简介

Git: 一个**分布式**版本控制系统

集中式vs分布式：

- 集中式: 版本库集中存放在中央服务器，使用时，从中央服务器获取最新的版本，修改完后，再推送给中央服务器。且集中式版本控制系统必须联网才能工作，不方便。
- 分布式：每个人的电脑都有一个完整的版本库，所以工作时不需要联网。将仓库进行同步就可以完成多人协作，但其实并不常用这种方法，而是通过一台充当“中央服务器”的电脑来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。

Git安装

Linux 下安装

输入git,查看是否已经安装Git:

```
$ git
The program 'git' is currently not installed. You can install it by typing:
sudo apt-get install git
```

像上面的命令，有很多Linux会友好地告诉你Git没有安装，还会告诉你如何安装Git。

如果碰巧用Debian或Ubuntu Linux，通过一条`sudo apt-get install git`就可以直接完成Git的安装，非常简单。

老一点的Debian或Ubuntu Linux，要把命令改为`sudo apt-get install git-core`，因为以前有个软件也叫GIT（GNU Interactive Tools），结果Git就只能叫`git-core`了。由于Git名气实在太大，后来就把GNU Interactive Tools改成`gnuit`，`git-core`正式改为`git`。

如果是其他Linux版本，可以直接通过源码安装。先从Git官网下载源码，然后解压，依次输入：`./config`，`make`，`sudo make install`这几个命令安装就好了。

Windows下安装

在Windows上使用Git，可以从Git官网直接下载安装程序，然后按默认选项安装即可。

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！

安装完成后，还需要最后一步设置，在命令行输入：

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

注意`git config`命令的`--global`参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

创建版本库

版本库(Repository、Repo)

又叫仓库。可以理解为一个目录，目录里所有的文件都可以被Git管理，每个文件的修改、删除、添加等操作都能被Git跟踪到，便于追踪历史或者还原到之前的某一个版本。

所以创建仓库非常简单，只需要把一个目录变成Git可以管理的仓库就可以.

创建仓库

使用git bash

- 方式一:在本地创建一个仓库

```
$ mkdir learn-git //创建一个目录（make directories）
$ cd learn-git //切换目录（change directories）
$ pwd //显示当前目录
/c/Users/14693/learn-git
```

```
$ git init
Initialized empty Git repository in C:/Users/14693/learn-git/.git/
```

git init 可以把这个目录变成git可以管理的仓库。

这样仓库就建好了，而且告诉我们这是一个空仓库(empty Git repository),当前目录下多了一个`.git`的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

- 方式二：从远程服务器克隆一个已经存在的仓库

```
$ git clone 地址
```

创建完仓库后，可以在当前库下用`ls -a`或`ls -ah`查看库中目录。

对版本状态的操作

工作区域和文件状态

工作区域

Git的本地数据管理分为三个区域：工作区(working directory)、暂存区(staging area)、本地仓库(local repository)

工作区:又叫工作目录或本地工作目录,我们在资源管理器中能看到的文件夹就是工作区

暂存区:又叫索引（index），是一种临时存储区域，它用于保存即将提交到git仓库的修改内容

本地仓库：通过`git init`创建的仓库就是本地仓库。包含了完整的项目历史和元数据，是Git存储代码和版本信息的主要位置

当我们修改完工作区的文件后，需要将他们添加到暂存区（`git add`）,然后将暂存区的内容提交到本地仓库中(`git commit`)。在版本控制系统中,这个过程叫做提交。

文件状态:

- 未跟踪（untrack）：就是我们创建的，还没有被Git管理起来的文件。
- 未修改（unmodified）：已经被Git管理起来，但是文件内容还没有发生变化，没有被修改过。
- 已修改（modified）：已修改的文件，还未添加到暂存区的文件
- 已暂存（staged）：已修改，并已添加至暂存区

