RESUMO CURSO GIT

Sumário

1. Introdução

Para servidor de repositório do Git será usado o github.com, através de uma conta gratuita.

Criar inicialmente uma chave publica no computador para que o github autentique a máquina que está sendo usada. E sabe qual usuário está usando o github no momento.

Como configurar no prompt: ssh-keygen –t rsa –C “e-mail usado no cadastro da conta no github”. Enter para gravar no diretório padrão. Enter para não gravar senha.

Abrir o arquivo id_rsa.pub, copiar o texto e inserir no github no menu de configurações SSH Public Keys.

Para testar a configuração clone um projeto qualquer no github, copie o endereço do projeto no github, escolha um diretório no seu computador, acesse através do prompt e digite: git clone “endereço no github”.

Comandos úteis: ls (mostra os arquivos), git tag (mostra as versões do projeto), git checkout v0.1 (para abrir uma versão X), git diff v0.1 v0.2 (mostra as diferenças – o que saiu ++ o que entrou), git blame index.html (mostra linha a linha quem efetuou a alteração e em que momento).

2. Inicialização de um repositório

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                    // criando um diretório no windows
```

```
cd curso-git                       //acessando o diretório
```

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                           //inicia um diretório git
```

Será exibida uma mensagem similar a: Initialized empty Git repository in /diretorio/repostorio/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, chamada 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                    // lista de arquivos cujas alterações serão rastreadas
```

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

```
git status                      //verifica se o arquivo está no work directory ou no index...
```

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              //para que o arquivo possa ir para a área de
                                Index ou Staging Area.
```

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"                    //commitando para a HEAD
```

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.

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 como 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 persisti-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".

3. Sincronização dos dados com o repositório

O github nos fornece a possibilidade de criarmos uma conta centralizada para acessar os nossos repositórios. Para criar o repositório você deve acessar a sua conta e clicar em “new repositor”.

Após isso, é preciso configurar para que o repositório local com seus arquivos saiba que irá se comunicar com o repositório remoto que acabamos de criar no github. Para isso, executamos o comando:

```
git remote add origin git@github.com:seu_endereco  //ta informando o endereço do repositório remoto
```

O próximo passo é enviar os arquivos do repositório local para o repositório remoto no github:

```
git push –u origin master                    //enviando para o repositório remoto (github)
```

Pronto, assim enviamos todos os commits para o github. Para visualiza-los no github, bastar acessar o repositório e clicar em commits.

Imagine que Ana tenha feito isso, agora Bob quer baixar os arquivos do repositório remoto e trabalhar no projeto também, ele deve acessar o endereço do projeto no github e clonar o proejto:

```
git clone git@github.com:endereco                    //clonando o repositório remoto
```

Os dois usuários agora tem os arquivos do projeto nas suas máquinas. Caso Ana faça uma alteração em um arquivo do projeto local. Após a modificação, ela deve adicionar o projeto para a Stage Area:

```
git add index.html                  //adicionando na stage area
```

Depois deve commitar o arquivo:

```
git commit –m “alterando o arquivo index”                commitando para a HEAD
```

Por fim ela vai enviar as alterações para o repositório remoto no github:

```
git push                                    //enviando para o github
```

Ao visualizar no github, um novo commit será incluído. Nesse momento, o repositório local de Bob está desatualizado, ele precisa buscar as alterações que Ana realizou, usando o comando:

