LearningGit

版本库（仓库，repository）

- 创建仓库 git init

  注意事项：

  1. 只能管理文本文件；推荐使用UTF-8编码；
  2. Windows下不要使用记事本编辑文件；推荐使用Notepad++，编码选择UTF-8 without BOM;
- 把文件添加到仓库 git add readme.txt
- 把文件提交到仓库 git commit -m "INI: this is a initial commit"

```
kinglee@DESKTOP-SLM34PF MINGW64 /d/Learning/LearningGit (master)
$ git commit -m "INI: this is a initial commit of LearningGit"
[master (root-commit) bcf7000] INI: this is a initial commit of LearningGit
 2 files changed, 19 insertions(+)
 create mode 100644 LearningGit.md
 create mode 100644 readme.txt
```

  返回信息：2个文件被改动，插入了19行内容；

- 比较文件的变动 git diff readme.txt

```
kinglee@DESKTOP-SLM34PF MINGW64 /d/Learning/LearningGit (master)
$ git diff readme.txt
diff --git a/readme.txt b/readme.txt
index b9ed2b3..1fe7979 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,2 @@
-Git is a verison control system.
+Git is a distribute verison control system.
 Git is a free software.
\ No newline at end of file
```

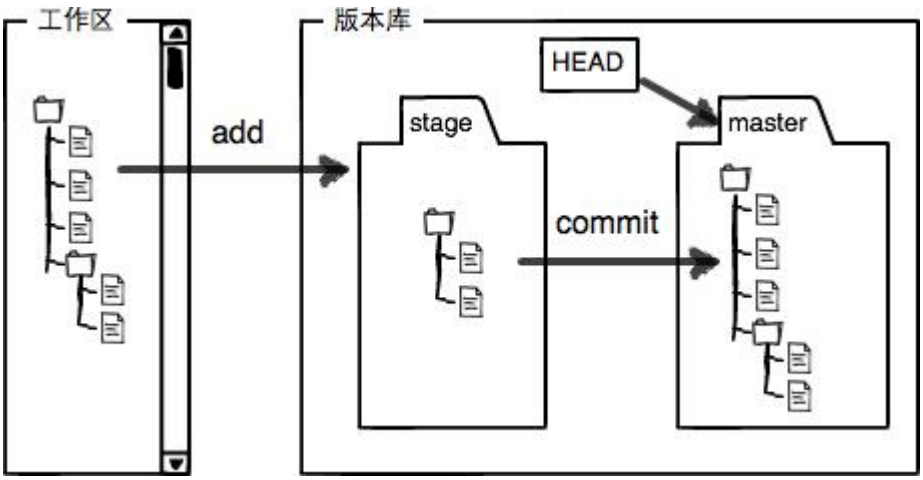
- 查看提交历史记录 git log （查看精简版历史记录 git log --pretty=oneline）
- 回退到上一版本 git reset --hard HEAD^

  注意事项：

  1. git用HEAD表示当前版本，上一个版本HEAD^，上上个版本HEAD^^，往上100个版本HEAD~100；
  2. 想回到某次提交，只需要知道该次提交的commit_id，git reset --hard commit_id;
  3. git版本回退的速度非常快，因为HEAD实际上是个指向当前版本的指针，当回退版本的时候，git仅仅是把HEAD指向了历史版本；
  4. 如果要重返未来，可以用git reflog查看命令历史，从中查看未来版本的commit_id

```
kinglee@DESKTOP-SLM34PF MINGW64 /d/Learning/LearningGit (master)
$ git reflog
bcf7000 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
bda3f52 HEAD@{1}: commit: NEW: Add some information
bcf7000 (HEAD -> master) HEAD@{2}: commit (initial): INI: this is a initial comm
it of LearningGit
```

工作区和暂存区



- 工作区：在电脑里能看到的目录；
- 版本库: .git目录，不算工作区，是git的版本库；版本库里有被称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支`master`，以及指向`master`的一个指针`HEAD`
- 上节将文件添加到git版本库中是分两步执行的：
  1. 用`git add`把文件添加进去，实际上就是把文件修改添加到暂存区；
  2. 用`git commit`提交更改，实际上就是把暂存区的所有内容提交到当前分支。

