**20180424**

* Git

Git是目前世界上最先进的分布式版本控制系统，免费、开源；

版本控制：记录一个或若干文件内容变化、以便将来查阅特定版本修订情况。

比如hikari用word写毕业论文，想删除某一段又怕删了找不到怎么办？当然是先复制一份，在新的word上继续修改…然后最终文件从ver1一直排到了ver20…

想恢复被删除的内容，又不知道在哪个文件了…

而Git的作用就是自动记录文件每次的改动，而且还可以让他人协作编辑。

Git作用：

① 记录文件的所有历史变化；

② 随时可恢复到任何一个历史状态；

③ 多人协作开发或修改；

④ 错误恢复。

* 集中式vs分布式

CVS及SVN都是集中式的版本控制系统，而Git是分布式版本控制系统

① 集中式版本控制系统：版本库集中存放在中央服务器；用自己的电脑先从中央服务器取得最新的版本，然后开始干活；结束后再把代码推送给中央服务器。



集中式版本控制系统最大的问题就是必须联网才能工作。局域网内还好，如果在互联网遇到网速慢的情况，那就干等着吧。

② 分布式版本控制系统：没有中央服务器，每人电脑上都是一个完整的版本库。这样工作就不需要联网，因为版本库就在自己的电脑上。当修改了文件，伙伴也修改了此文件，两人只需把各自修改推送给对方，就可以互相看到对方的修改。

分布式安全性要高很多。每个人电脑里都有完整的版本库，如果电脑挂了，随便从其他人那里复制一份就可以了；而集中式的中央服务器要是出了问题，所有人都没法干活了。



实际使用分布式版本控制系统时，很少在两人之间的电脑上推送版本库的修改。

通常也有一台充当中央服务器的电脑，但这个服务器的作用仅仅是用来方便交换大家的修改，没有它一样干，只是交换修改不方便而已。

* 安装Git

① Ubuntu：sudo apt-get install git

也可以通过源码安装。从Git官网下载源码，解压，依次输入：./config、make、sudo make install就可以安装。

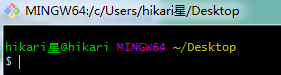
② windows：Git官网[下载](https://git-scm.com/download/win)，速度堪忧，移步[淘宝镜像](https://npm.taobao.org/mirrors/git-for-windows/)。

选择安装目录后，一路确定

桌面右键看到Git GUI和Git Bash说明安装成功！



点击Git Bash：



太丑了…

|  |
| --- |
| $ git config --global user.name "my name"  $ git config --global user.email "email@example.com" |

Git是分布式版本控制系统，每台机器都必须自报家门：name和email。

git config命令的--global参数表示全局，使这台机器上所有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name和email。

* 创建版本库

版本库又名仓库(repository)，可以简单理解为一个目录，目录里面所有文件都可以被Git管理，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻还原。

1. 创建版本库：

① 选择一个合适的地方，创建空目录：

|  |
| --- |
| $ mkdir hikari\_git  $ cd hikari\_git  $ pwd  /d/hikari\_git |

对于Windows系统，为了避免各种奇怪的问题，目录不要有中文!

② 通过git init命令将此目录变成Git可以管理的仓库：

|  |
| --- |
| $ git init  Initialized empty Git repository in D:/hikari\_git/.git/ |

Git将空仓库建立好了，而且当前目录下多了一个.git的隐藏目录(用命令ls -ah可以看见)，此目录是Git用来跟踪管理版本库的，没事千万不要手动修改里面的文件，否则不小心就把Git仓库破坏了。

2. 将文件添加到版本库

所有版本控制系统只能跟踪文本文件的改动，比如txt文件、网页、程序代码等；而图片、视频等二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只知道图片大小变化，但修改了什么版本控制系统不知道，也无法知道。

不幸的是，Word是二进制格式，版本控制系统无法跟踪Word文件改动。所以要真正使用版本控制系统，就要以纯文本方式编写文件。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。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。

千万不要使用Windows自带的~~记事本~~编辑任何文本文件。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。这产生了很多不可思议的问题：比如网页第一行可能会显示一个?、程序一编译就报语法错误等。

