ddp

Author: vincentzhwg@gmail.com
Date: 2013-05-14
Version: **1.1.1**

ddp 介绍

ddp 是基于 python 语言开发的，用于 ssh 登陆机器后执行一系列命令的工具，主要有以下特性：

支持并发执行，也可串行执行
命令脚本具备逻辑结构
可设定脚本命令最终的返回值及字符串
scp 操作通过特定命令可双向判断，选取可行方向执行，避免 scp 单通问题
既可直接当脚本工具使用，也可集成到 python 程序中，提供 API 调用接口

ddp 中文名称为“电灯泡”。

1.1.X 版本为多进程版本，支持 python2.6 至 2.7 版本，3.0 以上版本未测试。

1.0.X 版本为多线程版本，支持 python2.4 至 2.7t 版本，3.0 以上版本未测试。

版本更新日志

这里只列出 1.1.X 版本的更新日志。

版本：**1.1.1**
Date：2013-05-21
更新说明：

增加 SCP_LOCAL_PUSH_PULL 命令
修正几个 bug

版本：**1.1.0**
Date：2013-05-14
更新说明：

多进程版本，执行时间大为缩短
脚本参数去掉 threadsNO，变更为 workersNO
python 版本支持从 2.6 至 2.7，3.0 未测试

脚本使用参数

参数，使用缩写时，前面带 - 符号，如 -v ；使用全称时，前面带 -- 符号，如 --version 。

参数缩写	参数全称	参数说明
h	help	显示帮助信息
v	version	显示版本信息
l	hostsFile	指定 hosts 文件路径，该参数与 hostsString 参数不能同时使用
s	hostsString	指定 hosts 的字符串，该参数与 hostsFile 参数不能同时使用
c	cmdsFile	指定脚本命令 cmds 文件路径
e	execCmds	指定脚本命令 cmds 的字符串，该参数与 cmdsFile 不能同时使用
eh	errorHostsFile	执行完后，保存错误 hosts 信息的文件路径。若不指定该参数，将默认使用 error_hosts.txt 作为其值。注意：若所指定的文件路径原本存在，原文本内容将被清除
sh	successHostsFile	执行完后，保存成功 hosts 信息的文件路径。若不指定该参数，将默认使用 success_hosts.txt 作为其值。注意：若所指定的文件路径原本存在，原文本内容将被清除
r	retryTimes	当执行失败（包括登陆失败及执行某一脚本命令失败）时，重试次数。默认值为 0，表示不重试。
w	workersNO	并发数。小于等于 1 表示串行执行，大于 1 则是所有 host 并发执行
q	quiet	若使用此参数，将不输出执行过程中信息到终端上
qq	quietFiles	不生成 errorHostsFile 及 successHostsFile 两个文件
j	jsonFormat	最终返回的结果以 json 字符串格式表示
pr	printResult	把最终结果输出到终端
o	output	把执行过程中的输出保存到指定文件中

脚本使用命令范例：

```
#使用 -l 参数获取 hosts,使用 -c 参数获取脚本命令 cmds
./ddp.py -l test_hosts.txt -c test_cmds.txt

#使用 -s 参数与 -e 参数
./ddp.py -s '1.1.1.1,"user";1.1.1.2,"user","passwd"' -e '`uname -a`;`whoami`'

#使用 -s 参数与 -c 参数
./ddp.py -s '1.1.1.1,"user";1.1.1.2,"user","passwd"' -c test_cmds.txt

#使用全部参数的范例
./ddp.py -l test_hosts.txt -c test_cmds.txt -w 2 -o /tmp/o.txt -j -pr -q -qq -r 5 -eh /tmp/e.txt -sh /tmp/s.txt
```

