BSD rc.d脚本编程实战

Yar Tikhiy <yar@FreeBSD.org>

修订：43126

2013-11-07 由 gabor.

摘要

初学者可能会发现，难以通过正式的文档， 基于 BSD 的 rc.d 框架，编写一些实际任务的 rc.d 脚本。 本文中，我们采用了一些复杂性不断增加的典型案例， 来展示适合每个案例的 rc.d 特性， 并探讨其中的工作原理。 这样的实验为大家进一步研究设计有效的 rc.d 应用程序提供了一些参考点。

目录

1. 简介

历史上 BSD 曾有过一个单一的启动脚本， /etc/rc。 该脚本在系统启动的时候被 init(8) 程序所引导，并执行所有多用户操作所需求的用户级任务： 检查并挂载文件系统，设置网络，启动守护进程，等等。 在每个系统中实际的任务清单也并不相同； 管理员需要根据需求自定义这样的任务清单。在一些特殊的情况中， 还不得不去修改 /etc/rc 文件， 一些真正的黑客乐此不疲。

单一脚本启动方法的真正问题是它没有提供对从 /etc/rc 启动的单个组件的控制。拿一个例子来说吧，/etc/rc 不能够重新启动某个单独的守护进程。 系统管理员不得不手动找出守护进程，并杀掉它， 等待它真正退出后，再通过浏览 /etc/rc 得到该守护进程的标识，最终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动作的话， 该任务将变得更加困难以及容易出错。简而言之， 单一脚本在实现我们这样的目的上是不成功的： 让系统管理员的生活更轻松。

再后来，为了将最重要的一些子系统独立出来， 便尝试将部分的内容从 /etc/rc 分离出来了。 最广为人知的例子就是用来启动联网的 /etc/netstart 文件。它容许从单用户模式访问网络， 但由于它的部分代码需要和一些与联网完全无关的动作交互， 所以它并没有完美地结合到自启动的进程中。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；它包括了庞大的，由 /etc/rc 在不同的系统启动级别中调用的凌乱的 sh(1) 函数。然而，当启动任务变得多样化以及久经更改， “类模块化” 方法变得比曾经的整体 /etc/rc 更缓慢费事。

由于没有一个干净和易于设计的框架， 启动脚本不得不全力更改以满足飞速开发中基于 BSD 的操作系统的需求。 它逐渐变得明朗并经过许多必要的步骤最终变成一个具有细密性和扩展性的 rc 系统。BSD rc.d 就这样诞生了。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。 它的名字引用为系统单独的服务脚本的位置，也就是 /etc/rc.d下面的那些脚本。 之后我们将学习到更多的 rc.d 系统的组件并看看单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启动任务那样， 通过属于它们的可启动该服务的 sh(1) 脚本，来停止服务， 重载服务，检查服务的状态。具体动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， 但现在它仅仅是使用 start 参数来一个个调用那些小的脚本。 这便于用 stop 来对运行中的同样的脚本很好地执行停止任务， 这是被 /etc/rc.shutdown 脚本所完成的。看，这是多么好地体现了 Unix 的哲学： 拥有一组小的专用的工具，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) 函数所实现。 现在一个典型的脚本只需要寥寥几行的 sh(1) 代码。最终， 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运行它们。它同样帮助 /etc/rc.shutdown 做类似的事情， 因为正确的关闭次序是相对于启动的次序的。

BSD rc.d 的设计在 Luke Mewburn 的原文 中有记录， 以及 rc.d 组件也被充分详细地记录在各自的 联机手册 中。然而， 它可能没能清晰展现给一个 rc.d 新手，如何将无数的块和片进行关联来为具体的任务创建一个好风格的脚本。 因此本文将试着以不同的方式来讲述 rc.d。 它将展示在某些典型情况中应该使用哪些特性，并阐述了为何如此。 注意这并不是一篇 how-to 文档，我们的目的不是给出现成的配方， 而是在展示一些简单的进入 rc.d 的范围的门路。 本文也不是相关联机手册的替代品。 阅读本文时记得同时参考联机手册以获取更完整正规的文档。

理解本文需要一些先决条件。首先，你需要熟悉 sh(1) 脚本编程语言以掌握 rc.d，还有，你需要知道系统是如何执行用户级的启动和停止任务，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 的开发者也同样有用，因为 BSD rc.d 的两个分支不只是共享了同样的设计， 还保留了对脚本编写者都可见的类似观点。

