# 七种武器设计篇之设计是自找的

[mnstory.net](http://mnstory.net/)

一直有个悖论，如果一个人，没有设计能力，那就不会给你模块设计；但是，一个人的设计能力，需要从实际的设计中锻炼出来，如果不给你模块锻炼，如何得来设计能力？

看样子是这样的，但是，也不尽然，我之前给同事吹过牛逼：**设计，是自找的。**

你可以从每天改BUG的生活中，找到设计，之前举了一个我在改BUG的时候如何为HCI引入redis的例子，我今天看一下，一个普通的API，如何自找设计。

## V1

写一个Python执行Shell命令的API，看似乎非常简单，我的需求是，可以输入一点数据也可不输入（不交互），主要是能分别获取STDOUT和STDERR，还有退出码，方便外部判断命令是否执行正确（然而我最害怕的是，有同事根本不关心返回值，那就没下面什么事了）。

一般来说，写到这个水平，已经差不多了：

**def** run(cmd, input=**None**):  
 **try**:  
 p = subprocess.Popen(cmd, shell=**True**, stdout=subprocess.PIPE, stderr=subprocess.PIPE)  
 (out, err) = p.communicate(input)  
 **except** Exception, e:  
 **return** (127, **""**, str(e))  
   
 **return** (p.returncode, out, err)

## V2

当然，作为老码农，写的代码总应该和新员工有所区别，必须细读API DOC，搞懂每个参数是做啥的，测试验证，有疑问的配合源码阅读，然后我又发现几个问题：

1. 是否应该记录一下程序的执行时间？

毕竟太多时候，定位性能问题，就靠这个时间。（经验）

1. 是否需要对输出的数据做一下formal处理？

例如out数据有的是带回车换行，有的不带，当然，作为通用API，我应该原封不动返回，但是我是个懒人，我不想每次外部获取到的out数据还要自己trim一下，事实上，我至今没有见过谁的命令调用，结果分析依赖于out的首位两端空白符的，所以，我认为应该API内部做formal处理。（个人需求）

1. 此API里面是否应该输出一些正常日志。

不是异常日志，异常日志我是一定会输出的，也会返回，但是正常日志，一般情况下，我是拒绝的。

但这个地方我认为有必要，因为我是一个反对在程序里面掉命令来完成任务的人，所以说，这个run函数，使用应该非常少，也需要非常明确哪些逻辑使用了，所以我输出一些日志，第一，可以警示使用者，命令是否调用过多；第二，调命令完成任务是最容易出错的逻辑，应该有全面的日志记录。（设计取舍）

1. 经验告诉我们，毫不相干的子进程应该close所有继承自Parent的句柄。（经验）

于是更改为如下版本：

\_lastOutDict={}  
**def** run(cmd, input=**None**):  
 *# 1. 记录执行时间* timeStart = time.time()  
 **try**:  
 *#4. close\_fds=True 关闭所有从父进程继承的句柄* p = subprocess.Popen(cmd, shell=**True**, stdout=subprocess.PIPE, stderr=subprocess.PIPE, close\_fds=**True**)  
 (out, err) = p.communicate(input)  
 **except** Exception, e:  
 timeEnd = time.time()  
 *# 3. 错误日志输出* l.error(**"<EXE>(%ds):%s failed(%s)"** % ((timeEnd-timeStart), cmd, str(e)))  
 **return** (127, **""**, str(e))  
 timeEnd = time.time()  
  
 *# 2. 对out和err做trim处理* **if** out:  
 out = out.strip()  
 **else**:  
 out = **""  
 if** err:  
 err = err.strip()  
 exitCode = p.returncode  
  
 *# 3. 正常日志输出的时候，要考虑是否太过冗余，所有对于超过256字节的相同输出信息，第二次就做了supress，防止日志干扰* debugSupressOut = out  
 **if** out **and** len(out) > 256:  
 **if** \_lastOutDict.get(cmd, **""**) == out:  
 debugSupressOut = **"<equal last...>"  
 else**:  
 \_lastOutDict[cmd] = out  
 *# 3. 正常日志输出* l.debug(**"<EXE>(%d,%ds):%s%s%s%s"** % (exitCode, (timeEnd-timeStart), cmd, (**" <IN>:%s"** % input) **if** input **else ""**, (**" <OUT>:%s"** % debugSupressOut) **if** debugSupressOut **else ""**, (**" <ERR>:%s"** % err) **if** err **else ""**))  
 **return** (exitCode, out, err)

