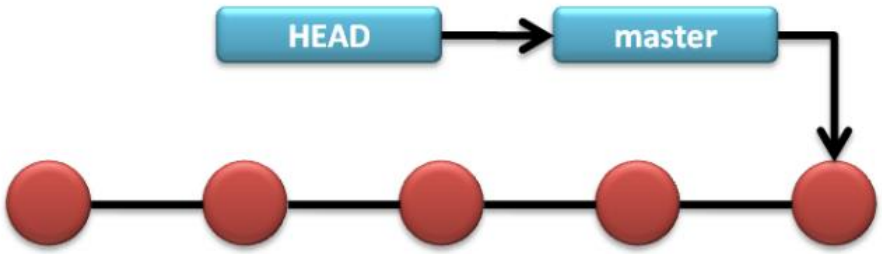
git 详解分支管理三

1.创建与合并分支

利用分支就可以实现多人开发的伟大模式，从而提高生产效率。在整个 GIT 之中，主分支(master)主要是作为程序 的发布使用，一般而言很少会在主分支上进行代码的开发，都会在各自的子分支上进行。

1）mastr 分支

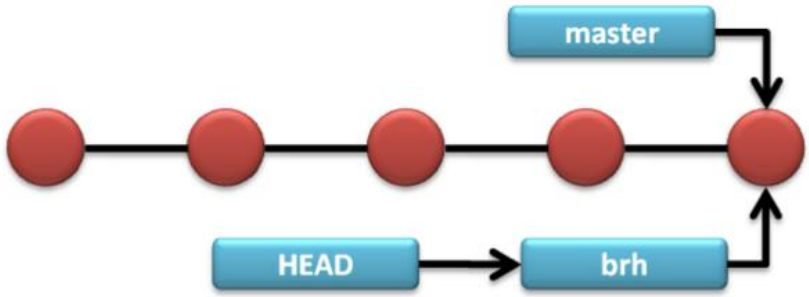
默认情况下，mastr 是一条线，git 利用 master 指向最新的提交，再用 "HEAD" 批向 "master",就能确定当前分支以及当前分支的提交点。



以上操作属于项目发布版本的执行顺序，因为最终发布就是 master 分支。但是对于其它的开发者，不应该应该在 mastr 分支上进行。所以应该建立分支，而子分支最起码建立的时候应该是当前的 master 分支的状态。而分支的一但创建之后， HEAD 指针就会发生变化。

2）分支提交

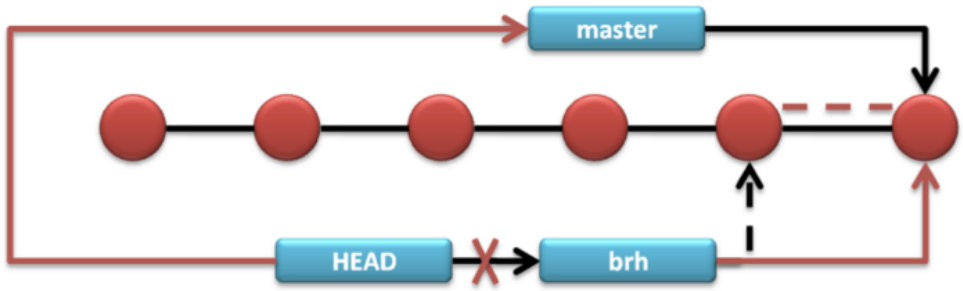
如果有新的提交，则 master 分支不会改变，只有 brh 分支会发生变化。



那么此时 master 分支的版本号就落后于子分支了。但是不管子分支再怎么开发，也不是最新发布版本，所有的发布版本都保存在 master 分支上，那么就必须将分支与 master 的分支进行合并。

