





GIT - Controle de Versão com Rapidez e Segurança

Autor: Lucas Medeiros de Freitas

Versão 1.12.4.6



Índice

1 - Sistema de Controle de Versão, uma breve explicação	1
1.1 Controle de Versão	1
1.1.1. O que é e por que usá-lo?	1
1.1.2. Como funciona?	1
1.2. Sistema de Controle de Versão Centralizado	2
1.3. Sistema de Controle de Versão Distribuído	5
2 - Introdução ao Git	9
2.1. Git, um SCM rápido e seguro	9
2.2. Projetos que usam o Git	9
2.3. Com o Git, tudo é local	10
2.4. O Git é rápido	10
2.5. O Git é facilmente adaptável ao seu fluxo de trabalho	11
2.6. Área intermediária	11
3 - Iniciando com o Git	12
3.1. Instalando o Git	12
3.1.1. Instalando o Git no Mac	12
3.1.2. Instalando o Git no Linux	14
3.1.2.1 <i>Instalando o Git em distribuições que usam o YUM</i>	14
3.1.2.2 <i>Instalando o Git em distribuições que usam o APT-GET</i>	14
3.1.2.3 <i>Instalando o Git a partir do source</i>	14

3.1.3. Instalando o Git no Windows	14
3.2. Configurando o Git	17
3.2.1. Exercício	17
3.3. Criando um repositório com o Git	17
3.3.1. Exercício	18
3.4. Realizando nosso primeiro commit	18
3.4.1. Exercício	20
4 - Trabalhando em Equipe	22
4.1. Hosterando nosso projeto no GitHub	22
4.1.1. Criando uma conta de usuário no GitHub	22
4.1.2. Criando um repositório no GitHub	24
4.1.3. Enviando nossas alterações ao repositório do GitHub	26
4.1.4. Exercício	27
4.2. Colaborando com o projeto	27
4.2.1. Adicionando um colaborador ao repositório do GitHub	27
4.2.2. Realizando um clone do repositório remoto	28
4.2.3. Enviando as alterações ao repositório remoto	30
4.2.4. Baixando as atualizações do repositório remoto	30
4.2.5. Resolução manual de conflitos	31
4.2.6. Git blame, o famoso dedo-duro	33
4.2.7. Exercício	33
5 - Recursos avançados do Git	34
5.1. Trabalhando com branches	34
5.1.1. Criando uma branch	34
5.1.2. Alterando entre branches	34

5.1.3. Trabalhando com a nova branch	35
5.1.4. Enviando as alterações para a branch master	39
5.1.5. Resolvendo conflitos no rebase	40
5.1.6. Enviando uma branch local para o repositório remoto	44
5.1.7. Baixando uma branch do repositório remoto para o repositório local	45
5.1.8. Exercício	45
5.2. Etiquetando nosso código com tags	47
5.2.1. Criando uma tag	47
5.2.2. Exercício	48
5.3. Usar o diff faz toda a diferença	49
5.3.1. Mostrando as alterações realizadas no working directory	49
5.3.2. Mostrando a diferença entre dois commits	49
5.3.3. Mostrando a diferença entre o commit atual e commits anteriores	50
5.3.4. Exercício	50
5.4. Desfazendo alterações	50
5.4.1. Descartando alterações no working directory	51
5.4.1.1. Exercício	52
5.4.2. Descartando alterações no Index	52
5.4.2.1. Exercício	53
5.4.3. Descartando alterações do último commit	53
5.4.3.1. Exercício	54
5.4.4. Descartando alterações de um commit antigo	54
5.4.4.1. Exercício	55
5.5. Procurando bugs em commits antigos	56
5.5.1. Exercício	58

5.6. Guardando alterações para depois	59
5.6.1. Exercício	60
6 - Ferramentas gráficas do Git	61
 6.1. Ferramentas gráficas para Mac	61
6.1.1. GitX	61
6.1.2. SmartGit	62
6.1.3. Tower	62
 6.2. Ferramentas gráficas para Linux	63
6.2.1. Git Cola	64
6.2.2. GitK	64
 6.3. Ferramenta gráfica para Windows	65
7 - Apêndice	67
 7.1. Montando um repositório centralizado	67
7.1.1. Montando um repositório centralizado no Mac e Linux	67
7.1.2. Montando um repositório centralizado no Windows	68

1 - Sistema de Controle de Versão, uma breve explicação

1.1 Controle de Versão

1.1.1. O que é e por que usá-lo?

Sistema de Controle de Versão, SCM (Source Code Management) ou VCS (Version Control System), é um sistema que registra alterações em um arquivo, ou em um conjunto de arquivos, ao longo do tempo, de forma que possamos voltar a uma versão específica mais tarde.

Embora seja mais usado para arquivos de código fonte de software, o Sistema de Controle de Versão pode ser usado para quase qualquer tipo de arquivo de computador.

O Sistema de Controle de Versão permite a um programador, por exemplo:

- Reverter seu código fonte a uma versão anterior;
- Comparar as alterações do código fonte ao longo do tempo;
- Manter um histórico de evolução do seu projeto;
- Saber quando e quem fez as alterações nos arquivos de código fonte;
- Procurar por um bug inserido no código fonte em um determinado período;
- Identificar as alterações feitas nos arquivos do projeto;
- Trabalhar em equipe de forma simples e prática;
- Criar ramificações do projeto com facilidade;
- Recuperar todos os arquivos do projeto em caso de perda dos arquivos por acidente.

1.1.2. Como funciona?

Os Sistemas de Controle de Versão trabalham de forma semelhante, possuindo um **repositório** para armazenar os arquivos e todo o histórico de evolução do projeto, e uma **área de trabalho** que nos permite trabalhar diretamente com os arquivos fazendo as alterações necessárias.

O **repositório** é como se fosse um banco de dados, que guarda todas as revisões do projeto, ou seja, todas as alterações feitas em seus arquivos bem como todo o histórico de evolução do projeto: quem fez, quando fez e o que fez.

A **área de trabalho** existe justamente para não trabalharmos diretamente nos arquivos armazenados no **repositório**, provendo uma cópia dos arquivos do projeto onde podemos alterá-los de forma segura, e estes são monitorados pelo SCM.

A **Figura 1** ilustra a comunicação entre a **área de trabalho** e o **repositório** de um SCM.

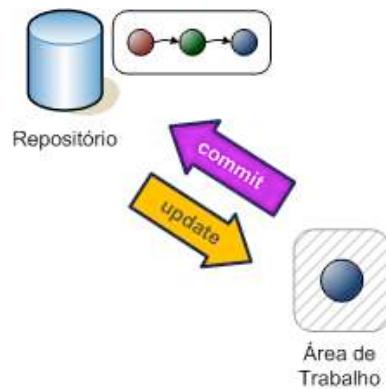


Figura 1. Funcionamento de um SCM.

Fonte: Pronus Engenharia de Software

Em um esquema lógico, um desenvolvedor realiza as alterações nos arquivos de código fonte diretamente na **área de trabalho** e, quando estas alterações estão maduras o suficiente, o desenvolvedor envia tais alterações para o **repositório**, criando assim uma nova revisão.

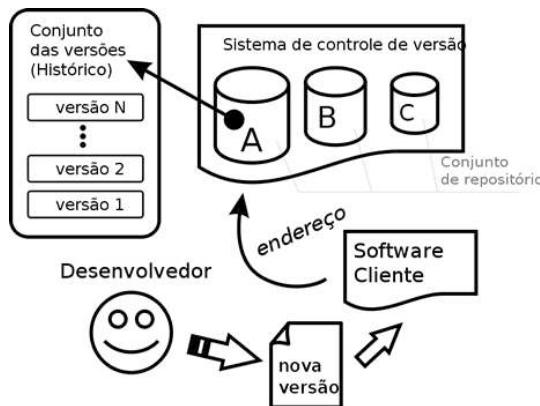


Figura 2. Esquema geral lógico de um SCM.

Fonte: Wikipédia

Cada revisão armazenada no **repositório** contém a parte sintética do projeto, ou seja, as alterações realizadas nos arquivos, e o histórico dessa revisão, com informações de quem fez tais alterações, quando foram feitas e o que foi feito.

Existem basicamente dois tipos de Sistemas de Controle de Versão: os **centralizados** e os **distribuídos**. Veremos nos próximos tópicos as diferenças entre os dois.

1.2. Sistema de Controle de Versão Centralizado

Um SCM centralizado trabalha de forma dependente de um servidor central. Neste modelo cada integrante de uma equipe de desenvolvedores tem sua própria **área de trabalho** em seu computador pessoal e o **repositório** fica armazenado em um servidor central. A sincronização entre o **repositório** e a **área de trabalho** só é possível quando se está conectado à rede e tem-se acesso a este servidor. Neste modelo toda a comunicação é feita através do repositório central, onde são realizados os **commits** e **updates**.

A **Figura 3** mostra o funcionamento de um SCM centralizado, ilustrando uma equipe de três desenvolvedores (Aline, Roberto e André) trabalhando em suas alterações, localmente em seus computadores, na **área de trabalho**, e sincronizando essas alterações com o **repositório central** através da rede.

Cada desenvolvedor pode enviar suas alterações para o **repositório central** através de um **commit** e baixar todas as atualizações feitas por outros desenvolvedores através de um **update**.

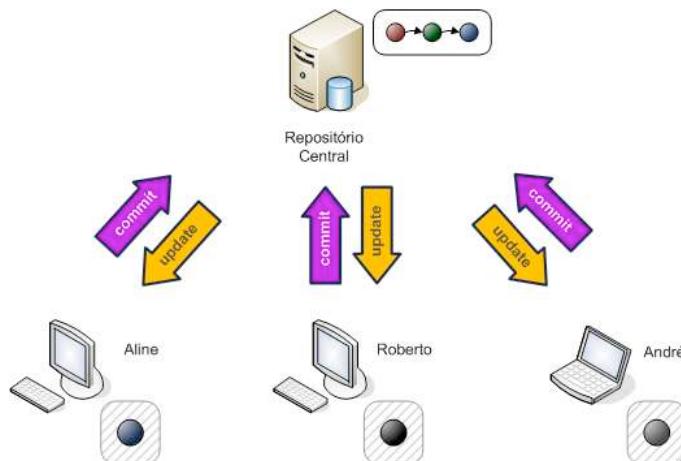


Figura 3. Funcionamento de um SCM centralizado.

Fonte: Pronus Engenharia de Software

Para obter uma cópia do projeto em um SCM centralizado, os desenvolvedores precisam realizar um **checkout**, como ilustra a **Figura 4**. Neste exemplo, os desenvolvedores Aline e Roberto obtém uma cópia do projeto e partem do mesmo ponto.

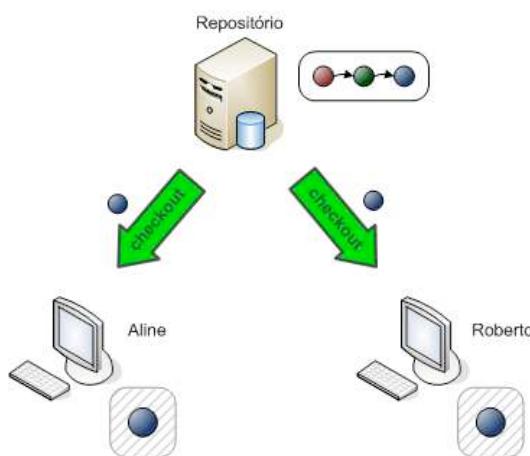


Figura 4. Desenvolvedores obtendo uma cópia do projeto através de um *checkout* no repositório central.

Fonte: Pronus Engenharia de Software

Após obterem uma cópia do projeto, Aline e Roberto trabalham em suas alterações. Então Aline finaliza suas alterações e as envia ao **repositório central** através de um **commit**, como ilustra a **Figura 5**.

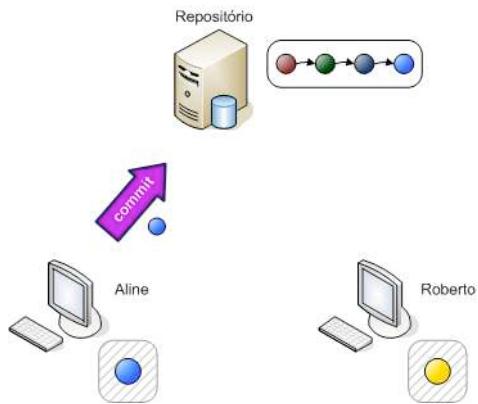


Figura 5. Ambos os desenvolvedores trabalham em modificações no projeto, mas Aline publica suas alterações primeiro.

Fonte: Pronus Engenharia de Software

Quando Roberto tenta enviar suas modificações, o servidor as recusa e informa que seus arquivos estão desatualizados, como ilustra a **Figura 6**.

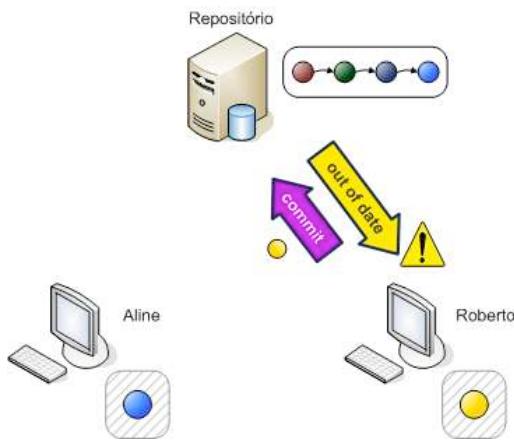


Figura 6. Roberto envia suas modificações logo após Aline ter feito um *commit*, e então o servidor recusa suas modificações.

Fonte: Pronus Engenharia de Software

Roberto precisa atualizar seu código para receber as atualizações de Aline, através de um **update**. Desta forma as atualizações de Aline são mescladas com o código que está em sua **área de trabalho**. Este processo de mesclagem é conhecido como **merge**.

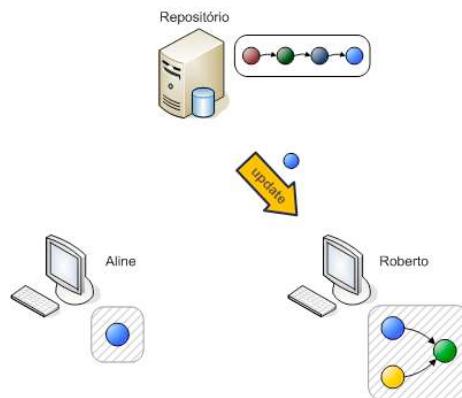


Figura 7. Roberto realiza um *update* para receber as atualizações de Aline.

Fonte: Pronus Engenharia de Software

Após realizar o **update**, Roberto obtém as novas atualizações do **repositório central**. Agora Roberto poderá enviar as suas modificações normalmente através de um **commit**.

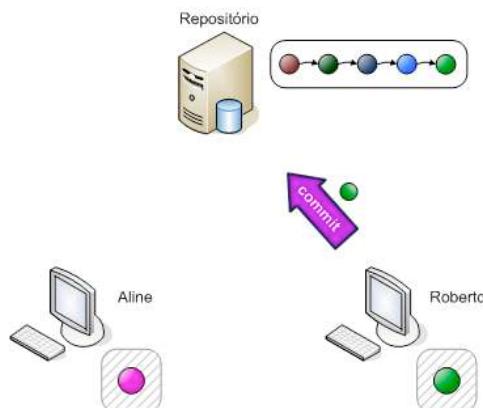


Figura 8. Após ter feito um *update*, Roberto estará apto a enviar suas modificações através de um *commit*.

Fonte: Pronus Engenharia de Software

Enquanto Roberto está atualizando seu código e enviando suas alterações para o **repositório central**, Aline trabalha em novas modificações no projeto, podendo enviá-las normalmente, caso não tenha alterado o mesmo arquivo que Roberto.

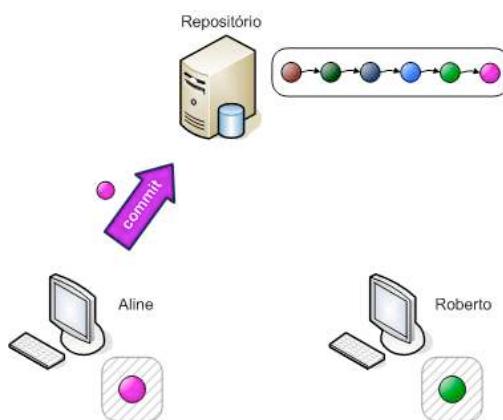


Figura 9. Aline envia suas alterações normalmente ao repositório central, já que estas não foram feitas no mesmo arquivo que Roberto modificou.

Fonte: Pronus Engenharia de Software

1.3. Sistema de Controle de Versão Distribuído

Em um SCM distribuído não existe **repositório central**. Cada desenvolvedor possui um **repositório** independente em seu computador local e uma **área de trabalho** para trabalhar diretamente nos arquivos. Desta forma todo o trabalho é feito localmente, não dependendo da rede para realizar **commits** e **updates**.

Como um SCM distribuído não depende da rede e todo o trabalho é feito localmente, o processo de sincronização com o **repositório** é muito mais rápido, pois não teremos que enfrentar os problemas de latência de rede, já que teremos um **repositório** em nosso sistema de arquivos local. Esta é uma grande vantagem dos SCM's distribuídos, pois podemos trabalhar desconectados. Imagine a comodidade e a praticidade de poder trabalhar em seu projeto e controlar sua versão enquanto está viajando em um avião, ou em um local que não possui acesso à rede. Isto aumenta muito a produtividade e a praticidade ao se trabalhar em um projeto versionado.

A **Figura 10** ilustra o funcionamento de um SCM distribuído.

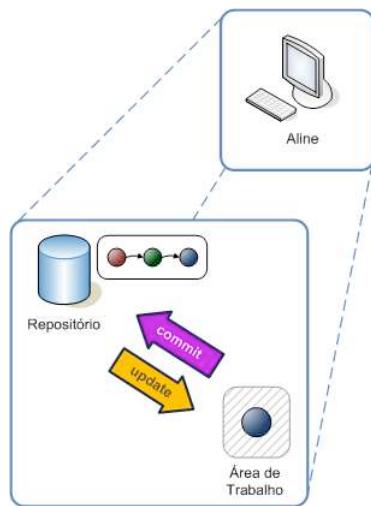


Figura 10. Funcionamento de um SCM distribuído. Neste modelo cada desenvolvedor possui um repositório independente e uma área de trabalho.

Fonte: Pronus Engenharia de Software

Mesmo sendo local o **repositório** em um SCM distribuído, podemos sincronizar facilmente com outros repositórios remotos na rede através dos comandos **pull** e **push**.

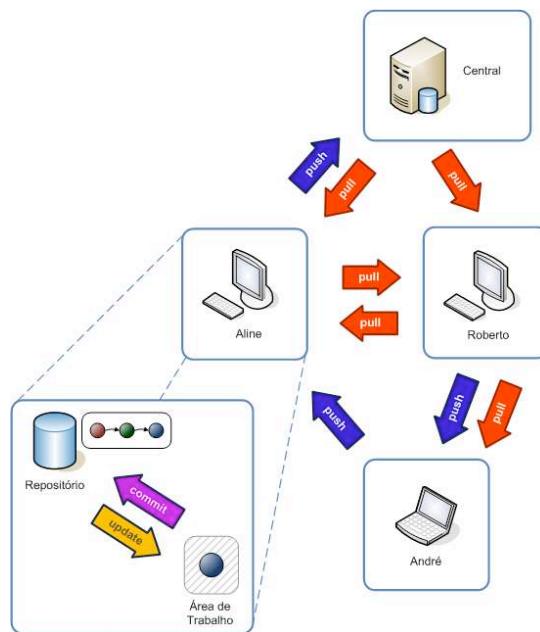


Figura 11. Podemos sincronizar nosso repositório com o de outros desenvolvedores em um SCM distribuído, ou até mesmo simular um repositório central.

Fonte: Pronus Engenharia de Software

Seguindo nosso exemplo, mas agora trabalhando em um SCM distribuído, o desenvolvedor Roberto realiza um **clone** do repositório de Aline e, a partir de então, ambos partem do mesmo ponto.

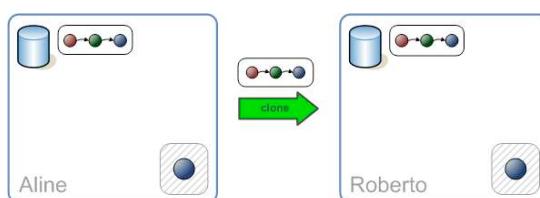


Figura 12. Roberto realiza um **clone** do repositório de Aline, obtendo uma cópia fiel de todo o conteúdo sintético e todo o histórico do projeto.

Fonte: Pronus Engenharia de Software

Aline e Roberto poderão trabalhar normalmente em seus repositórios locais, sem interferir no repositório um do outro.

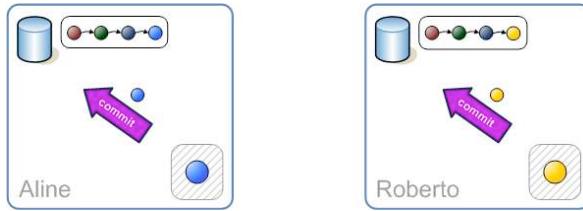


Figura 13. Aline e Roberto trabalham em novas modificações em seus repositórios locais, sem interferir no repositório um do outro.

Fonte: Pronus Engenharia de Software

Mesmo possuindo um repositório local, Roberto poderá sincronizar seu repositório com o de Aline, obtendo todas as modificações realizadas por ela, através de um comando **pull**.

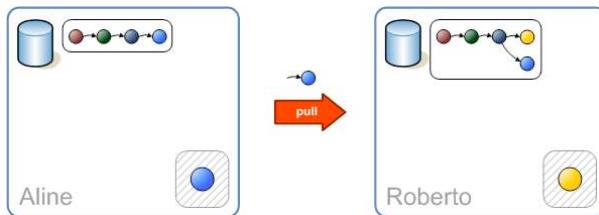


Figura 14. Roberto baixa as atualizações feitas por Aline, através de um comando *pull*.

Fonte: Pronus Engenharia de Software

Ao baixar as atualizações de Aline, estas serão mescladas no repositório local de Roberto através de um **merge**.

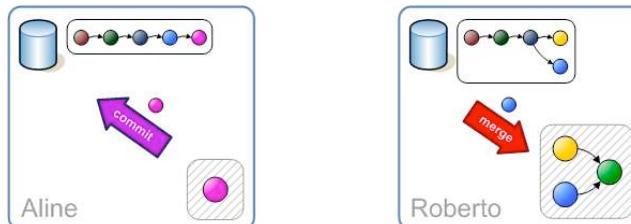


Figura 15. Um *merge* é feito no repositório local de Roberto, após baixar as atualizações do repositório da Aline.

Fonte: Pronus Engenharia de Software

Depois de ter baixado as atualizações de Aline, Roberto publica suas modificações em seu repositório local. Enquanto isso, Aline trabalha normalmente e publica novas alterações no repositório local dela.

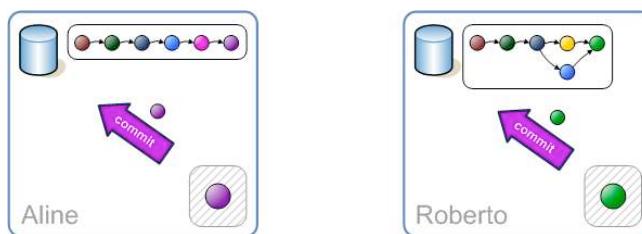


Figura 16. Aline e Roberto publicam alterações em seus repositórios locais.

Fonte: Pronus Engenharia de Software

GIT - Controle de Versão com Rapidez e Segurança

Finalmente Roberto envia suas atualizações para o repositório local de Aline através do comando **push**, mesclando suas atualizações no repositório local dela. Aline agora possuirá todas as alterações feitas por ela e por Roberto, bem como todo o histórico de alterações efetuadas por eles.

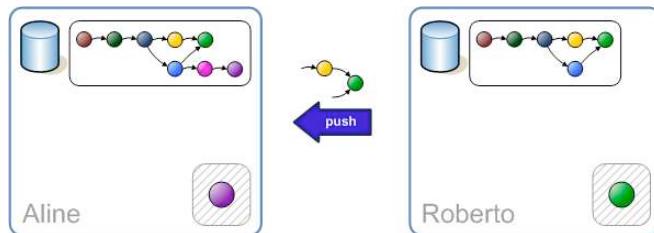


Figura 17. Roberto envia suas alterações para o repositório local de Aline, através de um *push*.

Fonte: Pronus Engenharia de Software

Como vimos, o Sistema de Controle de Versão distribuído tem uma vantagem muito grande por trabalhar desconectado, sem a necessidade de um servidor central. O Git é um SCM distribuído e iremos aprender mais sobre ele nos próximos tópicos.

2 - Introdução ao Git

2.1. Git, um SCM rápido e seguro

O Git é um SCM open source distribuído, moderno e poderoso, e vem sendo utilizado cada vez mais em grandes projetos no mundo inteiro.

O Git é um SCM muito rápido, foi criado por Linus Torvalds (pai do Linux e mantenedor do Kernel do Linux) e inicialmente foi projetado para trabalhar com o versionamento do Kernel do Linux, ou seja, desde seu princípio o Git já trabalha bem com sistemas muito grandes.

O Git surgiu a partir da necessidade de se ter um SCM capaz de acompanhar a velocidade com que o Kernel do Linux é desenvolvido. Antes do Git, o Kernel do Linux era versionado pelo BitKeeper, um SCM também distribuído porém comercial (pago) e, mesmo sendo um tanto rápido, o BitKeeper não era rápido o suficiente para acompanhar a velocidade com que o Kernel do Linux sofria alterações em seu código. Centenas de programadores contribuem com o Kernel do Linux no mundo inteiro e muitas alterações são feitas todos os dias, sendo assim apenas um SCM distribuído muito rápido para atender a este cenário. A partir desta necessidade, Linus Torvalds criou um SCM distribuído, chamado Git, que se demonstrou tão rápido que ganhou a atenção de muitos desenvolvedores e se tornou o SCM oficial do Kernel do Linux.

Hoje o Git é mantido por Junio Hamano, responsável pela evolução deste incrível SCM.

Para quem está acostumado com SCM's centralizados, como o CVS e o Subversion, e quer iniciar com o GIT, é preciso ter em mente algumas diferenças cruciais:

- Primeiro, o GIT não possui um servidor central;
- Segundo, o GIT não possui um servidor central.

Em SCMs distribuídos, todos os usuários possuem um clone do repositório original com o histórico completo do projeto, e essa é a essência dos SCMs distribuídos.

O GIT não possui um sistema de autenticação próprio. Isto é uma vantagem, pois você pode usar o sistema de autenticação de acordo com o protocolo que escolher ao se trabalhar com sincronização remota de repositórios.

