附录 C: Git 命令

在这一整本书里我们介绍了大量的 Git 命令，并尽可能通过讲故事的方式来介绍它们，慢慢介绍了越来越多的命令。 但是这导致这些命令的示例用法都散落在全书的各处。

在此附录中，我们会将本书中所提到过的命令都过一遍，并根据其用途进行大致分类。 我们会大致地讨论每个命令的作用，指出其在本书中哪些章节使用过。

较长的选项可以。 例如，你可以输入 `git commit --a`，它的行为与 `git commit --amend` 相同。 这种方式只有在 `--` 后的字母对于该选项唯一时才可行。 请在编写脚本时使用完整的选项。

设置与配置

有两个十分常用的命令：`config` 和 `help`。 从第一次调用 Git 到日常微调及阅读参考，它们一直陪伴着你。

git config

Git 做的很多工作都有一种默认方式。 对于绝大多数工作而言，你可以改变 Git 的默认方式，或者根据你的偏好来设置。 这些设置涵盖了所有的事，从告诉 Git 你的名字，到指定偏好的终端颜色，以及你使用的编辑器。 此命令会从几个特定的配置文件中读取和写入配置值，以便你可以从全局或者针对特定的仓库来进行设置。

本书几乎所有的章节都用到了 `git config` 命令。

在 初次运行 Git 前的配置 一节中，在开始使用 Git 之前，我们用它来指定我们的名字，邮箱地址和编辑器偏好。

在 Git 别名 一节中我们展示了如何创建可以展开为长选项序列的短命令，以便你不用每次都输入它们。

在 变基 一节中，执行 `git pull` 命令时，使用此命令来将 `--rebase` 作为默认选项。

在 凭证存储 一节中，我们使用它来为你的 HTTP 密码设置一个默认的存储区域。

在 关键字展开 一节中我们展示了如何设置在 Git 的内容添加和减少时使用的 smudge 过滤器 和 clean 过滤器。

最后，基本上 配置 Git 整个章节都是针对此命令的。

git config core.editor 命令

就像 初次运行 Git 前的配置 里的设置指示，很多编辑器可以如下设置：

表格 5. 详细的 `core.editor` 设置命令列表

编辑器	设置命令
Atom	`git config --global core.editor "atom --wait"`
BBEdit (Mac, with command line tools)	`git config --global core.editor "bbedit -w"`
Emacs	`git config --global core.editor emacs`

编辑器	设置命令
Gedit (Linux)	`git config --global core.editor "gedit --wait --new -window"`
Gvim (Windows 64-bit)	`git config --global core.editor "'C:/Program Files/Vim/vim72/gvim.exe' --nofork '%*'"` (Also see note below)
Kate (Linux)	`git config --global core.editor "kate"`
nano	`git config --global core.editor "nano -w"`
Notepad (Windows 64-bit)	`git config core.editor notepad`
Notepad++ (Windows 64-bit)	`git config --global core.editor "'C:/Program Files/Notepad/notepad.exe' -multiInst -notabbar -nosession -noPlugin"` (Also see note below)
Scratch (Linux)	`git config --global core.editor "scratch-text-editor"`
Sublime Text (macOS)	`git config --global core.editor "/Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl --new-window --wait"`
Sublime Text (Windows 64-bit)	`git config --global core.editor "'C:/Program Files/Sublime Text 3/sublime_text.exe' -w"` (Also see note below)
TextEdit (macOS)	`git config --global --add core.editor "open -W -n"`
Textmate	`git config --global core.editor "mate -w"`
Textpad (Windows 64-bit)	`git config --global core.editor "'C:/Program Files/TextPad 5/TextPad.exe' -m` (Also see note below)
Vim	`git config --global core.editor "vim"`
VS Code	`git config --global core.editor "code --wait"`
WordPad	`git config --global core.editor '"C:\Program Files\Windows NT\Accessories\wordpad.exe"'`
Xi	`git config --global core.editor "xi --wait"`

如果你在 64 位 Windows 系统上安装了 32 位的编辑器，那么它会被安装在 `C:\Program Files (x86)\` 而非上表中所写的 `C:\Program Files\` 中。

git help

`git help` 命令用来显示任何命令的 Git 自带文档。 但是我们仅会在此附录中提到大部分最常用的命令，对于每一个命令的完整的可选项及标志列表，你可以随时运行 `git help <command>` 命令来了解。

