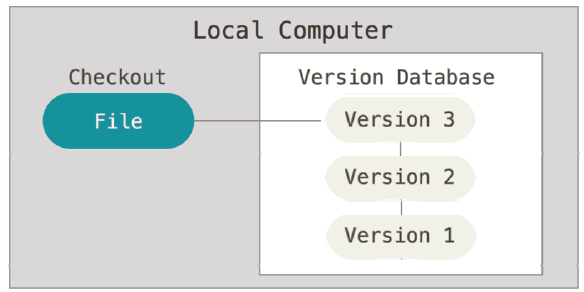
**Introdução com ao Git**

O Git é um sistema de controle de versão (VCS), um software projetado para registrar alterações feitas em um arquivo ou conjunto de arquivos ao longo do tempo. Isso permite reverter ou recuperar alterações em um arquivo ou em um conjunto de arquivos após as alterações. Se você estiver trabalhando em um projeto que contém muitos arquivos, o controle de versão oferece a capacidade de controlar todo o projeto e você pode reverter apenas um ou vários arquivos para as versões anteriores ou todo o projeto de volta. Você pode comparar as alterações feitas nos arquivos de uma versão para outra e pode ver exatamente o que foi alterado em cada arquivo quando a alteração foi introduzida ou quem a introduziu.

O melhor de um VCS é que você controla quase qualquer tipo de arquivo em um computador, não apenas os arquivos de código-fonte. Estudantes costumam usar o controle de versão para acompanhar diferentes versões de um ensaio ou trabalho de pesquisa. Um designer pode usar o controle de versão para manter as versões de uma imagem sem se preocupar em modificar a única versão dessa imagem e o controle de versão oferece a capacidade de explorar novas ideias com seu projeto, com a garantia de que você sempre pode voltar a uma versão anterior.

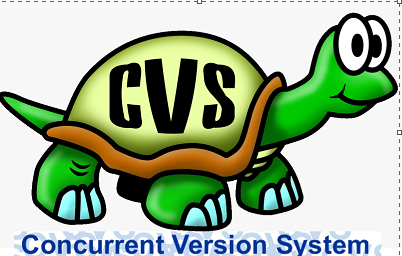
**Sistemas Locais de Controle de Versão**

O método de controle de versão de muitas pessoas é copiar os arquivos para outro diretório (talvez um diretório com carimbo de tempo, se eles forem espertos). Esta abordagem é muito comum porque é muito simples, mas também é incrivelmente propensa a erros. É fácil esquecer em qual diretório você está e acidentalmente sobreescrever o arquivo errado ou copiar arquivos que não quer.

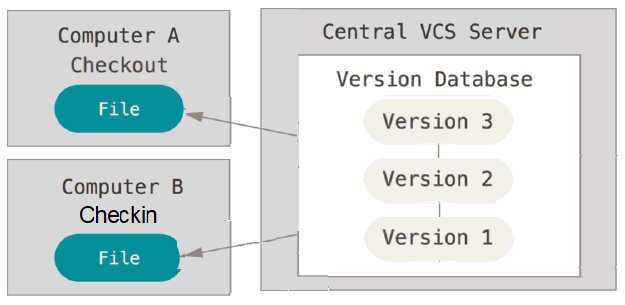


Para lidar com este problema, programadores há muito tempo desenvolveram VCSs locais que tem um banco de dados simples que mantêm todas as alterações nos arquivos sob controle de revisão. No começo dos anos 70 já havia o conceito de versionamento de código fonte. Em 1972 já havia o SCCS (Source Code Control System) desenvolvido na linguagem Snow Ball na Bell Labs e usado em sistemas como o IBM system 370 até o PDP11. Nos 80 foi surgiu sua evolução que foi o RCS (Revision Control System) mantido até hoje. Até mesmo o popular sistema operacional Mac OS X inclui o comando rcs quando você instala as Ferramentas de Desenvolvimento. RCS funciona mantendo conjuntos de alterações (ou seja, as diferenças entre os arquivos) em um formato especial no disco; ele pode, em seguida, re-criar como qualquer arquivo se parecia em qualquer ponto no tempo, adicionando-se todas as alterações.

**Sistemas Centralizados de Controle de Versão**

A próxima questão importante que as pessoas encontram é que elas precisam colaborar com desenvolvedores em outros sistemas. Para lidar com este problema, Sistemas Centralizados de Controle de Versão (CVCSs) foram desenvolvidos Finalmente por volta de 1990 surgiu o CVS (concorrent Version System) para substituir o RCS, têm um único servidor que contém todos os arquivos de controle de versão, e um número de clientes que usam arquivos a partir desse lugar central modelo chamado hoje de centralizado ou cliente-servidor que era arquitetura da moda nos anos 80 e 90.

Neste modelo configurava-se um servidor onde ficava o projeto e na máquina de cada desenvolvedor era instalado o cliente que conseguia se conectar nesse servidor. O servidor era responsável por coisas como autenticar o login de cada desenvolvedor para dar permissão de acesso ao código daí o desenvolvedor podia fazer check-out para baixar todo o código fazer check-in para subir as modificações e de tempos em tempos dar update para baixar as últimas modificações do servidor. O servidor também guardava os deltas de modificação, possível graças a ideia de patches. Novas versões adicionaram coisas como delta compresion a economizar espaço no disco guardando os patches zipados e novas funcionalidades foram aparecendo com uma ideia the branchs.



Esta configuração oferece muitas vantagens, especialmente sobre VCSs locais. Por exemplo, todo mundo sabe, até certo ponto o que todo mundo no projeto está fazendo. Os administradores têm controle refinado sobre quem pode fazer o que; e é muito mais fácil de administrar um CVCS do que lidar com bancos de dados locais em cada cliente.

Os branch é um conceito simples, mas com operação complicada. O branch é duplicar o diretório do projeto, trabalhar no diretório separado e quando terminar, gerar os patches para aplicar no diretório principal e no fim apagar o diretório duplicado. É uma forma de trabalhar numa funcionalidade que pode demorar ou testar correção de bugs sem atrapalhar o diretório principal. Isso funciona bem se você trabalha sozinho ou com poucas pessoas ou com pessoas que estão muito bem coordenadas entre si caso contrário isso se torna um pesadelo muito rápido. Nos dias de hoje você tem pull requests no GitHub que basicamente um branch do projeto.

 Após o CVS, surgiu o Subversion com o lema de “CVS feito direito” que ainda era no modelo centralizado e com muito problemas ainda em branchs. Só conseguiram consertar branchs tarde demais adicionando a funcionalidade de Merge tracking, na versão 1.5 que saiu em julho de 2008. Porém foi neste mesmo ano que o GitHub atingiu o pico de popularidade.

O conceito de bloquear arquivos para só uma pessoa usar de cada vez é uma desgraça pior ainda para gerenciar Releases ou fechar versões. Nos anos 90 e começo nos 2000 havia a figura de um gestor de configuração pago em tempo integral para fazer o equivalente fazer commits e tagear versões para não corromper muito o código-fonte.

O ponto único de falha que o servidor centralizado representa era outro problema. Se esse servidor der problema por uma hora, durante essa hora ninguém pode colaborar ou salvar as alterações de versão para o que quer que eles estejam trabalhando. Se o disco rígido do banco de dados central for corrompido, e backups apropriados não foram mantidos, você perde absolutamente tudo - toda a história do projeto, exceto imagens pontuais que desenvolvedores possam ter em suas máquinas locais. Sistemas VCS locais sofrem com esse mesmo problema - sempre que você tenha toda a história do projeto em um único lugar, há o risco de perder tudo.

Outro problema eram as branchs e a razão principal de porque CVS e Subversion tiveram reputação tão ruim. Conflitos de merge é o pesadelo de todo desenvolvedor de sistemas antigos e incompetentes., pois nos sistemas antigos, fazer branches e merges eram complicados você evitava ao máximo fazer, só que quanto mais você espera também mais caro vai acumulando as diferenças no branch e conflitos no merge. Por isso, a maioria simplesmente desistia de fazer branches e todo mundo trabalhava na mesma versão que no Subversion se chama **trunk** e no Git é chamado de **máster** e havia muita discussão sobre quem podia ter acesso de committ no máster. No Git, trabalhar em branches e merges é comum, você raramente encontra conflitos e mesmo quando encontra eles costumam ser fáceis de resolver pois, geralmente, você faz merges com frequência e evita deixar código não mergeado esquecido num branch por semanas ou meses.

Outro grande defeito estrutural do CVS é que ele guardava a revisões por arquivo ele não entende o conceito de um único arquivão (TAR) de patch de uma funcionalidade inteira como um pull request de hoje (que é um conjunto de commits). De forma simplória um commit é nada mais que um arquivo de patch com o cabeçalho com metadados (autor, data e hora). Era difícil de gerenciar a versão no CVS, pois cada desenvolvedor precisava trabalhar em um branch ou fork diferente (uma duplicata do projeto original) só que para combinar as modificações de dezenas de desenvolvedores em uma única versão eram um trabalho que podia levar um dia inteiro ou mais para conseguir fazer esse merge.

Mesmo Subversion não era nem uma grande maravilha era mais a filosofia do menos pior mesmo, o mundo open-source e o mundo corporativo estavam bem mal servidos, haviam produto comercial bem porcaria como Microsoft Source Safe ou Rational Clear Case. Mundo open-source ainda não tinha tanta força muitos projetos ainda usavam CVS alguns poucos estavam indo para Subversion.

Na virada do século surge a Bit Mover e seu produto Bitkeeper muita gente pensa que a principal diferença entre os sistemas de versionamento anteriores é que seria descentralizado.



Na realidade a principal novidade foi tornar branchs, forks e principalmente merges operações razoavelmente triviais. Um trabalho que podia levar horas ou dias reduzido para minutos e com muito menos margem de erros. Assim em 2002, apesar de ser um produto comercial de código fechado, o Linus resolve migrar o desenvolvimento do kernell do Linux para o Bitkeeper pois, assim como GitHub, apesar de serem uma ferramenta fechada eles tinham uma política que deixavam projetos open-source usar a ferramenta gratuitamente.

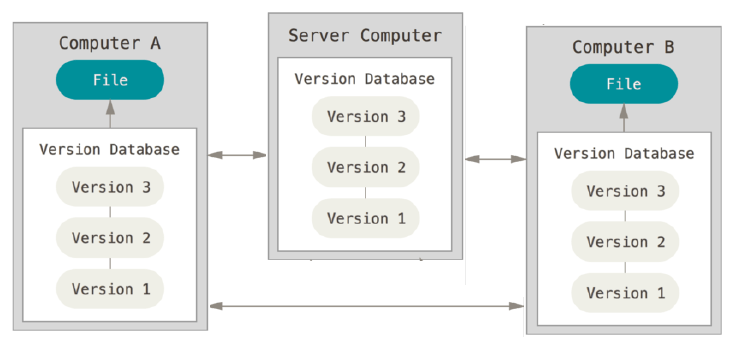
O problema de confiar uma ferramenta comercial é que por um tempo eles permitem usar de graça mas chega um dia que eles podem simplesmente proibir de usar. E isso que ocorreu na relação do BitKeeper com o Linux, após dois ou três anos, por que alguém tentou fazer engenharia reversa do protocolo do Bitkeeper, a Bit Mover resolveu botar restrições demais ao ponto que o Linux ia ficar sem o Bitkeeper.

**Sistemas Distribuídos de Controle de Versão**

Outras alternativas comerciais surgiram em 2003 (Darcs), em 2005 (Mercurial, Bazaar e Git), é aqui que Sistemas Distribuídos de Controle de Versão (DVCS) entram em cena.



Em um DVCS, clientes não somente usam o estado mais recente dos arquivos: eles duplicam localmente o repositório completo. Assim, se qualquer servidor morrer, e esses sistemas estiverem colaborando por meio dele, qualquer um dos repositórios de clientes podem ser copiado de volta para o servidor para restaurá-lo. Cada clone é de fato um backup completo de todos os dados. O foco dos DVCS era també em resolver o problema de branchs e merges.



Além disso, muitos desses sistemas trabalham muito bem com vários repositórios remotos, tal que você possa colaborar em diferentes grupos de pessoas de maneiras diferentes ao mesmo tempo dentro do mesmo projeto. Isso permite que você configure vários tipos de fluxos de trabalho que não são possíveis em sistemas centralizados, como modelos hierárquicos.

**Uma Breve História do Git**

O núcleo (kernel) do Linux é um projeto de código aberto com um escopo bastante grande. A maior parte da vida da manutenção do núcleo o Linux (1991-2002), as mudanças no código eram feitas com patches e tarball.

* **Patches:** termo da língua inglesa que significa, literalmente, "remendo". É um programa de computador criado para atualizar ou corrigir um software de forma a melhorar sua usabilidade ou performance. Quando patches corrigem bugs ou vulnerabilidades de segurança, se dá o nome de bugfix. Esta técnica pode ser utilizada como uma das formas de distribuir uma liberação de software. É muito utilizada em jogos para computadores de forma a atualizar o conteúdo disponível.
* **Tarball**: tar é um utilitário de software de computador para coletar muitos arquivos em um único arquivo, geralmente chamado de tarball, para fins de distribuição ou backup. O nome é derivado de "Tape ARchive", pois foi desenvolvido originalmente para gravar dados em dispositivos de E/S seqüenciais sem sistema de arquivos próprio. Os conjuntos de dados de archive criados pelo tar contêm vários parâmetros do sistema de arquivos, como nome, data e hora, propriedade, permissões de acesso a arquivos e organização de diretórios. O utilitário de linha de comando foi introduzido pela primeira vez na versão 7 Unix em janeiro de 1979, substituindo o programa tp. A estrutura do arquivo para armazenar essas informações foi padronizada no POSIX.1-1988 e posteriormente no POSIX.1-2001, e tornou-se um formato suportado pela maioria dos sistemas modernos de arquivamento de arquivos.

Por volta dos anos 70 na época dos Unix que surgiram duas ferramentas que existem até hoje no Linux. Que são termos que todo mundo de Unix e Linux sempre usa mas pouco sabem o que realmente é:

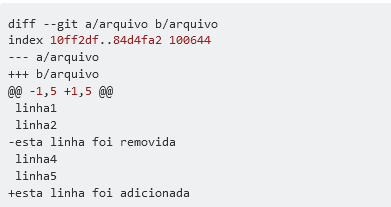
1. **Ferramenta diff:**
2. **Ferramenta patch:**

**Ferramenta diff**

**N**asceu por volta de 1976 e é usado para tirar diferença entre dois arquivos em um terceiro arquivo. É um problema exponencial que fica lento muito rápido se implementar de jeito incorreto mas existe um algoritmo bem conhecido para encontrar a subsequência comum mais longa (**Longgest Common Subsequence**) entre dois strings:

* No pior caso a complexidade vai ser de O(2m) sendo m o tamanho de um dos arquivos.
* No melhor caso a complexidade cai para O(mv) sendo n e m o tamanho dos dois arquivos

Para entender como você consegue achar essa subsequencia e a partir dessa subsequencia se um conteúdo existe no arquivo original mas não existe na subsequencia podemos marcar com o caracter com o sinal de menos(-) se não existe no arquivo original mais existe na subsequencia podemos marcar com o sinal de mais(+). Adiciona nessa subsequência alguns metadados como em qual linha do arquivo aquela subsequencia se localiza e chegamos no que chamamos o formato de um patch. Abaixo um exemplo de saída com comando diff do Git:



Então a ferramenta diff que todo o linux tem consegue achar a subsequencia como mais longa entre dois arquivos adiciona metadados como o número da linha no arquivo e gera esse arquivo de patch (em inglês, remendo) alguns dizem que essa é a origem do nome Apache o servidor web que era feito com um monte de patches ou seja o tipo disso a parte enfim dentro de um arquivo de patch você tem **chunk** ou **hunk** que é literalmente nacos de código. Vários nacos de código ou hunks formam um patch que é um remendo daí digamos que você queira mandar as modificações que você fez no arquivo para um amigo seu que tem o mesmo arquivo, você só envia o arquivo de patch

**Ferramenta patch**

E no linux tem a segunda ferramenta que se chama **patch** e é feita para aplicar um arquivo de patch no arquivo original para ter a versão novas. Se observar um projeto GitHub já deve ter visto o formato de patches só que colorizado mas é basicamente a mesma coisa. A ferramenta diff tornou mais econômico guardar só as modificações feitas sobre o arquivo original em vez de duplicar o arquivo toda vez só por causa de uma linha modificada então economizamos tanto espaço de disco como banda de rede já que para transferir as modificações pela rede agora pode ir só o patch. Mas observe que se você tiver outro programador que já está colaborando no mesmo projeto não basta só dar o patch para ele ele precisa ter alguma versão do projeto baixado para aplicar esse patch

Para baixar o projeto nos primórdios da internet era outro problema. Uma forma de fazer isso seria baixar de um servidor de arquivos, só que com baixa velocidade de tráfego de rede, dependendo do tamanho do projeto pode ser um processo bem lento. Nos anos 70 para trás disco rígido era uma coisa bem cara e a forma mais comuns nos mainframes era armazenada rolos de fita magnética e era preciso tentar desperdiçar menos espaço possível. Daí no fim dos anos 70 nasceu o formato de gravação de arquivos em fita ou “tape arquive”(TAR) a estrutura de dados de um tar é gravar uma série de objetos um atrás do outro tá é traduzido literalmente com alcatrão, poluente de petróleo, um bola de piche. Então se você tem uma bola de piche onde objetos vão grudando você tem um TAR. Na prática pense que em vez de ter 100 arquivos você combinou eles um atrás do outro em um único arquivão. Na prática pense que em vez de ter 100 arquivos você combinou eles um atrás do outro em um único arquivão.

E para ser mais eficiente o comando TAR dos de unix e linux ganhou capacidade de adicionar compressão sem perda a partir dos anos 90. Compressão existem dois tipos

* **Lossy(que perde):** É extensão jpg para imagens ou mp3 para músic
* **Lossless(que não perde):**  É extensão zip sevenzip, um exemplo algoritmo de compressão mais simples é o algoritmo de **Huffman** para a compreensão de texto. O formato gzip você vê até hoje em linux usa oalgoritmo **Deflate** que é uma combinação de uma variação do **LZ(Lempel-Ziv)** e do Huffman

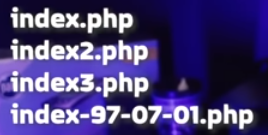
Algoritmos de compressãoé outra área que os russos dominam o formato RAR vem de **Roschal Archive** que é o nome do inventor russo Eugeny **Roschal. O** zip e o sevenzip vem de outro russo Igor Pavlov. No fim dos anos 80 uns compressores mais famosos na época do MS-DOS eram pkzip que era do americano Phil Katz que também inventou o Deflate do Unix.

Com essas ferramentas já era possível montar um fluxo de trabalho para um projeto como a kernel do Linux. O Linus Torvalds fez um TARball da versão 1.0 da Kernell do Linuz e colocava no servidor FTP público um outro desenvolvedor como Alan Cox ou Andrew Morton baixa do FTP e começa a desenvolver alguma nova funcionalidade quando termina faz um diff dos arquivos e envie anexado no e-mail de volta para o Linus. No unix e Linux existem comandos de terminal para conectar em servidores pop3 ou imap que a gente usa hoje em dia então o Linus conseguia automatizar, baixando o e-mail separava o anexo e usava o comando patch do Linux para aplicar os patches na versão na sua máquina. Se tudo estiver certoi ele fazia um novo TARball versão 1.1 e colocava no servidor. Se você já viu um servidor de FTP já deve ter visto diretório de projetos com TARball de diversas versões. Não é um fluxo de trabalho eficiente em 2020 mais nos anos 90 a gente tá falando do que havia de mais avançado em fluxo de trabalho distribuído. Então a internet, somado com ferramental Linux como diff e patch TAR, gzip, FTP e e-mail permitiram um fluxo de trabalho distribuído e o nascimento do mundo open source como conhecemos hoje alguma variação disso já existia no mundo em Unix nos anos 80 mas foi com o Linux que esse modelo de trabalho começou a ganhar chão.

Em 2002, o projeto do núcleo do Linux começou usar uma DVCS proprietária chamada BitKeeper. Em 2005, a ferramenta passou a ser paga. Depois de lançar a kernell 2.6, por volta de 2005, o Linus Torvalds desenvolve em duas semanas a sua própria ferramenta baseada em lições aprendidas ao usar o BitKeeper. Algumas metas do novo sistema era os seguintes:

* Velocidade
* Projeto simples
* Forte suporte para desenvolvimento não-linear (milhares de ramos paralelos)
* Completamente distribuído
* Lidar com projetos grandes como o núcleo o Linux com eficiência (velocidade e tamanho dos dados)

Os problemas resolvidos com o Git, e que não eram pelos versionadores da época, foram:

1. **Versionamento:** é quando você escreve um código e vai salvando em algum diretório e cai nos mesmos erros que todo programador sempre caiu, você vai escrever em cima do seu código de ontem, salvar e depois descobri que fez errado e agora perdeu a versão de ontem. Ou pode ser que você queira renomear ou apagar arquivos e uma hora vai apagar um arquivo que não devia. Depois de apagar coisas por acidente algumas vezes você vai começar a duplicar os arquivos antes de editar e vai começar a ficar com versões diferentes do mesmo arquivo no mesmo diretório e encontra projetos dos outros cheios de arquivos velhos que não deviam estar lá. Esse procedimento de cópias de arquivos, além de ser super ineficiente não só porque vai sujando seu diretório com arquivos desnecessários mais vira um enorme desperdício de espaço em disco.
2. **Compartilhamento:** o segundo problema é qunado você precisa que outro desenvolvedor trabalhe no mesmo projeto que você. Uma das maneiras de fazer isso é colocaro diretório do projeto em alguma pasta compartilhada na rede local daí acontecede os dois abrir o mesmo arquivo para editar você digita e salva o primeiro daí o segundo cara edita a parte dele e salva por cima e pronto você acabou de perder o que estava fazendo. Para resolver parcialmente esse problema que muitos produtos comerciais daquela época como o Source Safe ou Clear Case trabalhavam com o princípio de ter uma camada por cima do sistema de arquivos de rede e ofereciam a opção de bloquear um arquivo ou diretório (dar check-out) daí se esse outro programador tentar abrir o mesmo arquivo sistema vai negar e ele tem que esperar você terminar salvar e dar check-in para desbloquear o arquivo. Isso resolvia parcialmente o problema pois em projetos distribuídos em regiões distantes, caso alguém deixasse de realizar o check-in de seus arquivos, bloqueava o arquivo para edição para outro programador. Esses produtos atendiam somente razoavelmente em casos de equipes pequenas lozalizadas no mesmo escritótio, da mesma empresa, cenários bem limitados.
3. **Segurança:** Uma das coisas que o Linus queria garantir é que os dados no repositório Git sejam sempre confiáveis, ou seja, que que o texto que eu recupero dele é sempre o mesmo que coloquei nele. Para garantir isso, o Git commit, após fazer o comando diff e fazer o patch das alterações ele passa por um algoritmo de hash SHA-1. O Git armazena tudo em seu banco de dados não pelo nome do arquivo, mas pelo valor de hash do seu conteúdo. Se modificar um bit dentro desse patch o SHA-1 vai mudar completamente e gerando uma nova assinatura, sabemos que o patch está corrompido.

O SHA-1 é uma sequência de 40 caracteres composta de caracteres hexadecimais (0-9 e-f) e é calculada com base no conteúdo de uma estrutura de arquivo ou diretório no Git. Um hash SHA-1 é algo como o seguinte:

24b9da6552252987aa493b52f8696cd6d3b00373

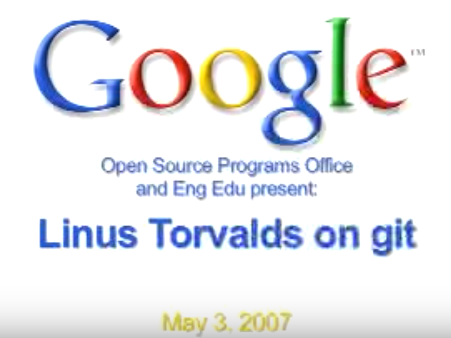
Os subdiretórios, .cvs (no CVC) e .svn (no Subversion), começavam com o ponto porque em Linux um arquivo que começa com ponto significa escondido. Quando mexia nesses diretórios tinha muita chance de estragar tudo e tem que dar um checkout de novo para reconstruir certo. Em 2003, nem Windows XP em Windows Server entendiam direito arquivos que começavam com o ponto e era um dos motivos principais de quase ninguém usar Subversion no Visual Studio pois não conseguia abrir os projetos. Hoje em dia o Windows já entende arquivos e diretórios que começa com ponto porque o Git faz a mesma coisa ele cria um diretório chamado .git mas ele é diferente do CVC e Subversion porque na na realidade agora os arquivos diretórios do projeto que você edita que não tem importância o conteúdo de verdade dos arquivos fica no repositório dentro desse .git e por isso só tem um a raiz e não em cada um dos subdiretórios do projeto (como era no CVC e Subversion).

No Git, suponhamos que você crie um projeto de Node.js, C# e faça o comando git init para iniciar, git add para preparar e o primeiro git commit para confirmar. Depois, por acidente, apague os arquivos originais com exceção do diretório .git, se você der o comando git checkout, o Git vai procurar o arquivo dentro desse .git e recuperar os arquivos sem precisar perguntar nada para nenhum servidor. Porém, se você listar o que tem dentro desse diretório não irá achar os arquivos do projeto, em vez disso vai ter subdiretórios em que o maior é o .git/object (a base de ddos). Nele vai achar um monte de arquivos com nomes de hashes de SHA-1 dos commits, trees e blobs que é a base que o git usa para recuperar os arquivos, criar branchs, etc.

**Performance:** Os versionadores Mercurial, Bazaar e Darcs apesar de serem descentralizados tinham problema de performance. Mercurial, Bazaar eram feitos em python então eles não tinham como escalar em performance até certo ponto. Darcs era feito em Haskell e também era criticado por problemas de performance. O Git sempre foi super-rápido, porém difícil de usar, mas em casos como esse performance e segurança têm prioridade. O git commit faz algo equivalente ao comando diff, é um patch de modificações desde o commit anterior. O merge faz algo equivalente ao comando patch e o pull request seria equivalente a um Tarball, ou seja, um conjunto de commits. Esses conceitos que garantiram a performance superior do Git em relação a outros versionadores.

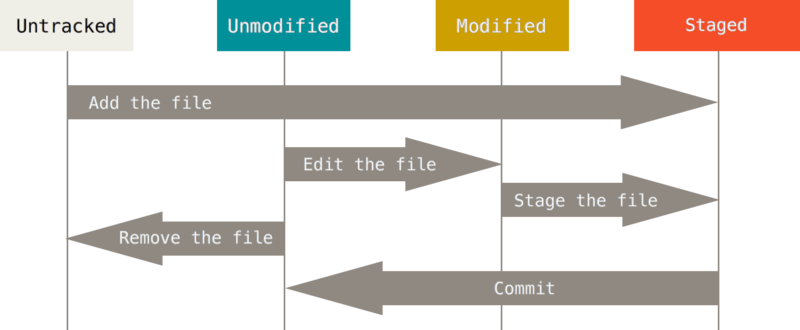
Em 2007 ninguém fora dos colaboradores da kernel do linux tinham muita ideia sobre Git. Em maio de 2007 o próprio Linus Torvalds dá uma palestra em uma tech-talk dentro do Google:

**https://www.youtube.com/watch?v=4XpnKHJAok8**



**Os Três Estados**

Quando você faz algo no Git, quase sempre dados são adicionados no banco de dados do Git - e não removidos. É difícil fazer algo no sistema que não seja reversível ou para fazê-lo apagar dados de forma alguma. Como em qualquer VCS, você pode perder alterações que ainda não tenham sido adicionadas em um commit; mas depois de fazer o commit no Git do estado atual das alterações, é muito difícil que haja alguma perda, especialmente se você enviar regularmente o seu banco de dados para outro repositório. Isso faz com que o uso do Git seja somente alegria, porque sabemos que podemos experimentar sem o perigo de estragar algo.



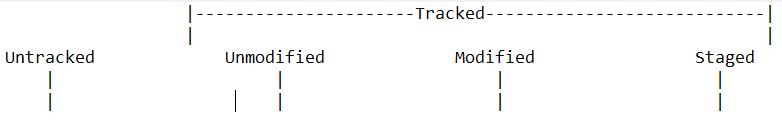
Cada um dos arquivos no diretório de trabalho está em um dos dois estados de alto nível:

1. **Arquivos rastreados (tracked files)** - são arquivos que estavam no último instantâneo (snapshot); eles podem estar em um dos seguintes sub-estados:

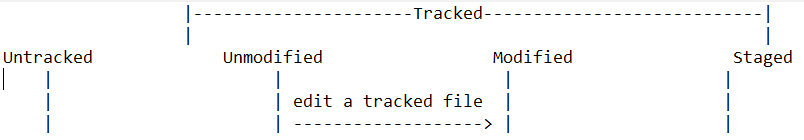
* **Não modificado (unmodified/commited)** - são todos os arquivos que não foram modificados desde a última confirmação. Eles ainda serão incluídos no próximo commit, mas permanecerão como estão.
* **Modificados (modified)** - Estes são os arquivos que foram modificados desde a última confirmação (provavelmente os modificamos como parte das correções). Esses arquivos serão incluídos no próximo commit, mas serão incluídos no respectivo novo formulário.
* **Preparado (staged)** - Esses são arquivos que não estão presentes no último commit (por exemplo, arquivos recém-criados) ou são arquivos "modificados" que dizemos ao git para incluir no próximo commit. Os arquivos são adicionados à preparação usando o comando "add" do git.

1. **Arquivos não rastreados (untracked files)** - todos os arquivos que não estão sendo rastreados, ou seja, o Git não existe. Você pode alterar o estado desses arquivos para "rastreado - preparado" usando o comando "add" do Git.

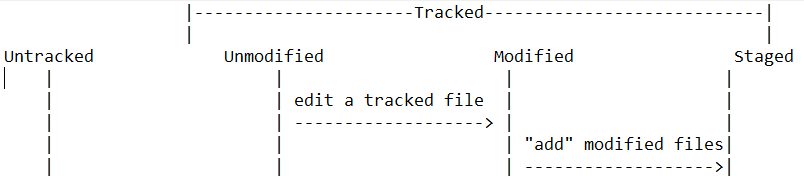
Se você clonar um repositório git, todos os arquivos do projeto serão iniciados no estado "não modificado".



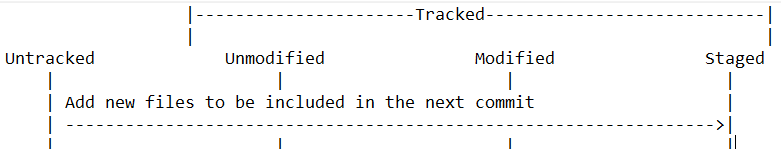
Se você editar qualquer um desses arquivos (por exemplo, usando um editor de texto como o vim), o estado desses arquivos será alterado para "modificado"



Nesse estágio, se você fizer um git "commit", o arquivo editado não será confirmado, porque os arquivos modificados precisam primeiro ser "testados" para serem incluídos no próximo commit. Isso é feito usando o comando "add" do git:

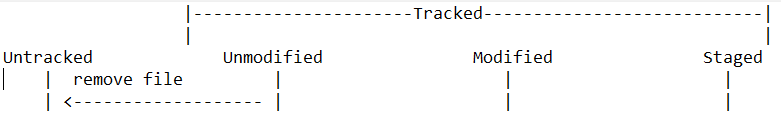


Você também usa o comando "add" do git para ter novos arquivos "staged" para que eles sejam incluídos no próximo commit:



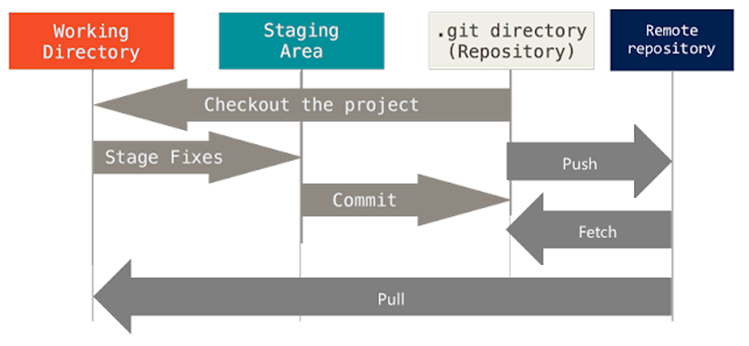
Depois disso, quando estiver satisfeito com tudo, você tira uma foto instantânea usando o comando "commit" do git. Isso resulta na sobreposição dos arquivos do commit anterior com os arquivos recentemente preparados, a fim de criar um novo instantâneo (também conhecido como commit). Você então repete o ciclo novamente.

Outra coisa que você pode querer fazer é remover um arquivo rastreado do projeto. Isso é feito usando o comando rm do git:



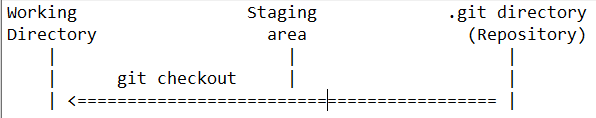
.

Um projeto git é composto de 3 partes:

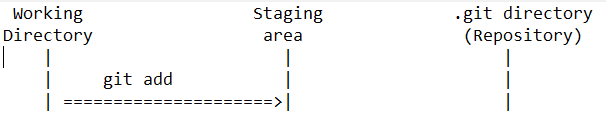


* **Diretório de trabalho** - Este é o diretório em que todos os arquivos e pastas do projeto residem (junto com a pasta .git). Cada um dos seus arquivos neste diretório está em um dos possíveis estados, não rastreados, não modificados, modificados, em etapas. Abordaremos mais sobre os estados dos arquivos posteriormente.
* **Área de preparação (staging area/Index) -** Essa é uma camada hipotética que fica no topo da camada da última confirmação. Quando você executa o comando "git commit", as duas camadas são mescladas. Todos os arquivos que estão na camada de confirmação, que possuem um arquivo mais novo diretamente acima (na "camada intermediária"), serão sobrescritos pelo arquivo mais recente (preparado). Observe que o conteúdo de um arquivo é rastreado para que você possa reverter a aparência do arquivo nos instantâneos anteriores (confirmações). Observe que a área de preparação também é conhecida como "índice"
* **Diretório. git** - este é basicamente o banco de dados do seu repositório git.

