

[返回 码云 首页\(目录\) 章节列表](#) ▾ [Pro Git \(中文版 \)](#)

4 服务器上的 Git

1. [4.1 协议](#)
2. [4.2 在服务器上部署 Git](#)
3. [4.3 生成 SSH 公钥](#)
4. [4.4 架设服务器](#)
5. [4.5 公共访问](#)
6. [4.6 GitWeb](#)
7. [4.7 Gitis](#)
8. [4.8 Gitolite](#)
9. [4.9 Git 守护进程](#)
10. [4.10 Git 托管服务](#)
11. [4.11 小结](#)

到目前为止，你应该已经学会了使用 Git 来完成日常工作。然而，如果想与他人合作，还需要一个远程的 Git 仓库。尽管技术上可以从个人的仓库里推送和拉取修改内容，但我们不鼓励这样做，因为一不留心就很容易弄混其他人的进度。另外，你也一定希望合作者们即使在自己不开机的時候也能从仓库获取数据——拥有一个更稳定的公共仓库十分有用。因此，更好的合作方式是建立一个大家都可以访问的共享仓库，从那里推送和拉取数据。我们将把这个仓库称为“Git 服务器”；代理一个 Git 仓库只需要花费很少的资源，几乎从不需要整个服务器来支持它的运行。

架设一台 Git 服务器并不难。第一步是选择与服务器通讯的协议。本章第一节将介绍可用的协议以及各自优缺点。下面一节将介绍一些针对各个协议典型的设置以及如何在服务器上实施。最后，如果你不介意在他人服务器上保存你的代码，又想免去自己架设和维护服务器的麻烦，倒可以试试我们介绍的几个仓库托管服务。

如果你对架设自己的服务器没兴趣，可以跳到本章最后一节去看看如何申请一个代码托管服务的账户然后继续下一章，我们会在那里讨论分布式源码控制环境的林林总总。

远程仓库通常只是一个裸仓库 (*bare repository*)——即一个没有当前工作目录的仓库。因为该仓库只是一个合作媒介，所以不需要从硬盘上取出最新版本的快照；仓库里存放的仅仅是 Git 的数据。简单地说，裸仓库就是你工作目录中 `.git` 子目录内的内容。

4.1 协议

Git 可以使用四种主要的协议来传输数据：本地传输，SSH 协议，Git 协议和 HTTP 协议。下面分别介绍一下哪些情形应该使用（或避免使用）这些协议。

值得注意的是，除了 HTTP 协议外，其他所有协议都要求在服务器端安装并运行 Git。

本地协议

最基本的就是本地协议 (*Local protocol*)，所谓的远程仓库在该协议中的表示，就是硬盘上的另一个目录。这常见于团队每一个成员都对一个共享的文件系统（例如 NFS）拥有访问权，或者比较少见的多人共用同一台电脑的情况。后面一种情况并不安全，因为所有代码仓库实例都储存在同一台电脑里，增加了灾难性数据损失的可能性。

如果你使用一个共享的文件系统，就可以在一个本地文件系统中克隆仓库，推送和获取。克隆的时候只需要将远程仓库的路径作为 URL 使用，比如下面这样：

```
$ git clone /opt/git/project.git
```

或者这样：

```
$ git clone file:///opt/git/project.git
```

如果在 URL 开头明确使用 `file://`，那么 Git 会以一种略微不同的方式运行。如果你只给出路径，Git 会尝试使用硬链接或直接复制它所需要的文件。如果使用了 `file://`，Git 会调用它平时通过网络来传输数据的工序，而这种方式效率相对较低。使用 `file://` 前缀的主要原因是当你需要一个不包含无关引用或对象的干净仓库副本的时候——一般指从其他版本控制系统导入的，或类似情形（参见第 9 章的维护任务）。我们这里仅仅使用普通路径，这样更快。

要添加一个本地仓库作为现有 Git 项目的远程仓库，可以这样做：

```
$ git remote add local_proj /opt/git/project.git
```

然后就可以像在网络上一样向这个远程仓库推送和获取数据了。

优点

基于文件仓库的优点在于它的简单，同时保留了现存文件的权限和网络访问权限。如果你的团队已经有一个全体共享的文件系统，建立仓库就十分容易了。你只需把一份裸仓库的副本放在大家都能访问的地方，然后像对其他共享目录一样设置读写权限就可以了。我们将在下一节“在服务器上部署 Git”中讨论如何导出一个裸仓库的副本。

这也是从别人工作目录中获取工作成果的快捷方法。假如你和你的同事在一个项目中合作，他们想让你检出一些东西的时候，运行类似 `git pull /home/john/project` 通常会比他们推送到服务器，而你再从服务器获取简单得多。

缺点

这种方法的缺点是，与基本的网络连接访问相比，难以控制从不同位置来的访问权限。如果你想从家里的笔记本电脑上推送，就要先挂载远程硬盘，这和基于网络连接的访问相比更加困难和缓慢。

另一个很重要的问题是该方法不一定就是最快的，尤其是对于共享挂载的文件系统。本地仓库只有在你对数据访问速度快的时候才快。在同一个服务器上，如果二者同时允许 Git 访问本地硬盘，通过 NFS 访问仓库通常会比 SSH 慢。

SSH 协议

Git 使用的传输协议中最常见的可能就是 SSH 了。这是因为大多数环境已经支持通过 SSH 对服务器的访问——即便还没有，架设起来也很容易。SSH 也是唯一一个同时支持读写操作的网络协议。另外两个网络协议（HTTP 和 Git）通常都是只读的，所以虽然二者对大多数人都可用，但执行写操作时还是需要 SSH。SSH 同时也是一个验证授权的网络协议；而因为其普遍性，一般架设和使用都很容易。

通过 SSH 克隆一个 Git 仓库，你可以像下面这样给出 `ssh://` 的 URL：

```
$ git clone ssh://user@server/project.git
```

或者不指明某个协议 — 这时 Git 会默认使用 SSH：

```
$ git clone user@server:project.git
```

如果不指明用户，Git 会默认使用当前登录的用户名连接服务器。

优点

使用 SSH 的好处有很多。首先，如果你想拥有对网络仓库的写权限，基本上不可能不使用 SSH。其次，SSH 架设相对比较简单 — SSH 守护进程很常见，很多网络管理员都有一些使用经验，而且很多操作系统都自带了它或者相关的管理工具。再次，通过 SSH 进行访问是安全的 — 所有数据传输都是加密和授权的。最后，和 Git 及本地协议一样，SSH 也很高效，会在传输之前尽可能压缩数据。

缺点

SSH 的限制在于你不能通过它实现仓库的匿名访问。即使仅为读取数据，人们也必须在能通过 SSH 访问主机的前提下才能访问仓库，这使得 SSH 不利于开源的项目。如果你仅仅在公司网络里使用，SSH 可能是你唯一需要使用的协议。如果想允许对项目的匿名只读访问，那么除了为自己推送而架设 SSH 协议之外，还需要支持其他协议以便他人访问读取。

Git 协议

接下来是 Git 协议。这是一个包含在 Git 软件包中的特殊守护进程；它会监听一个提供类似于 SSH 服务的特定端口（9418），而无需任何授权。打算支持 Git 协议的仓库，需要先创建 `git-export-daemon-ok` 文件 — 它是协议进程提供仓库服务的必要条件 — 但除此之外该服务没有什么安全措施。要么所有人都能克隆 Git 仓库，要么谁也不能。这也意味着该协议通常不能用来进行推送。你可以允许推送操作；然而由于没有授权机制，一旦允许该操作，网络上任何一个知道项目 URL 的人将都有推送权限。不用说，这是十分罕见的情况。

优点

Git 协议是现存最快的传输协议。如果你在提供一个有很大访问量的公共项目，或者一个不需要对读操作进行授权的庞大项目，架设一个 Git 守护进程来供应仓库是个不错的选择。它使用与 SSH 协议相同的数据传输机制，但省去了加密和授权的开销。

缺点

Git 协议消极的一面是缺少授权机制。用 Git 协议作为访问项目的唯一方法通常是不可取的。一般的做法是，同时提供 SSH 接口，让几个开发者拥有推送（写）权限，其他人通过 `git://` 拥有只读权限。Git 协议可能也是最难架设的协议。它要求有单独的守护进程，需要定制 — 我们将在本章的“Gitosis”一节详细介绍它的架设 — 需要设定 `xinetd` 或类似的程序，而这些工作就没那么轻松了。该协议还要求防火墙开放 9418 端口，而企业级防火墙一般不允许对这个非标准端口的访问。大型企业级防火墙通常会封锁这个少见的端口。

