

III 用户的第一个项目做出贡献

- (1) Fork 第一个 GitHub 仓库，并为其贡献自己的代码。
- (2) 解决忘记 Fork 带来的问题。
- (3) 创建有效的提交消息来展示所做的更改。
- (4) 创建 PR 以开始合并代码的过程。
- (5) 探索有效的 PR 工作流。
- (6) 审查 PR。

第 6 章 Fork 一个 Github 仓库

本章学习重点：

- (1) 理解 Fork 操作；
- (2) Fork 一个仓库；
- (3) 通过 Fork 和克隆来摆脱困境；
- (4) 通过 Fork 贡献代码；

如果用户在不是所有者或协作者的仓库上工作，当想执行除浏览文件以外的任何其他操作，就必须 Fork 仓库。本章将解释什么是 Fork？如何 Fork 仓库？并将 Fork 与克隆还有复制进行比较；本章还会讨论如何通过 Fork 贡献代码，并演示了用户如果在 Fork 之前已经对克隆进行了一些更改，如何将需要贡献的代码放入 Fork 中。

6.1 Fork 的介绍

仓库的 Fork 本质上就是仓库的一个副本。本着开源精神，Fork 是一种与其他开发者分享和学习的方式，开发人员可以 Fork 一个仓库的动机非常多，但最常见的三个原因如下：

- (1) 为别人的项目做贡献；
- (2) 使用别人的项目作为起点；
- (3) 在不影响别人项目的情况下试验这些代码。

如果用户不是项目的所有者或协作者，并且想做除浏览文件以外的任何事情，就必须 Fork 仓库。如果用户想对一个既不是所有者也不是协作者的仓库进行更改，就必须先对该仓库进行 Fork，以便在开始探索和修改代码时能处于正确的状态。不过无须担心，如果用户忘记了 Fork——请参阅本章后面的“Getting unstuck when cloning”一节，以获得帮助。

提示：在 GitHub 之前，一个开源项目的 Fork 往往带有负面含义。因为它不是源代码的副本，还是社区的分裂，它意味着一个小组将项目带向新方向的岔路口。在实践中，Forks 往往有利于整个生态系统，因为它们引入了新的想法。在某些情况下，Fork 中的最佳想法会回归到原始项目中，在 GitHub 上，Forks 更像是短期分支，要么合并回主代码，要么删除。

6.2 克隆、Fork 和复制

当克隆 GitHub 仓库时，表示正在计算机上创建该项目的本地副本。Fork 一个 GitHub 仓库会在 GitHub 账户上创建该仓库的副本，然后可以从那里克隆该仓库。原始仓库和 Fork 的那个仓库之间的链接将被保留，并允许用户将原始仓库上所做的更改拉入 Fork 的仓库中，或将所做的更改推送到原仓库中。

复制一个仓库是指用户制作的副本仓库不再具有指向原始仓库的链接。复制并不是开源工作流中常见的部分，因为它使推送修改到原始仓库变得更加困难。即便如此，复制一个仓库有时还是很有用的，例如当原始项目不再处于活跃状态而用户打算用副本仓库保持项目的活跃时。

6.3 克隆仓库

任何公共仓库都可以克隆，用户可以在计算机上运行代码并对其进行更改，但是如果用户没有对仓库的推送权限，将无法将这些更改推送到远程仓库。在第 4 章就描述了当用户是所有者时，如何在 GitHub Desktop 中克隆仓库。无论用户是所有者、协作者还是访客，这个过程都是一样的。

在克隆仓库之前，用户应该验证自己是否能够将更改推送到原仓库。验证是否拥有此能力的最简单方法是转到仓库主页：如果在主页右侧看到“Settings tab”选项卡，则表示可能拥有推送权限；如果没有，用户可能必须先 Fork 这个仓库，或者尝试在 GitHub 上编辑一个文件。若收到如图 6-1 所示的消息，表示没有直接为该项目做出贡献的权限。

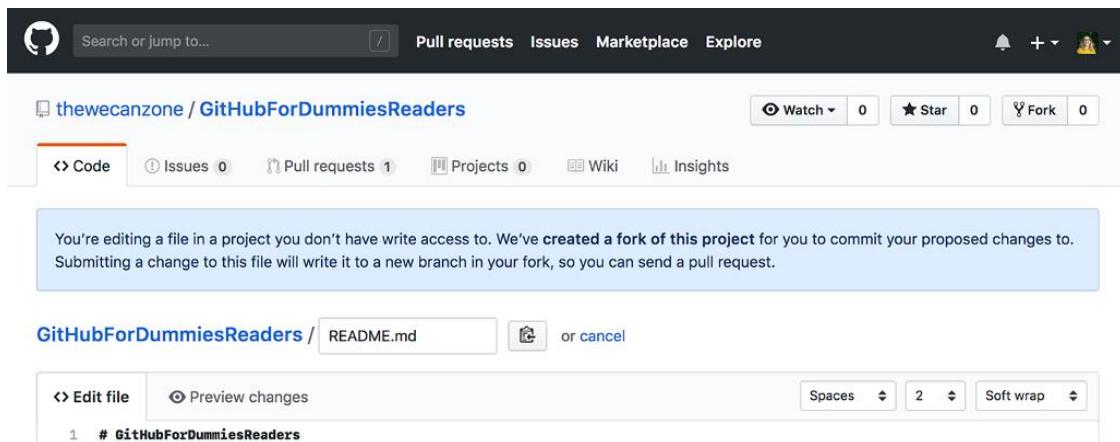


图 6-1：如果用户没有对仓库的编辑权限，GitHub.com 会出现的错误信息

警告：如果用户最终克隆了一个无权对其进行贡献的仓库，即使最终做了修改也无法推送它们。本章后面的“Getting unstuck when cloning”部分提供了如何摆脱这种状态的步骤。

在本地计算机上克隆仓库后，用户可以在终端中查看和修改它的元数据。打开终端并转到拥有 GitHub 仓库的目录。如果需要一个示例，请通过键入 `git clone`，代码如下：

```
$ git clone https://github.com/thewecanzone/
  GitHubForDummiesReaders
Cloning into 'GitHubForDummiesReaders'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 15 (delta 4), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (15/15), done.
$ cd GitHubForDummiesReaders
```

用户可以使用以下命令验证远程/目标仓库的位置：

```
$ git remote -v
origin https://github.com/thewecanzone/
  GitHubForDummiesReaders.git (fetch)
origin https://github.com/thewecanzone/
  GitHubForDummiesReaders.git (push)
```

如果用户克隆了同一个仓库，就会看到 `fetch` 和 `push` 的源 URL 完全相同，也可以看到远程仓库是由 `thewecanzone` 拥有的，而不是由 `sarah-wecan`。或者，如果用户在自己的仓库上运行相同的命令，应该会看到自己的用户名。例如，如果在克隆第 4 章中创建的网站仓库的目录中运行命令，就能看见。

```
$ git remote -v
origin https://github.com/sarah-wecan/sarah-wecan.github.io.git (fe
tch)
```

```
origin https://github.com/sarah-wecan/sarah-wecan.github.io.git (push)
```

如果用户尝试在一个没有远程源的 Git 仓库上使用该命令（这意味着它没有在 GitHub.com 或任何其他远程平台托管），将无法获得任何信息。例如，在第 1 章中，曾创建了一个名为 git-practice 的简单 Git 仓库，在该目录中运行命令不会给出任何反馈：

```
$ git remote -v
```

6.4 Fork 仓库

开源的目标是鼓励世界各地的软件开发人员之间进行协作，因此能够向不是项目所有者或明确协作者的仓库贡献代码，是 GitHub 工作流程和使命的重要组成部分。要成为开源项目的协作者，可以联系仓库的所有者并请求。但是必须首先获得他们的信任，否则可能会因为所有者无法确认对象从而被拒绝。

记住：Fork 仓库不需要项目所有者的许可。用户可以做出贡献并与所有者分享来展现对于项目的价值。

要 Fork 一个仓库，请转到仓库主页并单击右上角的 Fork 按钮。如果用户愿意，可以使用 (<https://github.com/thewecanzone/GitHubForDummiesReaders>) 来练习 Fork 并为公共仓库做出贡献。单击“Fork”按钮后，网页将刷新，可以看到该仓库的略微修改版本，如图 6-2 所示。在仓库的顶部，能看到仓库已附加到账户，但它仍然引用了原始仓库。

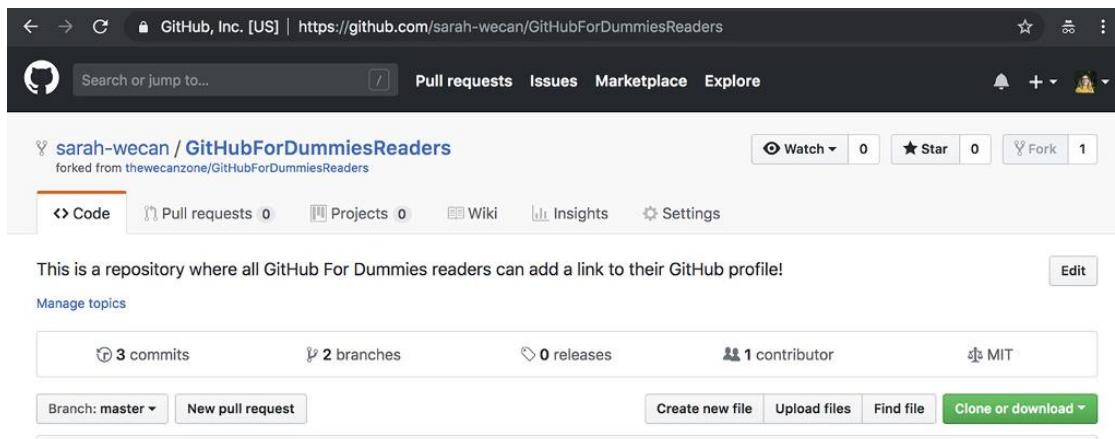


图 6-2：GitHub.com 上的 Fork 仓库。

警告：如果用户是多个 GitHub 组织的成员，那么在单击 Fork 按钮后，网页刷新之前，系统会要求用户选择要将仓库 Fork 到哪个组织。在拥有自己的 Fork 版本的仓库后，可以在本地计算机上克隆它以开始进行更改。第 7 章介绍了编写代码和创建提交，无论使用的是 Fork 的仓库还是原始仓库，过程都是相通的。如果使用 GitHub Desktop 克隆仓库，用户本地 git 仓库就会知道其 Fork 版本（远程 origin）和原始仓库（远程 upstream）。

提示：remote 的概念可能会让那些不熟悉分布式版本控制系统（如 Git）的人感到困惑。克隆 GitHub 仓库时，用户在本地计算机上拥有该仓库的完整副本。用户可能会认为 GitHub 上的仓库副本是标准副本，但是 Git 中是没有标准概念的，标准副本是项目工作人员通过共识决定的内容。

Git 确实有一个 remote 的概念，是指向托管在其他地方的相同 Git 仓库副本的指针。通常，remote 是指向 GitHub 等 Git 托管平台的 URL，也可能是包含仓库副本的目录的路径。当用户克隆一个仓库时，Git 会添加一个称为 origin 的 remote，其中包含其位置（通常是一个 URL）。但也可以将多个 remote 添加到 Git 仓库来表明希望从其他位置进行 push 和 pull 操作。例如，当用户克隆了仓库的一个 Fork，可能有一个名为 upstream 的远程指向原始仓库。

如果使用命令行克隆仓库，用户可能需要设置上游 remote，本章后面“Getting unstuck when cloning without forking”这节中对此进行了解释。如果在克隆仓库的目录中运行 git remote -v 命令，就可以看到 Fork 的仓库的远程 origin 和远程 upstream。

```
$ git remote -v
origin    https://github.com/sarah-wecan/
          GitHubForDummiesReaders.git (fetch)
origin    https://github.com/sarah-wecan/
          GitHubForDummiesReaders.git (push)
upstream  https://github.com/thewecanzone/
          GitHubForDummiesReaders.git (fetch)
upstream  https://github.com/thewecanzone/
          GitHubForDummiesReaders.git (push)
```

origin 是从 fetch/pull 更改并将更改 push 到的位置，带有用户的用户名（在此示例中为 sarah-wecan）。upstream 是原始代码所在的位置，也是最终希望想把自己写的代码贡献给它的位置，带有原始作者的用户名（在本例中为 thewecanzone）。

6.4.1 从上游获取更改

将上游仓库链接到用户的 Fork 仓库很重要。当开始进行更改时，用户希望能够将对原始代码所做的任何更改 fetch/pull 到代码中，以确保能拥有最新版本。

例如，假设一周前 Fork 并克隆了一个网站项目，并计划更改网站的“关于”页面。在用户进行这些更改时，其他用户对“关于”页面进行了更改。两边的更改可能会相同或发生冲突，所以在将更改提交回原始仓库之前，将这些更改拉入本地仓库是有意义的。它降低了不同用户间更改发生冲突的可能性，并使仓库所有者更可能接受更改。

如果用户发现自己需要从上游原始仓库获取更改，则可以转到自己所 Fork 的仓库所在的目录并键入。

```

$ git fetch upstream
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/thewecanzone/GitHubForDummiesReaders
  8404f3b..e02a4d2 master -> upstream/master
$ git checkout -b new-branch
Switched to a new branch 'new-branch'
$ git merge upstream/master
Updating 8404f3b..e02a4d2
Fast-forward
 README.md | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)

```

这三个命令从上游仓库中获取更改的部分，并确保在本地仓库和 Fork 的仓库中处于新分支，然后会将上游仓库中的新更改合并到自己的 Fork 仓库中。

6.4.2 为上游仓库做出贡献

在进行更改并将其发布到 Fork 仓库后，用户就可以向仓库的所有者提出更改建议了。如果访问 GitHub.com 上的原始上游仓库所有的分支就会显示在主页上，并且 GitHub 会询问是否要新建 pull request 以将更改与原始仓库合并（见图 6-3）。

上游项目名

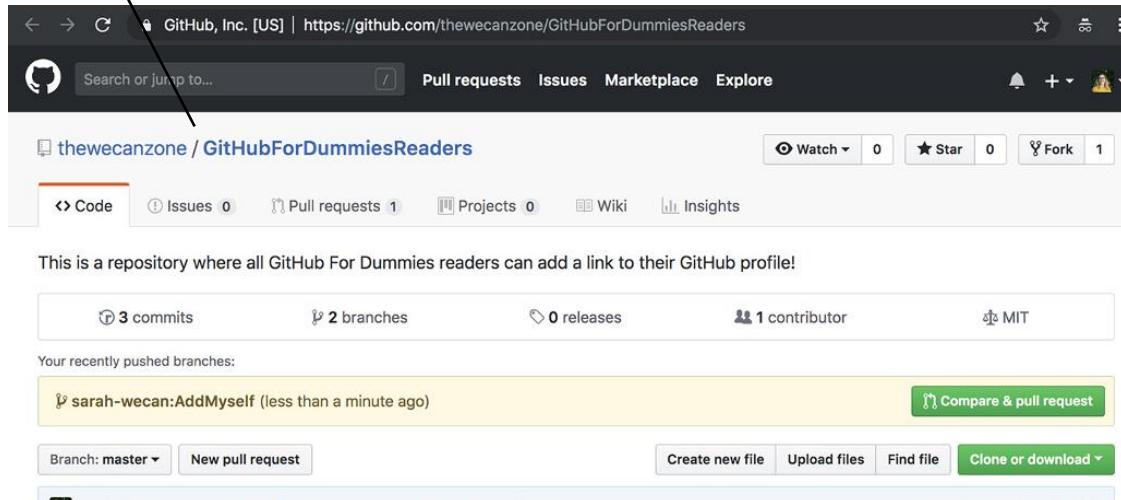


图 6-3 上游仓库检测到来自 Fork 仓库的分支

在 GitHub.com 上的 Fork 仓库中，会显示用户的分支，并且 GitHub 会询问是否要为其创建 PR（见图 6-4）。

单击“Compare & pull request”按钮，PR 创建页面为用户提供了请求将更改与上游仓库或 Fork 仓库合并的选项（参见图 6-5）。选择上游仓库，添加评论，然后单击创建 pull request 按钮。

