4.1 服务器上的 Git - 协议

到目前为止，你应该已经有办法使用 Git 来完成日常工作。 然而，为了使用 Git 协作功能，你还需要有远程的 Git 仓库。 尽管在技术上你可以从个人仓库进行推送（push）和拉取（pull）来修改内容，但不鼓励使用这种方法，因为一不留心就很容易弄混其他人的进度。 此外，你希望你的合作者们即使在你的电脑未联机时亦能存取仓库 — 拥有一个更可靠的公用仓库十分有用。 因此，与他人合作的最佳方法即是建立一个你与合作者们都有权利访问，且可从那里推送和拉取资料的共用仓库。

架设一台 Git 服务器并不难。 首先，选择你希望服务器使用的通讯协议。 在本章第一节将介绍可用的协议以及各自优缺点。 下面一节将解释使用那些协议的典型设置及如何在你的服务器上运行。 最后，如果你不介意托管你的代码在其他人的服务器，且不想经历设置与维护自己服务器的麻烦，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帐户然后继续下一章，我们会在那里讨论分布式源码控制环境的林林总总。

一个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。因为该仓库仅仅作为合作媒介，不需要从磁盘检查快照；存放的只有 Git 的资料。 简单的说，裸仓库就是你工程目录内的 .git 子目录内容，不包含其他资料。

协议

Git 可以使用四种不同的协议来传输资料：本地协议（Local），HTTP 协议，SSH（Secure Shell）协议及 Git 协议。 在此，我们将会讨论那些协议及哪些情形应该使用（或避免使用）他们。

本地协议

最基本的就是 *本地协议（Local protocol）* ，其中的远程版本库就是同一主机上的另一个目录。 这常见于团队每一个成员都对一个共享的文件系统（例如一个挂载的 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。 后者并不理想，因为你的所有代码版本库如果长存于同一台电脑，更可能发生灾难性的损失。

如果你使用共享文件系统，就可以从本地版本库克隆（clone）、推送（push）以及拉取（pull）。 像这样去克隆一个版本库或者增加一个远程到现有的项目中，使用版本库路径作为 URL。 例如，克隆一个本地版本库，可以执行如下的命令：

```
$ git clone /srv/git/project.git
```

或你可以执行这个命令：

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

如果在 URL 开头明确的指定 `file://`，那么 Git 的行为会略有不同。 如果仅是指定路径，Git 会尝试使用硬链接（hard link）或直接复制所需要的文件。 如果指定 `file://`，Git 会触发平时用于网路传输资料的进程，那样传输效率会更低。 指定 `file://` 的主要目的是取得一个没有外部参考（extraneous references） 或对象（object）的干净版本库副本—通常是在从其他版本控制系统导入后或一些类似情况需要这么做 （关于维护任务可参见 Git 内部原理 ）。 在此我们将使用普通路径，因为这样通常更快。

要增加一个本地版本库到现有的 Git 项目，可以执行如下的命令：

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

然后，就可以通过新的远程仓库名 `local_proj` 像在网络上一样从远端版本库推送和拉取更新了。

优点

基于文件系统的版本库的优点是简单，并且直接使用了现有的文件权限和网络访问权限。 如果你的团队已经有共享文件系统，建立版本库会十分容易。 只需要像设置其他共享目录一样，把一个裸版本库的副本放到大家都可以访问的路径，并设置好读/写的权限，就可以了， 我们会在在服务器上搭建 Git 讨论如何导出一个裸版本库。

这也是快速从别人的工作目录中拉取更新的方法。 如果你和别人一起合作一个项目，他想让你从版本库中拉取更新时，运行类似 `git pull /home/john/project` 的命令比推送到服务器再抓取回来简单多了。

缺点

这种方法的缺点是，通常共享文件系统比较难配置，并且比起基本的网络连接访问，这不方便从多个位置访问。 如果你想从家里推送内容，必须先挂载一个远程磁盘，相比网络连接的访问方式，配置不方便，速度也慢。

