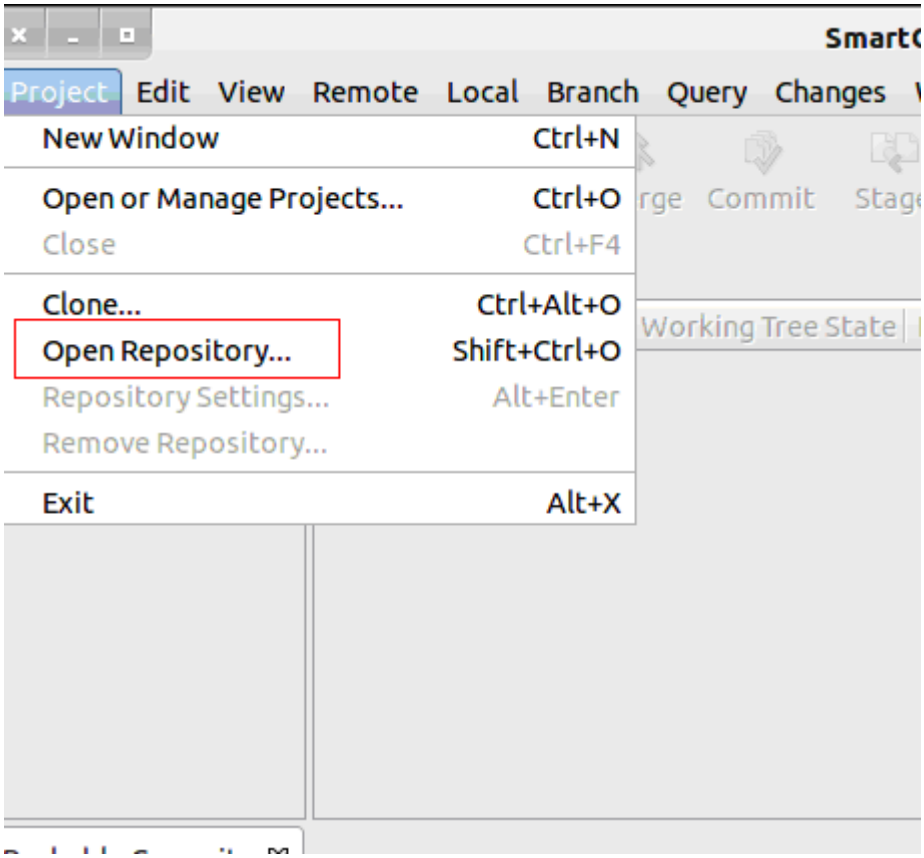
1.克隆 git clone

先把该项目的 Git 仓库复制一份出来

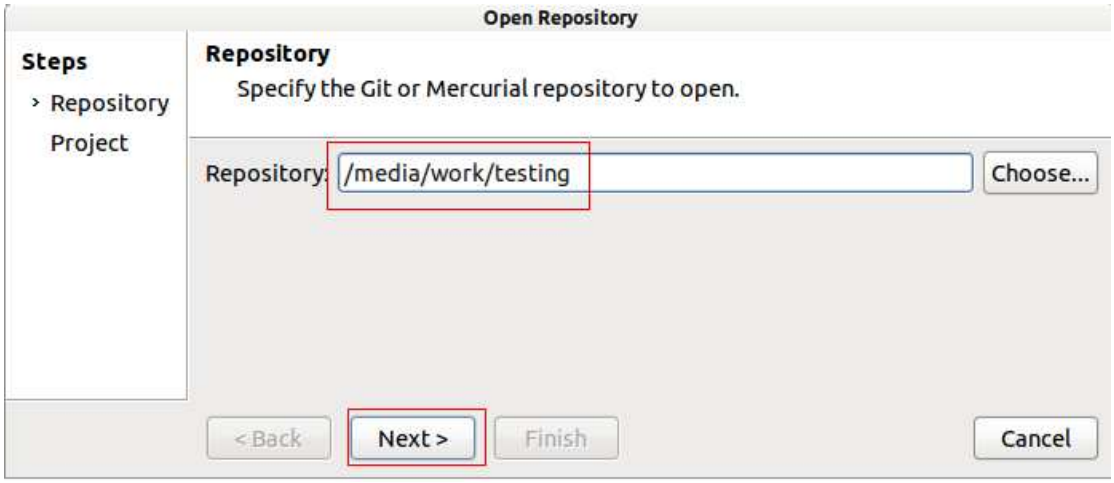
git clone git@10.219.68.248:testing

2. smart git 基础

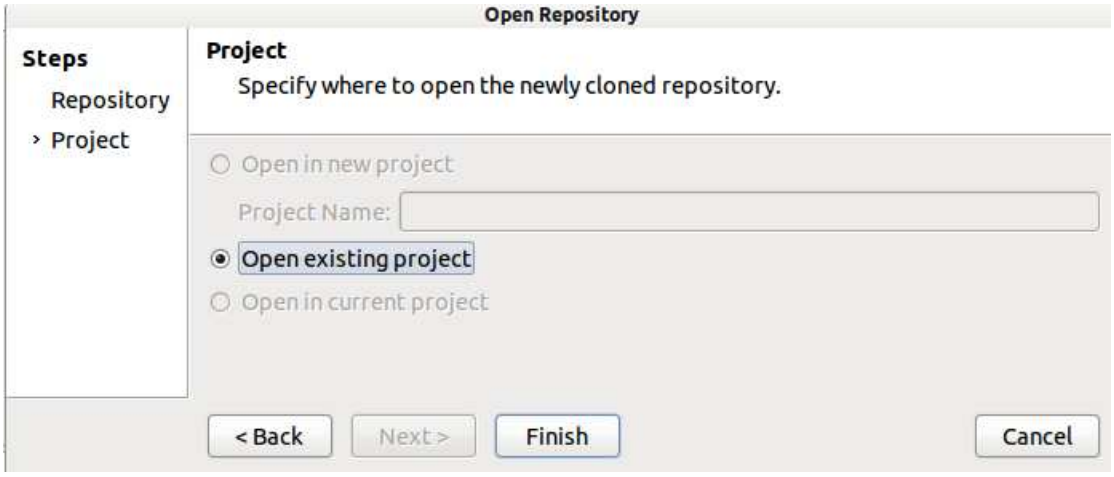
2.1 从程序中启动 smart git

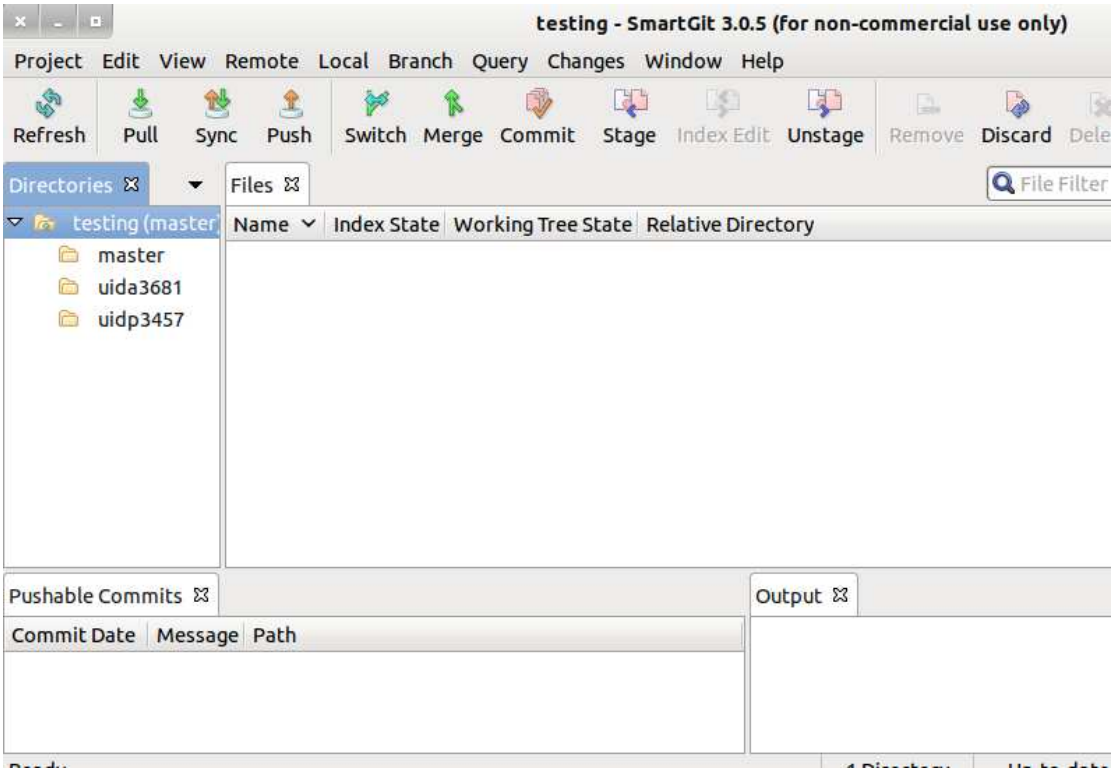


从工程 Project –> Open Repository 如下



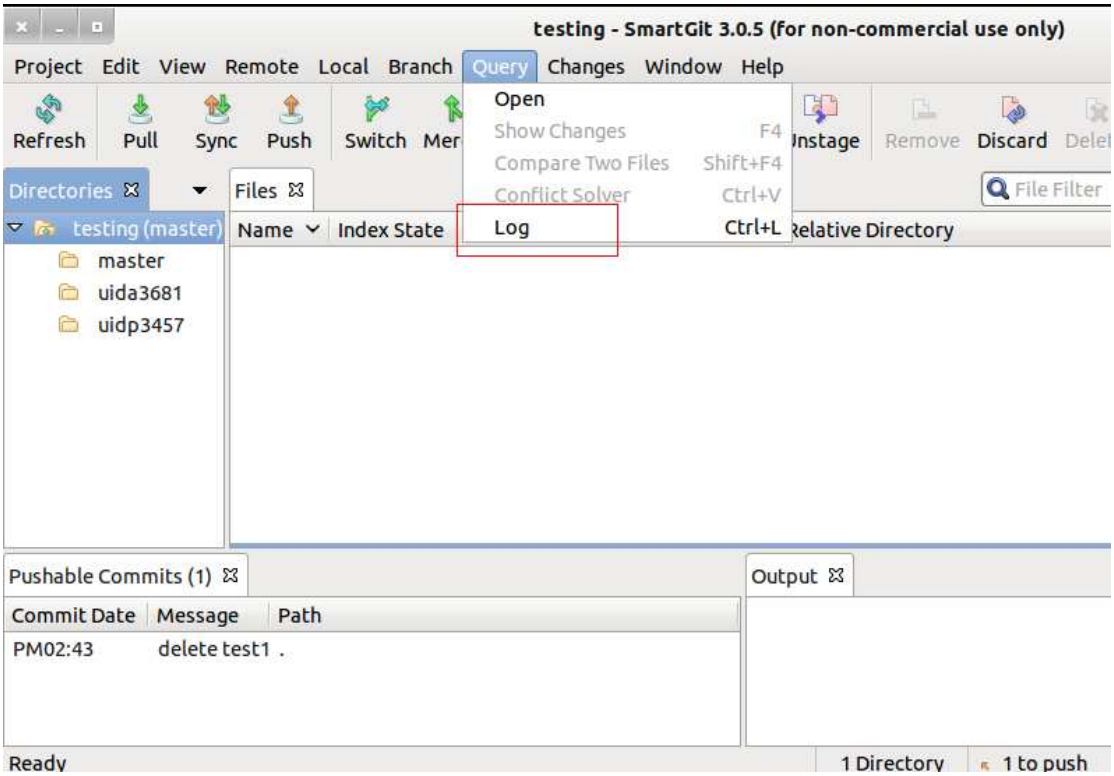
选择 clone 下来的 git 库的路径，点击 next
点击 finish。

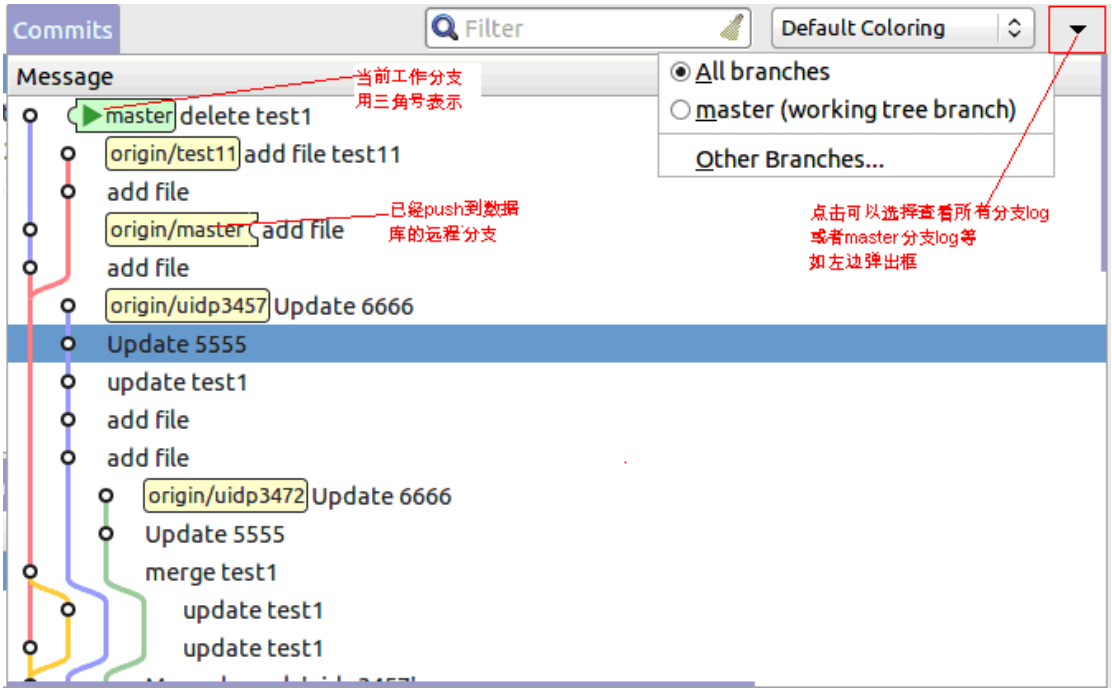




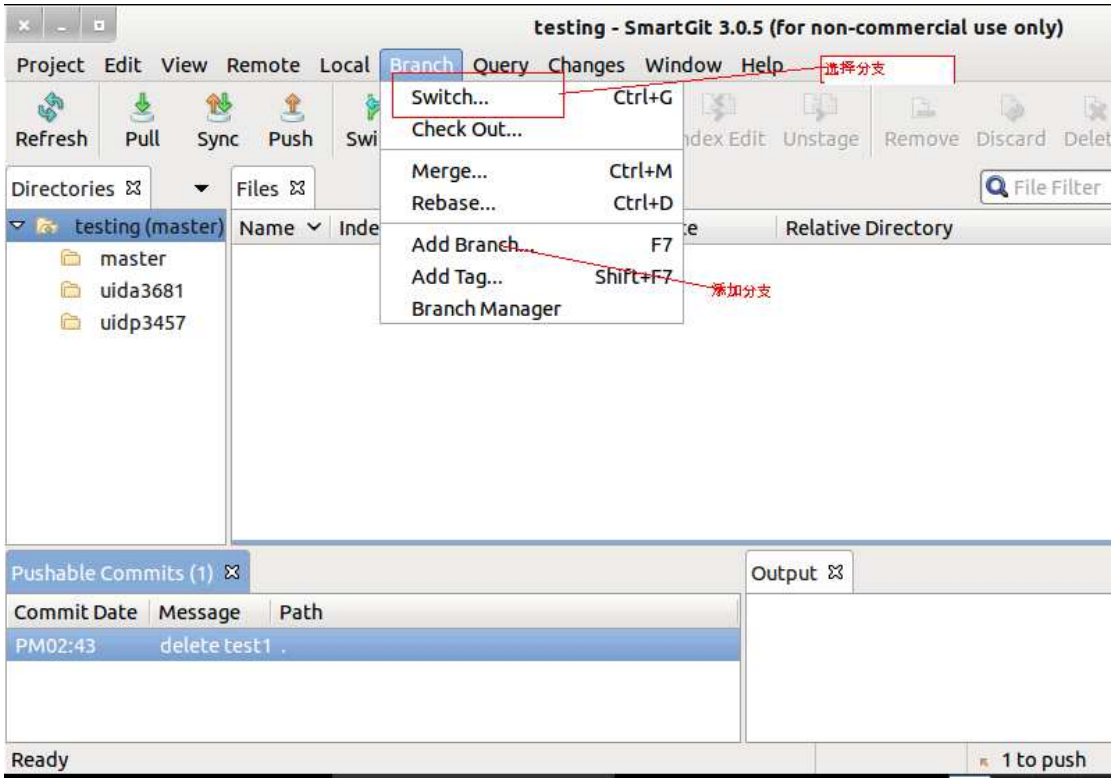
2.2 查看 log

Query -> Open


testing - SmartGit 3.0.5 (for non-commercial use only)
Project Edit View Remote Local Branch Query Changes Window Help
Refresh Pull Sync Push Switch Mer
Open
Show Changes F4
Compare Two Files Shift+F4
Conflict Solver Ctrl+V
Log Ctrl+L
Unstage Remove Discard Delet
Directories
Files
File Filter
testing (master)
master
uida3681
uidp3457
Name
Index State
Relative Directory
Pushable Commits (1)
Output
Commit Date
Message
Path
PM02:43
delete test1 .
Ready
1 Directory
1 to push

