编写Shell脚本的最佳实践

作者: Myths 发布时间: 2017-08-05 16:39 阅读: 20038 次 推荐: 46 原文链接 [收藏]

前言

由于工作需要，最近重新开始拾掇shell脚本。虽然绝大部分命令自己平时也经常使用，但是在写成脚本的时候总觉得写的很难看。而且当我在看其他人写的脚本的时候，总觉得难以阅读。毕竟shell脚本这个东西不算是正经的编程语言，他更像是一个工具，用来杂糅不同的程序供我们调用。因此很多人在写的时候也是想到哪里写到哪里，基本上都像是一段超长的main函数，不忍直视。同时，由于历史原因，shell有很多不同的版本，而且也有很多有相同功能的命令需要我们进行取舍，以至于代码的规范很难统一。

考虑到上面的这些原因，我查阅了一些相关的文档，发现这些问题其实很多人都考虑过，而且也形成了一些不错的文章，但是还是有点零散。因此我就在这里把这些文章稍微整理了一下，作为以后我自己写脚本的技术规范。

代码风格规范

开头有"蛇棒"

所谓shebang其实就是在很多脚本的第一行出现的以"#!"开头的注释，他指明了当我们没有指定解释器的时候默认的解释器，一般可能是下面这样：

```
#!/bin/bash
```

当然，解释器有很多种，除了bash之外，我们可以用下面的命令查看本机支持的解释器:

```
$ cat /etc/shells
#/etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/screen
```

当我们直接使用`./a.sh`来执行这个脚本的时候，如果没有shebang，那么它就会默认用`$SHELL`指定的解释器，否则就会用shebang指定的解释器。

不过，上面这种写法可能不太具备适应性，一般我们会用下面的方式来指定：

```
#!/usr/bin/env bash
```

这种方式是我们推荐的使用方式。

代码有注释

注释，显然是一个常识，不过这里还是要再强调一下，这个在shell脚本里尤为重要。因为很多单行的shell命令不是那么浅显易懂，没有注释的话在维护起来会让人尤其的头大。

注释的意义不仅在于解释用途，而在于告诉我们注意事项，就像是一个README。

具体的来说，对于shell脚本，注释一般包括下面几个部分：

1. shebang
2. 脚本的参数
3. 脚本的用途
4. 脚本的注意事项
5. 脚本的写作时间，作者，版权等
6. 各个函数前的说明注释

搜索文章

推荐链接
- 程序员找工作，就在博客园招聘频道
- 程序员问答平台，解决您的技术难题

编程基础热门文章
- Fiddler 教程
- SSL与TLS的区别以及介绍
- 字符串匹配的KMP算法
- 序列化和反序列化
- 数字证书及CA的扫盲介绍

编程基础最新文章
- 作为一个程序员，数学对你到底有多重要
- 以操作系统的角度述说线程与进程
- NASA的10条代码编写原则
- 编写Shell脚本的最佳实践
- 也许，这样理解HTTPS更容易

最新新闻
- Opera 53开发者版上线：Speed Dial新增新闻栏目
- 微信发布谣言治理报告：辟谣中心全年科普4.9亿次
- 因为一张黑板Word图画 微软向加纳老师捐赠电脑
- 运营商被曝即将迎来大调整：流量不分省内省外
- 贾跃亭终爆仓 谁要进场接盘被割韭菜

热门新闻
- 我是一名朝九晚五的程序员（你也可以！）
- 京东缴纳60亿社保惹了谁?
- 95后回乡见闻：尴尬的伪互联网生活
- 微信"跳一跳"首个广告诞生：一个2000万的Nike鞋
- 去开房，为何我的房钱比别人贵80 | 大数据反手一

7. 一些较复杂的单行命令注释

参数要规范

这一点很重要，当我们的脚本需要接受参数的时候，我们一定要先判断参数是否合乎规范，并给出合适的回显，方便使用者了解参数的使用。

最少，最少，我们至少得判断下参数的个数吧：

```
if [[ $# != 2 ]];then
    echo "Parameter incorrect."
    exit 1
fi
```

变量和魔数

一般情况下我们会将一些重要的环境变量定义在开头，确保这些变量的存在。

```
source /etc/profile
export PATH="/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/apps/bin/"
```

这种定义方式有一个很常见的用途，最典型的应用就是，当我们本地安装了很多java版本时，我们可能需要指定一个java来用。那么这时我们就会在脚本开头重新定义JAVA_HOME以及PATH变量来进行控制。

同时，一段好的代码通常是不会有很多硬编码在代码里的“魔数”的。如果一定要有，通常是用一个变量的形式定义在开头，然后调用的时候直接调用这个变量，这样方便日后的修改。