3）分支提交

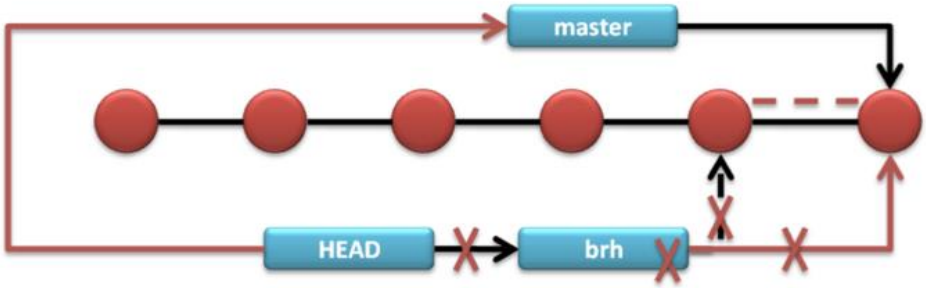
如果有新的提交，刚 master 分支不会改变，只有 bth 分支会发生改变。



当分支合并之后，实际上就相当于 master 的分支的提交点修改为子分支的提交点，而后这个合并应该在 master 分支上完成，而后 HEAD 需要修改指针，断开 brh 分支，而指向原本的 master 分支。

4）删除子分支

如果有新的提交，刚 master 分支不会改变，只有 brh 分支会发生改变。



分支删除掉之后所有的内容也就都取消了。

5）创建一个分支

```
git branch brh
```

6）当分支创建完成之后可以通过如下命令进行查看

```
git branch
[root@git ~]# git branch
  brh
* master
```

可以发现现在提示当前工作区中有两个分支:一个是 brh 分支，另外一个是

master 分支，而现在的分支指向的 是 master 分支。

7）切换到 brh 分支

```
git checkout brh
[root@git ~]# git checkout brh
 * brh
   master
```

但是很多时候我们创建分支的最终目的就是为了切换到此分支上进行开发，所以为

了方便操作，在 git 之中提供了一 个更加简单的功能。

创建并切换分支

如果想要删除子分支，那么不能在当前分支上，所以切换回了 master 分支

```
git checkout master
```

删除子分支

```
git branch -d brh
```

建立分支的同时可以自动的切换到子分支

```
git checkout -b brh
```

8）切换到 brh 分支

现在已经成功的在 brh 分支上了，那么下面进行代码的修改;

修改 hello.js

```
btn.onclick = function() {
   console.log('git 分支管理练习！');
}
```

这个时候的 hello.js 文件是属于子分支上的，而现在也在子分支上，那么下面查

询一下子分支的状态。

```
# On branch brh
# 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)
#   (commit or discard the untracked or modified content in
submodules)
#
#       modified:   hello.js
#
```

no changes added to commit (use "git add" and/or "git commit -a")

此时更新的是子分支的内容，但是主分支上的数据呢?

9）在子分支上将修改进行提交

```
git commit -a -m "modified hello.js file"
```

当子分支的数据提交之后实际上并不会去修改 master 分支的内容。这就证明

了，两个分支上的内容是彼此独立的。

10）既然分支都已经存在了，那么现在为了更加清楚，将 master 和 brh 两个分支都提交到远程服务器上(GITHUB)

```
git remote set-url origin https://github.com/yootk/mldn.git
git push origin master
git push origin brh
```

11）最终发布的版本一定是在 master 分支上，所以下面需要将 brh 分支与 master 分支进行合并(在主分支上)

```
git merge brh
[root@git ~]# git merge brh
Updating a26c954..682212c
Fast-forward
 Hello.js | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

在之前讲解的时候说过实际上是修改了 master 指针为 brh 分支的指针信息。所以此时的合并方式为“Fast-forward”,表示是快速合并方式，快速的合并方式并不会产生任何的 commit id。 它只是利用了合并子分支的 commit id 继续操作。

12）此时的 brh 分支没有任何的用处了，那么就可以执行删除操作

```
git branch -d brh
```

13）提交 master 分支

```
git push origin master
```

现在在本地上已经没有了子分支，但是在远程服务器上依然会存在子分支。那么下面要删除远程分支。

14）删除远程分支

