# 我的github token

ghp\_LTci7SqFv8qgBvLJz0Y6l5u44keceX3fRhf6

# 基本概念

Git是一个版本控制系统，用来追踪和管理代码的变化历史。

# 核心功能

1、记录文件的每次修改

2、保存历史版本，可随时回退

3、支持多人协作开发

4、管理不同分支的代码

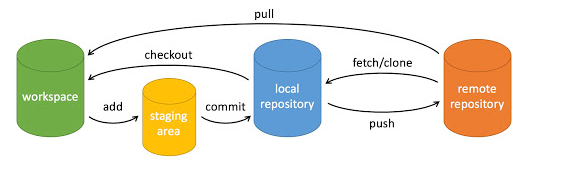
# 常用场景

1、个人项目：防止代码丢失，追踪修改历史

2、团队协作：多人同时开发，合并代码变更

3、开源项目：GitHub、GitLab等平台都基于Git

# git基本操作



workspace：工作区

staging area：暂存区/缓存区

# 工作流程

1、克隆 Git 资源作为工作目录。

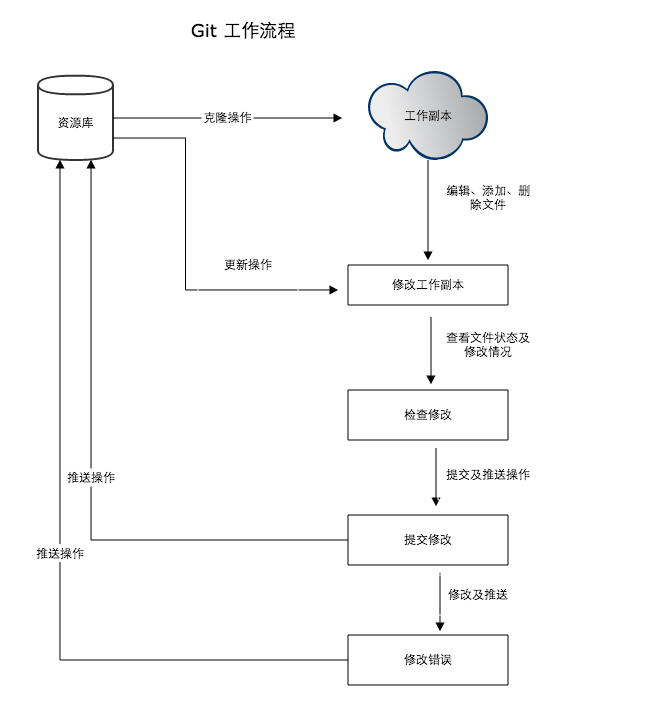
2、在克隆的资源上添加或修改文件。

3、如果其他人修改了，你可以更新资源。

4、在提交前查看修改。

5、提交修改。

6、在修改完成后，如果发现错误，可以撤回提交并再次修改并提交。



# git merge和rebase的区别

Git中的merge和rebase是两种常用的分支合并策略，它们在合并分支时有一些区别。

## ****Merge（合并）****

### 工作原理

1、将两个分支的历史合并在一起。

2、创建一个新的"合并提交"来连接两条分支线

3、保留完整的分支历史结构

### 命令

git merge feature-branch

### 结果特点

1、历史记录呈现分叉和汇合的网状结构。

2、保留所有原始提交的时间戳和作者信息

3、合并点清晰可见。

## ****Rebase（变基）****

### 工作原理

1、将当前分支的提交"重新应用"到目标分支的最新提交上

2、重写提交历史，让分支看起来像线性发展。

3、实际上是创建新的提交（相同内容，不同哈希值）

### 命令

git rebase main

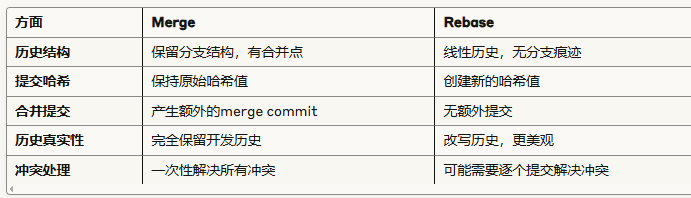
### 结果特点

1、历史记录呈现线性、干净的直线结构

2、提交时间戳会被更新

3、没有额外的合并提交

## 主要区别对比



## 使用场景建议

### 使用 Merge 的情况

1、公共分支（如main）的合并

2、需要保留完整开发历史

3、团队协作，避免历史混乱

4、已经推送到远程的分支

### 使用 Rebase 的情况

1、本地私有分支整理

2、希望保持干净的线性历史

3、功能分支在合并前的整理

4、未推送到远程的提交

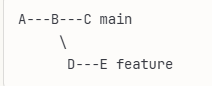
## 重要注意事项

永远不要在公共分支上使用rebase！

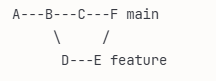
原因：rebase会改变提交哈希，如果其他人已经基于原来的提交工作，会造成历史冲突。