添加和提交文件

几个命令:

```
git init 创建仓库
```

```
git add  添加到暂存区
还可使用通配符
git add *.txt 添加所有txt文件
git add . 添加所有文件
```

```
git status 查看仓库状态
```

```
git commit -m ""   提交
-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样就能从历史记录里方便地找到改动记录。
如果忘记-m，在重新输入提交说明后，esc退出，:wq保存即可。
```

```
git log  查看仓库提交历史记录
可以使用 git log --oneline来查看简洁的提交记录
```

git reset 回退版本

```
git reset --soft
回退版本，并且保留回退前版本工作区和暂存区的内容
```

```
git reset --hard
回退版本，并丢弃回退前版本工作区和暂存区的内容
```

```
git reset --mixed  (默认)
回退版本，保留回退前版本工作区的内容，丢弃回退前版本暂存区的内容
```

hard参数使用应当谨慎，但误操作也可以通过`git reflog`查看版本号，再用`git reset --hard (使用hard前的版本号)`来回溯。

小结

现在总结一下：

- `HEAD`指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令`git reset --hard commit_id`。
- 穿梭前，用`git log`可以查看提交历史，以便确定要回退到哪个版本。
- 要重返未来，用`git reflog`查看命令历史，以便确定要回到未来的哪个版本。

git diff 查看差异

可以查看工作区、暂存区、本地仓库之间的差异。也可以查看不同版本之间的差异，还可以查看不同分支之间的差异

```
git diff
不加参数默认查看工作区和暂存区之间的差异，会显示更改的文件和更改的详细信息
```

```
git diff <file>
工作区与暂存区file文件之间的差异
```

```
git diff HEAD <file>
工作区与版本库
```

```
git diff --cached <file>
暂存区和版本库
```

```
git diff <版本号> <版本号> <file>
不同版本之间的差异
通过git log来获取版本号
HEAD表示当前版本
HEAD^或HEAD~表示上一个版本
HEAD~2 表示当前版本上上个版本
```

```
git diff <branch_name> <branch_name>
分支之间的差异
```

删除文件

```
rm <file>;git add file
先从工作区删除文件，再删除暂存内容
这个rm是linux中的命令,所以需要手动删除暂存区中的内容
```

```
git rm <file>
把文件从工作区和暂存区同时删除
```

```
git rm --cached <file>
把文件从暂存区删除，但保留在当前工作区中
```

```
git rm -r*
递归删除某个目录下所有的子目录和文件
```

在删除后，要记得commit

此外，还可以用版本库里的版本替换工作区的版本：

```
git checkout
将工作区里误操作的版本恢复到最新提交的版本
```

命令`git checkout -- readme.txt`意思就是，把`readme.txt`文件在工作区的修改全部撤销，这里有两种情况：

一种是`readme.txt`自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是`readme.txt`已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

远程仓库

ssh配置

```
ssh-keygen -t rsa -b 4096
生成ssh密钥
私钥文件：id_rsa
公钥文件:id_rsa.pub
```

将公钥文件复制到github进行配置即可

克隆仓库

`git clone <address>` 克隆仓库

`git push <remote> <branch>` 推送更新内容,本地>远程

`git pull <remote>` 拉取更新内容,远程>本地

关联本地仓库和远程仓库

关联远程仓库

- step1:

```
将一个已有的本地仓库与远程仓库关联
git remote add origin <远程仓库的地址>
origin是git对远程仓库的默认叫法
```

- step2:

```
把本地库的所有内容推送到远程库上
git push -u origin master
由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的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：
git push origin master
```

查看远程仓库

```
git remote -v
```

拉取远程仓库内容

```
git pull <远程仓库名(origin)> <远程分支名>:<本地分支名>
如果远程分支名和本地分支名相同,可以省略冒号后的部分
```

删除远程库

```
git remote -v  先查看远程库信息
git remote rm <name>
解除本地和远程的绑定关系,并不是直接删除远程库
```

分支

简介

分支在实际中有什么用呢？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。

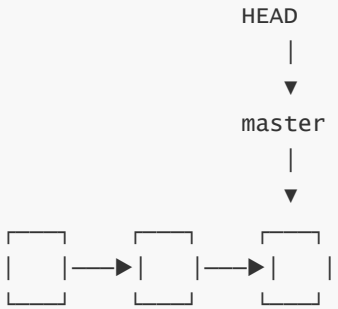
现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