```
git push origin --delete brh
```

那么此时远程分支就已经被成功的删除掉了。

2.分支的操作管理

上面演示了分支的各个操作，包括使用分支、以及合并分支，同时也清楚了对于分支有两种方式一种 是本地分支，另外一种是远程分支，但是对于分支在 GIT 使用之中依然会有一些小小的问题，所以下面进行集中式的说明：

1）为了方便还是建立一个新的分支 —— brh

```
git checkout -b brh
```

2）在此分支上建立一个 helloword.js 文件

```
public class HelloWorld() {
  console.log('Hello World');
}
    git add .
    git commit -a -m "Add Emp.java File"
```

以上的代码是在子分支(brh)上建立的。

3）此时并没有进行分支数据的提交，但是有人觉得这个 brh 分支名称不好，应该使用自己的姓名简写完成“wzy”

```
git branch -m brh wzy
```

现在相当于分支名称进行了重新的命名。

4）将分支推送到远程服务器端

```
git push origin wzy
```

5）在本地查看远程的分支

```
// 查看全部的分支，包括远程和本地的分支
git branch -a
// 只查看远程的分支
git branch -r
// 只查看本地分支
git branch -l
```

6）此时“wzt”分支上已经做出了修改，但是并没有与 master 分支进行合并，因为现在所开发的功能开发到一半发现不再需要了，所以就要废除掉所作出的修改。于是发出了删除 wzy 分支的命令

```
git branch -d wzy
[root@git ~]# git branch -d wzy
error: The branch 'wzy' is not fully merged.
If you are sure you want to delete it, run 'git branch -D wzy'.
```

此时直接提示，分支并不能够被删除掉，因为这个分支所做出的修改还没有进行合并。如果要想强制删除此分支， 则可以使用“-D”的参数完成。

```
[root@git ~]# git branch -D wzy
Deleted branch wzy (was 2c60599).
```

可是现在在远程服务器上依然会存在此分支，那么就必须也一起删除掉，但是对于删除操作，除了之前使用过的方式之外，也可以推送一个空的分支，这样也表示删除。

删除方式一

```
git push origin --delete wzy
```

删除方式二

```
git push origin :wzy
```

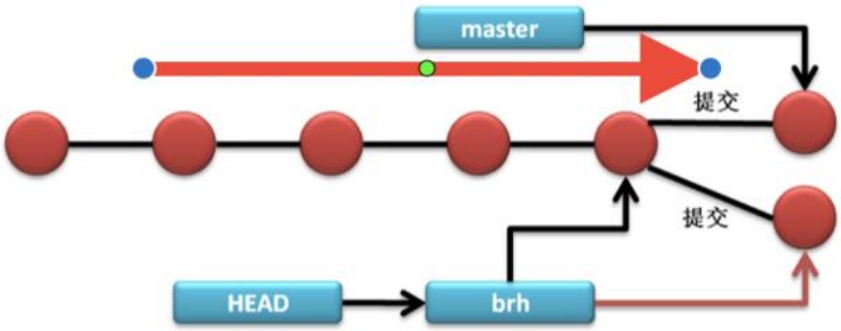
3.冲突解决

分支可以很好的实现多人开发的互操作，但是有可能出现这样种情况:

- 现在建立了一个新的分支 brh，并且有一位开发者在此分支上修改了 hello.js 文件。
- 但是这个开发者由于不小心的失误，又将分支切换回了 master 分支上，并且在 master 分支上也对 hello.js 文件进行修改。

等于现在有两个分支对同一个文件进行了修改，那么在进行提交的时候一定会出现一个冲突。因为系统不知道到底 提交那一个分支的文件。

master 和 brh 两个分支上都有各自的信息提交，那么此时就形成了冲突：



那么很明显，此时有两个提交点，那么会出现怎样的冲突警告呢?为了更好的说明问题，下面通过代码进行验证:

1）建立并切换到 brh 分支上

```
git checkout -b brh
```

2）在此分支上修改 hello.js 文件

```
btn.onclick = function() {
   console.log('git 分支管理练习！');
   console.log('git 分支冲突练习！')
}
```

3）在 brh 分支上提交此文件

