Git版本管理

<https://github.com/doggy8088/Learn-Git-in-30-days/blob/master/docs/01%20%E8%AA%8D%E8%AD%98%20Git%20%E7%89%88%E6%9C%AC%E6%8E%A7%E7%AE%A1.markdown>

Git: distributed version control system (DVCS)

文章作者，开始使用Git版本管控，其实是“选择使用GitHub当作版控平台”

* GitHub整合很好
* 完整的Git版控支援
* 议题追踪与管理
* 线上Wiki文件管理
* 友善的代码审核code review

Git: 分散式版本管控，每个人都有一份完整的repository，但必须经常合并档案。分支与合并是常态，但只要有合并，就有冲突，要学会如何解决冲突

* Git完全不需要server的支援就可以运行版本管控，因为每个人都有一份完整的储存库副本
* 每个人都有一份完整的储存库副本，每次提交版本，仅提交到本地repository，因此速度非常快，也不用网络
* 每人都有一份完整的储存库副本，使用Git版控，没有“权限”，每个成员都能clone，然后在本地提交变更。使用Git，需要设定的权限是，能否存取上层储存库upstream repository或远端储存库remote repository
* 如果需要跟别人交换变更的版本，随时可以通过“合并”，Git有非常强悍的合并追踪能力merge tracing
* 要合并多人版本，只要有存取共用shared repository的权限或管道即可。如在一台server上可以通过资料夹权限进行共用，或通过SSH远端存取另一台server的Git储存库，也可通过Web伺服器等方式来共用Git储存库

# 1. 建立储存库

在本机，建立本地储存库local repository

在本机，建立公用储存库shared repository

在GitHub或其他Git平台建立远端储存库remote repository

## 1.1 Local repository

Git Shell > mkdir git-demo > cd git-demo

这是空目录，不包含任何Git repository，用git init建立储存库，预设储存库会放在工作目录下的.git下

## 1.2 Shared repository

Shared repository指建立一个Git储存库，但不包含工作目录，常发生在Linux下，因为Linux通常多人使用同一台主机。用git init –bare建立，会在当前目录建立所有Git储存库的相关档案与资料夹。这个资料夹不能直接编辑，透过git指令操作。因为“没有工作目录的纯储存库”，也叫bare repository

Git是“分布式版本管控”，每人都有一份完整的repository。当你想建立一个工作目录working directory时，必须先取得这个bare repository内容回来：git clone [REPO\_URL]

建立工作目录，而且把这个shared repository完整地复制回来。

**注意**：工作目录下，.git也是一个储存库，但是工作目录下.git还包含工作目录下的索引资讯，记录工作目录下的状态、信息，这些不会出现在shared repository中。Shared repository里只有版本资讯。

## 1.3 Remote repository

可以在网站上完成相关操作，也可以使用command line

touch README.md

git init

git add README.md

git commit –m “first commit”

git remote add origin <https://github.com/cqbaizuo/LearnGit.git>

git push –u origin master

# 2. 常用的Git版本管控指令

[master +10 ~1 -2!]

master表示目录工作目录是master 分支，也是Git默认分支

“**红色**”都代表Untracked的档案，即这些变更都不会进入版本管控

+10 新增10个文件

~1 修改1个文件

-2 删除2个文件

## 2.1 新增档案

git add . 告知Git，这些变化“即将”加入Git版本库，但还没建立任何版本

git add app: 加入app资料夹与其下的所有档案

git add .\*: 加入所有”.”开头的档案

变成“**绿色**”，staged，变更会进入版本管控

## 2.2 提交变更、建立版本

git commit –m “words to explain this version”

git help commit git help add

## 2.3 查询历史记录

git log git log -10

## 2.4 档案更名

git mv test unit-test

## 2.5 显示工作目录的状态

git status git status –s: 精简版本

## 2.6 重置当前工作目录

git reset: 只会重置“索引”状态，用git rm删除的目录、文件，git mv更名的目录、文件，不能通过git reset把实体档案找回来

如果想把工作目录还原到目前的最新版，git reset --hard

## 2.7 还原其中一个档案

git checkout master Gruntfile.js 把master分支最新版的Gruntfile.js还原，可以避免使用git reset --hard 将所有档案还原

