Shell公共函数：/etc/init.d/functions详解

作者：An
原文链接：https://ld246.com/article/1389683304827
来源网站：链滴

在学习Shell的时候看到很多脚本都会有. /etc/init.d/functions 这一句，在网上查询了此数的作用，故收藏在博客中
functions这个脚本是给/etc/init.d里边的文件使用的。提供了一些基础的功能，看看里边究竟有什么。首先会设置umask，path，还有语言环境，然后会设置success,failure,warning,normal几种况下的字体颜色。下面再看看提供的重要方法：
checkpid:检查是否已存在pid，如果有一个存，返回0（通过查看/proc目录）
daemon:启动某个服务。/etc/init.d目录部分脚本的start使用这个
killproc:杀死某个进程。/etc/init.d目录部分脚本的stop使用到这个
pidfileofproc:寻某个进程的pid
pidofproc:类似上面的，只是还查找了pidof命令
status:返回一个服务的状
echo_success,echo_failure,echo_passed,echo_warning分别输出各类信息
success,failur,passed,warning分别记录日志并调用相应的方法
action:打印某个信息并执行给定的命令，它会据命令执行的结果来调用 success,failure方法
strstr:判断$1是否含有$2
confirm:显示 "Start service $1 (Y)es/(N)o/(C)ontinue? [Y]"的提示信息，并返回选择结果
详细分析：
<pre class="brush: bash; toolbar: false; auto-links: false"># -*-Shell-script-*-
#
functions This file contains functions to be used by most or all # 注释：该脚本几乎被 /etc/ini.d/ 下的所有脚本所调用，因为它包含了大量的
shell scripts in the /etc/init.d directory. # 的基础函数。同时也被 /etc/rc.d/rc.sysinit，例如 success、action、failure 等函数
#
TEXTDOMAIN=initscripts # 设置 TEXTDOMAIN 变量
#某些系统使用 LC_MESSAGES shell 变量所指定的消息类型. 其他一些系统根据
shell 变量 TEXTDOMAIN 的值来创建消息类型的名称, 可能还会加上后缀'.mo'. 如果
你使用 TEXTDOMAIN 变量, 你可能需要设置变量 TEXTDOMAINDIR 指向消息类型文件所
在的位置. 还有某些系统以这种形式两个变量都使用: TEXTDOMAINDIR/LC_MESSAGES
/Lc_Messages/TEXTDOMAIN.mo.
##
<h2 id="Make-sure-umask-is-sane---确保-root-用户的-umask-是正确的-022--也就是-rwxr-xr-x">Make sure umask is sane # 确保 root 用户的 umask 是正确的 022 （也就是 rwxr-xr-x）</h2>
umask 022
<h2 id="Set-up-a-default-search-path----设置默认的-PATH-变量">Set up a default search pat. # 设置默认的 PATH 变量</h2>
PATH="/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin" # 默认为 /sbin:/usr/sbin:/bin:/usr/bin/usr/X11R6/bin
export PATH # 导出为环境变量
<h2 id="Get-a-sane-screen-width---设置正确的屏幕宽度">Get a sane screen width # 设置正确屏幕宽度</h2>
[-z "${COLUMNS:-}"] && COLUMNS=80 # 如果 COLUMNS 变量的值为空，则置为 80 （列）
[-z "${CONSOLETYPE:-}"] && CONSOLETYPE="<code>/sbin/consoletype</coe>" # 如果 CONSOLETYPE 为空则设置 CONSOLETYPE 为 /sbin/consoletype 命令返回的值
<h2 id="一般是-vt-或者-pty--serial">一般是 vt 或者 pty 、serial</h2>
##
if [-f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}"] ; then # 如果存在 /etc/sysconfig/i18n 且 NOLOCALE 变量的值为空，则
. /etc/sysconfig/i18n # 执行 /etc/sysconfig/i18n 文件，取得 LANG 变量的值
if ["$CONSOLETYPE" != "pty"]; then # 如果当前 console 类型不是 pty（远程登录），而是 vt 或 serial，则
case "${LANG:-}" in # 根据 LANG 的值作出选择
ja_JP*|ko_KR*|zh_CN*|zh_TW*|bn_|bd_|pa_|hi_|ta_|gu_) # 如果 LANG 是 日文、中文简体、中文繁体、韩文等，则
export LC_MESSAGES=en_US # 把 LC_MESSAGES 设置为 en_US
export LANG # 同时导出为环境变量

