**Git Fluxos de Trabalho**

**ENCONTRANDO SEU FLUXO DE TRABALHO**

Para melhorarmos a forma como utilizamos os controles de versão, podemos escolher alguns fluxos de trabalho para ajudar na organização do processo de desenvolvimento de software. Essas três coisas juntas definem o que você pode chamar de fluxo de trabalho distribuído:

1. **Modelos de distribuição:** definindo como organizar os repositórios remotos. Quantos repositórios você possui e como eles interagem? Tem muitos repositórios compartilhados? Ou apenas um repositório compartilhado que seja visível para todos os desenvolvedores? Todos os desenvolvedores podem enviar suas confirmações para o repositório compartilhado ou alguns desenvolvedores têm acesso somente leitura? Os modelos mais utilizados são:
   * **Modelo ponto a ponto:** sem repositório central, sem controle centralizado, puramente distribuído.
   * **Modelo centralizado**: tem um repositório central e todos enviam para ele, não distribuído.
   * **Modelo pull request/Monorepo:** possui um repositório central, com mantenedores com o direito de enviar por push mas a maioria das pessoas só fazer pull.
   * **Modelo Dictator-Lieutenants/Monotree**, sem repositório central, onde você tem vários subprojetos, cada um no modelo de pull request distribuídos em níveis hierárquicos de integração.
2. **Modelos de ramificação:** quais ramos você tem no seu projeto e para que os usa? Qual ramo você usa para integrar seu trabalho ao trabalho de outros desenvolvedores? Qual ramo você usa para compactar um release? Alguns dos modelos mais comuns são:
   * Utilizar apenas a branch master;
   * Ter uma branch para cada nova funcionalidade, deixando a máster para código pronto para ser entregue;
   * Ter algumas branches por etapa de desenvolvimento, como uma branch de longo prazo para código ainda em construção e uma de curto prazo para correções de bugs urgentes.
3. **Restrições:** todas as regras adicionais que realmente não pertencem ao modelo de distribuição ou ao modelo de ramificação, como por exemplo, quando você recebe um monte de novas confirmações de um controle remoto, você deve mesclar essas confirmações no seu próprio repositório ou deve fazer rebase? Você pode enviar seu código para um controle remoto se as tarefas forem interrompidas ou deve garantir que o código esteja estável e funcionando antes de enviar?

**1 - MODELOS DE DISTRIBUIÇÃO**

A primeira coisa que você deve decidir ao configurar um fluxo de trabalho Git, a decisão que influencia a maioria das outras decisões, é o seu modelo de distribuição.

**Modelo ponto a ponto**

O modelo mais simples onde nenhum repo é mais importante que os outros. Imagine que você é um desenvolvedor em uma equipe de três. Todos esses repositórios contêm o mesmo projeto, foram originalmente clonados um do outro, mas divergem à medida que os desenvolvedores fazem commits em suas próprias cópias do repositório. Cada desenvolvedor pode ver os repositórios de outros desenvolvedores pelo mecanismo de fetch. Se houver novos commits que deseja em outro repositório, basta fazer um pull e inserir essas alterações no seu repositório. Os outros desenvolvedores podem fazer o mesmo, para que as mudanças se espalhem assim.

**Modelo centralizado**

É o mesmo que as pessoas usam com sistemas de versão não distribuídos, como o subversion ou o Team Formation Server. Atualmente, muitas empresas estão substituindo esses sistemas de versão para o Git, mas não necessariamente mudam o modelo de distribuição. Muitos deles ainda usam um modelo centralizado por pares. Como no modelo ponto a ponto nenhum repo é mais importante que os outros, se torna um problema quando você deseja fazer algo tão simples quanto liberar o projeto.

É preciso decidir então de qual repositório liberar, e cada repositório pode conter coisas ligeiramente diferentes. Esse repositório especial/abençoado é acessível a todos no projeto, tanto para o pull (checkout) quanto para push (checkin), para que você possa confirmar dados em seu próprio repositório local, mas também possui um controle remoto que está apontando para o repositório abençoado e pode enviar dados para esse controle remoto. Você pode simplesmente chamar o repo abençoado de servidor, que é um repositório simples, o que, em termos do Git, significa que é apenas o repositório sem uma área de trabalho ou um índice. É usado apenas para compartilhar dados e, talvez, para hospedar a máquina/robô de compilação, o sistema que executa os testes de unidade, pacotes, versões e assim por diante. Agora, neste modelo, você não acessa os repositórios de seus colegas de equipe e vê os dados deles, como no modelo ponto a ponto. Em vez disso, você se preocupa apenas com os dados no repositório do servidor. O problema desse fluxo é a grande possibilidade de existir conflitos constantes.

**Monorepos**

Nos sistemas de controle de versão, um monorepo é uma estratégia de desenvolvimento de software em que o código para muitos projetos é armazenado no mesmo repositório e todo mundo clona de um único lugar. A partir de 2017, algumas formas dessa prática de engenharia de software tinham mais de uma década, mas o conceito geral havia sido nomeado apenas recentemente.

Google, Facebook, Microsoft, Uber, Airbnb e Twitter todos empregam monorepos muito grandes com estratégias variadas para dimensionar sistemas de construção e software de controle de versão com um grande volume de código e alteração diária. As definições variam, um monorepo tem as seguintes características:

* O repositório contém mais de um projeto lógico (por exemplo, um cliente iOS e um aplicativo da web)
* É provável que esses projetos não estejam relacionados, estejam vagamente conectados ou possam ser conectados por outros meios (por exemplo, por meio de ferramentas de gerenciamento de dependências)
* O repositório é grande de várias maneiras:
* Número de confirmações
* Número de branches e / ou tags
* Número de arquivos rastreados
* Tamanho do conteúdo rastreado (conforme medido consultando o diretório .git do repositório)

Digamos que você é uma empresa de tecnologia gigante, mais especificamente uma com poucos produtos muito populares e centenas de desenvolvedores internos. Para complicar ainda mais a logística, esse produto é composto por dezenas de bibliotecas e microserviços.

Você tem dezenas de pull requests aparecendo toda hora em dezenas de repositórios de código. Alguns deles podem acabar quebrando as integrações entre os diversos componentes. Num exemplo hipotético simples, digamos que num dos repositórios você atualize uma biblioteca de parsing de JSON por conta de uma melhoria de performance. Mas digamos que em outro repositório você precise voltar uma versão pra trás dessa biblioteca porque a versão nova causa um erro de compatibilidade com seu código. E agora imagine um terceiro repositório que importa o código desses dois repositórios, cada um dependendo da mesma biblioteca em versões diferentes.

Agora imagine situações como essa acontecendo na mão de centenas de desenvolvedores, toda hora, todos os dias. Imagine a dificuldade que é no final empacotar tudo junto pra gerar um deploy coerente. Milhões de linhas de código, divididos em dezenas ou centenas de repositórios, recebendo modificações o dia inteiro pela mão de centenas de desenvolvedores, sendo vários desses times distribuídos geograficamente e separados até por fusos horários diferentes, e no final precisa empacotar uma versão de produção sem bugs. E isso tem que acontecer com bastante frequência, tipo uma vez por semana ou mesmo uma vez por dia. Ah sim, e o deploy acontece em centenas de máquinas espalhadas em dezenas de data centers pelo mundo.

E para piorar imagine 20 anos de histórico de código acumulado no caso de alguém como Google. Falando em Google, no caso dele estamos falando de mais de um bilhão de linhas de código. Mais de 25 mil engenheiros. E quase 20 anos de histórico num mono repositório gigante com todo o código da empresa que hoje já passou dos 80 terabytes de tamanho. Esse é o tipo de desafio que empresas com Twitter, Google, Facebook, Netflix, e outros gigantes enfrentam todos os dias.

**Modelo Pull-Request**

Este modelo é similar ao anterior com a diferença que a maioria dos desenvolvedores, às vezes são chamados de colaboradores, não pode gravar diretamente no repositório abençoado. Somente uma pessoa, ou algumas pessoas no projeto, podem realizar o push ao repo abençoado. Eles geralmente são chamados de mantenedores. Agora, digamos que um colaborador tem novos commits para contribuir, mas ela não é a mantenedora. Esse colaborador deve avisar o mantenedor de alguma outra maneira para avaliar e, se aprovado, enviar essas alterações ao repositório central. Essa mensagem pode ser um email, por exemplo, ou pode ser gerenciada por algum tipo de serviço, como o GitHub, e isso geralmente é chamado de solicitação pull. Se o manteedor estiver de acordo com essas mudanças, ele poderá colocá-las em seu próprio repositório, resolver qualquer conflito de mesclagem que ele possa ter e enviá-las para o repositório abençoado

E essa é uma das razões pelas quais serviços como o GitHub são tão populares, porque automatizam o envio de solicitações pull, e o modelo pull request se tornou o modelo mais popular para desenvolvimento de código aberto. A vantagem mais significativa do modelo de pull request é que ele ajuda a gerenciar a confiança. Você pode obter contribuições de fontes confiáveis, do mantenedor, e também de fontes menos confiáveis, do colaboradores, e isso é necessário em projetos de código aberto, pois qualquer pessoa na Internet pode ser um colaborador e você provavelmente não confia na Internet inteira para enviar diretamente para o repositório do seu projeto.

Esse conceito de gerenciamento de confiança também é útil para projetos de código fechado. Sempre que você tiver um projeto em que não deseja conceder acesso push ao repositório principal para toda a equipe de desenvolvimento. Nesse caso, você pode nomear um mantenedor interno, talvez chamá-lo de gerente de integração, e os desenvolvedores devem solicitar que essa pessoa faça suas alterações.

A maioria esmagadora dos projetos, adotam um fluxo de trabalho como **GitFlow do GitHub**, com um repositório principal, diversos forks dos repositórios, que recebem colaborações via pull requests de outros forks que retornam para o repositório principal e tem um único lugar na interface gráfica do repositório pra registrar e discutir Issues. Esse ainda é o caso de uso mais produtivo invés de adotar fluxos de trabalho de empresas gigantes como os monorepos do Google e Facebook ou o monotree da kernel do Linux.

**Modelo Dictator-Lieutenants ou Monotree**

Este é como o modelo de pull request, no entanto, o projeto é dividido em subprojetos, e cada subprojeto tem seu próprio repositório abençoado e seu próprio mantenedor, ou mantenedores, e existem outros colaboradores em geral. Neste modelo, qualquer pessoa pode extrair dados de praticamente qualquer lugar, mas apenas os mantenedores podem enviar dados para os repositórios abençoados. Contribuintes regulares geralmente enviam PRs para subprojetos, e os mantenedores do subprojeto enviam PRs para o mantenedor do projeto principal, e todo mundo puxa dados para seus próprios repositórios, para que os dados se espalhem para cima em resposta às pull request, e para baixo à medida que as pessoas o fazem o pull dos níveis superiores. Esse modelo é usado em projetos muito grandes que são grandes demais para uma única equipe de mantenedores como o kernel do Linux.

**Monotree**

No Linux, os mantenedores do subprojeto são chamados de tenentes, e o mantenedor global é chamado de ditador benevolente; portanto, você pode chamar esse modelo de modelo de ditador e tenentes. A kernel do Linux é um dos projetos open source mais antigos em atividade hoje, e com maior quantidade de código, e com a maior quantidade de voluntários colaborando ao redor do mundo. Dezenas de empresas, de dezenas de setores, dependem do Linux. A kernel em si é tecnicamente um monorepo. Mais corretamente ela é um monotree. No caso da kernel, existe o repositório do Linus, só que o único desenvolvedor que trabalha nesse repositório é o próprio Linus.

O fluxo de trabalho da kernel do Linux envolve dezenas de repositórios distribuídos que são forks do repositório do Linux, com discussões acontecendo simultaneamente em dezenas de mailing lists, e esses e-mails servindo como canal de distribuição de dezenas de patches de código que são automatizados de diversas maneiras diferentes por cada empresa ou mantenedor. Uma das primeiras coisas que é diferente no fluxo é que no GitHub só tem como fazer pull request de volta para o repositório de onde se fez fork. Na kernel você quer distribuir os patches para múltiplos repositórios de uma só vez. Além de pode discutir o assunto em paralelo entre diversos repositórios ao mesmo tempo.

Cada distro como Ubuntu, Fedora ou outros tem seus próprios repositórios principais, que não necessariamente clonam direto do Linus. Cada fabricante de hardware que desenvolve drivers tem repositórios diferentes. Mais importante, diversos produtos estão trabalhando em versões diferentes da kernel. Por exemplo, na própria Canonical que faz o Ubuntu, você tem o Ubuntu LTS que é o Long Term Service e o Ubuntu com versões mais novas dos componentes. A idéia do LTS é que ele atualize o menos possível para não ficar quebrando compatibilidade o tempo todo.

Porém, digamos que seja descoberto um problema grave de segurança num driver na kernel nova. E esse bug afeta várias versões da kernel para trás. O bug é descoberto no Fedora primeiro. Então a correção é feita no repositório deles. O patch da correção começa a ser distribuída em mailing lists. Cada empresa ou comunidade com um repositório tem seu mailing list. A forma de distribuir para tanta gente é literalmente usar o recurso de CC ou carbon copy de e-mail, e assim distribuir o patch para diversos lugares diferentes.

É por essas e várias outras razões que, hoje, não tem como suportar o fluxo de trabalho da kernel no GitHub ou GitLab ou qualquer um que siga esse mesmo modelo e tenha essa escala. O fluxo deles segue uma árvore, onde o principal é a raíz e ele vai expandindo numa única direção para cada galho. No caso do Linux é uma rede, um mesh, onde não existe direção definida, cada nó da rede pode se comunicar com qualquer outro nó da rede, sem um que coordena ou bloqueia essa comunicação. O Git foi originalmente feito com essa estrutura totalmente distribuída em mente. Hoje em dia nós simplificamos de uma forma mais centralizada, como num GitHub, para ficar mais usável para a maioria dos mortais.

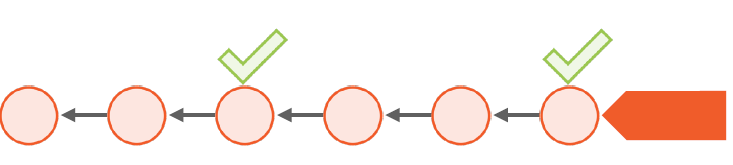
Em resumo, sim, se o GitHub ou GitLab resolverem criar uma estrutura de projeto de múltiplas árvores do mesmo repositório separados para mantenedores diferentes com uma forma mais simples de unificar a cooperação entre repositórios, tanto do ponto de vista de Issues e Pull Requests distribuídos, talvez, bem talvez, um projeto como Linux poderia viver nessas plataformas. Mas seria um tanto de funcionalidade extra para atender um nicho muito pequeno de projetos. Pouquíssimos projetos tem os requerimentos da kernel do Linux.

Pros colaboradores do kernel, tirando alguns subgrupos que podem trabalhar mais isoladamente; digamos, equipes de drivers ou algo assim, a grande maioria ainda vai ser mais produtiva com simples mailing lists e distribuindo patches mesmo, sem ficar na dependência de nenhuma plataforma controlando a comunicação.

