos outros desenvolvedores do projeto, e aí não será possível excluí-los. Então como descartamos as alterações do commit que gerou o bug neste caso?

Quando desejamos remover commits que foram realizados há algum tempo, a melhor maneira seria revertendo-os, isto é, apenas desfazendo as alterações daqueles commits. Todos os outros commits serão mantidos em seu respectivo estado. Isso é feito utilizando o comando git revert passando como argumento o hash do commit que se deseja reverter. Ao digitar o comando, um novo commit revertendo as alterações do commit escolhido será realizado e o editor de texto padrão se abrirá para que se possa digitar a mensagem do commit. Para que o comando seja utilizado, é necessário que o Working Directory e o Index estejam "limpos", ou as alterações atuais serão descartadas.

Uma boa alternativa para maior flexibilidade é a opção "-n", para que as alterações sejam revertidas e adicionadas ao nosso Working Directory e Index. Assim podemos fazer alterações adicionais antes de criar um novo commit de reversão.

git revert -n [hash\_do\_commit]

# **Buscando por bugs em muitos commits**

Quando conhecemos as alterações indesejadas, caso sejam poucas, é possível que façamos um novo commit com as alterações necessárias nos arquivos, desfazendo o que foi feito antes. Infelizmente, nem sempre temos um cenário tão simples, então precisamos utilizar um recurso mais avançado do Git.

Imagine que após algum tempo, uma funcionalidade que estava correta, parou de funcionar subitamente em nosso projeto. Um link, que antes funcionava, parou de funcionar e não sabemos quando fizemos a alteração que causou esse problema. Sabemos, porém, uma data aproximada de quando funcionava, por exemplo na segunda-feira.

Com o comando "git log", podemos encontrar o hash do primeiro commit daquele dia; é um bom ponto de partida. Vamos supor que, em nosso projeto, o primeiro commit da última segunda-feira é o "02bfc44...". Se desejarmos, podemos utilizar o comando git checkout 02bfc44 e verificar se estava funcionando. Supondo que está, devemos voltar ao HEAD, com git checkout HEAD.

Agora que estamos de volta ao HEAD, veremos quantos commits temos entre ele e o commit da segunda-feira que temos certeza que funciona:



Pronto, agora temos que testar commit a commit, ou seja, realizando um git checkout com o hash de cada commit. Imagine o trabalho e o tempo que isso tomará! Por isso, o Git nos fornece o comando git bisect. Vamos utilizá-lo e acompanhar seu funcionamento.

git bisect start

git bisect bad HEAD

Acima, iniciamos uma sessão de "bisect" e marcamos o commit HEAD como "bad" (ruim), ou seja, indicamos que ele contém o bug o qual queremos encontrar o momento em que foi introduzido.



Agora precisamos marcar qual commit deve ser utilizado como estando OK:

git bisect good 02bfc44

Agora que o Git sabe qual commit funciona e qual não funciona, ele automaticamente faz o checkout de um commit intermediário para que possamos verificar se funciona. Por exemplo o commit c93e5da:



Podemos testar agora se, nesse ponto, nossa funcionalidade estava OK. Caso não esteja, precisamos marcar o commit atual como "ruim":

git bisect bad



Por dedução lógica, todos os commits posteriores ao atual também estão defeituosos. Agora o Git conhece um novo intervalo de commits para buscar pelo erro, com metade do tamanho do intervalo anterior:



Agora com esse novo cenário, o Git faz automaticamente o checkout de um commit intermediário desse novo intervalo, por exemplo o commit af654d1:



Novamente podemos testá-lo e verificar se ele contém o erro que procuramos. Digamos que esse commit está OK, então temos marca-lo como tal:

git bisect good



Utilizando o mesmo conceito de antes, o Git encontra um novo intervalo, sendo que agora temos o cenário inverso de antes: podemos procurar por commits posteriores ao atual.



