

开源项目阅读与管理

六、开源项目阅读与管理-分支管理

开源项目阅读与管理

步骤 1：Bug 分支

步骤 2：Feature 分支

步骤 3：多人协作

步骤 1：BUG 分支

软件开发中，bug 不可避免。有了 bug 就需要修复，在 Git 中，由于分支是如此的强大，所以，每个 bug 都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

当你接到一个修复一个代号 101 的 bug 的任务时，很自然地，你想创建一个分支 issue-101 来修复它，但是，当前正在 dev 上进行的工作还没有提交：

```
李响1@Lixiang MINGW64 /d/myrepo (dev)
$ git add ContractManage.java

李响1@Lixiang MINGW64 /d/myrepo (dev)
$ git status
on branch dev
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ContractManage.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.docx
```

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需 1 天时间。但是，必须在两个小时内修复该 bug，怎么办？幸好，Git 还提供了一个 stash 功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：\$ git stash

```
李响1@Lixiang MINGW64 /d/myrepo (dev)
$ git stash
Saved working directory and index state WIP on dev: 05ced57 modify symbol
```

现在，用 git status 查看工作区，就是干净的（除非有没有被 Git 管理的文件），因此可以放心地创建分支来修复 bug。

```
李响1@Lixiang MINGW64 /d/myrepo (dev)
$ git status
on branch dev
nothing to commit, working tree clean
```

首先确定要在哪个分支上修复 bug，假定需要在 master 分支上修复：\$ git checkout master

```
李响1@LiXiang MINGW64 /d/myrepo (dev)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
  (use "git push" to publish your local commits)

李响1@LiXiang MINGW64 /d/myrepo (master)
$ |
```

从 master 创建临时分支：\$ git checkout -b issue-101

```
李响1@LiXiang MINGW64 /d/myrepo (master)
$ git checkout -b issue-101
Switched to a new branch 'issue-101'

李响1@LiXiang MINGW64 /d/myrepo (issue-101)
$ |
```

现在修复 bug，需要把 license.txt 文件里的 “This is my git repository” 改为 “This’s my git repository”，然后提交：

```
$ git add license.txt
$ git commit -m "fix bug 101"
```

```
李响1@LiXiang MINGW64 /d/myrepo (issue-101)
$ git add license.txt

李响1@LiXiang MINGW64 /d/myrepo (issue-101)
$ git commit -m "fix bug 101"
[issue-101 a82826e] fix bug 101
1 file changed, 1 insertion(+), 1 deletion(-)
```

修复完成后，切换到 master 分支，并完成合并，最后删除 issue-101 分支：

```
$ git switch master
$ git merge --no-ff -m "merged bug fix 101" issue-101
```

```
李响1@LiXiang MINGW64 /d/myrepo (issue-101)
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
  (use "git push" to publish your local commits)

李响1@LiXiang MINGW64 /d/myrepo (master)
$ git merge --no-ff -m "merged bug fix 101" issue-101
Merge made by the 'ort' strategy.
 license.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

再执行删除：

```
$ git branch -D issue-101
```

太棒了，原计划两个小时的 bug 修复只花了 5 分钟！现在，是时候接着回到 dev 分支干活了！

```
$ git switch dev
```

```
$ git status
```

```
李响1@LiXiang MINGW64 /d/myrepo (master)
$ git switch dev
Switched to branch 'dev'

李响1@LiXiang MINGW64 /d/myrepo (dev)
$ git status
on branch dev
nothing to commit, working tree clean
```

工作区是干净的，刚才 dev 分支的工作现场存到哪去了？用 git stash list 命令看看：

```
李响1@LiXiang MINGW64 /d/myrepo (dev)
$ git stash list
stash@{0}: WIP on dev: 05ced57 modify symbol
```

工作现场还在，Git 把 stash 内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用 git stash apply 恢复，但是恢复后，stash 内容并不删除，你需要用 git stash drop 来删除；另一种方式是用 git stash pop，恢复的同时把 stash 内容也删了：