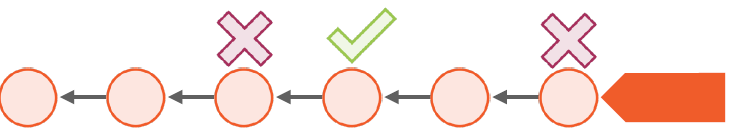
**2 - MODELOS DE RAMIFICAÇÃO**

**Ramificações estáveis ​​e instáveis**

Uma ramificação é estável quando a ponta da ramificação sempre contém uma versão funcional do projeto, ou seja, os testes são verdes e não há bugs conhecidos. Mesmo se alguém adicionar novas confirmações à ramificação, a ponta da ramificação deve conter um sistema em funcionamento.



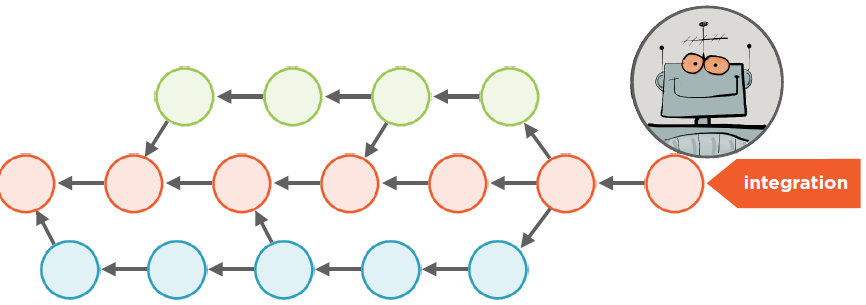
Em um ramo instável, você não necessariamente tem isso. Quando alguém envia mais uma confirmação ao ramo, a ponta do ramo pode estar funcionando ou pode estar quebrada. Não há garantia.



**Padrões comuns de ramificação,**

**Ramo de Integração**

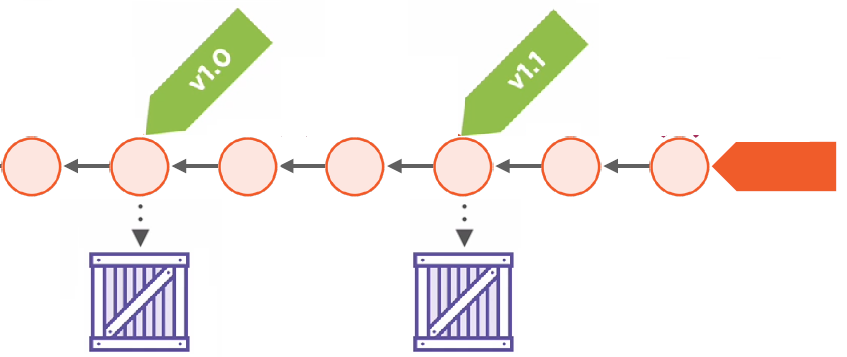
Quase todo projeto tem um ramo principal que você usa para juntar tudo. Esse ramo pode ser chamado de ramo de integração, ramo mestre, ramo principal, linha principal, ou ramo de desenvolvimento. Geralmente, esse é o ramo que as pessoas consideram o ramo mais importante do sistema. As pessoas podem trabalhar em outros ramos, mas os outros ramos tendem a se ramificar nesse ramo principal. Eles tendem a permanecer razoavelmente alinhados com o ramo principal e acabam voltando ao ramo principal onde você resolve os conflitos.



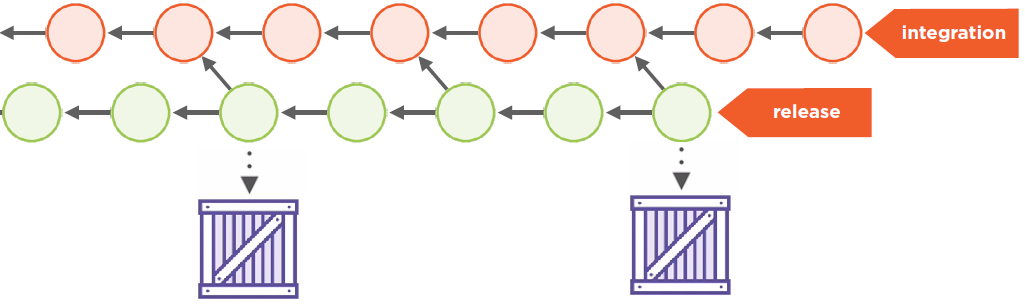
Apesar de ser difícil manter uma ramificação estável quando você está constantemente integrando coisas novas sobre ela, quanto mais estável for o ramo de integração, melhor. O principal trabalho de uma máquina de compilação na Jenkins, ou algum outro tipo de sistema de compilação automatizado, é verificar o que estiver no ramo de integração, provavelmente executar os testes e informar se a compilação atual está funcionando ou quebrada. Afinal, ninguém gosta de trabalhar em uma base de código instável.

**Ramo de Liberação (Release)**

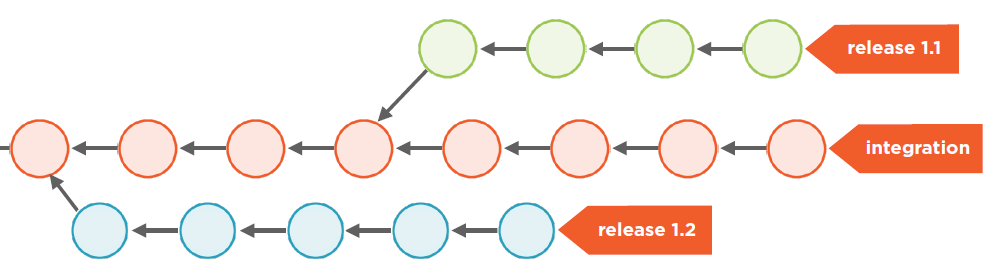
Em algum momento, você precisa implantar o software em um servidor da Web, ou talvez empacotá-lo e distribuí-lo em uma loja de aplicativos ou qualquer que seja o seu método de distribuição. Alguns projetos fazem isso a partir do ramo de integração, colocando tags em commits específicos para marcar os pontos de onde são liberados.



Outros projetos preferem ter uma ramificação separada para liberações para manter o código nesse ramo mais estável do que no ramo de integração. Por exemplo, você pode mesclar a ramificação de integração na ramificação de liberação somente depois de verificar se é estável. Essencialmente, um ramo de liberação separado fornece um buffer para manter as alterações liberáveis ​​separadas das alterações ainda não totalmente liberáveis.



Em alguns projetos você precisa manter vários lançamentos ao mesmo tempo e, nesse caso, você pode ter várias ramificações de liberação. Talvez eles se ramifiquem do ramo de integração no momento em que o lançamento acontece e depois prossigam. Se você precisar adicionar documentação específica ou corrigir um bug ou a versão 1.1, mas não a versão 1.2, bem, você terá um ramo de versão específico para trabalhar.

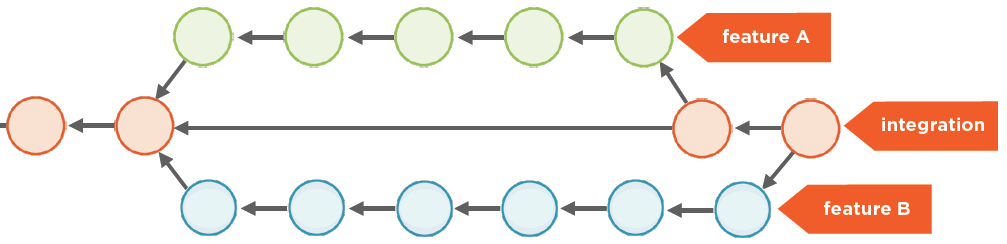


**Ramo de Recurso (Feature/Topic)**

Geralmente são ramificações leves que você cria localmente e que têm um nome que é significativo para você. O recurso/tópico é a razão da criação da ramificação, essencialmente. Eles são onde você pode trabalhar para uma correção ou recurso de bug que deve levar algum tempo para ser concluído. Digamos que você tenha dois desenvolvedores trabalhando em dois recursos diferentes. Em alguns projetos, ambos iriam diretamente para o ramo de integração.



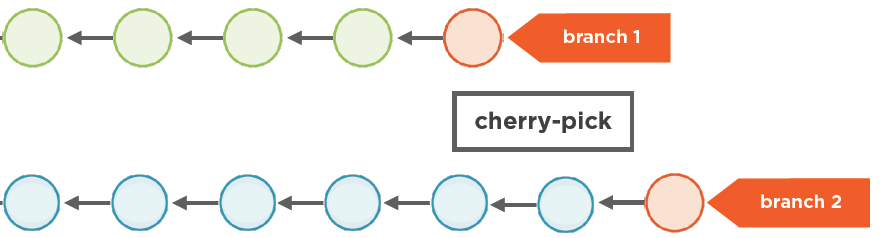
Com essa maneira de trabalhar, você pode ter uma integração muito frequente, mas às vezes pode ser difícil fazer o certo e, além disso, sua história fica difícil de entender porque os commits pertencentes a diferentes recursos podem ser todos interligado. Com essa abordagem, você também precisa viver com recursos semi-desenvolvidos no ramo de integração na maior parte do tempo. Por exemplo, agora, talvez o recurso verde esteja pronto, mas o recurso azul está ainda em andamento, você terá metade do recurso azul no ramo de integração. Uma alternativa possível é criar uma nova ramificação para cada recurso.



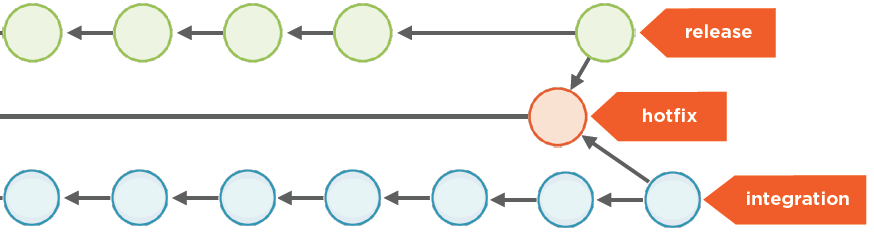
Assim, por exemplo, um ramo para o recurso A e um para o recurso B. E quando as pessoas começam a avançar para esses ramos, elas divergem e progridem em paralelo até eventualmente voltarem para o ramo de integração. Nesse caso, estamos integrando ao mesclar, mas você também pode fazer rebase do processo. E depois disso, você pode excluir esses ramos, ou deixá-los lá para referência futura.

**Ramo de Correção (Hotfix)**

Mas há um problema adicional que pode resultar em mais ramificações para alguns projetos. Imagine que você tenha uma situação como essa com alguns ramos que estão divergindo há algum tempo, mas a ramificação 1 tem uma confirmação vermelha que você também deseja ter na ramificação 2. Você não não quero nenhum dos outros commit das ramificações 1 e 2, apenas a vermelha. O Git tem um comando especial chamado de **cherry-pick**.



No cherry-pick, você pode escolher uma única confirmação, ou algumas confirmações específicas, de uma ramificação e copiá-las em cima de outra ramificação. Porém, o cherry-pick é como um pequeno rebase de um commit específico. Porém neste projeto específico não utilizam rebase e querem usar a mesclagem em todos os lugares. Em casos assim, quando você deseja que o mesmo commit seja compartilhado por dois ramos separados, basta ter um terceiro ramo e colocar esse commit compartilhado no terceiro ramo. E então você mescla o terceiro ramo no primeiro e no segundo ramo.



Agora você tem um lugar para colocar confirmações compartilhadas e acaba de mesclar, sem rebases. Esta é realmente uma situação comum, especialmente quando você tem uma correção de bug. Por exemplo, você tem um ramo de lançamento e um ramo de integração, e eles estão divergindo. Você acabou de encontrar um bug desagradável em sua versão mais recente e deseja corrigi-lo imediatamente e preparar outra versão. Mas você também deseja que a mesma correção de bug esteja no ramo de integração, para que você também tenha o bug corrigido na próxima versão. Bem, basta colocar a correção de bug em outro ramo chamado hotfix, e mesclar o hotfix no ramo de lançamento e de integração. Agora, essa correção está no histórico dos dois ramos. Você manteve as ramificações separadas e ainda compartilhou dados entre elas.

**Ramificação Remota**

Esse tipo de ramificação segue o desenvolvimento do trabalho de outra pessoa e é armazenado em seu próprio repositório. Você atualiza periodicamente esse ramo (usando git fetch) para rastrear o que está acontecendo em outro lugar. Quando estiver pronto para acompanhar as alterações de todos os outros, você usaria o git pull para buscar e mesclar.

**3 - RESTRIÇÕES**

Todo projeto tem suas próprias regras e restrições. Em alguns projetos, algumas pessoas preferem sempre mesclar e outras preferem fazer o rebase na ramificação de integração. Uma decisão para todo o projeto é decidir se a equipe toda vai usar mesclagem ou rebase.

Outra questão importante para alguns projetos é quem pode fazer o que com quais branches? Talvez seja esperado que alguns desenvolvedores façam commits em apenas com alguns ramos e não com outros, ou talvez seu projeto esteja usando tags no ramo de lançamento para marcar um novo lançamento, para que apenas o responsável pelas liberações possa marcar o ramo de lançamento.

De vez em quando, os testes de compilação podem ter erros onde é melhor interromper de mandar push para o servidor. Em casos como esse é até possível utilizar um recurso Git chamado ganchos (hooks), que é basicamente uma maneira de executar um script sempre que um evento específico acontece.

Aqui está outra decisão que algumas equipes optam por regular. Que tipo de histórico ou rebase você deve fazer nas ramificações de recursos antes de mesclá-las na ramificação de integração? Alguns projetos gostam de compactar o recurso inteiro em um único commit, por exemplo, enquanto outros preferem manter um commit pequeno e granular no histórico da linha principal.

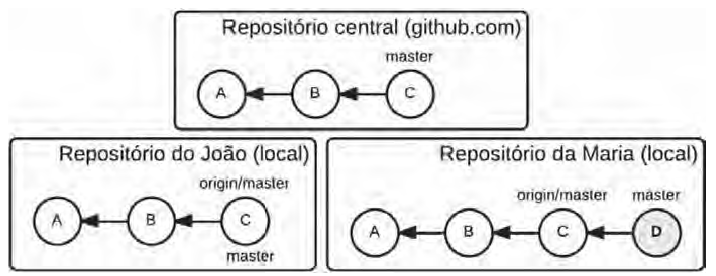
**MODELOS DE FLUXO DE TRABALHO**

Combinando os 3 fatores analisados anteriormente (Modelo de distribuição, de ramificação e restrições) chegamos aos fluxos de trabalho possíveis ao se trabalhar mais comuns que serão descritos em detalhes a seguir:

1. **Utilizando só a branch master com um repositório central**
2. **Utilizando branches por funcionalidade com um repositório central**
3. **Utilizando branches por etapa de desenvolvimento com um repositório central**
4. **Colaborando com projetos open source com Fork e Pull Request**
5. **Organizando projetos open source gigantescos com Ditador e Tenentes**

**Utilizando só a branch master com um repositório central**

Um dos fluxos de trabalhomais simples é utilizar apenas um repositório central hospedado, por exemplo, no Github, comitando tudo diretamente na branch local master. Para enviar os commits locais, cada desenvolvedor faz um push para o repositório central. Por isso, todos osmembros da equipe devem ter permissões de push.



