TIC em trilhas

Resumo do Capítulo 1 sobre Desenvolvimento Colaborativo com Git e GitHub

Desenvolver colaborativamente envolve o uso de ferramentas que facilitam e agilizam a experiência em um projeto. Uma das ferramentas mais úteis para isso é o **pull request**, que permite a validação e revisão de código antes de sua integração na branch principal.

Pull Requests

- **Criação de Pull Request**:
    1. Envie suas alterações para a branch remota.
    2. No GitHub, vá para 'Pull requests' e clique em 'Novo pull request'.
    3. Selecione a branch fonte e a branch destino.
    4. Solicite a revisão de outro colaborador.
- **Processo de Revisão**:
    1. O revisor analisa o código, testa e comenta sobre possíveis melhorias ou correções.
    2. O autor faz as alterações necessárias e envia novamente para revisão.
    3. Uma vez aprovado, o pull request é mesclado na branch principal e a branch fonte pode ser deletada.

Resolução de Conflitos

- **Tipos de Conflitos**:
    - **Current Changes**: Manter as mudanças atuais.
    - **Incoming Changes**: Manter as mudanças que estão vindo de outra branch.
    - **Both Changes**: Manter ambas as mudanças.
    - **Custom Changes**: Escrever manualmente o código necessário.
- **Deletar Branch Remota**: Use `git push origin -d <branch>` para deletar uma branch remota.
- **Atualizar Conexões de Branches**: Use `git remote update origin --prune` para podar branches que não possuem mais conexão entre o remoto e o local.

Dicas Importantes

1. **Revisão de Código**: Sempre solicite revisões de código para garantir a qualidade e conformidade com as diretrizes do projeto.
2. **Comunicação**: Mantenha uma comunicação clara com os colaboradores durante o processo de pull request.
3. **Resolução de Conflitos**: Esteja preparado para resolver conflitos de merge de forma eficiente, utilizando ferramentas adequadas.
4. **Deleção de Branches**: Após a conclusão de um pull request, delete a branch fonte para manter o repositório limpo.

5. **Documentação**: Documente o processo de pull request e resolução de conflitos para referência futura.

Resumo do Capítulo 2 sobre Tópicos Avançados de Git: Rebasing e Squashing

Rebasing

Rebasing é uma técnica alternativa ao merge para integrar mudanças de diferentes branches. Ele reescreve o histórico de commits, criando uma linha de commits linear, o que pode ser vantajoso para manter um histórico mais limpo. No entanto, isso também significa que alguns commits podem ser perdidos.

Processo de Rebasing:

1. Faça checkout para a branch fonte: `git checkout <branch-fonte>`.
2. Execute o rebase: `git rebase <branch-destino>`.
3. Faça checkout para a branch destino: `git checkout <branch-destino>`.
4. Realize o merge: `git merge <branch-fonte>`.

Vantagens:

- Histórico de commits linear.
- Facilita a identificação de falhas ao longo dos commits.

Desvantagens:

- Reescreve o histórico de commits, podendo perder alguns commits.

Usos Comuns:

- **Rodar Testes**: `git rebase -i --exec 'npm test' <branch>`
- **Padronização de Código**: `git rebase -i --exec 'eslint .' <branch>`
- **Checar Builds**: `git rebase -i --exec 'make build' <branch>`
- **Rodar Scripts Customizados**: `git rebase -i --exec './validate-commit.sh' <branch>`
- **Gerar Documentação**: `git rebase -i --exec 'generate-docs' <branch>`
- **Verificações de Segurança**: `git rebase -i --exec 'npm audit' <branch>`

Dicas Importantes

1. **Histórico Limpo**: Use rebase para manter um histórico de commits linear e mais fácil de seguir.

2. **Cuidado com o Histórico**: Evite rebase em branches compartilhadas para não perder commits importantes.
3. **Execução de Testes**: Utilize a flag `--exec` para rodar testes e outras verificações durante o rebase.
4. **Documentação e Segurança**: Garanta que a documentação e verificações de segurança sejam realizadas em cada commit.
5. **Automatização**: Combine rebase com scripts customizados para automatizar verificações e padronizações.