创建与合并分支

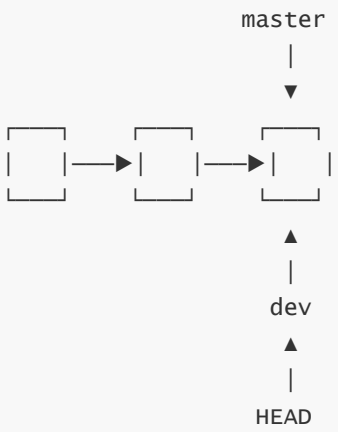
过程

- 在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即`master`分支。`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。

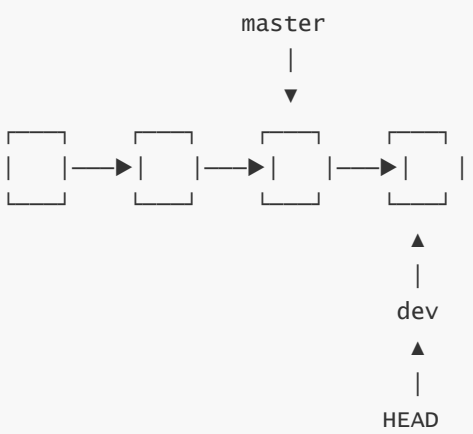
  一开始的时候，`master`分支是一条线，Git用`master`指向最新的提交，再用`HEAD`指向`master`，就能确定当前分支，以及当前分支的提交点。



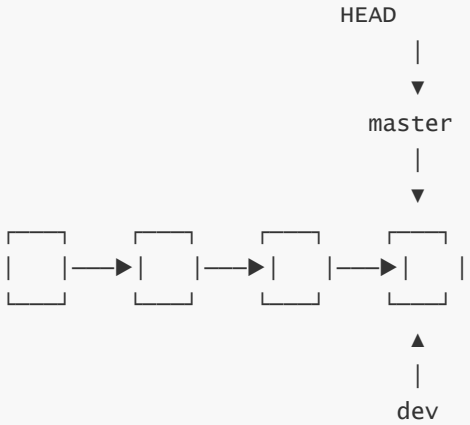
- 当我们创建新的分支，例如`dev`时，Git新建了一个指针叫`dev`，指向`master`相同的提交，再把`HEAD`指向`dev`，就表示当前分支在`dev`上。



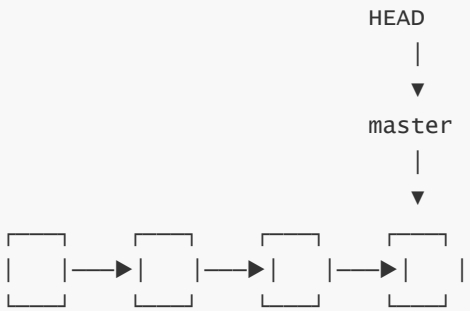
- 从现在开始，对工作区的修改和提交就是针对`dev`分支了，比如新提交一次后，`dev`指针往前移动一步，而`master`指针不变：



- 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：



代码

- 创建&切换

```
git checkout -b dev
加-b,-c表示创建并切换到dev分支。相当于如下两条命令
git branch dev
git checkout dev
```

- 查看当前有的分支

```
git branch
当前分支前有*号
```

- 合并分支

```
git merge dev
将dev分支合并到当前分支上
```

使用后，会出现以下提示:

```
Updating d46f35e..b17d20e
Fast-forward
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

注意到上面的`Fast-forward`信息，Git告诉我们，这次合并是“快进模式”，也就是直接把`master`指向`dev`的当前提交，所以合并速度非常快。

当然，也不是每次合并都能`Fast-forward`，我们后面会讲其他方式的合并。

- 删除分支

```
git branch -d dev
```

因为创建、合并和删除分支非常快，所以Git鼓励你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。

switch:

我们注意到切换分支使用`git checkout <branch>`，而前面讲过的撤销修改则是`git checkout -- <file>`，同一个命令，有两种作用。

实际上，切换分支这个动作，用`switch`更科学。因此，最新版本的Git提供了新的`git switch`命令来切换分支：

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
```