Novamente, continuamos o bisect a partir do commit intermediário do novo intervalo, por exemplo o commit df83cc9:

Novamente, vamos testá-lo e, em nosso exemplo, constatamos que a funcionalidade não está OK. Vamos marcar o commit atual como "ruim":

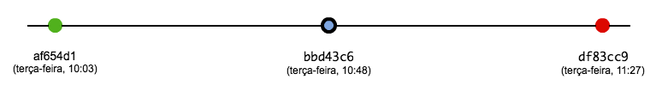
git bisect bad



Seguindo o ciclo, temos um novo intervalo para continuar nossa busca pelo erro:



No nosso caso, como tínhamos um número pequeno de commits para verificar, encontramos o commit onde o bug foi inserido em nosso projeto. O bisect faz o checkout automático do commit defeituoso, mas a saída no prompt de comando agora traz novas informações sobre ele.



Apesar de fazer checkout dos commits, o ciclo de bisect trabalha em uma branch exclusiva no repositório local. Portanto, para corrigir o erro, conhecemos o hash do commit e podemos utilizar os comandos git reset, git revert ou ver as alterações realizadas naquele commit específico com git show bbd43c6 e decidir a melhor maneira de corrigir.

Note que esse caso demonstra muito bem a importância de realizarmos pequenos commits, gravarmos pequenos avanços em nossos projetos, mesmo que em muitos commits, pois temos ferramentas para encontrar qualquer problema posteriormente de maneira automatizada.

# 

# **Branch e arquivo com mesmo nome**

Como vimos, o comando git checkout serve tanto para trocar de branch quanto descartar alterações de um arquivo do working directory. Mas se possuímos um arquivo e uma branch com o mesmo nome, o *Git* trata como padrão considerar que o usuário quer mudar de branch. Precisamos então de uma alternativa para diferenciar a branch do arquivo.

Para isso existe uma notação no comando git checkout, que indica que *daqui pra frente só serão listados arquivos*. Digamos que temos as branches design e master (e estamos na master) e o arquivo design que na master contém alterações. Para garantir que vamos restaurar o arquivo, precisamos usar o seguinte comando:

git checkout -- design

Essa notação de dois hífens -- indica que os parâmetros seguintes serão todos nomes de arquivos, permitindo que restauremos o arquivo design ao estado original.

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

Como descartamos **definitivamente** as alterações que estão no último commit?

git reset --hard HEAD~1

Usando esse comando, descartamos definitivamente as mudanças feitas no último *commit*.

git stash

Este comando faz com que as alterações que estão no working directory e no index sejam salvas, retornando o estado para o último *commit*.

git stash pop

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

git stash apply

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

git stash drop

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

git whatchanged -p

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

**Organização do trabalho com branches**

**É comum, durante o desenvolvimento de novos recursos ou de correção de bugs, interrompermos o trabalho por falta de tempo ou porque surgiu uma nova prioridade do projeto. Mas o que fazer com as alterações que estão pela metade? Deletar e depois ter que refazer tudo novamente? Colocar no repositório algo pela metade, podendo quebrar todo o sistema?**

**Uma solução que é bem utilizada no dia a dia é a de criar uma seção separada do projeto, uma bifurcação, uma branch. Tal solução possibilita desenvolver separadamente cada uma das funcionalidades sem interferir no desenvolvimento de uma outra parte do projeto.**

# **Trabalhando com branches: primeiros passos**

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

**git branch design**

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

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

**git branch**

**design**

**\* master**

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

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

**git checkout design**

**Switched to branch 'design'**

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

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

**body {**

**background-color: blue;**

**}**

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

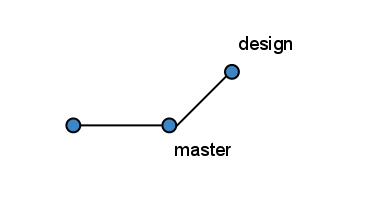
**<link rel="stylesheet" type="text/css" href="/design.css"/>**

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

**git add design.css index.html**

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

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

****

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

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

## **Verificando atualizações do repositório**

**git fetch origin**

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

# **Compartilhando branches locais com outros desenvolvedores**

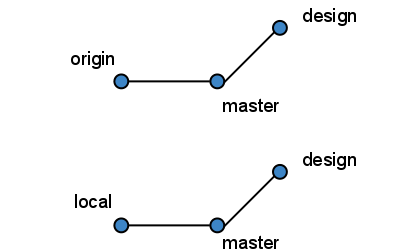
**Agora, o que acontece se você começou alguma alteração em um projeto e não terminou? Será que é bom manter apenas localmente? E se alguém quiser continuar as alterações que você iniciou?**

