

Artigo

Principais características do Git

Aprenda neste artigo as principais características do Git e como utilizá-las sem complicações.



Voltar

Suporte ao aluno

Anotar

Marcado como concluído



Artigos > Java > Principais características do Git



236



Como usar o Git?

Seja no desenvolvimento de novos softwares ou na manutenção de sistemas, é comum ter mais de uma pessoa, às vezes grandes equipes, editando arquivos de código fonte. Muitas dessas edições necessitam ser realizadas paralelamente no mesmo arquivo, inclusive por pessoas em projetos diferentes. Nesses cenários, fica a dúvida: como gerenciar as contribuições de diferentes desenvolvedores ao longo do tempo? O controle de versão há muito tempo se tornou imprescindível no **desenvolvimento de software** justamente por fornecer uma solução para esses dilemas. No entanto, a simples adoção de uma **ferramenta de controle de versão** não é suficiente para o sucesso. Com o grande número de ferramentas, cada qual tecendo seus próprios conceitos e paradigmas no versionamento de artefatos, a única maneira de ser bem-sucedido nessa tarefa é conhecendo bem o que cada ferramenta oferece, saber diferenciar os conceitos por trás de cada ferramenta e definir a que melhor se encaixa no contexto em questão. Este artigo, além de **explorar o correto uso do Git** – uma das ferramentas mais populares atualmente para controle de versão.

Em muitas **equipes de desenvolvimento** há dois sentimentos que vem se tornando muito comuns: o receio de fazer alterações no código que possam quebrar outras funcionalidades e o receio de compilar um projeto com código a mais ou código a menos. Além disso, geralmente esses códigos são repletos de comentários documentando detalhadamente e de forma exaustiva aspectos que não deveriam ser registrados em comentários, tais como relacionar a inclusão de uma variável a um projeto específico.

Uma solução antiga para esses problemas é o versionamento de controle, que visa dar segurança aos programadores em relação às versões corretas a serem compiladas e fornecer comentários relacionados as alterações implementadas na própria ferramenta de controle de versão, eliminando assim a complexidade de um código poluído repleto de explicações e o receio dos desenvolvedores de estarem alterando versões erradas do sistema.

Entretanto, mesmo em projetos que fazem uso de versionamento, esses sentimentos insistem em fazer parte do cotidiano das pessoas envolvidas e não é raro a ocorrência de problemas decorrentes do controle de versão. Mesmo com ferramentas os problemas ainda ocorrem, o que evidenciam o fator humano como causa, em boa parte, devido à falta de entendimento por completo do fluxo de trabalho do controle de versão ou da ferramenta adotada pela equipe. É

comum, por exemplo, vermos pessoas acostumadas a usar o SVN sem entender seu fluxo de trabalho e **migrarem para o Git** pensando ser a mesma coisa e vice-versa. Esses equívocos são os grandes geradores de problemas em relação ao versionamento e alimentam o receio da equipe em mexer no código, especialmente no Git, onde o erro mais comum é fazer o commit pensando que as alterações serão enviadas para o servidor de controle de versão como acontece no SVN.

Tendo como ponto de partida a necessidade do entendimento, tanto conceitual quanto prático das ferramentas de controle de versão para a solução desses cenários, esse artigo abordará os diferentes tipos de controle de versão a fim de fornecer uma base conceitual para se entender o fluxo de trabalho e focará no Git, mostrando como o usá-lo da maneira correta.

Controle de Versão com Git

Boa parte das tarefas profissionais que temos no dia a dia giram em torno de uma série de interações com algum tipo de conteúdo, que varia de acordo com a função de cada indivíduo. Esse conteúdo pode assumir muitas formas, tais com dados financeiros em uma planilha, textos e relatórios em documentos de texto, apresentações e, no caso dos programadores, código fonte em arquivos de texto. As tarefas diárias dos profissionais, em geral, podem ser resumidas em um ciclo de criar conteúdo, salvar, editar e salvar o conteúdo novamente, como ilustra a **Figura 1**.