;;
*)
export LANG # 如果是其他类型的语言，则直接导出 LANG
; ;
esac
else # 如果当前 consle 是 pty
[-n "$LC_MESSAGES"] && export LC_MESSAGES # 且如果 LC_MESSAGES 不为空，直接导出 LC_MESSAGES
export LANG
fi
fi
########################**********************************####################
</pre>
case语句 ： 它能够把变量的内容与多个模板进行匹配,再根据成功匹配的模板去决定应该执行哪分代码。
 使用格式：
 case 匹配母板 in
 模板1 [| 模板2] ...) 语句组 ;;
 模板3 [| 模板4] ...) 语句组 ;;
 esac
 case语句的匹配是从上往下地匹配顺序。因此，case语句编写原则是从上往下，模板从特殊到普通。在C语言里，case语句中有default模板，而在shell程序设计中可能将模板写成*，就可以完成相同的功能。
case语句的模板支持匹配
 匹配以n开头的所有情况： n*
 匹配yes的所有字母大小不同情况： [yY][eE][sS]
 但不支持{}匹配，因为模板可以使用 | 就可以达到目的。
例程:
<pre class="brush: bash; toolbar: false; auto-links: false">#!/bin/sh
echo "Please input "yes" or "no""
read var
case "$var" in
[yY][eE][sS]) echo "Your input is YES" ;;
[nN][oO]) echo "Your input is YES" ;;

) echo "Input Error!" ;;

esac
exit 0
##
##
<h2 id="下面是设置-success-failure-passed-warning-4种情况下的字体颜色的">下面是设置 success、failure、passed、warning 4 种情况下的字体颜色的</h2>
<h2 id="Read-in-our-configuration">Read in our configuration</h2>
if [-z "${BOOTUP:-}"]; then # 首先如果 BOOTUP 变量为空，则
if [-f /etc/sysconfig/init]; then # 如果存在 /etc/sysconfig/init 文件，执行 /etc/sysconfig/init 件
. /etc/sysconfig/init
else # 否则我们就手工设置
<h2 id="This-all-seem-confusing--Look-in--etc-sysconfig-init-">This all seem confusing? Look in /etc/sysconfig/init,</h2>
<h2 id="or-in--usr-doc-initscripts---sysconfig-txt">or in /usr/doc/initscripts-*/sysconfig.txt</h2>
BOOTUP=color # 第一设置 BOOTUP 变量，默认就是 color
RES_COL=60 # 第二设置设置在屏幕的第几列输出后面的 "[xxx]" ，默认是第 60 列
MOVE_TO_COL="echo -en \033[${RES_COL}G" # MOVE_TO_COL 是用于打印 "OK" 或者 "FAILED" ,或者 "PASSED" ,或者 "WARNING" 之前的部分，不含 "["
SETCOLOR_SUCCESS="echo -en \033[1;32m" # SETCOLOR_SUCCESS 设置后面的字体都为绿色
SETCOLOR_FAILURE="echo -en \033[1;31m" # SETCOLOR_FAILURE 设置后面将要输出的字体都