用户看到分支可以合并，但是因为不是目标分支的所有者无法亲自合并该 PR；只有所有者（或指定的合作者）才有权合并代码。图 6-6 显示了没有合并选项的仓库上的 pull request。

Fork 仓库名称和来源：

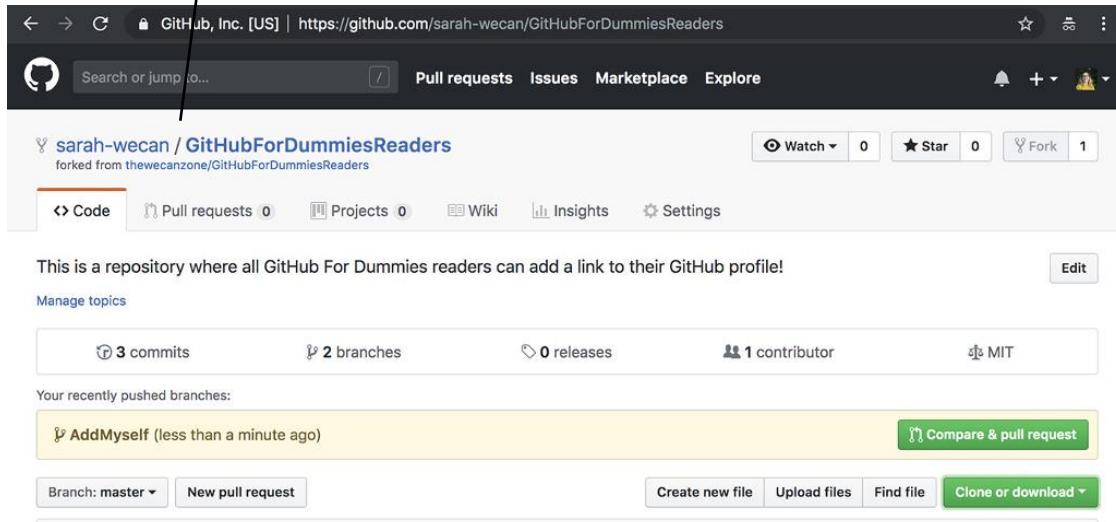


图 6-4：Fork 仓库检测到新分支。

目标仓库的选项

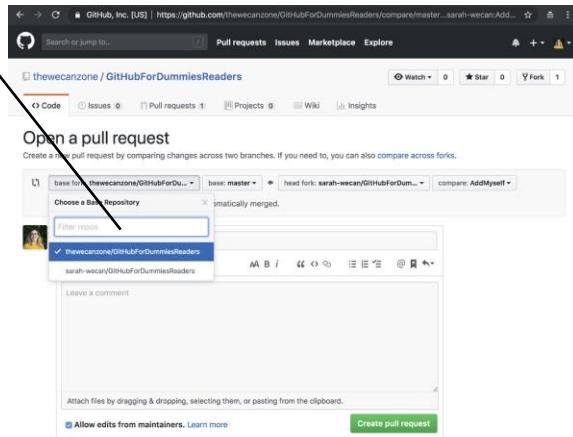


图 6-5：Fork 仓库检测到新的 pull request，并带有上游仓库或 Fork 仓库选项

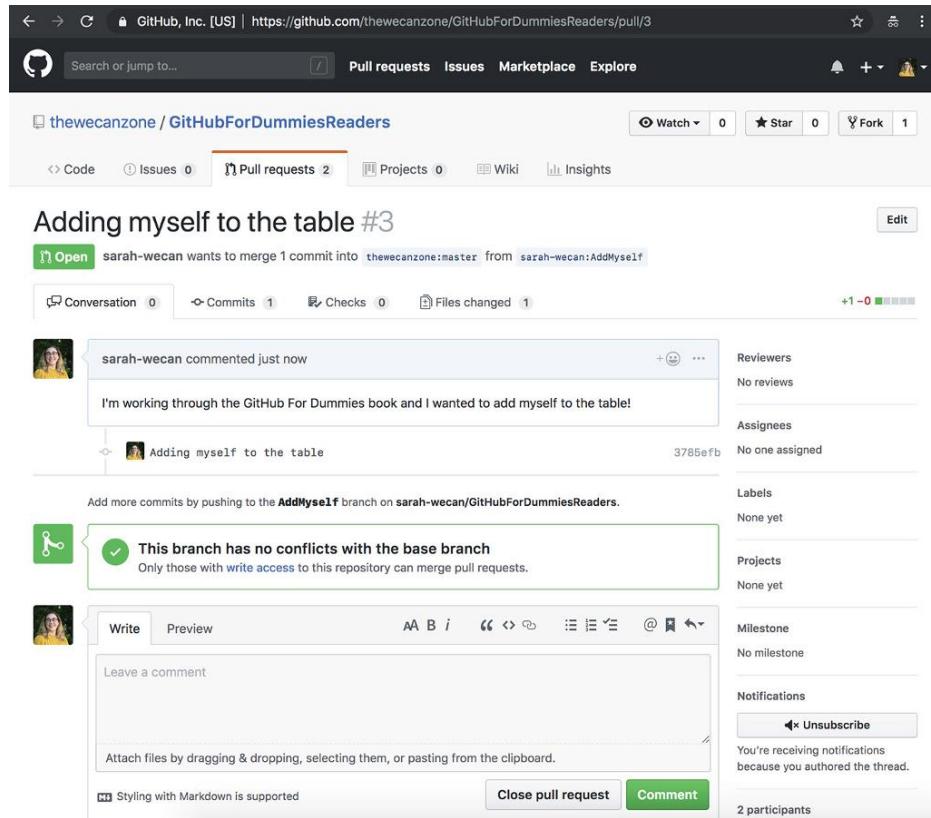


图 6-6：没有合并选项的 PR

作为上游仓库的所有者，就可以看到 pull request 并可以选择合并它（见图 6-7），如果正在此仓库上创建 pull request，将不断合并 pull request，以便可以保持所有 GitHub For Dummies 读者的最新表格。

提示：用户如果有很多修改想在请求合并到上游仓库之前添加到自己的 Fork 仓库中，那么可以首先创建 pull request，以自己的分支仓库而不是上游仓库为目标。这是对图 6-5 所示内容的更改。当准备好将更改合并到上游后就可以创建一个新的 pull request 以请求合并目标为上游仓库。

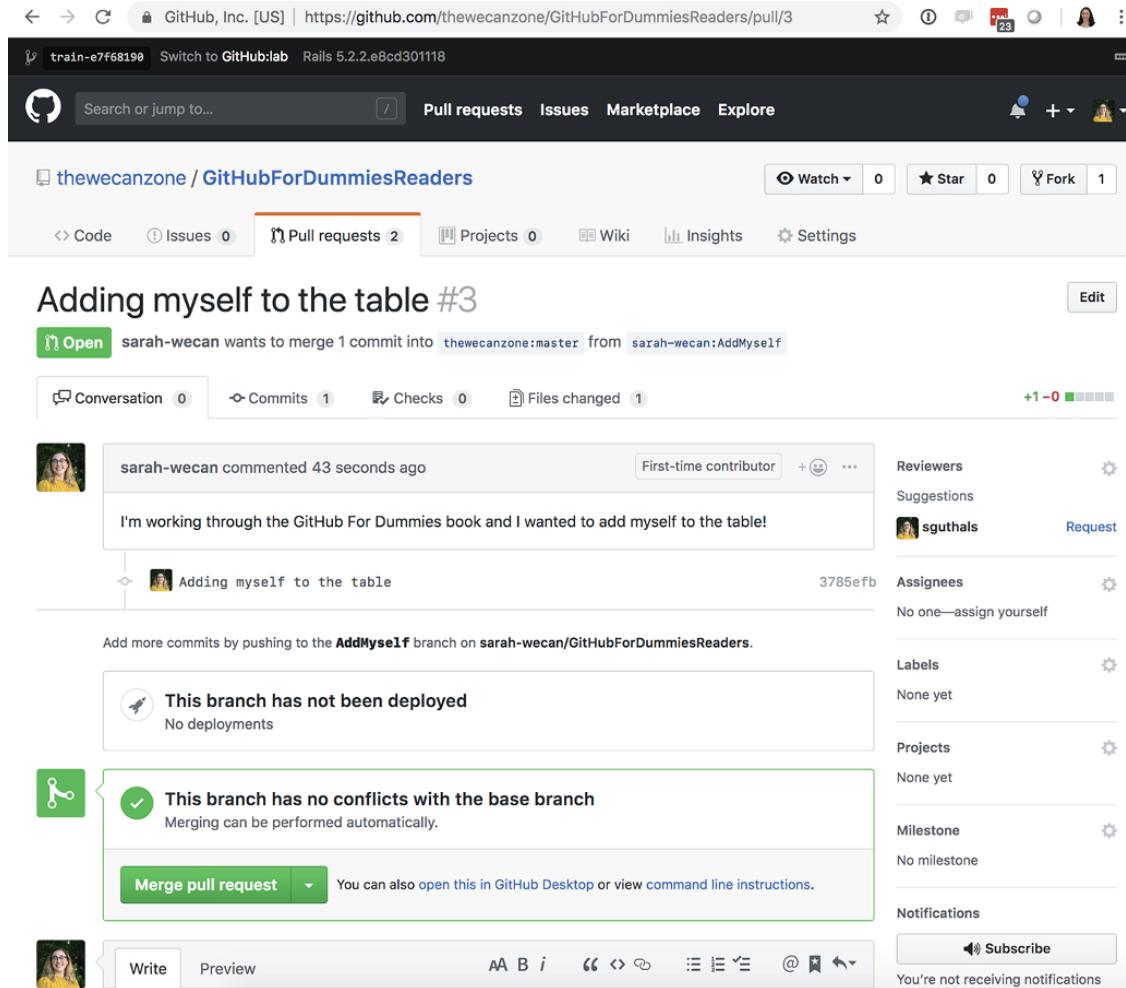


图 6-7：带有合并选项的 PR

6.4.3 解决忘记 Fork 带来的问题

人们遇到的一个常见问题是他们在尝试对其做出贡献之前忘记对仓库进行 Fork。以下场景描述了陷入这种情况的一个示例。

场景如下：用户将仓库克隆到本地计算机上，修改代码，将更改提交到 master，并准备好推送更改，但是随后会在 Atom（见图 6-8）、GitHub Desktop（见图 6-9）或终端中，收到一条看起来很吓人的错误消息：

```
$ git push origin master
remote: Permission to thewecanzone/GitHubForDummiesReaders.git
      denied to sarah-wecan.
fatal: unable to access 'https://github.com/thewecanzone/
      GitHubForDummiesReaders.git/': The requested URL returned
      error: 403
```

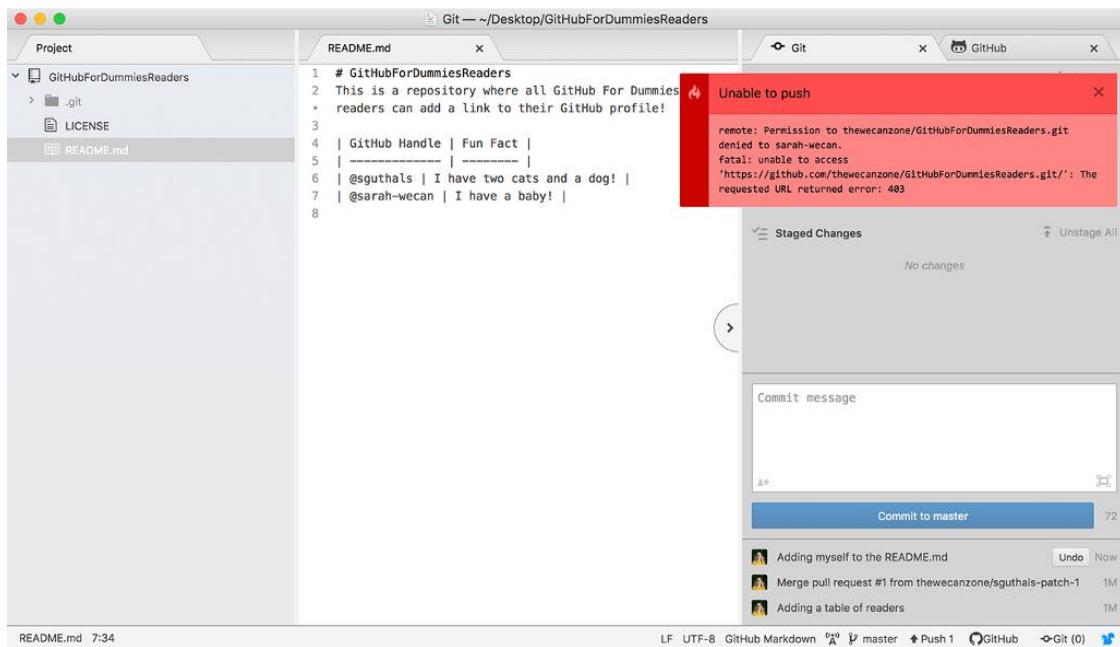


图 6-8：Atom 中的推送权限错误消息

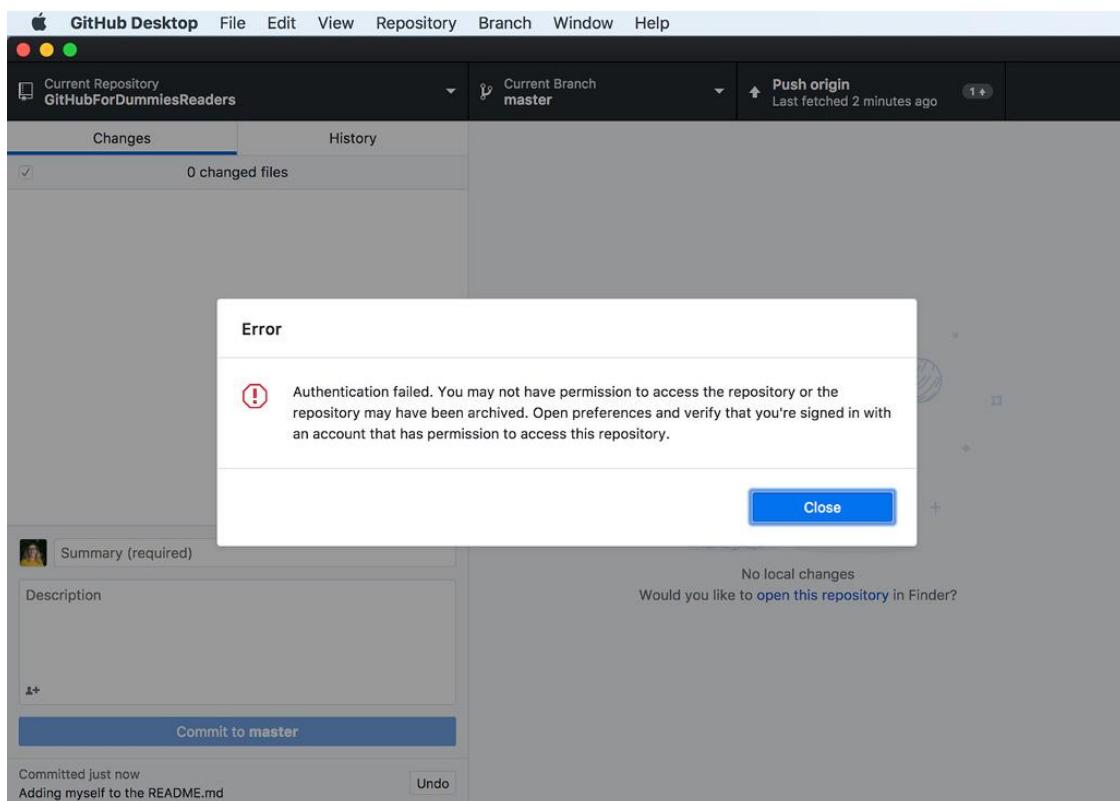


图 6-9：GitHub Desktop 中的推送权限错误消息

错误消息显示用户无权推送代码到此仓库，是因为应该先 Fork 仓库，并且不能直接提交给 master 分支。正如本书其他地方所推荐的那样，在一个分支中进行所有更改是一个很好的做法。

