

BSD rc.d脚本编程

摘要

初学者可能会问，何以通过正式的文章，基于 BSD 的 rc.d 框架，编写一些任意性的 rc.d 脚本。本文中，我采用了一些性质不断增加的典型案例，来展示符合各个案例的 rc.d 特性，并探讨其中的工作原理。期望大家能对研究有效的 rc.d 应用程序提供一些参考点。

目录

1. 简介.....	1
2. 任务描述.....	2
3. 虚拟脚本.....	2
4. 可配置的虚拟脚本.....	4
5. 并行停止守护进程.....	6
6. 并行停止高守护进程.....	7
7. 连接脚本到 rc.d 框架.....	10
8. 给予 rc.d 脚本更多的活性.....	12
9. 附一.....	14

1. 简介

历史上 BSD 曾有一个唯一的脚本，`/etc/rc`。脚本在系统启动的时候被 `init(8)` 程序所引用，并执行所有多用户操作所需求的各项任务：并行挂载文件系统，配置网络，守护进程，等等。在这个系统中所有的任务清单也并不相同；管理需要根据需求自定义的任务清单。在一些特殊的情况下，不得不去修改 `/etc/rc` 文件，一些真正的黑客对此不疲。

单一脚本方法的真正问题是它没有提供对从 `/etc/rc` 脚本的整个组件的控制。拿一个例子来说，`/etc/rc` 不能重新启动某个单独的守护进程。系统管理不得不手动启动守护进程，并杀掉它，等待它真正退出后，再通过 `/etc/rc` 得到守护进程的 ID，最后输入全部命令来再次启动守护进程。如果重新启动的服务包括不止一个守护进程或需要更多操作的，任务将变得更加困难以及容易出错。而言之，单一脚本在实现我们的目的上是不成功的：系统管理的生活更轻松。

再后来，为了将最重要的一些子系统独立出来，便要将部分的内容从 `/etc/rc` 分离出来了。最广为人知的例子就是用来启动网络的 `/etc/netstart` 文件。它容许从用户模式启动网络，但由于它的部分代码需要和一些与网络完全无关的操作交互，所以它并没有完美地合并到自身的程序中。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；它包括了大的，由 `/etc/rc` 在不同的系统启动中使用的凌乱的 `sh(1)` 函数。然而，当任务变得多变化以及久经更改，“模块化”方法总比曾有的整体 `/etc/rc` 更慢更事。

由于没有一个干净和易于使用的框架，脚本不得不全力更改以满足快速开发中基于 BSD 的操作系统的请求。它逐渐变得明朗并需要多必要的脚本最终成为一个具有私密性和扩展性的 rc 系统。BSD rc.d 就诞生了。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

独立的服务脚本的位置，也就是 `/etc/rc.d` 下面的那些脚本。之后我将学到更多的 `rc.d` 系统的组件并看看每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每个基本“服务”就象系统守护进程或原始任务那样，通常属于它的可管理服务脚本的 `sh(1)` 脚本，来停止服务，重启服务，管理服务状态。具体工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启动，但在它启动是使用 `start` 参数来一个个调用那些小的脚本。它便于用 `stop` 来调用行中的同类的脚本很好地运行停止任务，它是被 `/etc/rc.shutdown` 脚本所完成的。看，它是多么好地体现了 Unix 的哲学：有一组小的调用的工具，每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。它在一个典型的脚本只需要两几行的 `sh(1)` 代码。最后，`rcorder(8)` 成为了 `rc.d` 框架中重要的一部分，它用来帮助 `/etc/rc` 管理小脚本之间的依赖关系并有次序地运行它们。它同时帮助 `/etc/rc.shutdown` 做类似的事情，因此正它的启动次序是相等于它的次序的。

BSD `rc.d` 的内容在 [Luke Mewburn 的原文](#) 中有，以及 `rc.d` 组件也被充分地描述在各自的 [手册](#) 中。然而，它可能没能清晰展示一个 `rc.d` 新手，如何将无数的碎片运行起来具体的任务建立一个好风格的脚本。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如何使用些特性，并描述了何如此。注意并不是一篇 `how-to` 文档，我的目的不是给出成的配方，而是在展示一些调用的输入 `rc.d` 的调用的路径。本文也不是相关手册的替代品。阅读本文得如同参考手册以获取更完整正确的文档。

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`，有，你需要知道系统是如何运行调用的启动和停止任务，有些在 `rc(8)` 中都有说明。

