Практическое руководство по написанию rc.d скриптов в BSD

Аннотация

Новичкам может быть сложно соотнести факты из официальной документации по фреймворку rc.d в BSD с практическими задачами написания скриптов для rc.d. В этой статье мы рассмотрим несколько типичных случаев возрастающей сложности, покажем возможности rc.d, подходящие для каждого случая, и обсудим, как они работают. Такое рассмотрение должно дать ориентиры для дальнейшего изучения устройства и эффективного применения rc.d.

Содержание

1. Введение

Исторически в BSD был монолитный стартовый сценарий /etc/rc. Он вызывался init(8) во время загрузки системы и выполнял все задачи пользовательского пространства, необходимые для многопользовательского режима: проверку и монтирование файловых систем, настройку сети, запуск демонов и так далее. Точный список задач не был одинаковым в каждой системе; администраторам требовалось его настраивать. За редкими исключениями, /etc/rc приходилось изменять, и настоящим хакерам это нравилось.

Основная проблема монолитного подхода заключалась в том, что он не предоставлял контроля над отдельными компонентами, запускаемыми из /etc/rc. Например, /etc/rc не мог перезапустить отдельный демон. Администратору системы приходилось вручную находить процесс демона, завершать его, ждать, пока он действительно завершится, затем искать в /etc/rc нужные флаги и, наконец, вводить полную командную строку для повторного

запуска демона. Задача становилась ещё сложнее и более подверженной ошибкам, если служба состояла из нескольких демонов или требовала дополнительных действий. Одним словом, единый скрипт не справлялся с тем, для чего скрипты вообще предназначены: облегчать жизнь администратору системы.

Позже была предпринята попытка разделить некоторые части /etc/rc для возможности отдельного запуска наиболее важных подсистем. Известным примером стал /etc/netstart, предназначенный для настройки сети. Это позволяло получить доступ к сети в однопользовательском режиме, но плохо интегрировалось в автоматический процесс запуска, так как части его кода требовалось переплетаться с действиями, по сути не связанными с сетью. Именно поэтому /etc/netstart превратился в /etc/rc.network. Последний больше не был обычным скриптом; он состоял из больших, запутанных функций sh(1), вызываемых из /etc/rc на разных этапах загрузки системы. Однако по мере того, как задачи при запуске становились разнообразнее и сложнее, "квазимодульный" подход стал ещё большей обузой, чем монолитный /etc/rc.

Без чистого и хорошо продуманного каркаса, стартовые скрипты вынуждены были идти на всевозможные ухищрения, чтобы удовлетворить потребности быстро развивающихся BSD-ориентированных операционных систем. В конце концов стало очевидно, что необходимы дополнительные шаги на пути к детализированной и расширяемой системе rc. Так появилась BSD rc.d. Её признанными создателями стали Люк Мьюберн и сообщество NetBSD. Позже она была импортирована в FreeBSD. Её название отсылает к расположению системных скриптов для отдельных служб, которое находится в /etc/rc.d. Вскоре мы узнаем больше о компонентах системы rc.d и увидим, как вызываются отдельные скрипты.

Основные идеи, лежащие в основе BSD rc.d, — это *тонкая модульность* и *повторное использование кода*. *Тонкая модульность* означает, что каждая базовая «служба», такая как системный демон или примитивная задача запуска, получает собственный сценарий sh(1), способный запустить службу, остановить её, перезагрузить или проверить её состояние. Конкретное действие выбирается аргументом командной строки, переданным в сценарий. Сценарий /etc/rc по-прежнему управляет запуском системы, но теперь он просто вызывает небольшие сценарии один за другим с аргументом `start`. Также легко выполнять задачи завершения работы, запуская тот же набор сценариев с аргументом `stop`, что и делает /etc/rc.shutdown. Обратите внимание, насколько это близко следует Unix-подходу, где используется набор небольших специализированных инструментов, каждый из которых выполняет свою задачу наилучшим образом. *Повторное использование кода* означает, что общие операции реализованы как функции sh(1) и собраны в /etc/rc.subr. Теперь типичный сценарий может состоять всего из нескольких строк кода sh(1). Наконец, важной частью инфраструктуры rc.d является rcorder(8), который помогает /etc/rc упорядоченно запускать небольшие сценарии с учётом зависимостей между ними. Он также может помочь /etc/rc.shutdown, поскольку правильный порядок завершения работы противоположен порядку запуска.