若要修复此错误就需要将更改移动到新分支，Fork 这个仓库，更改本地仓库的远程 URL 以指向刚刚 Fork 的仓库，然后再推送更改。这个过程可能会很棘手，但下面这些步骤可以帮助摆脱这种困境。

(1) 将更改迁移到新分支

当发现当前的目标是错误的远程仓库时，应该将更改移动到一个新分支。意外地将上游仓库的更改拉入刚刚完成的所有工作中是不当的。这一步可能会很棘手，但幸运地是 Phil 创建了一个 Git 别名来提供帮助。请参阅侧边栏 “Creating a Git Alias” 以获取帮助。获得 git migrate 别名后，转到终端中仓库所在的目录并键入：

```
$ git migrate new-branch
Switched to a new branch 'new-branch'
Branch 'master' set up to track remote branch 'master' from
  'origin'.
Current branch new-branch is up to date.
```

确认新分支已经创建：

```
$ git status
On branch new-branch
nothing to commit, working tree clean
```

提示：用户还可以通过运行 log 命令比较两个分支来确认提交仅在这个新分支中，而不再在旧分支中：

```
$ git log master..new-branch --oneline
```

这条命令列出了不在 master 中但存在于 new-branch 中的提交。--oneline 参数表示将每个 commit 打印在一行上，当只需要 commit 的摘要而不是 commit 的完整详细信息时非常有用。

(2) 将上游远程设置为原始仓库

要将上游远程添加到用户的仓库中，请转到终端并键入

```
$ git remote add upstream https://github.com/thewecanzone/GitHubForDummiesReaders.git
```

确认上游远程已正确添加：

```
$ git remote -v
origin https://github.com/thewecanzone/ GitHubForDummiesReaders.git
(fetch)origin https://github.com/thewecanzone/ GitHubForDummiesReaders.git
(push)upstream https://github.com/thewecanzone/ GitHubForDummiesReaders.git
(fetch)upstream https://github.com/thewecanzone/ GitHubForDummiesReaders.git
(push)
```

(3) Fork 仓库。

返回 GitHub.com，转到原始仓库并单击仓库主页右上角的 Fork。页面刷新后，就会看到用户的 Fork 仓库，它引用了原始仓库（参见图 6-2）。

(4) 将 origin remote 设置为用户的 Fork 仓库

在拥有了 Fork 仓库后，就可以将远程 origin 更改自己的版本：

```
$ git remote set-url origin https://github.com/sarah-wecan/GitHubForDummiesReaders.git
```

用户还可以确认所有远程 URL 都已正确设置：

```
$ git remote -v
origin https://github.com/sarah-wecan/GitHubForDummiesReaders.git (fetch)
origin https://github.com/sarah-wecan/GitHubForDummiesReaders.git (push)
upstream https://github.com/thewecanzone/GitHubForDummiesReaders.git (fetch)
upstream https://github.com/thewecanzone/GitHubForDummiesReaders.git (push)
```

(5) 将分支推送到自己的 Fork 仓库。

用户现在处于与克隆之前，Fork 仓库时所处的状态相同的状态。回到 Atom，可以发布自己的分支。

(6) 创建 pull request。

就像在本章前面的图 6-4 中一样，用户的 Fork 仓库检测到一个新分支并供创建 pull request。

6.4.4 创建一个 Git 别名

Git Alias 是一种自动化和扩展 Git 命令的简单方法。如果用户在终端上执行大量 Git 命令，那么创建 Git 别名可以使软件开发更加高效。例如，在终端中可以输入：

```
$ git config --global alias.st status
```

现在，可以只输入 git st，而不是输入 git status，Git 会返回仓库的当前状态。仅仅去掉四个字母似乎有点傻，但随着时间的推移，它最终会让 Git 命令体验更加高效。

Git 别名的功能远不止是减少需要按键的数量。Phil 在他的博客文章中描述了一个棘手的场景，用户必须把一个分支上所做的提交迁移到另一个分支。如果没有权限在一个仓库的克隆下开始工作并且陷入困境时，此迁移非常重要，正如本文前面的“Getting unstuck when cloning without forking”部分中所讨论的。

将提交从一个分支迁移到另一个分支的 Git 别名很复杂，这是将几个复杂的步骤全部合并到一个简单的 git migrate 命令中。在没有 Fork 的克隆仓库时如果需要使用此命令，请在终端中执行以下步骤：

```
$ open ~/.gitconfig
```

```
$
```

在默认编辑器中打开 .gitconfig 文件， 将以下代码添加到 .gitconfig 文件的底部：

```
[alias]
migrate = "!f(){ CURRENT=$(git symbolic-ref --short HEAD);
git checkout -b $1 && git branch --force $CURRENT
${3-$CURRENT@{u}} && git rebase --onto ${2-master} $CURRENT;
}; f"
```

如果 .gitconfig 文件已经有 [alias] 部分，请不要重新键入该行。 保存并关闭 .gitconfig 文件。

现在可以使用 git migrate 命令将 commits 从一个分支迁移到另一个分支，这个 Git 命令有一个必需参数和两个可选参数：

```
git migrate <new-branch-name> <target-branch> <commit-range>
```

参数 <new-branch-name> 是必需的。 这个分支是将 commit 移动到的地方。 如果用户不指定任何其他内容，那么 migrate 命令会将所有提交从 master 分支移动到这个新分支。

参数 <target-branch> 和 <commit-range> 是可选的。 <target branch> 允许用户将 commits 从 master 分支以外的分支移动到第一个参数中指定的 <new-branch-name>。 <commit-range> 允许指定要移动的提交。 如果用户不小心在错误的分支上进行了一次提交，并且只想将该提交移至 <new-branch-name>，则此参数会很有用。

第7章 编写并提交代码

本章学习重点：

- (1) 在终端中提交代码；
- (2) 创建一个好的 commit；
- (3) 编写 commit 消息；
- (4) 提交其他工具；

在本章中，我们将编写和提交代码。第一部分，编写代码，是一个非常广泛的主题，广泛到以至于无法在这本书（或任何一本书）中涵盖全部内容。本章中所提的编写代码仅为介绍如何创建良好的 commit 而服务。而本章的大部分内容都集中在如何提交代码上。无论编写何种类型的代码，提交该代码的行为都是相同的。

在本章中使用的代码示例可能看起来很简单。但是，不要让简单分散注意力，因为本章中的信息也适用于大型代码库。

7.1 创建仓库

commit 是 Git 的最小工作单元。它代表了对仓库的相关更改的一个小的逻辑组。commit 还表示时间快照，也就是整个仓库的状态可以通过引用单个 commit 来表示。

在编写代码之前，需要创建一个本地仓库来存储代码。在以下示例中，我们在名为 best-example 的目录中创建了一个仓库。随意将 best-example 更改为选择的目录。幸运的是，这个过程快速而轻松：

- (1) 在计算机上打开终端，请参阅第 1 章以获得指导。
- (2) 转到要存储项目文件夹的目录并输入以下命令：

```
$ git init best-example  
$ cd best-example
```

第一个命令在指定目录中创建一个空的 Git 仓库 best-example。由于 best-example 目录尚不存在，因此 Git 会创建它。第二个命令将当前目录更改为这个新目录。

提示：我见过的几乎所有涵盖初始化 Git 仓库的 Git 教程都是通过调用不带参数的“git init”或“git init.”在当前目录中执行此操作，其中“.”代表当前目录。事实上，可将创建仓库目录和初始化当前目录合并为一个步骤，即将这两个命令组合成一个命令：“git init best-example && cd best-example”。这个技巧可以获得效率较低的同伴的钦佩和崇拜！

7.2 编写代码

进入 Git 仓库目录后，开始添加文件。如果不在目录中，请参阅上一节“创建仓库”，我们在其中创建了 best-example 目录。对于本示例，通过键入以下代码创建三个文件：

```
$ touch README.md  
$ touch index.html  
$ mkdir js  
$ touch js/script.js
```

请注意，创建的文件之一是 README.md 文件。要了解为什么每个仓库都应该有一个 README.md 文件，请参阅第 10 章。运行这些命令后有三个文件 README.md、index.html、script.js。其中，script.js 位于名为 js 的子目录中，先丰富 README.md 文件。在这个例子中，我们使用 Atom 打开和编辑当前目录中的文件。（如果需要任何设置 Atom 的指导，请参阅第 2 章。）

我们可以便捷地使用其他编辑器（例如 Visual Studio Code）通过将 Atom 替换为以下示例中的代码来编辑文件：

```
$ atom .
```

在 README.md 文档中添加一些简单的 Markdown 文本。Markdown 是一种语言，它提供了一种简单的方式来设置文本格式和样式。在 GitHub 指南上查看 Markdown 指南。

通过单击 Atom 中的文件树，在编辑器中打开 README.md。然后添加一些与项目相关的 Markdown。在此示例中，添加以下文本：

```
# The Best Example Ever  
Which will be a part of the best commit ever.
```

然后将以下代码添加到 index.html。

```
<!doctype html>  
<html lang="en">  
<head>  
  <meta charset="utf-8">  
  <title>It is the cod3z</title>  
  <script src="js/script.js"></script>  
</head>  
<body>  
  <h1>The Best Cod3z!</h1>  
</body>  
</html>
```

此 HTML 文件引用了 scripts.js。在 Atom 中打开 script.js 并添加以下代码。

```
document.addEventListener(  
  "DOMContentLoaded",  
  function(event) {
```

```
    alert('The page is loaded and the script ran!')  
}  
);
```

现在从浏览器中打开 index.html 文件来测试代码。

```
$ open index.html
```

该页面将加载到默认浏览器中，并出现如图 7-1 所示的警告消息。

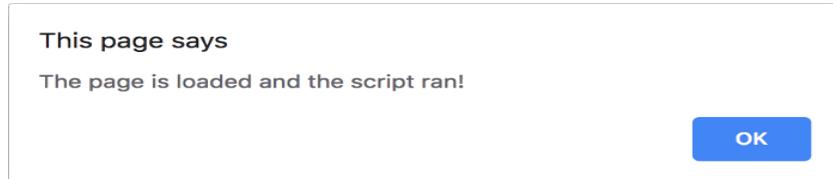


图 7-1：来自我们正在运行的代码的警告消息。

7.3 创建 Commit

本节假设在本地计算机上更改了代码，并且代码处于工作状态。如果需要工作代码示例，请参阅本章的前一节以达到此阶段。运行代码后，可以将其提交到仓库。创建提交有两步过程：①暂存要提交的更改；②创建带有 commit 消息的 commit。

7.3.1 暂存更改

暂存更改可能会让 Git 初学者感到困惑。从概念上讲，它类似于网站的 staging environment（准生产环境）。暂存更改可看成是一个中间步骤，允许在正式提交更改之前查看将要提交的更改。为什么要在提交之前进行暂存更改？在 Git 中，提交通常应该包含一组相关的更改。事实上，这样的做法是 Git 鼓励的。

假设已经工作了几个小时，现在有大量不相关的更改未提交到 Git 仓库。可能会想用一些通用的提交指示信息（比如提交“*A bunch of changes.*”）来提交所有的东西。提交一堆不相关的更改通常是个坏主意。仓库的提交历史讲述了项目如何随时间变化的故事。每个提交都应该代表一组明显的连贯变化。这种提交的方法不仅仅是讲究和有条理。拥有干净的 Git 历史具有具体的好处。

提示：干净的 Git 历史记录的一个好处是，当每次提交都是一个逻辑工作单元时，像 git bisect 这样的命令会更有用。git bisect 命令是一个高级命令，对其功能的完整介绍超出了本书的范围。简而言之，git bisect 提供了一种通过对 Git 历史进行二分查找的方法，以便找到引入特定行为（例如 bug）的特定提交。如果每个提交都包含了一大组不相关的更改，那么找到引起 bug 的 commit 就没有那么简单。

在本章的示例中，我们可能会创建两个 commit：一个只包含 README.md 文件的文件；另一个包含 index.html 和 script.js 文件。因为 index.html 文件引用了 script.js 文件，此时提交一个而不提交另一个是没有意义的。

首先暂存 README.md 文件：

```
$ git add README.md
```

README.md 文件被添加到 Git 索引中。Git 索引用来暂存仓库中正在创建的 commit。检查仓库的状态以查看文件是否已添加到索引中：

```
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file: README.md      Untracked files:
              (use "git add <file>..." to include in what will be committed)
    index.html  js/
```

README.md 文件已暂存以进行提交。同时，index.html 和 js/ 目录还未被此仓库跟踪。

提示：为什么 script.js 没有列在未跟踪文件部分？Git 在这里走了捷径。

它注意到没有跟踪 js 目录中的文件，因此它可以简单地列出目录而不是列出目录中的每个文件。在更大的代码库中，会很高兴 Git 没有列出每个子目录中的每个文件。

7.3.2 提交文件

暂存更改后（请参阅上一节），创建 commit。在这个例子中，我们使用 -m 标志和 git commit 命令来指定一个简短的 commit 消息。以下命令演示了如何一步创建 commit 并指定 commit 消息：

```
$ git commit -m "Add a descriptive README file"
[master (root-commit) 8436866] Add a descriptive README file
1 file changed, 3 insertions(+), 0 deletions(-)
create mode 100644 README.md
```

文件已提交。如果再次运行 git status 命令会看到仍有未跟踪的文件。git commit 命令仅提交暂存的更改。

7.3.3 提交多个文件

提交第一个文件后，就可以暂存其余文件以进行提交。

```
$ git add -A
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file: index.html
    new file: js/scripts.js
```

-A 标志表示要将工作目录中的所有更改添加到 Git 索引。当运行 git status 命令时，可以看到已经暂存了两个文件。

提示：当 js 目录未跟踪时，git status 只列出 js 目录，不列出目录中的任何文件。现在正在尝试暂存 js 目录，Git 却列出了 js 目录中的文件。为什么会出现这样的差异？Git 实际上并不跟踪目录。它只跟踪文件。因此，当目录添加到 Git 仓库时，需要将每个文件添加到索引中。

