Version Control with Git

第2版
涵盖GitHub

Git

版本控制管理

[美] *Jon Loeliger* *Matthew McCullough* 著
王迪 丁彦 等译

O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

内容提要

Git 是一款免费、开源的分布式版本控制系统，最早由 LinilusTorvalds 创建，用于管理 Linux 内核开发，现已成为分布式版本控制的主流工具。

本书是学习掌握 Git 的最佳教程，总共分为 21 章，其内容涵盖了如何在多种真实开发环境中使用 Git；洞察 Git 的常用案例、初始任务和基本功能；如何在集中和分布式版本控制中使用 Git；使用 Git 管理合并、冲突、补丁和差异；获得诸如重新定义变基（rebasing）、钩子（hook）以及处理子模块（子项目）等的高级技巧；Git 如何与 SVN 版本库交互（包括 SVN 向 Git 的转换）；通过 GitHub 导航、使用开源项目，并对开源项目做贡献。

本书适合需要进行版本控制的开发团队成员阅读，对 Git 感兴趣的开发人员也可以从中获益。

译者序

作为分布式版本控制系统中的佼佼者，Git 拥有许多简易但功能强大的操作。人民邮电出版社的编辑邀请我们翻译本书时，正值我们在开发一个名为 GeaKit（集盒）的代码托管项目，于是便愉悦地承接了本书的翻译工作。

作为一个一直使用 Git 作为技术开发版本控制系统的团队，我们对 Git 有着非比寻常的感情。以自身为例，我们团队现在开发的 Dotide 时序数据服务平台，就一直使用 Git 作为我们的版本控制工具。在开发中，Git 帮我们忠实地记录着版本库的历史。无论哪里、何时、是谁出了问题，Git 都可以帮我们甄别是非，迅速定位到问题所在。另外，当我们需要独立开发新功能时，我们也从来不会去担心自己的开发会影响到别人，Git 允许我们在自己的本地库中完成所有的开发，检查无误后再推送给别人，这样就可以随心所欲地处置自己本地的版本库了。

与很多讲述如何使用 Git 的图书不同，本书不但讲解了如何使用 Git，而且更进一步地剖析了 Git 是怎么做到的。也就是说，如果把 Git 比作一种魔法，那么本书不仅教会你如何使用魔法，还掀开了魔法的红盖头。本书内容翔实，章节编排有理、有序、有节，无论你是第一次接触 Git，还是有 Git 使用经验但是不了解其背后的运作机制，本书都会让你收获颇丰。

本书的翻译工作由我们团队成员王迪、丁彦、范乃良、张戈、刘天琴、冷涵、白煜诚共同完成，最后由王迪统稿整理，在此向他们表示感谢。最后，感谢人民邮电出版社在翻译过程中给予的理解和大力支持。

最后，要说的是，由于译者自身水平有限，书中难免出现错误，恳请广大读者批评指正。

GeaKit 团队

2014 年 12 月于南京

前言

本书读者

如果读者有一定的版本控制系统使用经验，再阅读本书是最好不过，当然，如果读者之前没有接触过任何版本控制系统，也可以通过本书在短时间内学会 Git 的基本操作，从而提升工作效率。水平更高的读者通过本书可以洞悉 Git 的内部设计机制，从而掌握更强大的 Git 使用技术。

本书假定读者熟悉并使用过 UNIX shell、基本的 shell 命令，以及通用的编程概念。

假定的框架

本书所有的示例和讨论都假定读者拥有一个带有命令行界面的类 UNIX 系统。本书作者是在 Debian 和 Ubuntu Linux 环境下开发的这些示例。这些示例在其他环境下（比如，Mac OS X 或 Solaris）应该也可以运行，但是可能需要做出微调。

书中有少量示例需要用到 root 权限，此时，你自然应该能清楚理解 root 权限的职责。

本书结构

本书是按照一系列渐进式主题进行组织编排的，每一个主题都建立在之前介绍的概念之上。本书前 11 章讲解的是与一个版本库相关的概念和操作，这些内容是在多个版本库上进行复杂操作（将在本书后 10 章涉及）的基础。

如果你已经安装了 Git，甚至曾经简单使用过，那么你可能用不到前两章中 Git 相关的介绍性知识和安装信息，第 3 章的知识对你来说也是可有可无。

第 4 章介绍的概念是深入掌握 Git 对象模型的基础，读者可以通过第 4 章清楚理解 Git 更为复杂的操作。

第 5 章～第 11 章更为详细地讲解了 Git 的各个主题。第 5 章讲解了索引和文件管理。第 6 章～第 10 章讨论了生成提交和使用提交来形成坚实的开发路线的基础。第 7 章介绍了分支，你可以在一个本地版本库中使用分支操作多条不同的开发路线。第 8 章解释了 diff 的来历和使用。

Git 提供了丰富、强大的功能来加入到开发的不同分支。第 9 章介绍了合并分支和解决分支冲突的基础。对 Git 模型的一个关键洞察力是意识到 Git 执行的所有合并是发生在当前工作目录上下文的本地版本库中的。第 10 章和第 11 章讲解了在开发版本库内进行更改、储藏、跟踪和恢复日常开发的操作。

第 12 章讲解了命名数据以及与另外一个远程版本库交换数据的基础知识。一旦掌握了合并的基础知识，与多个版本库进行交互就变成了一个交换步骤加一个合并步骤的简单组合。交换步骤是第 12 章中新出现的概念，而合并步骤则是在第 9 章讲解的。

第 13 章则从哲学角度对全局的版本库管理提供了抽象的讲解。它还为第 14 章建立了一个环境，使得使用 Git 原生的传输协议无法直接交换版本库信息时，能够打补丁。

接下来的 4 章则涵盖了一些有意思的高级主题：使用钩子（第 15 章）、将项目和多个版本库合并到一个超级项目中（第 16 章），以及与 SVN 版本库进行交互（第 17 章、第 18 章）。

第 19 章和第 20 章提供了一些更为高级的示例和提示、技巧、技术，从而使你成为真正的 Git 大师。
最后，第 21 章介绍了 GitHub，并解释了 Git 如何围绕着版本控制开启了一个有创造力的社会发展进程。
Git 仍然在快速发展，因为当前存在一个活跃的开发团体。这并不是 Git 很不成熟，你无法用它来进行开发；相反，对 Git 的持续改进和用户界面问题正在不断增强。甚至在本书写作的时候，Git 就在不停发展中。因此，如果我不能保持 Git 的准确性，还请谅解。
本书没有完整地覆盖 gitk 命令，尽管本书应该这样做。如果你喜欢以图形方式来呈现版本库中的历史，建议你自行探索 gitk 命令。当前也存在其他历史可视化工具，这些也没有在本书中介绍。本书甚至不能完全涵盖 Git 自带的核心命令和选项。再次向读者道歉！
但是，我仍然希望读者能够从本书中找到足够的线索、提示和方向，从而激励读者自行研究、积极探索 Git。

本书约定

 **提示**

这个图标用来强调一个提示、建议或一般说明。

 **警告**

这个图标用来说明一个警告或注意事项。
此外，读者应该熟悉基本的 shell 命令，以用来操作文件和目录。许多示例会包含一些命令，来完成添加/删除目录、复制文件和创建简单的文件等操作。

```
$ cp file.txt copy-of-file.txt
$ mkdir newdirectory
$ rm file
$ rmdir somedir
$ echo "Test line" > file
$ echo "Another line" >> file
```

需要使用 root 权限来执行的命令会作为一个 sudo 操作出现。

```
# Install the Git core package
$ sudo apt-get install git-core
```

如何在工作目录中编辑文件或效果如何改变则完全取决于你。你应该熟悉一款文本编辑器的用法。本书会通过直接注释或者伪代码的方式来表示文件的编辑过程。

```
# edit file.c to have some new text
$ edit index.html
```

代码示例的使用

本书的目的是为了帮助读者完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码，而且也没有必要取得我们的许可。但是，如果你要复制的是核心代码，则需要和我们打个招呼。例如，你可以在无须获取我们许可的情况下，在程序中使用本书中的多个代码块。但是，销售或分发 O’Reilly 图书中的代码光盘则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书的属性信息是最好不过。一个属性信息通常包括书名、作者、出版社和 ISBN。例如：“Version Control with Git by Jon Loeliger and Matthew McCullough. Copyright 2012 Jon Loeliger, 978-1-449-31638-9.”

在使用书中的代码时，如果不确定是否属于正常使用，或是否超出了我们的许可，请通过 permissions@oreilly.com 与我们联系。

联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社：

美国：

O’Reilly Media Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

关于本书的技术性问题或建议，请发邮件到：

bookquestions@oreilly.com

欢迎登录我们的网站（http://www.oreilly.com），查看更多我们的书籍、课程、会议和最新动态等信息。

Facebook: http://facebook.com/oreilly

Twitter: http://twitter.com/oreillymedia

YouTube: http://www.youtube.com/oreillymedia

致谢

没有众多人士的帮助，本书根本不可能问世。我要感谢 Avery Pennarun 为第 15 章、第 16 章和第 18 章贡献了大量资料，他还贡献了第 4 章和第 9 章的部分内容。感谢他的付出！感谢 Matthew McCullough 对第 17 章和第 21 章贡献的资料，此外还提供了大量的建议和意见。Martin Langhoff 对第 13 章的一些版本库发布建议进行了阐述。Bart Massey 的无须跟踪来保存文件的技巧也出现在了本书中。我要公开感谢在各个阶段花费时间来审阅本书的所有人：Robert P. J. Day、Alan Hasty、Paul Jimenez、Barton Massey、Tom Rix、Jamey Sharp、Sarah Sharp、Larry Streepy、Andy Wilcox 和 Andy Wingo。重点感谢 Robert P. J. Day，他参与审阅了本书的所有版本。

我还要感谢爱妻 Rhonda 和爱女 Brandi、Heather，她们提供了各种支持，包括黑皮诺葡萄酒以及偶尔在语

法上给予提示。还要感谢我的长矛猎犬 Mylo，在我写作期间，它一直温柔地蜷缩在我的怀中。
最后，感谢 O’Reilly 的全体工作人员以及编辑 Andy Oram 和 Martin Streicher。

第 1 章 介绍

1.1 背景

现如今，难以想象有创意的人会在没有备份策略的情况下启动一个项目。数据是短暂的，且容易丢失—例如，通过一次错误的代码变更或者一次灾难性的磁盘崩溃。所以说，在整个工作中持续性地备份和存档是非常明智的。

对于文本和代码项目，备份策略通常包括版本控制，或者叫“对变更进行追踪管理”。每个开发人员每天都会进行若干个变更。这些持续增长的变更，加在一起可以构成一个版本库，用于项目描述，团队沟通和产品管理。版本控制具有举足轻重的作用，只要定制好工作流和项目目标，版本控制是最高效的组织管理方式。

一个可以管理和追踪软件代码或其他类似内容的不同版本的工具，通常称为：版本控制系统（VCS），或者源代码管理器（SCM），或者修订控制系统（RCS），或者其他各种和“修订”、“代码”、“内容”、“版本”、“控制”、“管理”和“系统”等相关的叫法。尽管各个工具的作者和用户常常争论得喋喋不休，但是其实每个工具都出于同样的目的：开发以及维护开发出来的代码、方便读取代码的历史版本、记录所有的修改。在本书中，“版本控制系统”（VCS）一词就是泛指一切这样的工具。

本书主要介绍 Git 这款功能强大、灵活而且低开销的 VCS，它可以让协同开发成为一种乐趣。Git 由 Linus Torvalds 发明，起初是为了方便管理 Linux[1]内核的开发工作。如今，Git 已经在大量的项目中得到了非常成功的应用。

1.2 Git 的诞生

通常来说，当工具跟不上项目需求时，开发人员就会开发一个新的工具。实际上，在软件领域里，创造新工具经常看似简单和诱人。然而，鉴于市面上已经有了相当多的 VCS，决定再创造一个却应该是要深思熟虑的。不过，如果有着充分的需求、理性的洞察以及良好的动机，则完全可以创造一个新的 VCS。Git 就是这样一个 VCS。它被它的创造者（Linus，一个脾气急躁又经常爆出冷幽默的人）称作“从地狱来的信息管理工具”。尽管 Linux 社区内部政治性的争论已经淹没了关于 Git 诞生的情形和时机的记忆，但是毋庸置疑，这个从烈火中诞生的 VCS 着实设计优良，能够胜任世界范围内大规模的软件开发工程。

[1] Linux 是 Linus Torvalds 在美国和其他国家的注册商标。—原注

在 Git 诞生之前，Linux 内核开发过程中使用 BitKeeper 来作为 VCS。BitKeeper 提供当时的一些开源 VCS（如 RCS、CVS）所不能提供的高级操作。然而，在 2005 年春天，当 BitKeeper 的所有方对他们的免费版 BitKeeper 加入了额外的限制时，Linux 社区意识到，使用 BitKeeper 不再是一个长期可行的解决方案。
Linus 本人开始寻找替代品。这次，他回避使用商业解决方案，在自由软件包中寻找。然而，他却发现，在现有的自由软件解决方案中，那些在选择 BitKeeper 之前曾经发现的，导致他放弃自由软件解决方案的一些限制和缺陷如今依然存在。那么，这些已经存在的 VCS 到底存在什么缺陷？Linus 没能在现有 VCS 中找到的有关特性到底是哪些？让我们来看看。

有助于分布式开发

分布式开发有很多方面，Linus 希望有一个新的 VCS 能够尽可能覆盖这些方面。它必须允许并行开发，各人可以在自己的版本库中独立且同时地开发，而不需要与一个中心版本库时刻同步（因为这样会造成开发瓶颈）。它必须允许许多开发人员在不同的地方，甚至是离线的情况下，无障碍地开发，

能够胜任上千开发人员的规模

仅仅支持分布式开发模型还是不够的。Linus 深知，每个 Linux 版本都凝聚了数以千计开发人员的心血。所以新的 VCS 必须能够很好地支持非常多的开发人员，无论这些开发人员工作在整个项目相同还是不同的部分。当然，新的 VCS 也必须能够可靠地将这些工作整合起来。

性能优异

Linus 决心要确保新的 VCS 能够快速并且高效地执行。为了支持 Linux 内核开发中大量的更新操作，他知道不管是个人的更新操作，还是网络传输操作，都需要保证执行速度。为了节约存储空间，从而节约传输时间，需要使用“压缩”和“差异比较”技术。另外，使用分布式开发模型，而非集中式模型，同样也确保了网络的不确定因素不会影响到日常开发的效率。

保持完整性和可靠性

因为 Git 是一个分布式版本控制系统，所以非常需要能够绝对保证数据的完整性和不会被意外修改。那如何确定，在从一个开发人员到另一个开发人员的过程中，或者从一个版本库到另一个版本库的过程中，数据没有被意外修改呢？又如何确定版本库中的实际数据就是认为的那样？
Git 使用一个叫做“安全散列函数”（SHA1）的通用加密散列函数，来命名和识别数据库中的对象。虽然也许理论上不是绝对的，但是在实践中，已经证实这是足够可靠的方式。

强化责任

版本控制系统的一个关键方面，就包括能够定位谁改动了文件，甚至改动的原因。Git 对每一个有文件改动的提交（Git 把一个历史版本叫做一个“提交”）强制使用“改动日志”。“改动日志”中存储的信息

由开发人员、项目需求、管理策略等决定。Git 确保被 VCS 管理的文件不会被莫名地修改，因为 Git 可以对所有的改动进行责任追踪。

不可变性

Git 版本库中存储的数据对象均为不可变的。这意味着，一旦创建数据对象并把它们存放到数据库中，它们便不可修改。当然，它们可以重新创建，但是重新创建只是产生新的数据对象，原始数据对象并不会被替换。Git 数据库的设计同时也意味着存储在版本数据库中的整个历史也是不可变的。使用不可变的对象有诸多优势，包括快速比较相同性。

原子事务

有了原子事务，可以让一系列不同但是相关的操作要么全部执行要么一个都不执行。这个特性可以确保在进行更新或者提交操作时，版本数据库不会陷入部分改变或者破损的状态。Git 通过记录完整、离散的版本库状态来实现原子事务。而这些版本库状态都无法再分解成更小的独立状态。

支持并且鼓励基于分支的开发

几乎所有的 VCS 都支持在同一个项目中存在多个“支线”。例如，代码变更的一条支线叫做“开发”，而同时又存在另一条支线叫做“测试”。每个 VCS 同样可以将一条支线分叉为多条支线，在以后再将差异化后的支线合并。就像大多数 VCS 一样，Git 把这样的支线叫做“分支”，并且给每个分支都命名。伴随着分支的就是合并。Linus 不仅希望通过简单的分支功能来促进丰富的开发分支，还希望这些分支的合并可以变得简单容易。因为通常来说，分支的合并是各 VCS 使用中最为困难和痛苦的操作，所以，能够提供一个简单、清晰、快速的合并功能，是非常必要的。

完整的版本库

为了让各个开发人员不需要查询中心服务器就可以得到历史修订信息，每个人的版本库中都有一份关于每个文件的完整历史修订信息就非常重要。

一个清晰的内部设计

即使最终用户也许并不关心是否有一个清晰的内部设计，对于 Linus 以及其他 Git 开发人员来说，这确实非常重要。Git 的对象模型拥有者简单的结构，并且能够保存原始数据最基本的部分和目录结构，能够记录变更内容等。再将这个对象模型和全局唯一标识符技术相结合，便可以得到一个用于分布式开发环境中的清晰数据对象。

免费自由（Be free, as in freedom）

—Nuff 曾说过。

有了创造一个新 VCS 的清晰理由后，许多天才软件工程师一起创作出了 Git。需求是创新之母！

1.3 先例

VCS 的完整历史已经超出了本书的讨论范围。然而，有一些具有里程碑、革新意义的系统值得一提。这些系统对 Git 的开发或者有重要的铺垫意义，或者有引导意义。（本节为可选章节，希望能够记录那些新特性出现的时间，以及在自由软件社区变得流行的时间。）

源代码控制系统（Source Code Control System, SCCS）是 UNIX[2]上最初的几个系统之一，由 M. J. Rochkind 于 20 世纪 70 年代早期开发。[“The Source Code Control System,” *IEEE Transactions on Software Engineering* 1(4) (1975): 364-370.]这是有证可查的可以运行在 UNIX 系统上的最早的 VCS。

SCCS 提供的数据存储中心称为“版本库”（repository），而这个基本概念一直沿用至今。SCCS 同样提供了一个简单的锁模型来保证开发过程有序。如果一个开发人员需要运行或者测试一个程序，他需要将该程序解锁并检出。然而，如果他想修改某个文件，他则需要锁定并检出（通过 UNIX 文件系统执行的转换）。当编辑完成以后，他又可以将文件检入到版本库中并解锁它。

修订控制系统（Revision Control System，RCS）由 Walter F. Tichy 于 20 世纪 80 年代早期引入[“RCS: A System for Version Control,” *Software Practice and Experience* 15(7) (1985): 637-654.]。RCS 引入了双向差异的概念，来提高文件不同版本的存储效率。

并行版本系统（Concurrent Version System，CVS）由 Dick Grune 于 1986 年设计并最初实现。4 年后又被 Berliner 和他的团队融入 RCS 模型重新实现，这次实现非常成功。CVS 变得非常流行，并且成为开源社（http:www.opensource.org）区许多年的事实标准。CVS 相对 RCS 有多项优势，包括分布式开发和版本库范围内对整个“模块”的更改集。