Considerando que temos um repositório para o nosso projeto configurado no Github, a primeira coisa seria fazer:

**git clone https://github.com/empresa/projeto.git**

Após o clone, temos uma cópia local completa do repositório. Podemos, então, criar novos arquivos ou editar arquivos existentes. Quando estivermos satisfeitos com o código, podemos adicionar as alterações à área de stage e depois comitá-las na master, utilizando comandos como:

**git add .**

**git commit -m "Otimizando consulta a clientes"**

Depois de alguns commits, podemos compartilhar nosso trabalho com a equipe:

**git push origin master**

Depois de termos publicado os novos commits, os outros desenvolvedores podem obtê-los executando:

**git pull --rebase origin master**

Encorajamos o uso deum git pull --rebase ao obter mudanças remotas porque simplifica o histórico do repositório, evitando commits de merge. Se ocorrer algum conflito no momento em que obtemos os novos commits, devemos resolver os problemas editando os arquivos apropriados para então executar:

**git add .**

**git rebase --continue**

Quando estivermos prontos para fazer uma entrega, podemos marcá-la criando uma tag e, depois, enviando essa tag para o repositório central:

**git tag v1.0**

**git push origin --tags**

**Quando utilizar?**

* Para equipes pequenas: Quando trabalhamos em equipes pequenas, com até 5 desenvolvedores, quanto mais simples o fluxo de trabalho, melhor. Como a equipe é pequena, muito provavelmente o projeto será pequeno e teremos controle no ritmo das entregas, não precisando de um fluxo mais poderoso (e complexo).
* Na adoção do Git: Esse fluxo é parecido coma maneira como são comumente utilizados sistemas de controle de versão centralizados, como o Subversion. Por isso, para equipes que estão iniciando comoGit, é o fluxo demais fácil adoção. A equipe pode focar em aprender os comandos básicos doGit, utilizados para lidar com um repositório local, para depois partir para fluxos mais avançados.

**Vantagens**

* A simplicidade desse fluxo permite uma adoção mais tranquila para quem está começando a utilizar o fluxo. Também há menos complicação para equipes reduzidas.
* É mais fácil de adotar integração contínua, uma prática bastante comum em projetos que usam metodologias ágeis. Nessa prática, o código deve ser integrado frequentemente, disparando builds e testes automatizados e detectando erros de integração o mais rápido possível.

Nesse fluxo, como toda alteração é comitada na branch master, todo código compartilhado depois de um push será integrado na máster do repositório central. A cada novo push, o build e os testes automatizados podem ser disparados utilizando esse código integrado. Além disso, possíveis conflitos são detectados a cada pull.

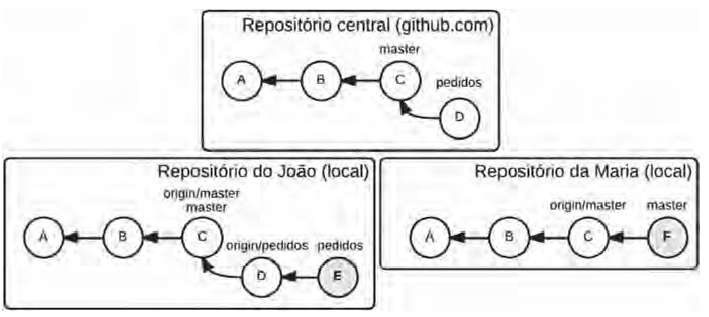
**Desvantagens**

* Ao corrigir defeitos urgentes, pode ser que o código de novas funcionalidades ainda em desenvolvimento já tenha sido compartilhado na branch master do repositório central. Com esse fluxo, fica difícil separar o código da correção do defeito do código das novas funcionalidades, o que pode gerar mais defeitos e insatisfação nos clientes.
* Como tudo é comitado na branch master, as entregas são feitas com todo o código que está no repositório no momento da entrega. Não é possível entregar só parte das funcionalidades que foram compartilhadas com o repositório central.
* Há a necessidade de permissão de push para todos os membros da equipe no repositório central, já que há apenas um repositório remoto. Para projetos open source, é algo inviável. Para projetos e equipesmuito grandes, pode também ser um problema.

**Utilizando branches por funcionalidade com um repositório central**

Podemos utilizar branches para isolar o código de novas funcionalidades ou alterações em funcionalidades existentes. A branch master passa a ser tratada de maneira especial, ficando estável durante todo o desenvolvimento do projeto. Só quando a funcionalidade estiver pronta, é feito um merge da branch da funcionalidade na master.

Se for preciso realizar uma correção urgente, pode ser utilizada a branch master diretamente. Para correções ou alterações mais demoradas, podem ser criadas branches para segregar os commits da correção. Essa maneira de trabalhar também é chamada de feature branching ou topic branching.



Vamos dizer que temos um repositório local clonado a partir de um repositório do Github, comofizemos anteriormente, como comando

**git clone https://github.com/empresa/projeto.git .**

A funcionalidade a ser desenvolvida é a criação de pedidos de nossa loja online. Antes de começar, vamos criar a partir da master uma branch chamada pedidos, já mudando pra essa nova branch:

**git checkout -b pedidos**

Já na branch pedidos, podemos começar a trabalhar na funcionalidade, criando novos arquivos e alterando arquivos existentes. Em seguida, podemos efetuar commits com os comandos que já conhecemos:

**git add .**

**git commit -m "Tela inicial de pedidos"**

Depois de alguns commits, mesmo sem termos terminado a funcionalidade de pedidos, podemos compartilhar o código que fizemos enviando-o para o repositório central. Basta executar:

**git push origin pedidos**

Com o código no repositório central, temos um backup do código e outros membros da nossa equipe podem obtê-lo para colaborar no desenvolvimento da funcionalidade. Tudo isso sem afetar a estabilidade da branch master. O restante da nossa equipe, para poder trabalhar na funcionalidade, precisa obter as últimas alterações do repositório central e, em seguida, criar uma cópia local da branch pedidos. Para isso, deve-se executar:

**git fetch origin**

**git checkout -t origin/pedidos**

Nós e outros membros da equipe podemos fazer commits e, quando apropriado, compartilhar o código por meio do repositório central através do comando:

**git push origin pedidos**

Para obtermos o novo código da funcionalidade, ainda na branch pedidos, devemos fazer:

**git pull --rebase origin pedidos**

No caso de conflitos após o pull com rebase, devemos mesclar os arquivos manualmente e executar:

**git add .**

**git rebase --continue**

A branch pedidos do repositório central terá o código que integra as mudanças de todos os membros da equipe. Testadores e outros interessados podem utilizar o código da branch pedidos para verificar o andamento das atividades e detectar problemas. Revisar a qualidade do código, por exemplo, fica mais fácil porque as mudanças relacionadas à funcionalidade ficam isoladas.

Caso haja alguma alteração feita diretamente na master, como uma correção de um defeito urgente, é importante obtê-la no repositório local. Para isso, continuando na branch pedidos, devemos executar:

**git pull --rebase origin master**

O comando anterior obtém as mudanças da branch remota origin/master e já faz o rebase na branch atual, que é a pedidos. Seria equivalente à sequência de comandos:

**git checkout master**

**git pull origin master**

**git checkout pedidos**

**git rebase master**

A diferença é que, nessa última sequência, a branch local máster ficaria atualizada com as últimas mudanças da origin/master. Quando a equipe estiver satisfeita com a funcionalidade e com o código, algum dos desenvolvedores pode fazer o merge na branch master. Antes, é importante nos certificarmos de que a branch master contém as últimas modificações do repositório central. Podemos utilizar um pull sem rebase, para deixarmos a master intacta. É tranquilo fazer o pull sem rebase nesse caso porque, muito provavelmente, será feito um merge do estilo fastforward, já que não é comum fazermos commits diretamente na master. Na branch pedidos, devemos executar os comandos:

**git checkout master**

**git pull origin master**

**git merge pedidos**

Há casos em que a equipe tem muitos membros e/ou tem problemas na comunicação. Em vez do merge, talvez seja melhor fazer um git pull origin pedidos (ainda na branch master). Também seria realizado um merge, mas com possíveis alterações de última hora. No caso de conflitos no merge da branch da funcionalidade, devemos resolver os conflitos manualmente para depois executar:

**git add .**

**git commit -m "Resolvendo conflitos no merge de pedidos**"

Depois de realizado o merge da branch pedidos com a branch local master, podemos compartilhar o código executando:

**git push origin master**

No momento adequado, podemos liberar uma nova versão do sistema, marcando a entrega com uma tag através do comando git tag v1.0 e compartilhando a tag criada com o comando

**git push origin --tags.**

Se não quisermos entregar alguma funcionalidade na nova versão do sistema, basta não fazermos o merge da branch da funcionalidade na branch master.

**Quando utilizar?**

* Em projetos um pouco maiores, principalmente se já tiverem algumas entregas feitas. É importante que a equipe tenha familiaridade no uso do Git.
* Para projetos que já estão a todo vapor, com melhorias e correções que precisam ser feitas de imediato, isolar código que ainda está sendo desenvolvimento é algo importante. Com o uso de uma branch para cada funcionalidade, podemos organizar o desenvolvimento das novas funcionalidades de maneira a não afetar demandas urgentes.
* Para que essa fluxo seja usado com fluência, é preciso que os desenvolvedores já estejam confortáveis com o uso básico do Git.

**Vantagens**

* Podemos isolar código mais estável na branch master, facilitando a realização de melhorias e correções imediatas.
* Revisões da qualidade do novo código que implementa uma funcionalidade podem ser feitas analisando os commits da branch da funcionalidade.
* Pode ser entregue apenas parte das funcionalidades que estão sendo desenvolvidas, possibilitando mudanças mais tranquilas na estratégia de negócio do nosso cliente.

**Desvantagens**

* Como a equipe precisa dominar o Git razoavelmente bem, o uso desse fluxo no início da adoção do Git fica dificultado.
* Como trabalhamos comumrepositório central, ainda há a necessidade de permissão de push para todos os membros da equipe. Por isso, esse fluxo pode ser problemático para grandes equipes e torna-se inviável para projetos open source.
* O código de uma funcionalidade só é efetivamente integrado com outras mudanças no momento do merge final com a branch master. Com outras funcionalidades em desenvolvimento, possíveis conflitos entre o código das funcionalidades só serão descobertos tardiamente, ao mesclarmos todas as branches.
* Realizar integração contínua, descobrindo problemas no código e nas funcionalidades rapidamente, fica mais difícil. A grande vantagem desse fluxo, que é a de isolar códigodas funcionalidades, torna-se a pior desvantagem sob a ótica de integração contínua. Por isso, especialistas em integração contínua não recomendam o uso de branches por funcionalidade, favorecendo o isolamento das funcionalidades com uma arquitetura mais modular, utilizando abstrações para grandes alterações no código (branch by abstraction) e configurações para desabilitar funcionalidades novas (feature toggles).

**Utilizando branches por etapa de desenvolvimento com um repositório central**

Ao comitarmos um novo código em uma branch separada para uma funcionalidade específica, mantemos a branch local master estável, mas corremos o risco de adiar demais a integração entre os códigos das novas funcionalidades.

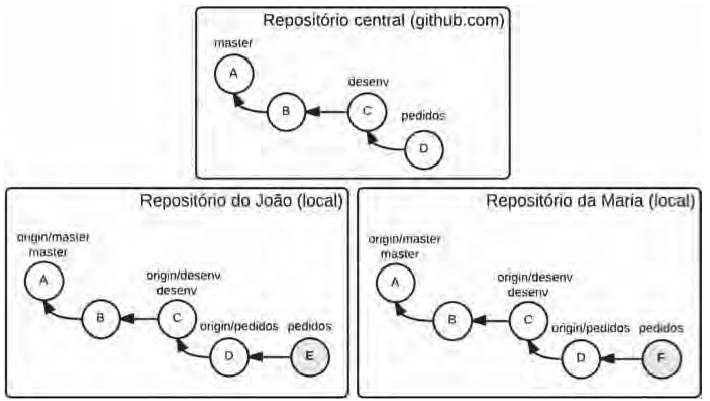
Essa demora pode levar a defeitos que deixariam nossos clientes insatisfeitos. Para evitar a integração tardia das funcionalidades, poderíamos ter uma branch para código ainda em desenvolvimento, chamada desenv. Seria uma branch de longo prazo, que existiria enquanto o projeto estiver sendo desenvolvido.

A branch master teria código já pronto para ser entregue. Já a branch desenv teria código para a próxima entrega. Quando tivermos um ponto estável no novo código, faríamos um merge da desenv na master. Podemos continuar utilizando branches por funcionalidade, mas, agora, criando-as a partir da branch desenv. Periodicamente, poderíamos efetuar um merge precoce das branches das funcionalidades, mesmo antes de o código estar totalmente finalizado. Assim, evitaríamos a integração tardia e deixaríamos de afetar a branch master.

Seria interessante termos branches de release, para comitarmos código referente a uma determinada entrega, como correções de última hora de pequenos bugs descobertos logo antes de liberar uma versão. Teriam nomes como release1.1 ou release2.0 e seriam criadas a partir da desenv.

Também poderiam ser comitados nessas branches de release códigos necessários para preparar uma entrega, como arquivos de versão e release notes. Seriam branches de curto prazo, que poderiamser deletadas quando não fizeremmais sentido. Os commits das correções feitas nessas branches de release precisariam ser aplicados na master e na desenv, através de um merge. Para bugs urgentes, que afetam versões em produção, poderíamos criar uma branch de hotfix. Se estivermos coma versão 1.0 em produção e acontecer um defeito que pode ser corrigido de maneira imediata, poderíamos criar a branch hotfix-1.0.1 a partir da master. A correção seria feita nessa nova branch e, quando finalizada, faríamos um merge na branch master.

Também é importante efetuarmos um merge na branch desenv, para obtermos a correção do defeito. Uma branch de hotfix é de curto prazo e, depois de feito o merge, podemos apagá-la.



Como nos fluxos de trabalho anteriores, cada desenvolvedor deve fazer o clone do repositório central executando:

**git clone https://github.com/empresa/projeto.git**

Logo no início do projeto, ou em algum outro momento apropriado, podemos criar uma branch local chamada desenv, a partir da master, para comitarmos código ainda em desenvolvimento. Também é importante que essa branch exista no repositório central. Para isso, um dos desenvolvedores deve executar:

**git branch desenv**

**git push origin desenv**

Para começarem a trabalhar na branch desenv, os desenvolvedores precisam criar uma branch local que aponta para a branch remota origin/desenv do repositório central. Para isso, devem executar:

**git fetch origin**

**git checkout -t origin/desenv**

Para trabalharmos em uma nova funcionalidade, por exemplo de estoque, podemos criar uma branch para essa funcionalidade a partir da branch desenv, executando:

**git checkout -b estoque desenv**

Podemos efetuar alguns commits e, depois, compartilhar a branch com a nossa equipe, através do comando git push origin estoque. No momento adequado, devemos fazer o merge da funcionalidade estoque na branch desenv, integrando-a com as outras funcionalidades que já foram mescladas com desenv. Porém, antes do merge, é importante obtermos as últimas mudanças em desenv feitas pelos outros membros da nossa equipe através de um pull de origin/desenv. Depois do merge, podemos compartilhar a branch desenv que já terá os últimos commits da estoque. Para fazer isso tudo, considerando que estamos na branch estoque, devemos executar:

**git pull origin desenv**

**git checkout desenv**

**git merge estoque**

**git push origin desenv**

Podemos repetir os comandos anteriores sempre que quisermos integrar a branch da funcionalidade estoque com a branch desenv. Ao terminarmos a funcionalidade, devemos fazer um último merge em desenv. Se quisermos, podemos deletar nossa branch local estoque com o comando

**git branch -d estoque.**

Quando estivermos satisfeitos comas funcionalidades que forammescladas na branch desenv, podemos criar uma branch para a próxima release com um nome como release-1.0 a partir da desenv. Para isso, devemos executar:

**git checkout -b release-1.0 desenv**

Na branch release-1.0 podemos comitar, por exemplo, um arquivo listando as novas funcionalidades (o release notes), modificar arquivos para preparar a entrega. Além disso, podemos comitar correções de pequenos bugs que descobrirmos antes de liberar a nova versão. Enquanto isso, outros desenvolvedores podem começar a trabalhar nas funcionalidades da próxima entrega, comitando na branch desenv.