有时需要编写更详细的 commit 消息。在这个例子中，我们在运行 commit 命令时没有指定 commit 消息，因为我们打算编写更详细的 commit 消息：

```
$ git commit
```

如果没有使用 -m 标志指定 commit 消息，Git 会启动一个编辑器来创建 commit 消息。如果尚未使用 Git 配置编辑器，它会使用系统默认编辑器，通常是 VI 或 VIM。

关于退出 VIM 有多困难的笑话有很多，所以我们不会在这里重述。

提示：对于记录，要退出 VIM，请按 ESC 键退出编辑模式并键入 :wq 退出并保存或 :q! 退出而不保存。

要将默认编辑器更改为 Atom 之类的内容，请在终端中运行以下命令：

```
$ git config --global core.editor"atom--wait"
```

编辑器打开一个名为 COMMIT_EDITMSG 的临时文件，其中包含一些被注释掉的指令：

```
# Please enter the commit message for your changes. Lines starting# w
ith '#' will be ignored, and an empty message aborts the commit.## O
n branch master## Initial commit## Changes to be committed:# new file:
README.md## Untracked files:# index.html# js/#
```

可在打开的文件中输入 commit 消息，也可以将 commit 消息写在所有注释之前，或者简单地用自己的 commit 消息替换文件中的所有内容。

在这种情况下，我们将该文件中的所有内容替换为

```
Add index.html and script.jsThis adds index.html to the project. This f
ile is the default page when visiting the website.This file references j
s/script.js, which is also added in this commit.
```

在保存 commit 消息并关闭文件或编辑器后，Git 会使用编写的消息创建一个 commit。

7.4 写一个好的 commit 消息

应该在怎样写 commit 消息？什么是好的 commit 消息？一个 Git Commit 应该包含一个合乎逻辑的、连贯的更改或一组更改。commit 消息应清楚地描述该更改，以便稍后阅读该消息的任何人都了解 commit 中的更改。

commit 消息的受众是项目当前和未来的合作者。这些合作者将来可能包括自己。写清楚的 commit 消息，对未来的会有帮助。

如果发现在描述 commit 时遇到问题，则可能是该 commit 包含太多更改。在编写代码时，我们经常注意到写得好的函数只做一件事并且做得很好。同样，一个 commit 应该代表对系统的一次更改。commit 消息描述了更改以及更改的原因。

一个好的 commit 消息也应该遵循特定的结构。一般来说，commit 消息有两部分：（1）总结应简短（50 个字符或更少）并使用祈使式现在时。例如，不应当写

“I added a method to Frobnciate widgets”，而要写“Add method that Frobnciates widgets”。

如果需要，说明提供了详细的解释性文字。并非每个更改都需要解释性文本。例如，如果重命名一个函数，那么仅包含“Rename Frobnciate to Bublecate”摘要的commit消息就足够了。应该将描述文本包裹在72个字符处。这确保它在终端中作为git log命令输出的一部分显示时看起来不错。按照惯例，换行符将摘要与描述分开。

这是某开源项目中的一位作者编写的示例commit消息：

```
Avoid potential race conditionIn theory, if "ClearFormCache" is called  
after wecheck `contains` but before we execute the `return` line, we cou-  
ld get an exception here.If we're concerned about performance here, we  
couldconsider switching to the ConcurrentDictionary.
```

可以在Git commit消息中使用一些约定，Git会忽略这些约定，但GitHub会识别这些约定。例如，可以指定一次commit解决特定的issue，例如“修复#123”，其中123是issue编号。当具有此模式的commit被推送到GitHub时，issue编号将链接到该issue。当包含该提交的分支合并到仓库的默认分支（通常是master）时，GitHub会关闭相关的issue。这很方便！

提示：还可以在commit消息中使用表情符号。例如，一些团队使用的一种模式是通过以:art:开头来指示提交仅包含外观更改。当该提交在GitHub.com上呈现时，GitHub会呈现表情符号。可以在图7-2中看到此操作，其中显示了来自GitHub for Visual Studio的某开源项目的提交列表。

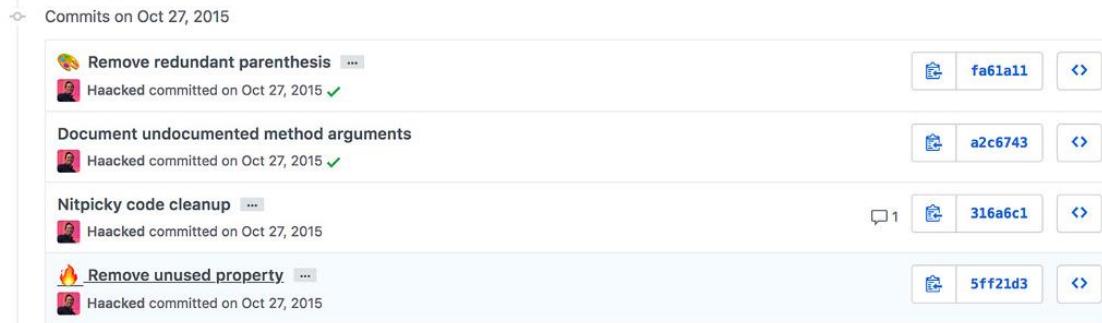


图7-2：commit消息列表，其中一些在摘要中带有表情符号

7.5 使用Desktop提交代码

尽管从终端提交commit非常简单，但许多人更喜欢使用GUI应用程序提交代码。使用GUI有几个好处：（1）GUI可以提供完成commit消息时的指导，例如将摘要保留为50个字符并通过新行将其与描述分开。GUI可以简单地呈现两个字段：摘要和描述。（2）GUI可以提供对GitHub的一些特殊约定的支持，例如指定某个commit专门解决某个issue。

GitHub Desktop（我们简称为 Desktop）是一个由 GitHub 创建的 GUI，非常适合提交代码。

7.5.1. 在 Desktop 上跟踪仓库

选择一个从未在 Desktop 中打开过但在本地计算机上拥有的仓库（使用 Desktop，请参阅第 2 章）。如果需要示例，请使用本章前面“创建一个仓库”部分中创建的 best-example 仓库。启动 Desktop 时，best-example 仓库未列在仓库列表中。

Desktop 不会扫描计算机以查找要管理的 Git 仓库。相反，必须将要管理的每个仓库告诉 Desktop。正如预期的那样，如果使用 Desktop 来克隆或创建仓库，那么 Desktop 就已经在跟踪。但有时有一个在 Desktop 之外克隆或创建的仓库——例如，我们使用终端创建了 best-example。现在需要告诉 Desktop 跟踪选择的仓库。幸运的是这个任务在终端上很容易完成。

Desktop 命令行工具允许用户从终端启动 Desktop，这可以根据需要轻松地将 Desktop 集成到现有的基于终端的 Git 工作流程中。

在 Windows 上，不需要安装命令行工具；它是自动完成的。在 Mac 上，必须采取单独的步骤。

在 Mac 上安装命令行工具：

(1) 确保 Desktop 正在运行中，然后在应用程序菜单栏中，选择“GitHub Desktop ⇄”安装命令行工具；

(2) 在终端进入要跟踪的文件目录。对于这个例子，我们选择 best-example 仓库。

(3) 运行以下命令：

```
$ github .
```

符号“.”在命令中代表当前目录。目录也可以是绝对路径。

GitHub Desktop 启动（如果它还没有运行）并打开指定的目录。由于当前目录已经是一个 Git 仓库，Desktop 会将其添加到它跟踪的仓库列表中。然后将该仓库设置为当前仓库，以便浏览仓库的历史记录、切换分支和创建提交，如图 7-3 所示。

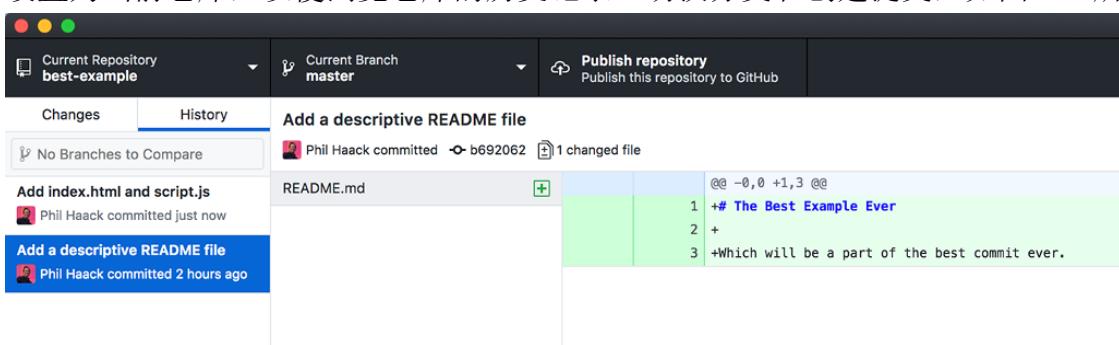


图 7-3：GitHub Desktop 打开到 best-example 仓库

如果当前目录不是仓库，Desktop 会提示在该目录中创建一个 Git 仓库，非常方便！

7.5.2. 在 Desktop 上发布仓库

要体验 Desktop 与 GitHub 集成的全部功能，需要将此仓库发布到 GitHub。

(1) 单击发布仓库按钮。会出现一个用于发布仓库的对话框（参见图 7-4）。

(2) 填写详细信息，然后单击“publish repository”按钮，仓库是在 GitHub.com 账户上创建的。

提示：GitHub Desktop 提供了一个键盘快捷键来打开浏览器到仓库： $\text{⌘} + \text{Shift} + \text{G}$ （在 Windows 上是 $\text{Ctrl} + \text{Shift} + \text{G}$ ）。

如果需要，在仓库中创建一些 issue（请参阅第 3 章了解如何创建 issue）。对于此示例，我们创建五个 issue：(1) 提供有关 README 的更多详细信息；(2) 在 readme 中提到项目需要共同努力；(3) 向 README 添加贡献部分；(4) 不要使用警告消息；(5) 设置网站警告。

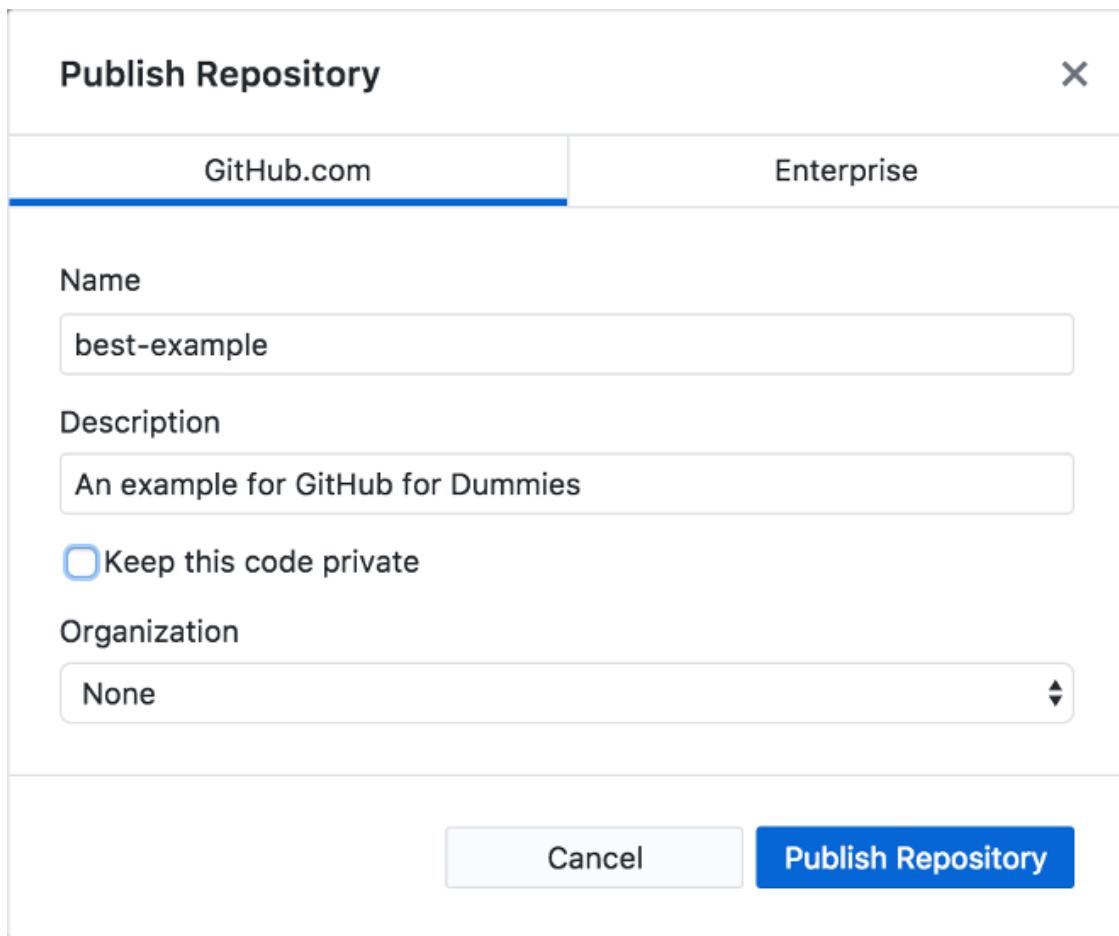


图 7-4 用于将仓库发布到 GitHub.com 的“发布”对话框。

还可以在我们的仓库上查看这些 issue。

7.5.3 在 Desktop 上提交

Desktop 仅适用于 Git 操作。如果要编辑仓库中的文件，仍然需要使用选择的编辑器。对文件进行一些更改，以便提交一些内容。在本章的示例中，对以粗体显示的 index.html 进行一些更改。

```
<!doctype html><html lang="en"><head> <meta charset="utf-8"> <title>**The Best Example**</title> <script src="js/script.js"></script></head><body> <h1>The Best Cod3z!</h1><div id="message"></div></body></html>
```

更新 script.js 以填充新的 DIV 元素，而不是弹出警告消息。更改以粗体显示：

```
document.addEventListener( "DOMContentLoaded", function(event) { var message = document.getElementById('message') message.innerText = 'The script ran!' });
```