2.2. Projetos que usam o Git

Atualmente vários projetos grandes e conhecidos usam o Git como SCM. Podemos citar abaixo alguns projetos importantes que usam o Git:

- O próprio Git;
- Kernel do Linux;
- Perl;
- Eclipse;
- Gnome;
- KDE;

- Ruby;
- Android;
- PostgreSQL;
- Debian Linux;
- X.org.

.. e inúmeros outros.

O Git trabalha bem com projetos grandes, garantindo segurança, performance e incríveis recursos para o controle de versão.

2.3. Com o Git, tudo é local

Com o Git, tudo é feito localmente em seu sistema de arquivos. Isso torna todo o processo de versionamento muito mais produtivo, permitindo que *commits* sejam feitos totalmente desconectado da rede. Imagine a praticidade de poder trabalhar em seu projeto, criar *branches*, fazer *merges* e *commits* e poder listar todo o histórico do seu projeto enquanto estiver em viagem, em um vôo. Isso é muito produtivo.

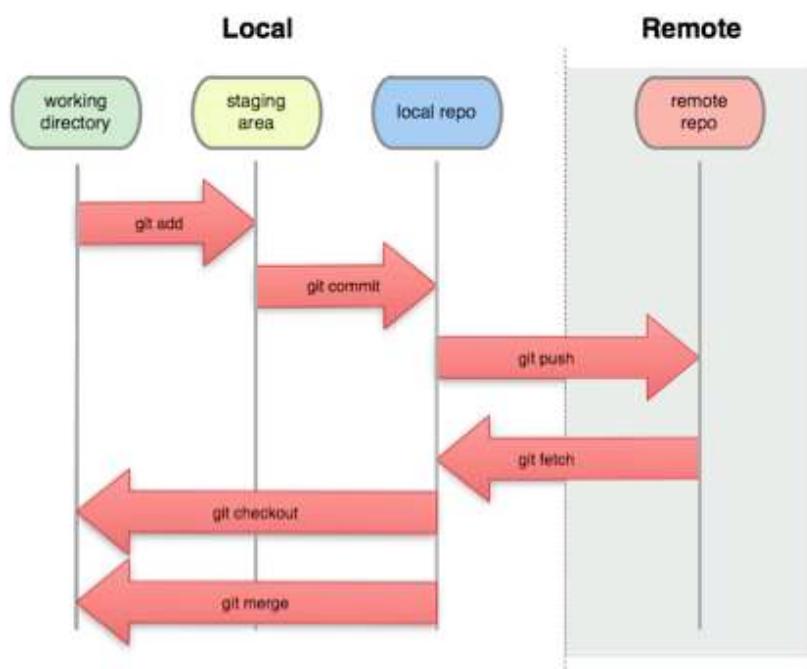


Figura 18. Todo o processo de versionamento é feito localmente com o Git, podendo sincronizar com repositórios remotos apenas quando necessário.

Fonte: whygitisbetterthanx.com

2.4. O Git é rápido

Em comparação com SCMs centralizados como o CVS e SVN, obviamente o Git é muito mais rápido por trabalhar localmente. Mas em comparação com outros SCMs distribuídos, como o Mercurial e o Bazaar, o Git também se demonstra muito mais rápido. Isto deve-se pelo fato de o Git ser totalmente escrito em C e seus desenvolvedores terem tido toda uma preocupação com performance desde seu início, já que desde seu nascimento o Git já trabalhava com repositórios enormes como o do Kernel do Linux.



Figura 19. O Git possui um tempo de resposta muito menor em diversas tarefas, comparado com outros SCM's distribuídos.

Fonte: whygitisbetterthanx.com

2.5. O Git é facilmente adaptável ao seu fluxo de trabalho

O Git é facilmente adaptável ao seu Fluxo de Trabalho. Se sua equipe tem a necessidade de ter um Repositório Central, por exemplo, mesmo sendo um SCM Distribuído, é possível criar um Repositório Central com o Git.

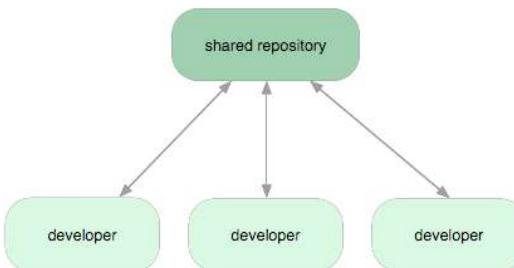


Figura 20. Exemplo de um fluxo de trabalho usando um repositório central com o Git.

Fonte: whygitisbetterthanx.com

2.6. Área intermediária

O Git possui uma área *intermediária* (*Index*), onde podemos decidir o que realmente vai ser commitado. A vantagem é que podemos adicionar apenas os arquivos desejados na área *intermediária* e fazer o *commit* apenas destes arquivos, sem a necessidade de fazer o *commit* de tudo o que foi modificado na área de *trabalho*.

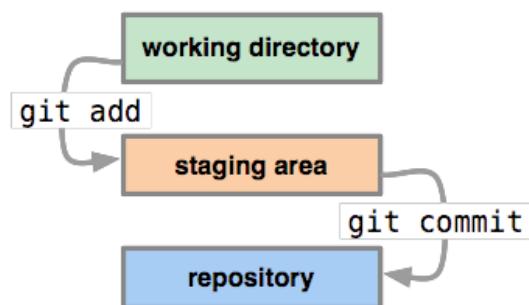


Figura 21. Área intermediária do Git, uma grande vantagem em relação a outros SCM's.

Fonte: whygitisbetterthanx.com

3 - Iniciando com o Git

3.1. Instalando o Git

O Git é fácil de ser instalado e possui suporte a diversos Sistemas Operacionais. Veremos nos próximos tópicos como instalar o Git nos três sistemas operacionais mais conhecidos: Mac OSX, Linux e Windows.

3.1.1. Instalando o Git no Mac

O Git possui um instalador para usuários Mac. Para instalá-lo, basta seguir os passos abaixo:

Entre no site <http://code.google.com/p/git-osx-installer/downloads/list?can=3> e baixe a última versão do instalador do Git para Mac, de acordo com a versão do seu Sistema Operacional.

The screenshot shows the 'Downloads' tab of the git-osx-installer project page. It lists several DMG files for different OS X versions and architectures. The files are ordered by filename, with the latest version at the top: 'git-1.7.9.4-intel-universal-snow-leopard.dmg' (Git Installer 1.7.9.4 - OS X - Snow Leopard - Intel Universal), followed by 'git-1.7.9.3-intel-universal-snow-leopard.dmg', 'git-1.7.9.1-intel-universal-snow-leopard.dmg', 'git-1.7.8.4-intel-universal-snow-leopard.dmg', 'git-1.7.8.3-intel-universal-snow-leopard.dmg', 'git-1.7.8.2-intel-universal-snow-leopard.dmg', 'git-1.7.7.3-intel-universal-snow-leopard.dmg', 'OpenInGitX-1.2.zip', and 'OpenInGitGui-2.1.zip'. Each file entry includes a download link, file size, and a 'Featured' badge.

Figura 22. Selecione a última versão da instalação do Git para download, de acordo com a versão do seu OSX.

The screenshot shows the detailed view for the 'git-1.7.9.4-intel-universal-snow-leopard.dmg' file. It displays the file's metadata: uploaded by 'lmchar...@gmail.com' on March 19, 2012, with 21593 downloads and the 'Featured' badge. Below this, there is a QR code. On the right side of the page, a Mac OS X 'Activity Monitor' window is open, showing a download progress bar for 'git-1.7.9.4-intel-universal-snow-leopard.dmg' at 9% completion.

Figura 23. Clique no link contendo o arquivo de instalação para baixar o instalador do Git.

GIT - Controle de Versão com Rapidez e Segurança

Após ter feito o download do instalador, basta abri-lo para iniciar a instalação. Na tela inicial do instalador, clique em “Continuar” para prosseguir com a instalação.



Figura 24. Tela inicial do instalador do Git.

Clique em instalar para iniciar a instalação do Git.



Figura 25. Tela do instalador do Git.

Após instalado, clique em “Fechar” para encerrar o instalador do Git.



Figura 26. Tela do instalador do Git.

Agora o Git está instalado e pronto para ser usado em seu Mac.

3.1.2. Instalando o Git no Linux

Instalar o Git no Linux é simples. Basta usar o gerenciador de pacotes de acordo com a sua distribuição.

3.1.2.1 Instalando o Git em distribuições que usam o YUM

Em distribuições do Linux que usam o YUM como gerenciador de pacotes, basta executar o seguinte comando (*como root*) para instalar o Git:

```
yum install git-core
```

3.1.2.2 Instalando o Git em distribuições que usam o APT-GET

Em distribuições do Linux que usam o APT-GET como gerenciador de pacotes, basta executar o seguinte comando (*como root*) para instalar o Git:

```
apt-get install git-core
```

3.1.2.3 Instalando o Git a partir do source

Outra opção para instalar o Git no Linux é a partir do source. Baixe o source do Git no link:

<http://git-scm.com/download>

Após baixar o source do Git, basta descompactá-lo e, em seu diretório, executar os seguintes comandos (*como root*) para instalar:

```
make prefix=/usr all
```

```
make prefix=/usr install
```

3.1.3. Instalando o Git no Windows

Para instalar o Git no Windows, baixe a última versão do **msysgit** no link: <http://code.google.com/p/msysgit/downloads/list>

Ao abrir o instalador, clique em *Next*.



Figura 27. Tela inicial do instalador do Git.

GIT - Controle de Versão com Rapidez e Segurança

Na tela de licença, clique em *Next*.

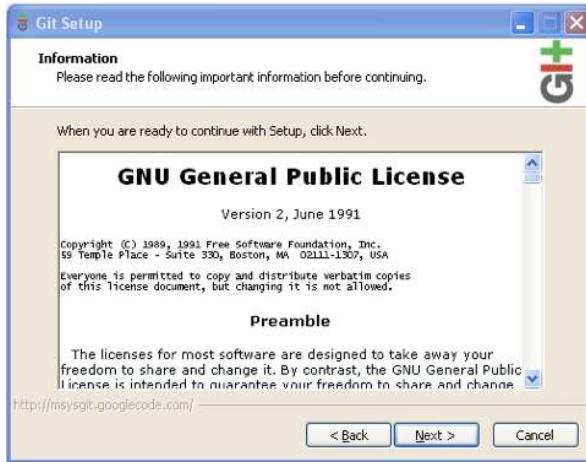


Figura 28. Tela do instalador do Git.

Escolha o diretório onde o Git deverá ser instalado e clique em *Next*.

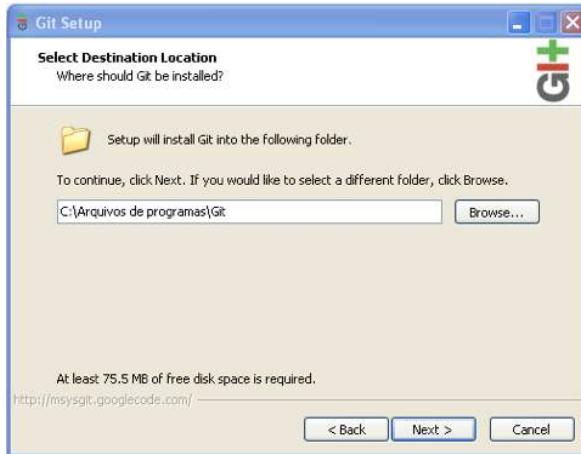


Figura 29. Tela do instalador do Git.

Na tela seguinte, deixe marcadas as opções padrões e clique em *Next*.

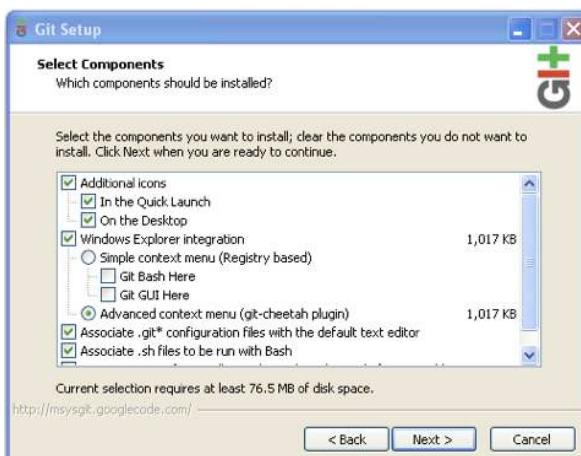


Figura 30. Tela do instalador do Git.

GIT - Controle de Versão com Rapidez e Segurança

Na próxima tela, defina como deve aparecer o nome do Git no menu Iniciar do Windows. Deixe o padrão, “Git”, e clique em *Next*.



Figura 31. Tela do instalador do Git.

Na próxima tela, devemos definir como devemos usar o Git no Windows. Marque a opção “Run Git from de Windows Command Prompt” e clique em *Next*, conforme imagem a seguir.



Figura 32. Tela do instalador do Git.

Na próxima tela, deixe marcado a primeira opção e clique em *Next*, conforme imagem a seguir.

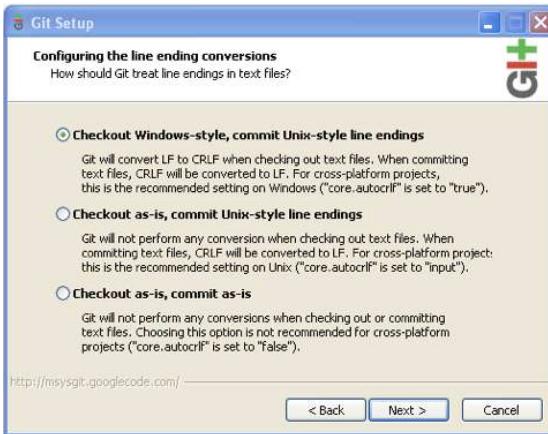


Figura 33. Tela do instalador do Git.

Uma mensagem de sucesso da instalação irá aparecer, aperte *Finish*!



Figura 34. Tela do instalador do Git.

3.2. Configurando o Git

Antes de começar a utilizar o Git, alguns ajustes precisam ser feitos. Precisamos definir o nome do nosso usuário e um e-mail, que serão usados para identificar as alterações que serão realizadas em nosso repositório.

Para que esta configuração seja feita, as variáveis globais **user.name** e **user.email** do Git precisam ser definidas.

3.2.1. Exercício

Vamos configurar as variáveis globais de usuário e e-mail para utilizar o Git:

1. Abra o terminal em seu Sistema Operacional e digite o comando:

git config --global user.name “<NOME DO SEU USUÁRIO>”

substituindo **<NOME DO SEU USUÁRIO>** pelo nome de usuário que deseja utilizar.

2. Após definir o usuário, ainda no terminal, defina o seu e-mail digitando o comando:

git config --global user.email “<SEU ENDEREÇO DE EMAIL>”

substituindo **<SEU ENDEREÇO DE EMAIL>** pelo seu endereço de e-mail.

3. Verifique as configurações realizadas com o comando: **git config -l**

3.3. Criando um repositório com o Git

Criar um repositório é uma tarefa simples com o Git. Um simples comando "**git init**" inicia um repositório do Git no diretório atual. Veja o exemplo na **Figura 35**, onde criamos um repositório do Git em um diretório chamado "**curso-git**".

A screenshot of a Mac OS X terminal window titled "curso-git – bash – 134x12". The window contains a command-line session. The user creates a directory named "curso-git" in their home folder, changes into it, and initializes a Git repository with the command "git init". Finally, they list the files in the directory with "ls -lahr".

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento ]# mkdir curso-git
[lucasfreitas@MacBook-de-Lucas:~/treinamento ]# cd curso-git
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ]# git init
Initialized empty Git repository in /Users/lucasfreitas/treinamento/curso-git/.git/
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master #)]# ls -lahr
total 0
drwxr-xr-x 12 lucasfreitas staff 408B 6 Abr 21:18 ..
drwxr-xr-x 10 lucasfreitas staff 340B 6 Abr 21:18 .git
drwxr-xr-x  3 lucasfreitas staff 102B 6 Abr 21:18 .
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master #)]#
```

Figura 35. Iniciando um repositório com o Git.

No exemplo acima criamos um repositório do Git no diretório “**curso-git**” e logo em seguida listamos seus arquivos. Perceba que um diretório “**.git**” foi criado no diretório raiz do nosso repositório.

#Fica a dica

- Quando se cria um repositório com o Git, apenas um diretório chamado “**.git**” é criado na raiz do projeto e este diretório conterá todo o histórico de controle de versão do nosso projeto. Isto é uma grande vantagem do Git em comparação a outros SCM's como o CVS, por exemplo, que cria um diretório “**.cvs**” no diretório raiz e em todos os sub-diretórios, poluindo muito o diretório do nosso projeto. Como apenas um diretório “**.git**” é criado, basta deletá-lo para deixar de usar o Git como controle de versão do nosso projeto.

3.3.1. Exercício

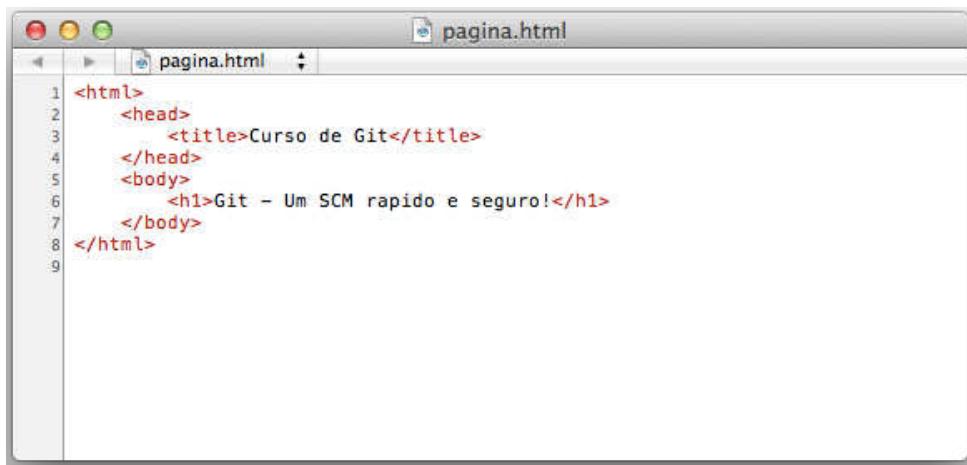
Agora que você aprendeu a criar um repositório do Git, vamos criar um repositório chamado “**curso-git**”, que será usado no decorrer do nosso curso.

1. Crie um diretório chamado “**curso-git**” em seu sistema de arquivos local;
2. Abra o terminal do seu Sistema Operacional e entre no diretório “**curso-git**” que foi criado;
3. Execute o comando: **git init**;
4. Liste os arquivos deste diretório e perceba o diretório “**.git**” que foi criado.

3.4. Realizando nosso primeiro commit

Agora que nós já temos um repositório, podemos criar um projeto dentro dele e ter seus arquivos monitorados pelo Git. Para o nosso curso, iremos utilizar um projeto simples: uma página html.

No diretório onde o repositório foi criado, criamos um arquivo **pagina.html** e definimos apenas um título e um cabeçalho para esta página:

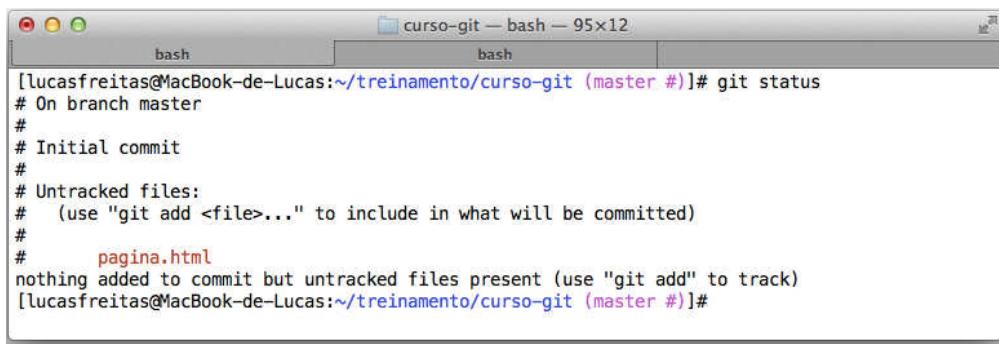


The screenshot shows a text editor window titled "pagina.html". The code inside the file is as follows:

```
1 <html>
2   <head>
3     <title>Curso de Git</title>
4   </head>
5   <body>
6     <h1>Git - Um SCM rapido e seguro!</h1>
7   </body>
8 </html>
```

Figura 36. Conteúdo inicial da nossa página

Após criar o arquivo **pagina.html** (nossa primeira arquivo do projeto) e salvá-lo, executamos o comando **git status** e percebemos a seguinte saída:



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master #)]# git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       pagina.html
nothing added to commit but untracked files present (use "git add" to track)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master #)]#
```

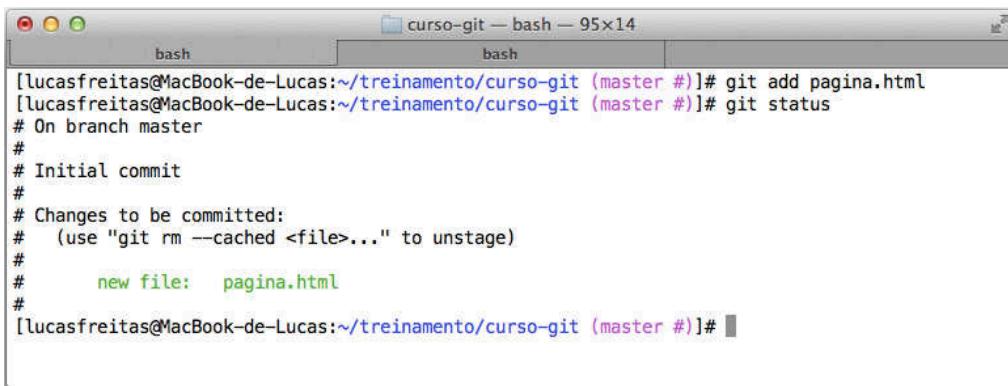
Figura 37. Saída do comando `git status`

A saída do comando **`git status`** nos informou que existe um novo arquivo no repositório (**`página.html`**), porém este arquivo ainda não está trackeado pelo Git. Um arquivo não trackeado (*untracked*) é um arquivo que está apenas na área de trabalho (*working directory*), onde trabalhamos diretamente com o arquivo. Precisamos então adicionar este arquivo na área intermediária (*staging area ou Index*) antes de fazer um *commit*.

#Fica a dica

- O comando **`git status`** é muito útil para verificarmos o estado dos nossos arquivos na *branch* atual. Ele nos informa se há alterações a serem inseridas na *staging area* ou se há arquivos na *staging area* prontos para serem *commitados*.

Para adicionar nosso arquivo `página.html` na área intermediária, usamos o comando **`git add`**:



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master #)]# git add página.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master #)]# git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   página.html
#
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master #)]#
```

Figura 38. Adicionando o arquivo `página.html` na *staging area*

Ao executar o comando **`git add página.html`** adicionamos o arquivo **`página.html`** na área intermediária. Com o comando **`git status`**, logo em seguida, podemos verificar que agora nosso arquivo `página.html` está trackeado, ou seja, ele está no *Index* (*staging area*) e pronto para ser adicionado definitivamente no repositório, em nossa *branch master*.

#Fica a dica

- Uma **`branch`** é uma ramificação do projeto, um espaço contendo uma cópia dos arquivos do repositório. Ao criar um repositório com o Git, automaticamente uma *branch* chamada *master* é criada e esta será a *branch* principal do projeto. Com o Git podemos ter quantas *branches* desejarmos. Aprenderemos mais sobre *branch* em outro capítulo.

Agora que nosso arquivo **`página.html`** está trackeado (adicionado ao *Index*), podemos realizar o *commit*, usando o comando **`git commit`**.

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git commit -m "Criando nossa primeira pagina"
[master (root-commit) 413271b] Criando nossa primeira pagina
 1 files changed, 8 insertions(+), 0 deletions(-)
  create mode 100644 pagina.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git status
# On branch master
nothing to commit (working directory clean)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 39. Realizando nosso primeiro *commit*

Perceba que ao executarmos o comando **git commit**, o Git emite uma saída nos informando que o arquivo **pagina.html** foi adicionado ao repositório, criando uma nova revisão. Após realizar o *commit*, o **git status** informa que nosso *working directory* está limpo, sem nenhum *commit* a ser feito.

Podemos verificar o Log de commits com o comando **git log**:

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

  Criando nossa primeira pagina
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 40. Verificando o Log de commits com o comando *git log*.

#Fica a dica

- O comando **git log** mostra o Log de todos os commits realizados na *branch* atual. O Log informa o autor do commit, a data e hora em que o commit foi realizado e a mensagem do commit. Perceba no Log que cada commit é identificado por uma hash SHA-1 de 40 caracteres hexadecimais. Diferente de SCM's centralizados, onde cada commit recebe um número como identificador (sendo que o commit mais recente sempre terá um número maior que os anteriores), em SCM's distribuídos, como o Git, cada commit é identificado por uma hash única.

Sempre que executamos um *commit*, uma mensagem deve ser informada identificando-o. Veja na **Figura 39** onde executamos o comando **git commit -m "Criando nossa primeira pagina"**. A flag **-m** indica que o conteúdo em seguida é a mensagem do *commit*. Caso a flag **-m** seja omitida, o próprio Git irá abrir um editor de texto pedindo-o para digitar uma mensagem para o *commit* a ser realizado. Informar uma mensagem/comentário em cada *commit* é muito importante para manter um histórico de evolução do código e facilita a leitura do que foi alterado naquele *commit* para todos os colaboradores do projeto.

3.4.1. Exercício

Agora que você aprendeu a fazer um *commit*, vamos colocar em prática.

- Crie um arquivo chamado **pagina.html** no diretório **curso-git** (onde foi criado o repositório) com o seguinte conteúdo:

```
<html>
  <head>
    <title>Curso de Git</title>
  </head>
  <body>
    <h1>Git - Um SCM rapido e seguro!</h1>
  </body>
</html>
```

2. Salve o arquivo e verifique o status de alterações no *working directory* com o comando **git status**.
3. Adicione o arquivo criado no *staging area* com o comando **git add pagina.html**.
4. Execute novamente o comando **git status** e perceba as alterações.
5. Faça o *commit* com o comando **git commit -m “Criando nossa primeira pagina”**.
6. Verifique o Log do *commit* realizado, com o comando **git log**.
7. Abra o arquivo **página.html** em seu browser e verifique seu conteúdo.

4 - Trabalhando em Equipe

Agora que já sabemos criar um repositório do Git e realizar commit, iremos trabalhar de forma colaborativa. Apesar de ser um SCM distribuído, com o Git podemos facilmente disponibilizar nosso repositório em um servidor central para que outros desenvolvedores possam colaborar com nosso projeto.

Em nosso curso iremos utilizar o GitHub para armazenar nosso repositório do Git. O GitHub é uma ferramenta na Web que nos possibilita compartilhar nosso projeto de forma rápida e possui muitos recursos que facilitam a utilização do Git. GitHub é praticamente uma rede social de desenvolvedores para projetos que usam o Git, permitindo que usuários do mundo inteiro possam ter acesso aos repositórios publicados.

O cadastro no GitHub é gratuito e você pode armazenar seus repositórios sem nenhum custo, desde que eles sejam open source. No GitHub é possível ter repositórios privados também. Os repositórios privados são pagos, porém seu custo é baixo: é possível ter repositórios privados a partir de 7 dólares por mês.

Utilizaremos então um repositório open source, sem custo mensal.

4.1. Hospedando nosso projeto no GitHub

Agora que já sabemos criar um repositório local e criar revisões com commits, iremos criar um repositório remoto no GitHub e permitir que outros desenvolvedores possam colaborar com nosso projeto.

Para isto, precisamos criar um usuário no GitHub.

4.1.1. Criando uma conta de usuário no GitHub

No site www.github.com basta clicar no link “Plans, Pricing and Signup”, localizado na página inicial. Uma página com os planos do GitHub irá se abrir. Basta clicar no link “Create a free account” para criar uma conta gratuita, sem custos mensais.

The screenshot shows the GitHub 'Plans & Pricing' page. At the top, there's a banner for the 'Free for open source' plan, which is \$0/mo and includes unlimited public repositories and unlimited public collaborators. A 'Create a free account' button is located in the top right of this banner. Below the banner, there are three paid plan options: Micro (\$7/mo), Small (\$12/mo), and Medium (\$22/mo). Each plan has its own section with a summary of features and a 'Create an account' button. The Micro plan includes 5 private repositories and 1 private collaborator, with unlimited public repositories and collaborators. The Small plan includes 10 private repositories and 5 private collaborators, with unlimited public repositories and collaborators. The Medium plan includes 20 private repositories and 10 private collaborators, with unlimited public repositories and collaborators.

Figura 41. Clique no link “Create a free account” para criar uma conta gratuita no GitHub.

GIT - Controle de Versão com Rapidez e Segurança

Na tela de cadastro, basta preencher um nome de usuário, endereço de e-mail e senha da conta, e clicar no link “Create an account”, conforme a **Figura 42**.

The screenshot shows the GitHub sign-up page at https://github.com/signup/free. At the top, it says "Sign up for GitHub". A yellow banner indicates a free plan with "\$0/mo" and a note about the cost being \$0 per month. Below this, there's a section to "Create your free personal account" with fields for "Username" (lucas-hachitecnologia), "Email Address" (lucas@hachitecnologia.com.br), "Password" (a masked password), and "Confirm Password" (also a masked password). To the right, there's a sidebar with social media links for Facebook, Twitter, Mozilla, and 37signals, and a message: "You're joining the smartest companies in the world". Below the sidebar, there's a list of benefits with checkmarks: "Email support", "Upgrade, downgrade or cancel at any time", and "Secure, reliable, always-available repository hosting". At the bottom left, a note states: "By clicking on "Create an account" below, you are agreeing to the Terms of Service and the Privacy Policy." A green "Create an account" button is at the bottom right.

Figura 42. Formulário para cadastro de conta no GitHub.com.

Precisamos agora gerar uma chave de segurança e cadastrá-la no GitHub para que ele autorize nossa máquina a gerenciar nossos repositórios remotos. Para isto, basta entrar no terminal (para usuários Windows basta abrir o Git Bash) e digitar o seguinte comando:

```
ssh-keygen -t rsa -C "<SEU ENDEREÇO DE E-MAIL>"
```

Substitua **<SEU ENDEREÇO DE E-MAIL>** pelo seu endereço de e-mail.

Após executar o comando acima, o gerador de chaves irá perguntar em qual arquivo deseja gravar a chave gerada. Basta pressionar **<Enter>** para que a chave seja gravada no local padrão, no arquivo **id_rsa** dentro do diretório padrão do seu usuário. Uma senha e sua confirmação também serão solicitados, bastando pressionar **<Enter>** nessas duas etapas, pois não iremos travar nossa chave com senha. Veja todo este processo na **Figura 43**.

The screenshot shows a terminal window titled "curso-git" with the command "ssh-keygen -t rsa -C "lucas@hachitecnologia.com.br"" being run. The output shows the generation of a public/private key pair, saving the private key to "/Users/lucasfreitas/.ssh/id_rsa" and the public key to "/Users/lucasfreitas/.ssh/id_rsa.pub". It also displays the key fingerprint and the randomart image. The terminal prompt "[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# " is visible at the bottom.

Figura 43. Gerando uma chave de segurança RSA para cadastrarmos no GitHub.com.

GIT - Controle de Versão com Rapidez e Segurança

Com a nossa chave de segurança gerada, precisamos cadastrá-la no GitHub. Para isto, com seu usuário logado, vá na sessão “Account Settings” do GitHub e entre no link “SSH Keys” no menu. Agora basta clicar no link “Add SSH Key”. Uma página de cadastro da chave de segurança irá se abrir. Informe um título para a chave e copie o conteúdo do arquivo “**id_rsa.pub**” do diretório “**.ssh**” (localizado no diretório home do seu usuário em seu computador) e cole no campo “Key” do formulário. Clique em “Add key” para concluir o cadastro da chave.

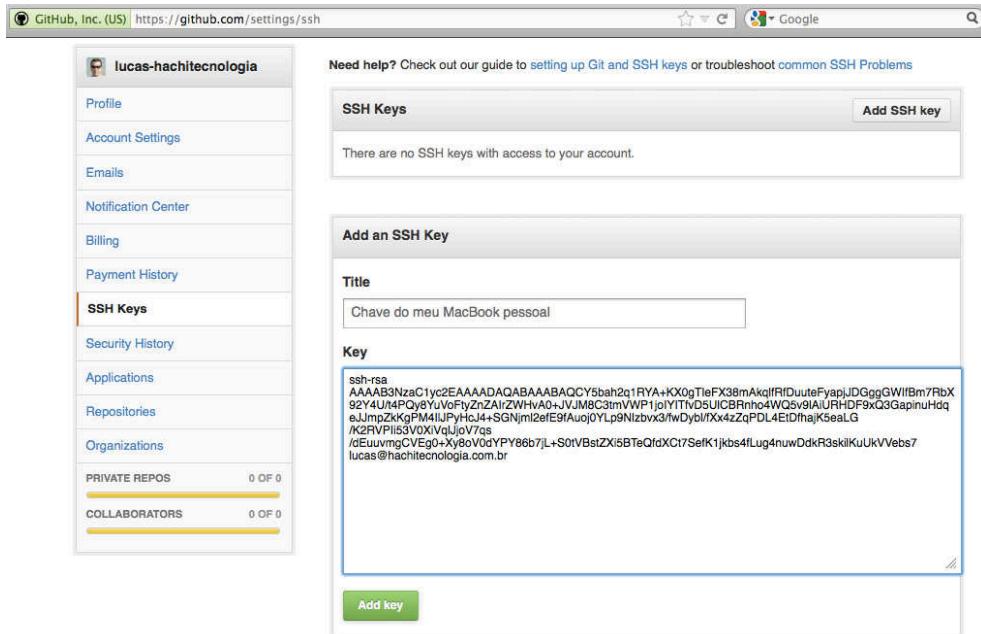


Figura 44. Cadastrando uma chave de segurança no GitHub.com

4.1.2. Criando um repositório no GitHub

Após criar nossa conta, iremos criar um repositório remoto no GitHub chamado “**curso-git**”. Para isto, basta entrar no GitHub.com com seu usuário e senha cadastrados e clicar no link “**New repository**”.

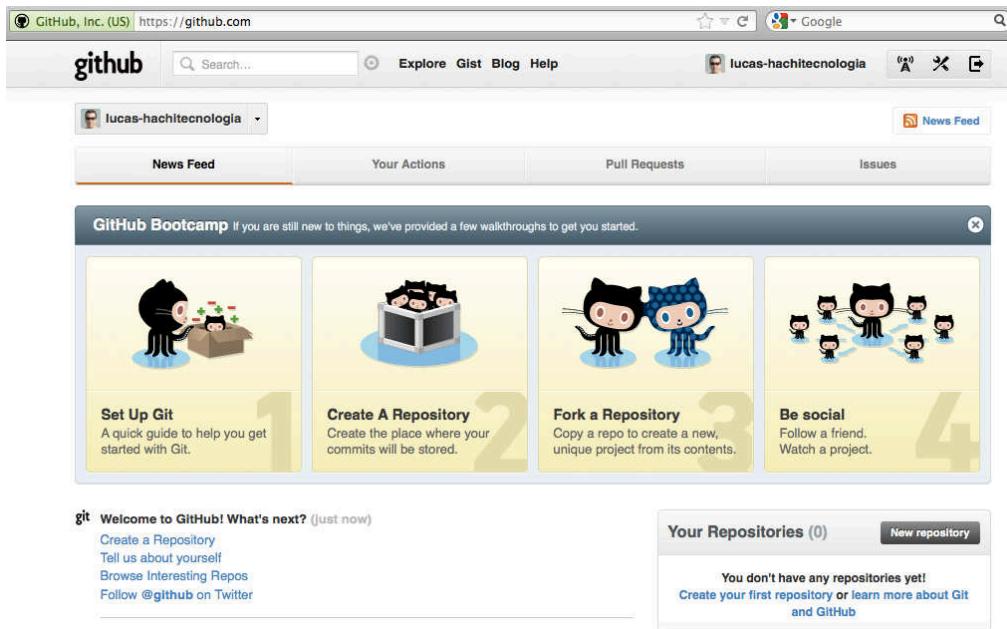


Figura 45. Página inicial da conta cadastrada no GitHub.com. Para criar um novo repositório, basta clicar no link “New Repository”.

Na página de cadastro do novo repositório, basta preencher um nome e uma descrição para o repositório, e clicar no link “Create repository”, conforme **Figura 46**.

The screenshot shows the GitHub 'Create a New Repository' page. The 'Project Name' field contains 'curso-git'. The 'Description (optional)' field contains 'Repositório do Curso de Git'. A note on the right says, 'If you intend to push a copy of a repository that is already hosted on GitHub, please fork it instead.' Below the fields are two radio buttons for access: 'Anyone (learn more about public repos)' (selected) and 'Upgrade your plan to create more private repositories!'. At the bottom is a green 'Create repository' button.

Figura 46. Formulário de cadastro de novo repositório no GitHub.com.

Nosso novo repositório remoto está criado e pronto para ser usado. Agora precisamos configurar nosso repositório remoto “**curso-git**” em nosso repositório local de mesmo nome para que seja feita a sincronização entre os dois.

Para configurar um repositório remoto do GitHub em um repositório local, basta usar o seguinte comando:

git remote add <NOME DO REPOSITÓRIO> git@github.com:<USUÁRIO>/<NOME DO REPOSITÓRIO>.git

Substituindo **<NOME DO REPOSITÓRIO>** por um nome que será usado para identificá-lo (por convenção, o Git recomenda utilizar o nome *origin*), **<USUÁRIO>** pelo nome do usuário cadastrado no GitHub e **<NOME DO REPOSITÓRIO>** pelo nome do repositório cadastrado no GitHub. Em nosso exemplo usaremos o comando:

git remote add origin git@github.com:luca-hachitecnologia/curso-git.git

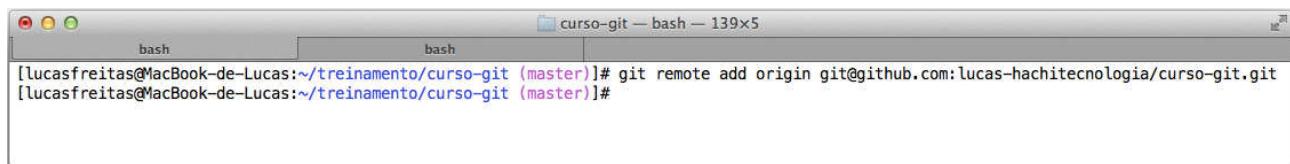


Figura 47. Adicionando o repositório remoto do GitHub em nosso repositório local.

#Fica a dica

- O comando **git remote add** adiciona um repositório remoto para sincronizarmos com o repositório local. Após cadastrar um repositório remoto podemos enviar todas as nossas alterações para ele e também baixar dele todas as alterações feitas por outros desenvolvedores. Quando sincronizamos com um repositório remoto, tanto nosso repositório local como o remoto conterá toda a parte sintética do projeto (código fonte) bem como todo o histórico de alterações realizados.

Agora que adicionamos um repositório remoto, podemos enviar a ele todas as nossas alterações realizadas no repositório local, para que outros desenvolvedores possam contribuir com o projeto.

Para enviar nossas alterações para o repositório remoto basta utilizar o comando:

git push <ALIAS DO REPOSITORIO> <NOME DA BRANCH LOCAL>

Substituindo **<ALIAS DO REPOSITORIO>** pelo nome que desejamos utilizar para nosso repositório remoto e **<NOME DA BRANCH LOCAL>** pelo nome da branch que desejamos sincronizar com o repositório remoto. No nosso caso, usaremos o seguinte comando:

git push origin master

Veja a **Figura 48** mostrando a sincronização entre os repositórios local e remoto na prática:

```
[luca...@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:luca.../curso-git.git
 * [new branch] master -> master
[luca...@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 48. Enviando as revisões do repositório local para o repositório remoto.

4.1.3. Enviando nossas alterações ao repositório do GitHub

A **Figura 48** mostra o envio das revisões feitas no nosso repositório local para nosso repositório remoto, e a saída do Git informa que tudo ocorreu conforme esperado.

Agora outros desenvolvedores poderão contribuir com nosso projeto fazendo um **clone** do repositório remoto e realizando as devidas alterações.

#Fica a dica

- Quando um desenvolvedor precisa colaborar com um projeto, ele faz um **clone** do repositório remoto. Um **clone** é uma cópia fiel de todo o repositório, incluindo tanto a parte sintética do projeto (código fonte) como todo o histórico de revisões (alterações) realizadas no projeto. Desta forma um desenvolvedor tem a liberdade de fazer suas alterações localmente, gerando suas revisões, e somente quando desejar poderá sincronizar com o repositório remoto, enviando todas as suas alterações feitas.

Ao visualizar nosso repositório remoto no site GitHub.com podemos ver claramente os arquivos enviados, juntamente com todo o histórico do projeto. Veja:

name	age	message	history
pagina.html	10 hours ago	Criando nossa primeira pagina [luca...-hachitecnologia]	

Figura 49. Visualizando o histórico do nosso projeto no site do GitHub.com.

Perceba na **Figura 49** que nosso arquivo ***página.html*** realmente foi enviado e uma informação mostra claramente o autor do último commit realizado, juntamente com a mensagem de commit.

4.1.4. Exercício

Agora que você já sabe como criar um repositório remoto no GitHub.com e sincronizar com seu repositório local, vamos colocar em prática.

1. Crie uma conta no site GitHub.com.
2. Gere uma chave de segurança RSA em seu computador e cadastre-a em sua conta do site GitHub.com.
3. Crie um repositório remoto chamado “**curso-git**” em sua conta no site GitHub.com.
4. Adicione o repositório remoto “**curso-git**” criado no site GitHub.com ao seu repositório “**curso-git**” local.
5. Sincronize seu repositório local com o repositório remoto enviando todas suas revisões a ele.
6. Abra a página do seu repositório remoto no GitHub.com e perceba as revisões enviadas.

4.2. Colaborando com o projeto

Agora que nosso projeto “**curso-git**” está hospedado no GitHub, outros desenvolvedores podem colaborar com o projeto enviando suas revisões. Por padrão, um projeto open source hospedado no GitHub permite que outros usuários façam apenas um **clone** do repositório, não permitindo que alterações sejam enviadas ao repositório remoto. Para permitirmos que outros desenvolvedores possam contribuir com o projeto, enviando suas alterações, precisamos autorizar seus usuários.

4.2.1. Adicionando um colaborador ao repositório do GitHub

No GitHub.com, autorizar um usuário para colaborar com nosso projeto é muito simples: basta abrir a área de administração do repositório que criamos, clicando no link “Admin” na página do repositório, e ir na opção “Collaborators” do menu. Na área “Collaborators” do nosso repositório “**curso-git**” podemos adicionar os usuários que desejamos autorizar como colaboradores do projeto. Veja na **Figura 50** o processo de adição de novos colaboradores.

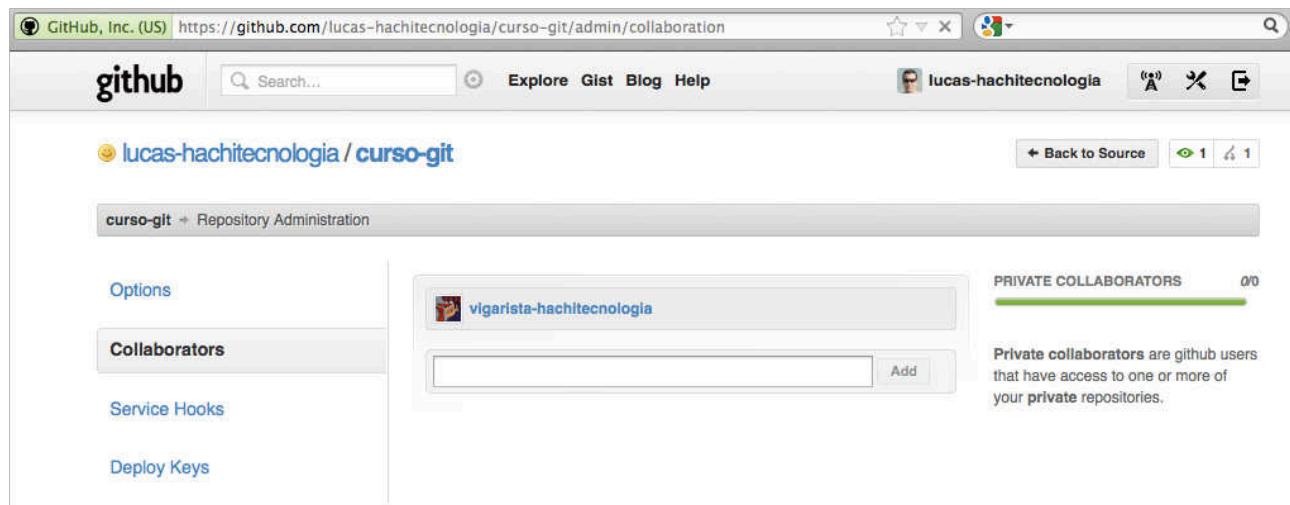


Figura 50. Adicionando um novo colaborador ao nosso repositório no GitHub.com.

Perceba na **Figura 50** que adicionamos o usuário (*vigarista-hachitecnologia*) pertencente ao Dick Vigarista. Agora Dick Vigarista está autorizado a realizar modificações em nosso projeto.

4.2.2. Realizando um *clone* do repositório remoto

O primeiro passo para Dick Vigarista colaborar com o projeto é fazer um **clone** do repositório remoto “**curso-git**”. O processo de *clonar* um repositório remoto no Git é muito simples, bastando utilizar o comando do Git:

```
git clone git@github.com:<USUARIO>/<NOME DO REPOSITÓRIO REMOTO>.git
```

No nosso caso, iremos utilizar o comando abaixo para clonar o repositório “**curso-git**”:

```
git clone git@github.com:lucas-hachitecnologia/curso-git.git
```

Veja o processo de **clone** do repositório na **Figura 51**:

```
[dickvigarista@MacBook-de-Lucas:~/git ]# git clone git@github.com:lucas-hachitecnologia/curso-git.git
Cloning into curso-git...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Receiving objects: 100% (3/3), done.
[dickvigarista@MacBook-de-Lucas:~/git ]#
```

Figura 51. Clonando o repositório remoto *curso-git*.

Após fazer o **clone**, o usuário Dick Vigarista poderá trabalhar no projeto fazendo suas alterações e sincronizando-as com o repositório remoto.

#Fica a dica

- Quando efetuamos o **clone** de um repositório, o Git comprime todos os dados que serão trafegados, usando a biblioteca *zlib*. Isto otimiza o tráfego de dados na rede, e é uma vantagem do Git, tornando o processo de sincronização mais rápido que outros SCM's.

O usuário Dick Vigarista resolve então alterar o arquivo **pagina.html** adicionando um novo parágrafo, conforme podemos ver na **Figura 52**.

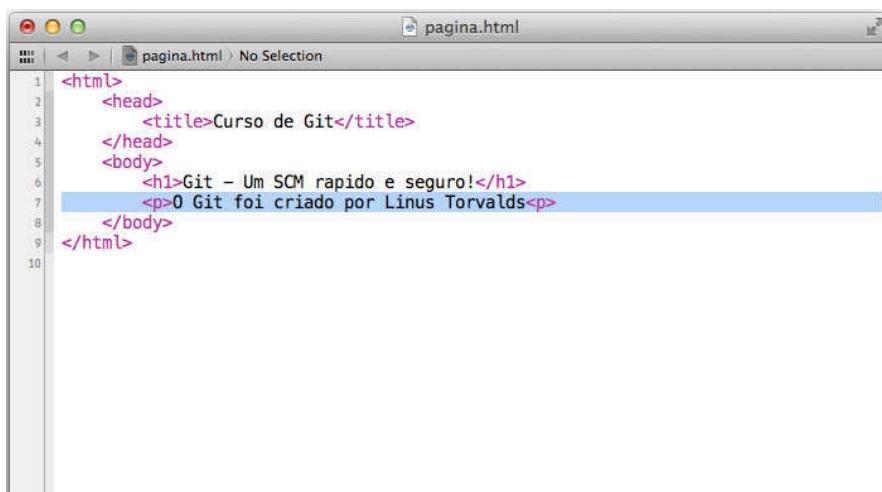
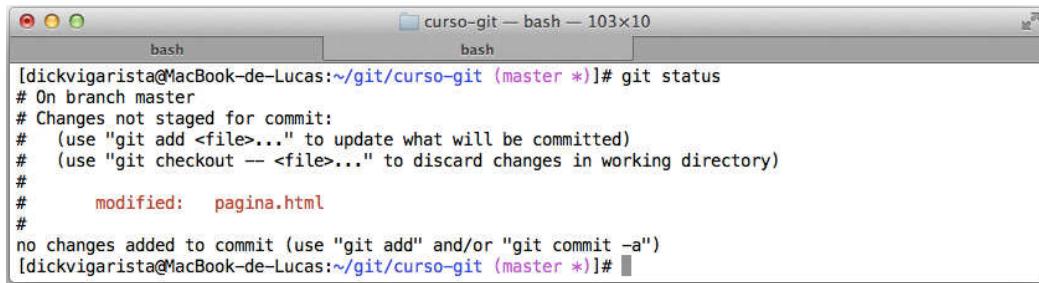


Figura 52. O usuário Dick Vigarista adiciona um novo parágrafo no arquivo *pagina.html*

GIT - Controle de Versão com Rapidez e Segurança

O **git status** informa a Dick Vigarista que o arquivo **pagina.html** foi modificado e está pronto para ser adicionado ao *staging area*, antes de ser commitado. Quando algum arquivo é alterado dizemos que o *working directory* está “sujo”, e essas alterações precisam ser adicionadas ao *staging area* para serem commitadas.



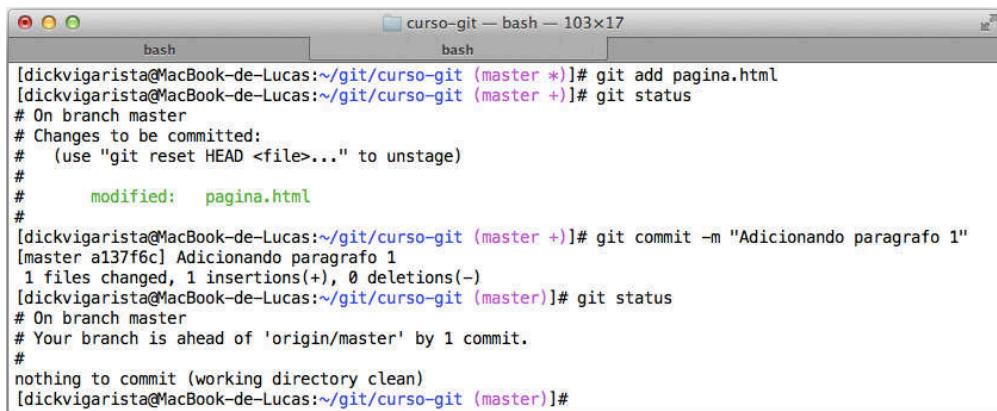
```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master *)]# git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   pagina.html
#
no changes added to commit (use "git add" and/or "git commit -a")
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master *)]#
```

Figura 53. A saída do comando *git status* informa que existem novas alterações a serem publicadas no repositório.

Para publicar suas alterações no repositório local, Dick Vigarista precisa executar a sequência de comandos:

git add pagina.html

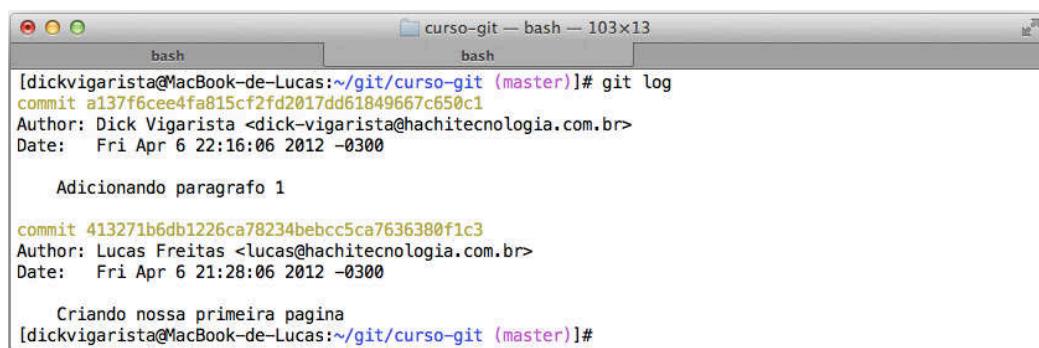
git commit -m “<COMENTÁRIO DO COMMIT>”



```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master *)]# git add pagina.html
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master +)]# git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   pagina.html
#
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master +)]# git commit -m "Adicionando paragrafo 1"
[master a137f6c] Adicionando paragrafo 1
 1 files changed, 1 insertions(+), 0 deletions(-)
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]#
```

Figura 54. Dick Vigarista commita suas alterações no repositório local e verifica o status logo em seguida.

As alterações foram commitadas no repositório local de Dick Vigarista. Podemos ver o Log dos commits com o comando **git log**.



```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git log
commit a137f6ceef4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Fri Apr 6 22:16:06 2012 -0300

    Adicionando paragrafo 1

commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

    Criando nossa primeira pagina
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]#
```

Figura 55. Log de commits mostrando todos os commits existentes no repositório de Dick Vigarista.

