使用Github协作开发dewey

目录

[使用Github协作开发dewey 1](#_Toc339120345)

[第一章github简单使用指南 2](#_Toc339120346)

[第二章 用eclipse加载本地库中的dewey 3](#_Toc339120347)

[**第三章 提交与更新** 7](#_Toc339120348)

[第一节 提交本地的更新 7](#_Toc339120349)

[第二节 把bettersaas的更新合并到自己Fork的版本中 11](#_Toc339120350)

[第四章 git电子书和其他资源 13](#_Toc339120351)

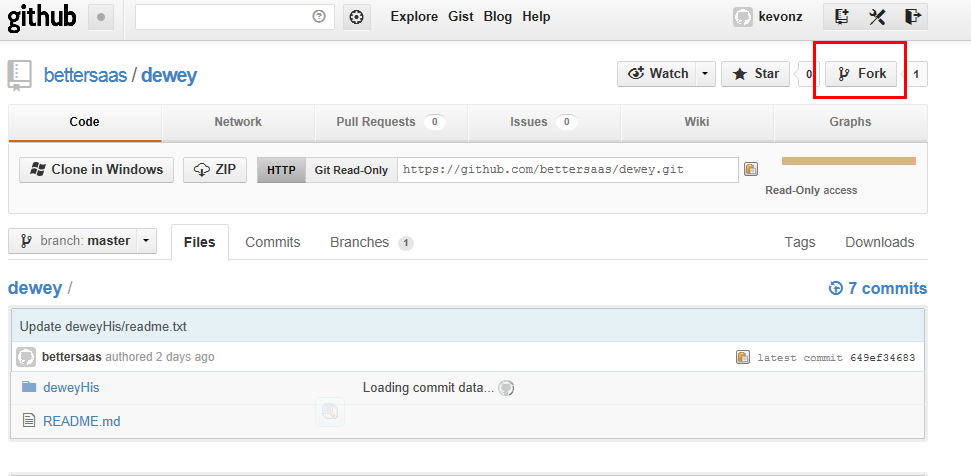
[官方文档http://help.github.com/ 13](#_Toc339120352)

[在线教程 http://gitimmersion.googol.im/ 13](#_Toc339120353)

[Git ppt介绍  ：http://www.wojilu.com/Forum1/Topic/1924 13](#_Toc339120354)

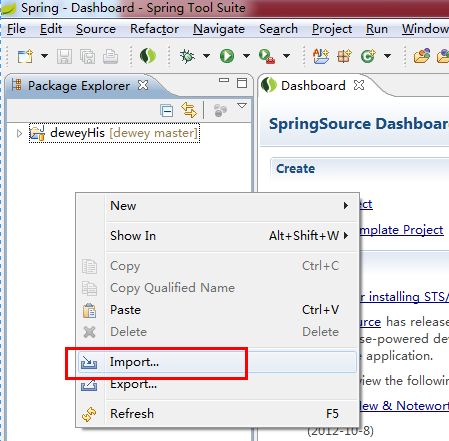
[转载一篇教程 Git入门教程 13](#_Toc339120355)

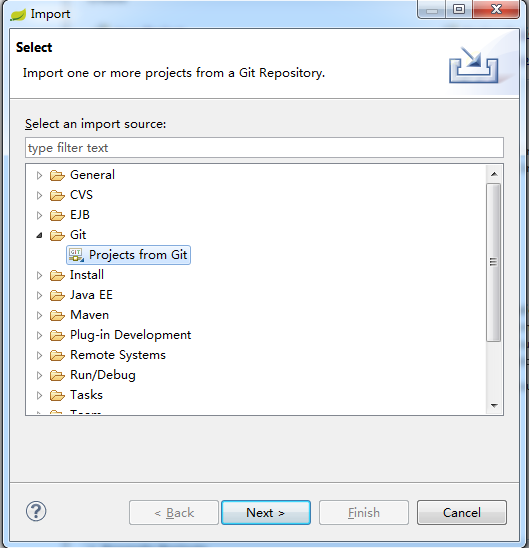
**第一章github简单使用指南**

1. 注册Github帐号. <https://github.com/signup/free>
2. 下载并安装Github的windows客户端<http://windows.github.com/>
3. 登录Github <https://github.com>
4. Fork dewey，先打开这个地址<https://github.com/bettersaas/dewey> 然后点击下图红框中的Fork
5. 打开Github客户端，并登录。客户端会自动加载Github信息
6. 选择左侧的Github，然后在主窗口的dewey上面点击右键，clone到本地。