```
git commit -a -m "add static attribute"
```

4）切换回 master 分支

```
git checkout master
```

5）在 master 分支上也修改 Hello.js 文件

```
btn.onclick = function() {
   console.log('git 分支管理练习！');
   console.log('git Mast 分支修改测试！ ')
}
```

6）在 master 分支上进行修改的提交

```
git commit -a -m "add master change file"
```

现在在两个分支上都存在了代码的修改，而且很明显，修改的是同一个文件，那么自然进行分支合并的时候是无法 合并的。

7）合并分支(此时已经存在于 master 分支上)

```
git merge brh
[root@git ~]# git merge brh
Auto-merging hello.js
CONFLICT (content): Merge conflict in hello.js
Automatic merge failed; fix conflicts and then commit the result.
```

此时会直接提示出现了冲突。

8）查看冲突的内容

```
wzydeMacBook-Pro:mypro wzy$ git status
On branch brh
You are currently rebasing branch 'brh' on 'ac4ef48'.
  (all conflicts fixed: run "git rebase --continue")

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:   hello.js

no changes added to commit (use "git add" and/or "git commit -a")
wzydeMacBook-Pro:mypro wzy$ git commit -a -m "add static attribute"
[brh f99b037] add static attribute
```

直接提示用户，两次修改了 Hello.java 文件。

9） 查看 hello.js 文件

```
btn.onclick = function() {
   console.log('git 分支管理练习！');
<<<<<<< HEAD
   console.log('git Mast 分支修改测试！  ')
=======
   console.log('git 分支冲突练习！')
>>>>>>> brh
}
```

它现在把冲突的代码进行了标记，那么现在就必须人为手工修改发生冲突的文件。

10）手工修改 hello.js 文件

```
btn.onclick = function() {
   console.log('git 分支管理练习！');
   console.log('git Mast 分支修改测试！  ')
   console.log('git 分支冲突练习！')
}
```

现在是希望这几个输出的内容都同时进行保留。

11）此时已经手工解决了冲突，而后继续进行提交

```
git commit -a -m "conflict print"
```

那么现在的冲突问题就解决了。

12）向服务器端提交信息

```
git push origin master
```

那么在实际的开发之中，一定会存在有许多的分支合并的情况，那么我怎么知道分支合并的历史呢?

13） 查看合并的情况

```
git log --graph --pretty=oneline
[root@git ~]# git log --graph --pretty=oneline
*   b5ec83968fa1f87195b6784cd0c1924a5c52ce2f conflict print
|\
| * 31f497cffe0a9c223da520b4bebfa6e25ec950f5 add static attribute
* | c7f81833a415b5cc047414a5328a36dd7806d787 add master change file
|/
* 87b22e585ddb7e99c4b397f59c8ad79772843f1c modified hello.js file
```

“-graph”指的是采用绘图的方式进行现实。

14） 删除掉 brh 分支

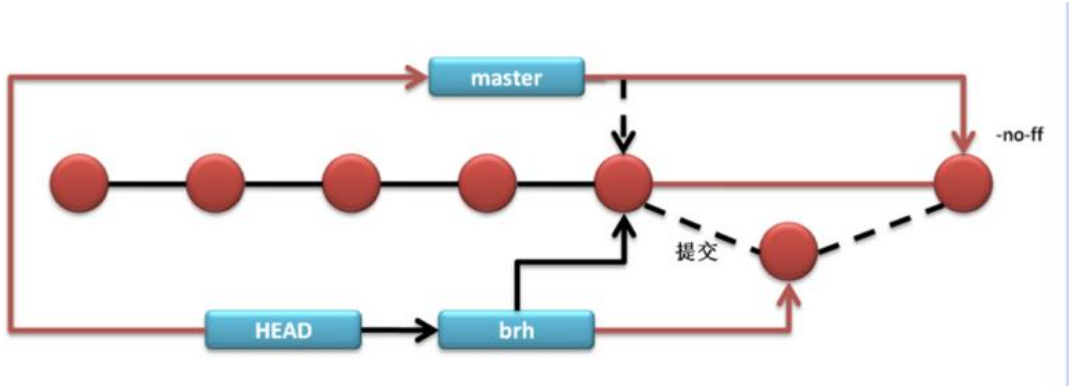
```
git branch -d brh
```

那么此时的代码就可以回归正常的开发模式。

4.分支管理策略

在之前进行分支合并的时候使用的全部都是“Fast forward”方式完成的，而此种方式只是改变了 master 指针，可是 在分支的时候也可以不使用这种快合并，即:增加上一个“--no-ff”参数，这样就表示在合并之后会自动的再生成一个新 的 commit id，从而保证合并数据的完整性。

"-no-ff": 合并后动创建一个新的 commit



1）创建一个新的分支