① 创建文本文件，如hello.py：

|  |
| --- |
| def **hello**():  **print**('hello world!') |

② git add <file>：将文件添加到暂存区

|  |
| --- |
| $ git add hello.py |

没有任何消息说明添加成功。Unix的哲学是没有消息就是好消息。

添加多个文件：

git add -A：提交所有变化

git add .：提交新文件、被修改的文件，不包括被删除的文件

③ git commit -m "message"：将文件从暂存区提交到仓库

|  |
| --- |
| $ git commit -m "add hello.py"  [master (root-commit) aa9f146] add hello.py  1 file changed, 2 insertions(+)  create mode 100644 hello.py |

-m后面是本次提交的说明，可以输入任意内容，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，方便自己和他人阅读。

为什么Git添加文件需要add、commit两步？因为commit可以一次提交很多文件，所以可以多次add不同文件一次性commit：

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

* 时光机穿梭

修改hello.py文件：

|  |
| --- |
| def **hello**(name):  **print**('hello {}!'.format(name)) |

可以使用git status查看仓库状态：

|  |
| --- |
| $ git status  On branch master  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.py  no changes added to commit (use "git add" and/or "git commit -a") |

git status命令可以时刻掌握仓库当前的状态，上面显示hello.py被修改过了，但还没有准备提交的修改。

git diff可以查看文件做了什么修改：

|  |
| --- |
| $ git diff hello.py  **diff --git a/hello.py b/hello.py**  **index 3b3f91b..4f6cc10 100644**  **--- a/hello.py**  **+++ b/hello.py**  @@ -1,2 +1,2 @@  -def hello():  - print('hello world!')  +def hello(name):  + print('hello {}!'.format(name)) |

git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式。

知道做了什么修改后，再把它提交到仓库就放心多了，步骤和前面一样。

|  |
| --- |
| $ git add hello.py  $ git status  On branch master  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  modified: hello.py  $ git commit -m "add change"  [master 1e6bd77] add change  1 file changed, 2 insertions(+), 2 deletions(-)  $ git status  On branch master  nothing to commit, working tree clean |

提交完之后，暂存区为空，也就是clean。

* 版本回退

hello.py再写一个hi()函数，add并commit。

每次文件修改到一定程度，就可以保存一个快照，commit就是这个这个快照。一旦文件改乱了或被误删，可以从最近的一个commit恢复。

git log命令显示从最近到最远的提交日志：

|  |
| --- |
| $ git log  commit 0a2f62e65c0b29106ee1eb8e8cc13f091b7c59e1 (HEAD -> master)  Author: hoshizorahikari <208343741@qq.com>  Date: Tue Apr 24 15:52:42 2018 +0800  add hi  commit 1e6bd77265d518ccc0cf7a2754f49d380f14fa27  Author: hoshizorahikari <208343741@qq.com>  Date: Tue Apr 24 15:38:26 2018 +0800  add change  commit aa9f146ce8cb85c37788c851bb5765f0d9889943  Author: hoshizorahikari <208343741@qq.com>  Date: Tue Apr 24 15:29:51 2018 +0800  add hello.py |

可以使用git log --pretty=oneline打印得稍微好看点

|  |
| --- |
| $ git log --pretty=oneline  0a2f62e65c0b29106ee1eb8e8cc13f091b7c59e1 (HEAD -> master) add hi  1e6bd77265d518ccc0cf7a2754f49d380f14fa27 add change  aa9f146ce8cb85c37788c851bb5765f0d9889943 add hello.py |

一长串字符串是commit id(版本号)，是一个SHA1值。这样的话，多人合作的项目版本号也很难冲突。

每提交一个新版本，Git就会把它们自动串成一条时间线。

在Git中用HEAD表示当前版本，上一个版本是HEAD^，上上一个版本是HEAD^^，往前100个版本写100个^很逗比，写成HEAD~100。

比如上面hello.py有3个版本，想要版本回退使用git reset命令：

|  |
| --- |
| $ git reset --hard HEAD^^  HEAD is now at aa9f146 add hello.py |

hello.py就回到过去，变为最初只有hello world的了。

此时再用git log查看版本库的信息：

|  |
| --- |
| $ git log  commit aa9f146ce8cb85c37788c851bb5765f0d9889943 (HEAD -> master)  Author: hoshizorahikari <208343741@qq.com>  Date: Tue Apr 24 15:29:51 2018 +0800  add hello.py |

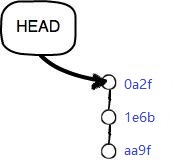
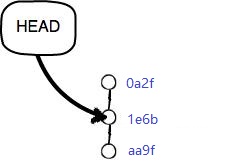
发现只有第1个版本，第2、3个版本不见了！怎么回到未来呢?

