GIT版本控制工具

# 1.GIT入门教程(廖雪峰)

GIT是分布式的版本管理工具，理论上来讲不需要中央服务器的参与，但一般为了资料共享的方便性总是会人为构造一个或几个服务器。

## 1.1.安装GIT

#apt-get install git

完成安装后，需要设置自己的名字和email地址，以作为这台机器上所有git仓库的配置信息，设置命令：

$git config –global user.name “Your name”

$git config –golbal user.email “Your [email@xxx.com](mailto:email@xxx.com)”

## 1.2.创建版本库

版本库又叫做仓库，英文名是repository。简单的理解，它就是一个目录，这个目录里的所有文件都可以被git管理起来，修改、删除、还原等操作都可以管理。

$mkdir test\_git

$cd test\_git

$git init

执行了该命令后，将在该文件夹中自动创建一个“.git”的隐藏文件，这几句是git来管理版本库的；而test\_git就认为这是一个git仓库，且提示你是一个空的仓库(empty git repository)。该文件非特殊情况，不要修改，否则导致git仓库被破坏。

将一个文件添加到git仓库中，需要两步：

$git add file.txt

$git commit –m “Create file.txt and save in git”

commit操作将文件提交到仓库，“-m”后的描述信息将记录在版本控制信息中，以供后续、其他人员查看文件记录。

可以使用“git add”命令添加多个文件，使用一次“commit”操作将其一次性的加入到版本控制仓库里。

## 1.3.时光机穿梭

命令：git status可以查看仓库当前的状态；如果使用该命令后，git告知有文件被修改过，可以通过命令“git diff mod\_file\_name”来查看修改的内容。

### 1.3.1.版本回退

要查看git仓库的历史记录，可以通过命令“git log”实现。该命令将记录显示出来，从最近到最远。可以通过加入参数，将主要信息在一行中显示出来以方便查看：“git log –pretty=oneline”。

要进行版本的回退，需要知道当前版本是哪个版本：在git中使用HEAD表示当前版本，也就是最新的提交，上一个版本就是HEAD^，上N个版本是HEAD~N。

可以使用“reset”命令将版本进行回退，例如要回退到上一个版本：“git reset –hard HEAD^”或“git reset –hard HEAD~1”都可以实现。

还可以有更加直接的方式，执行git log命令后，将每一次的版本修改信息显示的同时，还有一个十六进制的数字串，用来表示这个版本的唯一标识，假设这个值是“cb926e7ea50ad11b8f9e909c05226233bf755030”，那么可以通过命令“git reset –hard cb926e7ea50ad11b8f9e909c05226233bf755030”回到这个版本。

但每次都输入这么长的数字显然麻烦，且没有必要，因为每个版本的这个值都是不同的，所以只需要有选择的输入前几个数字，git可以自行查找匹配的版本并进行跳转，例如：“git reset –hard cb926e”。

如果从当前版本返回到了上一个版本，又想重新回到当前版本，就相当于要重返未来，需要使用“git reflog”命令找到“未来”版本的标示符，并通过该值实现跳转。

### 1.3.2.工作区和暂存区

所谓工作区就是存放在个人电脑上的、个人可见的目录，比如上面建立的test\_git文件夹就是一个工作区。

版本库的概念已经描述过，又称作仓库，英文名repository，是工作区中的隐藏目录“.git”。

GIT的版本库里包含了很多东西，最重要的就是称为stage(或者index)的暂存区，还有git自动为我们创建的第一个分支master，以及指向master的一个指针HEAD。

执行“git add”的时候，可以理解为将文件存放到了stage中；而执行“git commit”的时候，则是从stage中将文件传到了master中。

### 1.3.3.撤销修改

命令“git checkout -- filename.txt”可以将该文件在工作区的修改全部撤销，可以分为两种情况：

* 如果该文件在工作区修改了，还没有执行add命令添加到暂存区，那么执行该命令工作区的文件就会退回到与版本库一样的版本；
* 如果工作区修改了后使用add命令添加到了暂存区，那么执行命令就会退回到暂存区的状态；

总体来说，就是让工作区的文件回到最后一次“git commit”或者“git add”时的状态。

对于第一种情况，执行checkout命令就已经将工作区的修改撤销了，由于暂存区没有改变，所以就已经达到了目的；但对于第二种情况，执行该命令后只是达到了暂存区的状态，显然没有达到我们的目的，我们想要将暂存区和工作区的内容都还原到版本库里的。

要达到这个目的，可以首先执行“git reset HEAD filename.txt”命令，将暂存区的文件更新到版本库里的最新版本；再执行checkout命令，就可以将工作区同步暂存区，显然达到了我们的目的。

如果已经将修改的文件上传到了版本库里，想要撤销修改就需要参考“版本回退”里的指令了。

### 1.3.4.删除文件

使用“git rm”命令，可以从版本库里删除指定文件，当然执行完该命令后，同样要执行“git commit”命令才能达到最终效果。

如果在本地工作区里删除了文件，想要从版本库里恢复可以使用“git checkout -- filename”。

“checkout”命令实际上是用版本库里的版本替换工作区的，无论工作区是修改还是删除都可以“一键还原”。

## 1.4.远程仓库

可以自行搭建服务器作为远程仓库，每个人都访问这个服务器；也可以使用免费的github提供的远程仓库。这里讲的是后者。

首先登陆github网站，使用自己的邮箱注册一个账户。由于git仓库和github仓库之间的传输是通过ssh加密的，所以需要做如下设置：

1. 创建本机器的ssh key。在用户主目录下，看是否存在.ssh目录，如果有该目录再确认目录下是否存在id\_rsa和id\_rsa.pub两个文件，如果已经有了可以不做余下操作；否则在终端：$ssh-keygen –t rsa –C “[email@xxx.com](mailto:email@xxx.com)”。正常来说，执行结束后应该在主目录下找到前面说的几个文件。
2. 登录github后，打开account settings—ssh keys页面后：

点击“add ssh key”，填上自己的title，在key文本框里粘贴id\_rsa.pub文件的内容，如下图所示：



点击“add key”后，可见如下图所示的结果：



github允许一个人添加多个key，这样就可以保证每台电脑都可以向github上推送内容。但注意，免费版的github仓库是任何人都可以读的，如果有敏感信息需要保密，最好自行搭建一个中央服务器。

### 1.4.1.添加远程库(本地->远程)

首先在github网站上建立对应的远程仓库：



在点击了“Create a new repo”按钮后，就应该已经成功创建了一个新的仓库。但显然，这时候的远程仓库还是空的， 需要将本地的仓库与其做关联。

* git remote add origin [git@github.com:EricWuJl/test\_git.git](mailto:git@github.com:EricWuJl/test_git.git)：该命令将本地的git仓库与远程仓库进行关联；远程仓库的名字指定是origin(默认指定，可以修改，但不建议修改)；“EricWuJl”是在github网站上注册的账户名，要更改成自己的；
* git push -u origin master：将本地仓库的master分支推送到远程仓库；由于是第一次推送分支，所以加上“-u”参数，建立本地的master分支和远程master分支的关联，以后推送内容的时候该参数可以省略；
* git push origin master：master分支有了改动后，无需“-u”参数即可成功推送到远程仓库；

### 1.4.2.从远程库克隆(远程->本地)

先在github上创建一个仓库gitskills，作为远程仓库：



为了方便这里查看效果，选择的是“initialize this repository with a README”的选项，这样将直接建立一个拥有README.md文件的远程仓库gitskills。之后可以再本地对这个仓库进行克隆：

git clone [git@github.com:EricWuJl/gitskills.git](mailto:git@github.com:EricWuJl/gitskills.git)：该命令将克隆远程仓库的内容到本地仓库；

git支持多种网络协议，包括https等，但是使用ssh将拥有最快的交互速度。

## 1.5.分支管理

### 1.5.1.创建与合并分支

创建git的时候，git将自动创建一个分支：master。这时候，git用master指向最新的提交，再用HEAD指向master，就能很容易的确定当前分支以及其提交点：



每次执行“commit”命令的时候，master分支会向前移动；随着提交的增加，master分支也不断延长。

当创建新分支(例如dev)，git只是新建一个指针dev，它指向master相同的提交，再把HEAD指向dev就说明当前分支在dev上：



经过上面的操作，已经将当前的分支修改为了dev(HEAD指向dev，当前分支就是dev)。在此之后对工作区的修改和提交，将针对dev分支，比如新提交一次的情况：



当dev分支上的工作完成后，可将其合并到master分支上。git直接将master指向dev的当前提交，就完成了合并操作：



一旦确定dev分支已经完成了其使命，可以直接将其删除：



下面介绍实现这些功能的git指令：

* git checkout –b dev：创建分支dev，并将当前分支切换为dev；
  + git branch dev：只创建分支dev；
  + git checkout dev：将当前分支切换到dev；
* git branch：查看分支情况，以“\*”标记的是当前分支；
* git merge dev：合并指定分支到当前分支，这里就是将dev合并到活动的master分支上；默认采用的是fast-forward(快进模式)的合并，速度快，但不是所有情况下都可用；
* git branch –d dev：删除dev分支；

由于git对分支的操作非常快，所以git鼓励开发者使用分支完成任务，合并后再删除分支即可，这种开发更加安全可靠。

### 1.5.2.解决冲突

上面的合并操作，直接使用默认的“快进模式”，git就完成了所有工作。但是，如果出现如下情况：



新建了一个分支feature1，对文件进行了修改；回到master分支后，对文件同样进行了修改。之后执行“merge”命令试图将其合并：

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict in readme.txt

Automatic merge failed; fix conflicts and then commit the result.