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

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

**git push origin design**

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

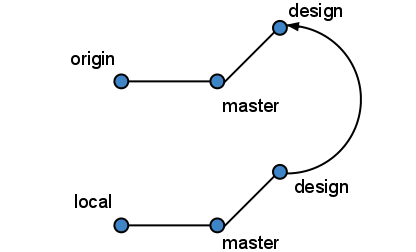
****

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

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

**git push -u origin design**

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

****

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

**git branch -r**

**origin/HEAD -> origin/master**

**origin/design**

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

**git branch -t design origin/design**

**Temos como resultado:**

**Branch design set up to track remote branch design from origin.**

**Switched to a new branch 'design'**

**git branch design**

**Com este comando criamos uma nova branch chamada design. Se quiser criar e já mudar para a branch nova, basta dar o comando git checkout -b design, que indica que você quer mudar para a branch design, mas ao mesmo tempo criá-la com a opção -b que representa *nova branch*.**

git branch

Ao digitar o comando git branch, todas as branches locais são listadas. Passando a opção -a as branches remotas são incluídas.

-d

Sim, este comando remove uma branch, porém somente se ela estiver sincronizada com outra. Senão, é necessário forçar com a opção -D.

-u

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

## 

## 

## 

## 

## >, = e <

## O *Git* utiliza os caracteres >, < e =. Entre o < e o = fica o conteúdo antigo e entre o > e = fica o conteúdo novo.

## 

## 

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

1. Criação de uma branch local com o mesmo nome da branch remota
2. Mudança para essa nova branch criada
3. Criação de link entre a branch local e remota

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

* Alternativa correta
* -t
* Com o comando git checkout -t origin/design, todo

Algumas das ferramentas gratuitas mais comuns são o [kdiff3](http://kdiff3.sourceforge.net/), [Meld](http://meldmerge.org/) ou [P4Merge](https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge).

# **Como aplicar o pull request**

Ao receber um pull request, os usuários selecionados pelo autor serão notificados e, a partir de então, o código enviado é avaliado. Caso o código não esteja adequado, o pull request pode ser negado. Em contrapartida, caso o código seja aceito, o mesmo deverá ser introduzido ao repositório do projeto.

Sempre que um pull request é criado, uma URL é atribuída a ele. Dessa maneira, é possível visualizar através do navegador as informações do pull request criado. Essa URL possui o seguinte formato: https://github.com/[dono\_do\_projeto]/[projeto]/pull/[id\_do\_pull\_request]. Entrando nesse endereço, é possível visualizar também as modificações introduzidas pelo pull request, que é como é possível avaliar se o pull request enviado está OK ou não.

O Git nos permite introduzir o diff do pull request de várias maneiras. Note que já temos o diff que o pull request gerou, e poderíamos simplesmente aplicá-lo ao nosso projeto alterando manualmente as linhas indicadas. Dessa forma, realizaríamos novos commits apenas refletindo o que foi sugerido pelo pull request. É uma abordagem funcional, mas que pode ser bastante trabalhosa, principalmente se o pull request envolver muitas modificações e arquivos.

Uma outra possibilidade é fazer todo esse processo através do próprio Git, sem precisar copiar código. Uma vez que o pull request foi gerado, basta baixá-lo para o nosso projeto e realizarmos o git merge para trazer as alterações para uma branch nossa. Com isso, bastaria primeiro fazer o git remote add [alias\_do\_repositorio\_remoto] [url\_do\_repositorio\_remoto]. E em seguida fazermos o git fetch. Com isso, já teremos todas as alterações baixadas em nosso computador. A seguir, basta realizar o git merge [alias\_do\_repositorio]/[branch]. Isso trará para a branch atual os commits existentes no repositório remoto. Por fim, temos que colocar os commits em nosso repositório remoto. Para isso, basta executar o git push e, com isso, introduzimos os commits do pull request de outro desenvolvedor ao nosso projeto.

Uma outra possibilidade é utilizar o botão "Merge pull request" existente na tela do pull request, visível apenas para os administradores do repositório. Esse botão desencadeia todo o processo de trazer os commits do pull request para a repositório automaticamente, sem precisar realizar nenhum comando.

