Git 简明手册

# 前言

欢迎使用 Git 简明手册，这是一份详尽的指南，旨在帮助新手和经验丰富的开发人员有效使用 Git（软件行业最流行的版本控制系统）所需的知识。无论您是管理个人项目还是与大型团队协作，这份简明手册都旨在成为您的首选资源。通过提供清晰的解释和实际示例，它旨在揭开 Git 的神秘面纱，并将其转化为直观、可操作的见解。

在本指南中，您将探索构成软件版本控制支柱的各种 Git 命令和概念。从初始化仓库和提交更改等基本操作，到分支、合并和变基等更高级的技术，这份简明手册涵盖了所有内容。您还将深入研究压缩提交、二分调试、处理子模块以及实现子树等专业主题，确保您做好充分准备，应对开发过程中出现的任何挑战。

随着学习的深入，您将学习如何维护数据完整性、管理多个工作树以及高效解决合并冲突。每个部分都提供循序渐进的指导，让您能够立即将所学知识运用到项目中。完成本课程后，您不仅会对 Git 有更深入的了解，还能自信地使用它来简化工作流程并增强与同事的协作。

为了充分利用本指南，建议读者具备命令行操作和通用编程原理的基础知识。熟悉终端或命令提示符的使用将极大地帮助您理解和应用本指南中提供的示例。此外，掌握版本控制的基本概念将增强您有效浏览本指南的能力。

# **基本 Git 命令**

在本节中，我们将深入探讨 Git 的基本命令，这些命令是高效管理和浏览 Git 仓库的基础。Git 是一个分布式版本控制系统，对于跟踪代码库的更改、与其他开发人员协作以及维护项目历史记录的完整性至关重要。对于任何希望在开发工作流程中充分利用 Git 功能的人来说，理解这些基本命令都至关重要。

我们将探索各种 Git 命令，涵盖 Git 使用的关键方面，例如初始化新仓库、提交更改、分支和合并。每个命令都用简短的句子来解释其用途，并通过实际示例来说明如何在实际场景中有效地使用它。无论您是创建新项目还是在现有代码库上工作，这些命令都能帮助您保持工作井然有序，并保持流畅的工作流程。

### git help

git help命令会打印 Git 帮助信息。它提供了 Git 基本用法和最常用 Git 命令的快速参考。当你需要快速回顾 Git 的功能或想要探索可用的命令时，此命令非常有用。

您还可以使用git help <command>显示任何特定 Git 命令的帮助信息。例如， git help git会打印特定于 Git 命令本身的帮助信息。

这些帮助命令对于初学者和有经验的用户来说都是宝贵的资源，可以快速访问有关 Git 的功能和用法的信息。

### git version

git version命令显示系统上安装的 Git 版本。此命令可用于验证当前正在使用的 Git 版本，这对于与某些功能的兼容性或解决问题非常重要。

### git init

git init命令用于在当前目录中初始化一个新的 Git 仓库。此命令会创建一个名为.git的新子目录，其中包含新仓库所需的所有元数据。这通常是您在启动要使用 Git 管理的新项目时运行的第一个命令。运行此命令后，您就可以开始在新的 Git 仓库中跟踪文件并进行提交。

### git clone <repository\_url>

git clone <repository\_url>命令会在本地计算机上创建远程 Git 仓库的副本。它会下载所有文件、分支和提交历史记录，让您可以立即开始项目工作。

### git status

git status命令显示 Git 仓库工作目录和git branch区的当前状态。它显示哪些文件已被修改、添加或删除，以及这些更改是否已git branch以进行下一次提交。

# **工作目录和git branch区**

工作目录和git branch区是 Git 中的基本概念，在版本控制过程中发挥着至关重要的作用。工作目录是您主动更改文件的环境，代表着项目的当前状态。它本质上是一个沙盒，您可以在开发项目时自由地编辑、删除和创建文件。但是，这些更改仅存在于您的本地计算机上，尚未成为版本历史记录的一部分。

另一方面，git branch区（也称为索引）充当工作目录和存储库之间的中间空间。它充当检查点，您可以在更改提交到存储库历史记录之前在其中选择性地组织更改。这使您可以准备一组逻辑上相关的更改，从而确保每次提交都有意义且连贯。

以下命令将简化工作目录和git branch区之间更改的管理。它们允许您将文件添加到git branch区、移除文件或修改现有文件，从而让您可以控制下次提交的内容。使用这些命令，您可以确保只提交预期的更新，从而使项目历史记录清晰有序。此过程对于维护清晰易懂的历史记录至关重要，因为它使您能够精确清晰地跟踪项目的进展。

### git checkout .

git checkout .命令会丢弃工作目录中的所有更改，将文件恢复到上次提交的状态。此命令对于快速撤消本地修改并将工作目录恢复到干净状态很有用。

### git reset -p

git reset -p命令允许你以交互方式重置工作目录中的更改。它提供了一种选择性撤消修改的方法，让你能够细粒度地控制哪些更改需要保留或丢弃。

### git add <文件>

git add <file>命令会将特定文件添加到 Git 的git branch区。这会为文件包含在下一次提交中做好准备，让您可以选择性地选择要包含在版本历史记录中的更改。

