Angelo Medeiros Nóbrega

Git + Git-flow para Leigos com GitLab (Git + Git-flow for Dummies with GitLab)

João Pessoa - PB

2017 - V0.1.0

Angelo Medeiros Nóbrega

Git + Git-flow para Leigos com GitLab
(Git + Git-flow for Dummies with GitLab)

Introdução ao controle de versão e boas práticas com o Git e o Git-flow, usando como repositório remoto o Gitlab.

João Pessoa - PB

2017 - V0.1.0

Lista de ilustrações

Lista de abreviaturas e siglas

CV	Controle de versão

Sumário

Capítulo 1

Antes de tudo, o que é controle de versão?

Antes de começar a utilizar o *git* para versionar seus trabalhos (códigos, imagens, layouts...) você deve entender o que é controle de versão. Após entender esse conceito você estará apto a usar todo o poder que o *git* proporciona.

O controle de versão(CV) é um sistema usado para ter controle sobre todas(isso se for usada corretamente) as mudanças feitas em um determinado arquivo. O CV permite você reverter sua aplicação que se encontra em um estado que está apresentando um *bug*, para um estado anterior onde o *bug* não havia se manifestado. Permite você também descobrir quem introduziu um problema, quando foi introduzido e onde foi introduzido. Se estiver usando um repositório remoto, você não correrá o risco de perder seu arquivos e melhor ainda, você também não perderá o controle sobre as mudanças feitas localmente.

O controle de versão ajudará na organização, facilitará na hora de trabalhar em equipe, sem aquela história de dois desenvolvedores alterarem um mesmo arquivo, ao mesmo tempo, por estarem desenvolvendo um mesmo projeto. Segurança, vocês desenvolverão, todos os projetos, sem medo de perder código ou acabar errando alguma atualização, sem ter como voltar. E você verá que existem muitas outras razões para usar um sistema para controle de versão.

Muitas vezes o controle de versão é confundido com backup, lembre-se que no controle de versão você terá acesso ao arquivo atual e todas alterações ligadas ao arquivo, e no backup você terá acesso apenas a última versão do arquivo.

Lembre-se também que todas essas *features* que o controle de versão oferece só existirão se forem usadas corretamente.

1.1 O Git

O Git é a ferramenta que você utilizará para fazer todo o controle de versão. O Git surgiu quando Linus Torvalds, o criador do Linux, começou a enfrentar problemas quando desenvolvia o kernel do linux (projeto open source, ou seja, o Linus trabalhava com apoio de uma comunidade para seu desenvolvimento) com as ferramentas de versionamento da época havendo a necessidade da criação de uma nova ferramenta. A proposta para o Git era aprensentar algumas *features* que o sistema antigo não oferecia:

1. Velocidade;
2. Projeto simples;
3. Forte suporte para desenvolvimento não-linear (milhares de ramos paralelos);
4. Completamente distribuído;
5. Capaz de lidar com projetos grandes como o núcleo o Linux com eficiência (velocidade e tamanho dos dados).

Desde 2005 quando o Git foi criado, ele passou por um longo período de evolução e ainda continua. Hoje ele está em uma versão estável oferecendo todas as *features* citadas acima.

1.2 O GitLab

O GitLab nada mais é que um gerenciador de repositório *git* remoto. O *Gitlab* apresenta algumas vantagens em relação ao *Github*:

1. Numero de Repositórios ilimitados;
2. Espaço ilimitado (futuramente será cobrado por projetos maiores que 5Gb), atualmente o *GitHub* limita em 1GB por projeto;
3. Integração continua integrada (*GitLab* CI);
4. Importação projetos do *GitHub*, *BitBucket* e *Gitorious*;
5. Armazenamento de repositórios em servidores privados.

A integração continua integrada funciona apenas para sistemas operacionais baseados no Linux.

1.3 O Git-flow

O *git-flow* é uma extensão do *git* para auxiliar o controle de versão usando comandos pré-definidos como boas práticas nesse quesito. Na minha opinião o *git-flow* é muito mais do que uma simples extensão, é uma filosofia, é uma nova maneira de pensar sobre o controle de versão.

Você pode aplicar a metodologia do *git-flow* sem a necessidade de ter ele instalado, usando apenas comandos nativos do *git*.

Capítulo 2

A estratégia de ramificação

A estratégia de ramificação assemelha-se muito a estruta de uma árvore, por isso alguns comandos do *git* usam opções como *branch*(que significa ramo ou galho), ferramentas como o *SourceTree* que serve para visualizar toda a ramificação do projeto em forma de grafos (a título de informação, *tree* significa árvore).