Squashing

Squashing é uma técnica usada para combinar vários commits em um único commit. Isso é especialmente útil em projetos grandes com muitos colaboradores, geralmente ao preparar um pull request. A técnica ajuda a manter o histórico de commits limpo e focado, evitando a inclusão de commits intermediários que não são relevantes para a branch principal.

Processo de Squashing

1. **Identificar Commits para Squash**:
    - Suponha que você tenha uma branch `feature4` com vários commits que deseja combinar.
    - Use `git rebase -i HEAD~3` para iniciar o rebase interativo dos últimos 3 commits.
2. **Rebase Interativo**:
    - Um arquivo será aberto listando os commits. Substitua a palavra “pick” por “squash” para todos os commits, exceto o primeiro.
    - Salve o arquivo. Outro arquivo abrirá para editar a mensagem do novo commit combinado. Edite se necessário e salve.
3. **Rebase da Branch**:
    - Faça checkout para a branch `feature4` e rebase com a branch `main`: `git rebase main`.
4. **Merge Final**:
    - Faça checkout para a branch `main` e realize o merge: `git merge feature4`.
    - O Git usará a estratégia de merge fast-forward, resultando em um histórico linear.

Dicas Importantes

1. **Rebase Interativo**: Utilize `git rebase -i` para realizar o squashing de commits de forma interativa.
2. **Mensagens de Commit**: Edite a mensagem do commit combinado para refletir claramente as mudanças feitas.
3. **Merge Estratégico**: Após o squashing, use merge com fast-forward para integrar as mudanças de forma linear.
4. **Deleção de Branches**: Após o merge, delete a branch fonte para manter o repositório organizado.

Essas práticas ajudarão a manter seu projeto organizado e o histórico de commits claro e conciso.

Resumo do Capítulo 3 sobre Resolução de Problemas e Recuperação no Git

Este capítulo aborda como recuperar erros no Git utilizando os comandos `reset`, `revert` e `reflog`. Esses comandos são poderosos e, em alguns casos, destrutivos, por isso devem ser usados com cuidado.

Comandos Destrutivos

- **Git Reset**: Desfaz commits, movendo o ponteiro da branch para um commit anterior.
  - **Opções**:
    - `--mixed` (padrão): Desfaz o commit e mantém as mudanças no Working Directory.
    - `--soft`: Desfaz o commit e mantém as mudanças no Working Directory e na Staging Area.
    - `--hard`: Desfaz o commit e descarta todas as mudanças no Working Directory e na Staging Area.
  - **Uso**: `git reset [opção] <SHA-hash do commit>`
- **Git Revert**: Cria um novo commit que reverte as mudanças de um commit específico, sem alterar o histórico de commits.
  - **Uso**: `git revert <SHA-hash do commit>`
  - **Exemplo**: `git revert HEAD` reverte o último commit.
- **Git Commit --amend**: Modifica o último commit, criando um novo commit com as alterações.
  - **Uso**: `git commit --amend [parâmetro] [novas alterações]`

Recomendações

- **Evitar em Branches Principais**: Não use comandos destrutivos em branches principais (e.g., `main`, `release`, `homolog`) para evitar inconsistências.
- **Uso Seguro**: Utilize `revert` em vez de `reset` para manter o histórico de commits intacto, especialmente em repositórios públicos.

Git Commit --amend

- **Uso**: Modifica o último commit, criando um novo commit com as alterações especificadas.
- **Comando**: `git commit --amend -m "Nova mensagem"`
- **Resultado**: Cria um novo commit com as alterações, removendo o commit antigo.

Git Cherry-pick

- **Uso**: Pega um commit de qualquer lugar do histórico e o aplica em uma branch específica.
- **Comando**: `git cherry-pick <commit>`
- **Exemplo**: Para aplicar o terceiro commit da branch `feature8` na branch `main`:
    1. Faça checkout para a branch `main`: `git checkout main`
    2. Execute o cherry-pick: `git cherry-pick <hash do terceiro commit da branch feature8>`
- **Flag**: `--no-commit` para aplicar as mudanças na Staging Area sem criar um novo commit.