# 3. 了解储存库、工作目录、物件、索引之间的关系

## 3.1 储存库

### 3.1.1 了解储存库

使用Git进行版本，需要一个“版本库”来储存这些版本信息，这就是Repository

### 3.1.2了解工作目录

$ mkdir git-demo

$ cd git-demo

$ git init

git-demo自动地成为了“工作目录”working directory，我们正在开发的专门文件夹，未来的操作都在这个目录下进行。我们操作Git相关指令、参数时，通常也在working directory下执行。

使用Git版本管控时，会遇到很多分支。适时地保持工作目录干净，是版本管控的一个基本原则。

## 3.2 了解Git的资料结构

Git里两个重要的资料结构：物件、索引。物件：保存版本库中文件与版本记录；索引：保存当下要进版本库之前的目录状态

### 3.2.1 关于物件

物件：特别的档案，内容取出，产生一组SHA1杂凑值，命名这个档案。

物件分为：目录资讯和档案内容，即tree object & blob object，都存放于object storage中。

### 3.2.2 关于索引

“索引”是一个经常异动的暂存档，通常位于.git目录下，名为index的档案。“索引”，记录“哪些档案即将要被提交到下一个commit版本中”，想要提交一个版本到Git repository，那么一定要先更新索引状态。

通常不直接编辑.git\index这个二进位档，而是透过git指令：

* git add
* git mv, git rm
* git status
* git commit
* git ls-files

索引：位于working directory和object storage之间的媒介。任何working directory的变化，都需要先更新索引，才能加入object storage版本管控中。

git branch rename: git branch -m <oldname> <newname> git branch -m <newname>, for the current branch

# 4. 物件结构

物件是immutable，所有储存在object storage中的档案，通常只出不进。

## 4.1 物件资料库

储存在.git\objects中，每个物件根据“档案内容”，进行SHA1杂凑运算，得到hash值，作为文档名称。

## 4.2 物件名称

* blob物件：某个档案的“内容”，并且只是内容而已。执行git add指令时，这些新增档案，立刻被写入成为blob物件，名称是杂凑值。
* tree物件：某个资料夹的快照snapshot
* commit物件：特定版本有哪些tree物件、版本提交时间、记录信息，通常还有上一层commit物件名称
* tag物件

# 5. 索引结构

## 5.1 关于索引

索引：记录“哪些档案即将被提交到下一个commit版本中”。想要提交一个版本到Git repository，必须先更新索引状态git add。有很多别名：index, cache, directory cache, current directory cache, staging area, staged files

如：git diff --cached == git diff --staged



$ git status:

# changes to be committed

# new file: c.txt

# changes not staged for commit:

# modified: a.txt

# untracked files

# b.txt

* changes to be committed
  + 最新版没有c.txt
  + 索引档加入了c.txt
  + git commit后，存入下一个版本
* changes not staged for commit
  + 最新版有a.txt
  + 索引档没加入a.txt
  + git commit不会进入下一版本
* untracked files
  + 最新版没有b.txt
  + 索引档也没有b.txt
  + git commit不会进入下一版本

实际上，git commit是将索引档里的文件加入创建的版本中。因此，只有目前最新版和索引档之间有差异的变更，才会被提交到下一个commit物件中

## 5.2 常用命令

* git add
* git rm git rm --cached a.txt 删除索引档中该文档，但保留实体文件
* git mv
* git commit
* git ls-files

# 6. 分支的具体观念和使用方式

## 6.1 分支的基本观念

最主要的目的：开发过程中，版本冲突的问题，可以：从远端pull一个分支到本地分支；将本地分支push到远端；也可以将本地任何一个分支merge到本地另一个分支

## 6.2 常用操作

* git branch NewBranch 创建branch，并停留在当前branch
* git checkout -b NewBranch 创建branch，切换到新的branch
* git branch 查看现有branch
* git checkout BranchName 查看BranchName
* git branch -d BranchName 删除BranchName
* git checkout CommitID 查看某个commitID

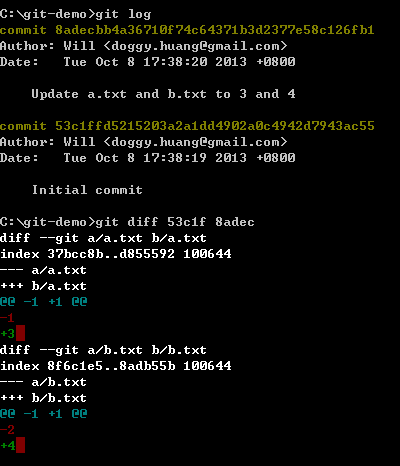
**注意**：此时，可能会切换到“旧的”版本（detached HEAD），“目前不在最新版”的提示。由于此版本已有下一版，如果在目前的“旧版”执行git commit，会导致刚刚创建的版本，无法被追踪。正确的方法是：建立分支。