我们在 获取帮助 一节中介绍了 `git help` 命令，同时在 配置服务器 一节中给你展示了如何使用它来查找更多关于 `git shell` 的信息。

获取与创建项目

有几种方式获取一个 Git 仓库。 一种是从网络上或者其他地方拷贝一个现有的仓库，另一种就是在一个目录中创

建一个新的仓库。

git init

你只需要简单地运行 `git init` 就可以将一个目录转变成一个 Git 仓库，这样你就可以开始对它进行版本管理了。

我们一开始在 获取 Git 仓库 一节中介绍了如何创建一个新的仓库来开始工作。

在 远程分支 一节中我们简单的讨论了如何改变默认分支。

在 把裸仓库放到服务器上 一节中我们使用此命令来为一个服务器创建一个空的裸仓库。

最后，我们在 底层命令与上层命令 一节中介绍了此命令背后工作的原理的一些细节。

git clone

`git clone` 实际上是一个封装了其他几个命令的命令。 它创建了一个新目录，切换到新的目录，然后 `git init` 来初始化一个空的 Git 仓库， 然后为你指定的 URL 添加一个（默认名称为 `origin` 的）远程仓库（`git remote add`），再针对远程仓库执行 `git fetch`，最后通过 `git checkout` 将远程仓库的最新提交检出到本地的工作目录。

`git clone` 命令在本书中多次用到，这里只列举几个有意思的地方。

在 克隆现有的仓库 一节中我们通过几个示例详细介绍了此命令。

在 在服务器上搭建 Git 一节中，我们使用了 `--bare` 选项来创建一个没有任何工作目录的 Git 仓库副本。

在 打包 一节中我们使用它来解包一个打包好的 Git 仓库。

最后，在 克隆含有子模块的项目 一节中我们学习了使用 `--recursive` 选项来让克隆一个带有子模块的仓库变得简单。

虽然在本书的其他地方都有用到此命令，但是上面这些用法是特例，或者使用方式有点特别。

快照基础

对于基本的暂存内容及提交到你的历史记录中的工作流，只有少数基本的命令。

git add

`git add` 命令将内容从工作目录添加到暂存区（或称为索引（index）区），以备下次提交。 当 `git commit` 命令执行时，默认情况下它只会检查暂存区域，因此 `git add` 是用来确定下一次提交时快照的样子的。

这个命令对于 Git 来说特别重要，所以在本书中被无数次地提及和使用。 我们将快速地过一遍一些可以看到的独特的用法。

我们在 跟踪新文件 一节中介绍并详细解释了 `git add` 命令。

然后，我们在 遇到冲突时的分支合并 一节中提到了如何使用它来解决合并冲突。

接下来，我们在 交互式暂存 一章中使用它来交互式的暂存一个已修改文件的特定部分。

最后，在 树对象 一节中我们在底层模拟了它的用法，以便你了解幕后发生了什么。

git status

`git status` 命令会显示工作区及暂存区域中不同状态的文件。 其中包含了已修改但未暂存，或已经暂存但没有提交的文件。 在一般的显示形式中，它会给你一些如何在这些暂存区之间移动文件的提示。

首先，我们在 检查当前文件状态 一节中介绍了 `status` 的基本及简单的形式。 虽然我们在全书中都有用到它，但是绝大部分的你能用 `git status` 做的事情都在这一章讲到了。

git diff

当需要查看任意两棵树的差异时，可以使用 `git diff` 命令。 此命令可以查看你工作环境与你的暂存区的差异（`git diff` 默认的做法），你暂存区域与你最后提交之间的差异（`git diff --staged`），或者比较两个提交记录的差异（`git diff master branchB`）。

首先，我们在 查看已暂存和未暂存的修改 一章中研究了 `git diff` 的基本用法，在此节中我们展示了如何查看哪些变化已经暂存了，哪些没有。

在 提交准则 一节中,我们在提交前使用 `--check` 选项来检查可能存在的空白字符问题。

在 确定引入了哪些东西 一节中,了解了使用 `git diff A…B` 语法来更有效地比较不同分支之间的差异。

在 高级合并 一节中我们使用 `-b` 选项来过滤掉空白字符的差异，及通过 `--theirs`、`--ours` 和 `--base` 选项来比较不同暂存区冲突文件的差异。