Git Reflog

- **Uso**: Lista o histórico de todas as operações feitas em um repositório local, permitindo reverter estados do Git.
- **Comando**: `git reflog`
- **Exemplo**: Para recuperar um commit perdido após um `git reset --hard`:
    1. Use `git reflog` para encontrar o SHA1-hash do commit perdido.
    2. Restaure o commit: `git reset --hard <SHA1-hash>`

Dicas Importantes

1. **Compreensão dos Comandos**: Entenda claramente o que cada comando faz antes de usá-lo para evitar perdas irreversíveis.
2. **Uso em Branches Isoladas**: Prefira usar comandos destrutivos em branches isoladas ou em repositórios locais.
3. **Verificação de Estado**: Use `git status` e `git log` para verificar o estado do repositório antes de aplicar comandos destrutivos.
4. **Revert para Segurança**: Utilize `git revert` para desfazer mudanças de forma segura sem alterar o histórico de commits.
5. **Amend com Cuidado**: Use `git commit --amend` para corrigir o último commit, mas lembre-se de que ele reescreve o histórico.
6. **Cherry-pick Flexível**: Utilize `git cherry-pick` para aplicar commits específicos em outras branches, com ou sem criar novos commits.
7. **Reflog para Recuperação**: Use `git reflog` para recuperar commits perdidos e reverter estados do repositório.

Essas práticas ajudarão a gerenciar erros e recuperar mudanças de forma eficiente no Git.

Resumo do Capítulo 4 sobre SHA-1 Hash no Git

O SHA-1 hash é um cálculo matemático usado para criptografar documentos no Git. Ele faz parte da família de funções hash criptográficas, que inclui SHA-1, SHA-256, SHA-512, entre outras. O Git utiliza o SHA-1 para gerar um código hash de 160 bits (40 caracteres hexadecimais) para cada arquivo e tipo de arquivo.

Geração do SHA-1 Hash

- **Entrada**: Qualquer tamanho de dados (conteúdo do arquivo, data de criação, nome do autor, etc.).
- **Saída**: Código hash de 160 bits (40 caracteres hexadecimais).

Perguntas Comuns

1. **Limite de Armazenamento**:
    - O Git pode armazenar até aproximadamente (2^{160}) combinações possíveis, o que é um número extremamente grande (1461501637330902918203684832716283019655932542976 combinações).
2. **Probabilidade de Colisão**:
    - A chance de dois arquivos diferentes gerarem o mesmo código hash é extremamente baixa, quase impossível.

Comparações para Compreensão

- O número de combinações possíveis do SHA-1 é maior do que:
    - A quantidade de grãos de areia na Terra ((7.5 \times 10^{18})).
    - A quantidade de estrelas no universo observável ((10^{24})).
    - A quantidade de segundos desde o Big Bang ((4.35 \times 10^{17})).

Dicas Importantes

1. **Segurança**: Utilize SHA-1 para garantir a integridade e segurança dos arquivos no Git.
2. **Compreensão dos Limites**: Entenda que o limite de armazenamento é vasto, mas a probabilidade de colisão é extremamente baixa.
3. **Uso em Projetos**: Confie no SHA-1 para gerenciar grandes volumes de arquivos e dados em projetos colaborativos.

Resumo do Capítulo 5 sobre .gitignore no Git

O arquivo `.gitignore` serve para ignorar arquivos desnecessários que não precisam ser versionados. Ele ajuda a manter o repositório limpo e eficiente, evitando que arquivos temporários, sensíveis ou irrelevantes sejam rastreados.

Importância do .gitignore

- **Redução de Clutter**: Previne que arquivos temporários ou desnecessários sejam rastreados.
- **Segurança**: Evita que arquivos sensíveis, como chaves privadas e senhas, sejam adicionados ao repositório.
- **Eficiência**: Reduz o tamanho do repositório e melhora a performance do Git.
- **Controle e Limpeza**: Mantém o repositório organizado, incluindo apenas arquivos necessários.