# 7. 对比档案与版本差异

## 7.1 git diff的基本观念

1. 先用git log得到版本资讯，并取得最近两个commit的id
2. 执行git diff commit1 commit2，对比两个版本差异。commit1用旧版，commit2用新版

$ git diff d120304519439dd8bade2eb2dae81b2226a4e02b dc1d4e5bebd863a9a1f4d5f5a3b4fa5cdf53c3f2



* diff --git: 对哪两个档案进行比较
* index ca90535..00b30ed 100644: git此次对比的标头资讯header line，后面两个hash id代表Git object storage中两个blob物件的id。100644是git属性，类似Linux的rwx
* ---a/a.txt 两个比对版本中“旧的”那个版本
* +++b/a.txt两个比对中“新的”那个
* @@ -1 +1 @@旧版的总行数与新版的总行数
* 变更的内容
  + “-”开头，代表旧版到新版，此行被删除
  + “+”开头，代表旧版到新版，此行新增
  + 空白字元开头，代表新旧版本都出现，没有变更

## 7.2 四种基本比较方式

1. git diff: 不加任何参数，对比“工作目录”和“索引”的差异
2. git diff commit: 对比“工作目录”与“指定commit物件里的那个tree物件”

最常用：git diff HEAD，对比“工作目录”与“当前分支最新版”，这种方法，不会对比“索引”

1. git diff --cached commit: 对比“索引”与“指定commit物件里的那个tree物件”

最常用：git diff --cached HEAD，对比“索引”与“当前分支最新版”

1. git diff commit1 commit2

# 8. Git物件的绝对名称

## 8.1 物件绝对名称

Git中，每个物件有一个SHA杂凑运算的ID，这就是“绝对名称”。

“简短语法”：不少于4个字元（4~40个长度都可用）

git log --pretty = oneline 一行显示结果

git log --pretty = oneline --abbrev-commit 一行，并且仅输出部分“绝对名称”

# 9. Git物件的一般参照与符号参照

Git版控最常用的“**参照名称**”

## 9.1 物件的参照名称

参照名称ref： Git物件的一个“指标”，相对绝对名称的另一个“好记名称”。

“分支名称”，HEAD，或之后学的“标签名称”，都是“参照名称”

$ git branch 得到分支名称。分支名称其实就是“参照名称”，代表这三个“参照名称”分别对应Git物件储存库中的三个commit物件。其实，分支的参照名称，实际就是一个档案而已（.git\ref\heads目录下）

档案如何跟“绝对名称”连接？

$ git branch # we are now in newbranch3

$ git log --pretty = oneline

打开.git\ref\heads\newbranch3，纯文字档，指向版本历史的“最新版”

$ git cat-file –p commitID 取得commit物件内容

$ git show commitID 取得版本变更记录

$ git cat-file newbranch3 $ git cat-file commitID 绝对名称和相对名称，执行得到相同结果

## 9.2 关于.git\refs\目录

所有**“参照目录”实际是一个档案**，包含文件的绝对名称，放在.git\refs下。输入git cat-files –p BranchName，Git会寻找“参照名称文档”，取出内容（即Git物件的绝对名称）

.git/refs/heads/

.git/refs/remotes/

.git/refs/tags/

## 9.3 物件的符号参照名称(symref)

符号参照名称(symref)也是参照名称(ref)的一种，只是内容不同。“符号参照”，指向另一个参照名称

$ type .git\HEAD # ref: refs/heads/newbranch1

* HEAD
  + 指向“工作目录”设定分支的最新版
  + git commit后，HEAD也更新至最新版
* ORIG\_HEAD
  + HEAD这个commit物件的“前一版”，常用来恢复上一版
* FETCH\_HEAD
  + 远端储存库时，可能会使用git fetch取回所有远端储存库的物件。FETCH\_HEAD符号参考，记录远端储存库每个分支的HEAD（最新版）的绝对名称
* MERGE\_HEAD
  + 合并工作时，“合并来源”的commit物件绝对名称会被记录在MERGE\_HEAD中