     因为我们创建Git版本库时，Git自动为我们创建了唯一一个`master`分支，所以，现在，`git commit`就是往`master`分支上提交更改。git diff 命令 实际上比较的是工作区与暂存区的差别；

- git一个重要的设计理念：git追踪并管理的是修改，而并非文件；修改需要先被加入到暂存区，然后才能被提交；没有加入暂存区的修改是不会加入到commit中的；

撤销修改

- git checkout -- readme.txt

  作用：把readme.txt在**工作区**的修改全部撤销，让readme.txt 回到最近一次git commit 或git add时的状态

  注意事项：

  1. 如果已经把readme.txt放入了暂存区，可以使用git reset HEAD 命令，把文件重新放回工作区；然后再使用git checkout --readme.txt 命令；
- 综上，想撤销修改分为三种场景：

1. 当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令`git checkout -- file`。
2. 当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令`git reset HEAD <file>`，就回到了场景1，第二步按场景1操作。
3. 已经提交了不合适的修改到版本库时，想要撤销本次提交，请使用上文讲过的版本回退，不过前提是没有推送到远程库；

删除文件

有两种场景：

1. 确实要从版本库中删除该文件

   工作流：直接删除->git rm test.txt (经测试，此处使用git add也是可以的)->git commit;

2. 误删了文件

   使用git checkout -- test.txt ，可以从版本库中恢复文件（恢复的是最近一次git commit 或git add时的文件）

注意事项：

1. 从来没有被添加到版本库就被删除的文件，是无法恢复的;
2. 添加到版本库的文件，你只能恢复文件到最新版本，你会丢失**最近一次添加到版本库后你修改的内容**;
3. 如果已经提交了删除操作到版本库，这个时候想要恢复文件，只能使用git reset回退版本了；

远程仓库

以使用github 为例：

- 创建SSH Key： ssh-keygen -t rsa -C “youremail@example.com”
- 切换到 ~/.ssh目录， 查看id_rsa.pub的内容（即公钥），添加到远程仓库的SSH Key；
- 在github创建一个仓库，然后与本地仓库关联：

  ```
  git remote add origin git@github.com:kinglee1992/LearningGit.git
  ```

  origin 是远程仓库的名字，这是git的默认叫法；

- 将本地库的内容推送到远程库：

  ```
  git push -u origin master
  ```

  将当前分支master推送到远程；

  注意事项：

  1. 由于远程库是空的，第一次推送`master`分支时，加上了`-u`参数，Git不但会把本地的`master`分支内容推送到远程新的`master`分支，还会把本地的`master`分支和远程的`master`分支关联起来，在以后的推送或者拉取时就可以简化命令（推送：git push origin master）
- 克隆远程仓库：git clone

分支管理

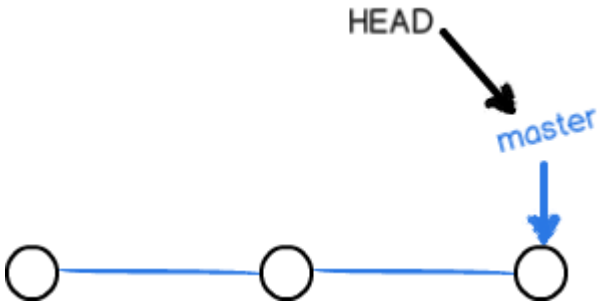
分支的作用

假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

分支的创建与合并

搞清楚HEAD与master的关系：

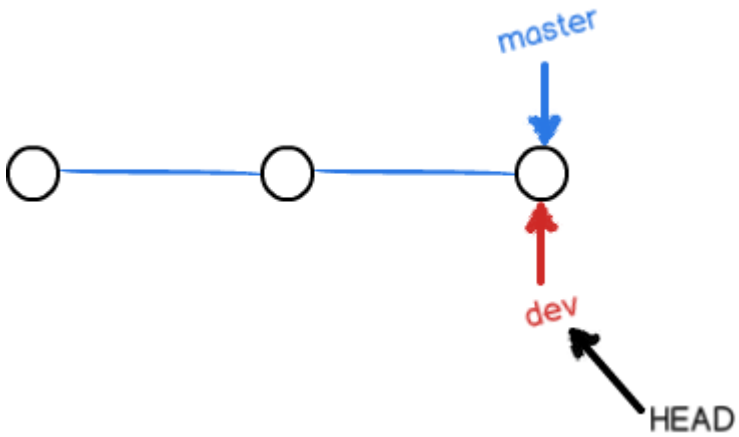


`HEAD`严格来说不是指向提交，而是指向`master`，`master`才是指向提交的，所以，`HEAD`指向的就是当前分支。

一开始的时候，`master`分支是一条线，Git用`master`指向最新的提交，再用`HEAD`指向`master`，就能确定当前分支，以及当前分支的提交点。