- **Personalização por Ambiente**: Permite regras específicas para diferentes ambientes de desenvolvimento.

Exemplos de Arquivos a Ignorar

- **Arquivos Temporários**: Logs, arquivos temporários de editores de texto.
- **Arquivos Sensíveis**: Chaves privadas, senhas.
- **Arquivos Grandes**: Arquivos que consomem muita memória.
- **Diretórios de Build**: `bin/`
- **Diretórios de Dependências**: `node_modules/`
- **Arquivos Compilados e de Histórico**: `*.pyc`, `*.log`
- **Arquivos do Sistema Operacional**: `.DS_Store`, `Thumbs.db`

Criação e Uso do .gitignore

1. **Adicionar ao Projeto**: Crie um arquivo `.gitignore` no diretório raiz do projeto.
2. **Definir Regras**: Adicione as regras de supressão necessárias.
3. **Commit e Push**: Faça commit do `.gitignore` e envie para o repositório remoto.

Lidando com Arquivos Já Commitados

- **Opção 1**:
    1. Adicione a nova regra ao `.gitignore`.
    2. Delete o arquivo no Working Directory.
    3. Faça commit das mudanças.
    4. Adicione novamente o arquivo deletado.
- **Opção 2**:
    1. Adicione a nova regra ao `.gitignore`.
    2. Delete o arquivo apenas do repositório: `git rm --cached <arquivo>`.
    3. Faça commit das mudanças.

Dicas Importantes

1. **Configuração Inicial**: Adicione o `.gitignore` logo no início do projeto.
2. **Ferramentas de Geração**: Use sites como toptal.com/developers/gitignore para gerar automaticamente o conteúdo do `.gitignore` baseado nas tecnologias usadas.
3. **Revisão Regular**: Revise e atualize o `.gitignore` conforme o projeto evolui.
4. **Consistência**: Certifique-se de que todos os colaboradores utilizem o mesmo `.gitignore` para evitar inconsistências.
5. **Documentação**: Documente as regras do `.gitignore` para que todos os membros da equipe entendam o que está sendo ignorado e por quê.

Resumo do Capítulo 6 sobre Tags no Git

Tags no Git são usadas para marcar versões importantes de commits, facilitando o acesso e a automação de processos. São ponteiros textuais estáticos que apontam para commits específicos, ao contrário das branches, que são dinâmicas.

Tipos de Tags

1. **Lightweight (Leve)**:
    - Armazenadas apenas no diretório `.git/refs/tags`.
    - Servem para demarcar um commit sem informações adicionais.
    - **Comando**: `git tag <nome-da-tag>`
    - **Exemplo**: `git tag v1.0.0`
2. **Annotated (Anotada)**:
    - Armazenadas nos diretórios `.git/refs/tags` e `.git/objects`.
    - Incluem data de criação, nome do autor e uma mensagem.
    - **Comando**: `git tag -a <nome> -m <mensagem>`
    - **Exemplo**: `git tag -a v1.0.0 -m "Nova tag"`

Uso de Tags

- **Marcar Versões de Lançamento**: Usadas para marcar versões de lançamento (release versions).
- **Referências Memoráveis**: Criam referências fáceis de lembrar para certos commits.
- **Automação de CI/CD**: Utilizadas como gatilhos para processos de implantação automatizados.

Comandos Úteis

- **Criar Tag Leve**: `git tag <nome-da-tag>`
- **Criar Tag Anotada**: `git tag -a <nome> -m <mensagem>`
- **Listar Tags**: `git tag`
- **Mostrar Informações da Tag**: `git show <tag>`
- **Deletar Tag Local**: `git tag -d <nome-da-tag>`
- **Enviar Tags para o Repositório Remoto**: `git push --tags`
- **Enviar Tag Específica para o Repositório Remoto**: `git push origin <tag>`
- **Deletar Tag do Repositório Remoto**: `git push --delete <repositório> <nome-da-tag>`

Dicas Importantes

1. **Marcação de Versões**: Use tags para marcar versões importantes do projeto, facilitando o acesso a essas versões.
2. **Automação**: Utilize tags em processos de CI/CD para automatizar implantações.
3. **Consistência**: Mantenha uma convenção de nomenclatura consistente para tags.