Дизайн BSD rc.d описан в оригинальной статье Люка Мьюберна, а компоненты rc.d подробно документированы в соответствующих страницах Справочника. Однако новичку в rc.d может быть неочевидно, как связать многочисленные элементы вместе, чтобы создать хорошо структурированный скрипт для конкретной задачи. Поэтому в этой статье будет предпринята попытка описать rc.d с другого ракурса. В ней будет показано, какие функции следует использовать в ряде типичных случаев и почему. Обратите внимание, что это не

руководство HowTo, поскольку наша цель — не предоставление готовых рецептов, а демонстрация нескольких простых способов входа в мир rc.d. Также эта статья не заменяет соответствующие страницы Справочника. Не стесняйтесь обращаться к ним для получения более формальной и полной документации во время чтения этой статьи.

Для понимания этой статьи есть предварительные требования. Прежде всего, вы должны быть знакомы с языком написания сценариев sh(1), чтобы освоить rc.d. Кроме того, вы должны знать, как система выполняет задачи запуска и завершения работы пользовательского пространства, что описано в rc(8).

Эта статья посвящена ветке FreeBSD в rc.d. Тем не менее, она может быть полезна и разработчикам NetBSD, потому что две ветки BSD rc.d не только разделяют одинаковый дизайн, но и остаются схожими в аспектах, видимых авторам скриптов.

2. Обрисовка задачи

Немного размышлений перед запуском `$EDITOR` не повредит. Чтобы написать хорошо продуманный скрипт rc.d для системной службы, сначала нужно ответить на следующие вопросы:

- Является ли служба обязательной или опциональной?
- Будет ли скрипт обслуживать одну программу, например, демон, или выполнять более сложные действия?
- От каких других служб зависит наша служба, и наоборот?

Из следующих примеров мы увидим, почему важно знать ответы на эти вопросы.

3. Примитивный скрипт

Следующий скрипт просто выводит сообщение каждый раз при загрузке системы:

```
#!/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
```

Вот что следует учитывать:

 Интерпретируемый скрипт должен начинаться с "волшебной" строки shebang. Эта строка указывает программу-интерпретатор для скрипта. Благодаря строке shebang скрипт может быть запущен точно так же, как бинарная программа, при условии что у него установлен бит выполнения. (См. chmod(1).) Например, системный администратор может запустить наш скрипт вручную из командной строки:

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

Для корректного управления в рамках rc.d скрипты должны быть написаны на языке sh(1). Если у вас есть служба или порт, который использует двоичную утилиту управления или процедуру запуска, написанную на другом языке, установите этот компонент в /usr/sbin (для системы) или /usr/local/sbin (для портов) и вызовите его из sh(1) скрипта в соответствующем каталоге rc.d.

Если вы хотите узнать подробности о том, почему скрипты rc.d должны быть написаны на языке sh(1), изучите, как /etc/rc вызывает их с помощью `run_rc_script`, а затем изучите реализацию `run_rc_script` в /etc/rc.subr.

 В файле /etc/rc.subr определено несколько функций sh(1), которые могут использоваться скриптами rc.d. Эти функции описаны в rc.subr(8). Хотя теоретически возможно написать скрипт rc.d без использования rc.subr(8), его функции оказываются чрезвычайно полезными и значительно упрощают задачу. Поэтому неудивительно, что все используют rc.subr(8) в скриптах rc.d. Мы не будем исключением.

Файл rc.d должен "подгрузить" (/etc/rc.subr, включить его с помощью “.”) *до* вызова функций rc.subr(8), чтобы у sh(1) была возможность знать об этих функциях заранее. Предпочтительный стиль — подгружать /etc/rc.subr в самом начале.

Некоторые полезные функции, связанные с сетью, предоставляются другим включаемым файлом — /etc/network.subr.

 Обязательная переменная `name` определяет имя нашего скрипта. Она требуется rc.subr(8). То есть, каждый скрипт в rc.d *должен* установить `name` перед вызовом функций rc.subr(8).

Теперь самое время раз и навсегда выбрать уникальное имя для нашего скрипта. Мы будем использовать его в нескольких местах при разработке скрипта. Содержимое переменной name должно соответствовать имени скрипта, так как некоторые части FreeBSD (например, сервисные клетки (jail) и функция cpuset в rc framework) зависят от этого. Таким образом, имя файла также не должно содержать символов, которые могут вызвать проблемы в скриптах (например, не используйте дефис "-" и другие).