红色
SETCOLOR_WARNING="echo -en \033[1;33m" # SETCOLOR_WARNING 设置后面将要输出的体都为黄色
SETCOLOR_NORMAL="echo -en \033[0;39m" # SETCOLOR_NORMAL 设置后面输出的字体都为色（默认）
LOGLEVEL=1
fi
if ["$CONSOLETYPE" = "serial"]; then # 如果是通过串口登录的，则全部取消彩色输出
BOOTUP=serial
MOVE_TO_COL=
SETCOLOR_SUCCESS=
SETCOLOR_FAILURE=
SETCOLOR_WARNING=
SETCOLOR_NORMAL=
fi
fi
##
##
if ["${BOOTUP:-}" != "verbose"]; then # 如果 BOOTUP 变量的值不为 verbose ，则
INITLOG_ARGS="-q" # 把 INITLOG_ARGS 的值设置为 -q （安静模式）
else # 否则
INITLOG_ARGS= # 把 INITLOG_ARGS 的值请空
fi
##
<h2 id="Check-if--pid--could-be-plural--are-running---下面定义一个函数-checkpid----目的是查--proc-下是否存在指定的目录-例如--proc-1--">Check if $pid (could be plural) are running # 下面定义一个函数 checkpid （），目的是检查 /proc 下是否存在指定的目录（例如 /proc/1/）</h>
checkpid() { # 如果有任意一个存在，则返回 0；
local i #局部变量定义
for i in $* ; do
[-d "/proc/$i"] && return 0
done
return 1 # 如果给出的参数全部不存在对应的目录，则返回 1
}
##
<h2 id="A-function-to-start-a-program----下面定义最重要的一个函数-daemon-函数-它的作用启动某项服务--etc-init-d--下的脚本的-start-部分都会用到它">A function to start a program. # 面定义最重要的一个函数，daemon 函数，它的作用是启动某项服务。/etc/init.d/ 下的脚本的 start 部分都会用到它</h2>
daemon() {
<h2 id="Test-syntax-">Test syntax.</h2>
local gotbase= force=
local base= user= nice= bg= pid=
nicelevel=0
while ["$1" != "${1##[-+]}"]; do # daemon 函数本身可以指定多个选项，例如 --check <value> ， --check=<value> ，
case $1 in
'') echo $"$0: Usage: daemon [+/-nicelevel] {program}" # 也可以指定 nice 值
return 1;;
--check)
base=$2

gotbase="yes"
shift 2
;;
--check=?*)
base=${1#--check=}
gotbase="yes"
shift
;;
--user) # 也可以指定要以什么用户身份运行（--user <usr> , --user=<usr>)
user=$2
shift 2
;;
--user=?*)
user=${1#--user=}
shift
;;
--force)
force="force" # --force 表示强制运行
shift
;;
[-+][0-9]*)
nice="nice -n $1" # 如果 daemon 的第一个参数是数字，则认为是 nice 值
shift
;;
*) echo $"$0: Usage: daemon [+/-nicelevel] {program}"
return 1;;
esac
done
<h2 id="Save-basename----basename-就是从服务器的二进制程序的-full-path-中取出最后的部分">Save basename. # basename 就是从服务器的二进制程序的 full path 中取出最后的部分</h2>
[-z "gotbase"] && base{1##*/}
<h2 id="See-if-it-s-already-running--Look-only-at-the-pid-file----检查该服务是否已经在运行-过-daemon-函数只查看-pid-文件而已">See if it's already running. Look only at th pid file. # 检查该服务是否已经在运行。不过 daemon 函数只查看 pid 文件而已</h2>
if [-f /var/run/${base}.pid]; then # 如果 /var/run 下存在该服务的 pid 文件，则
local line p
read line < /var/run/${base}.pid # 从该 pid 文件每次读取一行，送给变量 line 。注意 pid 文件能有多行，且不一定都是数字
for p in $line ; do # 对于 line 变量的每个 word 进行检查
[-z "{p//[0-9]/}" -a -d "/proc/p"] && pid="pid p" # 如果 p 全部是数字，且在 /proc/$p/ 目录，则认为该数字是一个 pid ，把它加入到 pid 变量
done # 到最后 pid 变量的值可能是有多个由空格分隔的数字组成
fi
[-n "{pid:-}" -a -z "{force:-}] && return # 如果 pid 变量最终为空，则 force 变量为空（不强制启动），则返回
<h2 id="make-sure-it-doesn-t-core-dump-anywhere-unless-requested---下面对该服务使用资源作一些设置">make sure it doesn't core dump anywhere unless requested # 下面对该服务用的资源作一些设置</h2>
ulimit -S -c ${DAEMON_COREFILE_LIMIT:-0} >/dev/null 2>&1 # ulimit 是控制该 shell 启动的进程能够使用的资源，-S 是 soft control 的意思，-c 是指最大的 core
<h2 id="dump-文件大小-如果-DEAMON-COREFILE-LIMIT-为空-则默认为-0">dump 文件大小，果 DEAMON_COREFILE_LIMIT 为空，则默认为 0</h2>