切换回 Desktop 并单击更改选项卡，如图 7-5 所示。

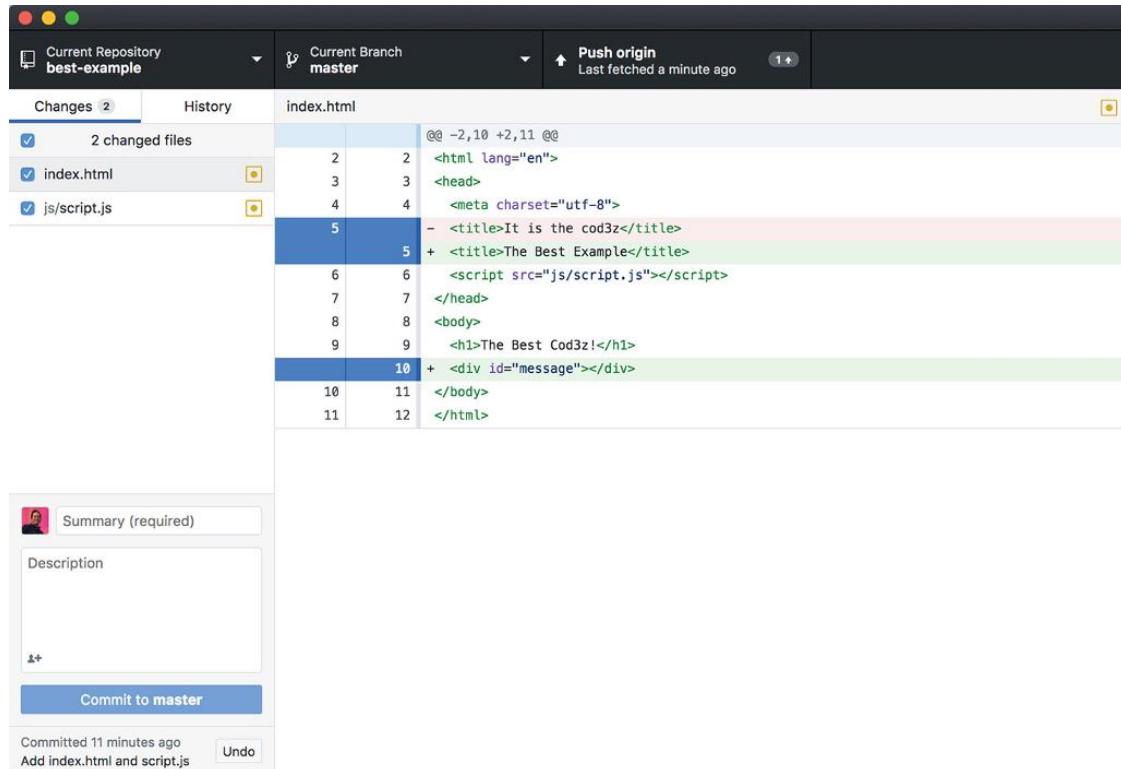


图 7-5：显示未提交更改的更改视图

左窗格列出了一组已更改的文件。如果选择一个文件，在右侧窗格中看到对该文件所做的特定更改，这称为“差异视图”。

将所有更改提交为一次 commit，但有时会进行不相关的更改。在这个例子中，我们有两个不相关的变化：（1）index.html 中标题的变化；（2）对于 index.html 和 script.js 的内容都有修改。

这些更改中的每一个都应该对应一个 commit。当 index.html 包含两个不相关的更改时，我们该怎么做？幸运的是，Desktop 提供了一种很好的方式来提交文件中的部分更改。此过程称为部分提交（partial commit）。

(1) 取消选择所有更改。

在左窗格中，取消选中标签“2 changed files”旁边的复选框以取消选择所有更改。

(2) 在左窗格中选择 index.html。

单击左窗格中的文件名以显示 index.html 的更改。

(3) 选择标题更改。

在差异视图中，单击左侧中的行号以选择要保留的更改。要选择整个代码块，请单击行号右侧的细线。单击第 5 行旁边的细线，选择第 5 行旁边的代码块。选择代码块后，标记为第 5 行的两行都应该被选中（选中的行显示为蓝色），如图 7-6 所示。

读者可能会对差异视图中有两行都标记为 5 感到困惑。左侧的数字 5 表示在进行更改之前文件的最初名称。右边行的数字 5 代表已更改的行号。因为我们改变了第 5 行，所以它被列出了两次。第 10 行是之前不存在的新行，因此仅在右侧列出。

(4) 选择这些行后，输入 commit 消息，然后单击提交到主分支按钮。

正如在图 7-6 的左下部分所看到的，Desktop 为 commit 消息提供了两个字段。继续并在摘要中输入“更改标题”，然后单击“commit to master”按钮。

请注意，差异视图更新为仅在第 10 行进行更改（见图 7-7）。那是因为我们已经提交了第 5 行的更改。

如果按照本章中的示例进行操作，请通过单击标签“2 changed files”旁边的复选框，直到选中该复选框，以确保选中所有剩余的更改。

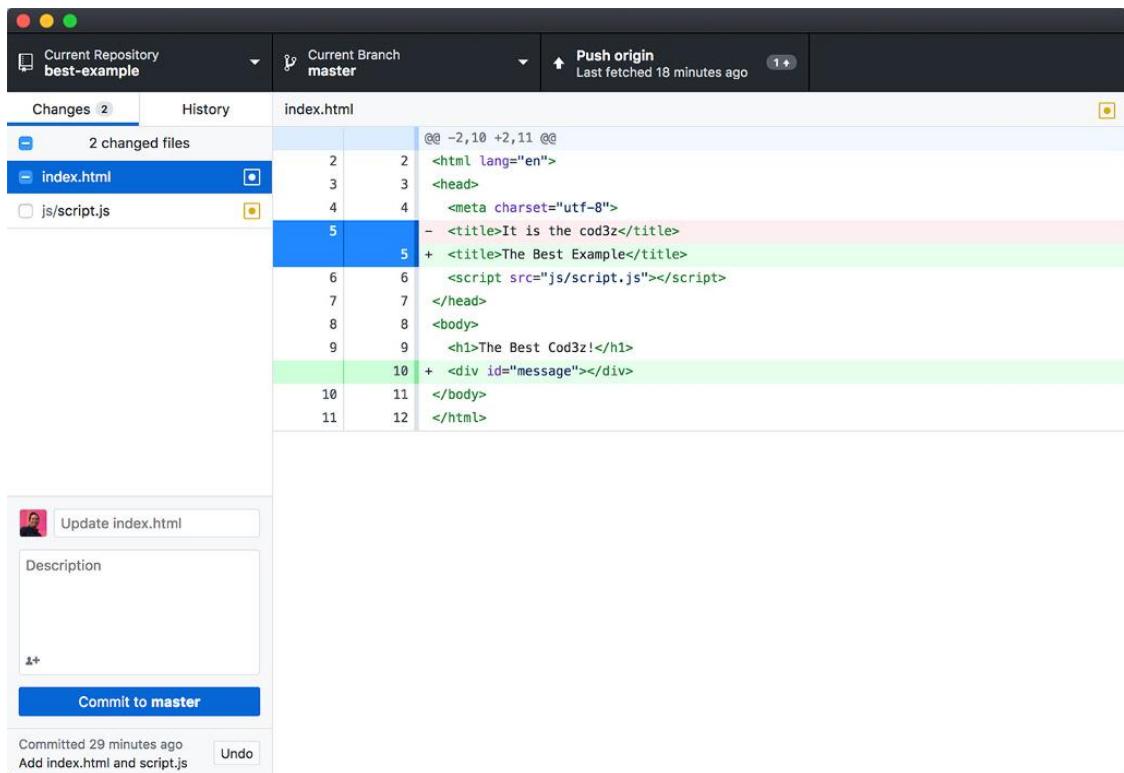


图 7-6：选择了一项更改的差异视图

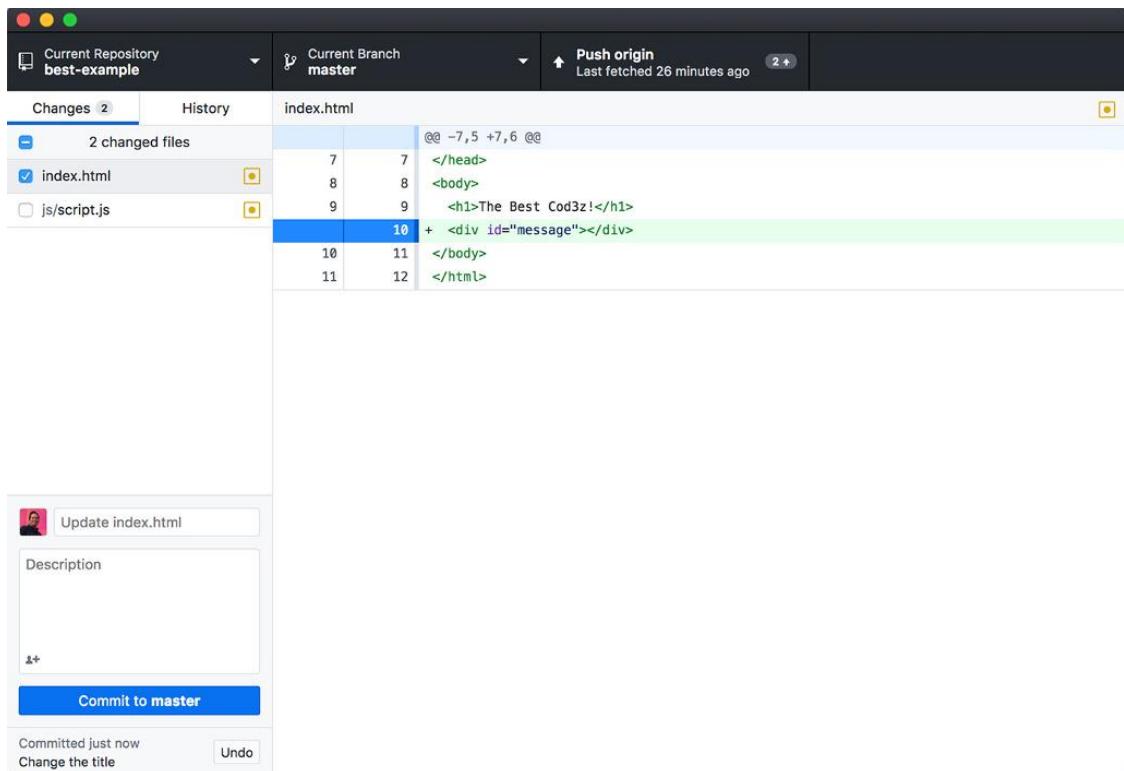


图 7-7：部分提交后的更改选项卡

7.6 使用 GitHub 约定 commit 消息

使用特定于 GitHub 的功能来增强 commit 消息，例如表情符号、issue 参考和协作者信息。

7.6.1 表情符号

表情符号是传达情感或概念的小图像或图标。在 GitHub.com 上广泛使用的表情符号可以为原本严肃的职业带来一些轻松和奇思妙想。

在输入 commit 消息的输入框中，通过键入 “:character” 来启动表情符号选择器。如果继续输入，列出所有以输入的字母开头的表情符号。例如，图 7-8 列出了所有以 ar 开头的表情符号作为结果。

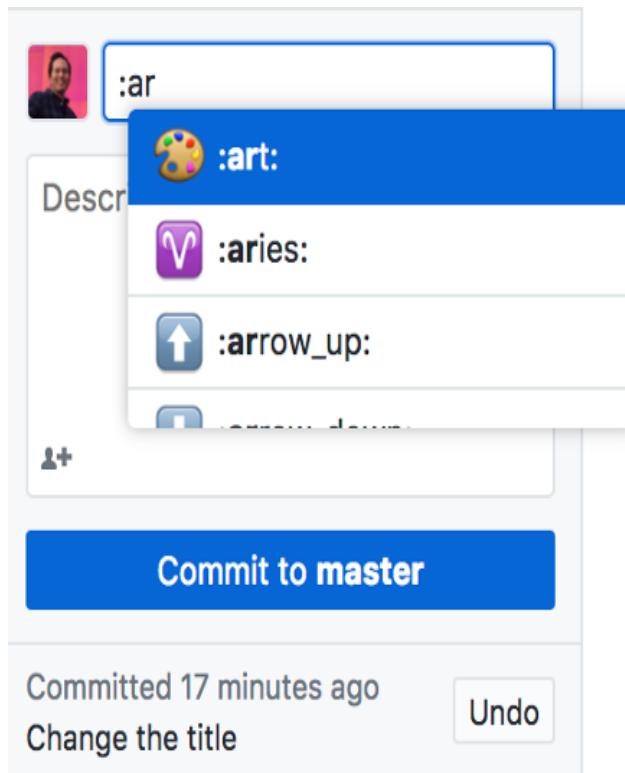


图 7-8：列出表情符号的表情符号选择器

使用箭头键选择想要的一个，然后按 Tab 键入表情。Desktop 会填写表情符号的全文，在这种情况下是 “:art:”。

7.6.2 Issue 参考

GitHub 还允许在 commit 消息中以 #123 格式引用 issue，其中 123 是 issue 的编号。Desktop 支持在编写 commit 消息时查找问题。要尝试此操作，请提前创建一个 issue，以便在 commit 消息中引用该 issue。例如，我们创建了一个 issue 来描述用更好的方法替换警告消息的需要。我们在此 commit 消息中引用了该 issue。

在 commit 消息中引用 issue：

(1) 在提交描述字段中，输入 Fixes #。

此时会出现一系列 issue。如果没有看到要参考的 issue 或不记得 issue 编号，开始输入 issue 中的单词。例如，当我们输入 #alert 时，会弹出几个 issue（见图 7-9）。

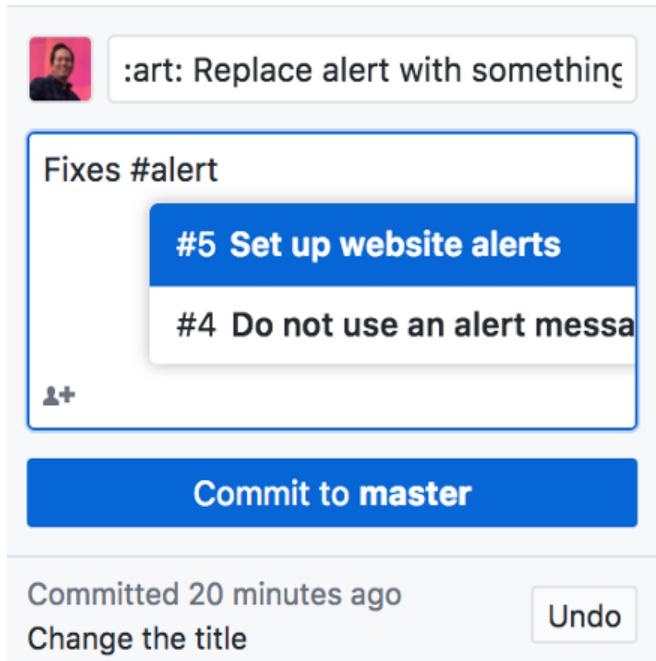


图 7-9：其中包含“alert”一词的 issue 列表

(2) 选择要参考的问题然后按 Tab。

在此示例中，我们选择 issue 4，桌面将 #alert 替换为 #4。

7.6.3 感谢协作者

Git 不支持多个作者。但是，Git 社区创建了一个约定，用于在 GitHub 支持的 commit 中指定多个协作者。

(1) 打开桌面后，在带有说明标签的提交框中，单击左下角带有人和加号的小图标。Desktop 添加了一个文本框来输入协作者的 GitHub 用户名。

(2) 单击@符号查看用户列表，如图 7-10 所示，GitHub 仅列出有权访问仓库的用户，例如，协作者和组织成员（如果仓库属于组织）。

提示：就像 issue 选择器一样，也可以通过在“@”后面附加一点来按名字、姓氏或用户名进行搜索。例如，如果我只记得 y 协作者的姓氏，我可以输入“@guth”来查找我的协作者。按 Tab 键，Desktop 会用所选用户的完整用户名替换目前输入的任何内容。