# **Redução dos comandos com Alias**

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

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

[alias]

st = status

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

[alias]

st = status

ci = commit

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

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

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

[alias]

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

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

[alias]

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

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

# **Adicionando cores na saída do console**

Por padrão, ao exibirmos um diff no console, o Git mostra as modificações identificadas com um sinal de - (menos) e + (mais). Dessa forma, conseguimos saber quais linhas foram modificadas, o que saiu e o que está entrando na nova versão do arquivo. Apesar de conseguirmos visualizar as modificações dessa maneira, quando o diff se torna grande, é muito fácil se perder no meio de tanta alteração.

Para facilitar esse trabalho, o Git permite que habilitemos cores para a visualização dos diffs. Para isso, basta editar o arquivo .gitconfig, adicionando a seção [color] no arquivo, com a entrada diff indicando que queremos cores para a visualização dos diffs:

[color]

diff = auto

O valor "auto" indica que as cores serão utilizadas apenas no terminal e se houver suporte para cores nele. Dessa maneira, as linhas removidas do arquivo agora são exibidas em vermelho e as linhas adicionadas, em verde.

Também é possível adicionar cores para as saídas de outros comandos como, por exemplo, o branch, onde a branch atual é mostrada em outra cor, e o status, onde cada estado do arquivo é mostrado em uma cor diferente.

# **Visualizações de Log**

Analisar os logs de commits é uma tarefa muito comum de ser realizada ao se trabalhar com um sistema de controle de versões, principalmente quando temos que rastrear uma alteração que foi feita na base de código da aplicação.

Uma alternativa para visualizar esses logs, caso se esteja utilizando o Github, é acessar a lista de commits disponível através do navegador. Assim, teremos uma lista do que fizemos. Apesar de ser uma alternativa funcional e viável, pode não ser a maneira mais produtiva, pois teremos que esperar a página abrir e, quando quisermos ver o que foi modificado por um commit, precisaremos abri-lo.

Uma alternativa também interessante é visualizar os logs localmente utilizando o comando git log, que, no caso do projeto "propostas\_homepage", pode gerar uma saída similar a:

commit 8b864af3b7d7e48e32731fc168db6c959677c964

Merge: bf04eb4 601c3dc

Author: jcfonsecagit <jcfonsecagit@gmail.com>

Date: Thu Dec 29 15:10:57 2011 -0200

Merge branch 'master' of github.com:mmsoaresgit/propostas\_homepage

commit bf04eb47a695e8dd79d08d16dbdd8ee5463ba78e

Author: jcfonsecagit <jcfonsecagit@gmail.com>

Date: Thu Dec 29 15:09:54 2011 -0200

trocando de h4 para h1

commit 601c3dc571023092141452420c42fdb119b1b52a

Author: mmsoaresgit <mmsoaresgit@gmail.com>

Date: Thu Dec 29 15:09:02 2011 -0200

alteracao do titulo da pagina

No entanto, é possível mudar essa visualização para termos na tela informações mais detalhadas ou apenas algumas informações que desejemos. Uma possibilidade é exibir no log de commits as alterações que foram feitas através da flag "-p". Com isso, o comando git log -p pode gerar uma saída similar a:

commit bf04eb47a695e8dd79d08d16dbdd8ee5463ba78e

Author: jcfonsecagit <jcfonsecagit@gmail.com>

Date: Thu Dec 29 15:09:54 2011 -0200

trocando de h4 para h1

diff --git a/index.html b/index.html

index 5ac7f9e..c6828da 100644

--- a/index.html

+++ b/index.html

@@ -7,6 +7,6 @@

Conteúdo do site

- <h4>Rodapé do site</h4>

+ <h1>Rodapé do site</h1>

</body>

</html>

\ No newline at end of file

Caso queiramos saber apenas um resumo do commit, com a quantidade de linhas adicionadas e removidas, temos a flag "--stat":

commit bf04eb47a695e8dd79d08d16dbdd8ee5463ba78e

Author: jcfonsecagit <jcfonsecagit@gmail.com>

Date: Thu Dec 29 15:09:54 2011 -0200

trocando de h4 para h1

index.html | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

Também é possível definir a quantidade de informações exibidas no log através de maneiras pré-definidas pelo próprio Git. Para isso, podemos utilizar a opção --pretty, onde precisamos passar um formato que será utilizado, podendo ser oneline, short, medium, full, fuller, email, raw ou então, caso nenhum atenda as necessidades, é possível criar sua própria formatação de log.