小结

Git鼓励大量使用分支：

查看分支：`git branch`

创建分支：`git branch <name>`

切换分支：`git checkout <name>`或者`git switch <name>`

创建+切换分支：`git checkout -b <name>`或者`git switch -c <name>`

合并某分支到当前分支：`git merge <name>`

删除分支：`git branch -d <name>`

解决合并冲突

在不同分支中对同一文件进行修改，并且修改内容不同。这时如果尝试合并两个分支，就会发生冲突。

git会告诉我们冲突的文件，我们也可以使用`git status`查看冲突文件

举例

准备新的`feature1`分支，继续我们的新分支开发：

```
$ git switch -c feature1
Switched to a new branch 'feature1'
```

修改`readme.txt`最后一行，改为：

```
Creating a new branch is quick AND simple.
```

在`feature1`分支上提交：

```
$ git add readme.txt

$ git commit -m "AND simple"
[feature1 14096d0] AND simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

切换到`master`分支：

```
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

Git还会自动提示我们当前`master`分支比远程的`master`分支要超前1个提交。

在`master`分支上把`readme.txt`文件的最后一行改为：

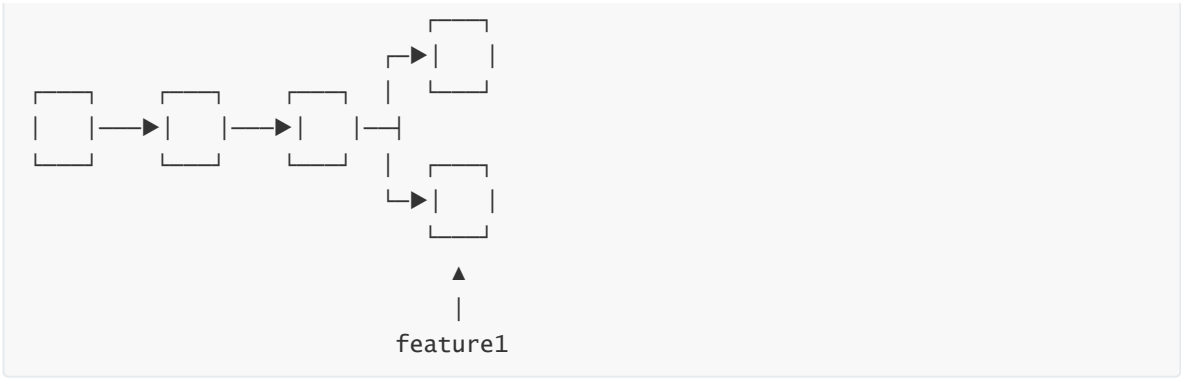
```
Creating a new branch is quick & simple.
```

提交：

```
$ git add readme.txt
$ git commit -m "& simple"
[master 5dc6824] & simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

现在，`master`分支和`feature1`分支各自都分别有新的提交，变成了这样：

```
                          HEAD
                           |
                           ▼
                        master
                           |
                           ▼
```



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：

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

果然冲突了！Git告诉我们，`readme.txt`文件存在冲突，必须手动解决冲突后再提交。`git status`也可以告诉我们冲突的文件：

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   readme.txt

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

我们可以直接查看readme.txt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
```

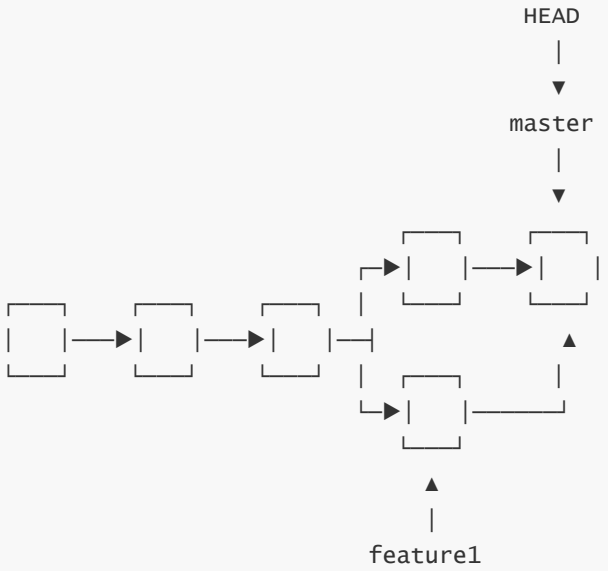
Git用`<<<<<<<`，`=======`，`>>>>>>>`标记出不同分支的内容，我们修改如下后保存：

```
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.
Creating a new branch is quick and simple.
```