值得一提的是，如果你使用的是类似于共享挂载的文件系统时，这个方法不一定是最快的。 访问本地版本库的速度与你访问数据的速度是一样的。 在同一个服务器上，如果允许 Git 访问本地硬盘，一般的通过 NFS 访问版本库要比通过 SSH 访问慢。

最终，这个协议并不保护仓库避免意外的损坏。 每一个用户都有“远程”目录的完整 shell 权限，没有方法可以阻止他们修改或删除 Git 内部文件和损坏仓库。

HTTP 协议

Git 通过 HTTP 通信有两种模式。 在 Git 1.6.6 版本之前只有一个方式可用，十分简单并且通常是只读模式的。 Git 1.6.6 版本引入了一种新的、更智能的协议，让 Git 可以像通过 SSH 那样智能的协商和传输数据。 之后几年，这个新的 HTTP 协议因为其简单、智能变的

十分流行。 新版本的 HTTP 协议一般被称为 **智能** HTTP 协议，旧版本的一般被称为 **哑** HTTP 协议。 我们先了解一下新的智能 HTTP 协议。

智能 HTTP 协议

智能 HTTP 的运行方式和 SSH 及 Git 协议类似，只是运行在标准的 HTTP/S 端口上并且可以使用各种 HTTP 验证机制， 这意味着使用起来会比 SSH 协议简单的多，比如可以使用 HTTP 协议的用户名/密码授权，免去设置 SSH 公钥。

智能 HTTP 协议或许已经是最流行的使用 Git 的方式了，它即支持像 `git://` 协议一样设置匿名服务， 也可以像 SSH 协议一样提供传输时的授权和加密。 而且只用一个 URL 就可以都做到，省去了为不同的需求设置不同的 URL。 如果你要推送到一个需要授权的服务器上（一般来讲都需要），服务器会提示你输入用户名和密码。 从服务器获取数据时也一样。

事实上，类似 GitHub 的服务，你在网页上看到的 URL（比如 https://github.com/schacon/simplegit），和你在克隆、推送（如果你有权限）时使用的是一样的。

