

BSD rc.d脚本编程

摘要

初学者可能会问，为什么要用正式的文件，基于 BSD 的 rc.d 框架，写一些任意的 rc.d 脚本。本文中，我采用了一些实用性不断叠加的典型案例，来展示综合各个案例的 rc.d 特性，并探讨其中的工作原理。以上的希望大家一研究有效的 rc.d 程序提供了一些参考点。

1

1. 介绍	1
2. 任务描述	2
3. 虚拟的脚本	2
4. 可配置的虚拟脚本	4
5. 并行停止守护进程	6
6. 并停止高优先级进程	7
7. 链接脚本到 rc.d 框架	10
8. 给 rc.d 脚本更多的灵活性	12
9. 总结	14

1. 介

史上 BSD 曾有一个唯一的启动脚本， /etc/rc。 该脚本在系统启动时被 [init\(8\)](#) 程序所调用，并执行所有多用途操作所需求的任务：挂载文件系统，设置网络，守护进程，等等。 在不同系统中的任务清理也并不相同；管理需要根据需求自定义任务清理。在一些特殊的情况下，不得不去修改 /etc/rc 文件，一些真正的黑客不疲。

『一腳本』方法的真正問題是它沒有提供從 /etc/rc 的各個事件的控制。拿一個例子來說，/etc/rc 不能重新啟動某个獨有的服務。系統管理不得不手動退出服務，並掉它，等待它真正退出後，再通過 /etc/rc 得到服務的命令，最進入全部命令來再次啟動服務。如果重新啟動的服務包括不止一個服務或需要更多動作的，任務將變得更加困難以及容易出錯。而言之，『一腳本』在『我』的目的上是不成功的：『系統管理』的生活更轻松。

再后来，为了将最重要的一些子系统独立出来，便将部分的内容从 /etc/rc 分离出来了。最广为人知的例子就是用来启动网的 /etc/netstart 文件。它可以从用户模式启动，但由于它的部分代码需要和一些与网完全无关的操作交互，所以它并没有完美地融合到自身的进程中。那便是为何 /etc/netstart 被演变成 /etc/rc.network 的原因了。后者不再是一个普通的脚本；它包括了大量的，由 /etc/rc 在不同的系统中用的凌乱的 `sh(1)` 函数。然而，当任得多变化以及久更改，“模块化”方法得比曾的整体 /etc/rc 更慢事。

由于没有一个干练和易于使用的框架，`rc` 脚本不得不全力更改以满足速口中基于 BSD 的操作系统的需求。它逐渐变得明朗并添加许多必要的功能最终形成一个具有保密性和扩展性的 `rc` 系统。BSD `rc.d` 就由此而生了。Luke Mewburn 和 NetBSD 社区是公认的 `rc.d` 之父。再之后它被引入到了 FreeBSD 中。它的名字引用了系统

唯一的服务器脚本的位置，也就是 /etc/rc.d 下面的那些脚本。之后我将学到更多的 rc.d 系统的文件并看看各个脚本是如何被调用的。

BSD rc.d 背后的基本理念是 良好的模块化和代码重用性。良好的模块化意味着一个基本 "服务" 就象系统守护程序或原始任务那样，通常属于它自己的可调用的服务的 sh(1) 脚本，来停止服务，重启服务，或服务的状态。具体操作由脚本的命令行参数所决定。/etc/rc 脚本仍然掌管着系统的服务，但是在它本身是使用 start 参数来一个个地调用那些小的脚本。这便于用 stop 来一行中的所有的脚本很好地执行停止任务，或是被 /etc/rc.shutdown 脚本所完成的。看，是多么好地体现了 Unix 的哲学：有一小的有用的工具，一个工具尽可能好地完成自己的任务。代码重用 意味着所有的通用操作由 /etc/rc.subr 中的一些 sh(1) 函数所完成。在一个典型的脚本只需要几行的 sh(1) 代码。最，rcorder(8) 成为了 rc.d 框架中重要的一部分，它用来帮助 /etc/rc 处理小脚本之间的依赖关系并有次序地执行它们。它同时也帮助 /etc/rc.shutdown 做类似的事情，因正确的次序是相对于正确的次序的。

BSD rc.d 的确在 [Luke Mewburn 的原文](#) 中有，以及 rc.d 文件也被充分地放在各自的机手册中。然而，它可能没能清晰展示一个 rc.d 新手，如何将无数的命令和片段结合起来具体的任务建一个好风格的脚本。因此本文将试着以不同的方式来描述 rc.d。它将展示在某些典型情况中使用一些特性，并描述了为何如此。注意并不是一篇 how-to 文章，我的目的不是给出成的配方，而是在展示一些进入 rc.d 的道路。本文也不是相机手册的替代品。本文只得同参考机手册以取更完整正确的文章。