<h2 id="if-they-set-NICELEVEL-in--etc-sysconfig-foo--honor-it---如果存在--etc-sysconfi-foo文件-且其中有-NICELEVEL-变量则用它代替-daemon-后面的那个-nice-值">if they set NICELEVEL n /etc/sysconfig/foo, honor it # 如果存在 /etc/sysconfi/foo 文件，且其中有 NICELEVEL 变量则它代替 daemon 后面的那个 nice 值</h2>
[-n "NICELEVEL"] && nic="nice -n NICELEVEL" # 注意，这里的 nice 赋值是用 nice -n <value> 格式，因为 nice 本身可以启动命令，用这个格式较方便
<h2 id="Echo-daemon---如果-BOOTUP-的值为-verbose--则打印一个服务名">Echo daemon # 果 BOOTUP 的值为 verbose ， 则打印一个服务名</h2>
["{BOOTUP:-}" = "verbose" -a -z "LSB"] && echo -n " $base"
<h2 id="And-start-it-up----下面是开始启动它了">And start it up. # 下面是开始启动它了</h2>
if [-z "$user"]; then # 如果 user 变量为空，则默认使用 root 启动它
nice initlog INITLOG_ARGS -c "*" # 执行 nice -n <nice_value> initlog -q -c "</san>"
else # 如果指定了用户，则
nice initlog INITLOG_ARGS -c "runuser -s /bin/bash -user -c \""" # 执行 nice n <nice_value> initlog -q -c "runuser -s /bin/bash - <user> -c "$*"
fi
["?" -eq 0] && success </spa>"base startup" || failure "$base startup" # 如果上面的命令成功，则显示一个绿色的 [OK] ，否则显示 [FAILURE]
}
##
<h2 id="A-function-to-stop-a-program----下面定义另外一个很重要的函数-killproc---etc-init-d-下面的脚本的-stop-部分都会用到它">A function to stop a program. # 下面定义另外一个很重要函数 killproc ， /etc/init.d/ 下面的脚本的 stop 部分都会用到它</h2>
killproc() {
RC=0 # RC 是最终返回的值，初始化为 0
<h2 id="Test-syntax--">Test syntax.</h2>
if ["$#" -eq 0]; then # killproc 函数的语法格式是 killproc <service> [<signal>] 例如 killproc sm-client 9
echo $"Usage: killproc {program} [signal]"
return 1
fi
notset=0 # noset 是用于检查用户是否指定了 kill 要使用的信号
<h2 id="check-for-second-arg-to-be-kill-level">check for second arg to be kill level</h2>
if [-n "$2"]; then # 如果 $2 不为空，则表示用户有设定信号，则
killlevel=$2 # 把 $2 的值赋予 killlevel 变量
else # 否则
notset=1 # notset 变量的值为 1，同时 killlevel 为 '-9' （KILL 信号）
killlevel="-9"
fi
<h2 id="补充--注意-并不是说用户没有指定信号地停止某项服务时-就会立即用-kill--9-这样的方式制杀死-而是先用-TERM-信号-然后再用-KILL">补充： 注意，并不是说用户没有指定信号地停止某项务时，就会立即用 kill -9 这样的方式强制杀死，而是先用 TERM 信号，然后再用 KILL</h2>
<h2 id="Save-basename-">Save basename.</h2>
base=${1##*/} # basename 就是得出服务的名称
<h2 id="Find-pid-">Find pid.</h2>
pid= # 把 pid 变量的值清空。注意，不是指 pid 变量的值等于下面脚本的执行结果，要看清楚<r>

