廖雪峰git入门教程

地址：[简介 - Git教程 - 廖雪峰的官方网站]

学习笔记：

1 git是什么

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

> 记录改动版本+协同编辑 = 版本控制

Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！

你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

太牛了，截个屏纪念、瞻仰、膜拜一下！（叩首）

git的开发语言：C，C++，PHP，Java，Python

集中式版本控制系统	分布式版本控制系统
联网，慢	强大的分支管理
中央服务器	每个人的电脑上都是一个完整的版本库；充当“中央服务器”的电脑
SVN	Git

2 安装Git

Git可以安装的操作系统：Linux，Mac，Windows，Raspberry Pi

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

`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。

3 创建版本库

版本库又名仓库（Repository）：里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪

所有的版本控制系统，其实只能跟踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。图片、视频、word这些二进制文件，只能知道改动的大小变化，具体改动了啥不清楚。

因为文本是有编码的，比如中文有常用的GBK编码，日文有Shift_JIS编码，如果没有历史遗留问题，强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

使用Windows的童鞋要特别注意：千万不要使用Windows自带的**记事本**编辑任何文本文件。原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Visual Studio Code代替记事本，不但功能强大，而且免费！

```
# cmd命令
E:   # 切换磁盘
dir  # 查看文件夹中文件，相当于bash的ls
dir /?  # dir命令使用帮助
echo > readme.txt  # 创建空文件。如果文件已存在，它会覆盖该文件。如果 echo 没有指定内容，Windows 会将 echo 命令的输出（即 echo is on）写入文件，echo 默认会输出当前的开关状态。
type nul > readme.txt  # 创建空文件。如果文件已存在，它不会覆盖该文件。
start readme.txt  # 打开文件，默认用关联的程序打开
start notepad++ readme.txt  # 用指定的程序打开指定文件
```

```
git config --global user.name "Your Name"  # 指定仓库的用户名。--global参数表示这台机器上所有的Git仓库都会使用这个配置。
```

```
git config --global user.email "email@example.com"  # 指定仓库的Email地址
git init  # 新建仓库。多了个.git的目录，用来跟踪管理版本库
git add readme.txt  # 把文件添加到仓库
git commit -m "wrote a readme file"  # 把文件提交到仓库。-m后面输入的是本次提交的说明，可以输入任意内容。
```

`git commit`命令执行成功后会告诉你，`1 file changed`：1个文件被改动（我们新添加的readme.txt文件）；`2 insertions`：插入了两行内容（readme.txt有两行内容）。

为什么Git添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：

```
$ git add file1.txt
$ git add file2.txt file3.txt
$ git commit -m "add 3 files."
```

注意：

Git命令必须在Git仓库目录内执行（git init除外），在仓库目录外执行是没有意义的。

添加某个文件时，该文件必须在当前目录下存在，用`ls`或者`dir`命令查看当前目录的文件，看看文件是否存在，或者是否写错了文件名。

4 时光穿梭机

4.1 版本回退

```
git status  # 查看文件状态，是否有过修改、添加、提交等
git diff readme.txt  # 查看文件具体修改内容
git log  # 显示由近到远的提交日志
git log --pretty=oneline  # 简洁版日志
git reset --hard "HEAD^"  # 回退到上一个版本(要加双引号！)
git reflog  # 记录每一次命令，可用于找回文件
```

`commit id`（版本号）：16进制，由SHA1计算出来

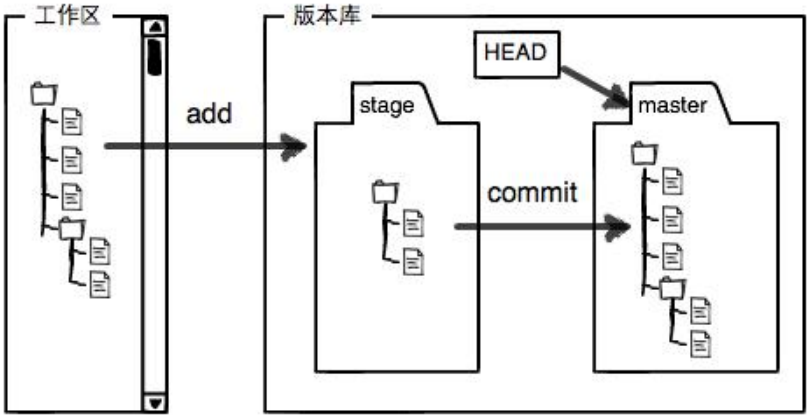
在Git中，用`HEAD`表示当前版本。上一个版本就是`HEAD^`，上上一个版本就是`HEAD^^`，当然往上100个版本写100个^比较容易数不过来，所以写成`HEAD~100`。

