ddp

Author: vincentzhwg@gmail.com

ddp 介绍

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

* 无需编写 python 脚本，直接使用命令脚本与机器列表即可完成所需操作
* ssh 连接为长连接，在执行脚本命令过程中一直为同一 ssh 连接
* 分发传输文件到各机器上
* 在各机器上执行一系列 shell 命令，并返回相应的命令结果
* 支持并发执行，也可串行执行
* 命令脚本可具备逻辑结构，可用于状态判定，根据状态选取相应的操作
* 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 2.7 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：**1.1.5**
Date：2013-07-16
更新说明：

修复 scp 操作时，当从操作机往 SSH 推送时，实际失败却判定为成功的 bug
修改 EXIT 命令使用第二 string 选项时的出错 bug

版本：**1.1.4**
Date：2013-06-25
更新说明：

修复当错误退出时不关闭 ssh 连接的 bug

版本：**1.1.3**
Date：2013-06-13
更新说明：

增加命令修饰符 LOCAL_CMD 用于在操作机上执行命令
修复若干 bug

版本：**1.1.2**
Date：2013-05-27
更新说明：

增加 ADD_USER 命令，方便在系统中增加用户，需要使用 root 帐户登录 SSH 机
修正当命令脚本或 hosts 文件有错时，不能完全退出的 bug

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

关键术语说明

术语名称	说明
操作机	也称为本地机器，运行 ddp 的所在机器。
SSH 机	也称为远程机器，在 ddp 运行过程中通过 ssh 连接登陆的机器。
ddp 分隔符	英文半角状态下的连续的两个冒号字符，即 :: 。
空白字符	英文半角状态下的空格字符或 tab 字符。注意：换行符不属于空白字符。
字符串	英文半角状态下的两个配对的双引号及之间的内容，将被视为一串字符串。两个配对双引号之间的内容，则为字符串的内容。如"a"，"b"，"abc"，这三个都是字符串，字符串内容依次是 a，b，abc。若想在字符串内容中含有双引号，可使用反斜杠字符进行转义，如 "a\"b" ，是一字符串，字符串内容为 a"b 。字符串能够支持反斜杠转义的字符有：\t （tab 字符）, \n （换行符）, \" （双引号）, \\ （反斜杠自身）。
空字符串	两个双引号之间无任何字符，即 "" ，视为空字符串。
变量	以下划线或字母作为首字符，后接任意多个下划线、数字或字母所组成的字符串，两端不需要有双引号。如 a , b , _a , var , v1, v2 。
string	表示字符串形式，字符串内容两端需要有 " 双引号
integer	表示整数形式。取值范围包括负整数，0，正整数。注意：正整数直接写整数值即可，无需加上 + 符号。
cmd	指代脚本命令，包括普通的 shell 命令，及特殊关键字定义的命令。
cmd_adjs	指代命令修饰符，其中包含一个或多个修饰符，各修饰符之间用 ddp 分隔符分隔开。如 TL :: 60 ， TL :: 60 :: ASSERT :: "success" 等。
cmds	指代一条或多条命令。各命令之间可用换行符分隔，也可用分号 ; 分隔。这些命令可带上修饰符，也可不带。

ddp 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	把执行过程中的信息保存到指定文件中，保存的数据中会带有一些执行过程中的附加的提示信息，如命令内容，执行成功或失败的提示等
oo	onlyOutput	只把运行过程中成功执行的命令输出保存到指定文件中，无其它任何多余的输出

ddp 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

#串行方式执行，即将 w 参数设为 1，非 1 时都是表示并发执行。
./ddp.py -l test_hosts.txt -c test_cmds.txt -w 1

#将执行过程信息输出到 /tmp/o.txt 文件，并将执行过程中成功命令的输出（只有输出，
没有多余的执行过程中信息）保存到 /tmp/oo.txt 文件中
./ddp.py -l test_hosts.txt -c test_cmds.txt -o /tmp/o.txt -oo /tmp/oo.txt
```

注释规范

在说明其他规范之前，先说明注释规范。注释规范适用于 hosts 文件规范中，也适用于命令脚本文件规范中。

注释以 # 符号开始。可以整行是注释，也可以在行内容的最后面加注释，以下是范例说明。

