GitHub Pull Request 命令行工作流指南

Chapter 1: 现代协同开发的基础原则

1.1. 什么是版本控制？协同工作中的挑战

版本控制，亦称源代码控制，是追踪与管理软件代码变更的实践[1]。版本控制系统（Version Control System, VCS）是一种软件工具，旨在帮助软件团队管理源代码随时间推移而发生的变化[1]。这些系统在现代软件开发，特别是 DevOps 团队中扮演着至关重要的角色，它们有助于缩短开发周期并提高部署成功率[1]。

其核心目的在于，版本控制软件在一个专门的数据库中记录了对代码的每一次修改[1]。这一机制使得开发人员在犯错时能够安全地回溯到代码的早期版本，从而在对团队造成最小干扰的情况下修复错误[1]。源代码被视为一项宝贵的资产，版本控制能够保护它免受灾难性数据丢失和因人为失误或意外后果导致的代码质量下降的影响[1]。

在团队环境中，开发人员持续不断地编写新代码和修改现有代码，这些工作常常跨越项目文件结构的多个部分[1]。版本控制通过追踪每个人的贡献来管理这些并发的变更，并有效防止冲突的发生[1]。它确保了由不同开发者所做的、可能不兼容的变更能够被有序地发现和解决，而不会阻碍团队其他成员的进度[1]。此外，由于任何代码变更都可能引入新的错误，版本控制系统促进了测试与开发的结合，直到一个新的稳定版本准备就绪[1]。

使用版本控制系统被认为是高效软件和 DevOps 团队的最佳实践，它能让开发者工作得更快，并帮助软件团队在规模扩大时保持效率和敏捷性[1]。版本控制系统主要带来以下三大核心优势：

1. 完整的长期变更历史：VCS 维护了每个文件每一次变更的全面历史记录，包括文件的创建、删除和内容编辑，这些变更可能由多人在多年间完成[1]。历史记录通常包含作者、日期以及解释每次变更目的的书面说明。这一完整的历史对于分析错误的根本原因、修复旧版本软件中的问题以及理解代码库的演进至关重要[1]。
2. 分支与合并：版本控制允许团队成员（甚至个人）在独立的工作流中进行开发[1]。“分支”（Branching）使这些工作流保持隔离，而“合并”（Merging）则提供了将它们重新整合的功能，

允许开发者验证不同分支上的变更不会相互冲突[1]。
3. 可追溯性：VCS 能够追踪每一次代码变更，并将其与项目管理和缺陷跟踪软件（如 Jira）相关联[1]。为每次变更附上描述性的信息，有助于根本原因分析和其他追溯性任务。在代码审查时，拥有这份带注释的历史记录有助于开发者理解代码的设计意图，从而做出更正确、更协调的修改，并提高未来工作的估算准确性[1]。

尽管在技术上没有版本控制也可以开发软件，但这样做会使项目面临巨大的风险，这是任何专业团队都不会建议的[1]。因此，问题不在于是否使用版本控制，而在于选择实施哪一个系统[1]。

1.2. Git 简介：一种分布式方法

Git 是一个免费、开源的分布式版本控制系统（Distributed Version Control System, DVCS），由 Linux 操作系统内核的创建者 Linus Torvalds 于 2005 年开发[2]。它被设计用来快速高效地处理从小型到非常大型的各种项目[2]。

与传统的集中式版本控制系统（Centralized Version Control Systems, CVCS），如 Subversion 或 CVS 相比，Git 的架构有着根本性的不同。在 CVCS 模型中，存在一个单一的中央服务器，存放着项目的完整版本历史[4]。开发者从这个中央服务器检出文件的一个快照进行工作。而在 DVCS 模型中，客户端不仅仅是检出文件的最新快照，而是完整地镜像（mirror）整个仓库，包括其完整的历史记录[3]。这意味着每个开发者的工作副本本身就是一个功能完备的仓库[4]。

这种分布式架构带来了显著的优势：

1. 每次克隆都是完整备份：如果用于协作的中央服务器发生故障，任何一个客户端仓库都可以被复制回服务器以进行恢复[4]。这意味着每一次克隆（clone）都相当于对所有项目数据的一次完整备份，从而减轻了 CVCS 中存在的单点故障风险[4]。
2. 增强的协同工作流：许多 DVCS 能够有效处理多个远程仓库，允许用户在同一个项目中以多种不同的方式与不同的人群同时协作[4]。这使得建立多样化的工作流程成为可能，包括集中式系统无法实现的层级模型[4]。
3. 卓越的性能与离线工作能力：由于每个开发者的本地副本都是一个完整的仓库，绝大多数操作，如提交新变更、创建分支、合并分支以及比较历史版本等，都在本地进行，因此速度极快[3]。这种本地化的特性也意味着开发者可以在没有网络连接的情况下继续工作，并在准备就绪后一次性推送所有变更[3]。

Git 的设计理念根植于性能、安全性和灵活性[3]。它已成为行业内的既成标准，这意味着大量的开发者已经具备 Git 经验，并且有众多的第三方工具和服务与之集成[3]。

1.3. 理解 GitHub：协同开发的平台

在学习 Git 的过程中，一个经常与之同时出现的名称是 GitHub。区分这两者至关重要：Git 是工具，而 GitHub 是服务 [5]。

GitHub 是一个基于 Web 的平台，它为 Git 仓库提供托管服务，并围绕这些仓库构建了一整套强大的协同开发工具 [5]。开发者可以在 GitHub 上寻找开源项目、为之做出贡献，并与全球的开发者社区进行连接和协作 [5]。从本质上讲，Git 负责在你的本地计算机上发生的一切与版本控制相关的事务，而 GitHub 则提供了一个云端的中心位置，用于存储你的仓库副本，并促进团队成员之间的互动 [7]。

虽然 Git 本身是一个命令行工具，但 GitHub 提供了一个图形化的 Web 界面，使得诸如代码审查、问题跟踪、项目管理等协同任务变得直观和高效 [7]。正是 Git 的分布式能力与 GitHub 平台功能的结合，催生了一种强大的、现代化的软件开发协作模式。这种模式不仅是开源社区的基石，也深刻地影响了全球范围内的远程工作和企业级软件开发。它允许开发者以一种异步的方式进行协作——开发者可以在本地独立工作，不受网络或中央服务器状态的限制，并通过 GitHub 的功能（如稍后将详述的“拉取请求”）在方便的时候与团队同步和讨论他们的工作。

1.4. 核心词汇：入门简介

在深入探讨具体命令之前，预先了解一些核心术语将有助于后续内容的理解。这些概念将在后续章节中进行详细阐述，此处仅作简要介绍。

- 仓库 **(Repository)**：仓库是 GitHub 的最基本元素。最简单的理解方式是将其想象成一个项目的文件夹 [8]。一个仓库包含了项目的所有文件（包括文档），并存储了每个文件的修订历史 [8]。仓库可以有多个协作者，并且可以是公开的或私有的 [8]。
- 提交 **(Commit)**：一次提交，或称“修订”，是对一个或一组文件的单次更改 [8]。当你进行一次提交来保存你的工作时，Git 会创建一个唯一的 ID（也称为“SHA”或“哈希”），它允许你记录下具体的变更内容、变更者以及变更时间 [8]。提交通常包含一条提交信息，这是对所做变更的简要描述 [8]。
- 分支 **(Branch)**：分支是仓库的一个并行版本 [8]。它包含在仓库内部，但不会影响主分支（或称
  main 分支），从而允许你自由地工作而不会干扰“线上”版本 [8]。当你完成了你想要做的变更后，你可以将你的分支合并回主分支以发布你的更改 [8]。
- 拉取请求 **(Pull Request)**：拉取请求是由用户提交的、提议对仓库进行变更的请求，它会被仓库的协作者接受或拒绝 [8]。与问题（Issues）类似，每个拉取请求都有其自己的讨论论坛，是代码审查和协作讨论的核心场所 [8]。

Chapter 2: Git 的心智模型：深入理解三个区域

要真正掌握 Git，最关键的一步是建立一个正确的心智模型。与其他一些系统不同，Git 的核心工作流并非简单地在“工作区”和“版本库”之间传递文件。它引入了一个中间层，这个设计是理解 Git 大部分命令的关键。若想顺利地学习后续内容，理解 Git 文件所处的三种主要状态至关重要 [10]。

2.1. 三种主要状态与三个核心区域

在 Git 的世界里，你的文件可以处于以下三种主要状态之一 [10]：

- 已修改 **(Modified)**：表示你已经修改了文件，但尚未将其提交到你的数据库中 [10]。
- 已暂存 **(Staged)**：表示你已经标记了一个已修改文件的当前版本，以使其包含在你的下一次提交快照中 [10]。
- 已提交 **(Committed)**：表示数据已经安全地存储在你的本地数据库中 [10]。