此外，CVS 引入了一个关于“锁”的新范式。而之前的系统需要开发人员在修改某个文件之前先锁定它，一个文件同时只允许一个开发人员进行修改，所有需要修改这个文件的开发人员需要有序等候。CVS 给予每个开发人员对于自己的私有版本写的权限。因此，不同开发人员的改动可以自动合并，除非两个开发人员尝试修改同一行。如果出现修改同一行的情况，那这一行将会作为“冲突”被标记出来，由开发人员手动去解决。这个关于“锁”的新规则使得多个开发人员可以并行地编写代码。

就像经常发生的那样，对 CVS 短处和缺点的改进，促进了新 VCS 的诞生：Subversion（SVN）。SVN 于 2001 年问世，迅速风靡了开源社区。不像 CVS，SVN 以原子方式提交改动部分，并且更好地支持分支。BitKeeper 和 Mercurial 则彻底抛弃了上述所有解决方案。它们淘汰了中心版本库的概念，取而代之的，数据的存储是分布式的，每个开发人员都拥有自己可共享的版本库副本。Git 则是从这种端点对端点（Peer to Peer）的模型继承而来。

最后，Mercurial 和 Monotone 首创了用散列指纹来唯一标识文件的内容，而文件名只是个“绰号”，旨在方便用户操作，再没有别的作用。Git 沿用了这个概念。从内部实现上来说，Git 的文件标识符基于文件

[2] UNIX 是 Open Group 在美国和其他国家的注册商标。—原注

的内容，这是一个叫做“内容可寻址文件存储”（Content Addressable File Store，CAFS）的概念。这不是一个新概念。见“The Venti Filesystem,” (Plan 9), Bell Labs, [http://www.usenix.org/events/fast02/quinlan/quinlan_html/index.html.] 据 Linus 的说法[3]，Git 直接从 Monotone 借用了这个概念。Mercurial 也同时实现了这个概念。

1.4 时间线

有了应用场景，有了一点额外的动力，再加上对新 VCS 的需求迫在眉睫，Git 于 2005 年 4 月诞生了。
4 月 7 日，Git 从以下提交起，正式成为自托管项目。

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:    Thu Apr 7 15:13:13 2005 -0700
Initial revision of "git", the information manager from hell
```

不久之后，Linux 内核的第一个提交也诞生了。

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:    Sat Apr 16 15:20:36 2005 -0700
Linux-2.6.12-rc2
Initial git repository build. I'm not bothering with the full history,
even though we have it. We can create a separate "historical" git
archive of that later if we want to, and in the meantime it's about
3.2GB when imported into git - space that would just make the early
git days unnecessarily complicated, when we don't have a lot of good
infrastructure for it.
Let it rip!
```

这一次提交将整个 Linux 内核导入 Git 版本库中[4]。这次提交的统计信息如下：

```
17291 files changed, 6718755 insertions(+), 0 deletions(-)
```

是的，这次提交足足引入了 670 万行代码！
仅仅过了 3 个小时，Linux 内核第一次用 Git 打上了补丁。Linus 在 2005 年 4 月 20 日向 Linux 内核邮件列表正式宣布，用上 Git 了！

---

[3] 私人电子邮件。—原注

[4] 关于旧的 Bitkeeper 日志如何导入 Git 版本库（2.5 之前）的悠久历史的起点，参见 http://kerneltrap. org/node/13996。—原注

Linus 非常清楚自己想回到 Linux 内核的开发中去，所以，在 2005 年 7 月 25 日，Linus 将 Git 代码维护工作交接给 Junio Hamano。并声称，Junio 是显而易见的选择。
大概两个月以后，版本号为 2.6.12 的 Linux 内核正式发布，所用 VCS 正是 Git。

1.5 名字有何含义

据 Linus 宣称，命名为 Git，是因为“我是一个自私的混蛋，我照着自己命名我所有的项目，先是 Linux，现在是 git”[5]。倘若，Linux 这个名字是 Linus 和 Minix 的某种结合。那么反用一个表示愚蠢无用之人的英语词汇也不是没可能。
那之后，也有人曾建议，使用一些其他也许更让人舒服的解释。其中最受欢迎的一个就是：全局信息追踪器（Global Information Tracker）。

[5] 参见 http://www.infoworld.com/article/05/04/19/HNtorvaldswork_1.html。—原注

第 4 章 基本的 Git 概念

4.1 基本概念

前一章介绍了 Git 的一个典型应用，并且可能引发了相当多的问题。Git 是否在每次提交时存储整个文件？.git 目录的目的是什么？为什么一个提交 ID 像乱码？我应该注意它吗？
如果你用过其他 VCS，比如 SVN 或者 CVS，那么对最后一章的命令可能会很熟悉。事实上，你对一个现代 VCS 期望的所有操作和功能，Git 都能提供。然而，在一些基本的和意想不到的方面，Git 会有所不同。

本章会通过讨论 Git 的关键架构组成和一些重要概念来探讨 Git 的不同之处和原因。这里注重基础知识并且演示如何与一个版本库交互；第 12 章会介绍如何操作很多关联的版本库。追踪多个版本库可能看起来是个艰巨的任务，但是你在本章学到的基本原则是一样适用的。

4.1.1 版本库

Git 版本库（repository）只是一个简单的数据库，其中包含所有用来维护与管理项目的修订版本和历史的信息。在 Git 中，跟大多数版本控制系统一样，一个版本库维护项目整个生命周期的完整副本。然而，不同于其他大多数 VCS，Git 版本库不仅仅提供版本库中所有文件的完整副本，还提供版本库本身的副本。
Git 在每个版本库里维护一组配置值。在前面的章节你已经见过其中的一些了，比如，版本库的用户名和 email 地址。不像文件数据和其他版本库的元数据，在把一个版本库克隆（clone）或者复制到另一个版本库的时候配置设置是不跟着转移的。相反，Git 对每个网站、每个用户和每个版本库的配置和设置信息都进行管理与检查。
在版本库中，Git 维护两个主要的数据结构：对象库（object store）和索引（index） 。所有这些版本库数据存放在工作目录根目录下一个名为*.git* 的隐藏子目录中。
对象库在复制操作的时候能进行有效复制，这也是用来支持完全分布式 VCS 的一种技术。索引是暂时的信息，对版本库来说是私有的，并且可以在需要的时候按需求进行创建和修改。
接下来的两节将对对象库和索引进行更详细的描述。

4.1.2 Git 对象类型

对象库是 Git 版本库实现的心脏。它包含你的原始数据文件和所有日志消息、作者信息、日期，以及其他用来重建项目任意版本或分支的信息。
Git 放在对象库里的对象只有 4 种类型：块（blob）、目录树（tree）、提交（commit）和标签（tag）。
这 4 种原子对象构成 Git 高层数据结构的基础。

块（blob）

文件的每一个版本表示为一个块（blob）。blob 是“二进制大对象”（binary large object）的缩写，是计算机领域的常用术语，用来指代某些可以包含任意数据的变量或文件，同时其内部结构会被程序忽略。一个 blob 被视为一个黑盒。一个 blob 保存一个文件的数据，但不包含任何关于这个文件的元数据，甚至连文件名也没有。

目录树（tree）

一个目录树（tree）对象代表一层目录信息。它记录 blob 标识符、路径名和在一个目录里所有文件的一些元数据。它也可以递归引用其他目录树或子树对象，从而建立一个包含文件和子目录的完整层次结构。

提交（commit）

一个提交（commit）对象保存版本库中每一次变化的元数据，包括作者、提交者、提交日期和日志消息。每一个提交对象指向一个目录树对象，这个目录树对象在一张完整的快照中捕获提交时版本库的状态。最初的提交或者根提交（root commit）是没有父提交的。大多数提交都有一个父提交，虽然本书后面（第 9 章）会介绍一个提交如何引用多个父提交。

标签（tag）

一个标签对象分配一个任意的且人类可读的名字给一个特定对象，通常是一个提交对象。虽然 9da581d910c9c4ac93557ca4859e767f5caf5169 指的是一个确切且定义好的提交，但是一个更熟悉的标签名（如 Ver-1.0-Alpha）可能会更有意义！

随着时间的推移，所有信息在对象库中会变化和增长，项目的编辑、添加和删除都会被跟踪和建模。为了有效地利用磁盘空间和网络带宽，Git 把对象压缩并存储在打包文件（pack file）里，这些文件也在对象库里。

4.1.3 索引

索引是一个临时的、动态的二进制文件，它描述整个版本库的目录结构。更具体地说，索引捕获项目在某个时刻的整体结构的一个版本。项目的状态可以用一个提交和一棵目录树表示，它可以来自项目历史中的任意时刻，或者它可以是你正在开发的未来状态。

Git 的关键特色之一就是它允许你用有条理的、定义好的步骤来改变索引的内容。索引使得开发的推进与提交的变更之间能够分离开来。

下面是它的工作原理。作为开发人员，你通过执行 Git 命令在索引中暂存（stage）变更。变更通常是添加、删除或者编辑某个文件或某些文件。索引会记录和保存那些变更，保障它们的安全直到你准备好提交了。还可以删除或替换索引中的变更。因此，索引支持一个由你主导的从复杂的版本库状态到一个可推测的更好状态的逐步过渡。

在第 9 章中，你会看到索引在合并（merge），允许管理、检查和同时操作同一个文件的多个版本中起到

的重要作用。

4.1.4 可寻址内容名称

Git 对象库被组织及实现成一个内容寻址的存储系统。具体而言，对象库中的每个对象都有一个唯一的名称，这个名称是向对象的内容应用 SHA1 得到的 SHA1 散列值。因为一个对象的完整内容决定了这个散列值，并且认为这个散列值能有效并唯一地对应特定的内容，所以 SHA1 散列值用来做对象数据库中对象的名字和索引是完全充分的。文件的任何微小变化都会导致 SHA1 散列值的改变，使得文件的新版本被单独编入索引。

SHA1 的值是一个 160 位的数，通常表示为一个 40 位的十六进制数，比如，9da581d910c9c4ac93557ca4859e767f5caf5169。有时候，在显示期间，SHA1 值被简化成一个较小的、唯一的前缀。Git 用户所说的 SHA1、散列码和对象 ID 都是指同一个东西。

> **全局唯一标识符**
>
> SHA 散列计算的一个重要特性是不管内容在哪里，它对同样的内容始终产生同样的 ID。换言之，在不同目录里甚至不同机器中的相同文件内容产生的 SHA1 哈希 ID 是完全相同的。因此，文件的 SHA1 散列 ID 是一种有效的全局唯一标识符。
>
> 这里有一个强大的推论，在互联网上，文件或者任意大小的 blob 都可以通过仅比较它们的 SHA1 标识符来判断是否相同。

4.1.5 Git 追踪内容

理解 Git 不仅仅是一个 VCS 是很重要的，Git 同时还是一个内容追踪系统（content tracking system）。这种区别尽管很微小，但是指导了 Git 的很多设计，并且也许这就是处理内部数据操作相对容易的关键原因。然而，因为这也可能是对新手来讲最难把握的概念之一，所以做一些论述是值得的。

Git 的内容追踪主要表现为两种关键的方式，这两种方式与大多数其他[1]修订版本控制系统都不一样。

首先，Git 的对象库基于其对象内容的散列计算的值，而不是基于用户原始文件布局的文件名或目录名设置。因此，当 Git 放置一个文件到对象库中的时候，它基于数据的散列值而不是文件名。事实上，Git 并不追踪那些与文件次相关的文件名或者目录名。再次强调，Git 追踪的是内容而不是文件。

如果两个文件的内容完全一样，无论是否在相同的目录，Git 在对象库里只保存一份 blob 形式的内容副本。Git 仅根据文件内容来计算每一个文件的散列码，如果文件有相同的 SHA1 值，它们的内容就是相同的，然后将这个 blob 对象放到对象库里，并以 SHA1 值作为索引。项目中的这两个文件，不管它们在用户的目录结构中处于什么位置，都使用那个相同的对象指代其内容。

如果这些文件中的一个发生了变化，Git 会为它计算一个新的 SHA1 值，识别出它现在是一个不同的 blob 对象，然后把这个新的 blob 加到对象库里。原来的 blob 在对象库里保持不变，为没有变化的文件所

---

[1] Monotone、Mercurial、OpenCMS 和 Venti 是一些值得注意的例外。—原注

使用。
其次，当文件从一个版本变到下一个版本的时候，Git 的内部数据库有效地存储每个文件的每个版本，而不是它们的差异。因为 Git 使用一个文件的全部内容的散列值作为文件名，所以它必须对每个文件的完整副本进行操作。Git 不能将工作或者对象库条目建立在文件内容的一部分或者文件的两个版本之间的差异上。
文件拥有修订版本和从一个版本到另一个版本的步进，用户的典型看法是这种文件简直是个工艺品。
Git 用不同散列值的 blob 之间的区别来计算这个历史，而不是直接存储一个文件名和一系列差异。这似乎有些奇怪，但这个特性让 Git 在执行某些任务的时候非常轻松。

4.1.6 路径名与内容

跟很多其他 VCS 一样，Git 需要维护一个明确的文件列表来组成版本库的内容。然而，这个需求并不需要 Git 的列表基于文件名。实际上，Git 把文件名视为一段区别于文件内容的数据。这样，Git 就把索引从传统数据库的数据中分离出来了。看看表 4-1 会很有帮助，它粗略地比较了 Git 和其他类似的系统。

表 4-1 数据库对比

系统	索引机制	数据存储
传统数据库	索引顺序存取方法（ISAM）	数据记录
UNIX 文件系统	目录（*/path/to/file*）	数据块
Git	*.git/objects/hash*、树对象内容	blob 对象、树对象

文件名和目录名来自底层的文件系统，但是 Git 并不真正关心这些名字。Git 仅仅记录每个路径名，并且确保能通过它的内容精确地重建文件和目录，这些是由散列值来索引的。
Git 的物理数据布局并不模仿用户的文件目录结构。相反，它有一个完全不同的结构却可以重建用户的原始布局。在考虑其自身的内部操作和存储方面，Git 的内部结构是一种更高效的数据结构。
当 Git 需要创建一个工作目录时，它对文件系统说：“嘿！我这有这样大的一个 blob 数据，应该放在路径名为 *path/to/directory/file* 的地方。你能理解吗？”文件系统回复说：“啊，是啊，我认出那个字符串是一组子目录名，并且我知道把你的 blob 数据放在哪里！谢谢！”

4.1.7 打包文件

一个聪明的读者也许已经有了关于 Git 的数据模型及其单独文件存储的挥之不去的问题：直接存储每个文件每个版本的完整内容是否太低效率了？即使它是压缩的，把相同文件的不同版本的全部内容都存储的效率是否太低了？如果你只添加一行到文件里，Git 是不是要存储两个版本的全部内容？
幸运的是，答案是“不是，不完全是！”
相反，Git 使用了一种叫做 打包文件（pack file） 的更有效的存储机制。要创建一个打包文件，Git 首先定位内容非常相似的全部文件，然后为它们之一存储整个内容。之后计算相似文件之间的差异并且只存储差异。例如，如果你只是更改或者添加文件中的一行，Git 可能会存储新版本的全部内容，然后记录那一行更改作为差异，并存储在包里。
存储一个文件的整个版本并存储用来构造其他版本的相似文件的差异并不是一个新伎俩。这个机制已经被其他 VCS（如 RCS）用了好几十年了，它们的方法本质上是相同的。

然而，Git 文件打包得非常巧妙。因为 Git 是由内容驱动的，所以它并不真正关心它计算出来的两个文件之间的差异是否属于同一个文件的两个版本。这就是说，Git 可以在版本库里的任何地方取出两个文件并计算差异，只要它认为它们足够相似来产生良好的数据压缩。因此，Git 有一套相当复杂的算法来定位和匹配版本库中潜在的全局候选差异。此外，Git 可以构造一系列差异文件，从一个文件的一个版本到第二个，第三个，等等。

Git 还维护打包文件表示中每个完整文件（包括完整内容的文件和通过差异重建出来的文件）的原始 blob 的 SHA1 值。这给定位包内对象的索引机制提供了基础。

打包文件跟对象库中其他对象存储在一起。它们也用于网络中版本库的高效数据传输。

4.2 对象库图示

让我们看看 Git 的对象之间是如何协作来形成完整系统的。

blob 对象是数据结构的“底端”；它什么也不引用而且只被树对象引用。在接下来的图里，每个 blob 由一个矩形表示。

树对象指向若干 blob 对象，也可能指向其他树对象。许多不同的提交对象可能指向任何给定的树对象。每个树对象由一个三角形表示。

一个圆圈表示一个提交对象。一个提交对象指向一个特定的树对象，并且这个树对象是由提交对象引入版本库的。

每个标签由一个平行四边形表示。每个标签可以指向最多一个提交对象。

分支不是一个基本的 Git 对象，但是它在命名提交对象的时候起到了至关重要的作用。把每个分支画成一个圆角矩形。

图 4-1 展示了所有部分如何协作。这张图显示了一个版本库在添加了两个文件的初始提交后的状态。两个文件都在顶级目录中。同时它们的 master 分支和一个叫 V1.0 的标签都指向 ID 为 1492 的提交对象。