本文关注的是 `rc.d` 的 FreeBSD 分支。不，它可能 NetBSD 的内容也同时有用，因为 BSD `rc.d` 的每个分支不只是共享了同的内容，它保留了脚本作者都可用的相似点。

2. 任务描述

在开始打 `$EDITOR` (编辑器) 之前进行小小的思考不是坏事。为了一个服务脚本写一个“听到的”`rc.d` 脚本，我首先可能回答以下问题：

- 服务是必需性的或是可选性的？
- 脚本将调用个程序服务，如一个守护进程，或是运行更调的操作？
- 我的服务依赖些服务？反过来些服务依赖我的服务？

从下面的例子中我将看到，什么知道些调用的答案是很重要的。

3. 虚拟的脚本

下面的脚本是用来在启动系统时输出一个信息：

```
#!/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：

一个解释性的脚本总以一行魔幻的 "shebang" 行开始。该行指定了脚本的解析程序。由于 shebang 行的作用，假如再有可执行位的设置，脚本就能象一个二进制程序一样被精确地执行。（参考 [chmod\(1\)](#)。）例如，一个系统管理可以从命令行运行我的脚本：

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



为了使 rc.d 框架正确地管理脚本，它的脚本需要用 [sh\(1\)](#) 编写。如果你的某个服务或 port 套件使用了二进制控制程序或是用其它语言写的例程，将其组件安装到 `/usr/sbin`（相对于系统）或 `/usr/local/sbin`（相对于 ports），然后从合适的 rc.d 目录的某个 [sh\(1\)](#) 脚本使用它。



如果你想知道什么 rc.d 脚本必须用 [sh\(1\)](#) 编写，先看下 `/etc/rc` 是如何依赖 [run_rc_script](#) 使用它，然后再去学 `/etc/rc.subr` 下 [run_rc_script](#) 的相关内容。

在 `/etc/rc.subr` 下，有多定量的 [sh\(1\)](#) 函数可供每个 rc.d 脚本使用。这些函数在 [rc.subr\(8\)](#) 中都有说明。尽管理论上可以完全不使用 [rc.subr\(8\)](#) 来编写一个 rc.d 脚本，但它的函数已表明了它真的很方便，并且能使任务更加的简单。所以所有人在编写 rc.d 脚本都会求助于 [rc.subr\(8\)](#) 也不足为奇了。当然我也不例外。

一个 rc.d 脚本在其调用 [rc.subr\(8\)](#) 函数之前必须先 "source" `/etc/rc.subr`（用 ``.`` 将其包含进去），而使 [sh\(1\)](#) 程序有机会来熟悉那些函数。首行格式是在脚本的最开始 `source /etc/rc.subr` 文件。



某些有用的与网络有关的函数由一个被包含进来的文件提供，`/etc/network.subr` 文件。

控制的变量 `name` 指定我的脚本的名字。它是 [rc.subr\(8\)](#) 所定义的。也就是，每个 rc.d 脚本在调用 [rc.subr\(8\)](#) 的函数之前必须设置 `name` 变量。

在是时候来我的脚本一次性给一个独一无二的名字了。在编写这个脚本的时候我将在很多地方用到它。在开始之前，我来脚本文件也取个相同的名字。



当前的 `rc.d` 脚本风格是把 `rc` 放在双引号中来测量。记住 `rc` 只是个风格，可能并不是。可以在只是 `rc` 的并不包括 `sh(1)` 元字符的语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)` 任何的量的解。程序是可以巧地由风格例悉其法以及使用的。

`rc.subr(8)` 背后主要的思想是 `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` 随着脚本命令行中所特指定的参数。我后将看到 `rc.subr(8)` 是如何准参数提供默认方法的。



除了 `rc.d` 中的代码更加一，常的是在任何合的地方都使用 `${name}` 形式。一来，可以松地将一些代码从一个脚本拷到一个中使用。

我 `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，我必使用无操作的 `sh(1)` 表式来改写准的方法。

比的方法主体可以用函数来。在能保函数名有意心的情况下，是个很不错的想法。



烈推我脚本中所定的所有函数名都添加似 `${name}` 的前，以使它永不会和 `rc.subr(8)` 或其它公用包含文件中的函数冲突。

是在求 `rc.subr(8)` 入 `rc.conf(5)` 量。尽管我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，仍然推脚本去装 `rc.conf(5)`。

通常是 `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行相的求作。

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)` 藏了所有化的西。下面个脚本使用 `rc.conf(5)` 通 `rc.subr(8)` 来看它是否在第一个地方被用，并取一条信息在示。事上这个任是相互独立的。一方面，`rc.d` 脚本要能支持和禁用它的服。一方面，`rc.d` 脚本必能具配置信息量。我将通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"
```