`--hard` 会回退到上个版本的已提交状态
`--soft` 会回退到上个版本的未提交状态
`--mixed` 会回退到上个版本未添加状态

4.2 工作区和暂存区

工作区：电脑里能看到的文件目录
暂存区：隐藏目录 `.git`，为Git的版本库

Git的版本库里存了很多东西，其中最重要的就是称为 `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的一个指针叫 `HEAD`。



`git add`：把文件修改从工作区添加到暂存区
`git commit`：把暂存区全部内体添加到分支

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。

【例】：

```
E:\git>start readme.txt

E:\git>type nul > LICENSE.txt

E:\git>git status
On branch master
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.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        LICENSE.txt

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

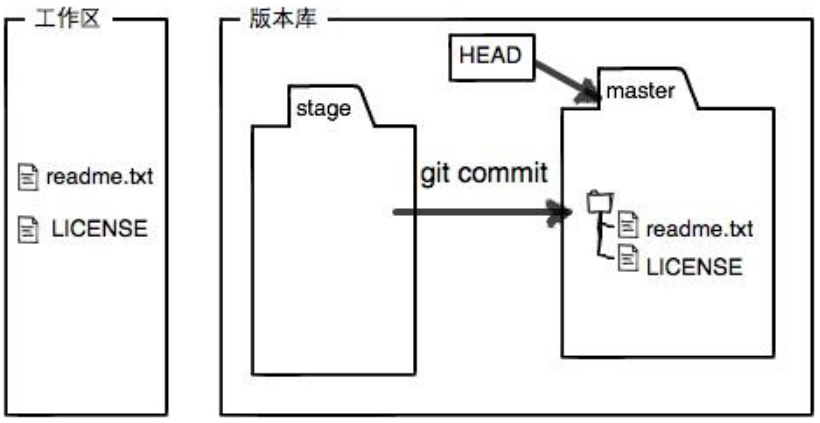
`Untracked`：未被添加过

```
E:\git>git add readme.txt LICENSE.txt

E:\git>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   LICENSE.txt
        modified:   readme.txt
```

```
E:\git>git commit -m "understand how stage works"
[master f88a6fc] understand how stage works
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 LICENSE.txt

E:\git>git status
On branch master
nothing to commit, working tree clean
```



4.3 管理修改

Git跟踪并管理的是修改，而非文件。

```
git diff HEAD -- readme.txt  # 查看工作区和版本库里面最新版本的区别
```

第一次修改 -> `git add` -> 第二次修改 -> `git commit`
此时只有第一次修改会被提交。

4.4 撤销修改

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。

```
git checkout -- readme.txt  # 把readme.txt文件在工作区的修改全部撤销
```

注意：`git checkout -- file`命令中的`--`很重要，没有`--`，就变成了“切换到另一个分支”的命令。

```
git reset HEAD readme.txt  # 既可以回退版本，也可以把暂存区的修改回退到工作区
```

若已经提交到版本库，那就用版本回退，前提是还没有把自己的本地版本库推送到远程。

【例】：

```
$ start readme.txt

$ git add readme.txt

$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.txt


$ git reset HEAD readme.txt
Unstaged changes after reset:
M       readme.txt

$ git status
On branch master
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.txt

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

$ git checkout -- readme.txt

$ git status
On branch master
nothing to commit, working tree clean
```

4.5 删除文件

【例】：

```
$ echo hello world > test.txt  # 新建文件，并写入hello world

$ start test.txt

$ git add test.txt

$ git commit -m "add test.txt"
```

```
[master f8e7512] add test.txt
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt

$ del test.txt  # 若删除文件（工作区）

$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    test.txt

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

$ git rm test.txt  # 在版本库中删除文件
rm 'test.txt'

$ git commit -m "remove test.txt"  # 提交删除
[master f40f4eb] remove test.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 test.txt
```

`git checkout`其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