```
$ git stash pop
```

```
李响1@LiXiang MINGW64 /d/myrepo (dev)
$ git stash pop
on branch dev
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ContractManage.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.docx

Dropped refs/stash@{0} (eda29dbc8b2b624ff8db7e68cd122ea54a8a09fd)
```

再用 git stash list 查看，就看不到任何 stash 内容了，而用 git status 查看，我们原先的工作又回来可以接着做了：

```
李响1@LiXiang MINGW64 /d/myrepo (dev)
$ git stash list

李响1@LiXiang MINGW64 /d/myrepo (dev)
$ git status
on branch dev
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ContractManage.java

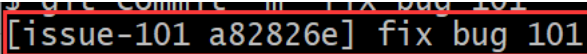
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   readme.docx
```

在 master 分支上修复了 bug 后，我们要想一想，dev 分支是早期从 master 分支分出来的，所以，这个 bug 其实在当前 dev 分支上也存在。

那怎么在 dev 分支上修复同样的 bug？重复操作一次，提交不就行了？有木有更简单的方法？有！

同样的 bug，要在 dev 上修复，我们只需要把 a82826e fix bug 101 这个提交所做的修改“复制”到 dev 分支。注意：我们只想复制 a82826e fix bug 101 这个提交所做的修改，并不是把整个 master 分支 merge 过来。

下面红框部分就是上面在提交时候生成的 id: a82826e



为了方便操作，Git 专门提供了一个 cherry-pick 命令，让我们能复制一个特定的提交到当前分支。

当执行 cherry-pick 命令之前我们需要在当前分支先临时提交一次：

```
$ git commit -m "temp commit"
```

```
$ git cherry-pick a82826e
```

Git 自动给 dev 分支做了一次提交，注意这次提交的 commit 是 1d4b803，它并不同于 master 的 a82826e，因为这两个 commit 只是改动相同，但确实是两个不同的 commit。用 git cherry-pick，我们就不需要在 dev 分支上手动再把修 bug 的过程重复一遍。

在开发中，不要使用 cherry-pick 来进行不同分支之间代码的同步，这很可能会造成最终合并时出现冲突，而且可能产生比冲突更严重的问题：该有冲突却没有冲突。

PS：何时使用 cherry-pick

当我们需要在不同分支之间移动提交时，可以使用 cherry-pick。因为项目需要有很多分支相互 merge，所以 cherry-pick 很容易造成问题。但是如果你的两个分支是两个单独的分支，永远不会相互 merge，那么就可以使用 cherry-pick 而不用担心上述问题。

■ 总结：

先说几个名词

未被追踪的文件：指的是新建的文件或文件夹且还没加入到暂存区（新建的还没有被 git add 过的）

未加入到暂存区的文件：指的是已经被追踪过，但是没有加入到暂存区（已经执行过 git add/commit 的但是这次修改后还没有 git add）

举例：readme.md 已经被 git add/git commit 过，但是呢 我这次只是修改了，而且没有修改完，不能 commit。

test 新建的文件夹，没有被 git add/git commit 过 有个急事需要处理，这时候我需要切换分支，去处理紧急任务，比如文中的举例去修改 bug，

正确的步骤：git add test（让 git 去追踪这个新文件）

git stash 保留现场

如果我不执行这两个命令，那么我在修改 BUG 完成之后

git status，就会发现 readme.md 没有添加到暂存区，

同时又多了个 test 文件，但是自己的 readme.md 没有完成，

万万不可以提交，这样导致 bug 的修改代码也提交不了。

所以你需要 git stash，这样你在提交修改 bug 代码的时候，

就不会看见 readme.md 和 test。可以安心提交修改 bug 的代码。

首先用 `git status` 看看，有没有add的，也就是新文件。

有执行： `git stash -a` ,其中-a代表所有（追踪的&未追踪的）

没有执行： `git stash`

总之:都可以执行 `git stash -a`

步骤 2：FEATURE 分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个 feature 分支，在上面开发，完成后，合并，最后，删除该 feature 分支。

现在，你终于接到了一个新任务：开发代号为 pro 的新功能，于是准备开发：