HTTP/S 协议

最后还有 HTTP 协议。HTTP 或 HTTPS 协议的优美之处在于架设的简便性。基本上，只需要把 Git 的裸仓库文件放在 HTTP 的根目录下，配置一个特定的 `post-update` 挂钩（hook）就可以搞定（Git 挂钩的细节见第 7 章）。此后，每个能访问 Git 仓库所在服务器上 web 服务的人都可以进行克隆操作。下面的操作可以允许通过 HTTP 对仓库进行读取：

```
$ cd /var/www/htdocs/  
$ git clone --bare /path/to/git_project gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```

这样就可以了。Git 附带的 `post-update` 挂钩会默认运行合适的命令（`git update-server-info`）来确保通过 HTTP 的获取和克隆正常工作。这条命令在你用 SSH 向仓库推送内容时运行；之后，其他人就可以用下面的命令来克隆仓库：

```
$ git clone http://example.com/gitproject.git
```

在本例中，我们使用了 Apache 设定中常用的 `/var/www/htdocs` 路径，不过你可以使用任何静态 web 服务——把裸仓库放在它的目录里就行。Git 的数据是以最基本的静态文件的形式提供的（关于如何提供文件的详情见第 9 章）。

通过 HTTP 进行推送操作也是可能的，不过这种做法不太常见，并且牵扯到复杂的 WebDAV 设定。由于很少用到，本书将略过对该内容的讨论。如果对 HTTP 推送协议感兴趣，不妨打开这个地址看一下操作方法：<http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt>。通过 HTTP 推送的好处之一是你可以使用任何 WebDAV 服务器，不需要为 Git 设定特殊环境；所以如果主机提供商支持通过 WebDAV 更新网站内容，你也可以使用这项功能。

优点

使用 HTTP 协议的好处是易于架设。几条必要的命令就可以让全世界读取到仓库的内容。花费不过几分钟。HTTP 协议不会占用过多服务器资源。因为它一般只用到静态的 HTTP 服务提供所有数据，普通的 Apache 服务器平均每秒能支撑数千个文件的并发访问——哪怕让一个小型服务器超载都很难。

你也可以通过 HTTPS 提供只读的仓库，这意味着你可以加密传输内容；你甚至可以要求客户端使用特定签名的 SSL 证书。一般情况下，如果到了这一步，使用 SSH 公共密钥可能是更简单的方案；不过也存在一些特殊情况，这时通过 HTTPS 使用带签名的 SSL 证书或者其他基于 HTTP 的只读连接授权方式是更好的解决方案。

HTTP 还有个额外的好处：HTTP 是一个如此常见的协议，以至于企业级防火墙通常都允许其端口的通信。

缺点

HTTP 协议的消极面在于，相对来说客户端效率更低。克隆或者下载仓库内容可能会花费更多时间，而且 HTTP 传输的体积和网络开销比其他任何一个协议都大。因为它没有按需供应的能力——传输过程中没有服务端的动态计算——因而 HTTP 协议经常会被称为傻瓜（*dumb*）协议。更多 HTTP 协议和其他协议效率上的差异见第 9 章。

4.2 在服务器上部署 Git

开始架设 Git 服务器前，需要先把现有仓库导出为裸仓库 — 即一个不包含当前工作目录的仓库。做法直截了当，克隆时用 `--bare` 选项即可。裸仓库的目录名一般以 `.git` 结尾，像这样：

```
$ git clone --bare my_project my_project.git
Initialized empty Git repository in /opt/projects/my_project.git/
```

该命令的输出或许会让人有些不解。其实 `clone` 操作基本上相当于 `git init` 加 `git fetch`，所以这里出现的其实是 `git init` 的输出，先由它建立一个空目录，而之后传输数据对象的操作并无任何输出，只是悄悄在幕后执行。现在 `my_project.git` 目录中已经有了一份 Git 目录数据的副本。

整体上的效果大致相当于：

```
$ cp -Rf my_project/.git my_project.git
```

但在配置文件中有若干小改动，不过对用户来讲，使用方式都一样，不会有什么影响。它仅取出 Git 仓库的必要原始数据，存放在该目录中，而不会另外创建工作目录。

把裸仓库移到服务器上

有了裸仓库的副本后，剩下的就是把它放到服务器上并设定相关协议。假设一个域名为 `git.example.com` 的服务器已经架设好，并可以通过 SSH 访问，我们打算把所有 Git 仓库储存在 `/opt/git` 目录下。只要把裸仓库复制过去：

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

现在，所有对该服务器有 SSH 访问权限，并可读取 `/opt/git` 目录的用户都可以用下面的命令克隆该项目：

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

如果某个 SSH 用户对 `/opt/git/my_project.git` 目录有写权限，那他就有推送权限。如果到该项目目录中运行 `git init` 命令，并加上 `--shared` 选项，那么 Git 会自动修改该仓库目录的组权限为可写（译注：实际上 `--shared` 可以指定其他行为，只是默认为将组权限改为可写并执行 `g+sx`，所以最后会得到 `rws`。 ）。

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

由此可见，根据现有的 Git 仓库创建一个裸仓库，然后把它放上你和同事都有 SSH 访问权的服务器是多么容易。现在已经可以开始在同一项目上密切合作了。

值得注意的是，这的确确实是架设一个少数人具有连接权的 Git 服务的全部 — 只要在服务器上加入可以用 SSH 登录的帐号，然后把裸仓库放在大家都有读写权限的地方。一切都准备妥当，无需更多。

下面的几节中，你会了解如何扩展到更复杂的设定。这些内容包含如何避免为每一个用户建立一个账户，给仓库添加公共读取权限，架设网页界面，使用 Gitosis 工具等等。然而，只是和几个人在一个不公开的项目上合作的话，仅仅是一个 SSH 服务器和裸仓库就足够了，记住这点就可以了。

小型安装

如果设备较少或者你只想在小型开发团队里尝试 Git，那么一切都很简单。架设 Git 服务最复杂的地方在于账户管理。如果需要仓库对特定的用户可读，而给另一部分用户读写权限，那么访问和许可的安排就比较困难。

SSH 连接

如果已经有了一个所有开发成员都可以用 SSH 访问的服务器，架设第一个服务器将变得异常简单，几乎什么都不用做（正如上节中介绍的那样）。如果需要对仓库进行更复杂的访问控制，只要使用服务器操作系统的本地文件访问许可机制就行了。

如果需要团队里的每个人都对仓库有写权限，又不能给每个人在服务器上建立账户，那么提供 SSH 连接就是唯一的选择了。我们假设用来共享仓库的服务器已经安装了 SSH 服务，而且你通过它访问服务器。

有好几个办法可以让团队的每个人都有访问权。第一个办法是给每个人建立一个账户，直截了当但略过繁琐。反复运行 `adduser` 并给所有人设定临时密码可不是好玩的。

第二个办法是在主机上建立一个 `git` 账户，让每个需要写权限的人发送一个 SSH 公钥，然后将其加入 `git` 账户的 `~/.ssh/authorized_keys` 文件。这样一来，所有人都将通过 `git` 账户访问主机。这丝毫不会影响提交的数据——访问主机用的身份不会影响提交对象的提交者信息。

另一个办法是让 SSH 服务器通过某个 LDAP 服务，或者其他已经设定好的集中授权机制，来进行授权。只要每个人都能获得主机的 shell 访问权，任何可用的 SSH 授权机制都能达到相同效果。

4.3 生成 SSH 公钥

大多数 Git 服务器都会选择使用 SSH 公钥来进行授权。系统中的每个用户都必须提供一个公钥用于授权，没有的话就要生成一个。生成公钥的过程在所有操作系统上都差不多。首先先确认一下是否已经有一个公钥了。SSH 公钥默认储存在账户的主目录下的 `~/.ssh` 目录。进去看看：