Hosts 文件规范

```
#第一个非空白字符为 # ,则表示这一行为注释行
#空白字符指空格或 tab 字符
#所有 host 的 IP 地址及域名不可重复，若重复将提示相应错误

        #这也是注释行，只需要行的首个非空白字符为 # 即可
#host 可分为四项值，每一项之间可用不少于一个的空格或 tab 字符隔开，也可使用逗号 , 隔开
#第一项为 hostName,可用 IP 地址或域名，使用 IP 地址时，直接写上 IP 地址，无需双引号 " 包围；使用域名时需要 " 包围
#第二项为用户名,需要使用双引号包围起来
#第三项为密码，若操作机（即运行 ddp 脚本的机器）到需要 ssh 登录的机器无需密码登陆，可不配置该项，或给予一空字符串，即 ""
#第四项为 ssh 端口，若不是特定端口，一般无需配置此项，留空即可，这样 ssh 会使用自身配置的端口选项进行连接


#各个 host 之间一般以换行符隔开，但也可使用 ; 隔开。使用 ; 隔开时，可在一行中配置多个 host
#host 可标注上标签，且一个 host 可有多个标签。标签的命名规范为下划线或字母开始，后接任意多个的字符或数字。
#标签的用途：在命令脚本中，若某命令有 TAG 修饰，那么该命令也是有标签的，该命令只会在有同样标签的 host 中执行，其余无该标签的 host 则不执行该命令。


10.10.10.10     "user"     "passwd"     #注释也可写在一行最后面，只需以 # 开始，四项配置之间用一个或多个 tab 字符隔开


    10.10.20.10     "user"     "passwd" #host 在配置时无需从行的首字符开始，可以有空格或 tab 字符在行首。未配置第四项，ssh 使用自身配置的端口进行连接


"ddp.oa.com"  "user"     "passwd"     #该行第一项的 hostName 使用域名，当使用域名时，需要用双引号 " 包围起来
```

10.10.30.10 "user" "pass\"wd" #若域名、用户名或密码中恰好有双引号字符，可使用反斜杠进行转义，该行的密码字符串即最终为 pass"wd 。支持反斜杠转义的只有 \t \n \" \\ 四个字符，最后一个表示反斜杠自身

10.10.10.11 "user" "passwd" 34567 #四项之间以空格隔开。配置了第四项端口，在 ssh 连接时将使用 34567 端口

10.10.10.12,"user" , "passwd", 34567 #四项之间以 , 隔开，且在使用逗号隔开时，在逗号前后也可有任意个空格或 tab 字符

10.10.10.13, "user" "passwd" , 22 #逗号与空白字符可混用，第一项与第二项之间使用了逗号，而第二项与第三项之间仅以空白字符隔开

10.10.10.14 "user" #未配置第三项密码。若操作机与要登录的机器无需密码即可登录，可不配置密码项

10.10.10.15 "user" "" #第三项密码为空字符串，这与不配置密码项的效果是等同的

h1::10.10.10.17 "user" "passwd" #符号 :: 表示分隔符，当需要给 host 标注标签时，需要使用该符号。标签值在分隔符 :: 前面，host 在分隔符后面。该行表示该 host 具有 h1 标签

h1 :: 10.10.10.18 "user" "passwd" #分隔符的前后可有任意多个的空白字符

h1 , h2,h3 :: 10.10.10.19 "user" "passwd" #多个标签之间用逗号 , 隔开，且逗号前后也可有任意多个的空白字符

h1, h3 :: 10.10.10.21 "user" "passwd"; 10.10.10.22 "user" "passwd" ;
10.10.20.30 "user" #一般一行中配置一个 host。若想在一行中配置多个 host，可在各个 host 之间用分号 ; 隔开，且分号前后可有任意多个空白字符

脚本命令 cmds 文件规范

#注释规则与 hostsFile 相同

`uname -a`　　#脚本命令必须用前后用 ` 字符包围起来，该字符即是键盘左上角（ESC 键下方的按键）

　　`whoami`　　#脚本命令无需在行首第一个字符开始，前面可有任意多个的空白字符

`cd ~` ; `pwd` ; `uname -a`　　#一般一行中一个命令。若想在一行中有多个命令，可使用分号 ; 隔开，分号前后可有任意个的空白字符

`counter=1`
`counter=\`expr $counter + 1\``　　#若命令中需要使用 ` 字符，可使用反斜杠进行转义。命令脚本中支持反斜杠转义的只有 \t \n \` \\ 四个字符，最后一个表示反斜杠自身，该行命令最终为 counter=`expr counter + 1` ，实现 counter 的值加 1

#命令支持修饰符进行修饰，有以下修饰符：
#NM：no matter，命令执行的成功与失败无关紧要。命令脚本在碰到某条命令执行失败时，将停止执行后面的命令，使用该修饰符可确保命令脚本在执行该命令时不会因执行失败而退出
#NTOL：no timeout limit，命令在执行时有个默认的超时时间，一般为 30s,若超过该时间，命令仍没有执行完毕或输出的话，将视为命令超时执行而失败。对于一些需要较长执行时间的命令，可使用该修改符
#TL：time limit，限定命令执行的超时等待时间，单位为秒。如认为该命令应当在 60 秒内执行完毕，即可设置为 60。
#TAG：标签，对命令标注标签，只有拥有同样标签的 host 才会执行该命令，其余 host 跳过该命令。变量命令规范为以下划线或字母开始，后接任意多个的字母或数字
#VAR：设置变量，将命令输出设置为某变量的值，在以后的命令中可使用
#SCP_PWD：应用于 scp 命令中，对于无需密码的 scp 操作可不需要该修饰符，而对于在 scp 操作时需要输入密码的则需要
#ASSERT：　断定命令应当输出的结果，若不符合，将认为命令执行失败