if [-f /var/run/${base}.pid]; then # 下面和上面的 daemon 函数一样找出 pid
local line p
read line < /var/run/${base}.pid
for p in $line ; do
[-z "{p//[0-9]/}" -a -d "/proc/p"
] && pid="pid p"
done
fi
if [-z "$pid"]; then # 不过和 daemon 不同的是，一旦 pid 为空不会直接 return 而是尝试用 pid
令再次查找
pid=<code>pidof -o $$ -o $PPID -o %PPID -x $1 || \ # -o 是用于忽略某个 pid ， -o $$ 是忽略
前 shell 的 pid、-o $PPID 是忽略 shell 的 pid pidof -o $$ -o $PPID -o %PPID -x $base</code>
-o %PPID 是忽略 pidof 命令的父进程，要查询的进程是 $1 (fullpath) 或者 $base
fi
<h2 id="Kill-it-">Kill it.</h2>
if [-n "${pid:-}"] ; then # 如果 pid 的值最终不为空，则
["BOOTUP" = "verbose" -a -
 "LSB"] && echo -n "$base " # 且 BOOTUP 的值为 verbose ，且
SB 变量不为空，则打印一个服务名
if ["$notset" -eq "1"] ; then # 如果 notset 变量不为 1，表示用户没有指定信号，则
if checkpid pid 2>&1; then # 调用 checkpid
pid 检查是否在 /proc/ 下存在进程目录，如果有
<h2 id="TERM-first--then-KILL-if-not-dead---先尝试用-TERM-信息-不行再用-KILL-信号">TERM
first, then KILL if not dead # 先尝试用 TERM 信息，不行再用 KILL 信号</h2>
kill -TERM pid >/dev/null 2>&1
执行 kill -TERM pid
usleep 100000 # usleep 和 sleep 一样，不过单位是百万分之 1 秒。这里休眠 1 秒
if checkpid pid && sleep 1 &&
amp;amp; # 如果 checkpid pid 还是查到有 /proc/<pid>/ 目录存在，则表示还没有
死，继续等待 1 秒
checkpid $pid && sleep 3 && # 如果 1 秒后用 checkpid 检查还是有，则再
待 3 秒；
checkpid $pid ; then # 如果还是没有杀死，则用 KILL 信号
kill -KILL $pid >/dev/null 2>&1 # 执行 kill -KILL 杀死它
usleep 100000 # 等待 1 秒种
fi
fi
checkpid $pid # 再次检查 pid 目录
RC=$? # 并把结果返回给 RC ，这就算是 killproc 的最后状态了
["RC" -eq 0] && failure </spa
>"base shutdown" || success "$base shut
own" # 如果 RC 的值为 0，则表示 kill -9 没有杀死了进程，则调用 failure 函数，否则调用 success
br>
RC=((! RC))
<h2 id="use-specified-level-only---上面都是在没有指定信号的情况的-下面是用户指定了信号的-
如-restart-或者-reload-部分">use specified level only # 上面都是在没有指定信号的情况的，下面
用户指定了信号的。例如 restart）或者 reload）部分</h2>
else # 这个 else 是针对 if ["$notset" -eq "1"] 的
if checkpid $pid; then # 如果检查到进程存在，则
kill killlevel pid >/dev/null 2>&1 # 执行 kill
命令，但使用指定的信号 $killlevel
RC=$? # 并把状态值返回给变量 RC
["RC" -eq 0] && success </sp
n>"base killlevel" || failure <span class="language-m