理解本文需要一些先决条件。首先，你需要熟悉 sh(1) 脚本语言以掌握 rc.d，还有，你需要知道系统是如何执行命令的启动和停止任务，这些在 [rc\(8\)](#) 中都有说明。

本文关注的是 rc.d 的 FreeBSD 分支。不过，它可能对 NetBSD 的用户也同样有用，因为 BSD rc.d 的两个分支不只是共享了同样的命令，保留了脚本编写者都可能的相似点。

2. 任务描述

在开始打开 \$EDITOR (编辑器) 之前进行小小的思考不是坏事。为了一个系统服务写一个 "听" 的 rc.d 脚本，我首先能回答以下：

- 服务是必须性的还是可选性的？
- 脚本将哪个程序服务，如一个守护进程，是执行更重要的操作？
- 我的服务依赖哪些服务？反过来哪些服务依赖我的服务？

从下面的例子中我将看到，什么知道些什么的答案是很重要的。

3. 虚拟的脚本

下面的脚本是用来在每次系统启动时输出一个信息：

```

#!/bin/sh ①

. /etc/rc.subr ②

name="dummy" ③
start_cmd="${name}_start" ④
stop_cmd=: ⑤

dummy_start() ⑥
{
    echo "Nothing started."
}

load_rc_config $name ⑦
run_rc_command "$1" ⑧

```

需要注意的是：

一个解口性的脚本口以一行魔幻的 "shebang" 行口。口行指定了脚本的解析程序。由于 shebang 行的作用，假如再有可口行位的口置，脚本就能象一个二口制程序一口被精口地口用口行。（口参考 [chmod\(1\)](#)。）例如，一个系口管理口可以从命令行手口行我口的脚本：

```
# /etc/rc.d/dummy start
```

为了使 rc.d 框架正口地管理脚本，它的脚本需要用 [sh\(1\)](#) 口言口写。如果口的某个服口或 port 套件使用了二口制控制程序或是用其它口言口写的例程，口将其口件安装到 /usr/sbin（相口于系口）或 /usr/local/sbin（相口于ports），然后从合口的 rc.d 目口的某个 [sh\(1\)](#) 脚本口用它。



如果口想知道口什口 rc.d 脚本必口用 [sh\(1\)](#) 口言口写的口，先看下 /etc/rc 是如何依口 [run_rc_script](#) 口用它口，然后再去学口 /etc/rc.subr 下 [run_rc_script](#) 的相口口。

在 /etc/rc.subr 下，有口多定口的 [sh\(1\)](#) 函数可供口个 rc.d 脚本使用。口些函数在 [rc.subr\(8\)](#) 中都有口明。尽管管理口上可以完全不使用 [rc.subr\(8\)](#) 来口写一个 rc.d 脚本，但它的函数已口明了它真的很方便，并且能使任口更加的口。所以所有人在口写 rc.d 脚本口都会求助于 [rc.subr\(8\)](#) 也不足口奇了。当然我口也不例外。

一个 rc.d 脚本在其口用 [rc.subr\(8\)](#) 函数之前必口先 "source"/etc/rc.subr (用 ``.`"将其包含口去)，而使 [sh\(1\)](#) 程序有机会来口悉那些函数。首口格是在脚本的最口始 source /etc/rc.subr 文件。



某些有用的与口网有口的函数由口一个被包含口来的文件提供，/etc/network.subr 文件。

口口制的口量 `name` 指定我口脚本的名字。口是 [rc.subr\(8\)](#) 所口的。也就是，口个 rc.d 脚本在口用 [rc.subr\(8\)](#) 的函数之前必口置 `name` 口量。

口在是口候来口我的脚本一次性口一个独一无二的名字了。口在口写口个脚本的口我口将在口多地方用到它。在口始之前，我口来口脚本文件也取个相同的名字。

当前的 rc.d 脚本风格是把`$`放在双引号中来定量。往往只是个风格，可能并不是`$`。可以在只是`$`的并不包括 `sh(1)` 元字符的句中放心地省略掉引号，而在某些情况下将需要使用引号以防止 `sh(1)` 对任何的量的解释。程序是可以巧妙地由风格例悉其法以及使用的。

rc.subr(8) 背后主要的意思是 rc.d 脚本提供管理程序，或者方法，来`rc.subr(8)` 用。特别是，`start`, `stop`, 以及其它的 rc.d 脚本参数都是被管理的。方法是存在一个以 `argument_cmd` 形式命名的量中的 `sh(1)` 表式，`argument` 表示脚本命令行中所特指的参数。我后面将看到 `rc.subr(8)` 是如何为准参数提供默认方法的。