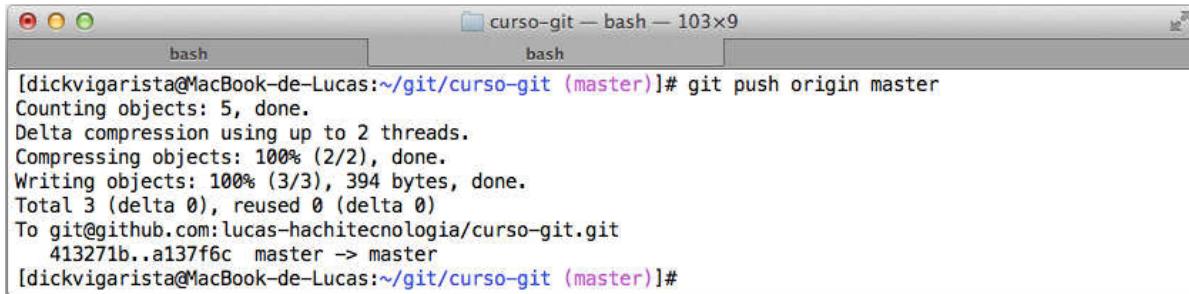
Agora Dick Vigarista possui dois commits em seu repositório: um commit realizado pelo usuário Lucas Freitas, que criou o arquivo **pagina.html**, e seu próprio commit.

4.2.3. Enviando as alterações ao repositório remoto

As alterações de Dick Vigarista foram commitadas apenas em seu repositório local. Se verificarmos a página do repositório “**curso-git**” no site do GitHub.com percebemos que o repositório remoto ainda não possui o commit de Dick Vigarista.

Para enviar as suas alterações, Dick Vigarista precisa executar um **push** ao repositório remoto, com o seguinte comando:

```
git push origin master
```



```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git push origin master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 394 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:lucas-hachitecnologia/curso-git.git
  413271b..a137f6c master -> master
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]#
```

Figura 56. Enviando as alterações para o repositório remoto com *git push*.

Agora nosso repositório remoto já possui as alterações feitas por Dick Vigarista, permitindo que outros desenvolvedores possam baixar essas alterações. Veja na página do repositório a revisão enviada por Dick Vigarista, na **Figura 57**.

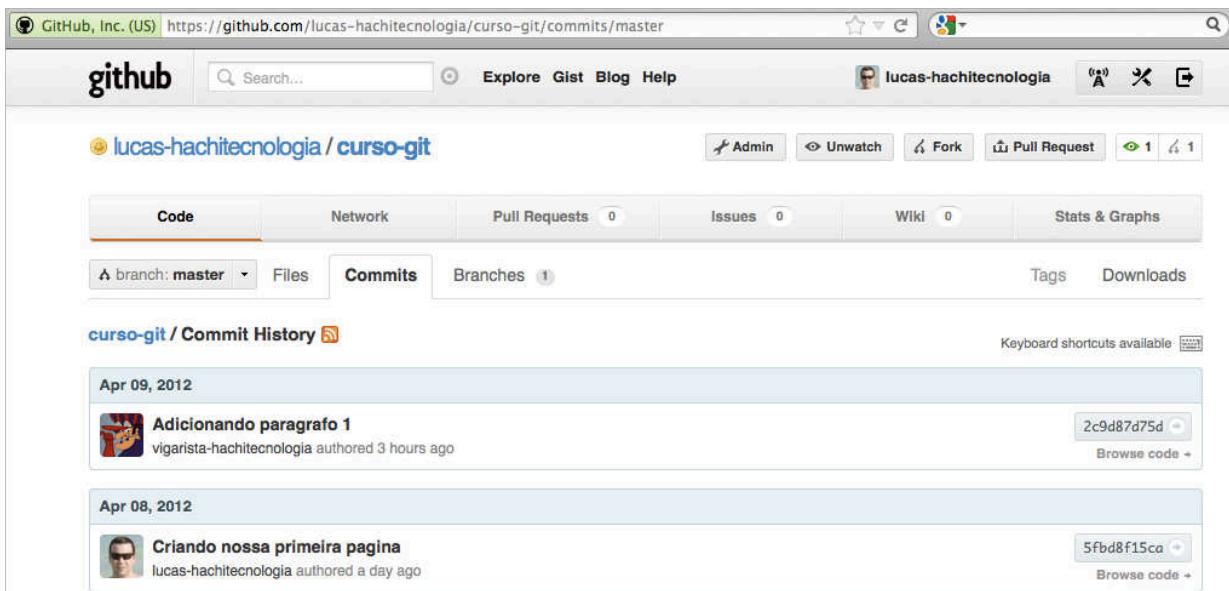


Figura 57. Log de *commits* do repositório remoto, na página do GitHub.com.

#Fica a dica

- Quando efetuamos o **clone** de um repositório remoto o Git automaticamente o adiciona em nosso repositório local, com o alias *origin*, para realizarmos as sincronizações posteriores.

4.2.4. Baixando as atualizações do repositório remoto

O usuário Dick Vigarista realizou alterações no projeto e as enviou para o repositório remoto hospedado no GitHub. Agora outros usuários poderão baixar essas atualizações com um **pull**, através do comando:

```
git pull origin master
```

```

[LucasFreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

    Criando nossa primeira pagina
[LucasFreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git pull origin master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From github.com:lucas-hachitecnologia/curso-git
 * branch            master      -> FETCH_HEAD
Updating 413271b..a137f6c
Fast-forward
 pagina.html |  1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
[LucasFreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit a137f6cee4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Fri Apr 6 22:16:06 2012 -0300

    Adicionando paragrafo 1

commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

    Criando nossa primeira pagina
[LucasFreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#

```

Figura 58. Usuário Lucas Freitas baixando as atualizações do repositório remoto enviadas por Dick Vigarista.

Veja, na **Figura 58**, que o usuário Lucas Freitas executou um **pull**, baixando todas as atualizações existentes no repositório remoto. Agora o usuário Lucas Freitas possui as alterações realizadas pelo usuário Dick Vigarista.

4.2.5. Resolução manual de conflitos

Quando estamos usando um SCM para controle de versão de um projeto, é comum dois ou mais desenvolvedores realizarem alterações no mesmo arquivo. Quando isto ocorre, ao realizar a sincronização dos repositórios, um **merge** automático é feito neste arquivo. Mas um problema que pode ocorrer é quando dois desenvolvedores resolvem modificar a mesma linha deste mesmo arquivo. Quando isto ocorre, o Git tenta aplicar o **merge** automático mas falha, e a única saída é resolver o conflito manualmente.

Imagine, por exemplo, que o usuário Lucas Freitas decide alterar o título da página para:

```
...
<title>Curso de Git SCM</title>
...
```

Após realizar a alteração, Lucas Freitas faz o commit e a envia para o repositório remoto.

Enquanto isso, o usuário Dick Vigarista decide também alterar o título da página em seu repositório local, deixando-o com o seguinte conteúdo:

```
...
<title>Curso sobre o SCM Git</title>
...
```

Dick Vigarista realiza o commit em seu repositório local mas, quando tenta enviar suas alterações para o repositório remoto, o **push** é rejeitado e o Git emite um erro informando que sua base de commits está desatualizada, sendo necessário baixar do repositório remoto todos os commits que seu repositório local ainda não possui

```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git push origin master
Password:
To ssh://lucasfreitas@localhost/Users/lucasfreitas/git/curso-git
 ! [rejected]      master --> master (non-fast-forward)
error: failed to push some refs to 'ssh://lucasfreitas@localhost/Users/lucasfreitas/git/curso-git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]#
```

Figura 59. Dick Vigarista tenta enviar suas alterações mas o repositório remoto as rejeita, devido à sua base de commits estar desatualizada.

Para resolver este problema, Dick Vigarista deverá realizar um **pull** para baixar as novas alterações do repositório remoto. Ao realizar o **pull**, o Git irá informar que um problema ocorreu ao fazer o **merge** automático devido ao usuário Lucas Freitas também ter alterado a mesma linha do arquivo **pagina.html**.

```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git pull origin master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://localhost/Users/lucasfreitas/git/curso-git
 * branch            master    -> FETCH_HEAD
Auto-merging pagina.html
CONFLICT (content): Merge conflict in pagina.html
Automatic merge failed; fix conflicts and then commit the result.
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master) **|MERGING]]#
```

Figura 60. Dick Vigarista executa o *pull* para baixar as novas alterações do repositório remoto, mas o *merge* automático falha.

Quando o **merge** automático falha, o Git cria uma branch temporária para que o conflito seja resolvido. Veja como ficou o arquivo após o conflito:

```
<html>
  <head>
    <title>Curso sobre o SCM Git</title>
    =====
    <title>Curso de Git SCM</title>
  </head>
  <body>
    <h1>Git - Um SCM rapido e seguro!</h1>
    <p>O Git foi criado por Linus Torvalds</p>
  </body>
</html>
```

Figura 61. Conteúdo do arquivo *pagina.html* após o conflito ocorrido ao executar o *merge* automático.

Agora Dick Vigarista precisa resolver o conflito manualmente, decidindo qual conteúdo irá permanecer no título da página. Dick Vigarista resolve o conflito, deixando o título com o seguinte conteúdo:

```
...
<title>Curso sobre o SCM Git</title>
...
```

Agora que Dick Vigarista editou o arquivo para resolver o conflito manualmente, basta adicionar o arquivo no *Index (staging area)* e realizar o commit com a nova alteração.

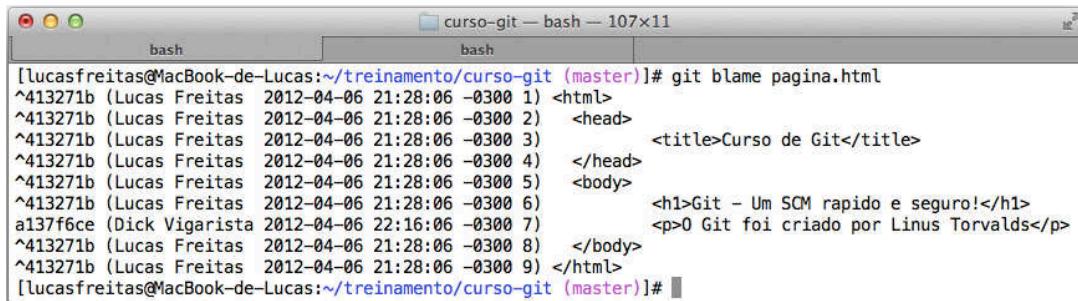
4.2.6. Git **blame**, o famoso dedo-duro

O Git possui um recurso muito útil que nos mostra o autor de cada linha de um arquivo em nosso repositório, bem como a data em que a linha foi alterada/adicionada, a hora e a hash do commit. Isto é muito útil para sabermos quem fez determinada modificação ou quem causou um determinado bug em um arquivo do código fonte, ou seja, um grande dedo-duro.

O **git blame** possui a seguinte sintaxe:

```
git blame <ARQUIVO>
```

Veja, na **Figura 62**, a saída do **git blame** no arquivo **pagina.html**:



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git blame pagina.html
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 1) <html>
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 2)  <head>
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 3)    <title>Curso de Git</title>
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 4)  </head>
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 5)  <body>
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 6)    <h1>Git - Um SCM rapido e seguro!</h1>
a137f6ce (Dick Vigarista 2012-04-06 22:16:06 -0300 7)    <p>O Git foi criado por Linus Torvalds</p>
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 8)  </body>
^413271b (Lucas Freitas 2012-04-06 21:28:06 -0300 9) </html>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 62. Saída do **git blame** no arquivo **pagina.html**.

4.2.7. Exercício

Agora que você já sabe contribuir com um repositório remoto, vamos colocar em prática:

1. Crie outro usuário no GitHub.com (não esqueça de cadastrar sua chave de segurança).
2. Adicione o novo usuário como colaborador do repositório “**curso-git**”.
3. Faça um **clone** do repositório a partir do novo usuário.
4. Modifique o arquivo **pagina.html** adicionando um novo parágrafo, deixando-o com o seguinte conteúdo:

```
<html>
  <head>
    <title>Curso de Git</title>
  </head>
  <body>
    <h1>Git - Um SCM rapido e seguro!</h1>
    <p>O Git foi criado por Linus Torvalds</p>
  </body>
</html>
```

5. Salve o arquivo e faça o commit no repositório local.
6. Envie as alterações para o repositório remoto do GitHub.com.
7. Baixe as alterações do repositório remoto com o primeiro usuário que você cadastrou no GitHub.com.

5 - Recursos avançados do Git

5.1. Trabalhando com branches

Uma **branch** é uma ramificação do projeto, uma cópia paralela do projeto que nos permite realizar modificações sem alterar o projeto original. Trabalhar com *branches* (ramificações) no Git é uma tarefa fácil.

A vantagem em se utilizar uma *branch* é simples: imagine que você está trabalhando em seu projeto e seu chefe solicita uma correção imediata de um bug ou uma rápida alteração no código, como mudar uma label ou trocar a cor de um botão. Quem trabalha com desenvolvimento de software vivencia diariamente essas “mudanças de prioridade”. Usando *branches*, no Git, você pode trabalhar no novo requisito sem ter que perder todas as alterações que havia feito antes da nova solicitação. Isto é muito útil, já que não queremos commitar nossas alterações pela metade ou ter que descartar todas elas para atender à nova solicitação e refazê-las em outro momento.

Uma boa prática é trabalhar sempre com uma *branch* local que não seja a *master* (branch padrão). Desta forma poluímos menos nosso projeto com commits indesejados e organizamos melhor as nossas revisões na *branch master*.

5.1.1. Criando uma branch

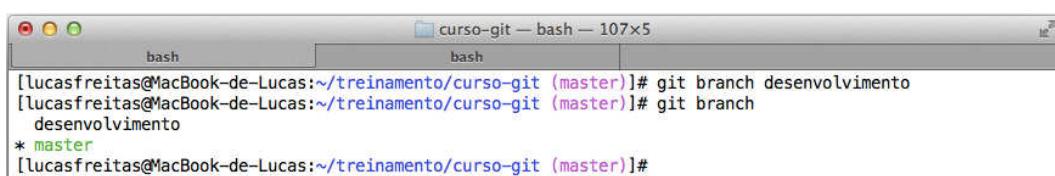
Para criar uma **branch**, basta executar o comando:

git branch <NOME DA BRANCH>

substituindo **<NOME DA BRANCH>** pelo nome desejado.

Vamos criar uma **branch** chamada *desenvolvimento* em nosso repositório local, usando o comando:

git branch desenvolvimento



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git branch desenvolvimento
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git branch
* master
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 63. Criando uma branch *desenvolvimento*.

#Fica a dica

- Podemos visualizar as *branches* do nosso repositório local apenas digitando o comando: **git branch**.

5.1.2. Alternando entre branches

Nossa branch *desenvolvimento* está criada, porém ainda estamos na branch *master* e precisamos alternar para a nova branch com o comando:

git checkout desenvolvimento

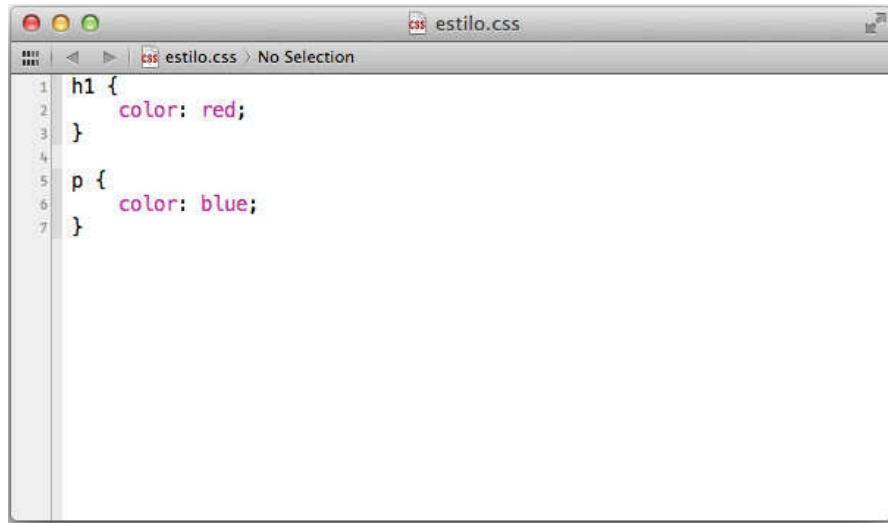
#Fica a dica

- Podemos criar uma *branch* e alternar para ela utilizando um único comando: **git checkout -b desenvolvimento**.

5.1.3. Trabalhando com a nova branch

Imagine que queremos fazer algumas alterações em nosso projeto e enviá-las ao repositório apenas quando estiverem maduras e homologadas. Podemos usar nossa branch *desenvolvimento* para isto. Para exemplificar, vamos alterar nosso projeto adicionando uma folha de estilo em nossa página, mas iremos fazer isso na branch *desenvolvimento*.

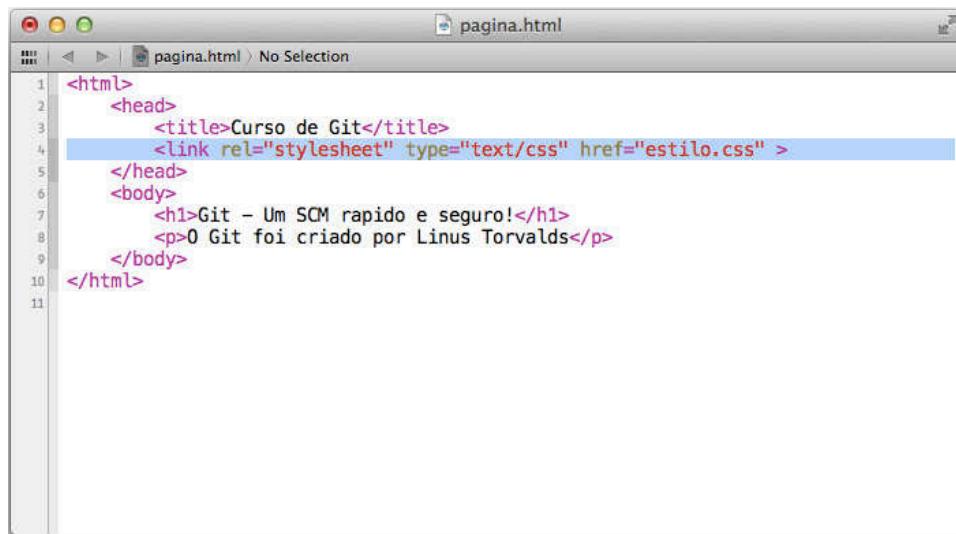
Criamos o arquivo **estilo.css** na branch **desenvolvimento** com o seguinte conteúdo:



```
1 h1 {  
2     color: red;  
3 }  
  
4 p {  
5     color: blue;  
6 }  
  
7
```

Figura 64. Criando uma folha de estilo para nossa página.

Definimos a cor vermelha para o cabeçalho e a cor azul para os parágrafos. Precisamos agora aplicar a folha de estilo em nossa página, no arquivo **pagina.html**, conforme **Figura 65**.



```
1 <html>  
2     <head>  
3         <title>Curso de Git</title>  
4         <link rel="stylesheet" type="text/css" href="estilo.css" >  
5     </head>  
6     <body>  
7         <h1>Git - Um SCM rápido e seguro!</h1>  
8         <p>O Git foi criado por Linus Torvalds</p>  
9     </body>  
10    </html>  
11
```

Figura 65. Aplicando o estilo em nossa página.

Após criar a folha de estilo e aplicá-la em nossa página, vamos realizar um commit das alterações feitas na branch *desenvolvimento*.

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git add pagina.html estilo.css
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento +)]# git commit -m "Adicionando estilo a nossa pagina"
[desenvolvimento 32c75bd] Adicionando estilo a nossa pagina
 2 files changed, 8 insertions(+), 0 deletions(-)
 create mode 100644 estilo.css
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]#
```

Figura 66. Realizando o *commit* das alterações realizadas na branch *desenvolvimento*.

Realizamos as alterações na branch *desenvolvimento* e fizemos o commit destas alterações. Perceba que as alterações foram commitadas apenas na branch *desenvolvimento*. Podemos verificar isso claramente quando comparamos o Log de commits da branch *desenvolvimento* com os da branch *master*.

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git log
commit 32c75bd8307c55cb6db660d1ff3b05aadf46f882
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 22:36:35 2012 -0300

  Adicionando estilo a nossa pagina

commit a137f6cee4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Fri Apr 6 22:16:06 2012 -0300

  Adicionando paragrafo 1

commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

  Criando nossa primeira pagina
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]#
```

Figura 67. Log da branch *desenvolvimento*.

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git checkout master
Switched to branch 'master'
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit a137f6cee4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Fri Apr 6 22:16:06 2012 -0300

  Adicionando paragrafo 1

commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

  Criando nossa primeira pagina
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 68. Log da branch *master*. Perceba que as alterações realizadas na branch *desenvolvimento* não refletiram na branch *master*.

Se uma nova prioridade no projeto for solicitada, como a correção de um bug por exemplo, e nossas alterações não estiverem maduras o suficiente para irem para a produção, podemos voltar para a branch *master* e realizar as novas solicitações, sem perdermos o nosso trabalho.

Imagine que seu chefe tenha pedido para adicionar um novo parágrafo à página, mas ele não quer que a folha de estilo que aplicamos entre em produção, pois ela ainda não está completa. Como as modificações que fizemos não foram feitas na branch *master*, ficará fácil atender a essa nova solicitação.

Voltaremos para a branch ***master*** com o comando:

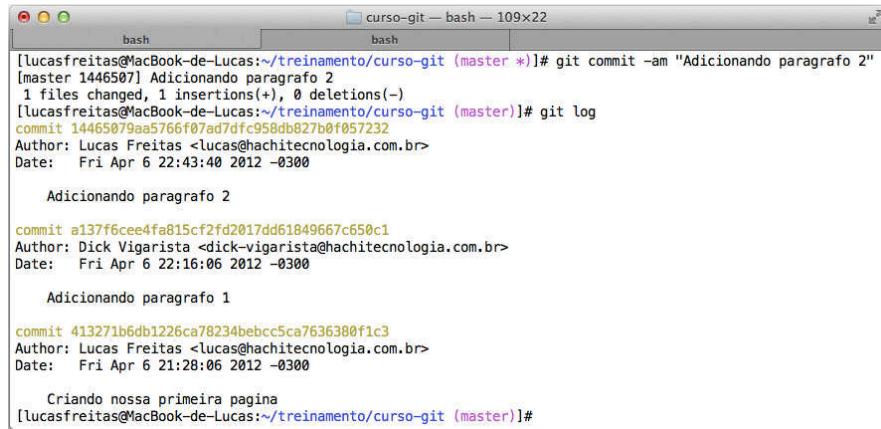
git checkout master

GIT - Controle de Versão com Rapidez e Segurança

e adicionamos o novo parágrafo

<p>O Git é um SCM distribuído.</p>

Após a alteração, realizamos um commit na branch **master** e a versão está pronta para ser usada em produção.



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git commit -am "Adicionando parágrafo 2"
[master 1446507] Adicionando parágrafo 2
 1 files changed, 1 insertions(+), 0 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit 14465079aa5766f07ad7dfc958db827b0f057232
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 22:43:40 2012 -0300

    Adicionando parágrafo 2

commit a137f6cee4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Fri Apr 6 22:16:06 2012 -0300

    Adicionando parágrafo 1

commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

    Criando nossa primeira pagina
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 69. Realizando o commit das novas alterações solicitadas.

Porém, ao abrir a página, seu chefe percebe que existe um bug causado pelo mal fechamento da tag de parágrafo, onde adicionamos <p> ao invés de </p>, acarretando no mal funcionamento do site.

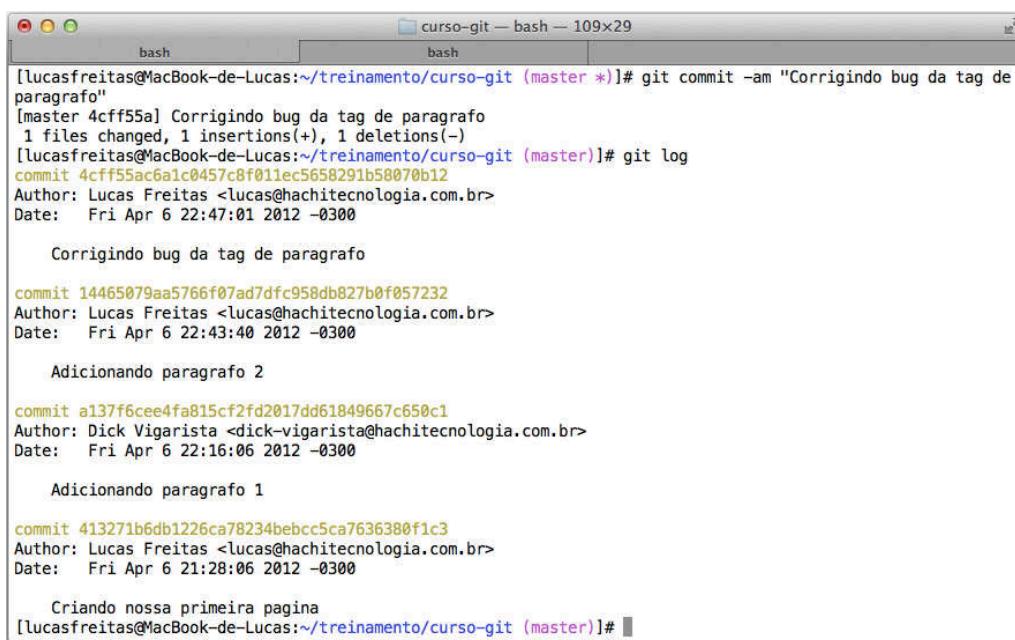
Devemos então corrigir o bug causado, trocando o parágrafo bugado de

<p>O Git é um SCM distribuído.</p>

para

<p>O Git é um SCM distribuído.</p>

Realizamos então um commit com a correção do bug, gerando uma nova revisão.



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git commit -am "Corrigindo bug da tag de parágrafo"
[master 4cff55a] Corrigindo bug da tag de parágrafo
 1 files changed, 1 insertions(+), 1 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit 4cff55ac6a1c0457cbf011ec5658291b58070b12
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 22:47:01 2012 -0300

    Corrigindo bug da tag de parágrafo

commit 14465079aa5766f07ad7dfc958db827b0f057232
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 22:43:40 2012 -0300

    Adicionando parágrafo 2

commit a137f6cee4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Fri Apr 6 22:16:06 2012 -0300

    Adicionando parágrafo 1

commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

    Criando nossa primeira pagina
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 70. Realizando um commit com a correção do bug.