### git add -p

允许您通过将工作目录中的更改分成块（块）来以交互方式git branch更改，从而使您能够在提交之前检查更改的部分并将其有选择地添加到索引中。

### git add -i

进入添加文件的交互模式。提供基于文本的交互式菜单，您可以在其中选择要执行的各种操作，例如git branch各个更改、更新文件或查看状态。

### git rm <文件>

从工作目录中删除文件并git branch删除

### git rm --cached <文件>

从git branch区（索引）中删除指定的文件，但将其完整地保留在工作目录中，从而有效地从版本控制中取消对文件的跟踪。

### git mv <旧路径> <新路径>

git mv <old\_path> <new\_path>命令用于移动或重命名 Git 仓库中的文件或目录。它会自动git branch更改，为下一次提交做好准备。

### git commit -m “消息”

git commit -m "message"命令用于在 Git 仓库中创建一个新的提交。它会保存已git branch（添加到索引）的更改以及一条描述性消息。该消息应简要说明本次提交中进行了哪些更改。

# **使用分支**

git branch是 Git 代码库中并行的开发线，允许您独立于主代码库开发不同的功能、修复错误或进行实验。每个分支可以拥有自己的提交历史记录，并且在合并之前，一个分支中的更改不会影响其他分支。这有助于组织工作并促进协作，因为多个开发人员可以同时处理项目的不同方面，而不会干扰彼此的进度。

在本节中，我们将介绍一些命令，用于在 Git 仓库中创建、切换、列出、重命名和删除分支。这些命令有助于管理并行开发，让您能够独立开发不同的功能或修复错误。您还将学习如何显示提交历史记录和分支关系，以及如何管理远程分支。

### git branch <分支名称>

创建新分支

### git checkout <分支名称>

切换到指定分支并更新工作目录

### git branch

列出所有分支

### git branch -d <分支名称>

删除分支

### git push --delete <远程> <分支>

删除远程分支

### git branch -m <旧名称> <新名称>

重命名分支

### git checkout -b <新分支>

基于当前分支，创建并切换到名为 的新分支。

### git switch <分支>

将工作目录切换到指定分支

### git show-branch <分支>

显示所有或选定分支的提交历史和分支关系的摘要，显示每个分支的分歧位置

### git show-branch --all

与上述相同，但适用于所有分支及其提交。

### git branch -r

列出本地存储库知道的所有远程分支

### git branch -a

列出存储库中的所有分支，包括本地分支和远程分支（本地存储库知道的分支）

### git branch --合并

列出已完全合并到当前分支的所有分支，如果不再需要，可以安全删除

### git branch --no-merged

列出尚未完全合并到当前分支的所有分支，显示尚未集成更改的分支

# **合并**

git merge 命令用于将一个分支的更改合并到另一个分支。它会整合两个分支的历史记录，创建一个包含两个源更改的新提交。此过程可以将多条开发线整合在一起，促进协作并确保所有更新都合并到主项目中。合并过程中，如果更改重叠，可能会发生冲突，需要手动解决以确保最终结果的一致性。

### git merge <分支>

将指定分支的更改整合到当前分支，并结合它们的历史记录

### git merge --no-ff <分支>

将指定分支合并到当前分支，即使可以进行快速合并，也始终创建新的合并提交

### git merge --squash <分支>

将指定分支中的所有更改合并为一个提交，准备在当前分支中提交更改而不合并分支的历史记录，允许您手动编辑提交消息。

### git merge --abort

取消正在进行的合并过程并将工作目录和索引的状态恢复到合并开始之前的状态。

### git merge -s ours <分支>

### git merge —-strategy=ours <分支>

使用“我们的”策略执行合并，该策略保留当前分支的更改并丢弃来自指定分支的更改，有效地合并历史记录而不集成来自其他分支的更改。

### git merge --strategy=theirs <分支>

使用“theirs”策略将指定分支合并到当前分支，该策略通过支持合并分支的更改来解决所有冲突（注意：“theirs”策略不是内置策略，通常需要自定义脚本或与工具一起用于处理冲突解决）。

# 远程

Git 远程仓库是指远程仓库，即托管在互联网或其他网络上的项目版本。它们允许多个用户与中央仓库共享和同步更改，从而实现协作。远程仓库的常见操作包括：使用 git fetch 获取更新、使用 git pull 获取并合并更改，以及使用 git push 将本地提交上传到远程仓库。远程仓库的管理包括添加、删除和重命名远程连接，以及配置 URL 以实现无缝协作。

### git fetch

从远程存储库获取更改但不将其合并到当前分支

### git pull

从远程存储库获取更改并立即将其合并到当前分支

### git push

将本地分支的更改上传到远程存储库

### git remote

列出为本地存储库配置的远程存储库的名称

### git remote -v

显示与本地存储库关联的远程存储库的 URL，包括获取和推送 URL

### git remote add <名称> <网址>

将具有指定名称和 URL 的新远程存储库添加到本地存储库配置。

### git remote remove <名称>

### git remote rm <名称>

从本地 git 配置中删除指定的远程存储库连接

### git remote rename <旧名称> <新名称>

更改本地 git 配置中现有远程存储库连接的名称

### git remote set-url <名称> <新网址>

更改本地 git 配置中现有远程存储库连接的 URL

### git fetch <远程>

从指定的远程存储库检索最新更改，更新远程分支的本地副本，而不将其合并到本地分支中。

### git pull <远程>

从指定的远程存储库获取更改并将其合并到当前分支

### git remote update

获取存储库跟踪的所有远程的更新。

### git push <远程> <分支>

将指定分支从本地存储库上传到给定的远程存储库

### git push <远程> --delete <分支>

从远程存储库中删除指定的分支

### git remote show <远程>

显示指定远程存储库的详细信息，包括其 URL、获取和推送配置以及它跟踪的分支

### git ls-remote <存储库>

列出指定远程仓库的引用（例如分支和标签）及其提交 ID。此命令允许您在不克隆远程仓库的情况下查看其中可用的分支和标签。

### git push origin <分支> --set-upstream

将本地分支推送到远程存储库 origin，并设置本地分支来跟踪远程分支，因此将来的 git push 和 git pull 命令将默认使用此远程分支。

### git remote add upstream <存储库>

向本地仓库添加一个名为“upstream”的新远程仓库，指向指定的 。这通常用于跟踪你从中 fork 出来的原始仓库，而 origin 通常指的是你自己的 fork。

