**版本控制**

**Git**

目录

[目录 2](#_Toc463470458)

[第1章 Shell和vi 4](#_Toc463470459)

[1.1 什么是shell 4](#_Toc463470460)

[1.2 shell分类 4](#_Toc463470461)

[1.3 认识bash这个shell 5](#_Toc463470462)

[1.4 vi编辑器 6](#_Toc463470463)

[1.5 SSH 8](#_Toc463470464)

[第2章 版本控制 9](#_Toc463470465)

[2.1 关于版本控制 9](#_Toc463470466)

[2.2 本地版本控制系统 10](#_Toc463470467)

[2.3 集中式版本控制系统 11](#_Toc463470468)

[2.4 分布式版本控制系统 12](#_Toc463470469)

[第3章 Git 13](#_Toc463470470)

[3.1 Git安装 13](#_Toc463470471)

[3.2 Git工作原理 13](#_Toc463470472)

[3.3 Git本地仓库 14](#_Toc463470473)

[3.3.1 Git基础 14](#_Toc463470474)

[3.3.2 Git分支 20](#_Toc463470475)

[3.4 Git远程（共享）仓库 26](#_Toc463470476)

[3.5 gitHub和gitLab 28](#_Toc463470477)

[3.6 命令汇总 32](#_Toc463470478)

[3.7 GitLab完整演示 33](#_Toc463470479)

[3.8 冲突解决 34](#_Toc463470480)

[3.9 Git高级 35](#_Toc463470481)

[3.9.1 gitignore忽略文件 35](#_Toc463470482)

[3.9.2 比较差异 35](#_Toc463470483)

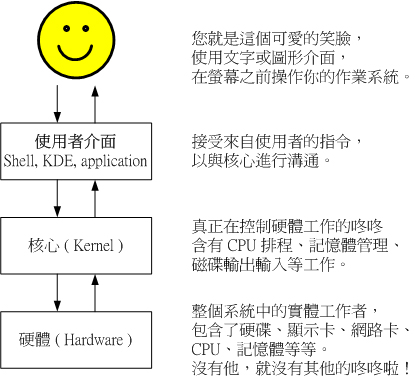
[3.9.3 更新仓库 36](#_Toc463470484)

[3.10 其它 37](#_Toc463470485)

# Shell和vi

## 什么是shell

在计算机科学中，Shell俗称壳，用来区别于Kernel（核），是指“提供使用者使用界面”的软件（命令解析器）。它类似于[DOS](http://baike.baidu.com/view/365.htm)下的command和后来的cmd.exe。它接收用户命令，然后调用相应的应用程序。



## shell分类

1、图形界面shell：通过提供友好的可视化界面，调用相应应用程序，如windows系列操作系统，Linux系统上的图形化应用程序GNOME、KDE等。

2、命令行shell：通过键盘输入特定命令的方式，调用相应的应用程序，如windows系统的cmd.exe、Windows PowerShell，Linux系统的Bourne shell ( sh)、Bourne Again shell ( bash)等。

## 认识bash这个shell

在window系统下使用bash，需要一个软件，这个软件模拟集成了bash大部分命令。

各个 shell 的功能都差不多， Linux 默认使用 bash ，所以我们主要学习bash的使用。

**1、bash命令格式**

命令 [-options] [参数]，如：tar zxvf demo.tar.gz

查看帮助：命令 --help

**2、bash常见命令**

pwd (Print Working Directory) 查看当前目录

cd (Change Directory) 切换目录，如 cd /etc

ls (List) 查看当前目录下内容，如 ls -al

mkdir (Make Directory) 创建目录，如 mkdir blog

touch 创建文件，如 touch index.html

cat 查看文件全部内容，如 cat index.html

more/less 查看文件，如more /etc/passwd、less /etc/passwd

rm (remove) 删除文件，如 rm index.html、rm -rf blog

rmdir (Remove Directory) 删除文件夹，只能删除空文件夹，不常用

mv (move) 移动文件或重命名，如 mv index.html ./demo/index.html

cp (copy) 复制文件，cp index.html ./demo/index.html

head 查看文件前几行，如 head -5 index.html

tail 查看文件后几行 –n –f，如 tail index.html、tail -f -n 5 index.html

tab 自动补全，连按两次会将所有匹配内容显示出来

history 查看操作历史

> 和 >>重定向，如echo hello world! > README.md，>覆盖 >>追加

wget 下载，如wget https://nodejs.org/dist/v4.4.0/node-v4.4.0.tar.gz

tar 解压缩，如tar zxvf node-v4.4.0.tar.gz

curl 网络请求，如curl http://www.baidu.com

whoami 查看当前用户

| 管道符可以将多个命令连接使用，上一次（命令）的执行结果当成下一次（命令）的参数。

grep 匹配内容，一般结合管道符使用

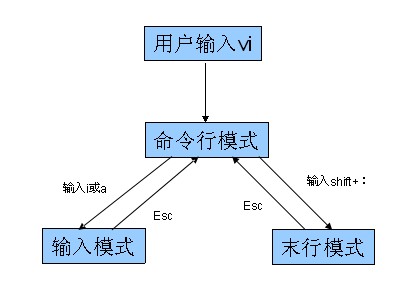
## vi编辑器

如同Windows下的记事本，vi编辑器是Linux下的标配，通过它我们可以创建、编辑文件。它是一个随系统一起安装的文本编辑软件。

**1、三种模式**

vi编辑器提供了3种模式，分别是命令模式、插入模式、底行模式，每种模式下用户所能进行的操作是不一样的。

3种模式的切换如下图所示：



通过上图我们发现，输入模式是不能直接切换到末行模式的，必须要先切回到命令模式（按ESC键）

**2、使用vi编辑器**

a) 打开/创建文件， vi 文件路径

b) 底行模式 :w保存，:w filenme另存为

c) 底行模式 :q退出

d) 底行模式 :wq保存并退出

e) 底行模式 :e! 撤销更改，返回到上一次保存的状态

f) 底行模式 :q! 不保存强制退出

g) 底行模式 :set nu 设置行号

h) 命令模式 ZZ（大写）保存并退出

i) 命令模式 u辙销操作，可多次使用

j) 命令模式 dd删除当前行

k) 命令模式 yy复制当前行

l) 命令模式 p 粘贴内容

m) 命令模式 ctrl+f向前翻页

n) 命令模式 ctrl+b向后翻页

o) 命令模式 i进入编辑模式，当前光标处插入

p) 命令模式 a进入编辑模式，当前光标后插入

q) 命令模式 A进入编辑模式，光标移动到行尾

r) 命令模式 o进入编辑模式，当前行下面插入新行

s) 命令模式 O进入编辑模式，当前行上面插入新行

当我们处在编辑模式的情况下，和我们在Windows编辑器的使用相似。

## SSH

SSH是一种网络协议，用于计算机之间的加密登录。

SSH只是一种协议，存在多种实现，既有商业实现，也有开源实现。本文针对的是[OpenSSH](http://www.openssh.com/)，它是自由软件，应用非常广泛。

如果要在Windows系统中使用SSH，会用到另一种软件[PuTTY](http://www.chiark.greenend.org.uk/~sgtatham/putty)，我们后面用到的Git客户也集成了SSH

格式：ssh user@host

user 代表真实存在的用户host代表要登录的远程计算机

常见有两种加密技术，分别是对称性加密和非对称性加密，SSH属于后者。

对称加密[算法](http://baike.baidu.com/view/7420.htm)在加密和解密时使用的是同一个密钥；而[非对称加密算法](http://baike.baidu.com/view/1490349.htm)需要两个[密钥](http://baike.baidu.com/view/934.htm)来进行加密和解密，这两个秘钥分别是[公开密钥](http://baike.baidu.com/view/1145160.htm)（public key，简称公钥）和私有密钥（private key，简称私钥）。

**工作原理**

公钥和私钥是成对出现，可以通过ssh-keygen -t rsa来创建，既可以通过密钥来加密数据，也可以通过私钥来加密数据，如果是以公钥进行的数据加密，只能与之相对应的私钥才可以解密，相反如果以私钥进行的数据加密，则只能与之对应的公钥才可以将数据进行解密，这样就可以提高信息传递的安全性。

**免密码登录**

我们可以将本地机器上的公钥保存到特定的远程计算机上，这样当我们再次登录访问这台远程计算机时就可以实现免密码登录了。

1、ssh-keygen -t rsa会创建公钥和密钥（默认在用户目录/.ssh目录下）

2、ssh-copy-id user@host添加到对应远程主机的用户目录/.ssh目录下

3、也可以登录远程主机，进入到用户目录/.ssh目录下手动创建authorized\_keys文件，并将自已的公钥粘入该文件。

这部分具体实现细节，参照我的演示有个印象就可以了。

# 版本控制

## 关于版本控制

版本控制（Version Control Systems）是一种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的系统。这个系统可以自动帮我们备份文件的每一次更改，并且可以非常方便的恢复到任意的备份（版本）状态。

举例：我们通常都是手动的重命名一个文件进行备份的，index.html改成index1.html或者index.html.bak等形式，然后这种方式对于单个文件我们还能够管理，但是对于整个项目而言，就会成为噩梦了！！！我们不得不借助于软件来实现。

实现版本控制的软件有很多种类，大致可以分为本地版本控制系统、集中式版本控制系统、分布式版本控制系统。

## 本地版本控制系统

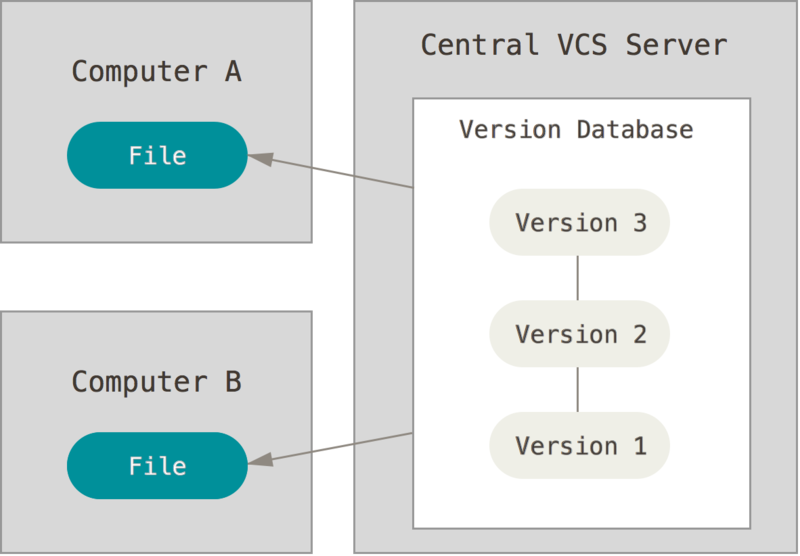
借助软件我们可以记录下文件的每一次修改，如下图所示，文件被修改后，记录下了3个版本，这样我们通过版本控制系统（软件）便可以非常方便的恢复到任意版本。



这种类型的版本控制系统，功能比较单一，比如很难实现多人协同开发，所以现在几乎很少使用了。

## 集中式版本控制系统

实际开发环境，一个项目通常是由多人协作共同完成的，如何让在不同终端上的开发者协同工作成了亟待解决的问题，集中式版本控制系统便应运而生了。它通过单一的集中管理的服务器，保存所有文件的修订版本，协同工作的开发者都通过客户端连到这台服务器，取出最新的文件或者提交更新。其代表为SVN，如下图所示。