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。为了给一个系统服务写一个 “听话的” rc.d 脚本， 我们首先应该能回答以下问题：

- 该服务是必须性的还是可选性的？
- 脚本将为单个程序服务，如一个守护进程，还是执行更复杂的动作？
- 我们的服务依赖哪些服务？反过来哪些服务依赖我们的服务？

从下面的例子中我们将看到，为什么说知道这些问题的答案是很重要的。

3．虚拟的脚本

下面的脚本是用来在每次系统启动时发出一个信息：

```
#!/bin/sh❶

. -/etc/rc.subr❷

name="dummy"❸
start_cmd="${name}_start"❹
stop_cmd=":"❺

dummy_start()❻
{
 echo -"Nothing started."
}

load_rc_config $name❼
run_rc_command -"$1"❽
```

需要注意的是：

❶ 一个解释性的脚本应该以一行魔幻的 “shebang” 行开头。 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调用执行。（请参考 chmod(1)。） 例如，一个系统管理员可以从命令行手动运行我们的脚本：

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

注意

为了使 rc.d 框架正确地管理脚本， 它的脚本需要用 sh(1) 语言编写。 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编写的例程， 请将其组件安装到 /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) 脚本调用它。

提示

如果你想知道为什么 rc.d 脚本必须用 sh(1) 语言编写的细节，先看下 /etc/rc 是如何依靠 run_rc_script 调用它们，然后再去学习 /etc/rc.subr 下 run_rc_script 的相关实现。

❷ 在 /etc/rc.subr 下， 有许多定义过的 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，但它的函数已经证明了它真的很方便， 并且能使任务更加的简单。所以所有人在编写 rc.d 脚本时都会求助于 rc.subr(8) 也不足为奇了。当然我们也不例外。

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) 程序有机会来获悉那些函数。首选风格是在脚本的最开始 source /etc/rc.subr 文件。

注意

某些有用的与联网有关的函数由另一个被包含进来的文件提供， /etc/network.subr 文件。

❸ 强制的变量 name 指定我们脚本的名字。 这是 rc.subr(8) 所强调的。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。

现在是时候来为我们的脚本一次性选择一个独一无二的名字了。 在编写这个脚本的时我们将在许多地方用到它。在开始之前， 我们来给脚本文件也取个相同的名字。

> ### 注意
>
> 当前的 rc.d 脚本风格是把值放在双引号中来给变量赋值。 请记住这只是个风格问题，可能并不总是这样。 你可以在只是简单的并不包括 sh(1) 元字符的词句中放心地省略掉引号， 而在某些情况下你将需要使用单引号以防止 sh(1) 对任何的变量的解释。 程序员是可以灵巧地由风格惯例获悉其语法以及使用的。

❹ rc.subr(8) 背后主要的构思是 rc.d 脚本提供处理程序，或者方法，来让 rc.subr(8) 调用。特别是，start， stop，以及其它的 rc.d 脚本参数都是这样被处理的。方法是存储在一个以 *argument_cmd* 形式命名的变量中的 sh(1) 表达式，该 *argument* 对应着脚本命令行中所特别指定的参数。我们稍后将看到 rc.subr(8) 是如何为标准参数提供默认方法的。

> ### 注意
>
> 为了让 rc.d 中的代码更加统一， 常见的是在任何适合的地方都使用 ${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。

❺ 我们应谨记 rc.subr(8) 为标准参数提供了默认的方法。 因此，如果希望它什么都不做的话，我们必须使用无操作的 sh(1) 表达式来改写标准的方法。

❻ 比较复杂的方法主体可以用函数来实现。 在能够保证函数名有意义的情况下，这是个很不错的想法。

> **重要**
>
> 强烈推荐给我们脚本中所定义的所有函数名都添加类似 ${name} 这样的前缀，以使它们永远不会和 rc.subr(8) 或其它公用包含文件中的函数冲突。

❼ 这是在请求 rc.subr(8) 载入 rc.conf(5) 变量。 尽管我们这个脚本中使用的变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) 变量存在的原因，仍然推荐脚本去装载 rc.conf(5)。