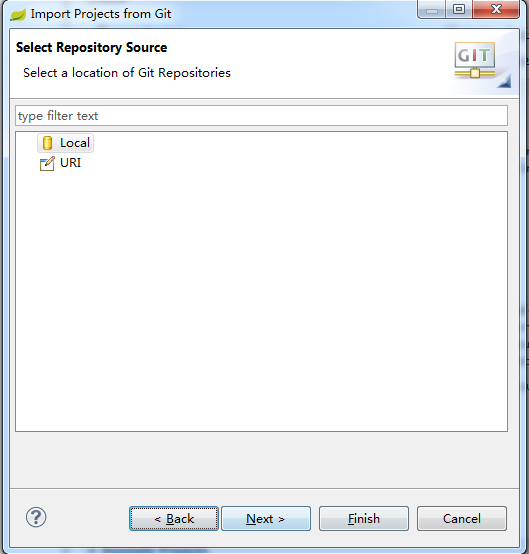
**第二章 用eclipse加载本地库中的dewey**

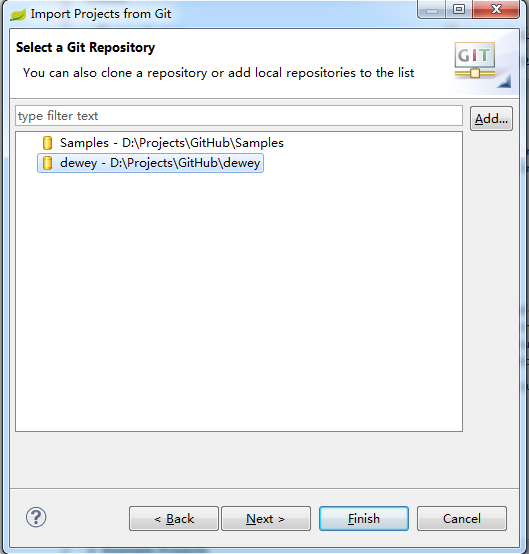
在eclipse（建议使用spring tool suites（简称STS） <http://www.springsource.org/downloads/sts-ggts>）中加载github中的dewey

1.在package explorer点右键，选择 import

2.依次选择Git -> Projects fromGit

3. 选择local，点击下一步

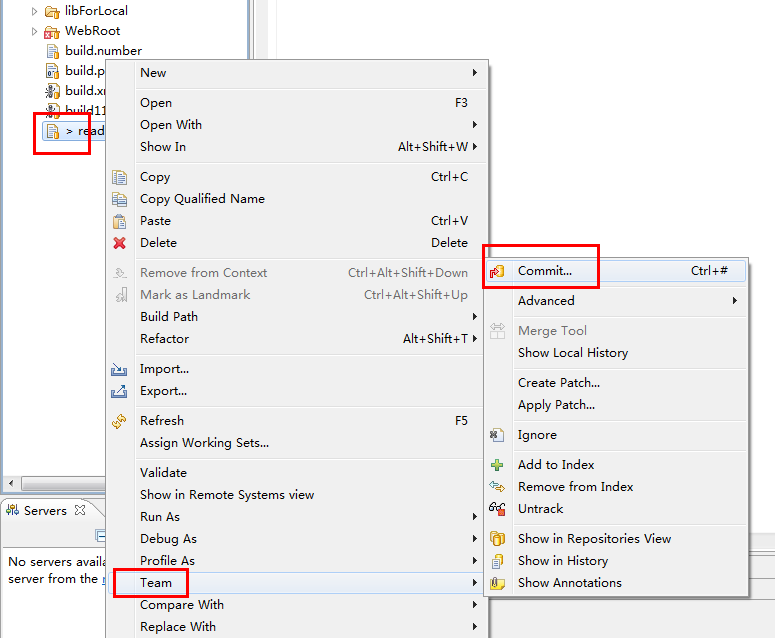
如果下图所示的界面上没有列出来本地的dewey路径，可以点击add手动选择本地计算机的github路径



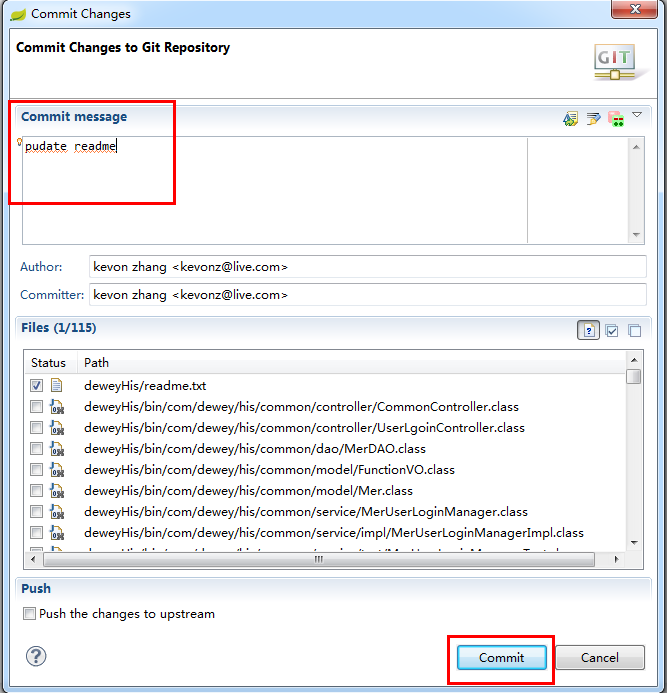
点击finish即可完成项目的导入。

**第三章 提交与更新**

### 第一节 提交本地的更新

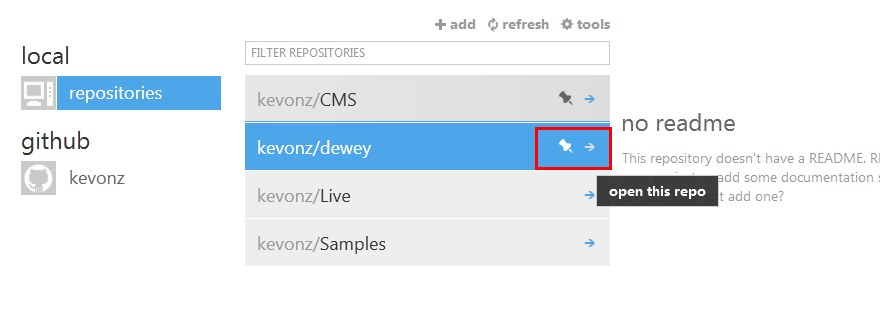
在STS中修改了代码以后，左侧package explorer中显示大于号(>)的文件表示本地更新的，要提交修改的服务器，在上面点击右键，如下图所示：