Para fecharmos a nova versão, devemos fazer o merge da nossa branch release-1.0 na branch master. Para isso, devemos executar os comandos:

**git checkout master**

**git merge release-1.0**

**git push origin master**

Também é importante, ainda na branch master, marcamos a nova versão com uma tag:

**git tag v1.0**

**git push origin --tags**

Se tivermos comitado correções de bugs na branch release-1.0, é importante aplicarmos o código corrigido na branch desenv, através de um merge. Para isso, devemos executar:

**git checkout desenv**

**git merge release-1.0**

**git push origin desenv**

Se quisermos, podemos deletar a branch local release-1.0 através do comando git branch -d release-1.0. Caso aconteça um bug em produção que deve ser corrigido imediatamente, podemos criar uma branch para trabalharmos na correção. Se a versão atual for a 1.0, poderíamos chamá-la de hotfix-1.0.1. Já que a branch master contém o código da última versão liberada para produção, devemos criar a nova branch de correção a partir da master. Para criá-la, devemos executar:

**git checkout -b hotfix-1.0.1 master**

Depois de descobrimos a causa do defeito e termos comitado o código com a correção, podemos fazer o merge da branch hotfix-1.0.1 na branch master. Então, devemos executar:

**git checkout master**

**git merge hotfix-1.0.1**

**git tag 1.0.1**

Não podemos esquecer de aplicar a correção do defeito na branch desenv, que contém o código que está sendo desenvolvido. Podemos fazer isso executando:

**git checkout desenv**

**git merge hotfix-1.0.1**

Feitos os merges da branch hotfix-1.0.1 nas branches máster e desenv podemos apagá-la com o comando

**git branch –d hotfix-1.0.1**

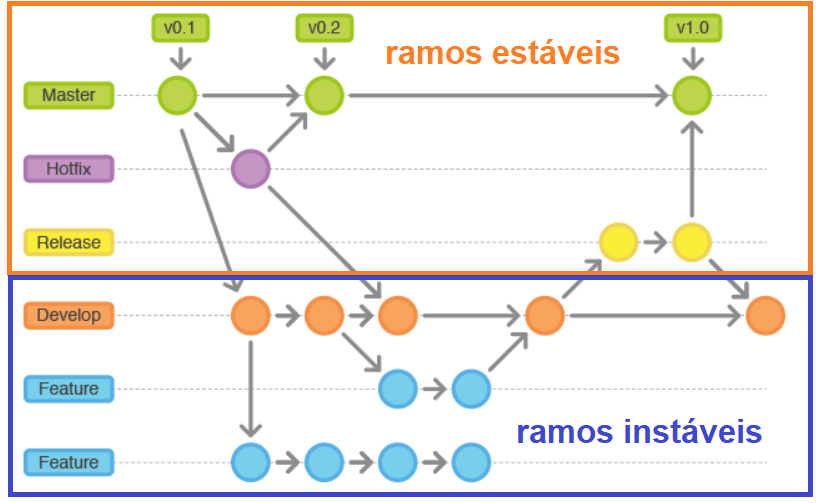
Um fluxo parecido com esse é chamado por alguns de Gitflow, mas não é um fluxo “oficial”, apesar do apelido. Foi descrito por Vincent Driessen em 2010, em seu blog:

**http://nvie.com/posts/a-successful-git-branching-model/**

**GitFlow**

Em um post de 2010 de Vincent Driessen, um desenvolvedor holandês, descreve um fluxo de trabalho muito específico que ele e sua equipe usaram que ficou conhecida como GitFlow. O GitFlow é baseado em um modelo de distribuição centralizado. Todo desenvolvedor pode enviar para um repositório abençoado central. Mas o GitFlow também incentiva os desenvolvedores a trocar dados diretamente quando apropriado, portanto esse é um modelo de distribuição misto, centralizado com elementos ponto a ponto.

O núcleo do GitFlow é seu modelo de ramificação, que é muito detalhado. Ele define um número de ramificações. Para entender as ramificações do GitFlow, você pode particioná-las amplamente em dois grupos, as ramificações instáveis, usadas para o trabalho de desenvolvimento, e as ramificações estáveis, usadas principalmente para lançamentos.



* **Historic Branches:​** Ao invés de trabalhar apenas com o branch de integração master, esse workflow utiliza dois branches principais para guardar histórico do projeto. O branch master guarda o histórico oficial das entregas, já o branch develop serve como integração entre todos os branches de funcionalidades (feature branches).
* **Feature Branches:​** Cada funcionalidade deve ter seu próprio branch, e ele deve ser criado a partir do branch develop. Quando uma funcionalidade for concluída, ela é mesclada (merged) novamente com o seu branch pai. As features nunca devem interagir diretamente com o master.
* **Release Branches**:​ Quando o branch develop estiver com funcionalidades suficientes para uma entrega, nós criamos um branch de entrega (release branch). Com isso, nós damos início ao próximo ciclo de entrega, ou seja, nenhuma nova funcionalidade pode ser incluída a partir desse momento. Quando estivermos prontos para realizar a entrega, o release é mesclada com os branches master e develop.
* **Maintenance Branches**:​ Também conhecidos como hotfix. Eles são usados para corrigir rapidamente algum problema em produção. Este é o único branch que deve ser criado a partir do master. Assim que a correção for finalizada, o branch é fechado e mesclado com o master e develop, mantendo assim as linhas completamente atualizadas.

E depois existem as restrições. Eles definem principalmente quais ramificações podem ramificar de quais outras ramificações e quais ramificações podem ser mescladas em quais outras ramificações. A propósito, eu digo mesclar, porque está sempre mesclando, nunca rebatendo. O GitFlow acredita em manter uma história verdadeira e rastreável. Você não deve mudar essa história com rebases. E há mais regras sobre o que pode ser colocado tags e quando. O GitFlow também fornece as convenções de nomenclatura que você deve usar para algumas de suas ramificações. Todas essas restrições significam que o GitFlow está bem definido e essa precisão tem algumas vantagens definidas. Por exemplo, você pode encontrar as extensões Git na Internet que fornecem comandos específicos para o GitFlow em:

**https://github.com/nvie/gitflow**

Operações como criar uma ramificação de recurso ou mesclar um hotfix estão bem definidas o suficiente para que possam ser automatizadas. No mínimo, a precisão contribui para uma documentação muito boa. Essas são outras razões pelas quais o GitFlow se tornou popular. Tornou-se o fluxo de trabalho do Git para muitas pessoas, e muitas equipes apenas adotaram como estão. Eles acessam a Internet, encontram o GitFlow e o tornam obrigatório para todos. Mas existem muitos projetos para os quais o GitFlow não é a solução ideal.

Por exemplo, projetos que podem evitar todas essas regras complexas de ramificação. Imagine que seu produto é um aplicativo da Web e não um aplicativo empacotado; portanto, você mantém apenas uma versão de produção por vez. Talvez você não precise de várias ramificações de lançamento. Ou talvez algumas das regras do GitFlow possam ser contraproducentes para você. Talvez você esteja em um projeto de ponta que faz implantação contínua. Ele implanta a produção toda vez que alguém integra um recurso estável. Algumas das restrições no GitFlow podem até incentivar comportamentos prejudiciais em projetos que não se encaixam nelas. Por exemplo, o GitFlow exige ramificações de recursos, e você deve mesclar uma ramificação de recursos ao desenvolvimento depois que o recurso for concluído. Porém, alguns grandes projetos herdados podem ter dezenas de recursos em desenvolvimento a qualquer momento, e cada recurso pode levar meses para ser implementado e tende a conter muito código. Portanto, quando esse processo é concluído e você finalmente mescla um recurso no ramo de desenvolvimento, tudo de uma vez, em uma única grande mesclagem, que pode causar grandes conflitos para outras pessoas na equipe. Em tais projetos, eu encorajaria as pessoas a se integrarem com mais frequência. Eu não sou um grande fã de ramos de recursos para dizer a verdade. Geralmente, prefiro fazer uma integração mais frequente quando posso. Mas algumas equipes respeitam essa restrição do GitFlow à letra e acabam tendo ramificações de recursos de folhas muito longas.

Você pode achar que seu fluxo de trabalho muito simples é quase tudo o que você precisa, na verdade, e quando se mostra insuficiente e você descobre que precisa de algo mais complexo, basta adicionar essa complexidade à medida que avança. Você pode adicionar restrições, adicionar novas ramificações, incorporar idéias de outros modelos de distribuição. Faça o seu fluxo de trabalho evoluir para se adequar ao seu projeto. Adicione apenas as regras em resposta a problemas reais que se tornam visíveis e sempre esteja pronto para remover regras que não estão tendo um impacto positivo no projeto.

**Quando utilizar?**

* Em projetos complexos, que já têm várias entregas e com diversas novas funcionalidades em desenvolvimento. Esse fluxo que usa branches por etapa de desenvolvimento deixa o trabalho bastante organizado. A branch de desenvolvimento serve como uma branch de integração para as diferentes branches de funcionalidade. Já a branch master não é afetada pelo dia a dia, ficando bastante estável porque só tem novo código nos momentos de entrega e correções urgentes. Ajustes finos e código para preparar a entrega tem seu lugar, nas branches de release. Bugs urgentes podem ser corrigidos nas branches de hotfix.
* A equipe já deve ter um bom domínio do Git. Devido à complexidade no uso das variadas branches, para trabalhar dessa maneira, o domínio dos conceitos do Git pelos membros da equipe é bastante importante.

**Vantagens**

* A branch master fica bem estável, podendo ser utilizada até para disparar implantações automáticas do software.
* Conseguimos descobrir conflitos ou erros entre códigos das novas funcionalidades mais cedo se mesclarmos as branches das funcionalidades na branch desenv periodicamente.
* Como utilizamos branches por funcionalidade, revisões do código das funcionalidades são fáceis de fazer.
* Podemos entregar apenas parte das funcionalidades, bastando deixar o código isolado na branch da funcionalidade que não entrará na nova versão. Porém, se fizermos integrações periódicas em desenv, isso pode ser um desafio.
* Correções urgentes têm um lugar definido nesse fluxo: as branches de hotfix.
* Trabalho relacionado com a preparação de uma nova versão e ajustes finos antes da liberação podem ser feitos em uma branch de release.

**Desvantagens**

* A equipe precisa de um bom domínio do Git.
* É um fluxo complexo. Por isso, é melhor utilizar esse fluxo em projetos grandes e/ou quando o projeto está a todo vapor. Aí, a organização do trabalho compensa a complexidade.
* Todos os membros da nossa equipe precisam de permissão de push no repositório central, inviabilizando o uso em projetos open source e em equipes grandes.
* Mesmo com a branch desenv, que serve como uma branch de integração, só integraremos efetivamente o código nos momentos em que fizermos o merge das branches das funcionalidades na desenv. Por isso, especialistas em integração contínua ainda criticam o uso desse fluxo, argumentando que essa integração ainda é feita tarde demais. Se esse fluxo for utilizado sem branches por funcionalidade, a situação é melhorada, mas perderíamos as vantagens de isolarmos o código das funcionalidades em desenvolvimento.

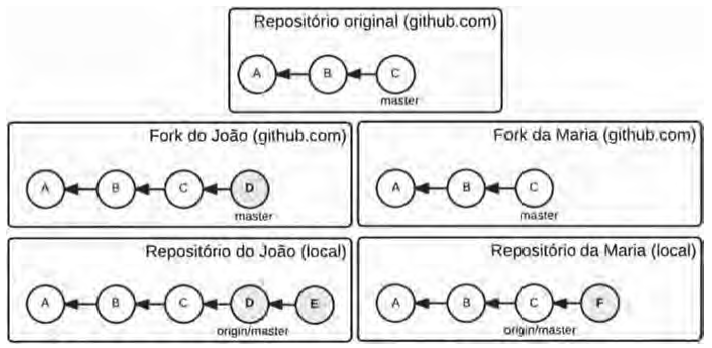
**Colaborando com projetos open source com Fork e Pull Request**

Utilizar um repositório central é algo bastante comum para projetos internos de empresas. Já para projetos open source precisamos de uma maneira mais flexível, que não necessite de permissões de push para as dezenas ou centenas de colabores do nosso projeto.

Serviços como o Github permitem que um colaborador faça forks de um projeto, criando uma cópia pública do repositório. Essa cópia fica publicada na web, servindo como o repositório remoto do colaborador. Assim, os colaboradores podem comitar mudanças em suas cópias do projeto, sem precisar de permissões de push para o projeto original.

O colaborador pode criar um repositório local que aponta para o seu repositório remoto. Depois de comitar algumas modificações, pode ser feito o push para sua cópia do projeto. Se desejar, pode até liberar acesso de push ao seu repositório remoto para outras pessoas colaborarem na sua cópia. Quando o colaborador estiver satisfeito comseu código, é possível enviar um pull request para o projeto principal. O mantenedor do projeto, a pessoa responsável pelo repositório original, pode revisar a cópia pública do colaborador e sugerir melhorias no código.

Quando o mantenedor estiver satisfeito, é possível aceitar o pull request, aplicando as mudanças no repositório original. É interessante que o colaborador faça seus commits em uma branch de funcionalidade, separada da master. Dessa maneira, na hora de aplicar o pull request, o mantenedor do projeto original teria os commits do colaborador em uma branch separada, podendo comitar melhorias. Também é possível utilizar branches por etapa de desenvolvimento para projetos open source maiores.



**Quando utilizar?**

* Em projetos open source de pequeno ou médio porte.
* Para projeto open sourcemuito grandes, commilhares de colaboradores, o número de pull requests seria tão grande que tornaria inviável o uso desse fluxo.
* Também é possível utilizar esse modelo em projetos de empresas, no caso de projetos que tenham colaboradores externos e/ou não confiáveis.

**Vantagens**

* Não é necessário dar permissões de push para todos os colaboradores do projeto.
* É um bom modelo para projetos open source de pequeno ou médio porte.

**Desvantagens**

* A integração das mudanças dos forks é feita de maneira bem tardia. Possíveis conflitos e/ou erros seriam descobertos apenas na hora de aplicarmos o pull request.
* Como só há um repositório original com, provavelmente apenas um mantenedor, o número de pull requests poderia ir acumulando. Para projetos open sourcemuito grandes é necessária uma outra abordagem.

**Organizando projetos open source gigantescos com Ditador e Tenentes**

Para projetos open source como o kernel do Linux, que tem milhões de linhas de código e milhares de colaboradores, utilizar o fluxo de trabalho anterior torna-se inviável.