```
#第一个非空白字符为 # ,则表示这一行为注释行

        #这也是注释行，只需要行的首个非空白字符为 # 即可

10.10.10.10        "user"        "passwd"      #注释也可写在一行最后面，只需以 # 开始，hostsFile 文件中注释行范例

`cd /tmp`      #命令脚本中的注释范例，该命令为进入 /tmp 目录
```

机器列表文件规范

Hosts 文件是用于列出 SSH 机的列表，应用于参数中的 l (hostsFile) 选项。

Hosts 文件有以下特性：

1.以换行符或 ; 分号字符（英文半角状态）为区分各 SSH 机的分隔符。通常使用一行中的各选项内容来指定一台 SSH 机的定义，这是最常用的用法。当想要在一行中定义多台 SSH 机时，各 SSH 机之间须以 ; 分隔。

2.在一台 SSH 机的定义中，有多个选项需要设定。各选项之间以至少 1 个以上的空白字符，或 , 逗号字符（英文半角状态），将各选项隔开。选项设定有顺序要求，依次为：

选项顺序	选项名称	说明	样例
1	**主机名**	可用 IP 地址或域名。使用 IP 地址时，直接写上 IP 地址即可，无需双引号；使用域名时，使用字符串形式。	IP 地址：10.0.2.151 域名："www.ddp.com"
2	**用户名**	ssh 登录时的帐户名，使用字符串形式。	"root"
3	用户密码	ssh 登录时帐户的密码，使用字符串形式。若操作机已与 SSH 机建立了无需密码登录的机制，可不用给出该选项，或给出一空字符串。注意：若在该选项中使用空字符串，相当于不配置此选项，此种用法可用于已经在操作机与 SSH 之间建立了无需密码登录的机制，但又想配置指定 ssh 端口时使用。	"password"
4	端口	操作机在登录 SSH 机时的 ssh 端口，直接以整数给出即可。一般无需配置此项，这样 ssh 会使用自身或系统指定的端口进行登录连接。	22

加粗显示的 1，2 选项为必选项。

3.SSH 机可加上标签分类。一台 SSH 机可拥有多个不同的标签，一个标签可用于多台 SSH

机。标签的用处在于，当在命令脚本中，对于某一条命令，若使用了 TAG 修饰符指定了某一些标签，那么只有拥有同样标签的 SSH 机才能执行这一条命令，其余的 SSH 机则跳过不执行此条命令。给 SSH 机加标签的方法，在主机名选项之前加标签，标签与主机名之间用 ddp 分隔符分隔开，在 ddp 分隔符前面写上做为标签名称的变量即可。给一台 SSH 机赋予多个不同的标签，在各标签之间用逗号 , 分隔开。

4.在同一个文件中 SSH 机的主机名不可以相同或重复，若相同或重复将提示相应错误。

5.在文件中允许存在空行或注释行。

Hosts 文件范例如下：

```
    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"     #该行第一项的主机名使用域名，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

10.10.10.16        "user"    ""    36000        #在操作机与 SSH 机已经建立了免密码登录机制后，无需配置密码，但又想配置端口，那么，即可把密码项用空字符串表示，这与不配置密码项效果是等同的，同时又可配置端口项
```

给 SSH 机配置上标签的范例如下：

```
h1::10.10.10.17     "user"     "passwd"              #符号 :: 为 ddp 分隔符，当需要给 host 标注标签时，需要使用该符号。标签值在 ddp 分隔符前面，host 在分隔符后面。该行表示该 host 具有 h1 标签。标签值内容采用变量形式。


h1    ::      10.10.10.18   "user"     "passwd"       #ddp 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"       #一个 SSH 机可拥有多个标签，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
```

同一行中配置多个 SSH 机的范例如下：