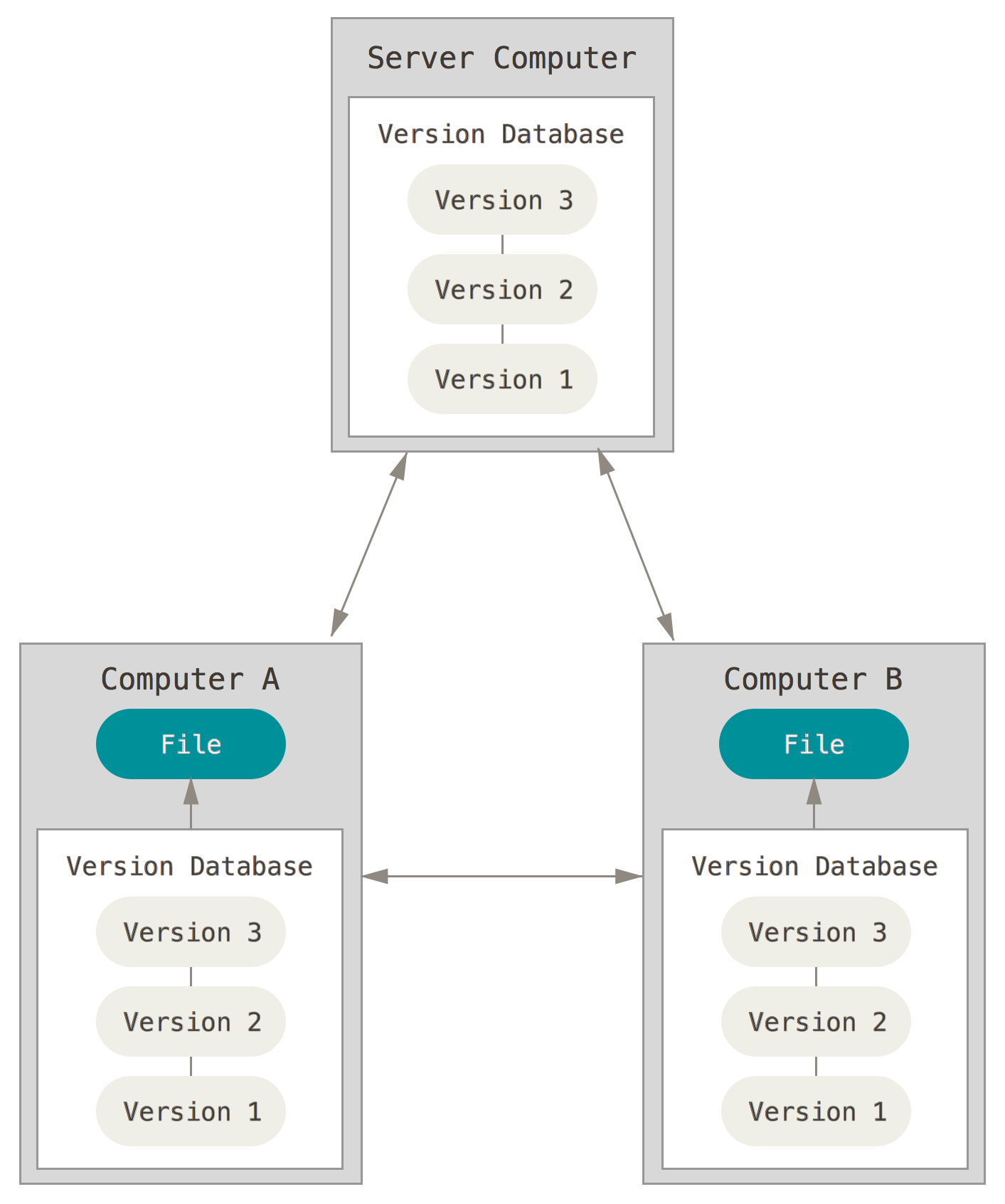


这种方式很好解决了多人协同开发的问题，但是也有一个弊端，如果集中管理的服务器出现故障，将会导致数据（版本）丢失的风险，另外协同开发者从集中服务器中更新数据时，严重依赖网络，如果网络不佳，也给开发带来诸多不便。

## 分布式版本控制系统

分布式版本控制系统，则不需要中央服务器，每个协同开发者都拥有一个完整的版本库，这么一来，任何协同开发者用的服务器发生故障，事后都可以用其它协同开发者本地仓库恢复。

由于版本库在本地计算机，也便不再受网络影响了。如果要将本地的修改，推送给其它协同开发者，还需要一台共享服务器，所有开发者通过这台共享服务器同步和更新数据。如下图所示。



分布式版本控制系统弥补了前面两种版本控制系统的缺陷，成为了版本控制的首选方案。其代表就是Git。

# Git

## Git安装

**Window安装**

<http://git-scm.com/download/win>下载Git客户端软件，和普通软件安装方式一样。

**Linux安装**

CentOS发行版：sudo yum install git

Ubuntu发行版：sudo apt-get install git

**Mac安装**

打开Terminal直接输入git命令，会自动提示，按提示引导安装即可。

## Git工作原理

为了更好的学习Git，我们们必须了解Git管理我们文件的3种状态，分别是已提交（committed）、已修改（modified）和已暂存（staged），由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。

Git仓库目录是Git用来保存项目的元数据和对象数据库的地方。 这是Git 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。

工作目录是对项目的某个版本独立提取出来的内容。这些从Git仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。

暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在Git仓库目录中。有时候也被称作“索引”（Index），不过一般说法还是叫暂存区域。



基本的Git工作流程如下：

1、在工作目录中修改文件。

2、暂存文件，将文件的快照放入暂存区域。

3、提交文件，找到暂存区域的文件，将快照永久性存储到Git仓库目录。

## Git本地仓库

Git本地仓库指的是开发者开发设备中的仓库。

### Git基础

命令行方式：任意目录（建议开发根目录）右键 > Git Bash Here

**1、配置用户**

配置用户的意义在于记录开发者信息，以便在版本控制记录开发者的操作行为，如lion于2016-08-24解决了一个bug。

git config --global user.name "自已的名字"

git config --global user.email "自已的邮箱地址"

--global 配置当前用户所有仓库

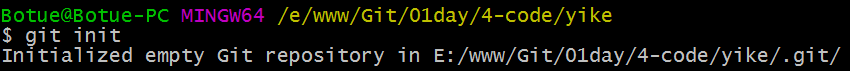
--system 配置当前计算机上所有用户的所有仓库

注：配置用户只需要执行1次，可以重复使用。

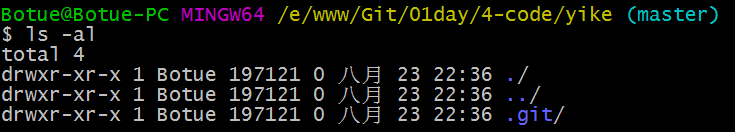
**2、初始化仓库**

我们如果想要利用git进行版本控制，需要将现有项目初始化为一个仓库，或者将一个已有的使用git进行版本控制的仓库克隆到本地。

a) git init

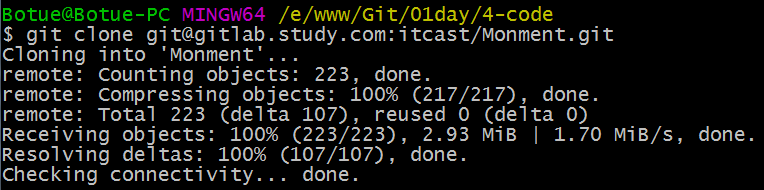


git init只是创建了一个名为.git的隐藏目录，这个目录就是存储我们历史版本的仓库，ls -al 可以查看。

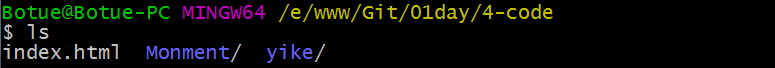


b) 假如公司已有项目用了Git，那我们就利用克隆

git clone 仓库地址



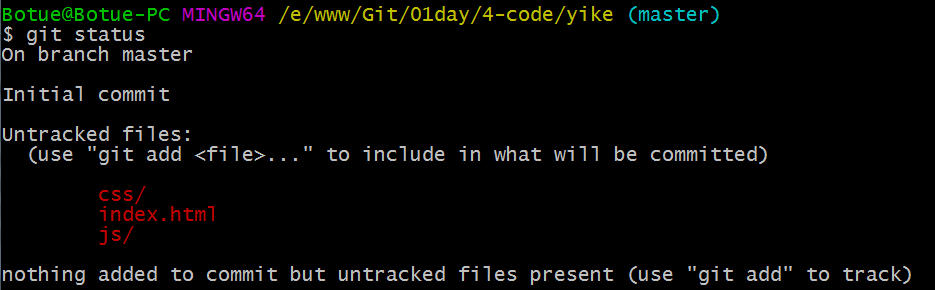
执行完这个命令，会在当前目录下生成一个Monment目录（默认和仓库名称相同），这个便是已有一个使用Git管理的项目。



**3、查看文件状态**

初始化仓库后便可以进行开发了，进入到刚刚创建好并初始为仓库的目录，添加我们开发需要的文件。

通过git status可以检测当前仓库文件的状态

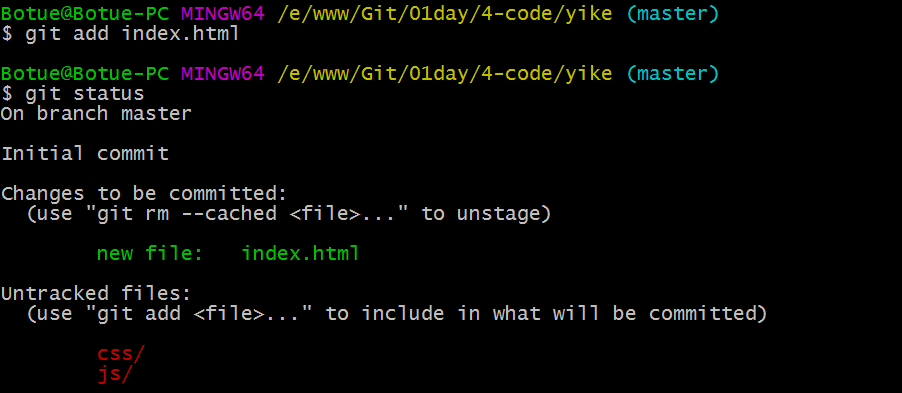


注：git会忽略空的目录

**4、添加文件到暂存区**

假设经过一段时间的开发后，需要把已开发的部分存起来，使用git add 添加到暂存区。

git add 文件名/ 文件路径 “\*”或-A代表所有



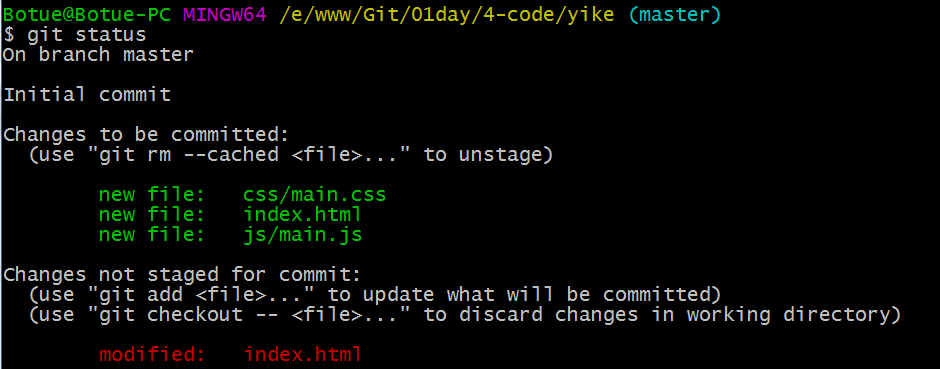
放到暂存区的文件被标记成了绿色，等待提交。

注：颜色是工具给添加的，目的是增加可读性并不是git统一的。

5、撤销更改

继续我们的开发

再次git status可以再次查看仓库状态



说明index.html再次被修改了，并被标记了红色。

又经过一段时间后发现新开发的部分有Bug，想要回到之前状态，可以使用git checkout 文件名。



注：从暂存区还原原到工作区

**5、提交文件**

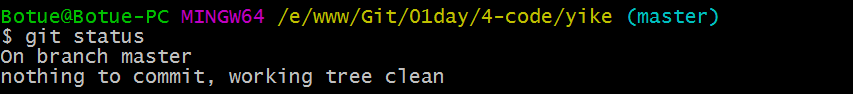
经过一个相对较长阶段开发或者一个功能开发完成了，就可以提交到本地仓库了，永久保存了。

git commit -m '备注信息'