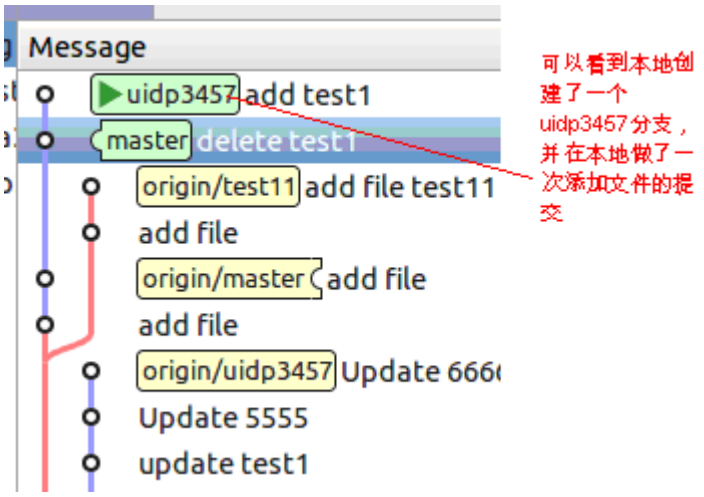

Commits
Filter
Default Coloring
Message
当前工作分支
用三角号表示
master delete test1
origin/test11 add file test11
add file
origin/master add file
已经push到数据
库的远程分支
add file
origin/uidp3457 Update 6666
Update 5555
update test1
add file
add file
origin/uidp3472 Update 6666
Update 5555
merge test1
update test1
update test1
All branches
master (working tree branch)
Other Branches...
点击可以选择查看所有分支log
或者master分支log等
如左边弹出框

2.3 选择分支

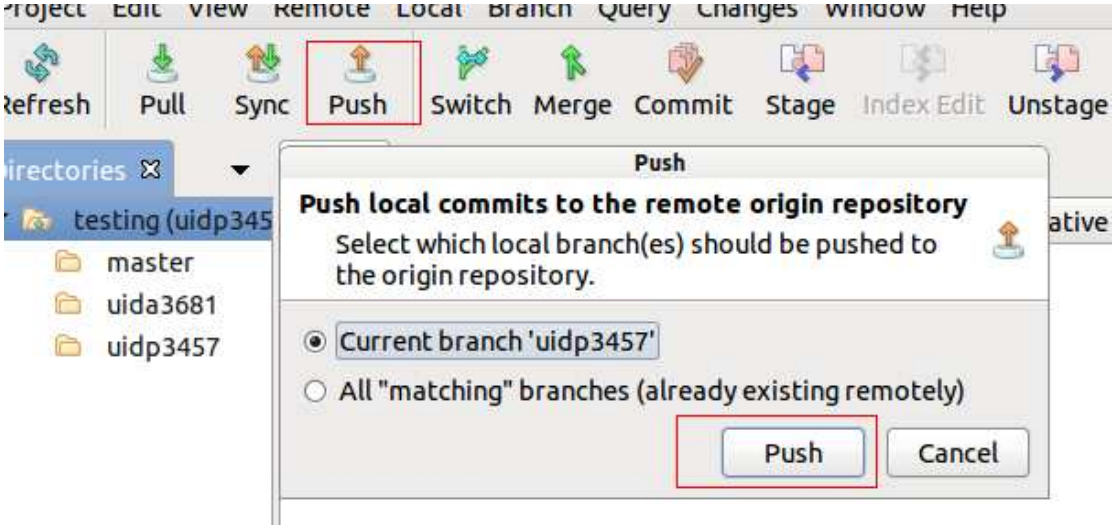


创建分支(git branch uidp3457)

点击 add Branch 创建分支
点击 Add Branch & Switch 创建分支并切换到创建的分支上。

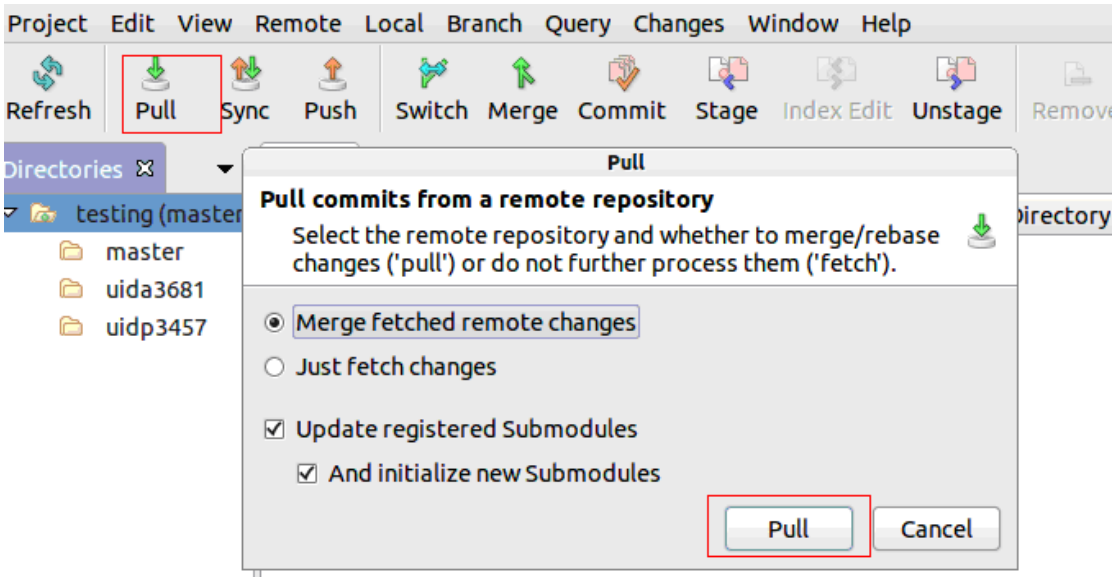


2.4 推送 Push



2.5 拉取数据 git pull：

相当于是从远程获取最新版本并 merge 到本地

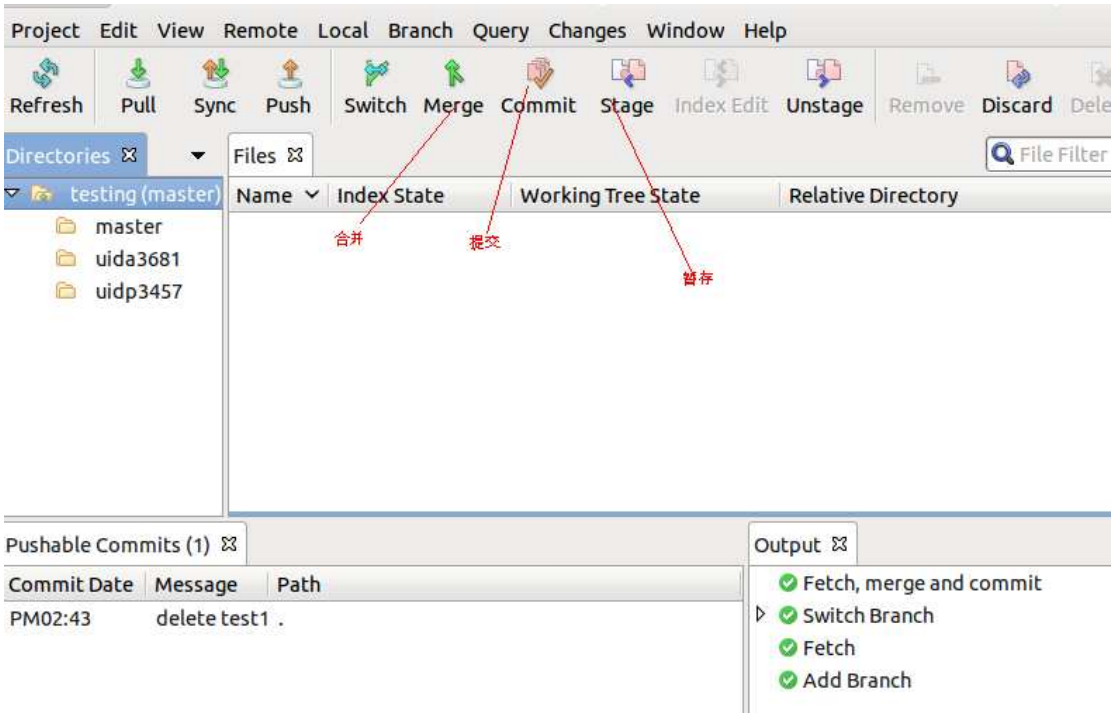


git fetch：相当于是从远程获取最新版本到本地，不会自动 merge

此命令会到远程仓库中拉取所有你本地仓库中还没有的数据。运行完成后，你就可以在本地访问该远程仓库中的所有分支，将其中某个分支合并到本地，或者只是取出某个分支，一探究竟。（我们会在第三章详细讨论关于分支的概念和操作。）