#Fica a dica

- Para adicionar as alterações realizadas na *staging area* e commitá-las ao mesmo tempo, podemos usar a flag “**-a**” no *commit*:

```
git commit -am "<mensagem do commit>"
```

Agora iremos dar continuidade ao nosso trabalho na branch **desenvolvimento**. Para isto, executamos novamente o comando:

```
git checkout desenvolvimento
```

para voltar para a branch **desenvolvimento**.

Podemos agora realizar as alterações que faltam em nossa folha de estilo. Seu chefe pediu que fosse adicionado um recurso para centralizar os cabeçalhos da página, e faremos isto na folha de estilo, deixando-a com o seguinte conteúdo:

Figura 71. Adicionando um recurso para centralizar os cabeçalhos em nossa página.

Realizada a alteração, executamos o commit:

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento *)]# git commit -am "Adicionando recurso para centralizar os cabecalhos"
[desenvolvimento 720f999] Adicionando recurso para centralizar os cabecalhos
 1 files changed, 1 insertions(+), 0 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git log
commit 720f999da8f136ba127c71719889a86d83f36f00
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 22:51:19 2012 -0300

    Adicionando recurso para centralizar os cabecalhos

commit 32c75bd8307c55cb6db660d1ff3b05aadf46f882
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 22:36:35 2012 -0300

    Adicionando estilo a nossa pagina

commit a137f6cee4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Fri Apr 6 22:16:06 2012 -0300

    Adicionando paragrafo 1

commit 413271b6db1226ca78234bebcc5ca7636380f1c3
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Fri Apr 6 21:28:06 2012 -0300

    Criando nossa primeira pagina
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]#
```

Figura 72. Commitando as alterações realizadas na branch *desenvolvimento*.

5.1.4. Enviando as alterações para a *branch master*

Agora que nossas alterações na branch **desenvolvimento** estão prontas, é hora de enviá-las para a branch **master**. Mas para isso, precisamos estar atualizados em relação aos commits da branch *master* que ainda não possuímos na branch *desenvolvimento*. Existem dois commits na branch *master* que a branch *desenvolvimento* ainda não possui. Se simplesmente jogássemos os commits da branch *desenvolvimento* na branch *master*, sem que ela esteja com a base de commits atualizada em relação à branch *master*, correríamos o risco de vários conflitos ocorrerem, sujando o log de commits do nosso projeto.

Para atualizar a nossa base de commits da branch **desenvolvimento**, precisamos puxar os commits da branch **master** que ainda não possuímos. Para isto, basta utilizar o comando:

git rebase master

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git rebase master
First, rewinding head to replay your work on top of it...
Applying: Adicionando estilo a nossa pagina
Applying: Adicionando recurso para centralizar os cabecalhos
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git log
commit 036a5a6b7e2812f3671b65f0c5b1f80dd4a075e
Author: Lucas Freitas <luca...@hachitecnologia.com.br>
Date:   Fri Apr 6 22:51:19 2012 -0300

    Adicionando recurso para centralizar os cabecalhos

commit f6c4776cb7b370699e77ac430fcdded3d053e6ec
Author: Lucas Freitas <luca...@hachitecnologia.com.br>
Date:   Fri Apr 6 22:36:35 2012 -0300

    Adicionando estilo a nossa pagina

commit 4cff55ac6a1c0457c8f011ec5658291b58070b12
Author: Lucas Freitas <luca...@hachitecnologia.com.br>
Date:   Fri Apr 6 22:47:01 2012 -0300

    Corrigindo bug da tag de paragrafo

commit 14465079aa5766f07ad7dfc958db827b0f057232
Author: Lucas Freitas <luca...@hachitecnologia.com.br>
Date:   Fri Apr 6 22:43:40 2012 -0300

    Adicionando paragrafo

commit a137f6cee4fa815cf2fd2017dd61849667c650c1
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
```

Figura 73. Commitando as alterações realizadas na branch *desenvolvimento*.

Agora que temos todos os commits da branch **master** podemos enviar as alterações feitas na branch **desenvolvimento** para ela. Para isto, precisamos alternar para a branch **master**, com o comando:

git checkout master

Estando na branch **master**, podemos puxar as alterações da branch **desenvolvimento** com o comando:

git merge desenvolvimento

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git checkout master
Switched to branch 'master'
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git merge desenvolvimento
Updating 4cff55a..036a5a6
Fast-forward
  estilo.css |    8 ++++++++
  pagina.html |    1 +
  2 files changed, 9 insertions(+), 0 deletions(-)
  create mode 100644 estilo.css
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 74. Puxando os commits da branch *desenvolvimento* para a branch *master*.

Pronto! Nossa branch **master** agora possui todos os commits realizados na branch **desenvolvimento**. Podemos agora enviar nossos commits para o repositório remoto, com o comando:

```
git push origin master
```

5.1.5. Resolvendo conflitos no rebase

No exemplo anterior, o **rebase** realizado para atualizar a base de commits da branch **desenvolvimento** ocorreu com sucesso. Mas e se outro desenvolvedor tivesse editado na branch **master** o mesmo arquivo que editamos na branch **desenvolvimento** alterando as mesmas linhas? Isto com certeza iria gerar um conflito na hora do *rebase*.

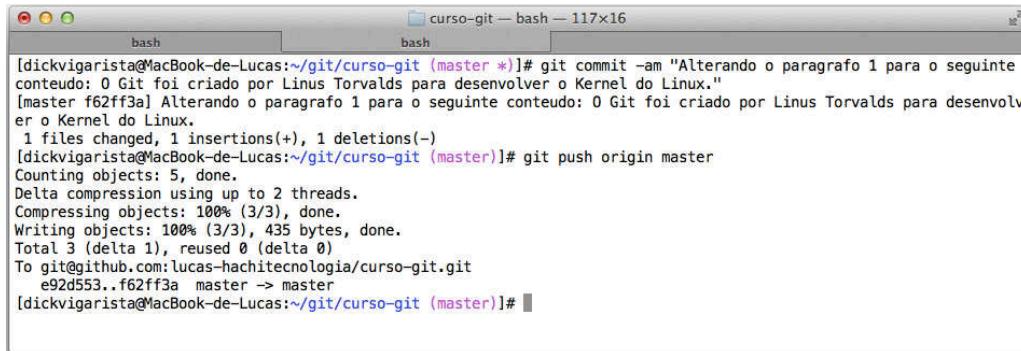
Imagine que o usuário Dick Viragista decide alterar o parágrafo:

```
<p>O Git foi criado por Linus Torvalds</p>
```

mudando-o para o seguinte conteúdo:

```
<p>O Git foi criado por Linus Torvalds para desenvolver o Kernel do Linux.</p>
```

Após realizar suas alterações, Dick Vigarista realiza um commit na branch **master** e envia sua revisão para o repositório remoto.



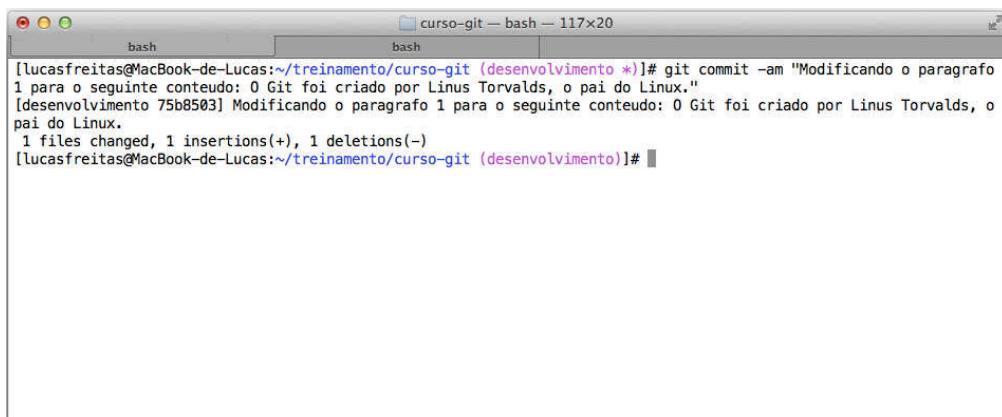
```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master *)]# git commit -am "Alterando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds para desenvolver o Kernel do Linux."
[master f62ff3a] Alterando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds para desenvolver o Kernel do Linux.
1 files changed, 1 insertions(+), 1 deletions(-)
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git push origin master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 435 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:luca-hachitecnologia/curso-git.git
 e92d553..f62ff3a master -> master
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]#
```

Figura 75. Dick Vigarista commita suas alterações e as envia para o repositório remoto.

Mas o usuário Lucas Freitas também decide alterar o mesmo parágrafo, deixando-o com o seguinte conteúdo:

```
<p>O Git foi criado por Linus Torvalds, o pai do Linux.</p>
```

O usuário Lucas Freitas commita sua alteração na branch **desenvolvimento**



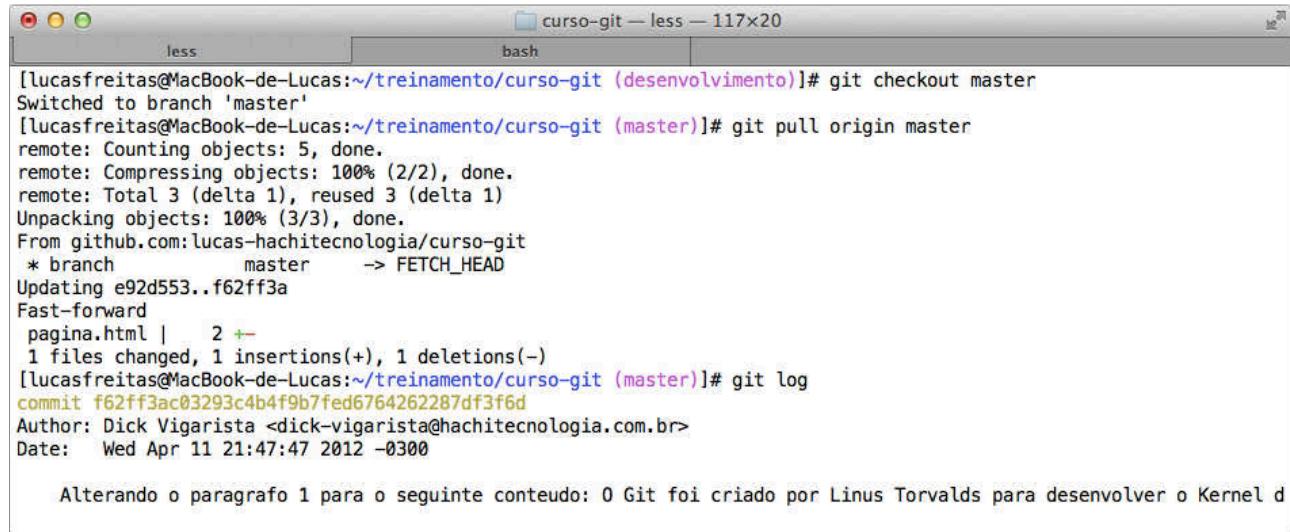
```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento *)]# git commit -am "Modificando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds, o pai do Linux."
[desenvolvimento 75b8503] Modificando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds, o pai do Linux.
1 files changed, 1 insertions(+), 1 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]#
```

Figura 76. Lucas Freitas faz suas alterações e as commita na branch *desenvolvimento*.

GIT - Controle de Versão com Rapidez e Segurança

Agora Lucas Freitas precisa enviar sua alteração da branch **desenvolvimento** para a branch **master**. Mas para isso, Lucas Freitas precisa atualizar sua base de commits puxando as novas alterações da branch **master**.

Lucas Freitas faz um **pull** em sua branch **master** para puxar as alterações feitas por outros usuários, no caso a alteração feita por Dick Vigarista.



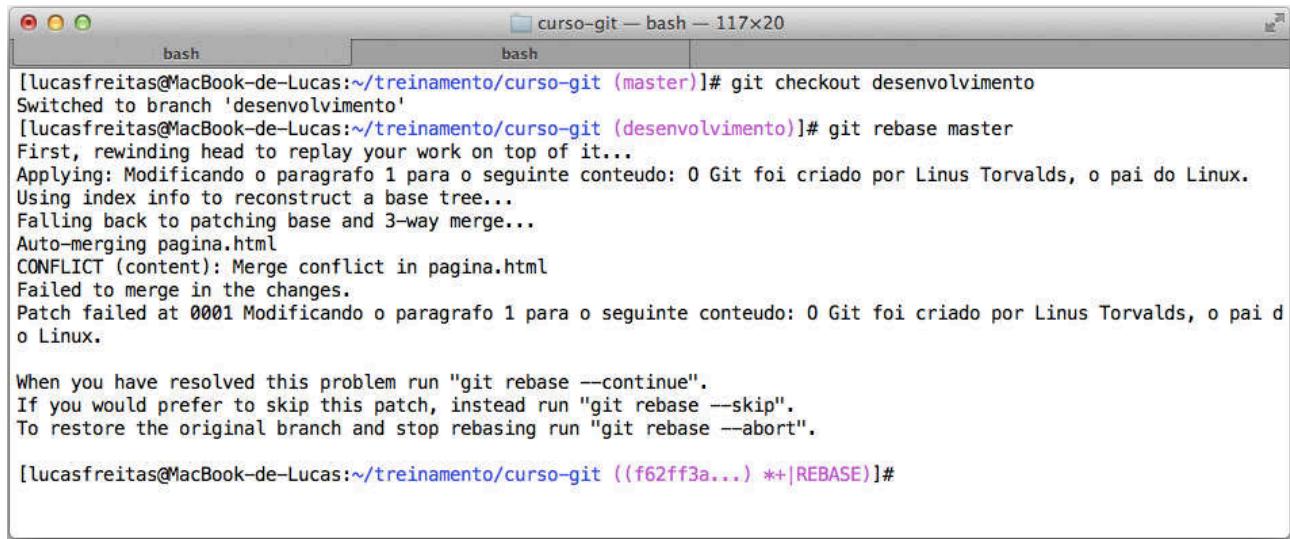
```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git checkout master
Switched to branch 'master'
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git pull origin master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From github.com:lucas-hachitecnologia/curso-git
 * branch            master      -> FETCH_HEAD
Updating e92d553..f62ff3a
Fast-forward
 pagina.html |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit f62ff3ac03293c4b4f9b7fed6764262287df3f6d
Author: Dick Vigarista <dick-vigarista@hachitecnologia.com.br>
Date:   Wed Apr 11 21:47:47 2012 -0300

    Alterando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds para desenvolver o Kernel d
```

Figura 77. Lucas Freitas baixa as alterações do repositório remoto para a branch **master** local.

Agora Lucas Freitas possui em sua branch **master** local todas as alterações existentes no repositório remoto, enviada por outros usuários. Porém, sua branch local **desenvolvimento** ainda não possui essas alterações. O usuário Lucas Freitas deverá agora fazer o **rebase** na branch **desenvolvimento** para atualizar a base de commits de acordo com a branch **master**.

Ao realizar o **rebase**, um conflito ocorre, gerando a seguinte saída:



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git checkout desenvolvimento
Switched to branch 'desenvolvimento'
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git rebase master
First, rewinding head to replay your work on top of it...
Applying: Modificando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds, o pai do Linux.
Using index info to reconstruct the base tree...
Fallling back to patching base and 3-way merge...
Auto-merging pagina.html
CONFLICT (content): Merge conflict in pagina.html
Failed to merge in the changes.
Patch failed at 0001 Modificando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds, o pai d
o Linux.

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".

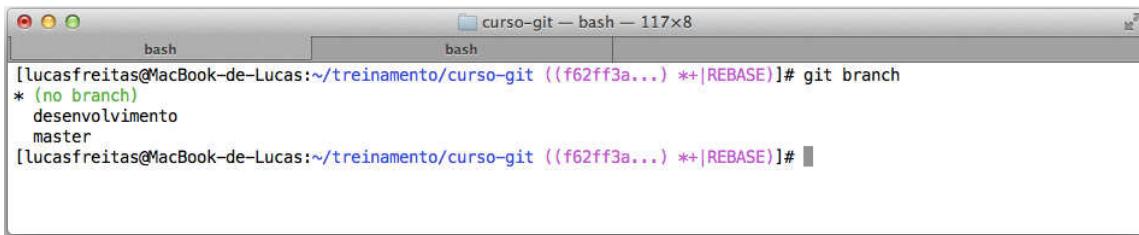
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)*+|REBASE)]#
```

Figura 78. Lucas Freitas realiza o rebase na branch **master** e percebe que ocorreram conflitos.

O Git informou que foi encontrado um conflito no arquivo **pagina.html**, ocorrido durante o merge automático no arquivo, devido às alterações realizadas pelos dois usuários na mesma linha do código. Quando um conflito ocorre durante o **rebase**, o Git cria uma branch temporária para que o conflito seja resolvido. Isto ocorrerá em todos os commits que gerarem

GIT - Controle de Versão com Rapidez e Segurança

conflito na hora do *rebase*, e deveremos resolver um a um. Perceba na **Figura 79** que ao executar o comando **git branch**, o Git nos informa que estamos em uma branch chamada de “**(no branch)**”, que é uma branch temporária.



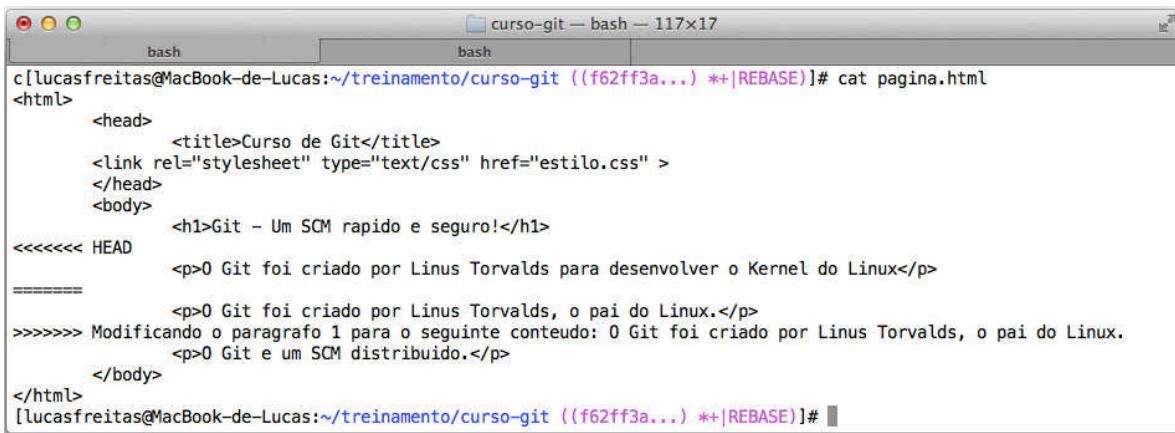
```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)*+[REBASE])# git branch
* (no branch)
  desenvolvimento
  master
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)*+[REBASE])# ]
```

Figura 79. O Git informa que estamos em uma branch temporária, chamada de *(no branch)*, para resolver os conflitos do *rebase*.

No aviso do conflito, na **Figura 78**, o próprio Git nos informa três possíveis soluções para resolvê-lo:

1. Usar o comando **git rebase --skip** para descartar o nosso commit, substituindo-o pelo commit feito por Dick Vigarista;
2. Usar o comando **git rebase --abort** para cancelar o *rebase* e descartar as atualizações vindas do branch **master**;
3. Ou resolver os conflitos manualmente, decidindo qual o conteúdo que deverá ficar no arquivo **pagina.html**, e usar o comando **git rebase --continue** após resolver estes conflitos para dar continuidade ao *rebase*.

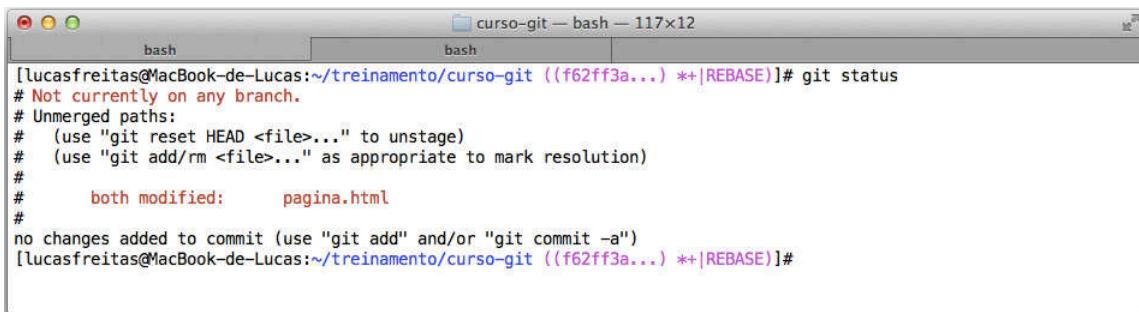
Veja o conteúdo do arquivo **pagina.html** após o conflito:



```
c[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)*+[REBASE])# cat pagina.html
<html>
  <head>
    <title>Curso de Git</title>
    <link rel="stylesheet" type="text/css" href="estilo.css" >
  </head>
  <body>
    <h1>Git - Um SCM rápido e seguro!</h1>
<===== HEAD
<p>0 Git foi criado por Linus Torvalds para desenvolver o Kernel do Linux</p>
=====
<p>0 Git foi criado por Linus Torvalds, o pai do Linux.</p>
>>>>> Modificando o parágrafo 1 para o seguinte conteúdo: O Git foi criado por Linus Torvalds, o pai do Linux.
<p>0 Git é um SCM distribuído.</p>
</body>
</html>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)*+[REBASE])# ]
```

Figura 80. Conteúdo do arquivo *pagina.html* com o conflito, após o *rebase*.

O comando **git status** nos informa a mensagem “*both modified: pagina.html*” indicando que ambos os desenvolvedores alteraram o mesmo arquivo, gerando um conflito. Devemos resolver o conflito para que esta mensagem desapareça.

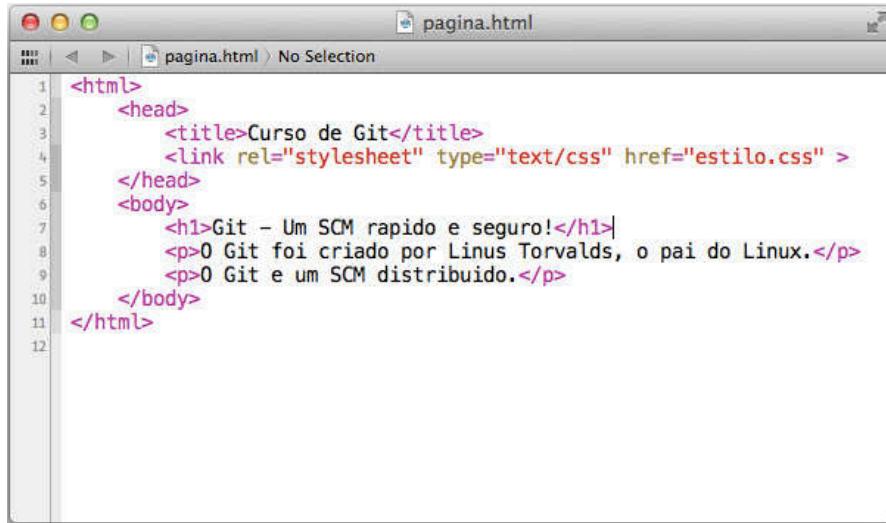


```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)*+[REBASE])# git status
# Not currently on any branch.
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      pagina.html
#
no changes added to commit (use "git add" and/or "git commit -a")
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)*+[REBASE])# ]
```

Figura 81. Saída do comando *git status* informando que existe um conflito no arquivo *pagina.html*.

GIT - Controle de Versão com Rapidez e Segurança

Optaremos por resolver o conflito manualmente, conforme aprendemos anteriormente no curso, decidindo ficar com o conteúdo de Lucas Freitas para o parágrafo 1 da página. Alteramos então o arquivo, para que ele fique com o seguinte conteúdo:



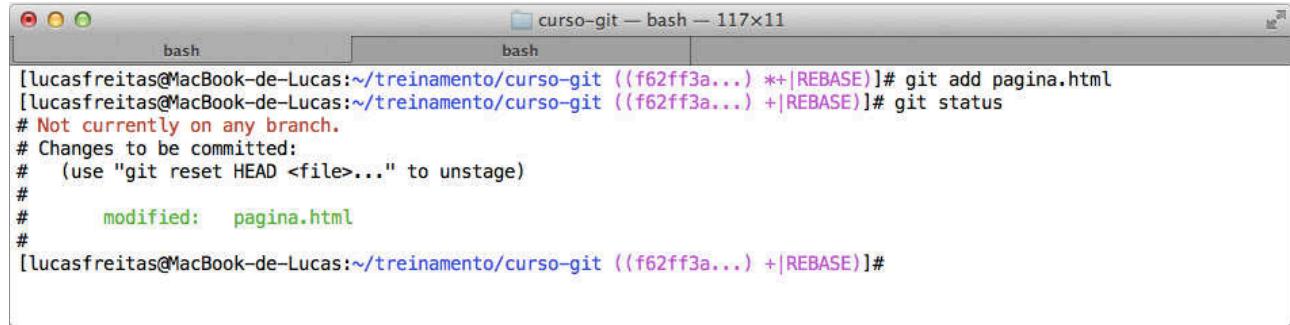
```
<html>
  <head>
    <title>Curso de Git</title>
    <link rel="stylesheet" type="text/css" href="estilo.css" >
  </head>
  <body>
    <h1>Git - Um SCM rápido e seguro!</h1>
    <p>O Git foi criado por Linus Torvalds, o pai do Linux.</p>
    <p>O Git é um SCM distribuído.</p>
  </body>
</html>
```

Figura 82. Conteúdo do arquivo `página.html` após resolução do conflito gerado pelo `rebase`.

Após resolver os conflitos manualmente, adicionamos o arquivo **`página.html`** ao `Index`, com o comando:

```
git add página.html
```

Verificando o status, percebemos que agora a mensagem “*both modified*” desapareceu, conforme **Figura 83**.

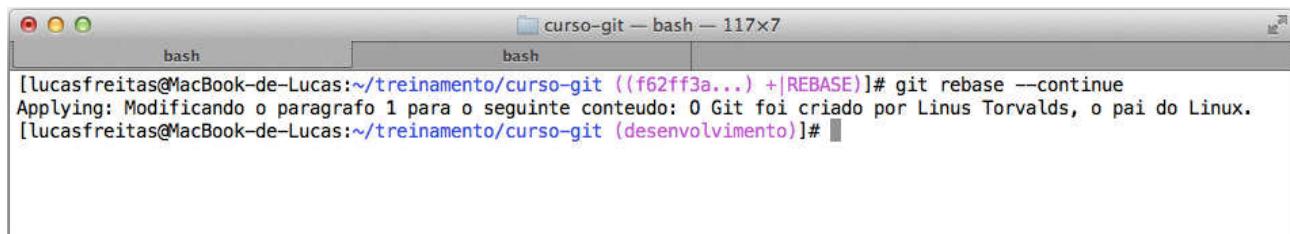


```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)+|REBASE)]# git add página.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)+|REBASE)]# git status
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   página.html
#
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)+|REBASE)]#
```

Figura 83. O status mostra que o conflito foi resolvido.

Após ter resolvido todos os conflitos, podemos dar continuidade ao `rebase` com o comando:

```
git rebase --continue
```



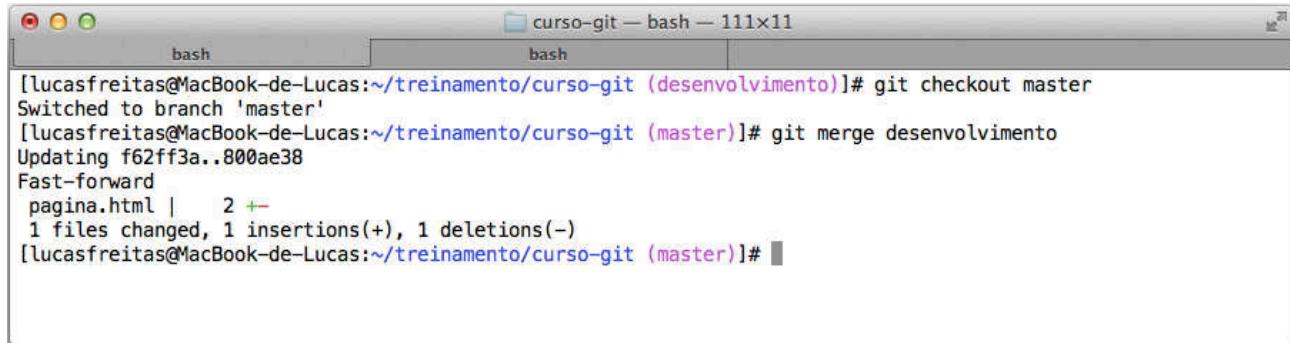
```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f62ff3a...)+|REBASE)]# git rebase --continue
Applying: Modificando o parágrafo 1 para o seguinte conteúdo: O Git foi criado por Linus Torvalds, o pai do Linux.
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]#
```

Figura 84. Dando continuidade ao `rebase`, após resolvidos os conflitos.

Agora que resolvemos os conflitos gerados pelo **rebase**, podemos enviar nossas alterações para a branch **master**, executando os comandos:

```
git checkout master
```

```
git merge desenvolvimento
```

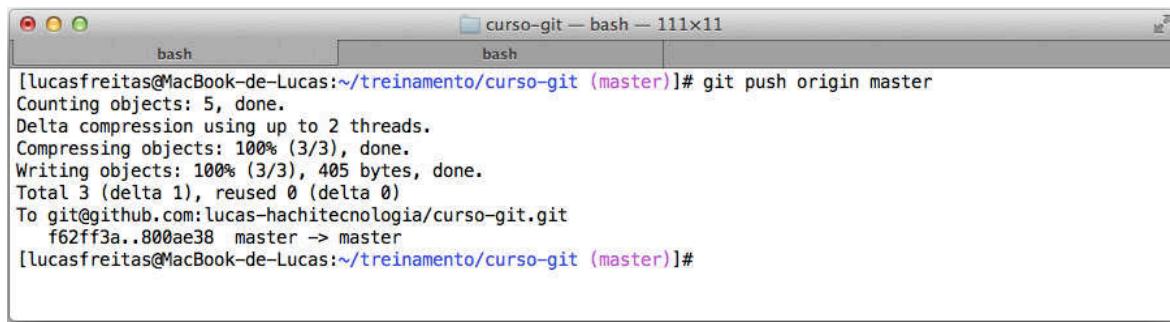


```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git checkout master
Switched to branch 'master'
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git merge desenvolvimento
Updating f62ff3a..800ae38
Fast-forward
 pagina.html |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 85. Puxando os commits da branch *desenvolvimento* para a branch *master*.