如果命令行窗口没关，可以往上面找到第2个版本的id：1e6b…

|  |
| --- |
| $ git reset --hard 1e6b  HEAD is now at 1e6bd77 add change |

版本号不必写全，写前几位就可以，Git会自动寻找。

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当回退版本时，仅仅是改变HEAD的指向：

然而，如果回退版本关了电脑，第2天醒来后悔了，想恢复到新版本，但是又找不到commit id了！

Git提供了一个命令git reflog用来记录每一次命令：

|  |
| --- |
| $ git reflog  1e6bd77 (HEAD -> master) HEAD@{0}: reset: moving to 1e6b  aa9f146 HEAD@{1}: reset: moving to HEAD^^  0a2f62e HEAD@{2}: commit: add hi  1e6bd77 (HEAD -> master) HEAD@{3}: commit: add change  aa9f146 HEAD@{4}: commit (initial): add hello.py |

① HEAD指向的版本就是当前版本。Git允许在版本的历史之间穿梭，使用命令git reset --hard commit\_id；

② 要回到以前，用git log可以查看提交历史，以便确定要回退到哪个版本；

③ 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。

* 工作区和暂存区

Git和其他版本控制系统(如SVN)一个不同之处就是有暂存区的概念。

① 工作区(Working Directory)：添加、编辑、修改文件等动作

电脑里创建的能看见的目录，比如之前的hikari\_git文件夹。

② 版本库(Repository)：工作区有一个隐藏目录.git就是Git版本库。

版本库最重要的是1) 暂存区stage：暂存已经修改的文件，最后统一提交到git仓库；2) Git自动创建的第一个分支master；3) 指向master的一个指针HEAD



git add实际上是把文件修改添加到暂存区；

git commit提交更改实际上是把暂存区的所有内容提交到当前分支。

因为创建Git版本库时，Git自动创建了唯一一个master分支，所以目前git commit就是往master分支上提交更改。可以简单理解为需要提交的文件修改全部放到暂存区，然后一次性提交暂存区的所有修改。

* 管理修改

Git比其他版本控制系统优势在于Git跟踪管理的是修改，而非文件。

先回到最近一次修改，版本号为0a2f …

|  |
| --- |
| $ git reset --hard 0a2f |

修改1：底部添加 # 修改1

修改2：底部再添加 # 修改2

如果顺序是：修改1 --> git add --> 修改2 --> git commit

git add将修改1添加到暂存区，修改2没有添加到暂存区，git commit只会将暂存区的修改提交到版本库。也就是修改1提交了，修改2没有提交。

用git diff HEAD -- hello.py可以查看工作区和版本库里面最新版本的区别：

|  |
| --- |
| $ git diff HEAD -- hello.py  **diff --git a/hello.py b/hello.py**  **index 002f29c..b33f7cb 100644**  **--- a/hello.py**  **+++ b/hello.py**  @@ -5,4 +5,5 @@ def hello(name):  def hi(name):  print('hi {}!'.format(name))  -# 修改1  +# 修改1  +# 修改2 |

也就是版本库里是修改1，工作区的修改2没有提交。

所以修改了别着急提交，先add了修改2，再commit：

修改1 --> git add --> 修改2 --> git add --> git commit

* 撤销修改

比如在hello.py最后写了一句*# stupid boss*如果提交了要扣工资啦

① hello.py在commit或add之后，不小心改错了，没有add到暂存区

|  |
| --- |
| $ git checkout -- hello.py |

撤销修改回到和最近一次commit或最近一次add一模一样的状态。

相当于直接丢弃工作区的修改，用git checkout -- filename (--左右都有空格)

② 修改错了，而且add到了暂存区，但没commit，想要撤回：

|  |
| --- |
| $ git reset HEAD hello.py  Unstaged changes after reset:  M hello.py  $ git checkout -- hello.py |

git reset既可以回退版本，也可以把暂存区的修改撤回，HEAD表示最新版本。

先用命令git reset HEAD filename回到了①，再按①操作。

③ 不合适的修改commit到了版本库：

|  |
| --- |
| $ git reset --hard HEAD^ |

需要版本回退，前提是没有推送到远程库。

* 删除文件

在Git中，删除也是一个修改操作。

通常直接删除或者rm命令删除：$ rm hello.py

Git知道删除了文件，因此工作区和版本库就不一致了。

git status命令显示哪些文件被删除了：

|  |
| --- |
| $ git status  On branch master  Changes not staged for commit:  (use "git add/rm <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  deleted: hello.py  no changes added to commit (use "git add" and/or "git commit -a") |