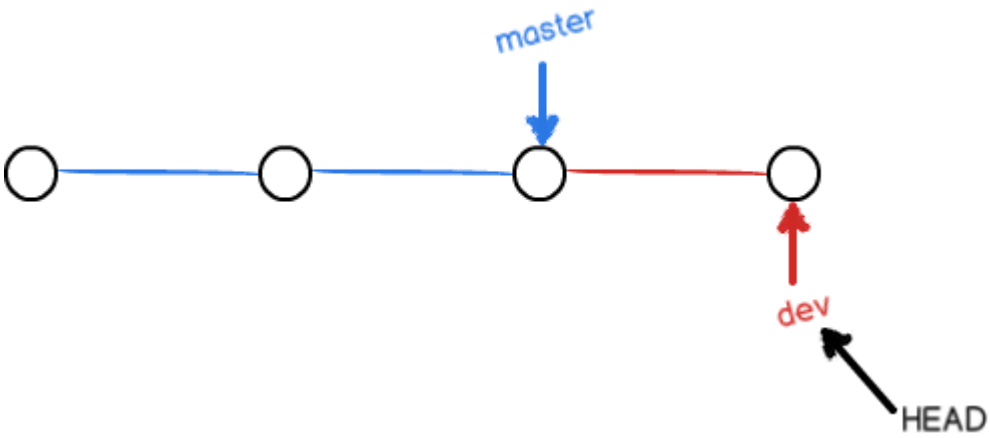
每次提交，`master`分支都会向前移动一步，这样，随着你不断提交，`master`分支的线也越来越长。

当我们创建新的分支，例如`dev`时，Git新建了一个指针叫`dev`，指向`master`相同的提交，再把`HEAD`指向`dev`，就表示当前分支在`dev`上。

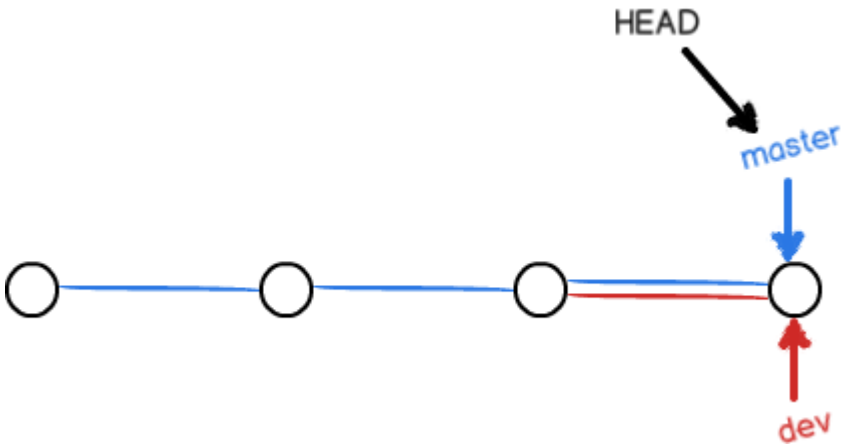


所以，Git创建一个分支很快，因为除了增加一个`dev`指针，改改`HEAD`的指向，工作区的文件都没有任何变化。

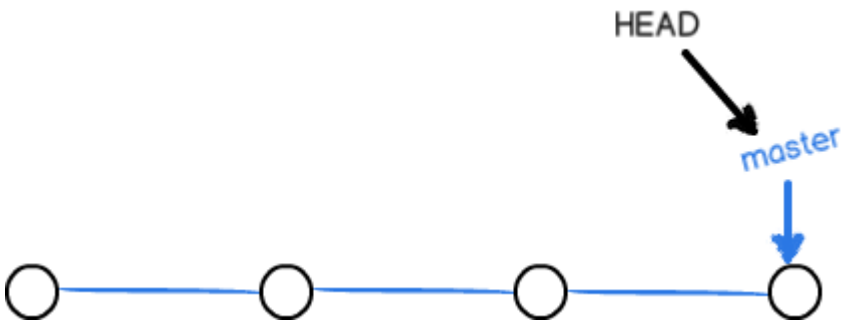
不过，从现在开始，对工作区的修改和提交就是针对`dev`分支了，比如新提交一次后，`dev`指针往前移动一步，而`master`指针不变：