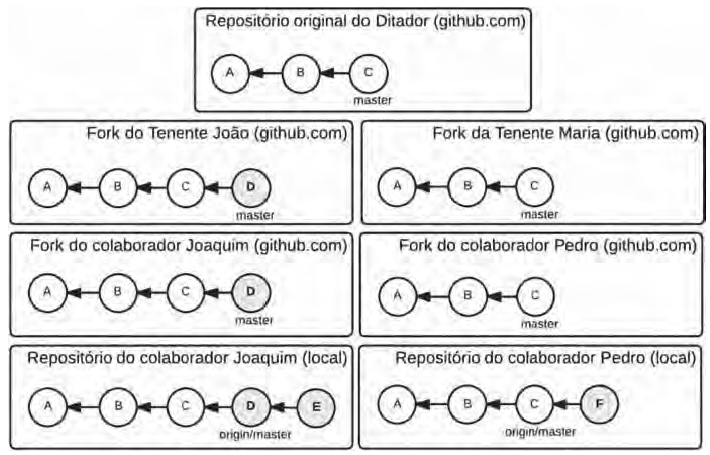
O número de pull requests criados seria enorme. Com apenas um mantenedor, seria impossível dar vazão às colaborações. Para esse tipo de projeto, o mantenedor do projeto original pode ficar como um ditador benevolente, que tem a última palavra sobre o código do projeto mas que aceitas sugestões. No caso do Linux, o ditador é Linus Torvalds.

O ditador elegeria colaboradores que se mostraram competentes no passado para manter repositórios públicos com cópias do projeto. Essas cópias, em geral, teriam um foco em algum módulo específico do projeto. Os eleitos pelo ditador serviriam como tenentes, recebendo pull requests dos milhares de outros colaboradores e revisando o código, filtrando apenas as colaborações realmente boas.

Um novo colaborador teria de escolher um dos repositórios dos tenentes para fazer seu fork do projeto, provavelmente considerando o módulo em que quer colaborar. Depois de feitos seus commits, faria o push para seu repositório. Então, poderia fazer um pull request para seu tenente, que faria uma revisão e daria um feedback.

Quando apropriado, o tenente faria pull requests para o repositório do ditador, sinalizando um pacote interessante de mudanças. Na verdade, o kernel do Linux não utiliza pull requests e nem o Github. No Github, há uma cópia só para leitura do repositório original. Os commits são enviados dos colaboradores para os tenentes e dos tenentes para o ditador Linus por e-mail.

É usado o comando git format-patch para criar um arquivo.patch com um conjunto de commits. Os patches são enviados por e-mail com o comando git send-mail. Então, deve ser utilizado o comando git am para aplicar os commits recebidos por e-mail. É interessante utilizar branches por funcionalidade e por etapa de desenvolvimento ao utilizar esse fluxo de trabalho.



**Quando utilizar?**

Para projetos open source grandes, com milhares de colaboradores.

**Vantagens**

* Assim como no fluxo anterior, não são necessárias permissões de push para o repositório original, do ditador, nem dos tenentes.
* Éum fluxo que funciona bem em projetos open source de grande porte.

**Desvantagens**

* É um fluxo de trabalho extremamente complicado, que requer muita familiaridade com o Git.
* A integração é feita de maneira tardia, só quando for aplicado o pull request (ou os patches recebidos por e-mail).

**GIT E OUTROS VCSS**

O cenário antes da implantação do VCS Git na empresa era um pouco caótico. Havia um servidor central com os projetos em SVN, o deploy não era feito de forma automática, ou seja, a equipe responsável por colocar o código em produção deveria autenticar­se nos servidores e fazer isso com comandos do SVN. Apesar disso, tínhamos duas linhas de produção, chamadas de trunk e branch. A linha trunk representava o produto que estava em produção. Já a linha branch eram as novas funcionalidades que iriam para produção quando estivessem prontas.

Podemos ver que esse modelo dificultava que tivéssemos mais linhas de novas funcionalidades (mais de um branch​), uma vez que manter apenas uma já era extremamente complicado – devido à quantidade enorme de conflitos que ocorriam no código ao mesclar com o trunk. Outra coisa importante era que os erros (bugs) encontrados no sistema no trunk nunca desciam para o branch, distanciando as duas linhas.

Após analisarmos esse cenário, vimos que poderíamos aumentar nossa produtividade, controle e qualidade de código se migrássemos para o GIT, mas para fazer isso precisávamos de um plano de ação. Esse plano foi composto pelos setores de Negócio, Gerência e Arquitetura. Chegamos a uma conclusão e criamos um passo a passo do que deveria ser feito e claro quanto tempo seria necessário para executar cada tarefa.

O projeto foi iniciado em maio de 2014 e foi feito para ser finalizado em 02 meses, por tanto precisávamos organizar as tarefas dentro desse cronograma. A preparação dos ambientes seria a primeira tarefa a ser executada e teve duração de 03 semanas. Após isso, a migração dos projetos do SVN para o GIT deveria ser feita – essa tarefa exigiu 03 semanas de trabalho. Em paralelo a essa tarefa, nós efetuamos a migração dos hooks​ do SVN para o GIT e também foi feita a integração com o bug​ tracker mantis. A última etapa do projeto foi a adequação das ferramentas de deploy​ ​ para o novo VCS​ ​ e foi finalizada em uma semana.

**Preparando o ambiente**

Primeiramente, montamos um servidor específico para o GIT. O servidor deveria ser robusto o suficiente para aguentar mais de 200 commits por dia, já que essa era nossa média. Como interface gráfica do servidor do GIT, escolhemos uma plataforma de código aberto chamada GitLab, que é responsável por facilitar o gerenciamento do VCS. Com tudo isso montado, nós poderíamos botar em prática a segunda etapa do projeto.

**Migrando para o GIT**

Na segunda fase, precisávamos migrar todos os projetos do antigo SVN para o GIT. Isso parece uma tarefa relativamente fácil, mas existiam alguns problemas. O primeiro deles era que em 2 projetos nós deveríamos manter a sincronia entre o SVN e GIT, pois não podíamos parar os desenvolvedores por mais de quatro dias para migrá-los. Por essa razão, precisávamos encontrar uma estratégia de sincronia de VCS e essa foi uma das etapas mais críticas do projeto, pois tínhamos que manter os repositórios funcionando e com seu histórico intacto. Após diversas pesquisas, encontramos uma empresa especializada em GIT e ela disponibilizou um artigo público que ajuda na criação dessa sincronia entre os diferentes VCSs.

É possível fazermos a sincronia em duas direções SVN para o GIT e GIT para SVN. Nós optamos fazer apenas uma via, onde os programadores devem apenas comitar para o repositório SVN e não para o GIT. Quando você estiver confiante que o seu time está pronto para fazer a troca, podemos finalizar a migração e começar a comitar diretamente para o Git.

Durante a migração, cerca de 4 projetos perderam o seu histórico, isso foi causado pela incompatibilidade das mensagens e dos autores dos commits com o formato do GIT e também pelo tamanho dos projetos, que poderiam ser superiores a 30 GB (contando com o histórico).

Com a migração dos projetos concluída, só restaram migrar os dois últimos, mas ainda encontramos um outro problema. Nós temos alguns servidores nos quais podemos colocar as linhas de produção, no geral são quatro. O servidor de produção é onde fica o código da linha master e onde o sistema está disponível para o cliente final. Também temos os servidores de demonstração, que servem para mostrar novas funcionalidades dos produtos antes de irem para o ar. Os servidores de homologação servem para homologarmos as novas funcionalidades (isso normalmente é feito pelas equipes de qualidade e donos do produto). E ainda temos os servidores de desenvolvimento que tem o propósito de disponibilizar um ambiente para os desenvolvedores testarem seu código em sua fase inicial. Nós conseguimos migrar rapidamente todos os servidores menos o de produção, pois ele não podia ser migrado sem subir as novas funcionalidades que já estavam em desenvolvimento. Dessa forma, observamos que a linha de produção master no GIT estava congelada e os desenvolvedores só podiam comitar no trunk do SVN (que era sincronizado com o master automaticamente), todas as outras linhas do Gitflow já estavam implementadas e rodando em todos os servidores e com sprints (expressão utilizada pela metodologia SCRUM) em andamento. Quando tivemos a aprovação da gerência e infraestrutura, nós efetuamos a migração do servidor de produção para o GIT e partir desse momento todas as equipes começaram a trabalhar com o Gitflow completo.

**Integrando o Git**

A terceira etapa do projeto constava em alterar os hooks do GIT para realizarmos integrações e validações. Os hooks são responsáveis por executar comandos automaticamente a partir de um evento que ocorra no VCS. Nós utilizávamos três hooks no SVN e precisávamos converte­los para o GIT. Os hooks eram os commit­msg​ ​, pre­commit​ e post­update​.

O commit­msg​ é o hook responsável por padronizar o formato que os desenvolvedores escrevem as mensagens dos commits. O pre­commit​ verifica se existem erros de sintaxe nos arquivos comitados, e se não existem configurações de debug ativadas. E o hook post­update​ é responsável em atrelar os commits com casos da nossa ferramenta de bug­tracker (rastreador de erros). Essa foi a etapa mais simples e rápida do projeto, pois tivemos que fazer poucas alterações nos hooks que são escritos em Shell Script.

**Deploy**

A última etapa do projeto era talvez a mais desafiadora: deveríamos implantar a automatização do deploy, fazendo integração com o bug­tracker. Para fazermos isso teríamos que alterar a cultura da empresa, mudando a forma como o fluxo é feito. Fizemos diversas pesquisas até chegarmos em algumas ferramentas de Integração Contínua e Entrega Contínua. Para Integração Contínua, nós utilizamos uma ferramenta chamada Jenkins​​, que é responsável por executar os testes unitários, de estresse e de carga na aplicação. A ferramenta também pode encarregar-­se de fazer o deploy automático a partir de pipelines – nós ainda não adotamos essa abordagem, apesar de estar caminhando para isso. Atualmente, fazemos o deploy integrado ao bug­tracker Mantis ​, onde criamos botões para publicação de cada caso em seu ambiente correto. Para isso, funcionar corretamente nós utilizamos a ferramenta Fabric​ que é escrita em Python.

Durante a execução do projeto, a equipe de arquitetura preparou diversos treinamentos para o Git e Gitflow que foram ministrados para as equipes de desenvolvimento, qualidade, gerencia e suporte, mostrando as vantagens da utilização desse novo fluxo de desenvolvimento. Com essa conscientização foi mais simples de quebrar as resistências das equipes.

**GIT E OUTROS SISTEMAS**

Normalmente, você não pode mudar imediatamente todos os projetos com os quais entra em contato com o Git. Às vezes, você fica preso em um projeto usando outro VCS e gostaria que fosse o Git. Muitas organizações ainda usam SVN e pretendem migrar, mas mesmo se você estiver num projeto subversion, ainda assim pode usar Git. A diferença é que você vai trabalhar muito melhor que todo mundo, porque você pode fazer branches locais para experimentar coisas e trabalhar aos poucos fazendo commits pequenos, e só no final pode fazer squash dos seus branchs e criar commits limpos para mandar de volta para o servidor legado.

**GIT COMO CLIENTE**

**Git e Subversion**

Uma grande fração dos projetos de desenvolvimento de código aberto e um bom número de projetos corporativos usam o Subversion para gerenciar seu código-fonte. Já existe há mais de uma década e, na maior parte do tempo, foi a opção de VCS de fato para projetos de código aberto.

**git svn**

Um dos grandes recursos do Git é uma ponte bidirecional para o Subversion chamada git svn. Esta ferramenta permite que você use o Git como um cliente válido para um servidor Subversion, para que você possa usar todos os recursos locais do Git e enviar para um servidor Subversion como se estivesse usando o Subversion localmente. Isso significa que você pode fazer ramificações e mesclagens locais, usar a área de armazenamento temporário, rebasing e cherry-picking, e assim por diante, enquanto seus colaboradores continuam trabalhando da maneira antiga.

É importante observar que, quando você está usando o git svn, está interagindo com o Subversion, que é um sistema que funciona de maneira muito diferente do Git. Embora você possa fazer ramificações e mesclagens locais, geralmente é melhor manter seu histórico o mais linear possível, fazendo rebases no seu trabalho e evitando coisas como interagir simultaneamente com um repositório remoto do Git. Não reescreva seu histórico e tente enviar novamente e não envie para um repositório Git paralelo para colaborar com outros desenvolvedores do Git ao mesmo tempo. O Subversion pode ter apenas uma única história linear, e confundi-la é muito fácil. Se você estiver trabalhando com uma equipe e alguns estiverem usando SVN e outros estiverem usando Git, verifique se todos estão usando o servidor SVN para colaborar. Isso fará com que sua vida seja mais fácil.

Para demonstrar essa funcionalidade, você precisa de um repositório SVN típico ao qual tenha acesso de gravação. Depois precisamos instalar o suporte a subversion do Git. Vamos voltar pro Ubuntu e instalar o pacote apt install git-svn. Todas as distros tem algum pacote similar.

**Se você deseja copiar esses exemplos, precisará fazer uma cópia gravável do meu repositório de teste. Para fazer isso facilmente, você pode usar uma ferramenta chamada svnsync que vem com o Subversion. Para acompanhar, primeiro você precisa criar um novo repositório local do Subversion:**

mkdir/tmp/test-svn

svnadmin create/tmp/test-svn

Em seguida, permita que todos os usuários alterem as revprops - a maneira mais fácil é adicionar um pré-script revprop-change que sempre retorna 0:

cat/tmp/test-svn/hooks/pré-revprop-change

!/bin/sh

exit 0;

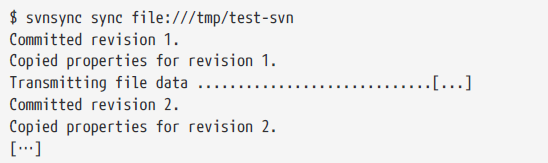
chmod x/tmp/test-svn/hooks/pre-revprop-change

Agora você pode sincronizar este projeto com sua máquina local chamando svnsync init com os repositórios to e from.

svnsync init file:///tmp/test-svn \ http://your-svn-server.example.org/svn/

Isso configura as propriedades para executar a sincronização. Você pode clonar o código executando o arquivo

svnsync sync:///tmp/test-svn



Embora esta operação possa demorar apenas alguns minutos, se você tentar copiar o repositório original para outro repositório remoto em vez de local, o processo levará quase uma hora, mesmo que haja menos de 100 confirmações. O Subversion precisa clonar uma revisão de cada vez e enviá-la novamente para outro repositório - é ridiculamente ineficiente, mas é a única maneira fácil de fazer isso.

Agora que você possui um repositório do Subversion ao qual tem acesso de gravação, pode passar por um fluxo de trabalho típico. Você começará com o comando git svn clone, que importa um repositório inteiro do Subversion para um repositório Git local. Lembre-se de que se você estiver importando de um repositório hospedado real do Subversion, substitua o arquivo: ///tmp/test-svn aqui pelo URL do seu repositório do Subversion:

git svn clone file:///tmp/test-svn -T trunk -b branches -t tags



Isso executa o equivalente a dois comandos - git svn init seguido de git svn fetch - no diretório URL que você fornece. Isso pode demorar um pouco. O projeto de teste tem apenas cerca de 75 confirmações e a base de código não é tão grande, mas o Git precisa verificar cada versão, uma de cada vez, e confirmar individualmente. Para um projeto com centenas ou milhares de confirmações, isso pode literalmente levar horas ou até dias para terminar. A parte -T trunk -b branches -t tags diz ao Git que este repositório do Subversion segue as convenções básicas de ramificação e marcação. Se você nomear seu tronco, ramificações ou tags de maneira diferente, poderá alterar essas opções. Como isso é muito comum, você pode substituir toda esta peça por -s, o que significa layout padrão e implica todas essas opções. O seguinte comando é equivalente:

git svn clone file:///tmp/test-svn -s

Quando você ter clonado o repositório em Git, você pode simplesmente pode dar git remote para adicionar um repositório vazio do Github por exemplo, e dar push pra lá. E a partir desse ponto, pode jogar fora o repositório subversion. Todo o histórico está salvo.