可以看到，自动合并失败了，原因是你同时修改了两个版本。由此可见，对于同一个文件，两个分支进行了同一个位置的修改，将导致无法git无法进行自动的版本管理，必须执行手动合并操作：

查看导致冲突的文件readme.txt，发现其内部有用“<<<<<<<”、“=======”、“>>>>>>>”标记的内容，这部分就是不同分支对应的不同修改，将这部分内容手动进行修改后，再次执行“commit”操作，效果图：



这里其实不是执行了理论上的合并，而只是将需要合并的内容进行了修正后，重新在master分支上进行了一次提交。这次修正兼顾了feature1分支的修改，也就达到了合并的效果。

也可以通过命令：git log –graph –pretty=oneline –abbrev-commit，查看分支的合并信息。当然在执行了这些操作后，可以使用：git branch –d feature1操作将该分支直接删除。

### 1.5.3.分支管理策略

使用“fast forward”模式执行合并操作时，分支历史上无法查看到关于合并的信息；如果想要查看该信息，需要使用禁用该模式，具体来讲：

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

由于要查看信息，所以“-m”参数肯定是需要的，方便查看信息的时候可以明确合并操作；参数“--no-ff”显然就是禁用“fast forward”模式。

git对于分支的管理，应有如下几个基本原则：

* master分支应该保证稳定。具体来说，应只是用该分支发布新版本，不在上面进行具体的开发工作；
* 开发工作宜在dev分支实现；每个开发者拥有自己的分支，并将自己的开发结果不断向dev合并；达到某一个稳定dev版本后，将dev向master合并即可。

### 1.5.4.Bug分支

在研发工作进行到一半的时候，现场的bug修复任务下发，需要马上修复bug。这时候直接将研发工作合并到分支里显然不合适，会造成其他人员拿到不能用的代码；放弃不理显然也不合适，会造成工作的浪费。这时候git提供了现场保存的功能。

执行命令“git stash”，git将当前分支上的内容保存起来。这时候执行“git status”可以发现没有了任何改动痕迹，是干净的。这时候就可以创建bug分支，并对bug进行修改后提交到指定分支上。

这时候需要恢复现场，执行“git stash list”命令可以查看有多少现场信息被保存，例如：

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

就说明有一个现场，id为“stash@{0}”，dev分支上的，被保存了。可以通过两种方式进行现场的恢复：

* git stash pop：恢复现场的同时，将stash里的内容也删除了；
* git stash apply：回复现场后stash的内容并不删除；git stash drop可以删除stash里的内容；

如果有多个现场信息被保存，恢复的时候可以指定现场号进行恢复：git stash apply stash@{0}，就是将stash@{0}的现场恢复。

**修复bug的标准流程应该就是如此：首先将没有完成的工作现场stash一下，然后新建分支修复bug，修复完成后将bug的分支删除，最后恢复现场。**

### 1.5.5.多人协作

当你从远程仓库克隆时，实际上git是把本地的master分支和远程的master分支对应起来了。且，远程仓库的默认名称是origin。

* “git remote origin”命令可以查看远程库的信息，要查看详细信息可以增加参数：“git remote –v origin”。
* git push origin master：将本地的master分支的内容推送到远程仓库对应的分支上；同理，“git push origin dev”就是推送dev分支的内容；
* git clone ...：将远程仓库的内容同步到本地工作区；这里只能同步master分支，如果需要dev等其他分支，需：git checkout –b dev origin/dev；
* git pull remote：假设你远程同步了dev分支，在修改过程中，另一个开发者对这个分支进行了修改并推送到了远程仓库，当你提交的时候，显然将面临版本的问题。这条指令就是处理这种冲突的。

该指令将最新的提交从当前的origin/dev分支上抓取下来，并与本地版本进行合并：如果提示该操作有错误，有可能出现1.5.2中说的冲突问题，需要手动解决；否则可再次尝试push操作，将其推送到远程。

* git branch --set-upstream branch-name origin/branch-name：如果git pull执行时提示“no tracking information”的错误，说明本地分支和远程分支没有建立连接关系，就需要执行本指令进行链接；

## 1.6.标签管理

发布一个程序版本时，通常先在版本库里打一个标签，将来要取某一个版本的程序，可以直接将打标签时刻的历史版本取出来。

### 1.6.1.创建标签

* git tag v1.0：对当前的分支、当前的版本，添加一个新标签“v1.0”。
* git tag：查看所有标签；
* git tag tagname commit\_id：在提交版本号是commit\_id处，新建一个标签；该功能可以为提交了很长时间的程序添加标签；该id可以通过git log操作获得；
* git tag –a tagname –m “infomation” commit\_id：在上一条指令基础上，增加了描述信息；
* git show tagname：查看指定标签的描述信息；

### 1.6.2.管理标签

* git tag –d tagname：删除本地标签，该标签尚未上传到远程仓库；
* git push origin tagname：将本地的某个标签(名字为tagname)推送到远程仓库；
* git push origin --tags：将本地的所有标签都推送到远程仓库；
* 删除远程标签需两步操作：
  + git tag –d tagname：先从本地删除；
  + git push origin ：refs/tags/tagname：再从远程删除；

## 1.7.自定义git

在1.1章节中只介绍了配置git的user.name和user.email属性，其实还有很多其他配置项可以配置。

例如：git config --global color.ui true，将在执行git命令的时候以不同颜色区分不同语句。

### 1.7.1.忽略特殊文件

有些文件，必须放在工作区，但又不能放到远程仓库里，例如数据库密码的配置文件、个人信息文件等。这些文件的存在将导致每次执行“git status”命令时总会显示“untracked files”。

为了解决这些问题git提供了“.gitignore”文件：在git工作区的根目录下创建该文件，把要忽略的文件名填进去，git就会忽略这些文件，不再提示上述问题。该文件有明确的格式信息，只需要按照规则填写、组合即可。

忽略文件的原则是：

* 忽略操作系统自动生成的文件，比如缩略图等；
* 忽略编译生成的中间文件、可执行文件等；
* 忽略包含个人信息的敏感文件，比如存放口令的配置文件等；

可以在：<https://github.com/github/gitignore> 上预览很多开源项目的“.gitignore”文件，并根据该类文件编写自己的忽略文件。

教程中并没有明确讲该文件的格式，初步观察应该是“#”后面可以跟类似于注释的内容，将需要忽略的文件直接放到文件中即可。

### 1.7.2.配置别名

可以使用简写代替原始的名字，这样可以照顾自己的简写并减少拼写错误出现的可能性。例如：git config --global alias.st status命令执行后，就可以执行“git st”，而不是“git status”查看效果。

还有很多其他缩写，例如：co----checkout；ci----commit；br----branch等。

### 1.7.3.搭建git服务器

# 2.GIT常用命令汇总

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Order** | **Usage(作用)** | |
| **ch** | **en** |
| **Create(创建)** | $git init | 新建本地库 | create a new local repository |
| $git clone *ssh://user@domain.com/repo.git* | 克隆远程库到本地 | clone an existing repository |
| **Local changes(本地更改)** | $git status | 工作区的文件的修改记录 | file change records in your working directory |
| $git diff *filename* | 指定文件的修改处 | changes to tracked file |
| $git add *filename* | 添加工作区文件到暂存区 |  |
| $git add . | 提交工作区中所有文件的所有改动 | add all current changes to the next commit |
| $git commit -m “*infomation*” | 提交当前暂存区内容 | commit previously staged changes |
| $git commit -a -m “*infomation*” | 提交所有工作区内容 | commit all local changes |
| **commit history(提交历史)** | $git log | 所有的提交记录，从最新的开始 | show all commits, start with newest one |
| $git log -p *filename* |  | show changes over time for a specific file |
| $git blame *filename* |  | who changed what and when |
| **branches&tags(分支和标签)** | $git branch | 显示所有分支 | list all exist branches |
| $git checkout *branchname* | 切换活动分支 | switch HEAD branch |
| $git branch *branchname* | 基于HEAD创建新分支 | Create new branch based on current HEAD |
| $git branch –d *branchname* | 删除一个本地分支 | delete a local branch |
| $git tag *tagname [commit\_id]* | 指定提交点，打标签 | Mark a tag in *commit\_id* |
| **update&publish(更新)** | $git pull *remotename* *branch* | 获取远程的最新修正并直接合并到HEAD | download changes and directly merge/integrete into HEAD |
| $git push *remotename* *branch* | 推送分支到远程 | publish local changes on remote |
| $git push remotename --tags | 所有标签推送到远程 | publish your tags to remote reop. |
| **merge (合并)** | $git merge *branchname* | 合并分支到活动分支 | merge branchname to HEAD |
| $git rm *filename* | 从版本库里删除文件 |  |
| **undo(撤销)** | $git reset --hard HEAD | 抛弃工作区所有改变 | discard all local changes |
| $git checkout HEAD *filename* |  |  |
| $git reset --hard *commit\_id* |  |  |
| $git reset *commit\_id* |  |  |