```
$ cd ~/.ssh
$ ls
authorized_keys2 id_dsa known_hosts
config id_dsa.pub
```

关键是看有没有用 `something` 和 `something.pub` 来命名的一对文件，这个 `something` 通常就是 `id_dsa` 或 `id_rsa`。有 `.pub` 后缀的文件就是公钥，另一个文件则是密钥。假如没有这些文件，或者干脆连 `.ssh` 目录都没有，可以用 `ssh-keygen` 来创建。该程序在 Linux/Mac 系统上由 SSH 包提供，而在 Windows 上则包含在 MSysGit 包里：

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
```

```
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

它先要求你确认保存公钥的位置（`.ssh/id_rsa`），然后它会让你重复一个密码两次，如果不想在使用公钥的时候输入密码，可以留空。

现在，所有做过这一步的用户都得把它们的公钥给你或者 Git 服务器的管理员（假设 SSH 服务被设定为使用公钥机制）。他们只需要复制 `.pub` 文件的内容然后发邮件给管理员。公钥的样子大致如下：

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpKDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GP1+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7x1ELEVF4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyB1WXFRCr+HAo3FXRitBqxiXlnKhXpHAZsMcilQ8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncMlQ9x5+3V0Ww68/eIFmb1zuUF1jQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+lnKatmIkjn2sold0lQraTlMqVSSbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

关于在多个操作系统上设立相同 SSH 公钥的教程，可以查阅 GitHub 上有关 SSH 公钥的向导：

<http://github.com/guides/providing-your-ssh-key>。

4.4 架设服务器

现在我们过一边服务器端架设 SSH 访问的流程。本例将使用 `authorized_keys` 方法来给用户授权。我们还将假定使用类似 Ubuntu 这样的标准 Linux 发行版。首先，创建一个名为 'git' 的用户，并为其创建一个 `.ssh` 目录。

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

接下来，把开发者的 SSH 公钥添加到这个用户的 `authorized_keys` 文件中。假设你通过电邮收到了几个公钥并存到了临时文件里。重复一下，公钥大致看起来是这个样子：

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x41hJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
Yjh6541NysnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIcTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBdLQlqMV0Fq1I2uPWQ0k0WQAHuKE0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

只要把它们逐个追加到 `authorized_keys` 文件尾部即可：

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

现在可以用 `--bare` 选项运行 `git init` 来建立一个裸仓库，这会初始化一个不包含工作目录的仓库。

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

这时，Join，Josie 或者 Jessica 就可以把它加为远程仓库，推送一个分支，从而把第一个版本的项目文件上传到仓库里了。值得注意的是，每次添加一个新项目都需要通过 shell 登入主机并创建一个裸仓库目录。我们不妨以 gitserver 作为 git 用户及项目仓库所在的主机名。如果在网络内部运行该主机，并在 DNS 中设定 gitserver 指向该主机，那么以下这些命令都是可用的：

```
# 在 John 的电脑上
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

这样，其他人的克隆和推送也一样变得很简单：

```
$ git clone git@gitserver:/opt/git/project.git
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

用这个方法可以很快捷地为少数几个开发者架设一个可读写的 Git 服务。

作为一个额外的防范措施，你可以用 Git 自带的 git-shell 工具限制 git 用户的活动范围。只要把它设为 git 用户登入的 shell，那么该用户就无法使用普通的 bash 或者 csh 什么的 shell 程序。编辑 /etc/passwd 文件：

```
$ sudo vim /etc/passwd
```

在文件末尾，你应该能找到类似这样的行：

```
git:x:1000:1000:./home/git:/bin/sh
```

把 bin/sh 改为 /usr/bin/git-shell（或者用 which git-shell 查看它的实际安装路径）。该行修改后的样子如下：

```
git:x:1000:1000:./home/git:/usr/bin/git-shell
```

现在 git 用户只能用 SSH 连接来推送和获取 Git 仓库，而不能直接使用主机 shell。尝试普通 SSH 登录的话，会看到下面这样的拒绝信息：

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

4.5 公共访问

匿名的读取权限该怎么实现呢？也许除了内部私有的项目之外，你还需要托管一些开源项目。或者因为要用一些自动化的服务器来进行编译，或者有一些经常变化的服务器群组，而又不想整天生成新的 SSH 密钥——总之，你需要简单的匿名读取权限。

或许对小型的配置来说最简单的办法就是运行一个静态 web 服务，把它的根目录设定为 Git 仓库所在的位置，然后开启本章第一节提到的 `post-update` 挂钩。这里继续使用之前的例子。假设仓库处于 `/opt/git` 目录，主机上运行着 Apache 服务。重申一下，任何 web 服务程序都可以达到相同效果；作为范例，我们将用一些基本的 Apache 设定来展示大体需要的步骤。

首先，开启挂钩：

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

如果用的是 Git 1.6 之前的版本，则可以省略 `mv` 命令 — Git 是从较晚的版本才开始在挂钩实例的结尾添加 `.sample` 后缀名的。

`post-update` 挂钩是做什么的呢？其内容大致如下：

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

意思是当通过 SSH 向服务器推送时，Git 将运行这个 `git-update-server-info` 命令来更新匿名 HTTP 访问获取数据时所需要的文件。

接下来，在 Apache 配置文件中添加一个 `VirtualHost` 条目，把文档根目录设为 Git 项目所在的根目录。这里我们假定 DNS 服务已经配置好，会把对 `.gitserver` 的请求发送到这台主机：

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

另外，需要把 `/opt/git` 目录的 Unix 用户组设定为 `www-data`，这样 web 服务才可以读取仓库内容，因为运行 CGI 脚本的 Apache 实例进程默认就是以该用户的身份起来的：

```
$ chgrp -R www-data /opt/git
```

重启 Apache 之后，就可以通过项目的 URL 来克隆该目录下的仓库了。

```
$ git clone http://git.gitserver/project.git
```

这一招可以让你在几分钟内为相当数量的用户架设好基于 HTTP 的读取权限。另一个提供非授权访问的简单方法是开启一个 Git 守护进程，不过这将要求该进程作为后台进程常驻 — 接下来的这一节就要讨论这方面的细节。

4.6 GitWeb

现在我们的项目已经有了可读可写和只读的连接方式，不过如果能有一个简单的 web 界面访问就更好了。Git 自带一个叫做 GitWeb 的 CGI 脚本，运行效果可以到 <http://git.kernel.org> 这样的站点

体验下（见图 4-1）。



Figure 4-1. 基于网页的 GitWeb 用户界面

如果想看看自己项目的效果，不妨用 Git 自带的一个命令，可以使用类似 `lighttpd` 或 `webrick` 这样轻量级的服务器启动一个临时进程。如果是在 Linux 主机上，通常都预装了 `lighttpd`，可以到项目目录中键入 `git instaweb` 来启动。如果用的是 Mac，Leopard 预装了 Ruby，所以 `webrick` 应该是最好的选择。如果要用 `lighttpd` 以外的程序来启动 `git instaweb`，可以通过 `--httpd` 选项指定：

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO WEBrick 1.3.1
[2009-02-21 10:02:21] INFO ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

这会在 1234 端口开启一个 HTTPD 服务，随之在浏览器中显示该页，十分简单。关闭服务时，只需在原来的命令后面加上 `--stop` 选项就可以了：

```
$ git instaweb --httpd=webrick --stop
```

如果需要为团队或者某个开源项目长期运行 GitWeb，那么 CGI 脚本就要由正常的网页服务来运行。一些 Linux 发行版可以通过 `apt` 或 `yum` 安装一个叫做 `gitweb` 的软件包，不妨首先尝试一下。我们将快速介绍一下手动安装 GitWeb 的流程。首先，你需要 Git 的源码，其中带有 GitWeb，并能生成定制的 CGI 脚本：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
  prefix=/usr gitweb/gitweb.cgi
$ sudo cp -Rf gitweb /var/www/
```

注意，通过指定 `GITWEB_PROJECTROOT` 变量告诉编译命令 Git 仓库的位置。然后，设置 Apache 以 CGI 方式运行该脚本，添加一个 VirtualHost 配置：

```
<VirtualHost *:80>
    ServerName gitserver
```

```
DocumentRoot /var/www/gitweb
<Directory /var/www/gitweb>
Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
AllowOverride All
order allow,deny
Allow from all
AddHandler cgi-script cgi
DirectoryIndex gitweb.cgi
</Directory>
</VirtualHost>
```

不难想象，GitWeb 可以使用任何兼容 CGI 的网页服务来运行；如果偏向使用其他 web 服务器，配置也不会很麻烦。现在，通过 `http://gitserver` 就可以在线访问仓库了，在 `http://git.server` 上还可以通过 HTTP 克隆和获取仓库的内容。

4.7 Gitis

把所有用户的公钥保存在 `authorized_keys` 文件的做法，只能凑和一阵子，当用户数量达到几百人的规模时，管理起来就会十分痛苦。每次改删用户都必须登录服务器不去说，这种做法还缺少必要的权限管理——每个人都对所有项目拥有完整的读写权限。

幸好我们还可以选择应用广泛的 Gitis 项目。简单地说，Gitis 就是一套用来管理 `authorized_keys` 文件和实现简单连接限制的脚本。有趣的是，用来添加用户和设定权限的并非通过网页程序，而只是管理一个特殊的 Git 仓库。你只需要在这个特殊仓库内做好相应的设定，然后推送到服务器上，Gitis 就会随之改变运行策略，听起来就很酷，对吧？

Gitis 的安装算不上傻瓜化，但也不算太难。用 Linux 服务器架设起来最简单——以下例子中，我们使用装有 Ubuntu 8.10 系统的服务器。

Gitis 的工作依赖于某些 Python 工具，所以首先要安装 Python 的 `setuptools` 包，在 Ubuntu 上称为 `python-setuptools`：

```
$ apt-get install python-setuptools
```

接下来，从 Gitis 项目主页克隆并安装：

```
$ git clone git://eagain.net/gitis.git
$ cd git
$ sudo python setup.py install
```