这三种状态直接对应于一个 Git 项目的三个主要工作区域：工作区、暂存区和 Git 目录 [10]。

- 工作区 (Working Directory / Working Tree)：
  工作区是项目某个版本的一个单独检出（checkout）10。当你从 Git 仓库中“检出”一个版本时，这些文件会从 Git 目录中被提取出来，以其原始的、非压缩的形式放置在你的硬盘上，供你使用或修改 10。这正是你在文件浏览器中看到的、可以直接用编辑器打开和修改的项目文件夹 11。
- 暂存区 (Staging Area / Index)：
  暂存区是一个文件，通常包含在你的 Git 目录中，它存储了关于下一次提交将包含哪些内容的信息 10。它的技术名称是“索引”（index），但“暂存区”这个说法同样有效且更易于理解 10。它是一个中间区域，在你正式提交变更之前，用于准备和组织这些变更 11。
- Git 目录 (Git Directory / Repository)：
  Git 目录是 Git 为你的项目存储元数据和对象数据库的地方 10。这是 Git 最重要的部分，当你从另一台计算机克隆一个仓库时，复制的就是这个目录 10。它包含了项目的所有历史记录、所有提交快照以及其他版本控制所需的一切信息 11。

2.2. 可视化流程：文件如何在各区域间流转

基本的 Git 工作流程可以概括如下[10]：

1. 你在你的工作区中修改文件。
2. 你选择性地将那些你希望成为下一次提交一部分的变更进行“暂存”，这会将这些变更的快照添加到暂存区。
3. 你执行一次“提交”，它会获取暂存区中的文件快照，并将该快照永久性地存储到你的 Git 目录中。

如果一个文件的特定版本存在于 Git 目录中，它就被认为是“已提交”的。如果它被修改过并且被添加到了暂存区，它就是“已暂存”的。如果它自从被检出后发生了变化，但尚未被暂存，那么它就是“已修改”的[10]。

这个流程揭示了暂存区的一个核心作用：它并非一个多余的步骤，而是一个强大的“策展空间”。它将你在编辑器中“保存文件”的行为与在版本控制历史中“记录一次变更”的行为解耦开来。在传统的版本控制系统中，一次提交通常会记录自上次提交以来所有被修改过的文件。而 Git 允许你通过暂存区精确地构建下一次提交的内容。

例如，一个开发者可能为了解决一个复杂问题，在他的工作区中修改了 10 个文件，这些修改逻辑上可以分为三个独立的任务：修复一个 bug、重构一个函数和更新文档。如果没有暂存区，他可能只能将这 10 个文件的所有变更混在一个庞大而混乱的提交中。然而，借助暂存区，他可以：

1. 首先，只将与 bug 修复相关的几个文件的变更添加到暂存区，然后进行一次提交，提交信息为“修复用户登录时的认证错误”。
2. 接着，将与函数重构相关的变更添加到暂存区，再进行一次提交，信息为“重构数据处理模块以提高性能”。
3. 最后，暂存并提交文档的更新。

通过这种方式，原本一次模糊的、包含多个目的的修改，被组织成了三个清晰、原子化且逻辑独立的提交。这极大地提高了项目历史的可读性和可维护性。Git 甚至提供了像 git add --patch 这样的命令，允许开发者暂存一个文件中的部分变更，而不是整个文件，从而实现更细粒度的控制[12]。因此，理解暂存区是理解

git add 和 git commit 命令背后设计哲学的关键，它将这些命令从简单的“保存”操作提升为一种“历史的精雕细琢”。

下方的图表演示了文件在三个区域之间的流转路径以及触发这些流转的核心命令。

图表 **1**：**Git** 的三区域工作流模型

起始区域	目标区域	触发命令	命令作用
工作区	暂存区	git add <file>	将工作区中指定文件的当前内容快照添加到暂存区，准备

			将其纳入下一次提交。
暂存区	Git 仓库	git commit	将暂存区中的所有内容创建一个新的永久性快照（提交），并将其保存在 Git 仓库中。
Git 仓库	工作区	git switch <branch>	从 Git 仓库中检出指定分支的最新快照，用其内容覆盖工作区的文件。

这个模型是后续所有操作的基础。每当你对文件进行修改后，都需要通过 git add 命令将其“登记”到暂存区，然后再通过 git commit 命令将其“存档”到仓库历史中。

Chapter 3: 初始配置：建立你的身份标识

在你开始使用 Git 进行任何实质性工作之前，需要在你的计算机上进行一次性的初始配置。这个步骤至关重要，因为它确立了你在协同开发环境中的身份标识。在 Git 的世界里，每一次贡献都是有署名的，而这些署名信息就来源于此处的配置 [14]。

3.1. git config 命令：作用域概览

Git 提供了一个名为 git config 的工具，它允许你获取和设置控制 Git 外观及行为的配置变量 [14]。这些变量可以存储在三个不同的位置，每个位置对应一个不同的作用域，并且它们之间存在一个明确的优先级覆盖关系：本地配置会覆盖全局配置，而全局配置会覆盖系统配置 [14]。

- 系统级 **(System)**：
  - 作用域：对系统上的所有用户和他们的所有仓库生效。
  - 配置文件：[path]/etc/gitconfig。
  - 命令选项：--system。
  - 使用场景：通常由系统管理员设置，用于统一团队或服务器范围内的默认行为。修改此文件通常需要管理员权限 [14]。

- 全局级 **(Global)**：
  - 作用域：对当前用户的所有仓库生效。
  - 配置文件：~/.gitconfig 或 ~/.config/git/config。
  - 命令选项：--global。
  - 使用场景：这是最常用的配置级别，用于设置用户个人的偏好，例如你的姓名和电子邮件地址。这些设置将在你的这台计算机上的所有项目中被默认使用 [14]。
- 本地级 **(Local)**：
  - 作用域：仅对当前所在的仓库生效。
  - 配置文件：当前仓库的 .git/config 文件。
  - 命令选项：--local（这也是默认行为，通常可以省略）。
  - 使用场景：用于设置特定项目的配置，例如，当你在一个工作项目和一个个人开源项目中使用不同的电子邮件地址时，就可以在各自的仓库中设置本地配置来覆盖全局设置 [14]。

3.2. 实践步骤：设置你的 user.name 和 user.email

安装 Git 后，你首先应该做的事情就是设置你的用户名和电子邮件地址 [14]。这一点至关重要，因为你所创建的每一次 Git 提交都会使用这些信息，并且这些信息会

被永久地、不可更改地嵌入到提交历史中 [14]。

对于初次使用的大学生来说，最直接有效的方式是进行全局配置。这样，你只需在你的计算机上设置一次，之后你在该计算机上进行的所有项目开发都会自动使用这些信息。

1. 设置全局用户名和邮箱

打开你的命令行终端（在 Windows 上是 Git Bash，在 macOS 或 Linux 上是 Terminal），然后输入以下命令，将引号内的内容替换为你自己的信息：

Bash

```
$ git config --global user.name "Zhang San"
$ git config --global user.email "zhangsan@example.com"
```

- --global 标志告诉 Git 将此配置写入全局配置文件，使其对你在这台机器上的所有项目生效 [14]。

- user.name 应该是你的真实姓名或一个易于识别的昵称。
- user.email 应该是一个有效的电子邮件地址，通常建议使用与你的 GitHub 账户关联的邮箱。

2. 检查你的配置

设置完成后，你可以通过以下命令来验证配置是否成功。

要查看所有 Git 能找到的配置项及其来源，可以使用：

Bash

```
$ git config --list --show-origin
```

如果你只想检查某个特定的配置项，例如用户名，可以这样做：

Bash

```
$ git config user.name
```

终端应该会返回你刚刚设置的名字。

下面是一个在命令行中设置并检查配置的示例截图：

Plaintext

```
# 设置全局用户名
$ git config --global user.name "Ada Lovelace"

# 设置全局邮箱
$ git config --global user.email "ada.lovelace@example.com"

# 检查用户名配置
$ git config user.name
```

```
Ada Lovelace

# 检查邮箱配置
$ git config user.email
ada.lovelace@example.com

# 查看所有配置项列表
$ git config --list
user.name=Ada Lovelace
user.email=ada.lovelace@example.com
... (其他配置项)...
```

图注：在命令行中使用 *git config --global* 设置用户信息，并使用 *git config <key>* 进行验证。

完成了这个简单的配置步骤，你的 Git 环境就已经为记录你的贡献做好了准备。现在，你可以开始你的第一个项目之旅了。

Chapter 4: 贡献者之旅的起点：派生与克隆

在开源世界或许多团队协作项目中，贡献代码的第一步通常不是直接在原始项目上工作，而是创建一个属于你自己的副本。这个过程被称为“派生”（Forking），它是 GitHub 协作模式的核心机制之一。本章将指导你完成从派生一个项目到在本地计算机上创建其工作副本的完整流程。

4.1. 理解派生工作流

当你希望为一个你没有直接写入权限的项目做出贡献时，派生工作流是标准的操作方式 [17]。

派生 **(Fork)** 是一个在你的 GitHub 账户下创建目标仓库完整副本的操作 [18]。这个副本（我们称之为“派生仓库”或“fork”）与原始项目（我们称之为“上游仓库”或“upstream”）完全独立，你对自己的派生仓库拥有完全的读写权限，可以自由地进行实验和修改，而不会影响到原始项目 [18]。