A estratégia que está representada na figura 1, já é algo que grandes e pequenas empresas usam a bastante tempo. Explicarei como ela é construída, como iremos trabalhar em cima dessa estratégia usando o *git*, e como usar a poderosa extensão do *git*, o *git-flow*, para facilitar ainda mais nosso trabalho.

2.1 Os branches

Na estratégia que iremos adotar vamos trabalhar com seis tipos de ramos, são eles, dois ramos principais e quatro tipos de ramos de apoio.

Os ramos principais são os ramos *master* e o *develop*. E os ramos de apoio serão os ramos, *feature*, *release*, *hotfix* e *buxfix*. A seguir irei descrever cada um desses ramos.

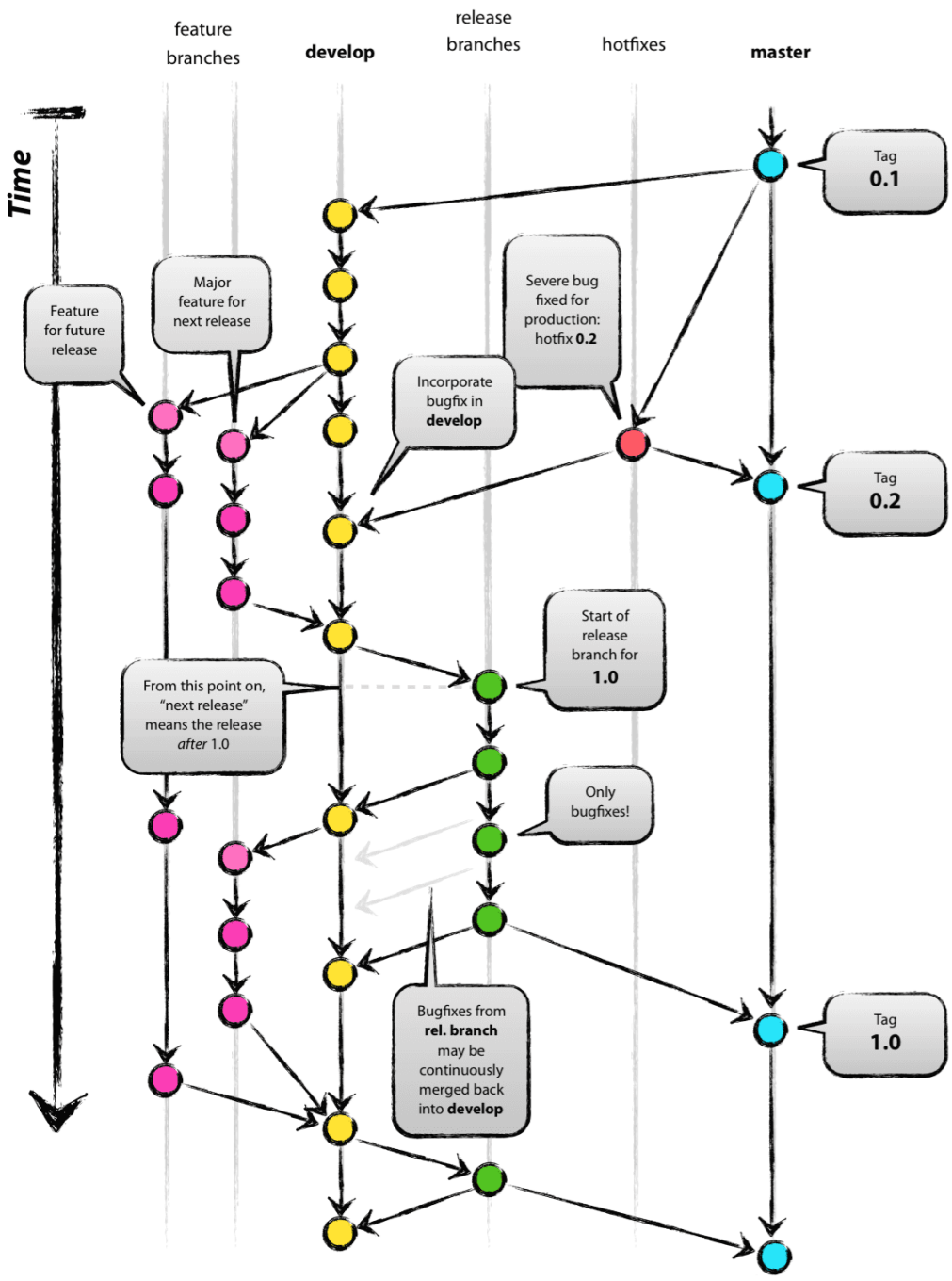
2.1.1 O *branch master*

O ramo *master* é o ramo que irá abrigar os códigos em suas versões mais estáveis, esse é o branch de produção. É o ramo que dará origem a aplicação final. Em algum momento do desenvolvimento todos os códigos produzidos em outros ramos farão um *merge*(ato de mesclar os ramos, colocar os arquivos de um *branch* em outro) com o *branch master*, de forma direta ou indireta.

2.1.2 O *branch develop*