## 实际示例

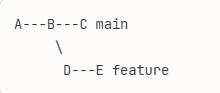
### Merge场景



执行git merge feature后



### Rebase场景



执行git rebase main后



总的来说，merge保留历史真实性，rebase追求历史整洁性。选择哪种方式取决于你的项目需求和团队约定。

# 命令

## git init

初始化仓库

## git clone

拷贝一份远程仓库，也就是下载一个项目。

## git add

添加文件到暂存区

## git status

查看仓库当前的状态，显示有变更的文件。

## git diff

比较文件的不同，即暂存区和工作区的差异。

## git commit

提交暂存区到本地仓库。

## git reset

回退版本。

## git rm

## git fetch

用于从远程仓库获取最新提交的 Git 命令。它会将远程仓库的分支和标签的更新内容下载到本地仓库，但不会自动合并或修改当前工作区的文件。

## git blame

### 基本概念

"git blame" 是 Git 版本控制系统中的一个命令，用于显示指定文件的每一行是由谁在何时修改的。该命令对于跟踪文件的修改历史非常有用，可以帮助开发人员了解每一行代码的修改者及修改时间，从而更好地理解代码的演变过程。

使用 "git blame" 命令时，会显示指定文件的每一行代码以及该行最近一次被修改的提交信息，包括提交的哈希值、作者、修改时间等。通过这些信息，开发人员可以追溯到代码的修改历史，找出特定功能或 bug 修复是由谁所引入的，也可以了解代码的演进轨迹。

例如，在命令行中执行以下命令可以查看指定文件的修改历史：

git blame file\_name

通过 "git blame" 命令，开发人员可以更好地理解代码的演变过程，定位问题的根源，也可以及时联系到相关的开发人员进行沟通和协作。这有助于团队更好地管理和维护代码库，并提高代码质量和可维护性。

### 应用场景

#### 1、****理解代码变更历史****

开发人员可以通过 git blame 查看每行代码的变更历史，从而理解代码的演变过程和每次修改的原因。

#### **2、**追踪 bug 或问题来源****

当出现问题或 bug 时，可以使用 git blame 来确定引入问题的具体提交，有助于定位和修复问题。

#### ****3、代码审查****

在进行代码审查时，可以使用 git blame 查看某段代码的作者及其修改的提交信息，帮助评估代码的质量和合理性。

#### **4、**了解团队成员的工作贡献****

团队成员可以使用 git blame 来了解彼此在项目中的贡献，以及每个人对代码库的贡献程度。

# origin

## 基本概念

在 Git 中，origin 是指一个远程仓库的默认名字，通常情况下它指向了代码仓库托管服务（如 Github、Gitlab）上的代码仓库。当我们使用 Git 管理代码时，通常需要进行代码的远程同步和协作。为了实现这个功能，我们需要首先在本地创建一个代码仓库。然后，我们可以将本地代码仓库与远程代码仓库建立联系，并给远程仓库起一个别名，而这个别名就是 origin。在 Git 中，我们可以使用 git remote 命令来查看当前代码仓库中所有可用的远程仓库。如果我们已经将代码仓库与某个远程仓库关联起来，那么我们可以使用 git remote -v 命令来查看远程仓库的别名（即 origin）和对应的 URL。

## 远程仓库和 origin

**1、远程仓库**：远程仓库是托管在网络上的 Git 仓库，允许多个开发者协作并共享代码。常见的远程仓库托管服务包括 GitHub、GitLab、Bitbucket 等。

2、origin **的作用**：origin 是 Git 默认使用的远程仓库名称。在克隆一个远程仓库到本地时，Git 会自动为其命名为 origin，并将远程仓库的 URL 关联到 origin 上。

## 使用 origin

**1、推送和拉取**：通过 origin，你可以将本地的修改推送（push）到远程仓库，或者从远程仓库拉取（pull）更新到本地。例如，使用 git push origin main 将本地的 main 分支推送到 origin 所关联的远程仓库。

**2、克隆时的默认行为**：当你使用 git clone 命令克隆一个远程仓库时，默认情况下 Git 会将远程仓库的别名设为 origin，例如 git clone <repository-url> 将会创建一个名为 origin 的远程仓库引用。