Текущий стиль написания скриптов в rc.d заключается в заключении значений, присваиваемых переменным, в двойные кавычки. Имейте в виду, что это всего лишь вопрос стиля, который может быть не всегда применим.

> Вы можете безопасно опустить кавычки вокруг простых слов без метасимволов sh(1), тогда как в некоторых случаях вам понадобятся одинарные кавычки, чтобы предотвратить интерпретацию значения sh(1). Программист должен уметь отличать синтаксис языка от стилевых соглашений и разумно использовать и то, и другое.

 Основная идея rc.subr(8) заключается в том, что скрипт rc.d предоставляет обработчики (или методы) для вызова rc.subr(8). В частности, аргументы `start`, `stop` и другие, передаваемые в скрипт rc.d, обрабатываются таким образом. Метод представляет собой выражение sh(1), сохранённое в переменной с именем `argument_cmd`, где *argument* соответствует тому, что может быть указано в командной строке скрипта. Далее мы увидим, как rc.subr(8) предоставляет стандартные методы для типовых аргументов.

> Чтобы сделать код в rc.d более единообразным, обычно используют `${name}` везде, где это уместно. Таким образом, множество строк можно просто копировать из одного скрипта в другой.

 Следует помнить, что rc.subr(8) предоставляет методы по умолчанию для стандартных аргументов. Следовательно, если мы хотим, чтобы стандартный метод ничего не делал, мы должны переопределить его с помощью no-op sh(1) выражения.

 Тело сложного метода может быть реализовано в виде функции. Хорошей практикой является использование осмысленного имени функции.

> Настоятельно рекомендуется добавлять префикс `${name}` к именам всех функций, определённых в нашем скрипте, чтобы они никогда не конфликтовали с функциями из rc.subr(8) или другого общего включаемого файла.

 Этот вызов rc.subr(8) загружает переменные rc.conf(5). Наш скрипт пока их не использует, но всё равно рекомендуется загружать rc.conf(5), потому что могут быть переменные rc.conf(5), управляющие самим rc.subr(8).

 Обычно это последняя команда в скрипте rc.d. Она вызывает механизм rc.subr(8) для выполнения запрошенного действия, используя переменные и методы, предоставленные нашим скриптом.

4. Настраиваемый фиктивный скрипт

Теперь добавим некоторые элементы управления в наш тестовый скрипт. Как вам может быть известно, скрипты rc.d управляются с помощью rc.conf(5). К счастью, rc.subr(8) скрывает от нас все сложности. Следующий скрипт использует rc.conf(5) через rc.subr(8), чтобы проверить, включен ли он вообще, и получить сообщение для отображения во время загрузки. Эти две задачи на самом деле независимы. С одной стороны, скрипт rc.d может просто поддерживать включение и выключение своего сервиса. С другой стороны, обязательный скрипт rc.d может иметь переменные конфигурации. Однако мы реализуем обе возможности в одном скрипте:

```
#!/bin/sh

. /etc/rc.subr

name=dummy
rcvar=dummy_enable ①

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

load_rc_config $name ②
: ${dummy_enable:=no} ③
: ${dummy_msg="Nothing started."} ④

dummy_start()
{
    echo "$dummy_msg" ⑤
}

run_rc_command "$1"
```

Что изменилось в этом примере?

① Переменная `rcvar` определяет имя переменной-переключателя ON/OFF.

② Теперь `load_rc_config` вызывается раньше в скрипте, до обращения к любым переменным rc.conf(5).

При изучении скриптов в rc.d следует помнить, что sh(1) откладывает вычисление выражений в функции до её вызова. Поэтому не будет ошибкой вызвать `load_rc_config` непосредственно перед `run_rc_command` и при этом обращаться к переменным rc.conf(5) из функций методов, экспортируемых в `run_rc_command`. Это связано с тем, что функции методов вызываются `run_rc_command`, который выполняется *после* `load_rc_config`.

③ `run_rc_command` выдаст предупреждение, если переменная `rcvar` установлена, но указанная переменная-флаг не задана. Если ваш скрипт rc.d предназначен для базовой системы, вы должны добавить значение по умолчанию для флага в /etc/defaults/rc.conf и задокументировать его в rc.conf(5). В противном случае ваш скрипт должен предоставить значение по умолчанию для флага. Канонический подход для последнего случая показан в примере.

