Entendendo o que é o Git e qual sua importância

O Git é um sistema de controle de versão de código aberto e distribuído, criado em 2005 por Linus Torvalds, o mesmo criador do kernel do Linux. Ele é projetado para gerenciar projetos de software com várias pessoas trabalhando neles, permitindo que os desenvolvedores trabalhem em diferentes partes do código simultaneamente e combinem seus trabalhos de forma colaborativa.

Com o Git, os desenvolvedores podem criar diferentes versões de um projeto e trabalhar em suas próprias cópias locais, mantendo um histórico completo de todas as mudanças realizadas ao longo do tempo. Isso torna mais fácil para eles acompanhar as alterações, rastrear bugs e corrigi-los. Além disso, o Git permite que os desenvolvedores colaborem facilmente uns com os outros, combinando seus trabalhos e resolvendo conflitos de forma automática ou manual, dependendo das necessidades.

A importância do Git está relacionada à sua capacidade de gerenciar projetos de software complexos e colaborativos. Com o Git, é possível trabalhar em projetos de software de qualquer tamanho, desde pequenos scripts até projetos complexos com milhares de arquivos e várias equipes trabalhando neles. Além disso, o Git torna mais fácil para os desenvolvedores colaborarem uns com os outros, permitindo que eles trabalhem de forma independente em diferentes partes do código e combinem seu trabalho de forma segura e eficiente.

Outra vantagem do Git é que ele é de código aberto, o que significa que qualquer pessoa pode usar e contribuir para ele. Isso torna o Git uma escolha popular entre desenvolvedores em todo o mundo e ajuda a impulsionar a inovação e a melhoria contínua do sistema. Em resumo, o Git é uma ferramenta essencial para o desenvolvimento de software moderno e é amplamente utilizado em todo o setor de tecnologia.

Comandos básicos e navegação CLI e instalação

Aqui estão alguns comandos Git que podem ser usados para manipular pastas e arquivos em um repositório Git:

cd: Este comando é usado para navegar pelas pastas. Ele é usado da mesma forma que em um terminal normal.

dir / ls: Este comando é usado para listar o conteúdo de uma pasta. Ele mostra os arquivos e pastas contidos na pasta atual.

mkdir: Este comando é usado para criar uma nova pasta no diretório atual. Por exemplo, para criar uma pasta chamada "meus_arquivos", você pode executar o ***comando mkdir meus_arquivos.***

touch: Este comando é usado para criar um novo arquivo vazio. Por exemplo, para criar um arquivo chamado "README.md", você pode executar o comando touch README.md.

del/rmdir: Este comando é usado para excluir um arquivo ou pasta. O comando rm com a opção -r pode ser usado para excluir uma pasta e todo o seu conteúdo. Por exemplo, para excluir a pasta "meus_arquivos", você pode executar o comando ***rm -r meus_arquivos***.

Tópicos fundamentais de Git

Aqui estão alguns tópicos fundamentais para entender o funcionamento do Git:

- **Controle de versão:** O Git é um sistema de controle de versão, o que significa que ele ajuda a gerenciar diferentes versões de um projeto ou arquivo. Ele permite que os usuários visualizem e comparem diferentes versões de um arquivo, voltem a versões anteriores e colaborem em equipe sem medo de perder dados.
- **Repositórios:** O Git armazena todo o histórico de um projeto em um repositório. Um repositório é basicamente um diretório onde o Git armazena todas as informações relevantes do projeto, incluindo arquivos, histórico de versão, branches e configurações.
- **Branches:** O Git permite que os usuários criem branches, ou seja, ramificações, do repositório principal. Isso permite que vários desenvolvedores trabalhem em diferentes partes do projeto simultaneamente, sem interferir no trabalho um do outro. As alterações em uma branch podem ser mescladas de volta ao branch principal quando estiverem prontas.
- **Commits:** Um commit é uma etapa fundamental do Git. Ele é usado para salvar as alterações feitas em um projeto em um determinado momento, juntamente com uma mensagem descritiva. Cada commit é salvo no histórico do projeto e pode ser visualizado e revertido posteriormente.
- **Push/pull:** O Git permite que os usuários enviem (push) as alterações feitas em um projeto para um repositório remoto e baixem (pull) as alterações feitas por outros desenvolvedores no mesmo projeto. Isso é essencial para trabalhar em equipe e manter o projeto atualizado.
- **Resolução de conflitos:** Quando várias pessoas trabalham em um projeto, é possível que haja conflitos entre as diferentes alterações feitas. O Git fornece ferramentas para resolver esses conflitos de forma eficiente e sem perder dados importantes.