#一条命令可同时使用多个修饰符，只需以分隔符 :: 隔开，且各修饰符的顺序没有限制，但同样的修饰符只能使用一次。

#命令与修饰符之间用分隔符隔开；各修饰符之间也用分隔符隔开。分隔符前后可有任意个空白字符

```
#修饰符有以下冲突：
#1. TL 与 NTOL 不可同时使用
#2. NM 与 ASSERT 不可同时使用

NTOL :: `sleep 150`              #使用 NTOL 修饰睡眠 150 秒的命令，这样即可避开一般 30 秒的命令超时等待时间
TL :: 160 :: `sleep 150`         #使用 TL 也可达到同样的效果，这里要睡眠 150 秒，那用 TL 设置命令的等待时间为 160s

NM :: `ps axu | grep application_name | grep -v grep | grep -v tail | grep -v vi | awk '{print $2}' | xargs kill -9`    #该命令实现强制杀死某进程的目的，但有时可能不一定存在正在运行的该进程，那就会导致命令失败，但其实又不关心是否已经存在，只是希望在重启前先杀掉可能已经在运行的进程，所以为了避免命令失败导致接下来的脚本命令不能被执行，那么即可加上 NM 修饰符，表示对于该命令的成功与失败都无所谓

TAG :: h1 , h2 :: `cp /tmp/o.txt ~`        #只有同样标注了 h1,h2 标签的 host 才会执行该命令

`cd ~`
VAR :: homePath :: `pwd`                   #将 home 目录路径的值保存到 homePath 变量中，以便接下来的命令中使用
`echo "HOME:{#homePath#}" > /tmp/o.txt`
#使用 homePath 变量，变量在命令中使用时，需要前面用 {# 开始，后面用 #} 结束。如果变量名为 varName, 那么使用时即写为 {#varName#} ,该命令即实现为将 home 目录路径写到/tmp/o.txt 文件中

SCP_PWD :: "passwd" :: `scp user@1.1.1.1:/tmp/o.txt /tmp`    #在执行该 scp 命令时需要密码时，即会使用 SCP_PWD 修饰符中设置的密码，密码需要用双引号包围起来

ASSERT :: 1 :: `ps axu | grep application_name | grep -v grep | awk '{print $2}' | wc -l`
#使用 ASSERT 即可假定该进程有一个在运行中，若命令输出不符合，那么将认为命令执行失败

ASSERT :: "/usr/local/java" :: `echo $JAVA_HOME`      #ASSERT 假定值中也可使用字符串，该命令判定$JAVA_HOME 值是否为/usr/local/java,若不是将判定命令失败

ASSERT :: "{#homePath#}" :: `pwd`          #ASSERT 假定值中也可使用到变量值，但只能用在字符串中，即有双引号包围起来

TAG :: h1, h2 :: VAR :: varA :: `whoami`    #修饰符可多个同时使用，且顺序可任意
VAR::varA::TAG::h1, h2::`whoami`            #该命令与上面的命令的效果是一样的
```

```
#EXIT : EXIT 命令用于指定在退出命令脚本时返回值。执行命令脚本时碰到 EXIT 命令时，将以 EXIT 设定的值退出命令脚本

#若正常执行完命令脚本，都未遇到 EXIT 命令，则默认退出值为 0 及空字符串,反映在返回结果中相应的 host 的 exit 及 msg 中

#EXIT 用法格式 : EXIT integer [, string]
#第一项为 integer，整数值，该项为必须项，当成功执行脚本命令时，该值将在返回结果中相应 host 的 exit 中
#第二项为 string，属于可选项。当成功执行脚本命令时，该值将在返回结果中相应 host 的 msg 中，若有配置此项，则为相应的 string 值；若无配置此项，则 msg 将为空字符串


#EXIT 命令不可用任何修饰符进行修饰


#下面一组命令通过判定某程序的状态，返回相应的值
IF :: ASSERT :: 1 :: `ps aux | grep app | grep -v grep | awk '{print $2}' | wc -l`
    EXIT 2, "running"
ELSE
    IF :: `cd ~/abc'
        EXIT 1, "installed, but no running"
    ELSE
        EXIT 0, "no installed"
    ENDIF
ENDIF
```