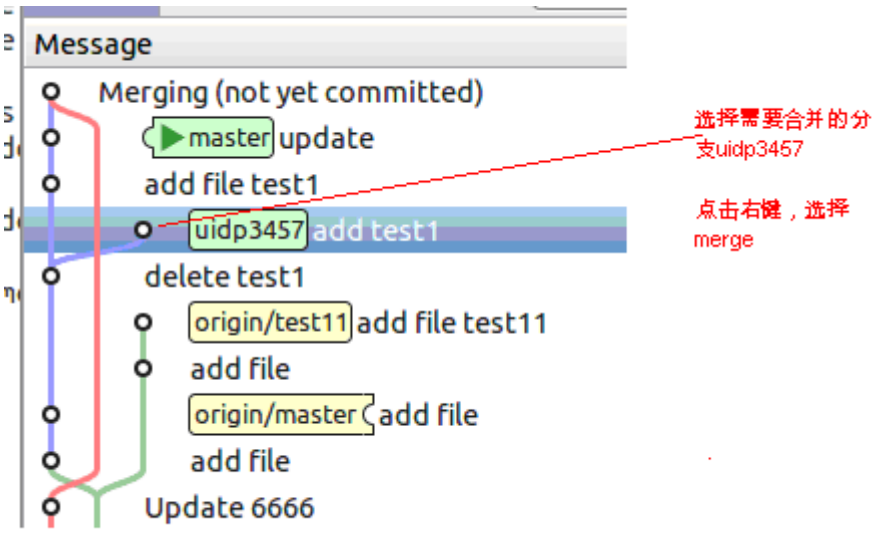
如果是克隆了一个仓库，此命令会自动将远程仓库归于 `origin` 名下。所以，`git fetch origin` 会抓取从你上次克隆以来别人上传到此远程仓库中的所有更新（或是上次 `fetch` 以来别人提交的更新）。有一点很重要，需要记住，`fetch` 命令只是将远端的数据拉到本地仓库，并不自动合并到当前工作分支，只有当你确实准备好了，才能手工合并。

如果设置了某个分支用于跟踪某个远端仓库的分支（参见下节及第三章的内容），可以使用 `git pull` 命令自动抓取数据下来，然后将远端分支自动合并到本地仓库中当前分支。在日常工作中我们经常这么用，既快且好。实际上，默认情况下 `git clone` 命令本质上就是自动创建了本地的 `master` 分支用于跟踪远程仓库中的 `master` 分支（假设远程仓库确实有 `master` 分支）。所以一般我们运行 `git pull`，目的都是要从原始克隆的远端仓库中抓取数据后，合并到工作目录中当前分支。



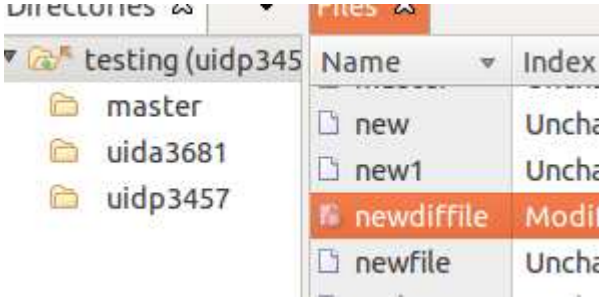
2.6 合并分支 merge

切换到合并分支上：
点击 Query->log

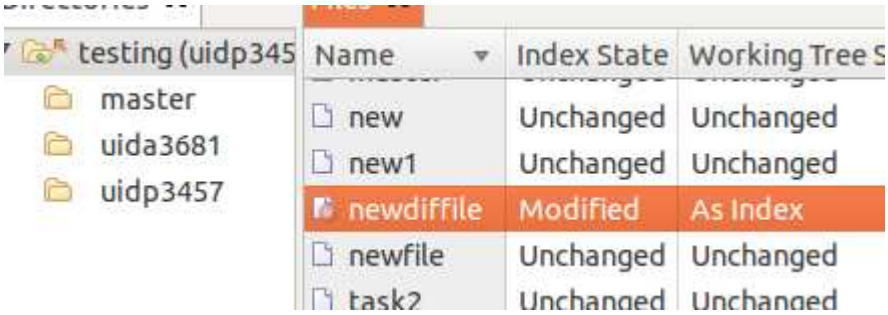


2.6 文件各种状态

2.6.1 Modified in working tree



2.6.2 Modified in Index



2.6.3 local change

	Direct Local Changes	There are local (or Index) changes within the directory itself.
	Indirect Local Changes	There are local (or Index) changes in one of the subdirectories of this directory.

Figure 4.3: Additional Directory States

2.6.4 file status

Icon	State	Details
	Unchanged	File is under version control and neither modified in working tree nor in Index.
	Unversioned	File is not under version control, but only exists in the working tree. Use **Stage** to add the file or **Ignore** to ignore the file.
	Ignored	File is not under version control (exists only in the working tree) and is marked to be ignored.
	Modified	File is modified in the working tree. Use **Stage** to add the changes to the Index or **Commit** the changes immediately.
	Modified (Index)	File is modified and the changes have been staged to the Index. Either **Commit** the changes or **Unstage** changes to the working tree.
	Modified (WT and Index)	File is modified in the working tree and in the Index in different ways. You may **Commit** either Index changes or working tree changes.
	Added	File has been added to Index. Use **Unstage** to remove from the Index.
	Removed	File has been removed from the Index. Use **Unstage** to un-schedule the removal from the Index.
	Missing	File is under version control, but does not exist in the working tree. Use **Stage** or **Remove** to remove from the Index or **Discard** to restore in the wirking tree.
	Modified (Added)	File has been added to the Index and there is an additional change in the working tree. Use **Commit** to either commit just the addition or commit addition and change.
	Intent-to-Add	File is planned to be added to the Index. Use **Add** or **Stage** to add actually or **Discard** to revert to unversioned.
	Conflict	A merge-like command resulted in conflicting changes. Use the **Conflict Solver** to fix the conflicts.

Figure 4.4: File States

3. 本地操作基本命令

3.1 查看状态 git status

git status

```
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
```

3.2 暂存文件 git add <file>

添加一个文件后，使用 git status 查看

```
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       test1
nothing added to commit but untracked files present (use "git add" to track)
```

使用 git add test1 暂存文件

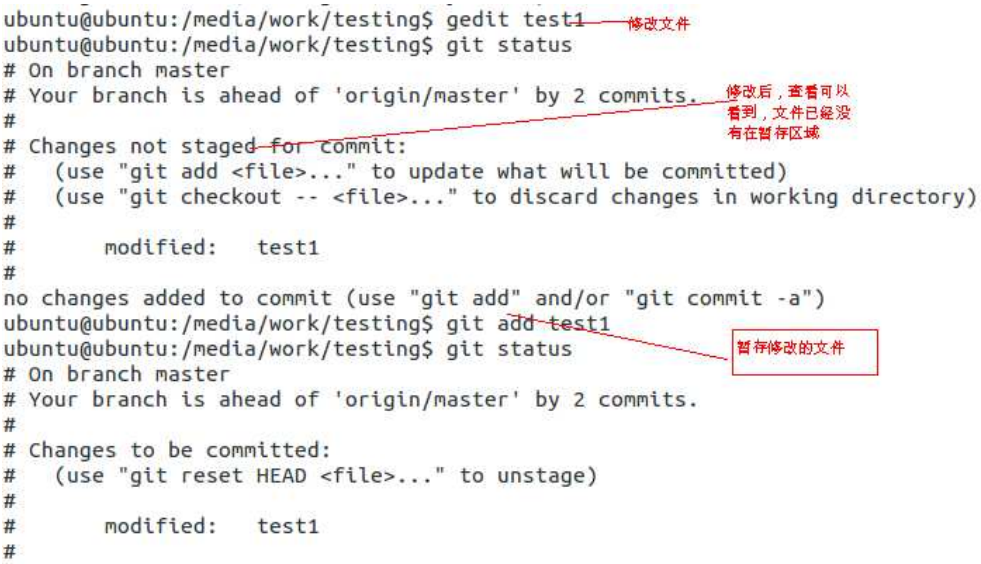
```
ubuntu@ubuntu:/media/work/testing$ git add test1
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   test1
#
```