Figura 1.Ciclo de vida de arquivos nas tarefas diárias da maioria dos profissionais

Esse ciclo pode se tornar muito complexo em pouco tempo em relação ao número de pessoas que compartilharão informações e necessidades em usar e alterar o mesmo conteúdo. Imagine um projeto de software com uma equipe pequena de, por exemplo quatro desenvolvedores. Nesse cenário muito provavelmente os desenvolvedores, ao longo do projeto, precisarão alterar um conteúdo (código-fonte) gerado por outro desenvolvedor e, talvez, essa necessidade apareça justamente enquanto houver outro desenvolvedor fazendo alterações no código. O que fazer nesse caso? O programador A que quer alterar o código só poderá fazê-lo quando o programador B que estiver alterando o código terminar suas tarefas? Mas, e se isso atrasar o programador A? Quanto mais gente no projeto e quanto mais longo for, maior a complexidade em lidar com questões como estas.

É aqui que entra o controle de versão, que nada mais é do que uma série de diretrizes para gerenciamento de conteúdo compartilhado. Por meio do controle de versão é possível saber quem fez as alterações, o motivo das mudanças e o que foi alterado em qualquer tipo de conteúdo.

A **Figura 2** ilustra as alterações em um arquivo de código-fonte feitas por um programador em diferentes momentos. Esse é o histórico do arquivo e fornece informações muito importantes sobre a evolução do mesmo, tais como as versões do arquivo ao longo do tempo e quais as mudanças em cada versão.

Figura 2.Exemplo de histórico de alterações em um arquivo de código fonte ao longo do tempo

Conhecer o histórico de um determinado arquivo é muito importante, pois fornece preciosas informações para o entendimento do arquivo em si. Contudo, o maior benefício do controle de versão é a colaboração. A **Figura 3** ilustra as alterações em um arquivo de código-fonte feitas por quatro programadores, cada um representado por uma cor diferente. Com esse histórico é possível saber quem foi o responsável pela criação de cada versão, quais mudanças ocorreram em cada versão, o motivo das alterações e, o recurso mais importante e poderoso no controle de versão, unir todas as contribuições de todos os indivíduos quando trabalharam em versões paralelas do arquivo em um determinado arquivo, contendo todas as alterações por meio do processo chamado `merge`. Graças a ele é possível ter diferentes pessoas alterando o mesmo arquivo ao mesmo tempo e, quando essas pessoas terminarem suas tarefas, une suas alterações em um arquivo que conte com as tarefas de ambos. Dessa forma, o trabalho se desenvolve muito mais rápido, pois não é mais preciso esperar que alguém que esteja trabalhando em um arquivo termine para que outra pessoa possa fazer suas alterações.

Figura 3. Exemplo de histórico de alterações em um arquivo de código-fonte feitas por diferentes programadores ao longo do tempo

Tudo o que foi falado até aqui diz respeito ao controle de versão. Existem muitas ferramentas que o implementam, cada qual a sua maneira, com algumas semelhanças e diferenças.

Atualmente, uma das ferramentas mais conhecidas e que vem tomando espaço a cada dia é o Git, que é usada por cerca de 68% dos desenvolvedores, de acordo com o Java Tools and Technologies Landscape Report 2016 (seção **Links**). Sendo assim, essa é uma ferramenta que vale a pena conhecer e dominar por completo e será o foco do próximo tópico, que apresenta o Git e suas particularidades, bem como do restante do artigo, que mostrará como trabalhar efetivamente com controle de versão utilizando esta ferramenta.

O que é Git?

O Git implementa de forma automatizada as diretrizes de controle de versão, fornecendo histórico de mudanças, facilitando a colaboração no manuseio de arquivos por diferentes pessoas ao mesmo tempo.