这会安装几个供 Gitis 使用的工具。默认 Gitis 会把 `/home/git` 作为存储所有 Git 仓库的根目录，这没什么不好，不过我们之前已经把项目仓库都放在 `/opt/git` 里面了，所以为方便起见，我们可以做一个符号连接，直接划转过去，而不必重新配置：

```
$ ln -s /opt/git /home/git/repositories
```

Gitis 将会帮我们管理用户公钥，所以先把当前控制文件改名备份，以便稍后重新添加，准备好让 Gitis 自动管理 `authorized_keys` 文件：

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

接下来，如果之前把 git 用户的登录 shell 改为 git-shell 命令的话，先恢复 'git' 用户的登录 shell。改过之后，大家仍然无法通过该帐号登录（译注：因为 authorized_keys 文件已经没有了。），不过不用担心，这会交给 GitoSis 来实现。所以现在先打开 /etc/passwd 文件，把这行：

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

改回：

```
git:x:1000:1000::/home/git:/bin/sh
```

好了，现在可以初始化 GitoSis 了。你可以用自己的公钥执行 gitosis-init 命令，要是公钥不在服务器上，先临时复制一份：

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

这样该公钥的拥有者就能修改用于配置 GitoSis 的那个特殊 Git 仓库了。接下来，需要手工对该仓库中的 post-update 脚本加上可执行权限：

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

基本上就算是好了。如果设定过程没出什么差错，现在可以试一下用初始化 GitoSis 的公钥的拥有者身份 SSH 登录服务器，应该会看到类似下面这样：

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
Connection to gitserver closed.
```

说明 GitoSis 认出了该用户的身份，但由于没有运行任何 Git 命令，所以它切断了连接。那么，现在运行一个实际的 Git 命令 — 克隆 GitoSis 的控制仓库：

```
# 在你本地计算机上
$ git clone git@gitserver:gitosis-admin.git
```

这会得到一个名为 gitosis-admin 的工作目录，主要由两部分组成：

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

gitosis.conf 文件是用来设置用户、仓库和权限的控制文件。keydir 目录则是保存所有具有访问权限用户公钥的地方 — 每人一个。在 keydir 里的文件名（比如上面的 scott.pub）应该跟你的不一样 — GitoSis 会自动从使用 gitosis-init 脚本导入的公钥尾部的描述中获取该名字。

看一下 gitosis.conf 文件的内容，它应该只包含与刚刚克隆的 gitosis-admin 相关的信息：

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = scott
```

它显示用户 `scott` — 初始化 Gitosis 公钥的拥有者 — 是唯一能管理 `gitosis-admin` 项目的人。

现在我们来添加一个新项目。为此我们要建立一个名为 `mobile` 的新段落，在其中罗列手机开发团队的开发者，以及他们拥有写权限的项目。由于 `'scott'` 是系统中的唯一用户，我们把他设为唯一用户，并允许他读写名为 `iphone_project` 的新项目：

```
[group mobile]
writable = iphone_project
members = scott
```

修改完之后，提交 `gitosis-admin` 里的改动，并推送到服务器使其生效：

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitosis-admin.git
fb27aec..8962da8 master -> master
```

在新工程 `iphone_project` 里首次推送数据到服务器前，得先设定该服务器地址为远程仓库。但你不用事先到服务器上手工创建该项目的裸仓库 — Gitosis 会在第一次遇到推送时自动创建：

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
* [new branch] master -> master
```

请注意，这里不用指明完整路径（实际上，如果加上反而没用），只需要一个冒号加项目名字即可 — Gitosis 会自动帮你映射到实际位置。

要和朋友们在一个项目上协同工作，就得重新添加他们的公钥。不过这次不用在服务器上一个一个手工添加到 `~/.ssh/authorized_keys` 文件末端，而只需管理 `keydir` 目录中的公钥文件。文件的命名将决定在 `gitosis.conf` 中对用户的标识。现在我们为 John，Josie 和 Jessica 添加公钥：

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

然后把他们都加进 `'mobile'` 团队，让他们对 `iphone_project` 具有读写权限：

```
[group mobile]
writable = iphone_project
```



```
members = scott john josie jessica
```

如果你提交并推送这个修改，四个用户将同时具有该项目的读写权限。

Gitosis 也具有简单的访问控制功能。如果想让 John 只有读权限，可以这样做：

```
[group mobile]
writable = iphone_project
members = scott josie jessica

[group mobile_ro]
readonly = iphone_project
members = john
```

现在 John 可以克隆和获取更新，但 Gitosis 不会允许他向项目推送任何内容。像这样的组可以随意创建，多少不限，每个都可以包含若干不同的用户和项目。甚至还可以指定某个组为成员之一（在组名前加上 @ 前缀），自动继承该组的成员：

```
[group mobile_committers]
members = scott josie jessica

[group mobile]
writable = iphone_project
members = @mobile_committers

[group mobile_2]
writable = another_iphone_project
members = @mobile_committers john
```

如果遇到意外问题，试试看把 `loglevel=DEBUG` 加到 `[gitosis]` 的段落（译注：把日志设置为调试级别，记录更详细的运行信息。）。如果一不小心搞错了配置，失去了推送权限，也可以手工修改服务器上的 `/home/git/.gitosis.conf` 文件 — Gitosis 实际是从该文件读取信息的。它在得到推送数据时，会把新的 `gitosis.conf` 存到该路径上。所以如果你手工编辑该文件的话，它会一直保持到下次向 `gitosis-admin` 推送新版本的配置内容为止。

4.8 Gitolite

Note: the latest copy of this section of the ProGit book is always available within the [gitolite documentation](#). The author would also like to humbly state that, while this section is accurate, and *can* (and often *has*) been used to install gitolite without reading any other documentation, it is of necessity not complete, and cannot completely replace the enormous amount of documentation that gitolite comes with.

Git has started to become very popular in corporate environments, which tend to have some additional requirements in terms of access control. Gitolite was originally created to help with those requirements, but it turns out that it's equally useful in the open source world: the Fedora Project controls access to their package management repositories (over 10,000 of them!) using gitolite, and this is probably the largest gitolite installation anywhere too.

Gitolite allows you to specify permissions not just by repository, but also by branch or tag names within each repository. That is, you can specify that certain people (or groups of people) can only push certain "refs" (branches or tags) but not others.

Installing

Installing Gitolite is very easy, even if you don't read the extensive documentation that comes with it. You need an account on a Unix server of some kind; various Linux flavours, and Solaris 10, have been tested. You do not need root access, assuming git, perl, and an openssh compatible ssh server are already installed. In the examples below, we will use the `gitolite` account on a host called `gitserver`.

Gitolite is somewhat unusual as far as "server" software goes -- access is via ssh, and so every userid on the server is a potential "gitolite host". As a result, there is a notion of "installing" the software itself, and then "setting up" a user as a "gitolite host".

Gitolite has 4 methods of installation. People using Fedora or Debian systems can obtain an RPM or a DEB and install that. People with root access can install it manually. In these two methods, any user on the system can then become a "gitolite host".

People without root access can install it within their own userids. And finally, gitolite can be installed by running a script *on the workstation*, from a bash shell. (Even the bash that comes with `msysgit` will do, in case you're wondering.)

We will describe this last method in this article; for the other methods please see the documentation.

You start by obtaining public key based access to your server, so that you can log in from your workstation to the server without getting a password prompt. The following method works on Linux; for other workstation OSs you may have to do this manually. We assume you already had a key pair generated using `ssh-keygen`.

```
$ ssh-copy-id -i ~/.ssh/id_rsa gitolite@gitserver
```

This will ask you for the password to the gitolite account, and then set up public key access. This is **essential** for the install script, so check to make sure you can run a command without getting a password prompt:

```
$ ssh gitolite@gitserver pwd
/home/gitolite
```

Next, you clone Gitolite from the project's main site and run the "easy install" script (the third argument is your name as you would like it to appear in the resulting gitolite-admin repository):

```
$ git clone git://github.com/sitaramc/gitolite
$ cd gitolite/src
$ ./gl-easy-install -q gitolite gitserver sitaram
```

And you're done! Gitolite has now been installed on the server, and you now have a brand new repository called `gitolite-admin` in the home directory of your workstation. You administer your gitolite setup by making changes to this repository and pushing.

That last command does produce a fair amount of output, which might be interesting to read. Also, the first time you run this, a new keypair is created; you will have to choose a

passphrase or hit enter for none. Why a second keypair is needed, and how it is used, is explained in the "ssh troubleshooting" document that comes with Gitolite. (Hey the documentation has to be good for *something*!)

Repos named `gitolite-admin` and `testing` are created on the server by default. If you wish to clone either of these locally (from an account that has SSH console access to the gitolite account via *authorized_keys*), type:

```
$ git clone gitolite:gitolite-admin
$ git clone gitolite:testing
```

To clone these same repos from any other account:

```
$ git clone gitolite@servername:gitolite-admin
$ git clone gitolite@servername:testing
```

Customising the Install

While the default, quick, install works for most people, there are some ways to customise the install if you need to. If you omit the `-q` argument, you get a "verbose" mode install -- detailed information on what the install is doing at each step. The verbose mode also allows you to change certain server-side parameters, such as the location of the actual repositories, by editing an "rc" file that the server uses. This "rc" file is liberally commented so you should be able to make any changes you need quite easily, save it, and continue. This file also contains various settings that you can change to enable or disable some of gitolite's advanced features.