假如我们在 `dev` 上的工作完成了，就可以把 `dev` 合并到 `master` 上。Git怎么合并呢？最简单的方法，就是直接把 `master` 指向 `dev` 的当前提交，就完成了合并：



合并完分支后，甚至可以删除 `dev` 分支。删除 `dev` 分支就是把 `dev` 指针给删掉，删掉后，我们就剩下了一条 `master` 分支：



- 创建一个名为dev的分支,并切换到dev分支： git checkout -b dev

  相当于以下两条命令：

  ```
  $ git branch dev   //创建dev分支
  $ git checkout dev  //切换到dev分支
  ```

- 列出所有分支： git branch

- 分支合并：

```
$git merge dev  //合并指定分支到当前分支
Updating fb7e51e..66c7bab
Fast-forward    //此次合并为“快进模式”，即直接把master指向dev的提交
 readme.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

- 分支删除：git branch -d dev

  注意事项：

  1. 因为创建、合并和删除分支非常快，所以Git鼓励使用分支完成某个任务，合并后再删掉分支，这和直接在`master`分支上工作效果是一样的，但过程更安全。
  2. git 2.23 以后支持新命令：switch 用于切换分支

     创建并切换到新的`dev`分支，可以使用：

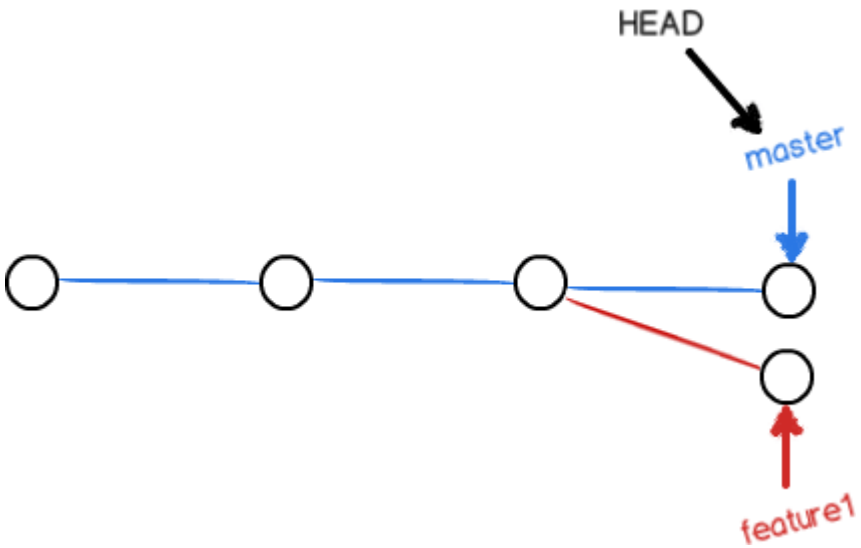
     ```
     $ git switch -c dev
     ```

     直接切换到已有的`master`分支，可以使用：

     ```
     $ git switch master
     ```

     使用新的`git switch`命令，比`git checkout`要更容易理解。

解决冲突



在master和dev分支上都有了新的提交，这个时候进行合并的话，有可能会发生冲突。

解决冲突的步骤：

1. 首先在master分支上试图合并feature1分支，提示有冲突；