点击Commit，在如下窗口中写入注释，点击Commit。

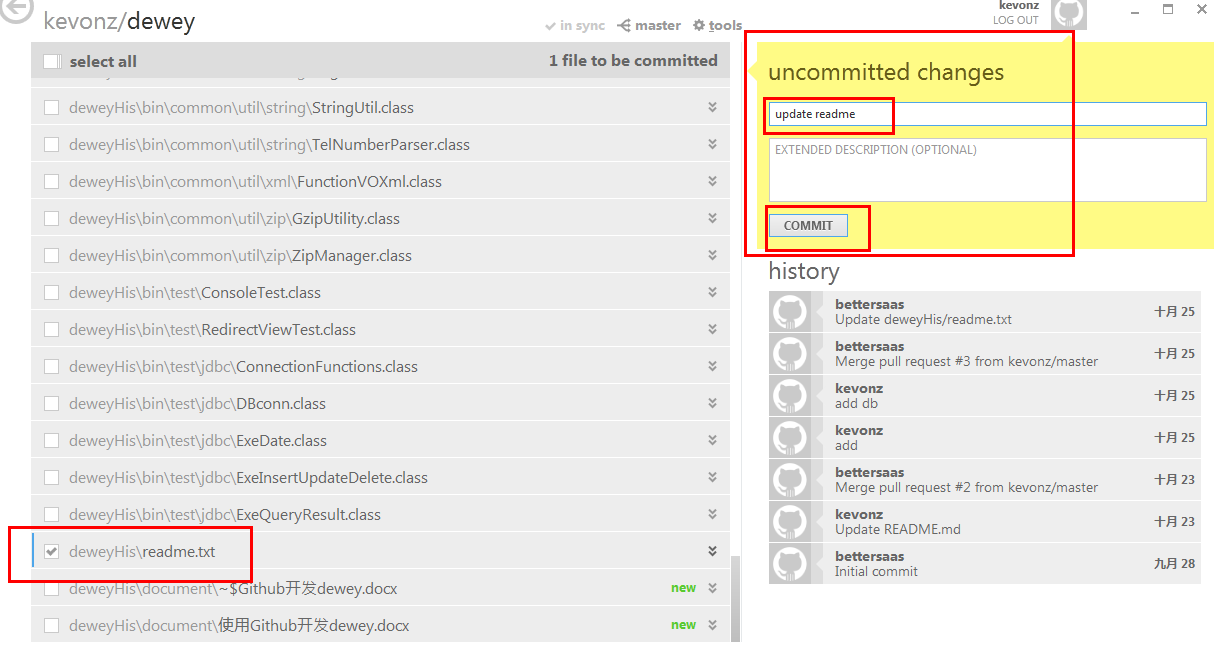


到这里，完成了代码向本地库的更新，要更新到主库（即到bettersaas中，还需要两步）。

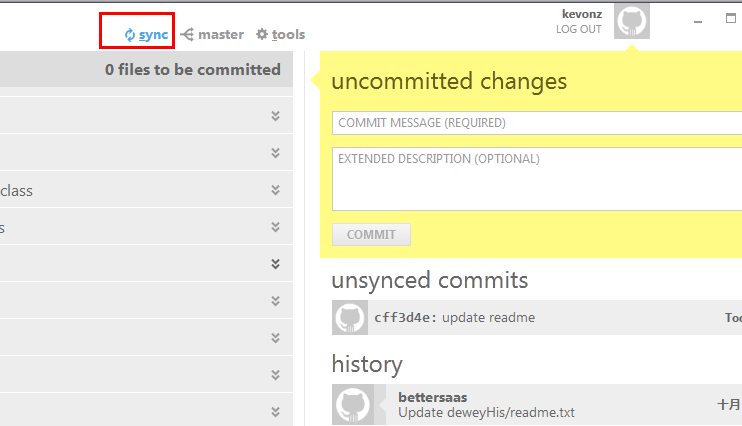
1. 打开github客户端，打开local的dewey



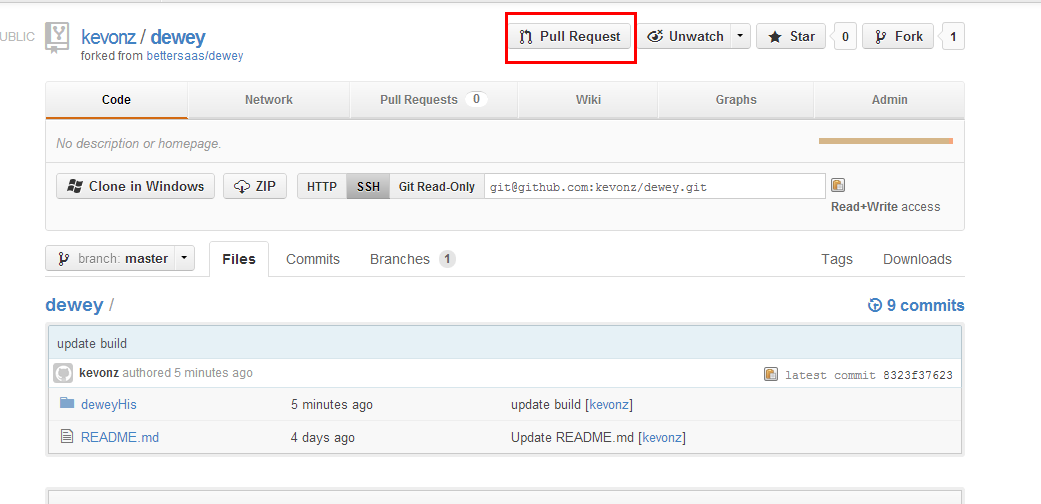
在左侧选择要更新的文件，在右上角写入注释。点击Commit



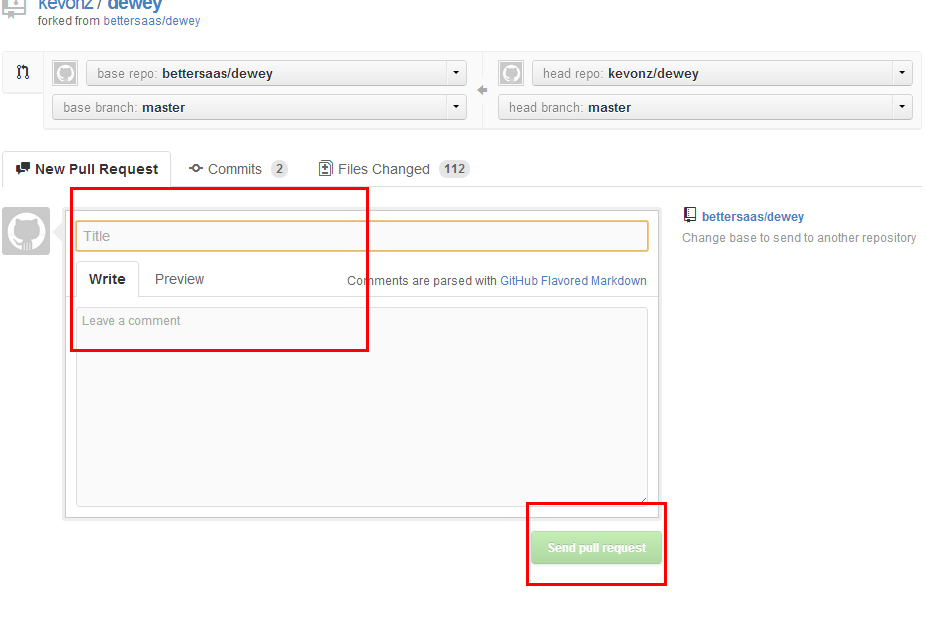
接下来，点击Sync



到这里，本地的更新已经提交到了自己的github库里面了，下面登录github网站，打开自己的dewey库，点击里面的Pull Request，如下图所示。



写入注释，点击send pull request即可完成提交到主库。



主库审核通过之后，即最终完成本次更新。

### 第二节 把bettersaas的更新合并到自己Fork的版本中

当主项目更新后，通过以下3条命令把更新合并到自己Fork的版本里面。

# 1 添加 bettersaas的 remote 地址（saas是别名），

D:\Projects\GitHub\dewey [master]> git remote add saas git://github.com/bettersaas/dewey.git

# 2 用 fetch 命令获取 saas 的所有分支

~D:\Projects\GitHub\dewey [master]> git fetch saas

From git://github.com/bettersaas/dewey

\* [new branch] master -> saas/master

#3 将 saas 的 master 分支的改动合并过来,目前是处与 master 分支

D:\Projects\GitHub\dewey [master]> git merge saas/master

Already up-to-date.

D:\Projects\GitHub\dewey [master]>

窗口截屏如下

|  |  |
| --- | --- |
|  |  |

**第四章 git电子书和其他资源**

官方文档<http://help.github.com/>

在线教程 <http://gitimmersion.googol.im/>

Git ppt介绍  ：<http://www.wojilu.com/Forum1/Topic/1924>

