

$ git clone <版本库的地址> [本地目录名]（可选）

Git clone 支持多种协议：除HTTP（s）外，还支持SSH、Git、本地文件协议等

除了Http协议之外，其他所有协议都要求在服务器端安装并运行Git。

例如：git clone ssh://code.ctrip.com//path/repo/Mice.Input.git/

注意：SSH的另一种写法如下

Git clone [ffma@]ctrip.com:path/to/repo.git/

要添加一个现有的本地仓库作为Git项目的远程仓库，可以用下面的命令：

$ git remote add local\_project /opt/git/project.git

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。SSH同时也是一个验证授权的网络协议，因为其普遍性，一般架设和使用都很容易。

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。

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。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-daemon-export-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

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。

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。

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

$ 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

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 的数据是以最基本的静态文件的形式提供的

$ git clone <http://example.com/gitproject.git>

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

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。

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。

$ git remote 命令用于管理主机名，Git要求每个远程主机都必须指定一个主机名

$ git remote –v 可以查看远程主机网址

在克隆版本库的时候，远程主机会自动被Git命名为origin，如果想用其他主机名，可以用 git clone –o

$ git clone –o jQuery <https://github.com/jquery/jquery.git>

$ git remote show <主机名> 可以查看该主机的详细信息

$ git remote add <主机名> <网址> 用于添加远程主机

$ git remote rm <主机名> 用于删除远程主机

$ git remote rename <原主机名> <新主机名>

$ git fetch 用于将远程主机的更新提交取回到本地

$ git fetch <远程主机名> 将某个远程主机的更新全部取回到本地，默认情况下，git fetch 会取回所有分支的更新，如果只想取回某个分支的更新，请补上 分支名

$ git fetch <远程主机名> <分支名>

所取回的更新，在本地主机上要用"远程主机名/分支名"的形式读取。比如origin主机的master，就要用origin/master读取。

取回远程主机的更新以后，可以在它的基础上，使用git checkout命令创建一个新的分支。

$ git checkout –b newBranch origin/master 在origin/master的基础上，创建一个新分支。

此外，也可以使用git merge命令或者git rebase命令，在本地分支上合并远程分支。

$ git merge origin/master

$ git rebase origin/master

$ git pull 取回远程主机某个分支的更新，再与本地的指定分支合并。它的完整格式稍稍有点复杂:

$ git pull <远程主机名> <远程分支名>：<本地分支名>

如果远程分支是与当前分支合并，则冒号后面的部分可以省略。

$ git pull origin release 意味着取回远程主机的release分支然后与本地的当前分支合并，这等同于先做git fetch ,再做 git merge。

在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系（tracking）。比如，在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是说，本地的master分支自动"追踪"origin/master分支。Git也允许手动建立追踪关系。

$ git branch –set-upstream master origin/next该命令指定本地的master分支追踪远程主机的next分支

如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名。

$ git pull origin

上面命令表示，本地的当前分支自动与对应的origin主机"追踪分支"（remote-tracking branch）进行合并。

如果当前分支只有一个追踪分支，连远程主机名都可以省略。

$ git pull

上面命令表示，当前分支自动与唯一一个追踪分支进行合并。

如果合并需要采用rebase模式，可以使用--rebase选项。

$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>

$ git push 命令用于将本地分支的更新，推送到远程主机。它的格式与git pull命令相仿。

$ git push <远程主机名> <本地分支名>：<远程分支名>

如果省略远程分支名，则表示将本地分支推送与之存在"追踪关系"的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支。

$ git push origin：master 表示删除origin 主机的master分支

如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

$ git push origin

上面命令表示，将当前分支推送到origin主机的对应分支。

如果当前分支只有一个追踪分支，那么主机名都可以省略。

$ git push

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。

$ git push –u origin master

不带任何参数的git push，默认只推送当前分支，这叫做simple方式。此外，还有一种matching方式，会推送所有有对应的远程分支的本地分支。Git 2.0版本之前，默认采用matching方法，现在改为默认采用simple方式。如果要修改这个设置，可以采用git config命令。

$ git config –global push.default matching

$ git config –global push.default simple

如果远程主机的版本比本地版本更新，推送时Git会报错，要求先在本地做git pull合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用--force选项。

$ git push --force origin

上面命令使用--force选项，结果导致远程主机上更新的版本被覆盖。除非你很确定要这样做，否则应该尽量避免使用--force选项。

最后，git push不会推送标签（tag），除非使用--tags选项。

$ git push origin –tags

$ git pull –rebase VS merge

当本地commit一个提交和远端服务器中的代码有冲突(别人也改了相同的文件)时可以在pull 中加 –rebase。加上 rebase 的意思是：

git pull --rebase

1. 把本地 repo. 从上次 pull 之后的变更暂存起來
2. 恢复到上次 pull 时的状态
3. 合并远端的变更到本地
4. 最后再合并刚刚暂存下來的本地变更

**合并前：**

D---E master

/

A---B---C---F origin/master

**使用 merge 合并后：**

D--------E

/ \

A---B---C---F----G master, origin/master

**如果是 rebase 的方式，就不會有 G 合并点：**

A---B---C---F---D'---E' master, origin/master

注意到，其中 D’, E’ 的 commit SHA 序号跟本來 D, E 是不同的，应为算是砍掉重新 commit 了。

**rebase vs. merge**

rebase 跟 merge 类似，出现 conflict 会暂停 rebase 动作，需要你手动修复后，然后才可以继续动作。这也是 rebase 比 merge 复杂一点的地方：merge 如果发生 conflict，你只需要解决冲突一次，然后commit 出去就完成了。而 rebase 的 conflict 可能会发生在上述步骤 4 的每一次重新套用上，所以可能需要解决冲突好几次 (rebase 时所谓的解决冲突，其实是直接修改你之前的变更內容，所以上图中变成 D’ 跟 E’ )。

**配置**

如果想要把 rebase 当做 git pull 的默认值，可以在 .git/config 加上

[branch "master"]

remote = origin

merge = refs/heads/master

rebase = true

也可以直接加到 ~/.gitconfig 让所有的 tracked branches 都自动套用这个设定：

[branch]

autosetuprebase = always