Utilizando no projeto o comando git log --pretty=oneline, teremos a seguinte saída:

8b864af3b7d7e48e32731fc168db6c959677c964 Merge branch 'master' of github.com:mmsoaresgit/propostas\_homepage

bf04eb47a695e8dd79d08d16dbdd8ee5463ba78e trocando de h4 para h1

601c3dc571023092141452420c42fdb119b1b52a alteracao do titulo da pagina

Note que, no oneline, não temos o nome da pessoa que realizou o commit. Podemos utilizar então nosso próprio formato para visualizar esse log mostrando também o nome do autor do commit e a data em que foi feita a alteração. Para isso, teremos que utilizar o comando --pretty, indicando o nosso próprio formato de log. Para indicar o formato do log, o Git disponibiliza algumas sequências de caracteres que possuem um significado. Por exemplo: %an significa o nome do autor, %ad signfica a data e hora do commit e %srepresenta a mensagem de commit. Dessa maneira, podemos utilizar o comando da seguinte maneira para mostrar uma saída elegante: git log --pretty='%an realizou commit em %ad: %s'

jcfonsecagit realizou commit em Thu Dec 29 15:10:57 2011 -0200: Merge branch 'master' of github.com:mmsoaresgit/propostas\_homepage

jcfonsecagit realizou commit em Thu Dec 29 15:09:54 2011 -0200: trocando de h4 para h1

Para uma lista completa das opções possíveis de se usar, pode-se acessar a documentação do comando git log através da execução do comando git log --help.

Por fim, uma alternativa para visualizar a evolução dos commits, quando se trabalha com diferentes branches e não se faz rebase das mesmas, é a utilização da flag --graph, que mostra um gráfico contendo a evolução do projeto e a criação e fechamento das branches. Podemos, inclusive, utilizá-la em conjunto com nossa formatação da saída, que utlizamos há pouco, executando o comando git log --pretty='%an realizou commit em %ad: %s' --graph, o que nos dará uma saída parecida com:

\* jcfonsecagit realizou commit em Thu Dec 29 15:10:57 2011 -0200: Merge branch 'master' of github.com:mmsoaresgit/propostas\_homepage

|\

| \* mmsoaresgit realizou commit em Thu Dec 29 15:09:02 2011 -0200: alteracao do titulo da pagina

\* | jcfonsecagit realizou commit em Thu Dec 29 15:09:54 2011 -0200: trocando de h4 para h1

|/

\* mmsoaresgit realizou commit em Thu Dec 29 15:05:00 2011 -0200: estado inicial do index.html

Note que, com essa visualização, conseguimos saber o momento em que a branch foi criada e quando ela foi trazida para o "master", nesse caso com o "merge" automático do Git.

**Criação de alias complexo**

**Como o fluxo do *Git* por vezes pode ser complexo, principalmente ao trabalhar com branches locais para organizar o desenvolvimento, é muito comum a criação de atalhos que façam todo o processo de publicação do código, ou seja: pull -> rebase em outra branch -> merge -> push.**

**Vamos criar um atalho chamado publica que faça todo esse processo de uma só vez para uma branch chamada dev! Para isso, abra o arquivo de configuração global (~/.gitconfig) e insira a seguinte linha no bloco alias:**

**[alias]**

**publica = !git checkout master && git pull && git checkout dev && git rebase master && git checkout master && git merge dev && git push**

**Agora, ao rodar o comando git publica, todo o processo abaixo será feito, em ordem, e interrompido caso o anterior falhe:**

1. **git checkout master: altera o local de trabalho para a branch master**
2. **git pull: atualiza o histórico da branch master**
3. **git checkout dev: altera o local de trabalho para a branch dev**
4. **git rebase master: atualiza o HEAD da branch dev para receber as alterações da branch master**
5. **git checkout master: altera o local de trabalho novamente para a branch master**
6. **git merge dev: mescla as alterações da branch dev na master**
7. **git push: envia suas alterações para o repositório remoto**