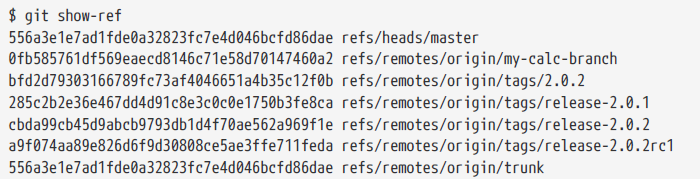
Neste ponto, você deve ter um repositório Git válido que importou suas ramificações e tags:

git branch -a



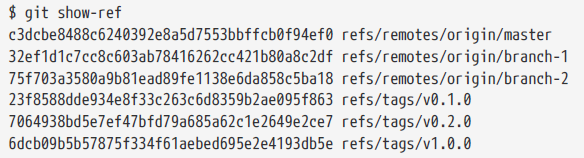
Observe como esta ferramenta gerencia tags do Subversion como referências remotas. Vamos dar uma olhada mais de perto com o comando plumbing do Git show-ref:

git show-ref



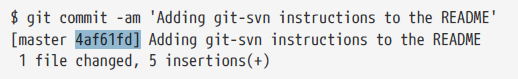
O Git não faz isso quando clona de um servidor Git; aqui está o que um repositório com etiquetas parece que após um clone fresco:

git show-ref



O Git busca as tags diretamente nas refs/tags. Confirmando o retorno ao Subversion Agora que você possui um repositório ativo, pode trabalhar no projeto e enviar seus commit de volta, usando o Git efetivamente como um cliente SVN. Se você editar um dos arquivos e confirmar, você possui um commit existente no Git localmente, que não existe no servidor Subversion:

git commit -am 'Adicionando instruções git-svn ao README'



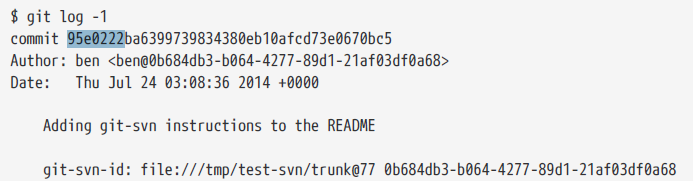
Em seguida, você precisa fazer o envio da sua mudança. Observe como isso muda a maneira como você trabalha com o Subversion - você pode fazer várias confirmações offline e enviá-las todas de uma vez para o servidor do Subversion. Localmente é um repositório Git normal. De tempos em tempos você pode fazer git svn rebase para puxar os últimos commits do subversion. E quando você tiver terminado basta agora fazer um git svn dcommit (observe o d antes da palavra commit) para enviar para um servidor Subversion:

git svn dcommit



Isso pega todos os commits que você fez sobre o código do servidor Subversion. Faz um commit do Subversion para cada um e, em seguida, reescreve o commit do Git local para incluir um identificador exclusivo. Isso é importante porque significa que todas as somas de verificação SHA-1 para seus commit são alteradas. Diferente de um commit de Git que é só local commit de subversion é para mandar para o servidor. Aqui o git svn vai pegar os commits que você trabalhou localmente no Git e criar commits de subversion. Outros desenvolvedores de subversion, quando derem svn update, vão baixar seus commits como se você nunca tivesse usado Git. Por esse motivo, trabalhar com versões remotas baseadas em Git de seus projetos simultaneamente com um servidor Subversion não é uma boa ideia. Se você observar o último commit, poderá ver o novo git-svn-id adicionado:

git log -1



A soma de verificação que começou originalmente com 4af61fd quando você confirmou agora começa com 95e0222. Se você deseja enviar por push para um servidor Git e um servidor Subversion, é necessário enviar (dcommit) para o servidor Subversion primeiro, porque essa ação altera seus dados de confirmação.

**Obtendo novas alterações**

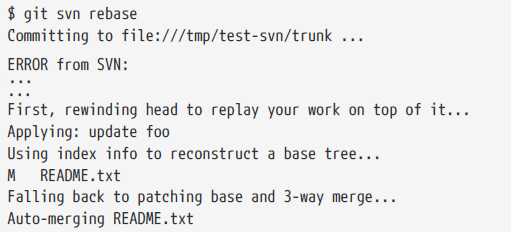
Se você estiver trabalhando com outros desenvolvedores, em algum momento um de vocês pressionará e depois o outro tentará pressionar uma alteração que conflite. Essa alteração será rejeitada até você se fundir no trabalho deles. No git svn, fica assim:

git svn dcommit



Para resolver essa situação, você pode executar o git svn rebase, que remove todas as alterações no servidor que você ainda não possui e refaz o trabalho que você tem sobre o que está no servidor:

git svn rebase ///tmp/test-svn/trunk



Agora, todo o seu trabalho está em cima do que está no servidor Subversion, para que você possa dcommit com êxito:

git svn dcommit



Observe que, ao contrário do Git que requer que você sincronize com um merge os arquivos locais e remotos antes de fazer um push, o git svn faz isso somente se as alterações entrarem em conflito (como o Subversion funciona). Se alguém enviar uma alteração para um arquivo e você enviar uma alteração para outro arquivo, seu dcommit funcionará normalmente.

git svn dcommit Comprometendo-se com o arquivo: /// tmp / test-svn / trunk ... M configure.ac R87 M confirmado autogen.sh r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs / remotes / origin / trunk) : 100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb65d4a7d0389ed6d M autogen.sh Primeiro, rebobine a cabeça para reproduzir novamente o trabalho quando o computador não foi lançado porque o resultado não existe, pois é importante lembrar . Se as alterações forem incompatíveis, mas não entrarem em conflito, você poderá encontrar problemas difíceis de diagnosticar. Isso é diferente de usar um servidor Git - no Git, você pode testar completamente o estado no sistema do cliente antes de publicá-lo, enquanto no SVN, você nunca pode ter certeza de que os estados imediatamente antes da confirmação e após a confirmação são idênticos. Você também deve executar este comando para obter alterações do servidor Subversion, mesmo que não esteja pronto para se comprometer. Você pode executar o git svn fetch para obter os novos dados, mas o git svn rebase faz a busca e atualiza as confirmações locais. $ git svn rebase M autogen.sh r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs / remotes / origin / trunk) Primeiro, rebobine a cabeça para reproduzir seu trabalho sobre ele ... Mestre de avanço rápido para refs / remotes / origin / trunk. A execução do git svn rebase de vez em quando garante que seu código esteja sempre atualizado. Você precisa ter certeza de que seu diretório de trabalho está limpo quando você executa isso. Se houver alterações locais, você deve ocultar seu trabalho ou enviá-lo temporariamente antes de executar o git svn rebase - caso contrário, o comando será interrompido se perceber que o rebase resultará em um conflito de mesclagem. Problemas de ramificação do Git Quando você se sentir confortável com um fluxo de trabalho do Git, provavelmente criará ramificações de tópicos, trabalhará com eles e depois os mesclará. Se você estiver enviando para um servidor Subversion via git svn, poderá rebase seu trabalho em uma única ramificação de cada vez, em vez de mesclar ramificações. O motivo para preferir rebasar é que o Subversion tem um histórico linear e não lida com mesclagens, como o Git, então o git svn segue apenas o primeiro pai ao converter os instantâneos em confirmações do Subversion. Suponha que seu histórico seja o seguinte: você criou um ramo de experiência, fez duas confirmações e depois as fundiu novamente no mestre. Quando você efetua o dcommit, vê uma saída como esta: $ git svn dcommit Confirmando o arquivo: /// tmp / test-svn / trunk ... M CHANGES.txt Confirmado r89 M CHANGES.txt r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs / remotes / origin / tronco) M COPYING.txt M INSTALL.txt Confirmado r90 M INSTALL.txt M COPYING.txt r90 = cb522197870e61467473391799148f6721bcf9a0 (refs / remotes / origin / trunk) Nenhuma alteração entre 71af502c214ba13123992338569f4669877f55fotes / / remotes / origin / trunk A execução de dcommit em uma ramificação com histórico mesclado funciona bem, exceto que quando você olha para o histórico do projeto Git, ele não reescreveu nenhum dos commits que você fez na ramificação da experiência - em vez disso, todas essas alterações aparecem na versão SVN do commit de mesclagem única. Quando alguém clona esse trabalho, tudo o que vêem é o commit de mesclagem com todo o trabalho compactado nele, como se você executasse o git merge --squash; eles não veem os dados de confirmação de onde vieram ou quando foram confirmados. Ramificação do Subversion Ramificar no Subversion não é o mesmo que ramificar no Git; se você pode evitar usá-lo muito, provavelmente é o melhor. No entanto, você pode criar e confirmar com ramificações no Subversion usando o git svn. Criando uma nova ramificação SVN Para criar uma nova ramificação no Subversion, execute git svn branch [branchname]: $ git svn branch opera Copiando arquivo: /// tmp / test-svn / trunk em r90 para o arquivo: /// tmp / test-svn / branches / opera ... Foi possível encontrar o ponto de ramificação possível: file: /// tmp / test-svn / trunk => arquivo: /// tmp / test- svn / branches / opera, 90 Pai branch encontrado: ( refs / remotes / origin / opera) cb522197870e61467473391799148f6721bcf9a0 Seguindo pai com do\_switch Pai seguido com sucesso r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs / remotes / origin / opera / subversion no servidor de cópia), que opera o subdomínio e o servidor. É importante observar que ele não faz check-in dessa branch do 365; se você confirmar neste momento, esse commit entrará no tronco no servidor, não no opera. Mudando de ramificações ativas O Git descobre em qual ramificação seus dcommits procuram a ponta de qualquer uma das ramificações do Subversion em sua história - você deve ter apenas uma e deve ser a última com um git-svn-id em seu atual histórico do ramo. Se você quiser trabalhar em mais de uma ramificação simultaneamente, poderá configurar ramificações locais para confirmar com ramificações específicas do Subversion iniciando-as no commit do Subversion importado para essa ramificação. Se você quer um ramo de ópera no qual possa trabalhar separadamente, pode executar $ git branch opera remotes / origin / opera Agora, se você deseja mesclar seu ramo de ópera no tronco (seu ramo mestre), pode fazê-lo com um procedimento normal. git mesclar. Mas você precisa fornecer uma mensagem de confirmação descritiva (via -m), ou a mesclagem dirá "Mesclar operação de ramificação" em vez de algo útil. Lembre-se de que, embora você esteja usando o git merge para fazer essa operação, e a mesclagem provavelmente será muito mais fácil do que seria no Subversion (porque o Git detectará automaticamente a base de mesclagem apropriada para você), essa não é uma mesclagem normal do Git confirmar. Você precisa enviar esses dados de volta para um servidor Subversion que não pode lidar com uma confirmação que rastreia mais de um pai; portanto, depois de pressionado, ele parecerá um único commit que compactará todo o trabalho de outro ramo em um único commit. Depois de mesclar uma ramificação em outra, você não pode voltar facilmente e continuar trabalhando nessa ramificação, como normalmente no Git. O comando dcommit que você executa apaga qualquer informação que diga em qual ramificação foi mesclada, portanto os cálculos subsequentes da base de mesclagem estarão incorretos - o dcommit fará com que o resultado do seu git merge pareça com o git merge --squash. Infelizmente, não há uma boa maneira de evitar essa situação - o Subversion não pode armazenar essas informações, para que você sempre fique prejudicado por suas limitações enquanto as usa como servidor. Para evitar problemas, você deve excluir o ramo local (neste caso, opera) após mesclá-lo no tronco. Comandos do Subversion O conjunto de ferramentas git svn fornece vários comandos para ajudar a facilitar a transição para o Git, fornecendo algumas funcionalidades semelhantes às que você tinha no Subversion. Aqui estão alguns comandos que mostram o que o Subversion costumava fazer. Histórico de estilos SVN Se você está acostumado ao Subversion e deseja ver seu histórico no estilo de saída SVN, pode executar o git svn log para visualizar seu histórico de consolidação na formatação SVN: 366 $ git svn log --------- -------------------------------------------------- ------------- r87 | escória | 2014-05-02 16:07:37 -0700 (sábado, 02 de maio de 2014) | Mudança automática de 2 linhas ---------------------------------------------- -------------------------- r86 | escória | 2014-05-02 16:00:21 -0700 (sábado, 2 de maio de 2014) | 2 linhas Mesclar ramo 'experimento' ------------------------------------------- ----------------------------- r85 | escória | 2014-05-02 16:00:09 -0700 (sábado, 2 de maio de 2014) | 2 linhas atualizaram o changelog Você deve saber duas coisas importantes sobre o git svn log. Primeiro, ele funciona offline, diferente do comando svn log real, que solicita os dados ao servidor Subversion. Segundo, ele mostra apenas as confirmações que foram confirmadas no servidor Subversion. O Git local confirma que você não se comprometeu a não aparecer; nem confirma que as pessoas fizeram no servidor Subversion enquanto isso. É mais como o último estado conhecido das confirmações no servidor Subversion. Anotação SVN Assim como o comando git svn log simula o comando svn log offline, você pode obter o equivalente à anotação svn executando git svn blame [FILE]. A saída é assim: $ git svn culpa README.txt 2 buffers de protocolo temporais - formato de intercâmbio de dados do Google 2 temporal Copyright 2008 Google Inc. 2 temporal http://code.google.com/apis/protocolbuffers/ 2 temporal 22 temporal C Instalação - Unix 22 temporal ======================= 2 temporal 79 schacon Confirmando no git-svn. 78 schacon 2 temporal Para criar e instalar o tempo de execução do C Protocol Buffer e o compilador Protocolo 2 do buffer temporário (protoc), execute o seguinte: 2 temporal Mais uma vez, ele não mostra confirmações feitas localmente no Git ou que foram enviadas para o Subversion enquanto isso. Informações sobre o servidor SVN Você também pode obter o mesmo tipo de informação que o svn info fornece, executando git svn info: 367 $ git svn info Caminho:. URL: https://schacon-test.googlecode.com/svn/trunk Repositório raiz: https://schacon-test.googlecode.com/svn Repositório UUID: 4c93b258-373f-11de-be05-5f7a86268029 Revisão: 87 tipo de nó : directory Agenda: normal Última alteração: autor: schacon Última alteração: 87 Data da última alteração: 2009-05-02 16:07:37 -0700 (sábado, 02 de maio de 2009) É como culpar e fazer login que ele é executado offline e está atualizado somente na última vez em que você se comunicou com o servidor Subversion. Ignorando o que o Subversion ignora Se você clonar um repositório do Subversion que possua propriedades svn: ignore definidas em qualquer lugar, provavelmente desejará definir os arquivos .gitignore correspondentes para não confirmar acidentalmente os arquivos que não deveria. O git svn possui dois comandos para ajudar com esse problema. O primeiro é o git svn create-ignore, que cria automaticamente os arquivos .gitignore correspondentes para você, para que seu próximo commit possa incluí-los. O segundo comando é git svn show-ignore, que imprime nas linhas padrão que você precisa colocar em um arquivo .gitignore, para que você possa redirecionar a saída para o arquivo de exclusão do seu projeto: $ git svn show-ignore> .git / info / exclude Dessa forma, você não desarruma o projeto com arquivos .gitignore. Esta é uma boa opção se você é o único usuário do Git em uma equipe do Subversion e seus colegas de equipe não desejam arquivos .gitignore no projeto. Resumo do Git-Svn As ferramentas git svn são úteis se você estiver preso em um servidor Subversion ou se estiver em um ambiente de desenvolvimento que exija a execução de um servidor Subversion. Você deve considerar o Git aleijado, no entanto, ou você encontrará problemas na tradução que podem confundir você e seus colaboradores. Para evitar problemas, tente seguir estas diretrizes: • Mantenha um histórico linear do Git que não contenha confirmações de mesclagem feitas pelo git merge. Rebase qualquer trabalho que você faz fora da ramificação da linha principal; não faça a mesclagem. • Não configure e colabore em um servidor Git separado. Possivelmente existe um para acelerar clones para novos desenvolvedores, mas não forneça nada que não tenha uma entrada git-svn-id. Você pode até adicionar um gancho de pré-recebimento que verifica cada mensagem de confirmação em busca de um git-svn-id e rejeita pushes que contenham confirmações sem ele. Se você seguir essas diretrizes, trabalhar com um servidor Subversion poderá ser mais suportável. No entanto, se for possível mudar para um servidor Git real, isso pode ganhar muito mais a sua equipe.