```
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

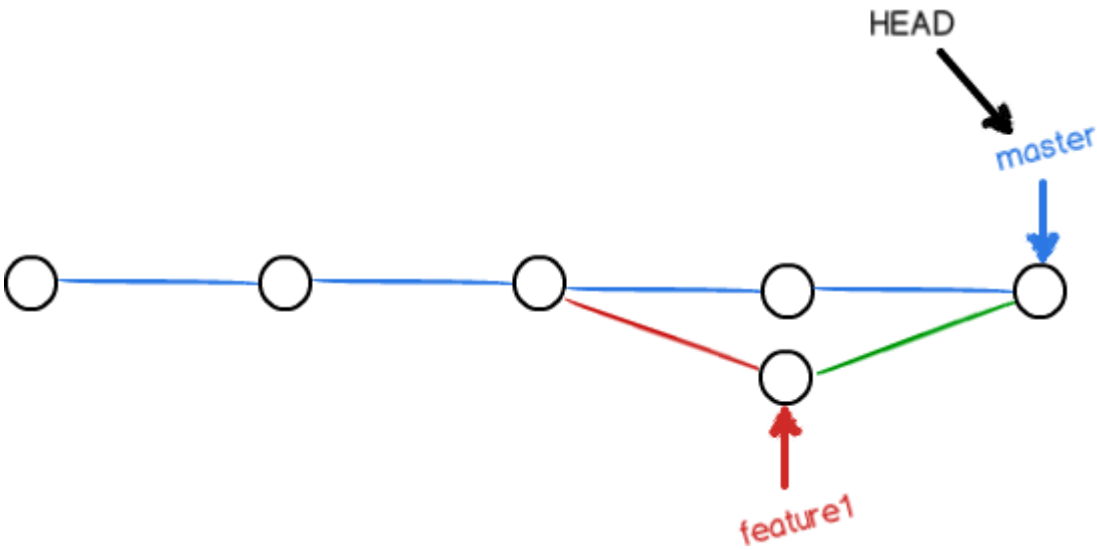
2. 查看冲突文件的内容，Git用`<<<<<<<`，`=======`，`>>>>>>>`标记出不同分支的内容；

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
<<<<<<< HEAD
Creating a new branch is quick & simple.
=======
Creating a new branch is quick AND simple.
>>>>>>> feature1
```

3. 手动对冲突文件进行修改并保存，然后提交；

```
$ git add readme.txt
$ git commit -m "conflict fixed"
[master cf810e4] conflict fixed
```

提交完成后分支情况如下：



4. 用`git log --graph`命令可以看到分支合并图（精简版分支图命令： git log --graph --pretty=oneline --abbrev-commit)

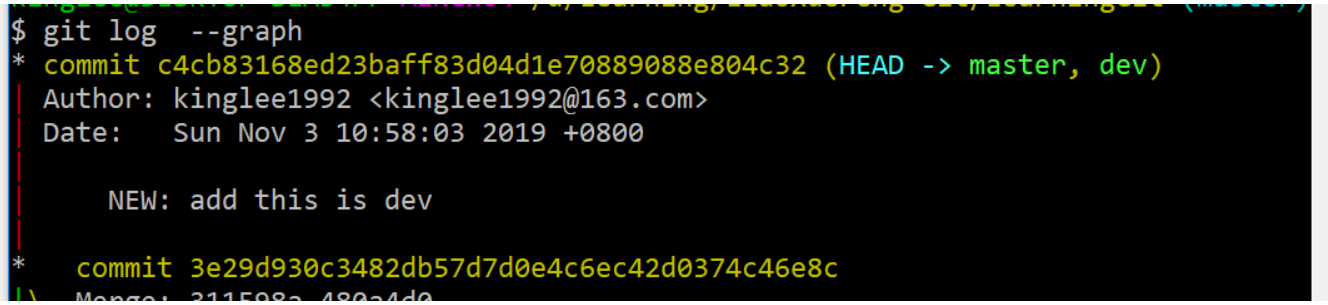
分支管理策略

- --no-ff参数

通常，合并分支时，如果可能，Git会用`Fast forward`模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用`Fast forward`模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

比如创建了dev分支，在dev分支上提交后，切回master分支进行合并；如果直接使用 git merge dev，则分支图如下：

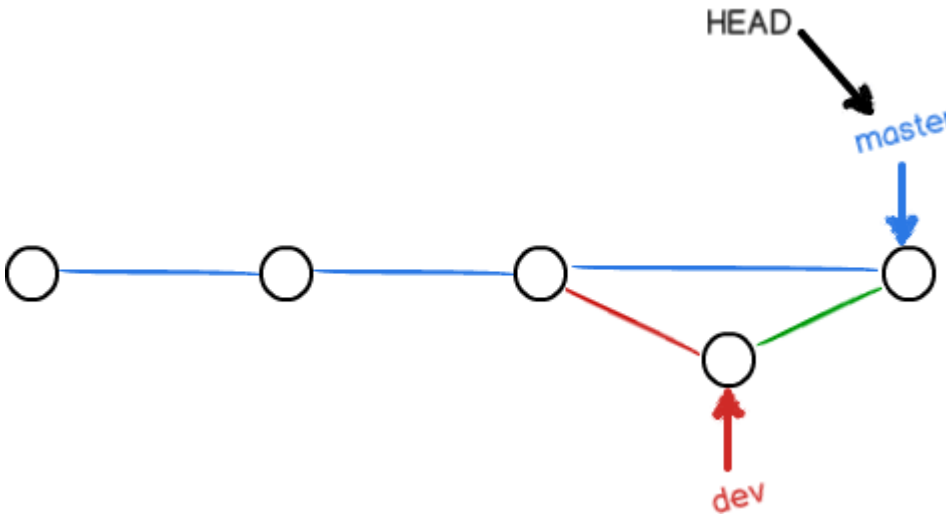


如果使用了`--no-ff`参数，表示禁用`Fast forward`：