① 确认删除

从暂存区和工作区删除：git rm hello.py

如果暂存区和工作区文件不一致可以git rm -f hello.py强制删除

然后git commit提交：

|  |
| --- |
| $ git rm hello.py  rm 'hello.py'  $ git commit -m "删除hello"  [master 512ce00] 删除hello  1 file changed, 9 deletions(-)  delete mode 100644 hello.py |

现在，文件就从版本库中被删除了。想要恢复需要版本回退。

② 撤销删除

误删想恢复，版本库里还存在，可以很轻松把误删的文件恢复到最新版本：

|  |
| --- |
| $ git checkout -- hello.py |

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。

**20180425**

* 远程仓库

如果只是在一个仓库里管理文件历史，Git和SVN真没啥区别。

远程仓库是Git的杀手级功能之一。

通常找一台电脑充当服务器角色，24小时开机，其他人都从这个服务器仓库克隆一份到自己电脑上，把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

然而搭建一个服务器有点大题小做，有个叫GitHub的神奇网站，提供Git仓库托管服务。只要注册GitHub账号，就可以免费获得Git远程仓库。



① 注册GitHub账号

② 创建SSH Key

SSH (Secure Shell)：安全外壳协议，是建立在应用层基础上的安全协议，SSH是

目前较可靠的专为远程登录会话和其他网络服务提供安全性的协议。

Git本地仓库与GitHub仓库之间的传输是通过SSH加密传输的。

Linux默认安装SSH；Windows默认不安装SSH，但是Git Bash自带SSH。

|  |
| --- |
| $ ssh-keygen -t rsa -C "208343741@qq.com" |

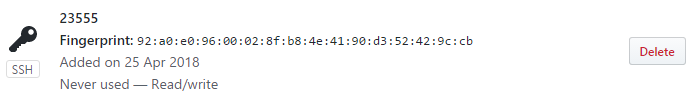
一路回车确认使用默认值即可。

使用RSA算法生成密钥；在用户目录找到.ssh 目录(Linux为~/.ssh；Windows为C:\Users\username\.ssh)，里面有id\_rsa和id\_rsa.pub两个文件，是SSH Key的秘钥对；id\_rsa是私钥，id\_rsa.pub是公钥。

如果已经存在这两个文件，此步跳过。

③ 登录GitHub --> Settings --> SSH and GPG keys --> SSH keys --> New SSH key

Title随便写，在Key文本框里粘贴id\_rsa.pub文件内容，点击Add SSH key



GitHub允许添加多个SSH Key。比如上班用公司电脑提交，在家用个人电脑提交，只要把每台电脑的SSH Key都添加到GitHub，就可以共享了。

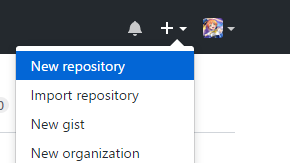
注意：在GitHub上免费托管的Git仓库，任何人都可以看到，但只能自己修改，不要把敏感信息放进去。

如果要私有Git库，一是交保护费，让GitHub把公开仓库变为私有，别人不可读更不可写；二是自己动手搭建一个Git服务器，公司内部开发必备。

* 添加远程库

在本地和GitHub创建Git仓库，让两个仓库进行远程同步，GitHub上的仓库既可以作为备份，又可以让其他人通过协作。

① GitHub右上角+号 --> New repository，创建一个新仓库



Repository name输入框输入仓库名字，如test

其他保持默认，点击Create repository创建了一个空的test仓库

② 根据GitHub的提示，在本地的hikari\_git仓库运行命令：

|  |
| --- |
| $ git remote add origin git@github.com:hoshizorahikari/test.git |

添加后远程库名字就是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。

③ 把本地hikari\_git仓库所有内容推送到远程库上：

|  |
| --- |
| $ git push -u origin master |

// 额…出现警告：The authenticity of host 'github.com (13.229.188.59)' can't be established. 但是输入yes还是push成功了…



git push：把本地库当前分支master推送到远程。

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master分支推送到远程新的master分支，还会把它们关联起来，在以后的push或者pull时就可以简化命令。

|  |
| --- |
| $ git push origin master |

* SSH警告

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 (13.229.188.59)' can't be established.  RSA key fingerprint is SHA256:xxxx.  Are you sure you want to continue connecting (yes/no)? |