# 3.Github的使用

想要将代码共享在服务器上，以方便在不同的工作地点都能够对同一套代码进行管理，个人使用者不可避免的要使用到github。

Github官网：<https://github.com/>；

创建用户是必须的操作，不做赘述。

对于github上代码的上传和下载等操作，实际使用后作如下记录：

## 3.1.上传新的代码库

要将一个目录上传到github上，可以在linux上直接执行，也可以在windows上通过模拟终端程序来实现:

* linux上直接使用“sudo apt-get install git”按照git即可；
* windows上下载msysgit后安装，之后就可以在任何目录通过右键找到“git bash here”，打开后即可同样通过命令行执行git操作；

### 3.1.1.创建一个仓库

要上传一部分代码到github上，首先要在github上创建一个新的代码仓库，这样github才能够为这个新上传的代码分配一个ssh路径，才能够执行上传的动作。

在github上创建新的仓库极为简单：在个人主页中，找到“new repository”后，点击创建即可；

### 3.1.2.创建ssh key

可以有https和ssh两种方式与github交互，我使用的是ssh，较为推荐的也是ssh。

本地要与github进行ssh交互，首先要在本地创建一个ssh key，之后将这个key注册给github上已经创建好的代码仓库，具体操作如下：

#### 本地创建ssh key

使用如下命令，在本地创建ssh key：ssh-keygen –t rsa –C “wujl\_0351@163.com”

创建时，会提示输入用户名密码等信息，不做任何输入，直接回车即可；

一旦创建完成，将会在 ~/.ssh目录下生成两个key，查看id\_rsa.pub文件内容如下：

$ cat id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD9A0uSZVuDtyxAsZwd/nAMwz/1djmhwlIs/KrnkuconlTBYvSNFdS2DozROC99Jf9YQ04qJ5RwH9p1RCLMpo9/q3dgasRMyvhCBg7cRxiaGTVt2gBE2SIm2OMiJ8zYCntkkTfzan7q+A5ldpFGjS5HTZZC1kDUP4fMj+L8MLP0WDDYximlSwMuMEj2VVbMKin8dm5Ou+R1Sxi+bf2S/k9IcLtLmU2noBjBLeSu9cu/0bloHZ0lHbMMSdTa6I2kMsE0MffOwSA6lKzj9i2vwFPzsx94iBhIsoCTW/dB0Y2LxWXNFE32CDjCNZ4AwRJreT1OJjF/+cXM0ZIniTTrnjK7 wujl\_0351@163.com

这个key的内容就是github中需要使用到的；

#### 将ssh key加入到代码仓库中

在github的个人页面中，选择“view profile and more”后，进入settings页面，选择“SSH and GPG keys”栏，在SSH keys中选择New SSH key按钮后，将提示输入一个名字和一个key；名字可以按照你对代码仓库的使用者来自行输入，key就要输入上述的id\_rsa.pub中的全部内容了。

注意要输入的是全部内容！包括最前面的“ssh-rsa”，也包括最后的邮箱名字！

#### 测试是否成功

使用如下命令可以测试ssh是否添加成功：ssh -T git@github.com

在没有成功添加key到github上的时候，其输出应该是：

$ ssh -T git@github.com

The authenticity of host 'github.com (192.30.253.112)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'github.com,192.30.253.112' (RSA) to the list of known hosts.

Permission denied (publickey).

在成功添加到github上之后，其输出应该是:

$ ssh -T git@github.com

Hi EricJinleiWu! You've successfully authenticated, but GitHub does not provide shell access.

### 3.1.3.本地代码提交

要将本地代码提交到github上，首先要将本地代码置入git管理中。

在要提交的代码目录中，执行如下命令：git init；执行该命令后，将在该目录下生成一个.git文件，这个文件中记录了git管理的所有信息；

之后再改目录中，执行：git add \*；可以将这个目录下的所有文件加入到git暂存区中，等待上传；

最后执行：git commit -m “”；就可以将这个目录中的内容，加入到本地的git管理中；

需要注意的是，由于windows、linux、macos对于一些符号的处理不同（最典型的就是换行符），因此执行add时如果出现大量的“LF will be replaced by CRLF in …”，就是不同平台导致的。

解决该问题的方法是：git config --global core.autocrlf false；这个命令可以将自动根据平台切换这些字符的处理方式这个特性置为false，那么再上传文件时就会将所有的文件按照原来的原始内容上传，不会做任何改动了。

### 3.1.4.将本地代码提交到github仓库中

首先，git config --global user.name “EricJinleiWu”

第二，git config --global user.email “wujl\_0351@163.com”

这两步操作执行完之后，用户信息将被记录，之后在github上查询到的commit记录中，都会有这两个信息。

第三，git remote add origin [git@github.com:EricJinleiWu/moCodes.git](mailto:git@github.com:EricJinleiWu/moCodes.git)

第四，git push -u origin master

执行完成后，github的个人主页上查看，此时就已经有了新提交的代码了。

## 3.2.从github上下载代码

下载已有的代码仓库到本地来，除创建用户、ssh key等动作外，需要执行的命令是clone，例如：git clone …/.git；

# 4.MyFAQ

## 4.1.同步最新代码到github出错

代码修改后，在本地执行了git add和git commit，将其作为本地版本记录存储成功后，执行git push命令向github同步时出错：

$ git push origin master

To github.com:EricJinleiWu/moCodes.git

! [rejected] master -> master (fetch first)

error: failed to push some refs to 'git@github.com:EricJinleiWu/moCodes.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

网上查询后发现出现该原因是没有远程同步github上的版本，需要首先：git pull origin master；之后再执行push命令即可成功。

出现这个的原因，是我在github上增加了一个readme文档，但是本地没有同步这个文件，导致出现了冲突。后来没有文件冲突的情况下进行push，都没有出现过这个错误。

VIM使用

# 1.VIM简明教程

网上博客找到的vim的教程，相对来说较为简单明了，网址：<http://blog.csdn.net/niushuai666/article/details/7275406>。该博客是翻译的国外牛人的，国外牛人的原地址：<http://yannesposito.com/Scratch/en/blog/Learn-Vim-Progressively/>。

其推荐的vim学习分成了四个等级：存活，感觉良好，觉得更好、更快、更强，使用vim的超能力。

## 1.1.存活

|  |  |  |
| --- | --- | --- |
| **命令符** | **含义** | **备注** |
| i | 从normal模式进入insert模式，光标位于当前位置 | “a”同样进入insert模式，光标位于下一个字符处 |
| x | 删除当前光标所在的一个字符 |  |
| :wq | 保存退出；:w，保存；:q，退出； |  |
| dd | 删除当前行，并将删除的内容复制到剪贴板 |  |
| p | 将剪贴板的内容粘贴到当前位置 |  |
| hjkl | 上下左右，也可以使用小键盘的上下左右键实现 | 推荐使用hjkl实现，但不强制 |
| :help <command> | 对某个命令的帮助信息 | 可以不跟command；退出需:q |

## 1.2.感觉良好

|  |  |  |  |
| --- | --- | --- | --- |
| **类别** | **命令符** | **含义** | **备注** |
| **各种插入模式** | a | 光标后插入 |  |
| o | 当前行后插入新行 |  |
| O | 当前行前插入新行 |  |
| **简单的移动光标** | 0 | 到行头 |  |
| ^ | 到本行第一个非blank字符 | blank指空格、tab、换行、回车等 |
| $ | 到本行行尾 |  |
| g\_ | 到本行最后一个非blank字符 | blank同上 |
| /pattern | 查找“值为pattern”的字符串 | 按n，可到下一个查找位置 |
| **拷贝/粘贴** | p | 将内容粘贴到当前位置之后 |  |
| P | 将内容粘贴到当前位置之前 |  |
| yy | 拷贝当前行，相当于ddP |  |
| **undo/redo** | u | undo |  |
| ctrl+r | redo |  |
| **保存等文件操作** | :e < filepath> | 打开指定路径的文件 |  |
| :saveas <path> | 将文件另存到指令路径 |  |
| :q! | 退出，不保存修改 |  |
| :bn，:bp | 同时打开多个文件，使用这两条命令进行切换 | 可以使用“:n”切换到下一个文件 |