```
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

注意：因为本次合并要创建一个新的commit，所以加上`-m`参数，把commit信息写进去。则分支图如下：

可以看到，不使用`Fast forward`模式，merge后就像这样：



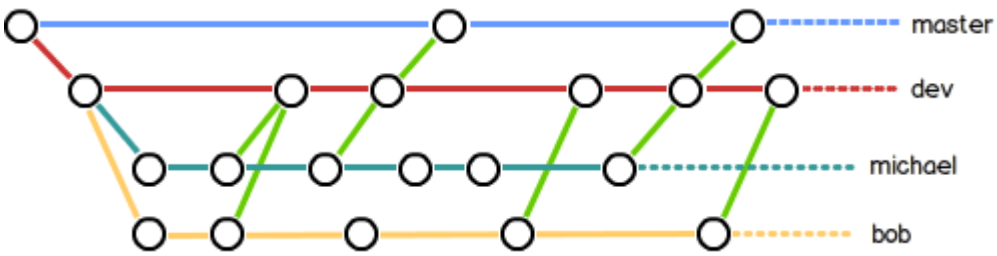
综上，合并分支时，加上`--no-ff`参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而`fast forward`合并就看不出来曾经做过合并。

- 分支策略

  在实际开发中，应该按照几个基本原则进行分支管理：

  - 首先，`master`分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；
  - 干活都在`dev`分支上，也就是说，`dev`分支是不稳定的，到某个时候，比如1.0版本发布时，再把`dev`分支合并到`master`上，在`master`分支发布1.0版本；
  - 每个人都在`dev`分支上干活，每个人都有自己的分支，时不时地往`dev`分支上合并就可以了。

  所以，团队合作的分支看起来就像这样：



Bug分支

想象这样一个场景：你正在dev分支做开发，当前在dev上进行的工作还没有提交（预计还要一天才能完成），这时突然有个bug 必须在两小时内修复，你想要切回master分支修复该bug。

工作流如下：

1. 使用git stash 保存dev分支的开发现场；（注意：这是因为切换分支前，必须保证工作区是干净的；否则切到master分支，你会发现dev分支在工作区的修改会被同步过来；想要保证工作区干净，要么commit，要么stash，由于工作还没做完，只能选择stash）
2. 切换到master 分支；
3. 在master 分支基础上创建bugfix分支；
4. 在bugfix分支上修改完bug后，合并到master分支；

5. 切换到dev分支继续干活，使用git stash pop命令恢复之前保存的工作区；

   （注意：也可以使用git stash apply 恢复，同pop的区别在于，恢复后stash的内容并不删除，需要git stash drop命令来删除）

   (可以多次stash，用git stash list 用来查看stash 记录，再用

   ```
   $ git stash apply stash@{0}
   ```

   命令来恢复指定的stash)

6. 由于master分支上的bug在dev分支也会存在，因此要使用cherry-pick命令，复制一个特定的提交(即修复bug那次的提交)到当前分支

```
git cherry-pick <commit_id>
```

（注意：使用cherry-pick时，也必须保证工作区是干净的）

使用cherry-pick后，Git会自动给dev分支做了一次提交，注意这次提交的commit_id并不同于master的commit_id，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。