th">"base $killlevel" # 如果 RC 为 0 则表示成功，调用 success；否则调用 fai
ure 函数
fi
fi
else # 这个 else 是针对 if [-n "${pid:-}"] 的，也就是说没有 pid 文件，pidof 命令也没有找到 pid
则
failure "base shutdown" # 调用 failure 函
，表示停止服务失败
RC=1 # 同时 RC 的值为 1
fi
<h2 id="Remove-pid-file-if-any----根据具体情况可能需要删除-pid-文件">Remove pid file if any
根据具体情况可能需要删除 pid 文件</h2>
if ["$notset" = "1"]; then # 如果 notset 不为 1 ，也就是用户没有指定信号的情况
rm -f /var/run/$base.pid # 自动删除 /var/run 下的 pid 文件
fi
return $RC # 并把 RC 作为 exit status 返回
}
<h2 id="补充--自所以删除-pid-文件只针对-notset-为1-的情况-是因为--HUP-信号-重读配置--并
杀死进程-所以不能删除它的-pid-文件">补充： 自所以删除 pid 文件只针对 notset 为 1 的情况，是
为 -HUP 信号（重读配置），并不杀死进程，所以不能删除它的 pid 文件</h2>
<h2 id="例如下面--">例如下面： </h2>
<h2 id="ps--ef--grep-xinetd">ps -ef |grep xinetd</h2>
root 2635 1 0 12:25 ? 00:00:00 xinetd -stayalive -pidfile /var/run/xinetd.pid
<h2 id="--xinetd-reload">./xinetd reload</h2>
Reloading configuration: [OK]
<h2 id="ps--ef--grep-xinetd-">ps -ef |grep xinetd</h2>
root 2635 1 0 12:25 ? 00:00:00 xinetd -stayalive -pidfile /var/run/xinetd.pid
root 3927 3412 0 16:43 pts/0 00:00:00 grep xinetd
#可以看到 pid 在 reload 后并没有变
##
################################
<h2 id="A-function-to-find-the-pid-of-a-program--Looks-only-at-the-pidfile">A function to f
nd the pid of a program. Looks only at the pidfile</h2>
<h2 id="下面的-pidfileofproc-函数和-checkpid-类似-但不执行-pidof-命令-只查询-pid-文件">下
的 pidfileofproc 函数和 checkpid 类似，但不执行 pidof 命令，只查询 pid 文件</h2>
pidfileofproc() {
local base=${1##*/}
<h2 id="Test-syntax---">Test syntax.</h2>
if ["$#" = 0] ; then
echo $"Usage: pidfileofproc {program}"
return 1
fi
<h2 id="First-try--var-run---pid-files">First try "/var/run/*.pid" files</h2>
if [-f /var/run/$base.pid] ; then
local line p pid=
read line < /var/run/$base.pid
for p in $line ; do
[-z "{p//[0-9]/}" -a -d /proc/p] &&
mp; pid="pid p"
done
if [-n "$pid"]; then
echo $pid
return 0
fi

fi
}
##
<h2 id="A-function-to-find-the-pid-of-a-program----下面的-pidofproc-函数和上面的-pidfileofproc-函数类似-但多了一步-pidof-命令">A function to find the pid of a program. # 下面的 pidofproc 函数和上面的 pidfileofproc 函数类似，但多了一步 pidof 命令</h2>
pidofproc() {
base=${1##*/}
<h2 id="Test-syntax----">Test syntax.</h2>
if ["$#" = 0]; then
echo $"Usage: pidofproc {program}"
return 1
fi
<h2 id="First-try--var-run---pid-files-">First try "/var/run/*.pid" files</h2>
if [-f /var/run/$base.pid]; then
local line p pid=
read line < /var/run/$base.pid
for p in $line ; do
[-z "{p//[0-9]/}" -a -d /proc/p] && pid="pid p"
done
if [-n "$pid"]; then
echo $pid
return 0
fi
fi
pidof -o
 -o $PPID -o %PPID -x $1 || \
pidof -o
-o PPID -o %PPID -x base
}
##
status() { # 注释： 下面的 status 函数是判断服务的状态，总共有 4 种
local base=${1##*/}
local pid
<h2 id="Test-syntax-----">Test syntax.</h2>
if ["$#" = 0] ; then
echo $"Usage: status {program}"
return 1
fi
<h2 id="First-try-pidof---同样是查找-pid-先-直接使用-pidof-命令">First try "pidof" # 同样是查找 pid 先。直接使用 pidof 命令</h2>
pid=<code>pidof -o $$ -o $PPID -o %PPID -x $1 || \ pidof -o $$ -o $PPID -o %PPID -x $base}</code>
if [-n "$pid"]; then # 如果 pid 变量的值不为空，则表示找到进程，
echo "{base} (pid $pid) is running..." # 则印 "xxx (pid nnn) is running " ,
return 0 # 并返回 0
fi
<h2 id="Next-try--var-run---pid-files---如果-pidof-命令没有找到-则尝试从-pid-文件找">Next try "/var/run/*.pid" files # 如果 pidof 命令没有找到，则尝试从 pid 文件找</h2>
if [-f /var/run/${base}.pid] ; then