```
git pull                                    //trazendo as atualizações do repositório remoto
```

O git pull indica quais arquivos foram trazidos. Agora que Bob tem todos os arquivos, ele também quer modifica-lo. Então agora ele irá remover alguma parte do arquivo. Após isso, ele fez todo o processo: adicionou no index, fez o commit. E por fim enviou as alterações para o repositório remoto. Para que vários usuários possam colaborar no projeto, eles devem ser

adicionados pelo administrador do repositório em Collaborators, passando o nome da conta no git hub.

4. Trabalhando com ramos (branches)

O que é branch e por que utilizá-las?

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                    //criando a brach 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                                  // listando as braches do repositório local
  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                  //para trabalhar na brach 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:

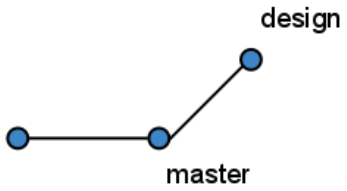
```
<link rel="stylesheet" type="text/css" href="/design.css"/>
```

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.

Compartilhando branches locais com outros desenvolvedores

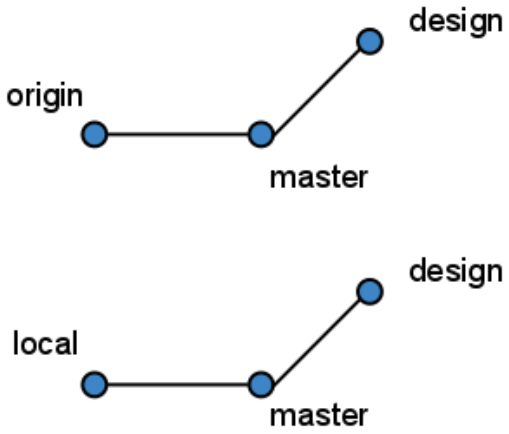
Agora, o que acontece se você começou alguma alteração em um projeto e não terminou? Será que é bom manter apenas localmente? E se alguém quiser continuar as alterações que você iniciou?

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                enviando as alterações para a branch remota design
```

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

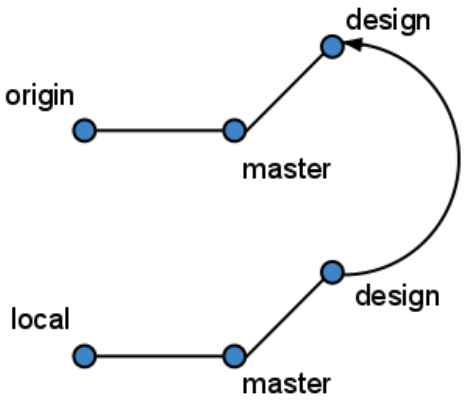


Porém, toda vez que atualizarmos tanto o nosso projeto local quanto o projeto remoto, precisaremos indicar qual o repositório e o nome da branch que a nossa branch local se refere no remoto, isto é, precisaremos digitar `git pull origin design` e `git push origin` design para atualizar os repositórios 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                                    // para visualizar todas as branches remotas
```

```
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'
```

5. Resolução de Conflitos

Imagine um projeto em que Bob e Alice clonaram e têm os arquivos nos seus repositórios locais. Bob altera o cabeçalho do arquivo index.html, add e faz o commit.

Em paralelo, Alice altera o rodapé do arquivo index.html, add e faz o commit.

Nesse momento os dois têm alterações para enviar ao repositório remoto. As alterações são em pontos diferentes e não conflitam.

Bob faz o `git push` e envia suas alterações para o github. Alice tenta fazer o `git push`, mas o push falha. O repositório de Alice não está sincronizado com o repositório remoto, e ela não pode fazer o push. Ela precisa pegar as alterações: `git pull`

Como os arquivos não conflitam, o git tratam as diferenças e faz o merge automático do arquivo. Se você tentar visualizar o histórico, usando o `git log`, verá que tem um novo commit que o git criou, chamado “Merge branch master”, o merge automático. Agora Alice pode realizar o push e enviar para o repositório remoto.

Mas e se as alterações conflitarem? Se os dois alterarem o cabeçalho?

Imagine que agora Bob resolve alterar a frase do conteúdo do arquivo. Alice também altera o conteúdo em paralelo. Bob novamente faz o add, commit e push antes de Alice. Alice ao tentar fazer o push não conseguirá. Ela então irá fazer o pull para trazer as alterações. Porém, agora as alterações conflitam, eles alteraram as mesmas partes do arquivo: *Automatic merge faliled*.

Nesse momento o git não sabe como resolver o conflito existente, e Alice precisa resolver manualmente. Ela pode verificar o status, e saber qual arquivo apresenta o conflito. Depois disso, aplicar o diff. Abrir o arquivo e verificar o que foi inserido pelo usuário Bob, e o que foi inserido pelo usuário Alice. O que pode ser alterado por Alice, decidindo o que fica e o que deve ser alterado. Depois Alice add, faz o commit, indicando que houve um merge “realizando merge manual” e depois realizando o push.

6. Boas Práticas ao trabalhar em equipe

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 por Bob quanto por Alice, é criar uma branch local chamada "desenvolvimento" e utilizá-la. Isso pode ser feito com o comando:

```
git checkout -b desenvolvimento  // criando uma branch chamada desenvolvimento
```

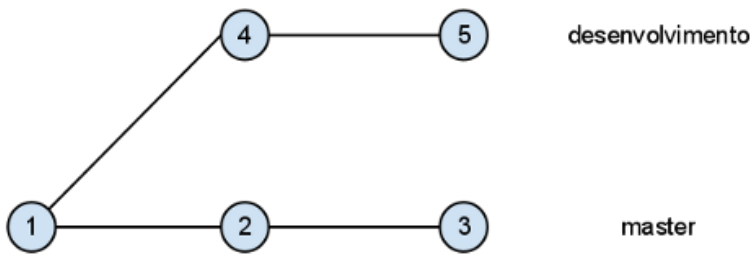
Nesse momento, a branch atual é trocada para a que acabamos de criar, ou seja, não está mais na branch master.

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.

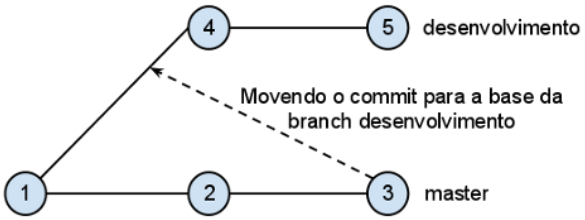
Envio dos commits das branches locais para o master remoto

Considere que o usuário Bob 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.

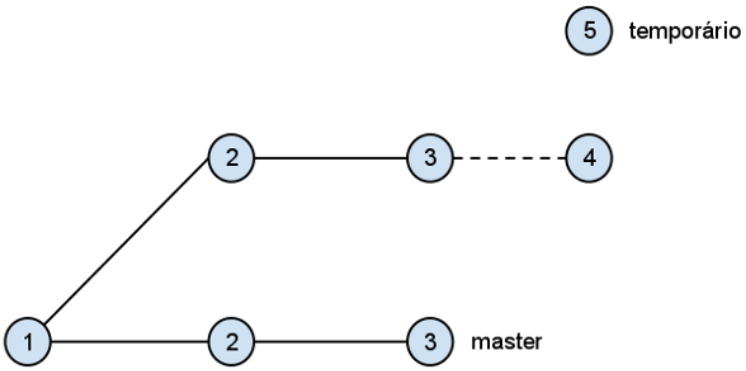
Temos a branch "desenvolvimento", que foi criada no commit 1. Porém, o último commit da master não é mais o commit 1 e sim, o 3.



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.

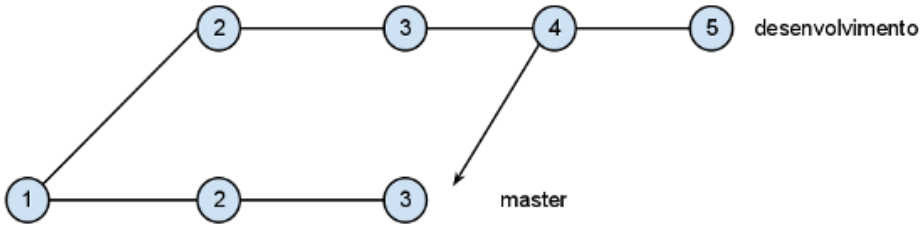


No caso, uma saída possível para a execução do git rebase master é a seguinte:

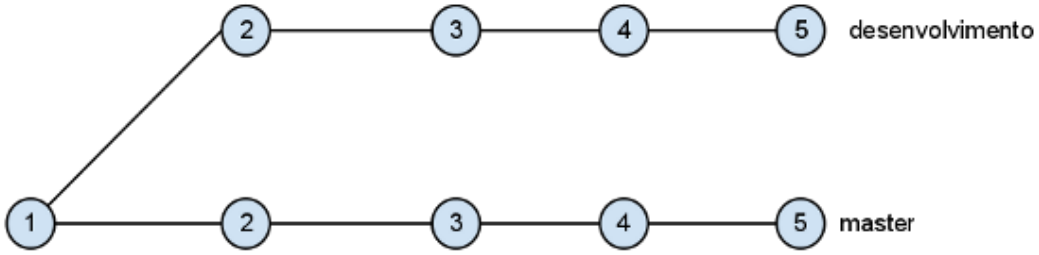
```
First, rewinding head to replay your work on top of it...
Applying: commit 1 de 2
Applying: commit 2 de 2
```

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, Bob e Alice, 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, Alice 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, Alice realiza o git push de suas alterações, enviando-as para o repositório remoto. Com isso, Bob 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 Alice 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 Bob, 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 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 Alice realizou. Para juntar as alterações que a Alice realizou com as suas, Bob 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 Bob 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 Bob 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 Bob realizou, adicionando o Copyright, conflitará com a alteração no rodapé que Alice 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 Bob para alterar a base de sua branch, utilizando os novos commits da outra usuária, Alice. Após a troca da base de commits, o primeiro commit realizado pelo Bob 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 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 Bob e manter o que a Alice fez com o comando git rebase --skip, excluindo esse último commit do rebase. Como não há mais nenhum commit que Bob 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).

7. Descartando alterações indesejadas

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 Bob decide alterar página, porém, por acidente, ele troca a "/" por "\" na Tag . 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 Bob 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 Bob criou a branch "desenvolvimento" e mudou para ela com o comando `git checkout -b desenvolvimento`.

Enquanto isso, a usuária Alice atualizou o repositório remoto modificando o arquivo "proposta_1.html". Ao realizar o git pull na branch master, o usuário Bob recebeu a atualização que a Alice realizou. Agora, ele deseja utilizar a versão apenas do arquivo "propsota_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 Bob 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 Bob 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 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, Bob 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 -am "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 Bob 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 -am "Alterando a tag div". Mas, ao ver a página, Bob 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 reset permite 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 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 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.

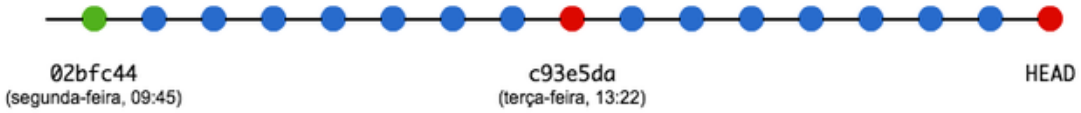
Agora precisamos marcar qual commit deve ser utilizado como estando OK:

```
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:

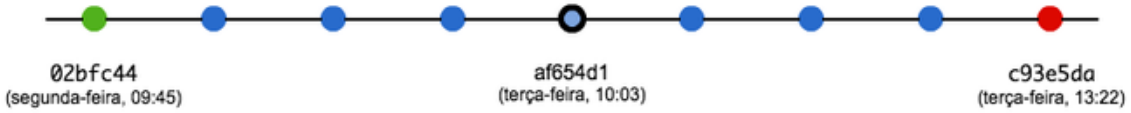
Podemos testar agora se, nesse ponto, nossa funcionalidade estava OK. Caso não esteja, precisamos marcar o commit atual como "ruim":

```
git bisect bad
```



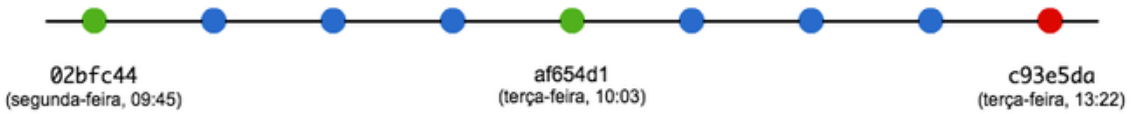
Por dedução lógica, todos os commits posteriores ao atual também estão defeituosos. Agora o Git conhece um novo intervalo de commits para buscar pelo erro, com metade do tamanho do intervalo anterior:

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

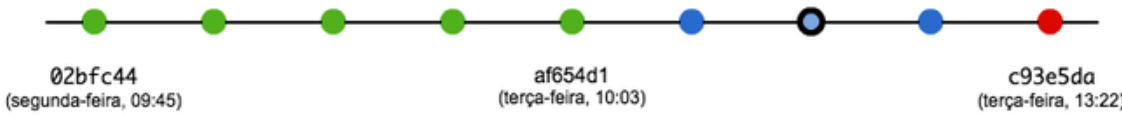


Novamente podemos testá-lo e verificar se ele contém o erro que procuramos. Digamos que esse commit está OK, então temos marca-lo como tal:

```
git bisect good
```



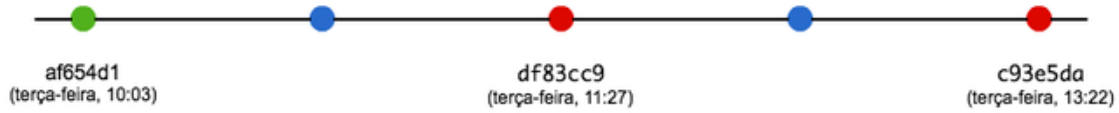
Utilizando o mesmo conceito de antes, o Git encontra um novo intervalo, sendo que agora temos o cenário inverso de antes: podemos procurar por commits posteriores ao atual.