### git fetch upstream

从上游远程存储库检索更新，更新对该远程分支和标签的本地引用，而无需修改工作目录或合并更改。

### git pull upstream <分支>

从上游远程仓库获取更新，并将这些更改合并到当前分支。这通常用于将原始仓库的更改集成到您自己的本地分支。

### git push origin <分支>

将本地分支上传到原始远程存储库，使您的分支及其提交在远程可用。

# **修改提交**

修改 Git 提交允许您修改最近的提交，通常是为了更正或更新其内容或消息。您可以使用 git commit –amend 命令完成此操作，该命令会在您的默认文本编辑器中打开提交以查看更改。修改功能尤其适用于修复小错误或添加被遗忘的更改，而无需创建新的提交，从而获得更清晰、更准确的提交历史记录。

### git commit --amend

修改最近的提交，合并git branch的更改。

### git commit --amend -m“新消息”

修改最近一次提交的提交消息。

### git commit --fixup=HEAD

--fixup选项创建一个新的提交，用于更正或修改最近的提交 (HEAD)。新的提交在提交信息中会带有 fixup! 前缀，并将在交互式 rebase 期间用于自动修复或修改指定的提交。

# git branch

Git 储藏功能允许您临时保存工作目录中尚未准备好提交的更改。使用 git stash 命令，您可以搁置这些更改并将工作目录恢复到干净状态，从而能够在不丢失进度的情况下切换分支或执行其他任务。之后，您可以使用 git stash apply 或 git stash pop 重新应用储藏的更改，从而继续上次中断的操作。此功能对于管理正在进行的工作尤其有用，尤其是在您需要解决紧急问题或尝试不同的代码路径时。

### git stash

### git stash save

暂时保存未提交的更改，允许您切换分支或执行其他操作而无需提交未完成的工作。

### git stash -m“消息”

### git stash save “消息”

与上述相同，但使用消息存储更改

### git stash show

显示最近存储条目的更改摘要，显示哪些文件被修改

### git stash list

显示存储库中所有存储的更改，并以编号列表的形式显示它们

### git stash pop

应用最近的存储，然后立即将其从存储列表中删除

### git stash drop

从存储列表中删除最新的存储条目，而不将其应用到工作目录

### git stash apply

将最近存储的更改重新应用到工作目录，而不将其从存储列表中删除

### git stash clear

清除所有存储的条目，永久删除存储中所有已保存的更改

### git stash branch <分支>

创建一个以您存储更改之前的提交命名的新分支，然后将存储的更改应用到该新分支。此命令允许您在单独的分支中继续处理存储的更改，并保留原始上下文和更改。

# 标签

Git 标记功能允许您使用有意义的名称将存储库历史记录中的特定点标记为重要，通常用于发布或重要的里程碑。与分支不同，标记通常是不可变的，不会更改，可作为对特定提交的永久引用。Git 中有两种类型的标记：轻量级标记（指向提交的简单指针）和带注释的标记（存储额外的元数据，例如标记者的姓名、电子邮件、日期和消息）。您可以轻松创建、列出标记、将标记推送到远程存储库以及删除标记，从而为管理和引用项目开发时间线中的关键点提供了一种便捷的方式。

### git tag <标签名称>

创建具有指定名称的新标签，指向当前提交（通常用于标记提交历史记录中的特定点，例如发布）

### git tag -a <标签名> -m “消息”

创建具有指定名称和消息的带注释的标签，其中包括标记者的姓名、电子邮件和日期等其他元数据，并指向当前提交。

### git tag -d <标签名称>

从本地存储库中删除指定的标签

### git tag -f <标签> <提交>

强制标签指向不同的提交。

### git show <标签名>

显示指定标签的详细信息，包括它指向的提交以及任何相关的标签消息或注释

### git push origin <标签名>

将指定的标签上传到远程仓库，供其他人使用

### git push origin --tags

将本地所有标签推送到远程仓库，确保所有标签与远程同步

### git push --follow-tags

推送提交和标签。

### git fetch --tags

从默认远程存储库中检索所有标签并使用它们更新本地存储库，而不会影响您当前的分支。

# **恢复更改**

在 Git 中，还原更改是指撤消对仓库历史记录所做的修改。这可以通过多个命令来实现，例如 git revert，它会创建一个新的提交，并否定指定先前提交的更改，从而有效地撤消其效果，同时保留提交历史记录。另一种方法是使用 git reset，它会将当前 HEAD 更改为指定的提交，并可以根据所选选项（-soft、-mixed 或 -hard）更新git branch区和工作目录。此外，git checkout 可用于放弃工作目录中的更改，将文件还原到上次提交时的状态。这些工具为管理和更正更改提供了灵活性，确保仓库保持准确和整洁。