将暂存区被标记成绿色的文件，全部提交到本地仓库存储。

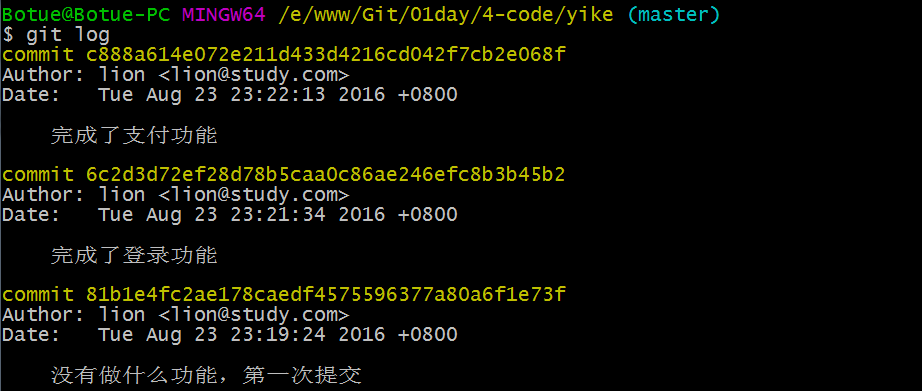
这时git status查看状态



没有什么可提交的，变的很干净。

**6、查看提交历史**

反反复复开发了很多的功能了，通过git log查看一下提交的历史。



我们可以查看到一次次提交记录

commit 81b1e4fc2ae178caedf4575596377a80a6f1e73f

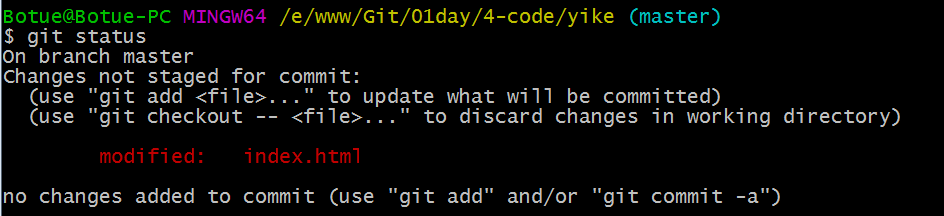
代表一次提交的唯一ID，一般称为SHA值。傻？

注：按键盘q键退出。

**7、再次检测仓库文件状态**

隔了好些天后，继续开发

git status 查看状态



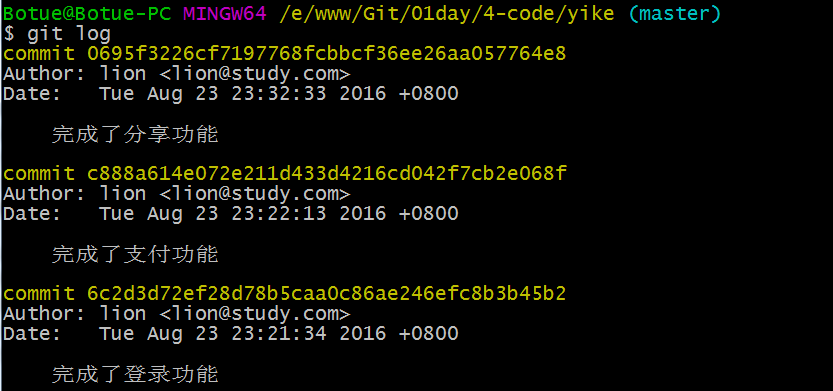
又提示有修改，等待重新添加到暂存区。

**8、重新添加暂存区然后提交**



**9、再次查看历史**

git log 可查到所有提交历史



这时可以查看到更多提交历史。

这时关掉所有目录甚至关机！

**10、恢复上一次提交的状态**

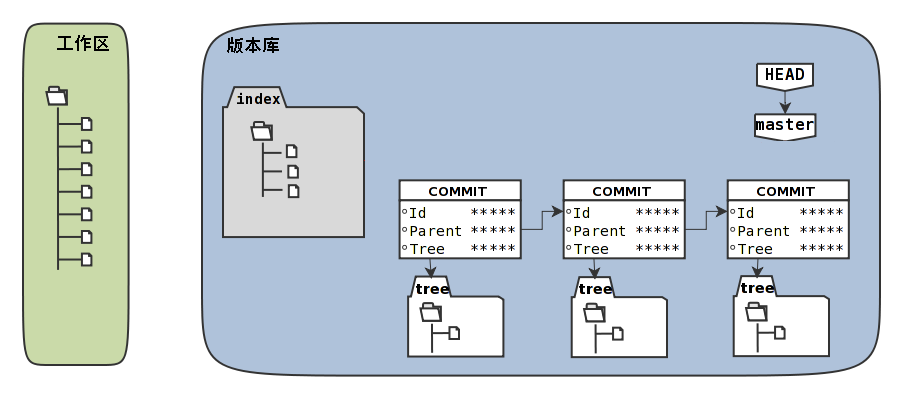
通过SHA值可以回到之前某一次的提交（时光倒流）

git reset --hard c888a614e072e2这样便回到了支付功能的状态

git log再次查看发现最后一次提交成了支付功能了



仓库示意图

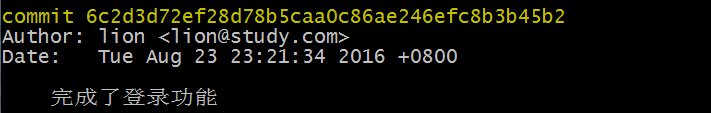


### Git分支

在我们的现实开发中，需求往往是五花八门的，同时开发个需求的情况十分常见，比如当你正在专注开发一个功能时，突然有一个紧急的BUG需要你来修复，这个时候我们当然是希望在能够保存当前任务进度，再去修改这个BUG，等这个BUG修复完成后再继续我们的任务。如何实现呢？

通过Git创建分支来解决实际开发中类似的问题。

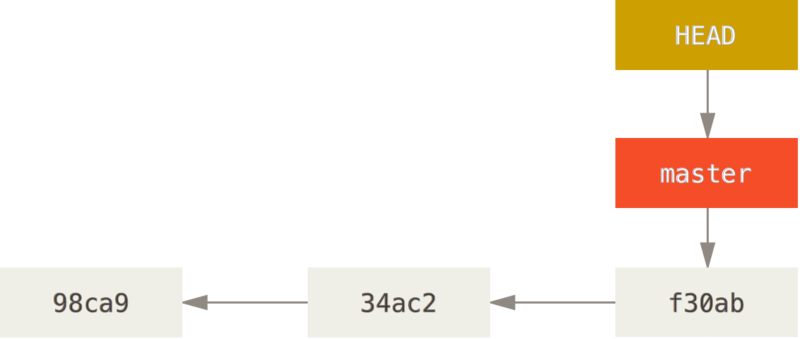
在Git的使用过程中一次提交称为历史记录（版本），并且会生成一个唯一的字符串，如下图



这个串可以代表某一个历史版本（实际使用只取前面几位就可以），

值得注意的是所有的提交（commit）实际上都是在分支（branch）的基础上进行的。

如下图所示：



当我们在初始化仓库的时候（实际上是产生第1次提交时），Git会默认帮我们创建了一个master的分支，并且有指针（HEAD）指到了末端。

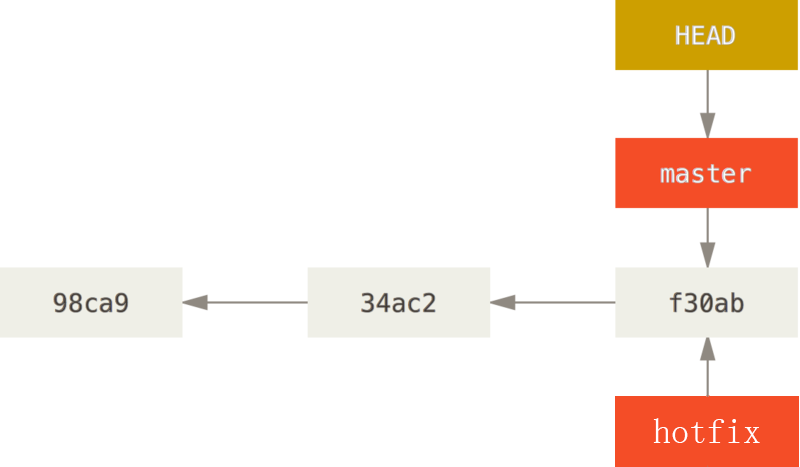
指针（HEAD）用来标明当前处于哪个分支的哪个版本，如上图指的处于master分支的最后1个版本。

我们也可以创建自已的分支

**1、创建分支**

git branch hotfix

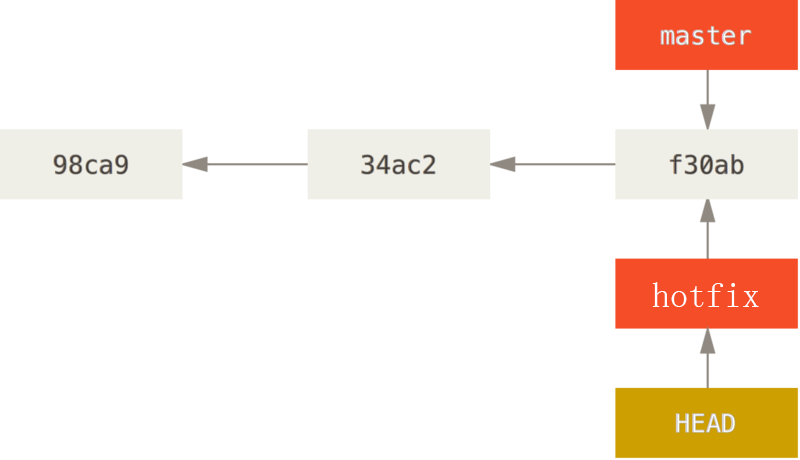
新的分支会在当前分支原有历史版本的结点上进行创建，我称其为子分支如下图



新建的子分支会继承父分支的所有提交历史。

**2、切换分支**

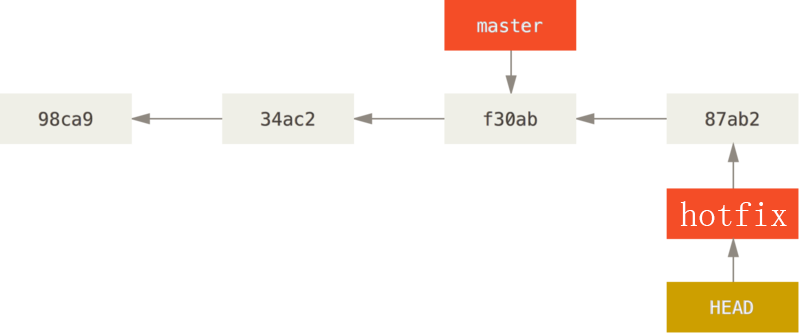
git checkout hotfix



我们发现HEAD现在又指向了hotfix的末端。

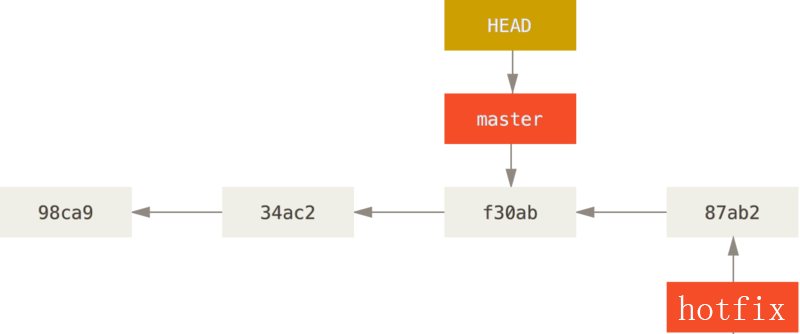
**3、再次提交操作**

修改bug后，提交



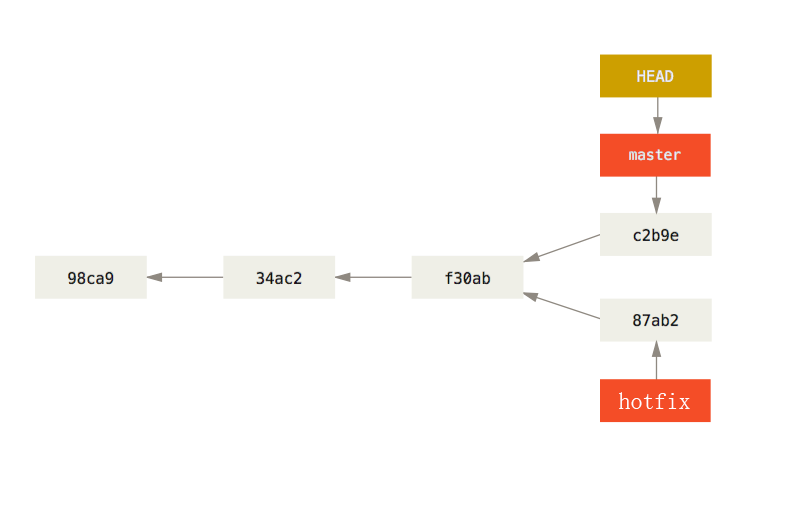
这次的提交历史版本就会记录在hotfix这个分支上了，并且HEAD伴随hotfix在移动。

**4、当我们再次切回到master时**



当我们切换回master后，HEAD指向了master分支的末端，并且我们观察发现我们的文件内容还是原来的“模样”。

**5、继续之前的开发**



总结：当我们git checkout branchname时，HEAD会自动指向对应分支的末端，工作目录中的源码也会随之发生改变。

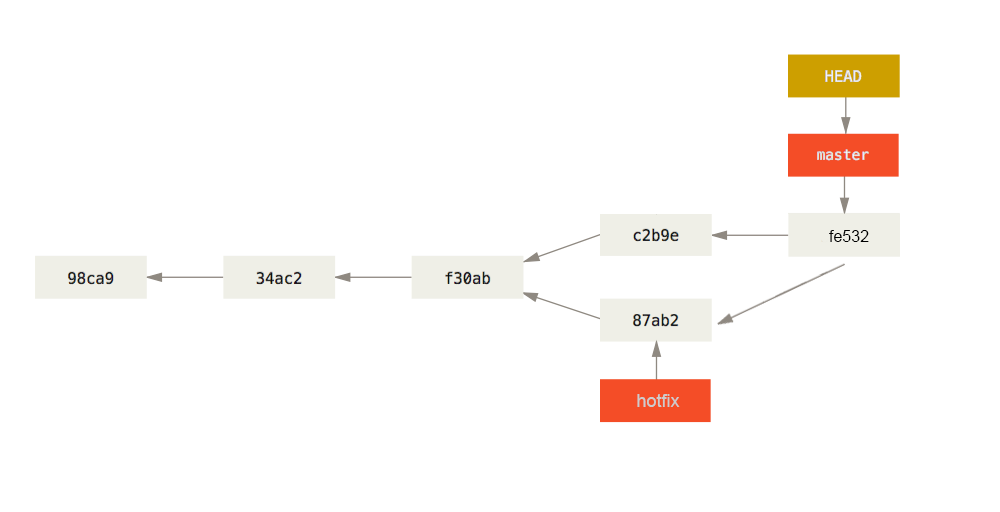
这个时候我们就在hotfix这个分支上修复了这个BUG，而我们原来在master分支上的操作并未受到影响。