缩进有规矩

对于shell脚本，缩进是个大问题。因为很多需要缩进的地方(比如if,for语句)都不长，所有很多人都懒得去缩进，而且很多人不习惯用函数，导致缩进功能被弱化。

其实正确的缩进是很重要的，尤其是在写函数的时候，否则我们在阅读的时候很容易把函数体跟直接执行的命令搞混。

常见的缩进方法主要有”soft tab”和”hard tab”两种。

- 所谓soft tab就是使用n个空格进行缩进(n通常是2或4)
- 所谓hard tab当然就是指真实的”\t”字符

  这里不去撕哪种方式最好，只能说各有各的优劣。反正我习惯用hard tab。

  对于if和for语句之类的，我们最好不要把then，do这些关键字单独写一行，这样看上去比较丑。。。

命名有标准

所谓命名规范，基本包含下面这几点：

1. 文件名规范，以.sh结尾，方便识别
2. 变量名字要有含义，不要拼错
3. 统一命名风格，写shell一般用小写字母加下划线

编码要统一

在写脚本的时候尽量使用UTF-8编码，能够支持中文等一些奇奇怪怪的字符。不过虽然能写中文，但是在写注释以及打log的时候还是尽量英文，毕竟很多机器还是没有直接支持中文的，打出来可能会有乱码。

权限记得加

这一点虽然很小，但是我个人却经常忘记，不加执行权限会导致无法直接执行，有点讨厌。。。

日志和回显

日志的重要性不必多说，能够方便我们回头纠错，在大型的项目里是非常重要的。

如果这个脚本是供用户直接在命令行使用的，那么我们最好还要能够在执行时实时回显执行过程，方便用户掌控。

有时候为了提高用户体验，我们会在回显中添加一些特效，比如颜色啊，闪烁啊之类的，具体可以参考 ANSI/VT100 Control sequences 这篇文章的介绍。

密码要移除

不要把密码硬编码在脚本里，不要把密码硬编码在脚本里，不要把密码硬编码在脚本里。

重要的事情说三遍，尤其是当脚本托管在类似Github这类平台中时。。。

太长要分行

在调用某些程序的时候，参数可能会很长，这时候为了保证较好的阅读体验，我们可以用反斜杠来分行：

```
./configure \
-prefix=/usr \
-sbin-path=/usr/sbin/nginx \
-conf-path=/etc/nginx/nginx.conf \
```

注意在反斜杠前有个空格。

编码细节规范

代码有效率

在使用命令的时候要了解命令的具体做法，尤其当数据处理量大的时候，要时刻考虑该命令是否会影响效率。

比如下面的两个sed命令：

```
sed -n '1p' file
sed -n '1p;1q' file
```

他们的作用一样，都是获取文件的第一行。但是第一条命令会读取整个文件，而第二条命令只读取第一行。当文件很大的时候，仅仅是这样一条命令不一样就会造成巨大的效率差异。

当然，这里只是为了举一个例子，这个例子真正正确的用法应该是使用`head -n1 file`命令。。。

勤用双引号

几乎所有的大佬都推荐在使用”$”来获取变量的时候最好加上双引号。

不加上双引号在很多情况下都会造成很大的麻烦，为什么呢？举一个例子：

```
#!/bin/sh
#已知当前文件夹有一个a.sh的文件
var="*.sh"
echo $var
echo "$var"
```

他的运行结果如下：

```
a.sh
*.sh
```

为啥会这样呢？其实可以解释为它执行了下面的命令：

```
echo *.sh
echo "*.sh"
```

在很多情况下，在将变量作为参数的时候，一定要注意上面这一点，仔细体会其中的差异。上面只是一个非常小的例子，实际应用的时候由于这个细节导致的问题实在是太多了。。。

巧用main函数

我们知道，像java，C这样的编译型语言都会有一个函数入口，这种结构使得代码可读性很强，我们知道哪些直接执行，那些是函数。但是脚本不一样，脚本属于解释性语言，从第一行直接执行到最后一行，如果在这当中命令与函数糅杂在一起，那就非常难读了。

用python的朋友都知道，一个合乎标准的python脚本大体上至少是这样的：

```
#!/usr/bin/env python
def func1():
    pass
def func2():
    pass
if __name__=='__main__':
    func1()
    func2()
```

他用一个很巧妙的方法实现了我们习惯的main函数，使得代码可读性更强。

在shell中，我们也有类似的小技巧：

```
#!/usr/bin/env bash
func1(){
    #do sth
}
func2(){
    #do sth
}
main(){
    func1
    func2
}
main "$@"
```

我们可以采用这种写法，同样实现类似的main函数，使得脚本的结构化程度更好。

考虑作用域

shell中默认的变量作用域都是全局的，比如下面的脚本：

```
#!/usr/bin/env bash
var=1
func(){
    var=2
}
func
echo $var
```

他的输出结果就是2而不是1，这样显然不符合我们的编码习惯，很容易造成一些问题。

因此，相比直接使用全局变量，我们最好使用`local readonly`这类的命令，其次我们可以使用`declare`来声明变量。这些方式都比使用全局方式定义要好。