Config File and Access Control Rules

Once the install is done, you switch to the `gitolite-admin` repository (placed in your HOME directory) and poke around to see what you got:

```
$ cd ~/gitolite-admin/
$ ls
conf/ keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/sitaram.pub
$ cat conf/gitolite.conf
#gitolite conf
# please see conf/example.conf for details on syntax and features

repo gitolite-admin
RW+ = sitaram

repo testing
RW+ = @all
```

Notice that "sitaram" (the last argument in the `gl-easy-install` command you gave earlier) has read-write permissions on the `gitolite-admin` repository as well as a public key file of the same name.

The config file syntax for gitolite is liberally documented in `conf/example.conf`, so we'll only mention some highlights here.

You can group users or repos for convenience. The group names are just like macros; when defining them, it doesn't even matter whether they are projects or users; that distinction is only made when you *use* the "macro".

```
@oss_repos = linux perl rakudo git gitolite
@secret_repos = fenestra pear

@admins = scott # Adams, not Chacon, sorry :)
@interns = ashok # get the spelling right, Scott!
@engineers = sitaram dilbert wally alice
@staff = @admins @engineers @interns
```

You can control permissions at the "ref" level. In the following example, interns can only push the "int" branch. Engineers can push any branch whose name starts with "eng-", and tags that start with "rc" followed by a digit. And the admins can do anything (including rewind) to any ref.

```
repo @oss_repos
  RW int$ = @interns
  RW eng- = @engineers
  RW refs/tags/rc[0-9] = @engineers
  RW+ = @admins
```

The expression after the `RW` or `RW+` is a regular expression (regex) that the refname (ref) being pushed is matched against. So we call it a "refex"! Of course, a refex can be far more powerful than shown here, so don't overdo it if you're not comfortable with perl regexes.

Also, as you probably guessed, Gitolite prefixes `refs/heads/` as a syntactic convenience if the refex does not begin with `refs/`.

An important feature of the config file's syntax is that all the rules for a repository need not be in one place. You can keep all the common stuff together, like the rules for all `oss_repos` shown above, then add specific rules for specific cases later on, like so:

```
repo gitolite
  RW+ = sitaram
```

That rule will just get added to the ruleset for the `gitolite` repository.

At this point you might be wondering how the access control rules are actually applied, so let's go over that briefly.

There are two levels of access control in gitolite. The first is at the repository level; if you have read (or write) access to *any* ref in the repository, then you have read (or write) access to the repository.

The second level, applicable only to "write" access, is by branch or tag within a repository. The username, the access being attempted (`w` or `+`), and the refname being updated are known. The access rules are checked in order of appearance in the config file, looking for a match for this combination (but remember that the refname is regex-matched, not merely string-matched). If a match is found, the push succeeds. A fallthrough results in access being denied.

Advanced Access Control with "deny" rules

So far, we've only seen permissions to be one or `R`, `RW`, or `RW+`. However, gitolite allows another permission: `-`, standing for "deny". This gives you a lot more power, at the expense of some complexity, because now fallthrough is not the *only* way for access to be denied, so the *order of the rules now matters*!

Let us say, in the situation above, we want engineers to be able to rewind any branch *except* master and integ. Here's how to do that:

```
RW master integ = @engineers
- master integ = @engineers
RW+ = @engineers
```

Again, you simply follow the rules top down until you hit a match for your access mode, or a deny. Non-rewind push to master or integ is allowed by the first rule. A rewind push to those refs does not match the first rule, drops down to the second, and is therefore denied. Any push (rewind or non-rewind) to refs other than master or integ won't match the first two rules anyway, and the third rule allows it.

Restricting pushes by files changed

In addition to restricting what branches a user can push changes to, you can also restrict what files they are allowed to touch. For example, perhaps the Makefile (or some other program) is really not supposed to be changed by just anyone, because a lot of things depend on it or would break if the changes are not done *just right*. You can tell gitolite:

```
repo foo
  RW = @junior_devs @senior_devs

  RW NAME/ = @senior_devs
  - NAME/Makefile = @junior_devs
  RW NAME/ = @junior_devs
```

This powerful feature is documented in `conf/example.conf`.

Personal Branches

Gitolite also has a feature called "personal branches" (or rather, "personal branch namespace") that can be very useful in a corporate environment.

A lot of code exchange in the git world happens by "please pull" requests. In a corporate environment, however, unauthenticated access is a no-no, and a developer workstation cannot do authentication, so you have to push to the central server and ask someone to pull from there.

This would normally cause the same branch name clutter as in a centralised VCS, plus setting up permissions for this becomes a chore for the admin.

Gitolite lets you define a "personal" or "scratch" namespace prefix for each developer (for example, `refs/personal/<devname>/*`); see the "personal branches" section in `doc/3-faq-tips-etc.mkd` for details.

"Wildcard" repositories

Gitolite allows you to specify repositories with wildcards (actually perl regexes), like, for example `assignments/s[0-9][0-9]/a[0-9][0-9]`, to pick a random example. This is a *very* powerful feature, which has to be enabled by setting `$GL_WILDREPOS = 1;` in the rc file. It allows you to assign a new permission mode ("C") which allows users to create repositories based on such wild cards, automatically assigns ownership to the specific user who created it, allows him/her to hand out R and RW permissions to other users to collaborate, etc. This feature is documented in `doc/4-wildcard-repositories.mkd`.

Other Features

We'll round off this discussion with a sampling of other features, all of which, and many more, are described in great detail in the "faqs, tips, etc" and other documents.

Logging: Gitolite logs all successful accesses. If you were somewhat relaxed about giving people rewind permissions (RW+) and some kid blew away "master", the log file is a life saver, in terms of easily and quickly finding the SHA that got hosed.

Git outside normal PATH: One extremely useful convenience feature in gitolite is support for git installed outside the normal `$PATH` (this is more common than you think; some corporate environments or even some hosting providers refuse to install things system-wide and you end up putting them in your own directories). Normally, you are forced to make the *client-side* git aware of this non-standard location of the git binaries in some way. With gitolite, just choose a verbose install and set `$GIT_PATH` in the "rc" files. No client-side changes are required after that :-)

Access rights reporting: Another convenient feature is what happens when you try and just ssh to the server. Gitolite shows you what repos you have access to, and what that access may be. Here's an example:

```
hello sitaram, the gitolite version here is v1.5.4-19-ga3397d4
the gitolite config gives you the following access:
R anu-wsd
R entrans
R W git-notes
R W gitolite
R W gitolite-admin
R indic_web_input
R shreelipi_converter
```

Delegation: For really large installations, you can delegate responsibility for groups of repositories to various people and have them manage those pieces independently. This reduces the load on the main admin, and makes him less of a bottleneck. This feature has its own documentation file in the `doc/` directory.

Gitweb support: Gitolite supports gitweb in several ways. You can specify which repos are visible via gitweb. You can set the "owner" and "description" for gitweb from the gitolite config file. Gitweb has a mechanism for you to implement access control based on HTTP authentication, so you can make it use the "compiled" config file that gitolite produces, which means the same access control rules (for read access) apply for gitweb and gitolite.

Mirroring: Gitolite can help you maintain multiple mirrors, and switch between them easily if the primary server goes down.