### git checkout -- <文件>

放弃工作目录中指定文件的更改，将其恢复到上次提交的状态，并有效撤消尚未git branch的任何修改。

### git revert <提交>

创建一个新的提交，撤消指定提交中的更改，有效地逆转其影响，同时保留历史记录。

### git revert -n <提交>

撤销提交但不提交结果。

### git 重置

将当前 HEAD 重置为指定状态，并选择性地更新git branch区和工作目录，具体取决于所使用的选项（-soft、-mixed 或 -hard）。

### git reset --soft <提交>

将 HEAD 移动到指定的提交，同时保持索引（git branch区）和工作目录不变，这样指定提交之后的所有更改都将保留在git branch区等待提交。当你想撤消提交但保留更改以便再次提交时，此功能非常有用。

### git reset --mixed <提交>

将 HEAD 移动到指定的提交并更新索引（git branch区）以匹配该提交，但保持工作目录不变，因此指定提交之后的更改将保留但不会被跟踪。

### git reset --hard <提交>

将 HEAD 移动到指定的提交并更新索引（git branch区）和工作目录以匹配该提交，丢弃指定提交后的所有更改和未跟踪的文件。

# **查看历史记录**

Git 历史记录指的是对仓库在一段时间内所做的所有更改的记录。它包含按时间顺序排列的提交，每个提交代表仓库在特定时间点的快照。通过提供详细的更改记录（包括更改者、更改时间和更改原因），这些历史记录可以帮助开发者追踪修改、了解代码库的演变，并实现高效的协作。像 git log 这样的工具可以帮助浏览这些历史记录，深入了解开发过程，并辅助调试和项目管理。

### git log

显示提交日志

### git log --oneline

显示每行提交的摘要。

### git log --graph

显示提交历史的图形表示。

### git log --stat

显示文件统计信息以及提交历史记录。

### git log --pretty=format:"%h %s"

根据指定的格式格式化日志输出。

### git log --pretty=format:“%h - %an，%ar：%s”

提供更易于阅读的日志格式

### git log --author=<作者>

显示指定作者所做的提交。

### git log --before=<日期>

显示在指定日期之前所做的提交。

### git log --after=<日期>

### git log --since=<日期>

显示在指定日期之后的提交。

### git log --cherry-pick

忽略两个分支之间等效的提交。

### git log --follow <文件>

显示文件的提交，包括重命名。

### git log --show-signature

显示提交的 GPG 签名信息。

### git shortlog

总结作者的 git log 输出。

### git shortlog -sn

按作者总结 git log 输出，并包含提交计数。

### git log --simplify-by-decoration

仅显示标签或分支引用的提交。

### git log --no-merges

从日志中忽略合并提交。

### git whatchanged

以类似于提交日志的格式列出提交数据。

### git diff-tree --pretty --name-only --root <提交>

显示提交树的详细信息。

### git log --first-parent

仅显示当前分支的提交并排除从其他分支合并的提交。

# **差异**

Git 差异是 Git 的一项功能，可让您查看代码库中不同状态之间的差异。这包括工作目录与git branch区之间的差异、git branch区与上次提交之间的差异，以及任意两次提交或分支之间的差异。通过逐行显示文件中的更改，差异可以帮助您在提交、合并或应用更改之前检查修改，从而确保代码库的准确性和一致性。

### git diff

显示存储库中不同状态之间的差异，例如工作目录与索引（git branch区）之间的差异、索引与上次提交之间的差异，以及两次提交之间的差异。它逐行显示文件中的更改，帮助您在提交或合并之前检查修改。

### git diff --stat

显示工作目录和索引（git branch区）之间的更改摘要，帮助您查看已修改的文件以及已添加或删除的行数。

### git diff --stat <提交>

查看提交和工作目录之间的更改

### git diff --stat <提交1> <提交2>

提供两次提交之间的更改摘要，显示哪些文件被更改以及它们之间的更改程度。

### git diff --stat <分支1> <分支2>

总结两个分支之间的差异，指出更改的文件和更改的幅度。

### git diff --name-only <提交>

仅显示在指定提交中更改的文件的名称。

### git diff --cached

显示git branch更改（索引）与上次提交之间的差异，帮助您查看下次提交将包含的内容

### git diff HEAD

显示工作目录和最新提交（HEAD）之间的差异，让您查看自上次提交以来所做的更改

### git diff <分支1> <分支2>

显示两个分支末端之间的差异，突出显示每个分支末尾提交之间的变化

### git difftool

启动差异工具来比较变化。

### git difftool <提交1> <提交2>

使用 diff 工具显示两个指定提交之间的差异。

### git difftool <分支1> <分支2>

打开差异工具来比较两个分支之间的变化。

### git cherry <分支>

将当前分支的提交与其他分支的提交进行比较，并显示每个分支独有的提交。它通常用于识别一个分支中哪些提交尚未应用到另一个分支。

# **Git 流程**

Git Flow 是 Git 的一种分支模型，它提供了一个强大的框架来管理大型项目。它定义了一套严格的分支策略，并根据项目发布周期进行设计，包含两个主要分支（主分支和开发分支）以及用于功能、发布和修补程序的支持分支。该模型有助于组织工作，确保历史记录清晰易管理，并通过明确定义不同类型开发工作的角色和流程来促进协作。