```
$ git switch -c feature-pro
```

```
李响1@LiXiang MINGW64 /d/myrepo (dev)
$ git switch -c feature-pro
Switched to a new branch 'feature-pro'

李响1@LiXiang MINGW64 /d/myrepo (feature-pro)
$ |
```

开发完毕以后执行 add 、 status 和 commit 命令：

```
$ git add ProManage.java
$ git status
$ git commit -m "add product manage"
```

```
李响1@LiXiang MINGW64 /d/myrepo (feature-pro)
$ git add ProManage.java

李响1@LiXiang MINGW64 /d/myrepo (feature-pro)
$ git status
on branch feature-pro
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ProManage.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   ContractManage.java

李响1@LiXiang MINGW64 /d/myrepo (feature-pro)
$ git commit -m "add product manage"
[feature-pro ba174c7] add product manage
1 file changed, 7 insertions(+)
create mode 100644 ProManage.java
```

接下来合并 feature 分支到 master 分支，然后删除。

但是！因为一些特殊原因，新功能必须取消！虽然白干了，但是这个包含机密资料的分支还是必须就地销毁：

```
$ git branch -d feature-pro
```

```
李响1@Lixiang MINGW64 /d/myrepo (master)
$ git branch -d feature-pro
error: The branch 'feature-pro' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-pro'.
```

销毁失败。Git 友情提醒，feature-pro 分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用大写的-D 参数。现在我们强行删除：

```
$ git branch -D feature-pro
```

```
李响1@Lixiang MINGW64 /d/myrepo (master)
$ git branch -D feature-pro
Deleted branch feature-pro (was ea22e7b).
```

■ 总结：

开发一个新 feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D <name>` 强行删除。

步骤 3：多人协作

当你从远程仓库克隆时，实际上 Git 自动把本地的 master 分支和远程的 master 分支对应起来了，并且，远程仓库的默认名称是 origin。

要查看远程库的信息，用 `git remote` 命令：

```
李响1@Lixiang MINGW64 /d/myrepo (master)
$ git remote
origin
```

或者，用 `git remote -v` 显示更详细的信息：

```
李响1@Lixiang MINGW64 /d/myrepo (master)
$ git remote -v
origin https://github.com/lixiangood/myrepo.git (fetch)
origin https://github.com/lixiangood/myrepo.git (push)
```

上面显示了可以抓取和推送的 origin 的地址。如果没有推送权限，就看不到 push 的地址。

1、推送分支：

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git 就会把该分支推送到远程库对应的远程分支上：`$ git push origin master`

```
李响1@Lixiang MINGW64 /d/myrepo (master)
$ git push origin master
Enumerating objects: 35, done.
Counting objects: 100% (35/35), done.
Delta compression using up to 4 threads
Compressing objects: 100% (28/28), done.
Writing objects: 100% (32/32), 25.07 KiB | 1.79 MiB/s, done.
Total 32 (delta 7), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (7/7), completed with 1 local object.
To https://github.com/lixiangood/myrepo.git
c06dd0d..592fe92 master -> master
```

如果要推送其他分支，比如 dev，就改成：`$ git push origin dev`