## V3

我对命令行调用的敬畏之心，远远超过很多人，所以，我还觉得差点什么。

是的，差一个TIMEOUT。

经验告诉我们，依赖外部命令的时候，有一个常见的风险，便是卡死，这是个头疼的问题。

Python2.7里面没有TIMEOUT执行命令的API，需要借助线程的TIMOUT来实现。

于是，有了第三个版本：

*# 杀进程树，而不是子进程，单杀子进程，孙子进程还在，残留逻辑没人收拾***def** killTree(rootPid, killRoot):  
 **try**:  
 rootProcess = psutil.Process(rootPid)  
 children = rootProcess.get\_children(recursive=**True**)  
 **for** child **in** children:  
 l.info(**"kill tree child %d:%s (parent %d:%s)"** % (child.pid, child.cmdline, rootProcess.pid, rootProcess.cmdline))  
 child.kill()  
 psutil.wait\_procs(children, timeout=7)  
  
 **if** killRoot:  
 l.info(**"kill tree root %d:%s"** % (rootProcess.pid, rootProcess.cmdline))  
 rootProcess.kill()  
 rootProcess.wait(5)  
 **except** Exception, e:  
 l.warning(**"kill tree %d found exception, %s"** % (rootPid, str(e)))  
  
**class** Command(object):  
 **def** \_\_init\_\_(self, cmd, input=**None**):  
 self.cmd = cmd  
 self.input = input  
 self.process = **None** self.out = **""** self.err = **""** self.errDesc = **""  
  
 def** \_target(self):  
 **try**:  
 self.process = subprocess.Popen(self.cmd, shell=**True**, stdout=subprocess.PIPE, stderr=subprocess.PIPE, close\_fds=**True**)  
 (self.out, self.err) = self.process.communicate(self.input)  
 **except** Exception, e:  
 **if** self.errDesc:  
 self.errDesc += **", "** self.errDesc += str(e)  
  
 **def** \_run(self):  
 self.\_target()  
  
 **def** \_runTimeout(self, timeout):  
 thread = threading.Thread(target=self.\_target)  
 thread.start()  
 thread.join(timeout)  
  
 **if** thread.is\_alive(): *# 超时后，线程还没有主动结束，表示还卡着，这个时候，就要主动KILL了* **if** self.errDesc:  
 self.errDesc += **", "** self.errDesc += **"timeout(%ds)"** % timeout  
 **if None** == self.process:  
 self.errDesc += **", no process object"  
 else**:  
 self.errDesc += **", kill process tree(%d)"** % self.process.pid  
 killTree(self.process.pid, **True**) *#全部杀死* thread.join(1) *#再给他一个机会* **def** run(self, timeout=-1):  
 **global** \_lastOutDict  
  
 timeStart = time.time()  
 **if** timeout <= 0:  
 self.\_run() *#如果是没有timeout，就不需要开启线程* **else**:  
 self.\_runTimeout(timeout) *#用新线程来等待* timeEscape = (time.time()-timeStart)  
  
 **if** self.out:  
 self.out = self.out.strip()  
 **else**:  
 self.out = **""  
  
 if** self.err:  
 self.err = self.err.strip()  
 **else**:  
 self.err = **""  
  
 if** self.errDesc:  
 exitCode = -1  
 **if** self.err:  
 self.err += **" "** self.err += **"<EXCEPTION>:"** +self.errDesc  
 **else**:  
 exitCode = self.process.returncode  
  
 debugSupressOut = self.out  
 **if** self.out **and** len(self.out) > 256:  
 **if** \_lastOutDict.get(self.cmd, **""**) == self.out:  
 debugSupressOut = **"<equal last...>"  
 else**:  
 \_lastOutDict[self.cmd] = self.out  
  
 l.debug(**"<EXE>(%d,%ds):%s%s%s%s"** % (exitCode, timeEscape, self.cmd, (**" <IN>:%s"** % self.input) **if** self.input **else ""**, (**" <OUT>:%s"** % debugSupressOut) **if** debugSupressOut **else ""**, (**" <ERR>:%s"** % self.err) **if** self.err **else ""**))  
 **return** (exitCode, self.out, self.err)  
  
**def** run(cmd, input=**None**, timeout=-1):  
 command = Command(cmd, input)  
 **return** command.run(timeout)

有TIMEOUT的逻辑和最开始的逻辑比起来，多了很多代码，很满意，这过程中，你是不是需要学习很多东西，例如，为何上面要KILL进程树而不是进程？例如，如何利用jone做线程协同？所有的细微知识，积累起来，就是功力。

2017/4/1