4.9 Git 守护进程

对于提供公共的，非授权的只读访问，我们可以抛弃 HTTP 协议，改用 Git 自己的协议，这主要是出于性能和速度的考虑。Git 协议远比 HTTP 协议高效，因而访问速度也快，所以它能节省很多用户的时间。

重申一下，这一点只适用于非授权的只读访问。如果建在防火墙之外的服务器上，那么它所提供的服务应该只是那些公开的只读项目。如果是在防火墙之内的服务器上，可用于支撑大量参与人员或自动系统（用于持续集成或编译的主机）只读访问的项目，这样可以省去逐一配置 SSH 公钥的麻烦。

但不管哪种情形，Git 协议的配置设定都很简单。基本上，只要以守护进程的形式运行该命令即可：

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

这里的 `--reuseaddr` 选项表示在重启服务前，不等之前的连接超时就立即重启。而 `--base-path` 选项则允许克隆项目时不必给出完整路径。最后面的路径告诉 Git 守护进程允许开放给用户访问的仓库目录。假如有防火墙，则需要为该主机的 9418 端口设置为允许通信。

以守护进程的形式运行该进程的方法有很多，但主要还得看用的是什么操作系统。在 Ubuntu 主机上，可以用 Upstart 脚本达成。编辑该文件：

```
/etc/event.d/local-git-daemon
```

加入以下内容：

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
--user=git --group=git \
--reuseaddr \
--base-path=/opt/git/ \
/opt/git/
respawn
```

出于安全考虑，强烈建议用一个对仓库只有读取权限的用户身份来运行该进程 — 只需要简单地新建一个名为 `git-ro` 的用户（译注：新建用户默认对仓库文件不具备写权限，但这取决于仓库目录的权限设定。务必确认 `git-ro` 对仓库只能读不能写。），并用它的身份来启动进程。这里为了简化，后面我们还是用之前运行 Gitosis 的用户 `'git'`。

这样一来，当你重启计算机时，Git 进程也会自动启动。要是进程意外退出或者被杀掉，也会自行重启。在设置完成后，不重启计算机就启动该守护进程，可以运行：

```
initctl start local-git-daemon
```

而在其他操作系统上，可以用 `xinetd`，或者 `sysvinit` 系统的脚本，或者其他类似的脚本 — 只要能让那个命令变为守护进程并可监控。

接下来，我们必须告诉 Gitis 哪些仓库允许通过 Git 协议进行匿名只读访问。如果每个仓库都设有各自的段落，可以分别指定是否允许 Git 进程开放给用户匿名读取。比如允许通过 Git 协议访问 `iphone_project`，可以把下面两行加到 `gitis.conf` 文件的末尾：

```
[repo iphone_project]
  daemon = yes
```

在提交和推送完成后，运行中的 Git 守护进程就会响应来自 9418 端口对该项目的访问请求。

如果不考虑 Gitis，单单起了 Git 守护进程的话，就必须到每一个允许匿名只读访问的仓库目录内，创建一个特殊名称的空文件作为标志：

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

该文件的存在，表明允许 Git 守护进程开放对该项目的匿名只读访问。

Gitis 还能设定哪些项目允许放在 GitWeb 上显示。先打开 GitWeb 的配置文件 `/etc/gitweb.conf`，添加以下四行：

```
$projects_list = "/home/git/gitis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

接下来，只要配置各个项目在 Gitis 中的 `gitweb` 参数，便能达成是否允许 GitWeb 用户浏览该项目。比如，要让 `iphone_project` 项目在 GitWeb 里出现，把 `repo` 的设定改成下面的样子：

```
[repo iphone_project]
  daemon = yes
  gitweb = yes
```

在提交并推送过之后，GitWeb 就会自动开始显示 `iphone_project` 项目的细节和历史。

4.10 Git 托管服务

如果不想经历自己架设 Git 服务器的麻烦，网络上有几个专业的仓库托管服务可供选择。这样做有几大优点：托管账户的建立通常比较省时，方便项目的启动，而且不涉及服务器的维护和监控。即使内部创建并运行着自己的服务器，同时为开源项目提供一个公共托管站点还是有好处的——让开源社区更方便地找到该项目，并给予帮助。

目前，可供选择的托管服务数量繁多，各有利弊。在 Git 官方 wiki 上的 Githosting 页面有一个最新的托管服务列表：

<http://git.or.cz/gitwiki/GitHosting>

由于本书无法全部——介绍，而本人（译注：指本书作者 Scott Chacon。）刚好在其中一家公司工作，所以接下来我们将会介绍如何在 GitHub 上建立新账户并启动项目。至于其他托管服务大体也是这么一个过程，基本的想法都是差不多的。

GitHub 是目前为止最大的开源 Git 托管服务，并且还是少数同时提供公共代码和私有代码托管服务的站点之一，所以你可以在上面同时保存开源和商业代码。事实上，本书就是放在 GitHub 上合作编著的。（译注：本书的翻译也是放在 GitHub 上广泛协作的。）

GitHub

GitHub 和大多数的代码托管站点在处理项目命名空间的方式上略有不同。GitHub 的设计更侧重于用户，而不是完全基于项目。也就是说，如果我在 GitHub 上托管一个名为 `grit` 的项目的话，它的地址不会是 `github.com/grit`，而是按在用户底下 `github.com/shacon/grit`（译注：本书作者 Scott Chacon 在 GitHub 上的用户名是 `shacon`。）。不存在所谓某个项目的官方版本，所以假如第一作者放弃了某个项目，它可以无缝转移到其它用户的名下。

GitHub 同时也是一个向使用私有仓库的用户收取费用的商业公司，但任何人都可以方便快捷地申请到一个免费账户，并在上面托管数量不限的开源项目。接下来我们快速介绍一下 GitHub 的基本使用。

建立新账户

首先注册一个免费账户。访问 Pricing and Signup 页面 <http://github.com/plans> 并点击 Free account 里的 Sign Up 按钮（见图 4-2），进入注册页面。



图 4-2. GitHub 服务简介页面

选择一个系统中尚未使用的用户名，提供一个与之相关联的电邮地址，并输入密码（见图 4-3）：

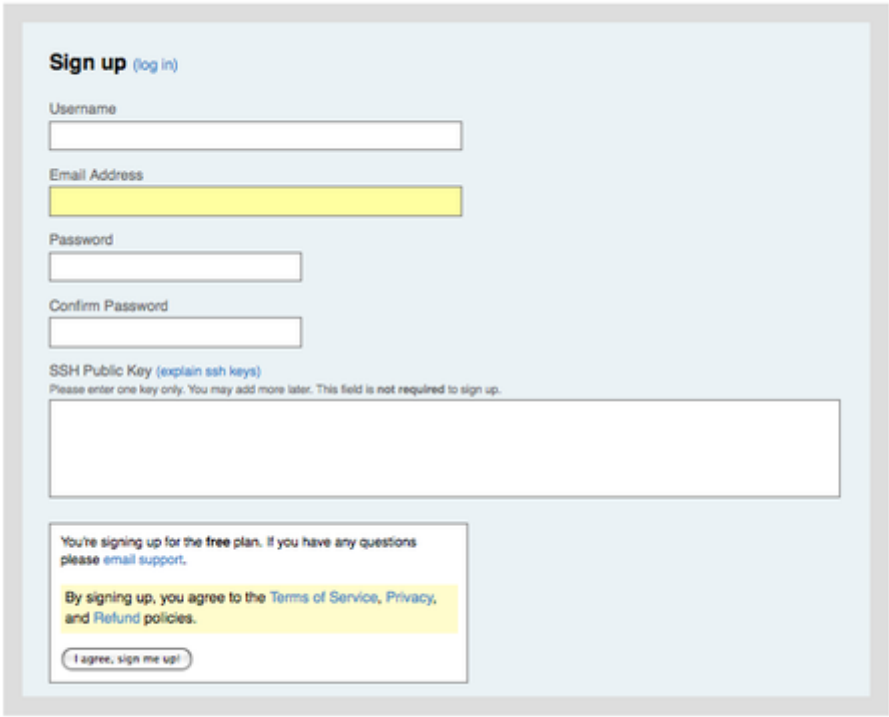
The image shows the GitHub 'Sign up' page. At the top, it says 'Sign up (log in)'. Below this are input fields for 'Username', 'Email Address', 'Password', and 'Confirm Password'. There is also a large text area for 'SSH Public Key' with a note: 'Please enter one key only. You may add more later. This field is not required to sign up.' At the bottom, there is a box containing text about the free plan, a link to email support, and a statement that by signing up, the user agrees to the Terms of Service, Privacy, and Refund policies. A button labeled 'I agree, sign me up!' is at the bottom of this box.

图 4-3. GitHub 用户注册表单

如果方便，现在就可以提供你的 SSH 公钥。我们在前文的"小型安装"一节介绍过生成新公钥的方法。把新生成的公钥复制粘贴到 SSH Public Key 文本框中即可。要是生成公钥的步骤不太清楚，也可以点击 "explain ssh keys" 链接，会显示各个主流操作系统上完成该步骤的介绍。点击 "I agree, sign me up" 按钮完成用户注册，并转到该用户的 dashboard 页面（见图 4-4）：



图 4-4. GitHub 的用户面板

接下来就可以建立新仓库了。

建立新仓库

点击用户面板上仓库旁边的 "create a new one" 链接，显示 Create a New Repository 的表单（见图 4-5）：

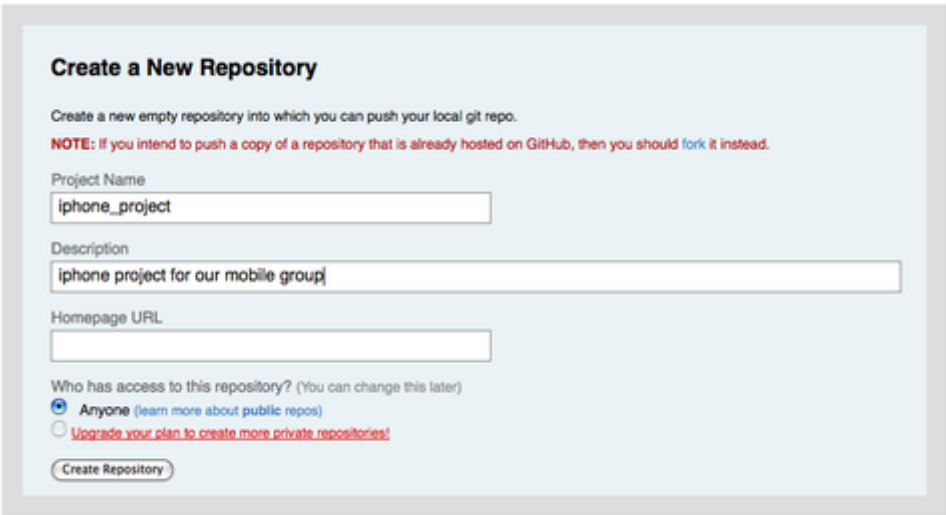
The image shows the 'Create a New Repository' form on GitHub. It starts with the title 'Create a New Repository' and a subtitle 'Create a new empty repository into which you can push your local git repo.' Below this is a red 'NOTE' about forking existing repositories. The form has input fields for 'Project Name' (containing 'iphone_project'), 'Description' (containing 'iphone project for our mobile group'), and 'Homepage URL'. At the bottom, there is a section for 'Who has access to this repository?' with radio buttons for 'Anyone' (selected) and 'Upgrade your plan to create more private repositories!'. A 'Create Repository' button is at the very bottom.

图 4-5. 在 GitHub 上建立新仓库

当然，项目名称是必不可少的，此外也可以适当描述一下项目的情况或者给出官方站点的地址。然后点击 "Create Repository" 按钮，新仓库就建立起来了（见图 4-6）：

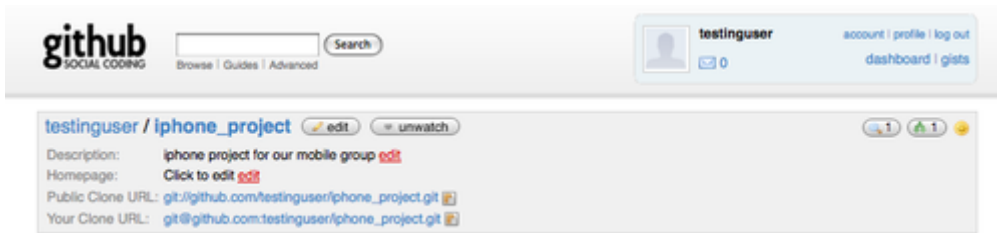


图 4-6. GitHub 上各个项目的概要信息

由于尚未提交代码，点击项目地址后 GitHub 会显示一个简要的指南，告诉你如何新建一个项目并推送上来，如何从现有项目推送，以及如何从一个公共的 Subversion 仓库导入项目（见图 4-7）：

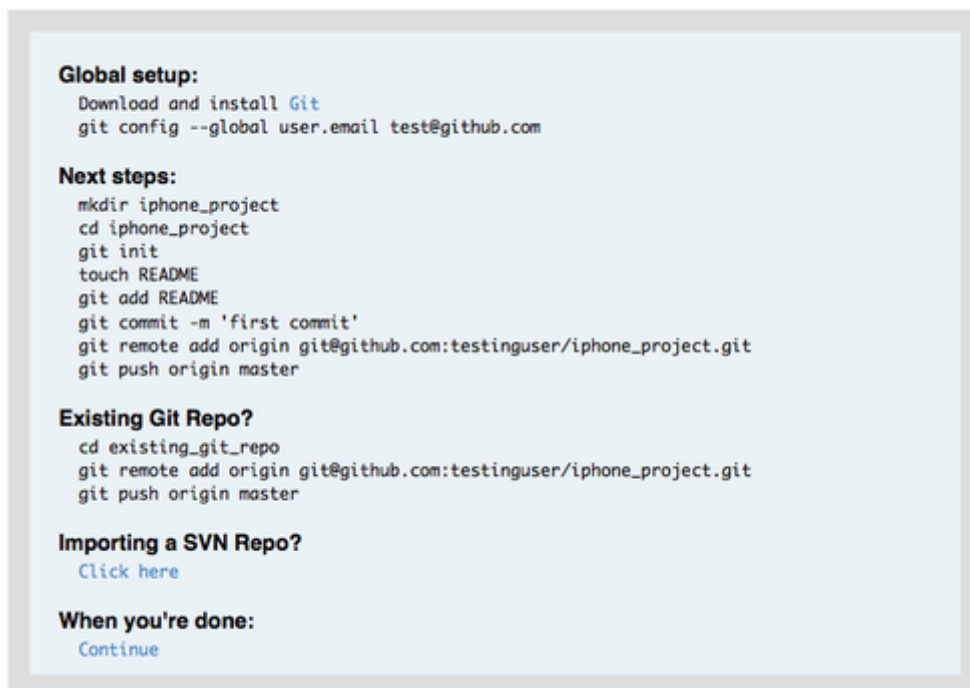


图 4-7. 新仓库指南

该指南和本书前文介绍的类似，对于新的项目，需要先在本地初始化为 Git 项目，添加要管理的文件并作首次提交：

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

然后在这个本地仓库内把 GitHub 添加为远程仓库，并推送 master 分支上来：

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

现在该项目就托管在 GitHub 上了。你可以把它的 URL 分享给每位对此项目感兴趣的人。本例的 URL 是 `http://github.com/testinguser/iphone_project`。而在项目页面的摘要部分，你会发现有两个 Git URL 地址（见图 4-8）：

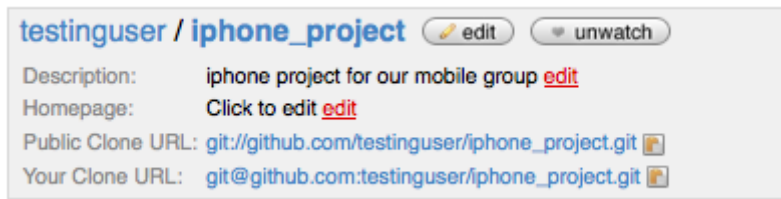


图 4-8. 项目摘要中的公共 URL 和私有 URL

Public Clone URL 是一个公开的，只读的 Git URL，任何人都可以通过它克隆该项目。可以随意散布这个 URL，比如发布到个人网站之类的地方等等。

Your Clone URL 是一个基于 SSH 协议的可读可写 URL，只有使用与上传的 SSH 公钥对应的密钥来连接时，才能通过它进行读写操作。其他用户访问该项目页面时只能看到之前那个公共的 URL，看不到这个私有的 URL。

从 Subversion 导入项目

如果想把某个公共 Subversion 项目导入 Git，GitHub 可以帮忙。在指南的最后有一个指向导入 Subversion 页面的链接。点击它会看到一个表单，包含有关导入流程的信息以及一个用来粘贴公共 Subversion 项目连接的文本框（见图 4-9）：

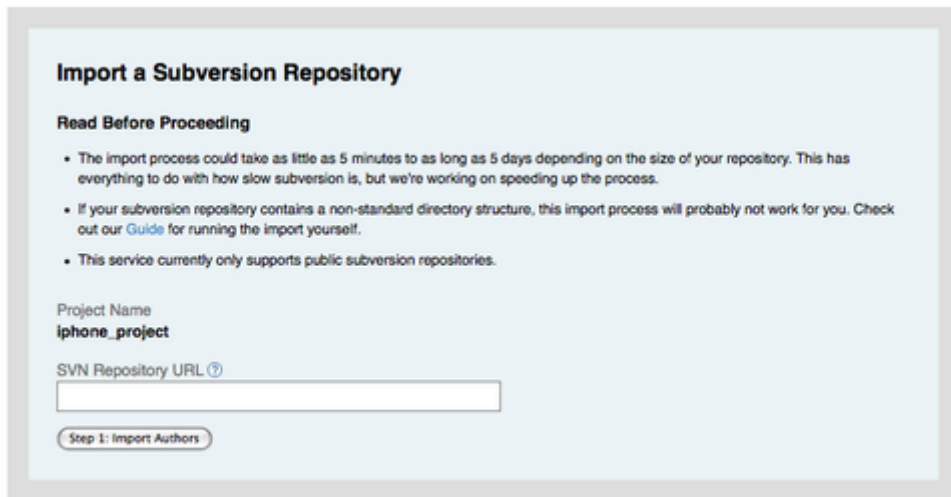


图 4-9. Subversion 导入界面

如果项目很大，采用非标准结构，或者是私有的，那就无法借助该工具实现导入。到第 7 章，我们会介绍如何手工导入复杂工程的具体方法。

添加协作开发者

现在把团队里的其他人也加进来。如果 John，Josie 和 Jessica 都在 GitHub 注册了账户，要赋予他们对该仓库的推送权限，可以把他们加为项目协作者。这样他们就可以通过各自的公钥访问我的这个仓库了。