完成修改后，你可以通过创建一个“拉取请求”（Pull Request）来提议将你的变更合并回上游仓库 [17]。整个工作流程通常遵循以下模式：派生 -> 克隆 -> 修改 -> 拉取请求 [17]。

4.2. 步骤详解：在 GitHub 上创建一个派生仓库

这个过程完全在 GitHub 的网页界面上完成。我们将以 GitHub 官方的测试仓库 octocat/Spoon-Knife 为例进行演示 [19]。

1. 导航至上游仓库：
   在浏览器中打开你想要贡献的项目的 GitHub 页面。例如：
   https://github.com/octocat/Spoon-Knife。
2. 点击“Fork”按钮：
   在页面的右上角，你会看到一个标有“Fork”的按钮。点击它 19。

   图注：GitHub 仓库页面右上角的 "Fork" 按钮。
3. 选择所有者并创建派生：
   GitHub 会提示你为这个新的派生仓库选择一个所有者（通常就是你自己的账户）19。你可以选择性地修改仓库名称和描述，然后点击“Create fork”按钮 19。

   图注：在创建 Fork 页面，选择所有者并确认创建。

片刻之后，GitHub 就会在你的账户下创建一个 Spoon-Knife 仓库的完整副本。你的浏览器会自动跳转到这个新仓库的页面，其 URL 会是 https://github.com/YOUR-USERNAME/Spoon-Knife。

4.3. 将项目带回家：git clone 命令

现在你有了一个云端的项目副本，下一步是将其下载到你的本地计算机上，以便进行实际的编码工作。这个过程被称为“克隆”（Cloning）。

git clone 命令用于创建一个远程仓库的本地副本 [22]。至关重要的是，它不仅仅是下载文件，而是复制整个仓库，包括所有的文件、所有的分支以及自项目诞生以来的全部提交历史 [24]。

1. 获取仓库 URL：
   在你自己的派生仓库（而不是上游仓库）的 GitHub 页面上，点击绿色的“<> Code”按钮。一个下拉菜单会出现，其中包含了克隆仓库所需的 URL。对于初学者，使用 HTTPS 链接是最简单的方式。点击旁边的复制按钮 19。

   !(https://user-images.githubusercontent.com/12345/67890125-example-clone-url.png)
   图注：点击 "Code" 按钮，复制 HTTPS 格式的 URL。
2. 在终端中执行克隆：

打开你的命令行终端，使用 cd 命令导航到你希望存放项目的本地目录（例如 ~/projects）。然后，输入 git clone 命令，并在其后粘贴你刚刚复制的 URL 19。

Bash

```
$ git clone https://github.com/YOUR-USERNAME/Spoon-Knife.git
```

请务必将 *YOUR-USERNAME* 替换为你自己的 *GitHub* 用户名。

3. 观察输出：
   按下回车后，Git 会开始下载仓库。你会在终端中看到类似下面的输出，显示下载进度 26。

Plaintext

```
$ git clone https://github.com/Ada-Lovelace/Spoon-Knife.git
Cloning into 'Spoon-Knife'...
remote: Enumerating objects: 32, done.
remote: Total 32 (delta 0), reused 0 (delta 0), pack-reused 32
Unpacking objects: 100% (32/32), 3. Spoon-Knife.19 KiB | 1.06 MiB/s, done.
```

图注：*git clone* 命令的典型终端输出。
当命令执行完毕后，你的本地文件系统中就会出现一个名为 Spoon-Knife 的新文件夹。这个文件夹就是一个功能齐全的本地 Git 仓库。

4.4. 验证你的设置：**git remote -v** 命令

git clone 命令非常智能，它在创建本地仓库的同时，还会自动设置一个指向你克隆来源的远程连接。这个远程连接的默认名称是 origin [27]。

为了验证这一点，并查看所有已配置的远程连接，我们可以使用 git remote -v 命令。-v 标志（verbose 的缩写）会显示每个远程连接的名称及其对应的 URL [28]。

首先，进入刚刚克隆的仓库目录：

Bash

```
$ cd Spoon-Knife
```

然后，运行 git remote -v：

Bash

```
$ git remote -v
```

终端的输出应该如下所示：

Plaintext

```
$ git remote -v
origin  https://github.com/Ada-Lovelace/Spoon-Knife.git (fetch)
origin  https://github.com/Ada-Lovelace/Spoon-Knife.git (push)
```

图注：*git remote -v* 命令的输出，显示了名为 *origin* 的远程连接，它指向用户的派生仓库。

这个输出确认了：

- 你有一个名为 origin 的远程连接。
- 这个 origin 指向的是你自己的派生仓库（Ada-Lovelace/Spoon-Knife.git）。
- Git 知道可以从这个 URL fetch（拉取）更新，也可以向这个 URL push（推送）你的变更。

此时，你的本地开发环境已经与你的 GitHub 派生仓库建立了连接。为了更好地理解整个协作流程中的各个实体及其关系，下面的图表演示了“上游”、“源”和“本地”三者之间的关系。

图表 **2**：派生工作流中的仓库关系模型

实体	位置	所有者	角色与关系
上游仓库 **(Upstream)**	GitHub.com	原始项目所有者	你希望贡献代码的中心仓库。你通常只有读取权限。
源仓库 **(Origin)**	GitHub.com	你自己	上游仓库在你账户下的一个完整副本。你拥有完全的读写权限。这是你推送本地变更的中心。

本地仓库 **(Local)**	你的计算机	你自己	源仓库（你的派生）在本地的克隆。这是你进行所有实际编码工作的地方。

关系路径：*Upstream --(Fork)--> Origin --(git clone)--> Local*

这个模型是理解整个贡献流程的心理地图。它清晰地界定了三个关键的仓库位置，并解释了为什么后续的命令（如 git push origin 和 git fetch upstream）会针对不同的目标。

Chapter 5: 保持同步：使你的派生仓库与上游项目保持最新

派生一个仓库，本质上是在某个时间点创建了原始项目的一个快照。然而，原始项目（即上游仓库）是持续演进的，其他贡献者会不断地合并新的变更。你的派生仓库不会自动接收这些更新。因此，在你开始进行任何新的开发工作之前，一个至关重要的步骤是：将你的派生仓库与上游仓库进行同步。

5.1. “上游”仓库以及同步的必要性

未能保持派生仓库的同步是导致后续在提交拉取请求时出现复杂合并冲突（Merge Conflicts）的主要原因之一。想象一下这样的场景：你基于一周前的代码版本开发了一个新功能。在这一周里，上游仓库的核心部分可能已经被重构了。当你提交拉取请求时，你的代码很可能与最新的代码库不兼容，从而产生大量的冲突，这会给项目维护者带来额外的审查负担，也增加了你修复问题的难度。

因此，同步并非一个可选的“良好实践”，而是派生工作流中一个强制性的、必不可少的前置步骤。它确保你的贡献是建立在项目最新版本的基础之上，从而最大程度地减少集成时的摩擦。这个“派生-同步-拉取请求”的因果链条是保证贡献质量和效率的关键。

同步的过程分为两步：首先，配置一个指向原始上游仓库的远程连接；然后，拉取上游的变更并将其合并到你的本地仓库中。

5.2. 配置上游远程连接：**git remote add**

你的本地仓库目前只知道一个远程连接，即 origin，它指向你在 GitHub 上的派生仓库。为了能够从原始项目中获取更新，你需要添加第二个远程连接，指向这个原始项目。按照惯例，这个远程连接被命名为 upstream [21]。

1. 获取上游仓库的 URL：
   在浏览器中导航到原始仓库的页面（例如 https://github.com/octocat/Spoon-Knife），像之前克隆时一样，点击“<> Code”按钮并复制 HTTPS URL。
2. 添加上游远程：
   在你的本地仓库目录中，执行 git remote add 命令。这个命令的语法是 git remote add <name> <url> 28。
   Bash
   ```
   $ git remote add upstream https://github.com/octocat/Spoon-Knife.git
   ```

3. 验证配置：
   再次运行 git remote -v 来检查你的远程连接配置。现在，你应该能看到 origin 和 upstream 两个远程连接 29。
   Plaintext
   ```
   $ git remote -v
   origin    https://github.com/Ada-Lovelace/Spoon-Knife.git (fetch)
   origin    https://github.com/Ada-Lovelace/Spoon-Knife.git (push)
   upstream  https://github.com/octocat/Spoon-Knife.git (fetch)
   upstream  https://github.com/octocat/Spoon-Knife.git (push)
   ```

   图注：*git remote -v* 的输出，显示了同时指向派生仓库的 *origin* 和指向原始仓库的 *upstream*。
   现在，你的本地仓库已经知道了两个远程端点：一个是你自己的、可读写的 origin，另一个是只读的原始 upstream。