## 1.3.更好、更快、更强

|  |  |  |  |
| --- | --- | --- | --- |
| **目的** | **命令符** | **含义** | **备注** |
| **更好** | .(小数点) | 重复上一次的命令 |  |
| N(command) | 重复某个命令N次 | 2dd，删除2行；3p，粘贴文本3次 |
| **更强** | : N | 到第N行 |  |
| gg | 到第一行 | 相当于: 1 |
| G | 到最后一行 |  |
| e | 到光标所在单词的结尾 | 这里的单词默认是字母、数字、下划线组成的(程序变量) |
| w | 到下一个单词的开头，见图1.3.1 |
| E/W | 功能同e/w，见图1.3.1 | 单词是由blank分隔的(程序语句) |
| **\*/#** | **匹配光标所在的单词，移动光标到下一处(\*)/上一处(#)** |  |
| % | 匹配括号移动，包括(、{、[ | 光标要移动到括号上 |
| **更快** | <start position> <command> <end position> | 命令与光标位置的组合使用 | 0y$:先到行头，在复制，复制到行尾；  ye：从当前位置拷贝到本单词的最后； |
| v | 可视化的选择 | 按v后移动光标，可看到文本被选择 |
| gU/gu | 选中内容变大写/小写 | 可用v可视化选中 |
| d | 删除选中内容 | 同上 |



图1.3.1 命令e/w与E/W的差别

Valgrind

# 1.安装

官网：<http://valgrind.org/>；

可以在官网下载最新的valgrind程序；

首先执行配置操作：./configure --prefix=/home/wujl/test/valgrind-3.12.0/output；使用“prefix”指定安装到哪个目录，默认会安装到/usr/local/中，因此如果不设置这个值，要添加root权限；

如果是交叉编译，需要指定交叉编译工具链：./configure --host=arm-hisiv100nptl-linux CC=arm-hisiv100nptl-linux-gcc CPP=arm-hisiv100nptl-linux-cpp CXX=arm-hisiv100nptl-linux-g++ --prefix=/home/wujl/test/valgrind-3.12.0/output；

此外，如果板子的型号并不是armv7，而是其他arm型号，需要手动修改配置文件configure，在其中的armv7\*) 改为 armv7\* | arm )；

之后:make；make install；

执行完成后，所有的相关文件，会产出到指定的目录中，例如我的output目录中；

# 2.基本使用

./valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --log-file=valgrind.log /mnt/a.out

* --tool：指定使用哪个工具检测，检测内存泄露的是memcheck，如果不适用—tool指定，那么默认使用的也是memcheck；
* --leak-check：指定是否检测泄漏；
* --show-reachable：是否检测“程序结束后依然可达”的内存泄漏；
* --log-file：输出检测结果到文件；
* Others

# 3.MyFAQ

Question：valgrind failed to start tool 'memcheck' for platform 'arm-linux': not a directory

Answer：运行时提示如上错误，是由于编译产出物中，有lib目录，如果默认路径安装，是安装在/usr/local/lib中的，可以直接找到并使用；如果不是默认路径安装，而是使用了prefix来指定，valgrind进程找不到库就会报这个错。解决方法是： export VALGRIND\_LIB="/mnt/output/lib/valgrind/"；

Question：illegal instruction

Answer：非法的汇编指令，我在板子上就遇到了这个问题，而且一直没有解决！问题出现的原因，是编译的时候连接的动态库，和板子上部署的动态库根本不是同一个；而且如果动态库没有调试信息，也会报这个错误；

看网上有人说，架构不同也会导致这个问题，而且valgrind现在只能支持armv7，因此需要在Makefile中增加：-march=armv7-a 这个信息，但是我增加了这个也没有效果；

Question：无法捕获到内存出错信息

Answer：有几个原因可能导致捕获不到内存出错信息：1. 编译时使用的编译级别过高，将调试信息都删除了，例如：CFLGAS修改O2-->O0；2.程序没有正常终止；

杂项

# 1.virtualBox使用代理服务器更新

virtualBox虚拟机中，装了ubuntu的系统，实体机是windows10，代理上网， 希望ubuntu能够apt-get到资源。

sudo vi /etc/apt/apt.conf

在您的apt.conf文件中加入下面这行（根据你的实际情况替换yourproxyaddress和proxyport）。

Acquire::http::Proxy "[http://proxyaddress:port](http://proxyaddress:port/" \t "_blank)";

如果需要用户名密码登陆：

Acquire::http::Proxy "[http://username:password@proxyaddress:port](http://username:password@proxyaddress:port/" \t "_blank)";  
保存apt.conf文件。（其他的协议自己可以适当修改）

# 2.svn客户端查看log时总是显示不全

**[最近](javascript:;" \t "_self)**遇到好几次，show log查看**[日志](javascript:;" \t "_self)**时，看不到最新的日志，只能看到几天前的  
日期处，也显示几天前，无法选择当前日期，还以为是版本太低，换了新版，也遇到同样情况，查了下原来是log被缓存了

**[解决](javascript:;" \t "_self)方案：**

TortoiseSVN-setting-Saved Data：

clear清空Log messages(show log dialog)

另外，也可以删除Log Caching

TortoiseSVN-setting-Log Caching-Cached Repositories

Cached Repositoried,删除对应的url记录

# 3.Virtual box扩容

在虚拟机的ubuntu中安装qt时，发现20G的容量，只剩下2G了，显然不够用了，于是查看了一下如何给虚拟机扩容。

按照网上的普遍说法，virtualbox只要是4.0以上的版本都支持动态扩容，基本命令如下：

* Cd E:\installDirectory //进入到virtualbox的安装目录
* VBoxManage.exe list hdds //查看现在的hdd信息，发现我的硬盘容量是20G
* VBoxManage.exe modifyhd 8365a282-8a2c-4794-a12a-eb07c515b2c0 --resize 40960 //修改虚拟机的容量到40G(4096M)，这里用了uuid做为虚拟机的唯一标识进行指定；

但是至此，并未发现如大家所描述的百分比进度条，而是提示了如下错误：

Progress state: VBOX\_E\_NOT\_SUPPORTED

VBoxManage.exe: error: Resize medium operation for this format is not implemented yet!

于是再次查阅资料，发现有些资料上描述是用的vdi文件的绝对路径，而不是uuid，猜测是否是这个原因呢？于是将uuid更改为绝对路径尝试，依然不可以，还是这个错误；

还有些资料说，vmware和virtualbox各自有各自的虚拟机文件格式，不是vdi格式的虚拟机文件，并不能支持这个操作，要先做一个格式转换。仔细查看了一下，我的就是vdi，没有这个原因。

终于，找到了一个英文说明：<https://www.intowindows.com/fix-resize-hard-disk-operation-for-this-format-is-not-implemented-yet-error-while-resizing-virtualbox-disk/>

这里说明，直接更改大小限制的操作，只能对动态分配size的情况使用，对于固定大小的这样不行，他提供了如下几个步骤：

1. VBoxManage.exe clonehd "E:\OSImages\Ubuntu1604\_virtualbox\_image\ubuntu16.04\_64bit.vdi" "E:\OSImages\Ubuntu1604\_virtualbox\_image\ubuntu16.04\_64bit\_40G.vdi" //克隆一个新的镜像
2. VBoxManage.exe modifyhd E:\OSImages\Ubuntu1604\_virtualbox\_image\ubuntu16.04\_64bit\_40G.vdi --resize 40960 //将新得到的vdi的size更改为40G，要注意，这个vdi的size分配方式是动态分配，最大size是40G.
3. 打开virtualBox，设置，存储，选择SATA选项，新加一个新的存储选项，选中ubuntu16.04\_64bit\_40G.vdi文件，新建一个；
4. 在这个页面中，将之前的那个固定size的vdi文件使用删除icon删除连接；
5. 将之前的vdi文件直接删除。

这样操作完后，查看设置页面，确实显示最大size改成了40G，但是实际使用fdisk -l /dev/sda查看容量的时候，却发现依然还是那么多，能用的空间一点也没有增加。

## 3.1.windows下新分配空间给虚拟机

dos命令行中，进入到virtualbox的安装目录。

vboxManage.exe list hdds //展示现在的虚拟机的硬盘信息

vboxManage.exe modifyhd diskUUID.vdi --resize SIZE\_IN\_MB //这里需要用到上面得到的uuid值填充到diskUUID.vdi；size是以M为单位计算的。