哑（Dumb） HTTP 协议

如果服务器没有提供智能 HTTP 协议的服务，Git 客户端会尝试使用更简单的“哑” HTTP 协议。 哑 HTTP 协议里 web 服务器仅把裸版本库当作普通文件来对待，提供文件服务。 哑 HTTP 协议的优美之处在于设置起来简单。 基本上，只需要把一个裸版本库放在 HTTP 根目录，设置一个叫做 `post-update` 的挂钩就可以了 （见 Git 钩子）。 此时，只要能访问 web 服务器上你的版本库，就可以克隆你的版本库。 下面是设置从 HTTP 访问版本库的方法：

```
$ 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 自带的 `post-update` 挂钩会默认执行合适的命令（`git update-server-info`），来确保通过 HTTP 的获取和克隆操作正常工作。 这条命令会在你通过 SSH 向版本库推送之后被执行；然后别人就可以通过类似下面的命令来克隆：

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

这里我们用了 Apache 里设置了常用的路径 `/var/www/htdocs`，不过你可以使用任何静态 Web 服务器 —— 只需要把裸版本库放到正确的目录下就可以。 Git 的数据是以基本的静态文件形式提供的（详情见 Git 内部原理）。

通常的，会在可以提供读 / 写的智能 HTTP 服务和简单的只读的哑 HTTP 服务之间选一个。 极少会将二者混合提供服务。

优点

我们将只关注智能 HTTP 协议的优点。

不同的访问方式只需要一个 URL 以及服务器只在需要授权时提示输入授权信息，这两个简便性让终端用户使用 Git 变得非常简单。 相比 SSH 协议，可以使用用户名 / 密码授权是一个很大的优势，这样用户就不必须在使用 Git 之前先在本地生成 SSH 密钥对再把公钥上传到服务器。 对非资深的使用者，或者系统上缺少 SSH 相关程序的使用者，HTTP 协议的可用性是主要的优势。 与 SSH 协议类似，HTTP 协议也非常快和高效。

你也可以在 HTTPS 协议上提供只读版本库的服务，如此你在传输数据的时候就可以加密数据；或者，你甚至可以让客户端使用指定的 SSL 证书。

另一个好处是 HTTPS 协议被广泛使用，一般的企业防火墙都会允许这些端口的数据通过。

缺点

在一些服务器上，架设 HTTPS 协议的服务端会比 SSH 协议的棘手一些。 除了这一点，用其他协议提供 Git 服务与智能 HTTP 协议相比就几乎没有优势了。

如果你在 HTTP 上使用需授权的推送，管理凭证会比使用 SSH 密钥认证麻烦一些。 然而，你可以选择使用凭证存储工具，比如 macOS 的 Keychain 或者 Windows 的凭证管理器。 参考 凭证存储 如何安全地保存 HTTP 密码。

SSH 协议

架设 Git 服务器时常用 SSH 协议作为传输协议。 因为大多数环境下服务器已经支持通过 SSH 访问 —— 即使没有也很容易架设。 SSH 协议也是一个验证授权的网络协议；并且，因为其普遍性，架设和使用都很容易。

通过 SSH 协议克隆版本库，你可以指定一个 ssh:// 的 URL：

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

或者使用一个简短的 scp 式的写法：

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

在上面两种情况中，如果你不指定可选的用户名，那么 Git 会使用当前登录的用的名字。

优势

用 SSH 协议的优势有很多。 首先，SSH 架设相对简单 —— SSH 守护进程很常见，多数管理员都有使用经验，并且多数操作系统都包含了它及相关的管理工具。 其次，通过 SSH 访问是安全的 —— 所有传输数据都要经过授权和加密。 最后，与 HTTPS 协议、Git 协议及本地协议一样，SSH 协议很高效，在传输前也会尽量压缩数据。

缺点

SSH 协议的缺点在于它不支持匿名访问 Git 仓库。 如果你使用 SSH，那么即便只是读取数据，使用者也 **必须** 通过 SSH 访问你的主机， 这使得 SSH 协议不利于开源的项目，毕竟人们可能只想把你的仓库克隆下来查看。 如果你只在公司网络使用，SSH 协议可能是你唯一要用到的协议。 如果你要同时提供匿名只读访问和 SSH 协议，那么你除了为自己推送架设 SSH 服务以外， 还得架设一个可以让其他人访问的服务。

Git 协议

最后是 Git 协议。 这是包含在 Git 里的一个特殊的守护进程；它监听在一个特定的端口（9418），类似于 SSH 服务，但是访问无需任何授权。 要让版本库支持 Git 协议，需要先创建一个 `git-daemon-export-ok` 文件 —— 它是 Git 协议守护进程为这个版本库提供服务的必要条件 —— 但是除此之外没有任何安全措施。 要么谁都可以克隆这个版本库，要么谁也不能。 这意味着，通常不能通过 Git 协议推送。 由于没有授权机制，一旦你开放推送操作，意味着网络上知道这个项目 URL 的人都可以向项目推送数据。 不用说，极少会有人这么做。

优点

目前，Git 协议是 Git 使用的网络传输协议里最快的。 如果你的项目有很大的访问量，或者你的项目很庞大并且不需要为写进行用户授权，架设 Git 守护进程来提供服务是不错的选择。 它使用与 SSH 相同的数据传输机制，但是省去了加密和授权的开销。

缺点

Git 协议缺点是缺乏授权机制。 把 Git 协议作为访问项目版本库的唯一手段是不可取的。 一般的做法里，会同时提供 SSH 或者 HTTPS 协议的访问服务，只让少数几个开发者有推送（写）权限，其他人通过 `git://` 访问只有读权限。 Git 协议也许也是最难架设的。 它要求有自己的守护进程，这就要配置 `xinetd`、`systemd` 或者其他的程序，这些工作并不简单。 它还要求防火墙开放 9418 端口，但是企业防火墙一般不会开放这个非标准端口。 而大型的企业防火墙通常会封锁这个端口。