脚本命令中的预定义变量

变量名称	说明
sshIP	ssh 机器的 IP 地址若 host 中 hostName 给出的是 IP 地址才能使用此变量，若给出的是域名（如 www.test.com），则不能使用此变量
sshUser	ssh 机器的登陆用户名
sshPort	ssh 机器的 ssh 端口。在配置 host 时有给出 port 这一项，才有此变量，否则没有
sshHomePath	ssh 机器上登陆用户的 home 目录路径

以上的预定义变量可直接在脚本命令中使用

#SCP_LOCAL_PUSH_PULL : 为一特殊命令，实现 scp 的双向操作，将操作机上的某一文件或目录通过 scp 方式放到 ssh 机器上。先测试从操作机向 ssh 机器的 push 方向，若成功即命令成功；若 push 方向操作不成功，则测试从 ssh 机器拉取操作机的 pull 方向，若成功则命令成功；若 push 方向与 pull 方向都失败，那么该命令失败

#SCP_LOCAL_PUSH_PULL 命令由以下选项构成：
#LOCAL_INTF : local interface name,操作机为 local 机，该选项指定操作机在获取其 IP 地址时使用哪一个网卡名称为参数，若不指定，将使用配置文件默认指定的接口名称，一般为 eth0
#LOCAL_PORT : 操作机的 scp 操作的端口，一般无需指定
#LOCAL_PWD : 操作机当前登陆用户的密码，若 scp 操作时无需密码，可不配置此项
#LOCAL_PATH : 操作机上需要被拉取的文件或目录的路径，必须是全路径
#LOCAL_ISDIR : 操作机上被拉取的是文件还是目录，若是文件，则不要配置此项；若是目录，则需要配置此项
#SSH_HOST_PATH : 从操作机拉取到的文件或目录，在 ssh 机器上的存放路径。路径可为相对路径或绝对路径，相对路径是相对于当前 ssh 机器的所处路径

#SCP_LOCAL_PUSH_PULL 的各选项在使用时无顺序限定，可随意排列，各选项之间用分隔符 :: 隔开

SCP_LOCAL_PUSH_PULL::LOCAL_PWD::"passwd"::LOCAL_PATH::"/tmp/o.txt"::SSH_HOST_PATH::"/tmp" #将操作机上的/tmp/o.txt 文件拉取到 ssh 机器的/tmp

SCP_LOCAL_PUSH_PULL::LOCAL_PATH::"/tmp/o.txt"::SSH_HOST_PATH::"/tmp"
#无需密码时不配置 LOCAL_PWD 选项

SCP_LOCAL_PUSH_PULL::LOCAL_PATH::"/tmp/o.txt"::SSH_HOST_PATH::"/tmp"::LOCAL_INTF::"eth1" #指定在获取操作机的 ip 地址时是获取 eth1 上的 ip 地址

SCP_LOCAL_PUSH_PULL::LOCAL_PATH::"/tmp/o"::SSH_HOST_PATH::"/tmp"::LOCAL_ISDIR #指明操作机上的/tmp/o 是一个目录路径

SCP_LOCAL_PUSH_PULL::LOCAL_PWD::"passwd"::LOCAL_PATH::"/tmp/o.txt"::SSH_HOST_PATH::"~" #将存放到 ssh 机器登录用户的 home 目录 ~ 下

`cd /tmp` #ssh 机器进入到/tmp 目录
SCP_LOCAL_PUSH_PULL::LOCAL_PATH::"/tmp/o.txt"::SSH_HOST_PATH::"."
#SSH_HOST_PATH 使用 . 值，表示当前目录，因上一条目录 ssh 机器已经进入到/tmp 目录，所以当前目录即为/tmp 目录，获取到的文件也会被放到/tmp 目录下

```
#SCP_LOCAL_PUSH_PULL 命令同样可使用修饰符进行修饰，修饰符需要在
SCP_LOCAL_PUSH_PULL 命令前，且也可使用变量值
#SCP_LOCAL_PUSH_PULL 命令不可用修饰符 SCP_PWD、TL、NTOL 修饰



TAG::h1,h2::SCP_LOCAL_PUSH_PULL::LOCAL_PWD::"passwd"::LOCAL_PATH::"/tmp/
o.txt"::SSH_HOST_PATH::"/tmp"     #只有标注了 h1,h2 标签的 host 才会执行该
SCP_LOCAL_PUSH_PULL 命令



`cd ~`
VAR :: homePath :: `pwd`
SCP_LOCAL_PUSH_PULL::LOCAL_PATH::"/tmp/o.txt"::SSH_HOST_PATH::"{#homePath
#}/qq/"                #在 SCP_LOCAL_PUSH_PULL 选项中使用变量，各选项都可使用变量
功能。该命令按照前面的理解，变量 homePath 存储的是 ssh 机器的 home 目录路径，那
么获取到的内容将存放到 home 目录下的 qq 目录中
```