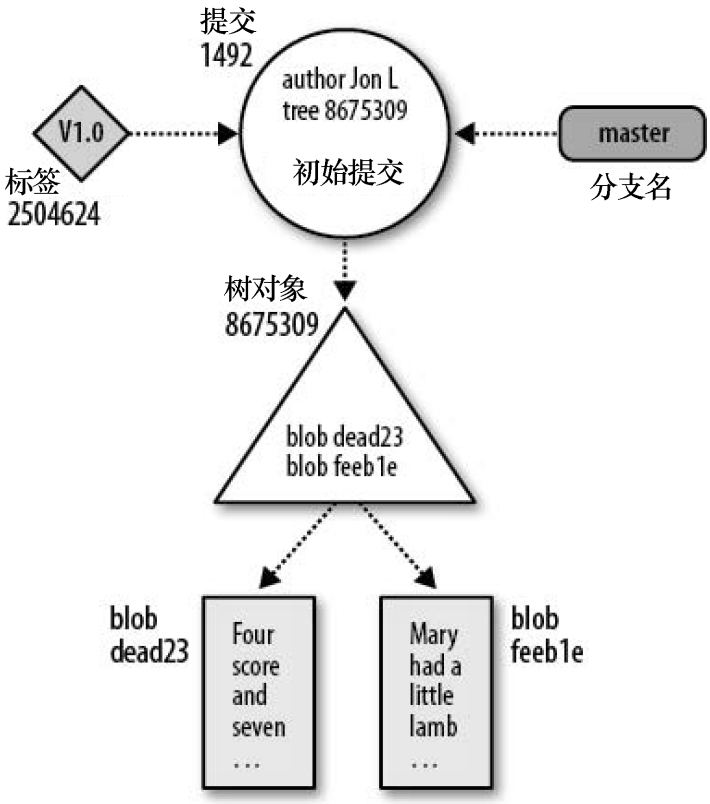


图 4-1 Git 对象

现在，让我们使事情变得复杂一点。保留原来的两个文件不变，添加一个包含一个文件的新子目录。对象库就如图 4-2 所示。

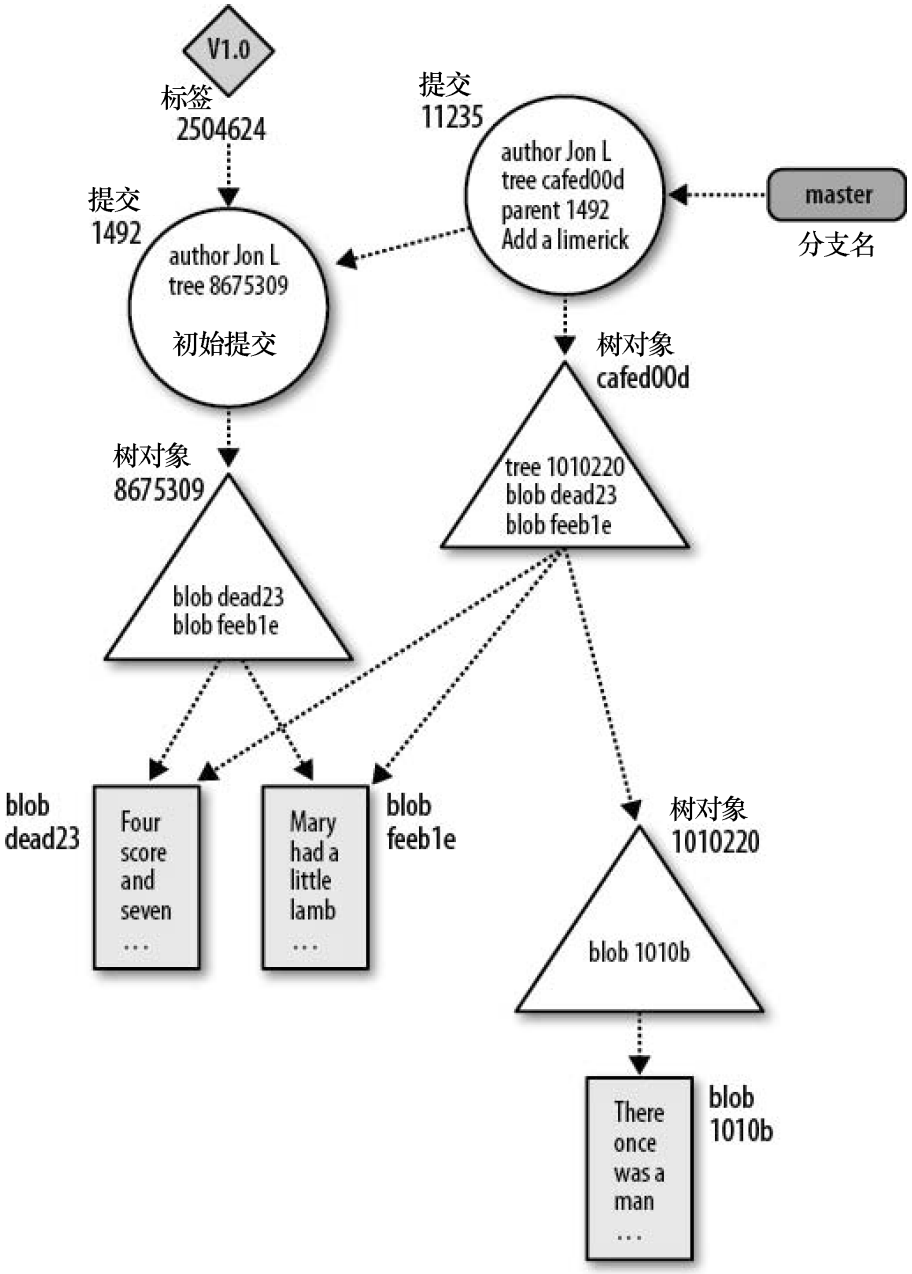


图 4-2 二次提交后的 Git 对象

就像前一张图里，新提交对象添加了一个关联的树对象来表示目录和文件结构的总状态。在这里，它是 ID 为 cafed00d 的树对象。

因为顶级目录被添加的新子目录改变了，顶级树对象的内容也跟着改变了，所以 Git 引进了一个新的树对象：cafed00d。

然而，blob 对象 dead23 和 feeb1e 在从第一次到第二次提交的时候没有发生变化。Git 意识到 ID 没有变化，所以可以被新的 cafed00d 树对象直接引用和共享。

请注意提交对象之间箭头的方向。父提交在时间上来得更早。因此，在 Git 的实现里，每个提交对象指回它的一个或多个父提交。很多人对此感到困惑，因为版本库的状态通常画成反方向：数据流从父提交流向子提交。

第 6 章扩展了这些图来展示版本库的历史是如何建立和被不同命令操作的。

4.3 Git 在工作时的概念

带着一些原则，来看看所有这些概念和组件是如何在版本库里结合在一起的。让我们创建一个新的版本库，并更详细地检查内部文件和对象库。

4.3.1 进入.git 目录

首先，使用 git init 来初始化一个空的版本库，然后运行 find 来看看都创建了什么文件。

```
$ mkdir /tmp/hello
$ cd /tmp/hello
$ git init
Initialized empty Git repository in /tmp/hello/.git/
# 列出当前目录中的所有文件
$ find .
.
./.git
./.git/hooks
./.git/hooks/commit-msg.sample
./.git/hooks/applypatch-msg.sample
./.git/hooks/pre-applypatch.sample
./.git/hooks/post-commit.sample
./.git/hooks/pre-rebase.sample
./.git/hooks/post-receive.sample
./.git/hooks/prepare-commit-msg.sample
./.git/hooks/post-update.sample
./.git/hooks/pre-commit.sample
./.git/hooks/update.sample
./.git/refs
./.git/refs/heads
./.git/refs/tags
./.git/config
./.git/objects
./.git/objects/pack
./.git/objects/info
./.git/description
./.git/HEAD
./.git/branches
```

```
./.git/info
./.git/info/exclude
```

可以看到，.git 目录包含很多内容。这些文件是基于模板目录显示的，根据需要可以进行调整。根据使用的 Git 的版本，实际列表可能看起来会有一点不同。例如，旧版本的 Git 不对*.git/hooks* 文件使用.*sample* 后缀。

在一般情况下，不需要查看或者操作.git 目录下的文件。认为这些“隐藏”的文件是 Git 底层（plumbing）或者配置的一部分。Git 有一小部分底层命令来处理这些隐藏的文件，但是你很少会用到它们。

最初，除了几个占位符之外，*.git/objects* 目录（用来存放所有 Git 对象的目录）是空的。

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/info
```

现在，让我们来小心地创建一个简单的对象。

```
$ echo "hello world" > hello.txt
$ git add hello.txt
```

如果输入的“hello world”跟这里一样（没有改变间距和大小写），那么 objects 目录应该如下所示：

```
$ find .git/objects
.git/objects
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

所有这一切看起来很神秘。其实不然，下面各节会慢慢解释原因。

4.3.2 对象、散列和 blob

当为 *hello.t__xt* 创建一个对象的时候，Git 并不关心 *hello.txt* 的文件名。Git 只关心文件里面的内容：表示“hello world”的 12 个字节和换行符（跟之前创建的 blob 一样）。Git 对这个 blob 执行一些操作，计算它的 SHA1 散列值，把散列值的十六进制表示作为文件名它放进对象库中。

> **如何知道一个 SHA1 散列值是唯一的？**
>
> 两个不同 blob 产生相同 SHA1 散列值的机会十分渺茫。当这种情况发生的时候，称为一次碰撞。然而，一次 SHA1 碰撞的可能性太低，你可以放心地认为它不会干扰我们对 Git 的使用。
>
> SHA1 是“安全散列加密”算法。直到现在，没有任何已知的方法（除了运气之外）可以让一个用户刻意造成一次碰撞。但是碰撞会随机发生吗？让我们来看看。

对于 160 位数，你有 2^{160} 或者大约 10^{48}（1 后面跟 48 个 0）种可能的 SHA1 散列值。这个数是极其巨大的。即使你雇用一万亿人来每秒产生一万亿个新的唯一 blob 对象，持续一万亿年，你也只有 10^{43} 个 blob 对象。

如果你散列了 2^{80} 个随机 blob，可能会发生一次碰撞。

不相信我们的话，就去读读 Bruce Schneier 的书吧[2]。

在这种情况下散列值是 3b18e512dba79e4c8300dd08aeb37f8e728b8dad。160 位的 SHA1 散列值对应 20 个字节，这需要 40 个字节的十六进制来显示，因此这内容另存为 *.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad*。Git 在前两个数字后面插入一个“/”以提高文件系统效率（如果你把太多的文件放在同一个目录中，一些文件系统会变慢；使 SHA1 的第一个字节成为一个目录是一个很简单的办法，可以为所有均匀分布的可能对象创建一个固定的、256 路分区的命名空间）。

为了展示 Git 真的没有对文件的内容做很多事情（它还是同样的内容“hello world”），可以在任何时间使用散列值把它从对象库里提取出来。

```
$ git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
hello world
```

 **提示**

Git 也知道手动输入 40 个字符是很不切实际的，因此它提供了一个命令通过对象的唯一前缀来查找对象的散列值。

```
$ git rev-parse 3b18e512d
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

4.3.3 文件和树

既然“hello world”那个 blob 已经安置在对象库里了，那么它的文件名又发生了什么事呢？如果不能通过文件名找到文件 Git 就太没用了。

正如前面提到的，Git 通过另一种叫做目录树（tree）的对象来跟踪文件的路径名。当使用 git add 命令时，Git 会给添加的每个文件的内容创建一个对象，但它并不会马上为树创建一个对象。相反，索引更新了。索引位于 *.git/index* 中，它跟踪文件的路径名和相应的 blob。每次执行命令（比如，git add、git rm 或者 git mv）的时候，Git 会用新的路径名和 blob 信息来更新索引。

[2] 《应用密码学》的作者、美国密码学学者、信息安全专家与作家。—译者注

任何时候，都可以从当前索引创建一个树对象，只要通过底层的 git write-tree 命令来捕获索引当前信息的快照就可以了。
目前，该索引只包含一个文件，*hello.txt.*

```
$ git ls-files -s
100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0      hello.txt
```

在这里你可以看到文件的关联，*hello.txt* 与 3b18e4...的 blob。
接下来，让我们捕获索引状态并把它保存到一个树对象里。

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ find .git/objects
.git/objects
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

现在有两个对象：3b18e5 的“hello　world”对象和一个新的 68aba6 树对象。可以看到，SHA1 对象名完全对应.*git/objects* 下的子目录和文件名。
但是树是什么样子的呢？因为它是一个对象，就像 blob 一样，所以可以用底层命令来查看它。

```
$ git cat-file -p 68aba6
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad     hello.txt
```

对象的内容应该很容易解释。第一个数 100644，是对象的文件属性的八进制表示，用过 UNIX 的 chmod 命令的人应该对这个很熟悉了。这里，3b18e5 是 *hello　world* 的 blob 的对象名，*hello.txt* 是与该 blob 关联的名字。
当执行 git ls-file -s 的时候，很容易就可以看到树对象已经捕获了索引中的信息。

4.3.4 对 Git 使用 SHA1 的一点说明

在更详细地讲解树对象的内容之前，让我们先来看看 SHA1 散列的一个重要特性。

```
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
$ git write-tree
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

每次对相同的索引计算一个树对象，它们的 SHA1 散列值仍是完全一样的。Git 并不需要重新创建一个新的树对象。如果你在计算机前按照这些步骤操作，你应该看到完全一样的 SHA1 散列值，跟本书所刊印

的一样。

这样看来，散列函数在数学意义上是一个真正的函数：对于一个给定的输入，它总产生相同的输出。这样的散列函数有时也称为摘要，用来强调它就像散列对象的摘要一样。当然，任何散列函数（即使是低级的奇偶校验位）也有这个属性。

这是非常重要的。例如，如果你创建了跟其他开发人员相同的内容，无论你俩在何时何地工作，相同的散列值就足以证明全部内容是一致的。事实上，Git 确实将它们视为一致的。

但是等一下—SHA1 散列是唯一的吗？难道万亿人每秒产生的万亿个 blob 永远不会产生一次碰撞吗？这在 Git 新手中是一个常见的疑惑。因此，请仔细阅读，因为如果你能理解这种区别，那么本章的其他内容就很简单了。

在这种情况下，相同的 SHA1 散列值并不算碰撞。只有两个不同的对象产生一个相同的散列值时才算碰撞。在这里，你创建了相同内容的两个单独实例，相同的内容始终有相同的散列值。

Git 依赖于 SHA1 散列函数的另一个后果：你是如何得到称为 68aba62e560c0ebc3396e8ae9335232cd93a3f60 的树的并不重要。如果你得到了它，你就可以非常有信心地说，它跟本书的另一个读者的树对象是一样的。Bob 通过合并 Jennie 的提交 A、提交 B 和 Sergey 的提交 C 来创建这个树，而你是从 Sue 得到提交 A，然后从 Lakshmi 那里更新提交 B 和提交 C 的合并。结果都是一样的，这有利于分布式开发。

如果要求你查看对象 68aba62e560c0ebc3396e8ae9335232cd93a3f60，并且你能找到这样的一个对象，同时因为 SHA1 是一个加密散列算法，因此你就可以确信你找的对象跟散列创建时的那个对象的数据是相同的。

反过来也是如此：如果你在你的对象库里没找到具有特定散列值的对象，那么你就可以肯定你没有持有那个对象的副本。总之，你可以判断你的对象库是否有一个特定的的对象，即使你对它（可能非常大）的内容一无所知。因此，散列就好似对象的可靠标签或名称。

但是 Git 也依赖于比那个结论更强的东西。考虑最近的一次提交（或者它关联的树对象）。因为它包含其父提交以及树的散列，反过来又通过递归整个数据结构包含其所有子树和 blob 的散列，因此可归结为它通过原始提交的散列值唯一标识整个数据结构在提交时的状态。

最后，我们在上一段中的声明可以推出散列函数的强大应用：它提供了一种有效的方法来比较两个对象，甚至是两个非常大而复杂的数据结构[3]，而且并不需要完全传输。

4.3.5 树层次结构

只有单个文件的信息是很好管理的，就像上一节所讲的一样，但项目包含复杂而且深层嵌套的目录结构，并且会随着时间的推移而重构和移动。通过创建一个新的子目录，该目录包含 *hello.txt* 的一个完全相同的副本，让我们看看 Git 是如何处理这个问题的。

```
$ pwd
/tmp/hello
```

[3] 对这个数据结构更详细的描述见 6.3.2 节。—原注

```
$ mkdir subdir
$ cp hello.txt subdir/
$ git add subdir/hello.txt
$ git write-tree
492413269336d21fac079d4a4672e55d5d2147ac
$ git cat-file -p 4924132693
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
040000 tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60    subdir
```

新的顶级树包含两个条目：原始的 *hello.txt* 以及新的 子目录 ，子目录是 树 而不是 blob 。

注意到不寻常之处了吗？仔细看 *subdir* 的对象名。是你的老朋友，68aba62e560c0ebc3396e8ae9335232cd93a3f60！

刚刚发生了什么？*subdir* 的新树只包含一个文件 *hello.txt* ，该文件跟旧的“hello world”内容相同。所以 *subdir* 树跟以前的顶级树是完全一样的！当然它就有跟之前一样的 SHA1 对象名了。

让我们来看看 *.git/objects* 目录，看看最近的更改有哪些影响。

```
$ find .git/objects
.git/objects
.git/objects/49
.git/objects/49/2413269336d21fac079d4a4672e55d5d2147ac
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

这只有三个唯一的对象：一个包含“hello world”的 blob；一棵包含 *hello.txt* 的树，文件里是“hello world”加一个换行；还有第一棵树旁边包含 *hello.txt* 的另一个索引的另一棵树。

4.3.6 提交

讨论的下一主题是提交（commit）。现在 *hello.txt* 已经通过 git add 命令添加了，树对象也通过 git write-tree 命令生成了，可以像这样用底层命令那样创建提交对象。

```
$ echo -n "Commit a file that says hello\n" \
    | git commit-tree 492413269336d21fac079d4a4672e55d5d2147ac
3ede4622cc241bcb09683af36360e7413b9ddf6c
```

结果如下所示。

```
$ git cat-file -p 3ede462
tree 492413269336d21fac079d4a4672e55d5d2147ac
author Jon Loeliger <jdl@example.com> 1220233277 -0500
```

```
committer Jon Loeliger <jdl@example.com> 1220233277 -0500

Commit a file that says hello
```

如果你在计算机上按步骤操作，你可能会发现你生成的提交对象跟书上的名字不一样。如果你已经理解了目前为止的一切内容，那原因就很明显了：这是不同的提交。提交包含你的名字和创建提交的时间，尽管这区别很微小，但依然是不同的。另一方面，你的提交确实有相同的树。这就是提交对象与它们的树对象分开的原因：不同的提交经常指向同一棵树。当这种情况发生时，Git 能足够聪明地只传输新的提交对象，这是非常小的，而不是很可能很大的树和 blob 对象。

在实际生活中，你可以（并且应该）跳过底层的 git write-tree 和 git commit-tree 步骤，并只使用 git commit 命令。成为一个完全快乐的 Git 用户，你不需要记住那些底层命令。

一个基本的提交对象是相当简单的，这是成为一个真正的 RCS 需要的最后组成部分。提交对象可能是最简单的一个，包含：

- 标识关联文件的树对象的名称；
- 创作新版本的人（作者）的名字和创作的时间；
- 把新版本放到版本库的人（提交者）的名字和提交的时间；
- 对本次修订原因的说明（提交消息）。

默认情况下，作者和提交者是同一个人，也有一些情况下，他们是不同的。

 **提示**

可以使用 git show --pretty=fuller 命令来查看给定提交的其他细节。

尽管提交对象跟树对象用的结构是完全不同的，但是它也存储在图结构中。当你做一个新提交时，你可以给它一个或多个父提交。通过继承链来回溯，可以查看项目历史。第 6 章会给出关于提交和提交图的更详细描述。

4.3.7 标签

最后，Git 还管理的一个对象就是标签。尽管 Git 只实现了一种标签对象，但是有两种基本的标签类型，通常称为 轻量级的（lightweight）和带附注的（annotated）。

轻量级标签只是一个提交对象的引用，通常被版本库视为是私有的。这些标签并不在版本库里创建永久对象。带标注的标签则更加充实，并且会创建一个对象。它包含你提供的一条消息，并且可以根据 RFC 4880 来使用 GnuPG 密钥进行数字签名。

Git 在命名一个提交的时候对轻量级的标签和带标注的标签同等对待。不过，默认情况下，很多 Git 命令只对带标注的标签起作用，因为它们被认为是“永久”的对象。

可以通过 git tag 命令来创建一个带有提交信息、带附注且未签名的标签：

```
$ git tag -m "Tag version 1.0" V1.0 3ede462
```

可以通过 git cat-file -p 命令来查看标签对象，但是标签对象的 SHA1 值是什么呢？为了找到它，使用 4.3.2 节的提示。