Este ramo será responsável por conter os códigos em nível de desenvolvimento para o próximo *deploy*(significa implementar, mas pode mudar de significado de acordo com o contexto). Lembre-se que os códigos não serão criados nesse *branch*, esse é responsável apenas em abrigar os códigos que estão em desenvolvimento. Após os códigos serem devidamente testados, o *branch develop* fará um merge com o *branch master*, isso se nem

Figura 1 – Estratégia de ramificação

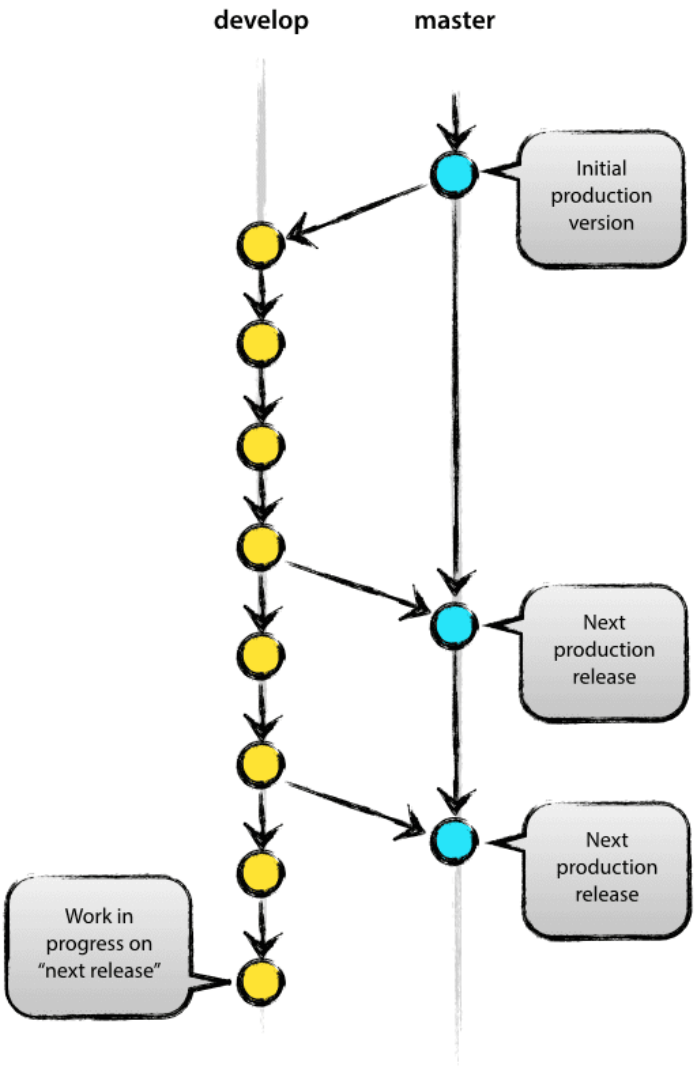


Fonte: (http://nvie.com/posts/a-successful-git-branching-model/)

um *bug* for encontrado no processo de testes, esse processo está representado na figura 2. Se o código aprensentar *bug* será criado outro branch a partir do *branch develop* para a correção dos *bugs* e posteriormente mesclado com o *branch master*.

Os branches *master* e *develop*, possuem vida infinita, ou seja, eles sempre estarão presentes durante o desenvolvimento, e serão criados sempre que o git for inicializado. Os

Figura 2 – O *branch develop*



Fonte: (http://nvie.com/posts/a-successful-git-branching-model/)

branches a seguir terão vida curta, eles serão criados e após suas utlizações eles serão finalizados e excluídos, esse processo firará mais claro na prática.