思考一个问题：

现在master这个分支上是否包含了hotfix的修复呢？

实际上从上图可以看出这时的master分支并没有包含有hotfix的修复。

**6、合并（融合）分支**

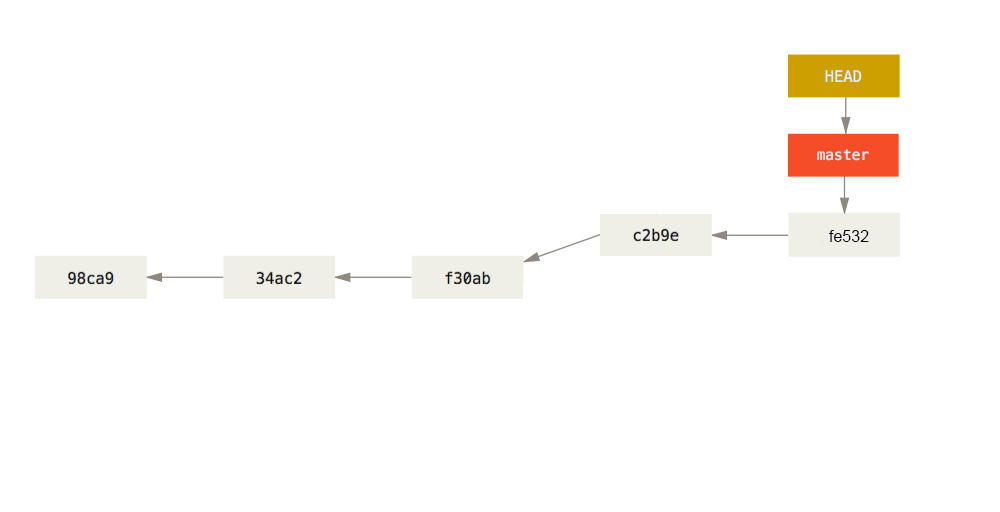


这时master会有两个父结点了，master便包含了hotfix里的修复了

**7、删除分支**

git branch -d hotfix

这时用来修复BUG创建的hotfix分支已经没有用处了，我们可以将它删除。



## Git远程（共享）仓库

通过上面学习我们可以很好的管理本地版本控制了，可是如果我们下班回到家里突然来了灵感觉得有部分代码可以优化，如果能接着公司电脑上的代码继续写该有多好呀！另一种情形，假设项目比较大，不同的功能模块由不同的开发人员完成，不同模块儿之间又难免会依赖关系，这时如果我们的代码互相合并（融合）该有多好呀！所有模块开发完毕后，需要整合到一起，要能做到准确无误该有多好呀！

借助一个远程仓库，大家可以共享代码、历史版本等数据，便可以解决以上遇到的所有问题，在学习远程仓库前我们先来学习git clone path这个命令。

**3、创建共享仓库**

Git要求共享仓库是一个以.git结尾的目录。

mkdir repo.git 创建以.git结尾目录

cd repo.git 进入这个目录

git init --bare 初始化一个共享仓库，也叫裸仓库 注意选项--bare



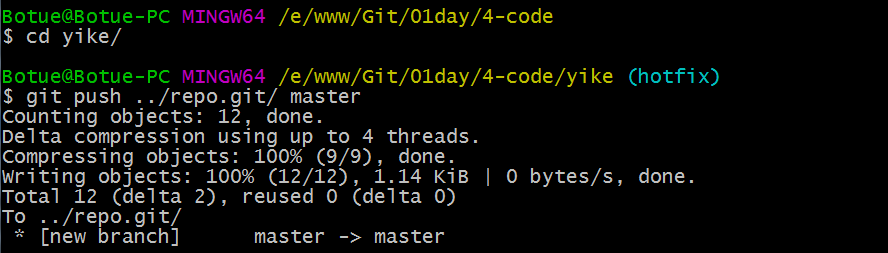
这样我们就建好了一个共享的仓库，但这时这个仓库是一个空的仓库，并且不允在这个仓库中进行任何修改。

**4、向共享仓库共享（同步）内容**

将自已开发的项目同步到这个目录中，其它开发者就可以共享你开发的项目了。

1、进入到yike目录

2、git push ../repo.git master

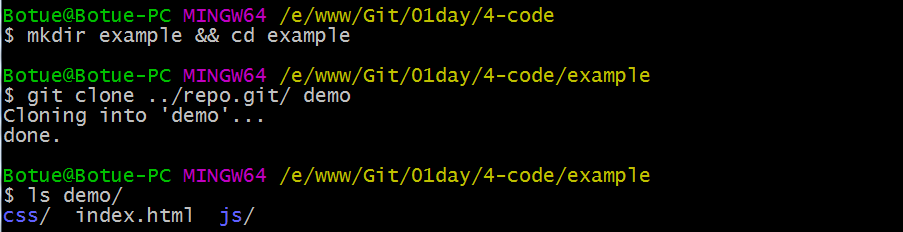


这样便把yike中的项目同步进了repo.git中。

**5、从共享仓库里取出内容**

1、新创建一个目录（模拟另一个开发者）

2、git clone ./repo.git demo



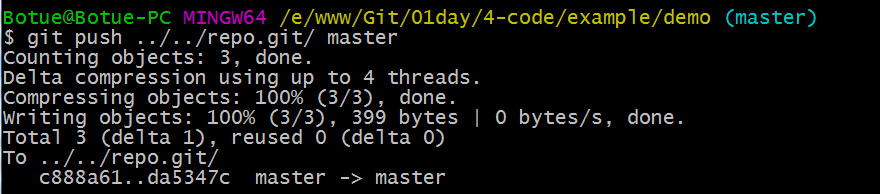
通过repo.git共享仓库，我们轻松得到了一个yike的副本

**6、通过demo仓库向repo.git共享内容**

进入到demo里，我们做一些修改

cd demo

git push ../repo.git master



**7、在360仓库从repo.git获取共享的内容**

cd yike

git pull ../repo.git master



奇迹似乎发生了，我们轻松的将demo仓库里的内容，通过repo.git共享给了yike仓库。

惊喜不断，问题也总是不断，我们发现我们这个共享的仓库只是放到了本地的，其它人是没有办法从我们这个共享仓库共享内容的！！！

然而现实是，办法总是有的！！！！

我们把这个共享的仓库放到一台远程服务器上，问题不就解决了吗？

## gitHub和gitLab

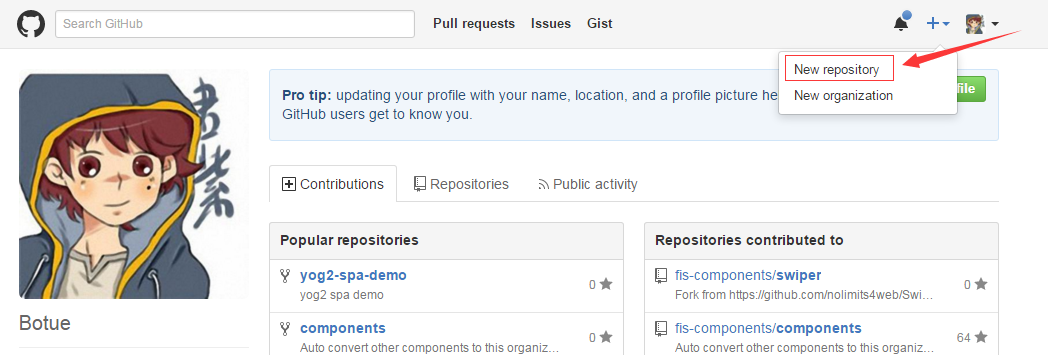
如果我们熟悉服务器的话，我们完全可以将上述的步骤在我们的远程服务器上进行操作，然后再做一些登录权限的设置，就可非常完美的搭建一个共享服务器了。其实为了更好的管理我们的仓库，一些第三方机构开发出了Web版仓库管理程序，通过Web界面形式管理仓库。

gitHub关于它的名气与意义，大家可以自行查阅，我们这里介绍它的使用

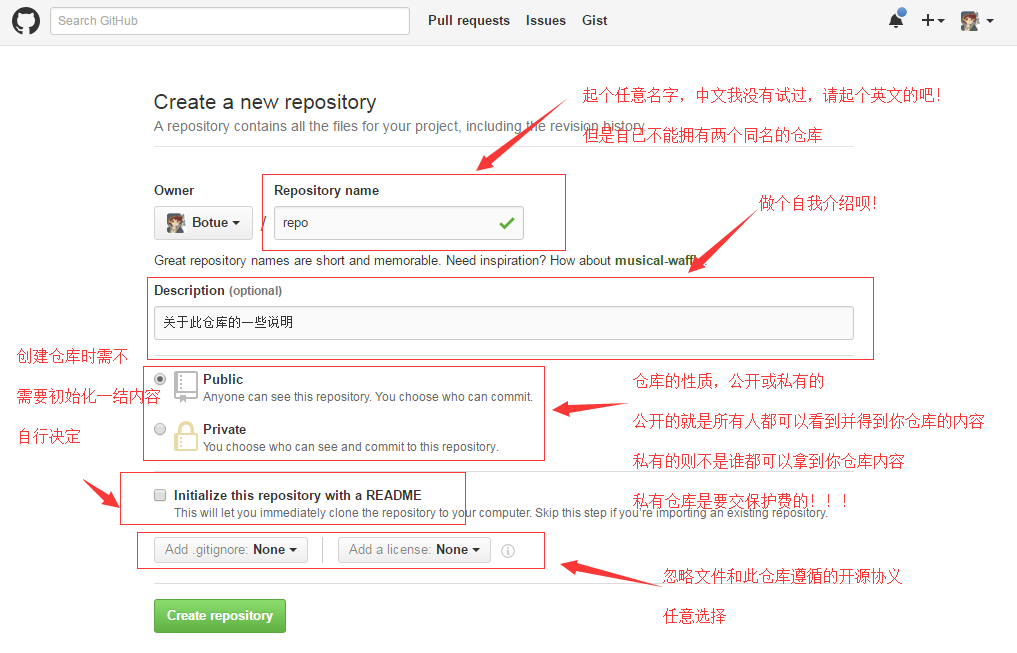
**1、注册账号并完善资料**

自行注册略过

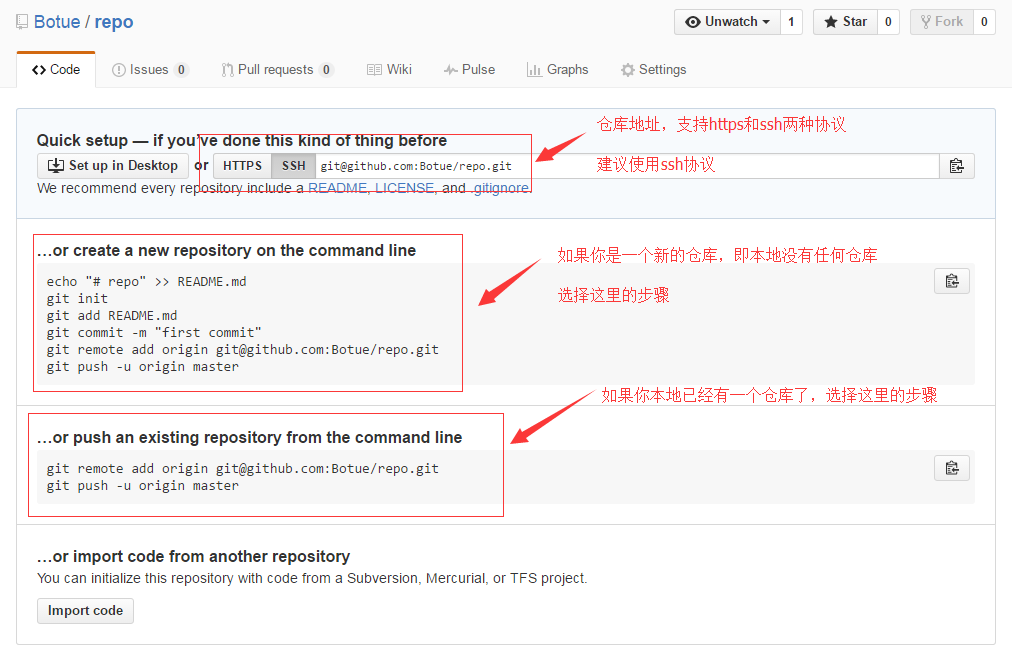
**2、创建共享仓库**



**3、填写仓库资料**



**4、共享仓库**

****

远程地址特别长，我们可以给他起一个别名

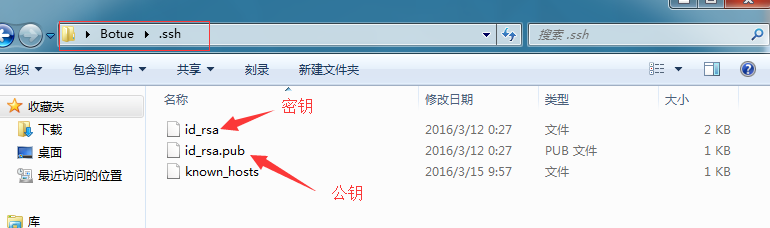
git remote add origin git@github.com:Botue/repo.git