### git flow init

为 git-flow 分支模型初始化存储库。

### git flow feature start <功能>

在 git-flow 中启动一个新的功能分支。

### git flow feature finish <功能>

在 git-flow 中完成一个功能分支。

# **探索 Git 引用**

Git 引用（通常称为 refs）是指向 Git 仓库中特定提交或对象的指针。这些引用可以包括分支、标签和其他引用，例如 HEAD，它指向工作目录中签出的当前提交。引用用于跟踪仓库的结构和历史记录，使 Git 能够高效地管理和导航项目时间线中的不同点。它们提供了一种命名和引用特定提交的方法，使使用和操作仓库历史记录变得更加容易。

### git show-ref --heads

列出所有头部（分支）的引用。

### git show-ref --tags

列出所有标签的引用。

# **配置**

Git 配置涉及设置各种选项和首选项，用于控制 Git 环境的行为。这包括指定用户名和电子邮件地址、设置默认文本编辑器、为常用命令创建别名以及配置全局忽略文件。配置设置可以应用于不同级别：全局（影响系统上的所有仓库）、本地（影响单个仓库）和系统范围。这些设置可确保定制且一致的用户体验，简化工作流程，并提高版本控制操作的整体效率。

### git config --global [user.name](http://user.name/) "您的姓名"

在全局级别设置用户名。

### git config --global [user.email](http://user.email/) "your\_email@example.com"

在全球范围内设置用户电子邮件。

### git config --global core.editor <编辑器>

设置默认文本编辑器。

### git config --global core.excludesfile <文件>

设置全局忽略文件。

### git config --list

列出所有配置设置。

### git config --list --show-origin

列出所有配置变量，显示它们的来源。

### git config <键>

检索指定键的值。

### git config --get <键>

检索指定配置键的值。

### git config --unset <键>

删除指定的配置键。

### git config --global --unset <key>

全局删除指定的配置键。

# **安全**

Git GPG 安全性涉及使用 GNU Privacy Guard (GPG) 对提交和标签进行签名，以确保其真实性和完整性。通过配置 GPG 密钥并启用自动签名，开发人员可以验证提交和标签是否由可信来源创建，从而防止篡改并确保存储库历史记录的完整性。此做法通过提供加密保证，确保更改来自合法贡献者，从而增强了安全性。

### git config --global user.signingKey <key>

配置用于签署提交和标签的 GPG 密钥。

### git config --global commit.gpgSign true

自动使用 GPG 签署所有提交。

# **设置别名**

Git 别名是您可以创建的自定义快捷方式，通过将较长的 Git 命令映射到更短、更易记的名称来简化和加快您的工作流程。通过在 Git 设置中配置别名，您可以快速执行常用命令，减少输入次数。这不仅提高了工作效率，还降低了出错的可能性。例如，您可以设置别名，例如 git st 来替换 git status，或 git co 来替换 git checkout。别名可以全局定义以应用于所有存储库，也可以本地定义以应用于单个项目，从而灵活地简化 Git 操作。

### git config --global alias.ci 提交

将 git ci 设置为 git commit 的别名。

### git config --global alias.st 状态

将 git st 设置为 git status 的别名。

### git config --global alias.co 签出

将 git co 设置为 git checkout 的别名。

### git config --global alias.br 分支

将 git br 设置为 git branch 的别名。

### git config --global alias.graph“log --graph --all --oneline --decorate”

为存储库历史记录的详细图表创建别名。

# **变基**

Git 变基会将您的更改重新应用到另一个分支的历史记录之上，从而创建更清晰、更线性的项目历史记录。实际上，这有助于顺利集成更新，避免不必要的合并提交，确保提交顺序清晰明了，并使项目的演变过程更容易理解。

### git rebase <分支>

git rebase 命令用于将提交重新应用到另一个基础提交之上。它允许你将一系列提交移动或合并到一个新的基础提交中。它通常用于：

1. 保持线性项目历史。
2. 将一个分支的变更集成到另一个分支。
3. 使用主分支的最新更改来更新功能分支。

基本用法是 git rebase ，它将当前分支重新定位到指定的分支上。

### git rebase --interactive <分支>

启动交互式 rebase 会话，允许你修改从当前 HEAD 开始的提交。这允许你重新排序、压缩、编辑或删除提交，从而提供一种在推送更改之前清理和优化提交历史记录的方法。简化版本： git rebase -i <branch>

### git rebase --continue

解决冲突后继续重新定基过程。

### git rebase --abort

中止变基过程并返回到原始分支。

### git fetch --rebase

从远程存储库获取并重新设置本地更改。

# **挑选**

Git 的 Cherry-Picking 功能允许您将特定提交引入的更改从一个分支应用到另一个分支。当您想选择性地合并来自不同分支的个别更改而不合并整个分支时，此功能尤其有用。通过使用 git cherry-pick 命令，您可以隔离并仅集成所需的提交，确保特定修改包含在当前分支中，同时避免来自分支其他部分的潜在冲突和不必要的更改。

### git cherry-pick <提交>

应用现有提交引入的更改。

### git cherry-pick --continue

解决冲突后继续挑选。

### git cherry-pick --abort

中止挑选过程。

### git cherry-pick --no-commit <提交>

挑选一个提交，但不自动提交，并允许进一步修改。简短版本： git cherry-pick -n <commit>

# **修补**

Git 补丁是一种将更改从一个仓库应用到另一个仓库，或将同一仓库内的一个分支应用到另一个分支的方法。它涉及创建补丁文件，这些文件是表示提交或分支之间差异的文本文件。然后，可以使用 git apply 或 git am 等命令将这些补丁文件应用到仓库，这样无需直接合并分支即可传输和集成更改。补丁对于在不同代码库之间共享特定的更改或更新特别有用，可确保仅应用预期的修改。