## 9.4 一般参考与符号参考使用方式

git update-ref自由建立“一般参考”

$ git update-ref NewRefName commitID

**注意**：可以指向任意Git物件，不是一定要commit物件；较正式，最好加上refs/开头，如git update-ref refs/NewRefName [objectID]

# 10. 认识Git物件的相对名称

## 10.1 相对名称表示法

**特殊符号**：^ ~

HEAD的前一版本：HEAD~/HEAD~1 f2e branch的前两个版本 f2e~~/f2e~2

没有分支与合并的储存库，^1与~1是一样的，都代表“前一版”；若有分支、合并，则不同意义

## 10.2 commit物件彼此间的连结

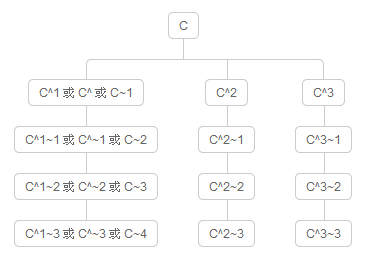
常见Git repository，预设一个“根commit物件”：Initial Commit。除了第一个commit物件，其他任何commit物件都会有一个、或多个parent commit（合并两个以上分支到另一个分支里，所以会有多个parent commit）

## 10.3 相对名称表示法~, ^的差异

~: 第一个上层commit物件

^: 拥有多个上层commit物件时，代表上一层第几个物件

如C^2，不能得到上面第二层。因为C^2找到的是：上一层中的第二个



## 10.4 git rev-parse指令

将任意“参考名称”或“相对名称”解析出“绝对名称”

# 11. 暂存工作目录与索引的变更状态

某个系统，开发到一半，被老板、客户“插单”，紧急修正一个现有Bug或添加一个功能：未完成的软件开发状态外加紧急调整的需求。Git可以很好地帮助你。

## 11.1 Git stash指令

Git版控，工作目录：有改到一半，还没完成的档案（unstaged），也有新增还没加入的档案（untracked）；索引：打算通过git commit建立版本。

当功能开发到一半，被紧急插单，尤其是手边写到一半的程序，不知如何是好。git stash，可以自动帮你把写到一半的档案建立一个“特殊的版本”（也是一个commit物件），称为stash版本，“暂存版”

## 11.2 建立暂存版

* 新增档案（未列入追踪）untracked
* 新增档案（已加入索引）tracked/staged
* 修改档案（未加入索引）tracked/unstaged
* 修改档案（已加入索引）tracked/staged
* 删除档案（未加入索引）tracked/unstaged
* 删除档案（已加入索引）tracked/staged

git stash**两个选择**：

* git stash将所有tracked的档案，建立暂存版
* git stash –u将所有tracked, untracked，全部建立暂存版。可将目前全部变更储存起来（包含untracked），储存结束后，重置到HEAD状态

$ git cat-file -p stash: 可以发现，物件内容里，有3个parent commit物件。这个特殊的暂存版，是从另外3个版本合并进来的。

## 11.3 取回暂存版本

“工作”目录已被重置，变更储存到stash里，需要取回暂存档，通过git stash pop重新“合并”回来

## 11.4 建立多重暂存版

正常开发情况下，不太会发生多重暂存版，除非：

* 开发习惯太差，积累一堆可能不用的暂存档
* 老板、客户“插单”太严重

具体做法，暂时不管，需要时再查

# 12. 标签-标记版控中的重要事件

在众多版本中，一定有值得我们记录的几个重要版本，这就是**标签Tag**能帮的事情

## 12.1 Tag的基本概念

Tag: 标记某一个版本、或commit物件，已一个“好记的名称”帮助记忆某个版本

* 轻量标签lightweight tag: 某个commit版本的“**别名**”，是一种相对名称存在.git\refs\下

与git update-ref refs\NewRefName commitID建立的相对名称，本质一样

* 标示标签annotated tag: 则是一种**Git物件**，存在.git\objects\下，并关联到另一个commit物件，建立annotated tag时，能像建立commit物件一样，包含“版本信息”

## 12.2 lightweight tag使用方式

$ git log -l

# commit efa1e0…

$ git tag 1.0.0-alpha

* git tag: 列出所有标签
* git tag [tagname]: 建立轻量标签
* git tag [tagname] -d: 删除轻量标签

查看轻量标签内容，可以通过git cat-file -p [tagname]；查看物件类型，git cat-file -t [tagname]