```
git checkout -b brh
```

2）建立一个新的 empty.js 文件

```
public class Empty() {
  console.log('empty file');
}
```

3） 提交修改

```
git add.
git commit -m "add empty.js file"
```

4） 切换回 master 分支

```
git checkout master
```

5） 使用非快速合并的方式进行代码合并

```
git merge --no-ff -m "no ff commit" brh
```

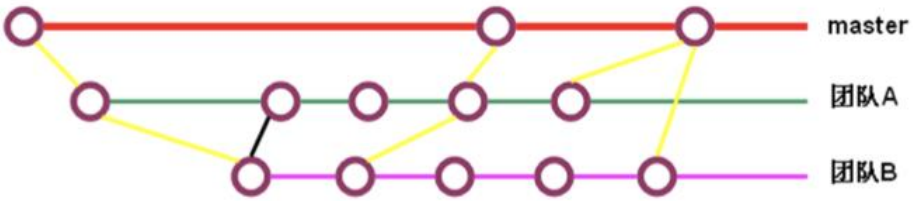
“--no-ff”方式会带有一个新的提交，所以需要为提交设置一个提交的注释。

6） 查看一下提交的日志信息

```
 git log --graph --pretty=oneline --abbrev-commit
[root@git ~]#  git log --graph --pretty=oneline --abbrev-commit
*   8ffe1b9 no ff commit
|\
| * 9305f6e add empty.js file
|/
*   b5ec839 conflict print
|\
| * 31f497c add static attribute
| * c54d3a0 modified hello.js file
* | c7f8183 add master change file
|/
```

分支策略

- master 分支应该是非常稳定的，也就是仅用来发布新的版本，不要在此分支上开发；
- 在各个子分支上进行开发工作；
- 团队中的每个成员都在各个分支上工作；



5.分支暂存

譬如说同在你正在一个分支上进行代码的开发，但是突然你的领导给了你一个新的任务，并且告诉你在半个小时内 完成，那么怎么办?

难道那开发一半的分支要提交吗?不可能的，因为对于版本控制的基本的道德方式:

你不能把有问题的代码提交上 去，你所提交的代码一定都是正确的代码，那么为了这样的问题，在 GIT 中提供了一个分支暂存的机制，可以将开发一半 的分支进行保存，而后在适当的时候进行代码的恢复。

那么下面首先创建一个基本的开发场景。

1) 创建并切换到一个新的分支

```
git checkout -b brh
```

2) 下面在分支上编写 empty.js 类的文件

```
public class Empty() {
  console.log('empty file');
  console.log('我正在开发一半中。。。。。。')
}
```

3) 将此文件保存在暂存区之中

```
git add .
```

这个时候由于代码还没有开发完成，所以不能够进行代码的提交。但是你的老板给了你一个新的任务，那么你就不得不去停止当前的开发任务，所以就需要将当前的开发进度进行“暂存” ，等日后有时间了继续进行恢复开发。

4) 将工作暂存

```
git stash
```

```
[root@git ~]# git stash
Saved working directory and index state WIP on brh: 9305f6e add
empty.js file
HEAD is now at 9305f6e add empty.js file
```

5）查看一下当前的工作区中的内容