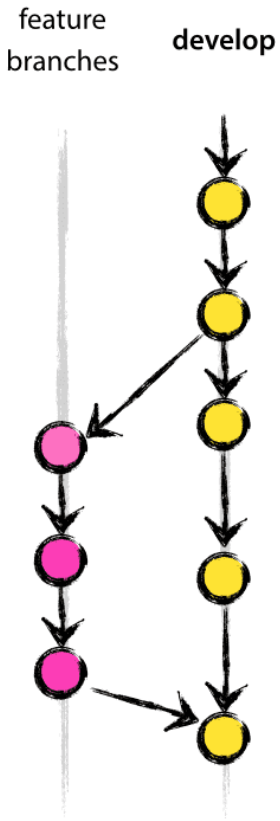
2.1.3 O *branch feature*

O branch feature representado na figura 3, será utlizado sempre que uma nova funcionalidade precisar ser criada. Ele sempre será criado a partir do *branch develop* e finalizado no *branch develop*, independente em qual ramo você esteja, isso se você estiver utlizando o *git-flow*.

Ao contrário do ramo *master* e *develop*, o ramo *feature* pode ser criado múltiplas vezes, uma para cada nova funcionalidade. Esse branch possui como prefixo *feature/**,

onde o asterisco(*) será substituido pelo nome do "sub-ramo" digamos assim, por exemplo, *feature/cadastrodeclientes*, *feature/teladelogin*.

Figura 3 – O *branch feature*



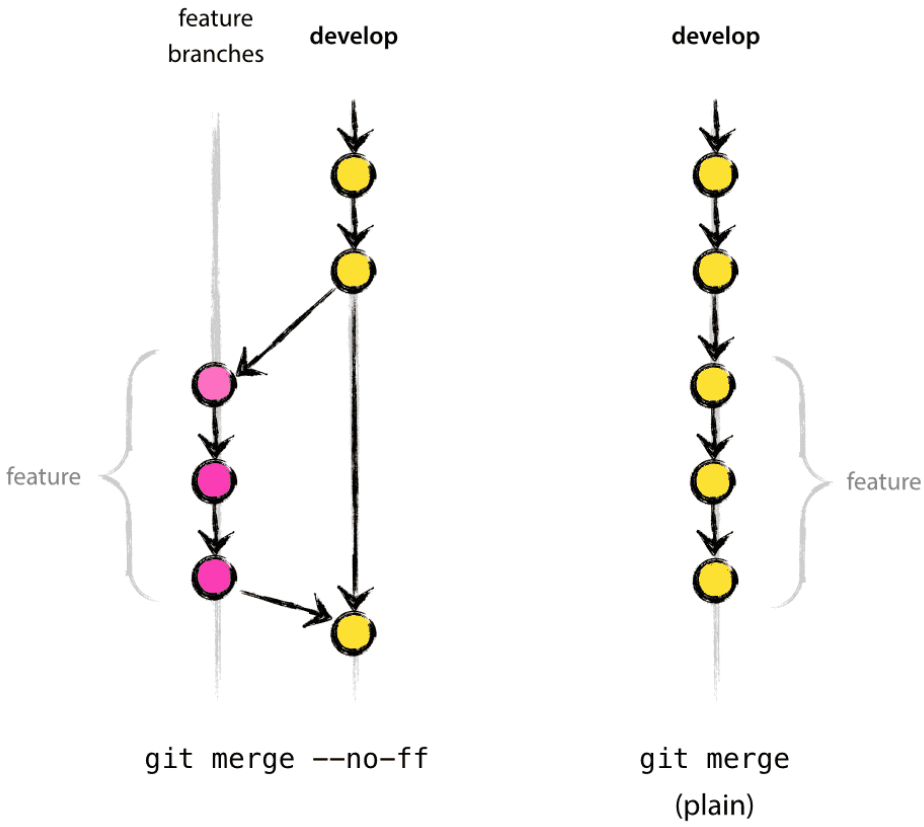
Fonte: (http://nvie.com/posts/a-successful-git-branching-model/)

São duas as principais maneiras de mesclar branches, figura 4. A primeira é fazendo a mesclagem criando um novo *commit* com as alterações dentro do ramo *develop*, sem adicionar os *commits* criados durante o desenvolvimento do ramo feature e, a outra é adicionando os *commits* criados no ramo *feature* dentro do ramo *develop* e, mais um novo *commit* indicando a mesclagem, o git-flow usa como padrão esse último. Novamente, esse proceso ficará mais claro durante a prática.