```
$ git rev-parse V1.0
6b608c1093943939ae78348117dd18b1ba151c6a
$ git cat-file -p 6b608c
object 3ede4622cc241bcb09683af36360e7413b9ddf6c
type commit
tag V1.0
tagger Jon Loeliger <jdl@example.com> Sun Oct 26 17:07:15 2008 -0500
Tag version 1.0
```

除了日志消息和作者信息之外，标签指向提交对象 3ede462。通常情况下，Git 通过某些分支来给特定的提交命名标签。请注意，这种行为跟其他 VCS 有明显的不同。
Git 通常给指向树对象的提交对象打标签，这个树对象包含版本库中文件和目录的整个层次结构的总状态。

回想一下图 4-1，V1.0 标签指向提交 1492—依次指向跨越多个文件的树（8675309）。因此，这个标签同时适用于该树的所有文件。
这跟 CVS 不同，例如，对每个单独的文件应用标签，然后依赖所有打过标签的文件来重建一个完整的标记修订。并且 CVS 允许你移动单独文件的标签，而 Git 则需要在标签移动到的地方做一个新的提交，囊括该文件的状态变化。

第 21 章 Git 和 GitHub

虽然本书前面的章节主要关注于 Git 的命令行工具，但是自从 2005 年开始，Git 支持和促进了一些关于它的社区和工具的发展。这些工具有数百个，并且有很多不同的形式，类型从桌面 GUI（如 SmartGit，http://syneveo.com/smartgit）到桌面备份工具（如 SparkleShare，http://sparkshare.org）。但是在这些工具之外，有一个在开发人员甚至非开发人员中心中都占有最显著的地位，它就是 GitHub。

图 21-1 所示就是这个网站的首页，网站所介绍的概念——社会化编程，很可能在前几年被忽略掉，但是现在确是我们中的许多人认为应该有的工作方式。社会化编程的这种模式首先用在开源项目中，但是在最近两年，在地理上的分布式协作开发已经在闭源的企业中有所发展。下来就让我们看看 GitHub 为我们提供什么。

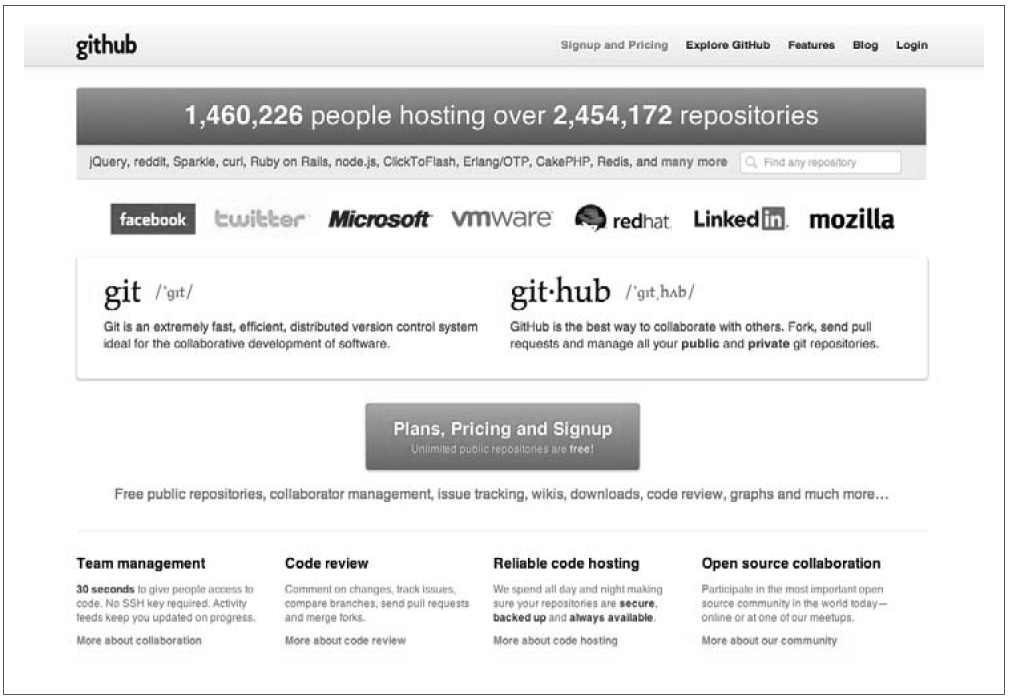


图 21-1 GitHub 主页

21.1 为开源代码提供版本库

有统计数据系显示，很多开发人员和 Git 第一次的接触就是从 GitHub 上克隆版本库。这正是 GitHub 的原始功能。它提供了一个能够通过 git://：、https://和 git+ssh://这些协议与版本库交互的界面。对于开源项目，账户是免费的，并且所有账户可以创建不限数量的公共版本库。这极大地促进了语言的开源社区（从 JavaScript 到 ClojureScript）对 Git 的使用。

在浏览器中打开网址 http://github.com，然后单击图 21-2 中的 Sign Up 链接就能开始创建新账户。

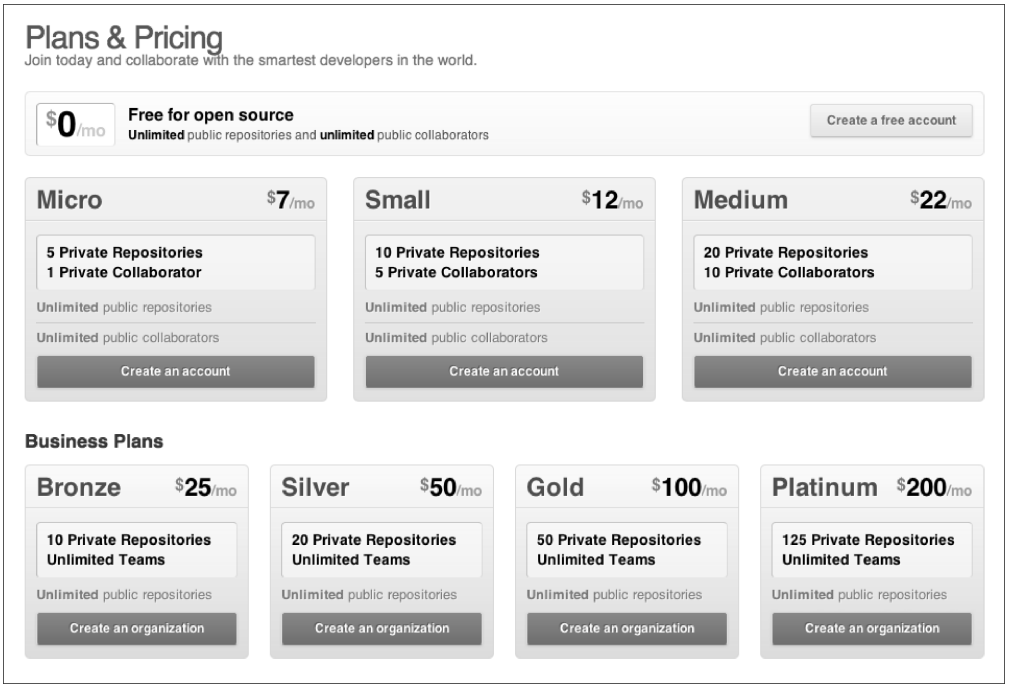


图 21-2 选择账户类型

GitHub 有 4 种账户类型：免费个人账户、付费个人账户、免费组织账户和付费组织账户。要加入一个组织必须要有个人账户。在选择用户名的时候需要慎重一点，因为默认一个账户只有一次修改用户名的机会（见图 21-3）。一个账户可以关联多个邮箱地址，并且可以随时更改。因此，用户名是最永久性的注册信息。

注册完最常见的账户类型——免费个人账户之后（见图 21-4），用户会被引导到 GitHub 的帮助页面，这里会提供一些教程，这些教程介绍设置开发人员桌面安装的 Git 的参数配置。

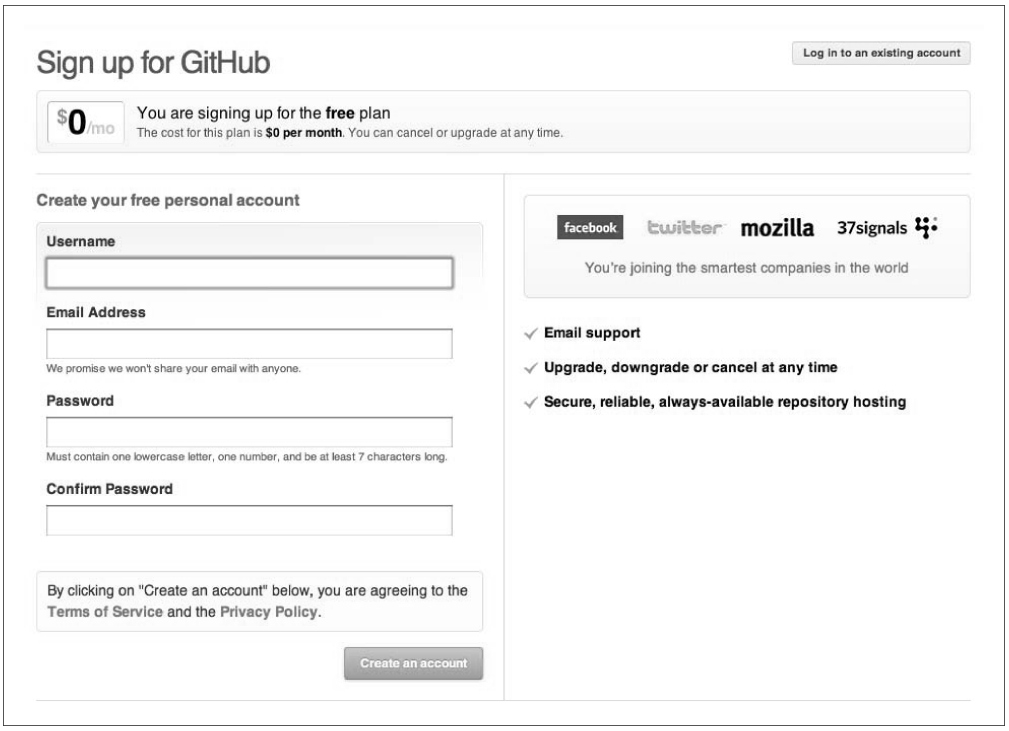


图 21-3 免费个人账户

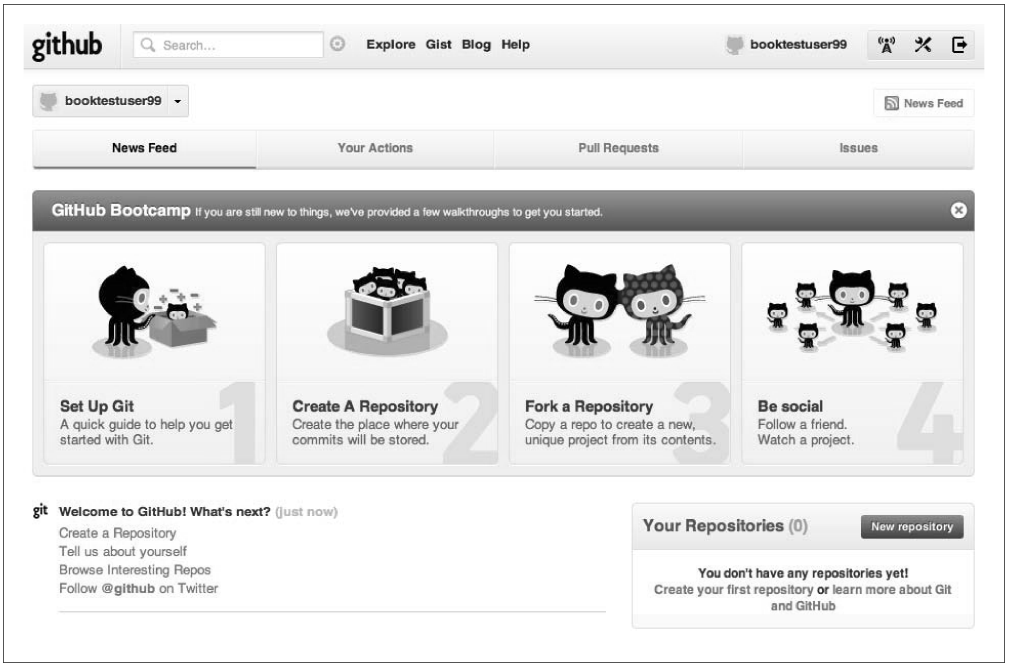


图 21-4 账户创建完成

21.2 创建 GitHub 的版本库

新版本库信息

在你创建账户之后，只需要单击最顶部的 New Repository 按钮就可以创建新版本库，这个按钮在你每次登录后的界面上都会显示。也可以通过输入 http://github.com/new 来创建新的版本库。

唯一必需的信息是版本库的名字。另外也可以设置项目的描述信息和项目首页的 URL 来使项目更加容易识别（见图 21-5）。

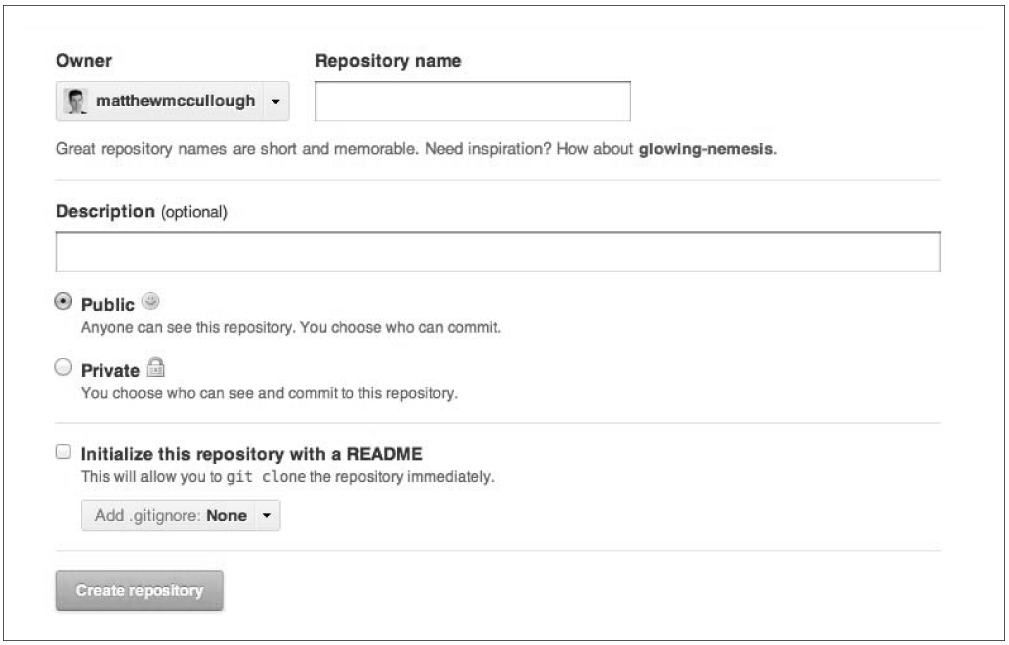


图 21-5 创建公开版本库

接下来，需要设置一些版本库的初始信息。根据你现在是否已经有提交，这里有两种途径。

README Seeding（选项一）

如果项目的第一步工作是在开始编写代码前创建 GitHub 版本库，那么你将需要创建一个占位符文件作为第一次提交。在创建新版本库的时候，你会被问及是否创建初始的 *README* 文件和.*gitignore* 文件。*README* 文本文件一般用来描述项目的意图。

至此，项目可以利用命令 git clone *url* 克隆到本地，之后，就可以在本地添加和提交代码了。

添加远程版本库（选项二）

如果你已经拥有了一个本地版本库，你可以将 GitHub 的地址和已经存在的本地版本库链接起来。要做到这一点，需要利用命令 git remote add *url* 将 GitHub 的 URL（Git 的一个远程地址）添加到已经存在的 Git 版本库中。

将本地的内容推送到 GitHub

在经过上面的步骤之后，本地版本库已经链接到远程版本库。本地版本库的内容可以推送到 GitHub。这需要利用命令 git push *remote branch*。如果分支之前从未发布，明确的命令 git push -u origin master 比较合适。-u 告诉 Git 去跟踪推送的分支，将它推送到 origin 远程版本库，同时将它推送到 master 分支。

一旦将 GitHub 的版本库和之前的版本库建立联系，之后进行的代码改变就可以简单地通过调用 git push 来推送。这便产生了使用可访问的中心 Git 版本库的好处，进行分布式协作的开发人员可以看见其他所有项目参与者提交的更改（见图 21-6），即使他们是离线的。

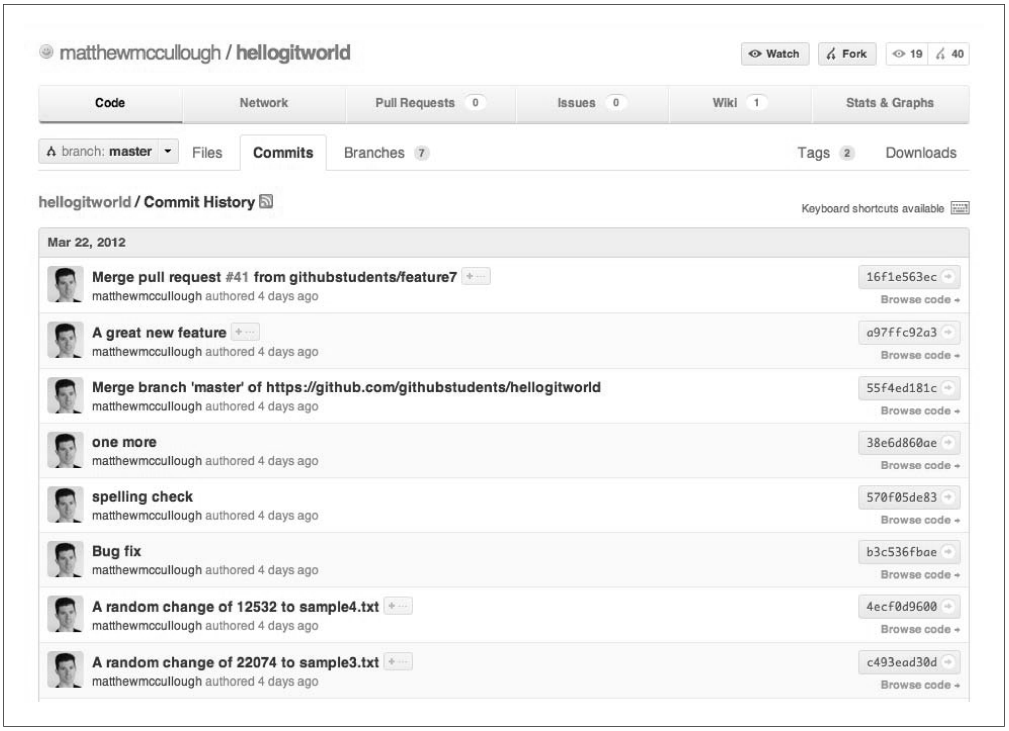


图 21-6 GitHub 上的提交历史记录

21.3 开源代码的社会化编程

GitHub 可以局限地被看做一个托管开源项目的地方。然而，在线创建版本库的概念其实已经被 SourceForge 和 Google Code 很好的利用，它们都有各自有优势的用户界面。其他一些关于组织条款、证书和提交权利的概念由 Apache 基金会、Codehaus 和 Eclipse 基金会进行了深化。

但是 GitHub 创造了完全不同的方式去促进协作，这便是利用社区贡献（见图 21-7）。GitHub 提供类似 Twitter、Facebook 等其他一些社交网站一样的社会化方面来记录编程中的社会化活动。Watch 使得用户可以关注感兴趣的项目，版本库 Fork 允许用户复刻项目，Pull Requests 可以使其他程序员向原项目的拥有者发送合并请求，行级 Comments 使得用户可以对提交进行具体的留言和评价，综合上面的功能来看，GitHub 确实将编码变成了一项社会化活动。正是因为这些流程，使得大量开源项目比之前用补丁文件报 bug 的时代有了更多的代码贡献者。