❽ 通常这是 rc.d 脚本的最后一个命令。 它调用 rc.subr(8) 体系使用我们脚本所提供的变量和方法来执行相应的请求动作。

4. 可配置的虚拟脚本

现在我们来给我们的虚拟脚本增加一些控制参数吧。正如你所知， rc.d 脚本是由 rc.conf(5) 所控制的。 幸运的是，rc.subr(8) 隐藏了所有复杂化的东西。 下面这个脚本使用 rc.conf(5) 通过 rc.subr(8) 来查看它是否在第一个地方被启用，并获取一条信息在启动时显示。 事实上这两个任务是相互独立的。一方面，rc.d 脚本要能够支持启动和禁用它的服务。另一方面， rc.d 脚本必须能具备配置信息变量。 我们将通过下面同一脚本来演示这两方面的内容：

```
#!/bin/sh

. -/etc/rc.subr

name=dummy
rcvar=dummy_enable❶

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

load_rc_config $name❷
eval -"${rcvar}=\${${rcvar}:-'NO'}"❸
dummy_msg=${dummy_msg:-"Nothing started."}❹

dummy_start()
{
 echo -"$dummy_msg"❺
}

run_rc_command -"$1"
```

在这个样例中改变了什么？

❶ 变量 rcvar 指定了 ON/OFF 开关变量的名字。

❷ 现在 load_rc_config 在任何 rc.conf(5) 变量被访问之前就在脚本中被预先调用。

> **注意**
>
> 检查 rc.d 脚本时，切记 sh(1) 会把函数延迟到其被调用时才对其中的表达式进行求值运算。 因此尽可能晚地在 run_rc_command 之前调用 load_rc_config， 以及仍然访问从方法函数输出到 run_rc_command 的 rc.conf(5) 变量并不是一个错误。这是因为方法函数将在 load_rc_config 之后， 被调用的 run_rc_command 调用。

❸ 如果自身设置了 rcvar， 但指示开关变量却未被设置，那么 run_rc_command 将发出一个警告。如果你的 rc.d 脚本是为基本系统所用的，你应当在 /etc/defaults/rc.conf 中给开关变量添加一个默认的设置并将其标注在 rc.conf(5) 中。 否则的话你的脚本应该给开关变量提供一个默认设置。 范例中演示了一个可移植接近于后者情况的案例。

> **注意**
>
> 你可以通过将开关变量设置为 ON 来使 rc.subr(8) 有效， 使用 one 或 force 为脚本的参数加前缀，如 onestart 或 forcestop 这样，会忽略其当前的设置。 切记 force 在我们下面要提到的情况下有额外的危险后果，那就是当用 one 改写了 ON/OFF 开关变量。例如， 假定 dummy_enable 是 OFF 的，而下面的命令将忽略系统设置而强行运行 start方法：
>
> ```
> # /etc/rc.d/dummy onestart
> ```

❹ 现在启动时显示的信息不再是硬编码在脚本中的了。 它是由一个命名为 dummy_msg 的 rc.conf(5) 变量所指定的。这就是 rc.conf(5) 变量如何来控制 rc.d 脚本的一个小例子。

重要

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

注意

当可以内部使用一个简短的名字时，如 msg 这样，为我们的脚本所引进的全部的共用名添加唯一的前缀 ${name}，能够避免我们与 rc.subr(8) 命名空间冲突的可能。

只要一个 rc.conf(5) 变量与其内部等同值是相同的， 我们就能够使用一个更加兼容的表达式来设置默认值：

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

尽管目前的风格是使用了更详细的形式。

通常，基本系统的 rc.d 脚本不需要为它们的 rc.conf(5) 变量提供默认值， 因为默认值应该是在 /etc/defaults/rc.conf 设置过了。但另一方面，为 ports 所用的 rc.d 脚本应提供如范例所示的默认设置。

❺ 这里我们使用 dummy_msg 来实际地控制我们的脚本，即，发一个变量信息。

5. 启动并停止简单守护进程

我们早先说过 rc.subr(8) 是能够提供默认方法的。 显然，这些默认方法并不是太通用的。 它们都是适用于大多数情况下来启动和停止一个简单的守护进况。 我们来假设现在需要为一个叫做 mumbled 的守护进程编写一个 rc.d脚本， 在这里：

```
#!/bin/sh

. -/etc/rc.subr

name=mumbled
```

```
rcvar=mumbled_enable

command="/usr/sbin/${name}"❶

load_rc_config $name
run_rc_command -"$1"
```

感到很简单吧，不是么？我们来检查下我们这个小脚本。 只需要注意下面的这些新知识点：

❶ 这个 command 变量对于 rc.subr(8) 来说是有意义的。当它被设置时，rc.subr(8) 将根据提供传统守护进程的情形而生效。 特别是，将为这些参数提供默认的方法： start，stop， restart，poll， 以及 status。