Se você deseja obter um novo arquivo rastreado pelo repositório git, use o comando “git checkout” para efetuar o checkout do projeto. Isso extrairá o commit mais recente (instantâneo) do banco de dados do repositório .git (para uma ramificação específica) e o colocará no diretório de trabalho. Todos os arquivos recém-retirados têm o estado do arquivo “não modificado” (mais sobre os estados dos arquivos posteriormente).



Adicione/crie o arquivo em algum lugar dentro do diretório de trabalho. Isso tornará o git ciente da existência desse arquivo e não o acompanhará. ou seja, o estado do arquivo é "não rastreado" (mais sobre os estados do arquivo posteriormente). Use o comando "git add" para colocar esse arquivo na área de preparação, esperando para ser mesclado na confirmação anterior (captura instantânea). Isso mudará o estado do arquivo para "staged" (mais sobre os estados do arquivo posteriormente).



Use o comando "git commit" para adicionar o arquivo e criar uma nova captura instantânea composta da "layer " de confirmação anterior, juntamente com os arquivos na "staged layer". Isso mudará o estado do arquivo para "não modificado" (mais sobre os estados do arquivo posteriormente).



**Usando a linha de comando**

Antes de instalar o Git, precisamos nos familiarizar um pouco com a linha de comando. É importante observar que muitos desses comandos dependem do local do diretório atual; portanto, você deve certificar-se de estar no diretório de projetos antes de executar seus comandos.

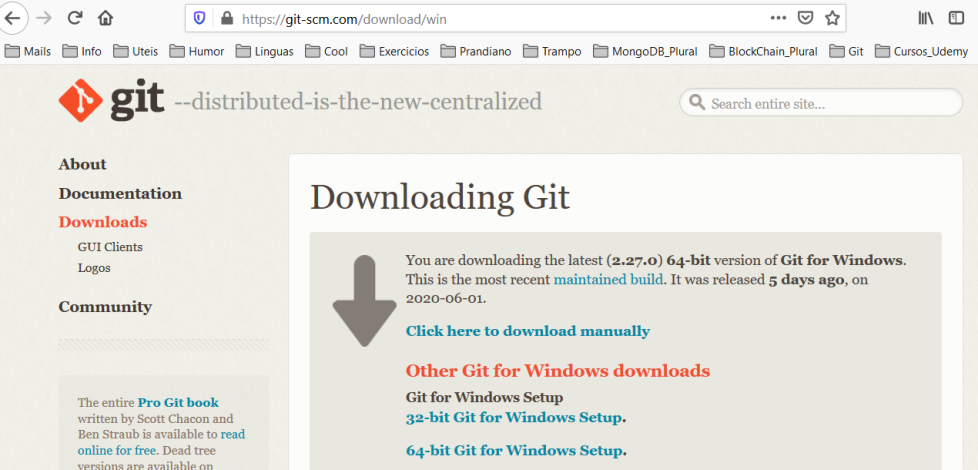
|  |  |  |
| --- | --- | --- |
| **Windows** | **Linux / Mac** | **Descrição** |
| date | date | Exibir a data atual. |
| time | cal | Exibir a hora atual. / Exibir um calendário. |
| echo %cd% | pwd | Exibir qual é o diretório atual. |
| dir | ls / ls –l / ls -la | Exibir arquivos/diretórios do diretório atual. |
| cd | cd | Mudar o diretório atual. |
| cd .. | cd .. | Voltar para o diretório acima. |
| mkdir | mkdir | Criar uma pasta. |
| copy | cp | Copiar (duplicar) um arquivo. |
| move | mv | Mover (ou renomear) um arquivo ou pasta. |
| del | rm | Deletar um arquivo |
| rmdir | rm -rf | Deletar uma pasta vazia. |
| rmdir /s |  | Deletar uma pasta que não esteja vazia. |
| type | cat | Exibir o conteúdo de um arquivo. |
| cls | clear | Limpar o terminal. |
| copy NUL | touch | Criar um arquivo em branco (vazio) |
| echo >> FILE |
| tree /f /a | tree | Mostra estrutura em árvore de arquivos e subdiretórios de um diretório especificado |
| start | open | Abre em visualização gráfica o diretório especificado |
| -h ou --help | -h ou --help | Ajuda com um determinado comando |

**Instale o Git**

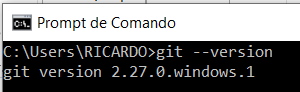
Para instalar o Git no Windows faça o download no link abaixo:

**https://git-scm.com/download/win**

O Git para Windows fornece uma emulação BASH chamada Git BASH usada para executar o Git a partir da linha de comando. Os usuários do NIX devem se sentir em casa, pois a emulação BASH se comporta exatamente como o comando "git" nos ambientes LINUX e UNIX. Como os usuários do Windows normalmente esperam interfaces gráficas, o Git for Windows também fornece a GUI do Git, uma alternativa poderosa ao Git BASH, oferecendo uma versão gráfica de quase todas as funções da linha de comando do Git, além de ferramentas visuais de diferenças.



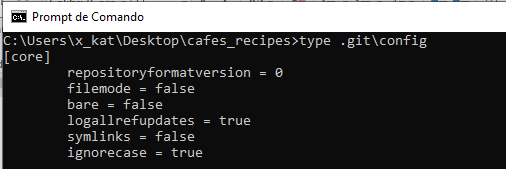
Depois de concluir a instalação, você pode verificar se a instalação foi bem-sucedida digite git –versionpara obter a versão do Git:



**Git config**

Agora que verificamos que o Git foi instalado, a primeira coisa que devemos fazer é definir nosso nome de usuário e endereço de email quando usamos o Git para fazer alterações em nossos arquivos do projeto. O Git usa essas informações para identificar quem fez essas alterações. O Git possui uma ferramenta chamada git config que permite definir configurações.

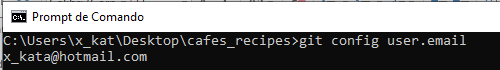
O git config é a função conveniente usada para definir os valores de configuração do Git em projetos de nível global ou local. Esses níveis de configuração correspondem aos arquivos de texto do .git/config.



Executar o git config modifica arquivos de texto de configuração. A gente vai falar das configurações comuns, como e-mail, nome de usuário e editor. Também vamos falar sobre aliases do Git, que permitem a criação de atalhos para operações usadas com frequência. Saber mais sobre o git config e sobre as diversas configurações do Git ajudam a criar fluxos de trabalho Git poderosos e personalizados.

Se você quiser verificar suas configurações, digite git config --list, e ele listará todas as configurações que Git possui. Se você deseja verificar uma configuração específica, é invocar o comando com o nome de configuração. Isso vai exibir o valor definido desse nome. Nomes de configuração são cadeias de caracteres delimitadas por pontos compostas de "seções" e "chaves" baseadas na hierarquia. Por exemplo: user.email

**git config user.email**



Nesse exemplo, o e-mail é propriedade filho do bloco de configuração do usuário. Isso retorna o endereço de e-mail configurado que o Git associa com commits criados no local.

**Níveis e arquivos do git config**

Antes de falar mais sobre o uso do git config, a gente vai discutir os níveis de configuração. O comando git config pode aceitar argumentos para especificar o nível de configuração no qual operar. Os seguintes níveis de configuração estão disponíveis:

|  |  |
| --- | --- |
| **Nível** | **Descrição** |
| --local | Por padrão, o git config grava em nível local se não houver nenhuma opção de configuração. A configuração de nível local é aplicada no repositório do contexto que o git config for invocado. Os valores de configuração local são armazenados no arquivo que pode ser encontrado no diretório .git do repositório: .git/config |
| --global | A configuração de nível global é específica do usuário. Isso quer dizer que ela é aplicada a usuários do sistema operacional. Os valores de configuração global são armazenados no arquivo localizado no diretório base do usuário. ~ /.gitconfig em sistemas Unix e C:\<username>\.gitconfig no Windows |
| --system | A configuração de nível do sistema é aplicada em toda a máquina. Ela abrange todos os usuários do sistema operacional e todos os repositórios. O arquivo da configuração de nível do sistema está no arquivo gitconfig localizado fora do caminho raiz do sistema. $(prefix)/etc/gitconfig nos sistemas Unix. No Windows, esse arquivo pode ser encontrado em C:\Program Files\Git\etc no Windows Vista e versões mais recentes. |

Portanto, a ordem de prioridade para os níveis de configuração é: local, global, sistema. Quando o Git procurar valores de configuração, ele vai começar no nível local e ir até o nível de sistema. A seguir estão alguns exemplos de comandos utilizados:

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git config --global user.name "Your Name" | Adicione seu nome à configuração global |
| git config --global user.email "Your email" | Adicione seu email à configuração global |
| git config --list | Exibir todas as configurações do git |
| git config user.name | Exibe o valor de configuração user.name |
| git config user.email | Exibe o valor da configuração user.email |
| git config --local --list | Lista a configuração local do git |
| git config --global --list | Lista a configuração global do git |
| git config --system --list | Lista a configuração do sistema git |
| git config --local log.date relative | Salve o formato de data relativo na configuração local do Git |
| git config --local --edit | Edita a configuração local do Git |
| git config --global --edit | Edita a configuração global do Git |
| git config --system --edit | Edita a configuração do sistema Git |
| git -c core.editor=echo config --local --edit | Imprime o nome do arquivo de configuração git |
| git -c core.editor=nano config --local --edit | Edite o arquivo de configuração git local usando o nano |
| git config core.excludesfile | Imprima o valor das configurações do git core.excludesfile |
| git flow | um comando git que fica disponível após a instalação do gitflow |

**Como gravar valores**

Para ampliar o conhecimento sobre git config, a gente vai analisar o exemplo no qual o seguinte valor foi gravado:

git config --global user.name "shyoutarou"

git config --global user.email "shyoutarou@gmail.com"

Nesse exemplo, o valor"Rish Rick" para user.name e o valor "shyoutarou@gmail.com" são definidos como o nome e e-mail. Ele usa o sinalizador --global, então esse valor é definido para o usuário atual do sistema operacional. Se você deseja alterar essa configuração global com um nome de usuário ou endereço de email diferente, mas apenas para um projeto específico, você pode executar os mesmos comandos, mas sem a opção --global.

**Git config editor – core.editor**

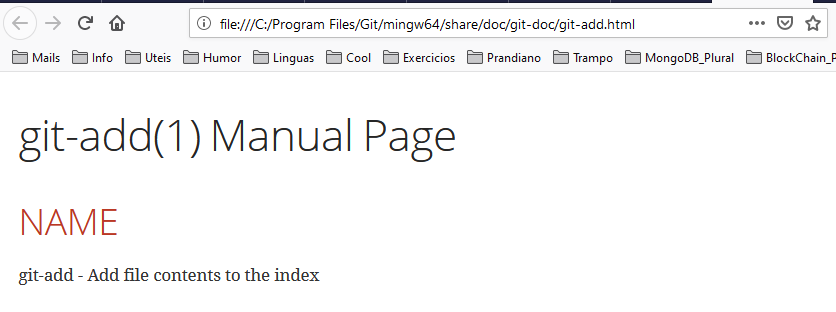
Muitos **comandos** Git iniciam o editor de texto para solicitar mais entradas. Um dos casos de uso mais comuns para git config é configurar qual editor o Git deve usar. A tabela de editores populares e comandos git config correspondentes é apresentada abaixo:

|  |  |
| --- | --- |
| **Editor** | **Comando** |
| Atom | git config --global core.editor "atom --wait"~ |
| Emacs | git config --global core.editor "emacs"~ |
| Nano | git config --global core.editor "nano -w"~ |
| Vim | git config --global core.editor "vim"~ |
| Sublime Text (Mac) | git config --global core.editor "subl -n -w"~ |
| Sublime Text (Win, instalação 32-bit) | git config --global core.editor "'c:/program files (x86)/sublime text 3/sublimetext.exe' -w"~ |
| Sublime Text (Win, instalação 64 bit) | git config --global core.editor "'c:/program files/sublime text 3/sublimetext.exe' -w"~ |
| Textmate | git config --global core.editor "mate -w"~ |
| [Visual Studio Code](https://code.visualstudio.com/) (VS Code) | git config --global core.editor "code --wait" |

**git help**

Para obter ajuda sobre um comando Git específico, como o comando "git add", esta sintaxe pode ser:

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git help help | Peça ao git para ajudar sobre seu sistema de ajuda |
| git help -a | Imprima todos os comandos disponíveis do git |
| git help -g | Imprima todos os guias disponíveis do git |
| git help glossary | Exibir o glossário git |
| git grep string | encontre todos os arquivos com o string fornecido |



**Comandos LESS**

Para mover a tela quando a quantidade de informações é muito grande (Ex.: comando help ou log), utilize o guia abaixo de comandos LESS:

|  |  |  |
| --- | --- | --- |
| **Comando** | **No** | **Descrição** |
| e ^E j ^N CR | \* | Avançar uma linha (ou N linhas) |
| y ^Y k ^K ^P | \* | Retrocede uma linha (ou N linhas) |
| f ^F ^V SPACE | \* | Avança uma janela (ou N linhas) |
| b ^B ESC-v | \* | Retrocede uma janela (ou N linhas) |
| z | \* | Avance uma janela (e defina a janela como N) |
| w | \* | Retroceda uma janela (e defina a janela como N) |
| ESC-SPACE | \* | Encaminha uma janela, mas não para no final do arquivo. |
| d ^D | \* | Avance uma meia janela (e defina meia janela como N). |
| u ^U | \* | Retroceda uma meia janela (e defina meia janela como N). |
| ESC-) RightArrow | \* | Meia largura da tela à direita (ou posições N) |
| ESC-( LeftArrow | \* | Esquerda e meia largura da tela (ou posições N) |
| ESC-} ^RightArrow |  | Direita para a última coluna exibida |
| ESC-{ ^LeftArrow |  | Esquerda para a primeira coluna |
| F |  | Avante para sempre; como "tail -f" |
| ESC-F |  | Como F, mas para quando o padrão de pesquisa é encontrado. |
| r ^R ^L |  | Repita a tela |
| R |  | Repita a tela, descartando a entrada em buffer. |
| Q |  | Sai da tela |

* Os comandos marcados com \* podem ser precedidos por um número, N.
* As notas entre parênteses indicam o comportamento se N for fornecido. Uma chave precedida por um sinal de intercalação indica a tecla Ctrl; assim ^K é ctrl-K.

**Comandos Plumbing e Porcelain**

O Git era inicialmente um kit de ferramentas para um sistema de controle de versão e, por isso, possui vários subcomandos que executam trabalhos de baixo nível e foram projetados para serem encadeados no estilo UNIX ou chamados de scripts. Existem dois tipos de comandos no Git:

1. **Plumbing:** comandos que dão acesso ao funcionamento interno do Git. Muitos desses comandos não devem ser usados ​​manualmente na linha de comando, mas como blocos de construção de novas ferramentas e scripts personalizados.
2. **Porcelain:** comandos mais amigáveis (checkout, branch, remote) ​​que tornam o Git um VCS fácil de usar.

**Conheça SHA-1**

Em sua essência, o Git é apenas um mapa, uma estrutura simples que mapeia chaves para valores, e essa estrutura é persistente, está armazenada no seu disco. Os valores são apenas sequências de bytes. Por exemplo, o conteúdo de um arquivo de texto ou mesmo um arquivo binário. Qualquer sequência de bytes pode ser um valor. Você pode atribuir um valor ao Git e ele calculará uma chave para ele, um hash. O Git calcula hashes com o algoritmo SHA-1, conhecido de maneira informal também como "Shawn". A seguir a descrição de alguns comandos Git que podem ser utilizados:

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| echo "Apple Pie" | git hash-object --stdin | Imprime e gera um SHA-1 da string. Se adicionar o parâmetro –w é gravado a string dentro do diretório .git/object |

Cada parte do conteúdo possui seu próprio SHA ‑ 1. Por exemplo, vamos pegar um conteúdo, a sequência "Apple Pie". Se você pedir ao Git para gerar um SHA-1 a partir dessa string, você obterá esse hash.

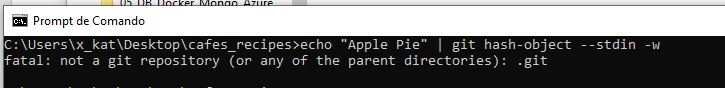
**Apple Pie = bb3918d5053fea31fc9a58fae1e5bdeabe3ec647**

Existe apenas um hash para essa sequência. Os SHA-1s são 20 bytes no formato hexadecimal, portanto, são uma sequência de 40 dígitos hexadecimais. Essa será a chave do Git para armazenar esse conteúdo no mapa. Também podemos calcular o SHA-1 na linha de comando com o comando git hash-object. Primeiro, devemos usar o comando echo para gerar esse conteúdo e canalizar o resultado em um objeto hash. Se digitarmos o comando diretamente o Git pensará que "Apple Pie" é o nome de um arquivo e emitirá erro.



Se você alterar alguma coisa no conteúdo, uma única letra, você obterá um SHA-1 completamente diferente. Todo objeto em um repositório Git tem um SHA1. Se você colocar a sequência "Apple Pie" no arquivo e armazená-lo no Git, o SHA-1 que acabamos de gerar identificará o arquivo. Os diretórios também têm seu próprio SHA-1, assim como confirmações e assim por diante. Para todos os fins práticos, os SHA1-s são únicos. Não apenas o único em seu projeto, você pode pensar neles como se fossem únicos no universo. Você pode colocar todos os dados que você escreverá em sua vida no mesmo repositório Git, e o Git atribuirá um SHA-1 diferente para cada versão de cada arquivo em cada pasta.

Se quiser que o conteúdo "Apple Pie" seja persistente, posso adicionar o argumento -w a este comando. -W significa gravação. Então agora, além de gerar o hash, o Git também salvará esse conteúdo em seu repositório. No entanto, agora ainda não temos um repositório, se eu tentar esse comentário imediatamente, o Git reclama.



**Inicializar um novo repositório Git**

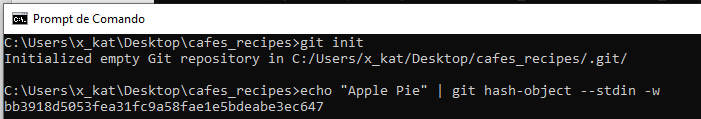
Com o Git instalado, é hora de adicionar o Git ao seu projeto. Isso é feito inicializando um novo repositório git, que são um armazenamento virtual do seu projeto. Em outras palavras, cada projeto no Git é referenciado como repositório para inicializar um novo projeto ou um projeto que já existe. Primeiro você precisa navegar até essa pasta do projeto usando o comando cd.

C:\Users\x\_kat\Desktop\cafes\_recipes

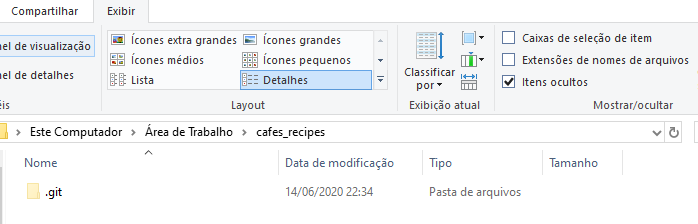
A seguir a descrição de alguns comandos Git que podem ser utilizados:

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git init | Inicialize um repositório git no repositório atual |
| git ls-files | Listar os arquivos no repositório |
| start PATH | Abre o Windows Explorer no caminho espicificado |

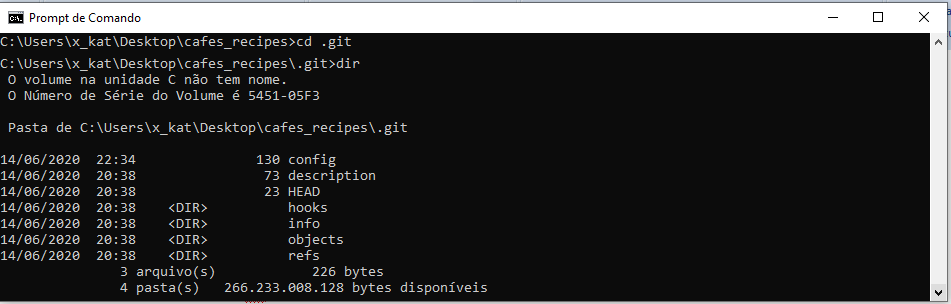
Agora que estamos na pasta, a inicialização de repositórios Git é feita digitando o comando Git init. Depois de executado, você verá uma mensagem de resposta que diz inicializada.



Quando você executa o git init em um diretório novo ou existente, o Git cria o diretório .git, que é onde quase tudo o que o Git armazena e manipula está localizado. Se você deseja fazer backup ou clonar seu repositório, copiar esse diretório único em outro lugar oferece quase tudo o que você precisa. O subdiretório .git que marcou essa pasta do projeto e todo o seu conteúdo para ser controlado por versão; esse subdiretório .git está oculto por padrão.



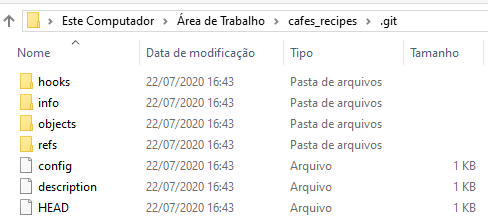
Mas você pode entrar nessa pasta digitando “cd .git” e, em seguida, você pode usar o comando dir para fornecer uma lista dos arquivos nessa pasta.



O diretório .git contém todos os metadados importantes para obter esses novos repositórios. Para abrir o diretório no Windows Explorer, digite:

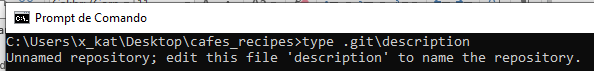
**start .git**



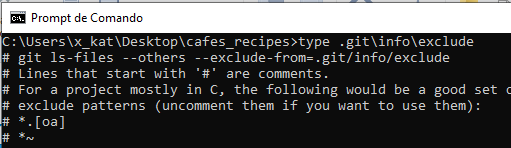


Você pode ver outros arquivos lá, mas este é um repositório git init novo, é o que você vê por padrão.

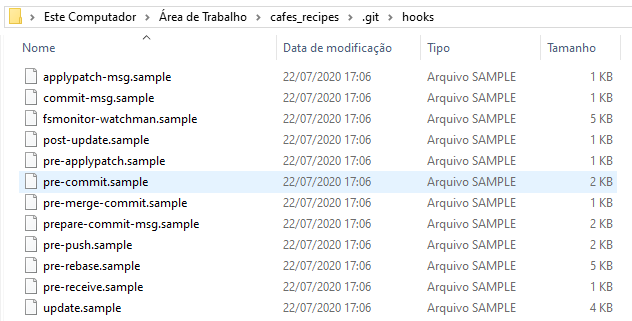
|  |  |
| --- | --- |
| **Entrada** | **Descrição** |
| arquivo description | é usado apenas pelo programa GitWeb |



|  |  |
| --- | --- |
| **Entrada** | **Descrição** |
| arquivo config | opções de configuração específicas do projeto |
| diretório info | mantém um arquivo de exclusão global para padrões ignorados que você não deseja rastrear em um arquivo .gitignore |



|  |  |
| --- | --- |
| **Entrada** | **Descrição** |
| diretório hooks | contém os scripts hook do lado do cliente ou do servidor |

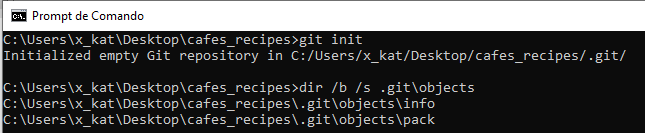


Isso deixa quatro partes principais do Git que iremos ver com mais detalhes:

|  |  |
| --- | --- |
| **Entrada** | **Descrição** |
| diretórios objects | armazena todo o conteúdo do seu banco de dados (blob, tree, commits e tags) |
| diretórios refs | armazena ponteiros em objetos commit nesses dados (ramificações) |
| arquivo HEAD | aponta para a ramificação que você fez check-out |
| arquivos de índice | armazena sua área de armazenamento temporário em formação |

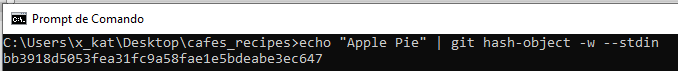
**Objetos Git**

Git é um sistema de arquivos endereçável por conteúdo, isso significa que, no centro do Git, há um simples armazenamento de dados de valores-chave. Você pode inserir qualquer tipo de conteúdo e fornecerá uma chave que você pode usar para recuperar o conteúdo novamente a qualquer momento. Para demonstrar, você pode usar o comando de plumbing hash-object, que pega alguns dados, armazena-os no diretório .git e devolve a chave em que os dados estão armazenados. Primeiro, você inicializa um novo repositório Git e verifica se não há nada no diretório de objetos:



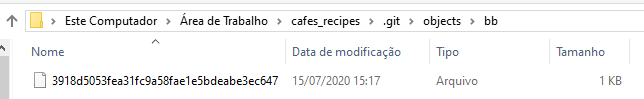
O Git inicializou o diretório de objetos e criou subdiretórios de pacote e informações, mas não há arquivos regulares. E se executarmos novamente o comando git hash-object para a string "Apple Pie" com –w, obteremos o hash e também salvaremos o conteúdo no diretório, objetos dentro do diretório .git.

**echo "Apple Pie" | git hash-object –w --stdin**



O -w diz ao objeto hash para armazenar o objeto; caso contrário, o comando simplesmente informa qual seria a chave. --stdin diz ao comando para ler o conteúdo do stdin; se você não especificar isso, o hash-object espera um caminho de arquivo no final. A saída do comando é um hash de soma de verificação de 40 caracteres. Esse é o hash SHA-1 - uma soma de verificação do conteúdo que você está armazenando e um cabeçalho. Agora você pode ver como o Git armazenou seus dados:





No subdiretório bb, que são os dois primeiros dígitos hexadecimais do SHA-1 da string que acabamos de salvar. E se olharmos dentro de bb, há um arquivo aqui e o nome desse arquivo são os 38 caracteres restantes do SHA-1. O Git usa esse esquema para organizar o conteúdo e distribuí-lo por vários diretórios. É apenas um truque para evitar acumular todo o conteúdo em um único diretório desordenado. Além disso dependendo do sistemas de arquivos o número máximo de arquivos em uma única pasta varia e, ao criar subdiretórios, ajuda a balancear a carga de arquivos por diretório.

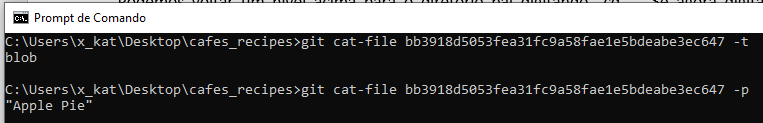
|  |  |
| --- | --- |
| **Sistema** | **Máximo de arquivos** |
| FAT/FAT16 | 512 |
| FAT32 | 65,534 |
| NTFS | 4,294,967,295 |
| exFAT | 2,796,202 |

**Blob de dados**

Nossa sequência original, "Apple Pie", está dentro desse arquivo. Isso é o que o Git chama de blob de dados. Um blob é uma parte genérica do conteúdo, onde o Git adicionou um cabeçalho pequeno e compactou o conteúdo para economizar espaço. Portanto, não podemos simplesmente abrir o arquivo e lê-lo. Para examinar o conteúdo podemos usar outro comando chamado git cat-file.

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git cat‑file SHA1 | Pega o SHA‑1 de um objeto e um argumento. Se o argumento for -t, retorna o tipo de arquivo, se o argumento for –p realiza impressão elegante. |
| echo 'version 1' > test.txt | Gera/Altera arquivo com conteúdo 'version 1' |
| git hash-object -w text.txt | Gera SHA‑1 de arquivo no banco de dados |

O comando git cat-file pega o SHA‑1 de um objeto e um argumento. Se o executarmos com o argumento -t, significa type, que no caso aqui é um blob. E se o executarmos novamente com –p para uma impressão elegante, o Git descompacta o objeto, remove o cabeçalho e imprime o conteúdo real do blob.

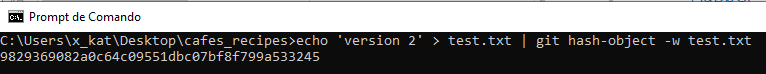


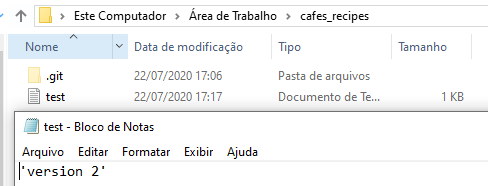
Até agora, vimos que o Git é capaz de pegar qualquer parte do conteúdo, gerar a chave para ele, um SHA-1, e depois persistir o conteúdo no repositório como um blob, um mapa persistente. Agora, você pode adicionar conteúdo ao Git e retirá-lo novamente. Você também pode fazer isso com o conteúdo dos arquivos. Por exemplo, você pode fazer algum controle de versão simples em um arquivo. Primeiro, crie um novo arquivo e salve seu conteúdo no seu banco de dados:

**echo 'version 1' > test.txt**

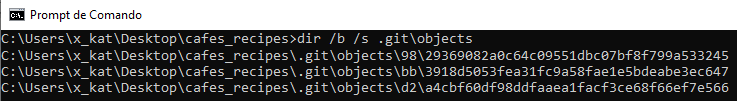


Em seguida, escreva um novo conteúdo no arquivo e salve-o novamente:



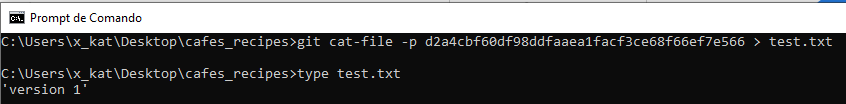


Seu banco de dados contém as duas novas versões do arquivo, bem como o primeiro conteúdo armazenado lá:



Agora você pode reverter o arquivo para a primeira versão

**git cat-file -p d2a4cbf60df98ddfaaea1facf3ce68f66ef7e566 > test.txt**



ou a segunda versão:

**git cat-file -p 9829369082a0c64c09551dbc07bf8f799a533245 > test.txt**



Mas lembrar a chave SHA-1 para cada versão do seu arquivo não é prático; além disso, você não está armazenando o nome do arquivo no seu sistema, apenas o conteúdo. Mas é apenas uma demonstração de que poderíamos até apagar o arquivo que conseguiríamos recuperar o conteúdo no arquivo a partir do versionamento de repositório é assim que comandos como o git check-out funcionam.

Outra coisa é que se tentar acessar diretamente o conteúdo de qualquer objeto SHA-1:



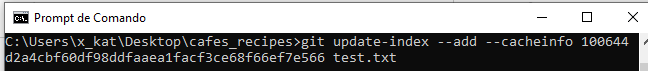
Isso ocorre pois o Git compacta todos objetos com o zlib, o que você pode fazer no Ruby com a biblioteca zlib e executando Zlib :: Deflate.deflate () no conteúdo.

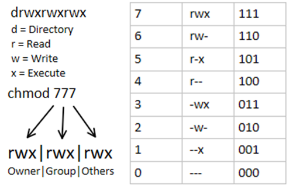
**Objetos Tree**

O próximo tipo que veremos é a árvore, que resolve o problema de armazenar o nome do arquivo e também permite que você armazene um grupo de arquivos juntos. O Git armazena conteúdo de maneira semelhante a um sistema de arquivos UNIX, mas um pouco simplificado. Todo o conteúdo é armazenado como objetos de árvore e blob, com árvores correspondentes às entradas do diretório UNIX e blobs correspondendo mais ou menos a inodes ou conteúdo de arquivo. Um único objeto de árvore contém uma ou mais entradas de árvore, cada uma contendo um ponteiro SHA-1 para um blob ou subárvore com seu modo, tipo e nome de arquivo associados.