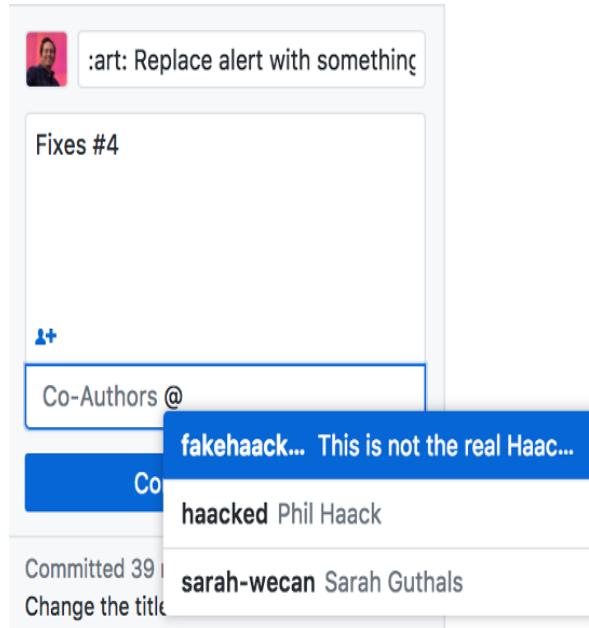


图 7-10：协作者列表

(3) 要创建 commit，请单击 Commit to master 按钮。要查看提交，请单击 History 选项卡并单击刚刚创建的条目（见图 7-11）。

index.html	7	7	@@ -7,5 +7,6 @@	
	8	8	</head>	
	9	9	<body>	
	10	10	<h1>The Best Cod3z!</h1>	
	11	11	+ <div id="message"></div>	
	12	12	</body>	
			</html>	

图 7-11：新创建的 Commit

在右侧窗格的 commit 消息中看到 Sarah 的用户名已被以下行替换

`Co-Authored-By: Sarah Guthals <sarah-wecan@users.noreply.github.com>`

这是在 Git commit 消息中指定协作者的实际约定。通过使用 Desktop，不必记住确切的格式。可以只指定一个用户名，让桌面处理其余的工作。

7.7.从编辑器提交代码

许多编辑器都内置了对提交代码的支持。这种内置支持允许用户无须切换到另一个应用程序即可提交代码。缺点是不同的编辑器对可在 commit 消息中使用的各种约定有不同等级的支持。

但是对于普通的 commit，内置支持非常有用。介绍每个编辑器如何支持 Git 提交超出了本书的范围，但在第 5 章中看到 Atom 的实际操作。对于其他编辑器，请参阅特定文档。

更多阅读

有很多很好的指导可以写出好的 commit。例如，Git 文档（<https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project>）有一个关于为项目做出贡献的部分，包括一些提交指南。

第 8 章 使用 Pull Request

本章学习重点：

- (1) 创建一个 PR
- (2) 编写一个很棒的 PR
- (3) 审查 PR
- (4) 探索 PR 工作流程

第 7 章指出 commit 是 Git 的最小工作单元。在 GitHub 上，pull request（简称 PR）是主要的工作单元。

本章将解释 PR 究竟是什么以及如何通过 PR 将代码推送到 GitHub，并描述打开、撰写和审查 PR 的过程。

8.1 理解 pull request

pull request 这个名称让一些人感到困惑。“我到底要请求拉取什么？”好问题。pull request 其实是请求仓库的维护者拉取一些（你的）代码。

当希望向某个仓库贡献一些代码时，可以创建并提交一个 PR。代码中包含着对目标仓库的更改，而 PR 正是向仓库维护者提供这些更改的方式。通过审查 PR 中的代码，仓库维护者可以接受或拒绝这些更改，或要求进一步的更改。

8.2 将代码推送到 GitHub

要将代码推送到 GitHub，你需要一个仓库。第 7 章介绍了如何创建一个仓库。

提示：如果想按照我们的示例进行操作，但尚未完成第 7 章中的步骤，请 fork 仓库 <https://github.com/FakeHaacked/best-example>，然后将 fork 的仓库克隆到计算机中。如果对上述操作不熟悉，那可能需要复习第 6 章，这章涵盖了 fork 和克隆相关的内容。

首先要做的是创建一个新分支。在编写新代码之前创建一个新分支是 Git 的标准操作。第 7 章的示例为了简单起见，刻意忽略了创建新分支这一步，而直接将代码提交到了 master 分支。

在本章中，你需要先创建一个分支并在这个新分支上开展工作，这才是正确的方式。为什么要这么做？一个重要的原因是 PR 本身不包含任何更改或 commit，它只是与一个分支相关联。换句话说，PR 是希望将一个分支合并到另一个分支的请求。

提示：虽然 PR 可以以任何分支为目标（除了它自身），但最常见的情况是向仓库的主分支发起 PR，该主分支通常被命名为 master。

PR 和分支之间的这种关系就是为什么应该在开始新工作前创建新分支的原因。我们将本示例的分支命名为 new-work，但它可以是任何名称：

```
$ git checkout -b new-work
```

得到一个新分支后，我们继续在该分支上创建一个 commit。commit 的具体内容不重要，你可以选择任何文件做一些修改，例如在末尾添加一些文本。如果你沿用第 7 章创建的仓库，请对 README.md 文件做些修改或运行以下命令在文件末尾添加一些文本：

```
$ echo "\n## Installation" >> README.md
```

运行下面的命令来提交所有更改。commit 消息在这里并不重要，重要的是在一个非 master 分支上增加一个 commit。

```
$ git add -A  
$ git commit -m "Add text to the README"
```

现在将这个新分支推送到 GitHub.com（注意用你的分支名称替换掉 new-work）：

```
$ git push -u origin new-work
```

git push 会将本地的 commit 推送到远程仓库。-u 标志指定推送到何处——上面的命令会将 commit 推送到名为 origin 的远程仓库的一个同名分支。-u 标志仅在第一次将新分支推送到服务器时才需要，它会将本地计算机上的 new-work 分支与 GitHub.com 上的 new-work 分支关联在一起。对该分支的任何后续推送都不需要该标志。

8.3 创建一个 PR

能创建 PR 的前提是仓库中至少有一个非默认分支。上一小节的操作使我们得到了一个尚未合并到 master 的分支 new-work。接下来访问 GitHub.com 并创建一个 PR。

当访问 GitHub.com 上的仓库时，如果你有最近推送的新分支，GitHub 会进行提示，如图 8-1 所示。

点击“Compare & pull request”按钮跳转到“Open a pull request”页面，如图 8-2 所示。目标分支是仓库的默认分支，你的分支被列在目标分支旁边，GitHub 会提示该分支是否可以合并到目标分支。最近的提交消息会作为 PR 的默认标题，描述部分留空。图 8-2 显示了示例说明。

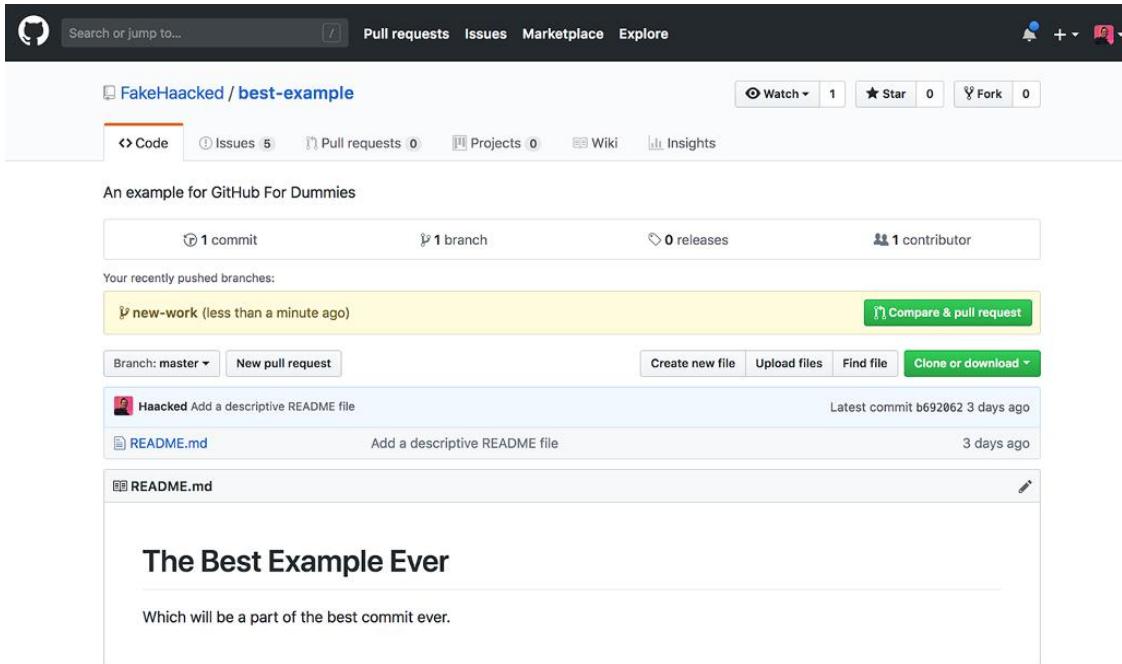


图 8-1：列出了最近推送的分支的仓库主页

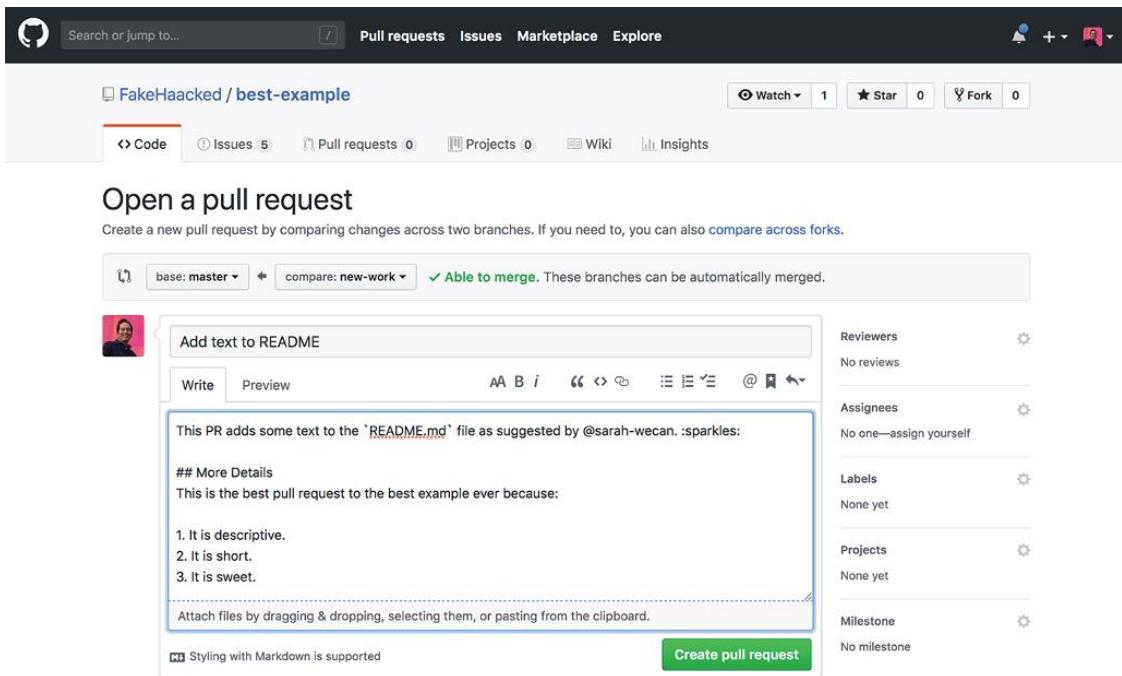


图 8-2：创建 PR 页面

提示：在仓库的 Settings⇒Branches 中可以更改默认分支。

(1) 使用 GIT 别名在终端打开 GITHUB.COM

在终端中访问本地仓库时，我们经常需要访问其在 GitHub.com 上对应的远程仓库。可以使用 Git 别名来创建一个快捷指令。第 6 章更详细地介绍了别名以及如何添加它们。在终端中运行以下命令来添加新的 git browse 别名。

```
$ git config --global alias/browse '!open `git config remote.origin.url`.'
```

要使用该别名，只需在仓库目录下运行以下命令。

```
$ git browse
```

该命令会在你的默认浏览器中打开仓库的原始 URL。这个别名假设仓库的 remote 设置使用 HTTPS 而不是 SSH。

(2) 使用 Git 别名打开分支比较页面

如果由于某种原因，你要为其创建 PR 的分支未在最近推送的分支中列出，你可以直接前往分支比较页面，其格式为

<https://github.com/{owner}/{repo}/compare/{BRANCH}>。

创建一个别名会更加便捷：

```
$ git config --global alias.pr '!f(){ URL=$(git config remote.origin.url); open ${URL%.git}/compare/$(git rev-parse --abbrev-ref HEAD); }; f'  
$ git pr
```

同样地，此别名假设仓库的 remote 设置使用 HTTPS，而不是 SSH。

8.3.1 描述 PR

在 PR 创建页面可以输入概要和描述。第 7 章介绍了一些撰写 commit 消息的 GitHub 约定。PR 中也支持大多数这些约定——例如，使用 @USERNAME 格式来提及别人。在图 8-2 中，我使用了 @sarah-wecan。当我创建 PR 时，@sarah-wecan 会收到通知。

使用 #ISSUEID 格式引用 Issue 和其他 PR。还可以使用 emoji，例如 :sparkles:。

在本节中，我们将更详细地介绍如何描述一个 PR。一个好的 PR 应包含很多内容。

8.3.2 添加审查者

PR 概要和描述的右侧是 PR 的一组选项。审查者选项允许你指定一个或多个要审查 PR 的人。

(1) 单击齿轮以查看你可以提及的人员列表。

对于拥有大量用户的仓库，键入用户名进行过滤。

(2) 点击每个用户将他们添加到审查者列表中。

当你添加审查者时，他们会在 PR 创建后立即收到通知。图 8-3 显示了包含两个审查者的审查者列表。

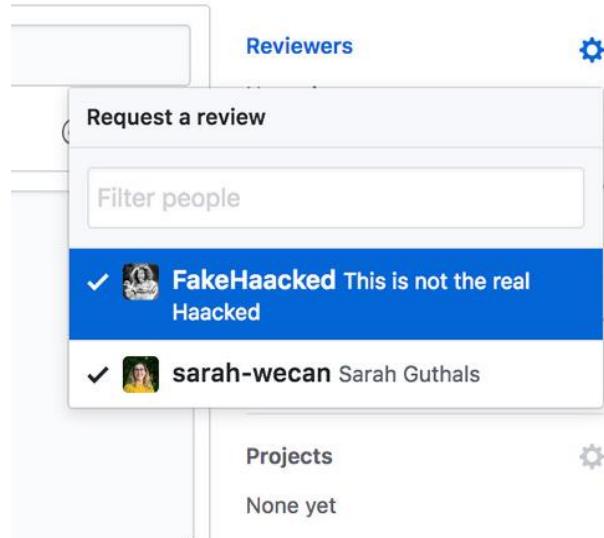


图 8-3：包含两个审查者的审查者列表

8.3.3 指定受理人

在审查者选项后是受理人选项。受理人负责执行该 PR 的工作。通常，PR 意味着工作正在进行中，而不是最终结果。如果一个 PR 还需要持续的投入，可以将 PR 分配给应该做这项工作的人。

- (1) 点击齿轮查看受理人列表，该列表与审查者列表类似。
- (2) 点击用户将他们添加到受理人列表中。

提示：在大多数情况下，最好只指定一个受理人，以避免受理人有多个却无人真正负责的情况。

8.3.4 指定标签

第 3 章介绍了为 Issue 打标签可以帮助明确下一步的工作。标签在 PR 中发挥同样的作用，我们可以利用标签对 PR 进行分组以及获取上下文信息，从而决定接下来要处理哪个 PR。

Issue 和 PR 共用一套标签，但一些标签只适用于二者之一。例如，许多仓库都有一个“ready for review”标签，这是 PR 专用的。

8.3.5 指定项目和里程碑

最后两个选项允许你指定此 PR 所属的项目和里程碑。项目和里程碑分别在第 3 和 11 章中介绍。

8.3.6 创建 PR

在完成 PR 描述的撰写并指定了所有 PR 选项后，单击“create pull request”按钮完成 PR 的创建。图 8-4 显示了我在自己仓库中创建的一个 PR。

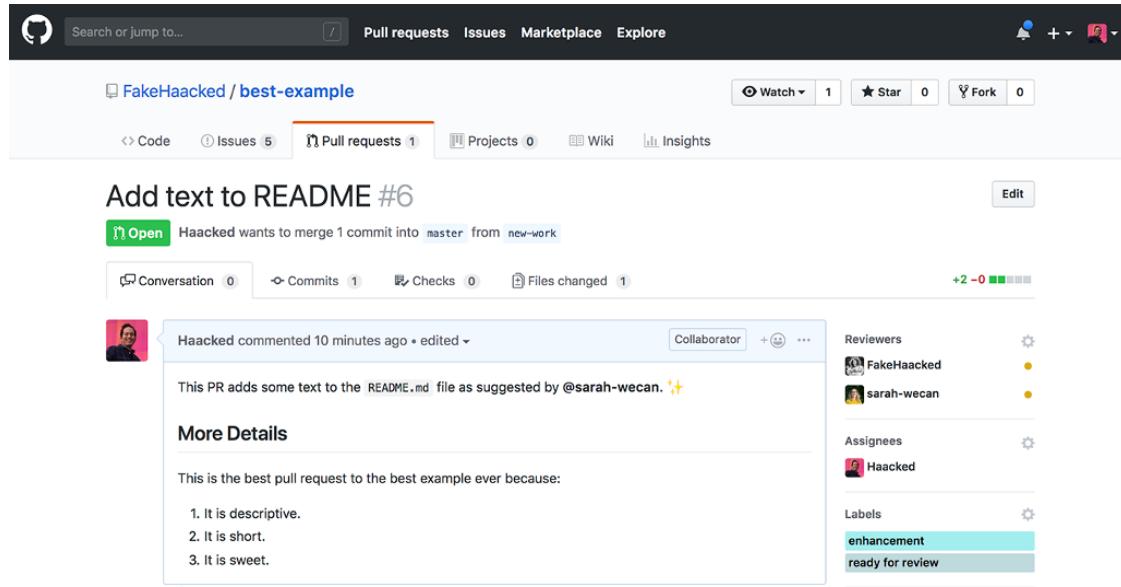


图 8-4：一个创建成功的 PR

8.4 撰写一个很棒的 PR

撰写一个很棒的 PR 是一门艺术。在开源项目中，交流大多发生在 PR 中。若希望贡献到一个项目中，PR 便是说服维护者将你的代码合并到主分支的绝佳机会。请务必展现出最好的一面。

8.4.1 了解受众

在落键前先想想 PR 的受众是谁，这能使描述的内容更为精准有效。一个 PR 的受众有许多，他们都非常重要，但你的主要关注对象是那些会审查代码并决定是否合并 PR 的人。这些人往往很忙，请尽量使他们能够轻松点。

即使项目维护者是主要受众，也不应忽略 PR 的其他读者。对于一个开源项目来说，这些读者可能来自全球各地，因此，请保持尊重、友好和包容。

警告：在愤怒中撰写 PR，事后却为措辞后悔不已，这种情况并不少见。

因此，若你写代码写得怒火中烧，请在创建 PR 之前花点时间冷静下来并整理你的想法。

8.4.2 明确目的

确保简明扼要，言之有物。例如，概要应明确 PR 的目的，因为它是唯一会显示在 PR 列表页面的部分。确保它能使人一目了然。

下面是一些优秀的概要：

- (1) Adds the About page to the website;
- (2) Minimize boilerplate setup code for JavaScript libraries;
- (3) Extract and isolate error handling from GitStore internals。

也有一些来自我个人仓库的反面教材。

- (1) Teams are forever;
- (2) Typo;
- (3) Small changes。

提示：在描述部分应该提供更多关于该 PR 的目的的解释。不需要长篇大论，但要阐明 PR 的意图。

8.4.3 保持专注

就像 commit 一样，一个 PR 中不应该包含太多不相关的更改。一个 PR 可以有许多 commit，但它们都应与手头的任务相关。

当发现撰写一个简明扼要的 PR 描述很困难的时候，通常是这个 PR 做得太多了。

即使 PR 专注于单个重大的变化，也要控制 PR 的体积，一个过大体积的 PR 不利于审查。

对于一个大任务，请将它分解为若干个小任务，并为每个任务单独提交 PR。

8.4.4 解释原因

除了解释 PR 做了什么，还应该解释为什么需要做这项工作。在 PR 描述部分附上指向其他文档的链接可以提供更多的上下文，毕竟你不能指望每个人都了解事情的来龙去脉。

如果有许多上下文需要提供，可以先写一句概要，再利用 `<details>` 标签给出更多细节。然后提供更多详细信息。例如，若你将下面的文本作为 PR 评论：

```
The reason we're embarking on this work is due to compliance reasons.  
<details>  
## More Details  
I don't want to bore everyone with all the nitty gritty details, but fo  
r those  
    who are interested, keep on reading...  
</details>
```

GitHub 会默认折叠 `<details>` 标签中的内容，如图 8-5 所示。

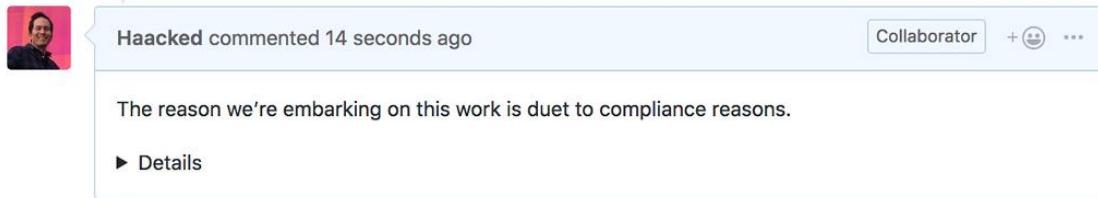


图 8-5：详细信息部分已折叠

单击三角形按钮以展开被折叠的内容，如图 8-6 所示。



图 8-6：详细信息部分已展开

8.4.5 一图胜千言

GitHub 允许在 PR 的描述部分使用图片，只要将图片拖拽到文本框中，GitHub 会将其自动上传并在文本框中填入对应的 markdown 代码。图 8-7 显示将图片拖入文本框后的上传状态。



图 8-7：将图像上传到 PR 中

访问 <https://github.com/FakeHaacked/best-example/pull/6#issuecomment-454927796c> 查看该图片评论。

提示：如果说一图胜千言，那么活动的图片价值更高。一个能够展现 PR 改动的 gif 动图会赢得审查者的赞赏。

8.4.6 明确反馈要求

告诉他人你希望的反馈。例如，若一个 PR 处于 WIP (working in progress) 状态，请一开始就明确这一点，这样人们就不会浪费时间审查尚未准备好的 PR。

每个仓库对描述这类情况都有一套自己的约定，通过主动探索仓库可以使你熟悉这些约定，就如第 5 章描述的那样。

遵循约定很重要，这样其他人才会对你的 PR 有正确的反应。

PR 的撰写还能更加全面，查看博客“How to write the perfect pull request”获取更多建议。<https://github.blog/2015-01-21-how-to-write-the-perfect-pull-request/>

8.5 审查 PR

一个 PR 由两方参与：提交方和审查方。这一小节，你将从审查者的视角来认识 PR。

审查 PR 是一项耗时耗力的工作。若只是粗粗一看，留下一条内容为 LGTM (Look Good To Me) 的评论而没提供什么实质反馈，这对 PR 提交者来说意义不大。

有时候如果实在腾不出时间，请说明 PR 中的哪些更改已被审查而哪些还没有被审查，并推荐另一位审查者来进行更加细致的审查。

当被添加为审查者后，你会通过邮件或 GitHub 网站右上角的通知铃铛接收到相关消息。此时访问 PR 页面，你将看到一条横幅消息，如图 8-8 所示。

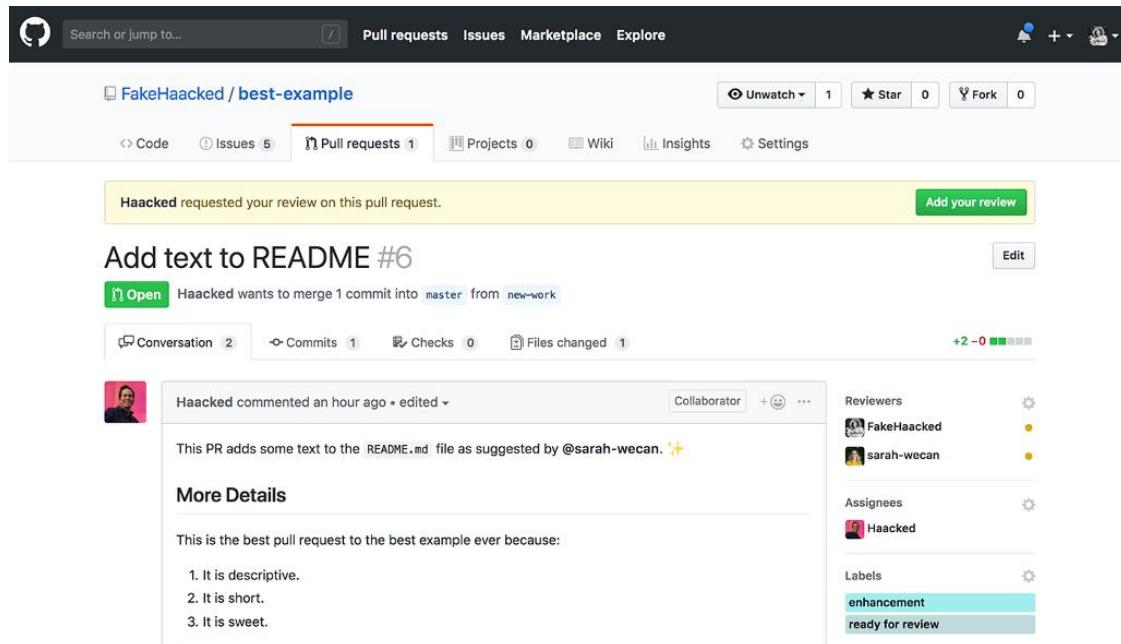


图 8-8：请求你对此 PR 进行审查的消息

警告：审查完毕后再点击“Add your review button”，因为该按钮会使页面跳转到一个对话框，你将在那里填入审查意见并完成这次审查。

8.5.1 审查“对话”选项卡

当审查一个 PR 时，先通读 PR 的“对话”选项卡中的内容，确保你了解 PR 的目的及其必要性。

GitHub 会运行仓库中的检查（如果有的话），可以在对话底部找到它们。如果未设置检查，你会看到消息“Continuous integration has not been set up”。这些检查通过与否是你的下一个关注点。许多仓库中都会设置一些检查，比如“代码是否成功编译”、“所有测试用例是否通过”等。如果这些测试中的任何一个失败，就不应该再花时间审查代码。PR 提交者可以看到自己的代码没有通过检查，错误应该由他们来修复。

提示：检查的运行需要些时间，如果 PR 提交者在检查运行完毕前就离开，没有发现检查未通过，对于这样的情况，尝试对提交者进行友好的提示，令其修复问题并将更改再次推送到 PR 分支。

图 8-9 显示了来自 GitHub Desktop 开源项目的 PR 示例，该请求未能通过持续集成 (CI) 编译的检查。

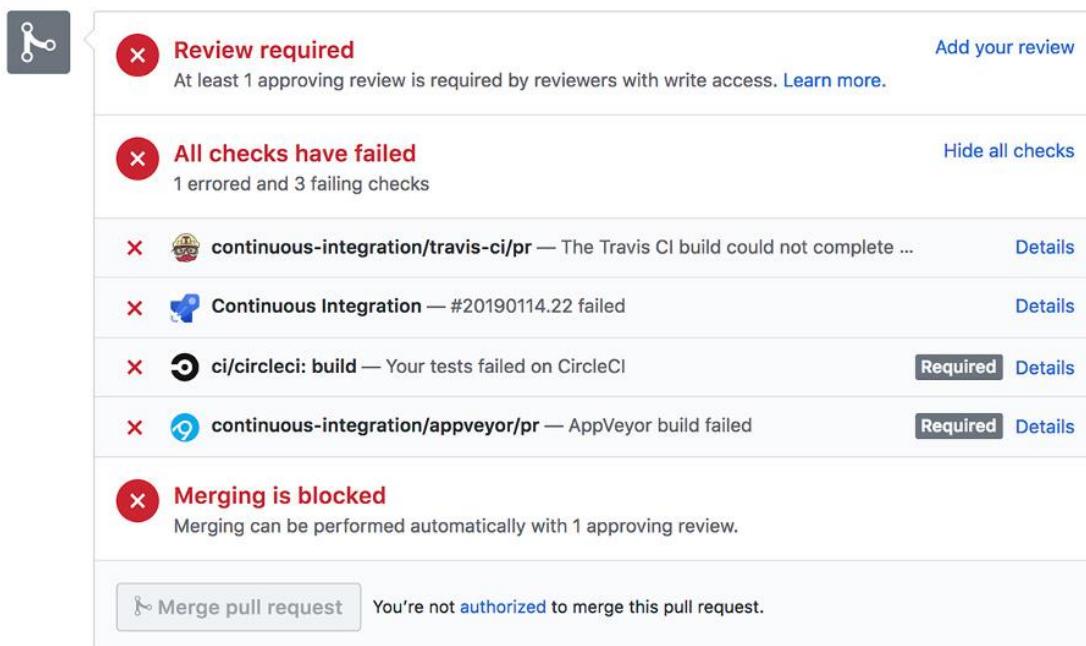


图 8-9：未通过检查的 PR

8.5.2 审查更改的文件

假设你对“对话”选项卡的内容感到满意并且所有检查已经通过，那么是时候深入细节去审查文件更改了。单击“文件更改”选项卡以查看 PR 中的所有更改。

以下是代码审查评判依据。

(1) 可读性: 代码是否易于阅读和理解? 还能更清楚吗? 代码中有适当的注释吗? 晦涩难懂的代码将成为未来维护的噩梦。

(2) 正确性: 代码功能是否如 PR 描述的那样? 是否有明显的错误? 是否有遗漏? 是否缺少测试用例?

(3) 安全性: 安全性审查也需要特定的意识。理想情况下, 你会和审查代码安全性的安全专家一起工作。这个想法的初衷是以恶意攻击者的角度找出所有潜在的对代码的攻击方式。有很多框架可以用于安全审查, 例如 STRIDE。你还应该考虑恶意攻击者如何利用代码来伤害其他用户。例如代码是否能保护用户的隐私? 它是否征求用户的同意从而代表用户产生行为。

(4) 约定和习惯: 正确的代码不一定是符合惯例的代码。代码审查是教和学特定仓库约定和习惯的好机会。

何谓约定? 约定是指完成某任务的常用方式。例如, 若项目中有特定的数据库查询方法, 那么 PR 中的代码也要遵循该方法。

警告: 值得注意的是, 我们没有把代码风格放到上面的评判依据中。代码审查不应该拘泥于如花括号是否应该独立一行之类的问题。过于迂腐挑剔的代码审查并不能营造友好的协作基调。根据上下文提供的信息, 你可以自己解决这些问题, 或者最好借助自动化工具来解决这类问题, 例如 linter 或 prettier。这将节约大家的时间。

8.5.3 评论代码

在审查文件更改时, 你可以向其中的代码行添加评论以指出问题, 给出建议, 或者用 :sparkle: 表情来庆祝优秀的编程技巧。

提示: 正面和鼓励的评论可以营造包容友好的协作氛围。维护者经常忘记第一次为项目贡献代码是多么令人生畏。不要吝啬 :sparkle: 表情!

以下操作可以实现在代码上评论。

(1) 将鼠标悬停在代码行上。

行号旁边会出现一个带加号的蓝色方块。

(2) 单击方块以显示该特定代码行的评论框, 如图 8-10 所示。

该评论框和 Issue、PR 的评论框一样, 引用, markdown, emoji, 这些功能一样都不少。

此时, 你可以点击 “Add single comment” 或 “Start a review”。前者会将评论立即添加到 PR 中并发送通知。这种方式适合一次性评论, 但大多数情况不被推荐, 因为这不利于一次深思熟虑的代码审查。

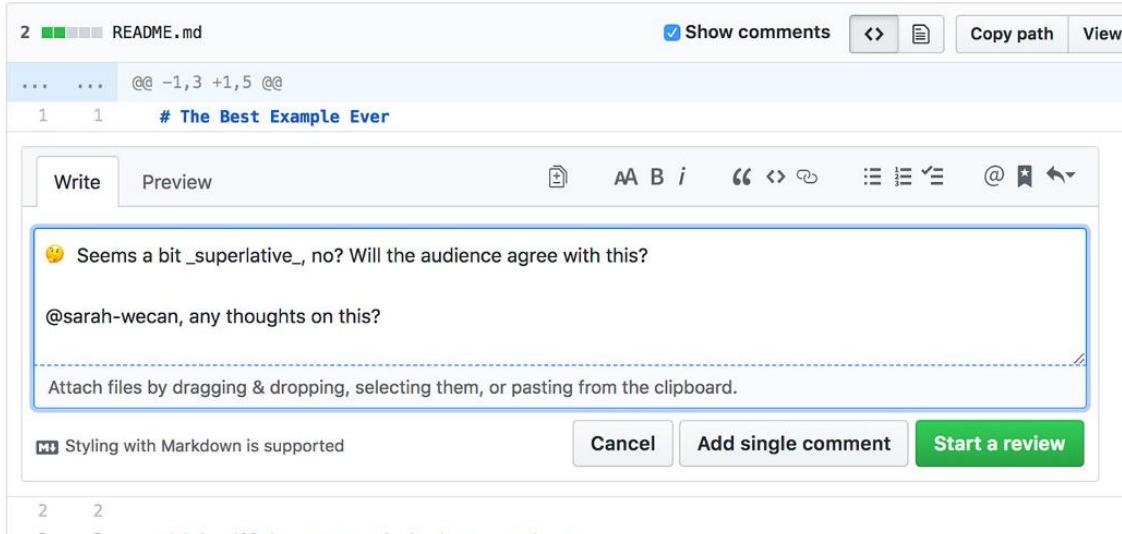


图 8-10：评论一行代码

(3) 点击“Start a review”开始一系列待定评论的创建，如图 8-11 所示。
待定标签表示你是目前唯一可以看到这个评论的人。这种方式允许在审查代码时继续添加（或编辑）待定评论，直到满意才将评论发布。

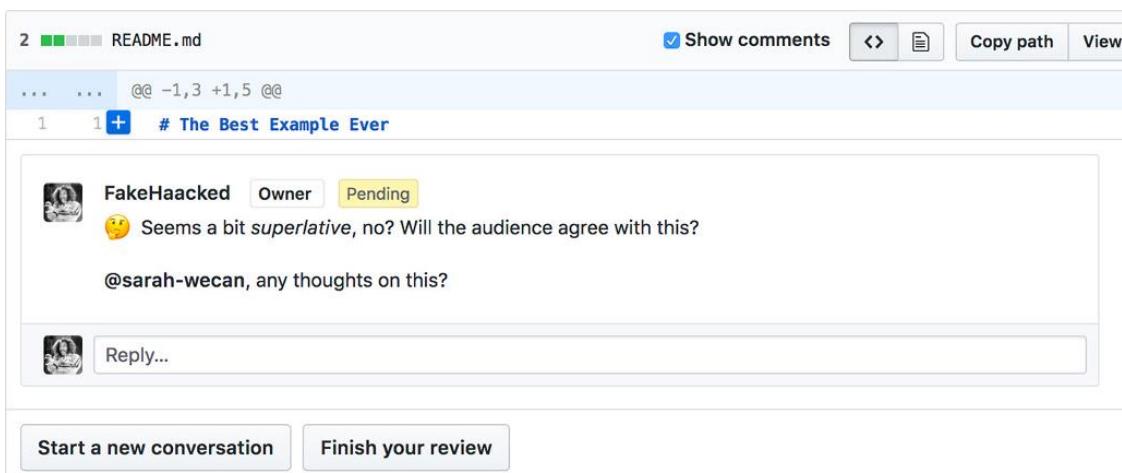


图 8-11：PR 审查中的待定评论

提示：“Start a review”更被提倡的原因是它使代码审查过程连贯且一致。读了后面的代码而想修改或删除前面的评论在代码审查中是常有的事，“Start a review”方式可以确保在发布审查评论前评论内容能被调整为最佳状态。

8.5.4 更改建议

有时，直接给出具体的更改比用文字解释省力得多，在一些诸如拼写错误的小问题上多费口舌甚至会增加噪音。对于这些情况，GitHub 支持通过 PR 评论来直接给出更改建议，并且 PR 的提交者可以点击按钮直接将更改建议应用到代码中。

以下操作可以给出更改建议。

(1) 在要更改的代码上打开评论框。

评论框上有一个带有 + 和 - 符号的图标。将鼠标悬浮在该图标上可以看到提示（见图 8-12），例如该功能的键盘快捷键是 CMD+G。

(2) 点击“Insert a suggestion”按钮。

一个建议块会插入到评论中，其中包含你要更改的代码行的当前内容。

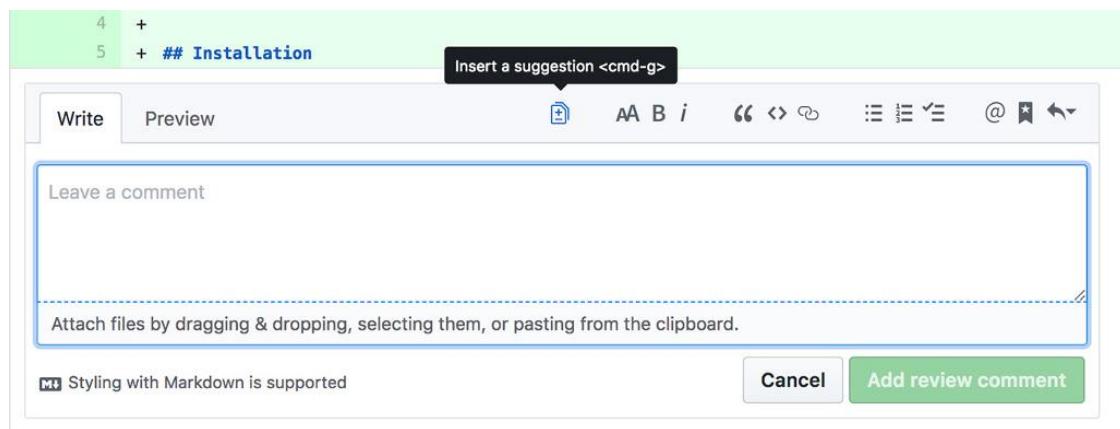


图 8-12：带有更改建议按钮的评论框

(3) 修改建议块中的内容，它会成为最终的更改建议。

图 8-13 的示例中，我建议在该行中添加“Instructions”一词。

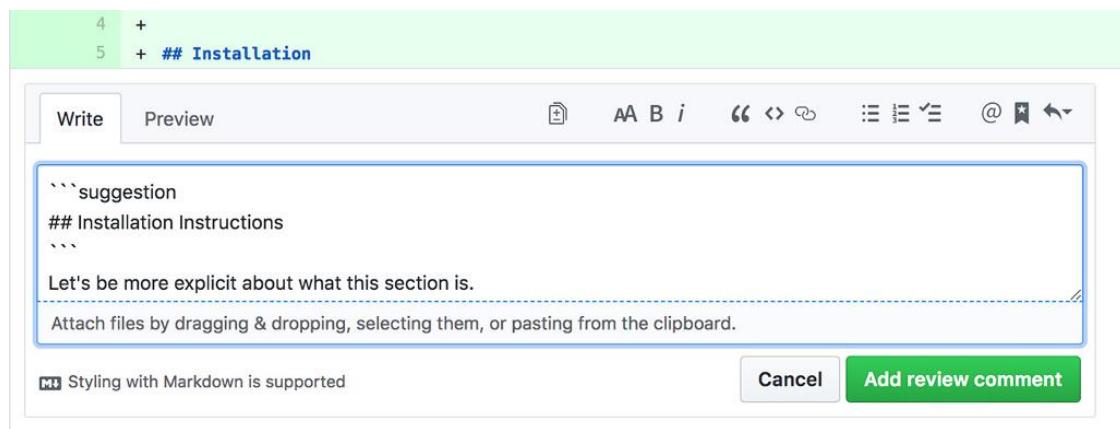


图 8-13：带有更改建议的评论

提示：任何建议块之外的内容不会包含在更改建议中，因此可以用它们来解释更改建议。除了非常简单或不言自明的更改建议外，尽量给出额外的解释。

创建带有更改建议的评论后，GitHub 会将更改建议渲染为一个内嵌的 diff 视图。如果查看建议的人具有 commit 权限，将看到提交更改建议的选项（见图 8-14）。他们也会感谢你为他们节省时间！

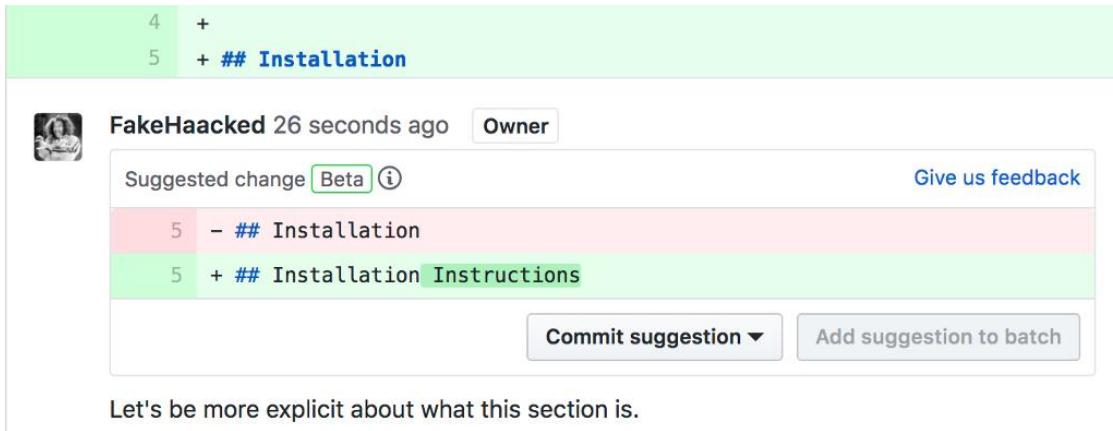


图 8-14：带有更改建议的评论

8.5.5 完成审查

在提出所有建议后，完成评论以使 PR 提交者接收到所有反馈并开始着手解决。在任何待定评论框中点击“finish your review”按钮完成评论（参见图 8-11）。

你会看到一个表单，可以在其中撰写有关 PR 的总体概要。该表单不特定于任何代码行，可以利用它表达赞誉，呼吁关注，倡议后续行动等。

键入评论后，选择其中一个审查选项。图 8-15 显示了带有审查选项的评论表单。

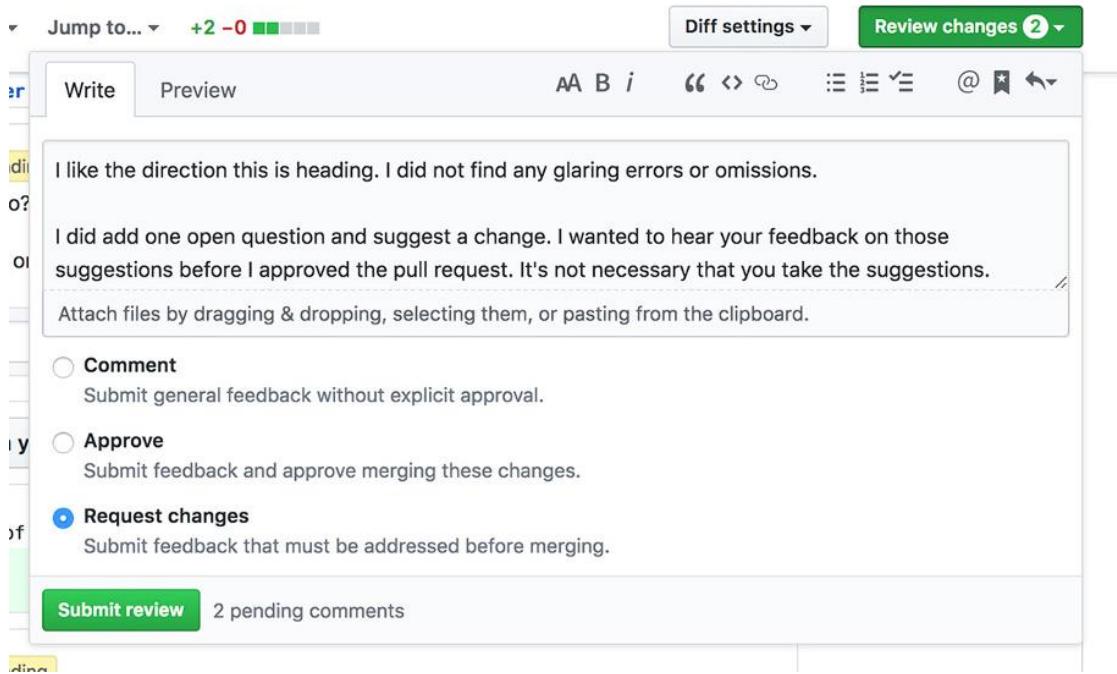


图 8-15：包含更改建议的评论

提示：如果你正在审查自己的 PR，唯一可用的选项是对 PR 发表评论。你的选择可能决定此 PR 是否能被合并到默认分支中，这视仓库的分支保护规则而定。例如，某些仓库可能需要一定数量的批准才能合并 PR 请求。在“设置分支”中管理分支保护规则。

8.6 阅读更多有关 PR 的信息

PR 和代码审查是软件开发生命周期中非常重要的部分，因此在这方面有非常多优秀建议，它们远不是一个章节的内容可以完全覆盖的。以下是一些可供阅读的文章，它们能帮助你充分利用 PR 和代码审查。

(1) 营造包容性的代码审查文化：代码审查的过程是项目文化的体现。<https://blog.plaid.com/building-an-inclusive-code-review-culture/>描述了一种具有包容性和协作性的代码审查方法。

(2) 代码审查，尽你所能：<https://haacked.com/archive/2013/10/28/code-review-like-you-mean-it.aspx/>这篇文章对代码审查的作用以及帮助做好这项工作的技巧做了简单介绍。