最后，在 开始使用子模块 一节中,我们使用此命令合 `--submodule` 选项来有效地比较子模块的变化。

git difftool

当你不想使用内置的 `git diff` 命令时。`git difftool` 可以用来简单地启动一个外部工具来为你展示两棵树之间的差异。

我们只在 查看已暂存和未暂存的修改 一节中简单的提到了此命令。

git commit

`git commit` 命令将所有通过 `git add` 暂存的文件内容在数据库中创建一个持久的快照，然后将当前分支上的分支指针移到其之上。

首先，我们在 提交更新 一节中涉及了此命令的基本用法。 我们演示了如何在日常的工作流程中通过使用 `-a` 标志来跳过 `git add` 这一步，及如何使用 `-m` 标志通过命令行而不启动一个编辑器来传递提交信息。

在 撤消操作 一节中我们介绍了使用 `--amend` 选项来重做最后的提交。

在 分支简介，我们探讨了 `git commit` 的更多细节，及工作原理。

在 签署提交 一节中我们探讨了如何使用 `-S` 标志来为提交签名加密。

最后，在 提交对象 一节中，我们了解了 `git commit` 在背后做了什么，及它是如何实现的。

git reset

`git reset` 命令主要用来根据你传递给动作的参数来执行撤销操作。 它可以移动 `HEAD` 指针并且可选的改变 `index` 或者暂存区，如果你使用 `--hard` 参数的话你甚至可以改变工作区。 如果错误地为这个命令附加后面的参数，你可能会丢失你的工作，所以在使用前你要确定你已经完全理解了它。

首先，我们在 取消暂存的文件 一节中介绍了 `git reset` 简单高效的用法，用来对执行过 `git add` 命令的文件取消暂存。

在 重置揭密 一节中我们详细介绍了此命令，几乎整节都在解释此命令。

在 中断一次合并 一节中，我们使用 `git reset --hard` 来取消一个合并，同时我们也使用了 `git merge --abort` 命令，它是 `git reset` 的一个简单的封装。

git rm

`git rm` 是 Git 用来从工作区，或者暂存区移除文件的命令。 在为下一次提交暂存一个移除操作上，它与 `git add` 有一点类似。

我们在 移除文件 一节中提到了 `git rm` 的一些细节，包括递归地移除文件，和使用 `--cached` 选项来只移除暂存区域的文件但是保留工作区的文件。

在本书的 移除对象 一节中，介绍了 `git rm` 仅有的几种不同用法，如在执行 `git filter-branch` 中使用和解释了 `--ignore-unmatch` 选项。 这对脚本来说很有用。

git mv

`git mv` 命令是一个便利命令，用于移到一个文件并且在新文件上执行`git add`命令及在老文件上执行`git rm`命令。

我们只是在 移动文件 一节中简单地提到了此命令。

git clean

`git clean` 是一个用来从工作区中移除不想要的文件的命令。 可以是编译的临时文件或者合并冲突的文件。

在 清理工作目录 一节中我们介绍了你可能会使用 `clean` 命令的大量选项及场景。

分支与合并

Git 有几个实现大部的分支及合并功能的实用命令。

git branch

`git branch` 命令实际上是某种程度上的分支管理工具。 它可以列出你所有的分支、创建新分支、删除分支及重命名分支。

Git 分支 一节主要是为 `branch` 命令来设计的，它贯穿了整个章节。 首先，我们在 分支创建 一节中介绍了它，然后我们在 分支管理 一节中介绍了它的其它大部分特性（列举及删除）。