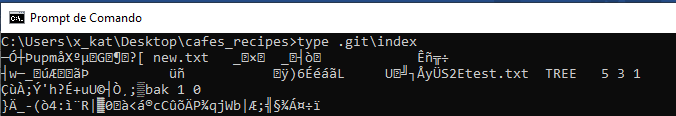
Você pode criar facilmente sua própria árvore. O Git normalmente cria uma árvore, tomando o estado da sua área de preparação ou índice e gravando uma série de objetos da árvore. Então, para criar um objeto de árvore, você primeiro precisa configurar um índice, preparando alguns arquivos. Para criar um índice com uma única entrada - a primeira versão do seu arquivo test.txt - você pode usar o comando plumbing update-index. Você usa este comando para adicionar artificialmente a versão anterior do arquivo test.txt a uma nova área de preparação. Você deve passar a opção --add porque o arquivo ainda não existe na sua área de preparação (você ainda nem tem uma área de preparação configurada) e --cacheinfo porque o arquivo que você está adicionando não está no seu diretório, mas está no seu banco de dados. Em seguida, você especifica o modo, SHA-1 e nome do arquivo:

**C:\Users\x\_kat\Desktop\cafes\_recipes>git update-index --add --cacheinfo 100644 d2a4cbf60df98ddfaaea1facf3ce68f66ef7e566 test.txt**



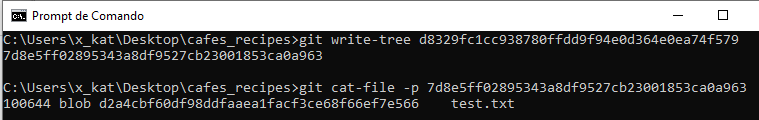
Nesse caso, você está especificando um modo de 100644, o que significa que é um arquivo normal. Outras opções são 100755, o que significa que é um arquivo executável; e 120000, que especifica um link simbólico. O modo é retirado dos modos normais do UNIX, mas é muito menos flexível - esses três modos são os únicos válidos para arquivos (blobs) no Git (embora outros modos sejam usados para diretórios e submódulos). No Linux esta coinvecção aparece no comando chmod(CHange MODe) onde 644 é um arquivo normal, 755 arquivo executável ou sendo mais exato o primeiro número é um modo para o usuário dono do arquivo, o segundo número é um modo para o grupo, e o terceiro é o modo para qualquer outros usuário. 7 é ler/ escrever/executar cinco é ler/executar quatro é só leitura e assim por diante conforme tabela abaixo:

Ao rodar esse comando ele cria o arquivo .git/index que não existia na inicialização do projeto, ele tem agora o conteúdo:

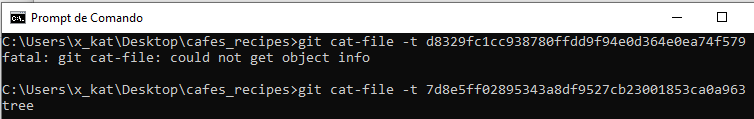


Agora, você pode usar o comando write-tree para gravar a área intermediária em um objeto em árvore. A opção no -w é necessária. A chamada write-tree cria automaticamente um objeto em árvore a partir do estado do índice se essa árvore ainda não existir:

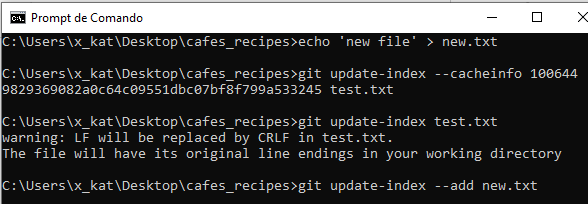
**git write-tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579**



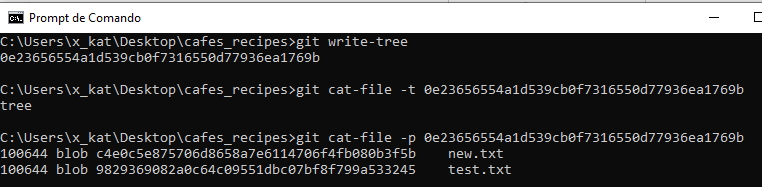
Você também pode verificar se este é um objeto de árvore



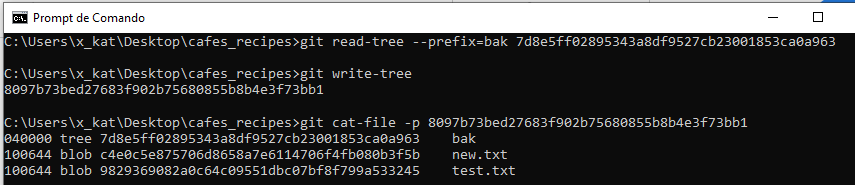
Agora você criará uma nova árvore com a segunda versão do test.txt e também um novo arquivo:



Sua área de preparação agora tem a nova versão do test.txt, bem como o novo arquivo new.txt. Escreva essa árvore (registrando o estado da área intermediária ou o índice em um objeto em árvore) e veja como ela se parece:



Observe que essa árvore possui as duas entradas de arquivo e também que o test.txt SHA-1 é o "versão 2" SHA-1 da versão anterior (1f7a7a). Apenas por diversão, você adicionará a primeira árvore como um subdiretório a essa. Você pode ler árvores na sua área de preparação chamando a árvore de leitura. Nesse caso, você pode ler uma árvore existente em sua área intermediária como uma subárvore usando a opção --prefix para ler a árvore:



Se você criou um diretório de trabalho a partir da nova árvore que você acabou de escrever, obteria os dois arquivos no nível superior do diretório de trabalho e um subdiretório chamado bak, que continha a primeira versão do arquivo test.txt. Você pode pensar nos dados que o Git contém para essas estruturas como este:

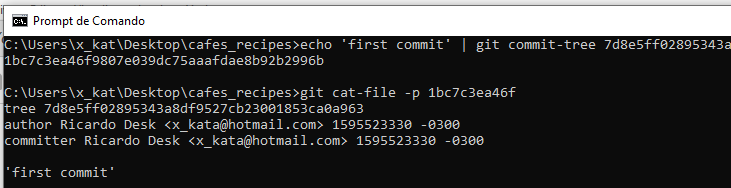


**Objetos Commit**

Você tem três árvores que especificam os diferentes instantâneos do seu projeto que deseja rastrear, mas o problema anterior permanece: você deve lembrar dos três valores do SHA-1 para recuperar os instantâneos. Você também não tem informações sobre quem salvou os instantâneos, quando eles foram salvos ou por que eles foram salvos. Essas são as informações básicas que o objeto de confirmação armazena para você.

Para criar um objeto de consolidação, chame **commit-tree** e especifique uma única árvore SHA-1 e quais objetos de consolidação, se houver, a precederam diretamente. Comece com a primeira árvore que você escreveu:

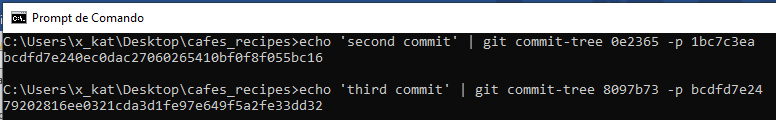
**echo 'first commit' | git commit-tree 7d8e5ff0**



Agora você pode olhar para o seu novo objeto de confirmação com o arquivo cat-file que tem um valor de hash diferente devido a diferentes horários de criação e dados do autor.

O formato para um objeto de confirmação é simples: especifica a árvore de nível superior para a captura instantânea do projeto naquele momento; as informações do autor/responsável (que usa as definições de configuração user.name e user.email e um carimbo de data/hora); uma linha em branco e, em seguida, a mensagem de confirmação.

Em seguida, você escreverá os outros dois objetos de confirmação, cada um referenciando o commit que veio diretamente antes dele:

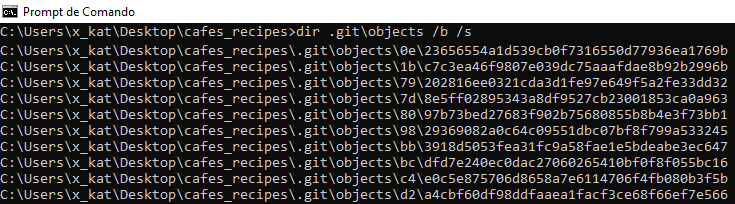


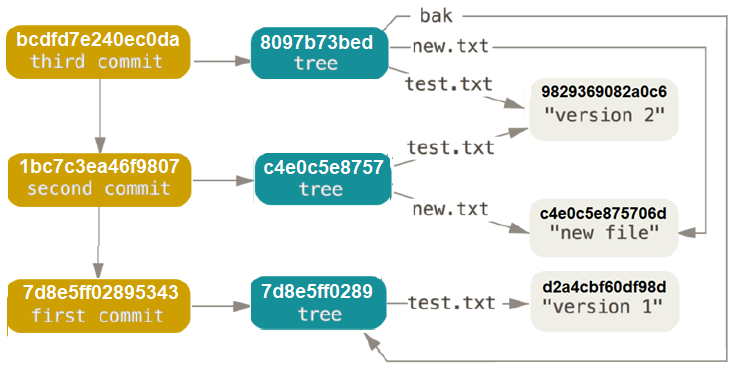
Cada um dos três objetos de confirmação aponta para uma das três árvores de captura instantânea que você criou. Curiosamente, você tem um histórico real do Git agora que pode ver com o comando git log, se o executar no último commit SHA-1:



Você acabou de executar as operações de baixo nível para criar um histórico do Git sem usar nenhum dos comandos de porcelana. Isso é essencialmente o que o Git faz quando você executa os comandos git add e git commit - ele armazena blobs para os arquivos que foram alterados, atualiza o índice, grava árvores e grava objetos de commit que fazem referência às árvores de nível superior e aos commit que veio imediatamente antes deles. Esses três objetos principais do Git - o blob, a árvore e o commit - são inicialmente armazenados como arquivos separados no diretório .git/objects. Aqui estão todos os objetos no diretório de exemplo agora, comentados com o que eles armazenam:

**dir .git\objects /b /s**

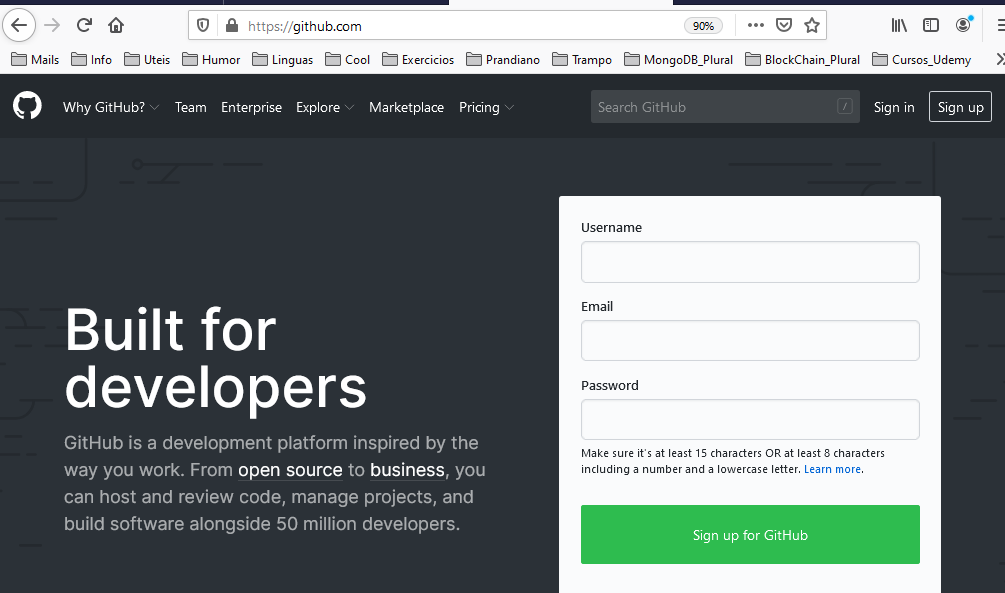




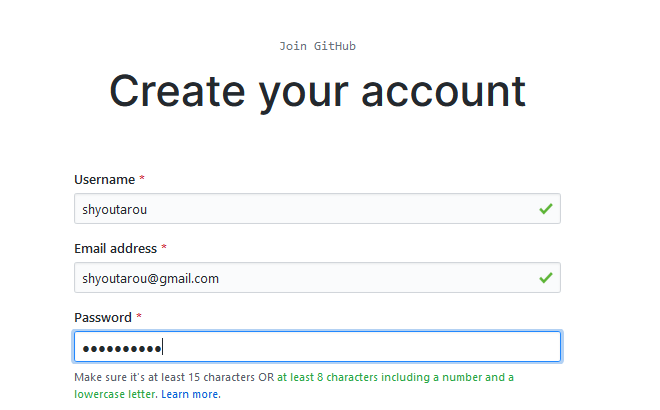
**PROVEDOR DE HOSPEDAGEM DE CÓDIGO**

Os arquivos agora estão sendo controlados por versão, mas tudo ainda está localmente no computador, não sendo possível o acesso remoto dos arquivos. Para resolver esse problema, precisamos de um provedor de hospedagem de código e os mais populares são o GitHub, o GitLab e o BitBucket. Além de ser um serviço de hospedagem de repositórios Git, os provedores de hospedagem de código oferecem recursos adicionais para ajudar a melhorar seus repositórios Git em colaboração e facilitar as contribuições entre outros desenvolvedores em todo o mundo. O recurso mais importante que esses provedores oferecem é a capacidade de integrar colaborações de maneira integrada aos seus repositórios Git.

Para criar uma conta GitHub basta ir para o seguinte endereço: github.com

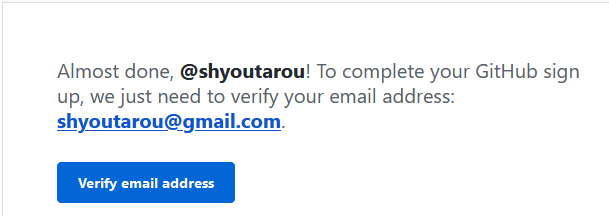


Vamos clicar em Sign Up e preencher um nome de usuário, endereço de email e senha **shyoutarou2020**.

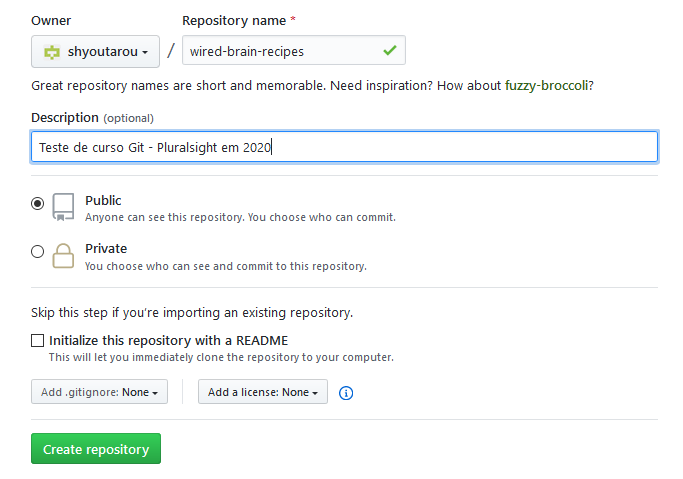


Há a etapa de escolha do tipo de plano pessoal que pode ser um plano gratuito para criar repositórios públicos ilimitados ou um plano pago para criar repositórios privados ilimitados. Depois, precisamos verificar o seu endereço de e-mail que nos levará de volta para página do GitHub e fim.

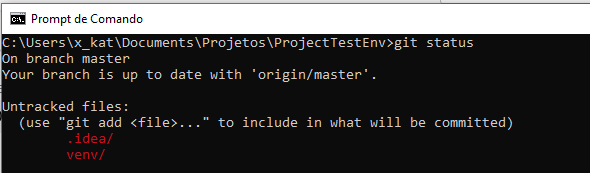
**https://github.com/shyoutarou**



Na tela para enviar um repositório Git para um provedor é necessário informar um nome válido, no qual, informamos wired-brain-recipes. Em seguida, podemos adicionar uma descrição opcional e temos a opção de tornar esses repositórios públicos ou privados.

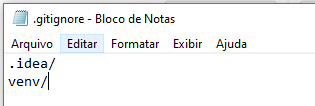


Há uma opção para inicializar esses repositórios com um arquivo leia-me, que é uma maneira rápida e simples de os colaboradores aprenderem mais sobre seus repositórios. E as outras duas opções são para adicionar um Git ignore, que é uma maneira de adicionar arquivos que você não deseja que a versão seja controlada pelo Git; e o outro é adicionar uma licença aos seus repositórios.



Caso tenha optado por não criar um arquivo gitignore, mas deseje cria-lo manualmente, siga a seguintes etapas:

1. Abra o bloco de notas.
2. Adicione o conteúdo do seu arquivo gitignore.
3. Clique em "Salvar como" e selecione "todos os arquivos".
4. Salve como .gitignore.





**Como .gitignore funciona**

Quando você faz confirmações em um repositório git, escolhe quais arquivos preparar e confirmar usando git add FILENAME e, em seguida, git commit. Mas e se houver alguns arquivos que você nunca deseja confirmar? É muito fácil comprometê-los acidentalmente (especialmente se você usar o git add. Para preparar todos os arquivos no diretório atual). É aí que um arquivo .gitignore é útil. Ele permite ao Git saber que deve ignorar certos arquivos e não rastreá-los. Os arquivos você deve ignorar geralmente são:

* Arquivos de log
* Arquivos com chaves / segredos de API, credenciais ou informações confidenciais
* Arquivos de sistema inúteis, como .DS\_Store no macOS
* Arquivos gerados como pastas dist
* Dependências que podem ser baixadas de um gerenciador de pacotes
* E pode haver outros motivos (talvez você faça pequenos arquivos todo.md)

Um arquivo .gitignore é um arquivo de texto sem formatação, em que cada linha contém um padrão para os arquivos / diretórios ignorarem. Geralmente, isso é colocado na pasta raiz do repositório, e é isso que eu recomendo. No entanto, você pode colocá-lo em qualquer pasta do repositório e também pode ter vários arquivos .gitignore. Os padrões nos arquivos são relativos ao local desse arquivo .gitignore.

Como o arquivo .gitignore é verificado no repositório, há algumas opções se você deseja ignorar alguns arquivos sem adicioná-lo às regras .gitignore do repositório. Por exemplo, você pode ter alguns arquivos especiais com os quais está trabalhando em um projeto específico ou pode usar um editor diferente dos seus colegas de equipe e sempre deseja ignorar esses tipos de arquivos. Se houver alguns arquivos que você deseja ignorar apenas para este repositório, você pode colocá-los em

**.git/ info/exclude.**

Se houver alguns arquivos que você deseja ignorar em todos os repositórios do seu computador, você pode colocá-los em um arquivo .gitignore global. Primeiro, você deve adicionar uma configuração ao Git com este comando:

**git config --global core.excludesFile ~/.gitignore**

Em seguida, você pode adicionar quaisquer regras globais ao ~/.gitignore.

O Git não ignorará o arquivo se você já o tiver confirmado. Você precisará deixar de rastrear o arquivo primeiro e, em seguida, ele começará a ignorá-lo. Você pode deixar de rastrear o arquivo com este comando:

**git rm --cached FILENAME**

Se você estiver com problemas, poderá descobrir por que certos arquivos estão sendo ignorados usando o comando git check-ignore com a opção detalhada.

**git check-ignore -v example.log**

A saída será mais ou menos assim:

**.gitignore: 1: \*. log example.log**

Neste exemplo, o arquivo .gitignore na raiz do projeto está fazendo com que o exemplo.log seja ignorado, e o padrão que está causando o fato de ser ignorado é \* .log na primeira linha.

O padrão mais fácil é um nome de arquivo literal, no exemplo abaixo ignorará qualquer arquivo chamado .DS\_Store, que é um arquivo comum no macOS.

**.DS\_Store**

Você pode ignorar diretórios inteiros, apenas incluindo seus caminhos e colocando um / no final. Se você não colocar a barra no final, ela corresponderá aos arquivos e diretórios com esse nome.

**node\_modules/**

**Histórico/**

O \* corresponde a 0 ou mais caracteres (exceto o /). Por exemplo, \* .log corresponde a qualquer arquivo que termine com a extensão .log. Outro exemplo é \*~, que corresponde a qualquer arquivo que termina com ~, como index.html~. Você também pode usar ?, Que corresponde a qualquer caractere, exceto o /.

Você pode usar um prefixo de! negar um arquivo que seria ignorado. No exemplo abaixo, example.log não é ignorado, mesmo que todos os outros arquivos que terminem com .log sejam ignorados.

**\*.log**

**!example.log**

Mas lembre-se, você não pode negar um arquivo dentro de um diretório ignorado. No exemplo abaixo, devido a razões de desempenho, o git ainda ignorará logs/example.log porque todo o diretório de logs é ignorado.

**logs/**

**!logs/example.log**

\*\* pode ser usado para corresponder a qualquer número de diretórios.

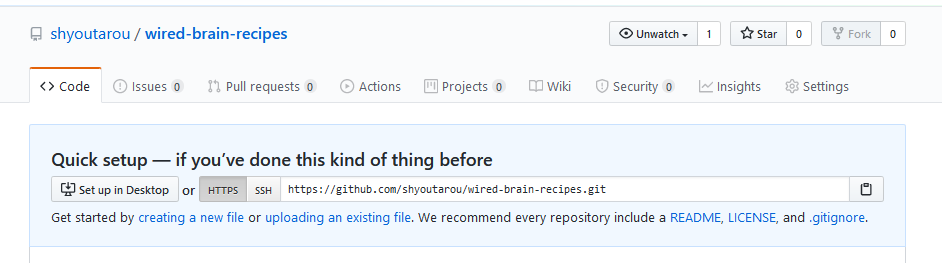
* \*\*/logs corresponde a todos os arquivos ou diretórios denominados logs (o mesmo que os logs padrão)
* \*\*/logs/\*. log corresponde a todos os arquivos que terminam com .log em um diretório de logs
* logs/\*\* corresponde a todos os arquivos em um diretório de logs.
* logs/\*\*/\*. log corresponde a todos os arquivos que terminam com .log no diretório logs e em qualquer um de seus subdiretórios

Quaisquer linhas que começam com # são comentários:

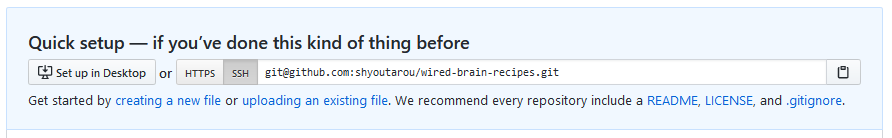
**# macOS Files**

**.DS\_Store**

Ao clicar Create repository, vamos para uma página que fornece alguns cenários diferentes e as etapas sobre como atualizar nossos repositórios.

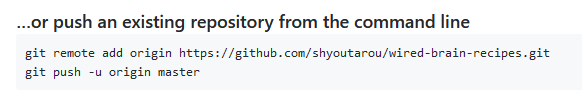


Temos a opção de utilizar URL https ou SSH, que funcionarão em qualquer lugar, mesmo se você estiver atrás de um firewall ou proxy usando https URL para enviar ou receber alterações do repositório o GitHub solicitará seu nome de usuário e endereço de e-mail para verificação.

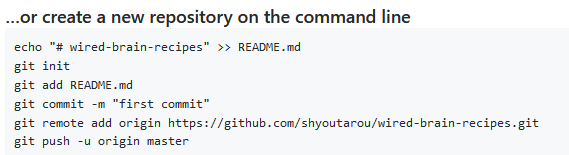


Há também uma opção para usar SSH URL para fornecer acesso aos seus repositórios git através do protocolo SSH Secure. Para usar SSH URL, você precisa primeiro gerar um par de chaves SSH e, em seguida, adicionar essa chave pública à sua conta Git nas configurações de perfil usando SSH URL para enviar ou receber alterações nos seus repositórios. O Github solicitará uma senha e você precisará digitar sua palacra-chave SSH.

Já que já temos um repositório Git, usaremos a segunda opção para enviar repositórios já existentes da linha de comando para o GitHub.



Mas usaremos algumas das etapas no primeiro cenário para criar um arquivo leia-me ao enviar nosso primeiro commit para atualizar nosso repositório remotamente.



**Preparando (Stage) e confirmando (Commit) alterações**

A seguir estão alguns exemplos de comandos utilizados:

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git status | Exibir status do diretório atual, no que se refere ao Git |
| git add FILE | Comece a rastrear FILE no Git; adiciona FILE à área de preparação |
| git add –dry-run | Mostrar o que o git add faria |
| git add . | Adicione todos os novos arquivos ao repositório git |
| git commit -m MSG | Confirme as alterações no repositório git, com uma mensagem entre aspas |
| git commit -a -m MSG | Adiciona os arquivos não preparados e cria um novo objeto de confirmação |
| git commit –allow-empty -m "Initial commit" | Crie um commit sem adicionar nenhum arquivo |
| git count-objects | Retorna o numero de objetos gerados em .git/objects |

Antes de passarmos para a linha de comando, será necessário copiar a URL https, no topo da página:

**https://github.com/shyoutarou/wired-brain-recipes.git**

Para adicionar o arquivo README.md digite o seguinte comando:

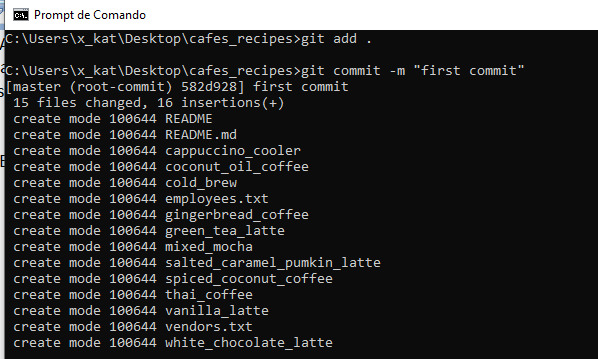
**echo "# wired-brain-recipes" >> README.md**

Agora que adicionamos o arquivo README.md ao nosso projeto, precisamos adicionar esse arquivo à nossa área de preparação para que eles possam ser confirmados e, portanto, rastreados por Git, podemos adicionar todos esses arquivos à nossa área de preparação digitando o comando:

**git add .**

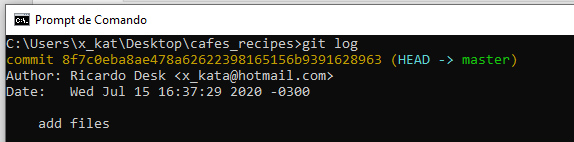
E para confirmarmos as alterações, digitamos o comando commit com o parâmetro –m para adicionarmos uma mensagem:

**git commit –m “first commit”**

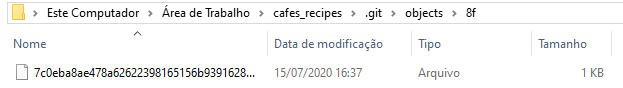


Sempre quando confirmarmos alterações elas resultam em novo SHA-1 de confirmação, como o seguinte:

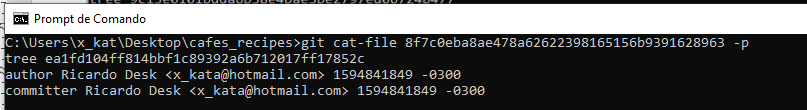
**commit 8f7c0eba8ae478a62622398165156b9391628963**



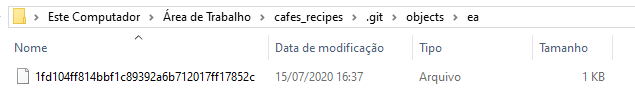
Ao navegarmos para pasta .git/object podemos ver que foi gerado o objeto de confirmação compactada como um blob:



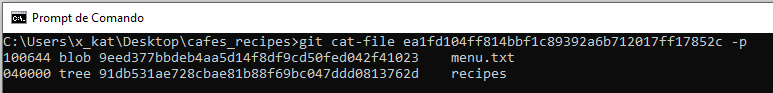
Ao usar o git cat-file para imprimir o conteúdo do commit, podemos observar que ele é um texto simples e muito curto, nada mais. O Git gera esse texto e o armazena da mesma maneira que armazena um blob. Ele gera seu SHA‑1, adiciona uma pequena letra ao texto para dizer que é uma confirmação, compacta o texto e armazena o resultado em um arquivo no banco de dados do objeto. O texto de confirmação contém todos os metadados sobre a confirmação, o nome do autor, o confirmador, data da confirmação e a mensagem e, em seguida, contém algo mais: o SHA-1 de uma árvore. Com exceção primeiro commit, a maioria dos commits são vinculados por um pai (parent).

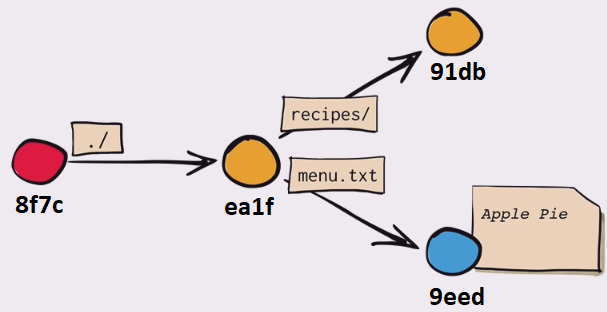


Assim como um blob é o conteúdo de um arquivo armazenado no Git, uma árvore é um diretório armazenado no Git. A confirmação está apontando para o diretório raiz do projeto. É isso que é essa árvore, a raiz do projeto. Se você procurar no banco de dados de objetos, verá um diretório nomeado com os dois primeiros dígitos do hash da árvore e, dentro dele, a árvore, um arquivo nomeado com os dígitos restantes do hash, como de costume.

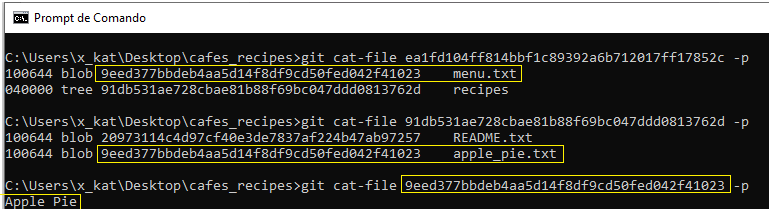


É como uma confirmação, uma parte do conteúdo que é gerada pelo Git e, em seguida, processada e armazenada no banco de dados do objeto. Assim como um commit, uma árvore é um pequeno pedaço de texto, é tudo o que existe e contém uma lista do conteúdo do diretório, uma lista dos SHA-1s na verdade.

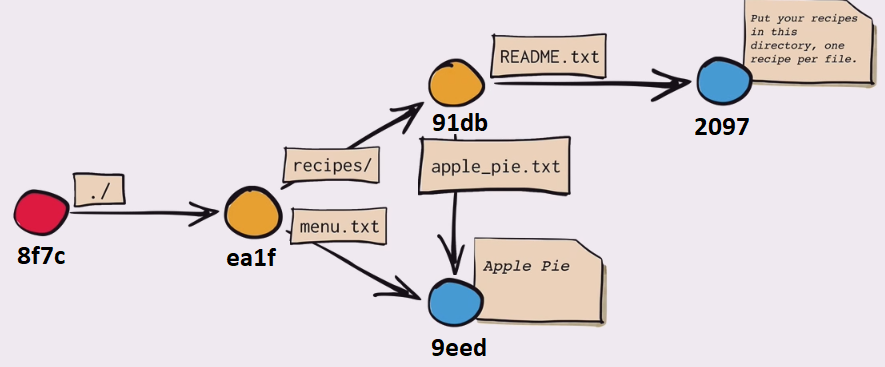




Os 6 digito iniciais são dados adicionais para as permissões de acesso do arquivo e do diretório. O nome do arquivo e as permissões do arquivo não são armazenados no blob, eles são armazenados na árvore que aponta para o blob.



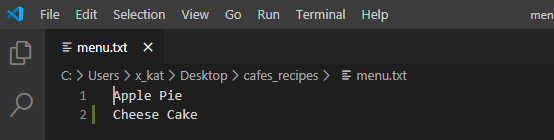
A árvore de receitas está apontando para o blob com o conteúdo de apple\_pie.txt, que é o mesmo conteúdo que o arquivo menu.txt, portanto, na verdade, é o mesmo blob. Como esses dois arquivos têm exatamente o mesmo conteúdo, o Git não criará dois objetos separados para eles, apenas reutilizará o objeto existente que já está no banco de dados. Portanto, um blob não é realmente um arquivo, um blob é apenas o conteúdo de um arquivo.



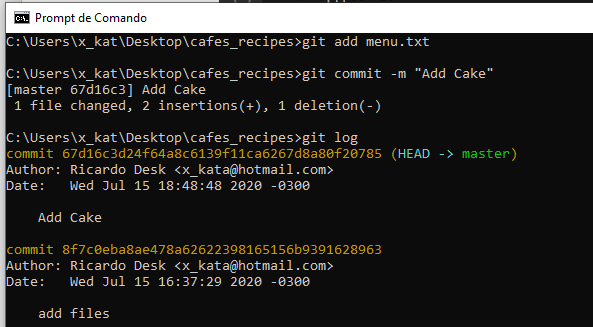
Se você tentar criar exatamente o mesmo projeto e fornece os mesmos comandos que outra pessoa fornece ao Git, verá que obtém exatamente os mesmos SHA-1s para todas as árvores e todos os blobs. O importante a entender aqui é que não há mágica por trás do SHA-1. Se você tem o mesmo conteúdo que outra pessoa, obtém os mesmos hashes. No entanto, o SHA‑1 do commit será diferente, porque você tem dados diferentes no commit, um autor diferente e uma data de commit diferente. Uma confirmação também é apenas uma parte do conteúdo, e sua confirmação tem um conteúdo diferente do meu, então você obterá um hash diferente.

Vamos fazer uma alteração no arquivo menu.txt chamando o editor VS Code assim:

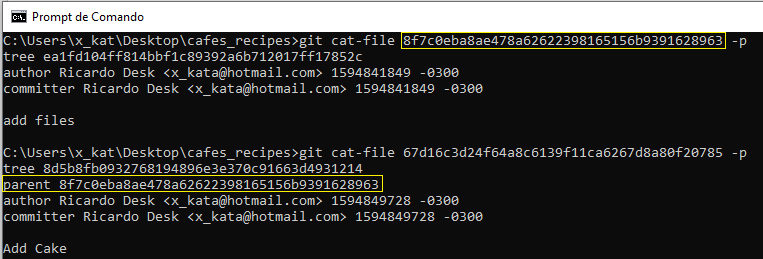
**start code menu.txt**



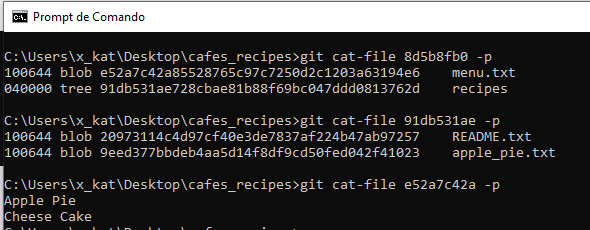
Depois de salvarmos o arquivo, preparamos e confirmamos as alterações:



Analisando o conteúdo dos commits podemos ver que o segundo commit possui o campo parent com o SHA-1 do primeiro commit vinculando os dois.



Se você observar o hash da árvore que este segundo commit está apontando, verá que essa é uma árvore totalmente nova. Não é a mesma árvore que o primeiro commit estava apontando. É como uma raiz diferente. Também podemos usar apenas os primeiros dígitos do SHA‑1 e o Git recuperará automaticamente todo o SHA ‑ 1 do banco de dados, a menos que haja vários SHA‑1s iniciando com esses primeiros dígitos.



Agora, menu.txt é um blob totalmente novo, porque esse arquivo foi alterado, incluindo o Apple Pie e o Cheesecake. No entanto, o conteúdo do diretório de receitas é o mesmo objeto que já tínhamos no banco de dados desde o primeiro commit. Como o conteúdo deste diretório não foi alterado, não há razão para criar um novo objeto. O Git pode apenas usar o objeto que já estava no banco de dados.