除了 rc.d 中的代码更加统一，常指的是在任何场合的地方都使用 `${name}` 形式。这样一来，可以轻松地将一些代码从一个脚本拷贝到另一个中使用。

我同意 `rc.subr(8)` 准参数提供了默认的方法。因此，如果希望它什么都不做的，我必须使用无操作的 `sh(1)` 表式来改写准的方法。

比`rc.conf(5)`的方法主体可以用函数来`rc.conf(5)`。在能保留函数名有意的情况下，是个很不错的想法。

强烈推荐我脚本中所定义的所有函数名都添加类似 `${name}` 的前缀，以便它永远不会和 `rc.subr(8)` 或其它公用包含文件中的函数冲突。

是在要求 `rc.subr(8)` 入 `rc.conf(5)` 量。尽管我这个脚本中使用的量并没有被其它地方使用，但由于 `rc.subr(8)` 自身所控制着的 `rc.conf(5)` 量存在的原因，仍然推荐脚本去装 `rc.conf(5)`。

通常我是 rc.d 脚本的最后一个命令。它用 `rc.subr(8)` 体系使用我脚本所提供的量和方法来执行请求操作。

4. 可配置的虚脚本

在我来我的虚脚本加一些控制参数。正如所知，rc.d 脚本是由 `rc.conf(5)` 所控制的。幸的是，`rc.subr(8)` 藏了所有`rc.conf(5)`的东西。下面两个脚本使用 `rc.conf(5)` 通过 `rc.subr(8)` 来看它是否在第一个地方被用，并取一条信息在`rc.conf(5)`。事实上两个任务是相互独立的。一方面，rc.d 脚本要能支持和禁用它的服务。另一方面，rc.d 脚本必能具备配置信息量。我将通过下面同一脚本来演示方面的内容：

```

#!/bin/sh

. /etc/rc.subr

name=dummy
rcvar=dummy_enable ①

start_cmd="${name}_start"
stop_cmd=""

load_rc_config $name ②
eval "$${rcvar}=\$${${rcvar}:-'NO'}" ③
dummy_msg=${dummy_msg:-"Nothing started."} ④

dummy_start()
{
    echo "$dummy_msg" ⑤
}

run_rc_command "$1"

```

在哪个例中改了什么？

变量 `rcvar` 指定了 ON/OFF 变量的名字。

变量在 `load_rc_config` 在任何 `rc.conf(5)` 变量被读之前就在脚本中被先用。



rc.d 脚本，一切 `sh(1)` 会把函数延申到其被调用才执行其中的表达式进行求值。因此尽可能地在 `run_rc_command` 之前调用 `load_rc_config`，以及仍然从方法函数输出到 `run_rc_command` 的 `rc.conf(5)` 变量并不是一个好主意。这是因为方法函数将在 `load_rc_config` 之后，被调用的 `run_rc_command` 使用。

如果自身设置了 `rcvar`，但指示变量却未被设置，那么 `run_rc_command` 将输出一个警告。如果的 rc.d 脚本是基本系统所用的，当在 `/etc.defaults/rc.conf` 中变量添加一个默认的设置并将其注在 `rc.conf(5)` 中。否则的脚本向变量提供一个默认值。例中演示了一个可移植接近于后者情况的案例。



可以通常将变量设置为 ON 来使 `rc.subr(8)` 有效，使用 `one` 或 `force` 脚本的参数加前缀，如 `onestart` 或 `forcestop`，会忽略其当前的设置。一切 `force` 在我下面要提到的情况下有外的危后，那就是当用 `one` 改写了 ON/OFF 变量。例如，假定 `dummy_enable` 是 OFF 的，而下面的命令将忽略系统设置而执行 `start` 方法：

```
# /etc/rc.d/dummy onestart
```

显示的信息不再是硬设在脚本中的了。它是由一个命名的 `dummy_msg` 的 `rc.conf(5)` 变量所指定的。就是 `rc.conf(5)` 变量如何来控制 rc.d 脚本的一个小例子。



我的脚本所独占使用的所有 `rc.conf(5)` 变量名，都必须具有同名的前缀：\${name}。例如：`dummy_mode`, `dummy_state_file`, 等等。

当可以内部使用一个短的名字，如 `msg`，我的脚本所引用的全部的共用名添加唯一的前缀 \${name}，能避免我与 `rc.subr(8)` 命名空间冲突的可能。

