子模块

经常有这样的事情，当你在一个项目上工作时，你需要在其中使用另外一个项目。也许它是一个第三方开发的库或者是你独立开发和并在多个父项目中使用的。这个场景下一个常见的问题产生了：你想将两个项目单独处理但是又需要在其中一个中使用另外一个。

这里有一个例子。假设你在开发一个网站，为之创建Atom源。你不想编写一个自己的Atom生成代码，而是决定使用一个库。你可能不得不像CPAN install或者Ruby gem一样包含来自共享库的代码，或者将代码拷贝到你的项目树中。如果采用包含库的办法，那么不管用什么办法都很难去定制这个库，部署它就更加困难了，因为你必须确保每个客户都拥有那个库。把代码包含到你自己的项目中带来的问题是，当上游被修改时，任何你进行的定制化的修改都很难归并。

Git 通过子模块处理这个问题。子模块允许你将一个 Git 仓库当作另外一个Git仓库的子目录。这允许你克隆另外一个仓库到你的项目中并且保持你的提交相对独立。

子模块初步

假设你想把 Rack 库（一个 Ruby 的 web 服务器网关接口）加入到你的项目中，可能既要保持你自己的变更，又要延续上游的变更。首先你要把外部的仓库克隆到你的子目录中。你通过git submodule add将外部项目加为子模块：

$ git submodule add git://github.com/chneukirchen/rack.git rack

Initialized empty Git repository in /opt/subtest/rack/.git/

remote: Counting objects: 3181, done.

remote: Compressing objects: 100% (1534/1534), done.

remote: Total 3181 (delta 1951), reused 2623 (delta 1603)

Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.

Resolving deltas: 100% (1951/1951), done.

现在你就在项目里的rack子目录下有了一个 Rack 项目。你可以进入那个子目录，进行变更，加入你自己的远程可写仓库来推送你的变更，从原始仓库拉取和归并等等。如果你在加入子模块后立刻运行git status，你会看到下面两项：

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: .gitmodules

# new file: rack

#

首先你注意到有一个.gitmodules文件。这是一个配置文件，保存了项目 URL 和你拉取到的本地子目录

$ cat .gitmodules

[submodule "rack"]

path = rack

url = git://github.com/chneukirchen/rack.git

如果你有多个子模块，这个文件里会有多个条目。很重要的一点是这个文件跟其他文件一样也是处于版本控制之下的，就像你的.gitignore文件一样。它跟项目里的其他文件一样可以被推送和拉取。这是其他克隆此项目的人获知子模块项目来源的途径。

git status的输出里所列的另一项目是 rack 。如果你运行在那上面运行git diff，会发现一些有趣的东西：

$ git diff --cached rack

diff --git a/rack b/rack

new file mode 160000

index 0000000..08d709f

--- /dev/null

+++ b/rack

@@ -0,0 +1 @@

+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

尽管rack是你工作目录里的子目录，但 Git 把它视作一个子模块，当你不在那个目录里时并不记录它的内容。取而代之的是，Git 将它记录成来自那个仓库的一个特殊的提交。当你在那个子目录里修改并提交时，子项目会通知那里的 HEAD 已经发生变更并记录你当前正在工作的那个提交；通过那样的方法，当其他人克隆此项目，他们可以重新创建一致的环境。

这是关于子模块的重要一点：你记录他们当前确切所处的提交。你不能记录一个子模块的master或者其他的符号引用。

当你提交时，会看到类似下面的：

$ git commit -m 'first commit with submodule rack'

[master 0550271] first commit with submodule rack

2 files changed, 4 insertions(+), 0 deletions(-)

create mode 100644 .gitmodules

create mode 160000 rack

注意 rack 条目的 160000 模式。这在Git中是一个特殊模式，基本意思是你将一个提交记录为一个目录项而不是子目录或者文件。

你可以将rack目录当作一个独立的项目，保持一个指向子目录的最新提交的指针然后反复地更新上层项目。所有的Git命令都在两个子目录里独立工作：

$ git log -1

commit 0550271328a0038865aad6331e620cd7238601bb

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Apr 9 09:03:56 2009 -0700

first commit with submodule rack

$ cd rack/

$ git log -1

commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

Author: Christian Neukirchen <chneukirchen@gmail.com>

Date: Wed Mar 25 14:49:04 2009 +0100

Document version change

克隆一个带子模块的项目

这里你将克隆一个带子模块的项目。当你接收到这样一个项目，你将得到了包含子项目的目录，但里面没有文件：

$ git clone git://github.com/schacon/myproject.git

Initialized empty Git repository in /opt/myproject/.git/

remote: Counting objects: 6, done.

remote: Compressing objects: 100% (4/4), done.

remote: Total 6 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (6/6), done.

$ cd myproject

$ ls -l

total 8

-rw-r--r-- 1 schacon admin 3 Apr 9 09:11 README

drwxr-xr-x 2 schacon admin 68 Apr 9 09:11 rack

$ ls rack/

$

rack目录存在了，但是是空的。你必须运行两个命令：git submodule init来初始化你的本地配置文件，git submodule update来从那个项目拉取所有数据并检出你上层项目里所列的合适的提交：

$ git submodule init

Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'

$ git submodule update

Initialized empty Git repository in /opt/myproject/rack/.git/

remote: Counting objects: 3181, done.

remote: Compressing objects: 100% (1534/1534), done.

remote: Total 3181 (delta 1951), reused 2623 (delta 1603)

Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.

Resolving deltas: 100% (1951/1951), done.

Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'

现在你的rack子目录就处于你先前提交的确切状态了。如果另外一个开发者变更了 rack 的代码并提交，你拉取那个引用然后归并之，将得到稍有点怪异的东西：

$ git merge origin/master

Updating 0550271..85a3eee

Fast forward

rack | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

[master\*]$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: rack

#

你归并来的仅仅上是一个指向你的子模块的指针；但是它并不更新你子模块目录里的代码，所以看起来你的工作目录处于一个临时状态：

$ git diff

diff --git a/rack b/rack

index 6c5e70b..08d709f 160000

--- a/rack

+++ b/rack

@@ -1 +1 @@

-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0

+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433

事情就是这样，因为你所拥有的指向子模块的指针和子模块目录的真实状态并不匹配。为了修复这一点，你必须再次运行git submodule update：

$ git submodule update

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 3 (delta 1), reused 2 (delta 0)

Unpacking objects: 100% (3/3), done.

From git@github.com:schacon/rack

08d709f..6c5e70b master -> origin/master

Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'

每次你从主项目中拉取一个子模块的变更都必须这样做。看起来很怪但是管用。

一个常见问题是当开发者对子模块做了一个本地的变更但是并没有推送到公共服务器。然后他们提交了一个指向那个非公开状态的指针然后推送上层项目。当其他开发者试图运行git submodule update，那个子模块系统会找不到所引用的提交，因为它只存在于第一个开发者的系统中。如果发生那种情况，你会看到类似这样的错误：

$ git submodule update

fatal: reference isn’t a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0

Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'

你不得不去查看谁最后变更了子模块

$ git log -1 rack

commit 85a3eee996800fcfa91e2119372dd4172bf76678

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Apr 9 09:19:14 2009 -0700

added a submodule reference I will never make public. hahahahaha!

然后，你给那个家伙发电子邮件说他一通。

上层项目

有时候，开发者想按照他们的分组获取一个大项目的子目录的子集。如果你是从 CVS 或者 Subversion 迁移过来的话这个很常见，在那些系统中你已经定义了一个模块或者子目录的集合，而你想延续这种类型的工作流程。

在 Git 中实现这个的一个好办法是你将每一个子目录都做成独立的 Git 仓库，然后创建一个上层项目的 Git 仓库包含多个子模块。这个办法的一个优势是你可以在上层项目中通过标签和分支更为明确地定义项目之间的关系。

子模块的问题

使用子模块并非没有任何缺点。首先，你在子模块目录中工作时必须相对小心。当你运行git submodule update，它会检出项目的指定版本，但是不在分支内。这叫做获得一个分离的头——这意味着 HEAD 文件直接指向一次提交，而不是一个符号引用。问题在于你通常并不想在一个分离的头的环境下工作，因为太容易丢失变更了。如果你先执行了一次submodule update，然后在那个子模块目录里不创建分支就进行提交，然后再次从上层项目里运行git submodule update同时不进行提交，Git会毫无提示地覆盖你的变更。技术上讲你不会丢失工作，但是你将失去指向它的分支，因此会很难取到。

为了避免这个问题，当你在子模块目录里工作时应使用git checkout -b work创建一个分支。当你再次在子模块里更新的时候，它仍然会覆盖你的工作，但是至少你拥有一个可以回溯的指针。

切换带有子模块的分支同样也很有技巧。如果你创建一个新的分支，增加了一个子模块，然后切换回不带该子模块的分支，你仍然会拥有一个未被追踪的子模块的目录

$ git checkout -b rack

Switched to a new branch "rack"

$ git submodule add git@github.com:schacon/rack.git rack

Initialized empty Git repository in /opt/myproj/rack/.git/

...

Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.

Resolving deltas: 100% (1952/1952), done.

$ git commit -am 'added rack submodule'

[rack cc49a69] added rack submodule

2 files changed, 4 insertions(+), 0 deletions(-)

create mode 100644 .gitmodules

create mode 160000 rack

$ git checkout master

Switched to branch "master"

$ git status

# On branch master

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# rack/

你将不得不将它移走或者删除，这样的话当你切换回去的时候必须重新克隆它——你可能会丢失你未推送的本地的变更或分支。

最后一个需要引起注意的是关于从子目录切换到子模块的。如果你已经跟踪了你项目中的一些文件但是想把它们移到子模块去，你必须非常小心，否则Git会生你的气。假设你的项目中有一个子目录里放了 rack 的文件，然后你想将它转换为子模块。如果你删除子目录然后运行submodule add，Git会向你大吼：

$ rm -Rf rack/

$ git submodule add git@github.com:schacon/rack.git rack

'rack' already exists in the index

你必须先将rack目录撤回。然后你才能加入子模块：

$ git rm -r rack

$ git submodule add git@github.com:schacon/rack.git rack

Initialized empty Git repository in /opt/testsub/rack/.git/

remote: Counting objects: 3184, done.

remote: Compressing objects: 100% (1465/1465), done.

remote: Total 3184 (delta 1952), reused 2770 (delta 1675)

Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.

Resolving deltas: 100% (1952/1952), done.

现在假设你在一个分支里那样做了。如果你尝试切换回一个仍然在目录里保留那些文件而不是子模块的分支时——你会得到下面的错误：

$ git checkout master

error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.

你必须先移除rack子模块的目录才能切换到不包含它的分支：

$ mv rack /tmp/

$ git checkout master

Switched to branch "master"

$ ls

README rack

然后，当你切换回来，你会得到一个空的rack目录。你可以运行git submodule update重新克隆，也可以将/tmp/rack目录重新移回空目录。