再提交：

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

现在，`master`分支和`feature1`分支变成了下图所示：



用带参数的`git log`也可以看到分支的合并情况：

```
$ git log --graph --pretty=oneline --abbrev-commit
*   cf810e4 (HEAD -> master) conflict fixed
|\
| * 14096d0 (feature1) AND simple
* | 5dc6824 & simple
|/
* b17d20e branch test
* d46f35e (origin/master) remove test.txt
* b84166e add test.txt
* 519219b git tracks changes
* e43a48b understand how stage works
* 1094adb append GPL
* e475afc add distributed
* eaadf4e wrote a readme file
```

最后，删除`feature1`分支：

```
$ git branch -d feature1
Deleted branch feature1 (was 14096d0).
```

工作完成。

小结

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

用`git log --graph`命令可以看到分支合并图。

分支管理策略

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。

下面我们实战一下`--no-ff`方式的`git merge`：

首先，仍然创建并切换`dev`分支：

```
$ git switch -c dev
Switched to a new branch 'dev'
```

修改readme.txt文件，并提交一个新的commit：

```
$ git add readme.txt
$ git commit -m "add merge"
[dev f52c633] add merge
 1 file changed, 1 insertion(+)
```

现在，我们切换回`master`：

```
$ git switch master
Switched to branch 'master'
```

准备合并`dev`分支，请注意`--no-ff`参数，表示禁用`Fast forward`：

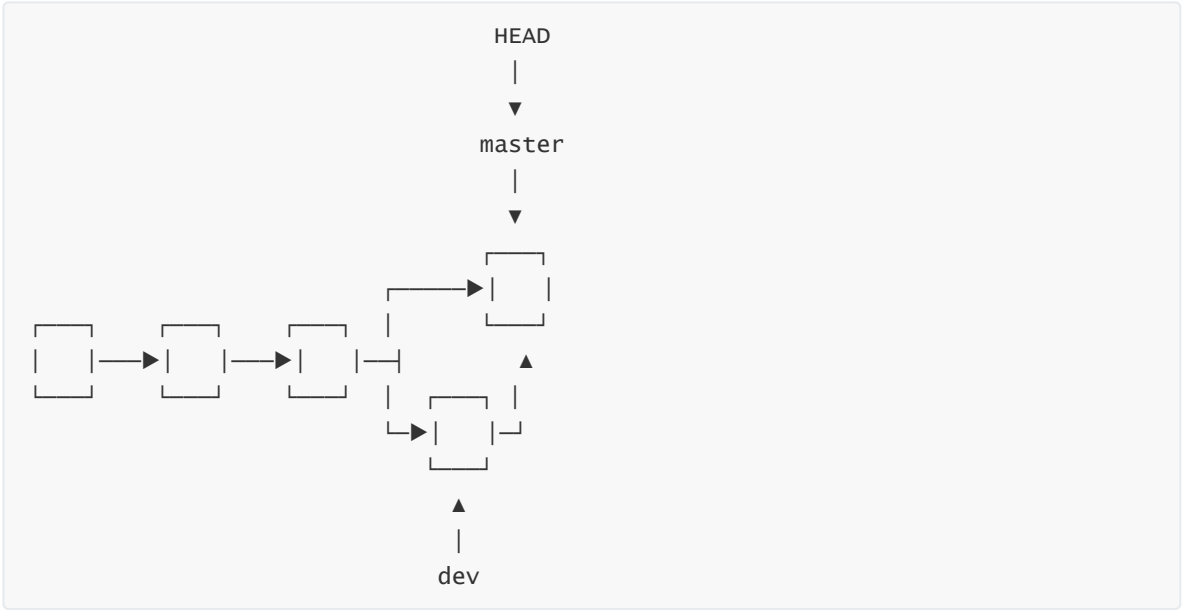
```
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

因为本次合并要创建一个新的commit，所以加上`-m`参数，把commit描述写进去。

合并后，我们用`git log`看看分支历史：

```
$ git log --graph --pretty=oneline --abbrev-commit
*   e1e9c68 (HEAD -> master) merge with no-ff
|\
| * f52c633 (dev) add merge
|/
*   cf810e4 conflict fixed
...
```

可以看到，不使用`Fast forward`模式，merge后就像这样：



分支策略

在实际开发中，我们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；

那在哪干活呢？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；

你和你的小伙伴们每个人都在`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。

小结

Git分支十分强大，在团队开发中应该充分应用。

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。

Bug分支

在git中，可以通过新建一个分支修复bug，但此时正在dev上进行的工作还没有提交

Git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
```

使用后，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。

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，就从`master`创建临时分支：

```
git checkout master
git checkout -b issue-101
```

修复ing....