只要一个 `rc.conf(5)` 变量与其内部等同是相同的，我就能使用一个更加兼容的表达式来置默值：



```
: ${dummy_msg:="Nothing started."}
```

尽管目前的风格是使用了更直接的形式。

通常，基本系列的 rc.d 脚本不需要它自己的 `rc.conf(5)` 变量提供默值，因为默值是在 /etc/default/rc.conf 置定了。但一方面，ports 所用的 rc.d 脚本提供如上例所示的默值。

这里我使用 `dummy_msg` 来直接地控制我的脚本，即，一个变量信息。

5. 启动并停止守护进程

我早先说 `rc.subr(8)` 是能提供默值方法的。当然，有些默值方法并不是太通用的。它们都是用于大多数情况下启动和停止一个守护进程。我来假定在需要写一个叫做 `mumbled` 的守护进程写一个 rc.d 脚本，在这里：

```
#!/bin/sh

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}" ①

load_rc_config $name
run_rc_command "$1"
```

感到很困惑，不是吗？我来写下我自己的小脚本。只需要注意下面的一些新知识点：

一个 `command` 变量对于 `rc.subr(8)` 来说是有意的。当它被置，`rc.subr(8)` 将根据提供守护进程的情形而生效。特别是，将一些参数提供默值的方法：`start`, `stop`, `restart`, `poll`, 以及 `status`。

守护进程将会由命令中的 `$command` 配合由 `$mumbled_flags` 所指定的命令行来。因此，默值的 `start` 方法来自，所有的输入数据在我脚本变量集合中都可用。与 `start` 不同的是，其他方法可能需要与进程相关的额外信息。一个例子，`stop` 必须知道进程的 PID 号来进程。在目前的情况下，`rc.subr(8)` 将扫描全部进程的清空，一个名字等同于 `$procname` 的进程。后者是一个 `rc.subr(8)` 有意的变量，并且默值它的跟

`command` 一`o`。而言之，当我`o` `command` `o`置`o`后，`procname` `o`上也`o`置了同`o`的`o`。我`o`的脚本来`o`死守`o`程并`o`它是否正在第一个位置`o`行。

某些程序上是可行的脚本。 系统脚本的解释器以脚本名命令行参数的形式来执行脚本。 然后被映射到进程列表中，会使 `rc.subr(8)` 迷惑。因此，当 `$command` 是一个脚本的，另外再设置 `command_interpreter` 来让 `rc.subr(8)` 知道进程的名字。



对于一个 rc.d 脚本而言，有一个可选的 rc.conf(5) 变量 command 指示其优先级。它的名字是下面的形式：\${name}_program， name 是我在之前讲过的必选性变量。如，在一个案例中它命名 emumbleled_program。其中是 rc.subr(8) 分配 \${name}_program 来改写 command 的。

当然，即使 `command` 未被置，`sh(1)` 也将允许从 `rc.conf(5)` 或自身来置 `${name}_program`。在那情况下， `${name}_program` 的特定属性失了，并且它成了一个能供的脚本用于其自身目的的普通量。然而，独使用 `${name}_program` 是并不是我所寄望的，因同使用它和 `command` 已成了 `rc.d` 脚本程的一个用的定。

对于默~~默~~方法的更~~多~~的信息，~~参~~考 [rc.subr\(8\)](#)。

6. 程序并停止高守并行

我来之前的“骨架”脚本加点“血肉”，并让它更□更富有特性□。默□的方法已能□我□做很好的工作了，但是我□可能会需要它□一些方面的□整。□在我□将学□如何□整默□方法来符合我□的需要。

```
#!/bin/sh

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}"
command_args="mock arguments > /dev/null 2>&1" ①

pidfile="/var/run/${name}.pid" ②

required_files="/etc/${name}.conf /usr/shared/misc/${name}.rules" ③

sig_reload="USR1" ④

start_precmd="${name}_prestart" ⑤
stop_postcmd="echo Bye-bye" ⑥

extra_commands="reload plugh xyzzy" ⑦

plugh_cmd="mumbled_plugh" ⑧
xyzzy_cmd="echo 'Nothing happens.'"

mumbled_prestart()
```

```

{
    if checkyesno mumbled_smart; then ⑨
        rc_flags="-o smart ${rc_flags}" ⑩
    fi
    case "$mumbled_mode" in
    foo)
        rc_flags="-frotz ${rc_flags}"
        ;;
    bar)
        rc_flags="-baz ${rc_flags}"
        ;;
    *)
        warn "Invalid value for mumbled_mode" ⑪
        return 1 ⑫
        ;;
    esac
    run_rc_command xyzzy ⑬
    return 0
}

mumbled_plugh() ⑭
{
    echo 'A hollow voice says "plugh".'
}

load_rc_config $name
run_rc_command "$1"