暂存后，使用 git status 再次查看。可以看到已经添加了一个文件 test1
并可以使用 git commit 命令提交

3.3 提交 git commit –m “提交信息”

```
ubuntu@ubuntu:/media/work/testing$ git commit -m "add file test1"
[master 503a216] add file test1
 1 file changed, 1 insertion(+)
 create mode 100644 test1
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
nothing to commit (working directory clean)
```

3.4 暂存修改文件 git add



3.5 查看已暂存和未暂存的更新 git diff

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 `git diff:`

```
ubuntu@ubuntu:/media/work/testing$ git diff
ubuntu@ubuntu:/media/work/testing$ git diff --cached
diff --git a/test1 b/test1
index a5bce3f..d982d73 100644
--- a/test1
+++ b/test1
@@ -1 +1 @@
-test1
+test1test1
```

看已经暂存起来的文件和上次提交时的快照之间的差异，可以用 `git diff --cached` 命令

3.6 移除文件 git rm

```
ubuntu@ubuntu:/media/work/testing$ git rm test1
error: 'test1' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
```

如果**删除之前修改过并且已经放到暂存区域的**话，则必须
要用强制删除选项-f（译注：即 force 的首字母），以防误删除文件后丢失修改的内容。

```
ubuntu@ubuntu:/media/work/testing$ git rm -f test1
rm 'test1'
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    test1
#
```

从工作目录中手工删除

```
ubuntu@ubuntu:/media/work/testing$ rm new
ubuntu@ubuntu:/media/work/testing$ ls
1234  3333  aaaa    new1   testreset       uida3681  uidp3457-1
2222  a     master  task2  testresetmixed  uidp3457
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    test1
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    new
#
```

3.7 移动文件（重命名）git mv

```
ubuntu@ubuntu:/media/work/testing$ ls
1234  3333  aaaa    new   task2      testresetmixed  uidp3457
2222  a     master  new1  testreset  uida3681        uidp3457-1
ubuntu@ubuntu:/media/work/testing$ git mv task2 task
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    task2 -> task
#       deleted:    test1
#
```

4. 远程仓库的操作

4.1 查看当前的远程库 git remote

-v：显示对应的克隆地址

```
ubuntu@ubuntu:/media/work/testing$ git remote
origin
ubuntu@ubuntu:/media/work/testing$ git remote -v
origin  git@10.219.68.248:testing (fetch)
origin  git@10.219.68.248:testing (push)
```

4.2 从远程仓库中抓取数据 git fetch

```
ubuntu@ubuntu:/media/work/testing$ git fetch origin
```

此命令会到远程仓库中**拉取所有你本地仓库中还没有的数据**。运行完成后，你就可以在本地访问该远程仓库中的所有分支，将其中某个分支合并到本地，或者只是取出某个分支，一探究竟。

如果是克隆了一个仓库，此命令会自动将远程仓库归于 origin 名下。所以，git fetch origin 会抓取从你上次克隆以来别人上传到此远程仓库中的所有更新（或是上次 fetch 以来别人提交的更新）。

有一点很重要，需要记住，**fetch 命令只是将远端的数据拉到本地仓库，并不自动合并到当前工作分支**，只有当你确实准备好了，才能手工合并。

使用**git pull 命令**

自动抓取数据下来，然后将远端分支自动合并到本地仓库中当前分支。在日常工作中我们经常这么用，既快且好。实际上，默认情况下git clone 命令本质上就是自动创建了本地的master 分支用于跟踪远程仓库中的master 分支（假设远程仓库确实有master 分支）。所以一般我们运行git pull，目的都是要从原始克隆的远端仓库中抓取数据后，合并到工作目录中当前分支。

4.3 推送数据到远程仓库 git push

git push origin uidp3457

4.4 查看远程仓库信息 git remote show

```
ubuntu@ubuntu:/media/work/testing$ git remote show origin
* remote origin
  Fetch URL: git@10.219.68.248:testing
  Push  URL: git@10.219.68.248:testing
  HEAD branch: master
  Remote branches:
    master               tracked
    new-uida3681-branch tracked
    task-1-aaaaa         tracked
    task-2-new           tracked
    test11               tracked
    uida3681-branch      tracked
    uidp3457             tracked
    uidp3457-1           tracked
    uidp3472             tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local refs configured for 'git push':
    master   pushes to master   (fast-forwardable)
    uidp3457 pushes to uidp3457 (local out of date)
ubuntu@ubuntu:/media/work/testing$
```

4.5 远程仓库的删除和重命名 git remote rename , git remote rm

git remote rename origin origin-2

git remote rm origin

5. git 分支

5.1 查看分支 git branch

```
ubuntu@ubuntu:/media/work/testing$ git branch
  master
* uidp3457
```

5.2 创建分支 git branch branchname

切换到创建的分支上：
git checkout branchname

新建并切换到该分支
git checkout –b branchname

5.3 分支合并 git merge

将本地分支 iss53 合并到 master 分支：

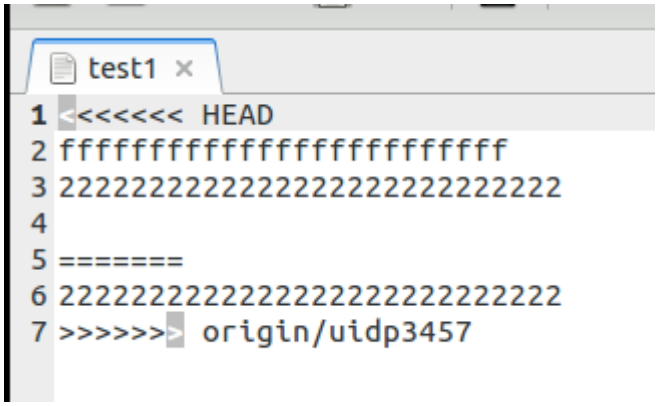
$ git checkout master

$ git merge iss53

5.4 解决冲突

```
ubuntu@ubuntu:/media/work/testing$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 3 commits.
#
# Changes to be committed:
#
#       new file:   5555
#       new file:   6666
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both added:         uidp3457/test1
#
```

可以看到 uidp3457/test1 文件有冲突
打开文件



手动解决冲突：

可以看到======= 隔开的上半部分，是HEAD（即master 分支，在运行merge 命令时检出的分支）中的内容，下半部分是在uidp3457 分支中的内容。

冲突解决后，暂存，提交。

5.5 推送分支 git push origin branchname

要想和其他人分享某个分支，你需要把它推送到一个你拥有写权限的远程仓库。你的本地分支不会被自动同步到你引入的远程分支中，除非你明确执行推送操作。换句话说，对于无意分享的，你尽可以保留为私人分支，而只推送那些协同工作的特性分支。

如果你有个叫serverfix 的分支需要和他人一起开发，可以运行

git push (远程仓库名) (分支名) :

$ git push origin serverfix

5.6 删除远程分支 git push origin :branchname

```
ubuntu@ubuntu:/media/work/testing$ git push origin :uidp3457
To git@10.219.68.248:testing
 - [deleted]         uidp3457
ubuntu@ubuntu:/media/work/testing$ git remote show origin
* remote origin
  Fetch URL: git@10.219.68.248:testing
  Push  URL: git@10.219.68.248:testing
  HEAD branch: master
  Remote branches:
    master               tracked
    new-uida3681-branch tracked
    task-1-aaaaa         tracked
    task-2-new           tracked
    test11               tracked
    uida3681-branch      tracked
    uidp3457-1           tracked
    uidp3472             tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (fast-forwardable)
ubuntu@ubuntu:/media/work/testing$
```