Feature分支

开发一个新feature，最好新建一个分支；

在新分支上已经开发完成并commit，但并未在master 上进行合并，此时要丢弃该分支，可以通过`git branch -D <name>`强行删除。

多人协作

- 查看远程库信息： git remote -v
- 推送分支：git push origin master

  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，如果要推送其他分支，比如`dev`，就改成：git push origin dev （在github上测试，此时如果远程库还没有该分支，则会自动创建该分支）

  并非所有的本地分支都要推送到远程：

  - `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分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
  - 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，分支完全可以只在本地开发，是否推送取决于实际情况。

- 抓取分支工作流：

  1. 另一台电脑，git clone 完成后，默认情况下，只能看到master分支；如果要同步远程的dev分支到本地，则必须使用git checkout -b dev origin/dev
  2. 使用`git push origin <branch-name>`推送自己的修改；

3. 如果推送失败，则因为远程分支比你的本地更新(即已经有小伙伴更新了代码并抢先push了上去)，需要先用`git pull`合并；
4. 如果合并有冲突，则解决冲突后，在本地提交，再push；

注意事项：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>`。

Rebase

考虑以下场景：

你从远程同步过代码后，继续开发，然后commit了两次；此时想要push到远程，发生冲突；于是你从远程pull下代码，解决冲突，然后再提交，再push。

此时的分支图：

```
$ git log --graph --pretty=oneline
*   eccbb304a2f203542269d791c76adf344c884b56 (HEAD -> dev) rebase:merge conflict
|\
| * 9cba4feb158923107353fe381d5f0336f921261a (origin/dev) to test rebase,add I p
ush first
* | 69885feaf5ddfcb3ab4bf027fc73339d1ec7bc71 rebase,change2
* | a1877ca5443c999dbb24032b218213c87d55f552 rebase,change1
|/
*   eaa8a59b1d2d3ce33285268a5060462872098b4a test push dev sovlving conflict
```

而使用git rebase命令可以将提交历史变成这样：

```
$ git log --graph --pretty=oneline
* 6351a2aad7c1525e93c1291018ca734fe07a0c93 (HEAD -> dev) rebase,change2
* 33f3dbe57a3a16d65d6f531738b1e1bb1b47204a rebase,change1
* 9cba4feb158923107353fe381d5f0336f921261a (origin/dev) to test rebase,add I pus
h first
*   eaa8a59b1d2d3ce33285268a5060462872098b4a test push dev sovlving conflict
```

注意事项：

- rebase过程中可能会发生冲突，解决冲突的办法：
  1. 修改冲突部分
  2. git add
  3. `git rebase --continue`
  4. （如果第三步无效可以执行`git rebase --skip`）

  不要在git add 之后习惯性的执行 git commit命令

标签管理

标签(tag)就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

创建标签

在Git中打标签非常简单：

1. 切换到需要打标签的分支上；
2. 敲命令`git tag <name>`就可以打一个新标签；默认标签是打在最新提交的commit上的；
3. 可以用命令`git tag`查看所有标签；

   注意事项：

   1. 如果要对历史提交打标签，先找到历史提交的commit id，然后打上就可以了,例如：

      ```
      $ git tag v0.9 f52c633
      ```

   2. 可以用`git show <tagname>`查看标签信息；还可以创建带有说明的标签，用`-a`指定标签名，`-m`指定说明文字；

      ```
      $ git tag -a v0.1 -m "version 0.1 released" 1094adb
      ```

   3. 标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支都可以看到这个标签；

操作标签

- 删除标签： git tag -d v0.1

  注意事项：

  1. 因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除；
  2. 如果要推送某个标签到远程，使用命令`git push origin <tagname>`；或者，一次性推送全部尚未推送到远程的本地标签：$ git push origin --tags
  3. 如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：$ git tag -d v0.9

     然后，从远程删除：git push origin :refs/tags/v0.9

GitHub的使用

如何参与一个开源项目：

1. 在项目主页，点“Fork”就在自己的账号下克隆了一个仓库，然后，从自己的账号下clone（要从自己账号下clone，否则没有推送权限）；
2. 如果你希望项目官方库能接受你的修改，可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了；

自定义Git