这样origin 就代表 git@github.com:Botue/repo.git

当我们通过git clone 从共享仓库获内容时，会自动帮我们添加origin到对应的仓库地址，例如：git clone git@github.com:Botue/repo.git 会自动添加origin 对应 git@github.com:Botue/repo.git

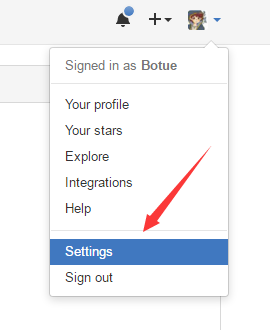
**5、生成密钥**

ssh-keygen -t rsa 然后一路回车，这里会在当前用户生成了一个.ssh的文件夹

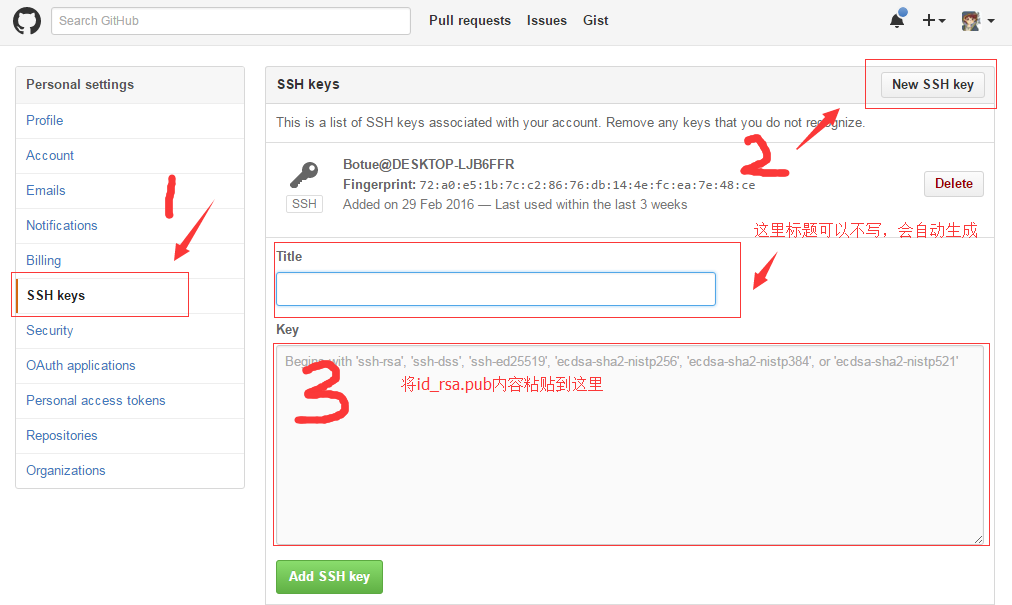


将id\_rsa.pub公钥的内容复制

打开gitHub的个人中心



打到SSH keys



到此我们便可以通过gitHub 提供的Web界面来管理我们的仓库了。

我们发现通过gitHub管理仓库实在是太方便了，可是只能免费使用公开仓库，自已公司的代码当然不能公开了，可是私有仓库又是需要交“保护费”的，无耐国人还是比较喜欢免费的，网络界总是有很多雷峰的，比如gitLab!!!

gitLab也是一个可以通过Web界面管理仓库的网站程序，我们可以把它架设到公司自已的服务器上，实现仓库私有化，这也是大部分公司通常采用的方法，其使用方法与gitHub十分相似。

我将闲置电脑配置成了一台服务器，上面架设了gitLab程序，我们接下来的练习全部会在gitLab上进行演示。

省略很多内容.....

## 命令汇总

git config配置本地仓库

常用git config --global user.name、git config --global user.email

git config --list查看配置详情

git init 初始一个仓库，添加--bare可以初始化一个共享（裸）仓库

git status 可以查看当前仓库的状态

git add“文件” 将工作区中的文件添加到暂存区中，其中file可是一个单独的文件，也可以是一个目录、“\*”、-A

git commit -m '备注信息' 将暂存区的文件，提交到本地仓库

git log 可以查看本地仓库的提交历史

git branch查看分支

git branch“分支名称” 创建一个新的分支

git checkout“分支名称” 切换分支

git checkout -b deeveloper 他健并切到developer分支

git merge“分支名称” 合并分支

git branch -d “分支名称” 删除分支

git clone “仓库地址”获取已有仓库的副本

git push origin “本地分支名称:远程分支名称”将本地分支推送至远程仓库，

git push origin hotfix（通常的写法）相当于

git push origin hotfix:hotfix

git push origin hotfix:newfeature

本地仓库分支名称和远程仓库分支名称一样的情况下可以简写成一个，即git push “仓库地址” “分支名称”，如果远程仓库没有对应分支，将会自动创建

git remote add “主机名称” “远程仓库地址”添加远程主机，即给远程主机起个别名，方便使用

git remote 可以查看已添加的远程主机

git remote show “主机名称”可以查看远程主机的信息

## GitLab完整演示

没错，Git非常强大！

但是，如果我们的分支不加以规范管理，也有可能适得其反！

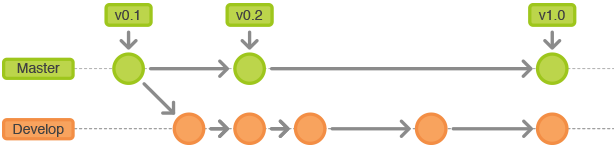
1、不要有太多的树杈（子分支）

2、要有一个“稳定分支”，即master分支不要轻意被修改

3、要有一个开发分支（developer），保证master分支的稳定性

4、所有的功能分支（feature）从developer创建

5、所有功能开发完成后新建发布分支（release）







## 冲突解决

假如两个开发同时改到同一文件的同一段内容会发生什么事情呢？

这时就会就会产生冲突了，当冲突产生后，需要开发者进行协商确认冲突的原因，然后将冲突代码删除重新提交就可以了。

## Git高级

熟悉掌握以上操作，基本上是可以满足日常开的需要的，但是在解决一些特殊问题时，就又需要我们能够掌握更多的命令。

### gitignore忽略文件

在项目根目录下创建一个.gitignore文件，可以将不希望提交的罗列在这个文件里，如项目的配置文件、node\_modules等

https://github.com/github/gitignore

### 比较差异

当内容被修改，我们无法确定修改哪些内容时，可以通过git diff来进行差异比较。

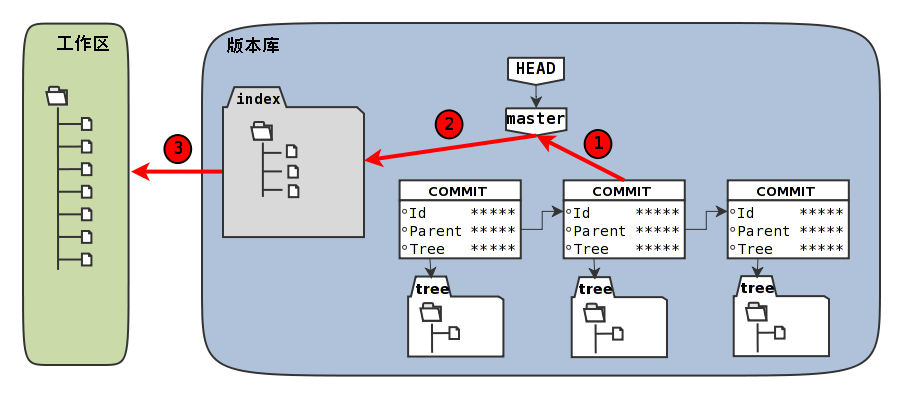
git difftool 比较的是工作区和暂存的差异

git difftool “SHA”比较与特定提交的差异

git difftool “SHA”“SHA”比较某两次提交的差异

git difftool 分支名称 比较与某个分支的差异

**2、回滚（撤销）操作**



HEAD 默认指向当前分支的“末端”，即最后的一次提交，但是我们通过git reset 可以改变HEAD的指向。

**看情况解释（稍微复杂一些，理解就好）**

1、git reset

--hard 工作区会变、历史(HEAD)会变， 暂存区也变

--soft 只会变历史(HEAD)

--mixed（默认是这个选项）历史(HEAD)会变、暂存区也变，工作区不变

2、git checkout

git checkout SHA -- "某个文件"，代表只是从SHA这个版中取出特定的文件，

和git reset 是有区别的，reset 重写了历史，checkout 则没有。

### 更新仓库

在项目开发过程中，经常性的会遇到远程（共享）仓库和本地仓库不一致，我们可以通过git fetch 命令来更新本地仓库，使本地仓库和远程（共享）仓库保持一致。

git fetch “远程主机”

或者

git fetch “远程主机” “分支名称”

我们要注意的是，利用git fetch 获取的更新会保存在本地仓库中，但是并没有体现到我们的工作目录中，需要我们再次利用git merge来将对应的分支合并（融合）到特定分支。如下

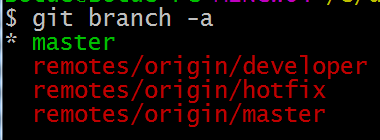
git pull origin 某个分支， 上操作相当于下面两步

git fetch

git merge origin/某个分支

**问题：如何查看远程主机上总共有多少个分支？**

git branch -a 便可以查看所有(本地+远程仓库)分支了



## 其它

删除远程分支git push origin --delete 分支名称

删除远程分支git push origin :分支名称

**AngularJS**

目录

[目录 39](#_Toc463689938)

[第1章 介绍 41](#_Toc463689939)

[1.1 特点 41](#_Toc463689940)

[1.2 下载 41](#_Toc463689941)

[1.3 体验AngularJS 42](#_Toc463689942)

[1.4 MVC 42](#_Toc463689943)

[第2章 模块化 43](#_Toc463689944)

[2.1 定义应用 43](#_Toc463689945)

[2.2 定义模块 44](#_Toc463689946)

[2.3 定义控制器 44](#_Toc463689947)

[第3章 指令 45](#_Toc463689948)

[3.1 内置指令 46](#_Toc463689949)

[3.2 自定义指令 47](#_Toc463689950)

[第4章 数据绑定 47](#_Toc463689951)

[4.1 单向绑定 47](#_Toc463689952)

[4.2 双向绑定 48](#_Toc463689953)

[4.3 相关指令 49](#_Toc463689954)

[第5章 作用域 50](#_Toc463689955)

[5.1 根作用域 50](#_Toc463689956)

[5.2 子作用域 50](#_Toc463689957)

[第6章 过滤器 51](#_Toc463689958)

[6.1 内置过滤器 51](#_Toc463689959)

[6.2 自定义过滤器 51](#_Toc463689960)

[第7章 依赖注入 52](#_Toc463689961)

[7.1 推断式注入 52](#_Toc463689962)

[7.2 行内注入 53](#_Toc463689963)

[第8章 服务 53](#_Toc463689964)

[8.1 内建服务 54](#_Toc463689965)

[8.2 自定义服务 56](#_Toc463689966)

[第9章 模块加载 57](#_Toc463689967)

[9.1 配置块 58](#_Toc463689968)

[9.2 运行块 59](#_Toc463689969)

[第10章 路由 59](#_Toc463689970)

[10.1 SPA 59](#_Toc463689971)

[10.2 路由 60](#_Toc463689972)

[10.2.1 使用 61](#_Toc463689973)

[10.2.1 路由参数 62](#_Toc463689974)

[第11章 其它 63](#_Toc463689975)

[11.1 jQuery 63](#_Toc463689976)

[11.2 bower 63](#_Toc463689977)

# 介绍

AngularJS是一款由Google公司开发维护的前端MVC框架，其克服了[HTML](http://baike.baidu.com/view/692.htm)在构建应用上的诸多不足，从而降低了开发成本提升了开发效率。

## 特点

AngularJS与我们之前学习的jQuery是有一定的区别的，jQuery更准确来说只一个类库（类库指的是一系列函数的集合）以DOM做为驱动（核心），而AngularJS则一个框架（诸多类库的集合）以数据和逻辑做为驱动（核心）。