```
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
```

脚本命令文件规范

脚本命令文件的内容为在 SSH 机所执行的一系列的脚本命令，按照文件的行号顺序依次执行，应用于参数中的 c (cmdsFile) 选项。

脚本命令

脚本命令文件是由一条条的脚本命令组成的，那么，什么样的文本内容可以被称为一条脚本命令呢？脚本命令有以下两种类别：

1.普通的 shell 命令，命令的两端需有 ` （反引号，这个字符对应的按键一般位于键盘的左上角，ESC 按键的下方）包围起来。如 `cd ~` ， ``tar zxf /tmp/a.tar.gz` 等。若在命令中需要使用到反引号，需要在其前面加上反斜杠进行转义，如 c=`expr 2 + 1` ，此命令计算 2 与 1 相加之后的和赋给 shell 变量 c，脚本命令表示为 `c=\`expr 2 + 1\`` 。

2.由特殊关键字所定义的命令，此类型的命令一般由各选项拼接成真正执行的命令，各选项之间用 ddp 分隔符进行分隔，ddp 分隔符前后可有任意个空白字符。如 SCP_LOCAL_PUSH_PULL ， SCP_LOCAL_PULL_PUSH ， ADD_USER ， EXIT 等。

脚本命令通常在一行中进行定义。若想在一行中定义多行，与机器列表文件类似，同样使用分号将各命令分隔开。命令范例如下：

```
`uname -a`                #命令必须用前后用反引号 ` 字符包围起来,该字符一般位于键盘左上角（ESC 键下方的按键）

    `whoami`      #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
```

第一种类型的普通 shell 命令，两端加反引号所组成的脚本命令，就无需太多介绍了。下面将一一介绍第二种由特殊关键字所定义的命令。

关键字命令： ADD_USER

ADD_USER 命令用于在系统中增加用户，使用此命令时需要用 root 帐户登陆 SSH 机。

ADD_USER 能使用以下选项，加粗选项为必选项，选项顺序可任意。

选项	说明	使用格式
USER_NAME	用户名称	USER_NAME :: string
USER_PWD	用户密码	USER_PWD :: string
USER_HOME	用户主目录	USER_HOME :: string
GROUP_NAME	用户的组名称	GROUP_NAME :: string

ADD_USER 使用范例如下：

```
# ADD_USER  使用范例
ADD_USER :: USER_NAME :: "u1"          #创建 u1 用户

ADD_USER :: USER_NAME :: "u2" :: USER_HOME :: "/home/u2" :: USER_PWD :: "u2"
      #创建 u2 用户，指定其 home 目录及密码

ADD_USER :: USER_NAME :: "u3" :: USER_HOME :: "/home/u3" :: GROUP_NAME :: "g3"       #创建 u3 用户，指定其 home 目录及组名称，请自行确认组名称 g3 存在，否则该命令将不成功
```

关键字命令：SCP_LOCAL_PUSH_PULL

SCP_LOCAL_PUSH_PULL 命令，是使用 scp 方式将操作机上的文件或目录传输至 SSH 机上。传输时会首先探测从操作机往 SSH 机推送方向是否可行，若可行则使用推送方式，若不可行，则再探测从 SSH 机拉取操作机的方向是否可行，若可行则使用拉取方式，若不可行则此命令失败。

SCP_LOCAL_PUSH_PULL 可使用以下选项，加粗选项为必选项，选项顺序可任意。

选项	说明	使用格式
LOCAL_PATH	操作机上待被传输的文件或目录路径，此路径必须是绝对路径。	LOCAL_PATH :: string
SSH_HOST_PATH	SSH 机上存放从操作机上传输过来的数据的文件或目录路径，此路径可以是相对路径或绝对路径。	SSH_HOST_PATH :: string
LOCAL_PWD	操作机当前登陆帐户的密码。当推送方向不成功时，需要使用到拉取方向时，若操作机与 SSH 无免密码设置时，需要此选项。	LOCAL_PWD :: string
LOCAL_ISDIR	用来表明 LOCAL_PATH 路径所指向的是目录。未使用此选项时，则表明是文件。	LOCAL_ISDIR
LOCAL_PORT	在拉取方向时，指定操作机的 scp 端口。	LOCAL_PORT :: integer
LOCAL_INTF	在拉取方向时，需要使用到操作机的 IP 地址，该选项用于指定从哪个网卡接口获取 IP 地址。默认值在配置文件中配置，一般为 eth0 或 eth1 。	LOCAL_INTF :: string

SCP_LOCAL_PUSH_PULL 范例如下：