Além do Git, existem outras ferramentas de controle de versão, tais como CVS e SVN, os mais utilizados nos anos anteriores ao Git. No entanto, são ferramentas cujo funcionamento e forma de trabalho são muito distintas e, para entender de forma mais fácil como cada uma funciona, o ideal é deixar de lado tudo o que se sabe das outras ferramentas a fim de evitar confusões entre termos e funcionalidades.

O Git é atualmente a ferramenta de controle de versão mais utilizada por desenvolvedores. Grande parte dessa adesão em massa está relacionada ao fato de o Git ser uma das ferramentas mais recentes e, portanto, trazer melhorias significativas no versionamento em relação a ferramentas mais antigas.

Convém um breve comparativo entre o Git e ferramentas como o CVS e SVN para entender a evolução dessas ferramentas, suas vantagens e desvantagens e o motivo da popularidade do Git.

Git versus CVS e SVN

Diferentemente do Git, o CVS e o SVN são sistemas de versionamento centralizados. Isso quer dizer que eles possuem um repositório central em que os usuários podem enviar (`commit`) e receber (`checkout`) artefatos versionados.

Essa abordagem centralizada oferece como principal vantagem o controle sobre os projetos, facilitando a implementação de segurança de acesso e bloqueio de arquivos ([lock](#)). Entretanto, parece haver mais desvantagens do que vantagens nesse modelo. Entre as inconveniências desse paradigma de versionamento podemos citar problemas de escalação, onde muitos usuários e projetos sob o mesmo repositório geralmente tornam essas ferramentas lentas e a impossibilidade de se trabalhar offline, obrigando os usuários a sempre estarem conectados no servidor para executar tarefas como criação de tags, branchs e merge.

Além disso, há diferenças entre o próprio CVS e o SVN, que também são diferenças significativas, sendo a mais relevante o tipo de commit, que no CVS é feito por arquivo enquanto que no SVN é possível fazer o commit agrupado de arquivos, facilitando o rollback e identificação de algum commit que possa quebrar o código.

Por outro lado, o Git implementa a abordagem de controle de versão descentralizada, que possui uma curva de aprendizado um pouco maior do que o paradigma centralizado. Entretanto, uma vez que se apreende os conceitos e passa-se a utilizar a ferramenta, todo o fluxo de trabalho se torna fácil e intuitivo.

A grande diferença entre os sistemas centralizados e os sistemas distribuídos, como o Git, é em relação ao commit. Em sistemas centralizados, o commit simplesmente envia todas as alterações nos artefatos para o repositório central. Nos sistemas descentralizados o commit apenas cria uma imagem das alterações em um repositório local e, só após o comando o [push](#) (que será abordado no decorrer do artigo) é que as alterações nos artefatos são enviadas para o repositório remoto. Essa forma de trabalho descentralizada torna o Git muito mais rápido em relação as ferramentas centralizadas, além de promover o uso de branches locais para experimentos e commits frequentes, práticas comuns e desenvolvidas na disciplina de métodos ágeis.

Atualmente o SVN é a ferramenta de controle de versão centralizada mais popular. Ela é especialmente adotada em projetos individuais e de pequenas equipes, especialmente por ser muito simples de usar (devido ao paradigma de controle centralizado adotado). Mas simplicidade de uso é um atributo relativo, afinal nada impede o uso do Git de forma simples, como o SVN, e ter a disposição muitas características e possibilidades de trabalho fornecidas pelo paradigma de versionamento distribuído. Além disso, quando se conhece bem os principais conceitos e características de uma ferramenta, seu uso torna-se naturalmente mais fácil. Por isso, os próximos tópicos elucidam os principais conceitos exclusivos do Git e mostram como essa ferramenta trata os dados que devem ser versionados.

Snapshots no Git

A maioria dos outros controladores de versão gerenciam diferenças entre arquivos, associando a cada um deles uma lista de mudanças. Mas no Git as coisas são bem distintas, pois ao invés de uma lista de mudanças associada a cada arquivo, ele usa snapshots para manter o histórico, ou seja, a cada commit o Git copia todos os arquivos do projeto e muda a referência dos mesmos marcando-os como sendo a versão atual. No caso de arquivos que não foram alterados, o Git apenas mantém um link para o arquivo sem copiá-lo novamente. Esse processo de funcionamento é ilustrado na [Figura 4](#).