Pronto! Nossas alterações estão na branch **master** e prontas para serem enviadas para o repositório remoto, usando o comando:

```
git push origin master
```



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git push origin master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 405 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:lucas-hachitecnologia/curso-git.git
 f62ff3a..800ae38 master -> master
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 86. Enviando as alterações da branch *master* local para o repositório remoto.

5.1.6. Enviando uma *branch* local para o repositório remoto

A branch **desenvolvimento** existe apenas em nosso repositório local e, mesmo realizando um *push* da branch **master** para o repositório remoto, a branch **desenvolvimento** não será enviada.

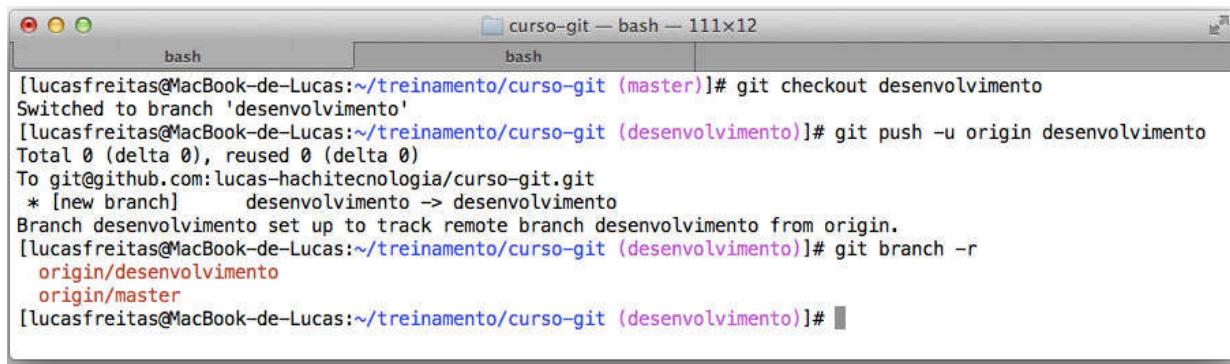
Para enviar nossa branch **desenvolvimento** para o repositório remoto, para que outros desenvolvedores possam trabalhar nesta branch e contribuir com alterações, devemos alternar para esta branch (com o comando **git checkout desenvolvimento**) e executar o seguinte comando:

```
git push -u origin desenvolvimento
```

#Fica a dica

- Para listarmos as branches existentes no repositório remoto, usamos o comando

```
git branch -r
```



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git checkout desenvolvimento
Switched to branch 'desenvolvimento'
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git push -u origin desenvolvimento
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:lucahs-hachitecnologia/curso-git.git
 * [new branch]      desenvolvimento -> desenvolvimento
Branch desenvolvimento set up to track remote branch desenvolvimento from origin.
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]# git branch -r
  origin/desenvolvimento
  origin/master
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (desenvolvimento)]#
```

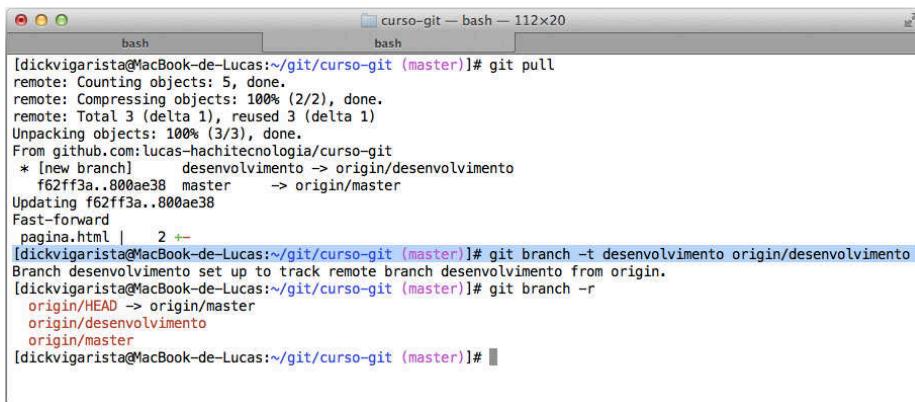
Figura 87. Enviando a branch **desenvolvimento** para o repositório remoto.

Pronto! Nossa branch **desenvolvimento** foi enviada para o repositório remoto. Agora outros desenvolvedores podem contribuir com alterações nesta branch. Perceba na **Figura 87** a saída do comando **git branch -r**, listando as branches remotas, mostrando que a branch **desenvolvimento** agora existe no repositório remoto.

5.1.7. Baixando uma **branch** do repositório remoto para o repositório local

Caso um usuário queira baixar a branch **desenvolvimento** do repositório remoto para contribuir com ela, basta atualizar seu repositório local (com o comando **git pull**) e executar o comando:

git branch -t desenvolvimento origin/desenvolvimento



```
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From github.com:lucahs-hachitecnologia/curso-git
 * [new branch]      desenvolvimento -> origin/desenvolvimento
   f62ff3a..800ae38 master      -> origin/master
Updating f62ff3a..800ae38
Fast-forward
  pagina.html |  2 ++
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git branch -t desenvolvimento origin/desenvolvimento
Branch desenvolvimento set up to track remote branch desenvolvimento from origin.
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]# git branch -r
  origin/HEAD -> origin/master
  origin/desenvolvimento
  origin/master
[dickvigarista@MacBook-de-Lucas:~/git/curso-git (master)]#
```

Figura 88. Usuário Dick Vigarista baixando a branch **desenvolvimento** do repositório remoto para o repositório local.

5.1.8. Exercício

Agora que você aprendeu a trabalhar com **branch**, vamos colocar em prática:

1. Crie uma branch chamada **desenvolvimento**.
2. Adicione um arquivo chamado **estilo.css** na nova branch com o seguinte conteúdo:

```
h1 {
  color: red;
}

p {
  color: blue;
}
```

3. Aplique a folha de estilo (**estilo.css**) que criamos ao arquivo **pagina.html**, conforme código abaixo:

```
...
<head>
    <title>Curso de Git</title>
    <link rel="stylesheet" type="text/css" href="estilo.css" >
</head>
...
```

4. Faça o commit das alterações feitas na branch **desenvolvimento**.

5. Volte para a branch **master** e adicione um novo parágrafo no arquivo **pagina.html** com o seguinte conteúdo:

```
...
<p>O Git é um SCM distribuído.</p>
...
```

6. Execute o commit na branch **master** com as alterações realizadas.

7. Agora altere novamente o arquivo **pagina.html**, corrigindo o bug na tag de parágrafo, modificando seu conteúdo de:

```
...
<p>O Git é um SCM distribuído.</p>
...
```

para

```
...
<p>O Git é um SCM distribuído.</p>
...
```

8. Realize o commit na branch **master** com a correção do bug.

9. Volte para a branch **desenvolvimento** e altere o arquivo **estilo.css**, adicionando o recurso de centralizar os cabeçalhos, deixando-o com o seguinte conteúdo:

```
h1 {
    color: red;
    text-align: center;
}
p {
    color: blue;
}
```

10. Realize o commit das alterações na branch **desenvolvimento**.

11. Realize o rebase na branch **desenvolvimento** para puxar a base de commits atualizada da branch **master**.

12. Envie os commits da branch **desenvolvimento** para a branch **master** local.

13. Envie todas as atualizações da branch **master** para o repositório remoto.

5.2. Etiquetando nosso código com tags

Quando estamos trabalhando em um projeto de software, um processo comum é a geração de uma nova release, onde o código estável e homologado do projeto é compilado e publicado em produção. Muitas vezes é preciso ter guardado o código fonte desta versão, caso em um determinado momento futuro decidimos compilar a aplicação exatamente como ela estava nessa versão específica, ou então para verificarmos uma particularidade no código desta versão. No Git isto é possível através das **tags**, que nos possibilita definir uma etiqueta (como “versão v1.0”, por exemplo) a um determinado ponto em nosso projeto.

Outra grande vantagem no uso de *tags* é que podemos comparar a diferença entre uma *tag* e outra, verificando, por exemplo, o que mudou no sistema da versão v1.0 pra versão v2.0.

5.2.1. Criando uma tag

Criar uma *tag* no Git é simples, e podemos fazer isto com um simples comando:

```
git tag -am "<MENSAGEM>" <NOME DA TAG>
```

substituindo **<MENSAGEM>** por uma mensagem sobre a versão que está sendo etiquetada, e **<NOME DA TAG>** pelo nome que deseja atribuir à *tag*.

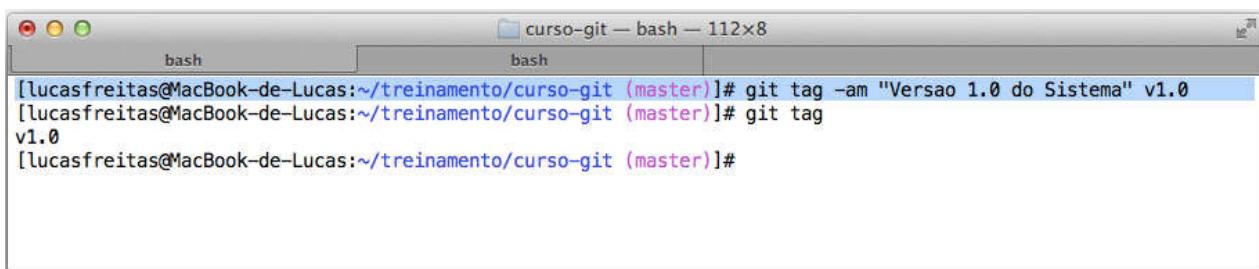
#Fica a dica

- Para listar as *tags* do nosso repositório, usamos o comando:

```
git tag
```

Vamos criar uma *tag* para etiquetar a versão 1.0 do nosso projeto, usando o comando:

```
git tag -am "Versão 1.0 do Sistema" v1.0
```



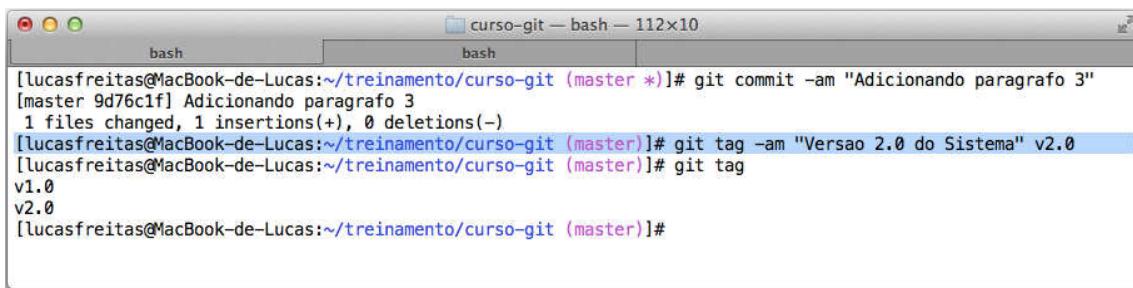
```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git tag -am "Versao 1.0 do Sistema" v1.0
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git tag
v1.0
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 89. Criando uma *tag* para identificar a Versão 1.0 do nosso projeto.

Para mostrar melhor o uso de uma *tag*, vamos realizar uma alteração na branch **master** e chamar o novo estado do projeto de **Versão 2.0**. Para isto, iremos adicionar mais um parágrafo em nosso arquivo **pagina.html**, com o seguinte conteúdo:

```
...
<p>O Git foi escrito em linguagem C.</p>
...
```

Realizamos o commit com a nova alteração e criamos uma *tag* com a etiqueta v2.0.



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master*)]# git commit -am "Adicionando paragrafo 3"
[master 9d76c1f] Adicionando paragrafo 3
 1 files changed, 1 insertions(+), 0 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git tag -am "Versao 2.0 do Sistema" v2.0
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git tag
v1.0
v2.0
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 90. Criando uma tag para identificar a Versão 2.0 do nosso projeto.

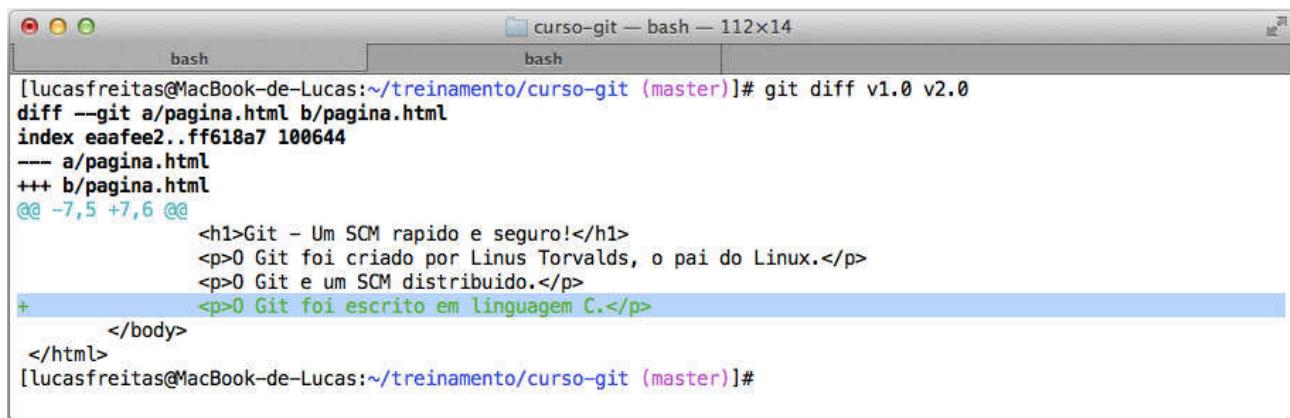
Agora temos duas tags em nosso repositório. Uma identificando a **Versão 1.0** e outra identificando a **Versão 2.0** do nosso projeto. Caso seja necessário voltar à Versão 1.0, para verificar como estava o código nessa versão ou mesmo para gerar novamente uma release com essa versão, basta realizar um **checkout** para a tag **v1.0**, usando o comando:

```
git checkout v1.0
```

Podemos também verificar a diferença entre uma tag e outra, usando o comando:

```
git diff <TAG 1> <TAG 2>
```

Veja a saída do **diff** mostrando a diferença entre as Versões **1.0** e **2.0** do nosso sistema:



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git diff v1.0 v2.0
diff --git a/pagina.html b/pagina.html
index eaafef2..ff618a7 100644
--- a/pagina.html
+++ b/pagina.html
@@ -7,5 +7,6 @@
<h1>Git - Um SCM rapido e seguro!</h1>
<p>O Git foi criado por Linus Torvalds, o pai do Linux.</p>
<p>O Git é um SCM distribuído.</p>
+ <p>O Git foi escrito em linguagem C.</p>
</body>
</html>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 91. Verificando a diferença entre as Versões 1.0 e 2.0 do nosso projeto, com o comando *git diff*.

Verifique a saída do **diff** mostrando que a diferença entre as duas versões é exatamente a linha:

```
+ <p>O Git foi escrito em linguagem C.</p>
```

que adicionamos na **Versão 2.0** da nossa página. O “**+**” significa que a linha foi adicionada. Caso a linha tivesse sido removida da Versão 1.0 pra 2.0 o símbolo seria “**-**”.

5.2.2. Exercício

Agora que aprendemos a trabalhar com tags, vamos colocar em prática:

1. Crie uma tag na branch **master** com a etiqueta “**v1.0**”.
2. Faça uma alteração no arquivo **pagina.html**, adicionando um novo parágrafo com o conteúdo:

```
... <p>O Git foi escrito em linguagem C.</p>
...
```

e realize o commit desta alteração.

3. Crie uma nova *tag* com a etiqueta “**v2.0**”.
4. Verifique a diferença entre as duas *tags* que você criou, usando o **diff**.

5.3. Usar o **diff** faz toda a diferença

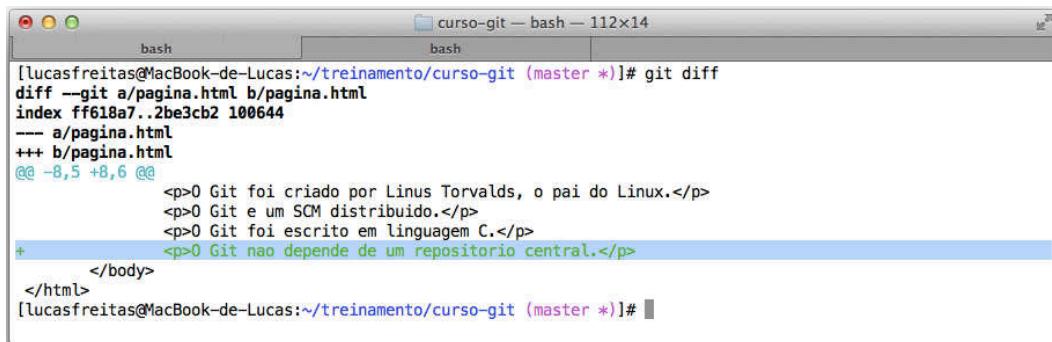
Como vimos anteriormente, o **diff** pode ser usado para verificar a diferença de conteúdo entre **tags**, mostrando o resultado no formato de um **patch**. Mas o **diff** vai muito além.

5.3.1. Mostrando as alterações realizadas no *working directory*

Podemos usar o **diff** também para verificar as alterações realizadas em nosso *working directory* que ainda não foram commitadas, apenas digitando o comando:

git diff

Veja a saída do comando **git diff** após inserirmos um novo parágrafo em nosso arquivo **pagina.html**:



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git diff
diff --git a/pagina.html b/pagina.html
index ff618a7..2be3cb2 100644
--- a/pagina.html
+++ b/pagina.html
@@ -8,5 +8,6 @@
<p>0 Git foi criado por Linus Torvalds, o pai do Linux.</p>
<p>0 Git é um SCM distribuído.</p>
<p>0 Git foi escrito em linguagem C.</p>
+ <p>0 Git não depende de um repositório central.</p>
</body>
</html>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]#
```

Figura 92. Comando **git diff** mostrando as alterações realizadas em nosso *working directory* que ainda não foram commitadas.

5.3.2. Mostrando a diferença entre dois commits

Também podemos utilizar o **diff** para verificar a diferença entre um commit e outro, através da hash identificadora do commit, usando a sintaxe:

git diff <HASH 1> <HASH 2>

Veja a saída do comando **diff** mostrando a diferença entre dois commits realizados:



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git diff e92d55338271d3bb2b22d27dd59a665505852
af1 9d76c1fa10f220ca08da4c9bdd1a69dbb7dfd62c
diff --git a/pagina.html b/pagina.html
index 17c7f1a..ff618a7 100644
--- a/pagina.html
+++ b/pagina.html
@@ -5,7 +5,8 @@
</head>
<body>
<h1>Git - Um SCM rápido e seguro!</h1>
- <p>0 Git foi criado por Linus Torvalds</p>
+ <p>0 Git foi criado por Linus Torvalds, o pai do Linux.</p>
<p>0 Git é um SCM distribuído.</p>
+ <p>0 Git foi escrito em linguagem C.</p>
</body>
</html>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]
```

Figura 93. Comando **git diff** mostrando diferença entre dois commits realizados em nosso repositório.

5.3.3. Mostrando a diferença entre o commit atual e commits anteriores

O Git também disponibiliza uma maneira fácil para verificar a diferença entre um commit anterior específico em relação ao commit atual. Vamos verificar por exemplo a diferença entre o commit atual e três commits atrás, usando o comando:

```
git diff HEAD~3
```

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git diff HEAD~3
diff --git a/pagina.html b/pagina.html
index 17c7f1a..ff618a7 100644
--- a/pagina.html
+++ b/pagina.html
@@ -5,7 +5,8 @@
 </head>
 <body>
     <h1>Git - Um SCM rápido e seguro!</h1>
-    <p>0 Git foi criado por Linus Torvalds</p>
+    <p>0 Git foi criado por Linus Torvalds, o pai do Linux.</p>
-    <p>0 Git é um SCM distribuído.</p>
+    <p>0 Git foi escrito em linguagem C.</p>
</body>
</html>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 94. Comando `git diff` mostrando diferença entre dois commits realizados em nosso repositório.

#Fica a dica

- A palavra **HEAD** é utilizada no Git como um ponteiro para a *branch* atual. Portanto lembre-se: **HEAD não** é uma *branch*, mas apenas um ponteiro para a *branch* atual.

5.3.4. Exercício

Agora que você aprendeu a trabalhar com o `diff`, vamos colocar em prática.

1. Realize uma alteração no arquivo ***página.html***, na branch ***master***, adicionando um parágrafo com o seguinte conteúdo:

```
... <p>O Git não depende de um repositório central.</p>
```

mas **NÃO** faça o commit desta alteração.

2. Verifique as diferenças realizadas no *working directory* e que ainda não foram committedas.
3. Verifique a diferença entre dois commits quaisquer, realizados em sua branch ***master***.
4. Verifique a diferença entre o commit atual e três commits atrás.

5.4. Desfazendo alterações

Quando estamos trabalhando com controle de versão em um projeto de software é comum às vezes realizarmos commits por acidente, ou incompletos, e querermos desfazê-los, ou mesmo voltar a um estado anterior do nosso código fonte. Para resolver este problema o Git possui recursos que nos permite desfazer alterações no *working directory*, no *Index* ou no **HEAD**. Outro recurso interessante é a possibilidade de podermos procurar por um *bug* commitado no repositório e corrigí-lo.

5.4.1. Descartando alterações no *working directory*

Se alguma alteração foi realizada em um arquivo no *working directory*, mas por algum motivo você decide desfazer estas alterações e voltar ao estado em que este arquivo estava no último commit, é possível fazê-lo usando o comando:

```
git checkout <ARQUIVO>
```

Substituindo **<ARQUIVO>** pelo nome do arquivo que deseja desfazer as alterações.

Para ver seu funcionamento, vamos alterar o arquivo **pagina.html** adicionando o seguinte parágrafo:

```
...<p>O Git nao depende de um repositorio central.</p>...
```

Após ter alterado o arquivo **pagina.html** podemos desfazer suas alterações e voltar ao estado do último commit, usando o comando:

```
git checkout pagina.html
```

The screenshot shows a terminal window titled 'curso-git' with two tabs: 'bash' and 'bash'. The terminal output is as follows:

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git diff pagina.html
diff --git a/pagina.html b/pagina.html
index ff618a7..2be3cb2 100644
--- a/pagina.html
+++ b/pagina.html
@@ -8,5 +8,6 @@
      <p>O Git foi criado por Linus Torvalds, o pai do Linux.</p>
      <p>O Git é um SCM distribuído.</p>
      <p>O Git foi escrito em linguagem C.</p>
+     <p>O Git não depende de um repositório central.</p>
   </body>
</html>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git checkout pagina.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git diff pagina.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 95. Desfazendo as alterações realizadas no arquivo *pagina.html* em nosso *working directory*.

Perceba, no exemplo, que usamos o **diff** para verificar as alterações realizadas no arquivo **pagina.html**, no *working directory*, antes de descartar as alterações. Logo em seguida, após o descarte, usamos o **diff** novamente e verificamos que realmente as alterações do arquivo **pagina.html** foram descartadas.