2.1.4 O *branch release*

Esse branch é o ramo intermediário entre o *develop* e o *master*. O objetivo desse branch é a criação de tags(são os números que indicam a versão da aplicação, 1.0.0, por exemplo). Ele inicia no *branch develop* e termina no *branch master* e no *develop*. Você deve está se perguntando porquê um *branch* que inicia-se no *develop* tem que terminar no *develop*. Isso acontece porquê o *branch develop* tem que ter a mesma *tag* do *branch master*.

Figura 4 – Tipos de merges



Fonte: (http://nvie.com/posts/a-successful-git-branching-model/)

O prefixo usado pelo *branch release* é *release/** , onde o asterisco(*) deve ser substituido pela tag da *realese*, por exemplo, *release/0.1.0*. Mais na frente iremos aprender um conjunto de regras para a criação dessas tags, conhecido como versionamento semântico.

Sempre quando falo que um branch "termina" ou "finaliza", me refiro a mesclagem de um branch em outro, usado por padrão pelo git-flow.

2.1.5 Os *branches hotfixes* e *bugfixes*

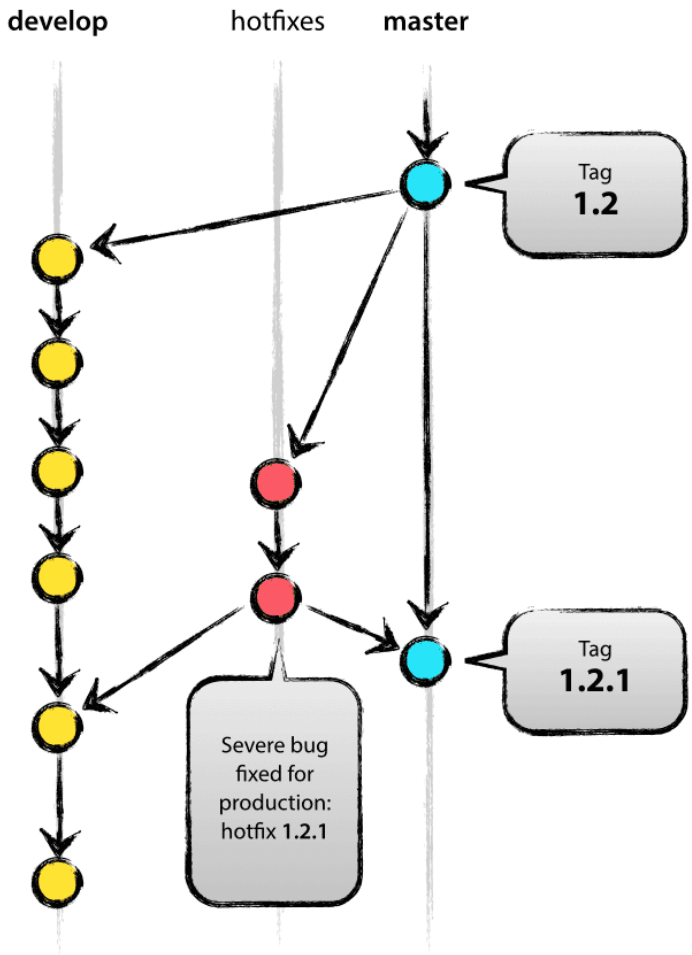
O *branch hotfix* e *bugfix*, tem o objetivo de correção de erros. A diferença entre eles é onde cada um é inicializado.

Se o *bug* for encotrado no ramo de produção(*branch master*), um *branch hotfix*, ver figura 5, deve ser criado para tratar o erro a partir do ramo *master*. Após a correção do erro, uma nova *tag* será criada automaticamente e, o *branch hotfix* será mesclado com o ramo *master* e também ao ramo *develop* para que esse não apresente o erro em futuras versões.

Análogo ao *branch release*, o prefixo do *branch hotfix* é *hotfix/** , onde o asterisco(*)

deve ser substituido pela nova tag, por exemplo, *hotfix/0.1.1*.

Figura 5 – O *branch hotfix*



Fonte: (http://nvie.com/posts/a-successful-git-branching-model/)

Já o *branch bugfix*, ele deve ser criado quando um *bug* é encontrado durante o desenvolvimento. Ele é iniciado e finalizado no *branch develop*. Seu prefixo é semelhante ao do *branch feature*, por exemplo, *bugfix/erro-cadastramento*.