```
[root@git ~]# git status
# On branch brh
# 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)
#   (commit or discard the untracked or modified content in
submodules)
#
#       modified:   work (modified content, untracked content)
#
no changes added to commit (use "git add" and/or "git commit -a")
```

此处会直接告诉用户当前的工作区之中没有任何的修改。

6）查看一下当前的工作区中的内容

而后现在假设要修改的代码还处于 master 分支上，所以下面切换到 master 分支。

那么现在假设说创建一个新的分支，用于完成老板的需求，假设分支的名称为“dev”(也有可能是一个 bug 调试)。

7）创建并切换分支

```
git checkout -b dev
```

8） 在新的分支中修改 hello.js 文件

```
btn.onclick = function() {
   console.log('git 分支管理练习！');
   console.log('git Mast 分支修改测试！  ')
   console.log('git 分支冲突练习！')
   console.log('临时任务 dev 上的修改')
}
```

9） 提交修改的操作

```
git commit -a -m "dev change"
```

10） 提交修改的操作

合并 deve 分支，使用 no fast forward

```
git merge --no-ff-m "merge dev branch" dev
```

11） 那么现在突发的问题已经被解决了，被解决之后对于 dev 的分支将没有任何的存在意义，可以直接删除;

```
git branch -d dev
```

12） 那么需要回归到已有的工作状态，但是有可能会存在有许多的暂存的状态，可以直接使用如下命令进行列出。

```
git stash list
```

```
wzydeMacBook-Pro:mypro wzy$ git stash list
stash@{0}: WIP on brh: dfab258 no ff commit brh
```

13）从暂存区之中进行恢复

暂存区恢复之后那么所暂停的操作将没有存在的意义，但是也有人会认为它有意义，所以对于恢复有两种形式:

形式一:先恢复，而后再手工删除暂存

```
git stash apply
git stash drop
```

形式二:恢复的同时也将 stash 内容删除

```
git stash pop
```

```
wzydeMacBook-Pro:mypro wzy$ git stash pop
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

You are currently rebasing branch 'brh' on 'ac4ef48'.
  (all conflicts fixed: run "git rebase --continue")

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:   empth.js

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (a299997ecbea5bbb61979a784808bf6c64f02aa2)
```

那么下面的任务就可以像之前那样进行代码的提交，而后删除掉 brh 分支:

```
git commit -a -m "change empty.js"
git branch -d brh
```

使用暂存策略可以很方便的解决代码突然暂停修改的操作，是非常方便。

6.补丁: patch

补丁并不是针对于所有代码的修改，只是针对于局部的修改。在很多的代码维护之中，如果按照最早克隆的方式将 代码整体克隆下来实际上所花费的资源是非常庞大的，但是修改的时候可能只修改很小的一部分代码，所以在这种情况下 就希望可以将一些代码的补丁信息发送给开发者。而发给开发者之后他需要知道那些代码被修改了，这样的话就可以使用 一个极低的开销实现代码的修改操作，而在 GIT 之中也提供了两种简单的补丁方案:

- 使用 git diff 生成标准的 patch
- 使用 git format-patch 声称 git 专用的 patch

1） 利用 git diff 生成标准的 patch

当前的 empty.js 文件

```
public class Empty() {
  console.log('empty file');
  console.log('我正在开发一半中。。。。。。')
```

```
}
```

2） 建立一个新的分支 —— cbrh

```
git checkout -b cbrh
```

3） 修改 empty.js 文件

```
public class Empty() {
  console.log('empty file');
  console.log('我正在开发一半中。。。。。。')

  console.log('补丁修改 1');
  console.log('补丁修改 2');
}
```

4） 而后查看前后代码的不同

```
git diff empth.js
```

```
[wzydeMacBook-Pro:mypro wzy$ git diff empth.js
diff --git a/empth.js b/empth.js
index 2ee9ae3..a1f98fe 100644
--- a/empth.js
+++ b/empth.js
@@ -1,3 +1,7 @@
 public class Empty() {
   console.log('empty file');
+  console.log('我正在开发一半中。。。。。。')
+
+  console.log('补丁修改1');
+  console.log('补丁修改2');
 }
```

此时可以发现 Emp.java 文件修改前后的对比情况。

5） 在 cbrh 上进行代码的提交