```
# 若误删文件后还想恢复文件
$ git checkout -- test.txt
```

注意：

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、会丢失最近一次提交后你修改的内容

6 远程仓库

Github：提供Git仓库托管服务

本地Git仓库和GitHub仓库之间的传输是通过SSH加密

```
# 创建SSH Key
ssh-keygen -t rsa -C "youremail@example.com"
```

创建完后可以在用户主目录里找到`.ssh`目录，里面有`id_rsa`和`id_rsa.pub`两个文件，这两个就是SSH Key的秘钥对，`id_rsa`是私钥，`id_rsa.pub`是公钥。

为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。

如果你不想让别人看到Git库：
1、交钱，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。
2、自己搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

6.1 添加远程库

让本地GIt库和Github上的Git库远程同步，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作。

```
git remote add origin git@github.com:<your_username>/<repo_name>.git
git remote add origin https://github.com/<your_username>/<repo_name>.git  # 或
git branch -M main
git push -u origin main
```

添加后，远程库的名字就是`origin`，这是Git默认的叫法，也可以改成别的，但是`origin`这个名字一看就知道是远程库。

把本地库的内容推送到远程，用`git push`命令，实际上是把当前分支`master`推送到远程。

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。

从现在起，只要本地作了提交，就可以通过命令：

```
$ git push origin master
```

把本地`master`分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

SSH警告

当你第一次使用Git的 `clone` 或者 `push` 命令连接GitHub时，会得到一个警告：

```
The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.
RSA key fingerprint is xx.xx.xx.xx.xx.
Are you sure you want to continue connecting (yes/no)?
```

这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入 `yes` 回车即可。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

```
Warning: Permanently added 'github.com' (RSA) to the list of known hosts.
```

这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入 `yes` 前可以对照GitHub的RSA Key的指纹信息是否与SSH连接给出的一致。

遇到问题：

```
$ git push -u origin master
error: src refspec master does not match any
```

这通常意味着你的本地分支名称与远程仓库中的分支名称不匹配，或者远程仓库中没有找到名为 master 的分支。

你的本地库名字叫learngit，远程库全名叫git@github.com:michaelliao/learngit.git

但每次推送让你敲个全名你会疯的，所以起个别名origin

每次敲命令git push origin master的时候，git看到origin，就去当前库的配置文件里找，看看它的全名到底写的啥

你也可以起别的名字，如果有多个远程库，那必然不同的远程库对应不同的名字

```
git remote rm origin # 删除关联的origin的远程库
git remote add origin https://gitee.com/xxxxxx.git  # 关联自己的仓库
git push origin master  # 推送到自己的仓库
```

运行完了刷新一下github就出来了！

删除远程库

```
git remote -v  # 查看远程库信息
```

```
git remote rm origin  # 删除远程库(解绑)
```

6.2 从远程库克隆

```
git clone git@github.com:<your_username>/<repo_name>.git
```

GitHub给出的地址不止一个，还可以用`https://github.com/michaelliao/gitskills.git`这样的地址。实际上，Git支持多种协议，默认的`git://`使用`ssh`，但也可以使用`https`等其他协议。

使用`https`除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放`http`端口的公司内部就无法使用`ssh`协议而只能用`https`。

小结

有些混乱了，整理一下学过的内容，并在Github上试验一下。

删除文件

网页版Github上删除文件有两种办法，一是在右上角的图标里点击删除，二是将建好的仓库克隆到本地，在本地删除后同步到Github。

```
# 法二删除Gitbub文件
git clone git@github.com:<your_username>/<repo_name>.git
del <file>  # 工作区删除文件
git rm <file>  # 版本库中删除文件
git commit -m "<注释>"
git push origin main
```

添加文件

网页版Github上添加文件有两种办法，一是在右上角的`Add file`点击添加文件，二是将建好的仓库克隆到本地，在本地添加后同步到Github，三是在本地仓库添加文件后将仓库添加到Github。

```
# 法二添加Github文件
git clone git@github.com:<your_username>/<repo_name>.git
```

```
# 在仓库中新建文件夹或放入文件（放入到工作区，添加到暂存区）
cd <dir>  # 若新建文件夹则需移动到该文件夹下再添加文件
git add <file>
git commit -m "<注释>"
git status
git push origin main
```

修改文件

在本地仓库修改文件后同步到Github。