该守护进程将会由运行中的 $command 配合由 $mumbled_flags 所指定的命令行标帜来启动。 因此，对默认的 start 方法来说， 所有的输入数据在我们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启动相关的额外信息。举个例子， stop 必须知道进程的 PID 号来终结进程。在目前的情况中，rc.subr(8) 将扫描全部进程的清单， 查找一个名字等同于 $procname 的进程。 后者是另一个对 rc.subr(8) 有意义的变量， 并且默认它的值跟 command 一样。 换而言之，当我们给 command 设置值后，procname 实际上也设置了同样的值。 这启动我们的脚本来杀死守护进程并检查它是否正在第一个位置运行。

注意

某些程序实际上是可执行的脚本。 系统启动脚本的解释器以传递脚本名为命令行参数的形式来运行脚本。 然后被映射到进程列表中，这会使 rc.subr(8) 迷惑。因此，当 $command 是一个脚本的时，你应该额外地设置 command_interpreter 来让 rc.subr(8) 知晓进程的实际名字。

对每个 rc.d 脚本而言， 有一个可选的 rc.conf(5) 变量给 command 指示其优先级。 它的名字是下面这样的形式：${name}_program， name 是我们 之前 讨论过的必须性变量。如，在这个案例中它应该命名为 emumbled_program。这其实是 rc.subr(8) 分配 ${name}_program 来改写 command 的。

当然，即使 command 未被设置， sh(1) 也将允许你从 rc.conf(5) 或自身来设置 ${name}_program。在那种情况下， ${name}_program 的特定属性丢失了， 并且它成为了一个能供你的脚本用于其自身目的的普通变量。 然

而，单独使用 ${name}_program 是并不是我们所寄望的，因为同时使用它和 command 已成为了 rc.d 脚本编程的一个惯用的约定。

关于默认方法的更详细的信息，请参考 rc.subr(8)。

6. 启动并停止高级守护进程

我们来给之前的 “骨架” 脚本加点 “血肉”，并让它更复杂更富有特性吧。 默认的方法已能够为我们做很好的工作了， 但是我们可能会需要它们一些方面的调整。 现在我们将学习如何调整默认方法来符合我们的需要。

```
#!/bin/sh

. -/etc/rc.subr

name=mumbled
rcvar=mumbled_enable

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

pidfile="/var/run/${name}.pid"❷

required_files="/etc/${name}.conf -/usr/share/misc/${name}.rules"❸

sig_reload="USR1"❹

start_precmd="${name}_prestart"❺
stop_postcmd="echo Bye-bye"❻

extra_commands="reload plugh xyzzy"❼

plugh_cmd="mumbled_plugh"❽
xyzzy_cmd="echo -'Nothing happens.'"

mumbled_prestart()
{
 if checkyesno mumbled_smart; then❾
  rc_flags="-o smart ${rc_flags}"❿
 fi
 case -"$mumbled_mode" in
 foo)
  rc_flags="-frotz ${rc_flags}"
  ;;
 bar)
```

```
  rc_flags="-baz ${rc_flags}"
  ;;
 *)
  warn -"Invalid value for mumbled_mode"⓫
  return 1⓬
  ;;
 esac
 run_rc_command xyzzy⓭
 return 0
}

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

load_rc_config $name
run_rc_command -"$1"
```

❶ 附加给 $command 的参数在 command_args 中进行传递。它们在 $mumbled_flags 之后将被添加到命令行。其实际的执行便是此后最终的命令行传递给 eval 运算，输入和输出以及重定向都可以在 command_args 中指定。

> **注意**
>
> 永远不要 在 command_args 包含破折号选项， 类似 -X 或 --foo 这样的。command_args 的内容将出现在最终命令行的末尾，因此它们可能是紧接在 ${name}_flags 中所列出的参数后面； 但大多的命令将不能识别出普通参数后的破折号选项。 更好的传递附加给 $command 的选项的方式是添加它们到 ${name}_flags 的起始处。另一种方法是像后文所示的那样来修改 rc_flags。

❷ 一个得体的守护进程会创建一个 pidfile 进程文件， 以使其进程能够更容易更可靠地被找到。如果设置了 pidfile 变量，告诉 rc.subr(8) 哪里能找到供其默认方法所使用的 pidfile 进程文件。