read pid < /var/run/${base}.pid
if [-n "$pid"]; then # 如果 pidof 命令找不到，但从 pid 文件找到了 pid ，则
echo "{base} dead but pid file exists" # 印 "xxx dead but pid file exists"，
return 1 # 并返回 1
fi
fi
<h2 id="base">See if /var/lock/subsys/$ exists # 如果 pidof 命令和 pid 文件都没有找到 pid，</h2>
if [-f /var/lock/subsys/${base}]; then # 如果在 /var/lock/subsys 下存在对应的文件，则
echo "{base} dead but subsys locked" # 印 “xxxx dead but subsys locked”，
return 2 # 并返回 2
fi
echo "{base} is stopped" # 如果 pidof 命、pidf 文件都没有找到 pid，且没有别锁，则打印 “xxx is stopped”
return 3 # 并返回 3
}
##
<h2 id="注释--下面的-echo-xxx-函数就是真正在屏幕上打印---ok------passed-----failure----warning---的部分了">注释： 下面的 echo_xxx 函数就是真正在屏幕上打印 [ok] 、[PASSED]、[FAILURE]、[WARNING] 的部分了</h2>
echo_success() { # 下面是 echo_success 部分
["BOOTUP" = "color"] && MOVE_TO_COL # 首先是打印 “[” 之前的空格
echo -n "[" # 然后打印 "["
["BOOTUP" = "color"] && SETCOLOR_SUCCESS # 设置字体为红色
echo -n $"OK" # 打印 OK
["BOOTUP" = "color"] && SETCOLOR_NORMAL # 返回字体为白色
echo -n "]" # 打印 "]"
echo -ne "\r" # 换行。
return 0 # 返回 0，其他一律返回 1
echo_failure() {
["BOOTUP" = "color"] && MOVE_TO_COL
echo -n "["
["BOOTUP" = "color"] && SETCOLOR_FAILURE
echo -n $"FAILED"
["BOOTUP" = "color"] && SETCOLOR_NORMAL
echo -n "]"
echo -ne "\r"
return 1
}
echo_passed() {
["BOOTUP" = "color"] && MOVE_TO_COL
echo -n "["
["BOOTUP" = "color"] && SETCOLOR_WARNING

echo -n $"PASSED"
["BOOTUP" = "color"] &a
p;amp;& SETCOLOR_NORMAL
echo -n "]"
echo -ne "\r"
return 1
}
echo_warning() {
["BOOTUP" = "color"] &a
p;amp;& MOVE_TO_COL
echo -n "["
["BOOTUP" = "color"] &a
p;amp;& SETCOLOR_WARNING
echo -n $"WARNING"
["BOOTUP" = "color"] &a
p;amp;& SETCOLOR_NORMAL
echo -n "]"
echo -ne "\r"
return 1
}
##
############################
<h2 id="Inform-the-graphical-boot-of-our-current-state">Inform the graphical boot of our c
rrent state</h2>
update_boot_stage() {
if ["$GRAPHICAL" = "yes" -a -x /usr/bin/rhgb-client]; then
/usr/bin/rhgb-client --update="$1"
fi
return 0
}
##
##########################
<h2 id="Log-that-something-succeeded">Log that something succeeded</h2>
success() { # success 函数除了打印 [xxx] 之外，还会使用 initlog 记录信息
if [-z "${IN_INITLOG:-}"]; then
initlog $INITLOG_ARGS -n $0 -s "$1" -e 1 # -n 是 --name 的意思，-s 是 --string ， -e
是 --event ， 1 表示完全成功
else
<h2 id="silly-hack-to-avoid-EPIPE-killing-rc-sysinit">silly hack to avoid EPIPE killing rc.sysinit
/h2>
trap "" SIGPIPE
echo "$INITLOG_ARGS -n $0 -s "$1\" -e 1" >&21
trap - SIGPIPE
fi
["BOOTUP" != "verbose" -a
z "LSB"] && echo_success
return 0
}
<h2 id="Log-that-something-failed">Log that something failed</h2>
failure() {
rc=$?
if [-z "${IN_INITLOG:-}"]; then
initlog $INITLOG_ARGS -n $0 -s "$1" -e 2 # failure 的话 --event 是 2 是失败
else