```
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 机的/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"

SCP_LOCAL_PUSH_PULL::LOCAL_PATH::"/tmp/o.txt"::SSH_HOST_PATH::"/tmp"::LOCAL_INTF::"eth1"   #指定在拉取方向时，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 机登录用户的 home 目录 ~ 下

`cd /tmp`     #SSH 机进入到/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
```

关键字命令：SCP_LOCAL_PULL_PUSH

SCP_LOCAL_PULL_PUSH 与 SCP_LOCAL_PUSH_PULL 命令一样，也是使用 scp 方式将操作机上的文件或目录传输至 SSH 机上，所能使用的选项也一样，区别只是在于先尝试拉取方向，再尝试推送方向。SCP_LOCAL_PULL_PUSH 的选项及使用方法，请参考 SCP_LOCAL_PUSH_PULL 的说明。

关键字命令：EXIT

EXIT 命令用于终止命令脚本的执行，并返回相应的整数数值，还可以附加返回的字符串内容，整数数值与字符串之间需用逗号分隔开。EXIT 命令，不能使用任何的命令修饰符（后面会介绍命令修饰符）。EXIT 命令的使用形式如下：

EXIT :: integer [, string]

integer 项是必须项，string 项属于可选项，若使用了 string 项，两者之间用逗号分隔开。若未配置 string 项，那么返回的附加字符串为空字符串。

若是正常执行完命令脚本中所有命令，未曾遇到 EXIT 命令，则默认返回的 integer 为 0，string 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
#里面用到了一些命令修饰符及 IF 的逻辑结构，后面都会有介绍
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
```

脚本变量

命令脚本文件中可使用变量功能。使用变量时，在变量名称的左端需有 {# 字符，右端需有 #} 字符，且之间不能有任何空白字符。如有一变量名称为 a ，在使用时需要写成 {#a#} 的形式。脚本变量的特性如下：

1.脚本命令中第一类型的 shell 命令的任何地方都可以使用变量，在实际执行命令时，会先把变量替换为实际的变量内容后，再执行此命令。

2.脚本命令选项或命令修饰符中使用 string 形式的地方，在两端的双引号之间可以使用变量。如在 SSH_HOST_PATH 选项中使用变量，SSH_HOST_PATH :: "{#sshHomePath#}/abc/" ，其中 sshHomePath 是一个变量，使用时在其两端加上了符号变成了 {#sshHomePath#} ，sshHomePath 变量属于预定义变量，其值为 SSH 机的 home 目录路径，可直接使用，后续会介绍，在实际执行此命令时，会先用 sshHomePath 变量的实际内容替换进去，最终 SSH_HOST_PATH 的路径指向 SSH 机 home 目录下的 abc 目录。

3.变量的名称需遵循变量命名规范，以下划线或字母作为首字符，后接任意多个下划线、数字或字母所组成的字符串，两端不需要有双引号。如 a , b , _a , var , v1, v2 。

4.变量的定义，可通过命令修饰符 VAR 进行定义，在命令修饰符这一章节中会详细介绍。

脚本变量中有一部分预定义变量，可直接进行使用。

脚本命令中的预定义变量

预定义变量名称	说明
sshHostName	机器列表文件中 SSH 机的主机名的名称字符串。若配置 host 时给出的为 IP 地址串则为 IP 地址串；若为域名字符串，则为域名字符串。
sshIP	机器列表文件中 SSH 机的主机名给出的是 IP 地址才能使用此变量，若给出的是域名（如 www.test.com），则为空字符串。
sshUser	机器列表文件中 SSH 机的登陆帐户。
sshPort	机器列表文件中 SSH 机的 ssh 端口。若未给出，为空字符串。
sshPassword	机器列表文件中 SSH 机的登陆密码。若未给出，为空字符串。
sshHomePath	SSH 机上登陆帐户的 home 目录路径。

