World Under Control

Happy coding

Search Go

- 首页
- 留言板
- 管理

CentOS 的自举
Fedora 上开启 WebDAV 服务

git 子模块的使用简介

Easior Lars posted @ 2014年1月22日 21:46 in Linux with tags linux Git git-module , 3238 阅读

通常，一个大型工程总会被分拆为一些子工程。这既有利于工程开发难度的降低，也有利于使用现成的方案或者第三方方案作为子工程。git-submodule 就是完成这样一种子工程拆分与整合的工具。下面开始简单介绍一下 git-submodule 的用法。

A、带 submodule 的 git 仓库的创建
正如 git 仓库有两种创建方式一样，带 submodule 的 git 仓库的创建也有两种。

1、新建一个大型工程

?

```
1$ cd /path/to/massproj
2$ echo Example of Mass Project > README.md
3$ git init
4$ git add README.md
5$ git commit -m 'Mass Project created.'
```

接着使用 git submodule add 为新建的工程增加一个第三方子工程，例如

?

```
1$ git submodule add git@domain.com:subproj.git subproj
2$ git status
```

可能会注意到，git 只记录了 submodule 目录，而没有记录目录下的文件。实际上，git 是按照 commit id 来比对 submodule 变动的。直接递交本次更改

?

```
1$ git commit -m 'thirdpart submodule added.'
```

若有管理该项目的 Git 服务器，可上传该项目至服务器，例如

?

```
1$ git remote add origin git@domain.com:massproj.git
2$ git push origin master
```

2、克隆一个带 submodule 的 git 仓库

?

```
1$ cd /path/to/hold/projcet/
2$ git clone git@domain.com:massproj.git
```

不过不同于常规的 git 仓库克隆，还需要将该仓库中的子模块初始化以及更新：

?

```
1$ git submodule init
2$ git submoudle update
```

如果觉得这样克隆一个 git 仓库太麻烦，那么试试

?

```
1$ git clone --rescursive git@domain.com:massproj.git
```

该命令行会自动完成一系列必需的动作。

B、带 submodule 的 git 仓库更新
git 仓库正常更新过程是这样的：

?

```
1$ git pull
```

然后根据修改情况，自动解决冲突、递交变更并合并到 git 仓库。不过对于带 submodule 的 git 仓库的更新，就有两种情况了。

1、更新主 git 仓库，如前，先执行

?

```
1$ git pull
```

就要留意 submodule 的变更，先试试

?

```
1$ git status
```

若发现 submodule 有修改，需立即执行

?

```
1$ git submodule update
```

更复杂一些的情况是，如果有一个 submodule 依赖另一个 submodule，那么很可能需要在 git pull 和 git submodule update 之后，再分别到每个有依赖关系的 submodule 目录中再执行一次 git submodule update。为了免去麻烦，可以执行

?

```
1$ git submodule foreach "git submodule update"
```

来实现一次性更新所有 submodule。

2、更新某个子模块，需先进入子模块目录，然后执行普通的更新操作，例如

```
1$ cd subproj
2$ git pull
```

若有很多子模块需要做类似的操作，可简单执行

```
1$ git submodule foreach "git pull"
```

若子模块之间还有依赖关系，可采用

```
1$ git submodule foreach "git pull" --rescursive
```

来一次性更新所有模块。

不过需要注意的是，这样的更新操作都是在本地 git 仓库没有做任何修改的前提之下完成的。假如本地仓库有所改动，特别是本地仓库中的子模块有所修改，那么有可能还需要解决一系列冲突才能合并。这需要用到下面这段的知识。

C、带 submodule 的 git 仓库的修改
本地仓库的普通修改不是这里的主要内容，这里要讨论一下本地仓库中子模块的更改。比如我们拥有某个 submodule 的远程仓库操作权限，此时正好碰到该子工程被大型工程调用时需要修改代码。切回到的该子工程在本地的原先仓库位置进行相关操作当然略显麻烦，若能在该大型工程的子模块仓库中直接修改无疑更方便一些。这里就来介绍一下子模块修改的大概流程。

1、有一个重要的事实需要特别强调，在执行 git submodule update 时 git 默认并不会将 submodule 切到任何 branch。因此，submodule 的 HEAD 默认是处于游离状态的(‘detached HEAD’ state)。在子模块修改前，记得一定要将当前的 submodule 分支切换到相应分支，例如切换到 master 分支:

```
1$ cd /path/to/massproj/subproj
2$ git checkout master
```

然后才能对子模块做修改和提交。另外，子模块中所跟踪的远程分支可被 .gitmodules 或者 .git/config 的配置覆盖，例如

```
1$ nano -w .gitmodules
2[submodule "subproj"]
3   path = subproj
4   url = git@github.com:subproj.git
5   branch = master
```