## 多个远程仓库和 origin

**1、多个远程仓库**：除了 origin 外，你也可以配置和管理多个远程仓库。通过 git remote add <name> <url> 可以为其他远程仓库添加别名，使得你可以同时与多个远程仓库进行交互。

**2、默认行为**：尽管可以有多个远程仓库，但 origin 通常是主要的远程仓库引用，因为它是 Git 在克隆操作中默认设置的。多数情况下，大部分操作都会默认使用 origin，除非显式指定其他远程仓库。

## 示例

假设你通过以下命令克隆了一个远程仓库：



Git 将会：

1、自动为远程仓库 https://github.com/example/repo.git 设置一个名为 origin 的别名。

2、你可以通过 git push origin main 将本地的 main 分支推送到 origin 所指向的远程仓库。

## 总结

总之，origin 在 Git 中表示一个默认的远程仓库别名，通常指向我们使用的代码仓库托管服务上的代码仓库。我们可以使用 Git 命令操作这个远程仓库，例如推送代码、拉取代码等等。

# origin/master

## 基本概念

1、origin：这是 Git 远程仓库的默认名称。当你克隆一个远程仓库时，Git 会自动将远程仓库命名为 origin。

2、master：这是 Git 仓库的默认主分支名称（尽管现在许多项目使用 main 作为默认分支名称）。

3、origin/master：这是一个**远程跟踪分支**（remote-tracking branch），表示远程仓库（origin）中的 master 分支在本地仓库中的状态。

## 作用

1、查看远程分支的状态：了解远程仓库的 master 分支的最新提交。

2、同步本地分支：将本地分支与远程分支保持一致。

3、比较差异：查看本地分支与远程分支之间的差异。

# HEAD

## 基本概念

在 Git 中，HEAD 是一个非常重要的概念，它指向当前所在的工作树（working tree）或者说当前检出的分支（branch）。理解 HEAD 的含义有助于我们理解 Git 的工作机制和版本控制的基本原理。

## 作用

1、**当前分支指针**: 在 Git 中，HEAD 是一个指针（pointer），它始终指向当前所在的分支或者提交（commit）。换句话说，HEAD 告诉 Git 当前在哪个分支上进行操作或者当前所处的提交状态。

2、**确定工作树状态**: HEAD 也可以理解为指向当前工作目录状态的指针，它指示了你当前工作树的内容是根据哪个分支或提交来的。

## 具体指向

**1、分支引用**：当你在一个分支上工作时，HEAD 将指向该分支的引用（比如 refs/heads/main 或简单的 main）。

**2、特定提交**：在特定情况下，比如进行 git checkout 操作后，HEAD 可能会直接指向一个具体的提交（commit），而不是一个分支。这种情况下称为 "detached HEAD"（脱离的 HEAD），通常用于查看历史版本或进行临时修改。

## 状态

**1、正常状态**：在正常的分支工作中，HEAD 指向当前分支的引用，例如 refs/heads/main。

**2、脱离状态**：当 HEAD 指向一个具体的提交而不是分支时，处于 "detached HEAD" 状态。在这种状态下，任何新的提交都不会更新任何分支引用，除非你创建新的分支或者切换回一个已存在的分支。

## 使用场景

**1、查看历史版本**：通过将 HEAD 指向历史提交，可以查看以前的工作状态或比较不同版本之间的差异。

**2、分支操作**：在创建新分支、合并分支或者进行其他分支操作时，HEAD 的正确指向非常重要，它决定了这些操作会影响哪个分支的状态。

## 示例

假设当前在 main 分支上进行工作：

1、HEAD 指向 refs/heads/main，表示你当前在 main 分支上进行操作。

2、通过 git checkout some-branch 切换到 some-branch 分支时，HEAD 将会指向 refs/heads/some-branch。

3、使用 git checkout abc123 切换到具体提交 abc123 时，HEAD 将会指向该提交，处于脱离状态。

## 总结

HEAD 在 Git 中是一个非常重要的概念，它指示了当前工作目录的状态是基于哪个分支或提交。理解和正确管理 HEAD 的指向有助于避免意外操作和理解版本控制的基本工作原理。

# tag

## 基本概念

在 Git 中，**Tag（标签）** 是一个用于标记特定提交（commit）的指针，通常用于标识项目的重要节点，例如版本发布（如 v1.0.0）。Tag 的作用是为某个提交打上一个易于记忆的名字，方便后续查找和引用。