### 转载一篇教程 [Git入门教程](http://hi.baidu.com/wangkangfeiyu/item/e68acbe7b9740b3b4cdcaf12)

**1.   概述**  
    对于软件版本管理工具，酷讯决定摒弃CVS而转向Git了。  
    为什么要选择Git？ 你真正学会使用Git时， 你就会觉得这个问题的回答是非常自然的。然而当真正需要用文字来回答时，却觉得文字好像不是那么够用。 咳，该则么回答呢？  
    其实，关键的问题不在于如何回答这个问题。 问题的关键是公司已经决定使用它了。那么，我们的程序员们！ 请开动你们的浏览器，请拿出你的搜索引擎工具，去自己发掘答案吧。在这里，我只能给你们一个最朦胧的感觉。  
    Git和 CVS、SVN不同，是一个分布式的源代码管理工具。Linux内核的代码就是用Git管理的。它很强，也很快。它给我们带来的直接好处有：  
1.       傻瓜都会的初始化，git init, git commit -a, 就完了。对于随便写两行代码就要放到代码管理工具里的人来说，再合适不过。也可以拿git做备份系统，或者同步两台机器的文档，都很方便。   
2.         绝大部分操作在本地完成，不用和集中的代码管理服务器交互，终于可以随时随地大胆地check in代码了。 只有最终完成的版本才需要向一个中心的集中的代码管理服务器提交。  
3.         每次提交都会对所有代码创建一个唯一的commit id。不像CVS那样都是对单个文件分别进行版本的更改。所以你可以一次性将某次提交前的所有代码check出来，而不用考虑到底提交过那些文件。（其实SVN也可以做到这点）  
4.         branch管理容易多了，无论是建立新的branch，还是在branch之间切换都一条命令完成，不需要建立多余的目录。   
5.         branch之间merge时，不仅代码会merge在一起，check in历史也会保留，这点非常重要。   
6.         … 太多了  
    当然，Git也会带给我们一些困难，首先，你想要使用好git，就要真正明白它的原理，理解它的观念， 对以那些CVS的熟手来说，改变你已经固有的纯集中式源代码管理的观念尤为重要，同时也会让你觉得有些困难。在使用git的初期，你可能会觉得有些困难，但等你逐渐明白它时，你绝对会喜欢上它。这是一定的，就像我问你“喜欢一个温吞如水、毫无感觉的主妇，还是喜欢一个奔放如火，让你爱的痴狂恨的牙痒的情人”一样毋庸置疑。  
    下面，就让我们进入学习Git之旅…  
    请记住，这只是一个非常简单而且初级的教程， 想要成为git的专家，需要各位同事不断的自己深入挖掘。  
**2.   Git基础命令**