**[color]**

**branch = auto**

**Incluir esta opção automatiza a coloração do comando git branch. Outra opção é utilizar branch = true.**

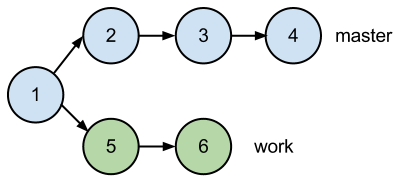
**[color]**

**branch = yes**

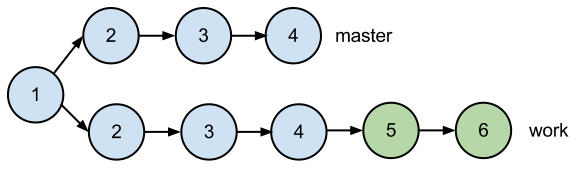
**Incluir esta opção automatiza a coloração do comando git branch. Outra opção é utilizar branch = on ou branch = 1.**

# **Selecionando commits**

**Quando realizamos um git merge ou um git rebase, todos os commits de uma branch são aplicadas numa outra branch. Contudo, às vezes desejamos que apenas alguns determinados commits sejam aplicados na outra branch. Vamos observar o seguinte exemplo:**

****

**Ao realizarmos um git rebase master na branch work, obtemos o seguinte resultado:**

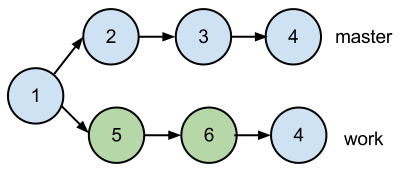
****

**Agora imagine que nós só queiramos aplicar o commit 4 da branch master em nossa brach work pois ela introduziu algum trecho de código que nos ajudará em nossa tarefa. Para conseguir esse resultado, utilizaremos o comando git cherry-pick.**

**Para usar o cherry-pick, precisamos indicar quais os commits que serão aplicados em nossa branch atual. Por exemplo, se quisermos mover apenas o commit 4 para a branch work, fazemos:**

**git cherry-pick 4**

**O resultado seria algo como:**

****

**Só que, em vez de passar o número 4, podemos passar a "hash" referente ao commit que você quer trazer. Por exemplo:**

**git cherry-pick 19f0bb7d8b4be8ecd687b48fca301b71b95eab41**

# **Perigos do Cherry Pick**

**Com cherry-pick, temos a liberdade de escolher quais commits queremos trazer para a nossa branch. Mas veja que isso pode ser perigoso: às vezes, trazer um commit isolado, sem os commits ao redor, pode gerar problemas de merge ou até mesmo problemas no código.**

**Os problemas de merge gerados pelo cherry-pick são resolvidos de maneira semelhante ao git merge e git rebase. Você deve abrir os arquivos com conflito, fazer as mudanças necessárias no código (eles também estão demarcados com >>>>) e, em seguida, adicionar (git add) os arquivos com conflito e commitar (git commit).**

## Ao executar o cherry-pick, o *commit* escolhido é copiado. Contudo, existe uma opção que permite não gerar este commit, fazendo com que as alterações sejam adicionadas ao index. Qual é essa opção? Se precisar de ajuda, use o comando git cherry-pick --help.

* **Alternativa correta**
* **-n**
* **A opção -n ou --no-commit permite que recuperemos as alterações de um dado *commit* sem precisar inseri-lo no histórico local.**

## Muitas vezes queremos passar um conjunto sequencial (range) de *commits* para o comando git cherry-pick. Como isso é feito? Se precisar, use o comando git help gitrevisions para encontrar a resposta.

## Imagine que o *commit* inicial tenha a hash abcd e o commit final tenha a hash 1234.

* **Alternativa correta**
* **git cherry-pick abcd..1234**
* **Correto! Para pegar todos os *commits* dentro de um intervalo, devemos escrever o hash do *commit* mais antigo seguido por dois pontos sem espaço e seguido pelo hash do *commit* mais recente. Cuidado, é importante não haver espaços e que sejam exatamente dois pontos!**