本操作执行完成后，再次list hdds，可以看到size有增加了。

但这时候虚拟机中还不认识这个分区，继续向下执行。

## 3.2.格式化新分区

df -H查看现在的占用情况。

fdisk /dev/sda //新建分区

进入到选项设置中后，输入n(new，新加一个分区)，p(执行过程中输出信息)，之后一路回车，系统自动将新加入的空间作为一个新分区存在并创建，最后输入w保存新加入的分区，并退出fdisk。

执行fdisk -l，查看所有分区信息，可以看到新多了一个分区：/dev/sda4；

但是这时候ls /dev/sda\*是看不到这个分区的，必须要**重启以后**才能看到。

重启后，执行：mkfs.ext4 /dev/sda4，将这个分区格式化成为ext4的文件格式。

这时候sda4已经是可以用的操作系统节点了，但是并不能被我们使用，我们还需要挂载到本地目录下才可以。

## 3.3.挂载

mkdir /home/wujl/sda4

mount /dev/sda4 /home/wujl/sda4

这种挂载可以正确执行，但是一旦重启就失效了。要永久设置，需要修改/etc/fstab文件，在其尾部增加如下一行：/dev/sda4 /home/wujl/sda4/ ext4 defaults 0 1

这样就可以永久挂载了。

树莓派

# 1.系统安装和环境配置

## 1.1.系统安装

安装的是树莓派官网提供的树莓派裁剪过的debian系统，按照官网所述，现在树莓派也有专门裁剪过的ubuntu系统了，但是还没有尝试过。

系统直接安装在了tf卡上。

1. tf卡，使用读卡器，插在电脑上(windows系统即可)；
2. 使用sdFormater工具，将sd卡格式化；
3. 使用win32DiskImager工具，将下载好的镜像文件(img)，烧写在tf卡上；
4. 弹出tf卡，插在树莓派上；
5. 树莓派，连接网线、hdmi线到显示器，插键盘(蓝牙鼠键也可，默认支持蓝牙)；
6. 启动，系统正常启动，默认用户名为pi，密码是raspberry；

至此，系统安装完成；

## 1.2.环境配置

### 1.2.1.ssh服务

ssh服务，是首先需要被配置的，因为我们不可能每次都使用树莓派连接在hdmi上，而更应该在内网里，使用secureCRT或者putty等工具，远程访问；

树莓派的系统，是支持ssh服务的，但是自2016年起，它取消了ssh的默认启动。要启动，很简单，进入其根目录下，创建一个空的ssh文件，就可以了。

因为我电脑上用的linux是在虚拟机下的，所以我直接在windows下进行了这个操作：tf卡插在读卡器上，插在电脑上，进入到tf卡目录下，直接创建一个ssh文件；拔出tf卡，插到树莓派上，启动；

验证ssh服务是否开启，可以通过“ps -ef”命令，查看是否有sshd这个进程的存在；

进程存在后，局域网内，通过putty或者secureCRT，访问树莓派的IP，通过ssh协议进行连接，就可以连接了。

### 1.2.2.samba服务

我做树莓派的目的，是作为家庭服务器使用(虽然硬件很低，但是我需求也少啊……)；

所以samba服务是不可缺少的，这个可以帮助我们，在windows下直接访问linux下的目录和文件。

安装samba服务：apt-get install samba samba-common-bin；

修改配置文件：vim /etc/samba/smb.conf，最末尾加上如下内容：

[Share] #指定了windows下访问的文件名，我修改为Share，表示是共享的

comment =share directory to windows # 共享文件夹说明

path = /home/wujl/shares # 共享文件夹目录

read only = no # 不只读

create mask = 0777 # 创建文件的权限

directory mask = 0777 # 创建文件夹的权限

guest ok = yes # guest访问，无需密码

browseable = yes # 可见

重启samba服务：/etc/init.d/samba restart

### 1.2.3.设置静态IP地址

需要将树莓派的地址固定为某一个，而不是自动分配的话，需要修改：/etc/dhcpd.conf；

找到其中的eth0，ipv4的地址修改为指定的192.168.11.152/24，后面的“/24”表示指定其子网掩码为255.255.255.0；在指定默认网关为192.168.11.1就可以了。

重启后就可以正常识别了。

最终示例如下：# Example static IP configuration:

interface eth0

static ip\_address=192.168.11.152/24

#static ip6\_address=fd51:42f8:caae:d92e::ff/64

static routers=192.168.11.1

static domain\_name\_servers=114.114.114.114 8.8.8.8

### 1.2.4.连接外网

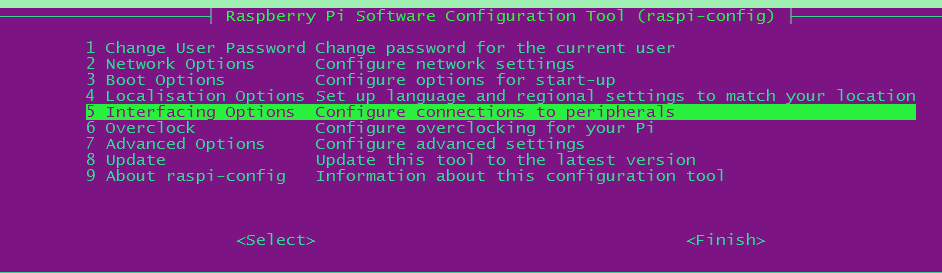
增加外网链接需要的网关：route add default gw 192.168.11.1 dev eth0

修改/etc/resolv.conf文件，“nameserver 8.8.8.8”，使用谷歌代理的服务；

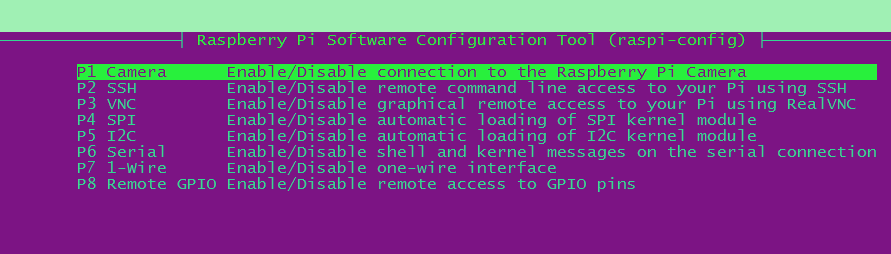
### 1.2.5.开启摄像头支持选项

摄像头插入后，树莓派并不是马上自动支持的，需要开启一些系统选项。

raspi-config命令，进入到menu中：



选中5选项，进入后，



第一个就是开启/关闭摄像头支持的选项。

# 2.第三方开源工具安装

由于暂时树莓派没有办法连接外网，且安装的又是最小系统，因此安装tcpdump时安装了一些其他基础包。一旦外网可以连接之后，就可以通过apt-get完成安装了。

## 2.1.Tcpdump

安装tcpdump，依次安装了如下几个工具：

1. m4;
2. Bison;
3. Flex;
4. Libpcap;
5. Tcpdump;

安装方式，都是源码包下载，之后 : ./configure; make; make install;

## 2.2.Sqlite

安装sqlite，通过源码包，执行了：./configure; make; make install;正常安装。

## 2.3.Valgrind

Valgrind安装时，外网已经通了，因此直接使用apt-get install进行了安装，但是安装完成后，无法正常运行！

于是又下载了tar.bz2的源文件，并同样执行了:.configure; make; make install;这样再次安装之后的，就可以运行了。

怀疑是自动安装的，安装的是arm交叉编译之后的valgrind，但是我所有的程序都是用gcc和g++编写的，因此不能正常运行。

## 2.4.Mysql

安装mysql耗时巨大。

由于之前树莓派的源太慢，因此将/etc/apt/sources.list文件，更新为了阿里云的源。

安装mysql-server时，直接使用apt-get install mysql-server命令，提示版本不符合的错误。

后来将源更新回树莓派的源，再安装就可以了。怀疑是两个源的一些依赖包的版本不匹配，导致了问题的出现。

具体安装：apt-get install mysql-server;

Apt-get install mysql-client;

Mysql-server安装完成后，通过如下命令进行测试：netstat -tap | grep mysql;如果发现有mysql的服务在运行，并且已经listen请求，说明mysql server安装成功。