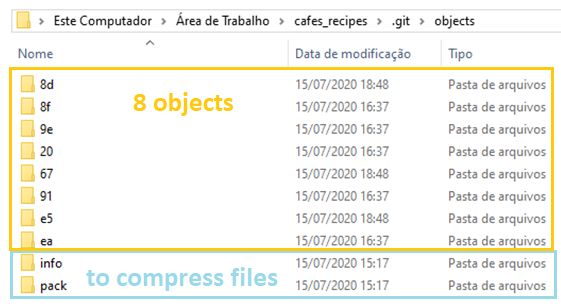
Essa é uma das razões pelas quais o Git é tão eficiente; não armazena as coisas mais de uma vez. Como alteramos um único arquivo, o Git armazenou um novo blob e, no nosso caso, uma nova árvore e uma nova confirmação. Como eles estão apontando para o novo arquivo, eles são alterados. As árvores e os commits são realmente pequenos, então isso ainda é extremamente eficiente.



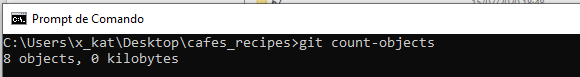
Falando em eficiência, você pode se surpreender ao ver que o Git armazena um novo blob toda vez que você altera um arquivo. E se eu tiver um arquivo enorme e alterar apenas uma única linha? O Git armazenará um novo blob inteiro nesse caso e duplicará o restante do arquivo? Bem, na verdade não. O Git também faz outra camada de otimizações para economizar mais espaço. Por exemplo, à medida que você continua trabalhando e adicionando conteúdo ao repositório, o Git pode decidir armazenar apenas as diferenças entre os dois arquivos ou até compactar vários objetos no mesmo arquivo físico. A propósito, esse tipo de coisa é a razão desses misteriosos diretórios **info** e **pack** no banco de dados.

No entanto, esses são realmente detalhes de implementação. Você pode ignorá-los com segurança. Para entender o modelo Git, basta pensar em cada commit, blob ou árvore como apenas arquivos, arquivos separados, que são divididos em hash e armazenados no banco de dados. No nível dos comandos, é assim que o Git realmente funciona.

Se você contar o número de objetos no diagrama acima, serão duas confirmações, mais 3 árvores e 3 blobs. total de oito objetos. Este é o número atual de objetos no banco de dados de objetos.



O próprio banco de dados está ficando um pouco cheio, então, em vez de contar os arquivos, vamos usar um desses comandos de encanamento raramente usados, o git count-objects. E aí estão oito objetos, e eles ocupam uma quantidade muito pequena de espaço em disco.



Observe o modelo inteiro de um ponto de vista abstrato. Temos os blobs, que contêm dados e existem outras coisas chamadas árvores que contêm blobs e outras árvores, criando uma estrutura recursiva. Os nomes dos blobs e árvores não estão armazenados nos objetos si mesmos mas na árvore que o contém. Portanto, você pode ter o mesmo objeto, digamos, o mesmo blob ou a mesma árvore apontada por diferentes árvores com nomes diferentes. Lembrando muito um sistema de arquivos.

Assim como em um sistema de arquivos, você tem conteúdo, arquivos ou blobs, e contêineres, diretórios ou árvores aninhadas nesse caso, e você pode ter links. O mesmo arquivo ou diretório pode ser acessado de diferentes locais com nomes diferentes. Afinal, como o Git foi escrito por Linus Torvalds', que escreveu o Linux, você pode argumentar que é isso que o Git é um sistema de arquivos de alto nível, construído sobre o sistema de arquivos nativo, como links no Linux ou atalhos no Windows.

**TAGS**

No banco de dados de objetos Git temos quatro tipos de objetos:

* **Blobs (conteúdo arbitrário)**
* **Trees (equivale a diretórios)**
* **Commits**
* **Tags**

Uma tag é como um rótulo para o estado atual do projeto, existem dois tipos de tags no Git:

1. **Tags anotadas:** são as que acompanham uma mensagem. Essa tag contém muitas informações úteis, como a data em que a tag foi criada, quem a criou, uma descrição e assim por diante.
2. **Tags leve não-anotada:** um rótulo simples para meu próprio uso.

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git rev-parse | traduza um nome ou tag de filial para um SHA1 específico |
| git tag tag\_name -m "message" SHA1id | crie uma tag chamada tag\_name, apontando para seu SHA1id |
| git tag | listar todas as tags |
| git show tag\_name | mostra informações sobre a tag denominada tag\_name |
| git tag -a tag\_name -m tag\_message SHA1 | crie uma tag no SHA1 com o nome tag\_name e a mensagem tag\_message |
| git tag -d tagname | remova a tag denominada tagname do repositório local |

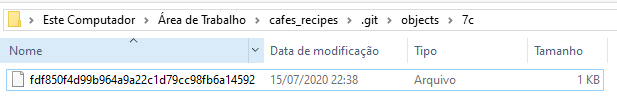
**Tags anotadas**

Para criar uma tag anotada, você pode usar o comando git tag com o argumento –a, e precisa de um nome para a tag e também de algum tipo de mensagem aqui.



É semelhante à criação de uma confirmação e, de fato, uma tag anotada também é um objeto no banco de dados de objetos do Git, como a confirmação.

**7cfdf850f4d99b964a9a22c1d79cc98fb6a14592**



Vamos usar o cat-file para espiar dentro dele. No caso de tags, o arquivo cat pode usar o hash ou o nome da tag.

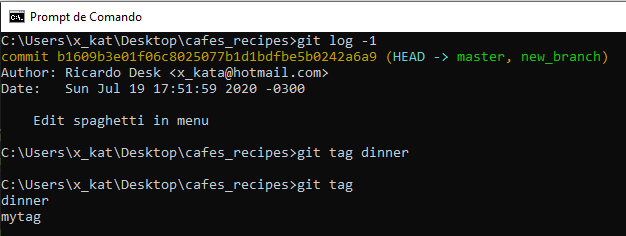


E aqui está a etiqueta. Ele contém metadados, como a mensagem da tag, o nome, o marcador, a data e, o mais importante, um objeto para o qual a tag está apontando. Nesse caso, é um commit.

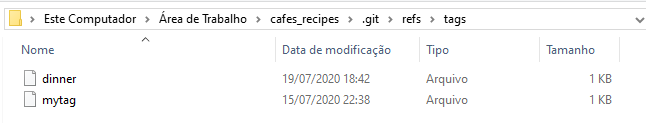
**Tags leve não-anotada**

Esse tipo de tag não tem um nome específico mas as pessoas às vezes as chamam de tags não anotadas ou leves. Para criar um, digamos que queremos marcar o commit mais recente com um tag simples para meu próprio uso chamado dinner. Digitamos então o comando tag ignorando a opção –a:

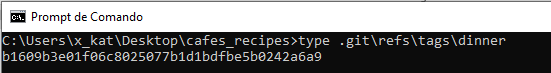
**git tag dinner**



Navegando agora para o diretório .git/refs, existem duas tags, que são dois arquivos simples que contêm o SHA‑1 de um objeto no banco de dados.



Veja, uma tag é uma referência a um objeto, neste caso, uma confirmação, como uma ramificação.



Na verdade, eu poderia transformar essa tag em um ramo apenas movendo-a para o diretório refs/head. Como é uma marca leve, ela contém o SHA‑1 de uma confirmação. Uma tag anotada é semelhante, mas contém o SHA-1 de um objeto de tag no banco de dados, e esse objeto, por sua vez, faz referência a uma confirmação, além de conter todas as informações extras, como a descrição da tag. A única diferença entre ramificação e tag é que ramos se movem, as tags não. Se eu criar um novo commit no momento, o master se moverá para rastreá-lo porque é o ramo atual, mas a tag permanecerá no mesmo objeto para sempre.

**Publicando no repositório remoto**

A seguir estão alguns exemplos de comandos utilizados:

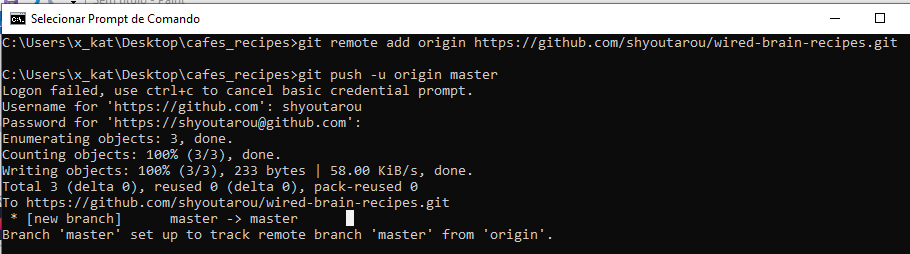
|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git remote add origin https:/// | Adicione um renomeado chamado origin que aponte para seu repositório remoto no Github |
| git remote | Exibe o nome do diretório remoto |
| git remote -v show | Exibe os nomes dos controles remotos junto com a URL correspondente |
| git remote add bob | Adicione um nome remoto bob que aponte para o repositório local ../math.bob |
| git ls-remote bob | Exibe as referências de um repositório remoto |
| GIT\_TRACE\_PACKET git ls-remote REMOTE | Exibir a interação de rede subjacente |
| git push -u origin master | Empurre seu ramo mestre para o remoto chamado origin |
| git push origin tagname | empurre a tag denominada tagname para a origem nomeada remota |
| git push –tags | envia todas as tags para o controle remoto padrão |
| git push origin :tagname | exclua a tag denominada tagname na origem nomeada remota |

Em seguida, precisamos conectar nossos repositórios locais ao que repositório remoto no Github, vinculando os dois. Isso pode ser feito com o comando:

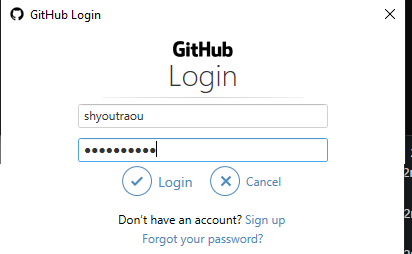
**git remote add origin https://github.com/shyoutarou/wired-brain-recipes.git**

Que indica a URL HTTPS que copiamos anteriormente como sendo origem do nosso repositório remoto, permitindo fazer push e pull de mudanças entre os dois. Por fim, vamos executar a operação de push com o comando:

**git push –u origin master**

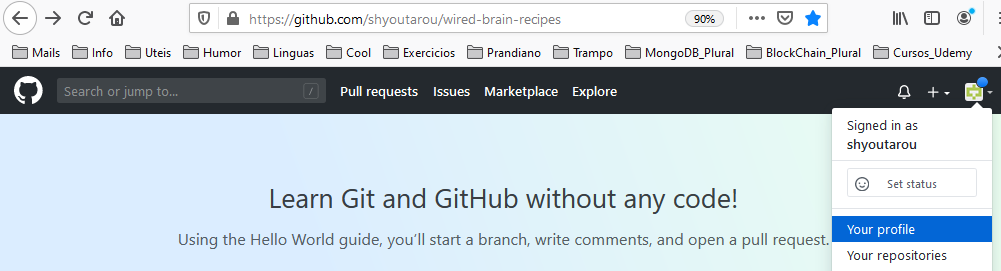


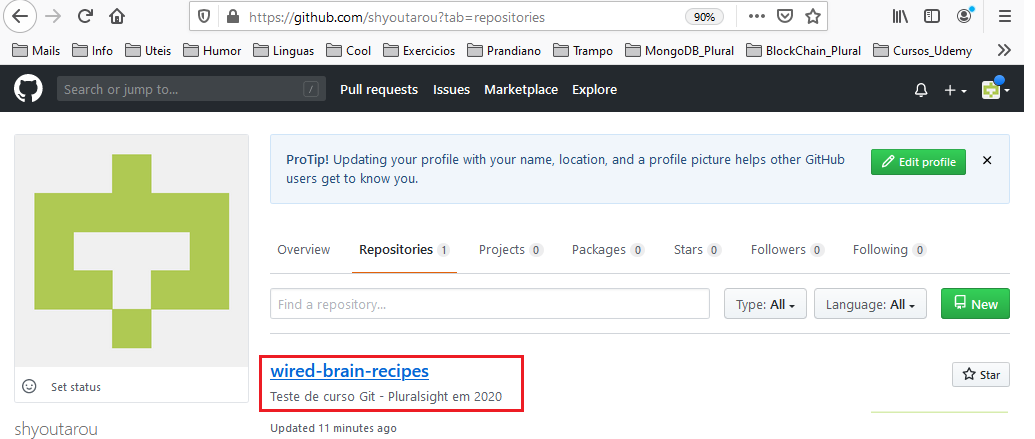
Neste comando é solicitado a inserir nosso nome de usuário e senha do Github que pode ser pelo console ou por uma caixa de diálogo:

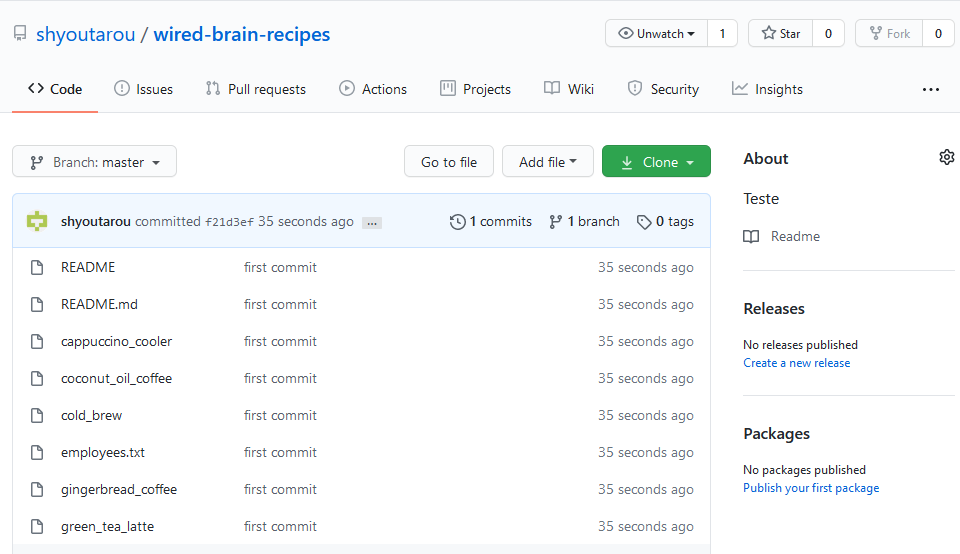


Se aparecerem algumas informações, na parte inferior, indicando que o Branch Master está configurado para rastrear o Branch Master remoto desde a origem significa que tudo funcionou e que os repositórios estão configurados corretamente.

Agora, se voltarmos à página principal dos perfis do hub de obtenção, veremos agora que temos um repositório listado e se clicarmos nessa seção de repositórios e, em seguida, nos repositórios reais, podemos ver que as receitas de traço cerebral com traço com fio recebem repositórios que foi local agora está sendo hospedado e rastreado no Github.



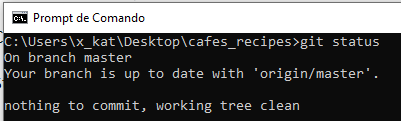




**COMANDOS BÁSICOS DO GIT**

**Status Git**

Você pode verificar o status do seu projeto a qualquer momento usando o comando git status. Se digitarmos o status Git em nossa linha de comando, Git responderá com algumas linhas de informação.



A primeira linha indica o ramo em que estamos, que é o ramo mestre. Quando inicializamos os repositórios Git, criamos uma ramificação principal automaticamente chamada de Ramificação Mestre. Uma ramificação (branch) no Git é um ponteiro para o seu projeto em um ponto específico no tempo. Agora, como nossos dois repositórios estão sincronizados, nossa ramificação principal local é igual à ramificação Master de Origem. A segunda linha na resposta menciona que nossa ramificação está atualizada com o mestre de origem e a terceira e última linha está dizendo que não há nada a se atualizar e que não modificamos nenhum arquivo, portanto ainda não há alterações em nosso diretório de trabalho e nada ainda foi preparado que ainda está aguardando confirmação.

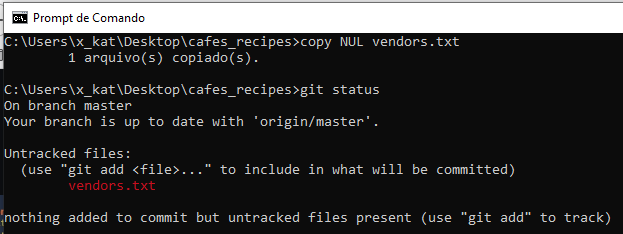
Como já foi mencionado cada arquivo em nosso projeto pode estar em um dos dois estados: **rastreado ou não rastreado**. Os arquivos rastreados são aqueles que estavam nos arquivos rastreados anteriores do snapshot anterior e podem estar em um dos três estados, que são: Não modificado (unmodified/commited), Modificados (modified) e Preparado (staged). Quando fazemos uma alteração em nosso arquivo confirmado ou não modificado, ele passa de um estágio confirmado para o estágio modificado. Quando estamos satisfeitos com nossas alterações e queremos prepará-las para confirmação, as adicionamos à nossa área de preparação. Em seguida, pegamos as alterações na área preparada e as confirmamos em nossos repositórios.

**copy NUL arquivo OU echo >> arquivo**

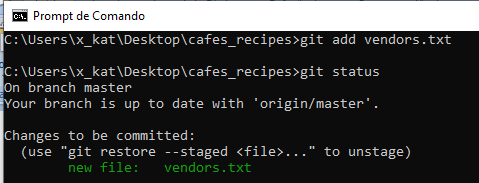
Por exemplo, ao adicionar um novo arquivo em seus repositórios chamado vendors com o comando:

**copy NUL vendors.txt OU echo >> vendors.txt**

Ao executarmos novamente o comando Git status, veremos agora que o Git destaca os arquivos não rastreados e Git fornece algumas informações úteis sobre como adicionar esse arquivo não rastreado.



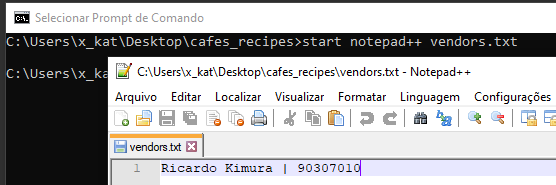
Ao rastrear esse arquivo digitando o comando: **git add vendors.txt** e executarmos, **git status**.



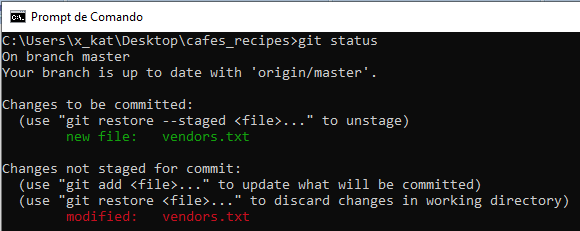
**start notepad++ arquivo OU start code arquivo**

Vemos que novo arquivo txt está sendo rastreado e que é há uma nova alteração aguardando confirmação. Os arquivos modificados que foram adicionados agora estão na fase de preparação aguardando a serem confirmadas. O Git status nos fornece outra mensagem que diz como redefinir ou remover arquivos da área de preparação. Antes de confirmar este arquivo, vamos adicionar algumas alterações neste arquivo que já está sendo rastreado. Abra o arquivo chamando seu editor padrão com o comando:

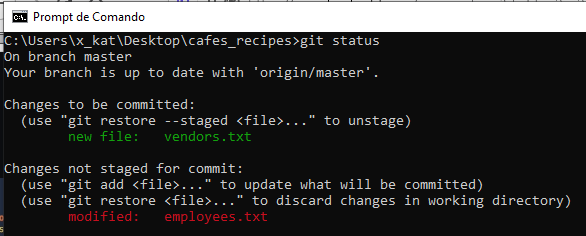
**start notepad++ vendors.txt**



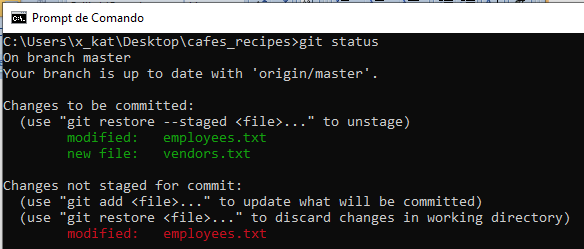
Adicione um novo vendedor que acabou de ser contratado, salve esse arquivo e volte à janela do terminal. Se digitarmos Git status novamente, veremos que o arquivo vendors.txt agora está na seção de mudanças, não para a confirmação.



Isso significa que o arquivo está sendo rastreado por Git e foi modificado em nosso diretório de trabalho, mas ainda não foi adicionado à nossa área de preparação para o commit. Para isso, é usado novamente o git add que serve tanto para rastrear novos arquivos, bem como arquivos modificados. Se alterarmos novamente um arquivo que já estivesse na área de preparados como, por exemplo employees.txt, teríamos o seguinte:



Agora se inserirmos este arquivo na aréa de preparados com o comando add mas, após isso, editarmos o arquivo novamente teríamos:

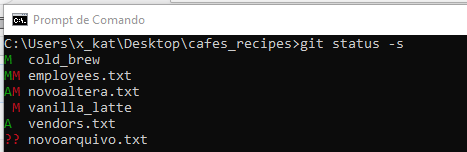


Observe que agora o arquivo employees.txt está em dois estágios ao mesmo tempo, no estágio pronto para serem confirmado, bem como no estágio de alterações modificadas ainda não preparadas. Quando preparamos pela primeira vez nossos funcionários, essas alterações foram adicionadas naquele momento para serem preparadas mas, quando alteramos o arquivo em seguida, o Git entende isso como uma nova alteração que não foi adicionado para área de preparação. Isso significa se confirmarmos com o comando comitt antes de adicionar o arquivo employees na área de preparados, as alterações do arquivo não serão inclusas na confirmação.

**git status –-short OU git status -s**

Para obter um status curto que fornece a mesma quantidade de informações, mas de uma maneira muito menor e concisa, digite os comandos:

**git status –-short OU git status -s**



Depois de executar este comando Git, o nome dos arquivos será colocado na coluna da extrema direita. A primeira coluna (em verde) indica que o arquivo foi preparado e a coluna do meio (em vermelho) indica que o arquivo foi modificado, mas permanece no diretório de trabalho. A letra M significa modificado e a letra A significa novos arquivos que foram adicionados à área de teste e pontos de interrogação (??) significam que há um novo arquivo que não é rastreado pelo Git.

|  |  |  |  |
| --- | --- | --- | --- |
| **Colunas** | | **Arquivo** | **Situação** |
| M |  | cold\_brew | Arquivo rastreado, modificado e preparado |
| M | M | employees.txt | Arquivo rastreado, modificado e preparado, novamente modificado e não preparado |
| A | M | novoaltera.txt | Arquivo novo rastreado, modificado e não preparado |
|  | M | vanilla\_latte | Arquivo rastreado, modificado e não preparado |
| A |  | vendors.txt | Arquivo novo rastreado, não modificado e preparado |
| ? | ? | novoarquivo.txt | Arquivo novo não rastreado, não modificado e não preparado |

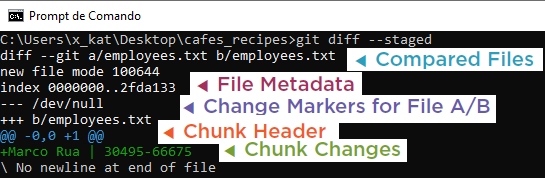
**git diff --staged**

Se o comando Git status for muito vago e você quiser ver as alterações reais sendo feitas nesses arquivos e não apenas saber quais arquivos foram alterados, você pode usar o comando git diff que pode ser usado para responder a duas perguntas.

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git diff | Mostrar qualquer diferença entre os arquivos rastreados no diretório atual/local e na área de preparação (índice sincronizado) |
| git diff –staged | Mostre qualquer diferença entre os arquivos na área de preparação e no repositório |
| git diff –cached | Exibe todas as diferenças entre o índice sincronizado e o último commit; |
| git diff HEAD | Exibe todas as diferenças entre sua cópia local e o último commit realizado; |
| git diff SHA1 SHA1 | Exibe todas as diferenças entre commits; |

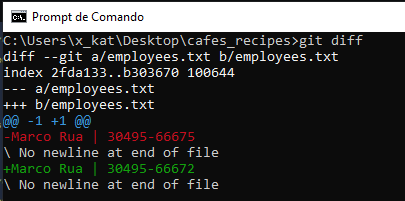
1. Que mudanças eu realizei que estão preparadas para serem confirmadas?

**git diff --staged OU git diff --staged --no-renames**



1. Que mudanças eu realizei, mas não estão preparadas?

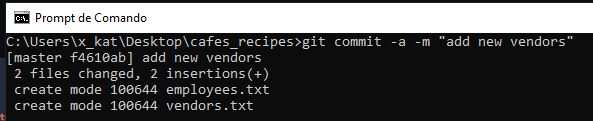
**git diff**



**GIT COMMIT**

**git commit -a -m "comentário"**

git commit -a -m "add new vendors"



Ao adicionar o parâmetro “-a” ao comando commit elimina a necessidade de aplicar o comando de adicionar (add) no estágio de preparados e adiciona-os diretamente ao estágio de confirmados. Mas isso acontece apenas aos arquivos que estejam sendo rastreados, arquivos novos (com a marcação ??), precisam ser aplicados o comando add antes de ser confirmados.

**GIT LOG**

A vantagem de um sistema de controle de versão é que ele registra alterações. Esses registros nos permitem recuperar os dados como confirmações, descobrir bugs, atualizações. Mas toda essa história será inútil se não pudermos navegar nela. Neste ponto, precisamos do comando git log. O Git log é uma ferramenta utilitária para revisar e ler um histórico de tudo o que acontece com um repositório. Várias opções podem ser usadas com um log git para tornar o histórico mais específico.

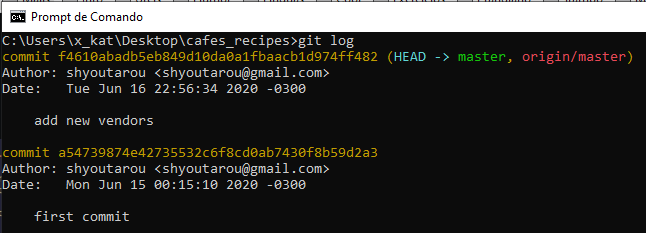
Geralmente, o log do git é um registro de confirmações. Um log git contém os seguintes dados:

* **Commit hash:** que é um dado de soma de verificação de 40 caracteres gerado pelo algoritmo SHA (Secure Hash Algorithm). É um número único.
* **Commit Author metadata:** as informações de autores, como nome e email do autor.
* **Commit Date metadata:** é um registro de data e hora para a hora da confirmação.
* **Commit title/message:** é a visão geral do commit fornecido na mensagem de commit.

Para sair do comando git log basta pressionar a tecla Q (Quit). Ele sairá da situação e voltará para a linha de comando.

O comando log do Git é um dos comandos mais comuns do git. É o comando mais útil para o Git. Toda vez que você precisar verificar o histórico, precisará usar o comando git log. O comando git log básico exibirá as confirmações mais recentes e o status do cabeçalho. Ele será usado como:

**git log**



O comando acima está listando todas as confirmações recentes. Cada confirmação contém algum sha-id exclusivo, gerado pelo algoritmo SHA. Ele também inclui a data, hora, autor e alguns detalhes adicionais.

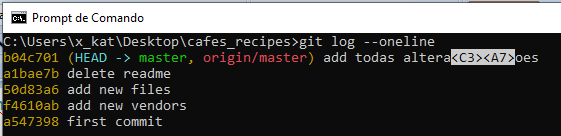
Podemos executar algumas ações, como rolar, pular, mover e sair na linha de comando. Para rolar na linha de comando, utilize os comandos LESS descrito na sessão Git help visto anteriormente. A seguir estão um resumo de comandos log utilizados:

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git log | Exibir o histórico do log |
| git log -3 | Exibir um quantidade específica de logs, aqui exibirá 3 |
| git log –parents | Mostra o histórico, exibindo o ID SHA1 do commit pai para cada commit |
| git log –parents –abbrev-commit | Igual ao comando anterior, mas reduza o ID SHA1 |
| git log –oneline | Exibir histórico de forma concisa, usando uma linha por cada confirmação |
| git log –patch | Exibe o histórico, mostrando as diferenças de arquivo entre cada confirmação |
| git log –stat | Exibe o histórico, mostrando um resumo das alterações do arquivo entre cada confirmação |
| git log –patch-with-stat | Exibe o histórico combinando o patch e a saída stat |
| git log –status | Exibe o log com os arquivos que foram modificados |
| git log –shortstat –oneline | Mostrar histórico usando uma linha por confirmação e listando cada arquivo alterado por confirmação |
| git log –merges | Lista confirmações resultantes de mesclagens |
| git log –oneline FILE | List on-line que afetam FILE |
| git log –oneline -all | Exibe todas as entradas de log de confirmação de todas as filiais |
| git log –simplify-by-decoration –decorate –all –oneline | Exibir o histórico de uma forma simplificada |
| git log –grep=STRING | Lista commits que possui STRING na mensagem de confirmação |
| git log –since MM/DD/YYYY –until MM//DD/YYYY | Lista commits entre duas datas |
| git shortlog | Resume commits de vários autores |
| git shorlog -e | Resume os commits de vários autores, incluindo email |
| git log –author=AUTHOR | Lista commits por autor |
| git log -stat HEAD^..HEAD | Lista a diferença entre o ramo com check-out atual e seu pai imediato |
| git log –oneline master..new\_feature | Mostra as confirmações entre o ramo mestre e o ramo do recurso |
| git log –graph –decorate –pretty=oneline –abbrev-commit | Veja o histórico do repositório em todas as ramificações |

**git log --oneline**

A opção on-line é usada para exibir a saída como uma confirmação por linha. Ele também mostra a saída resumidamente, como os sete primeiros caracteres do SHA de confirmação e a mensagem de confirmação.

**git log --oneline**



Portanto, geralmente podemos dizer que o sinalizador --oneline faz com que o git log seja exibido:

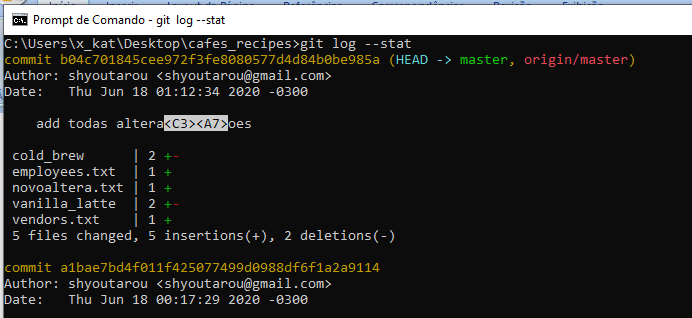
* Um commit por linha
* Os sete primeiros caracteres do SHA
* A mensagem de confirmação

Como podemos ver com mais precisão a partir da saída acima, todo commit é dado apenas em uma linha com um número sha de sete dígitos e uma mensagem de commit.

**git log stat**

O comando log exibe os arquivos que foram modificados. Também mostra o número de linhas e uma linha de resumo do total de registros que foram atualizados.

**git log --stat**



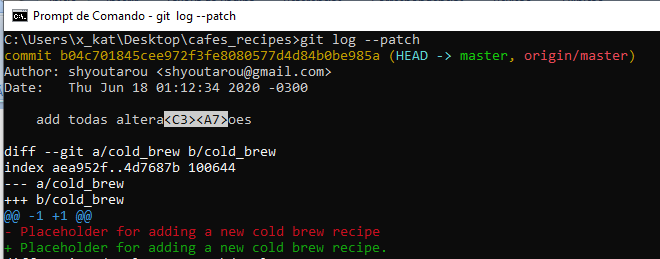
Geralmente, podemos dizer que a opção stat é usada para exibir

* os arquivos modificados,
* O número de linhas que foram adicionadas ou removidas
* Uma linha de resumo do número total de registros foi alterada
* As linhas que foram adicionadas ou removidas.

**git log p**

O comando git log patch exibe os arquivos que foram modificados. Ele também mostra o local das linhas adicionadas, removidas e atualizadas.

**git log -–patch Ou git log -p**



Geralmente, podemos dizer que o sinalizador --patch é usado para exibir:

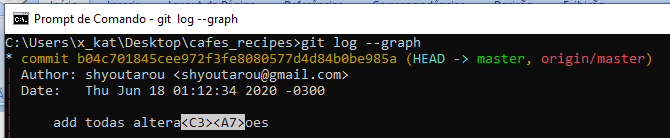
* Arquivos modificados
* O local das linhas que você adicionou ou removeu
* Alterações específicas que foram feitas.

A saída acima está exibindo os arquivos modificados com o local das linhas que foram adicionadas ou removidas.

**git log graph**

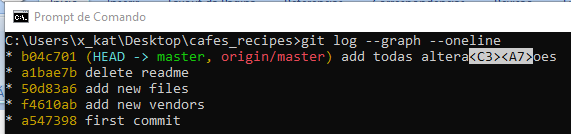
O comando Git log permite visualizar o seu git log como um gráfico. Para listar as confirmações na forma de um gráfico, execute o comando git log com a opção --graph. Ele será executado da seguinte maneira:

**git log --graph**



Para tornar a saída mais específica, você pode combinar este comando com a opção --oneline. Funcionará da seguinte maneira:

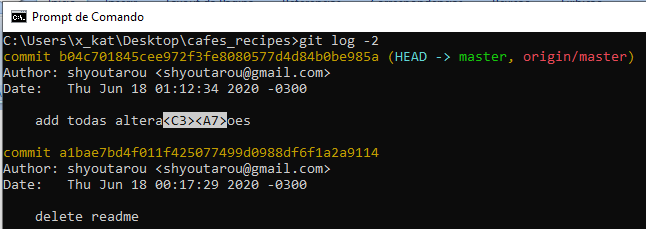
**git log --graph --oneline**



**Filtrando o histórico de Commits**

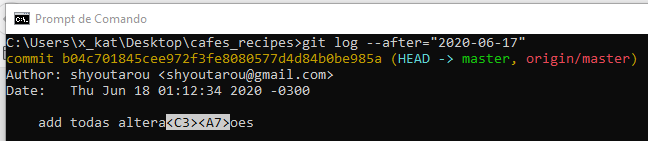
Podemos filtrar a saída de acordo com nossas necessidades. É uma característica única do Git. Podemos aplicar muitos filtros como quantidade, data, autor e muito mais na saída. Cada filtro tem suas especificações. Eles podem ser usados ​​para implementar algumas operações de navegação na saída.