图 21-7 社会化编程

21.4 关注者

在 GitHub 上最简单的社会化编程功能就是“关注”，这个操作就是图 21-8 中的 Watch 按钮。关注和 Twitter 上的跟随者和 Facebook 上的朋友类似，它意味着你对某个 GitHub 用户、组织或者特定项目感兴趣。

图 21-8 Watch 按钮

关注者的人数也是开源项目受欢迎程度的标志。在 GitHub 的探索页面里面，对项目的搜索就可以是基于版本库跟随者数量的（见图 21-9）。当这些和编程语言结合起来时，关注者数量可以显示出在某一领域内有用的样例代码。

图 21-9 探索并搜索关注者数量

21.5 新闻源

在你关注了用户、组织或者版本库之后，在图 21-10 的 News Feed 中会根据你关注的对象显示内容。这些新闻源就是你关注的用户版本库或者组织的活动信息。

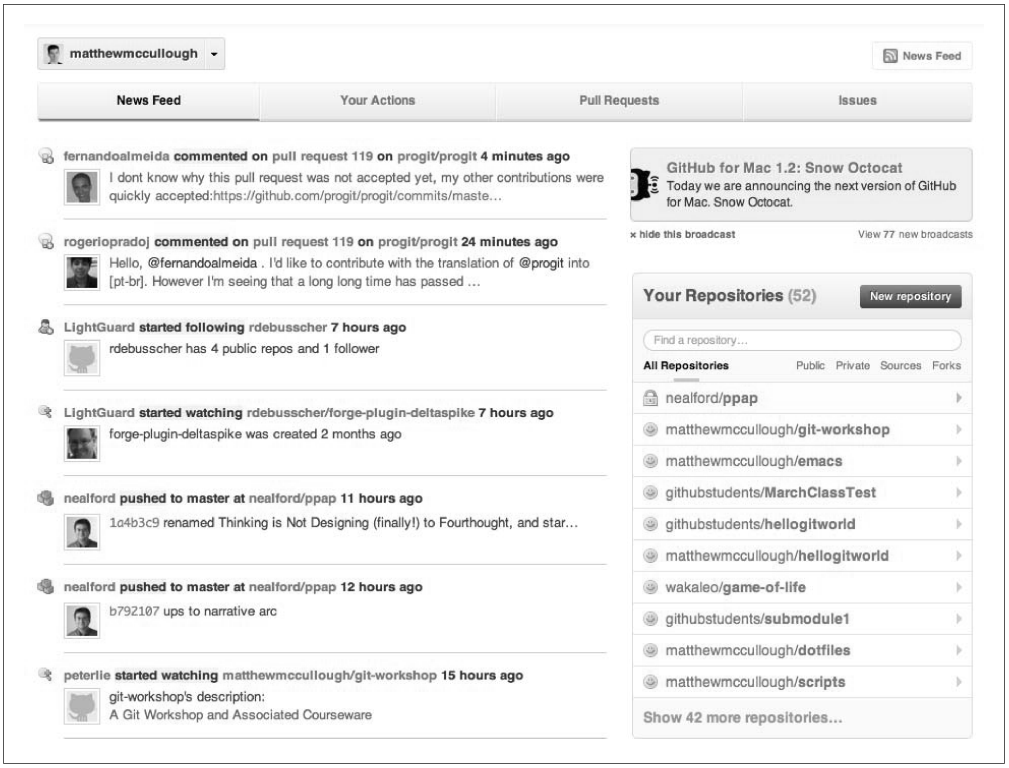


图 21-10 新闻源

新闻源既可以在 GitHub.com 的网站上查看，也可以作为 RSS 源由你选择的阅读器应用订阅。

21.6 复刻

复刻其他人的项目是 GitHub 普及的另外一个概念，这个概念甚至传播到了其他领域（见图 21-11）。复制这个词通常带有负面的内涵。在不久之前，从编码角度来看，复制意味着通过比较鲁莽的方式将程序引入完全不同方向的复制操作。

图 21-11 Fork 按钮

GitHub 的复刻思想是一个积极的操作。它使得大量贡献者可以用可控制的、可见的方式对项目做出大量的代码贡献。复刻是一个自发的动作，它使得任何贡献者可以获得项目代码的一个私人副本。这个私人副本（用 GitHub 的名词来说就是 fork）可以不经原作者的任何明确授权进行更改。这不会对核心项目造成任何危险，因为这些更改是发生在复刻的版本库中的，而不是原始版本库中。

这和 Apache 或者 Eclipse 项目的概念是相反的，它们的补丁被当做 bug 报告的文件附件提交。GitHub 的这种模式使得社区的贡献是透明和公共可见的（见图 21-12），即使在它们提交回核心项目去讨论和合并前也是这样的。

图 21-12 的网络图显示了核心项目的分支与提交和非主项目的分支与提交，包括复刻的版本库。这提供了社区关于这个项目所有活动的概览，还有某个复刻的项目是否明显偏离了核心项目。这使得即使没有发送合并请求，仍然能够提供很多有意义的分散的社区贡献的概览。

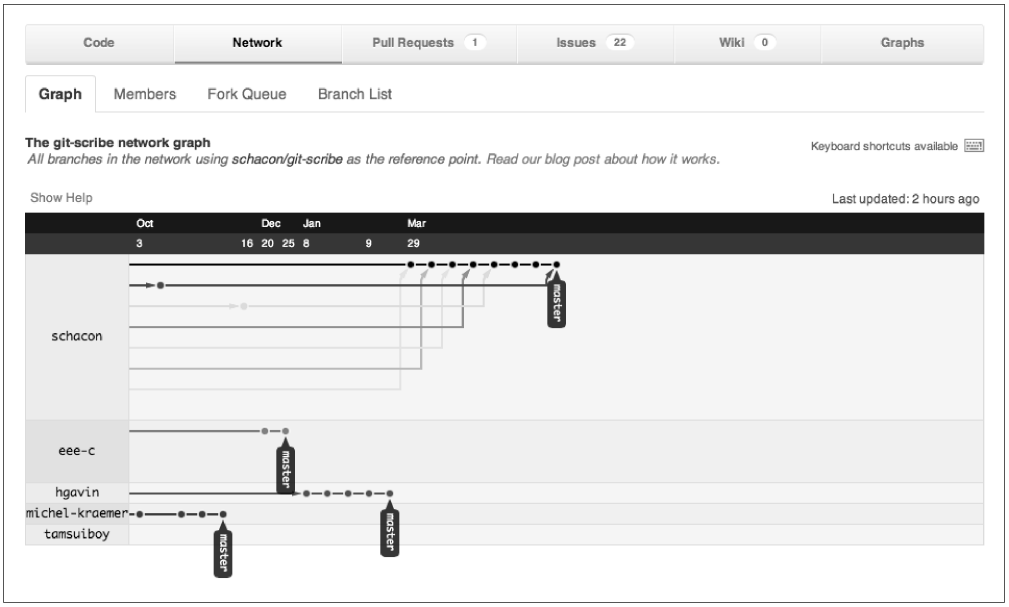


图 21-12 网络图

经过数年对社区行为的观察，项目越来越多的边缘用户开始提交修复和小的改善代码，这是因为这样的事在 Github 上已经非常方便。很多开源项目已经同时使用原来的 bug 附带补丁模式以及新的复刻和合并请求方式。在老的模式下对项目做出贡献的最大障碍在于，准备补丁需要的时间和真正修复 bug 所花时间的悬殊。

21.7 创建合并请求

复刻只是对一个项目创建一个私人副本，真正为核心项目带来价值的是合并请求（pull request）。合并请求是在任何用户进行了他觉得有用的提交操作的情况下，向核心项目拥有者发送的通知。

当一个贡献者完成了一个特性的编码，将它提交到某个命名分支，并且将这个新分支推送到复刻时，它可以转变成合并请求。合并请求可以精确地描述成“围绕一个主题的提交列表”。合并请求通常都基于某个主题分支的全部内容。但是，当不是整个分支的内容都准备好推送到一个发布的分支时，合并请求也可以是一个规模更小的提交。当从下拉的分支选择器中选择了最新推送的分支后，可以单击上下文相关的 Pull Request 按钮（见图 21-31），这便会出发合并请求。

图 21-13 Pull Request 按钮

合并请求的默认动作包括在当前主题分支上的所有提交。然而，当调用它的时候，可以手动更改提交的范围以及源分支和目标分支（见图 21-14）。

图 21-14 合并请求范围

至此，合并请求已经创建，接下来核心项目的拥有者将会查看、评估、评论，并且有可能合并这些更改。从概念上来看，这个过程可以拿来和在 Crucible 和 Gerri 上进行的代码审查来比较。然而，在 GitHub 看

来这个流程很好，刚好达到轻量级与能够满足代码审查间的平衡。在 GitHub 上还可以自动进行融合新代码的繁重过程，这只需要在 Pull Request 页面上单击一个按钮。

21.8 管理合并请求

在 GitHub 上，一个成功的项目会有一系列合并请求需要管理（见图 21-15）。核心项目实例的所有协作者都有管理和处理合并请求的权限。需要注意的是，合并请求可能不是来自复刻的。一些时候，拥有核心项目协作者权限的开发人员仍然会发送合并请求，这是为了在合并分支时能够获取反馈。

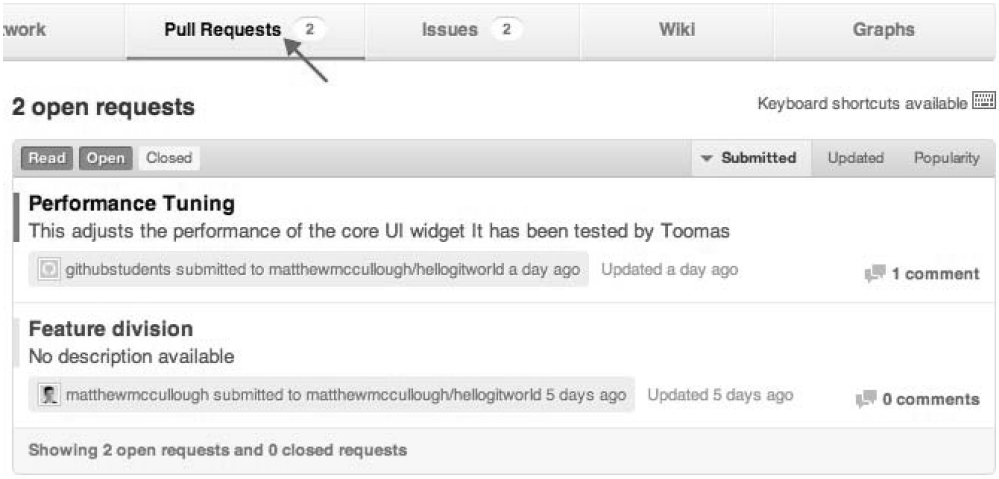


图 21-15 项目的合并请求队列

合并请求是 GitHub 生态圈中十分重要的一部分，以至于每个用户都会有一个属于他自己的自定义面积，显示的是他作为协作者的所有项目上的合并请求（见图 21-16）。

图 21-16 全局合并请求队列

在合并请求之后的概念其实是将通常二元的接受或拒绝操作转变成讨论。讨论伴随着关于合并请求的评论或者关于具体提交的评论（见图 21-17）。评论可以是指导性的，意味着已经提交的解决方案还需要进一步完善。如果贡献者在合并请求的某个分支上又进行了一些提交操作，推送这些提交后，仍然会在合并请求线程中按顺序显示出来。

图 21-17 合并请求评论

评论可以分为三个层次：对合并请求的评论，对提交的评论和对某行代码的评论。其中，行级评论对技术的改善最有用（见图 21-18），它使评论者能够对代码作者提供具有相同逻辑代码的建议。

图 21-18 合并请求行级提交评论

当对合并请求的代码做了足够的更改并且已经可以合并到主项目时，可以通过很多方式进行合并。这其中最创新和节省时间的方式就是通过 GitHub 用户界面上的自动融合按钮进行融合（见图 21-19）。这个操作会执行真正的 Git 提交操作，就像从命令行中操作一样。这会省去将代码下载到本地进行融合，然后再推送回 GitHub 的过程。

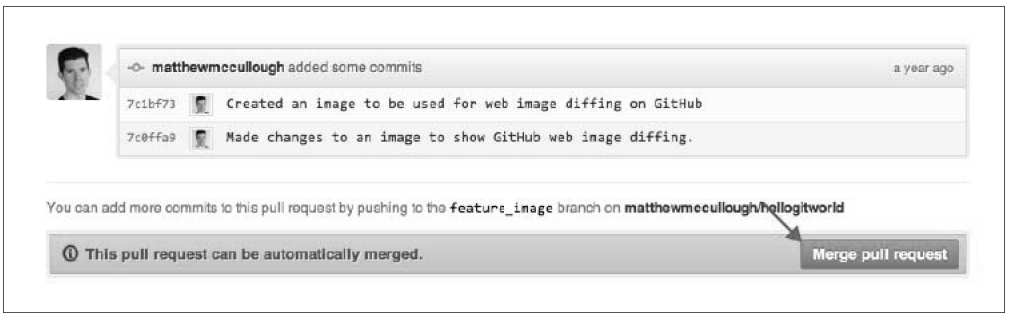


图 21-19 合并请求自动融合

发送合并请求会很自然地被看做完成了某个特性的开发、修复了一个 bug 或者完成其他开发工作。其实，

合并请求也可以在开始一个概念时使用。更常见的是，通过一张 JPEG 原型图片或者一个概括主题分支目标的文本文件启动合并请求。这些合并请求通常会征求团队的反馈，而这种反馈就是通过上文所述的合并请求评论方法实现的。贡献者可以继续向 GitHub 推送关于那个主题分支的变更，而合并请求会自动地随着最近的提交而被更新。

21.9 通知

像 GitHub 这样的社会化系统需要一个强大的通知机制，从而能够通知用户他们关注的项目、组织和用户发生了哪些改变。如你合理地猜测，通知主要就是上面提到的三种对象产生的对应类型。
所有关于你的通知概览会集中在一个通知页面。这个页面可以通过顶部导航栏的那个消息通知图标进入（见图 21-20）。

图 21-20 “通知”按钮

相关通知列表会显示事件源的图标。这些图标包括版本库、用户和组织级的活动。每一条活动的概览还有事件详细信息的链接（见图 21-21）。

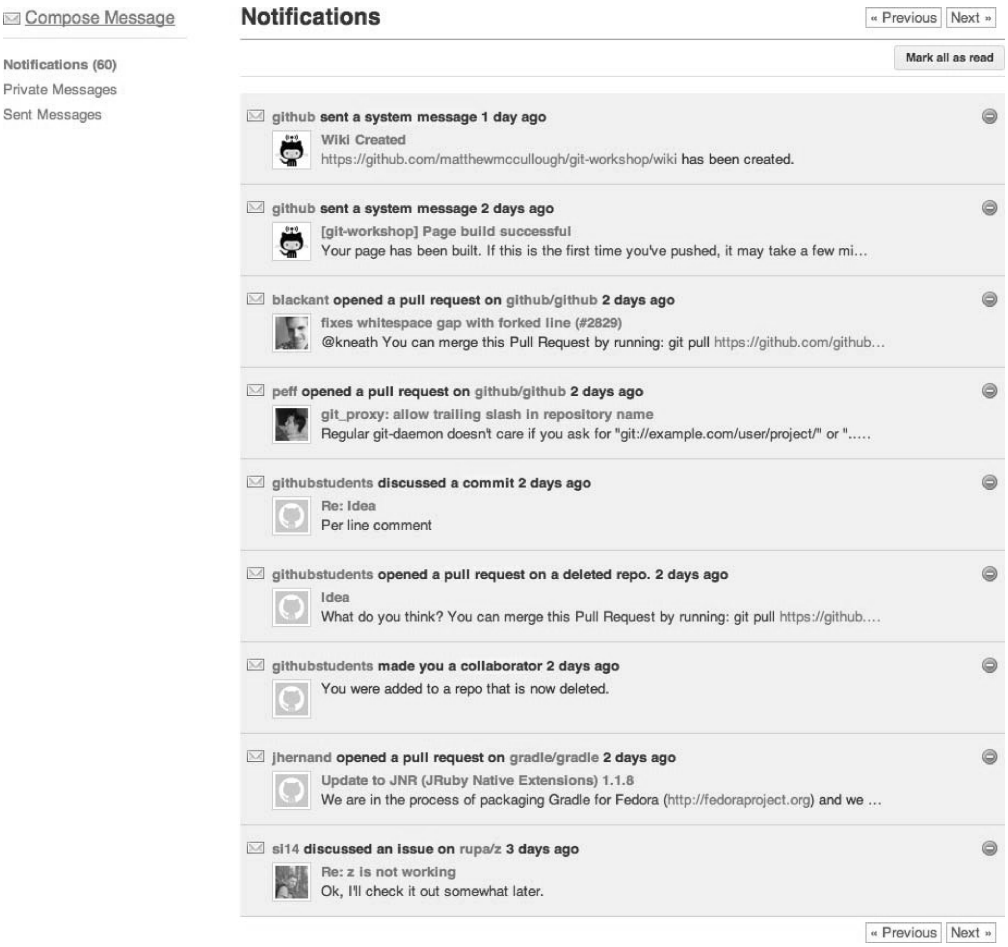


图 21-21 通知列表

通知可以通过每个版本库页面底部的一个超链接来控制是否开启（见图 21-22）。