#Fica a dica

Lembre-se que já usamos o comando **git checkout** anteriormente, mas para alternar entre **branches** e entre **tags**. O **git checkout** tem várias utilidades, e pode ser usado para:

- Alternar entre **branches**, usando a sintaxe:

```
git checkout <NOME DA BRANCH>
```

- Alternar entre **tags**, usando a sintaxe:

```
git checkout <NOME DA TAG>
```

- Desfazer alterações em um arquivo no *working directory*, usando a sintaxe:

```
git checkout <ARQUIVO>
```

- Navegar para um **commit** específico, usando a sintaxe:

```
git checkout <HASH DO COMMIT>
```

5.4.1.1. Exercício

Agora que aprendemos a descartar alterações no *working directory*, vamos colocar em prática.

1. Altere o arquivo ***página.html*** adicionando o seguinte parágrafo:

```
... <p>O Git não depende de um repositório central.</p>
...
```

mas **NÃO** adicione esta alteração ao *Index*, e nem faça o commit.

2. Descarte a alteração que você realizou no ***working directory***.

5.4.2. Descartando alterações no *Index*

Assim como no *working directory*, é possível também descartar alterações enviadas à *staging area (Index)*, mas o processo é um pouco diferente. Para descartar alterações no ***Index***, usamos a seguinte sintaxe:

```
git reset HEAD <ARQUIVO>
```

Para exemplificar, vamos alterar o arquivo ***página.html***, adicionando novamente o parágrafo:

```
... <p>O Git não depende de um repositório central.</p>
...
```

Após salvar o arquivo com a alteração e adicioná-lo ao ***Index*** (usando o comando ***git add página.html***), ele está pronto para ser committed. Porém, em alguns casos, podemos mudar de idéia e desfazer a alteração, caso necessário. Mesmo tendo adicionado nossas alterações ao ***Index***, decidimos então desfazê-la, usando o comando:

```
git reset HEAD página.html
```

The screenshot shows a terminal window with the title 'curso-git — bash — 105x21'. The terminal output is as follows:

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git add página.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master +)]# git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   página.html
#
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master +)]# git reset HEAD página.html
Unstaged changes after reset:
M     página.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]# git status
# 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:   página.html
#
no changes added to commit (use "git add" and/or "git commit -a")
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]#
```

Figura 96. Desfazendo alterações no arquivo *página.html* adicionadas ao *Index*.

Perceba, no exemplo da **Figura 96**, que adicionamos nossa alteração feita no arquivo ***página.html*** e a adicionamos ao ***Index***. Logo em seguida, a saída do comando ***git status*** nos informa que o arquivo foi modificado e já está no ***Index***, pronto para ser committed. Porém mudamos de idéia e desfazemos a alteração com o comando ***git reset HEAD página.html***. Logo em seguida a saída do ***git status*** nos mostra que as alterações já **NÃO** estão mais no ***Index***, e sim no *working directory*.

Como você pôde perceber, quando desfazemos uma alteração no **Index**, esta volta para o *working directory*. Se desejar desfazer esta alteração também no *working directory*, basta realizar o procedimento para descartar as alterações no *working directory*, conforme aprendemos anteriormente neste curso.

5.4.2.1. Exercício

Agora que você aprendeu a descartar alterações no *Index*, vamos colocar em prática.

1. Altere o arquivo **pagina.html**, adicionando o seguinte parágrafo:

```
... <p>O Git nao depende de um repositorio central.</p>
...
```

e adicione a alteração ao **Index**, **SEM** fazer o commit.

2. Descarte a alteração que você realizou no **Index**.
3. Descarte a alteração também no **working directory**.

5.4.3. Descartando alterações do *último commit*

Vimos anteriormente como descartar alterações no *working directory* e no *Index*, porém muitas vezes acabamos por commitar alterações indevidas, ou mudamos de idéia e decidimos desfazer alterações após o commit. O Git nos permite desfazer o último commit, ou seja, deixar tudo como estava de acordo com o penúltimo commit realizado. Para que isto seja feito, basta usar a sintaxe:

```
git reset <HASH DO PENÚLTIMO COMMIT>
```

Seguindo nosso projeto, vamos novamente adicionar em nosso arquivo **pagina.html** o parágrafo:

```
... <p>O Git nao depende de um repositorio central.</p>
...
```

Mas desta vez adicionamos as alterações no *working directory* e a commitamos. Logo em seguida, por algum motivo, mudamos de idéia e queremos desfazer o commit realizado. Veja o processo sendo realizado na **Figura 97**.

The screenshot shows a terminal window with two tabs: 'bash' and 'bash'. The terminal content is as follows:

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master +)]# git commit -m "Adicionando paragrafo 4"
[master 3df44e8] Adicionando paragrafo 4
 1 files changed, 1 insertions(+), 0 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git status
# On branch master
nothing to commit (working directory clean)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit 3df44e8f5c423fafb64b01309f96ca83224aea2b
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Sat Apr 14 12:26:48 2012 -0300

        Adicionando paragrafo 4

commit 9d76c1fa10f220ca08da4c9bdd1a69dbb7dfd62c
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Thu Apr 12 20:59:17 2012 -0300

        Adicionando paragrafo 3

commit 800ae38778cd47f9116851ab6d058fb175880741
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date:   Wed Apr 11 22:04:42 2012 -0300

        Modificando o paragrafo 1 para o seguinte conteudo: O Git foi criado por Linus Torvalds, o pai do Linux.
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git reset 9d76c1fa10f220ca08da4c9bdd1a69dbb7dfd62c
Unstaged changes after reset:
M    pagina.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master *)]#
```

Figura 97. Desfazendo alterações do último commit.

No exemplo da **Figura 97** executamos um **git reset** passando como parâmetro a hash do penúltimo commit, pois queremos voltar ao estado em que o repositório se encontrava no penúltimo commit, descartando o último commit.

Se executarmos o **git log**, percebemos que o último commit realmente foi descartado. Mas quando executamos um **git status** percebemos que as alterações do último commit, que descartamos, foram revertidas para o **working directory**. Portanto, se desejar descartar essas alterações também no **working directory**, basta realizar o procedimento para descartar as alterações no **working directory**, conforme aprendemos anteriormente neste curso.

5.4.3.1. Exercício

Agora que você aprendeu a descartar alterações do *último commit*, vamos colocar em prática.

1. Altere o arquivo **pagina.html**, adicionando o seguinte parágrafo:

```
...<p>O Git nao depende de um repositorio central.</p>...
```

e faça o commit.

2. Descarte a alteração que você realizou neste **último commit**.

3. Descarte a alteração também no **working directory**.

5.4.4. Descartando alterações de um *commit antigo*

A opção de descartar commits, usando o **git reset**, funciona perfeitamente para o último commit, mas não é bom usá-la em commits mais antigos. Descartar commits mais antigos é uma tarefa um pouco delicada, ainda mais se esses commits já foram enviados para um repositório remoto correndo o risco de outros desenvolvedores já os terem baixado. Ainda assim o GIT dispõe de um recurso capaz de reverter um commit mais antigo. Trata-se do comando **git revert**.

O comando **git revert** irá reverter as alterações do commit desejado e fazer um commit novo, descartando estas alterações. Para isto nosso **working directory** deve estar limpo, ou corremos o risco de perder as alterações realizadas nele. Uma boa alternativa é usar a opção "**-n**" para que as alterações sejam revertidas a adicionadas no **working directory**.

Para exemplificar, vamos realizar uma alteração no cabeçalho do nosso arquivo **pagina.html** para o seguinte conteúdo:

```
...<h1>Git - O SCM feito da maneira correta!</h1>...
```

e logo em seguida fazemos o commit.

Após alterar o cabeçalho e fazer o commit, vamos adicionar os seguintes parágrafos no arquivo **pagina.html**:

```
...<p>O Git nao depende de um repositorio central.</p><p>O Git foi criado inicialmente para desenvolver o Kernel do Linux.</p>...
```

e logo em seguida fazemos o commit.

Agora iremos adicionar mais um paragrafo no arquivo **pagina.html** com um link para a página oficial do Git, com o seguinte conteúdo:

```
...<p><a href="http://www.git-scm.com">Link para o site oficial do Git</a></p>...
```

e logo em seguida fazemos o commit.

Depois de realizar estes três commits, mudamos de idéia e percebemos que não queríamos alterar o cabeçalho, como fizemos há três commits atrás. Se utilizarmos o **git reset** nesta situação, perderíamos todos os commits realizados após a alteração do cabeçalho, pois o **git reset** é usado para descartar apenas o último commit, como vimos anteriormente.

Para solucionar nosso caso, de desfazer a alteração do cabeçalho que fizemos há três commits atrás, usaremos o comando **git revert**, com a seguinte sintaxe:

git revert <HASH DO COMMIT A SER DESFEITO>

Quando usamos o **git revert**, o Git abre um editor de texto para informarmos a mensagem que será usada para o commit automático que será feito desfazendo as alterações. Veja o processo de revert na **Figura 98**.

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git revert c4aff36f969b77d816567b52ad12de5e4af33a1
[master f53be0f] Revertendo a alteracao de cabecalho
 1 files changed, 1 insertions(+), 1 deletions(-)
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git log
commit f53be0fc426bf9d9349fb567cc581c1f93bbb0bc
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date: Mon Apr 16 11:38:52 2012 -0300

    Revertendo a alteracao de cabecalho

  This reverts commit c4aff36f969b77d816567b52ad12de5e4af33a1.

commit e580a60c95ec879e85d22221345ee9769c69540a
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date: Mon Apr 16 11:34:27 2012 -0300
```

Figura 98. Desfazendo alterações de um commit antigo usando o *git revert*.

Perceba, na **Figura 98**, que revertemos o commit com a alteração do cabeçalho sem afetarmos os commits posteriores. No log podemos ver a mensagem que foi inserida no commit do revert: “Revertendo a alteração de cabeçalho”. Essa mensagem foi inserida manualmente no editor de texto aberto pelo Git (apenas não mostramos esse passo).

Agora nosso cabeçalho ficou exatamente como estava no início do projeto, com o seguinte conteúdo:

```
...<h1>Git - Um SCM rapido e seguro!</h1>...
```

5.4.4.1. Exercício

Agora que você aprendeu a descartar alterações em um *commit antigo*, vamos colocar em prática:

1. Altere cabeçalho do arquivo **pagina.html**, deixando-o com o seguinte conteúdo:

```
...<h1>Git - O SCM feito da maneira correta!</h1>...
```

e faça o commit.

2. Altere novamente o arquivo **pagina.html**, adicionando os seguintes parágrafos:

```
...  
<p>O Git não depende de um repositório central.</p>  
<p>O Git foi criado inicialmente para desenvolver o Kernel do Linux.</p>  
...
```

e faça o commit.

3. Faça mais uma alteração no arquivo ***página.html***, adicionando mais o seguinte parágrafo:

```
...  
<p><a href="http://www.git-scm.com">Link para o site oficial do Git</a></p>  
...
```

e faça o commit.

4. Agora reverta (desfaça) a alteração de cabeçalho realizada no **item de número 1** deste exercício.

5.5. Procurando *bugs* em commits antigos

Vimos que o GIT tem recursos avançados para descartar commits antigos em nosso projeto. Mas imagine a situação onde um **bug** foi commitado no projeto mas não sabemos em qual commit ele foi inserido. Sabendo a data aproximada de quando o *bug* foi commitado podemos caçá-lo através de um recurso avançado do GIT, chamado **git bisect**. O **git bisect** nos exime de ter que dar o checkout commit por commit até achar o *bug* que procuramos.

Imagine, por exemplo, que um usuário telefone informando que há um *bug* no sistema, mas que há 4 dias atrás não existia esse *bug*. Com o **git bisect** podemos procurar esse *bug* pela data aproximada.

Para exemplificar, vamos fazer uma alteração no cabeçalho da nossa página, deixando-o com o seguinte conteúdo:

```
...  
<h1>Git - O SCM feito da maneira correta!<\h1>  
...
```

Realizamos o commit e logo em seguida adicionamos outro parágrafo em nossa página, com o seguinte conteúdo:

```
...  
<p><a href="http://www.git-scm.com/download">Link para download do Git</a></p>  
...
```

Realizamos novamente o commit e decidimos adicionar mais um parágrafo, com o seguinte conteúdo:

```
...  
<p><a href="http://www.git-scm.com/documentation">Link para documentação do Git</a></p>  
...
```

Finalmente, realizamos mais uma alteração na página, adicionando o parágrafo abaixo e realizando o commit:

```
...  
<p><a href="http://www.git-scm.com/about">Link para informações sobre o Git</a></p>  
...
```

Após realizar essas alterações, um usuário telefonou para o suporte e informou que a página está com um **bug**, mas que esse *bug* não existia há 4 dias atrás. Trata-se de um *bug* inserido na tag do cabeçalho, quando o alteramos, fechando a tag incorretamente com a barra invertida, ficando: **<\h1>**.

Em nosso exemplo, realizamos apenas 3 commits após a alteração que foi commitada com *bug*. Mas imagine um cenário onde dezenas de commits tenham sido realizados após a inserção de um *bug* no sistema. Ficaria bem mais difícil encontrá-lo, se não fosse o **git bisect**.

Para iniciar a busca de um *bug* com o **git bisect**, usamos o seguinte comando:

```
git bisect start
```

Quando executamos o comando acima, o Git inicia o *bisect* para procurarmos o *bug*. Como o *bug* existe de 4 dias pra cá, então a nossa branch atual, a branch **master**, está com o *bug* e devemos marcá-la como ruim, ou seja, *bugada*. Para isto, usamos o seguinte comando:

```
git bisect bad HEAD
```

O comando **git bisect bad** marca um commit como sendo ruim, ou seja, *bugado*.

Conforme nosso usuário havia dito, há 4 dias atrás o *bug* não existia na página. Imagine que o commit com a hash **<290fdfa9935512c0a945c72b50ed2925de66d0d6>** tenha sido realizado há 4 dias atrás, então pela lógica iremos marcá-lo como bom, ou seja, *não bugado*, e fazemos isso com o comando:

```
git bisect good 290fdfa9935512c0a945c72b50ed2925de66d0d6
```

Quando executamos o comando acima, estamos dizendo ao Git que o commit com a hash informada está bom, ou seja, não está *bugado*. O Git, então, irá automaticamente fazer o checkout de uma versão intermediária, entre a que marcamos como ruim e a que marcamos como bom, para testarmos. Se a versão que o Git fez o checkout automaticamente estiver com o *bug*, devemos marcá-la como ruim, com o comando **git bisect bad**. Mas se a versão estiver sem o *bug*, devemos marcá-la como boa, usando o comando **git bisect good**. O Git irá fazer checkouts automaticamente para testarmos até encontrar o commit em que o *bug* foi inserido.

Veja, na **Figura 99**, todo o processo de busca do *bug*, usando o *bisect*:

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git bisect start
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master|BISECTING)]# git bisect bad HEAD
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master|BISECTING)]# git bisect good 290fdfa9935512c0a945c72b50ed2925de66d0d6
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[8f8fcad247d6607932c5825c0160d994d9103d12] Alterando novamente o cabecalho da pagina
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((8f8fcad...)|BISECTING)]# git bisect bad
Bisecting: 0 revisions left to test after this (roughly 1 step)
[f53be0fc426bf9d9349fb567cc581c1f93bbb0bc] Revertendo a alteracao de cabecalho
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f53be0...)|BISECTING)]# git bisect good
8f8fcad247d6607932c5825c0160d994d9103d12 is the first bad commit
commit 8f8fcad247d6607932c5825c0160d994d9103d12
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date: Mon Apr 16 17:57:30 2012 -0300

        Alterando novamente o cabecalho da pagina

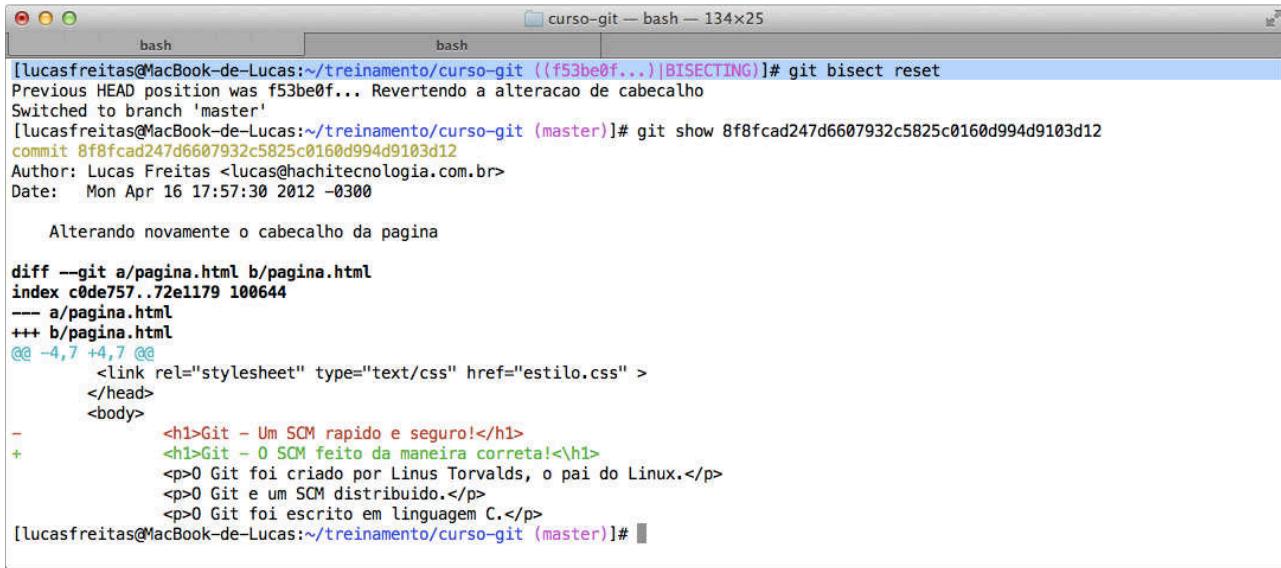
:100644 100644 c0de7577392f5d2a55328ed8e6aad9d78b832566 72e1179ebc24a11c527f64cf077f9c501adfabb4 M      pagina.html
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f53be0...)|BISECTING)]#
```

Figura 99. Procurando um *bug* no repositório usando o **git bisect**.

Todo checkout que o *bisect* nos retorna para testar, devemos marcar se ele está bom (*good*) ou ruim (*bad*), até o Git achar qual o commit em que foi inserido o *bug*. Fizemos isto no exemplo da **Figura 99**, e ao final o Git nos informou exatamente o commit em que foi inserido o *bug*.

Após o Git nos informar em qual commit foi inserido o *bug*, devemos sair do modo *bisect* e usar o *revert* para corrigir este commit. Seguindo nosso exemplo, vamos então sair do *bisect* com o comando:

```
git bisect reset
```



```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git ((f53be0f...)|BISECTING)]# git bisect reset
Previous HEAD position was f53be0f... Revertendo a alteracao de cabecalho
Switched to branch 'master'
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git show 8f8fcad247d6607932c5825c0160d994d9103d12
commit 8f8fcad247d6607932c5825c0160d994d9103d12
Author: Lucas Freitas <lucas@hachitecnologia.com.br>
Date: Mon Apr 16 17:57:30 2012 -0300

    Alterando novamente o cabecalho da pagina

diff --git a/pagina.html b/pagina.html
index c0de757..72e1179 100644
--- a/pagina.html
+++ b/pagina.html
@@ -4,7 +4,7 @@
     <link rel="stylesheet" type="text/css" href="estilo.css" >
     </head>
     <body>
-
+        <h1>Git - Um SCM rapido e seguro!</h1>
+        <h1>Git - O SCM feito da maneira correta!<\h1>
+        <p>O Git foi criado por Linus Torvalds, o pai do Linux.</p>
+        <p>O Git é um SCM distribuído.</p>
+        <p>O Git foi escrito em linguagem C.</p>
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 100. Saindo do modo *bisect* após encontrado o commit com *bug*.

Veja, na **Figura 100**, que realmente o Git nos informou corretamente o commit que estava com *bug*. O fato é comprovado na saída do **git show**, mostrando a tag do cabeçalho que foi invertida por um descuido do nosso programador.

Agora que sabemos o commit em que foi inserido o *bug*, podemos iniciar o processo de *revert* do commit, conforme aprendemos anteriormente neste curso.

#Fica a dica

- Para visualizarmos todas as alterações realizadas em um commit, podemos usar o comando:

git show <HASH DO COMMIT>

5.5.1. Exercício

Agora que você aprendeu a procurar um *bug*, usando o *git bisect*, vamos colocar em prática.

1. Altere o cabeçalho do arquivo **pagina.html**, deixando-o com o seguinte conteúdo:

```
...
<h1>Git - O SCM feito da maneira correta!<\h1>
...
```

e faça o commit. (**A tag de fechamento do cabeçalho deve ficar invertida, conforme escrito acima**).

2. Altere novamente o arquivo **pagina.html**, adicionando um parágrafo com o seguinte conteúdo:

```
...
<p><a href="http://www.git-scm.com/download">Link para download do Git</a></p>
...
```

e faça o commit.

3. Adicione mais um parágrafo no arquivo **pagina.html**, com o seguinte conteúdo:

```
...
<p><a href="http://www.git-scm.com/documentation">Link para documentacao do Git</a></p>
...
```

e faça o commit.

4. Novamente, adicione outro parágrafo no arquivo **pagina.html**, com o seguinte conteúdo:

```
... <p><a href="http://www.git-scm.com/about">Link para informacoes sobre o Git</a></p>
...
```

e faça o commit.

5. Após realizar estas 4 alterações, devemos procurar o *bug* que fez a página ficar desformatada, deixando todos os parágrafos com o mesmo estilo de um cabeçalho. Procure o *bug* na tag de cabecalho (**<h1>**) usando o **git bisect**.

6. Após encontrar o *bug*, reverta o commit para corrigí-lo.

5.6. Guardando alterações para depois

Quando estamos trabalhando em um projeto de software é comum termos que parar nosso trabalho para atender a uma nova solicitação. Imagine, por exemplo, que você tenha trabalhado em várias alterações no código mas seu chefe pede para fazer uma simples alteração no projeto. O que fazer com as alterações que você estava trabalhando? Não podemos colocar essas alterações em produção, pois ainda não estão completas, e por outro lado não queremos perdê-las e ter que refazer tudo de novo. Para resolver este problema, o Git tem um recurso chamado **git stash**, que nos permite guardar nossas alterações para usá-las depois.

Para exemplificar, vamos realizar uma alteração em nosso arquivo **pagina.html**, adicionando dois novos parágrafos:

```
... <p>Com o Git podemos trabalhar desconectados</p>
<p>O Git mantem todo o historico localmente</p>
...
```

Após realizar as alterações no *working directory*, uma nova prioridade surge, solicitando uma simples alteração no cabeçalho da página para o seguinte conteúdo:

```
... <h1>Git - Um SCM moderno e poderoso!</h1>
...
```

A alteração solicitada deverá ser feita urgentemente, mas as alterações que estávamos trabalhando não poderão ainda entrar em produção, pois não foram homologadas ou estão incompletas. Para realizar a alteração do cabeçalho sem perder nosso trabalho, usamos o seguinte comando:

git stash

Após executar o *stash*, nossas alterações serão guardadas para recuperarmos mais tarde. Agora podemos trabalhar normalmente na nova solicitação.

Realizamos a alteração do cabeçalho, conforme solicitado, e efetuamos o commit. Agora que atendemos à nova solicitação e realizamos o commit, podemos recuperar as alterações que estávamos trabalhando, usando o comando:

git stash pop

#Fica a dica

- Podemos guardar várias alterações com o **git stash**, e para listar todos os *stashes* guardados usamos o comando:

git stash list

- Por padrão o comando **git stash pop** recupera o último *stash* salvo, mas podemos recuperar algum outro *stash* salvo usando o número do *stash* conforme aparece na saída do comando **git stash list**.

5.6.1. Exercício

Agora que você aprendeu a usar o *git stash* para guardar alterações para mais tarde, vamos colocar em prática.

1. Altere o arquivo **pagina.html**, inserindo os seguintes parágrafos:

```
...<p>Com o Git podemos trabalhar desconectados</p>
<p>O Git mantem todo o histórico localmente</p>
```

mas **NÃO** faça o commit.

2. Guarde as alterações com o **git stash**.

3. Altere o cabeçalho do arquivo **pagina.html**, deixando-o com o seguinte conteúdo:

```
...<h1>Git - Um SCM moderno e poderoso!</h1>
```

e faça o commit.

4. Recupere as alterações de parágrafo, salvas pelo *stash*.

5. Faça o commit das alterações de parágrafo recuperadas do *stash*.

6 - Ferramentas gráficas do Git

Durante o curso, aprendemos como trabalhar com o Git através da linha de comando, que é a forma padrão, independente do Sistema Operacional. A vantagem de se trabalhar com a linha de comando é que podemos usufruir de todos os recursos do Git. Mesmo assim, existem ferramentas gráficas que facilitam o uso do Git, deixando sua usabilidade muito mais amigável para aqueles que não gostam muito de trabalhar em linha de comando. Vamos indicar algumas destas ferramentas.

6.1. Ferramentas gráficas para Mac

Para Mac, existem algumas opções para facilitar o uso do Git através de ferramentas gráficas.

6.1.1. GitX

Uma das opções é o **GitX**, uma interface gratuita e bastante simples que facilita o uso das funcionalidades básicas do Git.

Para baixá-lo, basta acessar o link: <http://gitx.frim.nl/index.html>

Veja algumas imagens do **GitX** em ação:

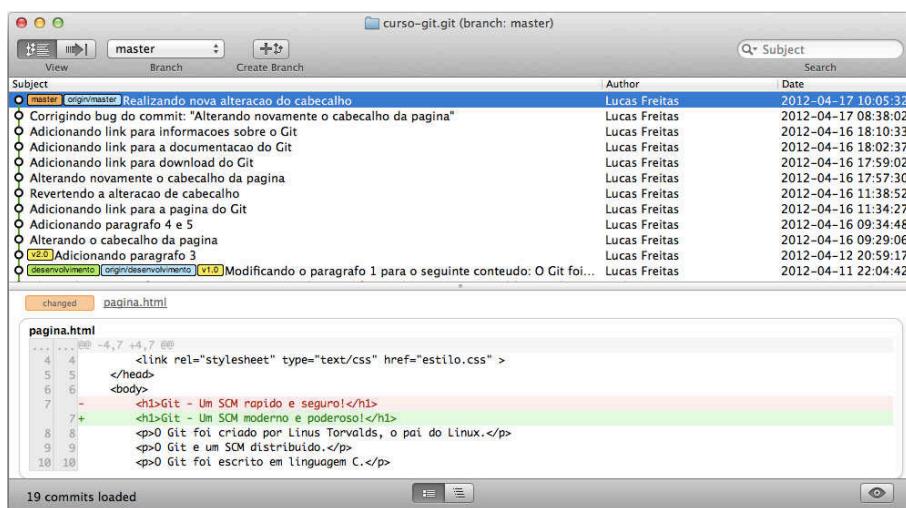


Figura 101. Log de commits de um repositório do Git sendo visualizada no *GitX*.

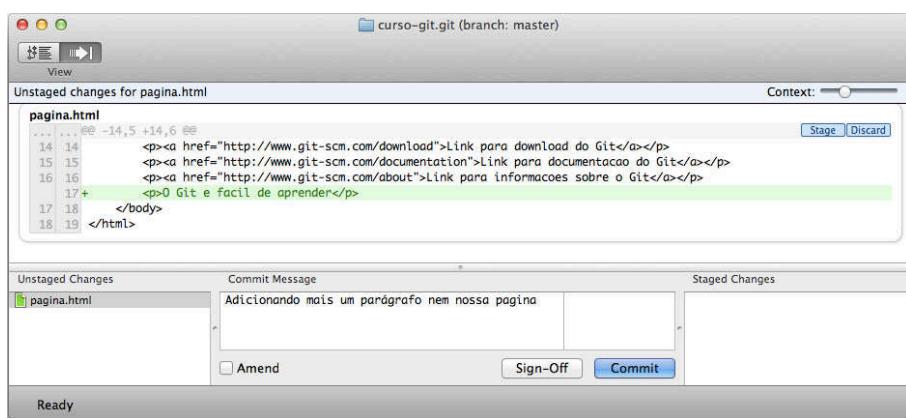


Figura 102. Realizando um commit em um repositório do Git através do *GitX*.

6.1.2. SmartGit

Outra ferramenta muito interessante para trabalhar com o Git em modo gráfico, no Mac, é o **SmartGit**. Esta ferramenta vai além do básico do Git, oferecendo funcionalidades avançadas, como fazer um *revert* de um commit, sincronizar com repositórios remotos e inclusive interagir com o GitHub.com. O **SmartGit**, apesar de ser pago, possui uma versão gratuita para fins não comerciais. Para baixá-lo, basta acessar o link: <http://www.synteko.com/smartgit/index.html>

Veja algumas imagens do **SmartGit** em ação:

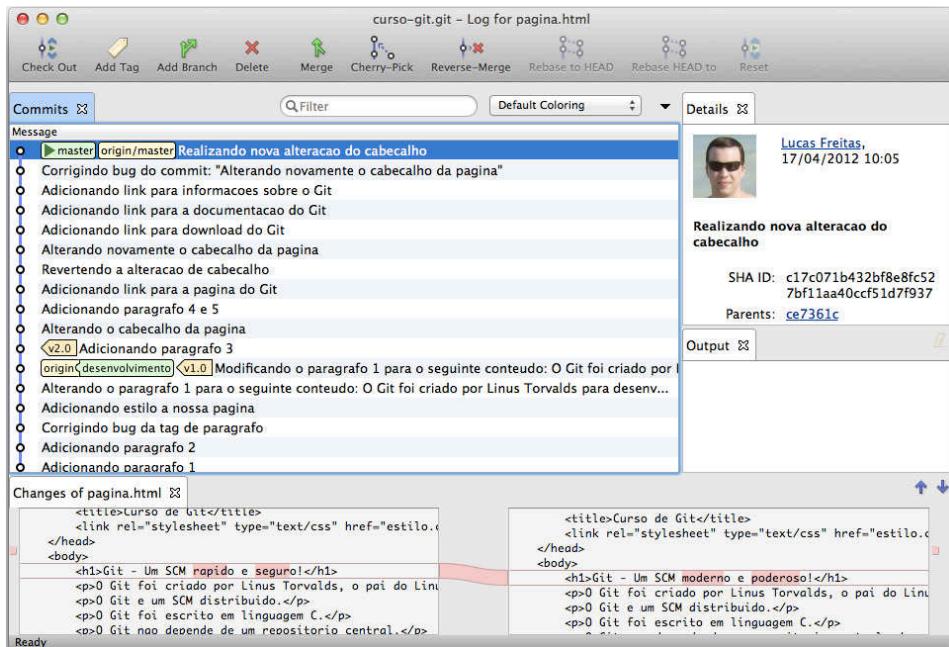


Figura 103. Log de commits de um repositório do Git sendo visualizado no *SmartGit*.

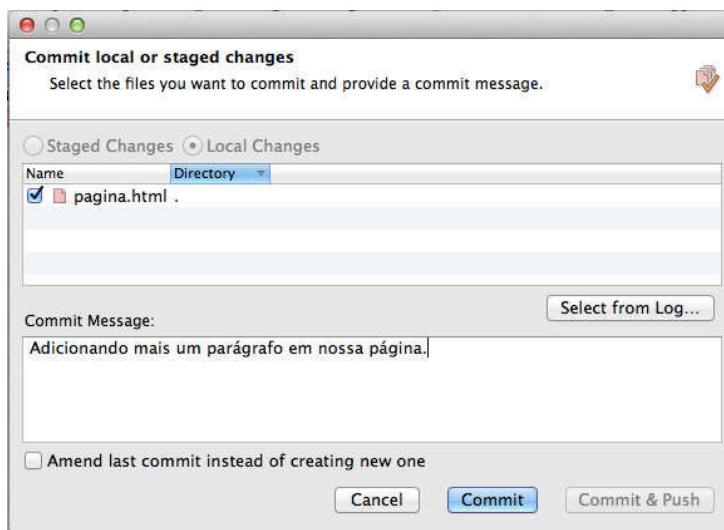


Figura 104. Realizando um commit em um repositório do Git através do *SmartGit*.

6.1.3. Tower

O **Tower** é uma das ferramentas gráficas mais poderosas para usar o Git no Mac. Possui suporte a diversas funcionalidades avançadas do Git, como fazer o *revert* de um commit, sincronizar com repositórios remotos, sincronizar com o GitHub.com, trabalhar com *stash*, e muitas outras opções. O **Tower** é uma ferramenta paga e possui uma versão trial para 30 dias de avaliação. Para baixá-lo, basta acessar o link: <http://www.git-tower.com>

GIT - Controle de Versão com Rapidez e Segurança

Veja algumas imagens do **Tower** em ação:

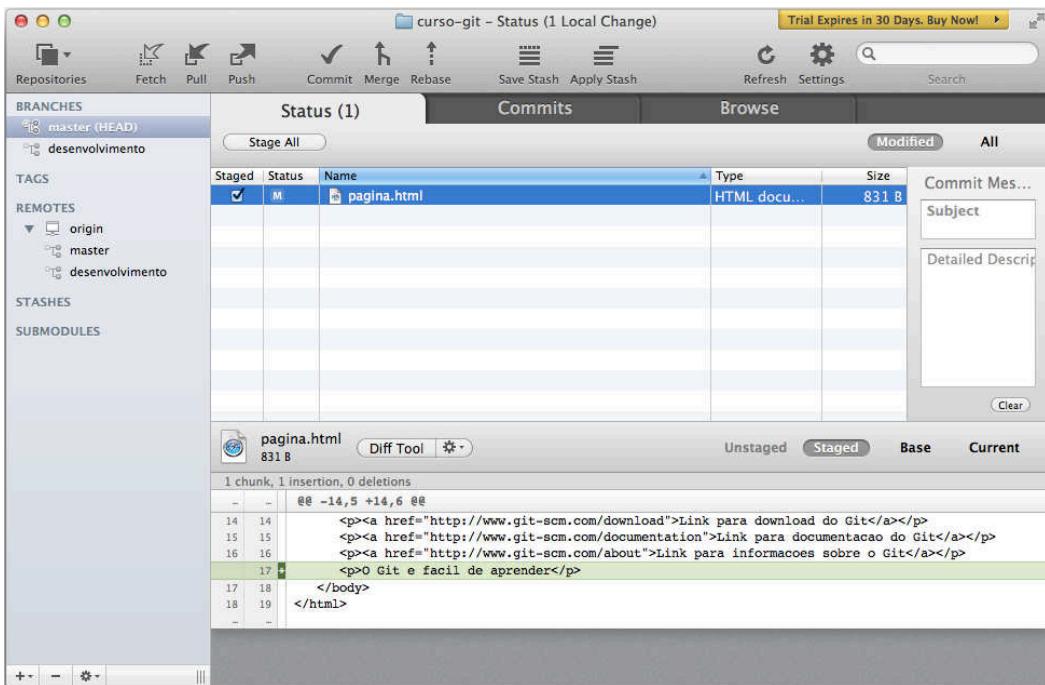


Figura 105. Visualizando o *status* da branch master em um repositório do Git, através do *Tower*.

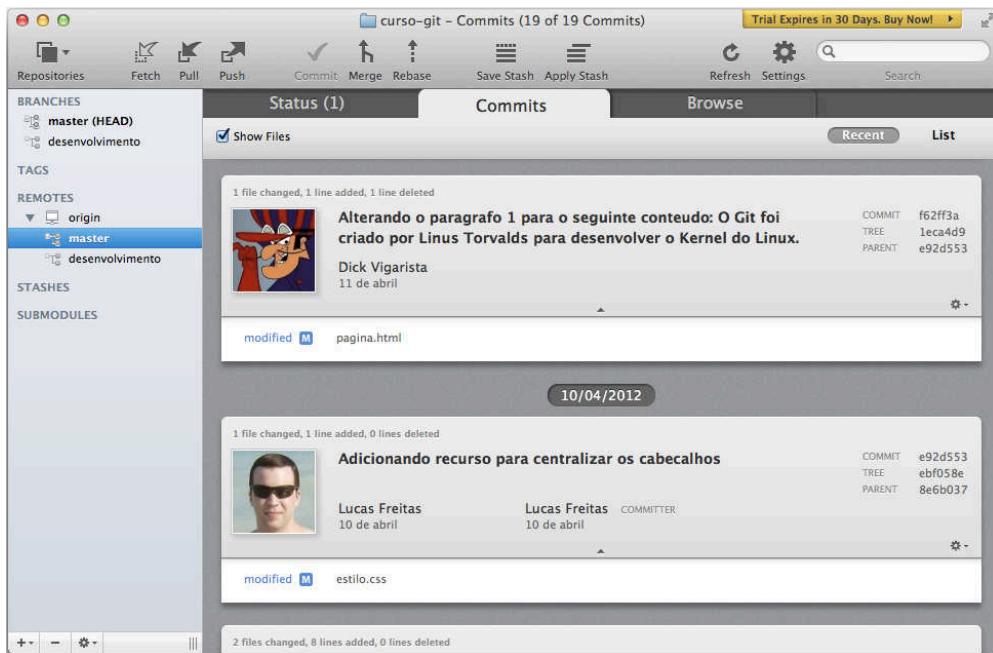


Figura 106. Log de commits de um repositório do Git sendo visualizado no *Tower*.

6.2. Ferramentas gráficas para Linux

Existem diversas ferramentas gráficas para trabalhar com o Git no Linux. Dentre elas, as mais usadas são o **Git Cola** e o **GitK**.

6.2.1. Git Cola

Uma ferramenta gráfica muito interessante para usar o Git no Linux é o **Git Cola**, com uma interface fácil de usar e atende bem às funcionalidades básicas do Git, compreendendo também alguns recursos mais avançados. Para baixá-lo, basta acessar o link: <http://git-cola.github.com>

Veja algumas imagens do **Git Cola** em ação:

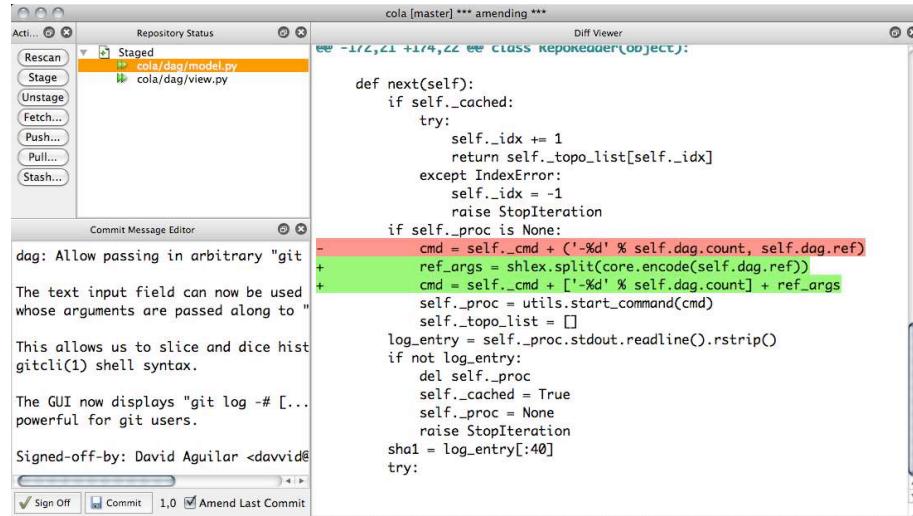


Figura 107. Realizando um commit em um repositório do Git, através do **Git Cola**.

6.2.2. GitK

Outra opção para usar algumas funcionalidades básicas do Git, em modo gráfico, é o **GitK**.

Veja uma imagem do **GitK** em ação:

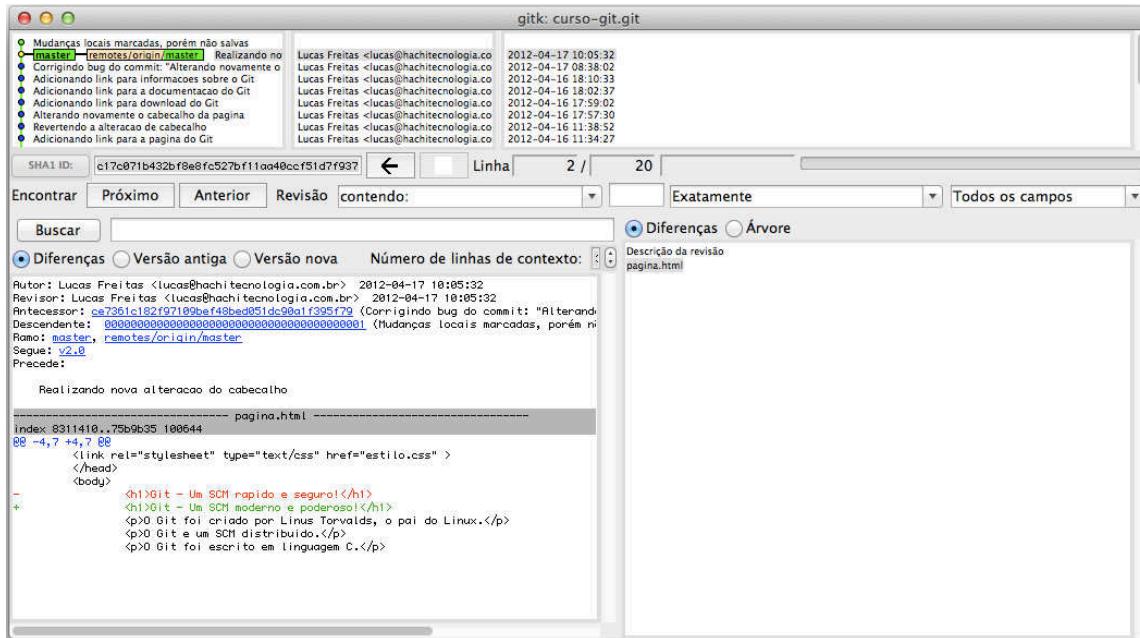


Figura 108. Log de commits de um repositório do Git sendo visualizado no **GitK**.

6.3. Ferramenta gráfica para Windows

No Windows, a mais completa ferramenta gráfica gratuita para trabalhar com o Git é o **TortoiseGit**. O **TortoiseGit** tem uma interface amigável e simples, e é totalmente integrado ao Windows Explorer. Para baixá-lo, basta acessar o link:

<http://code.google.com/p/tortoisegit/downloads/list>

Veja algumas imagens do **TortoiseGit** em ação:

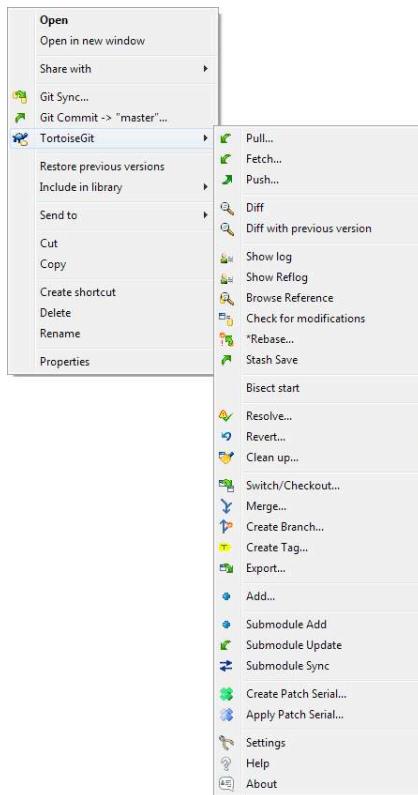


Figura 109. Menu do TortoiseGit interagindo com um repositório, no Windows Explorer.

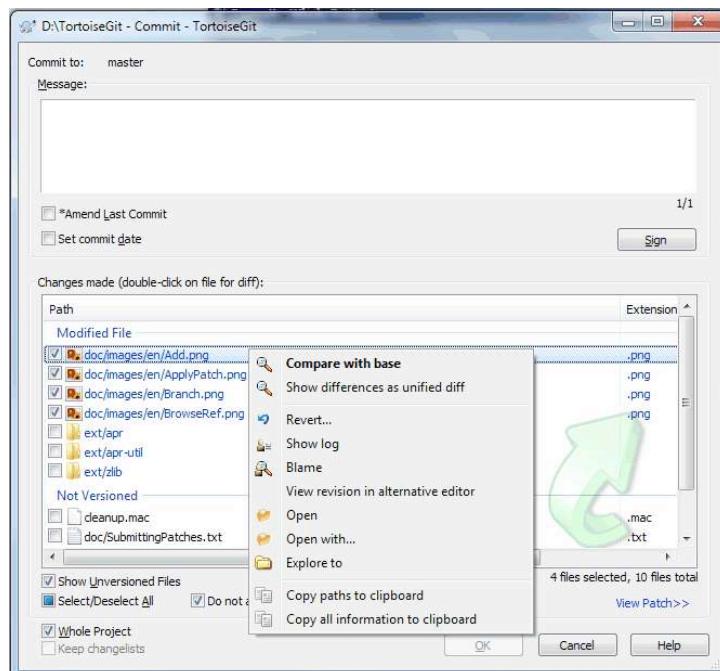


Figura 110. Tela de commit do TortoiseGit.

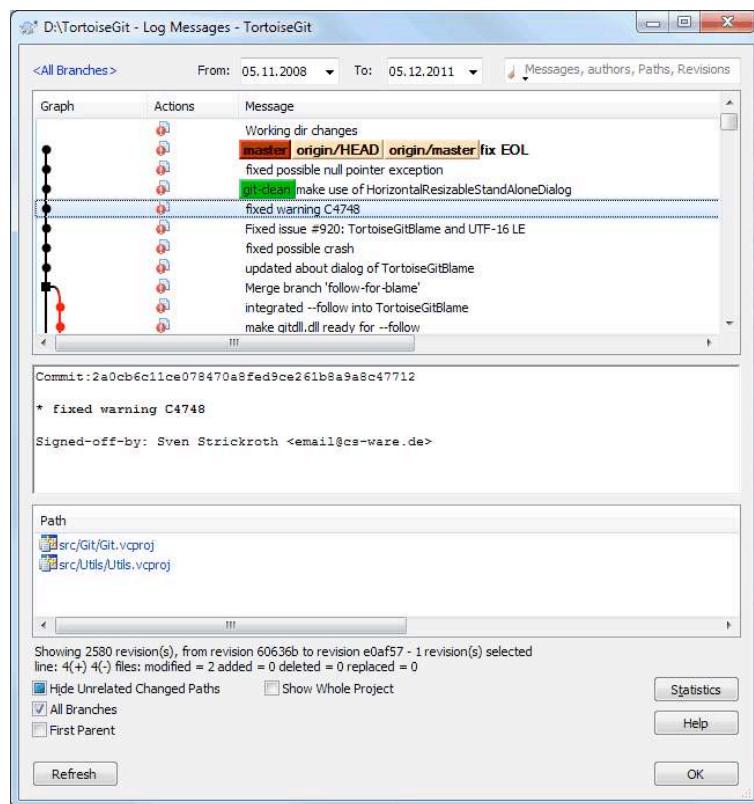


Figura 111. Visualizando Logs de commit no TortoiseGit.

7 - Apêndice

7.1. Montando um repositório centralizado

Vimos durante o curso que o Git trabalha localmente, e vimos também que é possível sincronizar com repositórios remotos através da rede. Usar o GitHub para simular um servidor centralizado em projetos que usam o Git é muito interessante, porém, em alguns casos, precisamos manter um repositório centralizado e privado, onde apenas um grupo restrito de desenvolvedores tenham acesso a ele.

Montar um repositório centralizado com o Git é simples. Além do seu próprio protocolo, o Git tem suporte a outros protocolos para a comunicação entre repositórios. Veja alguns protocolos que o Git suporta:

- Git
- SSH
- HTTP(S)
- FTP(S)

É possível ver a lista completa dos protocolos suportados pelo Git na página do manual do ***push***, digitando o comando:

```
git help push
```

O Git não possui sistema de autenticação próprio, deixando esta competência a cargo do protocolo de comunicação escolhido. Veremos como montar um repositório centralizado com o Git no Mac, Linux e Windows.

7.1.1. Montando um repositório centralizado no Mac e Linux

Iremos utilizar o protocolo SSH para a comunicação com o repositório centralizado, já que o servidor SSH já vem instalado nativamente no Mac e na maioria das distribuições do Linux. Faremos isto em poucos passos:

1. Crie um usuário chamado “**git**” no servidor centralizado, definindo seu diretório padrão para **/git**, e defina sua senha;
2. Logue no servidor com o usuário “**git**” que você criou;
3. Crie um diretório chamado “**curso-git.git**” dentro do diretório home (**/git**) definido para o usuário **git**;
4. No diretório que você criou (**/git/curso-git.git**), inicie um repositório do Git com o comando: **git init --bare**

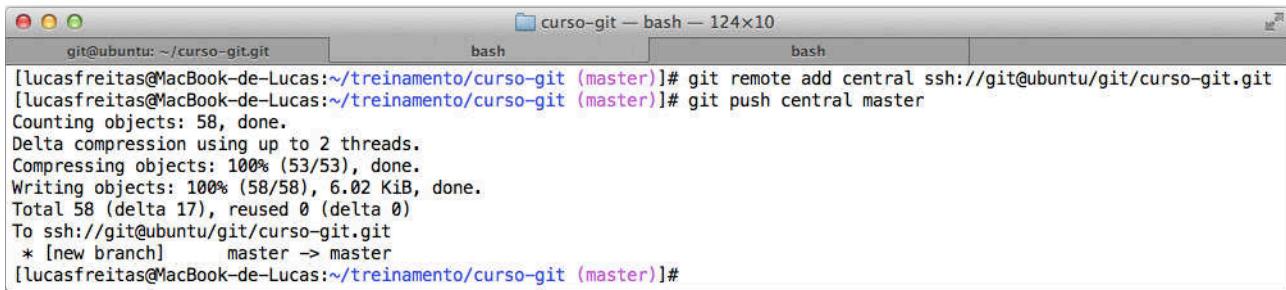
```
git@ubuntu:~/curso-git.git$ mkdir curso-git.git
git@ubuntu:~/curso-git.git$ cd curso-git.git/
git@ubuntu:~/curso-git.git$ pwd
/git/curso-git.git
git@ubuntu:~/curso-git.git$ git init --bare
Initialized empty Git repository in /git/curso-git.git/
git@ubuntu:~/curso-git.git$ ls -lahtr
total 40K
drwxr-xr-x 7 git git 4.0K 2012-04-19 11:42 ..
drwxr-xr-x 4 git git 4.0K 2012-04-19 11:42 refs
drwxr-xr-x 4 git git 4.0K 2012-04-19 11:42 objects
drwxr-xr-x 2 git git 4.0K 2012-04-19 11:42 info
drwxr-xr-x 2 git git 4.0K 2012-04-19 11:42 hooks
-rw-r--r-- 1 git git 23 2012-04-19 11:42 HEAD
-rw-r--r-- 1 git git 73 2012-04-19 11:42 description
-rw-r--r-- 1 git git 66 2012-04-19 11:42 config
drwxr-xr-x 2 git git 4.0K 2012-04-19 11:42 branches
drwxr-xr-x 7 git git 4.0K 2012-04-19 11:42 .
git@ubuntu:~/curso-git.git$
```

Figura 112. Criando um repositório do Git no servidor centralizado.

5. Adicione o repositório remoto em seu repositório local, usando o comando:

```
git remote add central ssh://git@<IP DO SERVIDOR>/git/curso-git.git
```

substituindo **<IP DO SERVIDOR>** pelo ip do seu servidor centralizado.



The screenshot shows a terminal window titled "curso-git — bash — 124x10". It displays the following command-line session:

```
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git remote add central ssh://git@ubuntu/git/curso-git.git
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]# git push central master
Counting objects: 58, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (53/53), done.
Writing objects: 100% (58/58), 6.02 KiB, done.
Total 58 (delta 17), reused 0 (delta 0)
To ssh://git@ubuntu/git/curso-git.git
 * [new branch]      master -> master
[lucasfreitas@MacBook-de-Lucas:~/treinamento/curso-git (master)]#
```

Figura 113. Adicionando o repositório remoto centralizado em nosso repositório local e fazendo a sincronização entre os dois.

#Fica a dica

- Quando criamos um repositório do Git que servirá apenas de repositório centralizado, iniciamos este repositório com a flag “**--bare**” ao executar o **git init**:

```
git init --bare
```

Pronto! Agora nosso servidor centralizado está pronto e podemos sincronizar nosso repositório local com ele através da rede, realizando *push* e *pull*.

7.1.2. Montando um repositório centralizado no Windows

Montar um repositório centralizado no Windows é bastante simples. Para isto, siga os passos:

1. Em seu usuário do Windows, crie um diretório que será usado para armazenar os repositórios do Git, por exemplo:

```
c:\git;
```

2. Compartilhe o diretório **c:\git** através do compartilhamento do windows (ative autenticação do compartilhamento do Windows para assegurar que todos se autentiquem para usar o repositório);

3. Abra o prompt de comando, entre no diretório **c:\git** que criamos:

```
cd c:\git
```

4. Inicie o repositório remoto com o comando:

```
git init --bare
```

Pronto! Nosso repositório remoto está criado. Para usá-lo em outras máquinas Windows, basta montar o compartilhamento nestas máquinas e usar o repositório localmente no diretório que você montou pela rede.