变量定义及使用范例如下：

```
`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，后面还会详细介绍 VAR 修饰符
`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_LOCAL_PUSH_PULL :: LOCAL_PATH :: "/tmp/a.txt" :: SSH_HOST_PATH :: "{#homePath#}/abc/"     #在 SSH_HOST_PATH 选项中使用变量
```

命令修饰符

命令修饰符，是用于给命令加上一些修饰符，赋予命令一些特殊的效果。命令修饰符的使用方法如下：

1.命令修饰符的位于命令之前，修饰符与命令之间用 ddp 分隔符 :: 隔开。

2.一条命令可带多个修饰符，不过，有些修饰符之间会有冲突而不能共用。

3.多个修饰符时，对于每一个修饰符并没有顺序上的要求，可随意排列。各个修饰符之间用 ddp 分隔符分隔开。

4.同一个修饰符在一条命令中只能使用一次。

下面将一一介绍各命令修饰符。

修饰符：TL

TL，即 TimeLimit 的简称，用于设置命令执行时的等待超时时间，时间单位为秒钟。若不设置，将使用配置文件所配置的默认值，一般为 20 秒。此修饰符的用处在于，若某条命令的执行时间会比较长，为了避免命令未执行完，却已经超过了等待的超时时间，导致认为此条命令执行失败的情况，则可以使用 TL 修饰符设定等待超时的时间。使用形式如下：

TL :: integer :: cmd

cmd 指代命令。integer 的值必须大于 1。TL 修饰符与 NTOL 修饰符存在冲突，不可共用。

TL 修饰符使用范例如下：

```
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，可避开一般 20 秒的命令超时等待时间的限制
```

修饰符：NTOL

NTOL：即 NoTimeOutLimit 的简称，使被修饰命令没有执行的等待超时时间的限制，会一直等到命令执行完毕才返回结果或执行下一条命令。NTOL 与 TL 存在冲突，不可共用。使用形式如下：

NTOL :: cmd

cmd 指代命令。

NTOL 修饰符的使用范例如下：

```
NTOL :: `sleep 150`        #使用 NTOL 修饰睡眠 150 秒的命令，这样即可避开一般 20 秒的命令超时等待时间的限制
```

修饰符：VAR

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该变量。使用形式如下：

VAR :: varName :: cmd

varName 为变量名称，cmd 为脚本命令。当脚本命令 cmd 成功执行之后，cmd 命令的终端输出（并不是命令的返回值），将做为变量 varName 的值。变量名称的命名规范为以下划线或字母作为首字符，后接任意多个下划线、数字或字母所组成的字符串，两端不需要有双引号。如 a , b , _a , var , v1, v2 。

VAR 修饰符的使用范例如下：

```
`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，后面还会详细介绍 VAR 修饰符
`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
```

关于变量的具体说明，可参考 **脚本变量** 这一章节。

修饰符：TAG

TAG：标签修饰符，用于给被修饰的命令加上标签，只有在机器列表中拥有相同标签的 SSH 机才会执行此命令，其余的 SSH 机则跳过此命令。使用形式如下：

TAG :: tag1 [, tag2 , tag3 , ...] :: cmd

tag1，tag2，tag3 等为标签名称，其命名规范与变量名称的命令规范一样，即以下划线或字母作为首字符，后接任意多个下划线、数字或字母所组成的字符串，两端不需要有双引号。如 a , b , _a , var , v1, v2 。cmd 指代命令。该使用形式，表示 cmd 命令被打上了标签，只有在机器列表文件中拥有同样的 tag1，tag2，tag3 等的 SSH 机才执行此命令，其它的 SSH 则跳过不执行此命令。