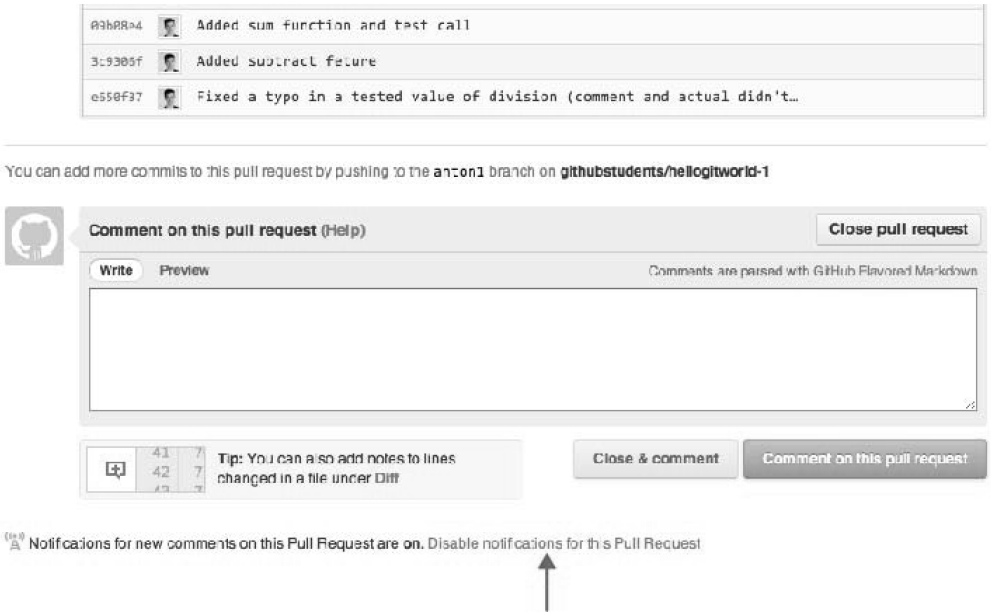


图 21-22 通知版本库开关

对全局通知的修改可以在用户的管理设置页面中进行。哪些类型的事件需要通知，以及是仅仅在网页上通知，还是也通过用户的邮件地址通知，这些都可以通过这个页面进行设置（见图 21-23）。

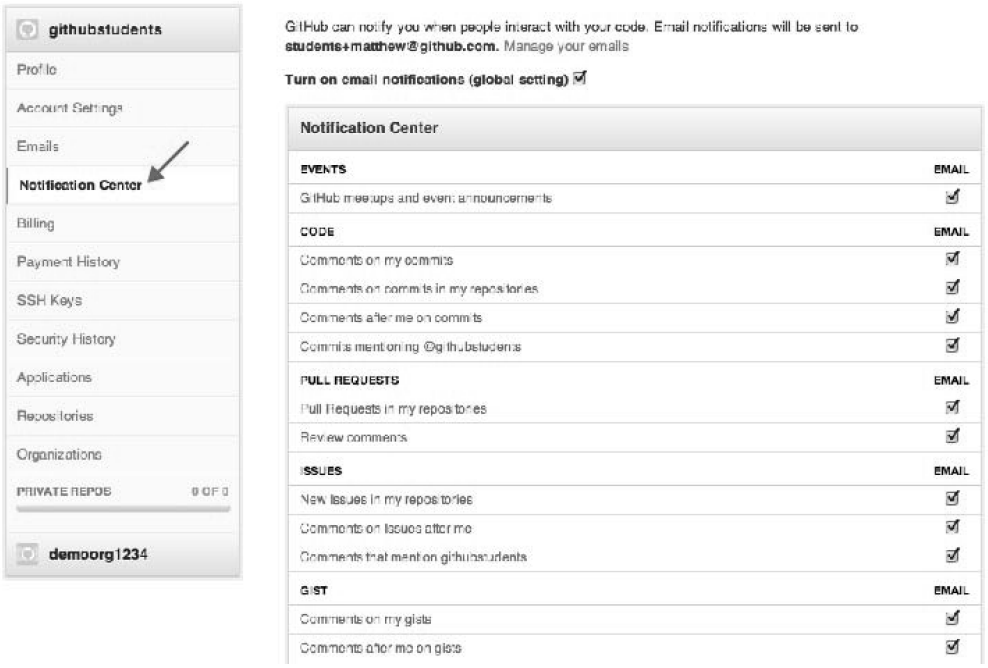


图 21-23 通知设置

21.10 查找用户、项目和代码

GitHub 主要集中了大量的开源项目，同时促进了开源项目的协作，但是，很大一部分开源社区关注寻找和应用开源库。GitHub 的 Explore 页面很好地满足了这个需求（见图 21-24）。这个开放的 Explore 页面

汇聚了一系列版本库，这些从统计数据上显示了趋势，同时能够让开源社区对其产生兴趣。

如果你想查找某种编程语言的示例代码，那么 Advanced Search 页面就是你想要的（见图 21-25）。对用户、受欢迎度、版本库名和编程语言的设置可以使你进行精准的搜索。

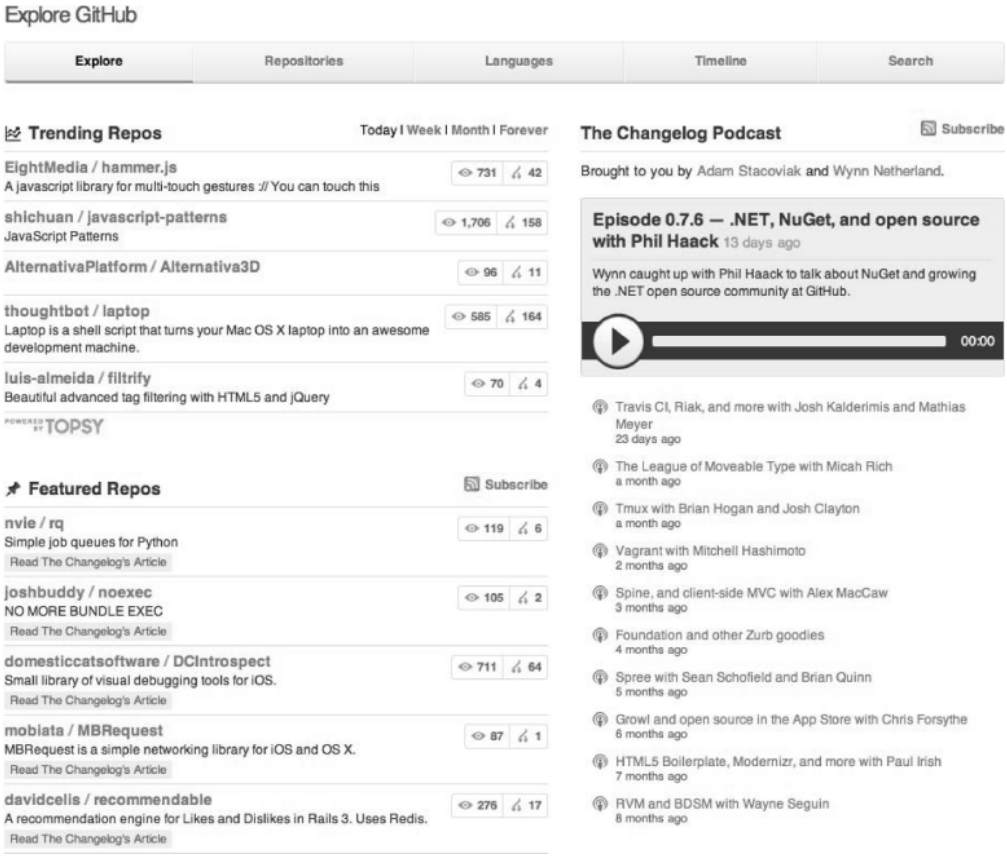


图 21-24 探索

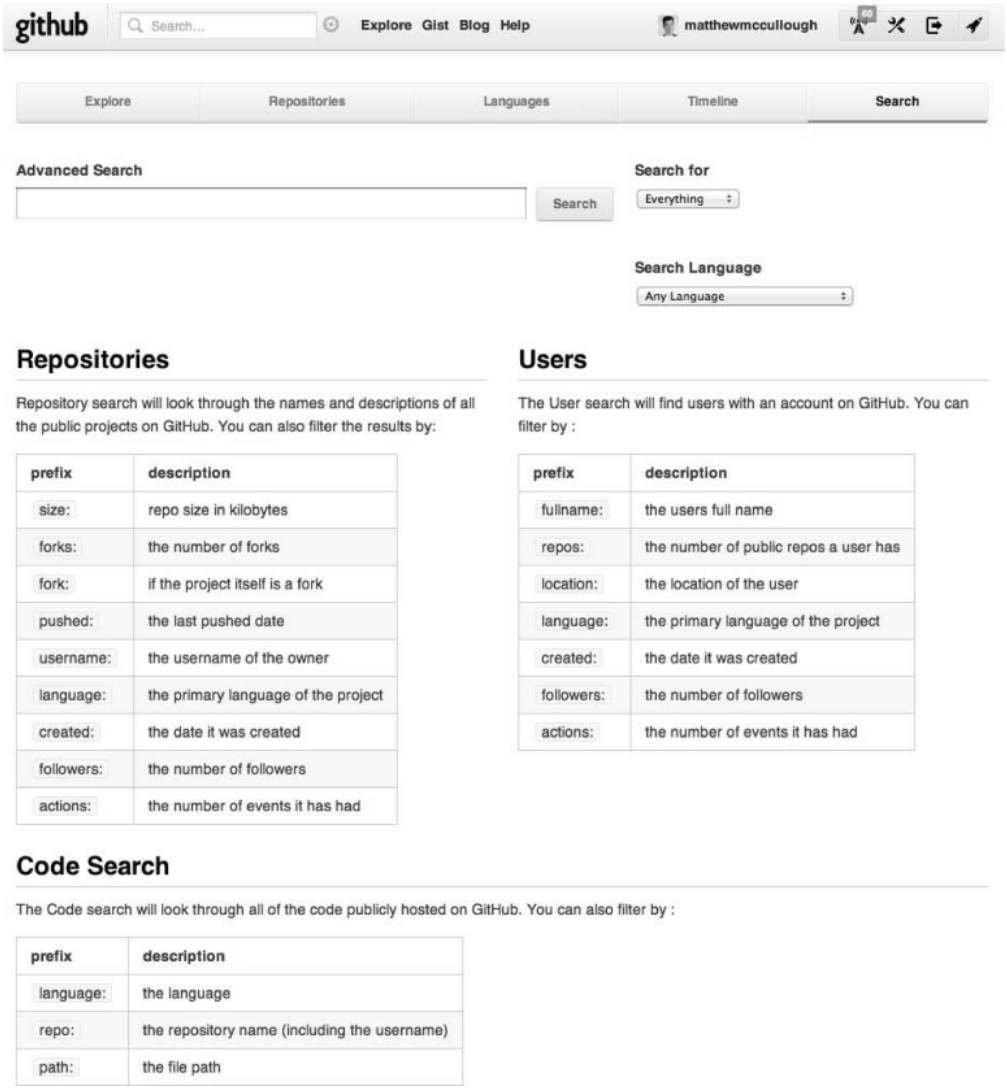


图 21-25 网站搜索

21.11 维基

在过去更新维基意味着你需要在浏览器上编辑页面。这是一种非常不可靠的并且没有版本控制的方式。浏览器稍稍刷新就会使更改丢失。

维基是用 Markdown（http://www.daringfireball/markdown）语法编辑的 Git 版本库，并且是依附在相应的项目上的。GitHub 的维基页面（见图 21-26）允许提交、评论、融合、变基 Git 用户可以使用的功能，还允许执行其他操作，而这些是之前维基用户从来没有使用过的。

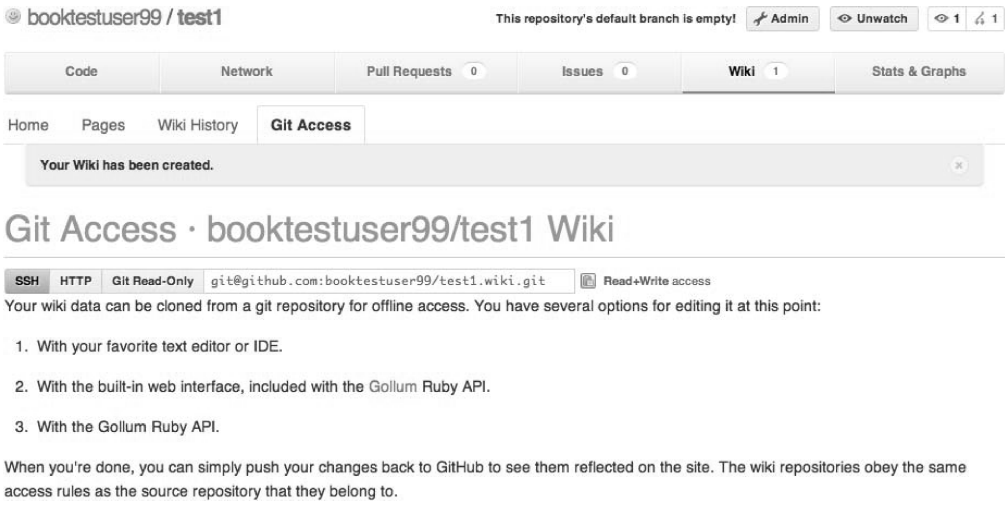


图 21-26 GitHub 维基

通过把版本库克隆到你的本地机器去编辑维基并不意味着放弃了在浏览器中的更改方式（见图 21-27）。在浏览器进行编辑也会写回底层 Git 版本库，这样用户就可以跟踪作者的活动和所有维基页面更改的历史记录。

21.12 GitHub 页面（用于网站的 Git）

如果维基主页听起来很吸引人，那么使用拥有 Git 版本库的 Markdown 文件作为发布整个网站的工具，这听起来如何？GitHub 页面基于 Jekyll（https://github.com/mojombo/jekyu），这个项目就提供上述功能，甚至可以用域名系统（Domain Name System，DNS）CNAME 记录映射关联到一个子域名或主域名（见图 21-28）。

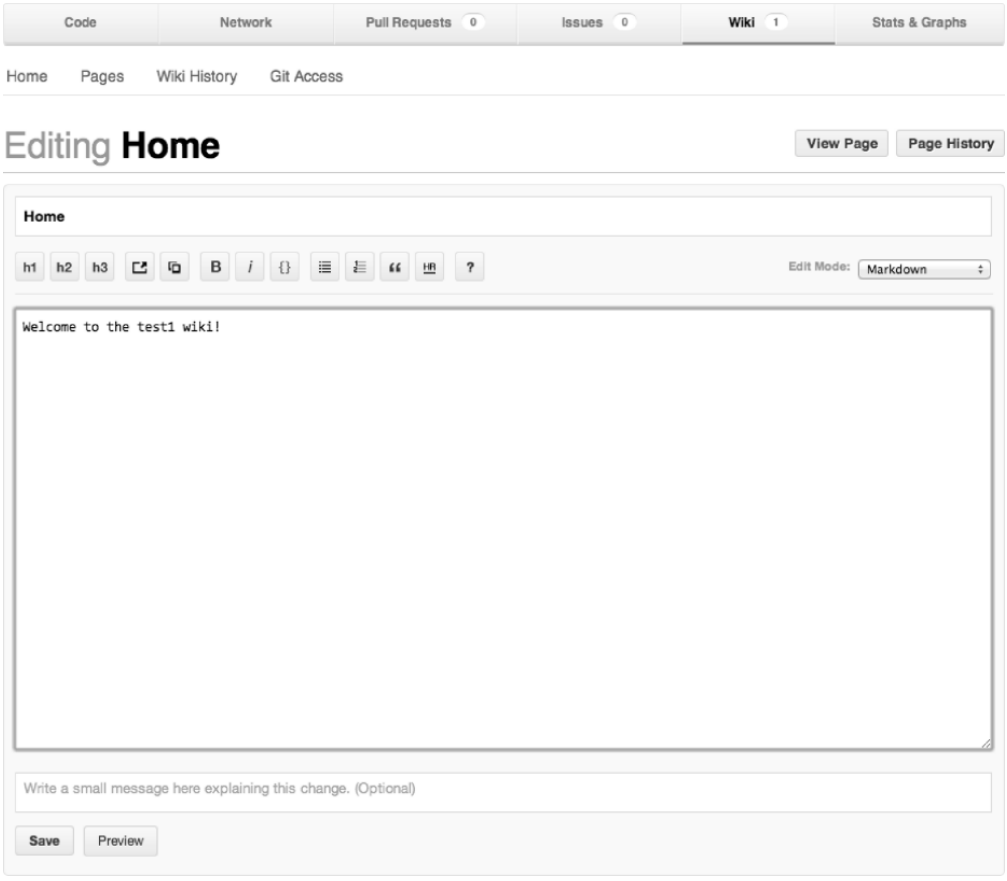


图 21-27 GitHub 在浏览器中编辑维基

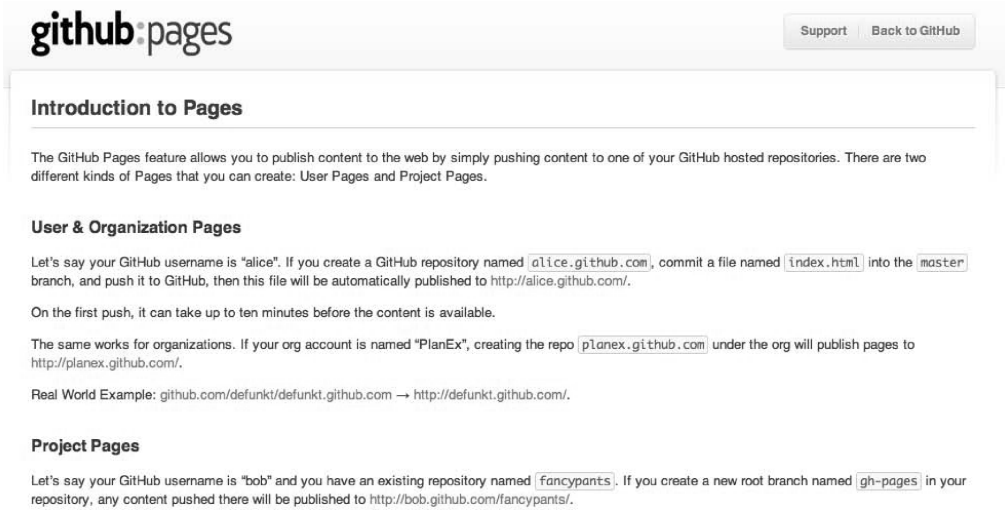


图 21-28 GitHub 页面介绍

Octopress（http://octopress.org，见图 21-29）已经从 Jekyll 和 GitHub 主页的混合模式中获益，这使得用一种静态方式发布动态内容变得非常方便。自身安全的缺陷与日益增多的攻击使得使用数据库和即时编译的动态网站逐渐转为静态页面。但是这不意味着要放弃动态站点的产生，我们只需要将动态处理过程迁移到产生内容的阶段而不是像使用 JSP（JavaServer Page）或者 PHP 那样在页面请求的阶段。

图 21-29 Octopress 主页

21.13 页内代码编辑器

通常情况下，用户会在台式机的编辑器中进行编程，但是对于一个微小的更改（比如，修改一个拼写错误）来说，拉取代码、修改代码、提交代码和推送代码这样的流程太不方便了。出于这样的原因，GitHub 提供了在浏览器中进行代码编辑的功能（见图 21-30）。

浏览器内编辑器基于 Mozilla 的 Ace（http://ace.ajax.org/），一个基于 JavaScript 的控件。这个控件还用于 Cloud9 IDE 和 Beanstalk。这个控件（见图 21-31）支持显示行数、代码突出显示和空格符、制表符格式化。这样，代码修改变得和在 GitHub 上浏览源文件一样简单，只需单击 Edit，在浏览器内编辑器下面输入提交消息，然后提交更改即可。微小的修改从来没有这么简单。

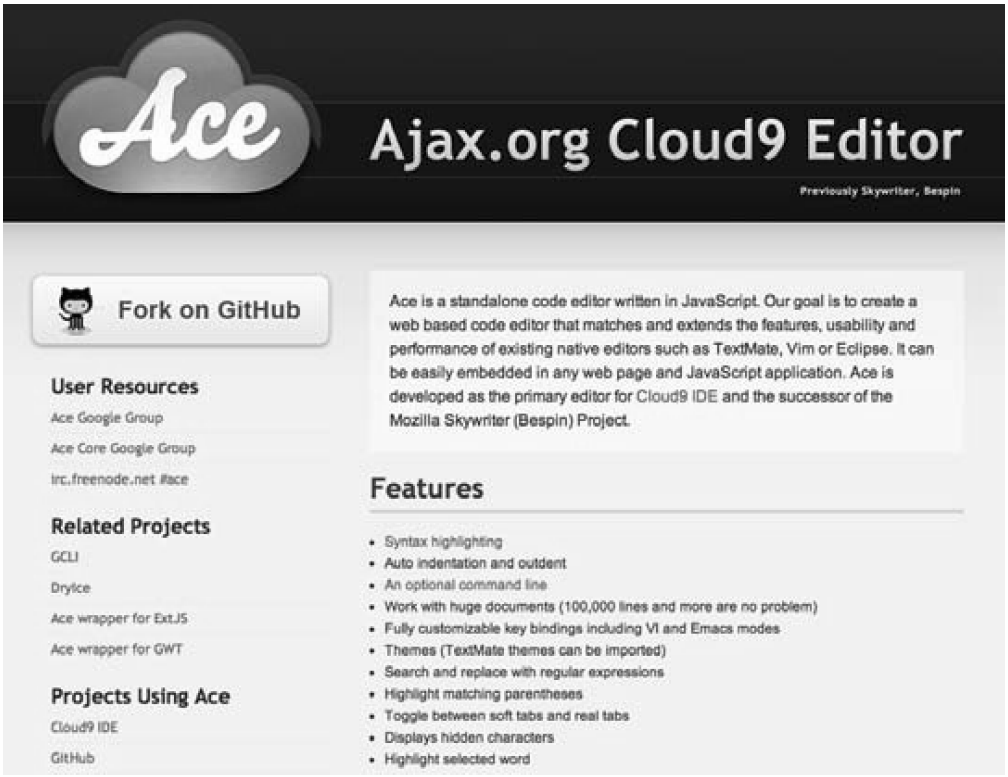


图 21-30 Ace 浏览器内编辑器

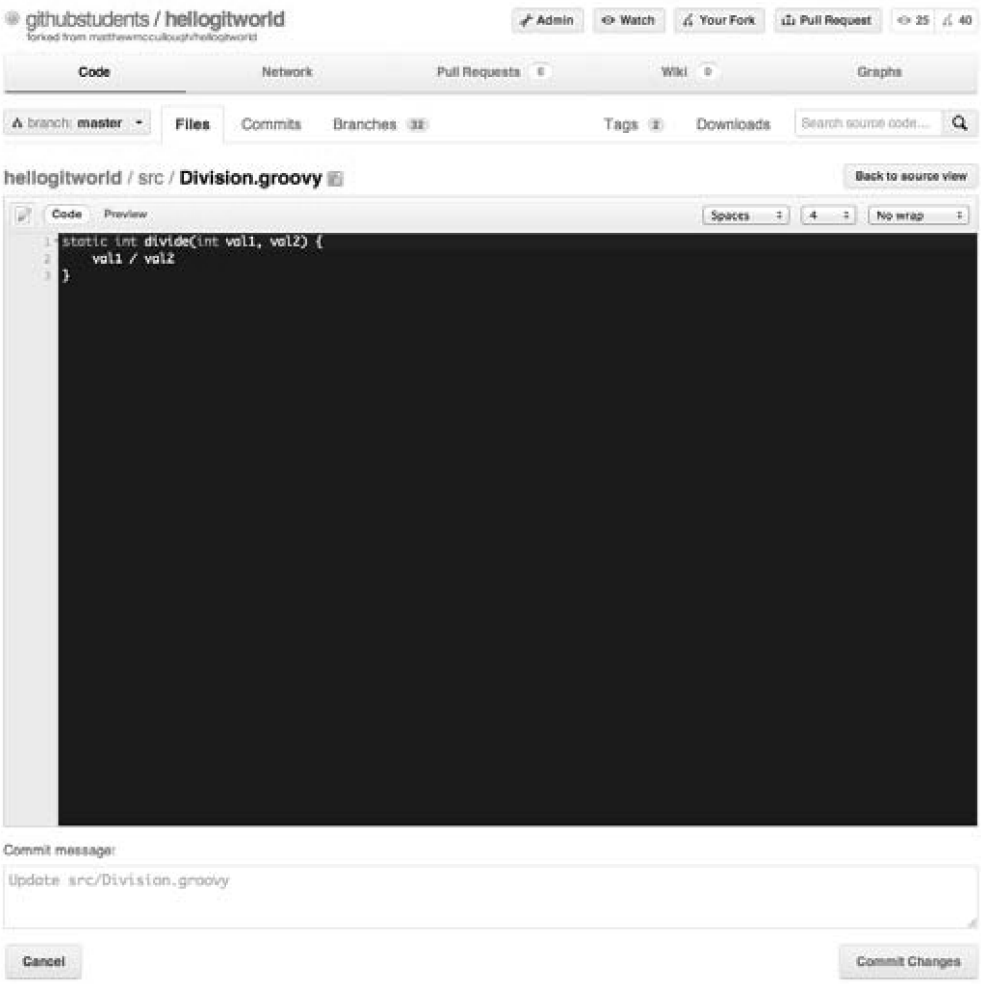


图 21-31 在浏览器内编辑代码

21.14 对接 SVN

虽然 GitHub 相信 Git 是 VCS 的未来，但是可以预计的是 SVN 仍然会使用很长时间。GitHub 支持这两种版本控制方式。

通常，Git 用户将版本库放在 SVN 中同时使用 git-svn 命令来对接这两种版本控制技术。然而，这种方法意味着只有很少的 SVN 元数据可以保留，而这些数据不包括 Git 作者、Git 提交者信息，以及 Git 先前提交的引用。

GitHub 让这两种技术对接成为可能，而且不需要客户端会话软件的支持。一个 Git 版本库可以在请求时动态转变为 SVN 版本库，而且这不改变用来克隆的 HTTPS URL（见图 21-32）。这是一个复杂的动态转化，而且只有在 Github 上的 Git 版本库才能提供。这个技术使得 SVN 到 Git 的转变能够以谨慎和渐变的方式进行。这个在服务器端的对接使 Git 和 SVN 的连接不仅能够通过 GUI 使用，而且能够用其他 SVN 遗留连接工具进行提交（见图 21-33）。Git 的默认分支通常为 master 分支，这个分支会自动映射到 SVN 界面的 trunk 分支上，这种映射正是提前考虑了在 SVN 领域中术语的意义。

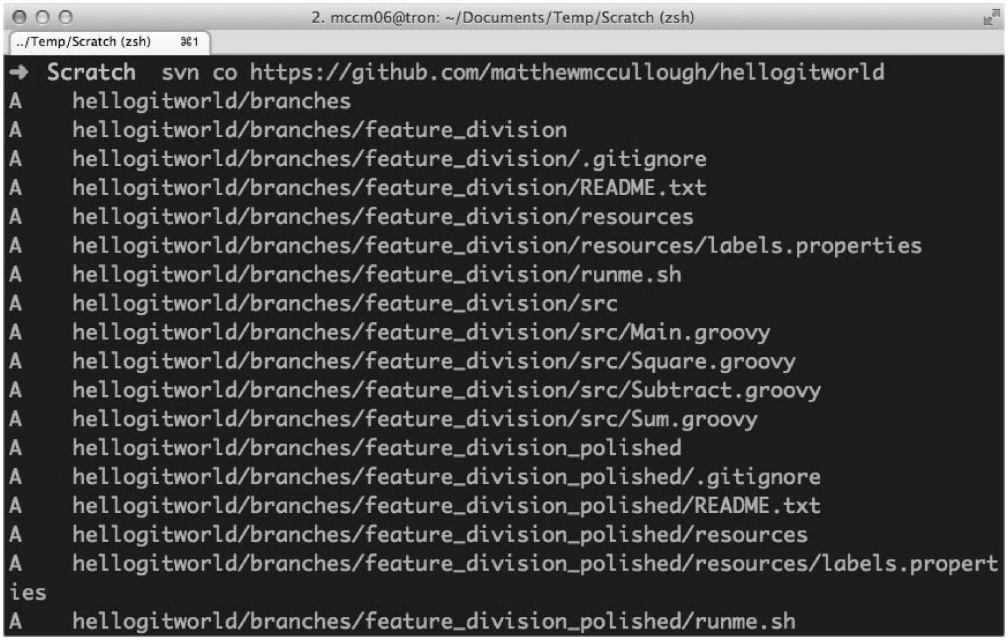
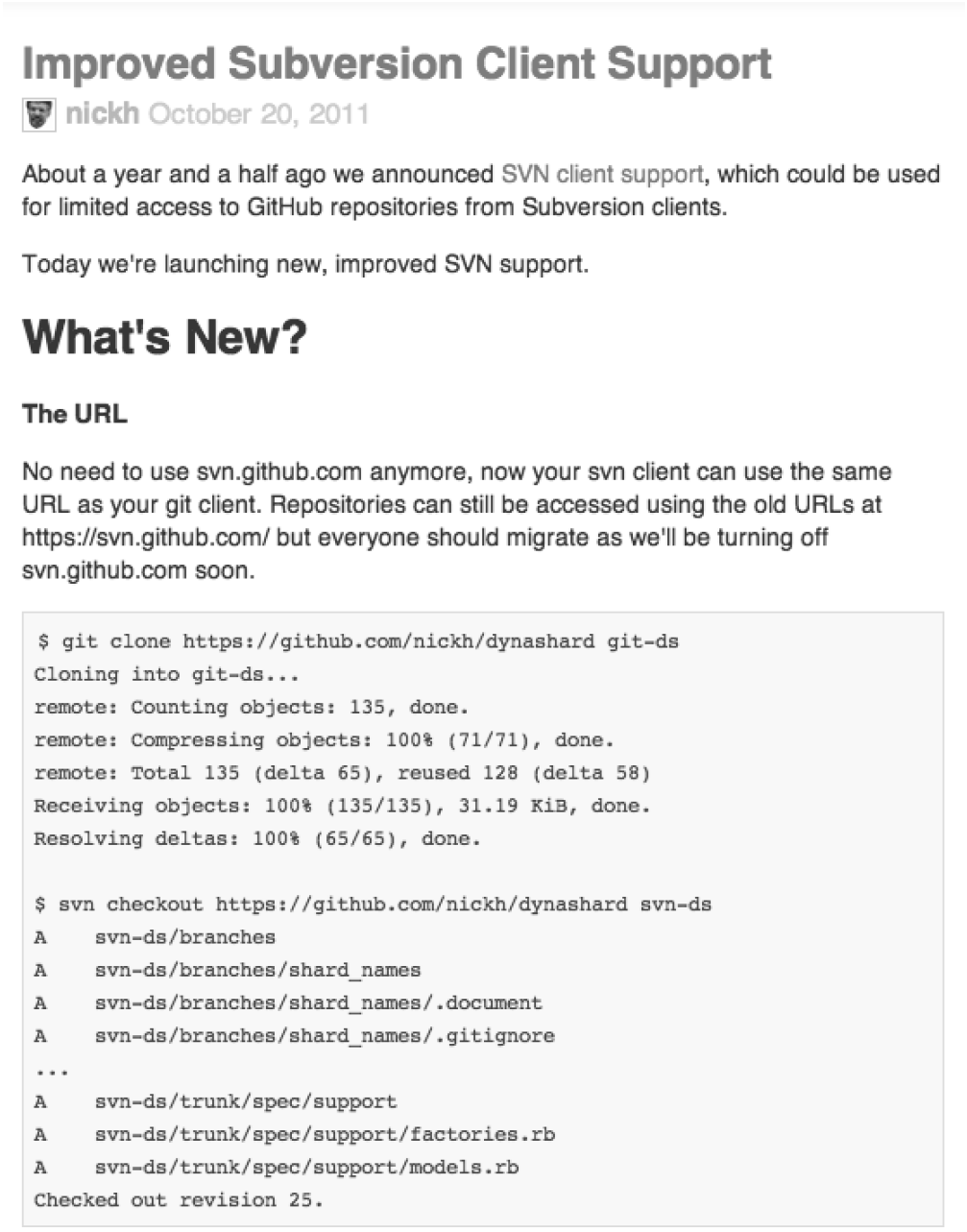


图 21-32 Git 版本库的 SVN 克隆



Improved Subversion Client Support

nickh October 20, 2011

About a year and a half ago we announced SVN client support, which could be used for limited access to GitHub repositories from Subversion clients.

Today we're launching new, improved SVN support.

What's New?

The URL

No need to use svn.github.com anymore, now your svn client can use the same URL as your git client. Repositories can still be accessed using the old URLs at https://svn.github.com/ but everyone should migrate as we'll be turning off svn.github.com soon.