在这个例子中改了什？

变量 `rcvar` 指定了 ON/OFF 变量的名字。

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



`rc.d` 脚本，切 `sh(1)` 会把函数延迟到其被用才其中的表式行求算。因此尽可能地在 `run_rc_command` 之前用 `load_rc_config`，以及仍然从方法函数出到 `run_rc_command` 的 `rc.conf(5)` 量并不是一个。是因方法函数将在 `load_rc_config` 之后，被用的 `run_rc_command` 用。

如果自身置了 `rcvar`，但指示变量却未被置，那 `run_rc_command` 将出一个警告。如果的 `rc.d` 脚本是基本系所用的，当在 `/etc/defaults/rc.conf` 中变量添加一个默的置并将其注在 `rc.conf(5)` 中。否的脚本变量提供一个默置。例中演示了一个可移植接近于后者情况的案例。



可以通过将变量置 ON 来使 `rc.subr(8)` 有效，使用 `one` 或 `force` 脚本的参数加前，如 `onestart` 或 `forcestop`，会忽略其当前的置。切 `force` 在我下面要提到的情况下有外的危后果，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置而行行 `start` 方法：

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

在显示的信息不再是硬在脚本中的了。它是由一个命名 `dummy_msg` 的 `rc.conf(5)` 量所指定的。就是 `rc.conf(5)` 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简短的名字时，如 `msg`，我的脚本所引入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)` 变量与其内部等同的是相同的，我就能使用一个更加兼容的表式来置默认：



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

尽管目前的风格是使用了更短的形式。

通常，基本系统的 `rc.d` 脚本不需要它的 `rc.conf(5)` 变量提供默认，因默认是在 `/etc/defaults/rc.conf` 置了。但一方面，`ports` 所用的 `rc.d` 脚本提供如例所示的默认置。

这里我使用 `dummy_msg` 来地控制我的脚本，即，一个变量信息。

5. 并停止守护进程

我早先 `rc.subr(8)` 是能提供默认方法的。然，些默认方法并不是太通用的。它都是用于大多数情况下来和停止一个的守护进程。我来假设在需要一个叫做 `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"
```

感到很奇怪，不是吗？我来写下我这个小脚本。只需要注意下面的一些新知识点：

一个 `command` 变量于 `rc.subr(8)` 来是有意。当它被置，`rc.subr(8)` 将根据提供守护进程的情形而生效。特是，将些参数提供默认的方法：`start`，`stop`，`restart`，`poll`，以及 `status`。

守护进程将会由行中的 `$command` 配合由 `$mumbled_flags` 所指定的命令行来。因此，默认的 `start` 方法来，所有的入数据在我脚本变量集中都可用。与 `start` 不同的是，其他方法可能需要与进程相关的额外信息。个例子，`stop` 必知道进程的 PID 号来进程。在目前的情况中，`rc.subr(8)` 将描全部进程的清，一个名字等同于 `$procname` 的进程。后者是一个 `rc.subr(8)` 有意的变量，并且默认它的跟

`command` 一。 而言之，当我 `command` 置后， `procname` 上也置了同的。 我脚本来死守 程并它是否正在第一个位置行。



某些程序上是可行的脚本。 系脚本的解器以脚本名命令行参数的形式来 行脚本。 然后被映射到程列表中，会使 `rc.subr(8)` 迷惑。因此，当 `$command` 是一个脚本的， 外地置 `command_interpreter` 来 `rc.subr(8)` 知程的名字。

个 `rc.d` 脚本而言， 有一个可的 `rc.conf(5)` 量 `command` 指示其先。 它的名字是下面 的形式：`${name}_program`， `name` 是我 之前 的必性量。如，在 个案例中它命名 `emumbled_program`。 其是 `rc.subr(8)` 分配 `${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。

于默方法的更 的信息， 参考 `rc.subr(8)`。

6. 并停止高守程

我来 之前的 “骨架” 脚本加点 “血肉”，并它更 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 `$command` 的参数在 `command_args` 中运行。它在 `$mumbled_flags` 之后将被添加到命令行。其运行便是此后最后的命令行。 `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 `${name}_flags` 中所列出的参数后面；但大多数的命令将不能出普通参数后的破折号。更好的附加 `$command` 的方式是添加它到 `${name}_flags` 的起始。 一方法是像后文所示的那来修改 `rc_flags`。

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



事实上， `rc.subr(8)` 在每一个守护程前会使用 `pidfile` 程文件来看它是否已在运行。使用了 `faststart` 参数可以跳个。

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



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

我可以在守护进程有正常的时候，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，但是可以通过设置 `sig_stop` 来行当更改的。



信号名称应当以不包含 `SIG` 前的形式指定 `rc.subr(8)`，就如例中所示的那。FreeBSD 版本的 `kill(1)` 程序能输出 `SIG` 前，不其它系统版本的就不一定了。

在默认的方法前面或后面行附加任是很容易的。于我脚本所支持的条件命令参数而言，我可以定义 `argument_precmd` 和 `argument_postcmd` 来完成。些 `sh(1)` 命令分在它各自的方法前后被用，然，从它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，以及自行自身命令之前所遇到更密集的条件。于是，将 `argument_precmd` 和 `argument_cmd` 一起使用，使我合理地将从作中独立了出来。

忘了可以将任意的有效的 `sh(1)` 表式入到方法和定义的 `pre-` 与 `post-commands` 命令中。在大部分情况下，用函数使任有好的格，但千万不要格限制了其幕后到底是回事的思考。

如果我同意一些自定义参数，些参数也可被作我脚本的命令，我需要在 `extra_commands` 中将它列出并提供方法以理它。



`reload` 是个特别的命令。一方面，它有一个在 `rc.subr(8)` 中置的方法。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的重方法，并且有些守护进程根本没有任何西可重的。所以而易，我需要去都提供了些的内建功能。我可以通过 `extra_commands` 来做。

我从 `reload` 的默认方法得到了什？守护进程常常在收到一个信号后重新入它的配置 - 一般来，也就是 `SIGHUP` 信号。因此 `rc.subr(8)` 发送一个信号守护进程来重它。信号一般 `SIGHUP`，但是如果必要的可以通过 `sig_reload` 量来自定它。

我的脚本提供了个非标准的命令，`plugh` 和 `xyzy`。我看到它在 `extra_commands` 中被列出来了，并且在是候它提供方法了。`xyzy` 的方法是内而 `plugh` 的是以 `mumbled_plugh` 形式完成的函数。

非标准命令在或停止的时候不被用。通常它是了系管理的方便。它能被其它的子系所使用，例如，`devd(8)`，前提是 `devd.conf(5)` 中已指定了。

全部可用命令的列表，当脚本不加参数地用，在 `rc.subr(8)` 打印出的使用方法中能到。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|xyzy|status|poll)
```