还有 SCP_LOCAL_PULL_PUSH 命令，与 SCP_LOCAL_PUSH_PULL 用法完全相同，区别只是在于 SCP_LOCAL_PULL_PUSH 先尝试 PULL 拉取方向，若失败再尝试 PUSH 推送方向。

IF 逻辑结构

```
#最简单的 IF 逻辑结构。IF 最后必须有 ENDIF 作为结尾
IF::`cd ~/abc/`                #判断在 home 目录下是否存在 abc 目录
    `echo "abc exists"`        #显示 abc 目录存在的打印信息
    `rm ~/abc -rf`             #删除掉 abc 目录
ENDIF
#在 IF 与 ENDIF 中的命令可不缩进。使用缩进可使脚本看起来更加清晰。



#带 ELSE 的 IF
IF::`cd ~/abc/`                #判断在 home 目录下是否存在 abc 目录
    `echo "abc exists"`        #显示 abc 目录存在的打印信息
    `rm ~/abc -rf`             #删除掉 abc 目录
ELSE
    `mkdir ~/abc`              #若在 home 目录下不存在 abc 目录，则新建一个。存在则删
除，不存在则新建，这逻辑很奇怪，纯粹是为了说明 IF 语法而已 ^_^
ENDIF
```

```
#IF 中的命令同样支持修饰符，但修饰符 NM 不可用，毕竟 IF 还是依赖于命令的成功与失败。
#修饰符须配置在 IF 与命令中间


IF :: TAG :: h1 :: `cd ~/abc`                #只有标注了 h1 的标签的 host 才会执行该 IF 整个的逻辑结构。没有 h1 标签的 host 不会进入 ELSE 分支，直接跳过整个 IF 逻辑结构，即相应的 ENDIF 后面。
    `mkdir cba`
ELSE
    `mkdir -p ~/abc/cba`
ENDIF
```

```
#IF 与 ELSE 分支中都支持无限嵌套 IF 与 ELSE 分支
IF :: `cd ~`
    IF :: `cd abc`
        `touch o.txt`
    ENDIF
    `touch a.txt`
ELSE
    IF :: `cd /tmp`
        `touch b.txt`
    ENDIF
    `touch c.txt`
    IF :: `mkdir cba`
        IF :: `cd cba`
            `touch d.txt`
        ENDIF
    ENDIF
ENDIF
```

WHILE 逻辑结构

```
#WHILE 逻辑结构，最后须用 ENDWHILE 做为结束


`ps axu | grep application_name | grep -v grep | grep -v tail | awk '{print $2}' | xargs kill`
WHILE :: ASSERT :: 1 :: `ps axu | grep application_name | grep -v grep | grep -v tail | awk
'{print $2}' | wc -l`
    `sleep 5`
ENDWHILE
#上面的几行命令实现使用 kill 命令杀死一进程，在进程未完全退出前，进入 WHILE 循
环，等待 5 秒，再次判断，若仍未杀死继续等待，直到进程被杀死后退出 WHILE 循环


#WHILE 循环也可无限嵌套


#ddp 不支持 break 关键字，但通过 WHILE 与 IF 相结合，可实现类似其他语言的 break
效果
NM :: `ps axu | grep app | grep -v grep | awk '{print $2}' | xargs kill`      #正常杀死进程，进程
通常情况下进入后期处理，并不会马上退出
`sleep 5`
`counter=0`                                          #在 shell 中设置一变量进行计数
WHILE :: ASSERT::1::`ps axu | grep app | grep -v grep| wc -l`   #若进程还未结束，进入循
环；若进程已经退出，则不用进入 WHILE 循环
    `counter=\`expr $counter + 1\``                  #利用 shell 进行计数器加 1
    IF :: ASSERT :: 4 :: `echo $counter`             #判断计数器是否到达 4，若到达 4 进入 IF
        NM :: `ps axu | grep app | grep -v grep | awk '{print $2}' | xargs kill -9`      #强制杀
死，不再等待进程被 kill 时的后期处理，这样下一次 WHILE 判定时应该会退出循环
    ENDIF
    `sleep 3`
ENDWHILE
```