在 跟踪分支 一节中，我们使用 `git branch -u` 选项来设置一个跟踪分支。

最后，我们在 Git 引用 一节中讲到了它在背后做一什么。

git checkout

`git checkout` 命令用来切换分支，或者检出内容到工作目录。

我们是在 分支切换 一节中第一次认识了命令及 `git branch` 命令。

在 跟踪分支 一节中我们了解了如何使用 `--track` 标志来开始跟踪分支。

在 检出冲突 一节中，我们用此命令和 `--conflict=diff3` 来重新介绍文件冲突。

在 重置揭密 一节中，我们进一步了解了其细节及与 `git reset` 的关系。

最后，我们在 HEAD 引用 一节中介绍了此命令的一些实现细节。

git merge

`git merge` 工具用来合并一个或者多个分支到你已经检出的分支中。 然后它将当前分支指针移动到合并结果上。

我们首先在 新建分支 一节中介绍了 `git merge` 命令。 虽然它在本书的各种地方都有用到，但是 `merge` 命令只有几个变种，一般只是 `git merge <branch>` 带上一个你想合并进来的一个分支名称。

我们在 派生的公开项目 的后面介绍了如何做一个 `squashed merge` （指 Git 合并时将其当作一个新的提交而不是记录你合并时的分支的历史记录。）

在 高级合并 一节中，我们介绍了合并的过程及命令，包含 `-Xignore-space-change` 命令及 `--abort` 选项来中止一个有问题的提交。

在 签署提交 一节中我们学习了如何在合并前验证签名，如果你项目正在使用 GPG 签名的话。

最后，我们在 子树合并 一节中学习了子树合并。

git mergetool

当你在 Git 的合并中遇到问题时，可以使用 `git mergetool` 来启动一个外部的合并帮助工具。

我们在 遇到冲突时的分支合并 中快速介绍了一下它，然后在 外部的合并与比较工具 一节中介绍了如何实现你自己的外部合并工具的细节。

git log

`git log` 命令用来展示一个项目的可达历史记录，从最近的提交快照起。 默认情况下，它只显示你当前所在分支的历史记录，但是可以显示不同的甚至多个头记录或分支以供遍历。 此命令通常也用来在提交记录级别显示两个或多个分支之间的差异。

在本书的每一章几乎都有用到此命令来描述一个项目的历史。

在 查看提交历史 一节中我们介绍了此命令，并深入做了研究。 研究了包括 `-p` 和 `--stat` 选项来了解每一个提交引入的变更，及使用`--pretty` 和 `--online` 选项来查看简洁的历史记录。

在 分支创建 一节中我们使用它加 `--decorate` 选项来简单的可视化我们分支的指针所在，同时我们使用 `--graph` 选项来查看分叉的历史记录是怎么样的。

在 私有小型团队 和 提交区间 章节中，我们介绍了在使用 `git log` 命令时用 `branchA..branchB` 的语法来查看一个分支相对于另一个分支, 哪一些提交是唯一的。 在 提交区间 一节中我们作了更多介绍。

在 <_merge_log>> 和 三点 章节中，我们介绍了 `branchA...branchB` 格式和 `--left-right` 语法来查看哪些仅其中一个分支。 在 合并日志 一节中我们还研究了如何使用 `--merge` 选项来帮助合并冲突调试，同样也使用 `--cc` 选项来查看在你历史记录中的合并提交的冲突。

在 引用日志 一节中我们使用此工具和 `-g` 选项 而不是遍历分支来查看 Git 的 `reflog`。

在 搜索 一节中我们研究了`-S` 及 `-L` 选项来进行来在代码的历史变更中进行相当优雅地搜索，如一个函数的历史。

在 签署提交 一节中，我们了解了如何使用 `--show-signature` 来为每一个提交的 `git log` 输出中，添加一个判断是否已经合法的签名的一个验证。

