Controle de Versão com Git

Comandos Essenciais para Gerenciamento Colaborativo

Página 1: Capa

Autor: Manus AI

Novembro de 2025

Página 2: Introdução ao Controle de Versão

O **Controle de Versão (VCS - Version Control System)** é um sistema que registra as alterações feitas em um arquivo ou conjunto de arquivos ao longo do tempo, permitindo que você recupere versões específicas posteriormente. É uma ferramenta essencial para o desenvolvimento de software e para qualquer trabalho colaborativo.

Por que usar Git?

O Git é o sistema de controle de versão distribuído (DVCS) mais popular do mundo. A principal diferença de um VCS centralizado é que, no Git, cada desenvolvedor tem uma cópia completa do repositório, incluindo todo o histórico de alterações. Isso oferece:

- **Resiliência:** Se o servidor central falhar, o histórico completo ainda está nas máquinas dos desenvolvedores.
- **Velocidade:** A maioria das operações (como `commit` e `branch`) é feita localmente.
- **Colaboração:** Facilita o trabalho em equipe e a fusão de alterações.

Instalação e Configuração Inicial

Após instalar o Git, o primeiro passo é configurar sua identidade:

```
git config --global user.name "Seu Nome"
git config --global user.email "seu.email@exemplo.com"
```

Página 3: O Ciclo de Vida do Git

O Git gerencia o estado dos arquivos em três áreas principais, que definem o ciclo de vida de uma alteração:

1. **Working Directory (Diretório de Trabalho):** Onde você edita e modifica os arquivos.
2. **Staging Area (Área de Preparação):** Uma área intermediária onde você seleciona quais alterações do Working Directory farão parte do próximo *commit*.
3. **Local Repository (Repositório Local):** Onde o Git armazena permanentemente as alterações na forma de *commits*.

Inicializando um Repositório

Para começar a usar o Git em um projeto:

```
# Inicializa um novo repositório Git no diretório atual
git init
```

Verificando o Status

O comando `git status` é seu melhor amigo, pois mostra o estado atual dos arquivos: quais foram modificados, quais estão na Staging Area e quais estão no Repositório Local.

```
# Mostra o estado dos arquivos
git status
```

Página 4: O Comando Essencial: Commit

Um **Commit** é o registro permanente de um conjunto de alterações no Repositório Local. É como um "instantâneo" do seu projeto em um determinado momento.

Preparando as Alterações (`git add`)

Antes de fazer um *commit*, você precisa mover as alterações do Working Directory para a Staging Area:

```
# Adiciona um arquivo específico à Staging Area
git add nome_do_arquivo.txt

# Adiciona todas as alterações de arquivos rastreados à Staging Area
git add .
```

Criando o Commit (`git commit`)

O *commit* só é criado a partir do que está na Staging Area. A mensagem de *commit* deve ser clara e concisa, descrevendo o que foi alterado.

```
# Cria um commit com a mensagem especificada
git commit -m "Mensagem clara sobre a alteração"
```

Visualizando o Histórico (`git log`)

Para ver o histórico de *commits* do projeto:

```
# Mostra o histórico completo de commits
git log

# Mostra o histórico de forma mais concisa e gráfica
git log --oneline --graph
```

Página 5: Repositórios Remotos

Um **Repositório Remoto** é uma versão do seu projeto hospedada em um servidor na internet (como GitHub, GitLab ou Bitbucket). Ele é usado para backup e, principalmente, para colaboração.

Clonando um Repositório

Para obter uma cópia local de um projeto existente em um servidor remoto:

```
# Cria uma cópia local do repositório remoto
git clone <URL_do_repositorio>
```

Adicionando um Repositório Remoto

Se você inicializou um repositório localmente e agora quer conectá-lo a um remoto:

```
# Adiciona um novo remoto chamado 'origin'
git remote add origin <URL_do_repositorio>
```

O nome padrão para o repositório remoto principal é `origin`.

Página 6: Sincronização: Push e Pull

A sincronização é o processo de mover alterações entre o Repositório Local e o Repositório Remoto.

Enviando Alterações (`git push`)

O comando `git push` envia seus *commits* locais (do Repositório Local) para o Repositório Remoto.

```
# Envia os commits da branch atual para o remoto 'origin'
git push origin nome_da_branch
```

Na primeira vez, use `git push -u origin nome_da_branch` para configurar o rastreamento.

Recebendo Alterações (`git pull`)

O comando `git pull` é usado para baixar o conteúdo do repositório remoto e, automaticamente, fundir (`merge`) essas alterações com sua *branch* local.

```
# Baixa e funde as alterações do remoto para a branch atual
git pull origin nome_da_branch
```

Diferença entre Pull e Fetch

- **`git fetch`:** Baixa as alterações do remoto, mas **não** as funde com sua *branch* local.
- **`git pull`:** É um atalho para `git fetch` seguido de `git merge`.

Página 7: Gerenciamento de Branches (Ramificações)

Uma **Branch** (ramificação) é um ponteiro móvel para um dos seus *commits*. O propósito principal é isolar o desenvolvimento de novas funcionalidades ou correções de bugs da *branch* principal (geralmente `main` ou `master`).