删除本地分支

git branch –d uidp3457-fix

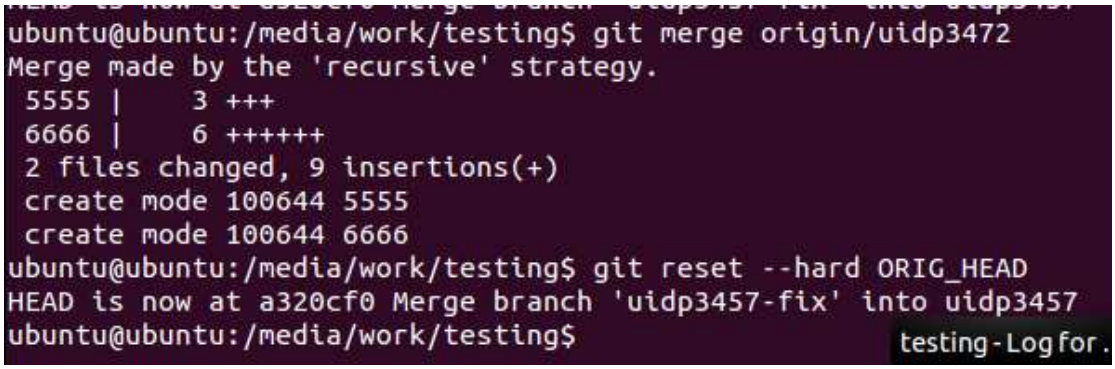
```
ubuntu@ubuntu:/media/work/testing$ git branch -d uidp3457-fix
Deleted branch uidp3457-fix (was c66f2bb).
```

5.7 合并远程分支 git merge origin/test11

```
ubuntu@ubuntu:/media/work/testing$ git merge origin/test11
Merge made by the 'recursive' strategy.
 test/test1  |    1 +
 test/test11 |    1 +
 2 files changed, 2 insertions(+)
 create mode 100644 test/test1
 create mode 100644 test/test11
ubuntu@ubuntu:/media/work/testing$
```

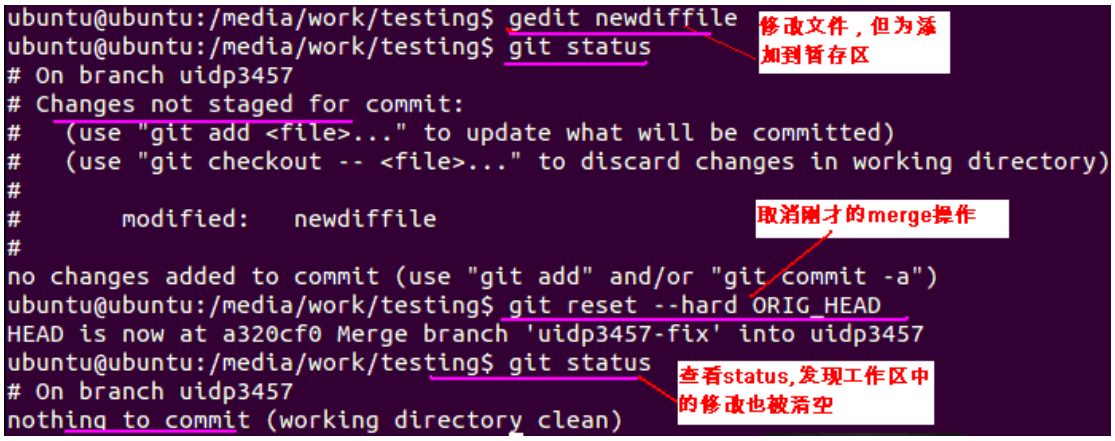
5.8 撤销合并

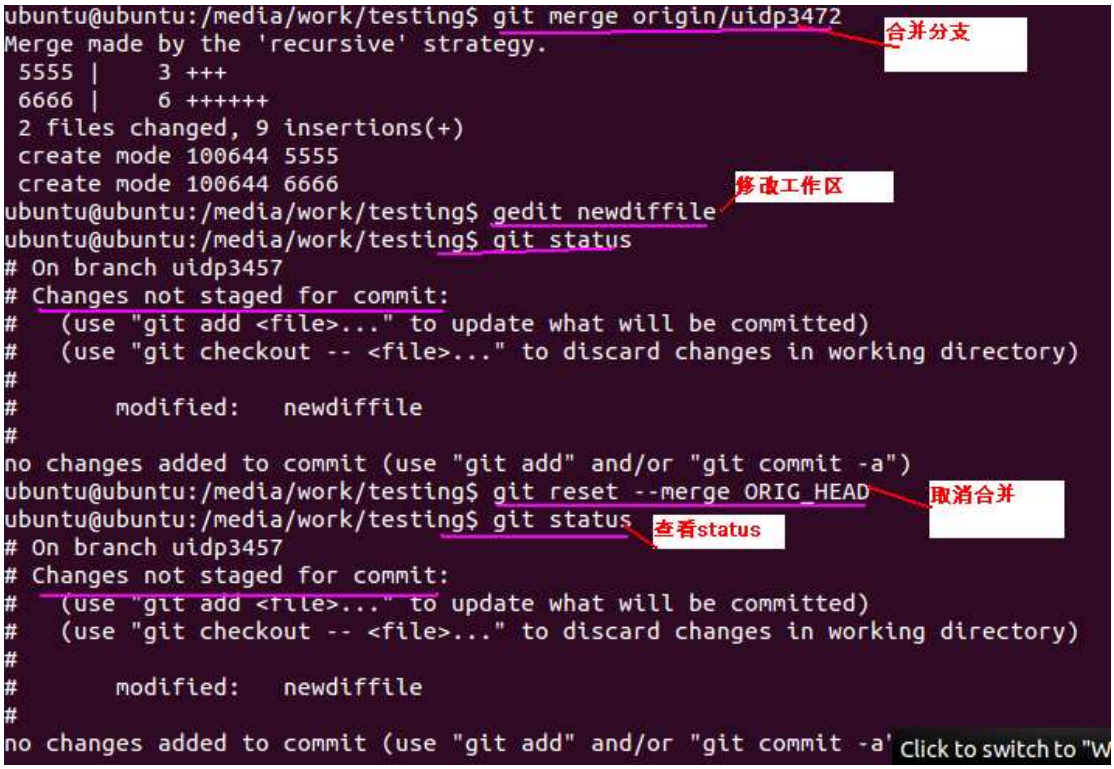
git reset --hard ORIG_HEAD （副作用，清除 workdirectory）



5.9 撤销合并并保存 work directory

git mergr origin/uidp3472





6.git 撤销提交和恢复 git reset ,git revert

git revert 和 git reset 的区别

1. **git revert** 是**用一次新的 commit 来回滚之前的 commit**，
git reset 是**直接删除指定的 commit**。

2. 在回滚这一操作上看，效果差不多。但是在日后继续 merge 以前的老版本时有区别。因为 git revert 是用一次逆向的 commit“中和”之前的提交，因此日后合并老的 branch 时，导致这部分改变不会再次出现，但是 git reset 是之间把某些 commit 在某个 branch 上删除，因而和老的 branch 再次 merge 时，这些被回滚的 commit 应该还会被引入。

3. **git reset 是把 HEAD 向后移动了一下**，

而 **git revert 是 HEAD 继续前进，只是新的 commit 的内容和要 revert 的内容正好相反，能够抵消要被 revert 的内容**。

6.1 撤销 git reset

6.1.1 取消放到暂存区域的文件

git reset HEAD filename

在 git 的一般使用中，如果发现错误的将不想 staging 的文件 add 进入 index 之后，想回退取消，则可以使用命令：git reset HEAD <file>...，同时 git add 完毕之后，git 也会做相应的提示，比如：

引用

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   Test.scala
```

6.1.2 取消提交等

git reset [--hard|soft|mixed|merge|keep] [<commit>或 HEAD]：将当前的分支重设（reset）到指定的<commit>或者 HEAD（默认，如果不显示指定 commit，默认是 HEAD，即最新的一次提交），并且根据[mode]有可能更新 index 和 working directory。mode 的取值可以是 hard、soft、mixed、merged、keep。下面来详细说明每种模式的意义和效果。

A). **--hard：重设（reset） index 和 working directory，自从<commit>以来在 working directory 中的任何改变都被丢弃**，并把 HEAD 指向<commit>。

具体一个例子，假设有三个 commit， git st:

```
commit3: add test3.c
commit2: add test2.c
commit1: add test1.c
```

执行 **git reset --hard HEAD~1** 后，(完全撤销)

显示：HEAD is now at commit2，运行 git log

```
commit2: add test2.c
commit1: add test1.c
```

运行 git st， 没有任何变化

B). **--soft：index 和 working directory 中的内容不作任何改变，仅仅把 HEAD 指向<commit>**。这个模式的效果是，执行完毕后，自从<commit>以来的所有改变都会显示在 git status 的"Changes to be committed"中。

具体一个例子，假设有三个 commit， git st:

```
commit3: add test3.c
commit2: add test2.c
commit1: add test1.c
```

执行 **git reset --soft(默认) HEAD~1** 后， （撤销 commit,保存在 stage 中）

运行 git log