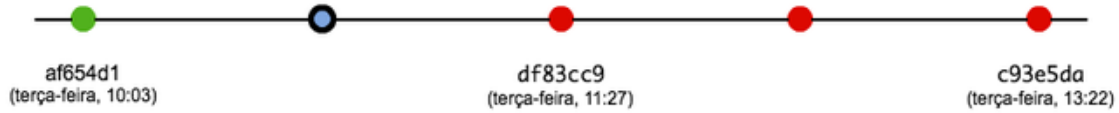
Novamente, continuamos o bisect a partir do commit intermediário do novo intervalo, por exemplo o commit df83cc9:

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.

8. Contribuindo com um projeto OpenSource

Para contribuir com um projeto opensource iresmos usar o recurso de Pull request. Imagine que você quer contribuir com algum projeto, mas não tem permissão para contribuir.

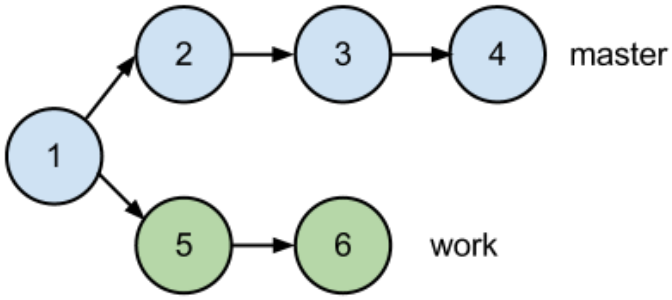
O primeiro passo é acessar o endereço do projeto e fazer uma cópia do repositório remoto para o seu repositório remoto, usando a opção `FORK`. Agora você tem uma cópia do projeto no seu repositório remoto. E pode fazer o clone do projeto para o repositório local, fazer pull, push, add, commit, como já vimos.

Após fazer a sua contribuição, você pode fazer o push para o seu repositório remoto. Em seguida, pode fazer o pull request, ou seja, fazer uma solicitação de pull para que o projeto pegue as alterações que você fez. Essa mensagem será enviada para algum responsável pelo repositório original.

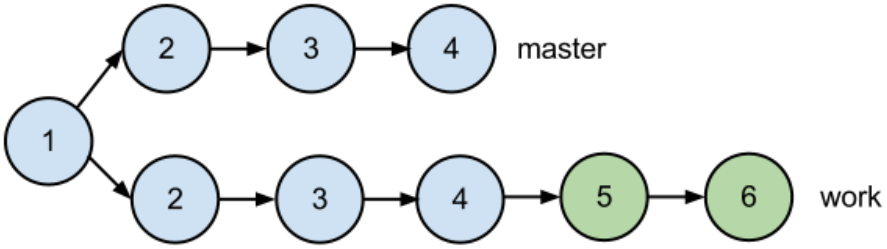
O usuário dono do projeto verifica o pull request e pode adicionar ou não aceitar a modificação, através da opção merge pull request.

9. 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:

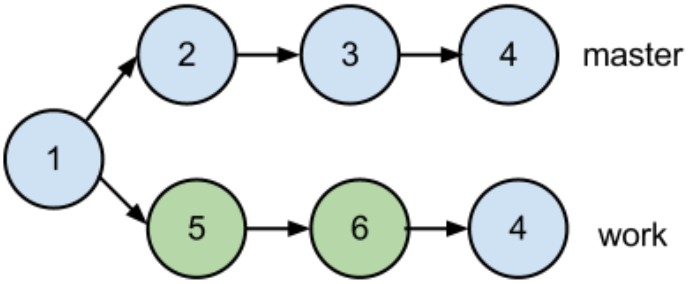


Agora imagine que nós só queiramos aplicar o commit 4 da branch master em nossa brach work pois ela introduziu algum trecho de código que nos ajudará em nossa tarefa. Para conseguir esse resultado, utilizaremos o comando git cherry-pick.

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).