### git apply <补丁文件>

将补丁文件的更改应用到工作目录。

### git apply --check

检查是否可以干净地应用补丁。

### git format-patch <since\_commit>

为自指定提交以来的每次提交创建补丁文件。

### git am <补丁文件>

从邮箱应用补丁。

### git am --continue

解决冲突后继续应用补丁。

### git am --abort

中止补丁应用过程。

### git diff > <文件.补丁>

根据差异创建补丁文件。

# **相对日期**

Git 相对日期允许用户使用易于理解的时间表达式来引用代码库历史记录中的特定时间点。例如，像 main@{1.week.ago} 或 @3.days.ago 这样的命令允许您访问分支的状态，或查看相对于当前日期的某个时间段以来所做的更改。此功能通过使用“昨天”、“两周前”或具体日期等直观的术语，简化了代码库时间线的导航，让您更轻松地跟踪和管理代码库的演变，而无需记住确切的提交哈希值或时间戳。

### git show main@{1.week.ago}

查看一周前主分支的状态：

### git diff @{3天前}

查看您在过去 3 天内所做的更改：

### git checkout main@{2周前}

检查一下两周前的存储库：

### git log @{1.month.ago}..HEAD

查看从 1 个月前到现在的提交日志

### @{2024-06-01}

### @{昨天}

### @{"1周2天前"}

其他用法示例。

# **责备**

Git Blaming 是 Git 的一项功能，它可以识别文件每一行的最后修改，并将更改归因于特定的提交和作者。此功能使用 git blame 命令完成，该命令提供文件的详细注释，显示更改的执行者和更改时间。此工具对于跟踪文件历史记录、了解代码演变以及识别错误或更改的来源特别有用。通过精确定位每行代码的提交和作者，开发人员可以深入了解开发过程，并促进团队内部的协作和问责。

### git blame <文件>

显示文件每一行的最后修改。

### git blame <文件> -L <开始>,<结束>

将 blame 输出限制在指定的行范围内。

### git blame <文件> <提交>

显示指定提交之前的责任信息。

### git blame <文件> -C -C

显示文件每一行最后修改的修订版本和作者，并带有复制检测。

-C选项用于检测同一文件中移动或复制的行。使用一次 ( -C ) 即可检测同一文件中移动或复制的行。使用两次( -C -C ) 则会让 git 检查未修改的文件作为复制源的候选文件。这意味着它不仅会尝试在同一个文件中查找复制行的来源，还会在其他文件中查找。

### git blame <文件> --reverse

向后操作，显示谁最后修改了指定文件中的每一行。

### git blame <文件> --first-parent

显示最近修改文件中每一行的人，仅遵循合并更改的第一个父提交。

# **归档**

Git 归档功能允许您创建包含特定提交、分支或标签内容的归档文件（例如 .tar 或 .zip）。此功能可用于打包特定时间点的仓库快照，让您能够分发或备份仓库的状态，而无需包含完整的 Git 历史记录。git archive 命令通常用于此目的，它提供了一种便捷的方法，可以将项目的当前状态导出为可移植的格式。

### git archive <格式> <树状>

创建一个存档文件（例如 .tar 或 .zip 文件），其中包含指定树状结构（例如提交、分支或标签）的内容，并以给定格式保存。例如：

git archive --format=tar HEAD创建当前提交（HEAD）的 .tar 档案。

git archive --format=zip v1.0创建 v1.0 标签中的文件的 .zip 档案。

此命令对于在特定时间点打包存储库的快照很有用。

# **追踪**

Git 跟踪是指监控和管理存储库中文件的过程。git ls-files 命令列出 Git 正在跟踪的所有文件，从而清晰地显示当前处于版本控制下的文件。另一方面，git ls-tree 命令显示指定分支的树状对象内容，从而展示存储库中该位置的结构和文件。这些命令共同帮助开发人员了解存储库中包含哪些文件以及它们的组织方式，从而确保高效地跟踪和管理项目的代码库。

### git ls-files

列出所有跟踪的文件。

### git ls-tree <分支>

列出树对象的内容。

# **指数操纵**

Git 索引操作涉及管理git branch区（也称为索引），用于在提交之前准备更改。这包括将文件标记为“假定未更改”以暂时忽略更改，或重置这些标记以再次跟踪更改。索引操作命令（例如 git update-index）允许您控制下次提交时包含哪些文件，从而灵活地处理更改并优化特定任务的工作流程。

### git update-index --assume-unchanged <文件>

将文件标记为假定未改变。

### git update-index --no-assume-unchanged <文件>

取消标记文件为假定未更改。

# **挤压**

Git 压缩是将多个提交合并为一个提交的过程。这通常用于在将更改合并到主分支之前清理提交历史记录，使历史记录更简洁易读。压缩可以使用交互式变基命令 (git rebase -i) 执行，该命令允许开发人员选择性地合并、重新排序或编辑提交。通过压缩提交，可以整合冗余或较小的更改，从而更清晰地呈现开发过程。

### git rebase -i HEAD~<n>

以交互方式压缩提交。

# **数据完整性**

Git 数据完整性是指 Git 用于确保存储库中数据准确性和一致性的机制和流程。Git 使用加密哈希（SHA-1 或 SHA-256）来唯一标识提交、树和 Blob 等对象。这种哈希不仅为每个对象提供唯一标识符，还能确保对对象内容的任何修改都会导致不同的哈希值，从而检测任何损坏或篡改行为。像 git fsck 这样的命令可用于验证数据库中对象的连接性和有效性，从而确保存储库的整体健康和完整性。