TAG 使用范例如下：

```
TAG :: h0 :: `cp /tmp/`          #只有同样标注了 h0 标签的 host 才会执行该命令

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
```

修饰符：ASSERT

ASSERT：表示假设、设定的意思，使用此修饰符的命令，首先命令得执行成功，其次命令的终端输出（非返回值）必须与 ASSERT 假定的值相同，否则认为命令执行失败。ASSERT 与 NM 修饰符冲突，不可共用。

ASSERT 使用形式如下：

ASSERT :: integer :: cmd

或

ASSERT :: string :: cmd

cmd 指代命令。ASSERT 有两种使用形式，一种假定命令输出的整数数值，一种假定命令输出的字符串。

ASSERT 使用范例如下：

```
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
```

修饰符：NM

NM：即 NoMatter 的简称，被 NM 修饰的命令，不关心其执行是否成功，都会接下去执行下一条命令。正常情况下，只有一条命令执行成功了，才会接下去执行下一条命令，若有一条命令执行失败了，将直接退出整个命令脚本，不再执行此条命令下面的命令。所以，当有些命令，如强制杀死某些进程，但该进程又不一定在运行中，只是为了确保在接下来的命令执行之前，那些进程不会存在，即可以使用 NM 修饰强制杀死那些进程的命令，以免那些进程原本未运行导致执行失败，而未能运行接下去的命令。NM 与 ASSERT 修饰符冲突，不可共用。

NM 使用范例如下：

```
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

NM :: `nohup ./start.sh 2>&1 &`      # 建议在 nohup 命令中都带上 NM 修饰符。验证是否真的启动成功，可接下来使用 ASSERT 修饰的命令来判定
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
```

修饰符：SCP_PWD

SCP_PWD：即 scp password，用于执行 scp 传输命令时，当需要密码时提供密码。若所执行的 scp 传输命令无需密码时，可不使用该修饰符。使用形式如下：

SCP_PWD :: string :: cmd

cmd 指代命令，通常为 scp 类型的命令。

SCP_PWD 使用范例如下：

```
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
```

修饰符：LOCAL_CMD

LOCAL_CMD：即 local command，是个比较特殊的修饰符，表示此条命令是一条本地命令，此命令将在操作机上执行，而不是在 SSH 机上执行。使用形式如下：

LOCAL_CMD :: cmd

cmd 提供命令。

LOCAL_CMD 使用范例如下：

```
`cd /tmp/`              #进入 tmp 目录，该命令在 SSH 机上执行
`echo -e "{#sshHostName#}\t{#sshUser#}" > {#sshHostName#}.txt`       #生成一 txt 文件，该命令在 SSH 机上执行
TL :: 30 :: LOCAL_CMD :: SCP_PWD :: "{#sshPassword#}" :: `scp {#sshUser#}@{#sshHostName#}:/tmp/{#sshHostName#}.txt /tmp/abc/`   #加上 LOCAL_CMD 修饰符，表示此条命令是在操作机上执行。该命令实现将生成的 txt 文件通过 scp 方式传回到操作机上的 tmp 目录下


VAR :: aSize :: LOCAL_CMD :: `ls -l --color=never --time-style=full-iso /tmp/a.txt | awk '{print $5}'`    #操作机上执行，获取 a.txt 文件大小放到变量 aSize 中
`cd /tmp/`        #SSH 机进入/tmp 目录
SCP_LOCAL_PULL_PUSH :: LOCAL_PWD :: "password" :: LOCAL_PATH :: "/tmp/a.txt" :: SSH_HOST_PATH :: "."       #将 a.txt 从操作机传输至 SSH 机
ASSERT :: "{#aSize#}" :: `ls -l --color=never --time-style=full-iso a.txt | awk '{print $5}'`
#通过 ASSERT 验证传输到 SSH 机上的文件大小是否与操作机上的一致
```