Nesse exemplo é possível observar o estado inicial onde todos os arquivos foram criados, Version 1. Em seguida, após alterações nos arquivos A e B, uma nova versão é criada, a Version 2, com os arquivos A1 e C1 copiados literalmente para esse snapshot e uma referência ao arquivo B, que não sofreu nenhuma alteração. Já na Version 3, quando apenas um arquivo é alterado (arquivo C2), apenas ele é copiado para o novo snapshot e os demais são links para os arquivos anteriores não alterados. As demais versões dessa linha do tempo seguem a mesma lógica.

Figura 4. A linha do tempo de histórico de versão do Git baseada em snapshots

Essa forma de lidar com as versões como se fossem snapshots traz muitas vantagens, especialmente em relação a criação de branches. Como no Git um branch é um ponteiro para algum commit, poderíamos criar facilmente um branch de qualquer uma das versões da **Figura 4**. O branch seria criado pelo Git simplesmente adicionando um ponteiro para algumas versões e, a partir de então, alterações e commits nesse novo branch seguiriam o mesmo processo de versões e snapshots a partir do ponto de criação desse novo branch em diante.

Trabalho local com Git

Outra característica única do Git é sua forma de trabalho local. Diferente de outros controladores de versão, a maior parte das operações do Git pode ser executada localmente, já que o Git mantém um histórico do projeto localmente. Isso faz do Git uma ferramenta extremamente rápida, que não sofre com a latência da rede para a execução da grande maioria de suas operações. As principais vantagens dessa característica são a rapidez nas operações (imagine ter que comparar diferentes versões de um arquivo de um ou dois meses atrás usando recursos locais versus recursos remotos) e a possibilidade de se trabalhar e ter a disposição quase todas as operações mesmo quando não se está conectado na rede.

Estados do Git

Um dos conceitos mais importantes do Git são os estados que podem ser aplicados em cada arquivo. São três estados fundamentais:

- **Committed** – arquivos armazenados na base local.
- **Modified** – arquivos que sofreram mudanças, mas não foram enviados para a base local (commit).
- **Staged** – arquivos modificados marcados para que façam parte do próximo commit.

A **Figura 5** ilustra a transição de estados de arquivos versionados pelo Git em seu fluxo de trabalho. Além dos estados, pode-se observar seções que estão relacionadas ao estado de cada arquivo. O Git os organiza no diretório de trabalho (working directory), área de preparação (staging area, também chamada de index) e repositório (git directory).

O repositório é o local onde são armazenados os metadados e o banco de objetos do projeto. Essa é a parte copiada quando um projeto é clonado.

O diretório de trabalho é um checkout de alguma versão do projeto. Os arquivos desse diretório são obtidos a partir do banco de dados comprimido do Git no repositório e disponibilizados no disco, possibilitando sua utilização e edição.

A área de preparação é, na verdade, um arquivo chamado de index dentro do repositório que especifica o conteúdo do próximo commit.

Em linhas gerais, o fluxo de trabalho com o Git pode ser descrito da seguinte forma:

1. Faz-se o checkout de um projeto do repositório, torna-se o diretório de trabalho.
2. Todas as alterações (correções, desenvolvimento, etc.) são realizadas no diretório de trabalho. As alterações vão para a área de preparação.
3. Quando o commit é feito, todos os arquivos da área de preparação são armazenados no repositório.

O entendimento dos estados, da forma de trabalho local e dos snapshots são fundamentais para se utilizar o Git de forma fácil e correta, pois essas três características são a base que define o versionamento de controle distribuído e o próprio Git.