巧用heredocs

所谓heredocs，也可以算是一种多行输入的方法，即在”<<”后定一个标识符，接着我们可以输入多行内容，直到再次遇到标识符为止。

使用heredocs，我们可以非常方便的生成一些模板文件：

```
cat>>/etc/rsyncd.conf << EOF
log file = /usr/local/logs/rsyncd.log
transfer logging = yes
log format = %t %a %m %f %b
syslog facility = local3
EOF
```

学会查路径

很多情况下，我们会先获取当前脚本的路径，然后一这个路径为基准，去找其他的路径。通常我们是直接用`pwd`以期获得脚本的路径。

不过其实这样是不严谨的，`pwd`获得的是当前shell的执行路径，而不是当前脚本的执行路径。

正确的做法应该是下面这两种：

```
script_dir=$(cd $(dirname $0) && pwd)
script_dir=$(dirname $(readlink -f $0 ))
```

应当先cd进当前脚本的目录然后再pwd，或者直接读取当前脚本的所在路径。

代码要简短

这里的简短不单单是指代码长度，而是只用到的命令数。原则上我们应当做到，能一条命令解决的问题绝不用两条命令解决。这不仅牵涉到代码的可读性，而且也关乎代码的执行效率。

最最经典的例子如下：

```
cat /etc/passwd | grep root
grep root /etc/passwd
```

cat命令最为人不齿的用法就是这样，用的没有任何意义，明明一条命令可以解决，他非得加根管道。。。

使用新写法

这里的新写法不是指有多厉害，而是指我们可能更希望使用较新引入的一些语法，更多是偏向代码风格的，比如

1. 尽量使用`func(){}`来定义函数，而不是`func{}`
2. 尽量使用`[[]]`来代替`[]`
3. 尽量使用`$()`将命令的结果赋给变量，而不是反引号
4. 在复杂的场景下尽量使用printf代替echo进行回显

事实上，这些新写法很多功能都比旧的写法要强大，用的时候就知道了。

其他小tip

考虑到还有很多零碎的点，就不一一展开了，这里简单提一提。

- 路径尽量保持绝对路径，绝多路径不容易出错，如果非要用相对路径，最好用./修饰
- 优先使用bash的变量替换代替awk sed，这样更加简短
- 简单的if尽量使用&& ||，写成单行。比如`[[ x > 2]] && echo x`
- 当export变量时，尽量加上子脚本的namespace，保证变量不冲突
- 会使用trap捕获信号，并在接受到终止信号时执行一些收尾工作
- 使用mktemp生成临时文件或文件夹
- 利用/dev/null过滤不友好的输出信息
- 会利用命令的返回值判断命令的执行情况
- 使用文件前要判断文件是否存在，否则做好异常处理
- 不要处理ls后的数据(比如`ls -l | awk '{ print $8 }'`)，ls的结果非常不确定，并且平台有关
- 读取文件时不要使用for loop而要使用while read

静态检查工具shellcheck

概述

为了从制度上保证脚本的质量，我们最简单的想法大概就是搞一个静态检查工具，通过引入工具来弥补开发者可能存在的知识盲点。

市面上对于shell的静态检查工具还真不多，找来找去就找到一个叫**shellcheck**的工具，开源在github上，有8K多的star，看上去还是十分靠谱的。我们可以去他的主页了解具体的安装和使用信息。

安装

这个工具的对不同平台的支持力度都很大，他至少支持了Debian,Arch,Gentoo,EPEL,Fedora,OS X,openSUSE等等各种的平台的主流包管理工具。安装方便。具体可以参照安装文档

集成

既然是静态检查工具，就一定可以集成在CI框架里，shellcheck可以非常方便的集成在Travis CI中，供以shell脚本为主语言的项目进行静态检查。

样例

在文档的Gallery of bad code里，也提供了非常详细的“坏代码”的标准，具有非常不错的参考价值，可以在闲下来的时候当成”Java Puzzlers“之类的书来读读还是很惬意的。

本质

不过，其实我觉得这个项目最最精华的部分都不是上面的功能，而是他提供了一个非常非常强大的wiki。在这个wiki里，我们可以找到这个工具所有判断的依据。在这里，每一个检测到的问题都可以在wiki里找到对应的问题单号，他不仅告诉我们”这样写不好”，而且告诉我们”为什么这样写不好”，”我们应当怎么写才好”，非常适合刨根问底党进一步研究。

参考资料

- 关于 shell 脚本编程的10 个最佳实践
- shell脚本编写规范
- Shellcheck Tool
- Best Practices for Writing Bash Scripts
- Good coding practices for bash
- Design patterns or best practices for shell scripts

- bashstyle(GITHUB)
- BashGuide/Practices
- Obsolete and deprecated syntax
- ANSI/VT100 Control sequences

标签： Shell Linux

« 上一篇：也许，这样理解HTTPS更容易
» 下一篇：NASA的10条代码编写原则