框架对开发的流程和模式做了约束，开发者遵照约束进行开发，更注重的实际的业务逻辑。

AngularJS有着诸多特性，最为核心的是：模块化、双向数据绑定、语义化标签、依赖注入等。

与之类似的框架还有BackBone、KnockoutJS、Vue、React等。

## 下载

1、通过[AngularJS](https://angularjs.org/)官网下载，不过由于国内特殊的国情，需要翻墙才能访问。

2、通过npm下载，npm install angular

3、通过bower下载，bower install angular

bower是什么？

## 体验AngularJS

见代码示例1-01.html

## MVC

MVC是一种开发模式，由模型（Model）、视图（View）、控制器（Controller）3部分构成，采用这种开发模式为合理组织代码提供了方便、降低了代码间的耦合度、功能结构清晰可见。

模型（Model）一般用来处理数据（读取/设置），一般指操作数据库。

视图（View）一般用来展示数据，比如通过HTML展示。

控制器（Controller）一般用做连接模型和视图的桥梁。



通过ThinkPHP来演示后端MVC的执行流程，其重点在于理解。

MVC更多应用在后端开发程序里，后被引入到前端开发中，由于受到前端技术的限制便有了一些细节的调整，进而出现了很多MVC的衍生版（子集）如MVVM、MVW、MVP、MV\*等。

注：做为初学可以不必过于在意这些概念。

# 模块化

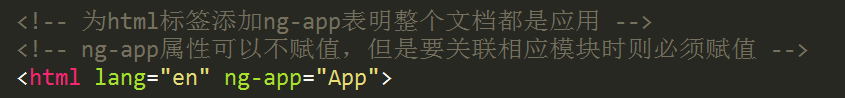
使用AngularJS构建应用（App）时是以模块化（Module）的方式组织的，即将整个应用划分成若干模块，每个模块都有各自的职责，最终组合成一个整体。

采用模块化的组织方式，可以最大程度的实现代码的复用，可以像搭积木一样进行开发。



## 定义应用

通过为任一HTML标签添加ng-app属性，可以指定一个应用，表示此标签所包裹的内容都属于应用（App）的一部分。



## 定义模块

AngularJS提供了一个全局对象angular，在此全局对象下存在若干的方法，其中angular.module()方法用来定义一个模块。



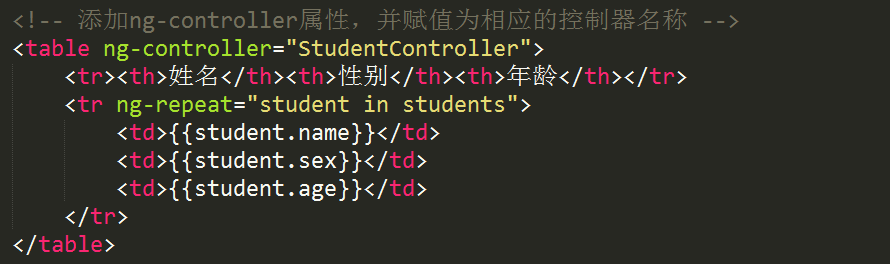
注：应用（App）其本质也是一个模块（一个比较大的模块）。

## 定义控制器

控制器（Controller）作为连接模型（Model）和视图（View）的桥梁存在，所以当我们定义好了控制器以后也就定义好了模型和视图。



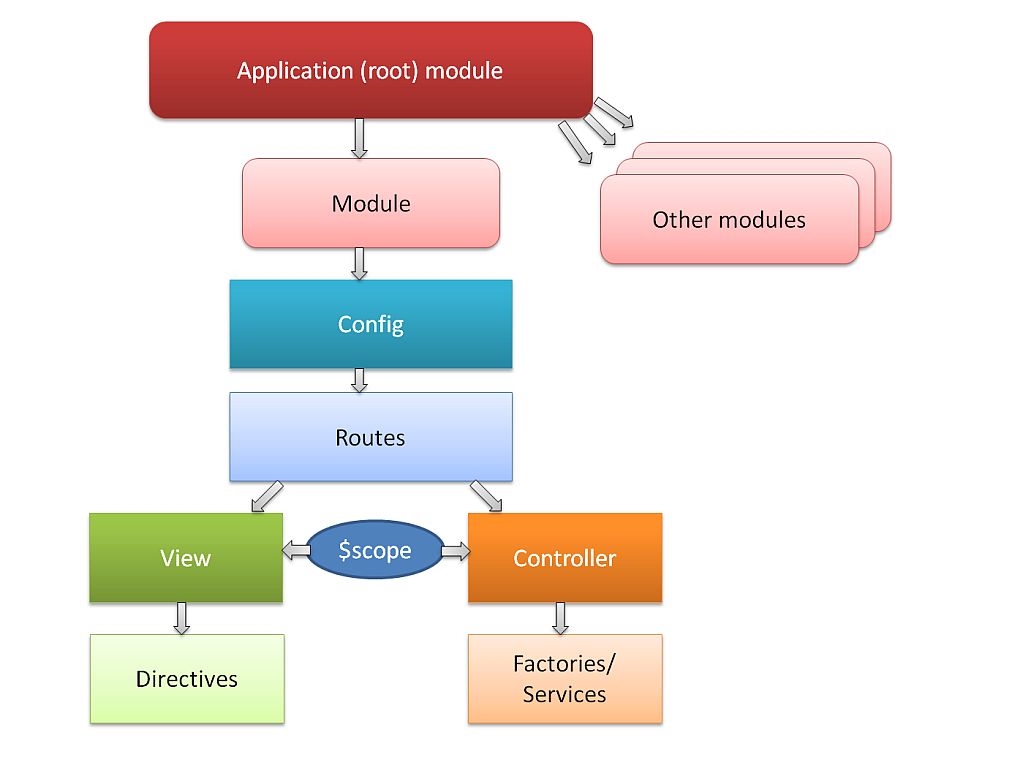
模型（Model）数据是要展示到视图（View）上的，所以需要将控制器（Controller）关联到视图（View）上，通过为HTML标签添加ng-controller属性并赋值相应的控制器（Controller）的名称，就确立了关联关系。



见代码示例2-01.html

以上步骤就是AngularJS最基本的MVC工作模式。

下图是AngularJS的结构，学习AngularJS会围绕下图的结构展开。



# 指令

HTML在构建应用（App）时存在诸多不足之处，AngularJS通过扩展一系列的HTML属性或标签来弥补这些缺陷，所谓指令就是AngularJS自定义的HTML属性或标签，这些指令都是以ng-做为前缀的，例如ng-app、ng-controller、ng-repeat等。

## 内置指令

ng-app 指定应用根元素，至少有一个元素指定了此属性。

ng-controller 指定控制器

ng-show控制元素是否显示，true显示、false不显示

ng-hide控制元素是否隐藏，true隐藏、false不隐藏

ng-if控制元素是否“存在”，true存在、false不存在

ng-src增强图片路径

ng-href增强地址

ng-class控制类名

ng-include引入模板

ng-disabled表单禁用

ng-readonly表单只读

ng-checked单/复选框表单选中

ng-selected下拉框表单选中

见代码示例3-01.html

注：后续学习过程中还会介绍其它指令。

## 自定义指令

AngularJS允许根据实际业务需要自定义指令，通过angular全局对象下的directive方法实现。



见代码示例3-02.html

# 数据绑定

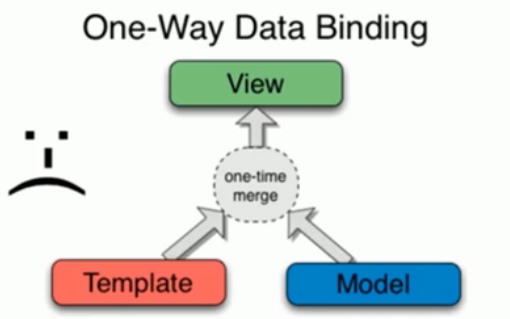
AngularJS是以数据做为驱动的MVC框架，所有模型（Model）里的数据经由控制器（Controller）展示到视图（View）中。

所谓数据绑定指的就是将模型（Model）中的数据与相应的视图（View）进行关联，分为单向绑定和双向绑定两种方式。

## 单向绑定

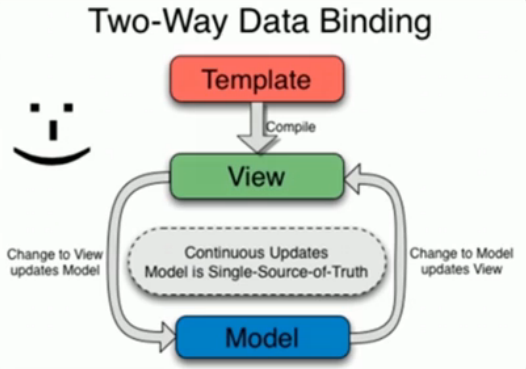
单向数据绑定是指将模型（Model）数据，按着写好的视图（View）模板生成HTML标签，然后追加到DOM中显示，如之前所学的artTemplate 模板引擎的工作方式。

如下图所示，只能模型（Model）数据向视图（View）传递。



## 双向绑定

双向绑定则可以实现模型（Model）数据和视图（View）模板的双向传递，如下图所示。



## 相关指令

在AngularJS中通过“{{}}”和ng-bind指令来实现模型（Model）数据向视图模板（View）的绑定，模型数据通过一个内置服务$scope来提供，这个$scope是一个空对象，通过为这个对象添加属性或者方法便可以在相应的视图（View）模板里被访问。

注：“{{}}”是ng-bind的简写形式，其区别在于通过“{{}}”绑定数据时会有“闪烁”现象，添加ng-cloak也可以解决“闪烁”现象，通过ng-bind-template可以绑定多个数据。

见代码示例4-01.html

通过为表单元素添加ng-model指令实现视图（View）模板向模型（Model）数据的绑定。

见代码示例4-02.html

通过ng-init可以初始化模型（Model）也就是$scope。

见代码示例4-03.html

AngularJS对事件也进行了扩展，无需显式的获取DOM元素便可以添加事件，易用性变的更强。通过在原有事件名称基础上添加ng-做为前缀，然后以属性的形式添加到相应的HTML标签上即可。如ng-click、ng-dblclick、ng-blur等。

见代码示例4-04.html

通过ng-repeat可以将数组或对象数据迭代到视图模板中，ng-switch、on、ng-switch-when可以对数据进行筛选。

见代码示例4-05.html

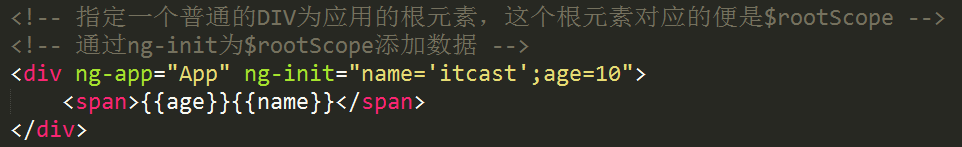
# 作用域

通常AngularJS中应用（App）是由若干个视图（View）组合成而成的，而视图（View）又都是HTML元素，并且HTML元素是可以互相嵌套的，另一方面视图都隶属于某个控制器（Controller），进而控制器之间也必然会产生嵌套关系。

每个控制器（Controller）又都对应一个模型（Model）也就是$scope对象，不同层级控制器（Controller）下的$scope便产生了作用域。

## 根作用域

一个AngularJS的应用（App）在启动时会自动创建一个根作用域$rootScope，这个根作用域在整个应用范围（ng-app所在标签以内）都是可以被访问到的。