* **Por quantidade:** Para limitar a saída do log do git, incluindo a opção - <n>. Se queremos apenas os três últimos commit, podemos passar o argumento -2 no comando git log. Considere a saída abaixo:



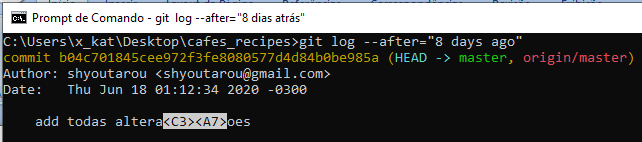
* **Por data e hora:** Podemos filtrar a saída por data e hora. Temos que passar o argumento --after ou -before para especificar a data. Esses dois argumentos aceitam uma variedade de formatos de data.:

**git log --after="2020-06-17"**



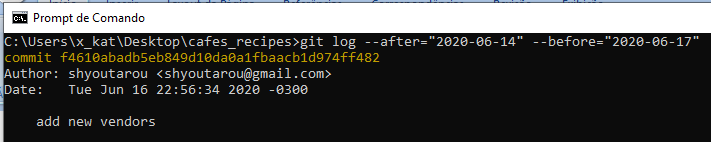
Também podemos passar uma referência aplicável como "yesterday," "1 week ago", "21 days ago,":

**git log --after="21 days ago"**



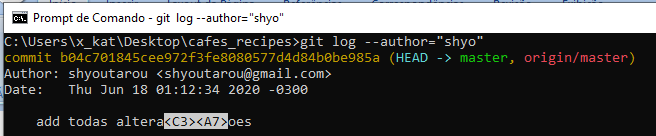
Também podemos rastrear os commits entre duas datas. Para rastrear as confirmações criadas entre duas datas, passe uma referência de instrução --after e --before da data. Podemos usar também os sinônimos --since e –unty.

**git log --after="2020-06-14" --before="2020-06-17"**



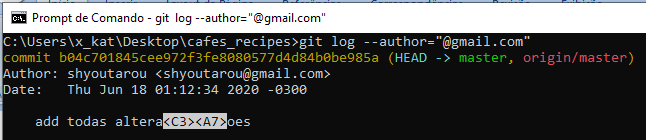
* **Por autor:** Podemos filtrar as confirmações por um usuário específico. Suponha, queremos listar os commits feitos apenas por um membro da equipe em particular. Podemos usar o sinalizador -author para filtrar os commit pelo nome do autor. Este comando usa uma expressão regular e retorna a lista de confirmações feitas por autores que correspondem a esse padrão. Você pode usar o nome exato em vez do padrão. O nome do autor não precisa ser uma correspondência exata; apenas possui a frase especificada.

**git log --author="Nome do autor"**



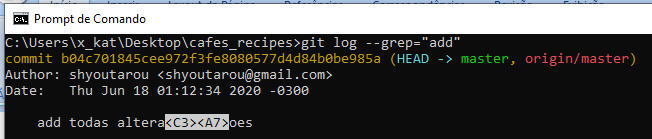
Como sabemos, o email do autor também está envolvido com o nome do autor, para que possamos usar o email do autor como padrão ou pesquisa exata. Suponhamos que desejamos rastrear os commits pelos autores cujo serviço de email é o google. Para fazer isso, podemos usar curingas como "@gmail.com".

**git log --author="@gmail.com"**



* **Por mensagem de confirmação:** Para filtrar as confirmações pela mensagem de confirmação. Podemos usar a opção grep, e ela funcionará como a opção do autor. Podemos usar a forma abreviada de mensagem de confirmação em vez de uma mensagem completa.

**git log --grep = "Confirmar mensagem."**



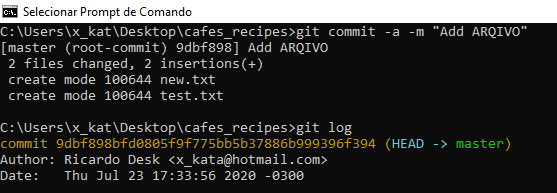
**Mensagens de Confirmação**

Muitas vezes, ao trabalhar com o Git, você pode revisar seu histórico de commits por algum motivo. Uma das grandes coisas do Git é que ele permite que você tome decisões no último momento possível. Você pode decidir em quais arquivos serão confirmados imediatamente antes de fazer o commi com a área de preparação, você pode decidir que ainda não estava planejando trabalhar em algo com o comando stash.

Pode reescrever commits que já aconteceram, para que pareçam ter acontecido de maneira diferente. Isso pode envolver alterar a ordem das commits, alterar mensagens ou modificar arquivos em uma confirmação, agrupar ou separar commits ou remover commits por completo - tudo antes de você compartilhar seu trabalho com outras pessoas.

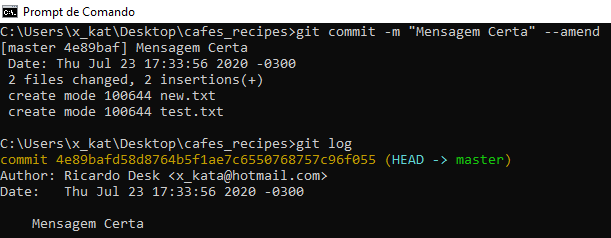
|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git commit --amend | Abre o editor de texto padrão para alterar o último commit |
| git commit –m --amend “Mensagem Certa” | Pega a área stage e a usa para realizar o commit. Se você não fez nenhuma alteração desde o último commit, então sua imagem dos arquivos irá ser exatamente a mesma, e tudo o que você alterará será a mensagem do commit e gerará um novo SHA-1. |
| git reset file | Redefina sua área de preparação, removendo as alterações feitas com o git add |
| git reset –hard HEAD@{4} | Redefina HEAD para apontar para o SHA1 ID representado por HEAD@{4}. |
| git reset --soft | Move um commit específico para sua área de preparação. Se você deseja reagrupar essas alterações em confirmações diferentes ou adicionar mais alterações antes de confirmar. |
| git reset --soft HEAD~3 | Navega por uma lista de commits e faz o HEAD apontar para três commits para trás |
| git reset  git reset --mixed | Definição padrão move de volta para o diretório de trabalho. |
| git reset --hard | Move as alterações para a lixeira. |
| git rebase -i HEAD~3 |  |

Um dos motivos mais comuns para desfazer um comando, aparece quando você executa um commit muito cedo e possivelmente esquecendo de adicionar alguns arquivos ou você escreveu a mensagem do commit de forma equivocada.



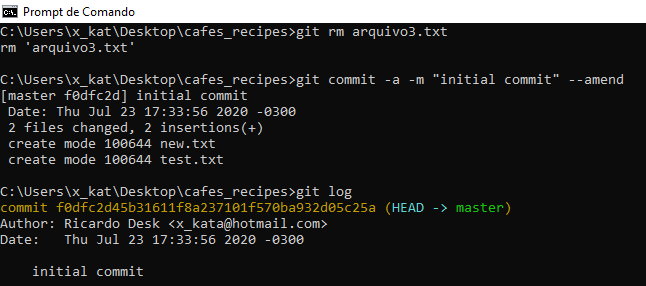
Se você quiser refazer este commit, execute o commit novamente usando a opção --amend:

**git commit --amend** OU **git commit –m --amend “Mensagem Certa”**



Esse comando pega a área stage e a usa para realizar o commit. Se você não fez nenhuma alteração desde o último commit (por exemplo, se você executar o comando imediatamente depois do commit anterior), então sua imagem dos arquivos irá ser exatamente a mesma, e tudo o que você alterará será a mensagem do commit.

O mesmo editor de mensagens de commit é acionado, porém o commit anterior já possui uma mensagem. Você pode editar a mensagem como sempre, porém esta sobrescreve a mensagem do commit anterior. Por exemplo, se você fizer um commit e então lembrar que esqueceu de colocar no stage as modificações de um arquivo que você quer adicionar no commit, você pode fazer algo semelhante a isto:



No final das contas você termina com um único commit – O segundo commit substitui o resultante do primeiro.

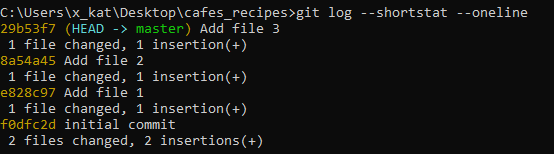
**Alterando várias mensagens de confirmação**

Para modificar uma consolidação mais antiga do seu histórico, você deve passar para ferramentas mais complexas. O Git não possui uma ferramenta de histórico de modificações, mas você pode usar duas ferramentas para alterar uma série de commits:

1. **Ferramenta Reset**
2. **Ferramenta Rebase**

**Alterar com Reset**

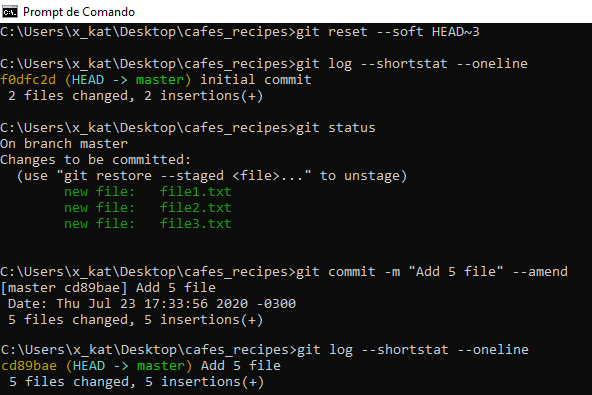
Se tivéssemos a situação de termos vários commits de correções locais. Neste momento, poderíamos enviar essas alterações para o mestre de origem no Github e incluiria esses 4 commits em sua origem.



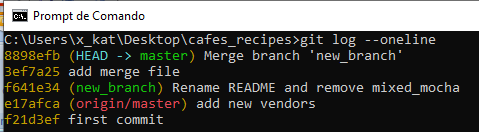
Mas e se não quisermos descartar algum desses commits ou todos eles ou alterar o histórico delas. Para isso há um comando chamada git reset que permite mover as alterações de volta à nossa área de trabalho ou de preparação. Também pode ser usado para jogar fora as alterações. Como é um comando destrutivo, é recomendável uma consideração cuidadosa ao usar. É uma prática recomendada não redefinir alterações que já tenham sido enviadas para uma origem remota, na qual outros colaboradores podem ter acessado as alterações.

Existem três opções diferentes ao usar git reset que decidem o que acontece com as confirmações que você deseja alterar: git reset --soft, git reset --mixed e git reset --hard. Se quisermos navegar por essa lista de commits podemos fazer o HEAD apontar para três commits para trás com o comando:

**git reset --soft HEAD~3**

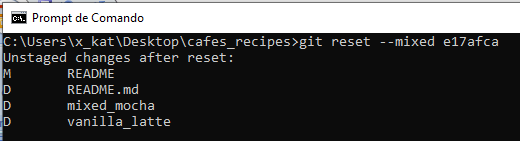


No caso de querer sincronizar o HEAD com algum branch pode ser feito como na situação abaixo:

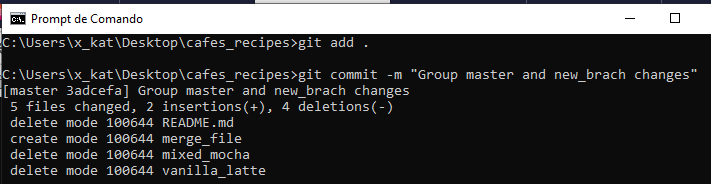


Podemos ver para aonde estão os ponteiros do HEAD local, do origin/master e do new\_branch. Nosso commit mais recente é o commit de quando mesclamos o new\_branch ao ramo mestre. As digitarmos para voltarmos ao origin/máster para o diretório de trabalho:

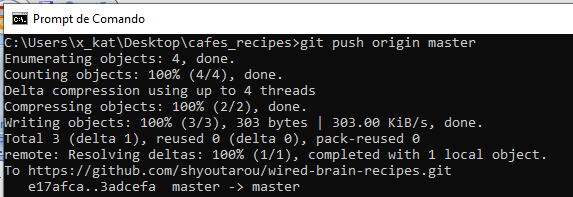
**git reset --mixed e17afca**



Vemos que nossas alterações do mestre e do ramo new\_branch, agora estão sendo exibidas em nosso diretório de trabalho. Ao adicionar essas alterações na área de preparação, vemos que todas as alterações serão feitas originalmente em nossa ramificação principal, e não mais separadas com alterações do ramo new\_branch.



Ao executarmos o commit, acabamos de alterar nosso histórico de consolidação para agrupar essas confirmações em uma única confirmação. Por fim executamos o git push origin master para atualizar o repositório remoto.

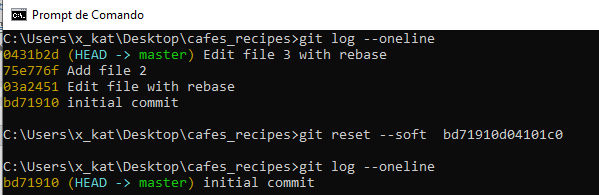


**Alterar com Rebase**

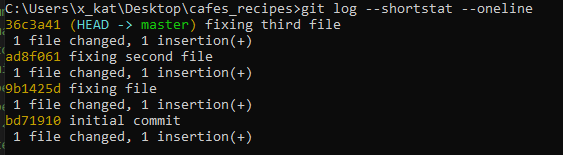
É possível utilizar o rebase em uma série de confirmações no HEAD em que elas se baseavam originalmente, em vez de movê-las para outra. Com a ferramenta de rebase interativa, você pode parar após cada confirmação que deseja modificar e alterar a mensagem, adicionar arquivos ou fazer o que quiser. Você pode executar o rebase interativamente adicionando a opção -i ao git rebase. Você deve indicar a quanto tempo você deseja reescrever confirmações, informando o comando em que se comprometer a reestruturar. Por exemplo, se você deseja alterar as três últimas mensagens de confirmação ou qualquer uma das mensagens de confirmação.

Se você errar após completar as operações abaixo e deseja refazer as operações é só dar o comando reset até o commit que pretende voltar, assim:

**git reset --soft bd71910d04101c0**



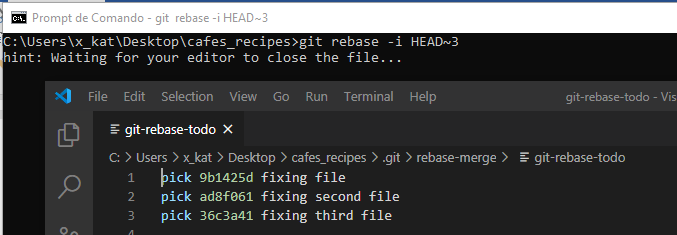
Por exemplo, se você deseja alterar as três últimas mensagens de confirmação ou qualquer uma das mensagens de confirmação desse grupo.



Você fornece como argumento para git rebase -i o pai do último commit que deseja editar, que é HEAD~2^ ou HEAD~3. Pode ser mais fácil lembrar dos ~3 porque você está tentando editar os três últimos commits; mas lembre-se de que você está designando quatro confirmações atrás, o pai da última confirmação que deseja editar:

**git rebase -i HEAD~3**

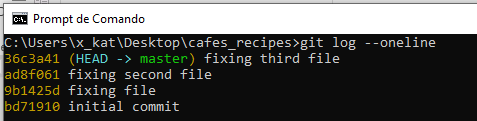
Lembre-se novamente de que este é um comando de rebasing - todos os commit incluídos no intervalo HEAD~3 até o HEAD serão reescritos(gera noco SHA-1), independentemente de você alterar ou não a mensagem. Não inclua nenhuma confirmação que você já enviou para um servidor central. Isso confundirá outros desenvolvedores ao fornecer uma versão alternativa da mesma alteração. A execução deste comando fornece uma lista de confirmações no seu editor de texto com a seguinte aparência:



Embaixo ele te ajuda dizendo quais opções você tem. É interessante testar todas as opções num repositório local.

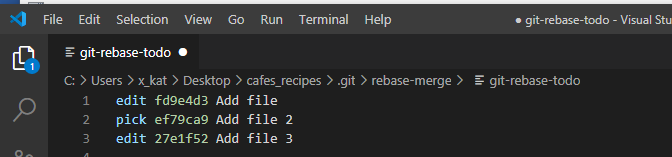
|  |  |  |
| --- | --- | --- |
| **Comando** | **Objeto** | **Descrição** |
| p, pick | <commit> | use confirmação |
| r, reword | <commit> | use confirmação, mas edite a mensagem de confirmação |
| e, edit | <commit> | use confirmação, mas pare para alterar |
| s, squash | <commit> | use a confirmação, mas mescla na confirmação anterior |
| f, fixup | <commit> | como "squash", mas descarta a mensagem de log desta confirmação |
| x, exec | <commit> | comando de execução (o resto da linha) usando shell |
| b, break | <commit> | Pare aqui e continue a rebase mais tarde com 'git rebase --continue' |
| d, drop | <commit> | remover confirmação |
| l, label | <label> | label HEAD atual com um nome |
| t, reset | <label> | redefinir HEAD para um rótulo |
| m, merge | [-C <commit> | -c <commit>] <label> [# <oneline>] | crie uma confirmação de mesclagem usando a mensagem confirmação de mesclagem original (ou on-line, se nenhuma confirmação de mesclagem original foi especificada). Use -c <commit> para reformular a mensagem de confirmação. |

É importante observar que esses commits estão listados na ordem oposta à que você normalmente os vê usando o comando log. Se você executar um log, verá algo assim:

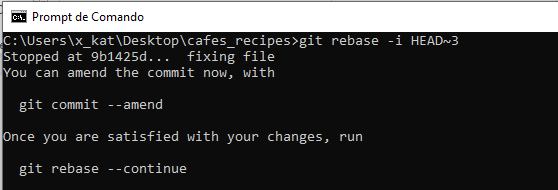


Observe a ordem inversa. O rebase interativo fornece um script que será executado. Ele começará na confirmação especificada na linha de comando (HEAD~3) e repetirá as alterações introduzidas em cada uma dessas confirmações de cima para baixo. Ele lista o mais antigo no topo, e não o mais novo, porque esse é o primeiro que será reproduzido.

Você precisa editar o script para que ele pare no commit que deseja editar. Para fazer isso, altere a palavra pick para a palavra 'edit' para cada confirmação que você deseja que o script pare depois. Por exemplo, para modificar apenas a primeira e terceira mensagem de confirmação, altere o arquivo para ficar assim:

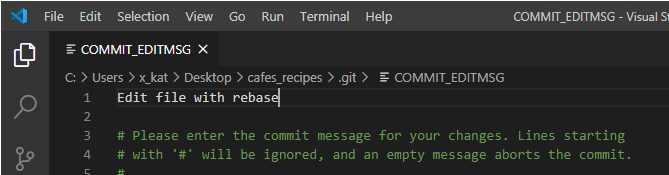


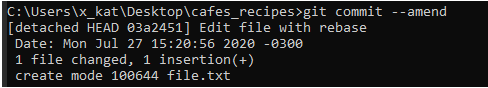
Quando você salva e sai do editor, o Git retrocede você até o último commit nessa lista e o coloca na linha de comando com a seguinte mensagem:



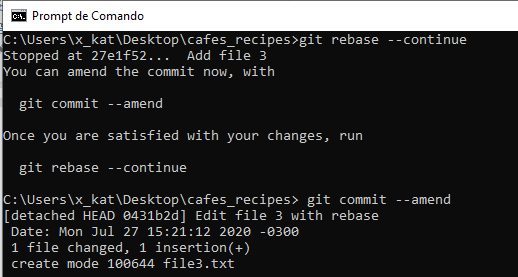
Estas instruções indicam exatamente o que fazer. Digite:

**git commit --amend**



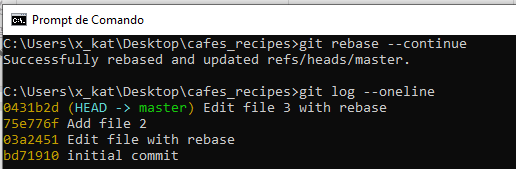


Altere a mensagem de confirmação e saia do editor. Como escolhemos para editar em mais linhas, devemos repetir essas etapas para cada confirmação alterada para editar. Sempre que o Git for interrompido, você poderá alterar o commit e continuar quando terminar.



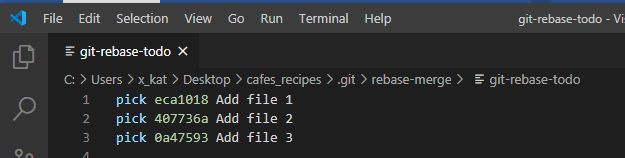
Então digite, até completar a operação, o último:

**git rebase --continue**

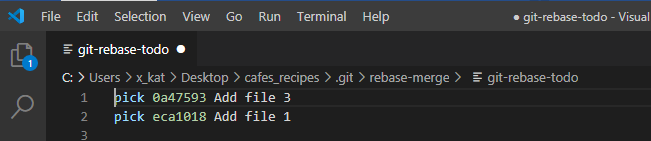


**Reordenar Commits**

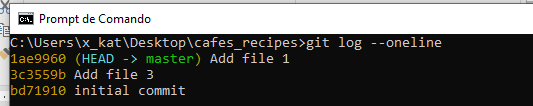
Você também pode usar rebases interativos para reordenar ou remover completamente as confirmações. Se você deseja remover o commit "arquivo-gato adicionado" e alterar a ordem em que os outros dois commits são introduzidos, você pode alterar o script rebase apartir desse estado:



Para assim:

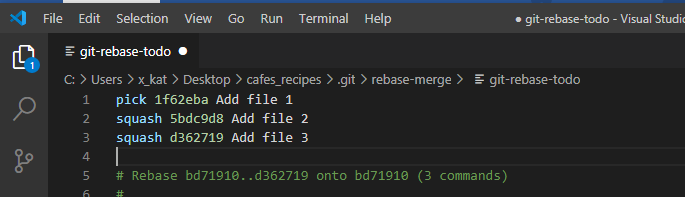


Quando você salva e sai do editor, o Git rebobina sua ramificação para o pai desses commits, aplica 0a47593 e, em seguida, eca1018 alterando a ordem desses commit e remove completamente o commit 407736a, " Add file 2".



**Squashing Commits**

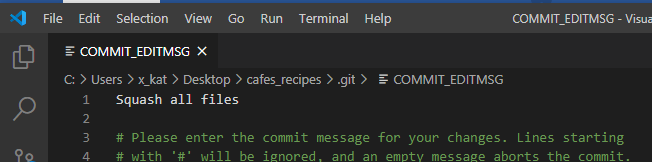
Também é possível obter uma série de confirmações e compactá-las em uma única confirmação. Portanto, se você quiser fazer um único commit a partir desses três commits, faça com que o script fique assim:



Squash é amassar tudo junto. Não precisa ser todos, eu posso inclusive pular um no meio se não fizer sentido mesclar tudo junto. Repare na barra de status do Visual Code ele indica eu está fazendo rebase:



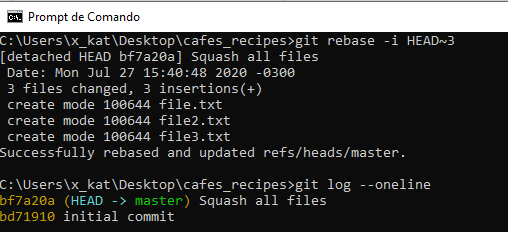
Ao fechar o editor a primeira vez, ele já abre automaticamente para edição da mensagem**:**



Ao editar, o Visual Code, gera um código indicando a alteração na barra de status:



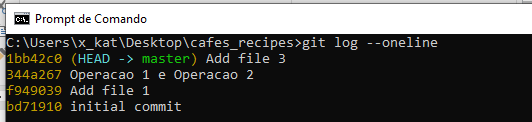
Ao retornar a janela de comando já está feita as alterações:



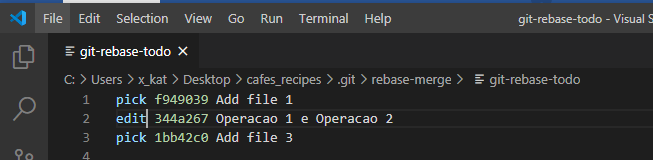
Este comando aplicará os outros dois commit automaticamente, e então você estará pronto.

**Divisão de um commit**

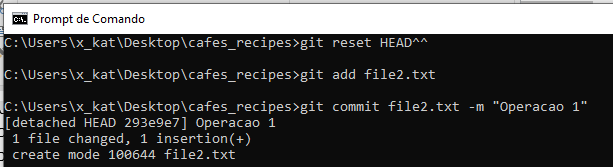
Por exemplo, suponha que você queira dividir o commit intermediário dos seus três commits. Em vez de “Operacao 1 e Operacao 2”, você deseja dividi-la em dois commits: “Operacao 1” para a primeira e “Operacao 2” para a segunda.



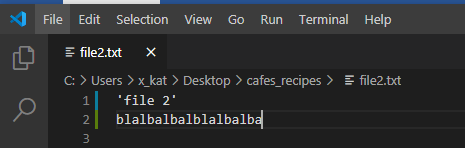
Você pode fazer isso no script rebase -i alterando a instrução no commit que deseja dividir para “edit":

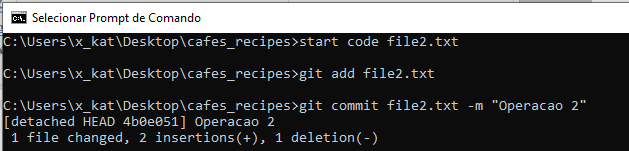


Em seguida, quando o script o coloca na linha de comando, você redefine essa confirmação, realiza as alterações que foram redefinidas e cria várias confirmações a partir delas. Quando você salva e sai do editor, o Git retrocede ao pai do primeiro commit na sua lista, aplica o primeiro commit (f7f3f6d), aplica o segundo (310154e) e o leva ao console. Lá, você pode fazer uma redefinição mista dessa confirmação com o git reset HEAD^^, que efetivamente desfaz a confirmação e deixa os arquivos modificados em estágios. Então, primeiro precisamos preparar e confirmar a inclusão do file2.txt confirmando com a mensagem “Operacao 1”;



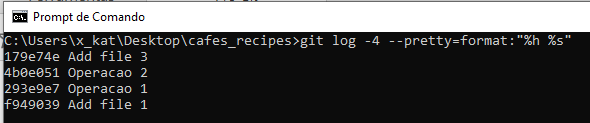
Após isso, podemos incluir arquivos ou alterar este mesmo para incluiormos um segundo commit com a mensagem “Operacao 2”:





Agora você pode preparar e confirmar arquivos até ter várias confirmações e executar o git rebase - continue quando tiver terminado:

**git log -4 --pretty=format:"%h %s"**



Mais uma vez, isso altera os SHA-1s de todas as confirmações na sua lista, portanto, verifique se nenhuma confirmação aparece nessa lista que você já enviou por push para um repositório compartilhado

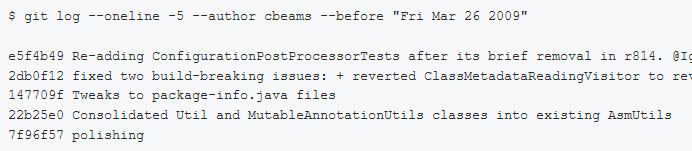
**7 Regras de mensagens**

Existem algumas regras e diretrizes gerais ao escrever uma mensagem de confirmação descritas em detalhes em um ótimo post de Chris Beams (2014):

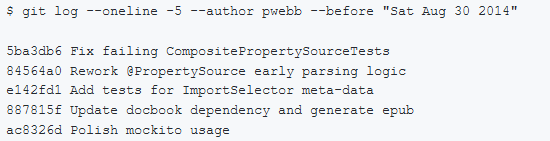
**https://chris.beams.io/posts/git-commit/**

Obviamente, essas orientações são apenas diretrizes e podem mudar dependendo de como o projeto está estruturado para ser mantido. Mas, no entanto, essas diretrizes são seguidas que facilitam no momento de reverter uma confirmação que introduziu um bug e precisarmos encontrar essa confirmação específica ou se apenas revisarmos alterações específicas introduzidas em uma confirmação. Se a mensagem de confirmação não for suficientemente detalhada, não sabemos quais arquivos foram alterados ou o que foi alterado nesses arquivos.

Se você procurar o log de qualquer repositório aleatório do Git, provavelmente encontrará suas mensagens de confirmação mais ou menos uma bagunça. Por exemplo,



Compare isso com esses commits mais recentes do mesmo repositório:



Mas um registro log bem cuidado é uma coisa bonita e útil. git blame, revert, rebase, log, shortlog e outros subcomandos ganham vida.

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git blame FILE | Exibe a saída de culpa de um FILE |
| git rebase master | Rebase sua ramificação atual com o commit mais recente do master |

Analisar solicitações de confirmação (commits) e recebimento (pull requests) de outras pessoas se torna algo que vale a pena ser feito e, de repente, pode ser feito de forma independente. Compreender por que algo aconteceu meses ou anos atrás se torna não apenas possível, mas eficiente. A abordagem de uma equipe ao log de confirmação não deve ser diferente. Para criar um histórico de revisão útil, as equipes devem primeiro concordar com uma convenção de mensagem de confirmação que defina pelo menos as três coisas a seguir:

1. **Estilo**. Sintaxe de marcação, margens de quebra automática, gramática, capitalização, pontuação. Soletre essas coisas, remova as suposições e torne tudo o mais simples possível. O resultado final será um registro notavelmente consistente, que não é apenas um prazer de ler, mas que na verdade é lido regularmente.
2. **Conteúdo**. Que tipo de informação o corpo da mensagem de confirmação (se houver) deve conter? O que não deve conter?
3. **Metadados**. Como devem ser referenciados os IDs de rastreamento, os números de solicitação pull, etc.

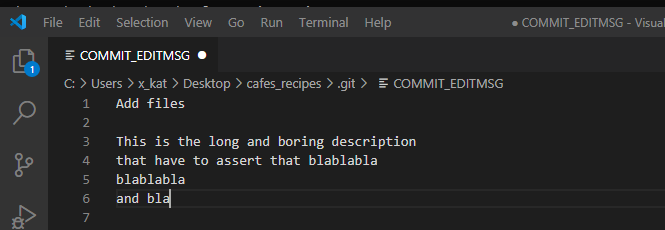
Felizmente, existem convenções bem estabelecidas sobre o que faz um Git idiomático enviar uma mensagem. De fato, muitos deles são assumidos na maneira como certos comandos do Git funcionam. Não há nada que você precise reinventar. Basta seguir as sete regras abaixo e você estará no caminho de se comprometer como um profissional.

1. **Separe o assunto do corpo com uma linha em branco**
2. **Limite a linha de assunto para 50 caracteres**
3. **Coloque em maiúscula a linha de assunto**
4. **Não termine o assunto com um período**
5. **Use o infinitivo na linha de assunto**
6. **Envolva o corpo com 72 caracteres**
7. **Use o corpo para explicar o que e por que versus como**
8. **Separe o assunto do corpo com uma linha em branco**

Em primeiro lugar, nem todo comprometimento requer um sujeito e um corpo. Às vezes, uma única linha é boa, especialmente quando a mudança é tão simples que nenhum contexto adicional é necessário. Se você está enviando algo assim na linha de comando, é fácil usar a opção -m para git commit:

**git commit -m "Corrigir erro de digitação na introdução ao guia do usuário"**

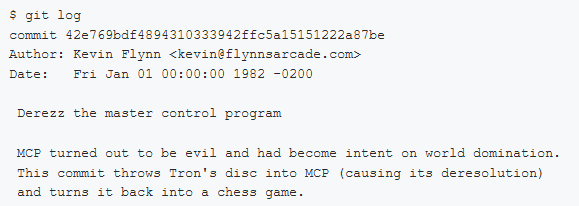
No entanto, quando um commit merece um pouco de explicação e contexto, você precisa escrever um corpo o que não é tão fácil de escrever com a opção -m. É possível escrever a mensagem em um editor de texto adequado digitando somente git commit sem parâmetros.



Ou você pode fazer isso facilmente na linha de comando. O truque é digitar assim:

**git commit -m "Title" -m "Description .........."**

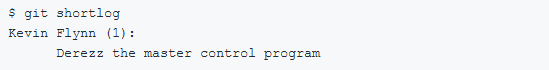
O GitHub exibirá corretamente a linha superior como sua mensagem básica e a linha inferior como a mensagem estendida. Aqui está a entrada completa do log:



E agora git log --oneline, que imprime apenas a linha de assunto:



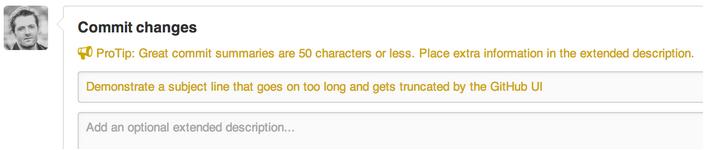
Ou, git shortlog, que os grupos confirmam pelo usuário, mostrando novamente apenas o assunto da concisão:



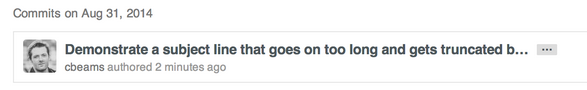
Existem vários outros contextos no Git em que a distinção entre linha de assunto e corpo entra em ação - mas nenhum deles funciona corretamente sem a linha em branco no meio.

1. **Limite a linha de assunto para 50 caracteres**

50 caracteres não é um limite rígido, apenas uma regra de ouro. Manter as linhas de assunto nesse comprimento garante que elas sejam legíveis e força o autor a pensar por um momento sobre a maneira mais concisa de explicar o que está acontecendo. A interface do usuário do GitHub está ciente dessas convenções. Ele avisará se você ultrapassar o limite de 50 caracteres:



E truncará qualquer linha de assunto com mais de 72 caracteres com reticências. Portanto, utilize 50 caracteres, mas considere 72 o limite máximo.



1. **Coloque em maiúscula a linha de assunto**

Isto é tão simples quanto parece. Comece todas as linhas de assunto com uma letra maiúscula.

* **Certo:** Acelerar para 88 milhas por hora
* **Errado:** acelerar para 88 milhas por hora

1. **Não termine o assunto com um período**

A pontuação à direita é desnecessária nas linhas de assunto. Além disso, o espaço é precioso quando você tenta mantê-los com 50 caracteres ou menos.