Os próximos tópicos se concentram no uso prático do Git conforme o fluxo de trabalho apresentado na **Figura 5**, demonstrando os principais comandos para lidar com versionamento. Veja na seção **Links** o link para fazer o download, instalar e configurar o Git. Partiremos da ideia que o mesmo está pronto em sua máquina.

Comandos do Git

Depois de conhecer tantos detalhes a respeito do Git, seu fluxo de trabalho e termos comumente utilizados ao se trabalhar com controle de versão, convém agora dominar o uso do Git e seus comandos para colocar em prática os conceitos vistos até o momento, tais como configurar o Git, criar repositórios, rastrear arquivos, verificar alterações, fazer commit, entre outras atividades de controle de versão. Ao longo dos exemplos, arquivos simples de código-fonte serão criados para serem versionados pelo Git.

Como configurar Git

O comando `git config` permite configurar diversas características do Git. A maioria delas tem um valor default que se encaixa na grande maioria dos casos. Entretanto, ao menos o usuário deve ser configurado para que o Git associe, por exemplo, seu nome e e-mail em seus commits. Para fazer isso, basta executar os comandos `git config --global user.name <name>` e `git config --global user.email <email>`.

Criando um repositório no Git

O primeiro passo para se trabalhar com controle de versão usando o Git é criar um repositório. Isso pode ser feito com o comando `git init` dentro do diretório que se deseja transformar em repositório, conforme exemplo a seguir:

```
1 | [user@localhost repository]$ git init
2 | Initialized empty Git repository in /home/user/repository/.git/
```

Esse comando inicializa um diretório em um repositório vazio. O diretório oculto criado `.git` contém todas as configurações e dados do repositório em questão, tais como links para repositórios remotos, banco de dados com os arquivos versionados, etc. Uma vez criado o repositório, todo o fluxo de trabalho apresentado anteriormente se torna possível.

Rastreando arquivos no Git

O repositório recém-criado ainda não contém nenhum arquivo para ser versionado (rastreado). Com o comando `git add` pode-se adicionar todos os arquivos do repositório no index (área de preparação) para o próximo commit e, quando finalmente o referido comando for executado, os arquivos passarão a ser rastreados.

O index é um snapshot das alterações do diretório de trabalho corrente e dita o conteúdo do próximo commit, por isso, após qualquer alteração é necessário adicioná-la no index, caso contrário a mesma não será gravada no repositório após commit. Esse comando adiciona todas as alterações nos diretórios abaixo do repositório, mas é possível especificar a adição de arquivos únicos informando o nome e o caminho do arquivo ao invés do ponto, como por exemplo, `git add /home/user/repository/MinhaClasse.java`.

Como exemplo, ao criar o arquivo MinhaClasse.java e executar o comando `git status`, que informa o atual estado do diretório de trabalho, obtém-se o resultado da **Listagem 1**.

Listagem 1. Estado atual do arquivo MinhaClasse.java

```
1 [user@localhost repository]$ git status
2
3 On branch master
4
5 Initial commit
6
7 Untracked files:
8   (use "git add <file>..." to include in what will be committed)
9
10      MinhaClasse.java
11
12 nothing added to commit but untracked files present (use "git add" to track)
```

Isso significa que houve alterações no diretório de trabalho, mas essas mudanças não foram adicionadas no index e, portanto, nada será adicionado ao repositório em caso de commit. Para fazer o commit desse arquivo é necessário, antes de tudo, adicioná-lo por meio do comando `git add`. Após adicionar o arquivo MinhaClasse.java o status agora é o mesmo apresentado na

Listagem 2.

Listagem 2. Estado atual do arquivo MinhaClasse.java após o comando add

```
1 [user@localhost repository]$ git add MinhaClasse.java
2 [user@localhost repository]$ git status
3 On branch master
4
5 Initial commit
6
7 Changes to be committed:
8   (use "git rm --cached <file>..." to unstage)
9
10      new file:   MinhaClasse.java
```

Agora há alterações no index prontas para serem adicionadas ao repositório. `MinhaClasse.java` se tornou um arquivo pronto para ser adicionado ao repositório e ter seu versionamento iniciado.