至此，所有修饰符介绍完毕。最后，给出一些修饰符共用的范例如下：

```
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

TL :: 60 :: ASSERT :: "success" :: `./run.sh`   # 运行 run.sh 脚本，可能运行需要的时间较长，所以使用 TL 修饰符将超时等待时间设为 60 秒。正常情况下，run.sh 的输出为 success ，使用 ASSERT 修饰符假定输出一定要为 "success" 字符串，否则命令将被认为执行失败，而不再执行后面的命令。
```

命令语法逻辑结构

ddp 的命令脚本文件中支持一定的语法逻辑结构，通过语法逻辑结构可以实现更加方便，更加复杂或巧妙的功能。所有的语法逻辑结构都支持无限嵌套使用。下面将一一介绍 ddp 所支持的语法逻辑结构。

逻辑结构：IF

IF 逻辑结构，根据命令执行的成功与失败，选择相应的逻辑分支执行相关命令。使用形如有两种，一种不带 ELSE 分支的，一种带 ELSE 分支的，具体如下：

不带 ELSE 分支：

IF :: [cmd_adjs ::] cmd

cmds

ENDIF

带 ELSE 分支：

IF :: [cmd_adjs] :: cmd

cmds

ELSE

cmds

ENDIF

其中，中括号括住的内容为可选项，cmd_adjs 为命令修饰符，可一个或多个修饰符，但不可使用 NM 修饰符。cmd 指代单条命令。cmds 指代一条或多条命令，修饰符可带可不带。IF 逻辑结构支持嵌套使用，即在 IF 的逻辑结构中，仍可再使用 IF，在使用范例中会给出样例。

在使用 IF 逻辑结构时，是否对命令进行缩进排版，无强制性要求。建议缩进排版，这样命令脚本看起来更加简洁，方便，易于察看。

IF 逻辑结构使用范例如下：

```
#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

#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。
IF :: TAG :: h1 :: `cd ~/abc`
    `mkdir cba`
ELSE
    `mkdir -p ~/abc/cba`
ENDIF
```

IF 逻辑结构嵌套使用的范例如下：

```
#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
```

逻辑结构：WHILE

WHILE 逻辑结构，用于实现循环功能。使用形式如下：

WHILE :: [cmd_adjs ::] cmd

cmds

END

其中，中括号括住的内容为可选项，cmd_adjs 为命令修饰符，可一个或多个修饰符，但不可使用 NM 修饰符。cmd 指代单条命令。cmds 指代一条或多条命令，修饰符可带可不带。

WHILE 逻辑结构使用范例如下：

```
#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



#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
#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
WHILE :: ASSERT::1::`ps axu | grep app | grep -v grep| wc -l`
    `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
```

逻辑结构：DOWHILE

DOWHILE 逻辑结构与 WHILE 逻辑结构类似，都是实现循环功能，区别只是在于 DOWHILE 先执行了一遍循环体后，才来执行判断。使用形式如下：

DO :: [cmd_adjs ::] cmd

cmds

DOWHILE :: [cmd_adjs ::] cmd

其中，中括号括住的内容为可选项，cmd_adjs 为命令修饰符，可一个或多个修饰符，但不可使用 NM 修饰符。cmd 指代单条命令。cmds 指代一条或多条命令，修饰符可带可不带。

DOWHILE 使用范例如下：

```
#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
```

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：

```
## 配置文件中，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

#当执行命令出错时，返回结果中针对 SSH 机的 exit 整数值无任何意义，给予无效值
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 **ddp 使用参数** 这一章节。

第一种调用方式使用 main 函数接口，其可以单独使用，即可完成所需的 ddp 功能。main 接口定义如下：

```
# 直接可调用的 main 接口函数
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
```

第二种调用方式，使用 ddp 函数接口，但其需要 getHostList 与 getFirstCmdNode 两个接口的配合使用，因为 ddp 函数接口其中有两个参数是分别使用它们的返回值。以下是它们的接口定义。

ddp 函数接口定义如下：

```
#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
```

getHostList 接口定义如下：