5.3. 同步流程：fetch 与 merge

配置好上游远程之后，同步你的本地 main 分支（或其他主分支）就变得非常简单，主要包含两个命令 [30]。

第一步：从上游仓库拉取变更 **(fetch)**

git fetch 命令会从指定的远程仓库下载你本地尚未拥有的所有数据，包括所有分支的最新提交 [31]。这个命令非常安全，因为它

只会下载数据到你的本地 **Git** 数据库，并更新你的远程跟踪分支（例如 **upstream/main**），但它

不会修改你自己的工作区文件或你自己的本地分支（例如 **main**）[31]。

Bash

```
$ git fetch upstream
```

执行后，你可能会看到类似下面的输出：

Plaintext

```
$ git fetch upstream
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/octocat/Spoon-Knife
 * [new branch]      main       -> upstream/main
 * [new branch]      test-branch -> upstream/test-branch
```

图注：*git fetch upstream* 的输出，表明已从上游仓库获取了新的数据，并更新了本地的 *upstream/main* 等远程跟踪分支。

现在，你本地仓库的 upstream/main 分支指针已经指向了上游仓库 main 分支的最新提交。

第二步：将上游变更合并到本地分支 **(merge)**

获取了上游的最新变更后，你需要将这些变更应用到你自己的本地主分支上。

1. 切换到你的本地主分支：
   确保你当前位于你想要更新的分支上。通常是 main 分支。
   Bash
   ```
   $ git switch main
   ```

2. 合并上游分支：
   使用 git merge 命令，将你刚刚更新的 upstream/main 分支合并到你当前的 main 分支中 30。
   Bash

```
$ git merge upstream/main
```

如果你的本地 main 分支自派生以来没有任何新的提交，Git 将会执行一次“快进式合并”（Fast-forward merge），这仅仅是将你的 main 分支指针向前移动到与 upstream/main 相同的位置 [30]。

```
Plaintext
$ git merge upstream/main
Updating 3497943..0c49c59
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

图注：一次成功的快进式合并的输出。

现在，你的本地 main 分支已经与上游仓库的 main 分支完全同步。最后，为了让你在 GitHub 上的派生仓库也保持同步，你需要将本地的更新推送上去：

```
Bash
$ git push origin main
```

通过定期执行这个“拉取上游，合并本地，推送源”的同步流程，你可以确保自己的工作始终基于项目的最新状态，为顺利的贡献铺平道路。

Chapter 6: 隔离的力量：掌握功能分支

在现代 Git 工作流中，一个最核心、最普遍遵循的原则是：永远不要直接在主分支（如 **main** 或 **master**）上进行开发。所有的开发工作，无论是添加一个新功能、修复一个 bug，还是进行一次实验，都应该在一个专门为此创建的、隔离的“功能分支”（Feature Branch）上进行。本章将深入探讨为何要使用分支，以及如何通过命令行有效地管理它们。

6.1. 为何要使用分支？隔离开发的重要性

在 Git 中，一个分支本质上只是一个轻量级的、可移动的、指向某次提交的指针 [33]。当你创建一个新分支时，你实际上是在说："我想从当前这个时间点开始，开辟一条新的开发路径，而不会影响到我当前所在的这条主路径。" [33]。

这种隔离开发的方式带来了巨大的好处：

- 保护主分支的稳定性：主分支（main）通常代表了项目的稳定、可部署的版本。将不稳定的、正在开发中的代码隔离在功能分支上，可以确保 main 分支随时都是可靠的 [20]。
- 并行开发：团队成员可以同时在各自的功能分支上开发不同的功能，而不会相互干扰。
- 清晰的代码审查：当一个功能开发完成后，可以通过拉取请求（Pull Request）的形式，将整个功能分支作为一个独立的、完整的单元进行审查和讨论。
- 方便的实验：如果你有一个不确定的想法，可以随时创建一个实验性分支去尝试。如果最终证明这个想法不可行，你只需简单地删除这个分支，主分支的历史将完全不受影响。

6.2. 创建、列出和删除分支：git branch 命令

git branch 命令是 Git 中用于管理分支的瑞士军刀 [35]。

- 列出分支：
  不带任何参数运行 git branch，会列出你本地仓库中所有的分支。当前所在的分支会以一个星号 * 作为前缀 36。
  Plaintext
  ```
  $ git branch
    iss53
  * main
    testing
  ```

  图注：*git branch* 命令的输出，显示 *main* 是当前分支。
- 创建新分支：
  使用 git branch <branch-name> 可以创建一个新的分支。这个新分支会指向你当前所在的提交 34。需要注意的是，这个命令
  只创建分支，并不会自动切换到新分支上。
  Bash
  ```
  $ git branch new-feature
  ```

- 删除分支：
  当你完成了一个功能分支的工作并已将其合并到主分支后，通常就可以删除它了。使用 git branch -d <branch-name>（-d 是 --delete 的缩写）可以安全地删除一个分支。说它"安全"，是因为如果该分支包含尚未被合并到当前分支的提交，Git 会阻止删除并给出提示 34。
  Bash
  ```
  $ git branch -d old-feature
  ```

如果你确定要丢弃某个分支上的所有工作，即使它们尚未合并，也可以使用 git branch -D <branch-name>（大写的 -D）来强制删除 [34]。

6.3. 导航你的工作空间：git switch 与 git checkout

创建了分支之后，你需要在它们之间进行切换，以在不同的开发任务上工作。Git 提供了两个命令来实现这一功能：git checkout 和 git switch。

git checkout 是一个历史悠久且功能强大的命令，但它的功能被过度“重载”了——它既可以用于切换分支，也可以用于恢复文件，这有时会让初学者感到困惑 [38]。为了解决这个问题，从 Git 2.23 版本开始，引入了两个新的、职责更明确的命令：

git switch 用于切换分支，git restore 用于恢复文件 [39]。

对于初学者而言，优先学习和使用 git switch 是更清晰、更安全的选择，因为它只专注于切换分支这一项任务。

- 切换到现有分支：
  使用 git switch <branch-name> 可以将你的工作区切换到指定的分支。Git 会更新你工作区的文件，使其与该分支最新提交的状态相匹配，同时，一个名为 HEAD 的特殊指针会移动到这个分支上，表示你现在正在此分支上工作 33。
  Bash
  ```
  $ git switch new-feature
  ```

- 创建并切换到新分支：
  这是一个非常常见的操作：创建一个新分支，然后立刻切换过去。git switch 提供了一个方便的快捷方式 -c（--create 的缩写）来实现这一点 33。
  Bash
  ```
  $ git switch -c another-new-feature
  ```

  这条命令等同于以下两条命令的组合：
  Bash
  ```
  $ git branch another-new-feature
  $ git switch another-new-feature
  ```

  你可能也会在一些旧的教程或文档中看到使用 git checkout -b <new-branch-name> 来实现同样的功能，这是 git checkout 命令的传统用法 [33]。了解它有助于你阅读他人的代码或文档，但在你自己的工作中，推荐使用更现代、更清晰的
  git switch -c。

下面是一个创建并切换分支的完整示例：

Plaintext

```
# 查看当前分支
$ git branch
* main

# 创建一个名为 "update-readme" 的新分支
$ git branch update-readme

# 再次查看分支列表，新分支已创建，但当前仍在 main
$ git branch
* main
  update-readme

# 切换到新分支
$ git switch update-readme
Switched to branch 'update-readme'

# 查看分支列表，确认已切换
$ git branch
  main
* update-readme

# 或者，使用一条命令创建并切换
$ git switch -c add-contribution-guide
Switched to a new branch 'add-contribution-guide'
```

图注：使用 *git branch* 和 *git switch* 命令管理和切换分支的流程。

掌握了分支的使用，你就掌握了 Git 工作流的精髓。现在，你可以在一个安全隔离的环境中，开始进行你的代码贡献了。

Chapter 7: 打造你的贡献：编辑-暂存-提交的循环

在功能分支上进行开发的核心，是一个不断重复的循环：修改文件、将变更暂存、然后将暂存的变更提交为历史记录中的一个快照。本章将详细拆解这个循环的每一个环节，并提供丰富的命令行输出截图，帮助你精确理解每个命令的作用和效果。

7.1. 明确你的方位：详解 git status

在你进行任何操作之前或之后，养成运行 git status 的习惯是至关重要的。这个命令是你了解仓库当前状态的“仪表盘”，它能告诉你工作区和暂存区发生了什么变化 [42]。它是一个完全安全的信息查询命令，不会对你的文件或历史记录做任何修改 [44]。

git status 的输出信息不仅是描述性的，更是指导性的。它会清晰地告诉你文件处于哪个阶段，并常常会提示你下一步应该使用哪个命令来推进工作流 [45]。

- 状态一：工作区干净
  当你刚刚完成一次提交，或者克隆了一个仓库而未做任何修改时，git status 会告诉你一切就绪。
  Plaintext
  ```
  $ git status
  On branch main
  Your branch is up to date with 'origin/main'.

  nothing to commit, working tree clean
  ```
  图注：一个干净的工作区，没有任何待处理的变更。
- 状态二：存在未暂存的变更 (Changes not staged for commit)
  如果你修改了一个已经被 Git 跟踪的文件，git status 会检测到这个变化，并将其列在“Changes not staged for commit”部分，通常以红色文本显示 47。
  Plaintext
  ```
  $ git status
  On branch main
  Your branch is up to date with 'origin/main'.

  Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
          modified:   README.md

  no changes added to commit (use "git add" and/or "git commit -a")
  ```
  图注：*README.md* 文件已被修改，但变更尚未被暂存。

- 状态三：存在未跟踪的文件 (Untracked files)
  如果你在项目中创建了一个新文件，Git 会注意到它的存在，但由于这个文件从未被提交过，Git 会将其归类为“Untracked files”，同样以红色显示 47。
  Plaintext
  ```
  $ git status
  On branch main
  Your branch is up to date with 'origin/main'.

  Untracked files:
    (use "git add <file>..." to include in what will be committed)
          CONTRIBUTING.md

  nothing added to commit but untracked files present (use "git add" to track)
  ```
  图注：*CONTRIBUTING.md* 是一个新文件，*Git* 尚未开始跟踪它。
- 状态四：存在待提交的变更 (Changes to be committed)
  当你使用 git add 命令将变更从工作区移入暂存区后，git status 会将这些文件列在“Changes to be committed”部分，通常以绿色文本显示 47。
  Plaintext
  ```
  $ git status
  On branch main
  Your branch is up to date with 'origin/main'.

  Changes to be committed:
    (use "git restore --staged <file>..." to unstage)
          new file:   CONTRIBUTING.md
          modified:   README.md
  ```
  图注：*CONTRIBUTING.md* 和 *README.md* 的变更都已暂存，准备好被提交。

7.2. 暂存你的变更：**git add** 的应用

git add 命令的作用是将工作区的变更（无论是修改过的文件还是新文件）的当前快照添加到暂存区，为下一次提交做准备 [12]。

- 暂存单个文件：
  这是最精确、最推荐的方式，因为它强迫你思考每次提交应该包含哪些具体的文件。
  Bash
  ```
  $ git add README.md
  ```

- 暂存所有变更：
  git add. 是一个常用的快捷方式，它会暂存当前目录及其子目录下所有的变更（包括新文件、修改的文件和被删除的文件）12。
  Bash
  ```
  $ git add.
  ```

  让我们通过一个完整的流程来观察 git status 在 git add 前后的变化：
  Plaintext
  ```
  # 初始状态，一个文件被修改，一个文件是新的
  $ git status
  Changes not staged for commit:
        modified:   README.md
  Untracked files:
        CONTRIBUTING.md

  # 使用 `git add.` 暂存所有变更
  $ git add.

  # 再次检查状态，所有变更都已移至暂存区
  $ git status
  Changes to be committed:
        new file:   CONTRIBUTING.md
        modified:   README.md
  ```

  图注：*git status* 的输出清晰地反映了 *git add* 命令的效果。

7.3. 创建快照：**git commit** 的艺术

git commit 命令会将暂存区中的所有内容记录为仓库历史中的一个新快照 [48]。

- 通过编辑器提交：
  直接运行 git commit 会打开你在 Git 配置中设置的默认文本编辑器（如 Vim 或 Nano）15。编辑器中会提示你输入提交信息。文件的第一行被视为主旨（subject），空一行后可以输入更详细的描述。
  Bash
  ```
  $ git commit
  ```

图注：git commit 命令后，在 Nano 编辑器中撰写提交信息。

- 通过命令行直接提交：
  对于简单的提交，使用 -m（--message 的缩写）标志可以直接在命令行中提供提交信息，从而跳过打开编辑器的步骤 48。
  Bash
  ```
  $ git commit -m "Add contribution guidelines and update README"
  ```
  执行后，终端会显示提交的摘要信息：
  Plaintext
  ```
  [main 24fbe3c] Add contribution guidelines and update README
   2 files changed, 10 insertions(+), 1 deletion(-)
   create mode 100644 CONTRIBUTING.md
  ```
  图注：使用 *-m* 标志进行提交后的确认信息。

7.4. 清晰历史的重要性：撰写有效的提交信息

提交信息是项目历史的叙述。一条好的提交信息能够让其他协作者（以及未来的你）快速理解某次变更的目的和背景。一个被广泛接受的规范是 [48]：

1. 用一行不超过 50 个字符的文字简要概括变更内容（主旨）。
2. 主旨行后留一个空行。
3. 之后是更详细的解释性文字，可以分段。解释“为什么”做出这个变更，而不仅仅是“做了什么”变更。

7.5. 回顾你的工作：**git log** 命令简介

git log 命令用于查看仓库的提交历史 [52]。它有许多强大的选项来格式化输出，帮助你以不同的方式审视历史。

- 默认输出：
  git log 的默认输出比较详细，包含了每次提交的 SHA 哈希、作者、日期和完整的提交信息。
- 常用格式化选项：
  - --oneline：将每次提交压缩到一行，只显示简短的 SHA 哈希和提交主旨，非常适合快速浏览 [52]。
  - --graph：在输出的左侧用 ASCII 字符绘制一个图形，清晰地展示分支和合并的历史脉络 [52]。
  - --decorate：显示指向各个提交的分支和标签名称（例如 HEAD -> main, origin/main）[52]

    - 。
    - --all：显示所有分支的历史，而不仅仅是当前分支。

将这些选项组合起来，可以得到一个非常强大且直观的历史视图命令：

Bash

```
$ git log --graph --oneline --decorate --all
```

这个命令的输出效果如下：

Plaintext

```
*   0e25143 (HEAD -> feature-branch) Add new feature implementation

| * c304897 (origin/main, origin/HEAD, main) Merge pull request #12 from user/fix-bug
|/|
| * 8a49b42 (origin/fix-bug, fix-bug) Fix the authentication bug
* | 24fbe3c Add contribution guidelines and update README

|/
* 3497943 (upstream/main) Initial commit
```

图注：*git log* 结合了 *--graph, --oneline, --decorate, --all* 选项的强大输出，直观地展示了分支、合并和提交历史。

熟练掌握“编辑-暂存-提交-查看日志”这个循环，是进行所有 Git 操作的基础。

Chapter 8: 分享你的工作：推送到你的远程派生仓库

在你本地的功能分支上完成了一系列提交之后，这些变更目前只存在于你的个人计算机上。为了让其他人（包括项目维护者）能够看到你的工作并进行代码审查，你需要将这些提交上传到你在

GitHub 上的派生仓库。这个过程被称为“推送”（Pushing）。

8.1. git push 命令详解

git push 命令用于将本地仓库的内容上传到一个远程仓库 [57]。它是

git fetch 和 git pull 的对应操作：fetch/pull 是从远程下载变更，而 push 则是向远程上传变更 [57]。

该命令的基本语法是：

Bash

```
git push <remote-name> <branch-name>
```

- <remote-name>：指定你要推向的远程仓库的名称。根据我们之前的设置，这通常是 origin（指向你自己的派生仓库）或 upstream（指向原始项目仓库）。在贡献工作流中，你总是推送到 **origin**，因为你对 upstream 没有写入权限。
- <branch-name>：指定你想要推送的本地分支的名称，例如 update-readme。

8.2. 首次推送新分支：-u 或 --set-upstream 标志

当你第一次推送一个在本地创建的新分支时，远程仓库（origin）并不知道这个分支的存在。此外，你的本地分支也不知道它应该与远程仓库的哪个分支相关联。这时，你需要使用 -u 或 --set-upstream 标志来建立一个“上游跟踪关系” [58]。

这个标志告诉 Git 两件事：

1. 将指定的本地分支推送到 origin 远程仓库，并在 origin 上创建一个同名分支。
2. 将你的本地分支与新创建的远程分支“链接”起来。

建立这种跟踪关系的好处是，在此之后，当你位于这个分支上时，你只需简单地运行 git push 或 git pull，Git 就会自动知道应该与 origin 上的同名分支进行同步，而无需再指定远程和分支名称 [59]。

假设你正在 update-readme 分支上工作，并且想要首次推送它，你应该运行以下命令：

Bash

```
$ git push --set-upstream origin update-readme
```

或者使用更简洁的 -u 标志：

Bash

```
$ git push -u origin update-readme
```

执行命令后，终端会显示推送的详细过程，并确认上游跟踪关系已经建立[60]：

Plaintext

```
$ git push -u origin update-readme
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 304 bytes | 304.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/Ada-Lovelace/Spoon-Knife.git
 * [new branch]      update-readme -> update-readme
Branch 'update-readme' set up to track remote branch 'update-readme' from 'origin'.
```

图注：首次推送新分支并设置上游跟踪的命令及其输出。最后一行确认了跟踪关系的建立。

对于后续的推送，只要你还在 update-readme 分支上，你只需运行：

Bash