O próximo tópico detalhará o comando de commit, mas antes note a dica do Git para remover arquivos do index: `git rm --cached <file>`. Caso o arquivo seja removido do index, ele não será adicionado ao repositório.

Git Commit

O comando `git commit` armazena o atual conteúdo do index em um novo commit no repositório juntamente com uma mensagem definida pelo usuário descrevendo as alterações realizadas. Em linhas gerais, a partir do momento em que o commit é realizado, o arquivo passa a ter suas alterações rastreadas, sendo possível visualizar as mesmas ao longo do tempo, comparar diferentes versões e até voltar para versões anteriores.

Para fazer o commit do arquivo MinhaClasse.java descrevendo as alterações com uma mensagem indicando se tratar do primeiro commit, é preciso executar o comando `git commit -m "primeiro commit"`. Na **Listagem 3** é apresentado o resultado.

Listagem 3. Resultado do comando commit

```
1 [user@localhost repository]$ git commit -m "primeiro commit"
2 [master (root-commit) 9026f18] primeiro commit
3   1 file changed, 3 insertions(+)
4   create mode 100644 MinhaClasse.java
```

Nesse caso o commit foi executado com sucesso e um arquivo foi alterado (MinhaClasse.java que foi adicionado ao repositório), já que houve três inserções (este arquivo contém três linhas

e, cada linha inserida entra nessa contagem). A partir de agora esse arquivo passa a ser rastreável e está no repositório local do Git, onde é possível consultar todo o seu histórico.

Histórico: Git commit history

Um dos usos mais importantes para controladores de versão é possibilitar o controle das versões, mostrando o histórico de evolução de um arquivo ou de um conjunto de arquivos por commits. O comando `git log <file>` possibilita visualizar o histórico dos commits por arquivo e o comando `git log` mostra todos eles. Após alteração e commit da classe `MinhaClasse.java`, na qual foi adicionada uma variável, o comando `git log MinhaClasse.java` mostra o seguinte histórico, apresentado a **Listagem 4**.

Listagem 4. Histórico após o comando git

```
1 [user@localhost repository]$ git log MinhaClasse.java
2 commit 10a1f9cb0a490a012dcb625c0571a1307aa41624
3 Author: Gabriel Amorim <gabriel@email.com>
4 Date:   Tue May 13 17:42:52 2017 -0300
5
6     //adicação de constante na MinhaClasse.java
7
8 commit 9026f184ed12772e8480ca59056eb3c29223a808
9 Author: Gabriel Amorim <gabriel@email.com>
10 Date:  Tue May 13 17:14:02 2017 -0300
11
12     primeiro commit
```

Note que cada commit é seguido por um código hash, que é gerado considerando o conteúdo de todo o commit. Dessa forma, cada um tem seu próprio hash e, consequentemente, é impossível perder qualquer alteração ou ter arquivos corrompidos sem que o Git seja capaz de detectar. Voltando ao histórico, veja que cada commit é apresentado mostrando o autor do mesmo, a data e um comentário descrevendo o seu motivo. Assim, é muito fácil consultar o histórico e a evolução de um arquivo ou de toda a base de código.

Comandos do Git ao se trabalhar em equipe

Os comandos apresentados nos tópicos anteriores são bastante úteis ao se trabalhar localmente e fazem parte dos comandos core do Git. No entanto, quando se trabalha em uma equipe que mantém um repositório remoto para versionamento, são necessários novos comandos para atuar nesse cenário. Os próximos tópicos apresentam os comandos mais úteis para se trabalhar em equipe no Git.

Git Clone

O cenário mais comum é uma equipe ter um repositório remoto ao qual todos têm acesso para enviar seus commits. Nesse caso, antes de começar a desenvolver, é necessário baixar o repositório remoto e isso é feito com o comando `git clone <caminho_para_o_repositório>`. Este faz uma cópia do trabalho local de um repositório. O exemplo da **Listagem 5** faz o clone de um repositório do GitHub – serviço que mantém diversos repositórios online – com o comando `git clone https://github.com/gabrielamorim/Java_repository.git`.