Вы можете заставить rc.subr(8) действовать так, как если бы переключатель установлен в `ON`, независимо от его текущего значения, добавив перед аргументом скрипта префикс `one` или `force`, например `onestart` или `forcestop`. Однако учтите, что `force` имеет другие опасные эффекты, которые мы затронем ниже, тогда как `one` просто переопределяет переключатель ON/OFF. Например, предположим, что `dummy_enable` установлен в `OFF`. Следующая

команда выполнит метод `start`, несмотря на настройку:

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

 Теперь сообщение, отображаемое при загрузке, больше не жёстко закодировано в скрипте. Оно задается переменной `dummy_msg` в rc.conf(5). Это простой пример того, как переменные rc.conf(5) могут управлять скриптом в rc.d.

Имена всех переменных rc.conf(5), используемых исключительно нашим скриптом, *должны* иметь один и тот же префикс: `${name}_`. Например: `dummy_mode`, `dummy_state_file` и так далее.

Хотя можно использовать более короткое имя внутри, например, просто `msg`, добавление уникального префикса `${name}_` ко всем глобальным именам, вводимым нашим скриптом, избавит нас от возможных конфликтов с пространством имён rc.subr(8).

Как правило, скрипты rc.d базовой системы не должны предоставлять значения по умолчанию для своих переменных rc.conf(5), поскольку значения по умолчанию должны быть установлены в /etc/defaults/rc.conf. С другой стороны, скрипты rc.d для портов должны предоставлять значения по умолчанию, как показано в примере.

 Здесь мы используем `dummy_msg` для фактического управления нашим скриптом, т.е., для выдачи переменного сообщения. Использование shell-функции здесь избыточно, так как она выполняет только одну команду; равнозначной альтернативой является:

```
start_cmd="echo \"$dummy_msg\""
```

5. Запуск и остановка простого демона

Мы ранее говорили, что rc.subr(8) может предоставлять методы по умолчанию. Очевидно, что такие методы не могут быть слишком общими. Они подходят для стандартного случая запуска и остановки простого демона. Предположим, что нам нужно написать скрипт rc.d для такого демона с именем `mumbled`. Вот он:

```
#!/bin/sh

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}" ①
```

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

Приятно просто, не так ли? Давайте рассмотрим наш небольшой скрипт. Единственное новое, на что стоит обратить внимание, это следующее:

 Переменная `command` имеет значение для rc.subr(8). Если она установлена, rc.subr(8) будет действовать по сценарию обслуживания обычного демона. В частности, будут предоставлены стандартные методы для таких аргументов: `start`, `stop`, `restart`, `poll` и `status`.

Демон будет запущен выполнением `$command` с флагами командной строки, указанными в `$mumbled_flags`. Таким образом, все входные данные для метода `start` по умолчанию доступны в переменных, установленных нашим скриптом. В отличие от `start`, другие методы могут требовать дополнительной информации о запущенном процессе. Например, `stop` должен знать PID процесса, чтобы завершить его. В данном случае, rc.subr(8) будет просматривать список всех процессов, ища процесс с именем, равным `procname`. Последний является ещё одной значимой переменной для rc.subr(8), и её значение по умолчанию совпадает со значением `command`. Другими словами, когда мы устанавливаем `command`, `procname` фактически устанавливается в то же значение. Это позволяет нашему скрипту завершить демон и проверить, запущен ли он вообще.

Некоторые программы на самом деле являются исполняемыми скриптами. Система запускает такие скрипты, запуская их интерпретатор и передавая ему имя скрипта в качестве аргумента командной строки. Это отражается в списке процессов, что может сбить с толку rc.subr(8). Дополнительно следует установить `command_interpreter`, чтобы rc.subr(8) знал фактическое имя процесса, если `$command` является скриптом.

Для каждого скрипта rc.d существует необязательная переменная rc.conf(5), которая имеет приоритет над `command`. Её имя формируется следующим образом: `${name}_program`, где `name` — это обязательная переменная, которую мы обсуждали ранее. Например, в данном случае это будет `mumbled_program`. Именно rc.subr(8) обеспечивает переопределение `command` с помощью `${name}_program`.