```
commit2: add test2.c
commit1: add test1.c
```

运行 git status， 则 test3.c 处于暂存区，处于准备提交状态。即此时 git commit 就会提交它。

在使用 git 进行协作开发时，我们经常需要将自己的修改生成 patch 发给别人，但是在修改代码的过程中我们进行了很多次的提交，**如何生成从最初的代码状态到最终代码状态的 patch 呢？**下面要介绍的功能是应对这中情况。

现假设我们 git 软件仓库中的分支情况如下：

a-->b-->c

也就是说我们的代码从状态 a 修改到状态 b，进行一次提交，然后再修改到状态 c，进行一次提交。这时我们已经肯定由 a 到 c 的修改是正确的，不再需要状态 b 了，并且要把从 a 到 c 的变化生成一个 patch 发送给别人。如果直接打包的话会生成两个 path，那么如何生成一个 patch 呢，这时就需要 git-reset 命令。

首先给状态 a 创建一个 tag，假设名称为 A，然后执行

git-reset --soft A

这样我们的软件仓库就变为

a

状态 b 和状态 c 都已经被删除了，但是当前的代码并没有被改变，还是状态 c 的代码，这时我们做一次提交，软件仓库变成下面的样子：

a-->d

状态 d 和状态 c 所对应的代码是完全相同的，只是名字不同。现在就可以生成一个 patch 打包发给别人了

C). **--mixed：仅 reset index，但是不 reset working directory**。这个模式是默认模式，即当不显示告知 git reset 模式时，会使用 mixed 模式。这个模式的效果是，working

directory 中文件的修改都会被保留，不会丢弃，但是也不会被标记成"Changes to be committed"，但是会打出什么还未被更新的报告。报告如下：

引用

Unstaged changes after reset:

M Test.Scala

M test.txt

6.1.3 使用实例

6.1.3.1 将 index 恢复到 working directory

A) **回滚 add 操纵**

引用

$ edit (1)

$ git add frotz.c filfre.c

$ mailx （收到邮件通知更新） (2)

$ **git reset** (3) **//将 index 恢复到 working directory 中**

$ git pull git://info.example.com/ nitfol (4)

(1) 编辑文件 frotz.c, filfre.c，做了些更改，并把更改添加到了 index

(2) 查看邮件，发现某人要你 pull，有一些改变需要你 merge 下来

(3) 然而，你已经把 index 搞乱了，因为 index 同 HEAD commit 不匹配了，但是你知道，即将 pull 的东西不会影响已经修改的 frotz.c 和 filfre.c，因此你可以 revert 这两个文件的改变。resetrevert 后，那些改变应该依旧在 working directory 中，因此执行 git reset。

(4) 然后，执行了 pull 之后，自动 merge，frotz.c 和 filfre.c 这些改变依然在 working directory 中。

6.1.3.2 回滚最后一次的提交

B) **回滚最近一次 commit**

引用

$ git commit ...

$ **git reset --soft HEAD^** (1)

$ edit (2)

$ **git commit -a -c ORIG_HEAD** (3) //使用 reset 之前那次 commit 的注释、作者、日期等信息**重新提交**

(1) 当提交了之后，你又发现代码没有提交完整，或者你想重新编辑一下提交的 comment，执行 git reset --soft HEAD^，让 working tree 还跟 reset 之前一样，不作任何改变。HEAD^指向 HEAD 之前最近的一次 commit。

(2) 对 working tree 下的文件做修改

(3) 然后使用 reset 之前那次 commit 的注释、作者、日期等信息重新提交。注意，当执行 git reset 命令时，git 会把老的 HEAD 拷贝到文件.git/ORIG_HEAD 中，在命令中可以使用 ORIG_HEAD 引用这个 commit。commit 命令中 -a 参数的意思是告诉 git，自动把所有修改的和删除的文件都放进 stage area，未被 git 跟踪的新建的文件不受影响。commit 命令中 -c <commit> 或者 -C <commit>意思是拿已经提交的 commit 对象中的信息（作者，提交者，注释，时间戳等）提交，那么这条 commit 命令的意思就非常清晰了，把所有更改的文件加入 stage area，并使用上次的提交信息重新提交。

6.1.3.3 将几次提交放到另一个分支

C) **回滚最近几次 commit，并把这几次 commit 放到叫做 topic 的 branch 上去**。引用

$ git branch topic/wip (1) //在当前 HEAD 创建分支

$ git reset --hard HEAD~3 (2) //删除最近三个 commit

$ git checkout topic/wip (3)//切换到 topic/wip 分支上

(1) 你已经提交了一些 commit，但是此时发现这些 commit 还不够成熟，不能进入 master 分支，但你希望在新的 branch 上润色这些 commit 改动。因此执行了 git branch 命令在当前的 HEAD 上建立了新的叫做 topic/wip 的分支。

(2) 然后回滚 master branch 上的最近三次提交。HEAD~3 指向当前 HEAD-3 个 commit 的 commit，git reset --hard HEAD~3 即删除最近的三个 commit（删除 HEAD, HEAD^, HEAD~2），将 HEAD 指向 HEAD~3。

删除分支 topic/wip git branch –d topic/wip

6.1.3.4 删除最后几个 commit

D) **永久删除最后几个 commit**

引用

$ git commit ...

$ git reset --hard HEAD~3 (1)

(1) 最后三个 commit（即 HEAD, HEAD^和 HEAD~2）提交有问题，你想永久删除这三个 commit。

6.1.3.5 撤销 merge

E) **回滚 merge 和 pull 操作**

引用

$ git pull (1)

Auto-merging nitfol

CONFLICT (content): Merge conflict in nitfol

Automatic merge failed; fix conflicts and then commit the result.

$ **git reset --hard** (2) //清除 index 和 working tree 中被搞乱的东西。

$ git pull . topic/branch (3) // (合并 topic/branch 分支)

Updating from 41223... to 13134...

Fast-forward

$ **git reset --hard ORIG_HEAD** (4) //回滚 merge 操作

(1) 从 origin 拉下来一些更新，但是产生了很多冲突，你暂时没有这么多时间去解决这些冲突，因此你决定稍候有空的时候再重新 pull。

(2) 由于 pull 操作产生了冲突，因此所有 pull 下来的改变尚未提交，仍然再 stage area 中，这种情况下 git reset --hard 与 git reset --hard HEAD 意思相同，即都是清除 index 和 working tree 中被搞乱的东西。

(3) 将 topic/branch 合并到当前的 branch，这次没有产生冲突，并且合并后的更改自动提交。

(4) 但是此时你又发现将 topic/branch 合并过来为时尚早，因此决定**退滚 merge，执行 git reset --hard ORIG_HEAD 回滚刚才的 pull/merge 操作**。说明：前面讲过，执行 git reset 时，git 会把 reset 之前的 HEAD 放入.git/ORIG_HEAD 文件中，命令行中使用 ORIG_HEAD 引用这个 commit。同样的，执行 pull 和 merge 操作时，git 都会把执行操作前的 HEAD 放入 ORIG_HEAD 中，以防回滚操作。

6.1.3.6 撤销 merge ,并保存 working tree

F) **在被污染的 working tree 中回滚 merge 或者 pull**

引用

$ git pull (1)

Auto-merging nitfol

Merge made by recursive.

nitfol | 20 +++++----

...

$ **git reset --merge ORIG_HEAD** (2)// 避免在回滚时清除 working tree

(1) 即便你已经**在本地更改了一些你的 working tree，你也可安全的 git pull**，前提是你知道将要 pull 的内容不会覆盖你的 working tree 中的内容。

(2) git pull 完后，你发现这次 pull 下来的修改不满意，想要回滚到 pull 之前的状态，从前面的介绍知道，我们可以执行 **git reset --hard ORIG_HEAD，但是这个命令有个副作用就是清空你的 working tree，即丢弃你的本地未 add 的那些改变**。为了避免丢弃 working tree 中的内容，可以使用 **git reset --merge ORIG_HEAD**，注意其中的--hard 换成了 --merge，这样就可以**避免在回滚时清除 working tree**。

6.1.3.7 被中断工作流

G) 被中断的工作流程

在实际开发中经常出现这样的情形：你正在开发一个大的 feature，此时来了一个紧急的 bug 需要修复，但是目前在 working tree 中的内容还没有成型，还不足以 commit，但是你又必须切换的另外的 branch 去 fix bug。请看下面的例子

引用

$ git checkout feature ;# you were working in "feature" branch and

$ work work work ;# got interrupted

$ git commit -a -m "snapshot WIP" (1) //临时提交

$ git checkout master

$ fix fix fix

$ git commit ;# commit with real log

$ git checkout feature

$ git reset --soft HEAD^ ;# go back to WIP state (2) //撤销临时提交

$ git reset (3)//清理 index

(1) 这次属于临时提交，因此随便添加一个临时注释即可。

(2) 这次 reset 删除了 WIP commit，并且把 working tree 设置成提交 WIP 快照之前的状态。

(3) 此时，在 index 中依然遗留着“snapshot WIP”提交时所做的 uncommit changes，**git reset 将会清理 index 成为尚未提交"snapshot WIP"时的状态便于接下来继续工作**。

(l) **保留 working tree 并丢弃一些之前的 commit**

假设你正在编辑一些文件，并且已经提交，接着继续工作，但是现在你发现当前在 working tree 中的内容应该属于另一个 branch，与这之前的 commit 没有什么关系。此时，你可以开启一个新的 branch，并且保留着 working tree 中的内容。

引用