Listagem 5. Clone de um repositório do GitHub

```
1 [user@localhost repository]$ git clone https://github.com/gabrielamorim/Java_repository.git
2 Cloning into 'Java_repository'...
3 remote: Counting objects: 126, done.
4 remote: Total 126 (delta 0), reused 0 (delta 0), pack-reused 126
5 Receiving objects: 100% (126/126), 28.53 KiB | 0 bytes/s, done.
6 Resolving deltas: 100% (35/35), done.
7 Checking connectivity... done.
```

Após realizar o clone do repositório, pode-se realizar quaisquer alterações nos códigos baixados.

Quando as alterações forem finalizadas, o comando `git status` mostrará as mesmas no diretório de trabalho, que devem ser adicionadas ao index com o comando `git add` e, finalmente deve-se fazer o commit com o comando `git commit -m "mensagem do commit"` para armazenar as alterações no repositório local e poder rastrear localmente o trabalho realizado.

Enviando e recebendo alterações no repositório remoto

As alterações realizadas e que forem adicionadas no repositório por meio do commit estarão apenas no repositório local. Como o trabalho é em equipe, as alterações locais devem ser enviadas para o repositório remoto para que todos os membros da equipe também tenham acesso e possam atualizar seus repositórios locais com as últimas alterações de outros membros da equipe.

Para isso utiliza-se o comando `git push`, que envia todos os commits do repositório local para o repositório remoto. O exemplo da **Listagem 6** mostra o resultado da execução do `git push`. Para enviar os commits para o repositório remoto é necessário fornecer as credencias de acesso.

Listagem 6. Resultado da execução do `git push`

```
1 [gamorim@localhost Java_repository]$ git push
2 Username for 'https://github.com': username
3 Password for 'https://novais.amorim@gmail.com@github.com':
4 Counting objects: 5, done.
5 Delta compression using up to 4 threads.
6 Compressing objects: 100% (5/5), done.
7 Writing objects: 100% (5/5), 491 bytes | 0 bytes/s, done.
8 Total 5 (delta 3), reused 0 (delta 0)
9 remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
10 To https://github.com/gabrielamorim/Java_repository.git
11     bf03c8d..35ffd16  master -> master
```

Da mesma forma que alterações são enviadas para o repositório remoto, em algum momento será necessário fazer o update do repositório local para baixar as últimas alterações disponíveis no repositório remoto. Isso é feito com o comando `git pull`, que simplesmente baixa todo o conteúdo atualizado do repositório remoto para o repositório local. Ao tentar executar o `git pull` no repositório https://github.com/gabrielamorim/Java_repository.git, obtém-se o resultado a seguir:

```
1 [user@localhost Java_repository]$ git pull
2 Already up-to-date.
```

Esse resultado significa que não há alterações no repositório remoto e que o local já está atualizado. Entretanto, caso houvessem alterações, elas seriam baixadas e o Git mostraria os arquivos alterados.

Git Branches e tags

Branches são ramificações, como uma cópia de um diretório de trabalho, criados para se desenvolver funcionalidades de forma isoladas uns das outros. Todo repositório Git se inicia com um branch padrão (master). O principal fluxo de trabalho consiste em criar novos branches para desenvolver uma funcionalidade e, quando finalizar o desenvolvimento, mesclar (merge) o branch criado com o principal.

Um branch pode ser criado com o comando `git checkout -b <nome_do_novo_branch>`. O exemplo da **Listagem 7** mostra a criação de um novo branch.