```
#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。
```

getFirstCmdNode 接口定义如下：

```
#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 两种调用方式的范例如下：

```
## 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
```

返回结果说明

不管是 API 调用，还是直接运行 ddp，其返回的结果（非终端输出），形式都是一样的。具体说明如下：

```
#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" : ".."
        },
        ...
    }
}
```

使用最佳建议

* 在 ddp.py 所在的目录下建立 hosts 目录（用于存放机器列表文件）和 cmds 目录（用于存放命令脚本），这样将迁移到新版本的 ddp 时，只需将 hosts 与 cmds 目录拷贝过去，而保持使用 ddp 的命令不变。

* nohup 命令推荐都带上 NM 修饰符，是否成功执行可接下来使用带 ASSERT 修饰符的命令来验证。

* 当有单条命令所产生的终端输出内容相当大量时，推荐使用多线程版本，这样可避免多进程版本的数据内容传输管道被撑爆而导致 ddp 异常的问题，因为多线程版本使用共享内存方式进行传输，不存在此问题。

附录一：ddp 安装

ddp 无需安装，把 ddp 压缩包解压之后，即可直接运行目录下的 ddp.py 进行使用。

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：与操作机进行 scp 操作时，在获取操作机的当前登陆用户名时发生异常

-5002：与操作机进行 scp 操作时，获取到操作机的 IP 为空

-5003：与操作机进行 scp 操作时，在获取操作机的 IP 时发生异常

-5004：与操作机进行 scp 操作时，指定操作机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：从操作机向 SSH 机发起 scp 的推送操作时，需要密码，但未提供

-5012：从操作机向 SSH 机发起 scp 的推送操作时，密码不正确或权限被禁止

-5013：从操作机向 SSH 机发起 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：从操作机向 SSH 机发起 scp 的推送操作时，超时，有经过输入密码操作

-5015：从操作机向 SSH 机发起 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：从操作机向 SSH 机发起 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 机向操作机发起 scp 的拉取操作时，操作失败

-5019：从 SSH 机向操作机发起 scp 的拉取操作时失败，因为未能通过前期的拉取测试

-5020：从操作机向 SSH 机发起 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

-5022：从操作机往 SSH 机发起 scp 的推送操作时，命令执行完成，但命令返回码（即 echo $?）非 0

SSH 机上命令操作错误：-6XXX

-6001：获取 SSH 机上当前目录路径时不成功

-6002：获取 SSH 机上当前登录用户的 home 目录时不成功

-6003：获取 SSH 机是 64 位还是 32 位系统时不成功

-6004：SSH 机执行完一条命令后，在获取 echo $?结果时发生 EOF

-6005：SSH 机执行完一条命令后，在获取 echo $?结果时，超时

-6006：SSH 机执行完一条命令后，在获取 echo $?结果时，发生异常

-6007：在 SSH 机上执行一条命令时，发生 EOF

-6008：在 SSH 机上执行一条命令时，超时

-6009：在 SSH 机上执行一条命令时，命令执行完毕，但获取其 echo $?的值时非 0

-6010：添加用户时，在创建用户的 home 目录时出错

-6011：添加用户时，用户名不能为空

-6012：执行添加用户命令时出错

-6013：执行完添加用户命令后，在获取命令返回值（即 echo $?）时出错

-6014：执行完添加用户命令后，获取到命令返回值（即 echo $?）非 0

-6015：在修改用户密码时出错，输入两次密码后仍然要求输入密码

-6016：在修改用户密码时发生 EOF

-6017：在修改用户密码时发生超时

-6018：执行完判断用户名是否已经存在的命令后，在获取命令返回值（即 echo $?）时出错

-6019：执行判断用户名是否已经存在的命令时，发生 EOF

-6020：执行判断用户名是否已经存在的命令时，发生超时

-6021：所要增加的用户名已经在系统中存在

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

执行本地命令时的错误：-8XXX

-8001：执行本地命令失败，因返回的 echo $?值不为 0

-8002：执行本地命令时发生异常