因为Git使用SSH连接，在第一次验证GitHub服务器的Key时，需要确认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输出一个警告，显示已经把GitHub的Key添加到本机的一个信任列表里：

|  |
| --- |
| Warning: Permanently added 'github.com,13.229.188.59' (RSA) to the list of known hosts. |

这个警告只会出现一次，后面操作就不会有任何警告。

分布式版本系统最大好处之一是在本地工作完全不需要考虑远程库的存在，有没有联网都可以正常工作，而SVN在没有联网的时候拒绝干活! 当有网络时，再把本地提交推送一下就完成了同步!

* 从远程库克隆

假设从零开发，最好的方式是先创建远程库，然后从远程库克隆。

① GitHub创建一个新仓库比如hello，勾选README.md，稍微修改一下

② git clone命令克隆到本地库：

|  |
| --- |
| $ git clone git@github.com:hoshizorahikari/hello.git  Cloning into 'hello'...  remote: Counting objects: 6, done.  remote: Compressing objects: 100% (3/3), done.  remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0  Receiving objects: 100% (6/6), done. |

README.md：

|  |
| --- |
| # **hello**  ---  ## **hello world**  ---  ```python  def **hello**(name):  **print**('hello, {}!'.format(name))  ``` |

如果多个人协作开发，那么每个人各自从远程克隆一份就可以了。

GitHub还可以用https://github.com/hoshizorahikari/hello.git这样的地址。实际上Git支持多种协议，默认的git://使用ssh，也可以使用https等其他协议。

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

* 分支管理

分支如同平行世界，当兔斯基正在电脑前努力学习Git的时候，另一个兔斯基正在平行世界里努力学习SVN。

如果两个平行世界互不干扰，自然没有什么影响。如果在某个时间点，两个平行世界合并了，结果兔斯基既学会了Git又学会了SVN !



假设准备开发一个新功能，可以建立一个属于自己的分支，别人看不到，还能在原来的分支继续正常工作；自己搬自己的砖，影响不到别人。当开发完成，如果感觉不错，可以合并到原来的分支。

其他版本控制系统如SVN等都有分支管理，但是创建和切换分支比蜗牛还慢，让人无法忍受；结果分支功能成了摆设，大家都不用。

但Git的分支与众不同，无论创建、切换和删除分支，Git在1s之内能完成! 无论版本库是1个文件还是1万个文件!

* 创建与合并分支

每次commit，Git把每个版本串成一条时间线，这条时间线就是一个分支。在Git里，主分支为master分支。HEAD严格来说不是指向commit，而是指向master；master才是指向commit。所以HEAD指向的就是当前分支，每次提交master分支都会向前移动一步，随着不断commit，master分支也越来越长：



当创建新的分支如dev时，Git新建了一个指针叫dev，指向与master相同的commit，再把HEAD指向dev，表示当前分支在dev上：



Git创建一个分支很快，因为只是增加一个dev指针和改变HEAD的指向。

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



如果在dev分支工作完成了，可以把dev合并到master上了。Git直接把master指向dev当commit，就完成了合并：



Git合并分支也很快！只改变指针，工作区内容也不变!

合并完分支后，可以删除dev分支。删除dev分支就是把dev指针给删除：



示例：在clone的hello仓库操作

① 创建并切换到dev分支

|  |
| --- |
| $ git checkout -b dev  Switched to a new branch 'dev' |

git checkout -b dev相当于两条命令：

|  |
| --- |
| $ git branch dev  $ git checkout dev  Switched to branch 'dev' |

② 查看当前分支：

|  |
| --- |
| $ git branch  \* dev  master |

git branch命令列出所有分支，当前分支前面标记一个\*号。

③ 修改README.md，底部添加：#### **dev branch**

提交：

|  |
| --- |
| $ git add README.md  $ git commit -m "test branch"  [dev 428f82b] test branch  1 file changed, 2 insertions(+) |

④ 切换回master分支：

|  |
| --- |
| $ git checkout master  Switched to branch 'master'  Your branch is up to date with 'origin/master'. |