DOWHILE 逻辑结构

```
#DOWHILE 逻辑结构与 WHILE 逻辑结构类似
#DOWHILE 以 DO 开始，以 DOWHILE 结尾

DO                          #DOWHILE 以 DO 作为开始
      `ps axu | grep app | grep -v grep | awk '{print $2}' | xargs kill`
DOWHILE :: ASSERT :: 1 :: `ps axu | grep app | grep -v grep | awk '{print $2}' | wc -l`          #
若进程仍然存在，则返回 DO 再杀进程一次



#DOWHILE 同样支持无限嵌套

#DOWHILE 与 命令之间同样支持修饰符，但不可使用 NM 修饰符
```

返回结果说明

```
#若没有使用 jsonFormat 参数，则为 python 中的 dict 字典结构
#若使用了 jsonFormat 参数，则为 json 格式的字符串

{
    "code" : 0,     #0 代表成功，非 0 代表失败。负数时表示参数错误，可参考附录错误码表；正数时表示操作失败，数值为多少即表示有多少台 host 没有执行脚本命令成功
    "msg" : "...",  #当 code 为 0 时为空字符串;当 code<0 时，表示错误的具体原因;当 code>0 时，内容为操作失败 host 的 ip 地址，用分号;隔开
    "hosts" : {     #hosts 内容为每一个操作的 host 的操作结果，每一项的 key 为 host 的 ip 地址，内容各有三项 code,exit,msg。
        "1.1.1.1" : {           #key 为 host 的 ip 地址
            "code" : 0,     #大于 0 代表成功。1：执行了 EXIT 命令后退出；0：执行完所有脚本命令，且未遇到 EXIT 命令；-1：登陆失败；-2：执行某一条脚本命令时失败；-3：执行 EXIT 命令时失败；-4：在设置 sshHomePath 时失败
            "exit" : 0,     #exit 相当于操作的返回值，成功时若没有碰到 EXIT 命令的情况下，默认设置为 0，若碰到 EXIT 命令，则为 EXIT 命令中的数值。当 code 为 1 失败时，该值为错误码，具体含义参考附录中的错误码表
            "msg" : "..."   #操作成功或失败时相应的辅助提示信息。成功时若没有碰到 EXIT 命令，则为空字符串；若碰到 EXIT 命令，则为 EXIT 命令中的字符串，若无则为空字符串。code 为 1 失败时，为失败相关的辅助提示信息
        },
        "2.2.2.2" : {
            "code" : 0,
            "exit" : 0,
            "msg" : ".."
        },
        ...
    }
}
```

配置文件

在 ddp/conf 下有一个配置文件 ddp.conf，用以设置脚本默认参数。配置选项如下

```
#-*- coding:utf-8 -*-
## 请注意，注释只能写在单独的一行，不能写在配置语句的后面

[ddp]
#同时并发执行的 host 个数
running_host = 10

#执行失败时的重试次数,若为 0 表示不重试
retry_times = 0

#保存成功结果的文件
success_hosts_file = success_hosts.txt

#保存失败结果的文件
error_hosts_file = error_hosts.txt

#当 host 执行命令出错时，返回的结果中针对该 host 的 exit 无任何意义，所给予的无效值
exit_useless_value = -99999

[pyssh]
#执行命令默认超时等待时间,单位为秒
pyssh_default_timeout = 20

#运行每条命令后的休眠时间,单位为秒
sleep_time_after_sendline_sec = 0.05

#执行 scp 命令时的等待超时时间,单位为秒
scp_wait_timeout_sec = 1200

#执行 scp 的测试时的等待超时时间,单位为秒
scp_test_timeout_sec = 20

#登陆时的等待超时时间,单位为秒
login_wait_timeout_sec = 20

#本地操作机器的网卡接口名称
local_interface = eth1
```