Esses são alguns dos tópicos fundamentais para entender o funcionamento do Git.

É importante também entender conceitos como diferenças (diffs), tags, submódulos e o fluxo de trabalho do Git, que ajudam a usar a ferramenta de forma mais eficiente e produtiva.

- **SHA1:** SHA1 (Secure Hash Algorithm 1) é um algoritmo criptográfico que é amplamente utilizado pelo Git para identificar e garantir a integridade de objetos e commits no repositório. SHA1 produz uma sequência de 40 caracteres hexadecimais que representa unicamente uma versão específica do objeto ou commit. O Git usa essa sequência para verificar se há corrupção em objetos ou commits, garantindo a segurança e a integridade do projeto.
- **Objetos fundamentais:** O Git armazena todo o histórico de um projeto em um formato de banco de dados. Esse banco de dados é composto de quatro tipos de objetos fundamentais: blobs, árvores, commits e tags. Cada objeto é identificado por um SHA1 exclusivo e contém informações sobre o projeto em diferentes níveis de granularidade. Por exemplo, um blob é um objeto que representa um arquivo específico em um projeto, enquanto uma árvore é um objeto que representa uma coleção de arquivos e subdiretórios.
- **Sistema distribuído:** O Git é um sistema de controle de versão distribuído, o que significa que não há um único servidor centralizado que mantém o histórico completo do projeto. Em vez disso, cada cópia local do repositório Git contém o histórico completo do projeto. Isso permite que os usuários trabalhem de forma independente em diferentes cópias do

repositório e, em seguida, sincronizem suas alterações em um momento posterior. Como resultado, o Git oferece maior flexibilidade e autonomia para os usuários em comparação com sistemas de controle de versão centralizados.

- **Por que o Git é seguro:** O Git é considerado um sistema de controle de versão seguro por várias razões. Em primeiro lugar, o SHA1 é usado para identificar e garantir a integridade de objetos e commits, como mencionado anteriormente. Além disso, o Git permite que os usuários criptografem o tráfego entre repositórios usando o protocolo SSH ou HTTPS. O Git também fornece ferramentas para gerenciar permissões de acesso a repositórios, rastrear mudanças e monitorar o uso do sistema. Por fim, o Git oferece uma rede de segurança adicional, permitindo que os usuários restaurem o projeto em caso de perda de dados, corrupção ou outros problemas.

Objetos internos do Git

O Git armazena todo o histórico de um projeto em um formato de banco de dados composto de objetos internos. Esses objetos são os blocos de construção fundamentais do sistema de controle de versão Git e representam diferentes níveis de granularidade dos arquivos e diretórios no repositório.

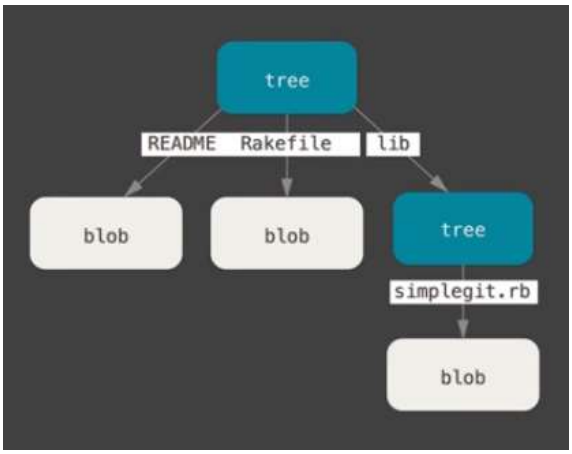
Os quatro tipos principais de objetos internos do Git são:

1. **Blob:** Um objeto blob representa um arquivo no repositório. Ele contém apenas o conteúdo do arquivo, sem metadados – tipo de objeto, tamanho da string-, como nome de arquivo ou informações de data e hora. Cada blob é identificado por um hash SHA-1 exclusivo, que é calculado a partir do conteúdo do arquivo.

   echo ‘conteudo’ | git hash-object --stdin

   Ex: echo -e blob 9\0conteudo’ | openssl sha1

2. **Tree:** Um objeto tree é uma representação de um diretório no repositório. Ele contém uma lista de entradas que especificam o nome, o tipo e o hash SHA-1 do objeto correspondente para cada arquivo ou diretório no diretório atual. Cada tree é identificado por um hash SHA-1 exclusivo, que é calculado a partir do conteúdo das entradas.