### 2.4.1.Mysql登录

按照网上所述，安装mysql-server的过程中，会提示输入密码，但是我在安装时并没有提示。

查看mysql安装后生成的配置文件：cat /etc/mysql/debian.cfg，内容如下：

root@raspberrypi:/etc/mysql# cat debian.cnf

# Automatically generated for Debian scripts. DO NOT TOUCH!

[client]

host = localhost

user = root

password =

socket = /var/run/mysqld/mysqld.sock

[mysql\_upgrade]

host = localhost

user = root

password =

socket = /var/run/mysqld/mysqld.sock

basedir = /usr

按照这个内容来看，应该是密码没有设置，为空。

于是使用root账户，直接尝试登陆：mysql -u root -p ；提示输入密码时，直接按回车键，果然登录成功！

### 2.4.2.修改密码

但是使用网上查找的方法，进行密码的修改时，并未成功！

表现是，修改后，直接使用回车键，依然能够直接登录到mysql上。。。。而且查看cfg文件，password还是空的。。。。

需要后续继续研究

### 2.4.3.安装C++的API支持

需要再次安装一个dev，以支持C++程序调用需要的so和h；

apt-get install libmysql++-dev；

安装完成后，通过：mysql\_config --cflags可以查看头文件的位置，通过mysql\_config --libs可以查看需要的库文件及位置，例如：

root@raspberrypi:/usr# mysql\_config --cflags --libs

-I/usr/include/mysql

-L/usr/lib/arm-linux-gnueabihf -lmariadbclient -lpthread -lz -lm -ldl

这样编译程序的时候，直接使用这些信息就可以。

## 2.5.Flask

树莓派上的os已经预置了2.7版本的python，需要部署flask环境，要用到pip：apt-get install python-pip python-dev；

再安装flask：pip install flask；

### 2.5.1.helloWorld

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "Hello World"

if \_\_name\_\_=="\_\_main\_\_":

app.run(host="192.168.11.152", port=8080, debug=True)

## 2.6.Python3

更新python到python3；

因为python2很快就不更新了，所以将默认的python更改到python3；

Apt-get install python3;

Cd /usr/bin;

Rm -rf python;

Ln -s python.3.5 python;

Python --version;

卸载python2的命令是：apt-get autoremove python2.7

## 2.7.Piccamera库

树莓派使用了摄像头后，默认可以通过raspistill和raspivid命令行的方式来进行拍照和录像。但是命令行的方式不利于程序设计和编写。因此找到了一个python实现的摄像头控制的库。

安装：apt-get install python3-picamera

如果安装失败，尝试apt-get update 和upgrade。因为摄像头插上后，需要更新一下树莓派的固件系统，才能安装需要的依赖库和内核更新。

这个库死活安装不上，逼得我重装了一次树莓派的系统，然后启动后就卸载了python2.7，以为这样就可以了，结果还是不行......

最终的解决，是将sources.list更新到清华的源就ok了。阿里云的源这次不能行~~~~

SWUpdate

# 1.概述

SWUpdate用来对一些嵌入式系统进行升级，可以从一个存储设备上的image文件升级，也可以从网络升级。然而，它更应该被称为一个框架(framework)，因为扩展的协议或安装方式，可以很方便的添加到其中。

一个典型的应用场景，是从外部设备的image文件升级，例如一个USB设备或者SD卡。这种情况下，升级操作与操作者之间不进行交互，可以认为是“一键升级”的类型。当用户重启设备(或其他任何可以识别升级文件的触发方式)后，升级程序自动启动，完成所有校验并执行升级。最后，升级程序只报告一个成功或者失败的升级结果给操作者。

结果可以输出到LCD设备上(使用frame-buffer的设备)，或者直接输出到一个串口设备(例如Linux的交互命令行窗口)。

通常来说，升级会使用initrd(使用Yocto方式产出升级image)、单份拷贝的方法来实现。当然，你也可以使用双份拷贝的方法(通过Software collections)来升级。

如果是远端网络升级，SWUpdate启动一个嵌入式的web-server，等待请求。使用者必须上传一个合适的image文件，SWUpdate将校验、安装。整个过程将通过AJAX的通知机制，发送到使用者的浏览器上。

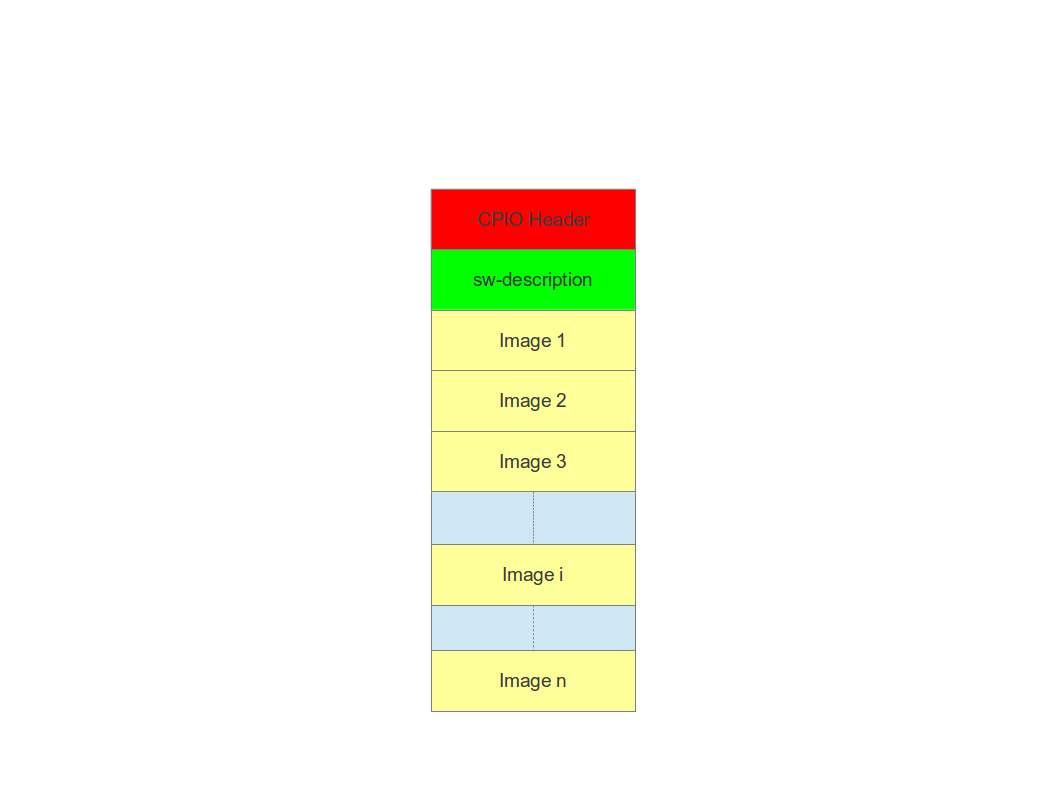
## 1.1.Features

### 1.1.1.概览

* 安装在一个嵌入式的设备上(eMMC、SD卡，raw nand、nor和spi-nor flash等)；
* 校验image是否有效。Image是组织在一个特殊的格式：cpio中，它必然包含一个软件描述信息，以此作为校验依据；
* SWUpdate用来更新设备上的***UBI volumes***和images，***UBI volumes***一般是针对NAND，但也有其他类型可以支持。因为一个完整的image文件可以作为一个sd卡分区，或者MTD分区。
* 支持压缩image格式，zip或者tgz文件的格式，都是可以支持的；
* 支持更新文件系统中一个单独的文件；
* 结构化的语言方式来描述一个image，这种结构化语言结构，可以被libconfig库作为默认解析器解析，这是一种类似于json的结构化语言结构；
* 可以使用一些其他的自有解析器，这是通过lua实现的。在example中提供了一个lua实现的xml格式的解析器。
* 支持设置/擦写：uboot参数；grub环境参数；EFI boot引导参数；
* 支持：preinstall/postinstall脚本。他们在升级image之前/之后运行；
* 支持配置，配置后可以校验软硬件的适配性。软件image必须包含一个入口，描述自己可以在哪种硬件平台/版本上运行。
* 支持image文件的部分安装。一个image文件可能指针很多平台，也可能包含很多自部分，SWUpdate支持部分安装image，或者提取当前平台需要的部分安装。
* 掉电安全。

### 1.1.2.Single image delivery--单镜像文件传送

这种类型的最大特点，就是制造商传送了一个超大的image。这个image，是被所有需要的subImage打包在一起组成的，除了这些subImage，还需要cpio的header，和sw-description描述文件结构和每一个单独的subImage信息。



### 1.1.3.Streaming feature -- 直接以流的形式传输升级文件

一般的升级思路，是将image文件存储到本地目录后，使用这些本地文件进行升级。

在远程网络升级中，SWUpdate接收相应的image文件，并将他们拷贝到环境变量TMPDIR锁指向的目录中，如果该环境变量没有设置，将拷贝到/tmp中。这样可以确保只有所有需要的文件都接收到了，并且都是正确的，才会执行升级操作。