注意

事实上，rc.subr(8) 在启动一个守护进程前还会使用 pidfile 进程文件来查看它是否已经在运行。使用了 faststart 参数可以跳过这个检查步骤。

❸ 如果守护进程只有在确定的文件存在的情况下才可以运行，那就将它们列到 required_files 中，而 rc.subr(8) 将在启动守护进程之前检查那些文件是否存在。还有相关的分别用来检查目录和环境变量的 required_dirs 和 required_vars 可供使用。它们都在 rc.subr(8) 中有详细的说明。

注意

来自 rc.subr(8) 的默认方法，通过使用 forcestart 作为脚本的参数，可以强制性地跳过预先需要的检查。

❹ 我们可以在守护进程有异常的时候，自定义发送给守护进程的信号。特别是，sig_reload 指定了使守护进程重新装载其配置的信号；默认情况也就是 SIGHUP 信号。另一个信号是发送给守护进程以停止该进程；默认情况下是 SIGTERM 信号，但这是可以通过设置 sig_stop 来进行适当更改的。

注意

信号名称应当以不包含 SIG 前缀的形式指定给 rc.subr(8)，就如范例中所示的那样。FreeBSD 版本的 kill(1) 程序能够识别出 SIG 前缀，不过其它系统版本的就不一定了。

❺❻ 在默认的方法前面或后面执行附加任务是很容易的。对于我们脚本所支持的每条命令参数而言，我们可以定义 *argument*_precmd 和 *argument*_postcmd 来完成。这些 sh(1) 命令分别在它们各自的方法前后被调用，显然，从它们各自的名字就能看出来。

> ## 注意
>
> 如果我们需要的话，用自定义的 *argument*_cmd 改写默认的方法，并不妨碍我们仍然使用 *argument*_precmd 和 *argument*_postcmd。特别是，前者便于检查自定义的方法，以及执行自身命令之前所遇到更严密的条件。于是，将 *argument*_precmd 和 *argument*_cmd 一起使用，使我们合理地将检查从动作中独立了出来。
>
> 别忘了你可以将任意的有效的 sh(1) 表达式插入到方法和你定义的 pre- 与 post-commands 命令中。在大部分情况下，调用函数使实际任务有好的风格，但千万不要让风格限制了你对其幕后到底是怎么回事的思考。

❼ 如果我们愿意实现一些自定义参数，这些参数也可被认作为我们脚本的 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。

> ## 注意
>
> reload 是个特别的命令。一方面，它有一个在 rc.subr(8) 中预置的方法。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样的重载方法，并且有些守护进程根本没有任何东西可重载的。所以显而易见，我们需要去询问都提供了哪些的内建功能。我们可以通过 extra_commands 来这样做。
>
> 我们从 reload 的默认方法得到了什么呢？守护进程常常在收到一个信号后重新载入它们的配置 — 一般来说，也就是 SIGHUP 信号。因此 rc.subr(8) 尝试发送一个信号给守护进程来重载它。该信号一般预设为 SIGHUP，但是如果必要的话可以通过 sig_reload 变量来自定义它。

❽❾ 我们的脚本提供了两个非标准的命令，plugh 和 xyzzy。我们看到它们在 extra_commands 中被列出来了，并且现在是时候给它们提供方法了。xyzzy 的方法是内联的而 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员的方便。它们还能被其它的子系统所使用， 例如，devd(8)，前提是 devd.conf(5) 中已经指定了。