3. **Commit:** Um objeto commit representa uma revisão ou alteração específica no repositório. Ele contém informações sobre o autor, o log de mensagens e um hash SHA-1 que aponta para o tree correspondente que representa o estado do diretório após a alteração. Cada commit é identificado por um hash SHA-1 exclusivo, que é calculado a partir do conteúdo das informações.

4. **Tag:** Um objeto tag é uma referência nomeada a um commit específico. Ele pode ser usado para marcar pontos importantes na história do repositório, como lançamentos ou versões. Cada tag contém o nome da tag, o hash SHA-1 do commit correspondente e informações adicionais, como autor e data. Cada tag é identificado por um hash SHA-1 exclusivo, que é calculado a partir do conteúdo das informações.

Esses objetos internos do Git são a base do histórico do repositório e são usados pelo Git para rastrear e gerenciar mudanças e versões no projeto.

Chave SSH e Tokens

Tanto a chave SSH quanto o token de acesso são usados para autenticar e autorizar o acesso a repositórios do Git.

A chave SSH é um par de chaves criptográficas, uma pública e uma privada, que são usadas para autenticar a conexão entre o computador do usuário e o servidor Git. A chave pública é armazenada no servidor Git e a chave privada é armazenada no computador do usuário. Quando o usuário se conecta ao servidor Git, a chave pública é usada para verificar se a chave privada correspondente é válida e, se for o caso, permitir o acesso ao repositório.

Já o token de acesso é uma cadeia de caracteres que é usada em vez de uma senha para autenticar o acesso ao Git. É gerado pelo provedor do serviço Git, como o GitHub, e pode ser usado em vez de uma senha para acessar o repositório de forma segura. O token de acesso pode ser gerado com diferentes níveis de permissão, permitindo que o usuário acesse o repositório com diferentes níveis de autorização, como leitura, gravação ou administração.

Ambas as opções, chave SSH e token de acesso, são opções seguras para autenticar o acesso ao Git e evitar a necessidade de digitar senhas, o que pode ser um risco de segurança. É importante seguir as melhores práticas de segurança ao usar esses recursos, como manter a chave SSH privada em um local seguro e não compartilhar o token de acesso com outras pessoas.

Gerando chave SSH

Para gerar uma chave SSH para autenticação no Git, você pode seguir os seguintes passos:

1. **Abra o terminal em seu sistema operacional.**
2. **Execute o comando ssh-keygen seguido pelo nome da chave que deseja usar. Por padrão, a chave é salva em ~/.ssh/id_rsa. Você pode usar outro nome, se preferir:**

```
$ ssh-keygen -t ed25519 -C geovane.duarte@castanhal.ufpa.br
```

Esse código é um comando utilizado para gerar uma chave SSH usando o algoritmo de criptografia ed25519, que é uma alternativa mais moderna e considerada mais segura do que o algoritmo RSA.

Aqui está o que cada parte do comando significa:

- ssh-keygen: é o comando utilizado para gerar chaves SSH.
- -t ed25519: especifica o tipo de algoritmo de criptografia utilizado para gerar a chave. Neste caso, estamos usando o algoritmo ed25519.
- -c geovanneduarte9@gmail.com: é um comentário que pode ser adicionado à chave, para ajudá-lo a identificar a chave se tiver várias chaves. Neste caso, estamos adicionando o endereço de e-mail geovanneduarte9@gmail.com ao comentário.