```
李响1@Lixiang MINGW64 /d/myrepo (master)
$ git push origin dev
Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 4 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (15/15), 1.39 KiB | 357.00 KiB/s, done.
Total 15 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), done.
remote:
remote: Create a pull request for 'dev' on GitHub by visiting:
remote:   https://github.com/lixiangood/myrepo/pull/new/dev
remote:
To https://github.com/lixiangood/myrepo.git
* [new branch]      dev -> dev
```

执行 push 以后查看远程仓库的状态：

lixiangood / myrepo Public

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

🔗 dev had recent pushes less than a minute ago [Compare & pull request](#)

🔗 master 2 branches 0 tags [Go to file](#) [Add file](#) [Code](#)

Switch branches/tags

Find or create a branch...

Branches Tags

✓ master default

dev

dev分支本来没有，刚刚push过来的.

592fe92 5 days ago 24 commits

fix bug 103	5 days ago
modify symbol	6 days ago
add test.txt	8 days ago

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

- master 分支是主分支，因此要时刻与远程同步；
- dev 分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
- bug 分支用于在本地修复 bug，不需要推到远程。
- feature 分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在 Git 中，分支完全可以在本地创建使用，是否推送，视你的心情而定！

2、抓取分支

多人协作时，大家都会往 master 和 dev 分支上推送各自的修改。现在，模拟一个你的同事，和你做协同开发：可以在另一台电脑（添加 SSH Key）或者同一台电脑的另一个目录下克隆：

用命令 `git clone` 克隆一个本地库，在 Git Bash 中先把当前目录切换到 C 盘：

```
李响1@Lixiang MINGW64 /d/myrepo (master)
$ cd c:

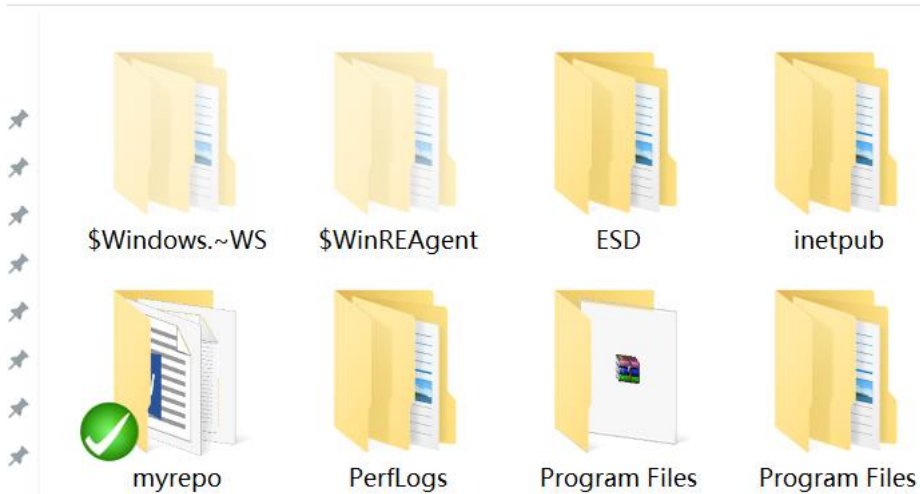
李响1@Lixiang MINGW64 /c
$
```

输入命令：\$ `git clone https://github.com/lixiangood/myrepo.git`

```
李响1@Lixiang MINGW64 /c
$ git clone https://github.com/lixiangood/myrepo.git
Cloning into 'myrepo'...
remote: Enumerating objects: 73, done.
remote: Counting objects: 100% (73/73), done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 73 (delta 20), reused 73 (delta 20), pack-reused 0
Receiving objects: 100% (73/73), 62.83 KiB | 6.00 KiB/s, done.
Resolving deltas: 100% (20/20), done.
```

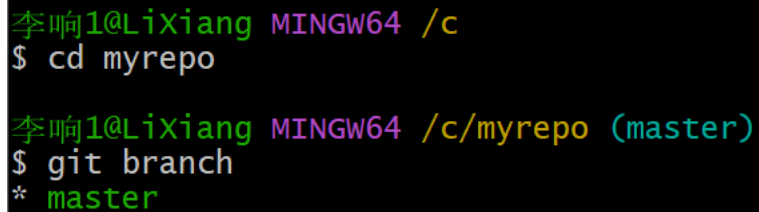
查看 C 盘，出现了刚刚抓取的 myrepo 目录。

此电脑 > OS (C:) >



切换到C盘地下的本地库myrepo，并执行命令查看分支：

```
$ cd myrepo
$ git branch
```

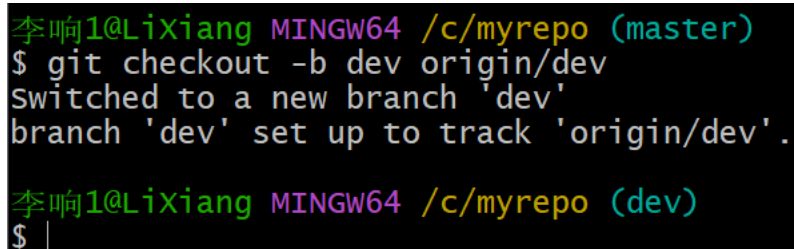


```
李响1@Lixiang MINGW64 /c
$ cd myrepo

李响1@Lixiang MINGW64 /c/myrepo (master)
$ git branch
* master
```

现在，你的同事要在 dev 分支上开发，就必须创建远程 origin 的 dev 分支到本地，于是他用这个命令创建本地 dev 分支：

```
$ git checkout -b dev origin/dev
```

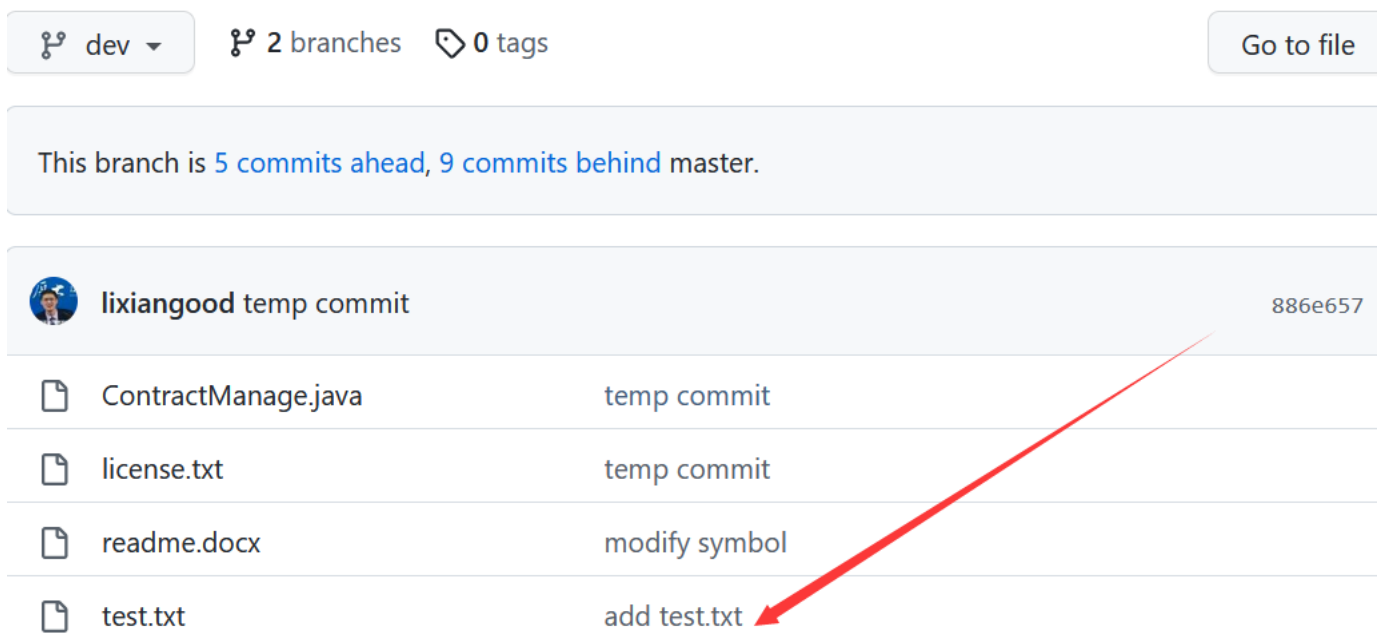


```
李响1@Lixiang MINGW64 /c/myrepo (master)
$ git checkout -b dev origin/dev
Switched to a new branch 'dev'
branch 'dev' set up to track 'origin/dev'.