**2.1   创建Git库—git-init**  
    你们曾经创建过CVS的库么？应该很少有人操作过吧？因为很多人都是从CVS库里checkout代码。同样，在合作开发中，如果你不是一个代码模块的发起者，也不会使用到这个命令，更多的是使用git-clone（见2.7节）。 但是，如果你想个人开发一个小模块，并暂时用代码管理工具管理起来（其实我就常这么做，至少很多个人开发过程都可以保留下来，以便备份和恢复），创建一个Git库是很容易和方便的。  
对于酷讯来说，当一个代码的Git库创建后，会添加代码文件到库里，并将这个库放到公司一个专门用来进行代码管理的服务器上，使大家可以在以后clone（不明白？没关系，继续往后看就明白了）它。对于个人来说，你可以随便将这个库放到哪里，只要你能访问的到就行。

    创建一个Git库是很容易和方便的，只要用命令 git-init 就可以了。在Git1.4之前（包括git1.4）的版本，这个命令是git-init。  
a)         $ mkdir dir  
b)        $ cd dir  
c)         $ git-init  
    这样，一个空的版本库就创建好了，并在当前目录中创建一个叫 .git 的子目录。以后，所以的文件变化信息都会保存到这个目录下，而不像CVS那样，会在每个目录和子目录下都创建一个讨厌的CVS目录。  
在.git目录下有一个config文件， 需要我们添加一下个人信息后才能使用。否则我们不能对其中添加和修改任何文件。  
    原始的config文件是这样的，  
[core]  
        repositoryformatversion = 0  
        filemode = true  
        bare = false  
        logallrefupdates = true  
我们需要加入  
[user]   
        name = xxx  
        emai= xxx@kuxun.cn  
    现在已经创建好了一个 git 版本库，但是它是空的，还不能做任何事情，下一步就是怎么向版本库中添加文件了。如果希望忽略某些文件，需要在git库根目录下添加. gitignore文件。  
**2.2   一条重要的命令 -- git-update-index**  
    在介绍如何向git库中添加文件前，不得不先介绍git-update-index命令。这条命令可能会使很多熟悉CVS的用户疑惑， 一般来说，我们向一个源代码管理库提交代码的更改，都会抽象为以下的动作：更改文件；向源码管理系统标识变化；提交。比如从一个CVS库里删除一个文件，需要先删除文件，然后cvs delete； 最后cvs commit。   
因此， git-update-index就是向源码管理系统标识文件变化的一个抽象操作。说的简要一些，git-update-index命令就是通知git库有文件的状态发生了变化（新添、修改、删除等待）。这条命令在早期的git版本中是非常常用的。 在新的git版本（1.5版本及以后）已经被其它命令包装起来，并且不推荐使用了。  
    git-update-index最常用的方式有以下两种，更多功能请man git-update-index。  
 方法一：git-update-index --add 文件名列表。 如果文件存在，则这条命令是向git库标识该文件发生过变化（无论是否该文件确实被修改过），如果文件不存在，则这条命令是向git库表示需要加入一个新文件。  
 方法二： git-update-index --force-remove 文件名列表。 这表示向git库表示哟啊从库中删除文件。无论该文件是否已经被删除，这条命令仅仅是通知git库要从库中删除这些文件。这些文件都不会受影响。  
    因此，git-update-index仅仅是向git库起到一个通知和标识的作用，并不会操作具体的文件。  
**2.3   向git库中添加或删除文件 – git-add、git-rm**  
    其实，说使用git-add命令向git库里添加文件是不对的， 或者说至少是不全面的。git-add 命令的本质是命令"git-update-index --add” 的一个包装。因此，git-add除了可以添加文件，还可以标识文件修改。在调用了git-add后，才可以做commit操作。git-rm 也是一样， 它是git-update-index --force-remove的一个包装。  
    对于git-add来说， 如果在一个目录下调用了git-add \* ,则默认是递归将子目录中所有文件都add到git库中。对于git-rm来说，也是一样。 这点和CVS有较大区别。  
    此外，我们还可以通过命令git-ls-files来查看当前的git库中有那些文件。