3. **Quando solicitado, escolha um local para salvar a chave e insira uma senha, se desejar.**
4. **Copie a chave pública (com a terminação .pub) para a sua área de transferência, usando o comando:**

cd /c/Users/geova/.ssh

cat id_ed25519.pub ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIIxeb/bxnpjCNhG8YbUkHY/11q5U4+QqnPe/557XhB 1 geovane.duarte@castanhal.ufpa.br

5. **Adicione a chave pública ao seu provedor de hospedagem Git, como GitHub ou GitLab. O processo de adição da chave pode variar dependendo do provedor de hospedagem.**

Com a chave SSH configurada, você pode se autenticar em seu provedor de hospedagem Git sem precisar digitar sua senha a cada vez que fizer push ou pull. Além disso, as operações do Git serão executadas com mais segurança, pois a chave SSH é criptografada e protegida com uma senha, se você escolher usar uma. Após a criação da chave, é necessário inicializar o SSJ EJA (Entidade que engarregada de pegar as chaves e lidar com ela) com os comandos abaixo:

eval $(ssh-agent -s

O comando eval $(ssh-agent -s) inicia um novo processo do agente SSH (SSH agent) e define variáveis de ambiente para o shell atual, permitindo que o agente SSH seja usado para gerenciar chaves de autenticação SSH.

O agente SSH é um programa que executa em segundo plano e armazena suas chaves privadas de forma segura, permitindo que você use suas chaves privadas sem a necessidade de digitar suas senhas repetidamente. Quando você executa esse comando, ele retorna um PID (Process ID) para o agente SSH

recém-criado, que será usado para se comunicar com o agente SSH para autenticar suas chaves SSH.

ssh-add id_ed25519

O comando "ssh-add id_ed25519" adiciona a chave privada armazenada no arquivo "id_ed25519" ao agente SSH em execução no terminal. Isso permite que o agente SSH use a chave privada para autenticar o usuário em um servidor remoto sem que seja necessário inserir a senha novamente. Dessa forma, ao acessar um servidor remoto usando SSH, o agente SSH pode usar a chave privada para autenticar o usuário sem a necessidade de inserir a senha novamente, o que é especialmente útil quando se trabalha com vários servidores remotos.

Iniciando e criando um commit no git

Para iniciar um repositório Git em um diretório local e criar um commit, você pode seguir os seguintes passos:

- **Inicie um repositório Git em um diretório local usando o comando git init:**

git init

git config –global user.email email@email.email

git config –global user.name name pasta

- **Adicione os arquivos que deseja monitorar ao índice Git (também conhecido como staging area) usando o comando git add:**

git add <arquivo>

- **ou para adicionar todos os arquivos no diretório atual:**

git add .

- **Crie um commit com uma mensagem descritiva usando o comando git commit:**

git commit -m "Mensagem de commit descritiva aqui"

- **Isso criará um commit com as alterações no índice Git, usando a mensagem de commit fornecida.**
- **É importante notar que o commit não será enviado para um repositório remoto, apenas estará salvo localmente. Se você quiser enviar seus commits para um repositório remoto, você pode usar o comando git push para fazer isso.**

git push <repositório-remoto> <branch>

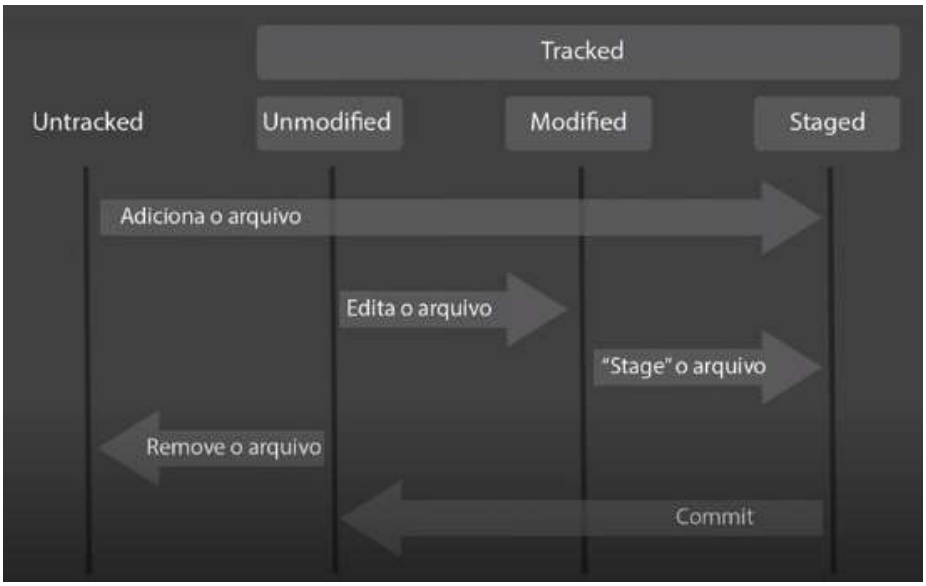
- **Isso enviará os commits para o repositório remoto especificado na branch especificada. Você precisará ter permissão para enviar commits para o repositório remoto.**

Ciclo de vida de arquivo no Git: Passo a passo

No Git, um arquivo é considerado "tracked" (rastreado) se ele já foi adicionado ao controle de versão. Isso significa que o Git está ciente das alterações feitas nesse arquivo e que ele será incluído no próximo commit. Um arquivo "tracked" pode ter uma das três condições possíveis: modificado, preparado ou confirmado.