实际上，.gitmodules 的修改可通过在增加 submodule 时直接指定跟踪分支自动完成，例如

```
1$ git submodule add git@github.com:subproj.git -b master
```

若有很多 submodule 需要类似分支切换操作，可使用如下命令

```
1$ git submodule foreach "git checkout master"
```

实际上，高版本的 git 解决这一问题的方式更简单，例如执行

```
1$ git submodule update --remote
```

来跟踪远程分支，默认是 origin/master；若子模块之间还有依赖关系，可采用

```
1$ git submodule update --remote --rescursive
```

一次性跟踪所有子模块的远程分支。

2、下面开始对子模块目录做一些修改，例如修改 README.md 文件：

```
1$ cd /path/to/massproj/subproj
2$ echo this is a new line. >> README.md
```

再将修改递交到 git 仓库中：

```
1$ git add README.md
2$ git commit -m "new comments added."
3$ git push
```

递交完之后，回到大型工程的顶层目录，该示例中就是

```
1$ cd ..
```

再试试

```
1$ git status
```

会看到子模块 subproj 还需要再做一次递交

```
1$ git add subproj
2$ git commit -m 'submodule updated'
3$ git push
```

由于 submodule 的更新只记录 commit id，因此，子模块的修改必须按上述流程递交版本。也即先在 submodule 内做递交，之后回到顶层目录再作一次递交，不然会引起 git 仓库的版本错乱。

3、如果你不慎在修改子模块之前忘记将它切换到远程跟踪分支，且又做了提交，此时可以用 cherry-pick 命令挽救。具体做法如下：先到该子模块将 HEAD 从游离状态切换到远程跟踪分支，例如 master 分支：

```
1$ cd subproj
2$ git checkout master
```

这时候，git 会报 Warning 说有一个提交没有在 branch 上，记住这个提交的 change-id（假如 change-id 为 aaaa)。随后将刚刚的提交重新作用远程跟踪分支上：

```
1$ git cherry-pick aaaa
```

最后只需将更新提交到远程版本库中：

```
1$ git push
```

D、git 仓库中 submodule 的更名或移动
随着大型工程的进行，有可能需要变更子模块，例如移动子模块的位置，或者第三方方案本身需要更名等。由于 submodule 的更名与移动两者差别不大，这里主要介绍子模块移动目录的流程。至于更名，只略微提及。假设要将 subproj 移动到 /massproj/platform 目录之下，先新建目录并移动子模块到里面

```
1$ cd /path/to/massproj
2$ mkdir platform
3$ mv subproj platform/
```

修改相关的配置文件，主要是 .gitmodules、.git/config、子模块目录中的 .git 以及 .git/modules 目录中几个文件与目录：

```
1$ sed -i 's@subproj@platoform/subproj@' .gitmodules
2$ sed -i 's@subproj@platoform/subproj@' platform/subproj/.git
3$ mv .git/modules/subproj .git/modules/platform/subproj
4$ sed -i 's@subproj@platoform/subproj@' .git/modules/platform/subproj/config
```

若是子模块更名，那么还请留意 .gitmodules、.git/config 以及 git/modules/<path/to/submodule>/config 中 url 设置，根据需要进行调整。最后让 git 记录所有的变更

```
1$ git rm --cached subproj
2$ git add platform/subproj .gitmodules
3$ git submodule sync -- platform/subproj
```

若觉得上述手续太烦且网络速度还可以的话，可直接删除原先的子模块，随后指定新路径添加模块。有关这个方法，需先了解一下后一段知识，再回过头来看一下以下操作

```
$ cd /path/to/massproj
$ git rm --cached subproj
$ rm -rf subproj
$ rm -rf .git/modules/subproj
$ nano -w .gitmodules
...remove subproj...
$ nano -w .git/config
...remove subproj...
$ git submodule add git@github.com:subproj.git platform/subproj -b master
$ git add .gitmodules
$ git commit -m 'subproj moved to platform/.'
$ git push
```

E、带 submodule 的 git 仓库中 submodule 的删除
这里以 platform/subproj 子模块为例介绍一下子模块删除的大概流程。先到大型工程的顶层目录清理子模块文件：

```
$ cd /path/to/massproj
$ git rm --cached platform/subproj
$ rm -rf platform/subproj
$ rm -rf .git/modules/platform/subproj
```

接着删除 git 中记录的相关数据

```
$ nano -w .gitmodules
...remove platform/subproj...
$ nano -w .git/config
...remove platform/subproj...
```

随后将变更递交到远程仓库

```
$ git add .gitmodule
$ git commit -m 'platform/subproj submodule removed.'
$ git push
```

评论 (0)

昵称 登录 E-mail: *
Web:
Twitter:

当有新评论通过 E-mail 通知我