* **Certo:** Abrir as portas do compartimento
* **Errado:** Abrir as portas do compartimento.

1. **Use o infinitivo na linha de assunto**

O infinitivo pode parecer estranho, meio robótico, e talvez por isso que não costumamos usá-lo. Mas é perfeito para o Git confirmar linhas de assunto. Uma razão para isso é que o próprio Git usa o imperativo sempre que cria um commit em seu nome. Por exemplo, a mensagem padrão criada ao usar a mesclagem git diz:

Merge branch 'myfeature'

E ao usar o git revert:

Revert "Add the thing with the stuff"

Ou ao clicar no botão "Mesclar" em uma solicitação de recebimento do GitHub:

Merge pull request #123 from someuser/somebranch

Uma linha de assunto de confirmação do Git formada corretamente deve sempre ser capaz de concluir a seguinte frase:

Se aplicado, esse commit irá “descrever linha de assunto”

Por exemplo:

* Se aplicado, esse commit irá refatorar o subsistema X para facilitar a leitura
* Se aplicado, esse commit irá atualizar a documentação de introdução
* Se aplicado, esse commit irá remover métodos obsoletos
* Se aplicado, este commit irá lançar a versão 1.0.0

Como estamos mais acostumados a falar no indicativo ou por outras formas não infinitivas, que trata de relatar fatos ou descrever conteúdos e objetos, com esse padrão não funciona:

* Se aplicado, esse commit irá Corrigido o erro com Y
* Se aplicado, esse commit irá Mudando o comportamento de X
* Se aplicado, esse commit irá Mais correções para coisas quebradas
* Se aplicado, esse commit irá Novos métodos API agradáveis

Lembre-se: O uso do infinitivo é importante apenas na linha de assunto. Você pode relaxar essa restrição ao escrever o corpo.

1. **Envolva o corpo com 72 caracteres**

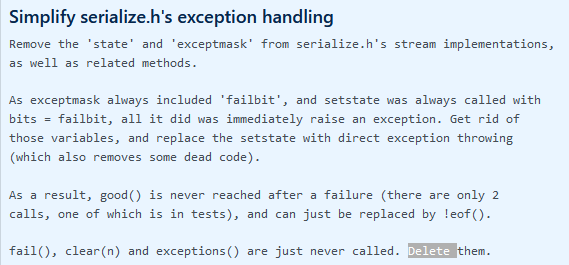
O Git nunca quebra o texto automaticamente. Ao escrever o corpo de uma mensagem de confirmação, você deve observar a margem direita e quebrar o texto manualmente. A recomendação é fazer isso com 72 caracteres, para que o Git tenha espaço de sobra para recuar o texto, mantendo tudo com menos de 80 caracteres no geral.

Um bom editor de texto pode ajudar aqui. É fácil configurar o Vim, por exemplo, para agrupar texto com 72 caracteres quando você estiver escrevendo uma confirmação do Git. Tradicionalmente, no entanto, os IDEs têm sido péssimos em fornecer suporte inteligente para quebra de texto em mensagens de confirmação (embora em versões recentes, o IntelliJ IDEA finalmente tenha melhorado sobre isso).

1. **Use o corpo para explicar o que e por que versus como**

Na maioria dos casos, você pode deixar de fora detalhes sobre como uma alteração foi feita. O código geralmente é auto-explicativo a esse respeito (e se o código é tão complexo que precisa ser explicado em prosa, é para isso que servem os comentários de origem). Apenas se concentre em esclarecer as razões pelas quais você fez a alteração em primeiro lugar - a maneira como as coisas funcionavam antes da mudança (e o que havia de errado nisso), a maneira como elas funcionam agora e por que você decidiu resolvê-la da maneira que fazia .

Este commit do Bitcoin Core é um ótimo exemplo de explicação do que mudou e por quê:



Dê uma olhada no diff completo e pense em quanto tempo o autor está economizando tempo de futuros colaboradores do projeto, dedicando um tempo para fornecer esse contexto do que mudou e por quê.

**Remover e mover arquivos**

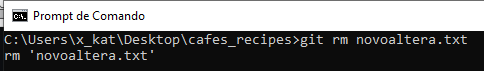
|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git rm file | Remover arquivo da área de preparação. Para diretório adicionar o parâmetro –r para executar o comando recursivamente. |
| git mv file1 file2 | Renomeie o arquivo1 para o arquivo2 na área de preparação |
| git rm --cached file | Deixa de rastrear o arquivo, mas não o remove do projeto. Para diretório adicionar o parâmetro –r para executar o comando recursivamente. |
| git revert | É sobre como fazer uma nova confirmação que reverte as alterações feitas por outras confirmações. |
| git restore | Trata-se de restaurar arquivos na árvore de trabalho a partir do índice ou de outro commit. Este comando não atualiza sua ramificação. O comando também pode ser usado para restaurar arquivos no índice de outra confirmação. |
| git reset file | Redefina sua área de preparação, removendo as alterações feitas com o git add |
| git checkout file | Confira a última versão confirmada do arquivo no seu diretório de trabalho |
| git checkout your\_SHA1id | Altere seu diretório de trabalho para corresponder a um SHA1id especificado |
| git ls-tree HEAD | Exibe todos os arquivos do HEAD |

**git rm**

Remover completamente um arquivo de seu projeto Git é algo que precisa de consideração especial, especialmente se tivermos modificado o arquivo em um projeto e sua versão do arquivo está sendo rastreada pelo Git.

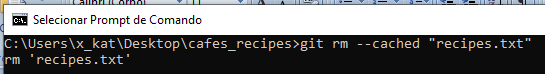
1. Digite “git ls-files” na linha de comando para exibir o arquivo.
2. Informar Git para parar de rastrear o arquivo. Podemos fazer isso usando o comando

**git rm “nomearquivo”**

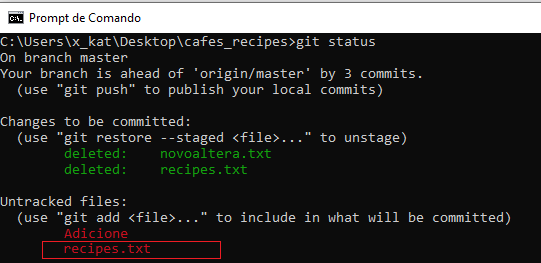


1. Para deixar de rastrear o arquivo, mas não remover-lo do projeto:

**git rm --cached “nomearquivo”**

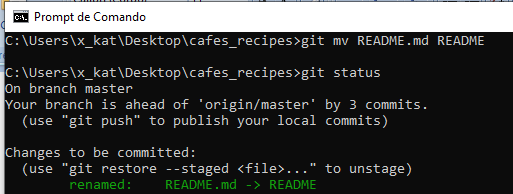


1. Digite o git status para veficar que o arquivo está sendo exibido como um arquivo excluído em nossa área de preparação. Na próxima vez que for executado o commit, esse arquivo não será mais rastreado pelo Git e será removido do nosso projeto.

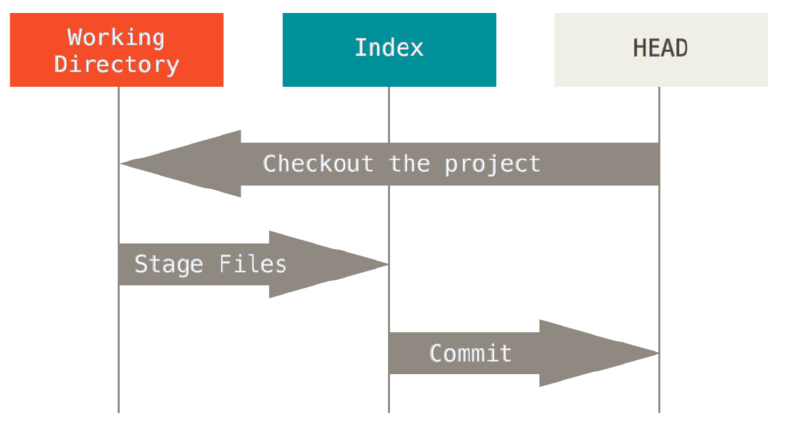


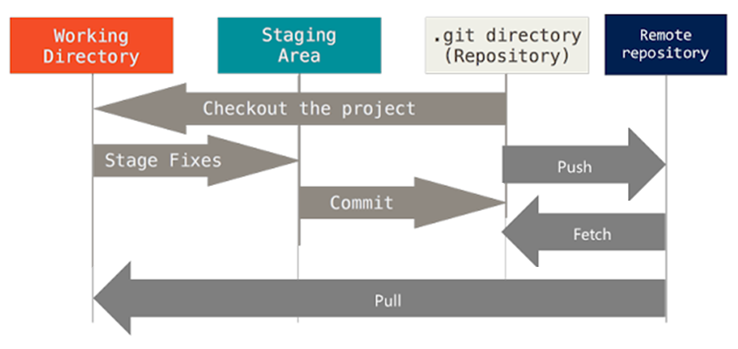
1. Para renomear um arquivo, basta digitar

**git mv file1 file2**



**GIT RESET e CHECKOUT**





**Desfazendo as Modificações de um Arquivo**

Se você não quiser manter as modificações de um arquivo. Como você pode reverter as modificações, voltando a ser como era quando foi realizado o último commit (ou clone inicial, ou seja como for que você chegou ao seu working directory):

**git checkout -- teste.txt**

Para entender melhor o comando git restore, ajuda a separar git checkout. Em dois comandos:

* Um para arquivos (git restore), que pode cobrir casos de git reset.
* Um para ramificações (git switch), que lida apenas com ramificações, não com arquivos.

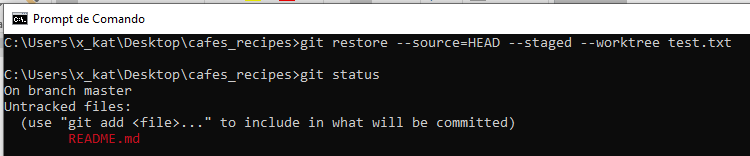
É importante entender que o git checkout -- <file> é um a comando perigoso. Qualquer modificação que você fez no arquivo se foi. O Git apenas substitui o arquivo pela última versão (mais recente) que sofreu commit. Nunca use este comando a não ser que você saiba com certeza que não quer salvar as modificações do arquivo.

Você pode restaurar o índice e a árvore de trabalho (é o mesmo que usar git-checkout)

**git restore --source=HEAD --staged --worktree test.txt**

Ou a forma abreviada que é mais prática, mas menos legível:

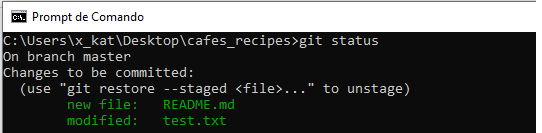
**git restore -s@ -SW teste.txt**



Lembre-se, qualquer coisa que está sobre commit com Git pode quase sempre ser recuperada. Até mesmo commits que estava em algum branches que foram deletados ou commits que forma sobre escritos através de um --amend podem ser recuperados. Contudo, qualquer coisa que você perder que nunca sofreu commit pode ser considerada praticamente perdida.

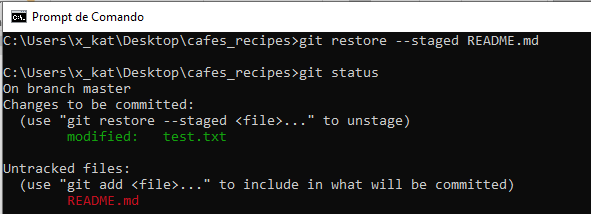
**Retirando um arquivo do Stage**

O comando **git status** que você usa para verificar o estado dessas duas áreas, também te relembra como desfazer as modificações aplicadas na área stage e work dir. Por exemplo, vamos supor que você alterou dois arquivos, e deseja realizar o commit deles separadamente, porém você acidentalmente digitou git add \* adicionando ambos ao stage. O comando git status lhe lembrará de como você pode retirar um deles do stage.



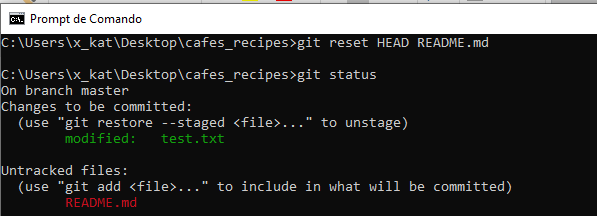
Logo abaixo do texto “Changes to be committed”, o Git indica o caminho mais seguro para retirar o arquivo do stage. Então, para restaurar um arquivo README.md no índice para corresponder à versão em HEAD:

**git restore --staged README.md**



O mesmo resultado pode ser obtido com o comando:

**git reset HEAD README.md**



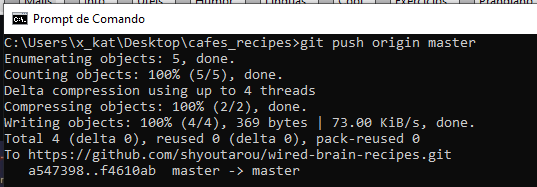
O arquivo README.md volta ao estado modificado porem está novamente fora do stage. É verdade que o comando git reset pode ser perigoso, especialmente se você usar a opção --hard. O objetivo do git-reset é "Redefinir o HEAD atual para o estado especificado", enquanto o git-restore "Restaurar arquivos da árvore de trabalho". Isso significa que muitas invocações de git-reset, se não forem cuidadosas, podem afetar outros arquivos à medida que são direcionados ao HEAD, em vez de arquivos individuais, como o git-restore.

**GIT PUSH**

**git push origin master**

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git push origin master | Empurre o ramo mestre para a origem do nome remoto |
| git push | Empurre a ramificação atual para a ramificação de rastreamento remoto padrão configurada pelo git checkout ou git push –set-upstream |

**git push origin master**



Podemos enviar nossas alterações para nosso projeto de origem usando o comando Git push, nosso nome remoto, que é origin e, em seguida, para onde iremos enviar, que é master. Vemos algumas informações de compactação, caminho URL do githu,, hashe, etc... que mostra que nosso envio foi foi enviado aos repositórios de origem sem erros.

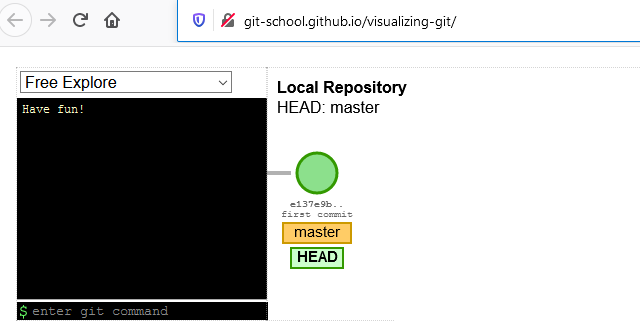
**Introdução a Branches**

Os branches são úteis nas situações em que temos alguns arquivos em nossa área de preparação aguardando confirmação (ramo mestre) mas deseja criar e modificar alguns outros arquivos (ramo secundário) que não serão confirmados juntamente com os arquivos do ramo principal.

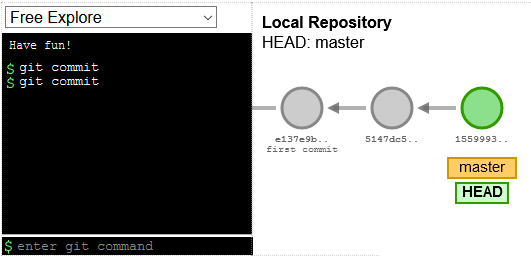
|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git branch | Listar todos os branches |
| git branch dev | Crie um novo ramo chamado dev |
| git checkout dev | Altere seu diretório de trabalho para o ramo chamado dev |
| git checkout -f master | Efetua checkout no ramo principal, descarta quaisquer alterações no ramo atual |
| git checkout -b new\_branch | Cria um branch e efetua checkout para esse ramo. |
| git checkout -b another\_fix\_branch fixing\_readme | Torna o nome de uma ramificação outro fix\_branch usando a ramificação fixa\_readme como ponto de partida |
| git branch new\_branch | Cria um branch chamado new\_branch |
| git branch -d master | Exclua o ramo chamado master |
| git branch -v | Lista todos os branches com informações do SHA1 |
| git branch -all | Mostra ramificações de rastreamento remoto além das ramificações locais |
| git branch –column | Lista os ramos pelo nome da coluna |
| git name-rev SHA1 | Dado um SHA1, ele fornece o nome da ramificação |
| git branch fixing\_readme YOUR\_SHA1ID | Criando um branch usando YOUR\_SHA1ID como ponto de partida |
| git diff BRANCH1…BRANCH2 | Indica a diferença entre BRANCH1 e BRANCH2 em relação a quando se tornaram diferentes |
| git diff –name-status BRANCH1…BRANCH2 | Resuma a diferença entre FILIAL1 e FILIAL2 |
| git push –set-upstream origin new\_branch | Cria uma ramificação de rastreamento remoto para new\_branch na origem nomeada remota |
| git config –get-regexp branch | Lista todas as definições de configuração do git que possuem a palavra branch no nome da variável |
| git branch -d local branch | Remova a ramificação local denominada ramificação local |
| git push origin :remotebranch | Remova a ramificação chamada remotebranch da origem nomeada remota |
| git cherry-pick SHA1 ID | Copie o commit no ramo atual em que você está |

Uma maneira realmente entender melhor o funcionamento dos branches é através do endereço:

**http://git-school.github.io/visualizing-git/**

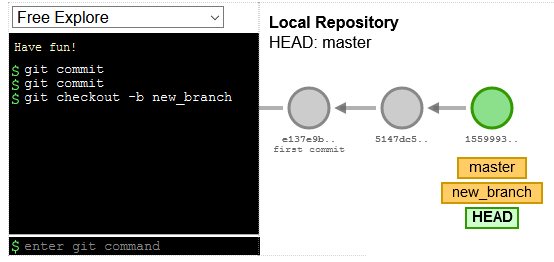


Aqui vemos um ponto verde que representa o ramo mestre dos projetos e no canto inferior esquerdo é possível inserir comandos do Git. Ao adicionar git commit ao visual, vemos que o HEAD, que é para onde estamos apontando no nosso projeto, está se movendo junto com o master. Tudo foi feito no ramo mestre até este ponto.

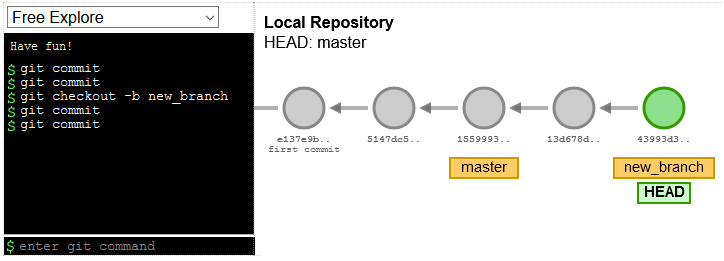


Um branch and Git é simplesmente um ponteiro leve e móvel para uma confirmação feita em um projeto. Vamos criar uma nova ramificação fora da ramificação principal digitando:

**git checkout -b new\_branch**

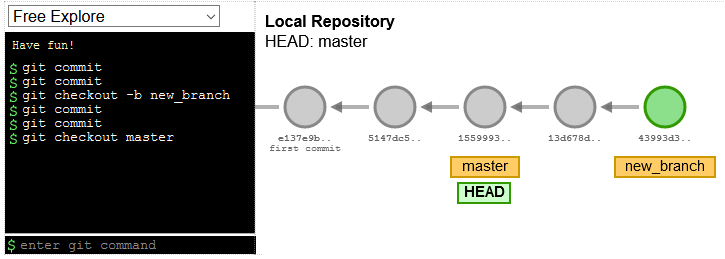


O comando Git checkout permite que você faça checkout para uma ramificação diferente, e a opção “-b” está dizendo ao git que queremos criar um novo ponteiro chamado new\_branch e depois dar o checkout. Observe que nosso novo ramo, mestre e chefe apontam (HEAD) para o mesmo commit em nosso projeto. Mas como fizemos check-out em nossa nova filial e não estamos mais em nosso mestre esse ponteiro agora está em new\_branch. Se dermos alguns commit no new\_branch:

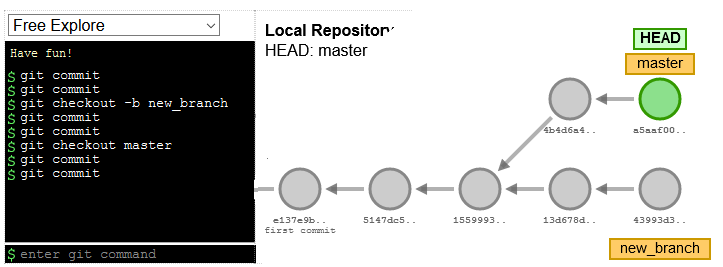


Observe que o master fica para trás e o new\_branch se move junto com o ponteiro HEAD, pois estamos fazendo commits no new\_branch e não mais no master. E podemos voltar ao nosso ramo principal digitando:

git checkout máster.



Note que o ponteiro HEAD agora foi de volta para o ramo principal. Se você confirmar em uma ramificação, a menos que sejam explicitamente alteradas, serão vinculadas a essa ramificação. Se fizermos alterações no ramo mestre, essas alterações estarão apenas no ramo mestre, elas não serão incluídos no novo ramo e vice-versa.



A ramificação do mestre nos permite criar um ambiente paralelo onde podemos experimentar novas idéias e fazer alterações ou ao trabalhar com vários colaboradores sem afetar o ramo mestre.

Quando criamos um projeto Git, e fazemos nosso primeiro commit, o Git cria uma ramificação para nós. Para consultar a lista de ramificações no projeto digite sem nenhum argumento:

**git branch**

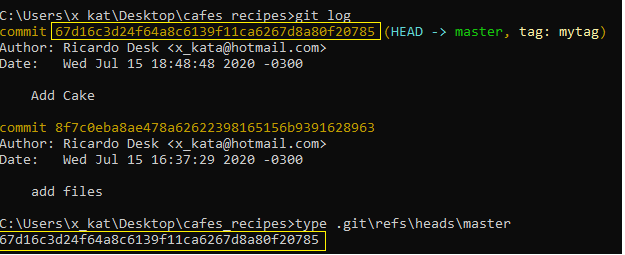


E aí está, nossa ramificação padrão, a ramificação mestre. O Git normalmente coloca ramificações aqui no diretório chamado refs e em um subdiretório chamado heads.



Observe que o ramo mestre realmente não possui status especial no Git, é apenas um ramo como qualquer outro. E tudo o que existe é esse pequeno arquivo. Na verdade, eu poderia excluir ou renomear o ramo mestre apenas excluindo ou renomeando este arquivo. Eu poderia até criar uma nova ramificação apenas escrevendo um novo arquivo nessa pasta contendo o SHA-1 de uma confirmação.

Como não está compactado, então não precisamos utilizar o comando cat-file e podemos utilizar o comando type para que imprimir seu conteúdo na tela.

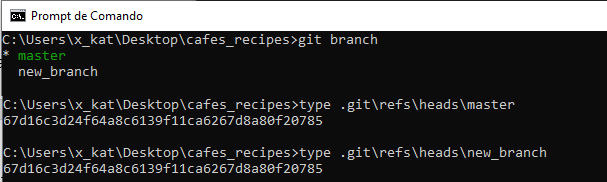


O arquivo contém uma única linha, um SHA‑1. E como você provavelmente espera, é o SHA‑1 do útimo commit. E o ramo nada mais é do que uma simples referência, um ponteiro para um commit. É por isso que o diretório que contém ramificações é chamado refs. Em vez disso, vamos criar uma nova ramificação da maneira correta, usando git branch com o nome da ramificação. Para criar um branch sem fazer checkout digite:

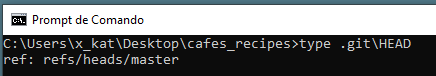
**git branch new\_branch**



Ao executar o comando git status verá que ele indica “On branch master”. Temos um novo ramo. Podemos vê-lo listado entre os branches e ao lado de master na pasta refs/heads. E se olharmos para ele, vemos que ele tem exatamente o mesmo conteúdo que o mestre, o mesmo commit. Então é isso que temos agora, dois commits, dois branches e os branches estão apontando para o mesmo commit.



Agora que temos dois ramos, se olharmos para a lista de ramos novamente, vemos que um ramo está marcado com um asterisco porque é o ramo atual. Se você olhar para a pasta .git novamente, verá um arquivo chamado HEAD. E se você olhar dentro do HEAD, verá que ele contém uma referência a um arquivo refs/heads/master, que indica qual ramificação é a ramificação atual. É isso que HEAD é, uma referência a um ramo, um ponteiro para um ponteiro.

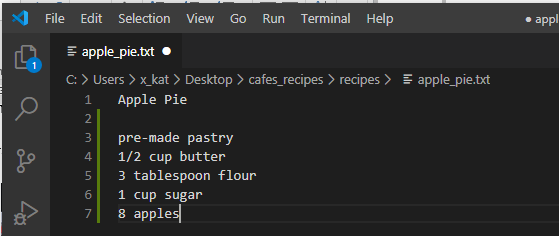


Então agora vamos alterar o arquivo apple\_pie.txt com o edit VS Code:

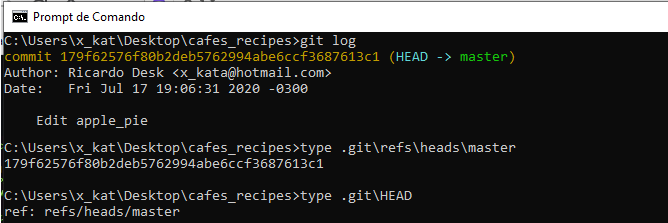
**start code recipes/apple\_pie.txt**



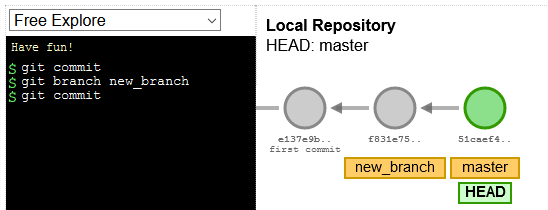
Vou adicionar a lista de ingredientes para a torta de maçã.



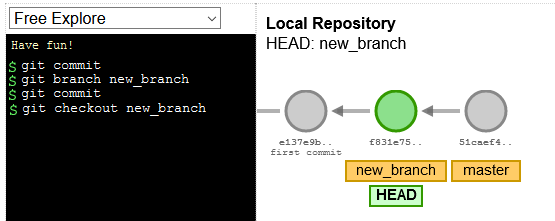
Vamos prepara as alterações e confirmá-la. O Git criou um novo objeto commit no banco de dados de objetos, e esse commit tem o commit anterior como pai. Em seguida, o Git olhou dentro do arquivo HEAD para descobrir qual é o ramo atual e moveu esse ramo para apontar para o novo commit.



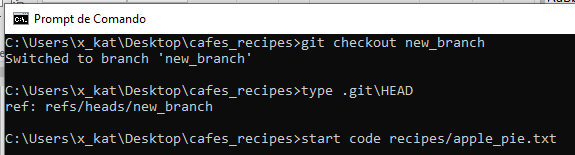
Portanto, o ramo mestre se moveu e observe que o HEAD estava apontado para o mestre antes do commit e continua pontando para o mestre.



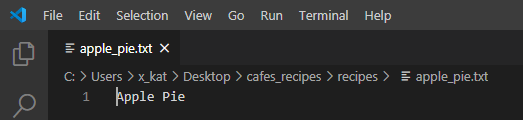
Até agora, não tocamos no novo ramo new\_branch, que está apontando para o commit anterior onde estava quando o criamos. Agora vamos fazer do new\_branch o ramo atual com a operação chamada git checkout.



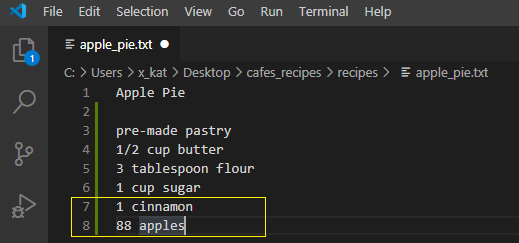
Quando eu faço check-out lisa, duas coisas acontecem. A primeira coisa que acontece é que o Git muda HEAD para apontar para new\_branch. Lá, agora o HEAD está apontando para refs/heads/new\_branch.



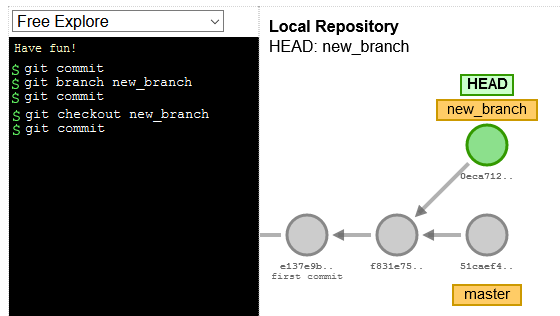
A segunda coisa que acontece é mais sutil. O Git acabou de substituir os arquivos e pastas em nossa área de trabalho, o diretório de trabalho, pelos arquivos e pastas neste commit. Portanto, após o checkout, nossa área de trabalho mudou para o conteúdo do commit apontado por new\_branch. Se eu olhar o conteúdo do arquivo da torta de maçã aqui, os ingredientes sumiram. Esta é a versão anterior do arquivo.



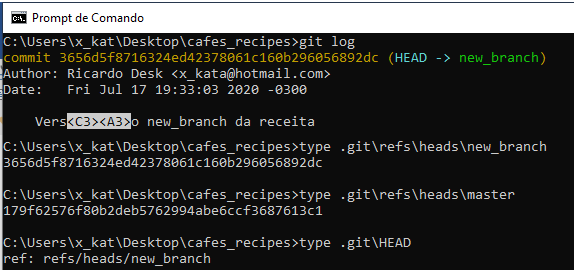
Isso que o checkout significa. Significa mover HEAD e atualizar a área de trabalho. Agora vamos modificar a receita da torta de maçã novamente. Estou usando quase a mesma lista de ingredientes que temos no ramo principal, mas new\_branch também usa canela em sua versão da torta e ela usa mais maçãs, 88 maçãs.



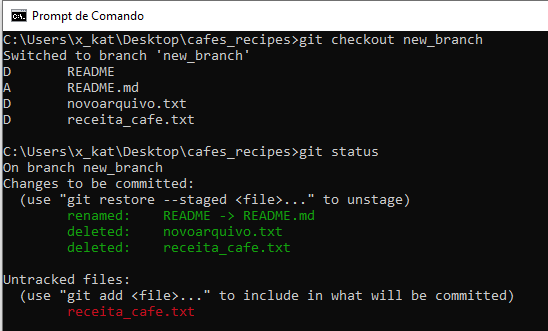
Vamos confirmar essas alterações. O Git solicitou a confirmação no banco de dados do objeto e move a ramificação atual, new\_branch, para apontar para a nova confirmação. HEAD não mudou. Mestre não mudou, é claro, mas new\_branch mudou. Agora ele aponta para o novo commit.



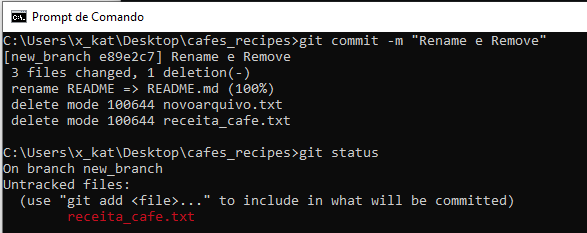
Agora, isso se parece um pouco mais com nossa noção intuitiva de ramificações, mas lembre-se de que ramificações ainda são apenas referências a confirmações.



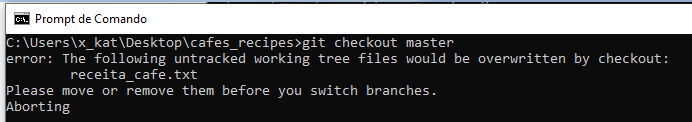
Mas repare que quando fizemos o comando git checkout new\_branch:



Vemos que as alterações em nossa branch principal são transferidas para o novo ramo, pois quando removemos e renomeamos esses arquivos em nossa branch principal, eles permaneceram em nossa área de preparação. Ainda não confirmamos essas alterações e, como criamos uma nova ramificação fora do master nesse momento, as alterações na área de preparação permaneceram. Vamos confirmar esses arquivos através do novo ramo:



E executando o status Git vemos que agora não temos nada a confirmar, mas ainda temos o arquivo não rastreado. Se executarmos o comando “git ls-files”, veremos que ou alterações neste ramo estão presentes. No entanto, se retornarmos ao nosso ramo mestre digitando git checkout master, ocorreremos um erro:



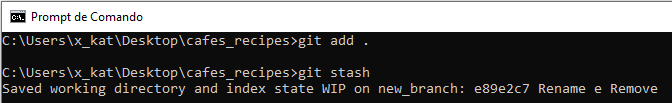
Isso ocorre pois confirmamos a exclusão do arquivo receita\_cafe.txt do controle de versão com um ramo, antes de incluir-lo na área de preparação no ramo principal. Como ele ainda está em um estado não rastreado, nós não o removemos ou excluímos do nosso sistema de arquivos. Quando tentamos fazer o checkout para obter um ramo diferente, estamos dizendo que precisamos fazer algo com esse arquivo, ou seja, removê-lo ou adicioná-lo novamente para ser rastreado.

**GIT STASH**

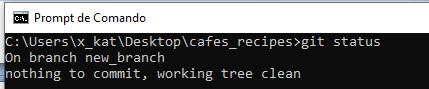
Como nosso intuito inicial era não remover esse arquivo (por isso, foi utilizado o comando git rm --cached file ) e como também não queremos adicioná-lo agora para ser incluído neste novo ramo de trabalho, o Git fornece uma opção para ocultar (stash) essa alteração.

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git stash | Defina o trabalho atual em andamento como stash, para que você possa executar um check-out do git |
| git stash list | Lista de trabalhos em andamento que você escondeu |
| git stash show | Para ter uma visão melhor detalhada de qual arquivo foi escondido em uma visão geral |
| git stash pop | Aplique o stash salvo mais recentemente no diretório de trabalho atual, remova-o do stash |

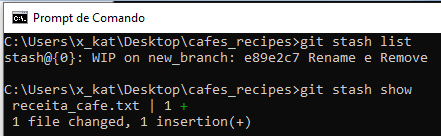
Podemos resolver adicionando o arquivo novamente digitando, “git add .”, em seguida o comando “git stash”.