```
$ git push
```

Git 就会自动将你的新提交推送到 origin/update-readme。

8.3. 在 GitHub 上验证推送

推送完成后，你可以回到浏览器，刷新你在 GitHub 上的派生仓库页面。你应该会看到几个变化：

1. 在分支选择下拉菜单中，可以看到你新推送的 update-readme 分支。
2. GitHub 通常会检测到新推送的分支，并在仓库主页上方显示一个黄色的提示横幅，建议你基于这个新分支创建一个拉取请求（Pull Request）。

!(https://user-images.githubusercontent.com/12345/67890127-example-pr-banner.png)
图注：成功推送到派生仓库后，GitHub 页面上会出现一个显眼的横幅，引导你发起一个拉取请求。
这个提示横幅是你工作流程的下一个入口点。你的代码现在已经安全地备份在云端，并准备好被提议合并到原始项目中了。

Chapter 9: 提议整合：在 GitHub 上创建拉取请求

将你的本地变更推送到派生仓库后，你的工作已经完成了一半。现在，你需要正式地向上游项目的维护者提议，请求他们将你的变更整合到主代码库中。这个正式的提议过程，在 GitHub 上被称为“创建拉取请求”（Creating a Pull Request，简称 PR）。一个 PR 不仅仅是一个代码合并请求，它更是一个协作和讨论的起点 [8]。

9.1. 导航至 GitHub 界面

在你将新分支推送到你的派生仓库后，GitHub 会让创建 PR 的过程变得非常简单。

通常，当你访问你的派生仓库主页时，会看到一个显眼的黄色横幅，其中包含一个绿色的

“Compare & pull request”按钮 [61]。这个横幅是 GitHub 智能检测到你的分支领先于上游仓库而提供的便捷入口。

图注：新分支推送后，GitHub 会自动显示创建拉取请求的提示。
点击这个按钮，你将被直接带到“Open a pull request”页面。

9.2. “Open a pull request” 页面详解

这个页面是创建 PR 的核心界面，你需要在这里明确指定你的变更将如何被合并。你需要确认四个关键信息，这些信息通常由 GitHub 预先填充好，但检查它们至关重要 [63]：

1. 基础仓库 **(base repository)**：这是你希望将变更合并进去的目标仓库，即上游仓库（例如 octocat/Spoon-Knife）。
2. 基础分支 **(base branch)**：这是上游仓库中你希望合并变更的目标分支，通常是 main 或 develop。
3. 头仓库 **(head repository)**：这是包含你的变更的源仓库，即你自己的派生仓库（例如 YOUR-USERNAME/Spoon-Knife）。
4. 比较分支 **(compare branch)**：这是你派生仓库中包含你所做变更的功能分支（例如 update-readme）。

由于你是从一个派生仓库向上游仓库发起 PR，你需要确保点击了“compare across forks”链接，以显示选择不同仓库的下拉菜单 [62]。

图注：一个经过标注的“Open a pull request”页面截图，清晰地指出了基础仓库、基础分支、头仓库和比较分支四个关键的下拉选择框。
在这个页面下方，GitHub 会显示一个差异（diff）视图，展示了你的分支与目标分支之间的所有代码变更，供你在提交前做最后一次检查。

9.3. 撰写一个有说服力的 PR 标题和描述

标题和描述是你的 PR 的“门面”，它们是项目维护者首先看到的内容。一个清晰、有说服力的 PR 描述能够极大地帮助维护者理解你的贡献，并加快审查过程。

- 标题 **(Title)**：应该是一个简明扼要的摘要，清晰地概括了这个 PR 的目的。例如，“更新 README 文件以添加贡献者链接”。
- 描述 **(Description)**：这里是你详细阐述的地方。你应该解释：

- 这个 **PR** 解决了什么问题？**(What)**：例如，“此变更向 README 文件添加了一个指向新的贡献指南的链接。”
- 为什么这个变更是必要的？**(Why)**：例如，“为了让新贡献者更容易找到如何参与项目的信息。”
- 你是如何解决的？**(How)**：如果适用，可以简要描述你的实现方法。

如果你的 PR 是为了解决一个已经存在的 Issue（问题），你可以在描述中使用特定的关键词（如 Fixes #123、Closes #123）来链接到那个 Issue。这样做的好处是，当你的 PR 被合并时，GitHub 会自动关闭那个关联的 Issue [64]。

9.4. 提交拉取请求

在填写完所有信息并检查了代码差异后，你就可以提交 PR 了。在页面底部，你会看到一个绿色的“Create pull request”按钮。

在点击之前，你可能会注意到一个选项：“Create draft pull request” [61]。

- **Create Pull Request**：创建一个准备好接受审查的正式 PR。它会通知项目维护者和代码所有者前来审查。
- **Create Draft Pull Request**：创建一个草稿状态的 PR。这表示你的工作仍在进行中，你可能只是想获得一些早期的反馈，或者运行一些自动化的 CI（持续集成）检查。草稿 PR 不能被合并，也不会自动请求审查，直到你手动将其标记为“Ready for review” [65]。

对于初学者，直接点击“Create pull request”即可。提交后，你和项目的协作者就会被带到一个新的页面，这个页面是这个 PR 的专属讨论和审查空间。你的贡献之旅，至此迈出了最关键的一步。

Chapter 10: 响应反馈：协同审查周期

提交拉取请求（PR）并不是贡献过程的终点，而是一个对话的开始。你的代码现在将接受项目维护者和其他协作者的审查。他们可能会提出问题、请求修改或给出改进建议。本章将指导你如何理解代码审查的反馈，并更新你的 PR 以回应这些反馈。

10.1. 在 GitHub 上理解代码审查反馈

在 PR 页面的“Files changed”标签页中，审查者可以对你的代码进行逐行评论 [66]。

- 行级评论 **(Line Comments)**：审查者可以在任何一行代码旁边留下评论，提出具体的问题或建议。
- 建议变更 **(Suggestions)**：审查者甚至可以提出具体的代码修改建议。这些建议会以特殊的格式显示，你只需点击一个按钮就可以将这些建议应用到你的代码中，并创建一个新的提交 [66]。
- 审查总结 **(Review Summary)**：在审查了全部或部分代码后，审查者会提交一个总结性的审查意见，其状态可以是以下三种之一 [67]：
  - **Comment**：留下一般性评论，不表示批准或反对。
  - **Approve**：表示同意这些变更，认为它们可以被合并。
  - **Request changes**：表示在合并之前，必须解决所提出的问题。

!(https://user-images.githubusercontent.com/12345/67890129-example-review-comment.png)
图注：一个 PR 的“Files changed”视图，展示了审查者留下的行级评论和一个可以直接应用的“建议变更”。

10.2. 更新拉取请求的简便流程

对于初学者来说，一个常见的困惑是：“我应该如何‘更新’我的 PR？” 答案比想象中要简单得多，因为它完美地利用了 Git 分支的工作机制。

一个拉取请求并非你一次性发送的静态代码包，而是一个动态的、指向你功能分支的实时指针。这意味着，任何被推送到该功能分支的新提交，都会自动地、实时地出现在这个拉取请求中 [65]。

这个设计原理极大地简化了更新流程。你不需要关闭旧的 PR 再开一个新的，也不需要学习任何复杂的“更新 PR”命令。你只需要在你本地的同一个功能分支上继续工作，然后像之前一样推送你的变更即可。

根据审查反馈更新 **PR** 的标准步骤如下：

1. 切换到本地功能分支：
   确保你的本地仓库位于你为这个 PR 创建的功能分支上。
   Bash
   ```
   $ git switch update-readme
   ```
2. 进行代码修改：
   根据审查者的反馈，在你的代码编辑器中进行必要的修改。例如，修正一个拼写错误，或者重构一个函数。
3. 暂存并提交变更：
   使用你已经熟悉的“编辑-暂存-提交”循环，将你的修改创建为一个或多个新的提交。撰写清晰的提交信息，说明你解决了哪些审查意见。