Конечно, sh(1) позволяет установить `${name}_program` из rc.conf(5) или самого скрипта, даже если `command` не задан. В этом случае специальные свойства `${name}_program` теряются, и она становится обычной переменной, которую ваш скрипт может использовать для своих целей. Однако использование `${name}_program` в одиночку не рекомендуется, так как совместное использование с `command` стало идиомой в rc.d скриптах.

Для получения более подробной информации о стандартных методах обратитесь к rc.subr(8).

6. Запуск и остановка продвинутого демона

Добавим немного мяса к костям предыдущего скрипта и сделаем его более сложным и функциональным. Стандартные методы могут хорошо справляться с задачами, но иногда требуется их тонкая настройка. Теперь мы узнаем, как адаптировать стандартные методы под наши нужды.

```
#!/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/${name}.conf /usr/share/misc/${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"
```

① Дополнительные аргументы для `$command` могут быть переданы в `command_args`. Они будут добавлены в командную строку после `$mumbled_flags`. Поскольку итоговая командная строка передаётся в `eval` для фактического выполнения, перенаправления ввода и вывода могут быть указаны в `command_args`.

Никогда не включайте параметры с дефисами, такие как `-X` или `--foo`, в `command_args`. Содержимое `command_args` будет добавлено в конец итоговой командной строки, поэтому, скорее всего, окажется после аргументов, указанных в `${name}_flags`; однако большинство команд не распознают параметры с дефисами после обычных аргументов. Лучший способ передать дополнительные параметры в `$command` — добавить их в начало `${name}_flags`. Другой способ — изменить `rc_flags` как показано далее.

② Вежливый демон должен создавать *pidfile*, чтобы его процесс можно было найти проще и надёжнее. Переменная `pidfile`, если она установлена, указывает rc.subr(8), где можно найти pidfile для использования его стандартными методами.

На самом деле, rc.subr(8) также использует pidfile для проверки, запущен ли демон, перед его запуском. Эту проверку можно пропустить, используя аргумент `faststart`.

③ Если демон не может работать без определённых файлов, просто укажите их в `required_files`, и rc.subr(8) проверит их наличие перед запуском демона. Также существуют `required_dirs` и `required_vars` для каталогов и переменных окружения соответственно. Все они подробно описаны в rc.subr(8).

Метод по умолчанию из rc.subr(8) можно принудительно заставить пропустить проверки предварительных условий, используя аргумент `forcestart` в скрипте.

④ Мы можем настроить сигналы, отправляемые демону, если они отличаются от общеизвестных. В частности, `sig_reload` указывает сигнал, который заставляет демона перезагрузить свою конфигурацию; по умолчанию это SIGHUP. Другой сигнал отправляется для остановки процесса демона; по умолчанию используется SIGTERM, но это можно изменить, установив `sig_stop` соответствующим образом.

Имена сигналов должны указываться для rc.subr(8) без префикса SIG, как показано в примере. Версия kill(1) в FreeBSD может распознавать префикс SIG, но версии из других типов ОС могут не поддерживать его.

Выполнение дополнительных задач до или после стандартных методов — это просто. Для каждого аргумента команды, поддерживаемого нашим скриптом, мы можем определить argument_precmd и argument_postcmd. Эти команды sh(1) вызываются до и после соответствующего метода, что очевидно из их названий.

Переопределение стандартного метода с помощью пользовательского argument_cmd всё равно не мешает нам использовать argument_precmd или argument_postcmd, если это необходимо. В частности, первый полезен для проверки пользовательских сложных условий, которые должны быть выполнены перед выполнением самой команды. Использование argument_precmd вместе с argument_cmd позволяет логически разделить проверки от действия.

Не забывайте, что вы можете вставлять любые допустимые выражения из sh(1) в определяемые вами методы, а также команды pre- и post-. Просто вызывать функцию, которая выполняет основную работу, — это хороший стиль в большинстве случаев, но никогда не позволяйте стилю ограничивать ваше понимание того, что происходит за кулисами.

Если мы хотим реализовать пользовательские аргументы, которые также можно рассматривать как *команды* для нашего скрипта, необходимо перечислить их в extra_commands и предоставить методы для их обработки.

Команда reload является особенной. С одной стороны, у неё есть предустановленный метод в rc.subr(8). С другой стороны, reload не предлагается по умолчанию. Причина в том, что не все демоны используют одинаковый механизм перезагрузки, а у некоторых вообще нет ничего для перезагрузки. Поэтому нам нужно явно запросить предоставление встроенной функциональности. Это можно сделать с помощью extra_commands.