### git fsck

验证数据库中对象的连接性和有效性。

### git fsck --unreachable

查找存储库中无法通过任何引用访问的对象。

### git prune

删除无法访问的对象

### git gc

运行垃圾收集过程。

Git 垃圾收集是一个维护过程，它通过移除不必要的文件并压缩文件修订来清理和优化存储库，从而节省空间。此过程由 git gc 命令触发，会合并并删除无法访问的对象（例如孤立提交和未引用的 blob），从而确保存储库保持高效和高性能。定期垃圾收集有助于有效管理存储，并保持存储库结构的井然有序。

# **清理**

Git 中的清理工作包括移除不必要的文件、引用和分支。这有助于保持仓库的井然有序和高效运行。定期进行清理活动（例如修剪远程跟踪分支、删除未跟踪的文件以及移除过时的引用）可确保您的仓库保持易于管理且整洁有序。实际上，这些操作可以提升性能、降低存储需求，并使项目导航和工作更加便捷。

### git fetch --prune

删除远程上不再存在的引用。

### git remote prune <名称>

修剪所有陈旧的远程跟踪分支。

### git fetch origin --prune

清理远程存储库中的过时的引用。

### git clean -f

从工作目录中删除未跟踪的文件，强制删除未被 Git 跟踪的文件。

### git clean -fd

从工作目录中删除未跟踪的文件和目录，包括 Git 未跟踪的任何文件和目录。

### git clean -i

进入交互模式来清理未跟踪的文件。

### git clean -X

仅从工作目录中删除被忽略的文件。

# **子树**

Git 子树是一种用于管理子项目并将其集成到主仓库的机制。与将子项目视为拥有独立仓库的独立实体的子模块不同，子树允许您将其他仓库的内容直接包含在主仓库的子目录中。这种方法简化了工作流程，无需多个仓库，并支持无缝集成、合并和从子项目拉取更新。子树提供了一种灵活便捷的方式来管理依赖项，并在需要合并外部代码库的项目上进行协作。

### git subtree add --prefix=<目录> <存储库> <分支>

添加存储库作为子树。

### git subtree merge --prefix=<目录> <分支>

合并子树。

### git subtree pull --prefix=<目录> <存储库> <分支>

从子树的存储库中提取新的更改。

# **搜索**

git grep 是 Git 中一个强大的搜索命令，允许用户在仓库的文件中搜索特定的字符串或模式。它会搜索工作目录和索引，快速高效地定位多个文件中指定模式的出现位置。对于希望在项目中查找代码、注释或文本实例的开发人员来说，此命令尤其有用，使他们能够轻松浏览和理解大型代码库。git grep 拥有各种选项和标志，可以执行有针对性的搜索，使其成为代码分析和维护的重要工具。

### git grep <模式>

在工作目录和索引中搜索字符串。

### git grep -e <模式>

搜索特定模式。

# **二等分**

Git bisecting 是一款强大的调试工具，可以帮助识别项目中引入错误或问题的具体提交。通过对提交历史记录进行二分搜索，git bisect 可以有效地缩小潜在问题提交的范围。该过程包括标记一个已知的正确提交和一个已知的错误提交，然后反复测试中间提交以确定它们是正确还是错误。这种迭代方法可以快速隔离错误提交，使开发人员能够精确定位导致问题的确切更改，从而实现更快、更准确的调试。

### git bisect start

开始二分会话。

### git bisect bad

将当前版本标记为不良版本。

### git bisect good <提交>

将指定的提交标记为良好。

### git bisect reset

结束二分会话并返回原始分支。

### git bisect visualize

启动一个可视化工具来协助平分。

# **属性**

Git 属性是定义 Git 如何处理仓库中特定文件或路径的设置。这些属性定义在名为 .gitattributes 的文件中，它们可以控制各种行为，例如文本编码、行尾规范化、合并策略和差异算法。通过设置属性，您可以确保在不同环境和协作者之间保持一致的行为，从而更轻松地管理具有特殊要求或复杂性的文件。例如，您可以将某些文件标记为二进制文件，以防止 Git 尝试合并它们，或者指定自定义差异驱动程序以进行更有意义的比较。

### git check-attr <属性> -- <文件>

显示 .gitattributes 配置中定义的给定文件的特定属性的值，帮助您了解 Git 如何根据文本编码、合并行为或差异处理等属性处理文件。

# **查看**

git checkout 是 Git 中的一个多功能命令，用于在仓库内的不同分支、标签或提交之间切换。通过更新工作目录和索引以匹配指定的分支或提交，它允许您查看或操作该点的仓库状态。此外，git checkout 还可用于创建新分支、从提交中恢复特定文件，甚至使用 –orphan 选项创建一个没有历史记录的新分支。此命令对于导航和管理项目代码库的不同版本至关重要。