**2.4   查看版本库状态—git-status**  
    通过该命令，我们可以查看版本库的状态。可以得知那些文件发生了变化，那些文件还没有添加到git库中等等。 建议每次commit前都要通过该命令确认库状态。以避免误操作。  
    其总，最常见的误操作是， 修改了一个文件， 没有调用git-add通知git库该文件已经发生了变化就直接调用commit操作， 从而导致该文件并没有真正的提交。如果这时如果开发者以为已经提交了该文件，就继续修改甚至删除这个文件，那么修改的内容就没有通过版本管理起来。如果每次在提交前，使用git-status查看一下，就可以发现这种错误。因此，如果调用了git-status命令，一定要格外注意那些提示为“Changed but not updated:”的文件。 这些文件都是与上次commit相比发生了变化，但是却没有通过git-add标识的文件。  
**2.5   向版本库提交变化 – git-commit**  
    直接调用git-commit命令，会提示填写注释。也可以通过如下方式在命令行就填写提交注释：git-commit -m "Initial commit of gittutor reposistory"。 注意，和CVS不同，git的提交注释必须不能为空。否则就会提交失败。  
    git-commit还有一个 –a的参数，可以将那些没有通过git-add标识的变化一并强行提交，但是不建议使用这种方式。  
    每一次提交，git就会为全局代码建立一个唯一的commit标识代码，用户可以通过git-revert命令恢复到任意一次提交时的代码。 这比CVS不同文件有不同的版本呢号管理可方便多了。（和SVN类似）  
    如果提交前，想看看具体那些文件发生变化，可以通过git-diff来查看， 不过这个命令的输出并不友好。因此建议用别的工具来实现该功能。在提交后，还可以通过git-log命令来查看提交记录。  
**2.6   分支管理 – git-branch**  
    我们迎来了git最强大，也是比CVS、SVN强大的多的功能 — 分支管理。  
    大概每个程序员都会经常遇到这样的情况：   
1.       需要立刻放下手头的工作，去修改曾经一个版本的bug并上线，然后再继续当的工作。  
2.       本想向中心库commit一个重要修改，但是由于需要经常备份代码，最终不得不频繁的向中心库commit。从而导致大量无用的commit信息被保留在中心库中。  
3.       将一次修改提交同事进行code review， 但是由于同事code review比较慢， 得到反馈时，自己的代码已经发生了变化，从而倒是合并异常困难  
    这些场景，如果用CVS或者SVN来解决，虽说不一定解决不了，但过程之繁琐，之复杂，肯定另所有人都有生不如死的感觉吧！究其关键，就是CVS或者SNV的branch管理太复杂，基本不具可用性。  
    在 git 版本库中创建分支的成本几乎为零，所以，不必吝啬多创建几个分支。当第一次执行git-init时，系统就会创建一个名为”master”的分支。 而其它分支则通过手工创建。下面列举一些常见的分支策略，这些策略相信会对你的日常开发带来很大的便利。   
 1.创建一个属于自己的个人工作分支，以避免对主分支 master 造成太多的干扰，也方便与他人交流协作。   
 2.当进行高风险的工作时，创建一个试验性的分支，扔掉一个烂摊子总比收拾一个烂摊子好得多。   
 3.合并别人的工作的时候，最好是创建一个临时的分支用来合并，合并完成后在“fatch”到自己的分支（合并和fatch后面有讲述，不明白就继续往下看好了）  
**2.6.1 查看分支 – git-branch**  
    调用git-branch可以查看程序中已经存在的分支和当前分支  
**2.6.2 创建分支 – git-branch 分支名**  
    要创建一个分支，可以使用如下方法:  
1.       git-branch 分支名称  
2.       git-checout –b 分支名  
    使用第一种方法，虽然创建了分支，但是不会将当前工作分支切换到新创建的分支上，因此，还需要命令”git-checkout 分支名” 来切换， 而第二种方法不但创建了分支，还将当前工作分支切换到了该分支上。  
    另外，需要注意，分支名称是有可能出现重名的情况的， 比如说，我在master分支下创建了a和b两个分支， 然后切换到b分支，在b分支下又创建了a和c分支。 这种操作是可以进行的。 此时的a分支和master下的a分支实际上是两个不同的分支。 因此，在实际使用时，不建议这样的操作，这样会带来命名上的疑惑。  
**2.6.3 删除分支 – git-branch –D**  
    git-branch –D 分支名可以删除分支，但是需要小心，删除后，发生在该分支的所有变化都无法恢复。   
**2.6.4 切换分支 – git-checkout 分支名**  
    如果分支已经存在， 可以通过 git-checkout 分支名 来切换工作分支到该分支名  
**2.6.5 查看分支历史 –git-show-branch**  
    调用该命令可以查看分支历史变化情况。 如：  
\* [dev1] d2  
! [master] m2  
--  
\* [dev1] d2  
\* [dev1^] d1  
\* [dev1~2] d1  
\*+ [master] m2  
    在上述例子中， “--”之上的两行表示有两个分支dev1和master， 且dev分支上最后一次提交的日志是“d2”,master分支上最后一次提交的日志是”m2”。 “--”之下的几行表示了分支演化的历史，其中 dev1表示发生在dev分支上的最后一次提交，dev^表示发生在dev分支上的倒数第二次提交。dev1~2表示发生在dev分支上的倒数第三次提交。  
**2.6.6 合并分支 – git-merge**  
    git-merge的用法为：git-merge “some memo” 合并的目标分支 合并的来源分支。如：  
    git-merge master dev1~2  
    如果合并有冲突，git会由提示，当前，git-merge已经很少用了， 用git-pull来替代了。  
    用法为：git-pull 合并的目标分支 合并的来源分支。 如git-pull . dev1^