```
$ git tag start
$ git checkout -b branch1
$ edit
$ git commit ...                    (1)
$ edit
$ git checkout -b branch2           (2)
$ git reset --keep start            (3)// 把在 start 之后的 commit 清除掉，但是保持 working tree 不变
```

(1) 这次是把在 branch1 中的改变提交了。

(2) 此时发现，之前的提交不属于这个 branch，此时你新建了 branch2，并切换到了 branch2 上。

(3) **此时你可以用 reset --keep 把在 start 之后的 commit 清除掉，但是保持 working tree 不变。**

6.2 恢复撤销

```
$ git reset --hard HEAD^
HEAD is now at 1a75c1d... added file1

$ cat file2
cat: file2: No such file or directory
```

$ git reflog

```
1a75c1d... HEAD@{0}: reset --hard HEAD^: updating HEAD
f6e5064... HEAD@{1}: commit: added file2
```

$ git reset --hard f6e5064 （恢复撤销的那次提交）

```
HEAD is now at f6e5064... added file2
```

6.3 git revert 和 reset 的区别

git revert 和 reset 的区别
这里讲一下 git revert 和 git reset 的区别：

git revert 是撤销某次操作，此次操作之前的 commit 都会被保留

git reset 是撤销某次提交，但是此次之后的修改都会被退回到暂存区

具体一个例子，假设有三个 commit， git st:
commit3: add test3.c
commit2: add test2.c
commit1: add test1.c
当执行 git revert HEAD~1 时， commit2 被撤销了
git log 可以看到：
commit1：add test1.c
commit3：add test3.c
git st 没有任何变化

如果换做**执行 git reset --soft(默认) HEAD~1 后**，运行 git log
commit2: add test2.c
commit1: add test1.c
运行 git st， 则 **test3.c 处于暂存区，准备提交**。

如果换做执行 **git reset --hard HEAD~1 后，**
显示：HEAD is now at commit2，运行 git log
commit2: add test2.c
commit1: add test1.c
运行 git st， 没有任何变化

7. index 和 working directory

1. Working Directory（工作目录）

Git 的工作目录是保存当前正在工作的文件所在的目录，和 working tree 是相同的意思。在这个目录中的文件可能会在切换 branch 时被 GIT 删除或者替换。这个目录是个临时目录，临时存储你从 GIT 库中取出的文件，这些文件一直会被保存，直到下次提交。

2. GIT Directory（GIT 库目录）

项目的**所有历史提交都被保存在了 GIT 库目录中**，只要你不作回滚操作，它应该不会丢失。

3. GIT Index（Git 索引）

Git index 可以看作是工作目录和 Git 库目录之间的**暂存区**，和 staging area 是相同的意思。可以使用 Git index 构建一组你准备一起提交的改变。Git Index 和 Git Staging area 是同一个意思，**都是指已经被 add 的但尚未 commit 的那些内容所在的区域**。最简单的查看目前什么内容在 index 中的方法是使用 **git status** 命令。

- 命令中”Changes to be committed“中所列的内容是在 Index 中的内容，commit 之后进入 Git Directory。
- 命令中“Changed but not updated”中所列的内容是在 Working Directory 中的内容，add 之后将进入 Index。
- 命令中“Untracked files”中所列的内容是尚未被 Git 跟踪的内容，add 之后进入 Index。

哪些操作能够改变 git index 中的内容？

A). **git add <path>...**会将 working directory 中的内容添加进入 git index。

B). **git reset HEAD <path>...**会将 git index 中 path 内容删除，重新放回 working directory 中

7.1 git diff

git diff 可以**比较 working tree 同 index 之间，index 和 git directory 之间，working tree 和 git directory 之间，git directory 中不同 commit 之间的差异，**

- **git diff [<path>...]**：这个命令最常用，在每次 add 进入 index 前会运行这个命令，查看即将 add 进入 index 时所做的内容修改，即 **working directory 和 index 的差异**。
-
- **git diff --cached [<path>...]**：这个命令初学者不太常用，却非常有用，它表示查看已经 add 进入 index 但是尚未 commit 的内容同最后一次 commit 时的内容的差异。即 **index 和 git directory 的差异**。

```
ubuntu@ubuntu:/media/work/testing$ git diff --cached newdiffile
diff --git a/newdiffile b/newdiffile
index 12db1ad..866716d 100644
--- a/newdiffile
+++ b/newdiffile
@@ -1,3 +1,5 @@
-fffffffffffffffffffff
+fffffffffffffffffffff444444%%%%%%%
 jjjjjjjjjjjjjjjjjjjjjjjKKKKKKKKKKKKKKKKK
 hhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
+44444444444
+
```

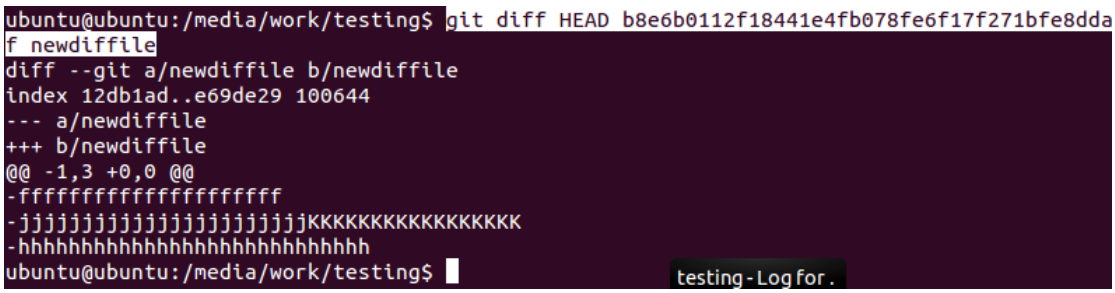
- git diff --cached [<commit>] [<path>...]：这个命令初学者用的更少，也非常有用，它表示查看已经 add 进入 index 但是尚未 commit 的内容同指定的<commit>之间的差异，和上面一条很相似，差别仅仅<commit>，即 index 和 git directory 中指定版本的差异。

```
ubuntu@ubuntu:/media/work/testing$ git diff --cached b8e6b0112f18441e4fb078fe6f17f271bfe8ddaf newdiffile
diff --git a/newdiffile b/newdiffile
index e69de29..866716d 100644
--- a/newdiffile
+++ b/newdiffile
@@ -0,0 +1,5 @@
+fffffffffffffffffffff444444%%%%%%%
+jjjjjjjjjjjjjjjjjjjjjjjKKKKKKKKKKKKKKKKK
+hhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
+44444444444
+
```

- git diff <commit> [<path>...]：这个命令用来查看工作目录和指定<commit>的 commit 之间的差别，如果要和 Git directory 中最新版比较差别，则<commit>=HEAD。如果要和某一个 branch 比较差别，<commit>=分支名字

```
ubuntu@ubuntu:/media/work/testing$ git diff HEAD
diff --git a/newdiffile b/newdiffile
index 12db1ad..b120944 100644
--- a/newdiffile
+++ b/newdiffile
@@ -1,3 +1,5 @@
-fffffffffffffffffffff
+fffffffffffffffffffff444444%%%%%%%
 jjjjjjjjjjjjjjjjjjjjjjjKKKKKKKKKKKKKKKKK
 hhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
+44444444444git work compare commit
```

- git diff <commit> <commit> [<path>...]：这个命令用来比较 git directory 中任意两个<commit>之间的差别，如果想比较任意一个<commit>和最新版的差别，把其中一个<commit>换成 HEAD 即可。


ubuntu@ubuntu:/media/work/testing$ git diff HEAD b8e6b0112f18441e4fb078fe6f17f271bfe8dda
f newdiffile
diff --git a/newdiffile b/newdiffile
index 12db1ad..e69de29 100644
--- a/newdiffile
+++ b/newdiffile
@@ -1,3 +0,0 @@
-fffffffffffffffffff
-jjjjjjjjjjjjjjjjjjjjjKKKKKKKKKKKKKKKKK
-hhhhhhhhhhhhhhhhhhhhhhhhhhhh
ubuntu@ubuntu:/media/work/testing$
testing-Log for .