O status do arquivo pode ser visto com o comando:
git status

- **Não modificado (Unmodified):** um arquivo unmodified é aquele que não sofreu alterações desde a última vez que foi confirmado (commited) no repositório Git. Em outras palavras, o arquivo foi adicionado ao repositório em um commit anterior e desde então não sofreu alterações
- **Modificado (Modified):** um arquivo é considerado modificado quando foi alterado desde o último commit. O Git rastreia essas modificações, mas o arquivo ainda não foi preparado para o próximo commit.
- **Preparado (Staged):** quando um arquivo modificado é adicionado ao "Index" (também conhecido como "Staging Area"), ele é considerado "preparado". A "Staging Area" é uma área intermediária onde você pode selecionar quais alterações serão incluídas no próximo commit.
- **Confirmado:** quando um arquivo é confirmado, significa que as alterações foram adicionadas ao histórico do Git e estão permanentemente registradas no repositório.



Por outro lado, um arquivo "untracked" (não rastreado) é aquele que não foi adicionado ao controle de versão. O Git não está ciente das alterações feitas nesse arquivo e ele não será incluído no próximo commit. Para adicionar um arquivo "untracked" ao controle de versão, você precisa primeiro adicioná-lo ao "Index" usando o comando "git add". A partir desse ponto, o arquivo será considerado "tracked" e poderá ser preparado e confirmado como explicado acima.

O ciclo de vida dos arquivos no Git pode ser dividido em três etapas e é armazenado em duas áreas:

1. O ambiente de desenvolvimento é o ambiente no qual os desenvolvedores trabalham, criando e modificando o código-fonte dos projetos. É nesse ambiente que os

programadores escrevem o código e executam testes antes de enviar suas mudanças para o servidor.

- **Remote repository**: é um repositório Git localizado em um servidor remoto, que pode ser acessado e gerenciado por várias pessoas. É um ponto central onde as alterações de código são compartilhadas entre diferentes colaboradores e equipes de desenvolvimento. Quando um desenvolvedor clona um repositório remoto, ele cria uma cópia local do repositório, que ele pode modificar e enviar de volta ao repositório remoto. Isso permite que vários desenvolvedores trabalhem em um projeto simultaneamente, enquanto mantêm um histórico completo de todas as alterações feitas no código. Alguns exemplos populares de serviços de hospedagem de repositórios remotos são o GitHub, GitLab e Bitbucket. Esses serviços oferecem recursos de gerenciamento de código e colaboração, como controle de versão, rastreamento de problemas, revisão de código e integração contínua.

2. Já o ambiente servidor é onde o código-fonte é armazenado e disponibilizado para uso e/ou distribuição. É nesse ambiente que o código-fonte é testado e integrado com outras partes do projeto antes de ser lançado para os usuários finais.
    - **Working Directory:** Nesta etapa, os arquivos estão em seu estado inicial e podem ser modificados. O Git ainda não sabe da existência desses arquivos e não os está rastreando.
    - **Staging Area:** Nesta etapa, os arquivos são adicionados à área de staging para serem preparados para o commit. O Git começa a rastrear as alterações nos arquivos e os prepara para serem salvos.
    - **Local repository:** Nesta etapa, os arquivos estão armazenados no repositório do Git como um novo commit. Eles são versionados e estão prontos para serem compartilhados com outros desenvolvedores.

O fluxo de trabalho típico com o Git segue os seguintes passos:

- Modificar os arquivos no diretório de trabalho.
- Adicionar os arquivos modificados à área de staging com o comando "git add".
- Confirmar as alterações com o comando "git commit".
- Repetir o processo sempre que houver novas alterações a serem versionadas.

Este processo permite que o Git mantenha um registro completo das mudanças nos arquivos ao longo do tempo, permitindo que desenvolvedores trabalhem em paralelo e possam colaborar de maneira mais eficiente.