```

附加到 `$command` 的参数在 `command_args` 中进行。它在 `$mumbled_flags` 之后将被添加到命令行。其一行便是此后最的命令行`eval` 计算，输入和输出以及重定向都可以在 `command_args` 中指定。



永远不要 在 `command_args` 包含破折号，似 `-X` 或 `--foo` 的。`command_args` 的内容将出现在最命令行的末尾，因此它可能是接在 `${name}_flags` 中所列出的参数后面；但大多的命令将不能普通参数后的破折号。更好的附加 `$command` 的方式是添加它到 `${name}_flags` 的起始。另一种方法是像后文所示的那来修改 `rc_flags`。

一个得体的守程会建一个 `pidfile` 程文件，使其程能更容易更可地被到。如果置了 `pidfile` 量，告 `rc.subr(8)` 里能到供其默方法所使用的 `pidfile` 程文件。



事上，`rc.subr(8)` 在一个守程前会使用 `pidfile` 程文件来看它是否已在行。使用了 `faststart` 参数可以跳一个。

如果守程只有在定的文件存在的情况下才可以行，那就将它列到 `required_files` 中，而 `rc.subr(8)` 将在守程之前那些文件是否存在。有相的分用来目和境的 `required_dirs` 和 `required_vars` 可供使用。它都在 `rc.subr(8)` 中有明。



来自 `rc.subr(8)` 的默方法，通使用 `forcestart` 作脚本的参数，可以制性地跳先需要的。

□ 我们可以在守护进程有异常的时候，自定义发送守护进程的信号。特别是，`sig_reload` 指定了使守护进程重新装入其配置的信号；默认情况也就是 `SIGHUP` 信号。一个信号是发送守护进程以停止进程；默认情况下是 `SIGTERM` 信号，但是是可以通过设置 `sig_stop` 来执行当更改的。



信号名称应当以不包含 `SIG` 前缀的形式指定为 `rc.subr(8)`，就如例中所示的那样。FreeBSD 版本的 `kill(1)` 程序能发出 `SIG` 前缀，而不是其它系统版本的就不一定了。

□ 在默认的方法前面或后面执行附加任务是很容易的。对于我脚本所支持的两条命令参数而言，我们可以定义 `argument_precmd` 和 `argument_postcmd` 来完成。这些 `sh(1)` 命令分在它们各自的方法前后被调用，当然，从它们各自的名字就能看出来。



如果我需要的，用自定义的 `argument_cmd` 改写默认的方法，并不妨碍我仍然使用 `argument_precmd` 和 `argument_postcmd`。特别是，前者便于自定义的方法，以及执行自身命令之前所遇到更复杂的条件。于是，将 `argument_precmd` 和 `argument_cmd` 一起使用，使我们合理地将命令从工作中独立了出来。

□ 忘了可以将任意的有效的 `sh(1)` 表达式插入到方法和自定义的 pre- 与 post-commands 命令中。在大部分情况下，用函数使任何任务有好的风格，但千万不要风格限制了幕后到底是怎么回事的思考。

□ 如果我愿意一些自定义参数，这些参数也可被用作我脚本的命令，我们需要在 `extra_commands` 中将它们列出并提供方法以处理它们。



`reload` 是个特殊的命令。一方面，它有一个在 `rc.subr(8)` 中设置的方法。另一方面，`reload` 命令默认是不被提供的。理由是并非所有的守护进程都使用同一种重启方法，并且有些守护进程根本没有任何东西可重启的。所以而易之，我们需要去都提供了些的内建功能。我们可以通过 `extra_commands` 来做。

我从 `reload` 的默认方法得到了什么？守护进程常常在收到一个信号后重新加载它的配置 - 一般来说，也就是 `SIGHUP` 信号。因此 `rc.subr(8)` 会发送一个信号给守护进程来重启它。信号一般为 `SIGHUP`，但是如果必要的，可以通过 `sig_reload` 量来自定义它。

□ 我的脚本提供了两个非标准的命令，`plugh` 和 `xyzzy`。我看到它们在 `extra_commands` 中被列出来了，并且是在时候它们提供方法了。`xyzzy` 的方法是内部的而 `plugh` 的是以 `mumbled_plugh` 形式完成的函数。