```
git commit -a -m "add 2 line empty.js "
```

此时并没有和主分支进行提交，但是代码已经改变了，需要的是将代码的变化提交

给开发者。

6） 生成补丁文件 —— mypatch

```
git diff master > mypatch
```

7）切换回 master 分支

此时会自动在项目目录中生成一个 mypat 的补丁文件信息。这个文件是可以由 git 读懂的信息文件，那么完成之后现在需要模拟另外一个开发者，另外一个开发者假设是专门进行补丁合并的开发者。

8）创建并切换一个新的分支

```
 git checkout -b patchbrh
```

9）应用补丁信息

```
git apply mypatch
```

此时补丁可以成功的使用了。

10）提交补丁的操作

```
git commit -a -m "patch apply"
```

11）切换回 master 分支之中进行分支合并

```
git checkout master
git merge --no-ff -m "Merge Patch" patchbrh
```

这样如果只是将补丁数据的文件发送给开发者，那么就没有必要进行大量代码的传输，并且在创建补丁的时候也可以针对于多个文件进行补丁的创建。

7. 利用 git format-patch 生成 GIT 专用补丁

1）创建并切换到 cbrh 分支

```
git branch -D cbrh
git branch -D patchbrh
git checkout -b cbrh
```

2）创建并切换到 cbrh 分支

```
public class Empty() {
```

```
  console.log('empty file');
  console.log('git format-patch 测试')
}
```

3）创建并切换到 cbrh 分支

```
git commit -a -m "add formatch test"
```

4）下面需要与原始代码做一个比较，而且比较后会自动的生成补丁文件

```
git format-patch -M master
```

现在表示要与 master 分支进行比较(而-M 参数就是指定分支)。

```
wzydeMacBook-Pro:mypro wzy$ git format-patch -M master
0001-add-formatch-test.patch
```

此时已经生成了一个补丁文件，因为只修改了一次的内容。这个补丁文件严格来将就是一个 email 数据，需要将此数据发送给开发者，而后开发者可以进行补丁的应用。

5）创建并切换到 patchbrh 分支上

```
git checkout master
git checkout -b patchbrh
```

6） 应用补丁的信息，利用“git am”完成

```
git am 0001-add-formatch-test.patch
```

```
wzydeMacBook-Pro:mypro wzy$ git am 0001-add-formatch-test.patch
Applying: add formatch test
```

现在是将发送过来的，带有 email 格式的补丁文件进行了应用。

7） 提交应用的更新

```
git commit -a -m "method patch apply"
```

那么此时就可以成功的应用补丁进行代码的更正。

关于两种补丁方式的说明

- 使用 git diff 生成补丁兼容性是比较好的，如果你是在不是 git 管理的仓库上，此类方式生成的补丁是非常容易接受的;
- 但是如果你是向公共的开发社区进行代码的补丁更正，那么建议使用 git format-patch，这样不仅标准，而且也可以将更正人的信息进行公布。

8. 多人协作开发

分支的处理实际上是为了更好的多人开发做出的准备，那么下面就将利用两个命令行方式(模拟其他的开发者)进行项目代码的编写。首先说明一下:

- 一般而言，master 分支项目的核心分支，只要进行代码的克隆，那么此分支一定会被保存下来;
- 开发者往往会建立一系列的分支，譬如，本次练习建立了一个 brh 的分支进行代码的编写;
- 如果要进行调试可以建立一个 bug 分支;
- 如果要增加某些新的功能则可以建立 feature 分支。

1） 创建并切换到一个新的分支:brh

```
git checkout -b brh
```

2）在新的分支上建立一个新的文件 —— Dept.js

```
public class Dept() {
  console.log('多人协作开发!');
}
```

3）将此代码进行提交

```
git commit -a -m 'add dept.js files'
```

4）将两个分支提交到服务器上去

```
git push origin master
git push origin brh
```

5） [二号]为了模拟第二个开发者，所以建立一个新的命令行窗口，并且将代码复制下来(d:proclone)