Что мы получаем от метода по умолчанию для reload? Довольно часто демоны перезагружают свою конфигурацию при получении сигнала — обычно, SIGHUP. Поэтому rc.subr(8) пытается перезагрузить демона, отправляя ему сигнал. Сигнал предустановлен на SIGHUP, но может быть изменён через sig_reload при необходимости.

Наш скрипт поддерживает две нестандартные команды: plugh и xyzzy. Мы видели их в списке extra_commands, и теперь пришло время реализовать методы для них. Метод для xyzzy просто встроен в код, а для plugh он реализован как функция mumbled_plugh.

Нестандартные команды не вызываются во время запуска или завершения работы. Обычно они предназначены для удобства системного администратора. Они также могут использоваться другими подсистемами, например, devd(8), если указаны в devd.conf(5).

Полный список доступных команд можно найти в строке использования, выводимой rc.subr(8), когда скрипт вызывается без аргументов. Например, вот строка использования из изучаемого скрипта:

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

 Скрипт может вызывать свои собственные стандартные или нестандартные команды, если это необходимо. Это может выглядеть похоже на вызов функций, но мы знаем, что команды и функции оболочки не всегда одно и то же. Например, `xyzzy` не реализован как функция в данном случае. Кроме того, могут существовать пред-команда и пост-команда, которые должны вызываться в определённом порядке. Поэтому правильный способ для скрипта выполнить свою собственную команду — с помощью rc.subr(8), как показано в примере.

 Полезная функция `checkyesno` предоставляется rc.subr(8). Она принимает имя переменной в качестве аргумента и возвращает нулевой код выхода только если переменная установлена в `YES`, `TRUE`, `ON` или `1`, без учёта регистра; в противном случае возвращается ненулевой код выхода. В последнем случае функция проверяет, установлена ли переменная в `NO`, `FALSE`, `OFF` или `0`, также без учёта регистра; если переменная содержит что-то иное (т.е. мусор), функция выводит предупреждение.

Имейте в виду, что для sh(1) нулевой код возврата означает истину, а ненулевой код возврата означает ложь.

Функция `checkyesno` принимает *имя переменной*. Не передавайте ей *значение* переменной; это не будет работать, как ожидается.

Ниже приведено правильное использование `checkyesno`:

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

Напротив, вызов `checkyesno`, как показано ниже, не сработает — по крайней мере, не так, как ожидается:

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

 Мы можем влиять на флаги, передаваемые команде `$command`, изменяя `rc_flags` в `$start_precmd`.

 В некоторых случаях может потребоваться вывести важное сообщение, которое также

должно попасть в `syslog`. Это можно легко сделать с помощью следующих функций rc.subr(8): `debug`, `info`, `warn` и `err`. Последняя функция завершает выполнение скрипта с указанным кодом.

 Коды выхода из методов и их предварительных команд не просто игнорируются по умолчанию. Если `argument_precmd` возвращает ненулевой код выхода, основной метод не будет выполнен. В свою очередь, `argument_postcmd` не будет вызван, если основной метод возвращает ненулевой код выхода.

Однако rc.subr(8) можно указать из командной строки игнорировать эти коды завершения и выполнять все команды в любом случае, добавив префикс `force` к аргументу, например `forcestart`.

7. Подключение скрипта к инфраструктуре rc.d

После написания скрипта его необходимо интегрировать в rc.d. Ключевой шаг — установка скрипта в /etc/rc.d (для базовой системы) или /usr/local/etc/rc.d (для портов). И bsd.prog.mk, и bsd.port.mk предоставляют удобные механизмы для этого, и обычно вам не нужно беспокоиться о правильных правах доступа и режиме. Системные скрипты должны устанавливаться из src/libexec/rc/rc.d через Makefile, находящийся там. Скрипты портов можно установить с помощью `USE_RC_SUBR`, как описано в Руководстве FreeBSD по созданию портов.

Однако следует заранее продумать место нашего скрипта в последовательности запуска системы. Скорее всего, обслуживаемый нашим скриптом сервис зависит от других сервисов. Например, сетевой демон не может работать без поднятых сетевых интерфейсов и маршрутизации. Даже если сервис, казалось бы, ничего не требует, он вряд ли сможет запуститься до проверки и монтирования основных файловых систем.