API 接口

API 接口只适用于 python 程序调用，将 ddp 做为模块导入即可使用。

API 中参数保持与脚本参数一致，只是全都使用全称，不使用缩写，参数说明及意义可参考“脚本使用参数”这一节。

```
# 直接可调用的主函数
def main(hostsFile=None, cmdsFile=None, hostsString=None, execCmds=None, output=None,
retryTimes=None, workersNO=None, quiet=False, jsonFormat=False, quietFiles=False,
printResult=False, successHostsFile=None, errorHostsFile=None)
    #参数解释参考 “脚本使用参数” 章节
    #返回结果参考 “返回结果说明” 章节
```

```
#使用该函数需要 getHostList 和 getFirstCmdNode 两个函数的配合使用
def ddp(hostList, firstCmdNode, output=None, retryTimes=None, workersNO=None,
quiet=False, jsonFormat=False, quietFiles=False, printResult=False, successHostsFile=None,
errorHostsFile=None)
    #参数 hostList 使用 getHostList 的返回值
    #参数 firstCmdNode 使用 getFirstCmdNode 的返回值
    #其余参数参考 “脚本使用参数” 章节
    #返回结果参考 “返回结果说明” 章节
```

```
#解析数据获取 host 的列表
def getHostList(hostsData)
    #hostsData : 字符串，未解析前的 hosts 内容。可为从 hosts 文件所获取到的文本内容，
也可以是拼接好的字符串。
    #返回结果 : dict 字典结构。成功时返回{'code':0, 'hostList': ...}，'hostList'的 key 所对应
的 value 即为解析出来的 host 的 list。失败时返回{'code':..., 'msg':...}，'code'的 key 所对应的
value 为非 0 的整数值的错误码，参考附录中的错误码表；'msg'的 key 所对应的 value 为错
误的辅助提示信息。
```

```
#解析出脚本命令的逻辑结构，并返回首个命令
def getFirstCmdNode(cmdsData)
    #cmdsData: 字符串，未解析前的命令脚本 cmds 的内容。可为从 cmds 文件所获取到的
文本内容，也可以是拼接好的字符串。
    #返回结果 : dict 字典结构。成功时返回{'code':0, 'cmdNode': ...}，'cmdNode'的 key 所对
应的 value 即为解析出来的 firstCmdNode。失败时返回{'code':..., 'msg':...}，'code'的 key 所对
应的 value 为非 0 的整数值的错误码，参考附录中的错误码表；'msg'的 key 所对应的 value
为错误的辅助提示信息。
```

```
## API 使用范例

#将 ddp 整个目录做为一个模块，放入所在工程的 lib 目录或其他目录下

#首先将存放 ddp 目录的路径加入 sys.path 中
sys.path.append( "替换成存放 ddp 目录的路径" )
from ddp import ddp


#第一种方式调用  ddp.main  接口
ddpRet = ddp.main(hostsFile="/tmp/test_hosts.txt", cmdsFile="/tmp/test_cmds.txt")
#这里只使用了两个参数，其余参数都使用默认值
#下面解析 ddpRet 结果值就可以了



#第二种方式调用  ddp.ddp  接口
#使用  ddp.ddp  接口需要使用到  getHostList  与  getFirstCmdNode  接口

getRet = ddp.getHostList(hostsData = open('/tmp/test_hosts.txt', 'r').read() )
# hostsData 也可以是字符串，如
hostsData='1.1.1.1,"user","passwd";"www.test.com","user","passwd"
#解析  getRet
if 0 != getRet['code']:
    #错误处理
    pass
else:
    hostList = getRet['hostList']

getRet = ddp.getFirstCmdNode(cmdsData = open('/tmp/test_cmds.txt', 'r').read() )
# cmdsData 也可以是字符串，如  cmdsData='`cd ~`;`uname -a`;'
#解析  getRet
if 0 != getRet['code']:
    #错误处理
    pass
else:
    firstCmdNode = getRet['cmdNode']

ddpRet = ddp.ddp(hostList=hostList, firstCmdNode=firstCmdNode)
#下面解析 ddpRet 结果值就可以了
```

附录一：安装步骤

ddp 无需安装，把 ddp 压缩包解压之后，即可直接使用。

附录二：错误码表

脚本参数错误：-1XXX

-1001：脚本命令参数无法正确识别
-1002：hostsFile 参数所指路径非文件路径
-1003：hostsFile 参数所指路径存在，但该文件不具备读权限
-1004：cmdsFile 参数所指路径非文件路径
-1005：cmdsFile 参数所指路径存在，但该文件不具备读权限
-1006：errorHostsFile 参数所指路径已经存在，但非文件
-1007：errorHostsFile 参数所指路径不具备写权限
-1008：errorHostsFile 参数所指路径在创建文件时出错
-1009：successHostsFile 参数所指路径已经存在，但非文件
-1010：successHostsFile 参数所指路径不具备写权限
-1011：successHostsFile 参数所指路径在创建文件时出错
-1012：output 参数所指路径已经存在，但非文件
-1013：output 参数所指路径不具备写权限
-1014：output 参数所指路径在创建文件时出错
-1015：hostsFile 参数为空字符串
-1016：cmdsFile 参数为空字符串
-1017：hostsFile 参数与 hostsString 参数需要使用其中一个，但不能同时使用
-1018：cmdsFile 参数与 execCmds 参数需要使用其中一个，但不能同时使用
-1019：hostsFile 参数或 hostsString 参数中没有 hosts
-1020：cmdsFile 参数或 execCmds 参数中没有 cmds
-1100：在处理脚本操作结果集的时候遇到参数错误，此时可认为脚本已经执行了一遍，但执行结果可成功可失败