Listagem 7. Criação de um novo branch

```
1 [user@localhost repository]$ git branch
2 * master
3 [user@localhost repository]$ git checkout -b funcionalidade_xyz
4 Switched to a new branch 'funcionalidade_xyz'
5 [user@localhost repository]$ git branch
```

```
6 | * funcionalidade_xyz  
7 | master
```

Primeiro foi executado o comando `git branch` para identificar o branch atual. Nesse caso, o único existente era o master, marcado com um asterisco para indicar o branch corrente. O comando `git checkout -b funcionalidade_xyz` criou um novo branch chamado `funcionalidade_xyz`. O Git então mudou para o novo branch e, uma nova execução do comando `git branch` mostrou a existência de dois branches, sendo o atual (aquele em que as alterações serão aplicadas) como sendo a `funcionalidade_xyz`.

Ao fazer uma alteração em algum arquivo desse novo branch, como o `MinhaClasse.java`, todas as mudanças serão adicionadas a esse branch.

Para voltar ao branch `master` utilize o comando `git checkout master`. Ao verificar o arquivo `MinhaClasse.java` do branch `master`, pode-se ver que não há nenhuma alteração, pois as mudanças foram feitas no `funcionalidade_xyz`. Agora é preciso fazer o merge dos dois branches, que é feito com o comando `git merge <nome_do_branch>`. Este mescla as alterações realizadas no branch onde ocorreram as alterações com o branch corrente que, no caso, é o principal. Ao avaliar o arquivo `MinhaClasse.java` agora, é possível ver que as alterações do `funcionalidade_xyz` agora estão no branch `master`. O Git sabe exatamente como fazer o merge caso não haja conflitos, no entanto, quando houver divergências que ele não saiba como resolver, o mesmo exibirá as diferenças entre os branches e solicitar o merge manual, como mostra a **Listagem 8**.

Listagem 8. Diferenças entre os branches

```
1 | [user@localhost repository]$ git merge funcionalidade_xyz  
2 | Updating 10a1f9c..a98668a  
3 | Fast-forward  
4 |   MinhaClasse.java | 1 +  
5 |     1 file changed, 1 insertion(+)  
6 | create mode 160000 Java_repository
```

Os branches auxiliam no desenvolvimento paralelo de funcionalidades, não impactando no código do sistema em produção ou a versão mais próxima de produção, que geralmente é o que está no branch `master`. O merge de um branch paralelo com o principal geralmente é devido a inclusão de uma funcionalidade ou correção de bug, então convém criar um ponto de release, ou seja, uma marcação indicando a versão do software a partir daquele commit, chamada de tag, e sua criação é feita com o comando `git tag <nome_da_tag> <identificador_do_commit>`, como no exemplo a seguir:

```
1 | [user@localhost repository]$ git tag 1.0.0 a98668ab880ba9059cf587de851b989f09c
```

O nome da tag é o identificador da mesma para consultas, que geralmente é o número de versão seguindo a convenção da equipe. O identificador do commit é o hash do commit que originou a criação dessa tag, que pode ser obtido por meio do histórico.

O fluxo de trabalho das ferramentas controladoras de versão tendem a ser mal interpretadas, especialmente quando acontece a migração de uma ferramenta para outra, fazendo com que os desenvolvedores continuem a utilizar a nova ferramenta com os mesmos conceitos da antiga. Isso acontece especialmente com o Git, que tem um fluxo de trabalho um pouco diferente das ferramentas mais utilizadas até então, fazendo o Git ser taxado até mesmo de muito complexo.

Os recursos apresentados nesse artigo são muito úteis para facilitar e melhorar o controle de versão e possibilitam o uso básico e corriqueiro da ferramenta de forma correta. Entender o Git corretamente é o primeiro passo de um longo caminho para o aperfeiçoamento do ambiente de desenvolvimento.

Links Utéis

Tecnologias:

Git

 Voltar

 Anotar

 Marcado como concluído



Por **Gabriel**

Em 2018



Menu

[Quem Somos](#)

Hospedagem web por Porta 80 Web Hosting

[Planos de estudo](#)

[Fale conosco](#)

[Plano para Instituição de ensino](#)

[Assinatura para empresas](#)

[Assine agora](#)