SWUpdate支持直接以流的方式进行升级，不是必须将文件存储到本地。

这个特性是用户可以设置的，可以通过设置一个：installed\_directly flag，选择直接将文件以stream(流)的形式传送到目标位置上，还是暂存到一个文件里。当然，如果选择了流的方式传输，那么就无法对整个image文件的完整性和正确性进行验证了。暂存目录只有在网络传输的时候，才有可能有必要使用，如果是使用本地image镜像(usb、sd卡等)，没有必要拷贝到本地存储中。

## 1.2.配置和编译

### 1.2.1.依赖性

对于SWUpdate来说，编译时的依赖不多，只需要一些库：

* Mtd-utils：mtd-utils产出libmtd和libubi。他们一般没有被安装和引用，但是他们会被SWUpdate引用，用来升级MTD和UBI volumes；
* Openssl：webserver依赖；
* Lua：liblua；
* Libz，libcrypto一般也要被连接；
* Libconfig：默认的解析器；
* Libcurl：network需要；
* Libarchive：可选择，针对不同平台的handler(安装程序)；
* Libjson：可选择，json解析和hawkbit；
* Libubootenv：可选择，如果支持uboot，就需要引用该库；
* Libebgenv：可选择，如果支持EFI boot 引导，就需要；

### 1.2.2.使用Yocto编译

提供了一个meta-swupdate层。这一层提供了mtd-utils和生成lua需要的所有变化，首先，需要拿到一份meta-swupdate的代码：

git clone https://github.com/sbabic/meta-swupdate.git；

将meta-SWUpdate加入到你的bblayers.conf中。你可以在其中加入machine信息：

MACHINE=<......> bitback swupdate-image

你可以在你的tmp/deploy/<......>目录中找到结果。

### 1.2.3.配置SWUpdate

SWUpdate支持“make menuconfig”进行配置。在默认配置中，很多选项都已经被打开了。

### 1.2.4.编译

对于交叉编译平台来说，在执行make之前首先设置CC和CXX是必要的。在make menuconfig中设置交叉编译前缀也是一种选项。

设置好必要的选项后，执行make，就可以编译代码。

编译完成后，得到的产出物是一个二进制文件：swupdate。除此之外，progress这个进程也会被创建，但这并不是直接需要用到的。这是一个例程，讲述你自己的接口如何挂在SWUpdate上展示进度条。

## 1.3.运行

### 1.3.1.SWUpdate运行时期待的结果

SWUpdate运行时，主要包含以下几个步骤：

* + 检查是否存在存储设备(例如u盘等)；
  + 检查是否存在image文件，image文件必须以“.swu”结尾；
  + 从image文件中解出sw-description；
  + 读取cpio信息，校验checksum值；
  + 检查软件硬件的兼容性；
  + 确认所有sw-description中描述的部件都确实存在在cpio中；
  + 如有有需要就更改分区。
  + 执行pre-install脚本；
  + 循环执行每个subImage的安装操作；
  + 执行post-install脚本；
  + 如果sw-description中指定了，更新bootloader环境；
  + 向调用者反馈升级结果；

几个命令：

* Swupdate -i <filename> : 启动swupdate，期待从文件“filename”中读取image；
* Swupdate -w “<web server options>” : 从一个嵌入式webserver中启动swupdate；例如：Swupdate -w “--document-root ./www --port 8080”，最重要的两个参数，就是“document-root”和“port”，嵌入式webServer使用的是mongoose；
* Swupdate -h：help；

默认的webServer的端口是8080，可以直接通过：http://<ip>:8080，连接到swupdate提供的主页。

### 1.3.2.命令行参数

|  |  |  |
| --- | --- | --- |
| 参数 | 类型 | 描述 |
| -f <file> | string | SWUpdate的配置文件 |
| -b <string> | String | 只有在CONFIG\_MTD被设置了的情况下，这个参数才能激活。  Active only if CONFIG\_MTD is set It allows to blacklist MTDs when SWUpdate searches for UBI volumes. Example: U-Boot and environment in MTD0-1: swupdate -b “0 1” |
| -c <sel> | string | sel is in the format <software>,<mode> It allows to find a subset of rules in the sw-description file. With it, multiple rules are allowed. One common usage is in case of the dual copy approach. Example: -e “stable, copy1” ==> install on copy1 -e “stable, copy2” ==> install on copy2 |
| -h |  | help |
| -k | string | 如果CONFIG\_SIGNED使用公钥设置了filename，这个参数有效 |
| -l | int | 设置log的级别 |
| -L |  | 输出log到syslog中 |
| -i <file> | string | 使用本地的.swu文件启动SWUpdate |
| -n |  | 在dry-run模式下运行SWUpdate |
| -o <file> | string | 保存流(SWU)到一个文件 |
| -v |  | Activate verbose(冗余的) output |
| -w <param> | string | 启动内置的webserver |
| -u <param> | string | start internal suricatta client daemon and pass to it a command line string. see suricatta’s documentation for details. |
| -H <board:rcv> | string | 设置板子名称和硬件版本号 |
| -c |  | 通过内部校验机制检查\*.swu文件 |
| -p | String | 执行post-update命令 |
| -d <pramas> | string | 只有在CONFIG\_DOWNLOAD开启的时候才有效， |
| -u <url> | string | URL地址，URL是一个连接地址，指向一个有效的swu镜像文件； |
| -t <timeout> | integer | 下载的时候，连接的超时时间 |
| -a <user:pwd> | string | 发送用户名和密码来做基础校验 |
|  |  |  |

### 1.3.3.Systemd Integration

SWUpdate可以支持systemd，这需要在编译的时候，指定CONFIG\_SYSTEMD配置项来完成。如果打开了该选项，SWUpdate发送信号给systemd。

TODO, 后续的我觉得用途不大，没有细看。

# 2.默认解析器的语法规则

## 2.1.简介

SWUpdate使用libconfig作为默认的、image描述信息的解析器。然而，对于SWUpdate来说，也可以扩展后，支持一个自有的解析器。在例程中有一个基于lua的解析器，以XML的形式构造描述信息。

一个示例文件(libconfig规则)：

software =

{

version = "0.1.0";

description = "Firmware update for XXXXX Project";

hardware-compatibility: [ "1.0", "1.2", "1.3"];

/\* partitions tag is used to resize UBI partitions \*/

partitions: ( /\* UBI Volumes \*/

{

name = "rootfs";

device = "mtd4";

size = 104896512; /\* in bytes \*/

},

{

name = "data";

device = "mtd5";

size = 50448384; /\* in bytes \*/

}

);

images: (

{

filename = "rootfs.ubifs";

volume = "rootfs";

},

{

filename = "swupdate.ext3.gz.u-boot";

volume = "fs\_recovery";

},

{

filename = "sdcard.ext3.gz";

device = "/dev/mmcblk0p1";

compressed = true;

},

{

filename = "bootlogo.bmp";

volume = "splash";

},

{

filename = "uImage.bin";

volume = "kernel";

},

{

filename = "fpga.txt";

type = "fpga";

}

);

files: (

{

filename = "README";

path = "/README";

device = "/dev/mmcblk0p1";

filesystem = "vfat"

}

);

scripts: (

{

filename = "erase\_at\_end";

type = "lua";

},

{

filename = "display\_info";

type = "lua";

}

);

bootenv: (

{

filename = "bootloader-env";

type = "bootloader";

},

{

name = "vram";

value = "4M";

},

{

name = "addfb";

value = "setenv bootargs ${bootargs} omapfb.vram=1:2M,2:2M,3:2M omapdss.def\_disp=lcd"

}

);

}

## 2.2.处理配置中的差异项

一个image文件中，可以包含多个不同类型的设备的各种升级数据。每一个设备可以有自己的kernel、dtb和root filesystem，或者各个设备可以共享一些公共的部分。

现在这些都是被自有的配置文件解析器管理的。它通过判断设备上软件运行的情况，校验那个image要被安装。

因为解析器是基于lua的，每个人都可以借助lua实现他自己的配置解析规则。最简单的例子，就是sw-description是一个xml格式的数据，我们自实现一个解析器解析它，也可以实现相应功能。

一份配置文件，也可以指定多个不同的设备，默认的配置文件格式可以如下所示定义：

Software =

{

Version = “0.1.0”;

Target-1 = {

Image : (

{

Device = “/dev/mtd4”

......