如果脚本需要的，它可以用自己的标准或非标准的命令。可能看起来有点像函数的用，但我知道，命令和 shell 函数并非一直都是同的西。个例子，`xyzy` 在里不是以函数来的。外，有被有序用的 `pre-command` 置命令和 `post-command` 后置命令。所以脚本行自己命令的合

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`。它以一个变量名作参数并返回一个零的退出，当且仅当变量置 `YES`，或 `TRUE`，或 `ON`，或 `1`，区分大小写；否返回一个非零的退出。在第二种情况中，函数变量的置 `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，。

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。不要大含将量的 它；否它不会如期那的工作。

下面是 `checkyesno` 的合理使用：

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

相反地，以下面的方式使用 `checkyesno` 是不会工作的——至少是不会如期的那：

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

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

某情况下我可能需要出一条重要的信息，那的 `syslog` 可以很好地日志。可以使用下列 `rc.subr(8)` 函数来松完成：`debug`，`info`，`warn`，以及 `err`。后者以指定的代退出脚本。

方法的退出和它的 `pre-commands` 命令不只是默被忽略掉。如果 `argument_precmd` 返回了一个非零退出，主方法将不会被行。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` 前的时候，如 `forstart`，`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`（基本系而言）或 `/usr/local/etc/rc.d`（ports而言）中去。在 `bsd.prog.mk` 和 `bsd.port.mk` 中都此提供了方便的接口，通常不必担心当的所有限和模式。系脚本当是通可以在 `src/etc/rc.d` 到的 Makefile 安装的。Port 脚本可以像 [Porter's Handbook](#) 中描述那通使用 `USE_RC_SUBR` 来被安装。

不，我首先考到我脚本在系序中的位置。我的脚本所理的服可能依赖于其它的服。个例子，没有网接口和路由的用的行，一个网守程是不起作用的。即使一个服看似什么都不需要，在基本文件系统挂完之前也很。