非标准命令在启动或停止的时候不被调用。通常它们是为了系统管理的方便。它们能被其它的子系统所使用，例如，`devd(8)`，前提是 `devd.conf(5)` 中已指定了。

全部可用命令的列表，当脚本不加参数地调用，在 `rc.subr(8)` 打印出的使用方法中能找到。例如，就是供学习的脚本用法的内容：

```
# /etc/rc.d/mumbled
Usage: /etc/rc.d/mumbled [fast|force|one]
(start|stop|restart|rcvar|reload|plugh|xyzzy|status|poll)
```

□ 如果脚本需要的，它可以调用自己的标准或非标准的命令。可能看起来有点像函数的调用，但我知道，命令和 shell 函数并非一直都是同一种东西。一个例子，`xyzzy` 在里不是以函数来调用的。另外，只有被有序调用的 pre-command 前置命令和 post-command 后置命令。所以脚本执行自己命令的合

一种方式就是利用 `rc.subr(8)`, 就像例中展示的那。

`rc.subr(8)` 提供了一个方便的函数叫做 `checkyesno`。 它以一个变量名作为参数并返回一个零的退出，当且仅当变量置为 `YES`, 或 `TRUE`, 或 `ON`, 或 `1`, 区分大小写；否则返回一个非零的退出。 在第二种情况下, 函数将变量置为 `NO`, `FALSE`, `OFF`, 或 `0`, 区分大小写；如果变量包含的内容的它打印一条警告信息, 例如, `foo`。

一切 `sh(1)` 而言零意味着真而非零意味着假。

`checkyesno` 函数使用一个变量名。不要大含将变量的值修改它；否则它不会如预期那样工作。

下面是 `checkyesno` 的合理使用：

```
! if checkyesno mumbled_enable; then
    foo
fi
```

相反地, 以下面的方式用 `checkyesno` 是不会工作的—至少是不会如预期的那：

```
if checkyesno "${mumbled_enable}"; then
    foo
fi
```

我可以通过修改 `$start_precmd` 中的 `rc_flags` 来影响到 `$command` 的。

某些情况下我可能需要输出一条重要的信息, 那么的 `syslog` 可以很好地记录日志。你可以使用下列 `rc.subr(8)` 函数来轻松完成：`debug`, `info`, `warn`, 以及 `err`。后者以指定的代码退出脚本。

方法的退出和它的 pre-commands 命令不只是默认被忽略掉。如果 argument_precmd 返回了一个非零退出, 主方法将不会被执行。依次地是, argument_postcmd 将不会被用, 除非主方法返回的是一个零的退出。

然而, 当一个参数使用 `force` 前的任何时候, 如 `forcestart`, `rc.subr(8)` 会听从命令行指示而忽略那些退出最后仍然用所有的命令。

7. 接脚本到 rc.d 框架

当写好了一个脚本, 它需要被整合到 `rc.d` 中去。一个重要的就是安装脚本到 `/etc/rc.d` (基本系统而言) 或 `/usr/local/etc/rc.d` (ports而言) 中去。在 `bsd.prog.mk` 和 `bsd.port.mk` 中都提供了方便的接口, 通常不必担心恰当的所有限制和模式。系统脚本当然是通过可以在 `src/etc/rc.d` 到的 Makefile 安装的。Port 脚本可以像 [Porter's Handbook](#) 中描述那样通过使用 `USE_RC_SUBR` 来被安装。

不, 我们先考虑到我的脚本在系统中的位置。我的脚本所处理的服务可能依赖于其它的服务。一个例子, 没有网口接口和路由的用处, 一个网守进程是不起作用的。即使一个服务看似什么都不需要, 在基本文件系统挂载之前也很。

之前我曾提到 [rcorder\(8\)](#)。在任何时候来密切地注下它了。本地， [rcorder\(8\)](#) 管理一个文件，本地的内容，并从文件集合打印一个文件列表的依序到 [stdout](#) 准出。点是用于保持文件内部的依信息，而一个文件只能明自己的依。一个文件可以指定如下信息：

- 它提供的 "条件" 的名字（意味着我服的名字）；
- 它需求的 "条件" 的名字；
- 本地先运行的文件的 "条件" 的名字；
- 能用于从全部文件集合中本地一个子集的外本地字（ [rcorder\(8\)](#) 可通过而被指定来包括或省去由特殊本地字所列出的文件。）

并不奇怪的是， [rcorder\(8\)](#) 只能管理接近 [sh\(1\)](#) 语法的文本文件。[rcorder\(8\)](#) 所解的特殊行看起来似 [sh\(1\)](#) 的注。本地特殊文本行的语法相当格外地化了其理。本地 [rcorder\(8\)](#) 以取更本地的信息。