}

);

};

Target-2 = {

Image : ({......});

};

}

这样定义的配置文件，就可以一个image文件，针对不同的硬件平台来使用。一般情况下，硬件平台的信息会定义在/etc/hwrevision文件中，这个文件只有一行：<boardname><revision>；

## 2.3.Software

Software标签定义了顶层结构，且可以指定不同的软件信息，例如上述例子中，就在image中指定不同的目标设备地址，来确认我们的image安装到不同的位置，例如上面的target-1中，就指定了这个镜像要安装到/dev/mtd4中。

## 2.4.hardware-compatibility

升级image兼容的硬件版本号，以如下格式定义：hardware-compatibility:[“major.minor”, “major.minor”, ...]；

例如：hardware-compatibility:[“1.0”, “1.3”]；

## 2.5.partitions

这部分可以用来改变UBI volumes的分区规则。

Partitons : (

{

Name = <volume name>; //name=”rootfs”

Size = <size in bytes>; //size=1048576

Device = <MTD device>; //device=”mtd4”

},

);

SWUpdate会寻找指定的partation，如果不存在，就创建一个指定size的分区；如果存在，就改变其分区size；

## 2.6.images

Images标签用来表示系统中将要被安装的image信息，语法如下：

images:(

{

filename/\*必须存在\*/ = <name in cpio archive>; /\*对应cpio中的名字\*/

volume/\*可以不存在\*/ = <destination volume>;

device/\*可以不存在\*/ = <destination volume>;

mtdname/\*可以不存在\*/ = <destination mtd name>;

type/\*可以不存在\*/ = <handler>;

offset/\*可以不存在\*/ = <offset>; /\*指定从设备的那个offset开始进行写\*/

compressed; /\*是否是压缩文件，例如文件是tar.gz就是一种压缩\*/

},

/\* next image \*/

);

注意，**volume和device，同一个image中只能选择一个**！

例如：

{filename=”core-image-base.ubifs”; volume=”rootfs”;}是用来更新ubi volume的；

{filename=”core-image-base.ext3”; device=”/dev/mmcblk0p1”;}是以raw模式更新的；

{filename=”uboot.bin”; device=”/dev/mmcblk0p1”; offset=16K;}是从offset=16K的位置开始更新的；单位换算机制是：K=1024, M=1024\*1024;

## 2.7.files

也可以不拷贝image，而是用单个文件来代替。这不是一个正常的选择，但很多情况下，可以作为一种调试手段或者解决一些特殊需求。

files : (

{

filename=<name in cpio archive>; //文件，必须与cpio中的名字一一对应

path = <path in filesystem>; //目标地址

device/\*可选择\*/ = <device node>;

filesystem/\*可选择\*/ = <filesystem for mount>;

}

);

filename和path是必须要存在的，filename是使用那个源文件；path是要拷贝到那个目标地址。

device和filesystem是可选择的，如果设置了这两个选项，那么就会先将device指定的目标设备格式化成filesystem指定的文件格式，之后再从filename拷贝到path；

## 2.8.scripts

脚本会按照他们在sw-description文件中的顺序依次运行。脚本的运行结果会被检测，如果发现结果不等于0，SWUpdate会停止升级。脚本会自动被拷贝到一个临时目录下， 之后才会被执行。一定要保证每个脚本的名字是唯一的，并且是和cpio中相对应的。

如果没有显示的定义脚本的类型，那么默认的类型就是LUA。

### 2.8.1.lua脚本

示例：

scripts:(

{

filename=<name in cpio archive>;

type=”lua”;

},

);

lua脚本会被内部的解释器执行，必须最少包含：function preinst() 和 function postinst()中的一个。

### 2.8.2.shell脚本

scripts:(

{

filename=<name in cpio archive>;

type=”shellscript”;

},

);

SWUpdate将使用系统命令执行shell脚本，传入preinst或者postinst作为第一个参数给脚本使用。如果data属性被赋值的话，他的值将作为其他参数，传入脚本。

## 2.9.bootloader

TODO

## 2.10.board特定设置

TODO

# 3.代码级编译及使用

## 3.1.编译

代码下载：<https://github.com/sbabic/swupdate>

下载后，make menuconfig，进行配置，此时可能出现curses.h头文件找不到的问题，需要安装相应的包：sudo apt-get install libncurses5-dev；

Make menuconfig后，make时需要安装一些依赖包：

Mtd/libmtd.h找不到，直接使用sudo apt-get install mtd-utils安装，依然没有见效，于是猜测是否是版本低的缘故，于是下载最新的版本尝试：<http://lists.infradead.org/pipermail/linux-mtd/2018-April/080308.html>

安装mtd前，需要先安装LZO：http://www.oberhumer.com/opensource/lzo/download/，安装：./configure; make; make install;

再去mtd目录：./configure; make; make install;

但是这样依然不能找到libmtd.h文件。于是查看了一下，确实没有将产出物放在/usr/local目录下，于是手动执行，将mtd-utils目录中的include直接拷贝到/usr/local/include/中；

同时将libmtd.a libubi.a拷贝到/usr/local/lib中；

这时候提示lua.h找不到，于是要安装lua：apt-get install lua5.3 lua5.3-dev；

但是并未见效。。。。想了想，我并不一定要用到lua，于是make menuconfig时，取消了lua选项~~哦耶，这次换成libconfig.h找不到了。。。。。

下载最新的libconfig：<https://github.com/hyperrealm/libconfig>

Autoreconf;

./configure --disable-cxx;

make; //提示没有makeinfo工具，apt-get install texinfo

make install;

最终编译提示了一个链接错误：libubootenv这个库找不到。OK，为了快速的对库进行验证，开展下一步工作，我直接去make menuconfig中，将uboot选项取消了》。。。。。

编译通过，执行命令：./swupdate -w “--document-root ./www --port 8080”,进程正常运行，之后再浏览器通过:http://<ip>:8080，来访问，得到了一个swupdate的网页。

make menuconfig时，打开image handlers--raw & script，不然不支持解析纯文件和脚本文件；

## 3.2.demo

做了一个简单demo，主要用来测试两部分的内容：

1. cpio文件格式是否正确；
2. sw-description中定义的规则是否按照预期的执行了；

### 3.2.1.生成cpio格式的升级文件

先看一下文件树：

myDemo

├── file1.h

├── file2.so

├── myTest.swu

├── shellscript.sh

├── sw-description

└── test.txt

相对简单，由于是64位虚拟机上操作，不期待更新image和partition信息，因此只做了两个纯文本文件file1.h file2.so，一个shell脚本文件shellscript.sh，此外就是必须要包含的sw-description文件。

test.txt文件，是将这四个文件名字组织起来：

cat test.txt

./sw-description

./file1.h

./file2.so

./shellscript.sh

在生成cpio文件时，要求sw-description文件一定要是第一个文件，因此这里通过test.txt文件手动指定了各个文件在cpio中的顺序。

MyTest.swu是生成的cpio格式的文件，按照SWUpdate的要求，修改为了swu后缀的文件名字。生成该cpio文件的命令：

cat test.txt | cpio -ov --format='crc' > myTest.swu

注意，format一定要指定为crc，不然生成的cpio文件的mark不是070702，而是070707，而且cpio header的长度也不是110个字节，而是76个；

附注一个普通的生成cpio文件的命令以作参考：

find . -name sw-description | cpio -ov --format='crc' > ver10\_newc.cpio

### 3.2.2.升级

启动SWUpdate进程：./swupdate -w “-r ./www -p 8080 -a 2”;对各个参数做一下说明：

* -w说明要启动webserver，支持http访问，能够通过http client发送http请求的方式，进行远程升级；
* -r说明要去www目录下寻找需要的html等网页文件；
* -p说明webserver的port是8080；
* -a是用来指定其内部使用的http server的版本(mongoose v1或v2)；

执行如上命令后，网页输入：http://<yourIp>:8080，就可以成功访问swupdate提供的网页了；

但是要注意，以v2的mongoose版本启动webserver，虽然可以通过浏览器访问，但是网页上提供的升级功能并不能正确使用，流程并未打通。后续要查看一下这些问题。

我用了一个折中的方式来测试升级流程：

1. 启动进程时，指定mongoose版本为v1：./swupdate -w “-r ./www -p 8080 -a 1”；
2. 通过curl工具，模拟一个客户端：curl <http://192.168.11.154:8080/handle_post_request;>
3. swupdate接收到这个http请求后，自动读取本地的myTest.swu文件，并执行后续的流程；

以这种模拟化的方式，我验证了升级流程是ok的；

## 3.3.后续工作

### 3.3.1.继续预研

后续有几个部分需要继续预研：

1. 嵌入式平台的基础环境搭建，目标是在36平台上可以正常运行期swupdate进程：
   1. 所需第三方库文件的编译；
   2. swupdate的交叉编译；
2. 应用层的其他功能调研：
   1. 直接通过browser执行升级流程；
   2. 通过U盘内的swu文件升级；
3. 联调，应用层的功能与嵌入式平台的结合；

### 3.3.2.V1

V1主要想坐到的目标是：定义好升级文件的格式，这种格式的文件，可以像现在的uptools一样正常升级到36的嵌入式平台上。

### 3.3.3.V2

V2想要做到的目标是：可以支持double firmware的方式执行升级操作；