见代码示例5-01.html

## 子作用域

通过ng-controller指令可以创建一个子作用域，新建的作用域可以访问其父作用域的数据。

见代码示例5-02.html

# 过滤器

在AngularJS中使用过滤器格式化展示数据，在“{{}}”中使用“|”来调用过滤器，使用“:”传递参数。

## 内置过滤器

1、currency将数值格式化为货币格式

2、date日期格式化，年（y）、月（M）、日（d）、星期（EEEE/EEE）、时（H/h）、分（m）、秒（s）、毫秒（.sss），也可以组合到一起使用。

3、filter在给定数组中选择满足条件的一个子集，并返回一个新数组，其条件可以是一个字符串、对象、函数

4、json将Javascrip对象转成JSON字符串。

5、limitTo取出字符串或数组的前（正数）几位或后（负数）几位

6、lowercase将文本转换成小写格式

7、uppercase将文本转换成大写格式

8、number数字格式化，可控制小位位数

9、orderBy对数组进行排序，第2个参数可控制方向

见代码示例6-01.html

## 自定义过滤器

除了使用AngularJS内建过滤器外，还可以根业务需要自定义过滤器，通过模块对象实例提供的filter方法自定义过滤器。



见代码示例6-02.html

# 依赖注入

AngularJS采用模块化的方式组织代码，将一些通用逻辑封装成一个对象或函数，实现最大程度的复用，这导致了使用者和被使用者之间存在依赖关系。

所谓依赖注入是指在运行时自动查找依赖关系，然后将查找到依赖传递给使用者的一种机制。

常见的AngularJS内置服务有$http、$location、$timeout、$rootScope等

## 推断式注入

没有明确声明依赖，AngularJS会将函数参数名称当成是依赖的名称。



这种方式会带来一个问题，当代码经过压缩后函数的参数被压缩，这样便会造成依赖无法找到。

见代码示例7-01.html

## 行内注入

以数组形式明确声明依赖，数组元素都是包含依赖名称的字符串，数组最后一个元素是依赖注入的目标函数。



推荐使用这种方式声明依赖

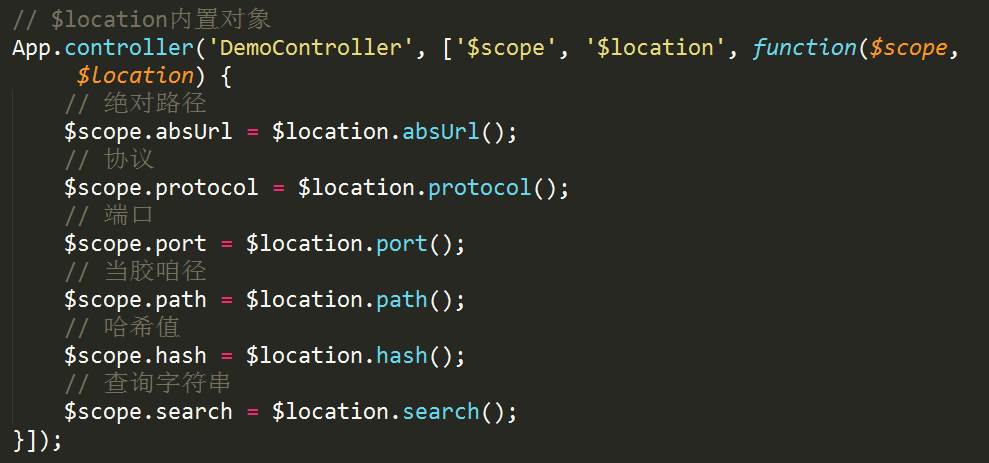
见代码示例7-02.html

# 服务

服务是一个对象或函数，对外提供特定的功能。

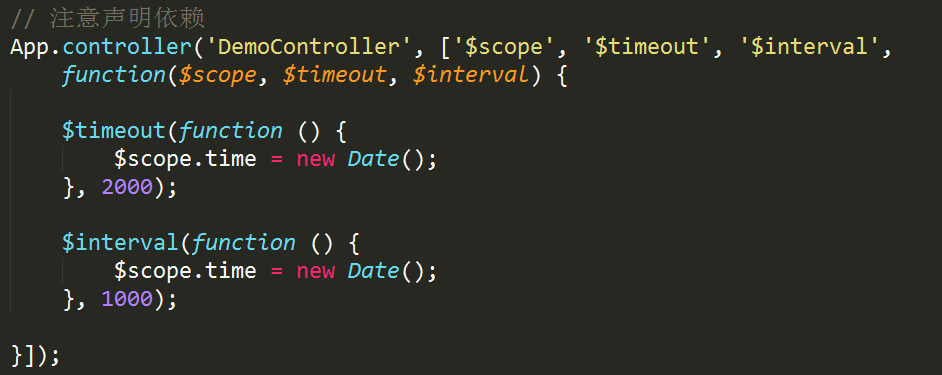
## 内建服务

1、$location是对原生Javascript中location对象属性和方法的封装。



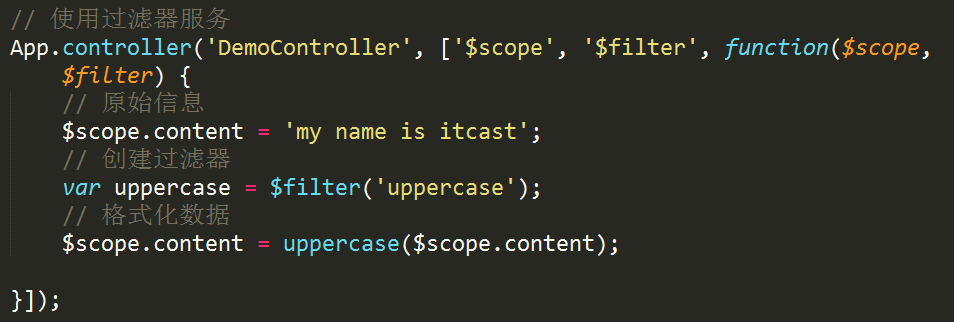
见代码示例8-01.html

2、$timeout&$interval对原生Javascript中的setTimeout和setInterval进行了封装。



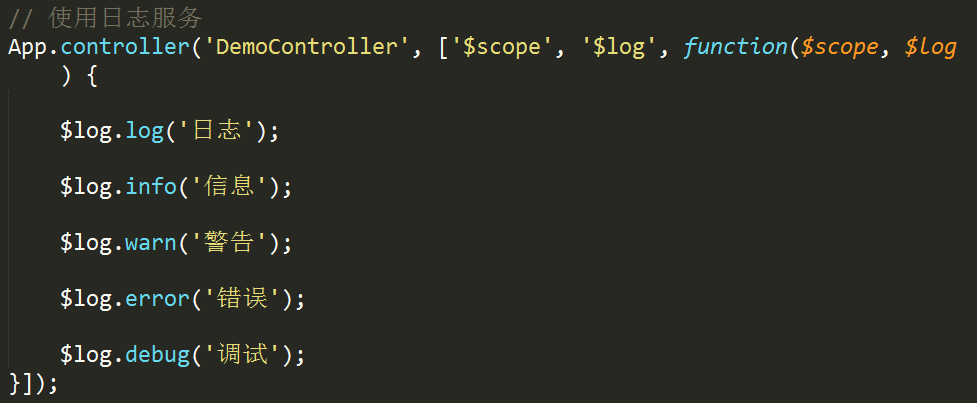
见代码示例8-02.html

3、$filter在控制器中格式化数据。



见代码示例8-03.html

4、$log打印调试信息



见代码示例8-04.html

5、$http用于向服务端发起异步请求。



同时还支持多种快捷方式如$http.get()、$http.post()、$http.jsonp。

注：各参数含义见代码注释。

见代码示例8-05.html

## 自定义服务

通过上面例子得知，所谓服务是将一些通用性的功能逻辑进行封装方便使用，AngularJS允许将自定义服务。

1、factory方法



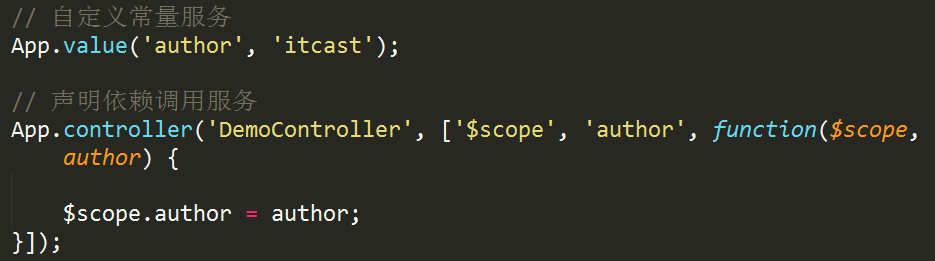
见代码示例8-06.html

2、service方法



见代码示例8-07.html

3、value方法定义常量

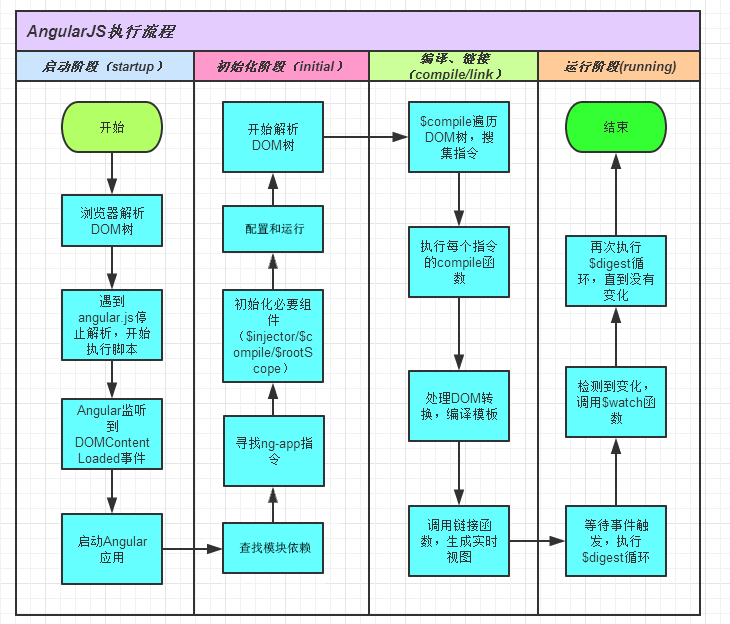


见代码示例8-08.html

在介绍服务时曾提到服务本质就是一个对象或函数，所以自定义服务就是要返回一个对象或函数以供使用。

# 模块加载

AngularJS模块可以在被加载和执行之前对其自身进行配置。我们可以在应用的加载阶段配置不同的逻辑。



## 配置块

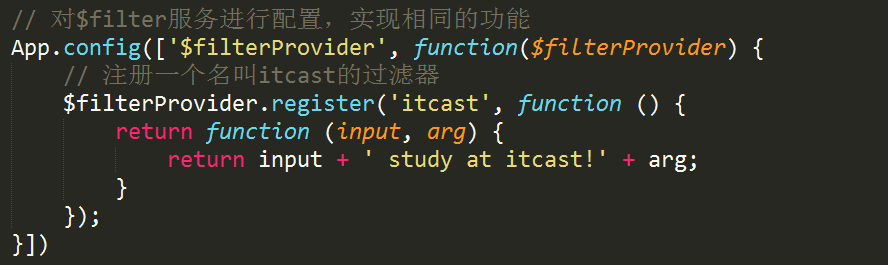
1、通过config方法实现对模块的配置，AngularJS中的服务大部分都对应一个“provider”，用来执行与对应服务相同的功能或对其进行配置。

比如$log、$http、$location都是内置服务，相对应的“provider”分别是$logProvider、$httpProvider、$locationPorvider。

下图以$log为例进行演示，修改了配置



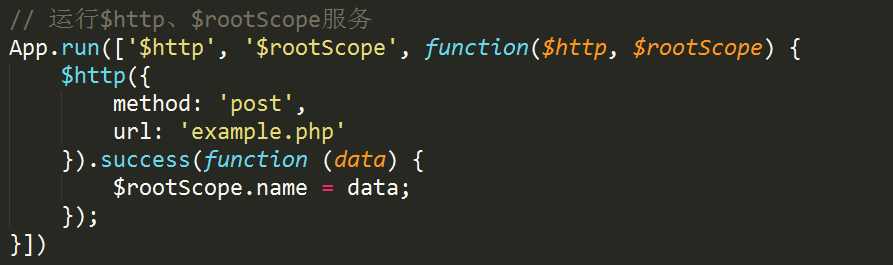
下图以$filter为例进行演示，实现相同功能



见代码示例9-01.html

## 运行块

服务也是模块形式存在的对且对外提供特定功能，前面学习中都是将服务做为依赖注入进去的，然后再进行调用，除了这种方式外我们也可以直接运行相应的服务模块，AngularJS提供了run方法来实现。



不但如此，run方法还是最先执行的，利用这个特点我们可以将一些需要优先执行的功能通过run方法来运行，比如验证用户是否登录，未登录则不允许进行任何其它操作。

见代码示例9-02.html

注：此知识点意在了解AngularJS的加载机制。

# 路由

一个应用是由若个视图组合而成的，根据不同的业务逻辑展示给用户不同的视图，路由则是实现这一功能的关键。

## SPA

SPA（Single Page Application）指的是通单一页面展示所有功能，通过Ajax动态获取数据然后进行实时渲染，结合CSS3动画模仿原生App交互，然后再进行打包（使用工具把Web应用包一个壳，这个壳本质上是浏览器）变成一个“原生”应用。

在PC端也有广泛的应用，通常情况下使用Ajax异步请求数据，然后实现内容局部刷新，局部刷新的本质是动态生成DOM，新生成的DOM元素并没有真实存在于文档中，所以当再次刷新页面时新添加的DOM元素会“丢失”，通过单页面应可以很好的解决这个问题。

## 路由

在后端开发中通过URL地址可以实现页面（视图）的切换，但是AngularJS是一个纯前端MVC框架，在开发单页面应用时，所有功能都在同一页面完成，所以无需切换URL地址（即不允许产生跳转），但Web应用中又经常通过链接（a标签）来更新页面（视图），当点击链接时还要阻止其向服务器发起请求，通过锚点（页内跳转）可以实现这一点。