除使用 [rcorder\(8\)](#) 的特殊行以外，脚本可以持将其依的其它服务性。当其它服是可的，并因系本地管理本地地在 [rc.conf\(5\)](#) 中禁用掉服而使其不能自行本地，会需要本地一点。

将些本地在心，我来考下本地合了依信息本地的守程脚本：

```
#!/bin/sh

# PROVIDE: mumbled oldmumble ①
# REQUIRE: DAEMON cleanvar frotz②
# BEFORE: LOGIN ③
# KEYWORD: nojail shutdown ④

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}"
start_precmd="${name}_prestart"

mumbled_prestart()
{
    if ! checkyesno frotz_enable && \
        ! /etc/rc.d/frotz forcestatus 1>/dev/null 2>&1; then
        force_depend frotz || return 1 ⑤
    fi
    return 0
}

load_rc_config $name
run_rc_command "$1"
```

跟前面一，做如下本地分析：

本地行声明了我脚本所提供的 "条件" 的名字。在其它脚本可以用那些名字来明我脚本的依。

通常脚本指定一个唯一的已提供的条件。然而，
从列出的那些条件中指定，例如，为了兼容性的目的。

并没有什么妨碍我



在其它情况，主要的名称，或者唯一的， **PROVIDE:** 条件`==> ${name}` 相同。

因此我的脚本指示了其依赖于它的脚本所提供的 "条件"。根据这些行的信息，脚本示 **rorder(8)** 以将其放在一个或多个提供 DAEMON 和 cleanvar 的脚本后面，但在提供 LOGIN 的脚本前面。

BEST: 一行不可以在其它脚本不完整的依存关系列表中使用。结合使用 **BEST:** 的情况是当其它脚本不关心我的脚本，但是我的脚本如果在一个之前执行的能更好地执行任务。一个典型的例子是网口接口和防火墙：虽然接口不依赖防火墙来完成自己的工作，但是网络安全将因一切网口流量之前后的防火墙而受益。



除了条件相同的那个独服，脚本使用元条件和它的 "占位符" 来保证某个操作在其它之前被执行。些是由 UPPERCASE 大写名字所表示的。它的列表和用法可以在 **rc(8)** 中找到。

一切将一个服务名称放入 **REQUIRE:** 行不能保证该服务会在我脚本中的时候执行。所需求的服务可能会丢失或在 **rc.conf(5)** 中被禁掉了。当然，**rorder(8)** 是无法追踪这些的，并且 **rc(8)** 也不会去追踪。所以，脚本中的应用程序当能对付任何所需求的服务的不可用情况。某些情况下，我可以使用 [下面](#) 所示的方式来帮助脚本。

如我从上述文字所起的，**rorder(8)** 字可以用来或省略某些脚本。即任何 **rorder(8)** 用字可以通过指定 `-k` 和 `-s` 来分别指定 "保留清 (keep list)" 和 "跳过清 (skip list)"。从全部文件到按依赖关系排列的清，**rorder(8)** 将只是挑出保留清（除非是空的）中那些字的以及从跳过清中挑出不字的文件。

在 FreeBSD 中，**rorder(8)** 被 `/etc/rc` 和 `/etc/rc.shutdown` 所使用。两个脚本定义了 FreeBSD 中 `rc.d` 字以及它的意义的准确列表如下：

以 **force_depend** 起始的行被用于更谨慎的情况。通常，用于修正相互的 `rc.d` 脚本分配置文件会更加妥。

如果仍不能完成不含 **force_depend** 的脚本，例提供了一个如何有条件地用它的用法。在例中，我的 **mumbled** 守护程序需求一个以高方式的程序，**frotz**。但 **frotz** 也是可的；而且 **rorder(8)** 些信息是一无所知的。幸的是，我的脚本已到全部的 **rc.conf(5)** 量。如果 **frotz_enable** 真，我希望的最好是依赖 `rc.d` 已给了 **frotz**。否我抑制 **frotz** 的状态。最，如果 **frotz** 依的服没有到或执行的，我将抑制其执行。而 **force_depend** 将输出一条警告信息，因为它只在到配置信息失的情况下被用。

8. 给 `rc.d` 脚本更多的活性

当行或停止的用，`rc.d` 脚本作用于其所的整个子系。例如，`/etc/rc.d/netif` 或停止 **rc.conf(5)** 中所描述的全部网口接口。个任都唯一地听从一个如 `start` 或 `stop` 的独命令参数的指示。在和停止之的，`rc.d` 脚本帮助管理控制行中的系，并其在需要的时候它将生更多的活性和精性。个例子，管理可能想在 **rc.conf(5)** 中添加一个新网口接口的配置信息，然后在不妨碍其它已存在接口的情况下将其。在下次管理可能需要一个独的网口接口。在魔幻的命令行中，的 `rc.d` 脚本用一个外的参数，网口接口名即可。