全部可用命令的列表，当脚本不加参数地调用时，在 rc.subr(8) 打印出的使用方法中能够找到。例如， 这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)
```

⓭ 如果脚本需要的话，它可以调用自己的标准或非标准的命令。 这可能看起来有点像函数的调用，但我们知道，命令和 shell 函数并非一直都是同样的东西。举个例子，xyzzy 在这里不是以函数来实现的。另外，还有应该被有序调用的 pre-command 预置命令和 post-command 后置命令。 所以脚本运行自己命令的合适方式就是利用 rc.subr(8)， 就像范例中展示的那样。

❾ 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为零的退出值， 当且仅当该变量设置为 YES，或 TRUE，或 ON，或 1，区分大小写；否则返回一个非零的退出值。 在第二种情况中，函数测试变量的设置为 NO， FALSE，OFF，或 0，区分大小写； 如果变量包含别的内容的话它打印一条警告信息，例如，垃圾。

切记对 sh(1) 而言零值意味着真而非零值意味着假。

> **重要**
>
> checkyesno 函数使用一个 变量名。不要扩大含义将变量的 值 传递给它； 否则它不会如你预期那样的工作。
>
> 下面是 checkyesno 的合理使用范围:
>
> ```
> if checkyesno mumbled_enable; then
>         foo
> fi
> ```
>
> 相反地，以下面的方式调用 checkyesno 是不会工作的 — 至少是不会如你预期的那样:
>
> ```
> if checkyesno -"${mumbled_enable}"; then
>         foo
> fi
> ```

❿ 我们可以通过修改 $start_precmd 中的 rc_flags 来影响传递到 $command 的标帜。

⓫ 某种情况下我们可能需要发出一条重要的信息，那样的话 syslog 可以很好地记录日志。 这可以使用下列 rc.subr(8) 函数来轻松完成： debug，info，warn，以及 err。 后者以指定的代码值退出脚本。

⓬ 方法的退出值和它们的 pre-commands 预命令不只是默认被忽略掉。如果 *argument*_precmd 返回了一个非零退出值，主方法将不会被执行。依次地是，*argument*_postcmd 将不会被调用，除非主方法返回的是一个为零的退出值。

> **注意**
>
> 然而，当给一个参数使用 force 前缀的时候，如 forcestart，rc.subr(8) 会听从命令行指示而忽略那些退出值最后仍然调用所有的命令。

7. 链接脚本到 rc.d 框架

当编写好了一个脚本，它需要被整合到 rc.d 中去。 一个重要的步骤就是安装脚本到 /etc/rc.d （对基本系统而言）或 /usr/local/etc/rc.d （对ports而言）中去。在 <bsd.prog.mk> 和 <bsd.port.mk> 中都为此提供了方便的接口， 通常你不必担心适当的所有权限和模式。系统脚本应当是通过可以在 src/etc/rc.d 找到的 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。

不过，我们应该预先考虑到我们脚本在系统启动顺序中的位置。 我们的脚本所处理的服务可能依赖于其它的服务。举个例子， 没有网络接口和路由选择的启用运行的话，一个网络守护进程是不起作用的。 即使一个服务看似什么都不需要，在基本文件系统检查挂载完毕之前也很难启动。

之前我们曾提到过 rcorder(8)。现在是时候来密切地关注下它了。 笼统地说，rcorder(8) 处理一组文件，检验它们的内容， 并从文件集合打印一个文件列表的依赖顺序到 stdout 标准输出。这点是用于保持文件内部的依赖信息， 而每个文件只能说明自己的依赖。一个文件可以指定如下信息：

- 它 提供 的 “条件” 的名字（意味着我们服务的名字）；
- 它 需求 的 “条件” 的名字；
- 应该 先 运行的文件的 “条件” 的名字；
- 能用于从全部文件集合中选择一个子集的额外 关键字 （ rcorder(8) 可通过选项而被指定来包括或省去由特殊关键字所列出的文件。）

并不奇怪的是，rcorder(8) 只能处理接近 sh(1) 语法的文本文件。rcorder(8) 所解读的特殊行看起来类似 sh(1) 的注释。这种特殊文本行的语法相当严格地简化了其处理。 请查阅 rcorder(8) 以获取更详细的信息。

除使用 rcorder(8) 的特殊行以外， 脚本可以坚持将其依赖的其它服务强制性启动。当其它服务是可选的， 并因系统管理员错误地在 rc.conf(5) 中禁用掉该服务而使其不能自行启动时，会需要这一点。

将这些谨记在心，我们来考虑下简单结合了依赖信息增强的守护进程脚本：