Comandos de Branch

Comando	Função
`git branch`	Lista todas as branches locais.
`git branch nova_feature`	Cria uma nova branch chamada `nova_feature`.
`git checkout nova_feature`	Alterna para a branch `nova_feature`.
`git switch -c nova_feature`	Cria e alterna para a nova branch (comando moderno).

Fluxo de Trabalho com Branches

O uso de *branches* é fundamental para o desenvolvimento colaborativo. O fluxo típico é:

1. Crie uma nova *branch* para sua tarefa.
2. Faça seus *commits* nessa *branch*.
3. Funda (`merge`) a *branch* de volta à *branch* principal quando a tarefa estiver concluída.

Página 8: O Fluxo de Trabalho: Merge

O `git merge` é o comando usado para integrar as alterações de uma *branch* em outra.

Fundindo Branches

Para fundir a *branch* `feature-x` na *branch* `main`:

```
# 1. Mude para a branch que receberá as alterações
git switch main

# 2. Funda a outra branch na branch atual
git merge feature-x
```

Conflitos de Merge

Um **Conflito de Merge** ocorre quando o Git não consegue fundir automaticamente as alterações, geralmente porque duas *branches* modificaram a mesma linha do mesmo arquivo de maneiras diferentes.

Resolução:

1. O Git marca os arquivos com conflito.
2. Você edita manualmente o arquivo, escolhendo qual versão manter.
3. Adiciona o arquivo resolvido à Staging Area (`git add nome_do_arquivo`).
4. Cria um novo *commit* de merge (`git commit`).

Página 9: Desfazendo Alterações

O Git oferece diversas maneiras de desfazer ou reverter alterações, dependendo do estágio em que a alteração se encontra.

Comando	Onde Desfaz	O que Faz
`git restore`	Working Directory	Descarta alterações não *commitadas* no Working Directory.
`git reset`	Staging Area / Local Repository	Move o ponteiro da *branch* para um *commit* anterior. Pode ser usado para desfazer um `git add`.
`git revert`	Local Repository	Cria um **novo commit** que desfaz as alterações de um *commit* anterior. É o método mais seguro para histórico compartilhado.

Exemplo de Revert (Seguro)

```
# Cria um novo commit que desfaz o commit com o ID especificado
git revert <ID_do_commit>
```

Exemplo de Reset (Local)

```
# Remove o último commit do histórico local, mas mantém as alterações no Working Directory
git reset HEAD~1
```

Página 10: Colaboração e Fluxo de Trabalho

O Git é a base para o desenvolvimento colaborativo moderno, utilizando plataformas como GitHub e GitLab.

Feature Branch Workflow

Este é o fluxo de trabalho mais comum:

1. A *branch* `main` (ou `master`) é sempre estável.
2. Novas funcionalidades são desenvolvidas em *branches* separadas (`feature-x`).
3. Quando a *feature* está pronta, ela é integrada à `main` via **Pull Request (PR)** ou **Merge Request (MR)**.

Pull Request / Merge Request

Um PR/MR é um mecanismo para notificar os membros da equipe que você concluiu uma *feature* e deseja fundir suas alterações. Ele permite:

- Revisão de Código (Code Review) por outros membros.
- Discussão e *feedback* sobre as alterações.
- Execução de testes automatizados antes da fusão.

Página 11: Dicas e Boas Práticas

Mensagens de Commit

Uma boa mensagem de *commit* deve ser:

- **Concisa:** Título com até 50 caracteres.
- **Informativa:** Corpo detalhando o *porquê* da mudança.
- **Padrão:** Use verbos no imperativo (Ex: “Adiciona validação de e-mail”, “Corrige bug de login”).

O Arquivo `.gitignore`

Este arquivo lista os arquivos e diretórios que o Git deve **ignorar** e não rastrear. É essencial para excluir:

- Arquivos de configuração local (senhas, chaves).
- Dependências de projeto (ex: pasta `node_modules`).
- Arquivos temporários ou de *build* (ex: `.log`, `.class`).

Comandos Úteis

Comando	Função
`git diff`	Mostra as diferenças entre o Working Directory e a Staging Area, ou entre *commits*.
`git stash`	Salva temporariamente as alterações não *commitadas* para que você possa alternar de *branch* e depois restaurá-las.

Página 12: Conclusão e Próximos Passos

O Git é a ferramenta definitiva para a gestão de projetos de software. Dominar os comandos de ciclo de vida (`add` , `commit`), sincronização (`push` , `pull`) e colaboração (`branch` , `merge`) é um requisito fundamental no mercado de TI.

Resumo dos Comandos Essenciais:

Categoria	Comandos
Inicialização	`git init` , `git clone` , `git config`
Ciclo de Vida	`git status` , `git add` , `git commit`
Sincronização	`git push` , `git pull` , `git fetch`
Colaboração	`git branch` , `git switch` , `git merge`
Desfazer	`git restore` , `git reset` , `git revert`

Próximos Passos para Aprofundamento:

1. **Git Rebase:** Aprenda a reescrever o histórico de *commits* de forma limpa.
2. **Git Flow:** Estude um modelo de *branching* mais estruturado para projetos grandes.
3. **Git Hooks:** Automatize tarefas em seu repositório local.

A prática diária com o Git é a chave para a fluência.