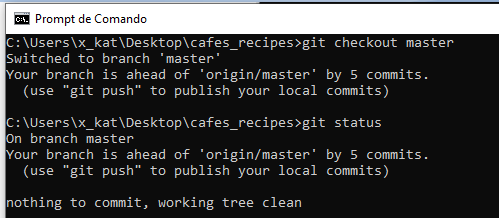
Se executarmos o git status e veja que nossa ramificação está limpa e não há nada a confirmar e nada em nosso diretório de trabalho. Então, para onde foi essa mudança?



Se digitarmos git stash list, Git nos fornecerá uma lista de alterações em andamento que armazenamos junto com a ramificação e alguns dados de confirmação, também podemos digitar git stash show para ter uma visão melhor detalhada de qual arquivo foi escondido em uma visão geral de alto nível das alterações.



Agora, se quiser tirar essa alteração do stash, teria que executar o comando “stash pop” e a alteração seria devolvida ao seu diretório de trabalho. Mas não iremos fazer isso agora, pois a utlizaremos depois. Vamos agora voltar ao nosso ramo principal com o checkout máster:



Agora, o git status mostra que existem outras confirmações feitas em nossa nova ramificação e não no mestre. No nosso ramo principal, crie um novo arquivo chamado merge\_file e edite-o no editor de texto padrão. Em seguida, adicione o arquivo para preparação e confirme nossas alterações.

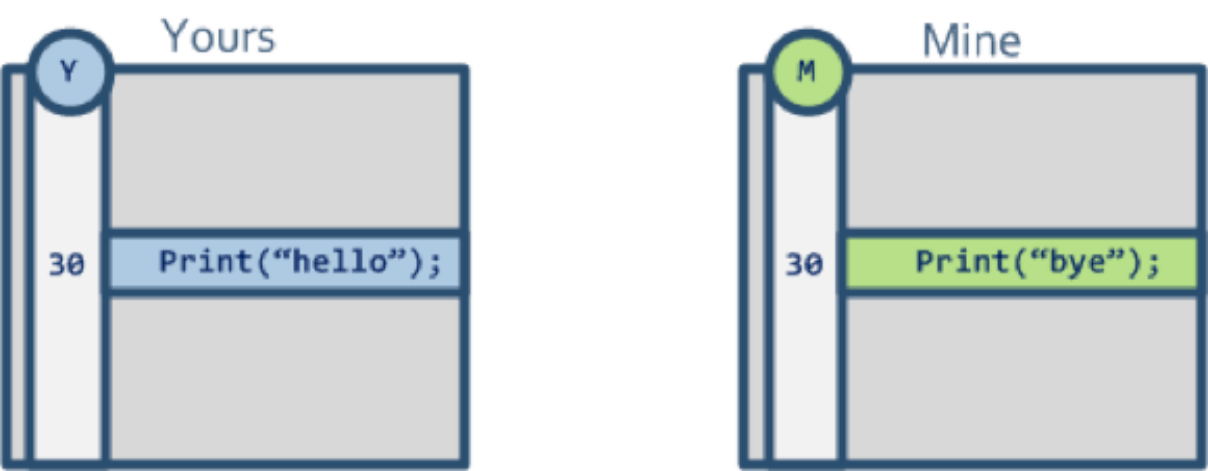


**GIT MERGE**

Em sistemas antigos com CVS, um branch é mais ou menos como duplicar o diretório inteiro do projeto e o merge é fazer diff no diretório duplicado gerar os patch de cada arquivo e aplicar de volta no diretório original. O CVS e o Subverssion antes da versão 1.5 não tinha essa informação do estado inicial do branch depois que você fazia o merge. Então não era possível mais continuar trabalhando no branch porque o próximo merge vai ter muitos mais conflitos. Esse tipo de merge é o que se chama de **“Two Way”**.

**Two-Way Merge**

Suponha que estamos modificando o mesmo arquivo simultaneamente. Você faz algumas alterações no arquivo e depois faço mais algumas alterações. Em algum momento, uma "terceira pessoa" olha para as duas cópias do arquivo.

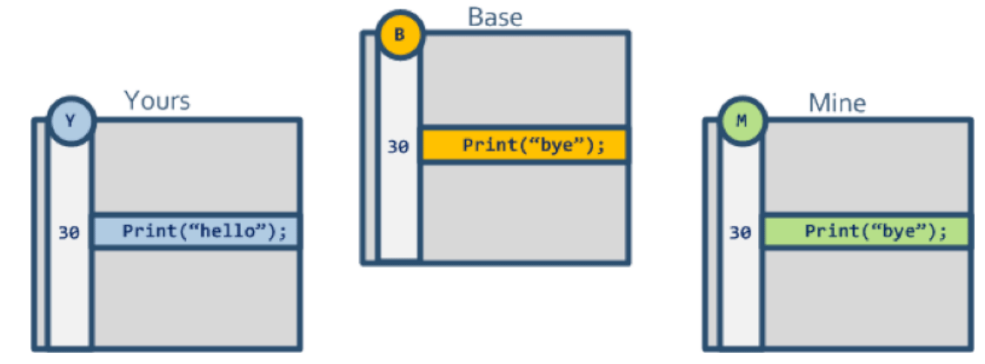


Essa "terceira pessoa" vê que há uma diferença na linha 30, mas como ele não sabe qual versão está incorreta e terá que ligar para os dois programadores e confiar em suas memórias para descobrir quem modificou o quê. É provável que essa "terceira pessoa" seja o sistema de controle de versão que tenta fazer uma mesclagem bidirecional simples, que apenas compara duas versões de um arquivo e tenta mesclá-las. Mas aqui, o VCS exigirá intervenção do usuário porque não será capaz de descobrir o que fazer.

É melhor evitar modificações simultâneas em um arquivo se você precisar confiar na mesclagem bidirecional, pois será um processo manual lento. Imagine 300 arquivos que requerem uma mesclagem simples .

**Three-Way Merge**

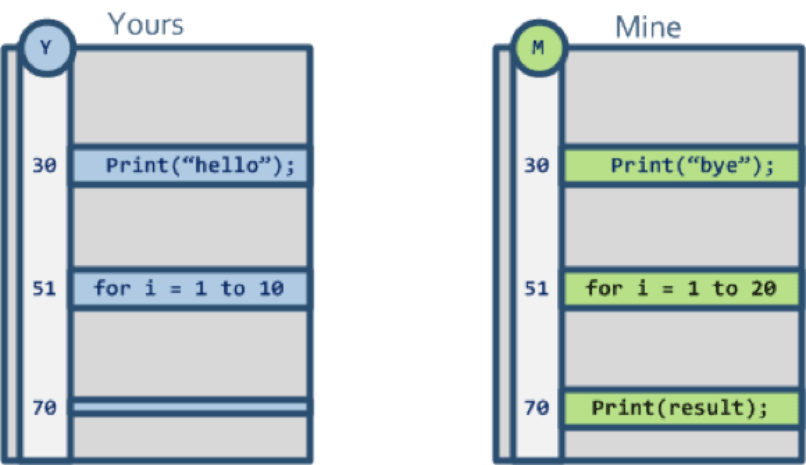
Vamos esquecer por um segundo sobre o controle de versão e voltar para essa "terceira pessoa", examinando os dois arquivos que modificamos: Para ela descobrir quem editou o arquivo, ela pode olhar para a versão original do arquivo que usamos como ponto de partida.



Então, olhando como o arquivo era originalmente ("ancestral comum de base" ou simplesmente "base"), ele pode descobrir o que fazer. Com base na Figura 2, apenas um desenvolvedor realmente alterou a linha 30, para que o conflito possa ser resolvido manualmente agora basta ligar para "Yours" pra resolver se essa alteração é válida e commitar ou inválida e desfazer.

É assim que o **Three-Way Merge** ajuda: Transforma um conflito manual em uma resolução automática. Parte da mágica aqui depende do VCS localizar a versão original do arquivo. Esta versão original é mais conhecida como "o ancestral comum mais próximo". O VCS passa o ancestral comum e os dois contribuidores para a ferramenta de mesclagem de três vias que usará os três para calcular o resultado.

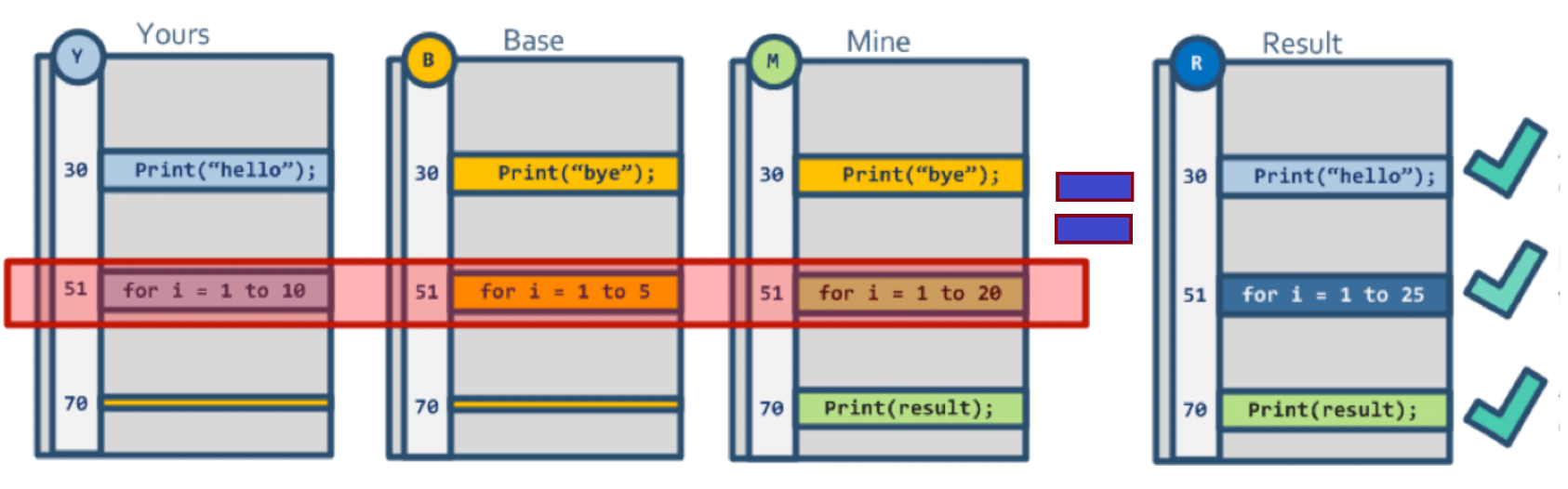
Usando apenas a fusão bidirecional, as linhas modificadas por dois desenvolvedores exigirão intervenção manual, enquanto todo o resto será mesclado automaticamente. Com a mesclagem tripla, é possível executar uma mesclagem indolor envolvendo centenas de arquivos. Vamos agora verificar um caso um pouco mais complexo, como mostra a Figura abaixo.



Comparando os dois arquivos lado a lado, podemos ver que há três linhas com diferenças:

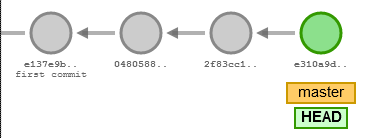
* Linha 30: com o mesmo conflito que tivemos antes.
* Linha 51: com um loop for sendo modificado.
* Linha 70: onde não sabemos se o "seu" removeu algum código ou o "meu" o adicionou.

Vamos agora olhar para o ancestral comum para poder resolver adequadamente os conflitos.

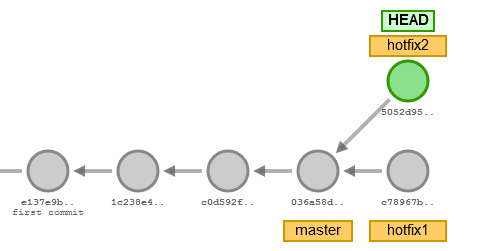


* O conflito na linha 30 pode ser resolvido automaticamente e o "Yours" (colaborador de origem)
* O conflito na linha 70 também pode ser resolvido automaticamente para "Mine" (colaborador de destino)
* O conflito na linha 51 precisa de resolução manual: você precisa decidir se deseja manter um dos contribuidores, o outro, ou mesmo modificá-lo manualmente.

O Git não prioriza a estrutura de diretórios, mas um grafo, um **DAG (Direct Acyclic Graf).** O DAG é um grafo de commits. É isso que o comando git commit sempre faz, cria um commit ligado ao anterior. É denominado Acíclico porque os nós desse grafo não apontam de volta um para o outro causando um loop infinito.



Um branch é nada mais nada menos que um desvio nesse gráfico com uma etiqueta com o nome do branch. A vantagem no Git é que todo branch tem a informação do ponto de origem a partir de qual branch começa o branch.

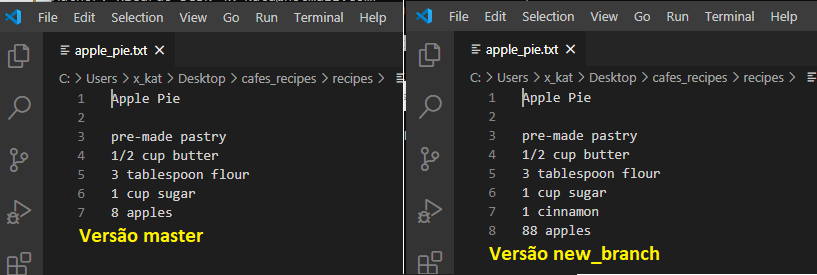


**Mesclar branch no mestre**

Mesclar uma ramificação em outra ramificação é um conceito-chave com ramificação e realmente o objetivo de criar uma ramificação.

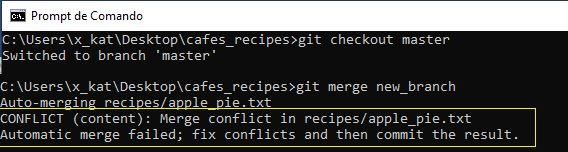
|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git merge branch2 | mesclar branch2 no ramo atual em que você está |
| git mergetool | abre uma ferramenta para ajudar a executar uma mesclagem entre dois ramos conflitantes |
| git merge –abort | abandona uma mesclagem entre dois ramos conflitantes |
| git merge-base branch1 branch2 | mostra a confirmação básica entre branch1 e branch2 |
| git merge –no-ff branch | mesclar branch na ramificação atual, criando um commit de mesclagem, mesmo que seja um commit de avanço rápido |

Primeiro, vamos voltar para a ramificação principal com o comando checkout. Se olharmos a receita da torta de maçã, encontraremos a versão de receita do master, não a versão de new\_branch.

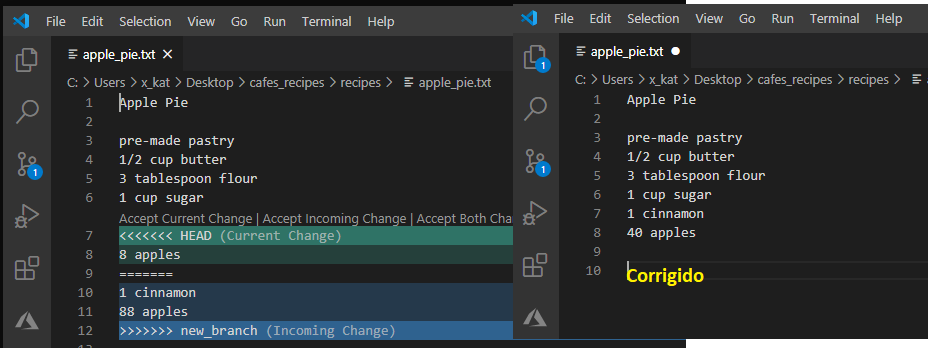


Agora, digamos que gosto mais da versão de new\_branch. Então, eu quero mesclar as alterações de new\_branch no ramo mestre com o comando git merge new\_branch e temos um conflito.

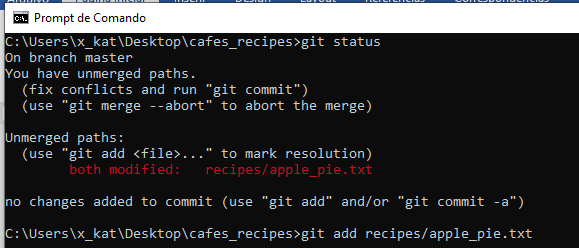
**git merge new\_branch**



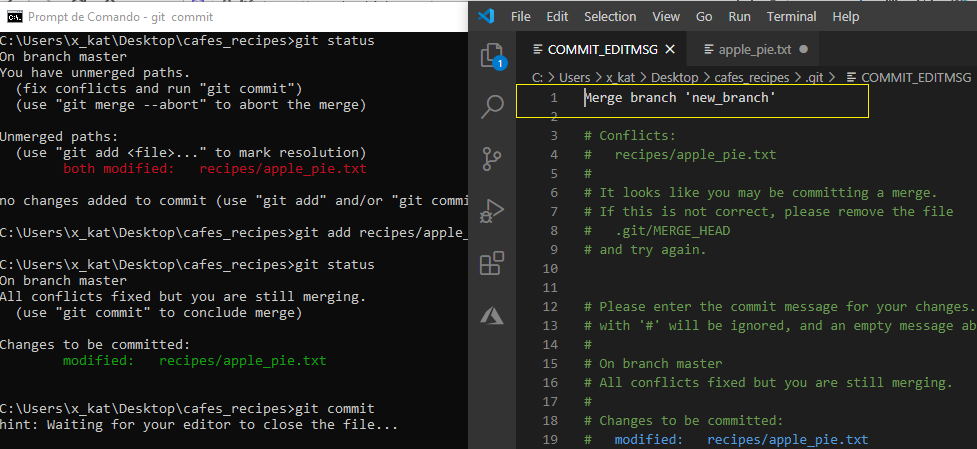
Precisamos resolver o conflito manualmente, se olharmos para dentro do arquivo da torta de maçã, veremos que esta linha foi alterada de maneira divergente em nossa receita e na receita de new\_branch, então agora precisamos nos posicionar e decidir quantas maçãs usar na torta.



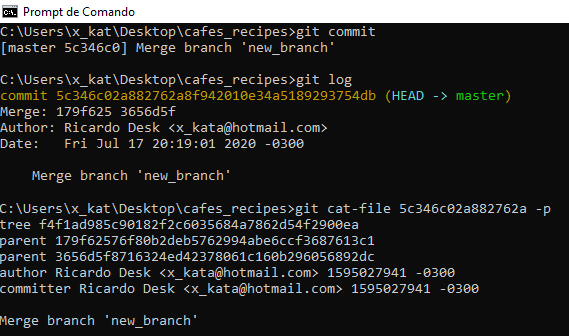
Agora, se obtivermos status, veremos que esse arquivo não está preparado para a próxima confirmação, precisamos adicioná-lo explicitamente. Esta é a nossa maneira de dizer ao Git que o conflito foi corrigido.



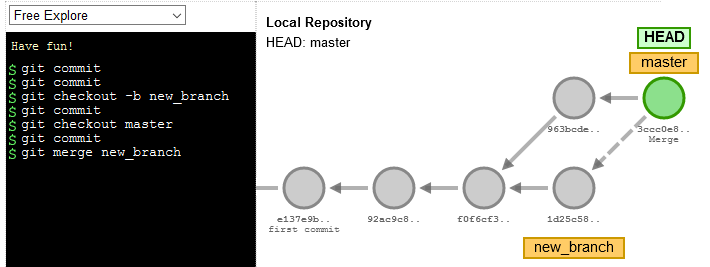
E agora podemos concluir a mesclagem. Se não tivéssemos conflitos, o Git teria feito esse último passo automaticamente, mas, como tivemos conflitos, temos que dizer, ok, terminamos de corrigir todos os conflitos no Git e fazemos isso com uma confirmação. Nem precisamos enviar uma mensagem de confirmação. O Git sabe que estamos no meio de uma mesclagem, por isso criará uma mensagem adequada automaticamente.



Poderíamos mudar a mensagem, mas não vou, apenas a aprovarei saindo do editor. Se você olhar para o log agora, verá um novo commit e, se olhar dentro desse commit com o comando de cat-file:

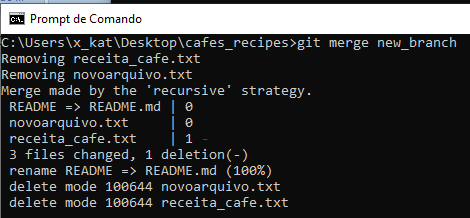


É como qualquer outro commit que vimos até agora. Uma mesclagem é apenas um commit que tem dois pais, é isso que a torna mesclada. Um commit no Git geralmente tem um pai, mas pode ter quantos pais você quiser, na verdade. A seguir temos um resumo das operações realizadas no ramo “new\_branch” e posteriormente no ramo principal que terminou com o checkout master e criação de um novo arquivo chamado merge\_file.



Se não houvesse conflitos e desejássemos mesclar alterações no ramo principal bastaria digitar:

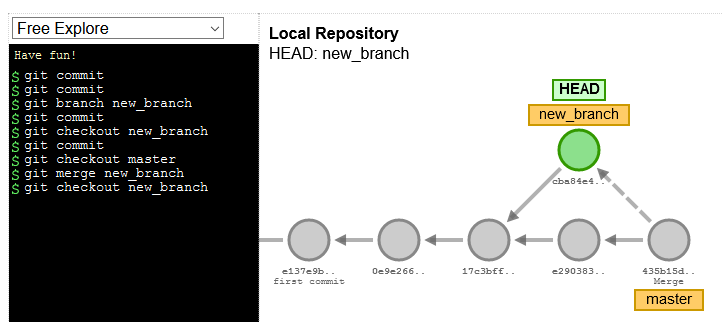
**git merge new\_branch**



Podemos ver que as alterações do new\_branch foram agora adicionadas ao nosso ramo principal.

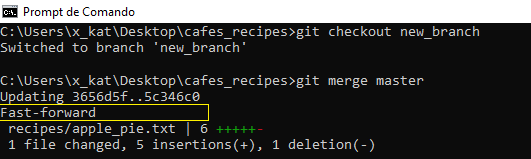
**Mesclar mestre no branch - Fast Forward**

Vimos como as ramificações e mesclagens funcionam, mas há alguns casos interessantes que não consideramos. O primeiro é um caso especial de mesclagem. Vamos iniciar dando um checkout no ramo new\_branch. Agora 0 HEAD está em new\_branch novamente.

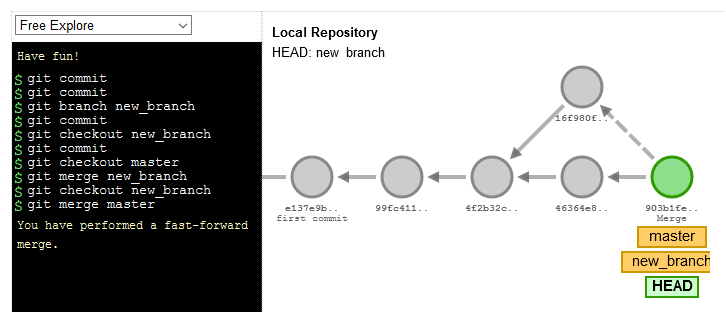


Anteriormente, fundimos new\_branch no mestre. Agora queremos mesclar o mestre no new\_branch. Isso poderia ser feito da maneira usual, exatamente como fizemos quando nos fundimos na outra direção. Ele pode criar um novo commit que tem dois pais, esses dois commit aqui seriam os pais e depois mover new\_branch para apontar para o novo commit. Esse novo commit seria garantido por não ter conflitos, porque já os resolvemos quando nos fundimos na outra direção. Portanto, seria fácil para o Git criar esse commit, mas também seria um desperdício.

Aprendemos até agora que o Git não gosta de desperdício. Portanto, você pode poupar um commit e apenas fazer isso. Move Lisa para apontar para o mesmo commit que Master. Portanto, o Git não precisou criar um novo commit. Esse truque acontece o tempo todo na prática. É chamado de avanço rápido (Fast Forward) .

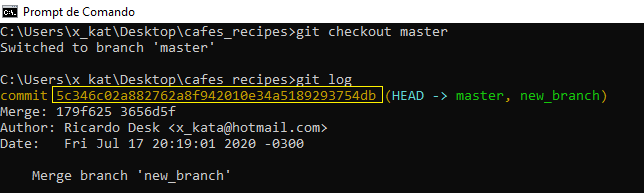


Sempre que você vê essa mensagem na tela, é o Git se gabando de poder poupar alguns objetos no banco de dados de objetos e tornar o histórico do seu projeto menos complicado. Bom Git.

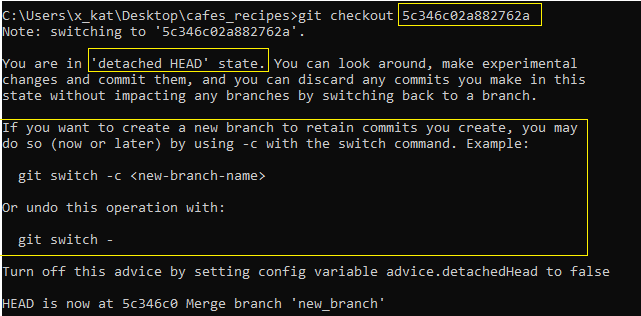


**HEAD desanexada**

O segundo caso é um recurso que se mostra bastante útil na prática. Vou começar dando um checkout no mestre e esquecer o ramo new\_branch por um tempo.

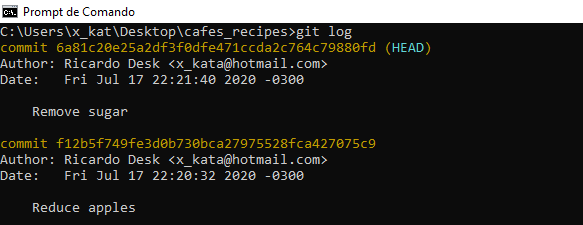


Até agora, eu sempre lhe disse que HEAD é uma referência a um ramo, que por sua vez é uma referência a um commit. Quando você faz check-out de uma ramificação, isso significa que você está alterando HEAD. No entanto, você também pode fazer algo diferente. Você pode fazer checkout diretamente um commit em vez de um branch. Vou fazer checkout este commit utilizando o SHA‑1 do commit.



Agora, se você olhar dentro do HEAD, ele não está apontando para uma ramificação, está apontando diretamente para uma confirmação e, de fato, não há nenhuma ramificação atual. Essa é uma situação chamada HEAD desanexada.

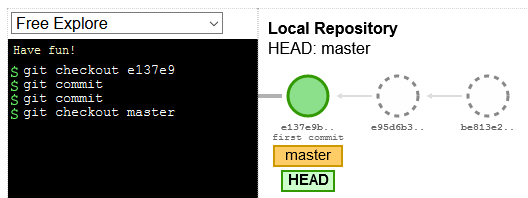
Agora, vamos fazer algumas alterações na receita de torta de maçã, algo que não tenho certeza se quero manter. Reduzimmos o número de maçãs e confirmamos. Depois altermos novamente a receita agora fazer a torta sem açúcar e outro commit.



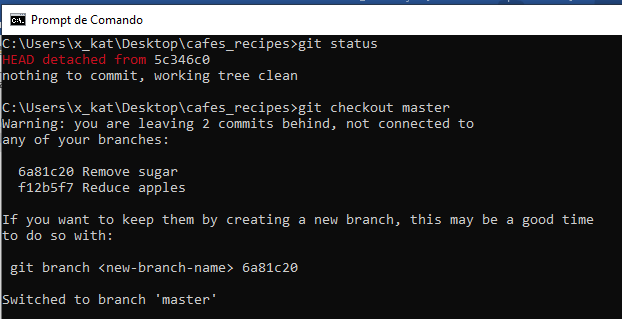
Como não há ramificação atual, ele rastreará a confirmação mais recente movendo HEAD diretamente. HEAD está funcionando exatamente como um ramo aqui.



Ao abandonar o experimento e fazer checkout no mestre novamente, o HEAD volta ao seu lugar no ramo principal. O mesmo acontece com nossos arquivos. Mas o que acontece com esses commits?

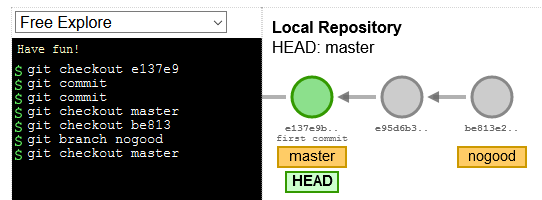


Bem, eles ainda estão no banco de dados de objetos em algum lugar junto com todas as suas árvores e blobs, mas, a menos que eu tenha anotado seus SHA-1s, esses commits e seus objetos conectados agora estão inacessíveis. Eles não podem ser alcançados iniciando-se por uma ramificação ou tag e caminhando pelos objetos no banco de dados. Só posso alcançá-los diretamente pelos seus SHA-1s e sou obrigado a esquecê-los em breve.



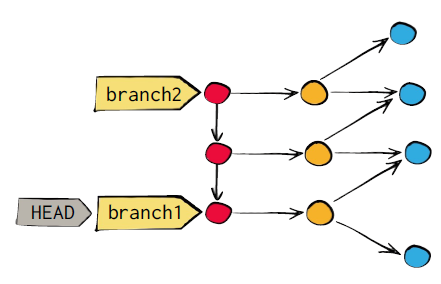
De vez em quando, no decorrer de outras operações, o Git decide que vale a pena executar uma coleta de lixo. O coletor de lixo procurará objetos no banco de dados que não possam ser alcançados por meio de uma ramificação, HEAD ou marca e os removerá para economizar espaço em disco. Lembre-se, cada objeto é apenas um arquivo no banco de dados, portanto, removê-los é tão fácil quanto excluir esses arquivos. Portanto, esses commits que eu criei provavelmente permanecerão no banco de dados por algum tempo e desaparecerão.

Se eu quiser salvá-los, uma coisa que posso fazer é voltar para os commits perdidos. Ainda posso fazer isso porque tenho seus SHA-1s aqui e o coletor de lixo ainda não foi executado. Primeiro, podemos fazer o checkout do commit utilizando o SHA-1 e mover o HEAD até o commit. E agora que estamos no commit, podemos colocar um banch nele, poe exemplo, um chamado nogood. Por fim, podemos fazer o checkout do mestre novamente, e desta vez os commits estão seguros.



Agora existe uma ramificação que atua como ponto de entrada para esta seção do gráfico de objetos, portanto esses objetos nunca serão coletados como lixo. E eu posso facilmente voltar a eles através do ramo criado nogood. Essa é uma maneira comum de usar um HEAD desanexado.Você pode desanexar HEAD, fazer seu experimento, ainda confirmar o experimento o quanto quiser e, para não perder dados, lembre-se de colocar um ramo no que você gosta antes de deixá-lo para trás.

Agora temos uma imagem melhor da natureza do Git, vamos recapitular. Um repositório Git é um monte de objetos vinculados entre si em um gráfico. Como você sabe, eles podem ser confirmações, blobs, árvores ou tags. Depois, existem ramificações que são referências a uma confirmação. E, finalmente, há HEAD. Isso também é uma referência, mas existe apenas uma e marca nossa posição atual no gráfico. Geralmente está apontando para um ramo, mas também pode ser desanexado e apontando diretamente para um commit.

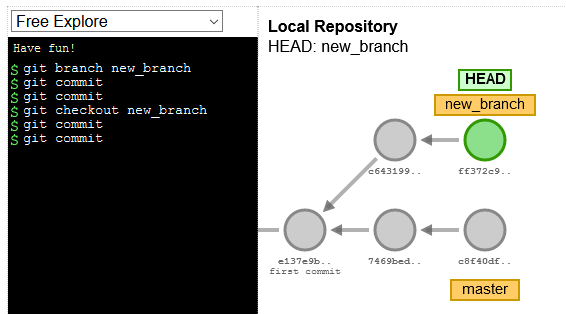


Existem algumas regras.

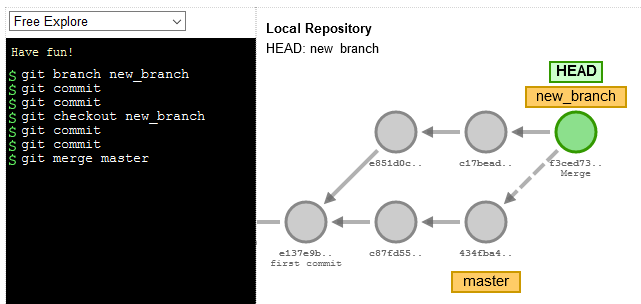
1. A ramificação atual controla novas confirmações. Portanto, se você criar um novo commit dizendo, git commit ou git merge, por exemplo, o ramo atual será movido para o novo commit. Se você estiver no estado HEAD desanexado, o próprio HEAD será movido para o novo commit.
2. Seu diretório de trabalho é atualizado automaticamente. Quando você muda para uma confirmação, por exemplo, com git checkout, o Git substitui o conteúdo do seu diretório de trabalho pelo conteúdo que pode ser alcançado a partir dessa confirmação.
3. Qualquer confirmação, blob ou árvore que não possa ser alcançada a partir de um ramo, HEAD ou uma marca, é considerada morta e pode ser coletada como lixo.

**GIT REBASE**

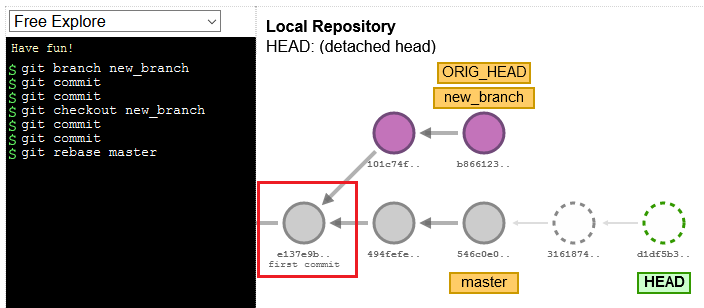
Suponhamos que existem dois ramos, o ramo principal e o ramo new\_branch recebeu 2 novos commits cada um como abaixo:



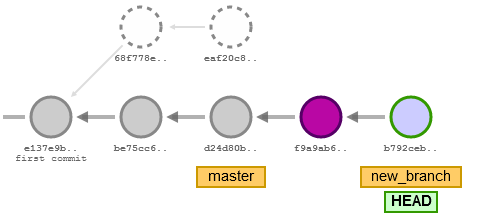
Temos dois ramos que divergiram mas queremos juntar o conteúdo dos dois ramos. Poderímos mesclar os dois ramos onde teríamos um novo commit e os pais desse novo commit seriam as os doid commits anteriores dos dois ramos. Além disso, o ramo atual passaria para esse novo commit mesclado.