**2.7   远程获取一个git库 git-clone**  
    在2.1节提到过，如果你不是一个代码模块的发起者，也不会使用到git-init命令，而是更多的是使用git-clone。通过这个命令，你可以从远端完整获取一个git库，并可以通过一些命令和远端的git交互。  
    基于git的代码管理的组织结构，往往形成一个树状结构，开发者一般从某个代码模块的管理者的git库通过git-clone取得开发环境，在本地迭代开发后，再提交给该模块的管理者，该模块的管理者检查这些提交并将代码合并到自己的库中，并向更高一级的代码管理者提交自己的模块代码。  
    对于酷讯来说，公司会有一个中心的git库， 大家在开发时，都是从中心库git-clone获取最新代码。  
git-clone的使用方法如下： git-clone [ssh://]username@ipaddr:path。 其中， “ssh://”可选，也有别的获取方式，如rsync。 Path是远端git的根路径，也叫repository。  
    通过git-clone获取远端git库后，.git/config中的开发者信息不会被一起clone过来。仍然需要为.git/config文件添加开发者信息。此外，开发者还需要自己添加. gitignore文件  
   另外，通过git-clone获取的远端git库，只包含了远端git库的当前工作分支。如果想获取其它分支信息，需要使用”git-branch –r” 来查看， 如果需要将远程的其它分支代码也获取过来，可以使用命令” git checkout -b 本地分支名 远程分支名”，其中，远程分支名为git-branch –r所列出的分支名， 一般是诸如“origin/分支名”的样子。如果本地分支名已经存在， 则不需要“-b”参数。

**2.8   从远程获取一个git分支 – git-pull**  
    与git-clone不同， git-pull可以从任意一个git库获取某个分支的内容。用法如下：  
    git-pull : 远端repository名 远端分支名:本地分支名。这条命令将从远端git库的远端分支名获取到本地git库的一个本地分支中。其中，如果不写本地分支名，则默认pull到本地当前分支。  
需要注意的是，git-pull也可以用来合并分支。 和git-merge的作用相同。 因此，如果你的本地分支已经有内容，则git-pull会合并这些文件，如果有冲突会报警。

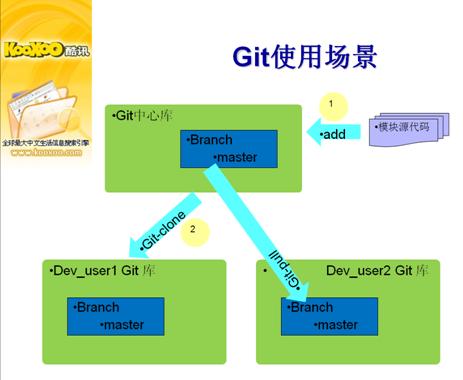
**2.9   将本地分支内容提交到远端分支 – git-push**  
    git-push和git-pull正好想反，是将本地某个分支的内容提交到远端某个分支上。用法：  
git-push : 远端repository名 本地分支名:远端分支名。这条命令将本地git库的一个本地分支push到远端git库的远端分支名中。

    需要格外注意的是，git-push好像不会自动合并文件。这点我的试验表明是这样，但我不能确认是否是我用错了。因此，如果git-push时，发生了冲突，就会被后push的文件内容强行覆盖，而且没有什么提示。 这在合作开发时是很危险的事情。  
**2.10     库的逆转与恢复 – git-reset**  
    库的逆转与恢复除了用来进行一些废弃的研发代码的重置外，还有一个重要的作用。比如我们从远程clone了一个代码库，在本地开发后，准备提交回远程。但是本地代码库在开发时，有功能性的commit，也有出于备份目的的commit等等。总之，commit的日志中有大量无用log，我们并不想把这些log在提交回远程时也提交到库中。 因此，就要用到git-reset。  
    Git-reset的概念比较复杂。它的命令形式：git-reset [--mixed | --soft | --hard] [<commit-ish>]   
命令的选项：  
--mixed   
    这个是默认的选项。 如git-reset [--mixed] dev1^(dev1^的定义可以参见2.6.5)。它的作用仅是重置分支状态到dev1^, 但是却不改变任何工作文件的内容。即，从dev1^到dev1的所有文件变化都保留了，但是dev1^到dev1之间的所有commit日志都被清除了，而且，发生变化的文件内容也没有通过git-add标识，如果您要重新commit，还需要对变化的文件做一次git-add。 这样，commit后，就得到了一份非常干净的提交记录。   
--soft   
    相当于做了git-reset –mixed，后，又对变化的文件做了git-add。如果用了该选项， 就可以直接commit了。  
--hard   
    这个命令就会导致所有信息的回退， 包括文件内容。 一般只有在重置废弃代码时，才用它。 执行后，文件内容也无法恢复回来了。

**2.11     更多的操作**  
    之前的10节只简要介绍了git的基本命令，更多的细节可以去linux下man git的文档。此外，<http://www.linuxsir.org/main/doc/git/gittutorcn.htm> 也有不少更详细的介绍。

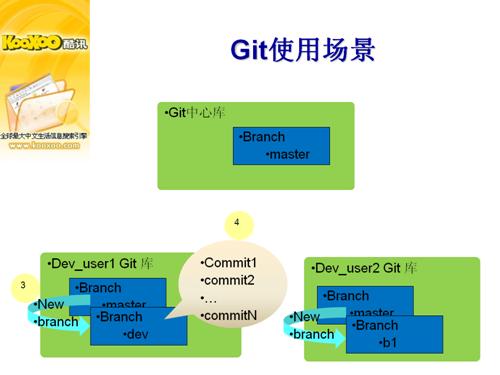
**3.   基于git的合作开发**  
对于酷讯来说，当我们采用了Git，如何进行合作开发呢？ 具体步骤如下：  
**3.1   获取最新代码**