```
#!/bin/sh

# PROVIDE: mumbled oldmumble ❶
# REQUIRE: DAEMON cleanvar frotz❷
# BEFORE:  LOGIN❸
# KEYWORD: nojail shutdown❹

. -/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 1❺
 fi
 return 0
}

load_rc_config $name
run_rc_command -"$1"
```

跟前面一样，做如下详细分析：

❶ 该行声明了我们脚本所提供的 “条件” 的名字。 现在其它脚本可以用那些名字来标明我们脚本的依赖。

> **注意**
>
> 通常脚本指定一个单独的已提供的条件。然而， 并没有什么妨碍我们从列出的那些条件中指定，例如， 为了兼容性的目的。

在其它情况，主要的名称，或者说唯一的， PROVIDE: 条件应该与 ${name} 相同。

❷❸ 因此我们的脚本指示了其依赖于别的脚本所提供的 “条件” 。根据这些行的信息，脚本请示 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。

> **注意**
>
> BEFORE: 这一行不可以在其它脚本不完整的依赖关系列表中滥用。 适合使用 BEFORE: 的情况是当其它脚本不关心我们的脚本， 但是我们的脚本如果在另一个之前运行的话能够更好地执行任务。 一个典型的实例是网络接口和防火墙： 虽然接口不依赖防火墙来完成自己的工作， 但是系统安全将因一切网络流量之前启动的防火墙而受益。
>
> 除了条件相对应的每个单独服务，脚本使用元条件和它们的 “占位符” 来保证某个操作组在其它之前被执行。 这些是由 UPPERCASE 大写名字所表示的。它们的列表和用法可以在 rc(8) 中找到。
>
> 切记将一个服务名称放进 REQUIRE: 行不能保证实际的服务会在我们的脚本启动的时候运行。 所需求的服务可能会启动失败或在 rc.conf(5) 中被禁掉了。 显然，rcorder(8) 是无法追踪这些细节的，并且 rc(8) 也不会去追踪。所以， 脚本启动的应用程序应当能够应付任何所需求的服务的不可用情况。 某些情况下，我们可以用 下面 所讨论的方式来协助脚本。

❹ 如我们从上述文字所记起的，rcorder(8) 关键字可以用来选择或省略某些脚本。即任何 rcorder(8) 用户可以通过指定 -k 和 -s 选项来分别指定 “保留清单 (keep list) ” 和 “跳过清单 (skip list) ” 。 从全部文件到按依赖关系排列的清单，rcorder(8) 将只是挑出保留清单 (除非是空的) 中那些带关键字的以及从跳过清单中挑出不带关键字的文件。

在 FreeBSD 中，rcorder(8) 被 /etc/rc 和 /etc/rc.shutdown 所使用。 这两个脚本定义了 FreeBSD 中 rc.d 关键字以及它们的意义的标准列表如下：

nojail
: 该服务不适用于 jail(8) 环境。 如果是在 jail 的内部的话，自动启动和关闭程序将忽略该脚本。

nostart
: 该服务只能手动启动否则将不会启动。 自动启动程序将忽略此脚本。结合 shutdown 关键字的话，这可以用来编写只在系统关闭时执行一些任务的脚本。

shutdown
: 这个关键字 明确 地列出了需要在系统关闭前停止的服务。

> ### 注意
>
> 当系统即将关闭的时候， /etc/rc.shutdown 在运行。 它假定认为大部分的 rc.d 脚本在那刻什么都不做。 因此， /etc/rc.shutdown 选择性地调用带有 shutdown 关键字的 rc.d 脚本， 有效地忽略其余的脚本。为了更快的关闭， /etc/rc.shutdown 传递 faststop 命令给其运行的脚本， 以跳过预置的检查，例如，进程文件 pidfile 的检查。 正如依赖性服务应该在其所依赖的服务之前停止， /etc/rc.shutdown 以相反的依赖次序来运行这些脚本。
>
> 如果写一个真正的 rc.d 脚本的话， 你应当考虑到其是否与系统关闭时有关系。例如， 如果你的脚本只通过响应 start 命令来运行任务，那么你不需要包含这个关键字。然而， 如果你的脚本管理着一个服务，那么，在系统进入 halt(8) 中所描述的其本身关闭顺序的最终阶段之前停止该脚本， 可能是个不错的主意。特别是， 你显然是应该关闭一个需要相当长时间， 或需要特定的动作才能干净地关闭的服务。 数据库引擎就是这样一个典型的例子。

❺ 以 force_depend 起始的行应被用于更谨慎的情况。通常，用于修正相互关联的 rc.d 脚本分层结构的配置文件时会更加稳妥。

如果你仍不能完成不含 force_depend 的脚本， 范例提供了一个如何有条件地调用它的习惯用法。在范例中，我们的 mumbled 守护进程需求另一个以高级方式启动的进程， frotz。但 frotz 也是可选的； 而且 rcorder(8) 对这些信息

是一无所知的。幸运的是， 我们的脚本已访问到全部的 rc.conf(5) 变量。如果 frotz_enable 为真，我们希望的最好结果是依靠 rc.d 已经启动了 frotz。否则我们强制检查 frotz 的状态。最终， 如果 frotz 依赖的服务没有找到或运行的话， 我们将强制其运行。这时 force_depend 将发出一条警告信息，因为它只应该在检查到配置信息丢失的情况下被调用。

8. 给予 rc.d 脚本更多的灵活性

当进行启动或停止的调用时，rc.d 脚本应该作用于其所负责的整个子系统。例如，/etc/rc.d/netif 应该启动或停止 rc.conf(5) 中所描述的全部网络接口。每个任务都唯一地听从一个如 start 或 stop 这样的单独命令参数的指示。在启动和停止之间的时间， rc.d 脚本帮助管理员控制运行中的系统， 并其在需要的时候它将产生更多的灵活性和精确性。举个例子， 管理员可能想在 rc.conf(5) 中添加一个新网络接口的配置信息， 然后在不妨碍其它已存在接口的情况下将其启动。 在下次管理员可能需要关闭一个单独的网络接口。在魔幻的命令行中， 对应的 rc.d 脚本调用一个额外的参数， 网络接口名即可。

幸运的是，rc.subr(8) 允许传递任意多（取决于系统限制）的参数给脚本的方法。 由于这个原因，脚本自身的改变可以说是微乎其微。

rc.subr(8) 如何访问到附加的命令行参数呢？直接获取么？ 并非是无所不用其极的。首先，sh(1) 函数没有访问到调用者的定位参数，而 rc.subr(8) 只是这些函数的容器。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传递给它的方法。

所以 rc.subr(8) 提供了如下的方法： run_rc_command 传递其所有参数但将第一个参数逐字传递到各自的方法。首先， 发出以方法自身为名字的参数：start，stop，等等。这会被 run_rc_command 移出， 这样命令行中原本 $2 的内容将作为 $1 来提供给方法，等等。

为了说明这点，我们来修改原来的虚拟脚本， 这样它的信息将取决于所提供的附加参数。从这里出发：