Trabalhando com o git

O Github é uma plataforma de hospedagem de código-fonte baseada em git. Ele oferece controle de versão, gerenciamento de projetos, colaboração em equipe, integração contínua e entrega contínua. É amplamente utilizado por desenvolvedores de software para armazenar e compartilhar seus códigos fonte com outras pessoas.

Ao trabalhar com o Github, o desenvolvedor cria um repositório no Github que corresponde ao seu projeto local. Esse repositório remoto é usado para armazenar e compartilhar o código com outras pessoas. O desenvolvedor pode enviar (push) as alterações do seu repositório local para o repositório remoto e pode sincronizar as alterações do repositório remoto para o seu repositório local (pull) utilizando o comando abaixo.

git remote add origin >url do repositório remoto<

git push origin master

Além disso, o Github também oferece recursos para gerenciamento de projetos, como issues, milestones, pull requests e outras ferramentas para ajudar os desenvolvedores a trabalharem juntos em projetos maiores. Esses recursos permitem que os desenvolvedores colaborem e comuniquem facilmente uns com os outros.

- **Issues:** são problemas ou tarefas que precisam ser resolvidos no projeto. São geralmente utilizados para comunicar problemas, bugs ou solicitar novas funcionalidades.
- **Milestones:** são pontos de referência que representam um estágio no ciclo de vida do projeto. São usados para acompanhar o progresso do projeto e estabelecer metas. Um milestone pode ser definido para um período de tempo, um conjunto de recursos ou um marco específico.
- **Pull Requests:** são pedidos para que as mudanças sejam revisadas e incorporadas ao branch principal do repositório. Eles são utilizados quando um colaborador deseja contribuir com alterações no projeto, permitindo que outros membros revisem o código

antes de ser fundido ao código principal. É uma forma de colaboração entre desenvolvedores.

O Github é uma ferramenta essencial para desenvolvedores, especialmente aqueles que trabalham em projetos de código aberto. Ele torna mais fácil para as pessoas colaborarem e contribuírem para projetos de software de todo o mundo.

Como os conflitos acontecem no GitHub e como resolvê-los

Os conflitos no GitHub ocorrem quando dois ou mais colaboradores modificam a mesma parte de um arquivo e tentam mesclar as alterações em um único arquivo. Isso pode acontecer quando diferentes pessoas trabalham em diferentes ramos e tentam mesclar suas alterações no mesmo ramo principal.

Para resolver conflitos no GitHub, o processo é o seguinte:

1. **Identificar o conflito:** é necessário saber quais arquivos apresentam conflitos. O GitHub geralmente exibe um aviso na interface quando há um conflito.
2. **Abrir o arquivo conflitante:** É necessário abrir o arquivo e procurar as partes que apresentam conflito. O GitHub geralmente adiciona marcadores especiais para identificar as partes conflitantes.
3. **Analisar as alterações conflitantes:** é necessário entender as mudanças que foram feitas por cada colaborador e decidir como combiná-las.
4. **Resolver o conflito**: Depois de entender as mudanças conflitantes, o próximo passo é decidir qual versão deve ser mantida. O GitHub permite que o colaborador escolha quais alterações manter e quais descartar.

   git pull origin master

5. **Commit e push:** Depois de resolver o conflito, é necessário confirmar as alterações e empurrá-las para o repositório remoto.

Para evitar conflitos, é recomendável manter o repositório atualizado, comunicar as mudanças que foram feitas e trabalhar em ramos separados sempre que possível.

O conflito de merge é uma situação que ocorre quando duas ou mais alterações são feitas no mesmo arquivo ou conjunto de arquivos em diferentes branches do Git, e essas alterações são conflitantes, ou seja, o Git não consegue mesclá-las automaticamente. Isso pode acontecer quando uma mesma linha do arquivo é alterada em ambos os branches, ou quando duas alterações são feitas em partes adjacentes do mesmo arquivo.

Quando isso ocorre, o Git exibe uma mensagem de conflito de merge e interrompe a operação de merge. É então necessário resolver manualmente o conflito, editando o arquivo em questão para remover as partes conflitantes e manter apenas as alterações que devem ser mantidas. Após a resolução do conflito, o arquivo deve ser salvo e o commit finalizado para que o merge seja concluído com sucesso.