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
```

现在回到dev继续工作

```
git switch dev
```

但是现在工作区没有内容,用`git stash list`查看。恢复：

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：

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。

同样的bug，要在`dev`上修复，我们只需要把`4c805e2 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`4c805e2 fix bug 101`这个提交所做的修改，并不是把整个`master`分支merge过来。

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：

```
git cherry-pick 4c805e2
```

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`4c805e2`，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。

小结

修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；

当手头工作没有完成时，先把工作现场`git stash`一下，然后去修复bug，修复后，再`git stash pop`，回到工作现场；

在`master`分支上修复的bug，想要合并到当前`dev`分支，可以用`git cherry-pick <commit版本号>`命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

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。

于是：

```
git switch -c feature
开发ing...
git add a.txt
git commit -m "add feature"
切回dev，准备合并：
git switch dev
```

但此时，需要取消新功能并删除这个feature分支

```
git branch -d feature
```

销毁失败。Git友情提醒，`feature-vulcan`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。

```
git branch -D feature
```

小结

开发一个新功能，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。

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`：

```
$ git remote
origin
```

或者，用`git remote -v`显示更详细的信息：

```
$ git remote -v
origin  git@github.com:michaelliao/learngit.git (fetch)
origin  git@github.com:michaelliao/learngit.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。

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
```

如果要推送其他分支，比如`dev`，就改成：

```
$ git push origin dev
```

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。

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，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

```
$ git clone git@github.com:michaelliao/learngit.git
Cloning into 'learngit'...
remote: Counting objects: 40, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 40 (delta 14), reused 40 (delta 14), pack-reused 0
Receiving objects: 100% (40/40), done.
Resolving deltas: 100% (14/14), done.
```

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的`master`分支。不信可以用`git branch`命令看看：

```
$ git branch
* master
```

现在，你的小伙伴要在`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
```

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：

```
$ git add env.txt

$ git commit -m "add env"
[dev 7a5e5dd] add env
 1 file changed, 1 insertion(+)
 create mode 100644 env.txt

$ git push origin dev
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 308 bytes | 308.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:michaelliao/learngit.git
```

```
   f52c633..7a5e5dd  dev -> dev
```

你的小伙伴已经向`origin/dev`分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

```
$ cat env.txt
env

$ git add env.txt

$ git commit -m "add new env"
[dev 7bd91f1] add new env
 1 file changed, 1 insertion(+)
 create mode 100644 env.txt

$ git push origin dev
To github.com:michaelliao/learngit.git
 ! [rejected]        dev -> dev (non-fast-forward)
error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，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
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
```

`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
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

再pull：

```
$ git pull
Auto-merging env.txt
CONFLICT (add/add): Merge conflict in env.txt
Automatic merge failed; fix conflicts and then commit the result.
```

这回`git pull`成功，但是合并有冲突，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 commit -m "fix env conflict"
[dev 57c53ab] fix env conflict

$ git push origin dev
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 621 bytes | 621.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To github.com:michaelliao/learngit.git
   7a5e5dd..57c53ab  dev -> dev
```

因此，多人协作的工作模式通常是这样：

1. 首先，可以尝试用`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。

小结

- 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 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。

Rebase(变基)

作用：将不同分支的修改内容合并到一起

作用原理

每个分支都有一个指针，指向当前分支的最新提交记录
在执行rebase时，找到两个分支的最近共同祖先，将当前分支在祖先后的内容移到目标分支的最新提交后面

与merge的区别

merge:

优点：不会破坏原分支的提交历史，方便回溯和查看

缺点：会产生额外的提交节点，分支图比较复杂

rebase:

优点：形成线性历史，直观、干净

缺点：会改变提交历史，改变了当前分支branch out的节点。避免在共享分支使用

标签管理

简介

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag?

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

如果换一个办法：

“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

所以，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

切换到分支并：