git stash

`git stash` 命令用来临时地保存一些还没有提交的工作，以便在分支上不需要提交未完成工作就可以清理工作目录。

贮藏与清理 一整个章节基本就是在讲这个命令。

git tag

`git tag` 命令用来为代码历史记录中的某一个点指定一个永久的书签。 一般来说它用于发布相关事项。

我们在 打标签 一节中介绍了此命令及相关细节，并在 为发布打标签 一节实践了此命令。

我也在 签署工作 一节中介绍了如何使用 `-s` 标志创建一个 GPG 签名的标签，然后使用 `-v` 选项来验证。

项目分享与更新

在 Git 中没有多少访问网络的命令，几乎所以的命令都是在操作本地的数据库。 当你想要分享你的工作，或者从其他地方拉取变更时，这有几个处理远程仓库的命令。

git fetch

`git fetch` 命令与一个远程的仓库交互，并且将远程仓库中有但是在当前仓库的没有的所有信息拉取下来然后存储在你本地数据库中。

我们开始在 从远程仓库中抓取与拉取 一节中介绍了此命令，然后我们在 远程分支 中看到了几个使用示例。

我们在 向一个项目贡献 一节中有几个示例中也都有使用此命令。

在 合并请求引用 我们用它来抓取一个在默认空间之外指定的引用，在 打包 中，我们了解了怎么从一个包中获取内容。

在 引用规范 章节中我们设置了高度自定义的 `refspec` 以便 `git fetch` 可以做一些跟默认不同的事情。

git pull

`git pull` 命令基本上就是 `git fetch` 和 `git merge` 命令的组合体，Git 从你指定的远程仓库中抓取内容，然后马上尝试将其合并进你所在的分支中。

我们在 从远程仓库中抓取与拉取 一节中快速介绍了此命令，然后在 查看某个远程仓库 一节中了解了如果你运行此命令的话，什么将会合并。

我们也在 用变基解决变基 一节中了解了如何使用此命令来来处理变基的难题。

在 检出冲突 一节中我们展示了使用此命令如何通过一个 URL 来一次性的拉取变更。

最后，我们在 签署提交 一节中我们快速的介绍了你可以使用 `--verify-signatures` 选项来验证你正在拉取下来的经过 GPG 签名的提交。

git push

`git push` 命令用来与另一个仓库通信，计算你本地数据库与远程仓库的差异，然后将差异推送到另一个仓库中。 它需要有另一个仓库的写权限，因此这通常是需要验证的。

我们开始在 推送到远程仓库 一节中介绍了 `git push` 命令。 在这一节中主要介绍了推送一个分支到远程仓库的基本用法。 在 推送 一节中，我们深入了解了如何推送指定分支，在 跟踪分支 一节中我们了解了如何设置一个默认的推送的跟踪分支。 在 删除远程分支 一节中我们使用 `--delete` 标志和 `git push` 命令来在删除一个在服务器上的分支。

在 向一个项目贡献 一整节中，我们看到了几个使用 `git push` 在多个远程仓库分享分支中的工作的示例。

在 共享标签 一节中，我们知道了如何使用此命令加 `--tags` 选项来分享你打的标签。

在 发布子模块改动 一节中，我们使用 `--recurse-submodules` 选项来检查是否我们所有的子模块的工作都已经在推送子项目之前已经推送出去了，当使用子模块时这真的很有帮助。

在 其它客户端钩子 中我们简单的提到了 `pre-push` 挂钩（hook），它是一个可以用来设置成在一个推送完成之前运行的脚本，以检查推送是否被允许。

最后，在 引用规范推送 一节中，我们知道了使用完整的 refspec 来推送，而不是通常使用的简写形式。 这对我

们精确的指定要分享出去的工作很有帮助。

git remote

`git remote` 命令是一个是你远程仓库记录的管理工具。 它允许你将一个长的 URL 保存成一个简写的句柄，例如 `origin`，这样你就可以不用每次都输入他们了。 你可以有多个这样的句柄，`git remote` 可以用来添加，修改，及删除它们。

此命令在 远程仓库的使用 一节中做了详细的介绍，包括列举、添加、移除、重命名功能。

几乎在此书的后续章节中都有使用此命令，但是一般是以 `git remote add <name> <url>` 这样的标准格式。

git archive

`git archive` 命令用来创建项目一个指定快照的归档文件。

我们在 准备一次发布 一节中，使用 `git archive` 命令来创建一个项目的归档文件用于分享。

git submodule

`git submodule` 命令用来管理一个仓库的其他外部仓库。 它可以被用在库或者其他类型的共享资源上。 `submodule` 命令有几个子命令, 如（`add`、`update`、`sync` 等等）用来管理这些资源。