## 作用

### 1、标记版本

常用于标记软件发布的版本号（如 v1.0.0）

### 2、标识重要节点

用于标识项目中的重要里程碑，例如功能完成、测试通过等。

### 3、快速定位

通过 Tag 可以快速定位到某个特定的提交，而不需要记住复杂的 commit hash。

### 4、发布管理

在开源项目中，Tag 通常与 GitHub/GitLab 的 Release 功能结合使用，用于管理发布版本。

## tag的类型

### 1、****轻量标签（Lightweight Tag）****

1、只是一个指向特定提交的引用，类似于一个分支。

2、不包含额外信息（如标签作者、标签信息等）。

3、适合临时标记。

### **2、**附注标签（Annotated Tag）****

1、是一个独立的 Git 对象，包含完整的标签信息（如标签作者、标签信息、日期等）。

2、适合正式发布版本。

# .gitignore文件

## 基本概念

.gitignore 文件在 Spring Boot 项目中起到过滤版本控制的关键作用。它告诉 Git 哪些文件和目录不应该被提交到代码仓库中。

## 作用

.gitignore 文件的主要目的是排除那些不需要版本控制的文件，包括：

1、自动生成的文件

2、编译产物

3、个人配置文件

4、敏感信息文件

5、临时文件

## 感叹号(!)的作用

在 .gitignore 中，! 表示否定规则，即**强制包含**某些文件或目录，即使它们匹配了前面的忽略规则。

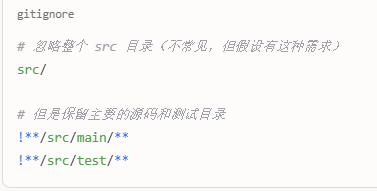
这些否定规则的主要目的是确保项目的核心源代码和必要的构建工具文件能够正确地被版本控制系统跟踪，即使存在较为宽泛的忽略规则。

### 实际应用场景

#### 1、广泛忽略后的精确包含



#### 2、目录级别的选择性忽略



#### 3、复杂项目结构的精确控制



### 规则执行顺序

Git 处理 .gitignore 规则是**从上到下**执行的。



### 示例

假设你的项目结构如下：



### 注意事项

1、顺序很重要：否定规则必须在相应的忽略规则之后。

2、路径匹配：确保路径模式正确匹配你的项目结构。

3、团队一致性：确保团队成员都使用相同的 .gitignore 配置。

## 最佳实践建议

### 1、项目初期就创建

在项目开始时就应该配置好 .gitignore，避免误提交不必要的文件。

### 2、团队统一标准

团队成员应该使用相同的 .gitignore 模板，确保一致性

### 3、分层配置

可以在不同目录层级使用不同的 .gitignore 文件，实现更精细的控制。

### 4、定期更新

随着项目发展，可能需要添加新的忽略规则。

### 5、安全考虑

确保包含敏感信息的配置文件（如数据库密码、API 密钥等）被正确忽略

# .gitattributes文件

## 基本概念

.gitattributes 文件是 Git 仓库中的一个配置文件，用于定义**特定路径或文件的属性**，从而控制 Git 对这些文件的行为。它在 IntelliJ IDEA 项目中的作用与在普通 Git 仓库中的作用是一致的。

## 作用

1、定义文件的换行符规则：确保不同操作系统下的换行符一致性。

2、指定文件的合并策略：控制 Git 在合并冲突时的行为。

3、标记文件的二进制属性：防止 Git 对二进制文件进行差异比较。

4、设置文件的差异比较方式：自定义文件的差异比较工具。

5、控制文件的导出行为：在导出或归档时忽略特定文件。

## ****在 IntelliJ IDEA 项目中的应用****

### 1、确保跨平台一致性

在团队协作中，不同开发者可能使用不同的操作系统（Windows、macOS、Linux），.gitattributes 文件可以确保换行符的一致性，避免因换行符差异导致的代码冲突。

### 2、优化 Git 行为

对于二进制文件（如图片、PDF 等），通过 .gitattributes 文件标记为二进制，可以避免 Git 尝试合并或差异比较，减少不必要的冲突。

### 3、自定义合并策略

对于特定文件（如配置文件），可以设置合并策略，避免在合并时丢失重要信息。

### 4、提高开发效率

通过 .gitattributes 文件，可以减少因文件属性问题导致的 Git 操作错误，提高开发效率。

## 注意事项

1、文件位置：.gitattributes 文件通常放在 Git 仓库的根目录下，但也可以放在子目录中，规则会递归生效。

2、优先级：子目录中的 .gitattributes 文件会覆盖父目录中的规则。

3、团队协作：建议将 .gitattributes 文件提交到 Git 仓库，确保所有开发者使用相同的规则。

## 是否可以删除

可以删除，但需要谨慎。

### 1、可以删除的情况

如果你的项目不需要特定的 Git 文件属性配置（如换行符控制、合并策略等）。

如果删除后不会影响团队协作或项目的正常运行。

如果你使用的是 Git 的默认行为，且没有遇到任何问题。

### 2、不建议删除的情况

如果项目中存在跨平台开发（如 Windows、macOS、Linux），且需要统一换行符。

如果项目中有二进制文件（如图片、PDF 等），且需要标记为二进制以避免 Git 尝试合并或差异比较。

如果项目中使用了自定义的合并策略或差异比较工具。

如果团队协作中依赖 .gitattributes 文件的配置。