之前我曾提到 `rcorder(8)`。在是时候来密切地关注它了。 本地，`rcorder(8)` 管理一个文件，它的内容，并从文件集合打印一个文件列表的依序到 `stdout` 输出。 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（意味着我服务的名字）；
- 它需求的"条件"的名字；
- 最先行的文件的"条件"的名字；
- 能用于从全部文件集合中一个子集的外 名字（`rcorder(8)` 可通过而被指定来包括或省去由特殊名字所列出的文件。）

并不奇怪的是，`rcorder(8)` 只能理解接近 `sh(1)` 法的文本文件。`rcorder(8)` 所解的特殊行看起来似 `sh(1)` 的注。 特殊文本行的法相当格地化了其理。 `rcorder(8)` 以取更的信息。

除使用 `rcorder(8)` 的特殊行以外，脚本可以持将其依的其它服制性。当其它服是可的，并因系管理地在 `rc.conf(5)` 中禁用掉服而使其不能自行，会需要一点。

将些在心，我来考下合了依信息的守程脚本：

```
#!/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，做如下分析：

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。



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

并没有什妨碍我

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

因此我的脚本指示了其依赖于的脚本所提供的 "条件"。根据些行的信息，脚本指示 `rcorder(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。

BEFORE: 一行不可以在其它脚本不完整的依赖系列表中用。合使用 **BEFORE:** 的情况是当其它脚本不我心我的脚本，但是我脚本如果在一个之前行的能更好地行行。一个典型的例是网接口和防火：然接口不依防火来完成自己的工作，但是系安全将因一切网流量之前防火而受益。



除了条件相的个独服，脚本使用元条件和它的 "占位符" 来保某个操作在其它之前被行。些是由 UPPERCASE 大写名字所表示的。它的列表和用法可以在 `rc(8)` 中到。

切将一个服名称放 **REQUIRE:** 行不能保的服会在我的脚本的候行。所需求的服可能会失或在 `rc.conf(5)` 中被禁掉了。然，`rcorder(8)` 是无法追踪些的，并且 `rc(8)` 也不会去追踪。所以，脚本的用程序当能付任何所需求的服的不可用情况。某些情况下，我可以用 下面 所的方式来助脚本。

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

在 FreeBSD 中，`rcorder(8)` 被 `/etc/rc` 和 `/etc/rc.shutdown` 所使用。个脚本定了 FreeBSD 中 `rc.d` 字以及它的意的准列表如下：

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

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` 守程需求一个以高方式程，`frotz`。但 `frotz` 也是可的；而且 `rcorder(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，我希望的最好果是依 `rc.d` 已了 `frotz`。否我制 `frotz` 的状。最，如果 `frotz` 依的服没有到或行的，我将制其行。 `force_depend` 将出一条警告信息，因它只在到配置信息失的情况下被用。

8. 予 `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` 脚本助管理控制行中的系，并在需要的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接口。在魔幻的命令行中，的 `rc.d` 脚本用一个外的参数，网接口名即可。

幸运的是，`rc.subr(8)` 允许多任意多（取决于系统限制）的参数脚本的方法。由于这个原因，脚本自身的改动可以是微乎其微。

`rc.subr(8)` 如何得到附加的命令行参数？直接取？并非是无所不用其的。首先，`sh(1)` 函数没有到用者的定位参数，而 `rc.subr(8)` 只是些函数的容器。其次，`rc.d` 指令的一个好的格式是由主函数来决定将些参数它的方法。

所以 `rc.subr(8)` 提供了如下的方法：`run_rc_command` 其所有参数但将第一个参数逐字到各自的方法。首先，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
```

能注意到脚本里产生了那些属性改变？

引入的所有在 `start` 之后的参数可以被当作各自方法的定位参数——被。我可以根据我的任何技巧 and 想法来以任何方式使用他。在当前的例子中，我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。里是脚本如何被用的：

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

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

同用于我脚本提供的任何方法，并不限于准的方法。我已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...
```

如果我只是所有附加参数任意的的方法，我只需要在脚本的最后一行我用 `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)` 脚本手册。在完全理解些表示式的意义之前不要使用它，因为它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的原本世界。也就是，有嵌入空白的参数可能不会被正确处理。缺陷是由于 `$*` 的用途。

9. 一

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

在 `rc(8)`, `rc.subr(8)`, 有 `rcorder(8)` 的手册中，`rc.d` 做了非常。在写脚本，如果不去学习和参考些手册的，是无法完全出 `rc.d` 的能量的。

工作中例的主要来源就是行的系统中的 `/etc/rc.d` 目录。它的内容可性非常好，因此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缺陷以及低劣的方案。但在可以来改它了！