```
git tag <name>
```

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办?

方法是找到历史提交的commit id，然后打上就可以了：

```
git log --pretty=oneline --abbrev-commit
12a631b (HEAD -> master, tag: v1.0, origin/master) merged bug fix 101
4c805e2 fix bug 101
e1e9c68 merge with no-ff
f52c633 add merge
cf810e4 conflict fixed
5dc6824 & simple
14096d0 AND simple
b17d20e branch test
d46f35e remove test.txt
b84166e add test.txt
519219b git tracks changes
```

```
e43a48b understand how stage works
1094adb append GPL
e475afc add distributed
eaadf4e wrote a readme file
```

比方说要对`add merge`这次提交打标签，它对应的commit id是`f52c633`，敲入命令：

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

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
```

用命令`git show <tagname>`可以看到说明文字

```
git show v0.1
```

注意

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，那么在这两个分支上都可以看到这个标签。

小结

- 命令`git tag <tagname>`用于新建一个标签，默认为`HEAD`，也可以指定一个commit id；
- 命令`git tag -a <tagname> -m "blablabla..."`可以指定标签信息；
- 命令`git tag`可以查看所有标签。

操作标签

如果标签打错了，也可以删除：

```
$ git tag -d v0.1
```

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。

如果要推送某个标签到远程，使用命令

```
git push origin v1.0
```

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
```

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。删除命令也是push，但是格式如下：

```
$ git push origin :refs/tags/v0.9
```

小结

- 命令`git push origin <tagname>`可以推送一个本地标签；
- 命令`git push origin --tags`可以推送全部未推送过的本地标签；
- 命令`git tag -d <tagname>`可以删除一个本地标签；
- 命令`git push origin :refs/tags/<tagname>`可以删除一个远程标签。

自定义Git

.gitingore忽略文件

应该忽略的文件

- 系统或者软件自动生成的文件
- 编译产生的中间文件和结果文件，比如.class文件
- 运行时生成的日志文件、缓存文件、临时文件
- 涉及身份、密码等敏感信息的文件

代码演示

```
echo access.log>.gitignore
忽略access.log配置文件
```

```
temp/
在.gitignore文件中写入temp/表示忽略temp文件夹
```

```
git add -f <file>
文件被忽略但是强制添加
```

在.gitignore文件中：

```
# 排除所有.开头的隐藏文件:
.*
# 排除所有.class文件:
*.class

# 不排除.gitignore和App.class:
!.gitignore
!App.class
```

!+文件名表示.gitignore的忽略作用对此文件不生效。

注意:

.gitignore放在哪个目录下，就对哪个目录包括其子目录起作用

.gitignore文件也要提交到版本库

.gitignore中的匹配规则

- 空行或以#开头的行会被Git忽略。一般空行用于可读性的分隔，#一般用于注释

- 使用标准的Blob模式。例如：
  - *通配任意个字符
  - ？通配单个字符
  - [ ]表示匹配列表中的单个字符,[abc]表示a/b/c
  - [0-9]表示任意一位数字,[a-z]表示任意一位小写字母
  - **表示匹配任意的中间目录

    (比如`doc/**/*.pdf`表示doc/目录及其所有子目录下的.pdf文件)
  - !取反

最后，github上有很多常用语言的忽略模板，可以自行取用、修改。

配置别名

```
git config --global alias.st status
```

这样`git st`就有`git status`的效果

很多人都用`co`表示`checkout`，`ci`表示`commit`，`br`表示`branch`

`--global`参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

配置文件

配置Git的时候，加上`--global`是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在`.git/config`文件中：

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = git@github.com:michaelliao/learngit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[alias]
    last = log -1
```

别名就在`[alias]`后面，要删除别名，直接把对应的行删掉即可。

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件`.gitconfig`中：

```
$ cat .gitconfig
[alias]
    co = checkout
    ci = commit
    br = branch
    st = status
[user]
    name = Your Name
    email = your@email.com
```

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置，或者直接删掉配置文件错误的那一行。