## 12.3 annotated tag的使用方式

$ git log -1

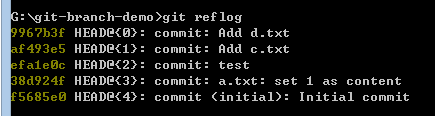
$ git tag 1.0.0-beta -a -m ‘Beta version’

$ git cat-file -t 1.0.0-beta # tag类型

# 13. 善用版本日志git reflog追踪变更轨迹

## 13.1 版本记录的过程

$ git reflog



## 13.2 复原意外的变更

使用Git，可能不小心执行错误，可以利用HEAD@{0}这个特殊的“参考名称”，对版本“定位”，将目前Git储存库版本恢复到任何一版

“取消”最近一次版本记录：git reset “HEAD@{1}” --hard复原变更（设置HEAD为@1）。但是Git里，这个操作，仍然会被记录。如果想复原git reset “HEAD@{1}” --hard，再执行一次git reset “HEAD@{1}” --hard 即可。

## 13.3 记录版本变更的原则

Git版控中，会产生很多“版本历史记录”，不是单纯的git log显示的，而是完整的变更历史记录。原则：只要通过指令修改了任何参照ref内容，或变更任何分支的HEAD参照内容，都会建立历史记录（改了ref引发的log）

## 13.3 reflog记录更多用法

$ git reflog HEAD

$ git reflog master

$ git reflog –g 显示详细版本记录

$ git reflog delete ref@{specifier} 删除特定历史记录

## 13.4 设定历史记录过期时间

默认，Git保存历史记录90天；如果这些记录有些commit物件不在分支线上，默认保存30天。

先建立分支branch1，然后commit了几个版本，最后把该分支直接删掉。这些commit仍在object storage里，但不能通过git log取得该版本，称为“不在分支线上的版本”

修改默认过期期限：git config gc.reflogExpire git config gc.reflogExpireUnreachable

$ git reflog gc.Expire “7 days”

$ git reflog gc.ExpireUnreachable “8 days”

## 13.5 清楚历史记录

立即清楚历史记录，通过git reflog expire --expire = now --all 然后配合git gc整理、清楚找不到、无法追踪的版本

# 14. 关于合并的基本观念与使用方式

## 14.1 合并的基本观念

Git下建立分支，可以让系统根据不同需求，分别开发、互不影响。如：原本稳定的系统放在master分支中；修正错误时则建立bugfix分支，修正错误，bugs修正后，通过“**合并**”将bugfix分支上的变更重新套用在master上。合并时，“**必须发生在同一个储存库里**”

合并时，如果两个分支有修改到相同档案，但只要修改的行数不同，Git会自动帮你套用/合并这两个变更。但如果刚好，两个分支里，修改到**“同一档案”的“同一行”**，会引发冲突事件。冲突时，Git不会决定任何事情，将“解决冲突”的工作交给你。发生冲突的档案，显示为”unmerged”状态。

## 14.2 “成功的合作”

$ mkdir git-merge-demo

$ cd git-merge-demo

$ git init

$ echo . > a.txt

$ git add .

$ git commit -m ‘Initial commit (a.txt created)’

$ echo 1 > a.txt

$ git add .

$ git commit -m ‘Update a.txt’

$ git checkout –b feature

$ echo 2 > b.txt

$ git add .

$ git commit -m ‘Add b.txt’

**注意**：切换“分支”时，随时查看保持“工作目录”是“干净的”，不要有任何档案异动

* 主分支：master
  + a.txt: 内容是1
* feature 分支
  + a.txt: 内容是1（从master继承）
  + b.txt: 内容是2

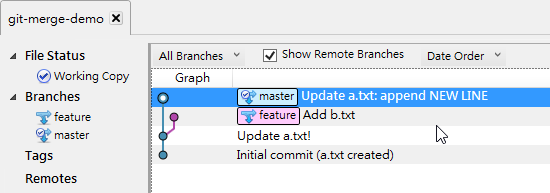
$ git checkout master

$ echo NEWLINE >> a.txt

$ git add .

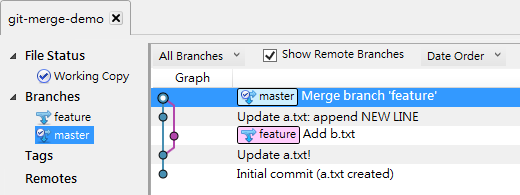
$ git commit –m ‘Update a.txt: append NEW LINE’

* 主分支：master
  + a.txt: 内容是两行，1 & NEW LINE
  + 没有feature分支的变更，即没有b.txt
* feature 分支
  + a.txt: 内容是1（从master继承）
  + b.txt: 内容是2 (feature分支新建立的)



建立feature分支后，master和feature都有修改，但修改不冲突，合并不会有问题

1. 合并前，清楚自己在哪个分支
2. 合并前，确保工作目录干净 ($ git status确保”nothing to commit”)
3. git merge [另一个分支] 将另一个分支变更合并回来
4. 合并后，用git log，发现“合并”会自动建立一个新版本



## 14.3 删除不必要的分支

如果feature还会持续开发新功能，可以暂时留着

如果确定不用的话，git branch –d feature删除

Git里，没有执行合并的分支，不能用-d删除，只能用-D强行删除

## 14.4 救回误删的分支

1. git reflog找出该分支最后一个版本的object id
2. git branch feature <SHA1>

## 14.5 冲突的合并

$ git checkout –b hotfixes

$ notepad a.txt (change line 1 to “bugfixed: 1”)

$ git add .

$ git commit –m ‘a.txt bug fixed’

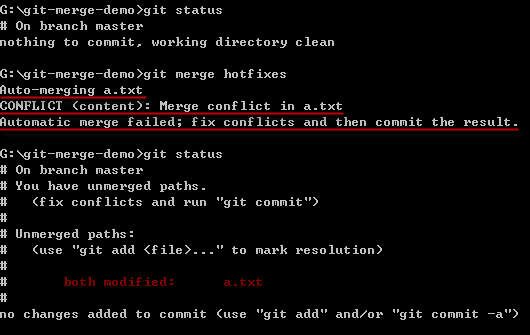
$ git checkout master

$ notepad a.txt (change line 1 to “bugfixed by cqbaizuo: 1”)

$ git add .

$ git commit –m ‘a.txt bug fixed by cqbaizuo’

$ git merge



## 14.6 查看冲突内容、解决问题

$ git diff

$ git add . 把当前工作目录的状态加入“索引”中，则可取消这些unmerged档案的冲突状态。但是commit版本会包含这些冲突文字（**不是好的解决方案**）

$ git status/git ls-files –u 找出冲突档案

**必须手动修改冲突的文件**，使用SourceTree可以很方便。

# 15. 修正commit过去版本历史记录

Git提供更多“修正版本记录”的机制，让你在“分享”版本给其他人时，能够预先整理

## 15.1 版本管控基本原则

* 做一个小功能修改就建立版本，才容易追踪变更
* 千万不要积累大堆修改才建立“大版本”
* 有逻辑、顺序的修正功能，确保相应的版本修正能够按顺序commit，便于追踪

## 15.2 修正commit的注意事项

希望“自我管控版本”，整理资讯，好让版本发布后，其他人能更清楚地理解你对版本做了哪些修改

* 储存库可以有许多分支
* 分析Git源代码最小单位是“分支”
* 任何修改某个branch的版本，只要还未“分享”给他人
* “分享”特定分支后，已分享的版本历史记录就别再改了

$ git reset --hard “HEAD^” 删除最新版，将HEAD指向前一个版本

这是危险操作，因此可以取消：git reset --hard ORIG\_HEAD

ORIG\_HEAD仅针对“**危险操作**”（如merge），而HEAD@{}链表则记录了每次HEAD的移动（包括commit），推荐使用**HEAD@{}**

$ git reset --soft “HEAD^” 删除最近一次版本记录，但保留最后一次版本的变更内容：保留更改+新的变更，重新执行commit，记录版本

## 15.3 重新提交最后一个版本HEAD

不小心执行了git commit，却发现有些档案忘记加入或讯息写错，想补上，执行git --amend即可。

# 16. 设定.gitignore忽略清单

开发时，可能会产生新的档案（如Visual Studio产生的暂存档案、快取档案；word编辑时，产生的暂存档案），不需列入版本管控“忽略清单”

Git通过.gitignore定义忽略清单。列在里面的档名或路径（可用regular expression）不会出现在git status结果中，也不会被git add添加（仅限untracked files）。已经列入版控 (staged files)，不受.gitignore控制

借助 GitHub建立默认忽略清单