Мы уже упоминали rcorder(8). Теперь пришло время рассмотреть его подробнее. В двух словах, rcorder(8) принимает набор файлов, анализирует их содержимое и выводит на `stdout` список этих файлов, упорядоченный по зависимостям. Главная идея заключается в том, чтобы хранить информацию о зависимостях *внутри* файлов, чтобы каждый файл мог описывать только себя. Файл может содержать следующую информацию:

- имена "условий" (что для нас означает сервисы), которые он *предоставляет*;
- имена "условий", которые он *требует*;
- имена "условий", перед которыми должен выполняться этот файл;
- дополнительные *ключевые слова*, которые могут использоваться для выбора подмножества из всего набора файлов (rcorder(8) может быть настроен с помощью опций для включения или исключения файлов, содержащих указанные ключевые слова.)

Неудивительно, что rcorder(8) может обрабатывать только текстовые файлы с синтаксисом, близким к sh(1). То есть специальные строки, понимаемые rcorder(8), выглядят как

комментарии в sh(1). Синтаксис таких специальных строк довольно жёсткий, чтобы упростить их обработку. Подробности см. в rcorder(8).

Помимо использования специальных строк rcorder(8), скрипт может настаивать на своей зависимости от другой службы, просто принудительно запуская её. Это может быть необходимо, когда другая служба является опциональной и не запускается самостоятельно, потому что системный администратор ошибочно отключил её в rc.conf(5).

С учётом этих общих знаний рассмотрим простой скрипт демона, дополненный зависимостями:

```
#!/bin/sh

# PROVIDE: mumbled oldmumble ①
# REQUIRE: DAEMON cleanvar frotz ②
# 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"
```

Как и ранее, следует детальный анализ:

 Эта строка объявляет названия "условий", которые предоставляет наш скрипт. Теперь другие скрипты могут указывать зависимость от нашего скрипта по этим именам.

Обычно скрипт указывает одно предоставленное условие. Однако ничто не мешает нам перечислить несколько условий, например, по причинам совместимости.

В любом случае, название основного или единственного условия `PROVIDE:` должно совпадать с `${name}`.

Таким образом, наш скрипт указывает, от каких "условий", предоставляемых другими скриптами, он зависит. Согласно строкам, наш скрипт просит rcorder(8) разместить его после скрипта(ов), предоставляющих DAEMON и cleanvar, но перед тем, который предоставляет LOGIN.

Строку `BEFORE:` не следует использовать для обхода неполного списка зависимостей в другом скрипте. Правильный случай для использования `BEFORE:` — когда другой скрипт не зависит от нашего, но наш скрипт может выполнить свою задачу лучше, если запустится до другого. Типичный пример из реальной жизни — сетевые интерфейсы и межсетевой экран: хотя интерфейсы не зависят от межсетевого экрана при выполнении своей работы, безопасность системы выиграет, если межсетевой экран будет готов до начала сетевого трафика.

Помимо условий, соответствующих отдельным службам, существуют метаусловия и их "заглушки" скриптов, используемые для обеспечения выполнения определённых групп операций в заданном порядке. Они обозначаются именами в ВЕРХНЕМ РЕГИСТРЕ. Их список и назначение можно найти в rc(8).

Имейте в виду, что указание имени службы в строке `REQUIRE:` не гарантирует, что служба действительно будет запущена к моменту старта нашего скрипта. Требуемая служба может не запуститься или быть отключена в rc.conf(5). Очевидно, rcorder(8) не может отслеживать такие детали, и rc(8) тоже этого не делает. Следовательно, приложение, запускаемое нашим скриптом, должно быть способно обрабатывать ситуации, когда требуемые службы недоступны. В некоторых случаях мы можем помочь ему, как описано в ниже

Как мы помним из текста выше, ключевые слова rcorder(8) могут использоваться для выбора или исключения некоторых скриптов. А именно, любой потребитель rcorder(8) может указать с помощью опций `-k` и `-s`, какие ключевые слова находятся в "списке сохранения" и "списке пропуска" соответственно. Из всех файлов, подлежащих сортировке по зависимостям, rcorder(8) выберет только те, которые имеют ключевое слово из списка сохранения (если он не пуст) и не имеют ключевого слова из списка пропуска.