幸的是，[rc.subr\(8\)](#) 允许多（取决于系统限制）的参数到脚本的方法。由于这个原因，脚本自身的修改可以是微乎其微。

[rc.subr\(8\)](#) 如何将附加的命令行参数到直接取？并非是无所不用其的。首先，[sh\(1\)](#) 函数没有将到用户的定位参数，而 [rc.subr\(8\)](#) 只是些函数的容器。其次，[rc.d](#) 指令的一个好的风格是由主函数来决定将哪些参数到它的方法。

所以 [rc.subr\(8\)](#) 提供了如下的方法：[run_rc_command](#) 将所有参数但将第一个参数逐字到各自的方法。首先，移出以方法自身名字的参数：`start`, `stop`, 等等。会被 [run_rc_command](#) 移出，命令行中原本 \$2 的内容将作 \$1 来提供方法，等等。

为了明点，我来修改原来的虚拟脚本，它的信息将取决于所提供的附加参数。从里出：

```
#!/bin/sh

. /etc/rc.subr

name="dummy"
start_cmd="${name}_start"
stop_cmd=:"
kiss_cmd="${name}_kiss"
extra_commands="kiss"

dummy_start()
{
    if [ $# -gt 0 ]; then ①
        echo "Greeting message: $*"
    else
        echo "Nothing started."
    fi
}

dummy_kiss()
{
    echo -n "A ghost gives you a kiss"
    if [ $# -gt 0 ]; then ②
        echo -n " and whispers: $*"
    fi
    case "$*" in
    *[.!?])
        echo
        ;;
    *)
        echo .
        ;;
    esac
}

load_rc_config $name
run_rc_command "$@" ③
```

能注意到脚本里发生了那些变化吗？

□ 所有的在 `start` 之后的参数可以被当作各自方法的定位参数一并被用。 我们可以根据我们的经验和想法来以任何方式使用它们。 在当前的例子中，我们只是以下行中字符串的形式参数 `echo(1)` 程序 - 注意 `$*` 是有双引号的。 这是脚本如何被用的：

```
# /etc/rc.d/dummy start
Nothing started.

# /etc/rc.d/dummy start Hello world!
Greeting message: Hello world!
```

□ 同可用于我们脚本提供的任何方法，并不限于标准的方法。 我们已经添加了一个自定义的叫做 `kiss` 的方法，并且它附加参数来的决不在于 `start`。 例如：

```
# /etc/rc.d/dummy kiss
A ghost gives you a kiss.

# /etc/rc.d/dummy kiss Once I was Etaoin Shrdlu...
A ghost gives you a kiss and whispers: Once I was Etaoin Shrdlu...
```

□ 如果我们只是所有附加参数任意的方法，我们只需要在脚本的最后一行我们用 `run_rc_command` 的地方，用 `"$@"` 代替 `"$1"` 即可。



一个 `sh(1)` 程序是可以理解 `$*` 和 `$@` 的微妙区别只是指定全部定位参数的不同方法。于此更深入的探讨，可以参考一个很好的 `sh(1)` 脚本编程手册。在完全理解这些表达式的意义之前不要使用它们，因为用它们将脚本引入缺陷和不安全的弊端。



□ 在 `run_rc_command` 可能有个缺陷，它将影响保持参数之中的原本界限。也就是说，如果有嵌入空白的参数可能不会被正确处理。缺陷是由于 `$*` 的作用。

9. 一

[Luke Mewburn 的原始文章](#) 中描述了 `rc.d` 的基本概要，并概述了其方案的原理。文章提供了深入了解整个 `rc.d` 框架以及其所在的现代 BSD 操作系统的内容。

在 `rc(8)`, `rc.subr(8)`, 也有 `rcorder(8)` 的手册中，`rc.d` 做了非常棒的工作。在写脚本时，如果不去学习和参考这些手册的话，是无法完全发挥出 `rc.d` 的能量的。

工作中主要的主要来源就是一行的系中的 `/etc/rc.d` 目录。它的内容可靠性非常好，因大部分的枯燥的内容都深藏在 `rc.subr(8)` 中了。切 `/etc/rc.d` 的脚本也不是神仙写出来的，所以它可能也存在着代码缺陷以及低级的方案。但可以在一起来改改它了！