```
git clone https://github.com/qq449245884/HelloGitHub.git
```

6) [二号] 查看分支信息

```
git branch -a
```

```
wzydeMacBook-Pro:HelloGitHub wzy$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/brh
  remotes/origin/master
wzydeMacBook-Pro:HelloGitHub wzy$
```

发现现在只是将 master 分支拷贝下来了，但是 brh 分支并没有存在。

7) [二号]建立并切换到 brh 分支上

```
git checkout -b brh
```

8) [二号]将远程服务器端上的 brh 分支的内容拷贝到本地的 brh 分支上

```
git merge origin/brh
```

9) [二号]现在开发者增加了一个 Admin.js 文件

```
public class Admin() {
  console.log('多人协作测试!:')
}
```

10) [二号]将新的代码进行提交

```
git add .
git commit -m 'add admin.js files'
```

11) [二号]现在本地的 brh 分支代码发生了变化，那么应该将此变化提交到远程的 brh 分支上

```
git push origin brh
```

现在代码已经发送到了服务器上了，并且在 brh 分支上增加了新的 Admin.java 文件。

12) [一号]这个时候最原始的开发者目录下还只是上一次提交的内容。那么需要取得最新的数据才可以

对于取得最新的分支数据有两种方式:

- git fetch: 此操作只是取得最新的分支数据，但是不会发生 merge 合并操作
- git pull: 此操作取出最新分支数据，并且同时发生 merge 合并操作

git pull

```
wzydeMacBook-Pro:mypro wzy$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/qq449245884/HelloGitHub
   e9e19fc..2bd6f5a  brh        -> origin/brh
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

    git branch --set-upstream-to=origin/<branch> brh

wzydeMacBook-Pro:mypro wzy$ ls
dept.js         empth.js        hello.js
```

实际上错误信息也很简单，指的是，当前的 brh 分支和服务器上的分支没有关系，所以如果要想读取代码，必须让两 个分支产生关联关系。

```
git branch --set-upstream-to=origin/brh
```

随后再次读取所有的代码。

13) [二号]修改 Admin.js 类文件

```
public class Admin() {
  console.log('多人协作测试!:')
  console.log('二号我来个性了！');
}
```

14) [二号]将以上的代码进行提交

```
 git commit -a -m 'update admin.js file'
```

15) [二号]向服务器端提交代码的修改

```
git push origin brh
```

16) [一号]开发者也进行 Admin.js 文件的修改

```
public class Admin() {
  console.log('多人协作测试!:')
  console.log('一号也进行修改了!')
}
```

17) [一号]将代码提交

```
git commit -a -m "1 update admin.js file"
```

但是这个时候很明显，两个用户一起修改了同一个文件。

18) [一号]抓取最新的更新数据

```
git pull
```

```
wzydeMacBook-Pro:mypro wzy$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/qq449245884/HelloGitHub
   2bd6f5a..a600e11  brh        -> origin/brh
Auto-merging admin.js
CONFLICT (content): Merge conflict in admin.js
Automatic merge failed; fix conflicts and then commit the result.
wzydeMacBook-Pro:mypro wzy$
```

现在可以发现，此时的程序，是两位开发者修改了同一个代码，所以产生了冲突。

同时一号开发者之中的 Admin.js 文件的内容已经变更为如下情:

```
public class Admin() {
  console.log('多人协作测试!:')
<<<<<<< HEAD
  console.log('一号也进行修改了!')
=======
  console.log('二号我来个性了！ ');
>>>>>>> a600e113d2d139efc73eee2052ad509fa95d16e3
}
```

19) [一号]手工解决冲突文件内容

```
public class Admin() {
  console.log('多人协作测试!:')
  console.log('一号也进行修改了!')
  console.log('二号我来个性了！ ');
}
```

20) 再次执行提交和服务器推送

```
 git commit -a -m "3 Update Admin.js File"
 git push origin brh
```