```
start <file>  # 打开文件，修改文件
git add <file>
git commit -m "<注释>"
git status
git remote -v
git push origin main
```

查看或撤销修改

```
# 查看修改
git diff HEAD -- <file>  # 查看工作区和版本库里面最新版本的区别
```

```
# 撤销修改

# 在工作区修改后未添加到暂存区
git status
git checkout -- <file>

# 修改添加到暂存区但未提交
git status
git reset HEAD <file>
git checkout -- <file>
```

版本回退

想要回到上一个版本。

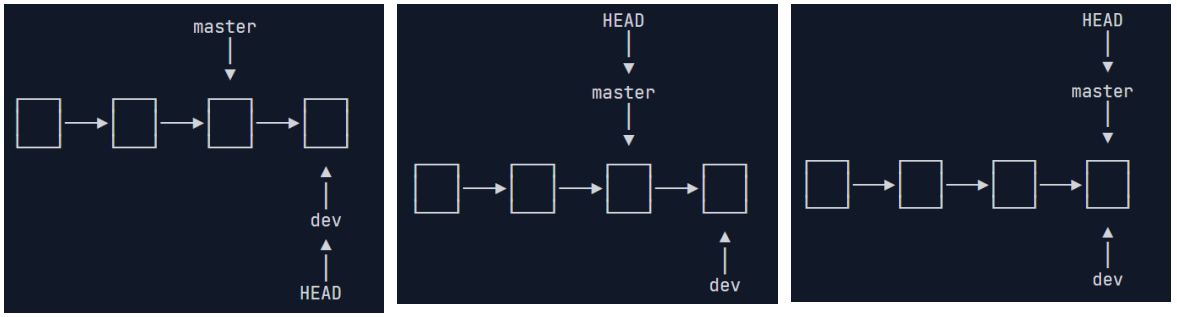
```
git log
git log --pretty=oneline
git reset --hard "HEAD^"
```

```
# 若又后悔了，不想回退了，趁命令窗口未关
git reset --hard <想要回到版本的版本号，可以只写前几位>
# 若窗口已关
git reflog  # 从历史命令记录中找到想要的版本号
```

---

7 分支管理

7.1 创建与合并分支




```
git checkout -b dev  # 加上-b，创建并切换到dev分支
# 等价于下面两条命令
git branch dev
git checkout dev

git branch  # 查看当前分支，加*的为当前分支

# 在dev分支上提交
start readme.txt
git add readme.txt
git commit -m "branch test"

git checkout master  # 切换为master分支
git merge dev  # 把dev分支的工作成果合并到master分支上
git branch -d dev  # 合并完后删除dev分支
git branch  # 查看分支，只剩master了
```

`Fast-forward`：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。但也不是每次都是这种方式。

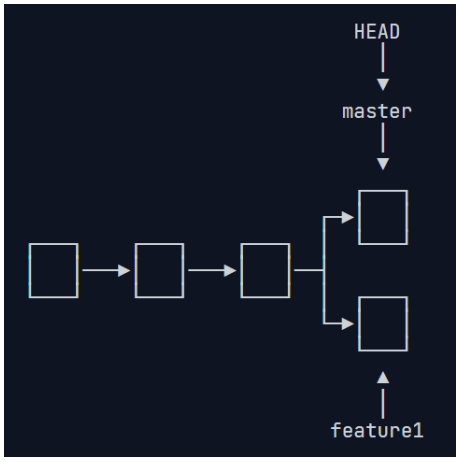
注意：

```
git checkout <branch>  # 切换分支
git checkout -- <file>  # 撤销修改
```

切换分支也可以用`git switch`命令。

```
git switch -c dev  # 创建并切换到新的dev分支
git switch master  # 直接切换到已有的master分支
```

7.2 解决冲突



此时无法快速合并，`git merge feature1`失败。

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

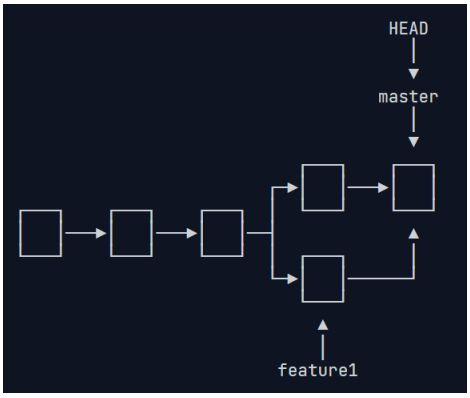
`git status`也可以告诉我们冲突的文件。

可以直接查看readme.txt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
```