点击项目页面上方的 "edit" 按钮或者顶部的 Admin 标签，进入该项目的管理页面（见图 4-10）：

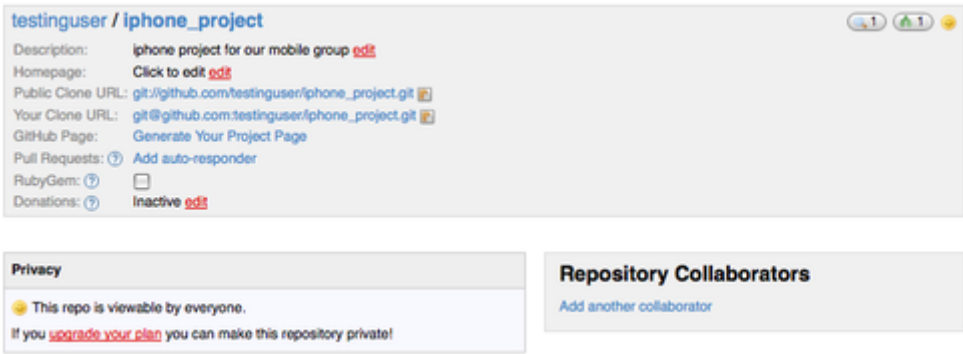


图 4-10. GitHub 的项目管理页面

为了给另一个用户添加项目的写权限，点击 "Add another collaborator" 链接，出现一个用于输入用户名的表单。在输入的同时，它会自动跳出一个符合条件的候选名单。找到正确用户名之后，点 Add 按钮，把该用户设为项目协作者（见图 4-11）：

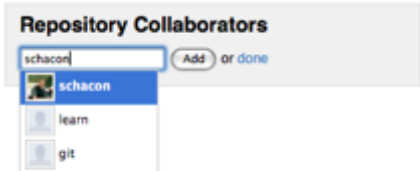


图 4-11. 为项目添加协作者

添加完协作者之后，就可以在 Repository Collaborators 区域看到他们的名单（见图 4-12）：

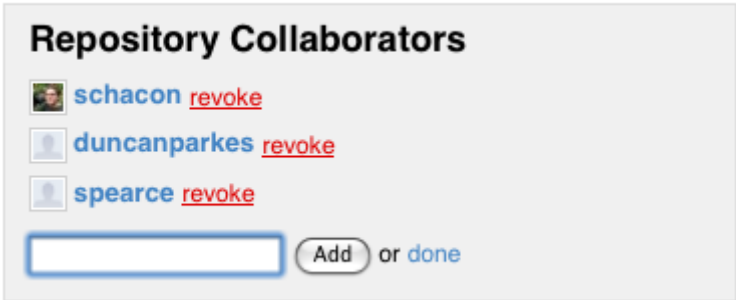


图 4-12. 项目协作者名单

如果要取消某人的访问权，点击 "revoke" 即可取消他的推送权限。对于将来的项目，你可以从现有项目复制协作者名单，或者直接借用协作者群组。

项目页面

在推送或从 Subversion 导入项目之后，你会看到一个类似图 4-13 的项目主页：

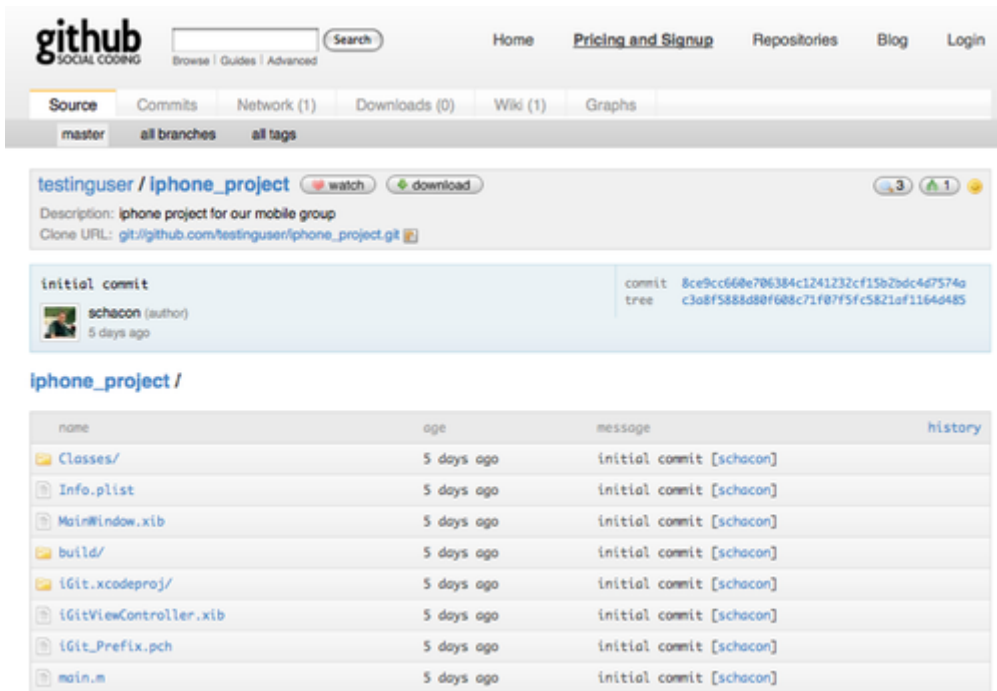


图 4-13. GitHub 上的项目主页

别人访问你的项目时看到的就是这个页面。它有若干导航标签，提交标签用于显示提交历史，最新的提交位于最上方，这和 `git log` 命令的输出类似。Network 标签展示所有派生了该项目并做出贡献的用户的关系图谱。Downloads 标签允许你上传项目的二进制文件，提供下载该项目各个版本的 tar/zip 包。Wiki 标签提供了一个用于撰写文档或其他项目相关信息的 wiki 站点。Graphs 标签包含了一些可视化的项目信息与数据。默认打开的 Source 标签页面，则列出了该项目的目录结构和概要信息，并在下方自动展示 README 文件的内容（如果该文件存在的话），此外还会显示最近一次提交的相关信息。

派生项目

如果要为一个自己没有推送权限的项目贡献代码，GitHub 鼓励使用派生（fork）。到那个感兴趣的项目主页上，点击页面上方的 "fork" 按钮，GitHub 就会为你复制一份该项目的副本到你的仓库中，这样你就可以向自己的这个副本推送数据了。

采取这种办法的好处是，项目拥有者不必忙于应付赋予他人推送权限的工作。随便谁都可以通过派生得到一个项目副本并在其中展开工作，事后只需要项目维护者将这些副本仓库加为远程仓库，然后提取更新合并即可。

要派生一个项目，到原始项目的页面（本例中是 `mojombo/chronic`）点击 "fork" 按钮（见图 4-14）：



图 4-14. 点击 "fork" 按钮获得任意项目的可写副本

几秒钟之后，你将进入新建的项目页面，会显示该项目派生自哪一个项目（见图 4-15）：

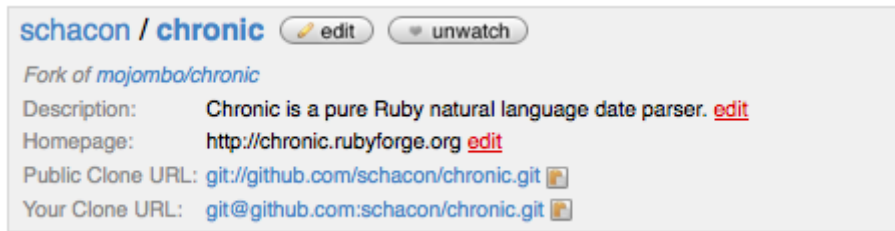


图 4-15. 派生后得到的项目副本

GitHub 小结

关于 GitHub 就先介绍这么多，能够快速达成这些事情非常重要（译注：门槛的降低和完成基本任务的简单高效，对于推动开源项目的协作发展有着举足轻重的意义。）。短短几分钟内，你就能创建一个新账户，添加一个项目并开始推送。如果项目是开源的，整个庞大的开发者社区都可以立即访问它，提供各式各样的帮助和贡献。最起码，这也是一种 Git 新手立即体验尝试 Git 的捷径。

4.11 小结

我们讨论并介绍了一些建立远程 Git 仓库的方法，接下来你可以通过这些仓库同他人分享或合作。

运行自己的服务器意味着更多的控制权以及在防火墙内部操作的可能性，当然这样的服务器通常需要投入一定的时间精力来架设维护。如果直接托管，虽然能免去这部分工作，但有时出于安全或版权的考虑，有些公司禁止将商业代码托管到第三方服务商。

所以究竟采取哪种方案，并不是个难以取舍的问题，或者其一，或者相互配合，哪种合适就用哪种。

[下一节](#) [上一节](#) [首页](#) ([目录](#)) | [返回 码云](#)