```
Bash
$ git add.
$ git commit -m "Fix typo in README as requested in review"
```

4. 推送到你的派生仓库：
   将包含修复的新提交推送到你的 origin 远程的同一个功能分支上。因为之前已经设置了上游跟踪关系，所以现在只需简单的 git push。
   ```
   Bash
   $ git push
   ```

自动更新的魔力

在你执行 git push 之后，回到 GitHub 上的 PR 页面并刷新。你会发现，你刚刚推送的新提交已经自动出现在了 PR 的提交列表和对话历史中。审查者会收到通知，可以看到你的更新，并对你的修改进行新一轮的审查。

!(https://user-images.githubusercontent.com/12345/67890130-example-pr-update.png)
图注：在本地推送了一个新的修复提交后，GitHub 的 PR 页面会自动显示这个新提交，并将其添加到变更历史中。
这个“本地修改 -> 提交 -> 推送 -> PR 自动更新”的循环会一直持续，直到你的 PR 获得批准并最终被合并。这个流畅的迭代过程是 GitHub 协同开发的核心所在。

Chapter 11: 当世界碰撞时：命令行解决合并冲突实战指南

在协同开发中，合并冲突（Merge Conflict）是不可避免的，但对于初学者来说，它往往是最令人畏惧的情景之一。当 Git 无法自动合并两个分支的变更时，冲突就会发生。本章旨在揭开合并冲突的神秘面纱，提供一个冷静、清晰的步骤化指南，帮助你在命令行中自信地解决它们。

11.1. 是什么导致了合并冲突？

合并冲突的根本原因在于，两个不同的分支对同一个文件的同一部分进行了不同的修改 [37]。例如：

- 你和另一位协作者在各自的分支上，修改了 styleguide.md 文件的同一行。
- 你在一个分支上修改了 README.md 文件，而另一位协作者在另一个分支上删除了这个文件。

在这些情况下，Git 无法替你做出决定——它不知道应该保留哪个版本的修改，或者是否应该删除文件。因此，它会暂停合并过程，将决策权交还给你，并要求你手动解决这些冲突 [37]。

11.2. 在终端中识别冲突

当你在执行 git merge 或 git pull 命令时遇到冲突，终端会明确地告诉你合并失败，并列出冲突的文件 [37]。

```
Plaintext
```

```
$ git merge feature-branch
Auto-merging styleguide.md
CONFLICT (content): Merge conflict in styleguide.md
Automatic merge failed; fix conflicts and then commit the result.
```

图注：一次失败的合并操作在终端中的输出。

此时，运行 git status 会提供更详细的信息。它会告诉你当前正处于一个合并冲突的状态，并列出所有“未合并的路径”（Unmerged paths）[37]。

```
Plaintext
```

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   styleguide.md

no changes added to commit (use "git add" and/or "git commit -a")
```

图注：在合并冲突状态下，*git status* 会清晰地列出存在冲突的文件。

11.3. 解读文件中的冲突标记

Git 会非常贴心地直接在你存在冲突的文件中插入一些特殊的标记，以精确地指出冲突发生的位置。当你用文本编辑器打开这个文件（例如 styleguide.md）时，你会看到类似下面的内容 [37]：

```
If you have questions, please
<<<<<<< HEAD
open an issue
=======
ask your question in IRC.
>>>>>>> feature-branch
```

这些标记的含义如下 [70]：

- <<<<<<< HEAD：这是冲突区域的开始。它下面的内容，直到 =======，是来自你当前所在分支（即 HEAD 指向的分支，这里是 main 分支）的修改。
- =======：这是一个分隔符，它将两个分支的冲突内容隔开。
- >>>>>>> feature-branch：这是冲突区域的结束。它上面的内容，从 ======= 开始，是来自你正在尝试合并进来的分支（这里是 feature-branch）的修改。

11.4. 解决冲突的工作流：编辑、暂存、提交

解决冲突的过程，本质上是一个手动的“微型”合并，它遵循我们已经熟悉的“编辑-暂存-提交”模式 [45]。

第一步：手动编辑文件
你的任务是打开这个包含冲突标记的文件，然后根据你的判断，决定最终的内容应该是什么。你可以选择保留其中一个分支的修改，或者将两者结合起来，或者完全写一个新的版本。
例如，在上面的例子中，你可能决定将两个建议都保留下来。你需要手动编辑文件，并删除所有的冲突标记 (<<<<<<<, =======, >>>>>>>)，最终文件内容如下：

```
If you have questions, please open an issue or ask your question in IRC.
```

第二步：暂存已解决的文件 (git add)
在你手动编辑并保存了文件之后，你需要明确地告诉 Git：“我已经解决了这个文件的冲突”。这个操作通过 git add 命令完成 70。将已解决的文件添加到暂存区，标志着该文件的冲突状态结束。

```
Bash
```

```
$ git add styleguide.md
```

第三步：完成合并提交 (git commit)
当所有冲突文件都通过 git add 标记为已解决后（你可以多次运行 git status 来确认），你就可以完成这次被暂停的合并了。只需运行 git commit 命令 70。

```
Bash
```

```
$ git commit
```

Git 会识别出你正在完成一次合并，并自动打开一个文本编辑器，里面已经为你准备好了一条默认的合并提交信息，例如 Merge branch 'feature-branch'。你通常可以直接保存并关闭这个文件，以完成提交。

提交完成后，一次新的“合并提交”就被创建了，它有两个父提交（分别指向 main 和 feature-branch 的末端），标志着两个分支的历史在此汇合。你的合并冲突也就被成功解决了。

如果你在解决冲突的过程中感到困惑，想要放弃这次合并，可以随时运行 git merge --abort，它会撤销这次合并，让你的仓库回到合并开始之前的状态 [72]。

Appendix A: 核心术语词汇表

本词汇表旨在为本指南中使用的核心 Git 和 GitHub 术语提供一个快速参考。

术语 (Term)	中文	定义 (Definition)
Branch	分支	仓库的一个并行版本。它包含在仓库内部，但不会影响主分支，允许你自由地工作而不会干扰“线上”版本 [8]。
Clone	克隆	创建一个远程仓库的本地副本。克隆操作会复制仓库的全部历史记录，包括所有文件、分支和提交 [24]。
Commit	提交	对一个或一组文件的单次更改的快照。每次提交都有一个唯一的 SHA 哈希值，并记录了作者、时间和提交信息 [8]。
Fetch	拉取	从一个远程仓库下载你本地尚未拥有的提交和文件，但它不会自动合并或修改你当前的工作。它只更新你的远程跟踪分支 [31]。
Fork	派生	在你的 GitHub 账户下创建一个其他用户仓库的个人副本。这允许你在不影响原始项目的情况下自由地进行修改 [17]。
HEAD	**HEAD**	一个特殊的指针，指向你当前所在的本地分支的最新提交。在“分离 HEAD”状态下，它直接指向一个提交而不是一个分支 [33]。
Index	索引	暂存区的技术名称。这是一个文件，存储了关于下一次提交将包含哪些内容的信息

		[10]。
Main / Master	主分支	Git 仓库中默认的开发分支。通常被视作项目的稳定、权威版本 [33]。
Merge	合并	将一个或多个分支的历史记录整合到当前分支中的操作。Git 会尝试自动合并变更，但如果存在冲突则需要手动解决 [31]。
Merge Conflict	合并冲突	当 Git 无法自动解决两个分支之间的差异时发生的情况。通常是因为两个分支修改了同一个文件的同一部分 [37]。
Origin	源	当你克隆一个仓库时，Git 为你克隆的那个远程 URL 创建的默认别名。它通常指向你自己的派生仓库 [27]。
Pull	拉取	git fetch 和 git merge 两个命令的组合。它从远程仓库获取更新，并立即尝试将其合并到你当前的本地分支中 [31]。
Pull Request (PR)	拉取请求	一个正式的提议，请求将一个分支（通常在你的派生仓库中）的变更合并到另一个仓库的主分支中。它是 GitHub 上代码审查和协作讨论的核心机制 [8]。
Push	推送	将你本地仓库中的提交上传到一个远程仓库，从而与他人分享你的变更 [57]。
Remote	远程	指向托管在别处的仓库版本的引用，例如在 GitHub 上的

		仓库。你可以通过 git remote 命令管理这些连接 [27]。
Repository	仓库	一个项目的集合，包含了所有的文件、目录以及每个文件的完整修订历史记录 [8]。
Staging Area	暂存区	一个介于工作区和仓库历史之间的中间区域。你使用 git add 命令将你希望包含在下一次提交中的变更放入暂存区 [10]。
Upstream	上游	一个通用的术语，指代你派生出的原始仓库。通常，你会配置一个名为 upstream 的远程，指向这个原始仓库，以便从中获取更新 [21]。
Working Directory	工作区	你在文件系统上看到的项目文件和目录。这是你进行实际编辑和修改的地方 [10]。

Appendix B: 常用命令参考速查表

本速查表汇总了完成一次标准 Pull Request 工作流所需的最核心的 Git 命令，旨在为你日常使用提供快速参考 [78]。

命令 (Command)	目的 (Purpose)	常见用法示例 (Example Usage)
git config --global	设置全局配置，如用户信息	git config --global user.name "Ada Lovelace"
git clone <url>	从远程 URL 克隆一个仓库到本地	git clone https://github.com/user/repo.git

git remote -v	列出所有远程连接及其 URL	git remote -v
git remote add <name> <url>	添加一个新的远程连接	git remote add upstream https://github.com/original/repo.git
git fetch <remote>	从远程仓库下载最新历史，但不合并	git fetch upstream
git merge <branch>	将指定分支的变更合并到当前分支	git merge upstream/main
git pull <remote> <branch>	拉取远程变更并立即尝试合并	git pull origin main
git branch	列出本地所有分支	git branch
git branch <name>	创建一个新分支	git branch new-feature
git switch <branch>	切换到指定分支	git switch new-feature
git switch -c <branch>	创建一个新分支并立即切换过去	git switch -c another-feature
git status	查看工作区和暂存区的状态	git status
git add <file>	将文件变更添加到暂存区	git add README.md
git add.	将当前目录所有变更添加到暂存区	git add.
git commit -m "message"	将暂存区的变更提交到本地仓库	git commit -m "Implement user authentication"
git log	查看提交历史	git log
git log --oneline --graph	以图形化单行格式查看提交历史	git log --oneline --graph --decorate --all

git push <remote> <branch>	将本地分支的提交推送到远程仓库	git push origin new-feature
git push -u <remote> <branch>	首次推送新分支并建立跟踪关系	git push -u origin new-feature