```
#!/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 -]; then❶
                echo -"Greeting message: $*"
        else
                echo -"Nothing started."
        fi
}

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

load_rc_config $name
run_rc_command -"$@"❸
```

能注意到脚本里发生了那些实质性改变么?

❶ 你输入的所有在 start 之后的参数可以被当作各自方法的定位参数一样被终结。 我们可以根据我们的任务、技巧和想法来以任何方式使用他们。 在当前的例子中，我们只是以下行中字符串的形式传递参数给 echo(1) 程序 — 注意 $* 是有双引号的。这里是脚本如何被调用的:

```
# /etc/rc.d/dummy start
Nothing started.
# /etc/rc.d/dummy start Hello world!
Greeting message: Hello world!
```

❷ 同样用于我们脚本提供的任何方法，并不仅限于标准的方法。 我们已经添加了一个自定义的叫做 kiss 的方法， 并且它给附加参数带来的戏耍决不亚于 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...
```

❸ 如果我们只是传递所有附加参数给任意的方法， 我们只需要在脚本的最后一行我们调用 run_rc_command 的地方， 用 "$@ 代替 "$1" 即可。

重要

一个 sh(1) 程序员应该是可以理解 $* 和 $@ 的微妙区别只是指定全部定位参数的不同方法。 关于此更深入的探讨，可以参考这个很好的 sh(1) 脚本编程手册。在你完全理解这些表达式的意义之前请不要使用它们， 因为误用它们将给脚本引入缺陷和不安全的弊端。

注意

现在 run_rc_command 可能有个缺陷， 它将影响保持参数之间的原本边界。也就是， 带有嵌入空白的参数可能不会被正确处理。该缺陷是由于对 $* 的误用。

9．进一步阅读

Luke Mewburn 的原始文章 中讲述了 rc.d 的基本概要， 并详细阐述了其设计方案的原理。该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)， 还有 rcorder(8) 的联机手册中，对 rc.d 组件做了非常详细的记载。 在你写脚本时，如果不去学习和参考这些联机手册的话， 你是无法完全发挥出 rc.d 的能量的。

工作中实际范例的主要来源就是运行的系统中的 /etc/rc.d 目录。 它的内容可读性非常好，因为大部分的枯燥的内容都深藏在 rc.subr(8) 中了。切记 /etc/rc.d 的脚本也不是神仙写出来的， 所以它们可能也存在着代码缺陷以及低级的设计方案。但现在你可以来改进它们了！