李响1@Lixiang MINGW64 /c/myrepo (dev)
$ |
```






现在，他就可以在 dev 上继续修改，然后，时不时地把 dev 分支 push 到远程。

比如：在你的本地库和远程库的 dev 分支已经有 test.txt 文件存在，并且里面有内容：



dev 2 branches 0 tags Go to file

This branch is 5 commits ahead, 9 commits behind master.







	lixiangood temp commit	886e657
	ContractManage.java	temp commit
	license.txt	temp commit
	readme.docx	modify symbol
	test.txt	add test.txt

如果这时模拟你的同事，在 C 盘 myrepo 下直接创建 dev.txt，输入内容 dev，并推送到了远程库的 dev 分支：

```
$ git add dev.txt
$ git commit -m "add new dev"
$ git push origin dev
```

```
李响1@Lixiang MINGW64 /c/myrepo (dev)
$ git push origin dev
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 261 bytes | 261.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/lixiangood/myrepo.git
886e657..d7e6044 dev -> dev
```

下面是在github上查看的你的同事在远程库中新同步的dev.txt文件：

 lixiangood add new dev		
	ContractManage.java	temp commit
	dev.txt	add new dev ←
	license.txt	temp commit
	readme.docx	modify symbol
	test.txt	add test.txt

这时你如果再创建dev.txt，提交就会推送失败，因为你的提交推送和你小伙伴之前的提交有冲突，解决办法也很简单，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：

```
$ git pull
```

```
李响1@Lixiang MINGW64 /d/myrepo (dev)
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), 241 bytes | 7.00 KiB/s, done.
From https://github.com/lixiangood/myrepo
886e657..d7e6044 dev -> origin/dev
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> dev
```

如果git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：

```
$ git branch --set-upstream-to=origin/dev dev
```

再执行pull:

```
$ git pull
```

A terminal window with a black background and green text. The prompt is '李响1@Lixiang MINGW64 /d/myrepo (dev)'. The command '\$ git pull' is entered. The output shows 'Updating 886e657..d7e6044', 'Fast-forward', 'dev.txt | 1 +', '1 file changed, 1 insertion(+)', and 'create mode 100644 dev.txt'.

```
李响1@Lixiang MINGW64 /d/myrepo (dev)
$ git pull
Updating 886e657..d7e6044
Fast-forward
 dev.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 dev.txt
```

这回git pull成功，如果合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后，提交，再push:

```
$ git add env.txt
$ git commit -m "fix env"
$ git push origin dev
```

因此，多人协作的工作模式通常是这样:

1. 首先，可以试图用 `git push origin <branch-name>` 推送自己的修改;
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并;
3. 如果合并有冲突，则解决冲突，并在本地提交;
4. 没有冲突或者解决掉冲突后，再用 `git push origin <branch-name>` 推送就能成功!

如果 `git pull` 提示 `no tracking information`，则说明本地分支和远程分支的链接关系没有创建，用命令 `git branch --set-upstream-to <branch-name> origin/<branch-name>`。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

■ 总结:

查看远程库信息，使用 `git remote -v`;

本地新建的分支如果不推送到远程，对其他人就是不可见的;

从本地推送分支，使用 `git push origin branch-name`，如果推送失败，先用 `git pull` 抓取远程的新提交;

在本地创建和远程分支对应的分支，使用 `git checkout -b branch-name origin/branch-name`，本地和远程分支的名称最好一致;

建立本地分支和远程分支的关联，使用 `git branch --set-upstream-to branch-name origin/branch-name`;

从远程抓取分支，使用 `git pull`，如果有冲突，要先处理冲突。