### git checkout <提交>

更新工作目录和索引以匹配指定的提交，允许您查看或使用该提交的存储库状态，但它会使您处于“分离的 HEAD”状态，这意味着您不在任何分支上

### git checkout -b <分支> <提交>

从指定的提交开始创建一个名为的新分支并切换到该分支，允许您从提交历史记录中的该点开始工作。

### git checkout <提交> -- <文件>

将指定文件从特定提交恢复到您的工作目录中，用该提交的版本替换文件的当前版本，而不更改提交历史记录或索引。

### git checkout --orphan <新分支>

创建一个名为 的新分支，没有提交历史记录，有效地启动一个以干净的工作目录和索引开始的新分支，就好像它是一个新的存储库一样。

# **引用日志**

Git reflog 是一个强大的工具，它记录了 Git 仓库中分支末端和 HEAD 引用的所有更改。这些更改包括提交、检出、合并和重置等操作。通过维护这些更改的历史记录，reflog 允许用户跟踪最近的修改并恢复丢失的提交，即使它们不包含在当前分支历史记录中。它提供了一种浏览仓库状态变化的方法，使其成为调试和撤销错误的宝贵资源。

### git reflog

显示对 HEAD 引用和分支提示的所有更改的日志，包括提交、签出、合并和重置，允许您恢复丢失的提交或跟踪存储库状态的最近更改

### git reflog show <ref>

显示指定引用 () 的 reflog，显示该引用的更改日志，包括 HEAD 或分支提示的更新，以及相关的提交消息和时间戳。

# **处理未跟踪的文件**

### git 清理

从工作目录中删除未跟踪的文件和目录。默认情况下，它仅显示要删除的内容，而不会实际删除任何内容。要执行实际清理，您需要使用其他标志：

* git clean -f ：删除未跟踪的文件。
* git clean -fd ：删除未跟踪的文件和目录。
* git clean -fx ：删除未跟踪的文件，包括被 .gitignore 忽略的文件。
* git clean -n ：显示哪些文件将被删除但实际上并未删除它们。

# **强制推动**

### git push --force

强制将本地分支推送到远程仓库，即使这会导致非快进式合并。这会用本地更改覆盖远程分支。当您重写了历史记录（例如，使用 rebase）并需要更新远程分支以匹配本地分支时，此功能可能很有必要，但它也可能会覆盖其他人的更改，因此应谨慎使用。

# **获取和拉取**

### git fetch --all

从为本地存储库配置的所有远程存储库中检索更新，从所有分支和标签中获取更改，而无需修改本地分支

### git pull --rebase

从远程仓库获取更改，并将本地提交变基到更新后的远程分支上，而不是合并它们。这可以使提交历史记录保持线性，并避免不必要的合并提交。

# **处理合并冲突**

在 Git 中处理合并冲突是与多位贡献者协作开展项目的关键技能。当不同分支或提交中的更改重叠或相互矛盾，导致无法自动合并时，就会发生合并冲突。解决这些冲突需要审查并手动协调差异，以确保最终代码准确整合各方贡献。实际上，有效地管理合并冲突有助于维护代码完整性，并通过确保每个人的更改都正确地纳入项目历史记录来促进顺畅的协作。

### git mergetool

启动合并工具，帮助您解决合并或变基期间出现的冲突。它会打开图形界面或 Git 设置中配置的文本工具，让您手动解决冲突并完成合并。

### git rerere

rerere 代表“重用已记录的解决方案”，这项功能可帮助您通过重用之前记录的冲突解决方案来自动解决将来合并或变基中的冲突。启用后，Git 会记录您解决冲突的方式，如果再次出现相同的冲突，它可以自动应用相同的解决方案。

# **工作树**

Git 中的工作树允许您将多个工作目录与单个存储库关联。这对于同时处理多个分支且无需在同一目录中不断切换分支的情况尤其有用。通过使用工作树，您可以在隔离的环境中轻松管理不同的功能、错误修复或实验，从而提高工作流程效率并降低冲突风险。

### git worktree add../new-branch feature-branch

根据“feature-branch”在名为“new-branch”的目录中创建一个新的工作树。

### git worktree list

列出与当前存储库关联的所有工作树，显示它们的路径和它们签出的分支。

### git worktree remove <路径>

删除给定处的指定工作树，删除工作目录并分离分支。

### git worktree prune

删除对不存在的工作树的引用，清理工作树列表。

### git worktree lock <路径>

在给定的位置锁定指定的工作树，以防止其被修剪。

### git worktree unlock <路径>

在给定的时间解锁指定的工作树，以便在必要时对其进行修剪。

# **子模块**

Git 中的子模块是一种在您自己的仓库中包含和管理外部仓库的方法。它们对于跨多个项目复用代码、维护依赖项或集成第三方库特别有用。通过使用子模块，您可以保持主仓库的整洁和模块化，同时仍然确保所有必要的组件都已包含并进行版本控制。

### git submodule init

初始化仓库中的子模块。此命令设置子模块所需的配置，但实际上并不克隆它们。

### git submodule update

克隆并检出子模块到指定路径。通常在 git submodule init 之后运行。

### git submodule add <存储库> <路径>

将新的子模块添加到指定路径的存储库，并将其链接到指定的存储库。

### git submodule status

显示所有子模块的状态，显示它们的提交哈希以及它们是否是最新的、已修改的还是未初始化的。

### git submodule foreach <命令>

在每个子模块中运行指定的命令。这对于跨所有子模块执行批量操作非常有用。

### git submodule sync

将配置文件中的子模块 URL 与 .gitmodules 文件中的子模块 URL 同步，确保它们是最新的。

### git submodule deinit <路径>

注销指定的子模块，并删除其配置。这不会删除子模块的工作目录。

### git submodule update --remote

从远程存储库获取子模块并将其更新为最新提交。

### git submodule set-url <路径> <新url>

将指定子模块的 URL 更改为新的 URL。

### git submodule absorptiongitdirs

将子模块的 Git 目录吸收到超级项目中以简化结构。