Controle avançado de alterações

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 "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>
```

```
    <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,

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 d
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

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 -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

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 -am "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>
```

```
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
```

```
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 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

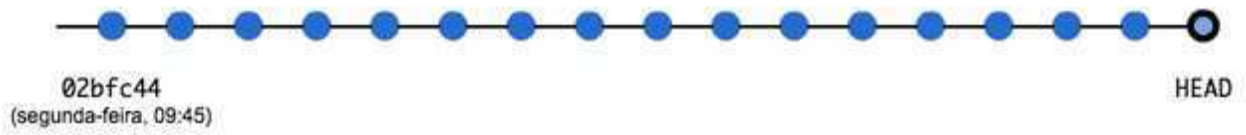
Quando conhecemos as alterações indesejadas, caso sejam poucas, é possível que façamos um novo commit com as alterações necessárias nos arquivos, desfazendo o que foi feito antes. Infelizmente, nem sempre temos um cenário tão simples, então precisamos utilizar um recurso mais avançado do Git.

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 o commit da segunda-feira que temos certeza que funciona:

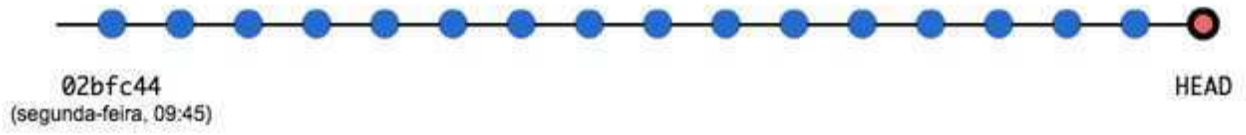
temos certeza que funciona.



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.