    酷讯会准备一个中心git代码库。首先，我们将整理好的代码分模块在git中心库中建立git库。并将文件add到中心库中。 接下来，开发者通过git-clone将代码从中心库clone到本地开发环境。   
对于较大的项目，我们还建议每个组选择一个负责人，由这个负责人负责从中心库获取和更新最新的代码，其它开发者从这个负责人的git代码库中clone代码。此时，对开发者来说，这个负责人的git库就是中心库了。



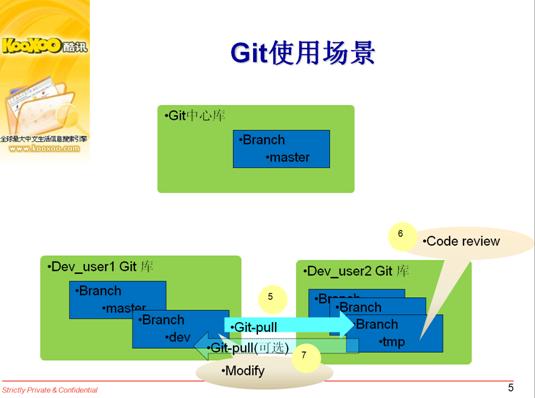
**3.2 开发者在本地进行迭代开发**

    当用户将代码clone到本地后， 就可以进行本地的迭代开发，建议用户不要在master分支上开发，而是建立一个开发分支进行开发。 在本地开发中，用户可以随意的创建临时分支，随意commit。



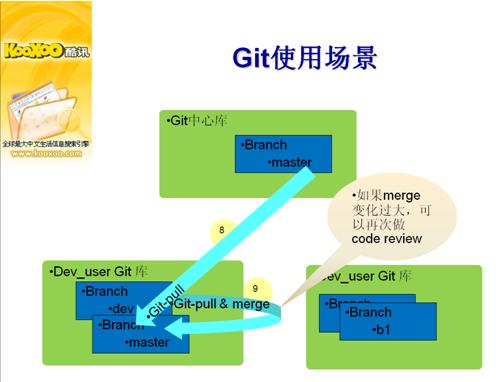
**3.3   开发者请其它同事进行code review**

    当本地开发完毕，可以请其它同事进行code review。过程为：  
1. user2通通过git-pull命令，将开发者(user1)的开发分支（dev）pull到user2本地的一个tmp分支，并切换工作分支到该分支上进行code review。  
2. 完成code review后， user2切换回其原有开发分支继续开发，并告知user1已经修改完毕。  
3. User1将user2的tmp分支git-pull到本地tmp分支，并和dev分支进行merge。最终得到一个code review后的dev分支。  
    当然，user2也可以直接坐在user1旁边在他的代码上进行review。而不需要走上述步骤。（图中第7步，不是git-pull,而是直接在dev分支上和user1边review边modify）



**3.4   和中心库进行代码合并**

    使用过CVS的人都知道， 在commit之前，都要做一次cvs update，以避免和中心库冲突。Git也是如此。  
现在我们已经经过了code review， 准备向中心库提交变化了， 在开发的这段时间，也许中心库发生了变化， 因此，我们需要在向中心库提交前，再次将中心库的master分支git-pull到本地的master分支上。并且和dev分支做合并。最终，将合并的代码放入master分支。  
    如果开发过程提交日志过多，可以考虑参照2.10节的介绍做一次git-reset。  
    此外，如果发现合并过程变化非常多， 出于代码质量考虑，建议再做一次code review



**3.5   提交代码到中心库**

    此时，已经完全准备好提交最终的代码了。 通过git-push就可以了。



**3.6   合作流程总结**  
    大家可以看到，使用git进行合作开发，这一过程和CVS有很多相似性，同时，增强了以下几个环节：  
1.       开发者在本地进行迭代开发，可以经常的做commit操作且不会影响他人。 而且即使不在线也可以进行开发。只需要最后向中心库提交一次即可。  
2.       大家都知道，如果CVS管理代码，由于我们会常常做commit操作。但是在commit之前cvs update时常会遇到将中心库上的其它最新代码checkout下来的情况，此时，一旦出现问题，就很难确认到底是自己开发的bug还是其它用户的代码带来了影响。 而使用git则避免了用户间的开发互相影响。  
3.       更有利于在代码提交前做code review。 以往用cvs， 都是代码提交后才做code view。如果发生问题， 也无法避免服务器上有不好的代码。 但是用git， 真正向中心库commit前，都是在本地开发，可以方便的进行code review， 然后才提交到中心库。更有利于代码质量。而且， 大家应该可以感到，使用git的过程中，更容易对代码进行code review，因为影响因素更小。  
4.       创建多分支，更容易在开发中进行多种工作，而使工作间不会互相影响。 比如user2对user1的代码进行code review时，就可以非常方便的保留当时的开发现场，并切换到user1的代码分支，在code review完毕后，也可以非常方便的切换会曾经被中断的工作现场。

    诚然，带来这些好处的同时，确实也使得操作比CVS复杂了一些。但我们觉得和前面所能获得的好处相比，这些麻烦是值得的。 当大家用惯了之后会发现，这并不增加多大的复杂性， 而且开发流程会更加自然。

    请大家多动手，多尝试! 去体验git的魅力所在吧！let’s enjoy it!