再次查看README.md，刚才添加的内容不见了！因为刚才是在dev分支上提交，而master分支此刻的提交点(//理解为快照?)并没有变：



⑤ 合并分支

将dev分支合并到master分支：

|  |
| --- |
| $ git merge dev  Updating d6f6104..428f82b  Fast-forward  README.md | 2 ++  1 file changed, 2 insertions(+) |

git merge命令用于合并指定分支dev到当前分支master。

合并后查看README.md，发现和dev分支的最新提交是完全一样的。

注：Fast-forward表示合并是快进模式，直接把master指针指向dev当前提交，所以合并速度非常快。

也有可能会发生冲突，比如不同分支有同名文件但是内容不同…

⑥ 删除分支

合并完成后可以放心地删除dev分支：

|  |
| --- |
| $ git branch -d dev  Deleted branch dev (was 428f82b).  $ git branch  \* master |

删除dev分支后，只剩master分支了。

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工作效果是一样的，但过程更安全。

分支小结：

① 查看分支：git branch

② 创建分支：git branch <name>

③ 切换分支：git checkout <name>

④ 创建+切换分支：git checkout -b <name>

⑤ 合并某分支到当前分支：git merge <name>

⑥ 删除分支：git branch -d <name>

* 解决冲突

人生不如意十有八九；合并分支往往也不是一帆风顺。

① 创建新的feature1分支：

|  |
| --- |
| $ git checkout -b feature1  Switched to a new branch 'feature1' |

修改README.md最后一行，改为：#### **feature1111**

② 在feature1分支上提交：

|  |
| --- |
| $ git add README.md  $ git commit -m "111"  [feature1 8a0299d] 111  1 file changed, 1 insertion(+), 1 deletion(-) |

③ 切换到master分支：

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

Git还会提示当前master分支比远程的master分支要超前1个提交。

④ master分支，README.md最后一行改为：#### **master**

提交：

|  |
| --- |
| $ git add README.md  $ git commit -m "master"  [master cc3e87b] master  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：



⑤ 合并feature1分支到master

这种情况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：

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

提示README.md存在冲突，必须手动解决冲突后再提交。

git status也可以显示冲突的文件：

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

查看README.md：

|  |
| --- |
| <<<<<<< HEAD  #### **master**  =======  #### **feature1111**  >>>>>>> feature1 |

Git用<<<<<<<、=======、>>>>>>>标记出不同分支的内容；

⑥ 修改最后一行：#### **master and feature1111**

提交：

|  |
| --- |
| $ git add README.md  $ git commit -m "conflict fixed"  [master 4e24c99] conflict fixed |

现在，master分支和feature1分支变成了下图所示：



⑦ git log --graph可以看到分支合并图：

|  |
| --- |
| $ git log --graph --pretty=oneline --abbrev-commit  \* 4e24c99 (HEAD -> master) conflict fixed  |\  | \* 8a0299d (feature1) 111  \* | cc3e87b master  |/  \* 428f82b test branch  \* d6f6104 (origin/master, origin/HEAD) Update README.md  \* ea7da2f Initial commit |

git log --graph --oneline --all好像显示是一样的…

⑧ 删除feature1分支：

|  |
| --- |
| $ git branch -d feature1  Deleted branch feature1 (was 8a0299d). |

开发工作完成。

* 分支管理策略

通常合并分支时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。

git merge --no-ff可以强制禁用Fast forward模式，Git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。

① 创建并切换dev分支：

|  |
| --- |
| $ git checkout -b dev  Switched to a new branch 'dev' |

② README.md最后再添加一行：#### **test git merge --no-ff**

提交：

|  |
| --- |
| $ git add README.md  $ git commit -m "test --no-ff"  [dev a154106] test --no-ff  1 file changed, 1 insertion(+) |

③ 切换回master：

|  |
| --- |
| $ git checkout master  Switched to branch 'master' |

④ 合并dev分支

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.md | 1 +  1 file changed, 1 insertion(+) |

因为合并要创建一个新的commit，所以加上-m参数添加描述。

⑤ git log查看分支历史

|  |
| --- |
| $ git log --graph --oneline  \* 1211b9f (HEAD -> master) merge with no-ff  |\  | \* a154106 (dev) test --no-ff  |/  \* 4e24c99 conflict fixed  |\  | \* 8a0299d 111  \* | cc3e87b master  |/  \* 428f82b test branch  \* d6f6104 (origin/master, origin/HEAD) Update README.md  \* ea7da2f Initial commit |