实现单页面应用需要具备：

a、只有一页面

b、链接使用锚点

见代码实例10-01.html

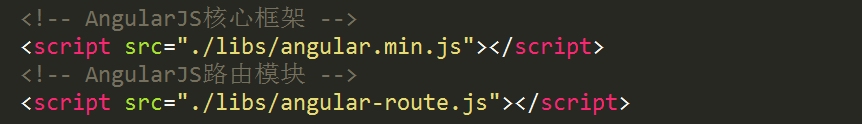
通过上面的例子发现在单一页面中可以能过hashchange事件监听到锚点的变化，进而可以实现为不同的锚点准不同的视图，单页面应用就是基于这一原理实现的。

AngularJS对这一实现原理进行了封装，将锚点的变化封装成路由（Route）,这是与后端路由的根本区别。

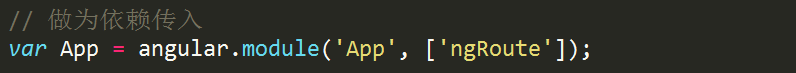
在1.2版前路由功能是包含在AngularJS核心代码当中，之后的版本将路由功能独立成一个模块，[下载angular-route.js](https://code.angularjs.org/)

### 使用

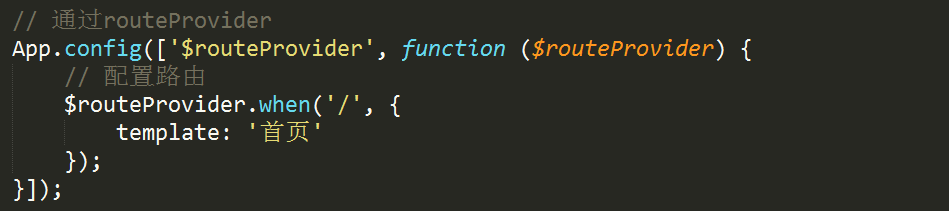
1、引入angular-route.js



2、实例化模块（App）时，当成依赖传进去（模块名称叫ngRoute）。

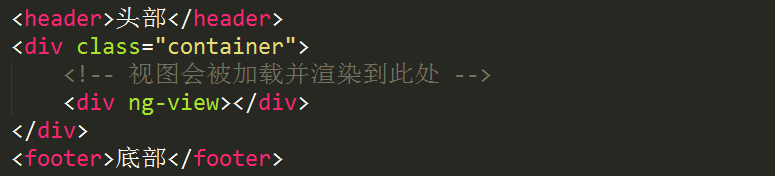


3、配置路由模块



4、布局模板

通过ng-view指令布局模板，路由匹配的视图会被加载渲染到些区域。



### 路由参数

1、提供两个方法匹配路由，分别是when和otherwise，when方法需要两个参数，otherwise方法做为when方法的补充只需要一个参数，其中when方法可以被多次调用。

2、第1个参数是一个字符串，代表当前URL中的hash值。

3、第2个参数是一个对象，配置当前路由的参数，如视图、控制器等。

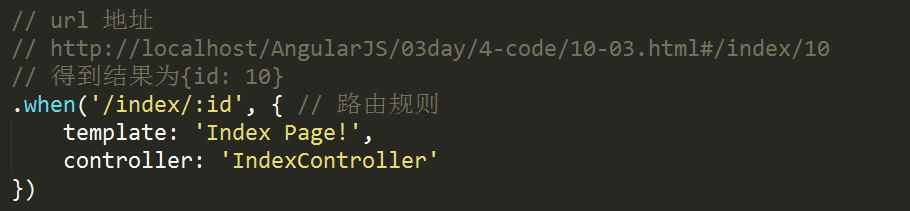
a、template 字符串形式的视图模板

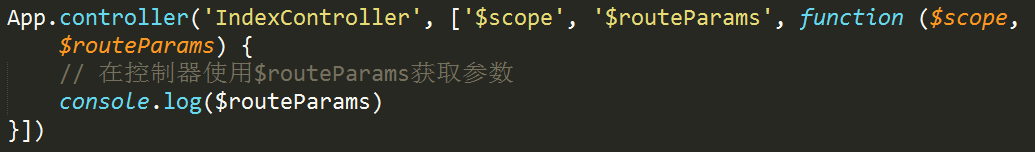
b、templateUrl 引入外部视图模板

c、controller 视图模板所属的控制器

d、redirectTo跳转到其它路由

4、获取参数，在控制中注入$routeParams可以获取传递的参数





# 其它

## jQuery

在没有引入jQuery的前提下AngularJS实现了简版的jQuery Lite，通过angular.element不能选择元素，但可以将一个DOM元素转成jQuery对象，如果引提前引入了jQuery则angular.element则完全等于jQuery。

见代码示例11-01.html

## bower

基于NodeJS的一个静态资源管理工具，由twitter公司开发维，解决大型网站中静态资源的依赖问题。

1、依赖NodeJS环境和git工具。

2、npm install -g bower安装bower

3、bower search 查找资源信息

4、bower install 安装（下载）资源，通过#号可以指定版本号

5、bower info 查看资源信息

6、bower uninstall 卸载（删除）资源

7、bower init初始化，用来记录资源信息及依赖。

**项目构建**

目录

[目录 2](#_Toc464248662)

[第1章 项目构建 3](#_Toc464248663)

[1.1 Gulp 3](#_Toc464248664)

[1.1.1 Gulp基础 3](#_Toc464248665)

[1.1.2 Gulp工作原理 5](#_Toc464248666)

[1.1.3 Gulp API 5](#_Toc464248667)

[1.1.4 常用Gulp插件 6](#_Toc464248668)

# 项目构建

多个开发者共同开发一个项目，每位开发者负责不同的模块，这就会造成一个完整的项目实际上是由许多的“代码版段”组成的；

使用less、sass等一些预处理程序，降低CSS的维护成本，最终需要将这些预处理程序进行解析；

合并css、javascript，压缩html、css、javascript、images可以加速网页打开速度，提升性能；

这一系列的任务完全靠手动完成几乎是不可能的，借助构建工具可以轻松实现。

所谓构建工具是指通过简单配置就可以帮我们实现合并、压缩、校验、预处理等一系列任务的软件工具。

常见的构建工具包括：Grunt、Gulp、F.I.S（百度出品）、webpack

## Gulp

Gulp是基于Nodejs开发的一个构建工具，借助[gulp插件](http://gulpjs.com/plugins/)可以实现不同的构建任务，以其简洁的配置和卓越的性能成为目前主流的构建工具。

全局安装 npm install -g gulp

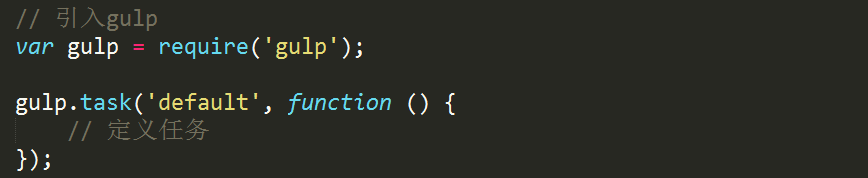
### Gulp基础

1、本地安装gulp

进入项目根目录执行npm install gulp --save-dev（添加--save-dev会在package.json记录依赖关系）。

2、任务清单

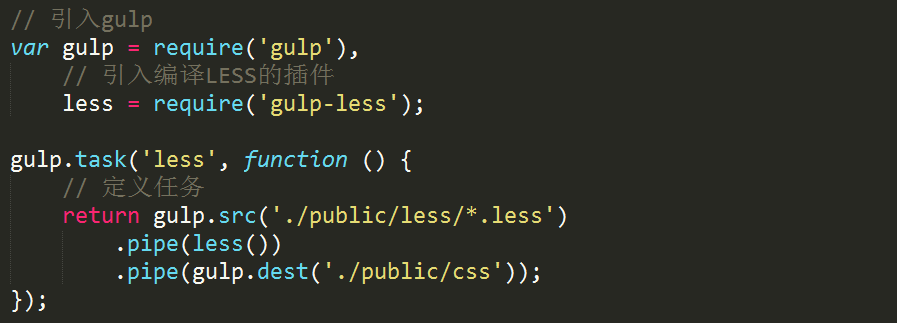
在项目根目录中创建gulpfile.js（这是一个配置文件）



3、定义任务

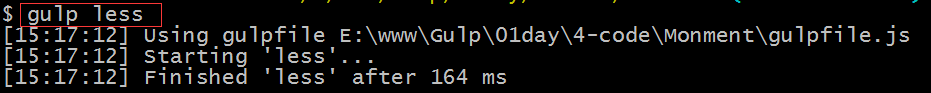
在gulpfile.js定义构建任务，如压缩、合并，Gulp自身并不执行任何任务，是通过调用具体插件来完成的。

以编译LESS为例，安装npm install gulp-less，如下图定义任务



4、执行任务

输入命令 gulp less



这样我们的LESS文件便会编译成CSS了。

### Gulp工作原理



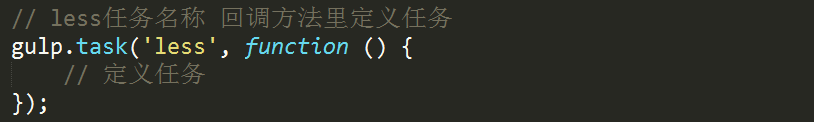
通过不同的插件实现构建任务，Gulp只是按着配置文件调用执行了这些插件。

### Gulp API

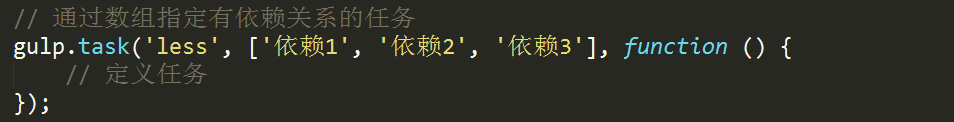
Gulp是基于NodeJS的，通过require可以引入一个NodeJS的包（模块），其作用类似于浏览器中的script标签引入资源，被引入的包存放在node\_modules目录下。

引入gulp包（模块）后返回一个对象，习惯赋值给变量gulp，通过该对象提供的方法（API）完成任务的配置。

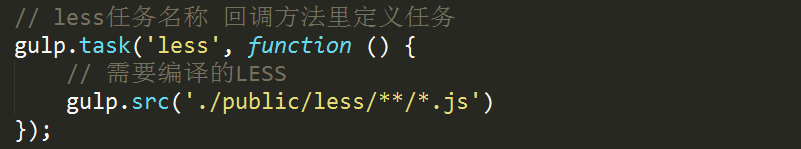
1、gulp.task() 定义各种不同的任务，如下图有两个参数



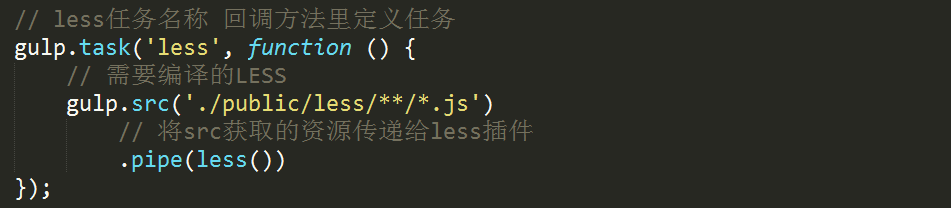
不同任务间存在依赖关系时，可以指定依赖，如下图



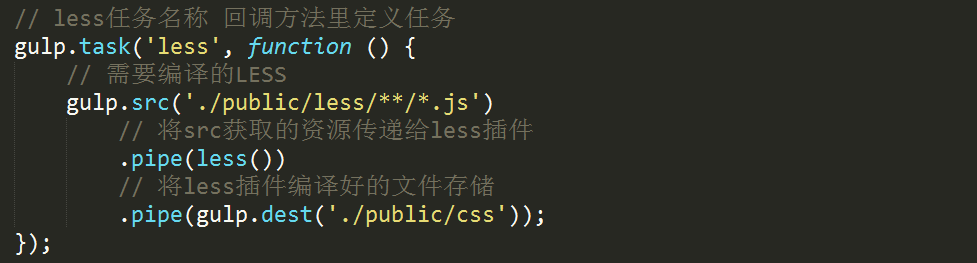
2、gulp.src() 需要构建资源的路径，字符串或数组（可以正则方式书写）



2、gulp.pipe() 管道，将需要构建的资源“输送”给插件。



3、gulp.dest() 构建任务完成后资源存放的路径（会自动创建）



4、gulp.watch()

### 常用Gulp插件

gulp-less 编译LESS文件

gulp-autoprefixer 添加CSS私有前缀

gulp-cssmin 压缩CSS

gulp-rname重命名

gulp-imagemin 图片压缩

gulp-uglify 压缩Javascript

gulp-concat 合并

gulp-htmlmin 压缩HTML

gulp-rev 添加版本号

gulp-rev-collector 内容替换

gulp-useref

gulp-if