若在`master`分支上修改错了，变成如下情况：



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
```

删除分支。`git branch -d feature1`

总结：

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。

```
# 此时可用命令
git merge feature1  # 合并分支
git status  # 查看冲突文件
start readme.txt  # 查看冲突内容
git log --graph --pretty=oneline --abbrev-commit  # 查看分支合并情况
git branch -d feature1  # 删除分支
```

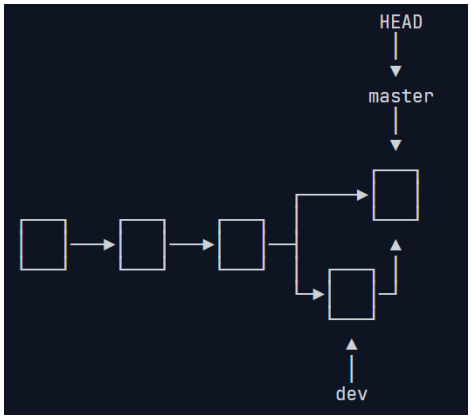
7.3 分支管理策略

通常合并分支时Git用`Fast Forward`模式，但这样删除分支后会丢失一些信息。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。

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

`--no-ff`参数，表示禁用`Fast forward`。`-m`参数，表示把commit描述写进去。

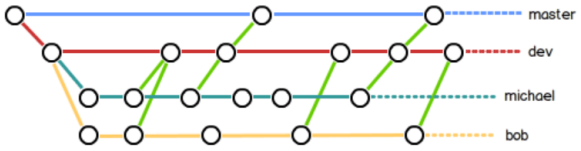
不使用`Fast forward`模式，merge后就像这样：



总结：

```
git merge dev  # Fast Forward模式合并
git merge --no-ff -m "merge with no-ff" dev  # 普通模式合并
```

`master`分支仅用来发布新版本，平时在`dev`分支上干活，干活时每个人都有自己的分支，把自己的分支往`dev`上合并就行了。



7.4 Bug分支

```
$ git stash  # 把当前工作现场“储藏”起来，等以后恢复现场后继续工作
Saved working directory and index state WIP on dev: f52c633 add merge
```

```
git checkout master  # 从master创建临时分支
git checkout -b issue-101
```

```
git add readme.txt
git commit -m "fix bug 101"
git switch master
git merge --no-ff -m "merged bug fix 101" issue-101
git switch dev
git status
git stash list  # 查看

git stash apply  # 恢复后，stash内容并不删除，需要用git stash drop来删除
git stash pop  # 恢复的同时把stash内容也删了

git stash apply stash@{0}  # 恢复指定的stash
```

7.5 Feature分支

7.6 多人协作

7.7 Rebase

8 标签管理

8.1 创建标签

```
# 切换到需要打标签的分支上
git branch
git checkout master

git tag v1.0  # 打标签
git tag  # 查看所有标签

git log --pretty=oneline --abbrev-commit  # 找到历史提交的commit id
git tag v0.9 <commit_id>  # 打标签
git show v0.9 查看标签信息

# 创建带有说明的标签
# -a指定标签名，-m指定说明文字
git tag -a v0.1 -m "version 0.1 released" <commit_id>
```

```
git show <tagname>  # 可以看到说明文字
```

注意：

1、标签不是按时间顺序，而是按字母顺序排列的。
2、标签与说明挂钩，如果标签同时出现在master和dev分支，那么两个分支上都可以看到这个表签。

8.2 操作标签

```
git tag -d v0.1  # 删除标签
git push origin v0.1  # 推送标签到远程
git push origin --tags  # 一次性推送所有尚未推送的标签到远程

# 若标签已经推送到远程，想要删除
git tag -d v0.9  # 先从本地删除
git push origin :refs/tags/v0.9  # 再从远程删除
```

创建的标签只储存在本地，不会自动推送到远程。

登录Github查看是否真从远程库删除了标签。