Git e TFS O Git está se tornando popular entre os desenvolvedores do Windows e, se você estiver escrevendo código no Windows, há uma boa chance de usar o Team Foundation Server (TFS) da Microsoft. O TFS é um conjunto de colaboração que inclui rastreamento de defeitos e itens de trabalho, suporte ao processo para Scrum e outros, revisão de código e controle de versão. Há um pouco de confusão pela frente: o TFS é o servidor, que suporta o controle do código fonte usando o Git e seu próprio VCS personalizado, que eles apelidaram de TFVC (Team Foundation Version Control). O suporte ao Git é um recurso um pouco novo para o TFS (fornecido com a versão 2013), portanto, todas as ferramentas anteriores a essa seção se referem à parte de controle de versão como "TFS", mesmo trabalhando principalmente com o TFVC. Se você se encontra em uma equipe que usa o TFVC, mas prefere usar o Git como seu cliente de controle de versão, existe um projeto para você. Qual ferramenta De fato, existem duas: git-tf e git-tfs. Git-tfs (encontrado em https://github.com/git-tfs/git-tfs) é um projeto .NET e (até o momento em que este artigo foi escrito) é executado apenas no Windows. Para trabalhar com os repositórios Git, ele usa as ligações do .NET para libgit2, uma implementação orientada a bibliotecas do Git, que tem alto desempenho e permite muita flexibilidade com o essencial de um repositório Git. O Libgit2 não é uma implementação completa do Git; portanto, para cobrir a diferença, o git-tfs realmente chamará o cliente Git da linha de comando para algumas operações, portanto, não há limites artificiais para o que ele pode fazer com os repositórios do Git. Seu suporte aos recursos TFVC é muito maduro, pois usa os assemblies do Visual Studio para operações com servidores. Isso significa que você precisará acessar esses assemblies, o que significa que você precisa instalar uma versão recente do Visual Studio (qualquer edição desde a versão 2010, incluindo o Express desde a versão 2012) ou o Visual Studio SDK. O Git-tf (cuja casa fica em https://gittf.codeplex.com) é um projeto Java e, como tal, é executado em qualquer computador com um ambiente de tempo de execução Java. Ele faz interface com os repositórios do Git por meio do JGit (uma implementação da JVM do Git), o que significa que praticamente não há limitações em termos de funções do Git. No entanto, seu suporte ao TFVC é limitado em comparação ao git-tfs - ele não suporta ramificações, por exemplo. Portanto, cada ferramenta tem prós e contras, e há muitas situações que favorecem uma sobre a outra. Abordaremos o uso básico de ambos neste livro. NOTA Você precisará acessar um repositório baseado em TFVC para seguir essas instruções. Estes não são tão abundantes na natureza quanto os repositórios Git ou Subversion, então você pode precisar criar um dos seus. O Codeplex (https://www.codeplex.com) ou o Visual Studio Online (http://www.visualstudio.com) são boas opções para isso. Começando: git-tf A primeira coisa que você faz, como em qualquer projeto Git, é clonar. Aqui está o que parece com o git-tf: $ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $ / myproject / Main project\_git O primeiro argumento é a URL de uma coleção TFVC, o segundo é de o formulário $ / project / branch e o terceiro é o caminho para o repositório Git local que deve ser criado (este último é opcional). O Git-tf pode trabalhar apenas com uma ramificação por vez; se você quiser fazer check-ins em uma ramificação TFVC diferente, precisará criar um novo clone a partir dessa ramificação. Isso cria um repositório Git totalmente funcional: $ cd projeto\_git $ git log --all --oneline --decorate 512e75a (HEAD, tag: TFS\_C35190, origin\_tfs / tfs, master) Mensagem de check-in Isso é chamado de clone superficial, o que significa que apenas o o último conjunto de alterações foi baixado. O TFVC não foi projetado para cada cliente ter uma cópia completa do histórico; portanto, o git-tf assume como padrão obter apenas a versão mais recente, que é muito mais rápida. Se você tiver algum tempo, provavelmente vale a pena clonar todo o histórico do projeto, usando a opção --deep: $ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $ / myproject / Main \ project\_git --deep Nome de usuário: domain \ user Senha: Conectando ao TFS ... Clonando $ / myproject em / tmp / project\_git: 100%, concluído. Clonou 4 changesets. Clonou as últimas alterações 35190 como d44b17a $ cd projeto\_git $ git log --all --oneline --decorate d44b17a (HEAD, tag: TFS\_C35190, origin\_tfs / tfs, master) Adeus 126aa7b (tag: TFS\_C35189) 8f77431 (tag: TFS\_C35189) (tag: TFS\_C35177) Criou a pasta do projeto de equipe $ / tfvctest por meio do \ Team Project Creation Wizard Observe as tags com nomes como TFS\_C35189; esse é um recurso que ajuda você a saber quais confirmações do Git estão associadas aos conjuntos de alterações TFVC. É uma boa maneira de representá-lo, pois é possível ver com um simples comando de log qual dos seus commit está associado a um instantâneo que também existe no TFVC. Eles não são necessários (e na verdade você pode desativá-los com git config git-tf.tag false) - o git-tf mantém os mapeamentos reais de commit-changeset no arquivo .git / git-tf. Introdução: git-tfs A clonagem de git-tfs se comporta de maneira um pouco diferente. Observe: PS> git tfs clone --with-branches \ https://username.visualstudio.com/DefaultCollection \ $ / project / Trunk project\_git Repositório Git vazio inicializado em C: /Users/ben/project\_git/.git/ C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9 C16 = c403405f4989d73a2c3c119e79021cb2104ce44a Tfs ramos encontrados: - $ / tfvc-test / featureA o nome da branch local será: featureA C17 = d202b53f67bde32171d5078968c644e562f1c439 C18 = 44cd729d8df868a8be20438fdeeefb961958b674 Aviso a bandeira --with-ramos. O Git-tfs é capaz de mapear ramificações TFVC para ramificações Git, e esse sinalizador indica para configurar uma ramificação Git local para cada ramificação TFVC. Isso é altamente recomendado se você já tiver ramificado ou mesclado no TFS, mas ele não funcionará com um servidor mais antigo que o TFS 2010 - antes desse lançamento, “branches” eram apenas pastas, portanto o git-tfs não pode diferenciá-los de pastas regulares. Deixei dê uma olhada no repositório Git resultante: PS> git log --online --graph --decorate --all \* 44cd729 (tfs / featureA, featureA) Adeus \* d202b53 Ramificado de $ / tfvc-test / Trunk \* c403405 (HEAD, tfs / padrão, mestre) Olá \* b75da1a Novo projeto PS> git log -1 commit c403405f4989d73a2c3c119e79021cb2104ce44a Autor: Ben Straub <ben@straub.cc> Data: sexta-feira, 1 de agosto de 03:41:59 2014 0000 Hello git-tfs- id: [https://username.visualstudio.com/DefaultCollectionBody/$project/Trunk;C16 Existem duas ramificações locais, mestre e featureA, que representam o ponto inicial inicial do clone (Trunk in TFVC) e uma ramificação filha (recursoA em TFVC). Você também pode ver que o tfs “remote” também possui algumas referências: default e featureA, que representam ramificações TFVC. O Git-tfs mapeia o ramo do qual você clonou para tfs / default, e outros obtêm seus próprios nomes. Outra coisa a se notar é o git-tfs-id: lines nas mensagens de confirmação. Em vez de tags, o git-tfs usa esses marcadores para relacionar os conjuntos de alterações TFVC com as confirmações do Git. Isso tem a implicação de que o seu Git confirma terá um hash SHA-1 diferente antes e depois de serem enviados ao TFVC. 397Git-tf [s] Fluxo de trabalho NOTA Independentemente de qual ferramenta você está usando, você deve definir alguns valores de configuração do Git para evitar problemas. $ git config set --local core.ignorecase = true $ git config set --local core.autocrlf = false A próxima coisa óbvia que você vai querer fazer é trabalhar no projeto. O TFVC e o TFS têm vários recursos que podem adicionar complexidade ao seu fluxo de trabalho: 1. As ramificações de recursos que não são representadas no TFVC adicionam um pouco de complexidade. Isso tem a ver com as diferentes maneiras pelas quais TFVC e Git representam ramificações. 2. Lembre-se de que o TFVC permite que os usuários façam check-out dos arquivos do servidor, bloqueando-os para que ninguém mais possa editá-los. Obviamente, isso não impedirá que você os edite em seu repositório local, mas poderá atrapalhar quando chegar a hora de enviar suas alterações para o servidor TFVC. 3. O TFS possui o conceito de check-in "fechado", em que um ciclo de teste de construção do TFS deve ser concluído com êxito antes que o check-in seja permitido. Isso usa a função "arquivar" no TFVC, que não abordamos em detalhes aqui. Você pode fingir isso de maneira manual com o git-tf, e o git-tfs fornece o comando checkintool que reconhece portas. Por uma questão de brevidade, o que abordaremos aqui é o caminho feliz, que evita ou evita a maioria desses problemas. Fluxo de trabalho: git-tf Digamos que você tenha feito algum trabalho, fez algumas confirmações do Git no master e está pronto para compartilhar seu progresso no servidor TFVC. Aqui está nosso repositório Git: $ git log --oneline --graph --decorate --all \* 4178a82 (HEAD, master) código de atualização \* 9df2ae3 update readme \* d44b17a (tag: TFS\_C35190, origin\_tfs / tfs) Adeus \* 126aa7b (tag: TFS\_C35189) \* 8f77431 (tag: TFS\_C35178) PRIMEIRO \* 0745a25 (tag: TFS\_C35177) Criou a pasta do projeto de equipe $ / tfvctest através do \ Team Project Creation Wizard Queremos tirar o instantâneo que está no commit 4178a82 e enviá-lo ao servidor TFVC . Primeiras coisas primeiro: vamos ver se algum de nossos colegas de equipe fez alguma coisa desde a última conexão: 398 $ git tf fetch Nome de usuário: domínio \ usuário Senha: Conectando ao TFS ... Buscando $ / myproject nas últimas alterações: 100%, concluído. Alterou o conjunto de alterações 35320 como confirmação 8ef06a8. Atualizado FETCH\_HEAD. $ git log --oneline --graph --decorate --all \* 8ef06a8 (tag: TFS\_C35320, origin\_tfs / tfs) apenas algum texto | \* 4178a82 (HEAD, mestre) código de atualização | \* 9df2ae3 update readme | / \* d44b17a (tag: TFS\_C35190) Adeus \* 126aa7b (tag: TFS\_C35189) \* 8f77431 (tag: TFS\_C35178) PRIMEIRO \* 0745a25 (tag: TFS\_C35177) Criou o projeto de equipe da pasta $ / tfvctest Parece que alguém está trabalhando também, e agora temos uma história divergente. É aqui que o Git brilha, mas temos duas opções de como proceder: 1. Fazer um commit de mesclagem parece natural como um usuário do Git (afinal, é isso que o git pull faz), e o gittf pode fazer isso por você com um simples git tf pull. Esteja ciente, no entanto, de que o TFVC não pensa assim e, se você pressionar a consolidação de mesclagem, seu histórico começará a parecer diferente nos dois lados, o que pode ser confuso. No entanto, se você planeja enviar todas as suas alterações como um único conjunto de alterações, essa é provavelmente a opção mais fácil. 2. O rebasing torna nosso histórico de consolidação linear, o que significa que temos a opção de converter cada um de nossos commit do Git em um conjunto de alterações TFVC. Como isso deixa em aberto a maioria das opções, recomendamos que você faça dessa maneira; O git-tf até facilita para você com o git tf pull --rebase. A escolha é sua. Para este exemplo, refazeremos: $ git rebase FETCH\_HEAD Primeiro, rebobinaremos a cabeça para reproduzir seu trabalho sobre ele ... Aplicando: update readme Aplicando: update code $ git log --oneline --graph --decorate - -all \* 5a0e25e (HEAD, master) código de atualização \* 6eb3eb5 leia-me de atualização \* 8ef06a8 (tag: TFS\_C35320, origin\_tfs / tfs) apenas alguns textos \* d44b17a (tag: TFS\_C35190) Adeus \* 126aa7b (tag: TFS\_C35189) \* 8f06: ) FIRST \* 0745a25 (tag: TFS\_C35177) Criou a pasta do projeto de equipe $ / tfvctest através do \ Team Project Creation Wizard Agora estamos prontos para fazer um check-in no servidor TFVC. O Git-tf oferece a opção de fazer um único conjunto de alterações que represente todas as alterações desde a última (--shallow, que é o padrão) e criar um novo conjunto de alterações para cada confirmação do Git (--deep). Neste exemplo, apenas criaremos um 399changeset: $ git tf checkin -m 'Atualizando o leia-me e o código' Nome de usuário: domínio \ usuário Senha: Conectando ao TFS ... Fazendo check-in em $ / myproject: 100%, pronto. Confirmado commit 5a0e25e no changeset 35348 $ git log --oneline --graph --decorate --all \* 5a0e25e (HEAD, tag: TFS\_C35348, origin\_tfs / tfs, master) código de atualização \* 6eb3eb5 update readme \* 8ef06a8 (tag: TFS\_C35320) apenas algum texto \* d44b17a (tag: TFS\_C35190) Adeus \* 126aa7b (tag: TFS\_C35189) \* 8f77431 (tag: TFS\_C35178) PRIMEIRO \* 0745a25 (tag: TFS\_C35177) Pasta de projeto de equipe criada $ / tfvctest através do \ Team Project Creation Wizard TFS\_C35348, indicando que o TFVC está armazenando exatamente o mesmo instantâneo que o commit 5a0e25e. É importante observar que nem todo commit do Git precisa ter uma contrapartida exata no TFVC; o commit 6eb3eb5, por exemplo, não existe em nenhum lugar do servidor. Esse é o principal fluxo de trabalho. Há algumas outras considerações que você deve ter em mente: • Não há ramificação. O Git-tf pode criar apenas repositórios Git a partir de uma ramificação TFVC por vez. • Colabore usando TFVC ou Git, mas não os dois. Diferentes clones git-tf do mesmo repositório TFVC podem ter hashes SHA-1 de confirmação diferentes, o que não causará dores de cabeça. • Se o fluxo de trabalho da sua equipe incluir a colaboração no Git e a sincronização periódica com o TFVC, conecte-se apenas ao TFVC com um dos repositórios do Git. Fluxo de trabalho: git-tfs Vamos percorrer o mesmo cenário usando git-tfs. Aqui estão os novos commits que fizemos na ramificação master em nosso repositório Git: PS> git log --online --graph --all --decorate \* c3bd3ae (HEAD, master) código de atualização \* d85e5a2 update readme | \* 44cd729 (tfs / featureA, featureA) Adeus | \* d202b53 Ramificado de $ / tfvc-test / Trunk | / \* c403405 (tfs / padrão) Olá \* b75da1a Novo projeto Agora vamos ver se mais alguém fez algum trabalho enquanto estávamos hackeando: 400PS> git tfs fetch C19 = aea74a0313de0a391940c999e51c5c15c381d5 git log --all --oneline --graph --decorate \* aea74a0 (tfs / default) atualização documentação | \* c3bd3ae (HEAD, mestre) código de atualização | \* d85e5a2 atualização leia-me | / | \* 44cd729 (tfs / featureA, featureA) Adeus | \* d202b53 Ramificado de $ / tfvc-test / Trunk | / \* c403405 Olá \* b75da1a Novo projeto Sim, verifica-se que nosso colega de trabalho adicionou um novo conjunto de alterações TFVC, que aparece como o novo commit de aea74a0 e a ramificação remota tfs / default mudou-se. Como no git-tf, temos duas opções fundamentais para resolver esse histórico divergente: 1. Rebase para preservar um histórico linear. 2. Mesclar para preservar o que realmente aconteceu. Nesse caso, faremos um check-in "profundo", em que cada confirmação do Git se torna um conjunto de alterações TFVC, portanto, queremos fazer uma nova recuperação. PS> git rebase tfs / default Primeiro, rebobine a cabeça para reproduzir seu trabalho sobre ele ... Aplicando: update readme Aplicando: update code PS> git log --all --oneline --graph --decorate \* 10a75ac (HEAD , mestre) código de atualização \* 5cec4ab readme de atualização \* aea74a0 (tfs / padrão) documentação de atualização | \* 44cd729 (tfs / featureA, featureA) Adeus | \* d202b53 Ramificado de $ / tfvc-test / Trunk | / \* c403405 Olá \* b75da1a Novo projeto Agora, estamos prontos para concluir nossa contribuição, verificando nosso código no servidor TFVC. Usaremos o comando rcheckin aqui para criar um conjunto de alterações TFVC para cada confirmação do Git no caminho de HEAD até a primeira ramificação remota do tfs encontrada (o comando checkin criaria apenas um conjunto de alterações, como as confirmações confirmadas pelo Git). 401PS> git tfs rcheckin Trabalhando com o tfs remote: padrão Buscando alterações no TFS para minimizar a possibilidade de conflito tardio ... Iniciando o check-in de 5cec4ab4 'readme de atualização' adicione README.md C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017 rebatido5fec4ab4ab4 ... Rebase feito com sucesso. Iniciando o check-in do b1bf0f99 'código de atualização', edite .git \ tfs \ default \ workspace \ ConsoleApplication1 / ConsoleApplication1 / Program.cs C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b Concluído com o b1bf0f9977b2d4855. Chega de rc heckin. PS> git log --all --oneline --graph --decorate \* ff04e7c (HEAD, tfs / padrão, mestre) código de atualização \* 71a5ddc update readme \* aea74a0 update documentation | \* 44cd729 (tfs / featureA, featureA) Adeus | \* d202b53 Ramificado de $ / tfvc-test / Trunk | / \* c403405 Olá \* b75da1a Novo projeto Observe como após cada check-in bem-sucedido no servidor TFVC, o git-tfs está reestruturando o trabalho restante para o que acabou de fazer. Isso ocorre porque ele adiciona o campo git-tfs-id na parte inferior das mensagens de confirmação, o que altera os hashes do SHA-1. É exatamente como foi projetado e não há com o que se preocupar, mas você deve estar ciente de que isso está acontecendo, especialmente se você estiver compartilhando o Git se compromete com outras pessoas. O TFS possui muitos recursos que se integram ao seu sistema de controle de versão, como itens de trabalho, revisores designados, checkins fechados e assim por diante. Pode ser complicado trabalhar com esses recursos usando apenas uma ferramenta de linha de comando, mas felizmente o git-tfs permite iniciar uma ferramenta gráfica de check-in com muita facilidade: PS> git tfs checkintool PS> git tfs ct Parece um pouco com isso: 402 148. A ferramenta de check-in git-tfs. Isso parecerá familiar para os usuários do TFS, pois é a mesma caixa de diálogo iniciada no Visual Studio. O Git-tfs também permite controlar ramificações TFVC no seu repositório Git. Como exemplo, vamos criar um: PS> git tfs branch $ / tfvc-test / featureBee O nome da ramificação local será: featureBee C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5 PS> git log --oneline --graph --decorate --all \* 1d54865 (tfs / featureBee) Ramo de criação $ / myproject / featureBee \* ff04e7c (HEAD, tfs / padrão, mestre) código de atualização \* 71a5ddc readme de atualização \* aea74a0 documentação de atualização | \* 44cd729 (tfs / featureA, featureA) Adeus | \* d202b53 Ramificado de $ / tfvc-test / Trunk | / \* c403405 Olá \* b75da1a Novo projeto A criação de uma ramificação no TFVC significa adicionar um conjunto de alterações onde essa ramificação agora existe e isso é projetado como uma confirmação do Git. Observe também que o git-tfs criou a ramificação remota tfs / featureBee, mas o HEAD ainda está apontando para o mestre. Se você deseja trabalhar no ramo recém-cunhado, baseará seus novos commit no commit 1d54865, talvez criando um branch de tópico a partir desse commit. Resumo do Git e TFS Git-tf e Git-tfs são ótimas ferramentas para fazer interface com um servidor TFVC. Eles permitem que você use o poder do Git localmente, evite constantemente ter que fazer uma ida e volta ao servidor TFVC central e facilitar muito a sua vida como desenvolvedor, sem forçar toda a sua equipe a migrar para o Git. Se você estiver trabalhando no Windows (provavelmente sua equipe está usando o TFS), provavelmente usará o git-tfs, pois o conjunto de recursos é mais completo, mas se você estiver trabalhando em outra plataforma, Eu vou usar git-tf, que é mais limitado. Como na maioria das ferramentas deste capítulo, você deve escolher um desses sistemas de controle de versão para ser canônico e usar o outro de maneira subordinada - Git ou TFVC devem ser o centro da colaboração, mas não os dois.