```
$ git clone https://github.com/nickh/dynashard git-ds
Cloning into git-ds...
remote: Counting objects: 135, done.
remote: Compressing objects: 100% (71/71), done.
remote: Total 135 (delta 65), reused 128 (delta 58)
Receiving objects: 100% (135/135), 31.19 KiB, done.
Resolving deltas: 100% (65/65), done.

$ svn checkout https://github.com/nickh/dynashard svn-ds
A    svn-ds/branches
A    svn-ds/branches/shard_names
A    svn-ds/branches/shard_names/.document
A    svn-ds/branches/shard_names/.gitignore
...
A    svn-ds/trunk/spec/support
A    svn-ds/trunk/spec/support/factories.rb
A    svn-ds/trunk/spec/support/models.rb
Checked out revision 25.
```

图 21-33 Git-SVN 对接

21.15 标签自动归档

当一个开源项目想在 GitHub 上创建该项目一个压缩的归档时，有一种非常方便的方法：只需要将某个版本的代码打上标签。Git 的标签会自动转变为 TGZ 和 ZIP 压缩归档，这些归档可以在标签页找到（见图 21-34）。

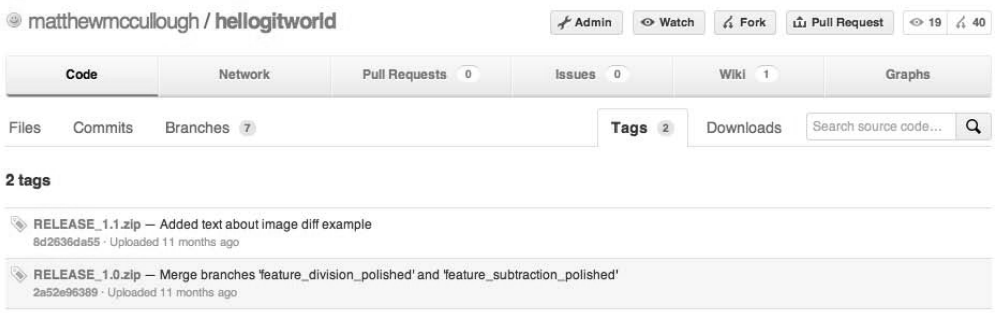


图 21-34 标签和归档

21.16 组织

至此，本书主要讨论的都是关于相对独立的少量个体 GitHub 用户的交互操作。然而，Git 已经吸引了大批内聚性很强的组织、小型公司和大型企业。GitHub 中的一组功能是为那些“组织”服务的（见图 21-35）。

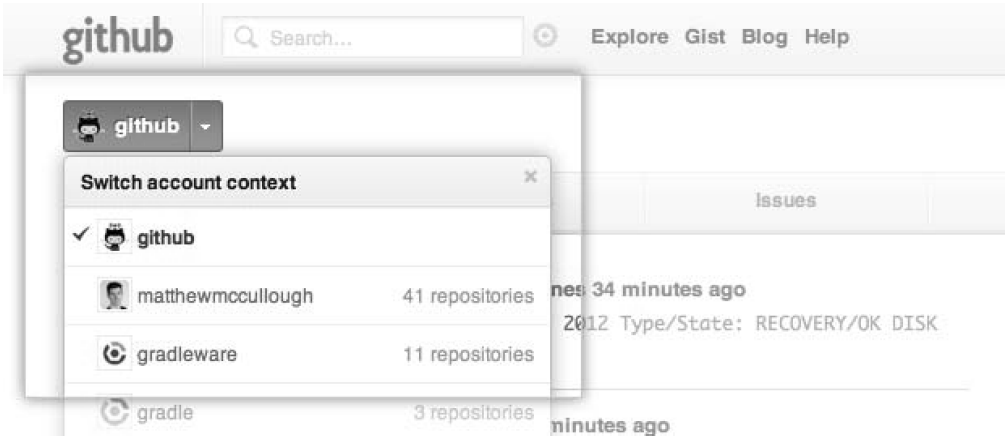


图 21-35 组织选择器

GitHub 的组织提供了相对个体用户在更高层次上对版本库的拥有权。为了支持这种机制，还有一个附加的安全结构：团队。团队是一种组织方式，它和一个特定的权限级别和一组版本库相关联。权限的三个层次是：仅拉取、拉取和推送，以及拉取、推送和管理（见图 21-36）。

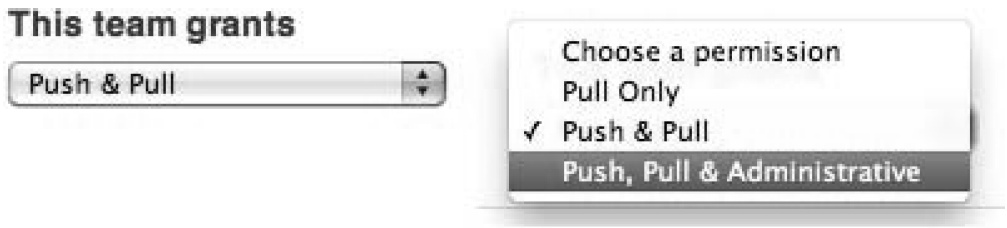


图 21-36 组织权限

21.17 REST 风格的 API

有一个 Web 应用确实是个非常好的开始，但是 GitHub 拥有很多社区开发人员，非常渴望能够使用真正的服务来构建有用的特性，而不仅仅是是页面。为了促进社区构建支持工具，GitHub 建立了全面的应用编程接口（API）。GitHub 的 API 已经历经了三个阶段，现在 API 的 V3 版本提供了 API 形式的几乎所有的 UI 能够提供的特性。在一些情况下，可能在 GitHub UI 上没有的功能，在 API 里已经提供了。

下面是一个通过用户获取其所在组织的 API（见例 21-1）。GitHub API 的所有响应都是 JSON（JavaScript Object Natation）格式的。注意，avatar_url 实际上是一个完整的长字符串，因为排版原因分为了几行。

例 21-1 调用 GitHub API