// 之前被删除的feature1的快照还存在，只是没有名字…

可以看到，不使用Fast forward模式，分支合并图：



在没冲突情况下合并分支，使用Fast forward，dev和master都指向a154，查看历史看不出曾经做过合并；加上--no-ff参数使用普通模式合并，master指向新创建的快照1211而不是dev快照a154，查看历史能看出曾经做过哪些合并。

* 分支策略

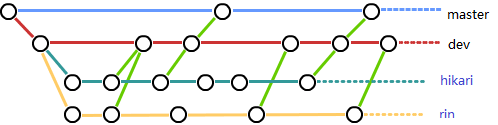
在实际开发中，应该按照几个基本原则进行分支管理：

① master分支应该非常稳定，仅用来发布新版本，平时不能在上面干活；

② 搬砖都在dev分支上，也就是dev分支是不稳定的；如果1.0版本要发布，把dev分支合并到master上，在master分支发布1.0版本；

③ 每个人都在dev分支上搬砖，每个人都有属于自己的分支，时不时往dev分支上合并自己的分支就可以。

所以，团队合作的分支看起来就像如下：



* Bug分支

软件开发中bug就像家常便饭一样，有了bug就需要修复。Git中每个bug都可以通过一个新的临时分支来修复，修复后合并分支，将临时分支删除。

比如正在兴致勃勃地在dev分支搬砖，突然stupid boss传来一个代号为101的bug任务。此时需要创建一个分支issue-101来修复它，但是当前在dev上进行的工作还没有提交。并不是不想提交，砖搬到一半，可能还要一天才能搬完，然而sb命令必须2h内修复bug，为之奈何?

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。

假如正在dev分支写hello.py项目，而issue-101是要删除master分支README.md最后几句屁话：

① 编写hello.py，不管add与否都可以暂存，也就是保存暂存区和工作区内容?

② 接到修复bug任务，git status查看工作区

|  |
| --- |
| $ git status  On branch dev  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: hello.py |

③ git stash将工作现场储存起来

|  |
| --- |
| $ git stash  Saved working directory and index state WIP on dev: 2d87fc9 test again |

再用git status查看工作区

|  |
| --- |
| $ git status  On branch dev  nothing to commit, working tree clean |

发现工作是干净的，可以放心创建分支修复bug

④ 切换到bug分支(此处为master)创建临时分支

|  |
| --- |
| $ git checkout master  Switched to branch 'master'  Your branch is ahead of 'origin/master' by 7 commits.  (use "git push" to publish your local commits)  $ git checkout -b issue-101  Switched to a new branch 'issue-101' |

⑤ 修复bug

此处将几句废话删除，添加一句 ### bug fixed

提交：

|  |
| --- |
| $ git add README.md  $ git commit -m "fix bug 101"  [issue-101 07e06d9] fix bug 101  1 file changed, 1 insertion(+), 3 deletions(-) |

⑥ 合并分支，删除临时分支

|  |
| --- |
| $ git checkout master  Switched to branch 'master'  Your branch is ahead of 'origin/master' by 7 commits.  (use "git push" to publish your local commits)  $ git merge --no-ff -m "merged bug fix 101" issue-101  Merge made by the 'recursive' strategy.  README.md | 4 +---  1 file changed, 1 insertion(+), 3 deletions(-)  $ git branch -d issue-101  Deleted branch issue-101 (was 07e06d9). |

⑦ 切回dev继续搬砖

|  |
| --- |
| $ git checkout dev  Switched to branch 'dev'  $ git status  On branch dev  nothing to commit, working tree clean |

工作区还是干净的，用git stash list命令查看：

|  |
| --- |
| $ git stash list  stash@{0}: WIP on dev: 2d87fc9 test again |

Git把stash内容存在某个地方，恢复有两个办法：

1) git stash apply：恢复后stash内容并不删除，需要用git stash drop删除；

2) git stash pop：恢复同时把stash内容删除。

|  |
| --- |
| $ git stash pop  On branch dev  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: hello.py  Dropped refs/stash@{0} (9b319c98913ef2f8a3a0985d598deea8317a4b9d) |

hello.py又回来了…

再用git stash list查看，看不到任何stash内容：

|  |
| --- |
| $ git stash list |

可以多次stash，恢复时，先用git stash list查看，然后恢复指定的stash用：

|  |
| --- |
| $ git stash apply stash@{0} |

// 但是dev分支的bug还是没修复啊…提交后切回master，然后合并dev还是会冲突…