trap "" SIGPIPE
echo "$INITLOG_ARGS -n $0 -s "$1\" -e 2" >&21
trap - SIGPIPE
fi
["BOOTUP" != "verbose" -a
z "LSB"] && echo_failure
[-x /usr/bin/rhgb-client] && /usr/bin/rhgb-client --details=yes
return $rc
}
<h2 id="Log-that-something-passed--but-may-have-had-errors--Useful-for-fsck">Log that s
mething passed, but may have had errors. Useful for fsck</h2>
passed() {
rc=$?
if [-z "${IN_INITLOG:-}"]; then
initlog $INITLOG_ARGS -n $0 -s "$1" -e 1 # passed 的话 --event 还是 1
else
trap "" SIGPIPE
echo "$INITLOG_ARGS -n $0 -s "$1\" -e 1" >&21
trap - SIGPIPE
fi
["BOOTUP" != "verbose" -a
z "LSB"] && echo_passed
return $rc
}
<h2 id="Log-a-warning">Log a warning</h2>
warning() {
rc=$?
if [-z "${IN_INITLOG:-}"]; then
initlog $INITLOG_ARGS -n $0 -s "$1" -e 1 # warning 的话 --event 也是 1
else
trap "" SIGPIPE
echo "$INITLOG_ARGS -n $0 -s "$1\" -e 1" >&21
trap - SIGPIPE
fi
["BOOTUP" != "verbose" -a
z "LSB"] && echo_warning
return $rc
}
##
##############################
<h2 id="Run-some-action--Log-its-output----action-函数是另外一个最重要的函数-它的作用是
印某个提示信息并执行给定命令">Run some action. Log its output. # action 函数是另外一个最重
的函数，它的作用是打印某个提示信息并执行给定命令</h2>
tion() {
STRING=$1
echo -n "$STRING "
if ["${RHGB_STARTED}" != "" -a -w /etc/rhgb/temp/rhgb-console]; then
echo -n "$STRING " > /etc/rhgb/temp/rhgb-console
fi
shift
initlog INITLOG_ARGS -c "*" &&
success "STRING" || failure <span class="l
nguage-math">"STRING"
rc=$?

echo
if ["${RHGB_STARTED}" != "" -a -w /etc/rhgb/temp/rhgb-console]; then
if ["$rc" = "0"]; then
echo_success > /etc/rhgb/temp/rhgb-console
else
echo_failed > /etc/rhgb/temp/rhgb-console
[-x /usr/bin/rhgb-client] && /usr/bin/rhgb-client --details=yes
fi
echo
fi
return $rc
}
##
<h2 id="returns-OK-if--1-contains--2---strstr-函数是判断--1-字符串是否含有--2-字符串-是则返0-否则返回1">returns OK if $1 contains $2 # strstr 函数是判断 $1 字符串是否含有 $2 字符串，则返回 0，否则返回 1</h2>
() {
["${1#*$2*}" = "$1"] && return 1
return 0
}
##
<h2 id="Confirm-whether-we-really-want-to-run-this-service---confirm-函数是用于交互式的动服务">Confirm whether we really want to run this service # confirm 函数是用于交互式的启动务</h2>
nfirm() {
[-x /usr/bin/rhgb-client] && /usr/bin/rhgb-client --details=yes
while : ; do
echo -n $"Start service $1 (Y)es/(N)o/(C)ontinue? [Y] " # 会打印一个提示信息
read answer
if strstr "yY" "answe" || ["$answer" = ""] ; then # 如果 answer 变量是 y 或者 Y 则
return 0 # 返回 0（但未真正启动）
elif strstr "cC" "anser" ; then # 如果 answer 是 c 或者 C，则
rm -f /var/run/confirm # 删除 /var/run/confirm 文件
[-x /usr/bin/rhgb-client] && /usr/bin/rhgb-client --details=no
return 2 # 返回 2
elif strstr "nN" "anser" ; then # 如果 answer 是 n 或者 N，则
return 1 # 直接返回 1
fi
done
} </pre>

原文：http://www.cnblogs.com/image-eye/archive/2011/10/26/2220405.html