4. **Envio de Tags**: Lembre-se de enviar tags para o repositório remoto usando `git push --tags`.
5. **Verificação de Tags**: Use `git show <tag>` para verificar informações detalhadas sobre uma tag.

Resumo do Capítulo 7 sobre Stashing no Git

Stashing é uma técnica útil no Git que permite salvar temporariamente as alterações feitas no Working Directory para que você possa mudar de branch sem perder essas mudanças. Isso é especialmente útil quando você precisa alternar entre branches sem fazer um commit.

Como Funciona o Stashing

- **Salvar Alterações**: Use `git stash` para guardar as alterações não commitadas. Isso limpa o Working Directory, permitindo o checkout para outra branch.
- **Recuperar Alterações**: Use `git stash pop` para aplicar as mudanças salvas e removê-las do stash. Alternativamente, use `git stash apply` para aplicar as mudanças sem removê-las do stash.

Comandos Úteis

- **Salvar Alterações**: `git stash`
- **Aplicar e Remover Alterações**: `git stash pop`
- **Aplicar Alterações sem Remover**: `git stash apply`
- **Listar Stashes**: `git stash list`
- **Remover Stash Específico**: `git stash drop <hash>`
- **Limpar Todos os Stashes**: `git stash clear`

Exemplos de Uso

1. **Salvar Alterações e Mudar de Branch**:
    - Faça alterações na branch `temp`.
    - Salve as alterações: `git stash`.
    - Mude para a branch `feature3`: `git checkout feature3`.
    - Volte para a branch `temp` e aplique as alterações: `git stash pop`.
2. **Resolver Conflitos**:
    - Se houver conflitos ao aplicar o stash, resolva-os manualmente e continue o desenvolvimento.
3. **Experimentação**:
    - Use `git stash apply` para experimentar mudanças sem remover o stash. Se não gostar das mudanças, você pode descartá-las e aplicar o stash novamente.

Dicas Importantes

1. **Uso Regular**: Utilize o stashing regularmente para manter seu Working Directory limpo e facilitar a troca entre branches.
2. **Resolução de Conflitos**: Esteja preparado para resolver conflitos ao aplicar stashes.
3. **Gerenciamento de Stashes**: Use `git stash list` para monitorar seus stashes e `git stash clear` para limpar todos os stashes quando não forem mais necessários.
4. **Flexibilidade**: Aproveite a flexibilidade do `git stash apply` para testar mudanças sem comprometer o stash original.

Essas práticas ajudarão a gerenciar suas alterações de forma eficiente e a manter seu fluxo de trabalho organizado no Git.

Resumo do Capítulo 8 sobre Chaves SSH no GitHub

O GitHub permite que os usuários se autentiquem de várias maneiras, incluindo a autenticação via SSH, que oferece mais segurança e conveniência em comparação com a autenticação por usuário e senha.

Benefícios das Chaves SSH

- **Segurança**: Chaves SSH são criptografadas e quase impossíveis de decifrar por força bruta, ao contrário das senhas.
- **Conveniência**: Após configurada, a chave SSH elimina a necessidade de digitar usuário e senha em cada interação com o repositório.
- **Controle de Acesso**: Chaves SSH podem ser facilmente adicionadas ou removidas sem afetar outras credenciais.
- **Resistência a Phishing**: Como a senha não é digitada, ela não trafega pela internet, reduzindo o risco de ser capturada por ataques de phishing.

Dicas Importantes

1. **Segurança**: Utilize chaves SSH para aumentar a segurança das suas interações com o GitHub.
2. **Configuração Inicial**: Configure a chave SSH logo no início para evitar a necessidade de digitar senhas repetidamente.
3. **Gerenciamento de Chaves**: Adicione e remova chaves SSH conforme necessário para manter o controle de acesso.
4. **Documentação**: Documente o processo de configuração da chave SSH para referência futura e para ajudar outros membros da equipe.