只在 子模块 章节中提到和详细介绍了此命令。

检查与比较

git show

`git show` 命令可以以一种简单的人类可读的方式来显示一个 Git 对象。 你一般使用此命令来显示一个标签或一个提交的信息。

我们在 附注标签 一节中使用此命令来显示带注解标签的信息。

然后，我们在 选择修订版本 一节中，用了很多次来显示不同的版本选择将解析出来的提交。

我们使用 `git show` 做的最有意思的事情是在 手动文件再合并 一节中用来在合并冲突的多个暂存区域中提取指定文件的内容。

git shortlog

`git shortlog` 是一个用来归纳 `git log` 的输出的命令。 它可以接受很多与 `git log` 相同的选项，但是此命令并不会列出所有的提交，而是展示一个根据作者分组的提交记录的概括性信息

我们在 制作提交简报 一节中展示了如何使用此命令来创建一个漂亮的 changelog 文件。

git describe

`git describe` 命令用来接受任何可以解析成一个提交的东西，然后生成一个人类可读的字符串且不可变。 这是一种获得一个提交的描述的方式，它跟一个提交的 SHA-1 值一样是无歧义，但是更具可读性。

我们在 生成一个构建号 及 准备一次发布 章节中使用 `git describe` 命令来获得一个字符串来命名我们发布的文件。

调试

Git 有一些命令可以用来帮你调试你代码中的问题。 包括找出是什么时候，是谁引入的变更。

git bisect

`git bisect` 工具是一个非常有用的调试工具，它通过自动进行一个二分查找来找到哪一个特定的提交是导致 bug 或者问题的第一个提交。

仅在 二分查找 一节中完整的介绍了此命令。

git blame

`git blame` 命令标注任何文件的行，指出文件的每一行的最后的变更的提交及谁是那一个提交的作者。 当你要找那个人去询问关于这块特殊代码的信息时这会很有用。

只有 文件标注 一节中提到此命令。

git grep

`git grep` 命令可以帮助在源代码中，甚至是你项目的老版本中的任意文件中查找任何字符串或者正则表达式。

只有 Git Grep 的章节中提到此命令。

补丁

Git 中的一些命令是以引入的变更即提交这样的概念为中心的，这样一系列的提交，就是一系列的补丁。 这些命令以这样的方式来管理你的分支。

git cherry-pick

`git cherry-pick` 命令用来获得在单个提交中引入的变更，然后尝试将作为一个新的提交引入到你当前分支上。 从一个分支单独一个或者两个提交而不是合并整个分支的所有变更是非常有用的。

在 变基与拣选工作流 一节中描述和演示了 `Cherry picking`

git rebase

`git rebase` 命令基本是是一个自动化的 `cherry-pick` 命令。 它计算出一系列的提交，然后再以它们在其他地方以同样的顺序一个一个的 `cherry-picks` 出它们。

在 变基 一章中详细提到了此命令，包括与已经公开的分支的变基所涉及的协作问题。

在 替换 中我们在一个分离历史记录到两个单独的仓库的示例中实践了此命令，同时使用了 `--onto` 选项。

在 Rerere 一节中，我们研究了在变基时遇到的合并冲突的问题。

在 修改多个提交信息 一节中，我们也结合 `-i` 选项将其用于交互式的脚本模式。

git revert

`git revert` 命令本质上就是一个逆向的 `git cherry-pick` 操作。 它将你提交中的变更的以完全相反的方式的应用到一个新创建的提交中，本质上就是撤销或者倒转。

我们在 还原提交 一节中使用此命令来撤销一个合并提交。

邮件

很多 Git 项目，包括 Git 本身，基本是通过邮件列表来维护的。 从方便地生成邮件补丁到从一个邮箱中应用这些补丁,Git 都有工具来让这些操作变得简单。

git apply

`git apply` 命令应用一个通过 `git diff` 或者甚至使用 GNU diff 命令创建的补丁。 它跟补丁命令做了差不多的工作，但还是有一些小小的差别。