Works cited

1. What is version control | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/what-is-version-control
2. Git SCM, accessed August 26, 2025, https://git-scm.com/
3. What is Git | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/what-is-git
4. About Version Control - Git, accessed August 26, 2025, https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control
5. Get started with GitHub documentation - GitHub Docs, accessed August 26, 2025, https://docs.github.com/en/get-started
6. About GitHub and Git - GitHub Docs, accessed August 26, 2025, https://docs.github.com/en/get-started/start-your-journey/about-github-and-git
7. GitHub Docs, accessed August 26, 2025, https://docs.github.com/
8. GitHub glossary - GitHub Docs, accessed August 26, 2025, https://docs.github.com/en/get-started/learning-about-github/github-glossary
9. GitHub glossary, accessed August 26, 2025, https://docs.github.com/articles/github-glossary
10. 1.3 Getting Started - What is Git?, accessed August 26, 2025, https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F
11. Recording Changes to the Repository - Git, accessed August 26, 2025, https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository
12. Git Guides - git add · GitHub, accessed August 26, 2025, https://github.com/git-guides/git-add
13. git add — Tuto git, accessed August 26, 2025, https://gdevops.frama.io/opsindev/tuto-git/commands/add/add.html
14. First-Time Git Setup - Git, accessed August 26, 2025, https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup.html
15. 1.6 Getting Started - First-Time Git Setup, accessed August 26, 2025, https://git-scm.com/book/ms/v2/Getting-Started-First-Time-Git-Setup
16. Git Configuration - Git, accessed August 26, 2025, https://git-scm.com/book/be/v2/Customizing-Git-Git-Configuration
17. Understanding the git fork and pull request workflow - Graphite, accessed August 26, 2025, https://graphite.dev/guides/understanding-git-fork-pull-request-workflow
18. Understanding the difference between a fork and a pull request in GitHub -

Graphite, accessed August 26, 2025, https://graphite.dev/guides/understanding-fork-vs-pull-request-github
19. Contributing to a project - GitHub Docs, accessed August 26, 2025, https://docs.github.com/en/get-started/quickstart/contributing-to-projects
20. Pull Request Workflow with Git — 6 steps guide | by Hybesis - H.urna | Medium, accessed August 26, 2025, https://medium.com/@urna.hybesis/pull-request-workflow-with-git-6-steps-guide-3858e30b5fa4
21. A Complete Guide to GitHub Forks: From Setup to Pull Requests - Gun.io, accessed August 26, 2025, https://gun.io/news/2024/11/a-complete-guide-to-github-forks-from-setup-to-pull-requests/
22. git clone | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone
23. git-clone Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/git-clone
24. Git Guides - git clone - GitHub, accessed August 26, 2025, https://github.com/git-guides/git-clone
25. Clone a Git repository | Bitbucket Cloud - Atlassian Support, accessed August 26, 2025, https://support.atlassian.com/bitbucket-cloud/docs/clone-a-git-repository/
26. Learn Git with Bitbucket Cloud | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud
27. Git Remote | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/syncing
28. Git Remote | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/syncing/git-remote
29. 2.5 Git Basics - Working with Remotes, accessed August 26, 2025, https://git-scm.com/book/ms/v2/Git-Basics-Working-with-Remotes
30. Syncing a fork - GitHub Docs, accessed August 26, 2025, https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/syncing-a-fork
31. Basic Git Commands | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/glossary
32. Git Guides - git pull - GitHub, accessed August 26, 2025, https://github.com/git-guides/git-pull
33. Branches in a Nutshell - Git, accessed August 26, 2025, https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
34. Git Branch | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/using-branches
35. A3.4 Appendix C: Git Commands - Branching and Merging, accessed August 26, 2025, https://git-scm.com/book/ms/v2/Appendix-C:-Git-Commands-Branching-and-Merging
36. Branch Management - Git, accessed August 26, 2025, https://git-scm.com/book/be/v2/Git-Branching-Branch-Management

37. Basic Branching and Merging - Git, accessed August 26, 2025, https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging
38. git-checkout Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/git-checkout
39. What's the difference between 'git switch' and 'git checkout'
40. Git Checkout | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/using-branches/git-checkout
41. git-switch Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/git-switch
42. Git Status: Inspecting a repository | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/inspecting-a-repository
43. Git - Status - GeeksforGeeks, accessed August 26, 2025, https://www.geeksforgeeks.org/git/git-status/
44. Git Guides - git status - GitHub, accessed August 26, 2025, https://github.com/git-guides/git-status
45. Resolving a merge conflict using the command line - GitHub Docs, accessed August 26, 2025, https://docs.github.com/articles/resolving-a-merge-conflict-using-the-command-line
46. Git Commands - Basic Snapshotting, accessed August 26, 2025, https://git-scm.com/book/en/v2/Appendix-C:-Git-Commands-Basic-Snapshotting
47. Basic Snapshotting - Git Reference, accessed August 26, 2025, https://git.github.io/git-reference/basic/
48. git-commit Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/git-commit
49. git-commit(1) - The Linux Kernel Archives, accessed August 26, 2025, https://www.kernel.org/pub/software/scm/git/docs/git-commit.html
50. squash)
51. How can I commit files with git? - Stack Overflow, accessed August 26, 2025, https://stackoverflow.com/questions/7080803/how-can-i-commit-files-with-git
52. Git Status: Inspecting a repository | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-log
53. git-log Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/git-log
54. git log | A Guide to Using the log Command in Git, accessed August 26, 2025, https://initialcommit.com/blog/git-log
55. Power 'git log' graphing - zwischenzugs, accessed August 26, 2025, https://zwischenzugs.com/2016/06/04/power-git-log-graphing/
56. Command Line Git Graph with Colors - Build47, accessed August 26, 2025, https://build47.com/command-line-git-graph-with-colors/
57. Git Push | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/syncing/git-push
58. Git Guides - git push - GitHub, accessed August 26, 2025, https://github.com/git-guides/git-push

59. Remote Branches - Git, accessed August 26, 2025, https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches
60. Learn Branching with Bitbucket Cloud | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/learn-branching-with-bitbucket-cloud
61. Creating a pull request - GitHub Docs, accessed August 26, 2025, https://docs.github.com/articles/creating-a-pull-request
62. Creating a pull request from a fork - GitHub Docs, accessed August 26, 2025, https://docs.github.com/articles/creating-a-pull-request-from-a-fork
63. Pull Requests | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/making-a-pull-request
64. GitHub flow - GitHub Docs, accessed August 26, 2025, https://docs.github.com/en/get-started/using-github/github-flow
65. About pull requests - GitHub Docs, accessed August 26, 2025, https://docs.github.com/articles/about-pull-requests
66. Reviewing proposed changes in a pull request - GitHub Docs, accessed August 26, 2025, https://docs.github.com/articles/reviewing-proposed-changes-in-a-pull-request
67. About pull request reviews - GitHub Docs, accessed August 26, 2025, https://docs.github.com/articles/about-pull-request-reviews
68. Review and comment on pull requests - Azure Repos | Microsoft Learn, accessed August 26, 2025, https://learn.microsoft.com/en-us/azure/devops/repos/git/review-pull-requests?view=azure-devops
69. Preferred Github workflow for updating a pull request after code review - Stack Overflow, accessed August 26, 2025, https://stackoverflow.com/questions/7947322/preferred-github-workflow-for-updating-a-pull-request-after-code-review
70. Git merge conflicts | Atlassian Git Tutorial, accessed August 26, 2025, https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts
71. Resolve Git merge conflicts - Azure Repos | Microsoft Learn, accessed August 26, 2025, https://learn.microsoft.com/en-us/azure/devops/repos/git/merging?view=azure-devops
72. 7.8 Git Tools - Advanced Merging, accessed August 26, 2025, https://git-scm.com/book/ms/v2/Git-Tools-Advanced-Merging
73. gitglossary Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/gitglossary
74. git-merge Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/git-merge
75. git-pull Documentation - Git, accessed August 26, 2025, https://git-scm.com/docs/git-pull
76. What's the use of the staging area in Git? - Stack Overflow, accessed August 26, 2025, https://stackoverflow.com/questions/49228209/whats-the-use-of-the-staging-ar

ea-in-git
77. How Git Works - KodeKloud, accessed August 26, 2025, https://kodekloud.com/blog/how-git-works/
78. GIT CHEAT SHEET - GitHub Education, accessed August 26, 2025, https://education.github.com/git-cheat-sheet-education.pdf