```
curl https://api.github.com/users/matthewmccullough/orgs
[
  {
    "avatar_url": "https://secure.gravatar.com/avatar/11f43e3d3b15205be70289ddedfe2de7
        ?d=https://a248.e.akamai.net/assets.github.com
        %2Fimages%2Fgravatars%2Fgravatar-orgs.png",
    "login": "gradleware",
    "url": "https://api.github.com/orgs/gradleware",
    "id": 386945
  },
  {
    "avatar_url": "https://secure.gravatar.com/avatar/61024896f291303615bcd4f7a0dcfb74
        ?d=https://a248.e.akamai.net/assets.github.com
        %2Fimages%2Fgravatars%2Fgravatar-orgs.png",
    "login": "github",
    "url": "https://api.github.com/orgs/github",
    "id": 9919
  }
]
```

GitHub 的所有操作都是用 RESTful API 来组织的，同时在 GitHub API 站点上有很详细的文档（见图 21-37）。除了能够获取用户列表、版本库列表或者文件列表之外，API 还提供了开发的标准身份认证接口 OAUTH，来以 GitHub 用户身份登录。这些使得查询、操作私有版本库的内容，使用版本库作为源代码之外的产品的存储容器，把构建应用程序从构建版本控制持久化层的困难中抽象出来都成为了可能。

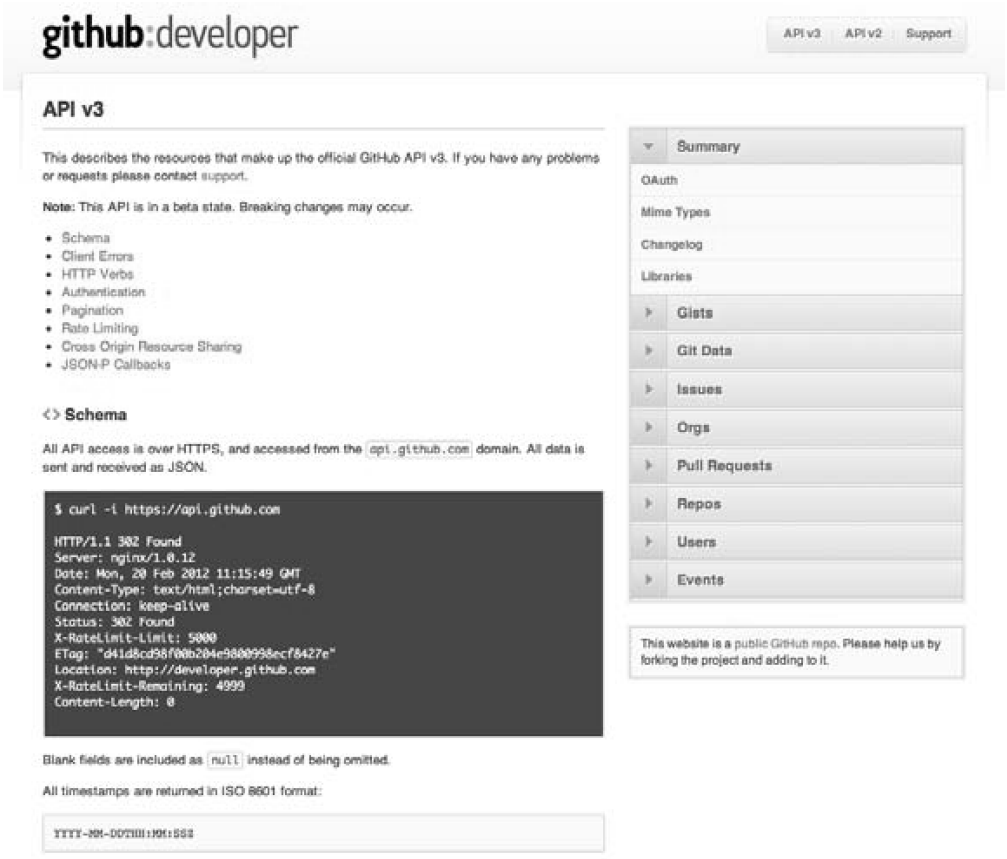


图 21-37 GitHub REST API

21.18 闭源的社会化编程

虽然像 GitHub 这种协作开发模式的来源是开源项目，但是几乎所有的特性都可以在公司内部使用。公司可以发挥每个员工的才能，即使他们没有赋予某个项目的开发权限。合并请求结合组织和团队内的拉取使得任何权限的员工都可对项目做出贡献，所需要的只是要有对代码进行审核的核心项目开发人员。

21.19 最终开放源代码

虽然很多项目一开始就开放源代码，但是越来越多的项目是经过一定时间的开发从而到达相对成熟的阶段或者在某个开发里程碑完成之后才开放源代码。这种最终开发源代码的方式会从 Git 中保留的历史记录和在 GitHub 上维护的版本库中获益。关于“为何这行代码为什么这样写”的问题可以从 Git 的提交历史记录上获得答案。同时，让项目转变为开源项目从而由 GitHub 的社会化编程中获益的操作就像在版本库管理页面中单击布尔型切换按钮这么简单（见图 21-38）。

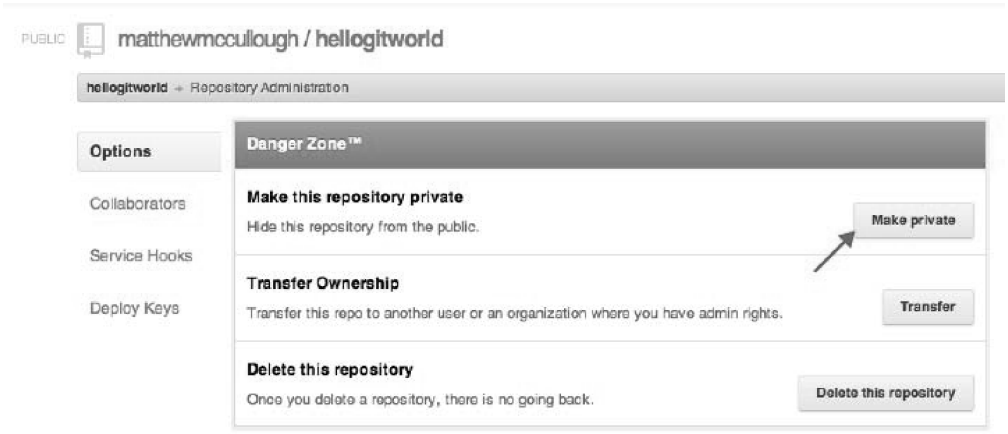


图 21-38 公开与私有版本库切换按钮

21.20 开发模型

使用 Git 作为 VCS，或者说得更详细点，使用 GitHub 托管版本库，有许多使用模式。以下简述其中的三种。

中心模型（见图 21-39）仍然提供本地提交，所以并不像 SVN 那样的真正的中心版本库。它是最简单的但最不吸引人的方式。这种简单的方式下，开发人员频繁地推送代码以保证“所有内容都在中心版本库中”的原则，就像使用以前的版本控制工具一样。虽然这是一种很容易使用 Git 的方式，但是非常不适合 Git 与 GitHub 提供的分布式和协作性模型。

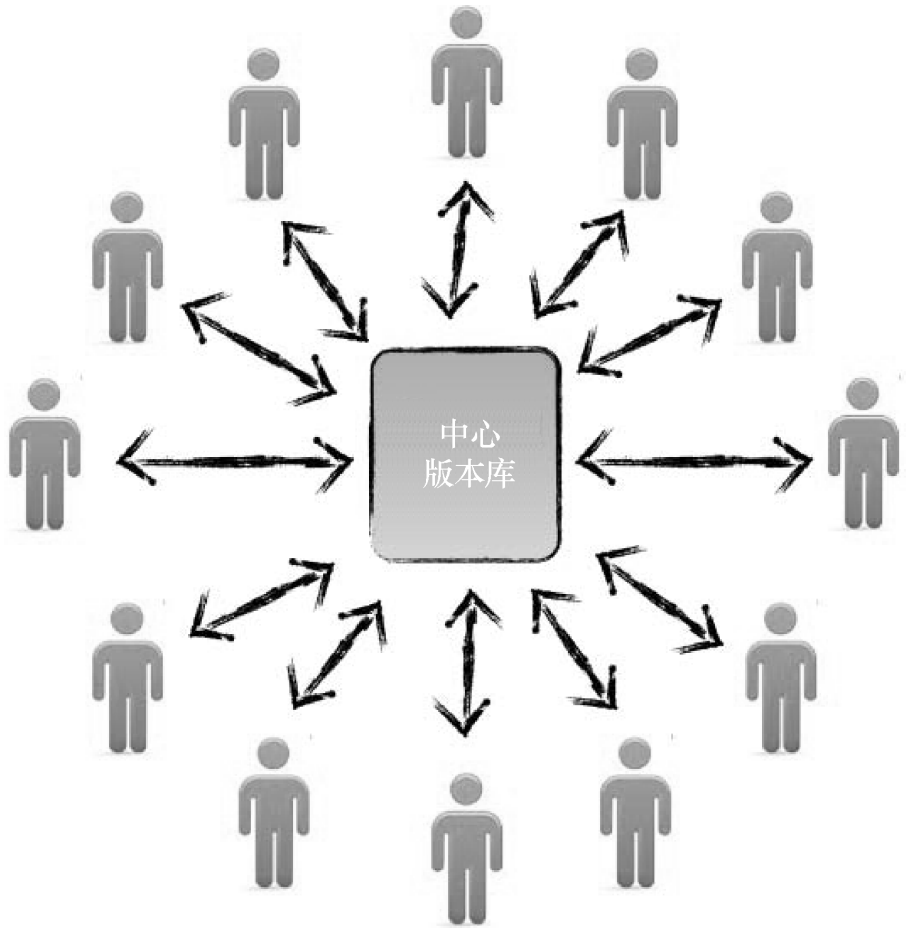


图 21-39 中心模型

下面介绍中尉指挥官模型（见图 21-40）。可以发现，这种方式很像 GitHub 提供的合并请求。需要注意的是，在没有 GitHub 之前，Git 项目正是通过邮件和链接来实现这种协作模式，但是和合并请求相比，这种方式明显很不方便。

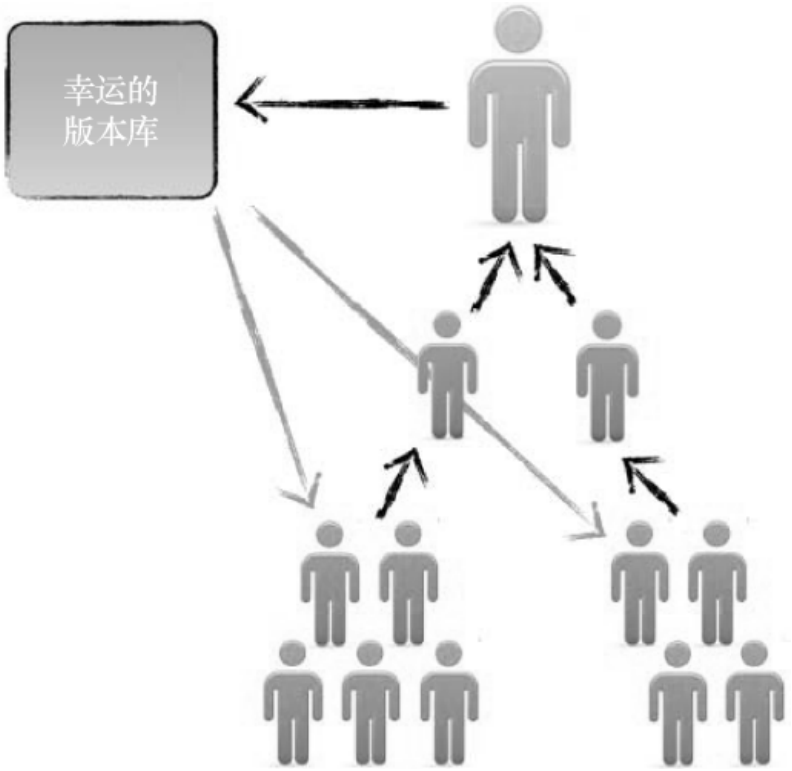


图 21-40 Linux 中尉与指挥官模型

最后一种方式是很多公司使用的开源模型。在享受到开源的优点的同时，想要把它们的 bug 修复贡献回去，但又要保持内部的革新，就要引入两个版本库的仲裁者。仲裁者（见图 21-41），挑选版本同时将它们推回到开源项目的公共区。这种方式被很多著名的项目使用（如红帽的 JBoss Server）。

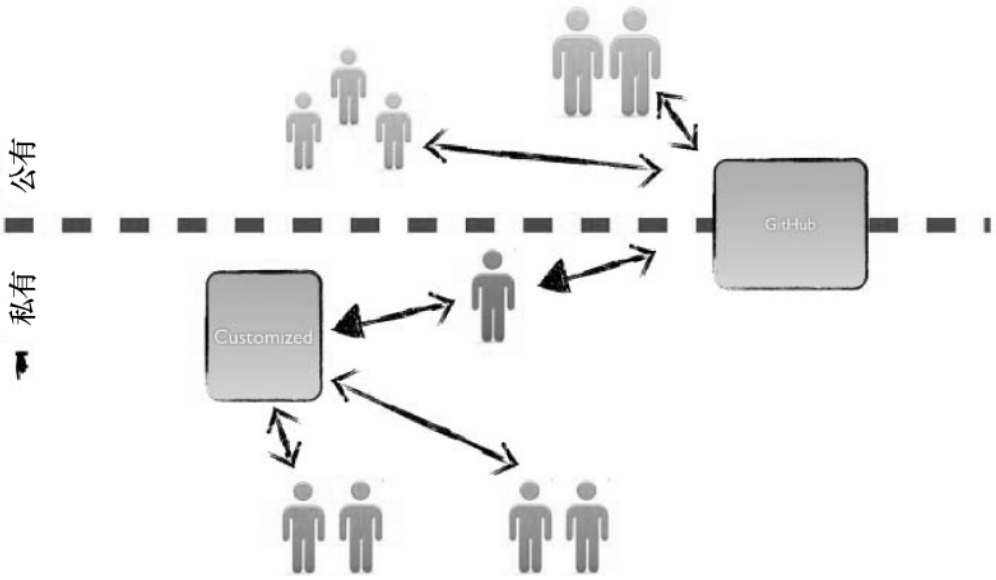


图 21-41 部分开源模型

21.21 GitHub 企业版

可能之前所述非常吸引人，但是你的公司有特殊的规定甚至法律禁止你们将代码存储在公共的互联网上，而无论安全措施如何完善。这种情况的解决方案就是 GitHub 企业版（主页如图 21-42 所示）。它提供了和公共 GitHub 相同的功能，但是以存储在企业内部主机中的虚拟机镜像形式存在（如图 21-43 中的 VirtualBox 所示）。同时，GitHub 企业版能够兼容许多企业已经使用的交换服务器轻量级目录访问协议（Lightweight Directory Access Protocol，LDAP）和集中身份验证服务（Central Authentication Service，CAS）。

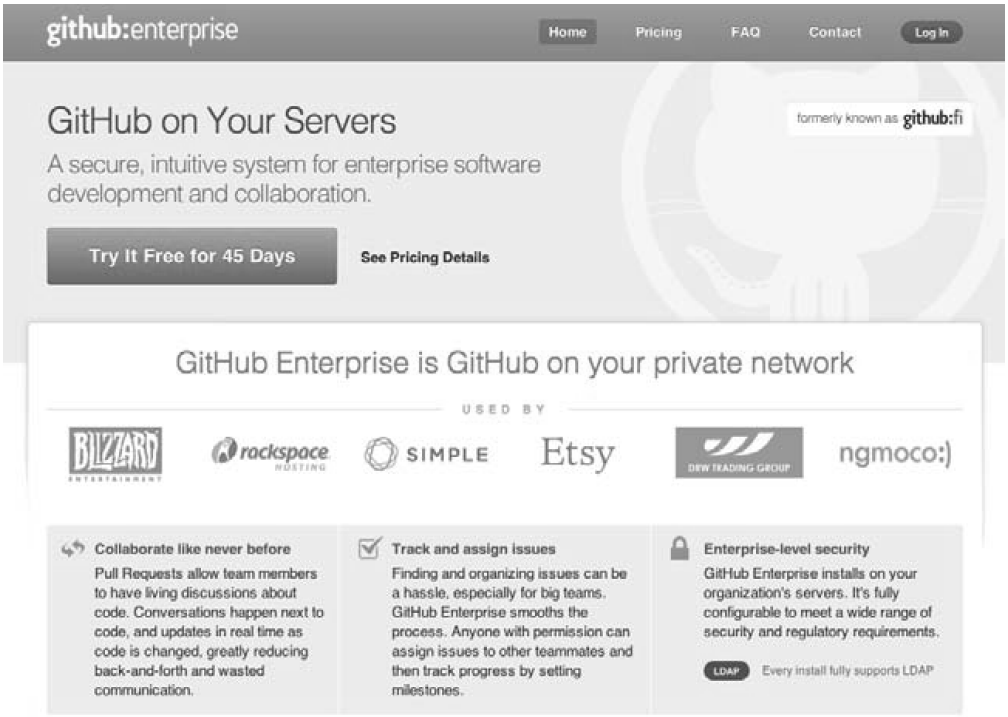


图 21-42 GitHub 企业版主页

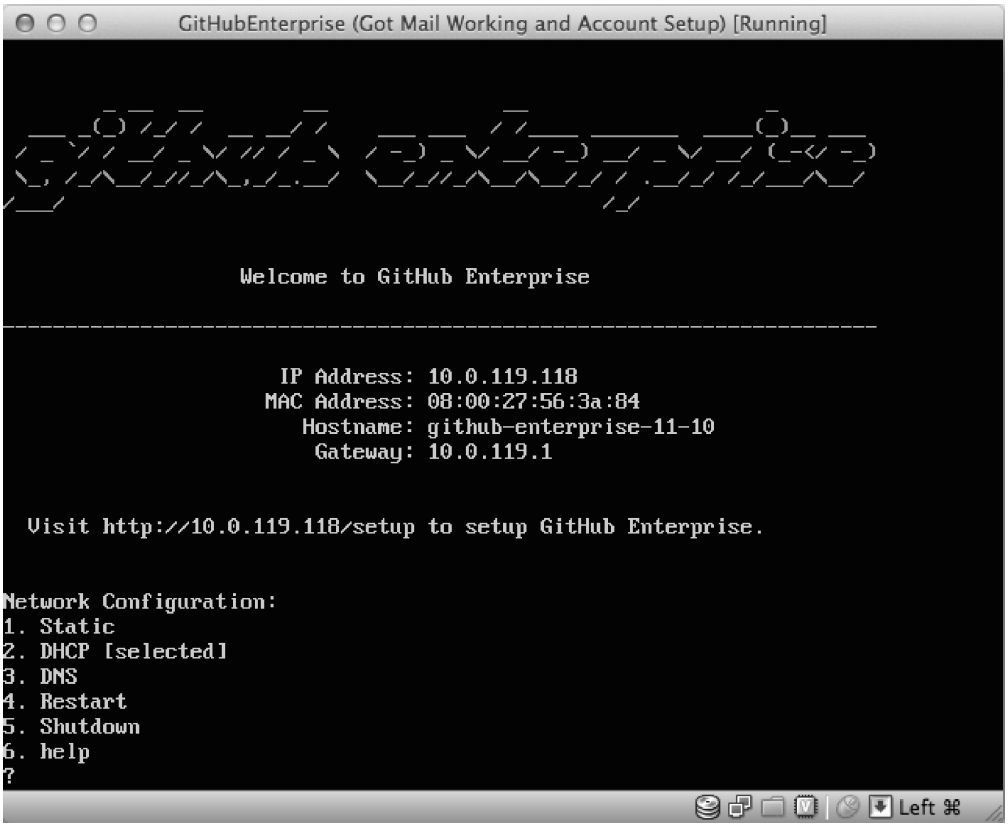


图 21-43 VirtualBox 中的 GitHub 企业版

21.22 关于 GitHub 的总结

Git 是一种版本控制工具，它已经撼动了 CVS、SVN、Perforce 以及 ClearCase 的地位。这正是因为它是高性能、协作化和分布式的开源版本控制系统。GitHub 是十分优秀的 Web 应用，它极大地减少了使用工具的负担，加快了微小修改的流程，同时允许大量开发人员对一个项目做出贡献，更重要的是，它真正使得编程成为真正社会化的活动。

Git版本控制管理

本书可以让读者迅速上手Git，对代码的变更进行跟踪、合并和管理。本书通过一系列步骤式教程，引导读者迅速掌握从Git基础知识到高级使用技巧在内的所有知识，并提供友好而严谨的建议，以帮助读者熟悉Git的许多功能。

本书在上一版的基础之上进行了全面更新，包含了操作树的技巧，全面覆盖了reflog和stash的用法，还全面介绍了GitHub。一旦你掌握了Git系统的灵活性之后，就可以用近乎无限的方式来管理代码开发，而本书则会告诉你怎么来做。

- 学习如何在多个真实的开发场景中使用Git；
- 深入理解Git的常见用例、初始任务以及基本功能；
- 针对集中式和分布式版本控制而使用Git系统；
- 学习如何管理合并、冲突、补丁和差异；
- 应用高级技术，比如变基、钩子和处理子模块的方法；
- 与SVN仓库进行交互——其中包括SVN到Git的转换；
- 通过GitHub来导航、使用开源项目，并对开源项目做贡献。

Jon Loeliger，Freescale半导体公司的软件工程师，从事开源项目的工作，比如Git、Linux和U-Boot。他在Linux World等会议上公开讲授Git，而且还经常为Linux Magazine撰写稿件。

Matthew McCullough，GitHub培训项目的VP，有15年的企业软件开发经验，而且还是一名开源教育家。Matthew是O'Reilly Git Master Class系列的创始人。

O'REILLY®
oreilly.com.cn

封面设计：Karen Montgomery，张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/软件工程及软件方法学

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-38243-6