我们在 应用来自邮件的补丁 一节中演示了它的使用及什么环境下你可能会用到它。

git am

`git am` 命令用来应用来自邮箱的补丁。特别是那些被 mbox 格式化过的。 这对于通过邮件接受补丁并将他们轻松地应用到你的项目中很有用。

我们在 使用 `am` 命令应用补丁 命令中提到了它的用法及工作流，包括使用 `--resolved`、`-i` 及 `-3` 选项。

我们在 电子邮件工作流钩子 也提到了几条 hooks，你可以用来辅助与 `git am` 相关工作流。

在 邮件通知 一节中我们也将用此命令来应用 格式化的 GitHub 的推送请求的变更。

git format-patch

`git format-patch` 命令用来以 mbox 的格式来生成一系列的补丁以便你可以发送到一个邮件列表中。

我们在 通过邮件的公开项目 一节中研究了一个使用 `git format-patch` 工具为一个项目做贡献的示例。

git imap-send

`git imap-send` 将一个由 `git format-patch` 生成的邮箱上传至 IMAP 草稿文件夹。 我们在 通过邮件的公开项目 一节中见过一个通过使用 `git imap-send` 工具向一个项目发送补丁进行贡献的例子。

git send-email

`git send-mail` 命令用来通过邮件发送那些使用 `git format-patch` 生成的补丁。

我们在 通过邮件的公开项目 一节中研究了一个使用 `git send-email` 工具发送补丁来为一个项目做贡献的示例。

git request-pull

`git request-pull` 命令只是简单的用来生成一个可通过邮件发送给某个人的示例信息体。 如果你在公共服务器上有一个分支，并且想让别人知道如何集成这些变更，而不用通过邮件发送补丁，你就可以执行此命令的输出发送给这个你想拉取变更的人。

我们在 派生的公开项目 一节中演示了如何使用 `git request-pull` 来生成一个推送消息。

外部系统

Git 有一些可以与其他的版本控制系统集成的命令。

git svn

`git svn` 可以使 Git 作为一个客户端来与 Subversion 版本控制系统通信。 这意味着你可以使用 Git 来检出内容，或者提交到 Subversion 服务器。

Git 与 Subversion 一章深入讲解了此命令。

git fast-import

对于其他版本控制系统或者从其他任何的格式导入，你可以使用 `git fast-import` 快速地将其他格式映射到 Git 可以轻松记录的格式。

在 一个自定义的导入器 一节中深入讲解了此命令。

管理

如果你正在管理一个 Git 仓库，或者需要通过一个复杂的方法来修复某些东西，Git 提供了一些管理命令来帮助你。

git gc

`git gc` 命令在你的仓库中执行 “garbage collection”，删除数据库中不需要的文件和将其他文件打包成一种更有效的格式。

此命令一般在背后为你工作，虽然你可以手动执行它-如果你想的话。 我们在维护 一节中研究此命令的几个示例。

git fsck

`git fsck` 命令用来检查内部数据库的问题或者不一致性。

我们只在 数据恢复 这一节中快速使用了一次此命令来搜索所有的悬空对象（dangling object）。

git reflog

`git reflog` 命令分析你所有分支的头指针的日志来查找出你在重写历史上可能丢失的提交。

我们主要在 引用日志 一节中提到了此命令，并在展示了一般用法，及如何使用 `git log -g` 来通过 `git log` 的输出来查看同样的信息。

我们同样在 数据恢复 一节中研究了一个恢复丢失的分支的实例。

git filter-branch

`git filter-branch` 命令用来根据某些规则来重写大量的提交记录，例如从任何地方删除文件，或者通过过滤一个仓库中的一个单独的子目录以提取出一个项目。

在 从每一个提交中移除一个文件 一节中，我们解释了此命令，并探究了其他几个选项，例如 `--commit-filter`，`--subdirectory-filter` 及 `--tree-filter` 。

在 Git-p4 的章节中我们使用它来修复已经导入的外部仓库。

底层命令

在本书中我们也遇到了不少底层的命令。

我们遇到的第一个底层命令是在 合并请求引用 中的 `ls-remote` 命令。我们用它来查看服务端的原始引用。

我们在 手动文件再合并、 Rerere 及 索引 章节中使用 `ls-files` 来查看暂存区的更原始的样子。

我们同样在 分支引用 一节中提到了 `rev-parse` 命令，它可以接受任意字符串，并将其转成一个对象的 SHA-1 值。

我们在 Git 内部原理 一章中对大部分的底层命令进行了介绍，这差不多正是这一章的重点所在。 我们尽量避免了在本书的其他部分使用这些命令。

索引