Migrando para o Git

Se você já possui uma base de código em outro VCS, mas decidiu começar a usar o Git, deve migrar seu projeto de uma maneira ou de outra. Esta seção aborda alguns importadores de sistemas comuns e, em seguida, demonstra como desenvolver seu próprio importador personalizado. Você aprenderá como importar dados de vários dos maiores sistemas SCM usados ​​profissionalmente, porque eles compõem a maioria dos usuários que estão trocando e porque as ferramentas de alta qualidade para eles são fáceis de encontrar.

Subversion

Se você leu a seção anterior sobre o uso do git svn, pode facilmente usar essas instruções para o git svn clonar um repositório; então, pare de usar o servidor Subversion, vá para um novo servidor Git e comece a usá-lo. Se você quiser o histórico, poderá fazer isso o mais rápido possível, retirando os dados do servidor Subversion (o que pode demorar um pouco). No entanto, a importação não é perfeita; e, como isso levará muito tempo, você também pode fazer o que é certo. O primeiro problema é a informação do autor. No Subversion, cada pessoa que confirma um usuário no sistema que está registrado nas informações de confirmação. Os exemplos na seção anterior mostram o schacon em alguns lugares, como a saída da culpa e o log do git svn. Se você deseja mapear isso para obter melhores dados do autor do Git, precisará de um mapeamento dos usuários do Subversion para os autores do Git. Crie um arquivo chamado users.txt que tenha esse mapeamento em um formato como este: schacon = Scott Chacon <schacon@geemail.com> selse = Someo Nelse <selse@geemail.com> Para obter uma lista dos nomes dos autores que o SVN usa , você pode executar isso: $ svn log --xml --quiet | autor grep | classificar -u | \ Perl -pe 's /.\*>(.\*?)<.\*/$ 1 = /' Isso gera a saída do log no formato XML, mantém apenas as linhas com informações do autor, descarta duplicatas e retira as tags XML. (Obviamente, isso só funciona em uma máquina com grep, sort e perl instalados.) Em seguida, redirecione essa saída para o arquivo users.txt para adicionar os dados de usuário Git equivalentes ao lado de cada entrada. Você pode fornecer esse arquivo ao git svn para ajudá-lo a mapear os dados do autor com mais precisão. Você também pode dizer ao git svn para não incluir os metadados que o Subversion normalmente importa, passando --no-metadata para o comando clone ou init (embora se você deseja manter os metadados de sincronização, sinta-se à vontade para omitir esse parâmetro). Isso faz com que seu comando de importação fique assim: 404 $ git svn clone http://my-project.googlecode.com/svn/ \ --authors-file = users.txt --no-metadata --prefix "" -s my\_project $ cd my\_project Agora você deve ter uma importação melhor do Subversion no seu diretório my\_project. Em vez de confirmações que se parecem com esta confirmação 37efa680e8473b615de980fa935944215428a35a Autor: schacon <schacon @ 4c93b258-373f-11de-be05-5f7a86268029> Data: Dom 3 de maio 00:12:22 2009 0000 instalação fixa - vá para o tronco git-svn-id : //my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de- be05-5f7a86268029 eles têm a seguinte aparência: commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2 Autor: Scott Chacon <schacon@geemail.com Data> 12:22 2009 0000 instalação fixa - vá para o tronco Não apenas o campo Autor parece muito melhor, mas o git-svn-id também não está mais lá. Você também deve fazer uma limpeza pós-importação. Por um lado, você deve limpar as referências estranhas que o git svn configurou. Primeiro, você move as tags para que elas sejam reais, em vez de ramificações remotas estranhas, e depois move o restante das ramificações para que elas sejam locais. Para mover as tags para as tags Git adequadas, execute: $ for t em $ (git for-each-ref --format = '% (refname: short)' refs / remotes / tags); git tag $ {t / tags \ //} $ t parece correto e um dos quais seria completamente inválido para uma confirmação do Git.

This takes the references that were remote branches that started with refs/remotes/tags/ and

makes them real (lightweight) tags.

Next, move the rest of the references under refs/remotes to be local branches:

$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); do git branch

$b refs/remotes/$b && git branch -D -r $b; done

It may happen that you’ll see some extra branches which are suffixed by @xxx (where xxx is a

number), while in Subversion you only see one branch. This is actually a Subversion feature called

405

“peg-revisions”, which is something that Git simply has no syntactical counterpart for. Hence, git

svn simply adds the svn version number to the branch name just in the same way as you would

have written it in svn to address the peg-revision of that branch. If you do not care anymore about

the peg-revisions, simply remove them:

$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D

$p; done

Now all the old branches are real Git branches and all the old tags are real Git tags.

There’s one last thing to clean up. Unfortunately, git svn creates an extra branch named trunk,

which maps to Subversion’s default branch, but the trunk ref points to the same place as master.

Since master is more idiomatically Git, here’s how to remove the extra branch:

$ git branch -d trunk

The last thing to do is add your new Git server as a remote and push to it. Here is an example of

adding your server as a remote:

$ git remote add origin git@my-git-server:myrepository.git

Because you want all your branches and tags to go up, you can now run this:

$ git push origin --all

$ git push origin --tags

All your branches and tags should be on your new Git server in a nice, clean import.

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit

more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool

called "hg-fast-export", which you’ll need a copy of:

$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

$ hg clone <remote repo URL> /tmp/hg-repo

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what

it will put in the author field for changesets, so this is a good time to clean house. Generating this is

a one-line command in a bash shell:

406

$ cd /tmp/hg-repo

$ hg log | grep user: | sort | uniq | sed 's/user: \*//' > ../authors

This will take a few seconds, depending on how long your project’s history is, and afterwards the

/tmp/authors file will look something like this:

bob

bob@localhost

bob <bob@company.com>

bob jones <bob <AT> company <DOT> com>

Bob Jones <bob@company.com>

Joe Smith <joe@company.com>

In this example, the same person (Bob) has created changesets under four different names, one of

which actually looks correct, and one of which would be completely invalid for a Git commit. Hgfast-

export lets us fix this by adding ={new name and email address} at the end of every line we want

to change, and removing the lines for any usernames that we want to leave alone. If all the

usernames look fine, we won’t need this file at all. In this example, we want our file to look like this:

TFS

Se sua equipe estiver convertendo o controle de origem de TFVC para Git, você desejará a conversão 413 da mais alta fidelidade possível. Isso significa que, enquanto cobrimos git-tfs e git-tf para a seção de interoperabilidade, apenas cobriremos git-tfs para esta parte, porque o git-tfs suporta ramificações, e isso é proibitivamente difícil usando o git-tf. NOTA Esta é uma conversão unidirecional. O repositório Git resultante não poderá se conectar ao projeto TFVC original. A primeira coisa a fazer é mapear nomes de usuários. O TFVC é bastante liberal com o que entra no campo do autor para conjuntos de mudanças, mas o Git deseja um nome e um endereço de email legíveis por humanos. Você pode obter essas informações no cliente de linha de comando tf, assim: PS> tf history $ / myproject -recursive> AUTHORS\_TMP Isso captura todas as alterações no histórico do projeto e as coloca no arquivo AUTHORS\_TMP que processaremos para extrair os dados da coluna Usuário (a segunda). Abra o arquivo e encontre em que caracteres iniciam e terminam a coluna e substitua, na seguinte linha de comando, os parâmetros 11-20 do comando cut pelos encontrados: PS> cat AUTHORS\_TMP | corte -b 11-20 | cauda -n 3 | classificar | uniq> AUTORES O comando recortar mantém apenas os caracteres entre 11 e 20 de cada linha. O comando tail ignora as duas primeiras linhas, que são cabeçalhos de campo e sublinhados de arte ASCII. O resultado de tudo isso é canalizado para classificar e uniq para eliminar duplicatas e salvo em um arquivo chamado AUTHORS. O próximo passo é manual; para que o git-tfs faça uso efetivo desse arquivo, cada linha deve estar neste formato: DOMÍNIO \ nome\_do\_usuário = Nome do Usuário <email@address.com> A parte à esquerda é o campo "Usuário" do TFVC e a parte no lado direito do sinal de igual é o nome de usuário que será usado para confirmações do Git. Depois de ter esse arquivo, a próxima coisa a fazer é criar um clone completo do projeto TFVC no qual você está interessado: PS> clone git tfs --with-branches --authors = AUTHORS https://username.visualstudio.com / DefaultCollection $ / project / Trunk project\_git Em seguida, você deseja limpar as seções git-tfs-id na parte inferior das mensagens de confirmação. O comando a seguir fará isso: PS> git filter-branch -f --msg-filter 'sed "s / ^ git-tfs-id:. \* $ // g"' '-' --all Isso usa o comando sed do ambiente Git-bash para substituir qualquer linha iniciada por "git-tfs-id:" pelo vazio, que o Git ignorará. 414 Quando tudo estiver pronto, você estará pronto para adicionar um novo controle remoto, empurrar todas as suas branches e fazer com que sua equipe comece a trabalhar no Git.