Em vez disso, usarei outra maneira de juntar os dois ramos. Vou refazer a ramificação atual sobre a outra ramificação. Se refazermos o new\_branch no mestre, eis o que acontece. O Git procura o primeiro commit no new\_branch, que também é um commit no master. O commit que é a base do ramo de new\_branch.



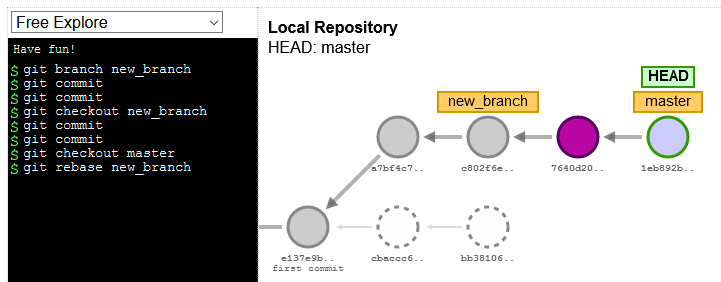
Todo o histórico antes desse commit já é compartilhado entre os dois ramos. Agora, o Git desanexa todo o ramo de new\_branch desse commit e o move sobre o master para que ele mude a base desse ramo. É por isso que é chamado de rebase. Como em uma mesclagem, talvez tenhamos que resolver conflitos para concluir a reformulação, mas nesse caso, não há conflitos. As duas ramificações alteram arquivos diferentes, então já terminamos. Agora, o ramo de new\_branch contém todos os commits do ramo mestre mais o material de new\_branch, que é o que queríamos.



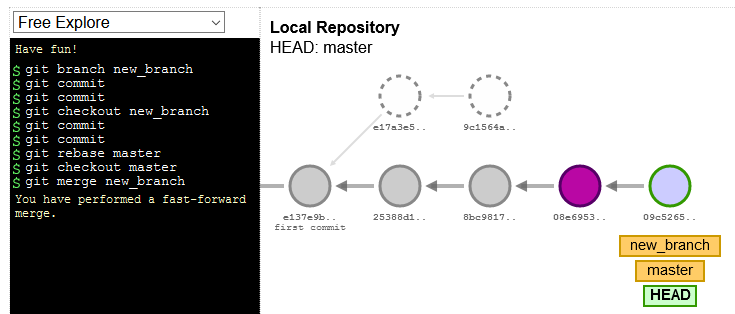
Quando você faz rebase, o Git desanexa o ramo atual de sua base e o move para o topo do ramo de destino. O Git faz cópias das confirmações. Ele cria novos commit com os mesmos dados, na verdade exatamente os mesmos dados, exceto para os pais. Portanto, essas novas confirmações se parecem quase exatamente com as confirmações originais, mas são novos objetos com novos SHA-1s. Portanto, eles são novos arquivos com novos nomes no diretório do banco de dados.

E, finalmente, o Git move o ramo rebased para os novos commits, deixando os antigos commits para trás. Algumas vezes as rebase podem ser complicadas e você pode evitar confusão se lembrar que rebasear é uma operação que cria novas confirmações.

O que acontece se queremos que funcione da outra maneira e queremos que as coisas sejam new\_branch no ramo principal. Assim como em uma mesclagem, podemos fazer o checkout do mestre e refazer o caminho para o outro lado.



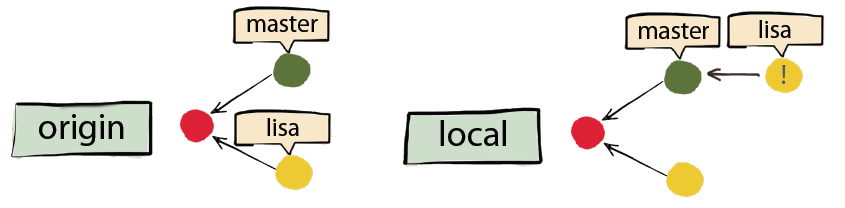
E agora vamos nos refazer. Na verdade, nesse caso em particular, eu poderia me refazer ou mesclar e isso não faria diferença alguma. Nos dois casos, o Git pode fazer o fast foward da ramificação. Uma rebase pode ser fast foward como uma mesclagem.



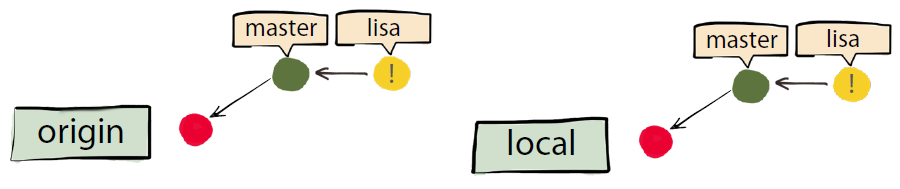
Então é isso que temos agora. Assim como em uma mesclagem, temos todos os commits que lidam com o new\_branch e todos os commits que lidam com a torta na mesma história. No entanto, diferente de uma mesclagem, obtivemos esse resultado não permitindo que vários ramos fiquem em paralelo, mas reorganizando os ramos para que pareçam um único ramo.

**Revendo Rebase**

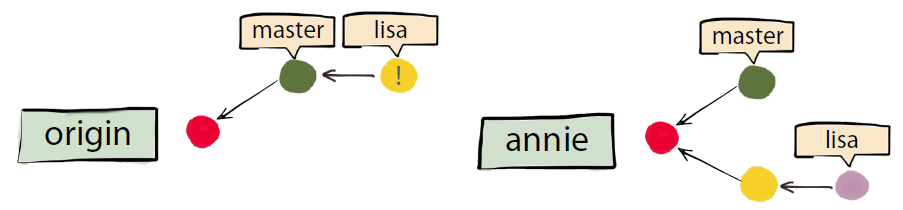
Existem alguns casos em que os rebasing não funcionam muito bem. Digamos que este repositório tenha sido clonado recentemente com dois ramos que estão rastreando ramos na origem. Estamos trabalhando no ramo lisa e decidimos fazer as alterações do mestre para o lisa. Você sabe que podemos fazer isso com uma mesclagem ou rebase. Então, vamos tentar a rebase desta vez. O Git copia o commit de lisa para que seu pai agora seja o último commit do master. No entanto, lembre-se de que esse novo commit amarelo (com um ponto de exclamação) que temos aqui não é o mesmo que o commit amarelo anterior. Em vez disso, é uma cópia, um objeto de banco de dados diferente.



O commit original será coletado em algum momento. Então agora temos um conflito novamente. Não podemos simplesmente fazer push porque temos histórias diferentes em nosso repositório local e na origem. Esse conflito em particular, podemos corrigi-lo facilmente com um force push ou um pull, seguido de um push. E então temos as mesmas coisas de origem que temos no local.



No entanto, outro colaborador, Annie, também está trabalhando no mesmo repositório e ainda tem o commit amarelo original sem marca de exclamação em nosso repositório. Não só isso, ela também continuou trabalhando no ramo lisa. Ela adicionou um commit lá. Então agora Annie tem um conflito bastante desagradável para resolver da próxima vez que sincronizar com a origem.

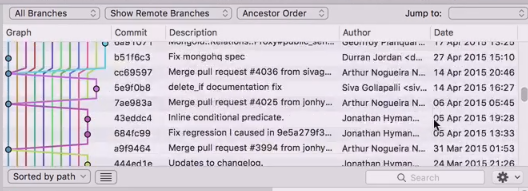


Ela precisa entender o que aconteceu primeiro e depois resolver os conflitos, mesmo que não os tenha causado. Há boas chances de que, mesmo depois de resolver os conflitos, ela tenha um histórico confuso que inclua os dois commits em amarelo, mesmo que pareçam exatamente iguais. Portanto, este é o ponto principal quando se trata de rebasear: como regra geral, nunca rebase coisas que foram compartilhadas com algum outro repositório. Não há problema em fazer rebase nas confirmações que você ainda não compartilhou, em seu repositório local. Mas lembre-se de que é fácil fazer rebase em confirmações compartilhadas por engano e ocorrer problemas.

**Os Trade-offs do Merges e Rebases**

Agora, sabemos como é uma rebase e como ela realmente funciona. No entanto, a rebasing e a merging parecem fazer algo muito semelhante. Ambos registram confirmações existentes no histórico de um branch, ma ambos têm diferentes vantagens e desvantagens. O ponto principal do merge é que preserva a história exatamente como aconteceu. Por exemplo, você pode ver claramente que as confirmações sendo criadas independentemente, possivelmente em paralelo, e depois foram mescladas em uma única linha do tempo. Se houver algum conflito durante a mesclagem, essa consolidação incluirá correções nos conflitos.

Mas o merge se torna um pouco menos simples quando você está olhando para um grande projeto em que é muito usado. Por exemplo, neste projeto Git para uma popular biblioteca de código aberto no Ruby abaixo.



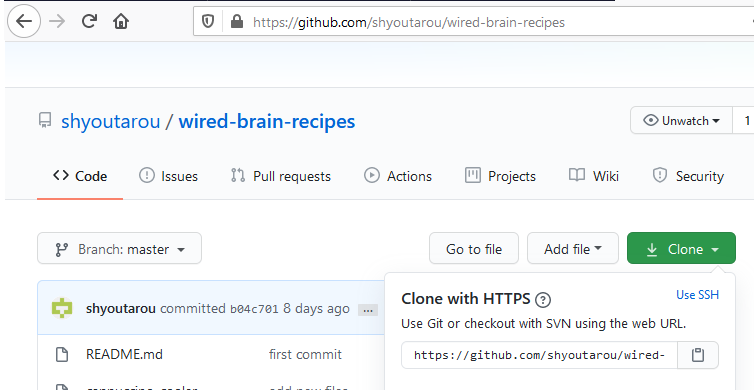
Estou usando uma ferramenta chamada Source Tree para visualizar o histórico do Git. Como você pode ver, há muitas ramificações e merges acontecendo, portanto pode ser muito difícil seguir o caminho em que todos esses ramos divergiram e depois convergiram novamente. Comparando esta ferramenta gráfica com o comando git log, em um projeto como este, o log do git pode ser enganoso, pois mostra o histórico como se fosse uma única linha do tempo, uma sequnência de confirmação após a outra, mas não é assim que o histórico do projeto realmente se parece. O log está esmagando a história real de alguma forma, intercalando commits não relacionados de diferentes ramos, como se estivessem conectados um ao outro enquanto não estão. Portanto, os merges preservam o histórico do projeto e, em geral, isso é bom, exceto que o histórico do projeto pode ser feio e confuso. Mas uma coisa é certa: a merge nunca mente.

No rebasing as confirmações já estão organizadas em uma única linha do tempo. Portanto, um projeto que utiliza muitas reformulações geralmente parece mais otimizado e limpo do que um projeto que utiliza muitas mesclagens no histórico. No entanto, essa história bem projetada não é real. Foi forjada por um rebasing, que é uma operação destrutiva. Pode dar a falsa impressão de que os commits foram criados primeiro em relação a outros commits. Mas na realidade foram criados em paralelo em diferentes ramos. Existem alguns comandos git avançados, por exemplo, que se tornam menos úteis se você violar o histórico do projeto. Em caso de dúvida, basta mesclar. Rebasing é bastante útil, mas você só deve usá-lo se souber o que está fazendo e entender as consequências.

**Clonar da Origem Remota**

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| git clone http: /// | Clone seu repositório do github |
| git clone source destination\_dir | Clona o repositório Git na origem para o destination\_dir |
| git clone –bare source destination\_dir "message" SHA1id | Clona o diretório bare do diretório de origem no destination\_dir |

A próxima coisa que precisamos fazer é baixar essas alterações nos outros computadores para que ele possa trabalhar nesse projeto usando seus computadores pessoais e de trabalho. Se navegarmos para a guia repositórios no GitHub clicarmos no projeto wired-brain-recipes, veremos uma lista de arquivos e conteúdo dentro do projeto. Para clonar esses repositórios em outro computador, você pode clicar no botão clone verde localizado no canto superior direito da página.

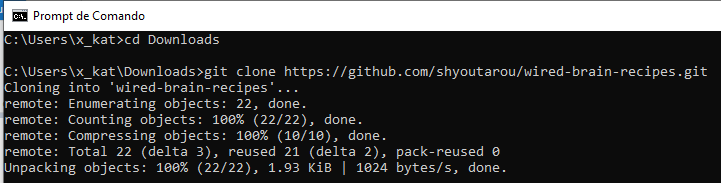


Podemos copiar esse https vinculado clicando n o ícone da área de transferência ao lado da Web URL

**https://github.com/wired-brain-coffee/wired-brain-recipes.git**

Com essa URL, podemos clonar o projeto simplesmente navegando para o diretório que iremos armazenar o projeto e, em seguida, cloná-lo nesse diretório. Vamos cloná-lo então para nossa pasta de Downloads, digite:

**git clone https://github.com/shyoutarou/wired-brain-recipes.git**

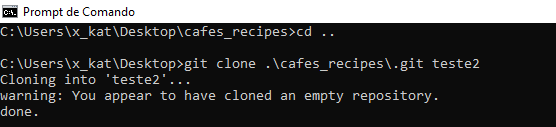


Ao navegarmos para este novo diretório vamos e digitar git status para ver se estamos atualizados com o origin master e, em seguida, digitar o comando “git ls-files” nos mostra a lista completa de arquivos em nosso projeto que foi retirado do origin master. Ele não copia literalmente todos os arquivos. Por exemplo, nas versões recentes do Git, o git clone copia apenas uma ramificação, a ramificação principal. Se eu quiser trabalhar com as outras ramificações no repositório remoto, preciso fornecer comandos específicos para isso.

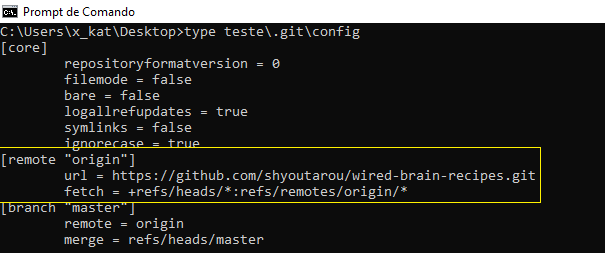
O Git não é como o Subversion ou outros sistemas tradicionais de controle de revisão que precisam de um servidor centralizado e todo mundo está apenas conversando com esse servidor. Em vez disso, agora os dois computadores contêm todo o projeto e seu histórico. Poderíamos ter quantos desses clones quisermos sincronizando uns com os outros. Obviamente, você ainda pode decidir que um clone específico é o mais importante. Por exemplo, se você tiver vários desenvolvedores trabalhando no mesmo projeto de software, provavelmente decidirá que o repositório no GitHub é o repositório de referência, o qual você constrói os lançamentos e todos devem sincronizar com esse. Por isso, desenhei o repositório GitHub no topo. Ainda é possível sincronizar os repositórios de desenvolvedores diretamente entre si, mas mesmo assim você provavelmente deseja nomear uma cópia de referência conhecida com a qual todos sincronizam. No entanto, no Git, isso não é um problema técnico. Agora, podemos seguir as mesmas etapas para clonar seu projeto em outros computadores.

É possível clonar através de um outro diretório local na mesma máquina:

**git clone .\cafes\_recipes\.git teste2**



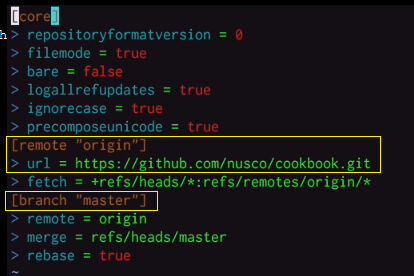
Se abrir o arquivo de configuração do teste2 svemos uma configuração de uma origem, e como remote é só um texto de configuração pode apontar para onde quiser.



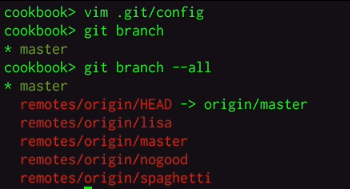
Se preferir que terceiros não tem acesso ao seu código fonte e dos clientes é possível montar uma instalação privada de GitLab.

**Local and Remote**

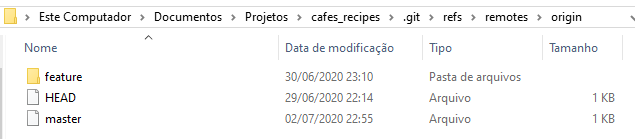
Agora, temos o mesmo projeto em dois repositórios separados. Quando emitimos o comando git clone, o Git adicionou algumas linhas à configuração do nosso repositório. Cada repositório Git pode lembrar informações sobre outras cópias do mesmo repositório. Cada outra cópia é chamada de controle remoto. Você pode definir quantos controles remotos quiser, mas quando você clona o projeto, o Git define imediatamente um controle remoto padrão e o chama com um nome convencional: origin. Aqui está a configuração de origem e aponta para a URL da qual clonamos o projeto. Apenas saiba que a configuração padrão diz que temos uma ramificação principal que mapeia a ramificação principal do controle remoto.



Então agora o Git se lembra de quais outros repositórios ou repositórios queremos sincronizar. Mas, para sincronizar, o Git também precisa saber o estado atual de origem, quais ramificações existem no controle remoto, que confirma essas ramificações atualmente apontadas, e assim por diante. E, de fato, o Git armazena essas informações. Se solicitarmos filiais, ele mostrará apenas as filiais locais. Agora só temos mestre. Mas se você listar as ramificações com a opção –all, verá todas as referências, incluindo as do controle remoto, as ramificações remotas e a posição atual da cabeça.



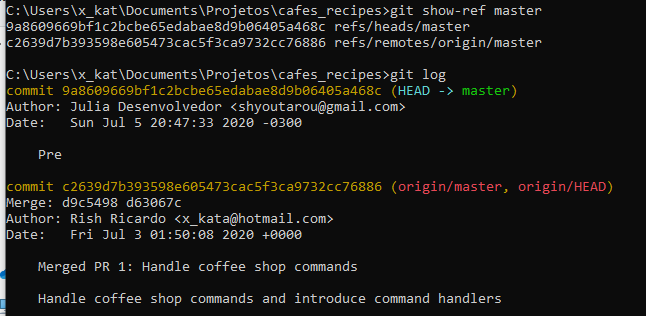
O Git rastreia ramificações remotas exatamente como rastreia ramificações locais, escrevendo essas ramificações como referências na pasta refs. Se você olhar dentro dessa pasta, verá uma pasta de origem aqui que contém as referências a ramificações, tags e o ponteiro de origem atual.



O Git atualizará automaticamente essas informações quando nos conectarmos a um controle remoto. Há uma ruga aqui. Se você olhar dentro desta pasta, poderá achar que alguns dos ramos estão ausentes. Nesse caso, só consigo ver os controles remotos HEAD aqui, não os ramos. Isso é devido a uma otimização de baixo nível no Git. Para evitar manter um arquivo pequeno para cada ramificação, o Git às vezes compacta alguns deles em um único arquivo chamado **packed-refs**.

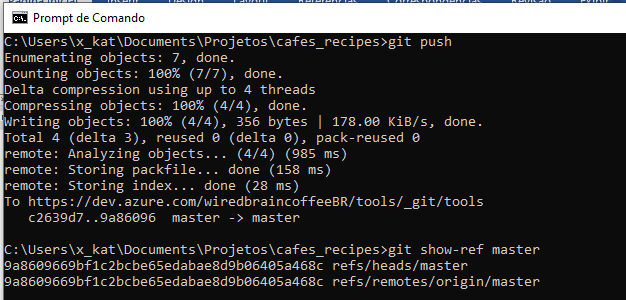
Não há um comando simples para descompactar esses arquivos; portanto, você terá que acreditar que os ramos que não estão no diretório refs devem estar nesse arquivo. Isso pode acontecer para ramificações locais e remotas, mas em ambos os casos, se as ramificações ainda são arquivos individuais ou empacotadas em pacotes-refs, elas ainda são conceitualmente a mesma coisa. Todas as ramificações, locais ou remotas, ainda são referências a uma confirmação e o Git rastreia todas elas. Como não podemos espiar dentro dos arquivos algumas dessas ramificações porque elas foram compactadas, vamos usar este comando de canalização, git show-ref, para ver quais confirmações estão apontando.

**git show-ref master**



O git show-ref master lista todos os ramos que possuem mestre em seus nomes, o que significa o ramo mestre local e o ramo mestre remoto. E, como você pode ver, eles estão apontando para o mesmo commit. Portanto, você sabe que a ramificação local no Git é apenas uma referência a uma confirmação. Bem, um ramo remoto é exatamente a mesma coisa. Sempre que você sincroniza com um controle remoto, o Git atualiza ramificações remotas.

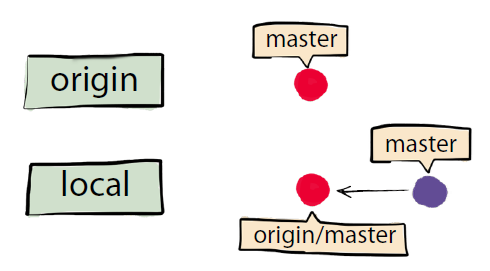
Então, vamos sincronizar, enviando os novos objetos e a ramificação atualizada para a origem com o comando git push.



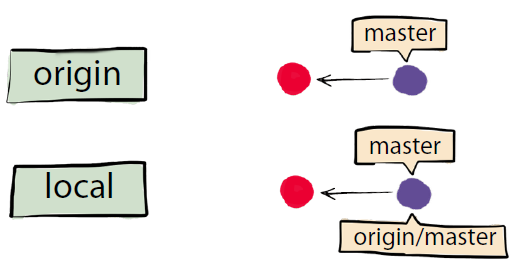
Aqui estamos, agora nossos novos objetos foram transferidos para o controle remoto e as ramificações na origem foram movidas para apontar para o último commit. Podemos verificar isso facilmente porque o Git atualizou nossas ramificações remotas para se alinhar ao estado de origem atual.

**Pulling**

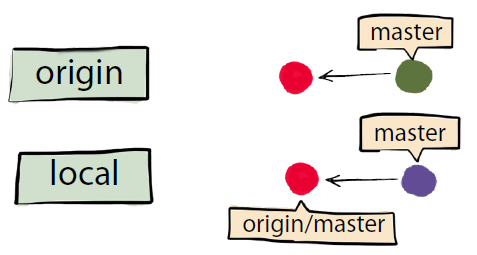
Agora não podemos apenas escrever alterações no controle remoto, também devemos ler as alterações no controle remoto. Imagine que temos um repositório remoto com um único commit. Quando clonamos esse repositório, obtemos os mesmos objetos em nosso cliente local, e aqui estão os ramos (em vermelho).



Agora, digamos que adicionamos um commit (em roxo) e fazemos push. Se não houver alterações na ramificação principal do controle remoto, as coisas serão fáceis. O Git copia nossa nova confirmação e os objetos associados para o controle remoto e atualiza a ramificação mestre do controle remoto para apontar para a nova confirmação, e também reflete a alteração nas ramificações na origem, atualizando a ramificação mestre de origem no repositório local.

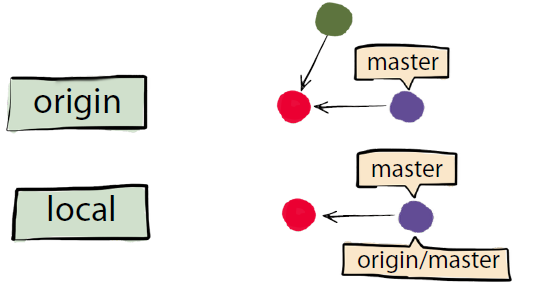


Agora imagine que tenhamos adicionado uma confirmação e nos preparamos para enviar, mas desta vez outra pessoa tenha enviado antes e colocado outro commit no controle remoto. Agora não podemos simplesmente fazer o push, pois temos um conflito que precisam ser reconciliado.



Nesse caso, basicamente temos duas opções. Uma opção, que eu não recomendaria, exceto em casos muito especiais, é forçar um push. Podemos fazer isso com a opção de linha de comando:

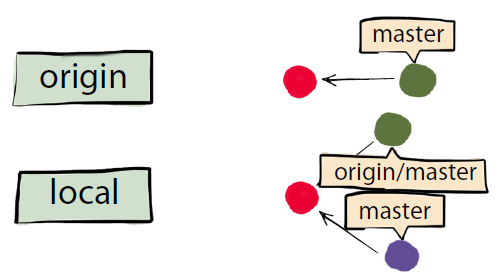
**git push –f**



Isso significa que estamos forçando o controle remoto a pegar nossos novos objetos e alterar seu histórico para corresponder ao histórico local. Então, provavelmente estamos perdendo dados de origem. Aqui estamos perdendo o commit verde. Nenhum ramo está apontando mais para esse commit, portanto ele será coletado como lixo eventualmente. Também estamos criando uma situação muito confusa para todas as outras pessoas sincronizando com o mesmo controle remoto, porque agora o histórico local estará em conflito com o histórico de origem, portanto, provavelmente forçar um push não é uma boa ideia.

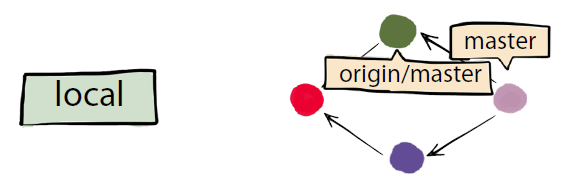
A outra opçãp é corrigir o conflito em nossa própria máquina antes de avançarmos. Para fazer isso, precisamos primeiro buscar os dados do controle remoto. Existe um comando para fazer isso chamado git fetch.

**git fetch**

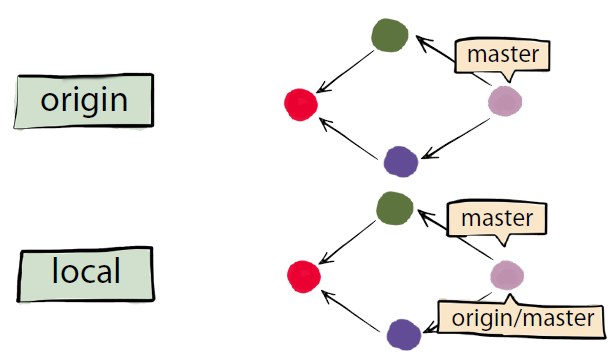


Obtemos os novos objetos do controle remoto e também atualizamos a posição atual das ramificações remotas, como de costume. Agora que temos o novo commit e os objetos relacionados, podemos mesclar nossas alterações locais com o histórico remoto. Então, fizemos a busca, agora fazemos uma mesclagem.

**git merge**



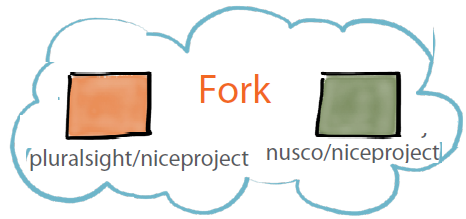
É claro que, durante o merge, talvez tenhamos que corrigir conflitos de merge e similares, mas o ponto importante aqui é que não estamos reescrevendo a história. Mesclagens nunca fazem isso. Em vez disso, eles apenas adicionam novos objetos. Então, quando fazemos a mesclagem, nossa história é a história do controle remoto e mais algumas coisas, e podemos enviar essas coisas novas para o controle remoto sem reescrever o histórico do controle remoto.



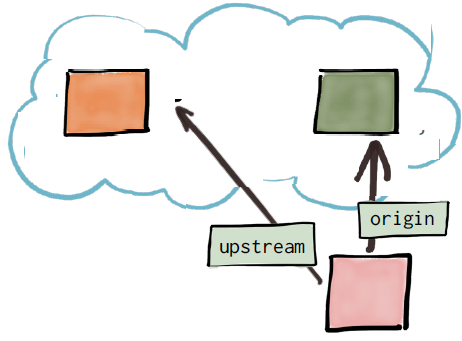
É isso que você faz na maioria das vezes. Você obtém as alterações no controle remoto, as mescla em seu próprio repositório e pressiona o resultado. Essa sequência de git fetch, seguida de git merge, é tão comum que existe um único comando que faz as duas coisas chamado, git pull.

**Multiple Remotes (Forks e Pull Request)**

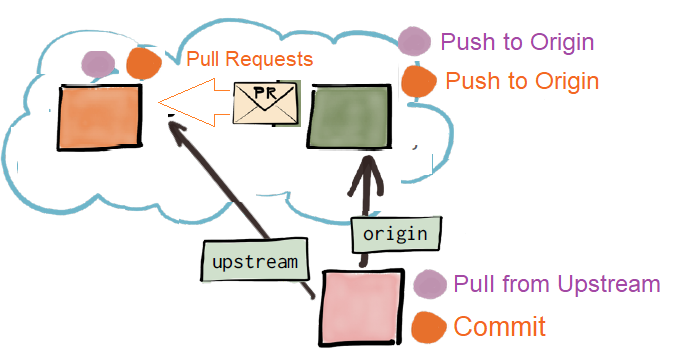
Imagine que existe este projeto no GitHub com o qual queremos contribuir. Pertence a um usuário chamado Pluralsight. Poderíamos simplesmente clonar esse projeto, mas ele ficaria preso em nossa máquina local porque não temos acesso de gravação ao repositório do Pluralsight, portanto não podemos avançar nele. O que podemos fazer a partir da interface da web do GitHub é criar nossa própria cópia do projeto no GitHub. Isso é chamado de fork. Um fork é como um clone, mas é um clone remoto. Estamos clonando o projeto da conta do GitHub de outra pessoa para a nossa própria conta do GitHub.



Portanto, agora temos um novo projeto na nuvem e podemos cloná-lo em nossa máquina local. Quando fazemos isso, o Git cria um controle remoto em nosso repositório local apontando a origem. A origem está apontando para o nosso próprio projeto GitHub, não o projeto original. Na verdade, do ponto de vista do Git, não há nenhuma conexão entre o nosso projeto e o projeto original do qual partimos. O GitHub sabe que os dois projetos estão conectados, mas o Git não. Portanto, se quisermos acompanhar as alterações no projeto original, precisamos adicionar outro controle remoto apontando para ele.



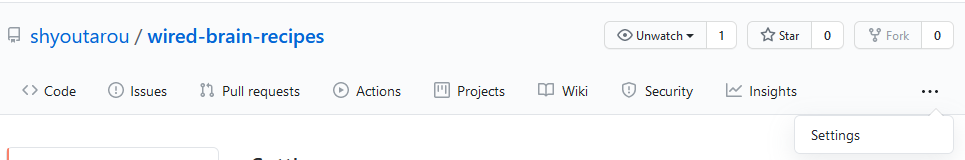
Isso não é algo que o Git faz automaticamente. Nós temos que fazer isso sozinhos. Uma convenção comum é chamar esse remoto upstream. Agora temos nosso projeto local com vários controles remotos. Podemos trabalhar nisso e podemos sincronizar todas as nossas alterações locais com a origem. Se confirmarmos alterações locais, podemos simplesmente enviar essas alterações para a origem. Se houver mudanças no upstream, podemos atraí-las para o nosso projeto local, resolver qualquer conflito e enviá-las à origem. Uma coisa que ainda não podemos fazer, no entanto, é levar as mudanças a montante. Por exemplo, gostaríamos de contribuir com nosso commit laranja para o projeto original, mas ainda não temos acesso de gravação ao upstream, portanto o GitHub nos oferece uma alternativa. Podemos enviar uma mensagem aos mantenedores do upstream e pedir que eles façam nossas alterações.



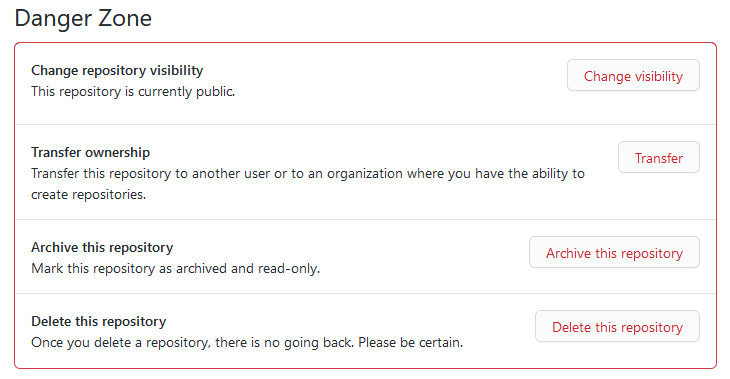
É uma pull request. Mais uma vez, as solicitações pull não são um recurso do Git. Nem sequer são um recurso de controle de versão. De certa forma, eles são um recurso de rede social. Você está apenas enviando uma mensagem para as pessoas. Se essas pessoas gostam de suas alterações, o GitHub facilita a execução remota e a obtenção das alterações desde a origem. E isso é tudo sobre forks e solicitações pull, duas das ferramentas mais importantes no desenvolvimento moderno de código aberto.

**Excluir Repositório**

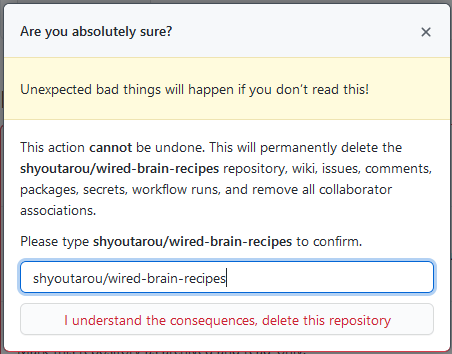
1. No GitHub Enterprise, navegue até a página principal do repositório.
2. Sob o nome do seu repositório, clique em Configurações.



1. Em Zona de perigo, clique em Excluir este repositório.



1. Leia os avisos. Para verificar se você está excluindo o repositório correto, digite o nome do repositório que deseja excluir. Por fim, clique em Eu entendo as consequências, exclua este repositório.



Ao tentar enviar alterações ao repositório remoto agora:

