

Pro Git

Scott Chacon, Ben Straub

目录

1. 账户的创建和配置	2
1.1. SSH 访问	3
1.2. 头像	3
1.3. 邮件地址	4
1.4. 两步验证	5
2. 对项目做出贡献	5
2.1. 派生 (Fork) 项目	6
2.2. GitHub 流程	6
2.3. 合并请求的进阶用法	11
2.4. Markdown	15
3. 维护项目	20
3.1. 维护项目	20
3.2. 创建新的版本库	20
3.3. 添加合作者	22
3.4. 管理合并请求	22
3.5. 提醒和通知	26
3.6. 通知页面	27
3.7. 网页通知	28
3.8. 邮件通知	29
3.9. 特殊文件	29
3.10. README	29
3.11. 贡献 CONTRIBUTING	30
3.12. 项目管理	30
4. 管理组织	31
4.1. 管理组织	31
4.2. 组织的基本知识	32

4.3. 团队	32
4.4. 审计日志	33
5. 脚本 GitHub	33
5.1. 钩子	34
5.2. GitHub API	38
5.3. 基本用途	38
5.4. 在一个问题上评论	39
5.5. 修改 Pull Request 的状态	41
5.6. Octokit	42
6. 总结	43

插图

1. GitHub 注册表单	2
2. “Account settings” 链接	3
3. “SSH keys” 链接	3
4. “Profile” 链接	4
5. 裁剪已上传的头像	4
6. 添加所有邮件地址	5
7. Security 标签页中的 2FA	5
8. “Fork” 按钮	6
9. 他想要做出贡献的项目	7
10. 合并请求按钮	9
11. 合并请求创建页面	10
12. 对合并请求内的特定一行发表评论	11
13. 通过电子邮件发送的评论提醒	11
14. 合并请求讨论页面	12
15. 最终的合并请求	13
16. 不能进行干净合并	14
17. 合并请求现在可以干净地合并了	14
18. 在合并请求中的交叉引用	15
19. 在合并请求中渲染后的交叉引用	16
20. 在合并请求中渲染后的交叉引用	16
21. 一个 Markdown 的例子和渲染效果	17
22. Markdown 评论中渲染后的任务列表	17
23. 任务列表示例	17
24. 渲染后的摘录代码示例	18
25. 渲染后的引用示例	18
26. 表情符号自动完成器	19
27. 使用了大量表情符号的评论	19
28. 通过拖拽的方式自动插入图片	20
29. “Your repositories” 区域	20
30. “new repository” 下拉列表	21
31. “new repository” 表单	21
32. 版本库设置链接	22
33. 版本库合作者	22
34. 新的合并请求的邮件通知	23
35. 邮件回复	24
36. 合并按钮和手工合并一个合并请求的指令	24

37.	手工修改合并请求的目标	27
38.	输入 @ 来提醒某人	27
39.	取消订阅一个问题或合并请求	27
40.	通知中心选项	28
41.	通知中心	28
42.	开启合并请求时有 CONTRIBUTING 文件存在	30
43.	改变项目的默认分支	31
44.	把项目移交给另一个 GitHub 用户或组织	31
45.	“New organization” 菜单项	32
46.	组织页面	33
47.	团队页面	34
48.	审计日志	35
49.	服务与钩子配置区域	36
50.	电子邮件服务配置	36
51.	Web 钩子配置	37
52.	Web 钩子调试信息	38
53.	从设置页的 “Applications” 标签生成访问令牌	40
54.	从 GitHub API 发布的一条评论	41
55.	通过 API 的提交状态	43

GitHub 是最大的 Git 版本库托管商, 是成千上万的开发者和项目能够合作进行的中心。大部分 Git 版本库都托管在 GitHub, 很多开源项目使用 GitHub 实现 Git 托管、问题追踪、代码审查以及其它事情。所以, 尽管这不是 Git 开源项目的直接部分, 但如果想要专业地使用 Git, 你将不可避免地 >与 GitHub 打交道, 所以这依然是一个绝好的学习机会。

本章将讨论如何高效地使用 GitHub。我们将学习如何注册和管理账户、创建和使用 Git 版本库、向已有项目贡献的通用流程以及如何接受别人向你自己项目的贡献、GitHub 的编程接口和很多能够让这些操作更简单的小提示。

如果你对如何使用 GitHub 托管自己的项目, 或者与已经托管在 GitHub 上面的项目进行合作没有兴趣, 可以直接跳到??这一章。

1 账户的创建和配置

你所需要做的第一件事是创建一个免费账户。直接访问<https://github.com>, 选择一个未被占用的用户名, 提供一个电子邮件地址和密码, 点击写着 “Sign up for GitHub” 的绿色大按钮即可。

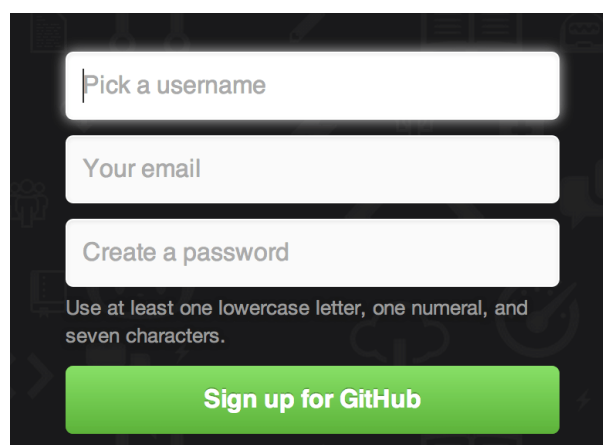
A screenshot of the GitHub sign-up form. It features three input fields: 'Pick a username', 'Your email', and 'Create a password'. Below the password field is a note: 'Use at least one lowercase letter, one numeral, and seven characters.' At the bottom is a prominent green button labeled 'Sign up for GitHub'.

图 1 GitHub 注册表单

你将看到的下一个页面是升级计划的价格页面, 目前我们可以直接忽略这个页面。GitHub 会给你提供的邮件地址发送一封验证邮件。尽快到你的邮箱进行验证, 这是非常重要的 (我们会在后面了解到这点)。

GitHub 为免费账户提供了完整功能, 限制是你的项目都将被完全公开 (每个人都具有读权限)。*GitHub* 的付费计划可以让你拥有一定数目的私有项目, 不过本书将不涉及这部分内容。

点击屏幕左上角的 Octocat 图标, 你将来到控制面板页面。现在, 你已经做好了使用 GitHub 的准备工作。

1.1 SSH 访问

现在, 你完全可以使用 `https://` 协议, 通过你刚刚创建的用户名和密码访问 Git 版本库。但是, 如果仅仅克隆公有项目, 你甚至不需要注册——刚刚我们创建的账户是为了以后 fork 其它项目, 以及推送我们自己的修改。

如果你习惯使用 SSH 远程, 你需要配置一个公钥。(如果你没有公钥, 参考 ??) 使用窗口右上角的链接打开你的账户设置:

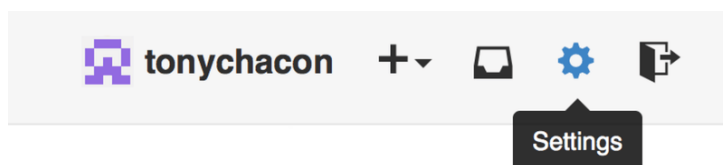


图 2 “Account settings” 链接

然后在左侧选择 “SSH keys” 部分。

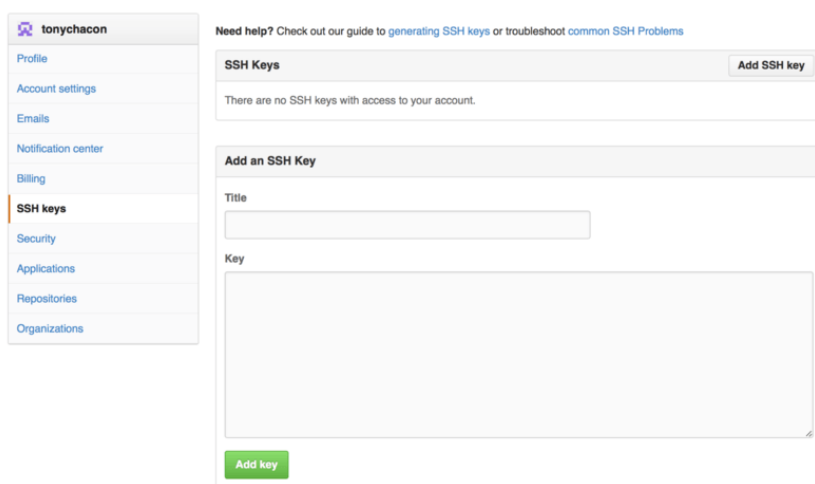


图 3 “SSH keys” 链接

在这个页面点击 “Add an SSH key” 按钮, 给你的公钥起一个名字, 将你的 `~/.ssh/id_rsa.pub` (或者自定义的其它名字) 公钥文件的内容粘贴到文本区, 然后点击 “Add key”。

确保给你的 *SSH* 密钥起一个能够记得住的名字。你可以为每一个密钥起名字 (例如, “我的笔记本电脑” 或者 “工作账户” 等), 以便以后需要吊销密钥时能够方便地区分。

1.2 头像

下一步, 如果愿意的话, 你可以将生成的头像换成你喜欢的图片。首先, 来到 “Profile” 标签页 (在 “SSH Keys” 标签页上方), 点击 “Upload new picture”。

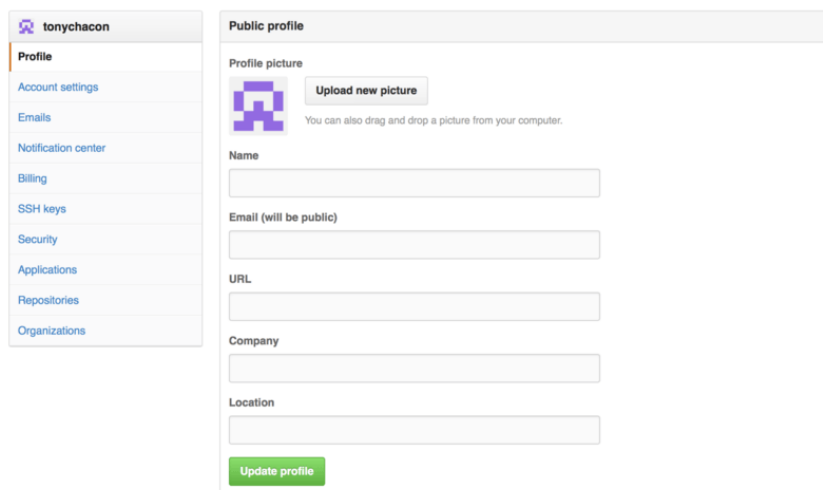


图 4 “Profile” 链接

我们选择了本地磁盘上的一个 Git 图标, 上传之后还可以对其进行裁剪。

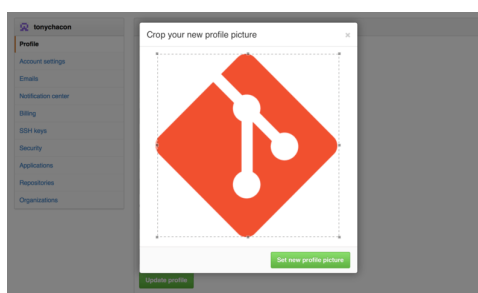


图 5 裁剪已上传的头像

现在, 在网站任意有你参与的位置, 人们都可以在你的用户名旁边看到你的头像。如果你已经把头像上传到了流行的 Gravatar 托管服务 (Wordpress 账户经常使用), 默认就会使用这个头像, 因此, 你就不需要进行这一步骤了。

1.3 邮件地址

GitHub 使用用户邮件地址区分 Git 提交。如果你在自己的提交中使用了多个邮件地址, 希望 GitHub 可以正确地将它们连接起来, 你需要在管理页面的 Emails 部分添加你拥有的所有邮箱地址。在[添加所有邮件地址](#)中我们可以看到一些不同的状态。顶部的地址是通过验证的, 并且被设置为主要地址, 这意味着该地址会接收到所有的通知和回复。第二个地址是通过验证的, 如果愿意的话, 可以将其设置为主要地址。最后一个地址是未通过验证的, 这意味着你不能将其设置为主要地址。当 GitHub 发现任意版本库中的任意提交信息包含了这些地址, 它就会将其链接到你的账户。

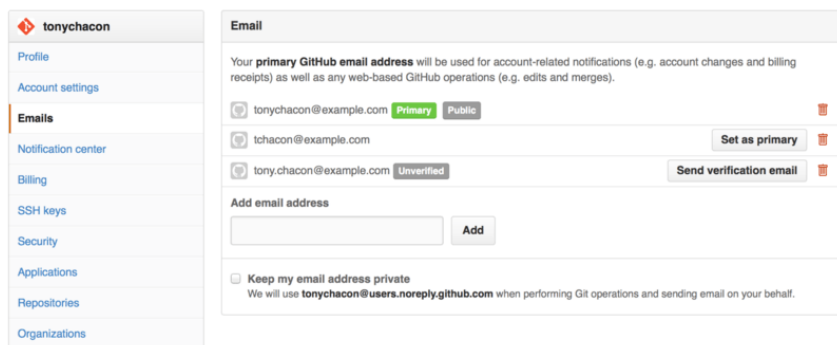


图 6 添加所有邮件地址

1.4 两步验证

最后, 为了额外的安全性, 你绝对应当设置两步验证, 简称为“2FA”。两步验证是一种用于降低因你的密码被盗而带来的账户风险的验证机制, 现在已经变得越来越流行。开启两步验证, GitHub 会要求你用两种不同的验证方法, 这样, 即使其中一个被攻破, 攻击者也不能访问你的账户。

你可以在 Account settings 页面的 Security 标签页中找到 Two-factor Authentication 设置。点击“Set up two-factor authentication”按钮, 会跳转到设置页面。该页面允许你选择是要在登

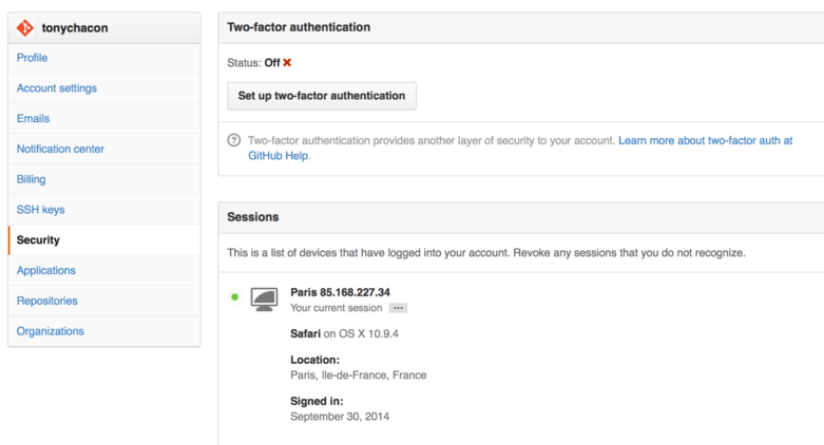


图 7 Security 标签页中的 2FA

录时使用手机 app 生成辅助码 (一种“基于时间的一次性密码”), 还是要 GitHub 通过 SMS 发送辅助码。

选择合适的方法后, 按照提示步骤设置 2FA, 你的账户会变得更安全, 每次登录 GitHub 时都需要提供除密码以外的辅助码。

2 对项目做出贡献

账户已经建立好了, 现在我们来了解一些能帮助你对现有的项目做出贡献的知识。

2.1 派生 (Fork) 项目

如果你想要参与某个项目, 但是并没有推送权限, 这时可以对这个项目进行“派生”。派生的意思是指, GitHub 将在你的空间中创建一个完全属于你的项目副本, 且你对其具有推送权限。

在以前, “fork” 是一个贬义词, 指的是某个人使开源项目向不同的方向发展, 或者创建一个竞争项目, 使得原项目的贡献者分裂。在 *GitHub*, “fork” 指的是你自己的空间中创建的项目副本, 这个副本允许你以一种更开放的方式对其进行修改。

通过这种方式, 项目的管理者不再需要忙着把用户添加到贡献者列表并给予他们推送权限。人们可以派生这个项目, 将修改推送到派生出的项目副本中, 并通过创建合并请求 (Pull Request) 来让他们的改动进入源版本库, 下文我们会详细说明。创建了合并请求后, 就会开启一个可供审查代码的板块, 项目的拥有者和贡献者可以在此讨论相关修改, 直到项目拥有者对其感到满意, 并且认为这些修改可以被合并到版本库。

你可以通过点击项目页面右上角的“Fork”按钮, 来派生这个项目。稍等片刻, 你将被转到



图 8 “Fork” 按钮

新项目页面, 该项目包含可写的代码副本。

2.2 GitHub 流程

GitHub 设计了一个以合并请求为中心的特殊合作流程。它基于我们在 ?? 的?? 中提到的工作流程。不管你是在一个紧密的团队中使用单独的版本库, 或者使用许多的“Fork”来为一个由陌生人组成的国际企业或网络做出贡献, 这种合作流程都能应付。

流程通常如下:

1. 从 `master` 分支中创建一个新分支
2. 提交一些修改来改进项目
3. 将这个分支推送到 GitHub 上
4. 创建一个合并请求
5. 讨论, 根据实际情况继续修改
6. 项目的拥有者合并或关闭你的合并请求

这基本和??中的一体化管理流程差不多, 但是团队可以使用 GitHub 提供的网页工具替代电子邮件来交流和审查修改。

现在来看一个使用这个流程的例子。

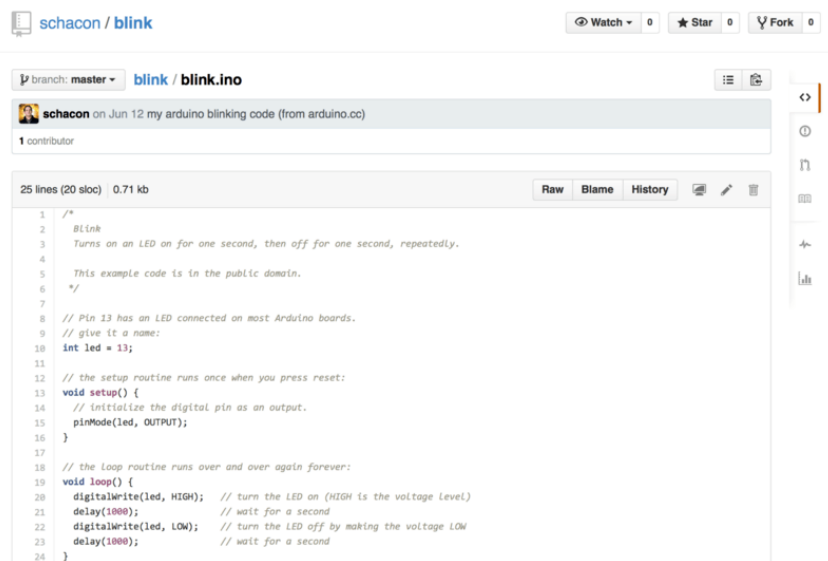


图 9 他想要做出贡献的项目

2.2.1 创建合并请求

Tony 在找一些能在他的 Arduino 微控制器上运行的代码，他觉得 <https://github.com/schacon/blink> 中的代码不错。

但是有个问题，这个代码中的闪烁频率太高，我们觉得 3 秒一次比 1 秒一次更好一些。所以让我们来改进这个程序，并将修改后的代码提交给这个项目。

首先，单击“Fork”按钮来获得这个项目的副本。我们使用的用户名是“tonychacon”，所以这个项目副本的访问地址是：<https://github.com/tonychacon/blink>。我们将它克隆到本地，创建一个分支，修改代码，最后再将改动推送到 GitHub。

```
$ git clone https://github.com/tonychacon/blink (1)
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink (2)
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (3)

$ git diff --word-diff (4)
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
```

```

void loop() {
    digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
    [-delay(1000);-]{+delay(3000);+}           // wait for a second
    digitalWrite(led, LOW);     // turn the LED off by making the voltage LOW
    [-delay(1000);-]{+delay(3000);+}           // wait for a second
}

$ git commit -a -m 'three seconds is better' (5)
[slow-blink 5ca509d] three seconds is better
1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink (6)
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
* [new branch]      slow-blink -> slow-blink

```

1. 将派生出的副本克隆到本地
2. 创建出名称有意义的分支
3. 修改代码
4. 检查改动
5. 将改动提交到分支中
6. 将新分支推送到 GitHub 的副本中

现在到 GitHub 上查看之前的项目副本, 可以看到 GitHub 提示我们有新的分支, 并且显示了一个大大的绿色按钮让我们可以检查我们的改动, 并给源项目创建合并请求。

你也可以到 “Branches” (分支) 页面查看分支并创建合并请求:
<https://github.com/<用户名>/<项目名>/branches>

如果你点击了那个绿色按钮, 就会看到一个新页面, 在这里我们可以对改动填写标题和描述, 让项目的拥有者考虑一下我们的改动。通常花点时间来编写个清晰有用的描述是个不错的主意, 这能让作者明白为什么这个改动可以给他的项目带来好处, 并且让他接受合并请求。

同时我们也能看到比主分支中所“领先” (ahead) 的提交 (在这个例子中只有一个) 以及所有将会被合并的改动与之前代码的对比。

当你单击了 “Create pull request” (创建合并请求) 的按钮后, 这个项目的拥有者将会收到一条包含改动和合并请求页面的链接的提醒。

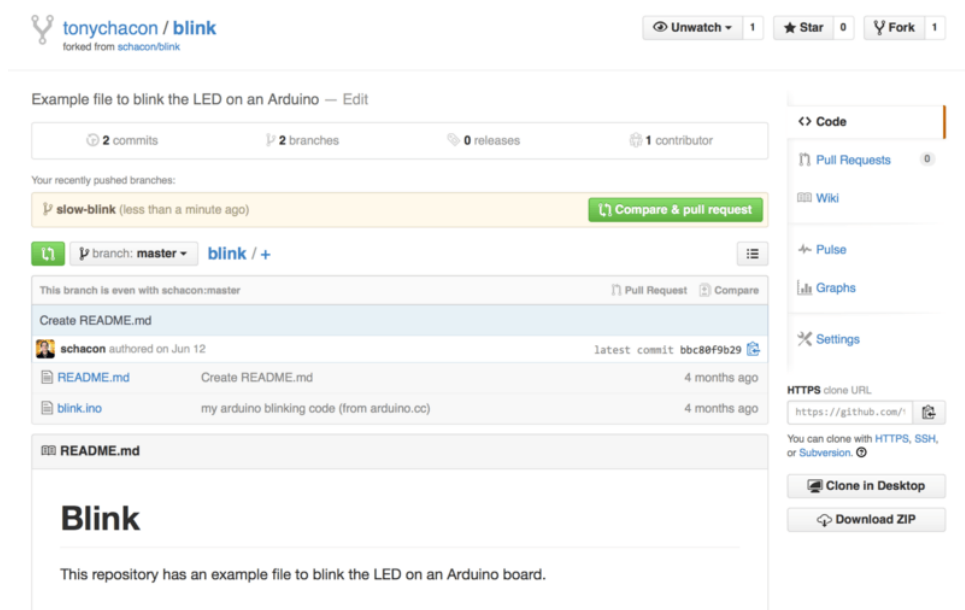


图 10 合并请求按钮

虽然合并请求通常是在贡献者准备好在公开项目中提交改动的时候提交，但是也被用在仍处于开发阶段的内部项目中。因为合并请求在提交后 **依然可以加入新的改动**，它也经常被用来建立团队合作的环境，而不只是在最终阶段使用。

2.2.2 利用合并请求

现在，项目的拥有者可以看到你的改动并合并它，拒绝它或是发表评论。在这里我们就当作他喜欢这个点子，但是他想要让灯熄灭的时间比点亮的时间稍长一些。

接下来可能会通过电子邮件进行互动，就像我们在??中提到的工作流程那样，但是在 GitHub，这些都在线上完成。项目的拥有者可以审查修改，只需要单击某一行，就可以对其发表评论。

当维护者发表评论后，提交合并请求的人，以及所有正在关注 (Watching) 这个版本库的用户都会收到通知。我们待会儿将会告诉你如何修改这项设置。现在，如果 Tony 有开启电子邮件提醒，他将会收到这样的一封邮件：

每个人都能在合并请求中发表评论。在[合并请求讨论页面](#)里我们可以看到项目拥有者对某行代码发表评论，并在讨论区留下了一个普通评论。你可以看到被评论的代码也会在互动中显示出来。

现在贡献者可以看到如何做才能让他们的改动被接受。幸运的是，这也是一件轻松的事情。如果你使用的是电子邮件进行交流，你需要再次对代码进行修改并重新提交至邮件列表，在 GitHub 上，你只需要再次提交到你的分支中并推送即可。

如果贡献者完成了以上的操作，项目的拥有者会再次收到提醒，当他们查看页面时，将会看到最新的改动。事实上，只要提交中有一行代码改动，GitHub 都会注意到并处理掉旧的变更集。

如果你点开合并请求的“Files Changed”（更改的文件）选项卡，你将会看到“整理

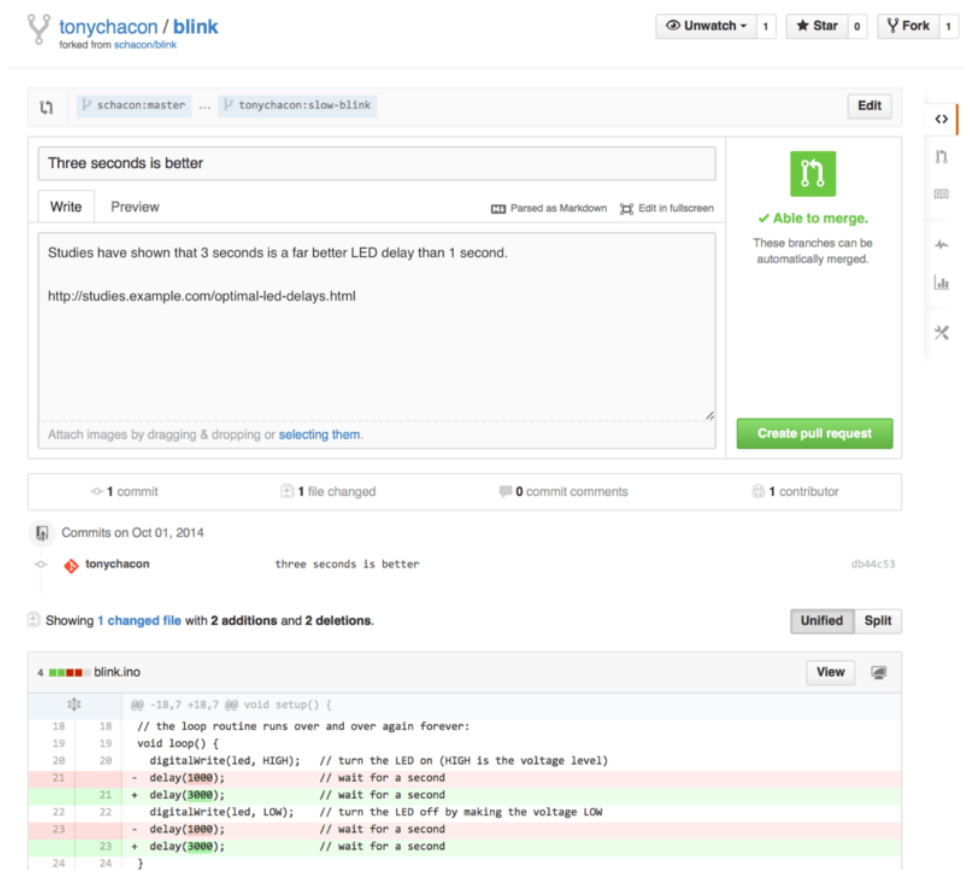


图 11 合并请求创建页面

过的”差异表——也就是这个分支被合并到主分支之后将会产生的所有改动，其实就是 `git diff master...<分支名>` 命令的执行结果。你可以浏览 ?? 来了解更多关于差异表的知识。

你还会注意到，GitHub 会检查你的合并请求是否能直接合并，如果可以，将会提供一个按钮来进行合并操作。这个按钮只在你对本版本库有写入权限并且可以进行简洁合并时才会显示。你点击后 GitHub 将做出一个“非快进式”（non-fast-forward）合并，即使这个合并能够快进式（fast-forward）合并，GitHub 依然会创建一个合并提交。

如果你需要，你还可以将分支拉取并在本地合并。如果你将这个分支合并到 `master` 分支中并推送到 GitHub，这个合并请求会被自动关闭。

这就是大部分 GitHub 项目使用的工作流程。创建分支，基于分支创建合并请求，进行讨论，根据需要在分支上进行修改，最终关闭或合并合并请求。

不必总是 *Fork*

有件很重要的事情：你可以在同一个版本库中不同的分支提交合并请求。如果你正在和某人实现某个功能，而且你对项目有写权限，你可以推送分支到版本库，并在 `master` 分支提交一个合并请求并在此进行代码审查和讨论的操作。不需要进行“*Fork*”。

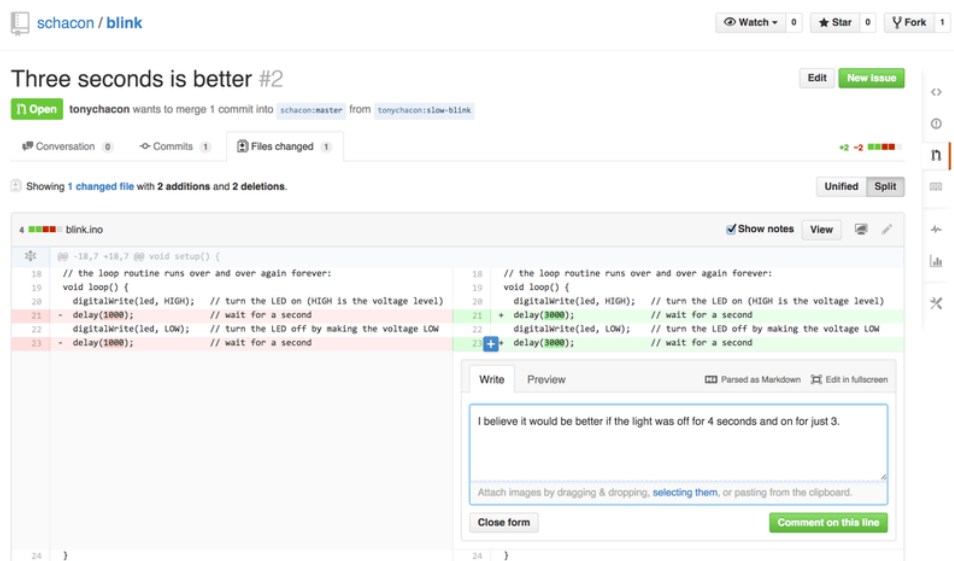


图 12 对合并请求内的特定一行发表评论



图 13 通过电子邮件发送的评论提醒

2.3 合并请求的进阶用法

目前, 我们学到了如何在 GitHub 平台对一个项目进行最基础的贡献。现在我们会教给你一些小技巧, 让你可以更加有效率地使用合并请求。

2.3.1 将合并请求制作成补丁

有一件重要的事情: 许多项目并不认为合并请求可以作为补丁, 就和通过邮件列表工作的项目对补丁贡献的看法一样。大多数的 GitHub 项目将合并请求的分支当作对改动的交流方式, 并将变更集合起来统一进行合并。

这是个重要的差异, 因为一般来说改动会在代码完成前提出, 这和基于邮件列表的补丁贡献有着天差地别。这使得维护者们可以更早的沟通, 由社区中的力量能提出更好的方案。当有人从合并请求提交了一些代码, 并且维护者和社区提出了一些意见, 这个补丁系列并不需要从头来过, 只需要将改动重新提交并推送到分支中, 这使得讨论的背景和过程可以齐头并进。

举个例子, 你可以回去看看[最终的合并请求](#), 你会注意到贡献者没有变基他的提交再提交一

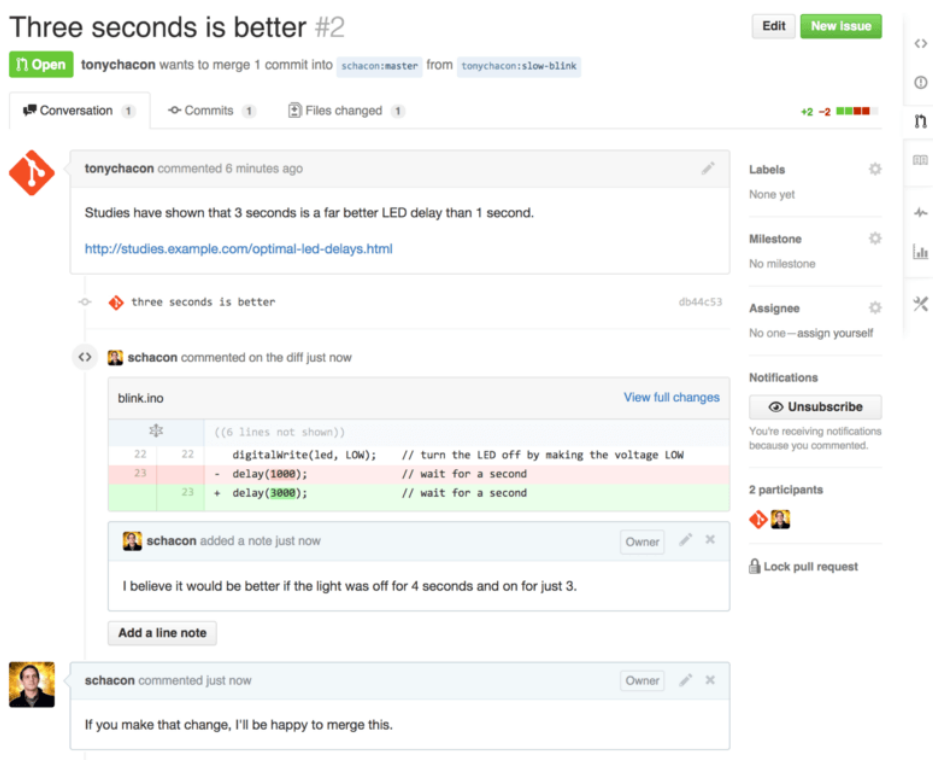


图 14 合并请求讨论页面

个新的合并请求, 而是直接增加了新的提交并推送到已有的分支中。如果你之后再回去查看这个合并请求, 你可以轻松地找到这个修改的原因。点击网页上的“Merge”(合并) 按钮后, 会建立一个合并提交并指向这个合并请求, 你就可以很轻松的研究原来的讨论内容。

2.3.2 与上游保持同步

如果你的合并请求由于过时或其他原因不能干净地合并, 你需要进行修复才能让维护者对其进行合并。GitHub 会对每个提交进行测试, 让你知道你的合并请求能否简洁的合并。

如果你看到了像不能进行干净合并中的画面, 你就需要修复你的分支让这个提示变成绿色, 这样维护者就不需要再做额外的工作。

你有两种方法来解决这个问题。你可以把你的分支变基到目标分支中去 (通常是你派生出的版本库中的master 分支), 或者你可以合并目标分支到你的分支中去。

GitHub 上的大多数的开发者会使用后一种方法, 基于我们在上一节提到的理由: 我们最看重的是历史记录和最后的合并, 变基除了给你带来看上去简洁的历史记录, 只会让你的工作变得更加困难且更容易犯错。

如果你想要合并目标分支来让你的合并请求变得可合并, 你需要将源版本库添加为一个新的远端, 并从远端抓取内容, 合并主分支的内容到你的分支中去, 修复所有的问题并最终重新推送回你提交合并请求使用的分支。

在这个例子中, 我们再次使用之前的“tonychacon” 用户来进行示范, 源作者提交了一个改动, 使得合并请求和它产生了冲突。现在来看我们解决这个问题的步骤。

Three seconds is better #2

Open tonychacon wants to merge 3 commits into schacon:master from tonychacon:slow-blink

Conversation 3 Commits 3 Files changed 1

tonychacon commented 11 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.

<http://studies.example.com/optimal-led-delays.html>

three seconds is better db44c53

schacon commented on an outdated diff 5 minutes ago [Show outdated diff](#)

schacon commented 5 minutes ago

If you make that change, I'll be happy to merge this.

tonychacon added some commits 2 minutes ago

- longer off time 0c1f66f
- remove trailing whitespace ef4725c

tonychacon commented 10 seconds ago

I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?

This pull request can be automatically merged.
You can also merge branches on the [command line](#).

[Merge pull request](#)

图 15 最终的合并请求

```
$ git remote add upstream https://github.com/schacon/blink (1)

$ git fetch upstream (2)
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master      -> upstream/master

$ git merge upstream/master (3)
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino (4)
```

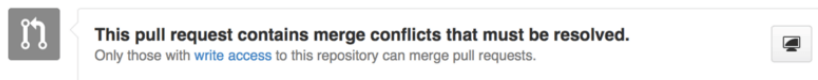



图 16 不能进行干净合并

```
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink (5)
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
ef4725c..3c8d735  slower-blink -> slow-blink
```

1. 将源版本库添加为一个远端, 并命名为“upstream”(上游)
2. 从远端抓取最新的内容
3. 将主分支的内容合并到你的分支中
4. 修复产生的冲突
5. 再推送回同一个分支

你完成了上面的步骤后, 合并请求将会自动更新并重新检查是否能干净的合并。

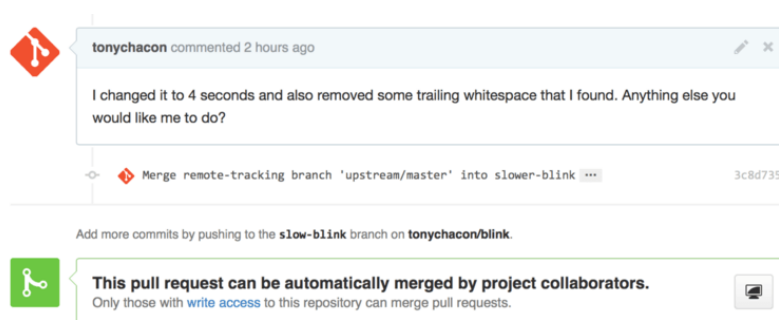


图 17 合并请求现在可以干净地合并了

Git 的伟大之处就是你可以一直重复以上操作。如果你有一个运行了十分久的项目, 你可以轻松地合并目标分支且只需要处理最近的一次冲突, 这使得管理流程更加容易。

如果你一定想对分支做变基并进行清理, 你可以这么做, 但是强烈建议你不要强行的提交到已经提交了合并请求的分支。如果其他人拉取了这个分支并进行一些修改, 你将会遇到??中提

到的问题。相对的, 将变基后的分支推送到 GitHub 上的一个新分支中, 并且创建一个全新的合并请求引用旧的合并请求, 然后关闭旧的合并请求。

2.3.3 参考

你的下一个问题可能是“我该如何引用旧的合并请求? ”。有许多方法可以让你在 GitHub 上的几乎任何地方引用其他东西。

先从如何对合并请求或议题 (Issue) 进行相互引用开始。所有的合并请求和议题在项目中都会有一个独一无二的编号。举个例子, 你无法同时拥有 3 号合并请求和 3 号议题。如果你想要引用任何一个合并请求或议题, 你只需要在提交或描述中输入 `#<编号>` 即可。你也可以指定引用其他版本库的议题或合并请求, 如果你想要引用其他人对该版本库的“Fork”中的议题或合并请求, 输入 `用户名#<编号>`, 如果在不同的版本库中, 输入 `用户名/版本库名#<编号>`。

我们来看一个例子。假设我们对上个例子中的分支进行了变基, 并为此创建一个新的合并请求, 现在我们希望能在新的合并请求中引用旧的合并请求。我们同时希望引用一个派生出的项目中的议题和一个完全不同的项目中的议题, 就可以像 [在合并请求中的交叉引用](#) 这样填写描述。

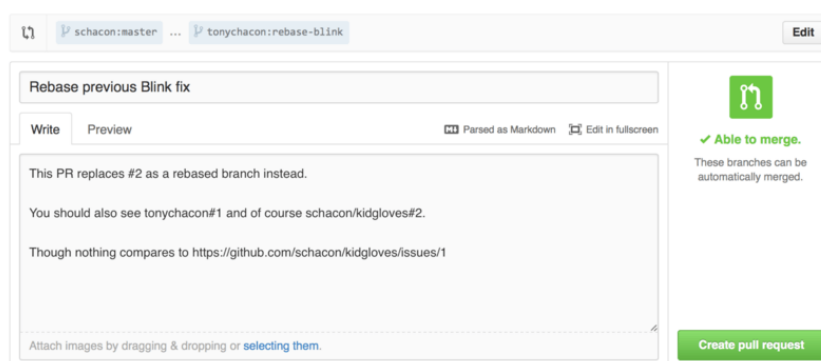


图 18 在合并请求中的交叉引用

当我们提交了这个合并请求, 我们将会看到以上内容被渲染成这样: [在合并请求中渲染后的交叉引用](#)。

你会注意到完整的 GitHub 地址被简化了, 只留下了必要的信息。

如果 Tony 回去关闭了源合并请求, 我们可以看到一个被引用的提示, GitHub 会自动地反向追踪事件并显示在合并请求的时间轴上。这意味着任何查看这个合并请求的人可以轻松地访问新的合并请求。这个链接就像[在合并请求中渲染后的交叉引用](#)中展示的那样。

除了议题编号外, 你还可以通过使用提交的 SHA-1 来引用提交。你必须完整的写出 40 位长的 SHA, GitHub 会在评论中自动地产生指向这个提交的链接。同样的, 你可以像引用议题一样对“Fork”出的项目中的提交或者其他项目中的提交进行引用。

2.4 Markdown

对于在 GitHub 中绝大多数文本框中能够做到的事, 引用其他议题只是个开始。在议题和合并请求的描述, 评论和代码评论还有其他地方, 都可以使用“GitHub 风格的 Markdown”。

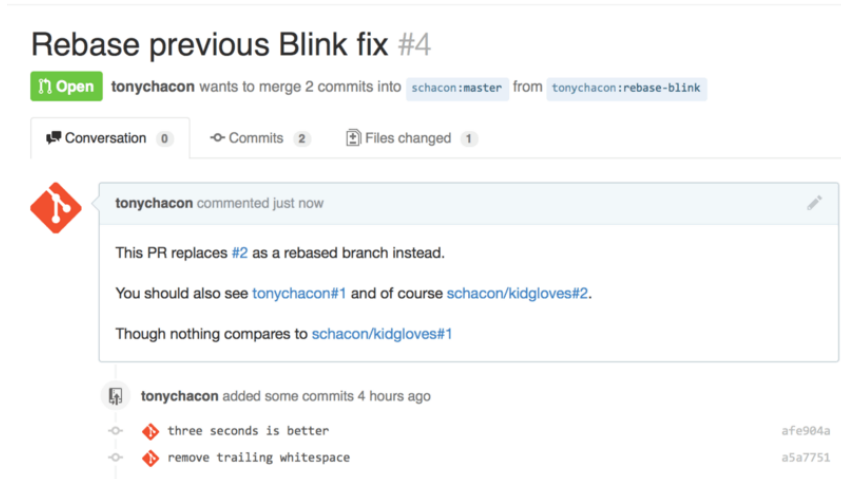


图 19 在合并请求中渲染后的交叉引用

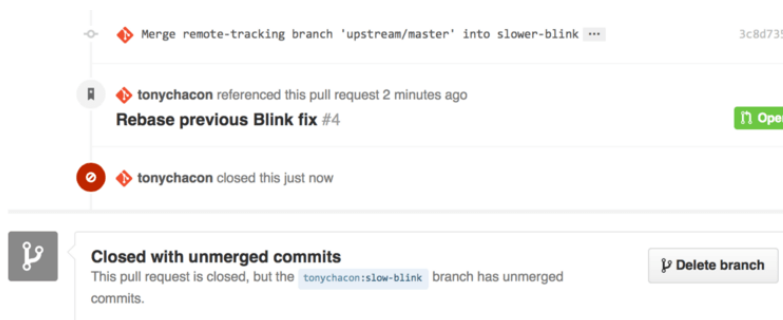


图 20 在合并请求中渲染后的交叉引用

Markdown 可以让你输入纯文本，但是渲染出丰富的内容。

查看一个 [Markdown 的例子和渲染效果](#)里的例子来了解如何书写评论或文本，并通过 Markdown 进行渲染。

2.4.1 GitHub 风格的 Markdown

GitHub 风格的 Markdown 增加了一些基础的 Markdown 中做不到的东西。它在创建合并请求和议题中的评论和描述时十分有用。

任务列表 第一个 GitHub 专属的 Markdown 功能，特别是用在合并请求中，就是任务列表。一个任务列表可以展示出一系列你想要完成的事情，并带有复选框。把它们放在议题或合并请求中时，通常可以展示你想要完成的事情。

你可以这样创建一个任务列表：

- [X] 编写代码
- [] 编写所有测试程序
- [] 为代码编写文档

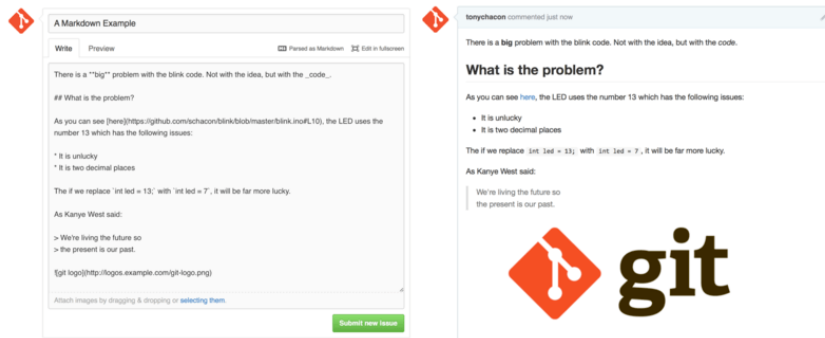


图 21 一个 Markdown 的例子和渲染效果

如果我们将这个列表加入合并请求或议题的描述中，它将会被渲染 [Markdown 评论中渲染后的任务列表](#) 这样。

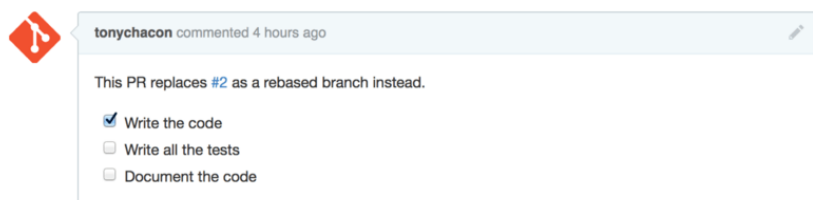


图 22 Markdown 评论中渲染后的任务列表

在合并请求中，任务列表经常被用来在合并之前展示这个分支将要完成的事情。最酷的地方就是，你只需要点击复选框，就能更新评论——你不需要直接修改 Markdown。

不仅如此，GitHub 还会将你在议题和合并请求中的任务列表整理起来集中展示。举个例子，如果你在一个合并请求中有任务清单，你将会在所有合并请求的总览页面上看到它的进度。这使得人们可以把一个合并请求分解成不同的小任务，同时便于其他人了解分支的进度。你可以在[任务列表示例](#)看到一个例子。

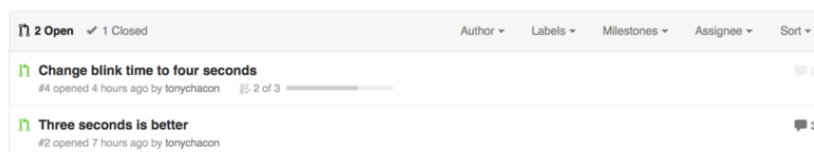


图 23 任务列表示例

当你在实现一个任务的早期就提交合并请求，并使用任务清单追踪你的进度，这个功能会十分的有用。

摘录代码 你也可以在评论中摘录代码。这在你想要展示尚未提交到分支中的代码时会十分有用。它也经常被用在展示无法正常工作的代码或这个合并请求需要的代码。

你需要用“反引号”将需要添加的摘录代码包起来。

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```
```

如果加入语言的名称, 就像我们这里加入的“java”一样, GitHub 会自动尝试对摘录的片段进行语法高亮。在下面的例子中, 它最终会渲染成这个样子: [渲染后的摘录代码示例](#)。



图 24 渲染后的摘录代码示例

引用 如果你在回复一个很长的评论之中的一小段, 你只需要复制你需要的片段, 并在每行前添加 > 符号即可。事实上, 因为这个功能会被经常用到, 它也有一个快捷键。只要你把你要回应的文字选中, 并按下 **r** 键, 选中的问题会自动引用并填入评论框。

引用的部分就像这样:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

经过渲染后, 就会变成这样: [渲染后的引用示例](#)。

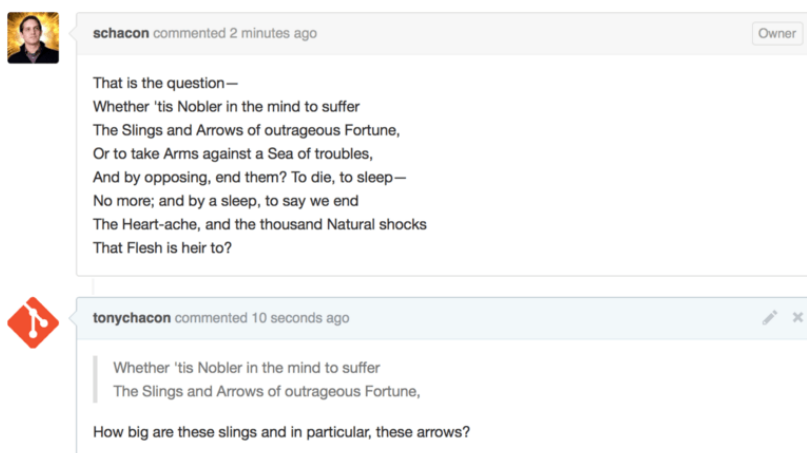


图 25 渲染后的引用示例

表情符号 (Emoji) 最后, 我们可以在评论中使用表情符号。这经常出现在 GitHub 的议题和合并请求的评论中。GitHub 上甚至有表情助手。如果你在输入评论时以 : 开头, 自动完成器会帮助你找到你需要的表情。

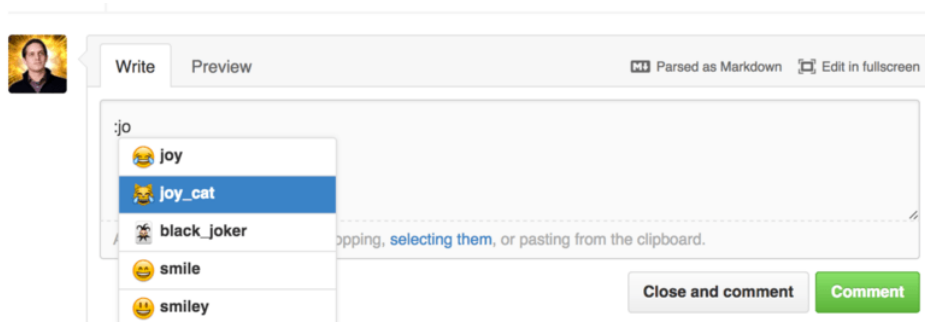


图 26 表情符号自动完成器

你也可以在评论的任何地方使用 :<表情名称>: 来添加表情符号。举个例子, 你可以输入以下文字:

```
I :eyes: that :bug: and I :cold_sweat:.  
  
:trophy: for :microscope: it.  
  
:+1: and :sparkles: on this :ship:, it's :fire::poop:!  
  
:clap::tada::panda_face:
```

渲染之后, 就会变成这样: 使用了大量表情符号的评论。

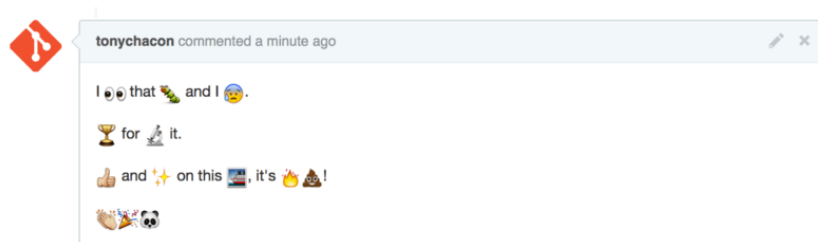


图 27 使用了大量表情符号的评论

虽然这个功能并不是非常实用, 但是它在这种不方便表达感情的媒体里, 加入了趣味的元素。

事实上现在已经有大量的在线服务可以使用表情符号, 这里有个列表可以让你快速的找到能表达你的情绪的表情符号:

<http://www.emoji-cheat-sheet.com>

图片 从技术层面来说,这并不是 GitHub 风格 Markdown 的功能,但是也很有用。如果不想使用 Markdown 语法来插入图片, GitHub 允许你通过拖拽图片到文本区来插入图片。

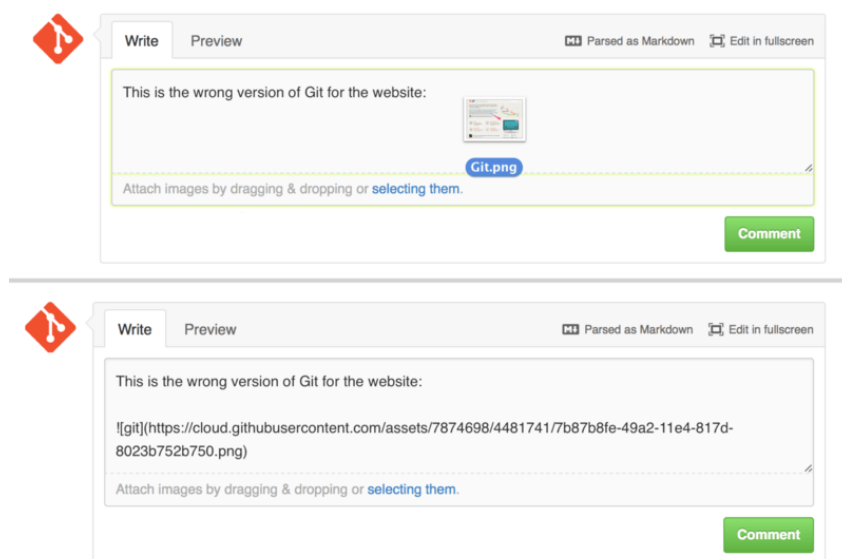


图 28 通过拖拽的方式自动插入图片

如果你回去查看[在合并请求中的交叉引用](#),你会发现文本区上有个“Parsed as Markdown”的提示。点击它你可以了解所有能在 GitHub 上使用的 Markdown 功能。

3 维护项目

3.1 维护项目

现在我们可以很方便地向一个项目贡献内容,来看一下另一个方面的内容:创建、维护和管理你自己的项目。

3.2 创建新的版本库

让我们创建一个版本库来分享我们的项目。通过点击面板右侧的“New repository”按钮,或者顶部工具条你用户名旁边的 + 按钮来开始我们的旅程。参见[“new repository”下拉列表](#)。

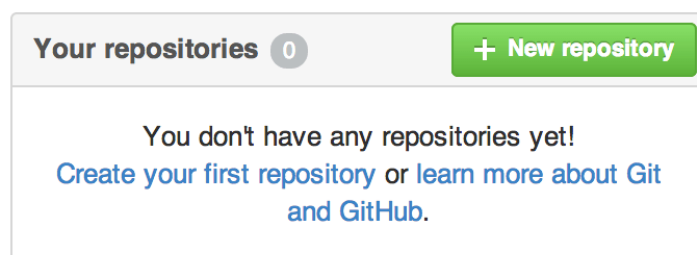


图 29 “Your repositories” 区域

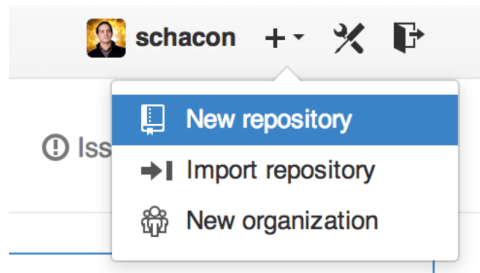


图 30 “new repository” 下拉列表

这会带你到 “new repository” 表单:

图 31 “new repository” 表单

这里除了一个你必须要填的项目名，其他字段都是可选的。现在只需要点击 “Create Repository” 按钮，Duang!!!——你就在 GitHub 上拥有了一个以 `<user>/<project_name>` 命名的新仓库了。

因为目前暂无代码，GitHub 会显示有关创建新版本库或者关联到一个已有的 Git 版本库的一些说明。我们不会在这里详细说明此项，如果你需要复习，去看 ??。

现在你的项目就托管在 GitHub 上了，你可以把 URL 给任何你想分享的人。GitHub 上的项目可通过 HTTP 或 SSH 访问，格式是：
HTTP: `https://github.com/<user>/<project_name>`, SSH: `git@github.com:<user>/<project_name>`。Git 可以通过以上两种 URL 进行抓取和推送，但是用户的访问权限又因连接时使用的证书不同而异。

通常对于公开项目可以优先分享基于 HTTP 的 URL，因为用户克隆项目不需要有一个 GitHub 帐号。如果你分享 SSH URL，用户必须有一个帐号并且上传 SSH 密钥才能访问你的项目。HTTP URL 与你贴到浏览器里查看项目用的地址是一样的。

3.3 添加合作者

如果你想与他人合作, 并想给他们提交的权限, 你需要把他们添加为“Collaborators”。如果 Ben、Jeff、Louise 都在 GitHub 上注册了, 你想给他们推送的权限, 你可以将他们添加到你的项目。这样做会给他们“推送”权限, 就是说他们对项目和 Git 版本库都有读写的权限。

点击边栏底部的“Settings”链接。

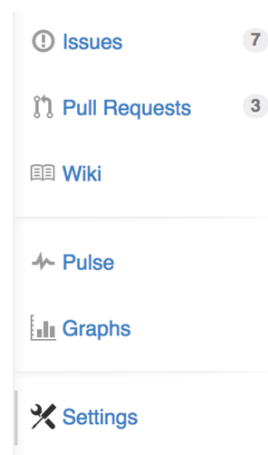


图 32 版本库设置链接

然后从左侧菜单中选择“Collaborators”。然后, 在输入框中填写用户名, 点击“Add collaborator.”。如果你想授权给多个人, 你可以多次重复这个步骤。如果你想收回权限, 点击他们同一行右侧的“X”

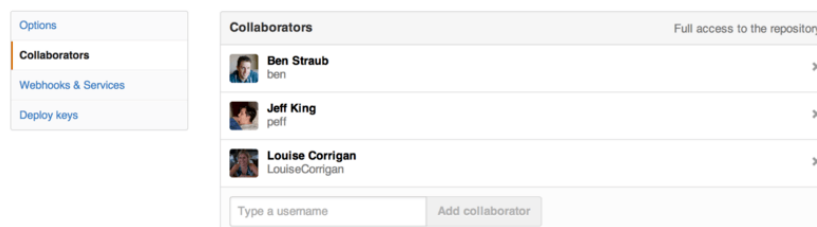


图 33 版本库合作者

3.4 管理合并请求

现在你有一个包含一些代码的项目, 可能还有几个有推送权限的合作者, 下面来看当你收到合并请求时该做什么。

合并请求可以来自仓库副本的一个分支, 或者同一仓库的另一个分支。唯一的区别是 fork 过来的通常是和你不能互相推送的人, 而内部的推送通常都可以互相访问。

作为例子, 假设你是“tonychacon”, 你创建了一个名为“fade”的 Arduino 项目。

3.4.1 邮件通知

有人来修改了你的代码, 给你发了一个合并请求。你会收一封关于合并请求的提醒邮件, 它看起来像[新的合并请求的邮件通知](#)。



图 34 新的合并请求的邮件通知

关于这个邮件有几个要注意的地方。它会给你一个小的变动统计结果——一个包含合并请求中改变的文件和改变了多少的列表。它还给你一个 GitHub 上进行合并请求操作的链接。还有几个可以在命令行使用的 URL。

如果你注意到 `git pull <url> patch-1` 这一行, 这是一种合并远程分支的简单方式, 无需必须添加一个远程分支。我们很快会在??讲到它。如果你愿意, 你可以创建并切换到一个主题分支, 然后运行这个命令把合并请求合并进来。

还有一些有趣的 URL, 像 `.diff` 和 `.patch`, 就像你猜的那样, 它们提供 diff 和 patch 的标准版本。你可以技术性地用下面的方法合并“合并请求”:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

3.4.2 在合并请求上进行合作

就像我们在[GitHub 流程](#)说过的, 现在你可以跟开启合并请求的人进行会话。你既可以对某些代码添加注释, 也可以对整个提交添加注释或对整个合并请求添加注释, 在任何地方都可以用 GitHub Flavored Markdown。

每次有人在合并请求上进行注释你都会收到通知邮件, 通知你哪里发生改变。他们都会包含一个到改变位置的链接, 你可以直接在邮件中对合并请求进行注释。

一旦代码符合了你的要求, 你想把它合并进来, 你可以把代码拉取下来在本地进行合并, 也可以用我们之前提到过的 `git pull <url> <branch>` 语法, 或者把 fork 添加为一个 remote, 然后进行抓取和合并。

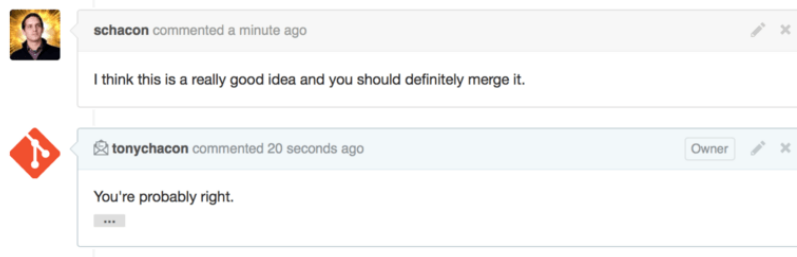


图 35 邮件回复

对于很琐碎的合并，你也可以用 GitHub 网站上的“Merge”按钮。它会做一个“non-fast-forward”合并，即使可以快进 (fast-forward) 合并也会产生一个合并提交记录。就是说无论如何，只要你点击 merge 按钮，就会产生一个合并提交记录。你可以在[合并按钮和手工合并一个合并请求的指令](#)看到，如果你点击提示链接，GitHub 会给你所有的这些信息。

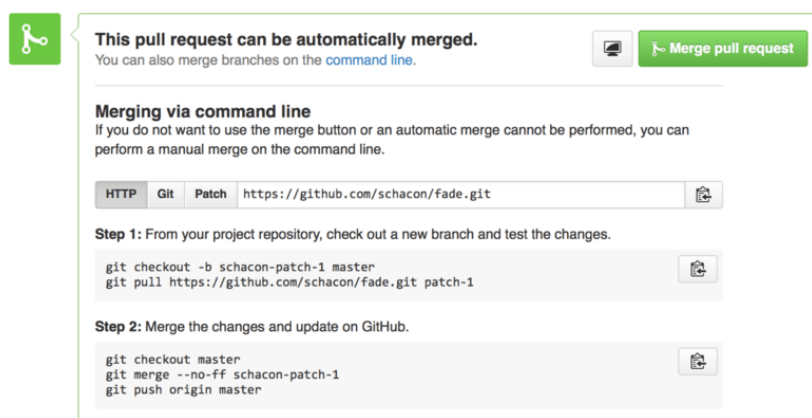


图 36 合并按钮和手工合并一个合并请求的指令

如果你决定不合并它，你可以把合并请求关掉，开启合并请求的人会收到通知。

3.4.3 合并请求引用

如果你正在处理许多合并请求，不想添加一堆 remote 或者每次都要做一次拉取，这里有一个可以在 GitHub 上用的小技巧。这是有点高级的技巧，但它相当有用，我们会在??有更多的细节说明。

实际上 GitHub 在服务器上把合并请求分支视为一种“假分支”。默认情况下你克隆时不会得到它们，但它们还是隐式地存在，你可以很容易地访问到它们。

为了展示这个，我们要用到一个叫做 `ls-remote` 的低级命令 (通常被叫做“plumbing”，我们会在??读到更多相关内容)。这个命令在日常 Git 操作中基本不会用到，但在显示服务器上有哪些引用 (reference) 时很管用。

如果在我们之前用过的“blink”版本库上使用这个命令，我们会得到一个版本库里所有的分支，标签和其它引用 (reference) 的列表。

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d      HEAD
10d539600d86723087810ec636870a504f4fee4d      refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e        refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3        refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1        refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d        refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a        refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c        refs/pull/4/merge
```

当然，如果你在你自己的版本库或其它你想检查的远程版本库中使用 `git ls-remote origin`，它会显示相似的内容。

如果版本库在 GitHub 上并且有打开的合并请求，你会得到一些以 `refs/pull/` 开头的引用。它们实际上是分支，但因为它们不在 `refs/heads/` 中，所以正常情况下你克隆时不会从服务器上得到它们——抓取过程正常情况下会忽略它们。

每个合并请求有两个引用——其中以 `/head` 结尾的引用指向的提交记录与合并请求分支中的最后一个提交记录是同一个。所以如果有人在我们的版本库中开启了一个合并请求，他们的分支叫做 `bug-fix`，指向 `a5a775` 这个提交记录，那么在**我们的**版本库中我们没有 `bug-fix` 分支（因为那是在他们的 fork 中），但我们可以有一个 `pull/<pr#>/head` 指向 `a5a775`。这意味着我们可以很容易地拉取每一个合并请求分支而不用添加一堆 `remote`。

现在，你可以像直接抓取引用一样抓取那些分支或提交。

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
* branch                refs/pull/958/head -> FETCH_HEAD
```

这告诉 Git：“连接到 `origin` 这个 `remote`，下载名字为 `refs/pull/958/head` 的引用。”Git 高高兴兴去执行，下载构建那个引用需要的所有内容，然后把指针指向 `.git/FETCH_HEAD` 下面你想要的提交记录。然后你可以用 `git merge FETCH_HEAD` 把它合并到你进行测试的分支，但那个合并的提交信息看起来有点怪。然而，如果你需要审查一大批合并请求，这样操作会很麻烦。

还有一种方法可以抓取**所有的**合并请求，并且在你连接到远程 (`remote`) 的时候保持更新。用你最喜欢的编辑器打开 `.git/config`，查找 `origin` 远程 (`remote`)。看起来差不多像下面这样：

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2
  fetch = +refs/heads/*:refs/remotes/origin/*
```

以 `fetch =` 开头的行是一个“refspec.”它是一种把 `remote` 的名称映射到你本地 `.git` 目录的方法。这一条（就是上面的这一条）告诉 Git，“remote 上 `refs/heads` 下面的内容在我本地版本库中都放在 `refs/remotes/origin`。”你可以把这一段修改一下，添加另一个 refspec：

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
```

```
fetch = +refs/heads/*:refs/remotes/origin/*
fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

最后一行告诉 Git: “所有看起来像 `refs/pull/123/head` 的引用应该在本地版本库像 `refs/remotes/origin/pr/123` 一样存储” 现在, 如果你保存那个文件, 执行 `git fetch`:

```
$ git fetch
# ...
* [new ref]          refs/pull/1/head -> origin/pr/1
* [new ref]          refs/pull/2/head -> origin/pr/2
* [new ref]          refs/pull/4/head -> origin/pr/4
# ...
```

现在所有的合并请求在本地像分支一样展现, 它们是只读的, 当你执行抓取时它们也会更新。这让在本地测试合并请求中的代码变得超级简单:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

你的鹰眼系统会发现在 `refspec` 的 `remote` 部分的结尾有个 `head`。在 GitHub 那边也有一个 `refs/pull/#/merge` 引用, 它代表的是如果你在网站上按了 “merge” 按钮对应的提交记录。这甚至让你可以在按按钮之前就测试这个合并。

3.4.4 合并请求之上的合并请求

你不仅可以在主分支或者说 `master` 分支上开启合并请求, 实际上你可以在网络上的任何一个分支上开启合并请求。其实, 你甚至可以在另一个合并请求上开启一个合并请求。

如果你看到一个合并请求在向正确的方向发展, 然后你想在这个合并请求上做一些修改或者你不太确定这是个好主意, 或者你没有目标分支的推送权限, 你可以直接在合并请求上开启一个合并请求。

当你开启一个合并请求时, 在页面的顶端有一个框框显示你要合并到哪个分支和你从哪个分支合并过来的。如果你点击那个框框右边的 “Edit” 按钮, 你不仅可以改变分支, 还可以选择哪个 fork。

这里你可以很简单地指明合并你的分支到哪一个合并请求或 fork。

3.5 提醒和通知

GitHub 内置了一个很好的通知系统, 当你需要与别人或别的团队交流时用起来很方便。

在任何评论中你可以先输入一个 @, 系统会自动补全项目中合作者或贡献者的名字和用户名。

你也可以提醒不在列表中的用户, 但是通常自动补全用起来更快。

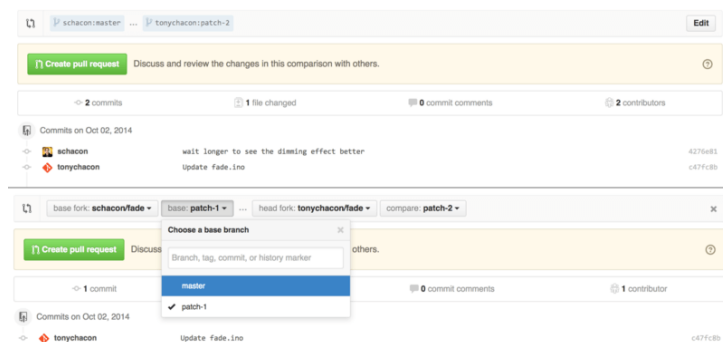


图 37 手工修改合并请求的目标



图 38 输入 @ 来提醒某人

当你发布了一个带用户提醒的评论, 那个用户会收到通知。这意味着把人们拉进会话中要比让他们投票有效率得多。对于 GitHub 上的合并请求, 人们经常把他们团队或公司中的其它人拉来审查问题或合并请求。

如果有人收到了合并请求或问题的提醒, 他们会“订阅”它, 后面有新的活动发生他们都会持续收到提醒。如果你是合并请求或者问题的发起方你也会被订阅上, 比如你在关注一个版本库或者你评论了什么东西。如果你不想再收到提醒, 在页面上有个“Unsubscribe”按钮, 点一下就不会再收到更新了。

Notifications

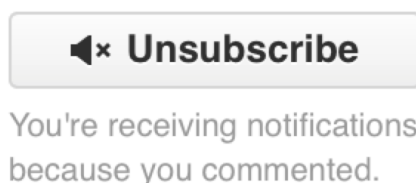


图 39 取消订阅一个问题或合并请求

3.6 通知页面

当我们在这提到特指 GitHub 的“notifications”, 指的是当 GitHub 上有事件发生时, 它通知你的方式, 这里有几种不同的方式来配置它们。如果你打开配置页面的“Notification center”

标签, 你可以看到一些选项。

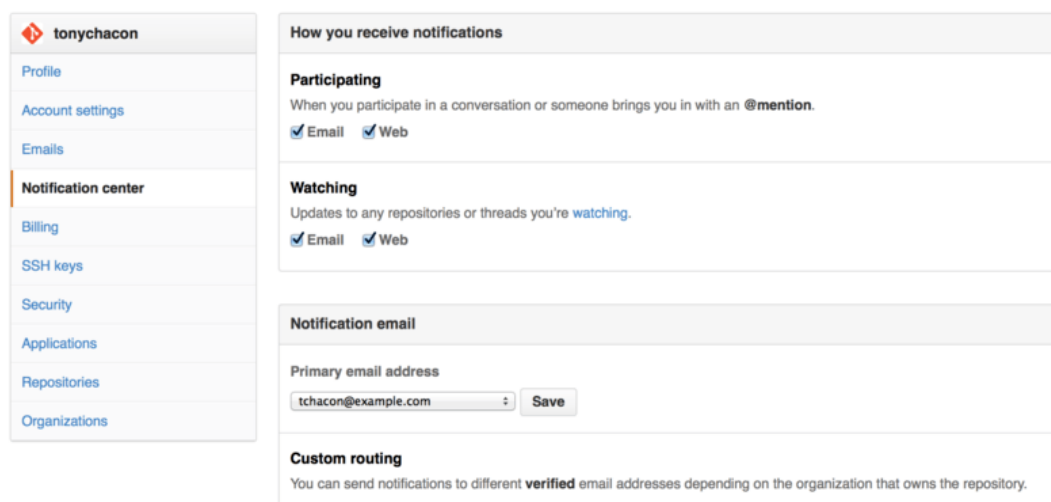


图 40 通知中心选项

有两个选项, 通过“邮件 (Email)”和通过“网页 (Web)”, 你可以选用一个或者都不选或者都选。

3.7 网页通知

网页通知只在 GitHub 上存在, 你也只能在 GitHub 上查看。如果你打开了这个选项并且有一个你的通知, 你会在你屏幕上方的通知图标上看到一个小蓝点。参见[通知中心](#)。

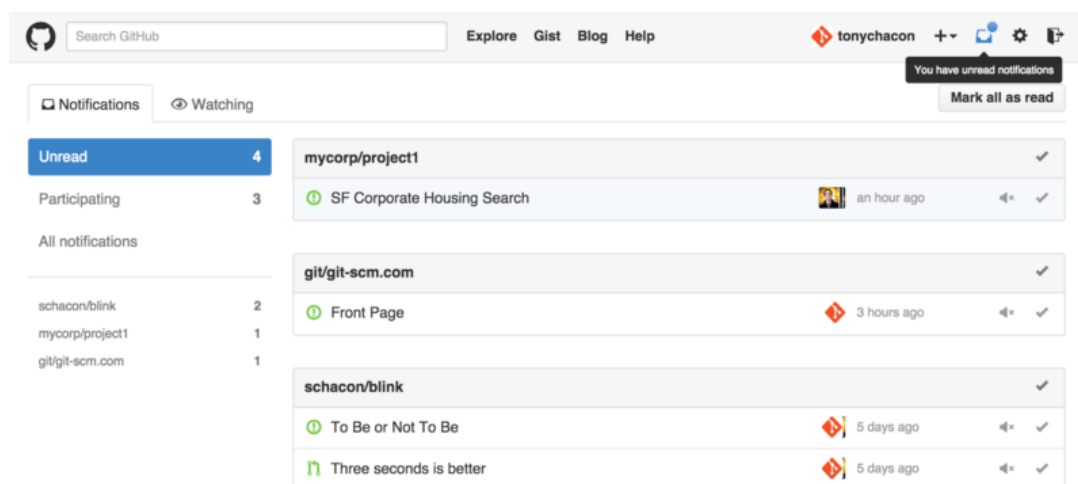


图 41 通知中心

如果你点击那个玩意儿, 你会看到你被通知到的所有条目, 按照项目分好了组。你可以点击左边栏的项目名字来过滤项目相关的通知。你可以点击通知旁边的对号图标把通知标为已读, 或

者点击组上面的图标把项目中**所有的**通知标为已读。在每个对号图标旁边都有一个静音按钮, 你可以点一下, 以后就不会收到它相关的通知。

所有这些工具对于处理大量通知非常有用。很多 GitHub 资深用户都关闭邮件通知, 在这个页面上处理他们所有的通知。

3.8 邮件通知

邮件通知是你处理 GitHub 通知的另一种方式。如果你打开这个选项, 每当有通知时, 你会收到一封邮件。我们在[通过电子邮件发送的评论提醒](#)和[新的合并请求的邮件通知](#)看到了一些例子。邮件也会被合适地按话题组织在一起, 如果你使用一个具有会话功能的邮件客户端那会很方便。

GitHub 在发送给你的邮件头中附带了很多元数据, 这对于设置过滤器和邮件规则非常有帮助。

举个例子, 我们来看一看在[新的合并请求的邮件通知](#)中发给 Tony 的一封真实邮件的头部, 我们会看到下面这些:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...
X-GitHub-Recipient-Address: tchacon@example.com
```

这里有一些有趣的东西。如果你想高亮或者转发这个项目甚至这个合并请求相关的邮件, Message-ID 中的信息会以 `<user>/<project>/<type>/<id>` 的格式展现所有的数据。例如, 如果这是一个问题 (issue), 那么 `<type>` 字段就会是 “issues” 而不是 “pull”。

List-Post 和 List-Unsubscribe 字段表示如果你的邮件客户端能够处理这些, 那么你可以很容易地在列表中发帖或取消对这个相关帖子的订阅。那会很有效率, 就像在页面中点击静音按钮或在问题/合并请求页面点击 “Unsubscribe” 一样。

值得注意的是, 如果你同时打开了邮件和网页通知, 那么当你在邮件客户端允许加载图片的情况下阅读邮件通知时, 对应的网页通知也将会同时被标记为已读。

3.9 特殊文件

如果你的版本库中有一些特殊文件, GitHub 会提醒你。

3.10 README

第一个就是 README 文件, 可以是几乎任何 GitHub 可以识别的格式。例如, 它可以是 README, README.md, README.asciidoc。如果 GitHub 在你的版本库中找到 README 文件, 会把它在

项目的首页渲染出来。

很多团队在这个文件里放版本库或项目新人需要了解的所有相关的信息。它一般包含这些内容：

- 该项目的作用
- 如何配置与安装
- 有关如何使用和运行的例子
- 项目的许可证
- 如何向项目贡献力量

因为 GitHub 会渲染这个文件，你可以在文件里植入图片或链接让它更容易理解。

3.11 贡献 CONTRIBUTING

另一个 GitHub 可以识别的特殊文件是 `CONTRIBUTING`。如果你有一个任意扩展名的 `CONTRIBUTING` 文件，当有人开启一个合并请求时 GitHub 会显示 [开启合并请求时有 CONTRIBUTING 文件存在](#)。

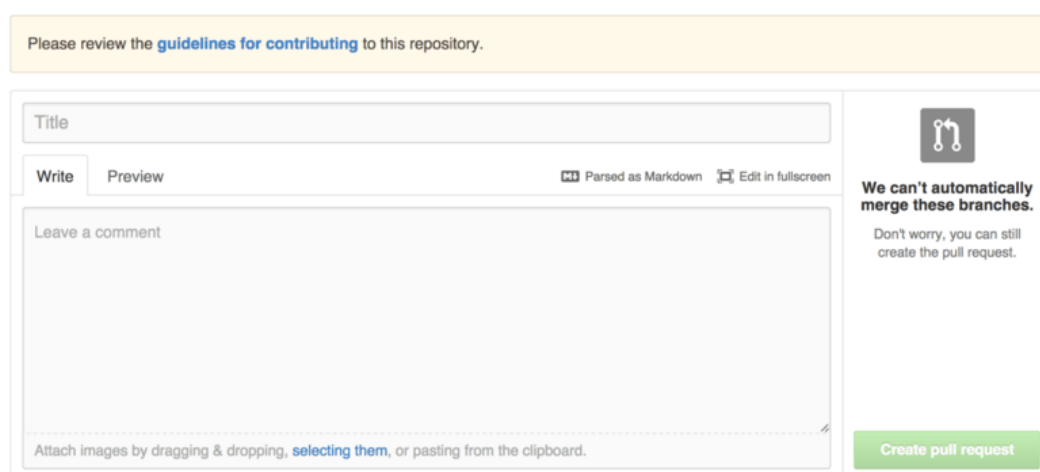


图 42 开启合并请求时有 `CONTRIBUTING` 文件存在

这个的作用就是你可以在这里指出对于你的项目开启的合并请求你想要的/想要的各种事情。这样别人在开启合并请求之前可以读到这些指导方针。

3.12 项目管理

对于一个单个项目其实没有很多管理事务要做，但也有几点有趣的。

3.12.1 改变默认分支

如果你想用 “master” 之外的分支作为你的默认分支，其他人将默认会在这个分支上开启合并请求或进行浏览，你可以在你版本库的设置页面的 “options” 标签下修改。

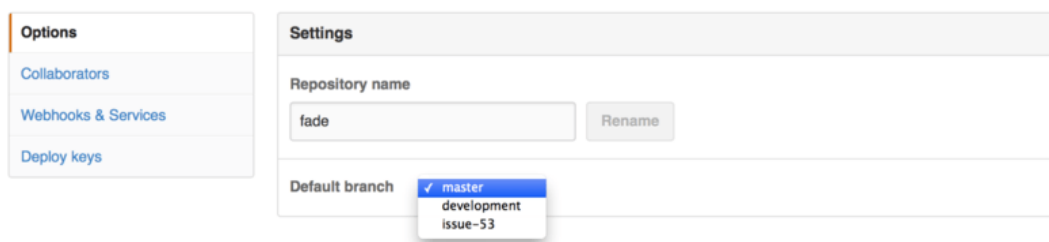


图 43 改变项目的默认分支

简单地改变默认分支下拉列表中的选项，它就会作为所有主要操作的默认分支，他人进行克隆时该分支也将被默认检出。

3.12.2 移交项目

如果你想把一个项目移交给 GitHub 中的另一个人或另一个组织，还是设置页面的这个“options”标签下有一个“Transfer ownership”选项可以用来干这个。

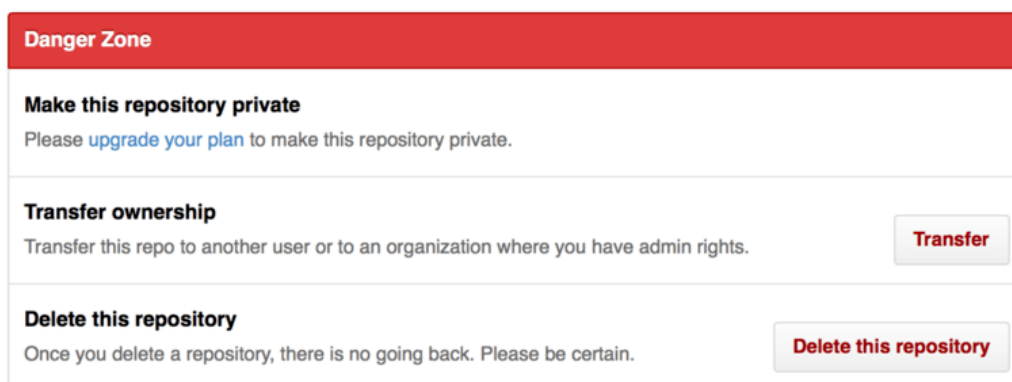


图 44 把项目移交给另一个 GitHub 用户或组织

当你正准备放弃一个项目且正好有别人想要接手时，或者你的项目壮大了想把它移到一个组织里时，这就管用了。

这么做不仅会把版本库连带它所有的观察和星标数都移到另一个地方，它还会将你的 URL 重定向到新的位置。它也重定向了来自 Git 的克隆和抓取，而不仅仅是网页端请求。

4 管理组织

4.1 管理组织

除了个人账户之外，GitHub 还提供被称为组织 (Organizations) 的帐户。组织账户和个人账户一样都有一个用于存放所拥有项目的命名空间，但是许多其他的东西都是不同的。组织帐户代表了一组共同拥有多个项目的人，同时也提供一些工具用于对成员进行分组管理。通常，这种

账户被用于开源群组 (例如: “perl” 或者 “rails”), 或者公司 (例如: “google” 或者 “twitter”)。

4.2 组织的基本知识

我们可以很简单地创建一个组织, 只需要点击任意 GitHub 页面右上角的 “+” 图标, 在菜单中选择 “New organization” 即可。

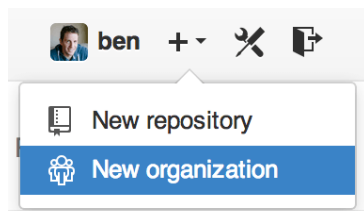


图 45 “New organization” 菜单项

首先你必须提供组织的名称和组织的主要联系邮箱。然后, 如果你希望的话, 也可以邀请其他用户作为共同拥有人。

完成以上步骤后, 你就会拥有一个全新的组织。类似于个人帐户, 如果组织的所有内容都是开源的, 那么你就可以免费使用这个组织。

作为一个组织的拥有者, 当你在派生一个版本库的时候, 你可以选择把它派生到你的组织的命名空间内。当你新建版本库时, 你可以把它存放到你的个人帐户或你拥有的组织内。同时, 你也会自动地 “关注” 所有这些组织内的新版本库。

就像头像, 你可以为你的组织上传头像, 使它更个性化。同时, 也和个人帐户类似, 组织会有一个着陆页 (landing page), 用于列出该组织所有的版本库, 并且该页面可供所有人浏览。

下面我们来说一些组织和个人帐户不同的地方。

4.3 团队

组织使用团队 (Teams) 来管理成员, 团队就是组织中的一组个人账户和版本库, 以及团队成员对这些版本库的访问权限。

例如, 假设你的公司有三个版本库: `frontend`、`backend` 和 `deployscripts`。你会希望你的 HTML/CSS/Javascript 开发者有 `frontend` 或者 `backend` 的访问权限, 操作人员有 `backend` 和 `deployscripts` 的访问权限。团队让这个任务变得更简单, 而不用为每个版本库管理它的协作者。

组织页面主要由一个面板 (dashboard) 构成, 这个仪表盘包含了这个组织内的所有版本库、用户和团队。

你可以点击[组织页面](#)右边的团队侧边栏 (Teams) 来管理你的团队。点击之后, 你会进入一个新页面, 在这里你可以添加新成员和版本库到团队中, 或者管理团队的访问权限和其它设置。每个团队对于版本库可以有只读、读写和管理三种权限。你可以通过点击在[团队页面](#)内的 “Settings” 按钮更改相应权限等级。

当你邀请一个用户加入团队, 该用户会收到一封通知他被邀请的邮件。

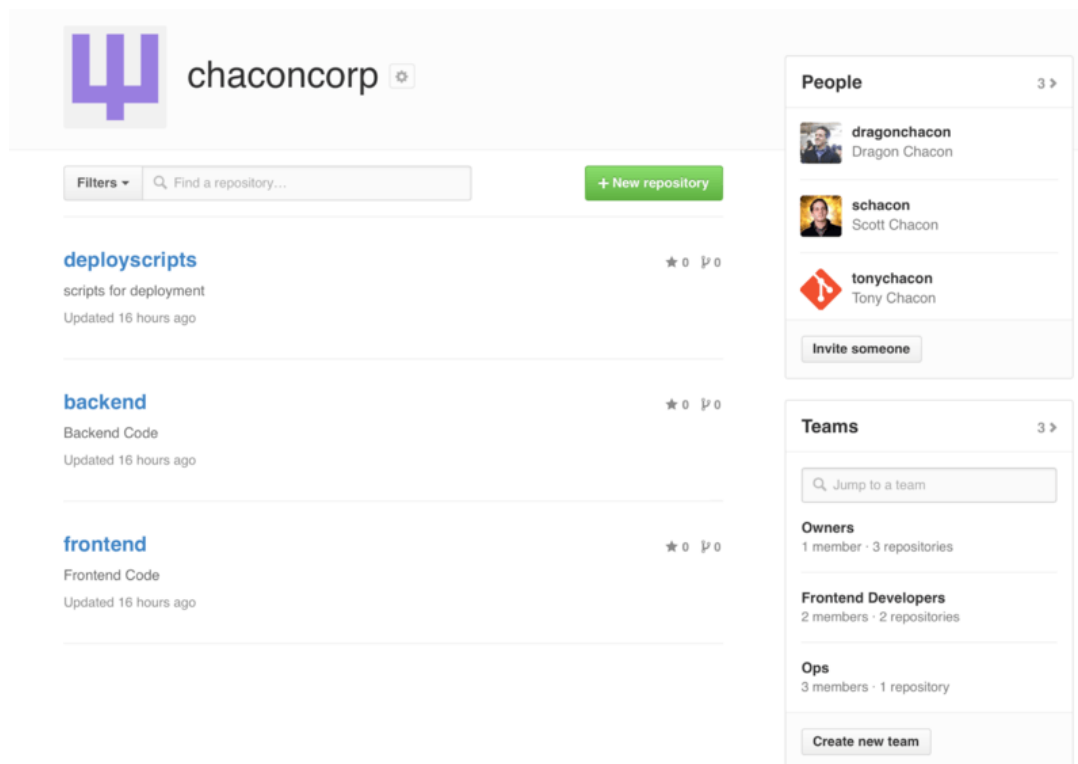


图 46 组织页面

除此之外，团队也类似于个人帐户，有 @mentions (例如: @acmecorp/frontend) 的功能，不同之处就在于被提及的团队内**所有**成员都会成为这个话题的订阅者。当你希望得到团队中某个人的关注，又不知道具体应该问谁的时候，这个功能就显得很有帮助。

一个用户可以加入任意数量的团队，所以别把自己局限于拥有访问控制的团队。对于某一类课题，像 `ux`, `css` 或者 `refactoring` 这样有着特殊关注点的团队就显得很有帮助，而像 `legal` 和 `colorblind` 这样的就完全是针对它们各自领域的。

4.4 审计日志

组织的拥有者还可以访问组织中发生的事情的所有信息。在 **Audit Log** 标签页有整个组织的日志，你可以看到谁在世界上哪个地方做了什么事。

你也可以通过选定某一类型的事件、某个地方、某个人对日志进行过滤。

5 脚本 GitHub

所以现在已经介绍了 GitHub 的大部分功能与工作流程，但是任意一个小组或项目都会去自定义，因为他们想要创造或扩展想要整合的服务。

对我们来说很幸运的是，GitHub 在许多方面都真的很方便 Hack。在本节中我们将会介绍如何使用 GitHub 钩子系统与 API 接口，使 GitHub 按照我们的设想来工作。

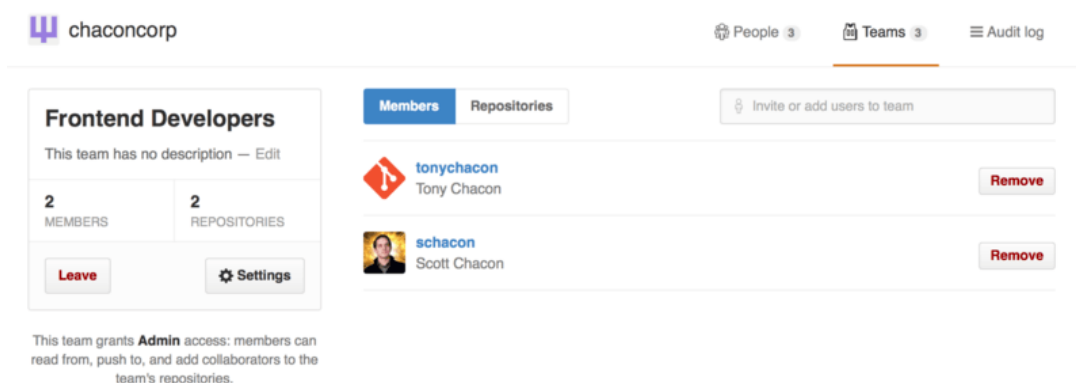


图 47 团队页面

5.1 钩子

GitHub 仓库管理中的钩子与服务区块是 GitHub 与外部系统交互最简单的方式。

5.1.1 服务

首先我们来看一下服务。钩子与服务整合都可以在仓库的设置区块中找到, 就在我们之前添加协作者与改变项目的默认分支的地方。在“Webhooks and Services”标签下你会看到与[服务与钩子配置区域](#)类似的内容。

有许多可以选择的服务, 大多数是整合到其他的商业与开源系统中。它们中的大多数是为了整合持续集成服务、BUG 与问题追踪系统、聊天室系统与文档系统。我们将会通过设置一个非常简单的例子来介绍。如果从“Add Service”选择“email”, 会得到一个类似[电子邮件服务配置](#)的配置屏幕。

在本例中, 如果我们点击“Add service”按钮, 每次有人推送内容到仓库时, 指定的电子邮件地址都会收到一封邮件。服务可以监听许多不同类型的事件, 但是大多数只监听推送事件然后使用那些数据做一些事情。

如果有一个正在使用的系统想要整合到 GitHub, 应当先检查这里看有没有已有的可用的服务整合。例如, 如果正使用 Jenkins 来测试你的代码库, 当每次有人推送到你的仓库时你可以启用 Jenkins 内置的整合启动测试运行。

5.1.2 钩子

如果需要做一些更具体的事, 或者想要整合一个不在这个列表中的服务或站点, 可以转而使用更通用的钩子系统。GitHub 仓库钩子是非常简单的。指定一个 URL 然后 GitHub 在任一期望的事件发生时就会发送一个 HTTP 请求到那个 URL 。

通常做这件事的方式是可以设置一个小的 web 服务来监听 GitHub 钩子请求然后使用收到的数据做一些事情。

为了启用一个钩子, 点击[服务与钩子配置区域](#)中的“Add webhook”按钮。这会将你引导至

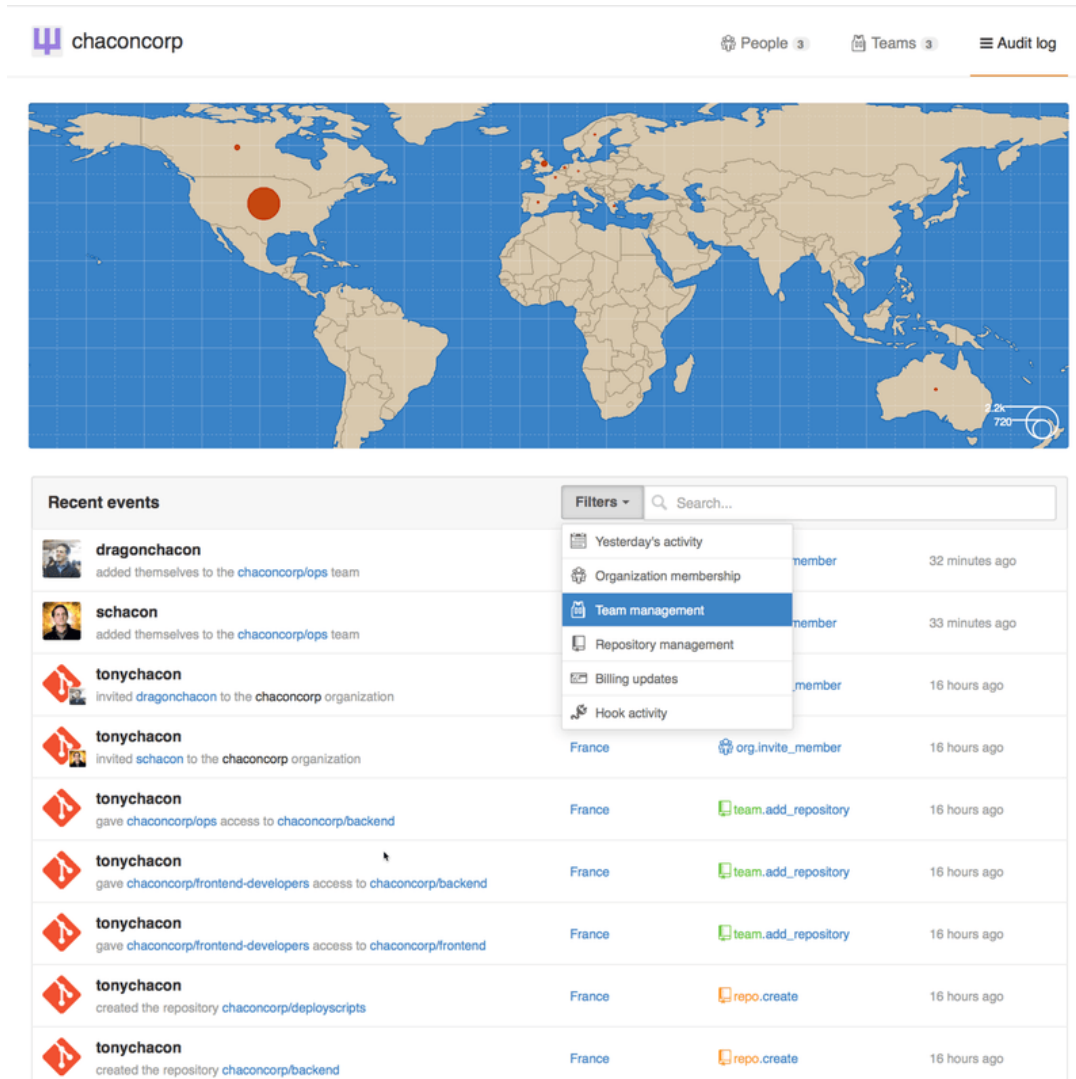


图 48 审计日志

一个类似[Web 钩子配置](#)的页面。

Web 钩子的设置非常简单。大多数情况下只需要输入一个 URL 与一个密钥然后点击“Add webhook”。有几个选项可以指定在哪个事件时想要 GitHub 发送请求——默认的行为是只有当某人推送新代码到仓库的任一分支时的 **push** 事件获得一个请求。

让我们看一个设置处理 web 钩子的 web 服务的小例子。我们将会使用 Ruby web 框架 Sinatra, 因为它相当简洁, 应该能够轻松地看到我们正在做什么。

假设我们想要在某个特定的人推送到我们的项目的特定分支并修改一个特定文件时得到一封邮件。我们可以相当容易地使用类似下面的代码做到:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
```

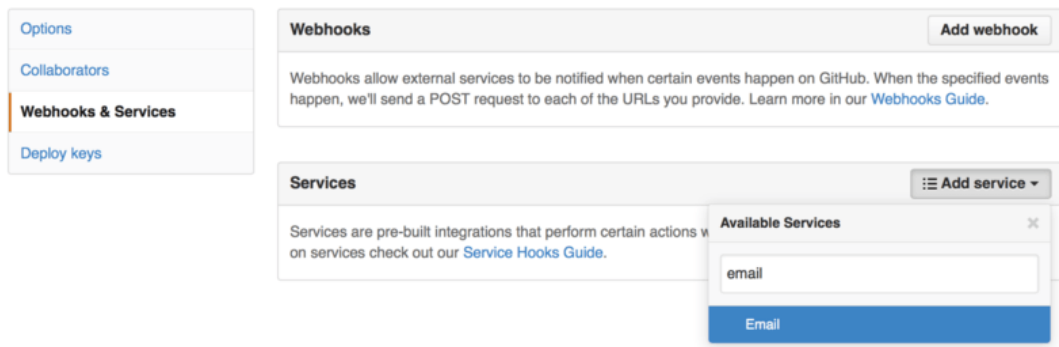


图 49 服务与钩子配置区域

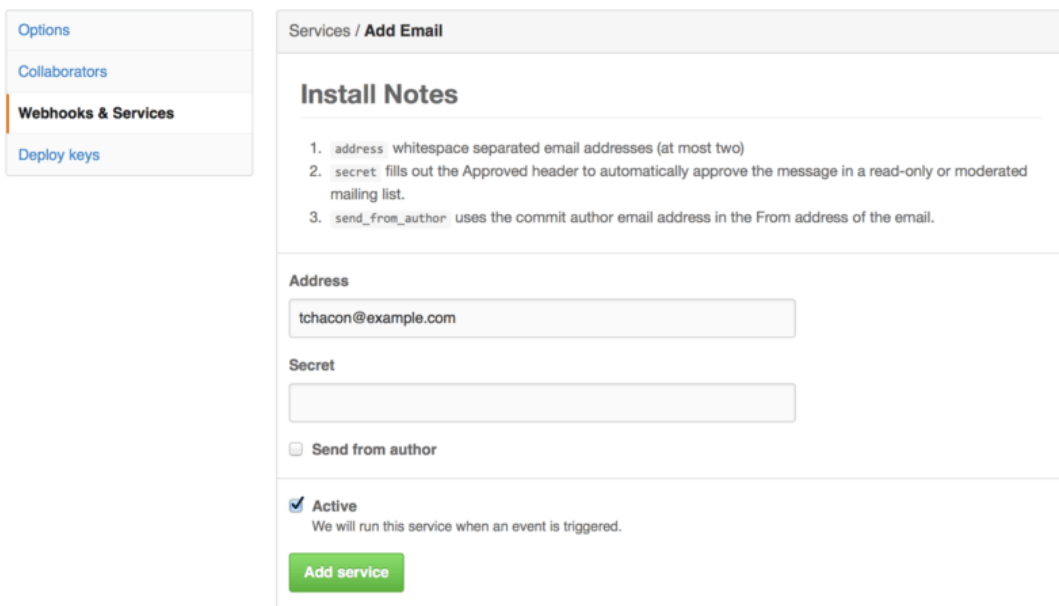


图 50 电子邮件服务配置

```
push = JSON.parse(request.body.read) # parse the JSON

# gather the data we're looking for
pusher = push["pusher"]["name"]
branch = push["ref"]

# get a list of all the files touched
files = push["commits"].map do |commit|
  commit['added'] + commit['modified'] + commit['removed']
end
files = files.flatten.uniq

# check for our criteria
```

Options

Collaborators

Webhooks & Services

Deploy keys

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

https://example.com/postreceive

Content type

application/json

Secret

Which events would you like to trigger this webhook?

☒ Just the push event.

☐ Send me **everything**.

☐ Let me select individual events.

☒ **Active**
We will deliver event details when this hook is triggered.

Add webhook

图 51 Web 钩子配置

```
if pusher == 'schacon' &&
  branch == 'ref/heads/special-branch' &&
  files.include?('special-file.txt')

  Mail.deliver do
    from      'tchacon@example.com'
    to        'tchacon@example.com'
    subject    'Scott Changed the File'
    body      "ALARM"
  end
end
end
```

这里我们拿到一个 GitHub 传给我们的 JSON 请求然后查找推送者，他们推送到了什么分支以及推送的所有提交都改动了哪些文件。然后我们检查它是否与我们的条件匹配，如果匹配则发送一封邮件。

为了开发与测试类似这样的东西，在设置钩子的地方有一个漂亮的开发者控制台。可以看到 GitHub 为那个 webhook 的最后几次请求。对每一个钩子，当它发送后都可以深入挖掘，检测它是否是成功的与请求及回应的消息头与消息体。这使得测试与调试钩子非常容易。

开发者控制台的另一个很棒的功能是可以轻松地重新发送任何请求来测试你的服务。

关于如何编写 web 钩子与所有可监听的不同事件类型的更多信息，请访问在 <https://developer.github.com/webhooks/> 的 GitHub 开发者文档。

Recent Deliveries

4aeae280-4e38-11e4-9bac-c130e992644b

2014-10-07 17:40:41

...

aff20880-4e37-11e4-9089-35319435e08b

2014-10-07 17:36:21

...

90f37680-4e37-11e4-9508-227d13b2ccfc

2014-10-07 17:35:29

...

Request

Response 200

Completed in 0.61 seconds.

Redeliver

Headers

Request URL: https://hooks.example.com/payload

Request method: POST

content-type: application/json

Expect:

User-Agent: GitHub-Hookshot/64a1910

X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc

X-GitHub-Event: push

Payload

```
{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bffa827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffa8...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
    }
  ]
}
```

图 52 Web 钩子调试信息

5.2 GitHub API

服务与钩子给你提供了一种方式来接收关于在仓库中发生的事件的推送通知,但是如何获取相关事件的详情呢?如何自动化一些诸如添加协作者或给问题加标签的事情呢?

这是 GitHub API 派上用场的地方。在自动化流行的趋势下, GitHub 提供了大量的 API 接口,可以进行几乎任何能在网站上进行的操作。在本节中我们将会学习如何授权与连接到 API,如何通过 API 在一个问题上评论与如何修改一个 Pull Request 的状态。

5.3 基本用途

可以做的最基本的事情是向一个不需要授权的接口上发送一个简单的 GET 请求。该接口可能是一个用户或开源项目的只读信息。例如,如果我们想要知道更多关于名为“schacon”的用

38

户信息, 我们可以运行类似下面的东西:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
  # ...
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

有大量类似这样的接口来获得关于组织、项目、问题、提交的信息——差不多就是你能在 GitHub 上看到的所有东西。甚至可以使用 API 来渲染任意 Markdown 或寻找一个 `.gitignore` 模板。

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "/*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

5.4 在一个问题上评论

然而, 如果要在网站上进行一个操作, 如在 Issue 或 Pull Request 上评论, 或者想要查看私有内容或与其交互, 你需要授权。

这里提供了几种授权方式。你可以使用仅需用户名与密码的基本授权, 但是通常更好的主意是使用一个个人访问令牌。可以从设置页的“Applications”标签生成访问令牌。

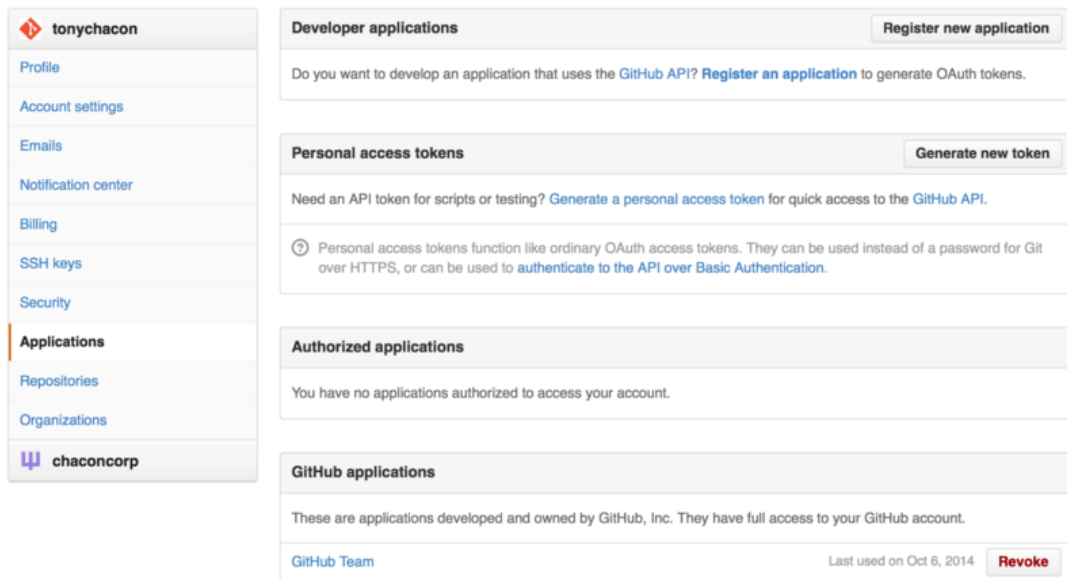


图 53 从设置页的“Applications”标签生成访问令牌

它会询问这个令牌的作用域与一个描述。确保使用一个好的描述信息，这样当脚本或应用不再使用时你会很放心地移除。

GitHub 只会显示令牌一次，所以记得一定要拷贝它。现在可以在脚本中使用它代替使用用户名写密码来授权。这很漂亮，因为可以限制想要做的范围并且令牌是可废除的。

这也会有一个提高频率上限的附加优点。如果没有授权的话，你会被限制在一小时最多发起 60 次请求。如果授权则可以一小时最多发起 5000 次请求。

所以让我们利用它来对我们的其中一个问题进行评论。想要对一个特定问题 Issue #6 留下一条评论，必须使用刚刚生成的令牌作为 Authorization 头信息，发送一个到 `repos/<user>/<repo>/issues/<num>/comments` 的 HTTP POST 请求。

```
$ curl -H "Content-Type: application/json" \
      -H "Authorization: token TOKEN" \
      --data '{"body":"A new comment, :+1:"}' \
      https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
}
```

```
"updated_at": "2014-10-08T07:48:19Z",  
"body": "A new comment, :+1:"  
}
```

现在如果进入到那个问题, 可以看到我们刚刚发布的评论, 像[从 GitHub API 发布的一条评论](#)一样。

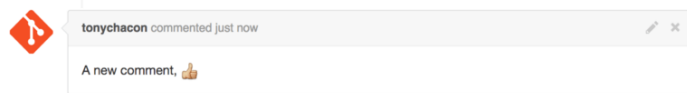


图 54 从 GitHub API 发布的一条评论

可以使用 API 去做任何可以在网站上做的事情——创建与设置里程碑、指派人员到 Issues 与 Pull Requests, 创建与修改标签、访问提交数据、创建新的提交与分支、打开关闭或合并 Pull Requests、创建与编辑团队、在 Pull Request 中评论某行代码、搜索网站等等。

5.5 修改 Pull Request 的状态

如果使用 Pull Requests 的话我们将要看到的最后一个例子会很有用。每一个提交可以有一个或多个与它关联的状态, 有 API 来添加与查询状态。

大多数持续集成与测试服务通过测试推送的代码后使用这个 API 来回应, 然后报告提交是否通过了全部测试。你也可以使用该接口来检查提交信息是否经过合适的格式化、提交者是否遵循了所有你的贡献准则、提交是否经过有效的签名——种种这类事情。

假设在仓库中设置了一个 web 钩子访问一个用来检查提交信息中的 Signed-off-by 字符串的小的 web 服务。

```
require 'httparty'  
require 'sinatra'  
require 'json'  
  
post '/payload' do  
  push = JSON.parse(request.body.read) # parse the JSON  
  repo_name = push['repository']['full_name']  
  
  # look through each commit message  
  push["commits"].each do |commit|  
  
    # look for a Signed-off-by string  
    if /Signed-off-by/.match commit['message']  
      state = 'success'  
      description = 'Successfully signed off!'  
    else  
      state = 'failure'  
      description = 'No signoff found.'    end  
  end  
end
```

```

end

# post status to GitHub
sha = commit["id"]
status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

status = {
  "state"      => state,
  "description" => description,
  "target_url" => "http://example.com/how-to-signoff",
  "context"    => "validate/signoff"
}

HTTParty.post(status_url,
  :body => status.to_json,
  :headers => {
    'Content-Type'  => 'application/json',
    'User-Agent'    => 'tonychacon/signoff',
    'Authorization' => "token #{ENV['TOKEN']}" }
)
end
end

```

希望这相当容易做。在这个 web 钩子处理器中我们浏览刚刚推送上来的每一个提交，在提交信息中查找字符串 **Signed-off-by** 并且最终使用 HTTP 向 `/repos/<user>/<repo>/statuses/<commit_sha>` API 接口发送一个带有状态的 POST 请求。

在本例中可以发送一个状态 (**success, failure, error**)、一个发生了什么的描述信息、一个用户可以了解更多信息的目标 URL 与一个 “context” 以防一个单独的提交有多个状态。例如，一个测试服务可以提供与一个类似这样的验证服务也可能提供一个状态——“context” 字段是用来区别它们的。

如果某人在 GitHub 中打开了一个新的 Pull Request 并且这个钩子已经设置，会看到类似[通过 API 的提交状态](#)的信息。

现在可以看到一个小的绿色对勾标记在提交信息中有 “Signed-off-by” 的提交旁边，红色的对勾标记在作者忘记签名的提交旁边。也可以看到 Pull Request 显示在那个分支上的最后提交的状态，如果失败的话会警告你。如果对测试结果使用这个 API 那么就不会不小心合并某些未通过测试的最新提交。

5.6 Octokit

尽管我们在这些例子中都是通过 curl 与基本的 HTTP 请求来做几乎所有的东西，还有一些以更自然的方式利用 API 的开源库存在着。在写这篇文章的时候，被支持的语言包括 Go、Objective-C、Ruby 与 .NET。访问<http://github.com/octokit>了解更多相关信息，它们帮你

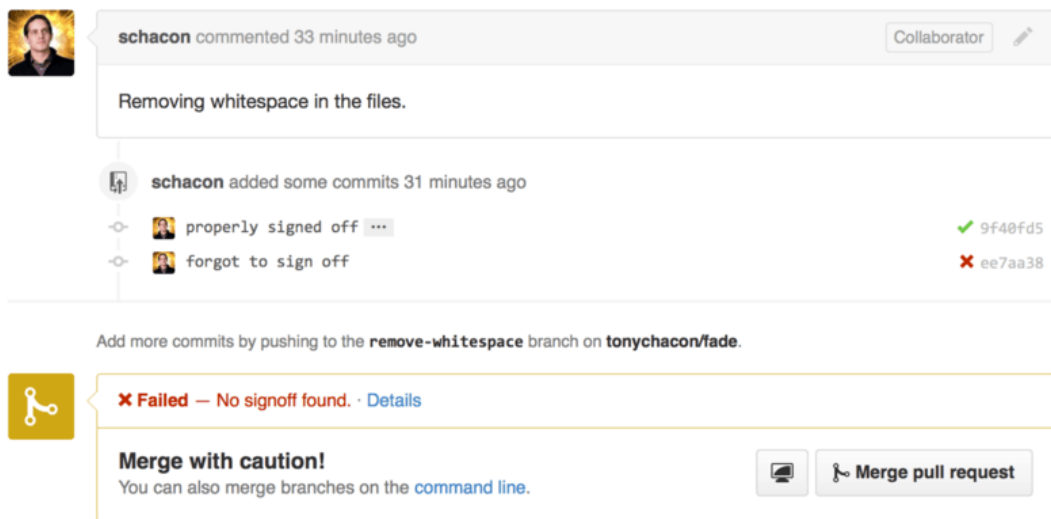


图 55 通过 API 的提交状态

处理了更多 HTTP 相关的内容。

希望这些工具能帮助你自定义与修改 GitHub 来更好地为特定的工作流程工作。关于全部 API 的完整文档与常见任务的指南, 请查阅 <https://developer.github.com>。

6 总结

现在你已经是一名 GitHub 用户了。你知道了如何创建账户、管理组织、创建和推送版本库、向别人的项目提供贡献以及接受别人的贡献。在下一章中, 你将学习更多强有力的工具, 以及处理复杂情况的知识, 这些将使你成为真正的 Git 大师。