Host 文件解析错误：-2XXX

-2001：词法错误，非法字符
-2002：语法错误，不符合语法规范
-2003：未知错误

Cmd 文件解析错误：-3XXX

-3001：词法错误，非法字符
-3002：语法错误，不符合语法规范
-3003：未知错误

Login 错误：-4XXX

-4001：登陆密码不正确
-4002：登陆成功，但未能获取到输入提示符 PS1
-4003：登陆时发生 EOF
-4004：登陆超时错误

Scp 错误：-5XXX

-5001：与 local 机器进行 scp 操作时，在获取 local 机器的当前登陆用户名时发生异常
-5002：与 local 机器进行 scp 操作时，获取到 local 机器的 IP 为空
-5003：与 local 机器进行 scp 操作时，在获取 local 机器的 IP 时发生异常
-5004：与 local 机器进行 scp 操作时，指定 local 机器上的操作目录应该是绝对路径
-5005：scp 命令需要密码，但未提供
-5006：scp 命令失败，因为密码不正确或权限被禁止
-5007：scp 命令执行结束，但通过 echo $?判定时未能获取到 0 而判定为失败
-5008：scp 命令失败，EOF
-5009：scp 命令失败，超时
-5010：scp 命令无需输入密码就执行结束，但通过 echo $?判定时未能获取到 0 而判定为失败
-5011：从 local 机器向 SSH 机器发起 scp 的推送操作时，需要密码，但未提供
-5012：从 local 机器向 SSH 机器发起 scp 的推送操作时，密码不正确或权限被禁止
-5013：从 local 机器向 SSH 机器发起 scp 的推送操作时，有经过输入密码操作，操作执行结束，但未能通过验证
-5014：从 local 机器向 SSH 机器发起 scp 的推送操作时，超时，有经过输入密码操作
-5015：从 local 机器向 SSH 机器发起 scp 的推送操作时，未经过输入密码操作，操作执行结束，但未能通过验证
-5016：从 local 机器向 SSH 机器发起 scp 的推送操作时，超时，未经过输入密码操作
-5017：执行 SCP_LOCAL_PULL_PUSH 操作时，拉取与推送双方向都失败
-5018：从 SSH 机器向 local 机器发起 scp 的拉取操作时，操作失败
-5019：从 SSH 机器向 local 机器发起 scp 的拉取操作时失败，因为未能通过前期的拉取测试
-5020：从 local 机器向 SSH 机器发起 scp 的推送操作失败，因为未能通过前期的推送测试
-5021：执行 SCP_LOCAL_PUSH_PULL 操作失败，拉取与推送双方向都失败

SSH 机器上命令操作错误：-6XXX

-6001：获取 SSH 机器上当前目录路径时不成功
-6002：获取 SSH 机器上当前登录用户的 home 目录时不成功
-6003：获取 SSH 机器是 64 位还是 32 位系统时不成功
-6004：SSH 机器执行完一条命令后，在获取 echo $?结果时发生 EOF
-6005：SSH 机器执行完一条命令后，在获取 echo $?结果时，超时
-6006：SSH 机器执行完一条命令后，在获取 echo $?结果时，发生异常
-6007：在 SSH 机器上执行一条命令时，发生 EOF
-6008：在 SSH 机器上执行一条命令时，超时
-6009：在 SSH 机器上执行一条命令时，命令执行完毕，但获取其 echo $?的值时非 0

执行脚本命令时的错误：-7XXX

-7001：在 EXIT 命令中的 msg 部分含有{#varName#}，但 varName 不符合变量命名规范
-7002：在 EXIT 命令中的 msg 部分含有{#varName#}，但 varName 是未定义使用的变量名
-7003：在脚本命令中含有{#varName#}，但 varName 不符合变量命名规范
-7004：在脚本命令中含有{#varName#}，但 varName 是未定义使用的变量名
-7005：在脚本命令修饰符中含有{#varName#}，但 varName 不符合变量命名规范
-7006：在脚本命令修饰符中含有{#varName#}，但 varName 是未定义使用的变量名
-7007：脚本命令执行成功，但与 ASSERT 设定的值不符