В FreeBSD, rcorder(8) используется /etc/rc и /etc/rc.shutdown. Эти два скрипта определяют стандартный список ключевых слов rc.d FreeBSD и их значения следующим образом:

nojail

Сервис не предназначен для окружения jail(8). Процедуры автоматического запуска и остановки будут игнорировать скрипт, если он находится внутри клетки.

nostart

Служба должна запускаться вручную или не запускаться вовсе. Процедура автоматического запуска проигнорирует скрипт. В сочетании с ключевым словом shutdown это может использоваться для написания скриптов, выполняющих действия только при выключении системы.

shutdown

Этот ключевой параметр должен быть указан *явно*, если службу необходимо остановить перед завершением работы системы.

Когда система собирается завершить работу, выполняется /etc/rc.shutdown. Предполагается, что большинству скриптов rc.d в этот момент нечего делать. Поэтому /etc/rc.shutdown выборочно запускает скрипты rc.d с ключевым словом shutdown, фактически игнорируя остальные скрипты. Для ещё более быстрого завершения работы /etc/rc.shutdown передаёт команду faststop запускаемым скриптам, чтобы они пропускали предварительные проверки, например, проверку pid-файла. Поскольку зависимые службы должны быть остановлены до своих зависимостей, /etc/rc.shutdown запускает скрипты в обратном порядке зависимостей. Если вы пишете настоящий скрипт rc.d, стоит подумать, актуален ли он во время завершения работы системы. Например, если ваш скрипт выполняет свою работу только в ответ на команду start, то включать это ключевое слово не нужно. Однако если ваш скрипт управляет службой, вероятно, стоит остановить её до того, как система перейдёт к финальной стадии завершения работы, описанной в halt(8). В частности, службу следует останавливать явно, если для её корректного завершения требуется значительное время или специальные действия. Типичный пример такой службы — система управления базами данных.

 Прежде всего, `force_depend` следует использовать с большой осторожностью. Обычно лучше пересмотреть иерархию конфигурационных переменных для ваших rc.d скриптов, если они взаимозависимы.

Если вам всё ещё не обойтись без `force_depend`, в примере показано, как вызвать его условно. В примере наш демон `mumbled` требует, чтобы другой демон, `frotz`, был запущен заранее. Однако `frotz` также является опциональным, и rcorder(8) ничего не знает о таких деталях. К счастью, наш скрипт имеет доступ ко всем переменным rc.conf(5). Если `frotz_enable` имеет значение true, мы надеемся на лучшее и полагаемся на rc.d, что `frotz` был запущен. В противном случае мы принудительно проверяем статус `frotz`. Наконец, мы принудительно устанавливаем зависимость от `frotz`, если обнаруживаем, что он не запущен. `force_depend` выдаст предупреждение, так как его следует вызывать только в случае обнаружения неправильной конфигурации.

8. Придание большей гибкости скрипту rc.d

При вызове во время запуска или завершения работы скрипт rc.d должен воздействовать на всю подсистему, за которую он отвечает. Например, /etc/rc.d/netif должен запускать или останавливать все сетевые интерфейсы, описанные в rc.conf(5). Любая из этих задач может быть однозначно указана единственным аргументом команды, таким как `start` или `stop`. Между запуском и завершением работы скрипты rc.d помогают администратору управлять работающей системой, и именно тогда возникает потребность в большей гибкости и

точности. Например, администратор может добавить настройки нового сетевого интерфейса в rc.conf(5), а затем запустить его, не затрагивая работу существующих интерфейсов. В следующий раз администратору может потребоваться остановить отдельный сетевой интерфейс. В духе командной строки, соответствующий скрипт rc.d требует дополнительного аргумента — имени интерфейса.

К счастью, rc.subr(8) позволяет передавать любое количество аргументов (в пределах системных ограничений) методам скрипта. Благодаря этому изменения в самом скрипте могут быть минимальными.

Как rc.subr(8) может получить доступ к дополнительным аргументам командной строки. Должен ли он просто захватывать их напрямую? Ни в коем случае. Во-первых, функция sh(1) не имеет доступа к позиционным параметрам своего вызывающего объекта, но rc.subr(8) — это просто набор таких функций. Во-вторых, хороший стиль rc.d предписывает, что именно главный скрипт должен решать, какие аргументы передавать его методам.

Итак, подход, принятый в rc.subr(8), следующий: `run_rc_command` передаёт все свои аргументы, кроме первого, в соответствующий метод в неизменном виде. Первый, опущенный аргумент — это имя самого метода: `start`, `stop` и т.д. Он будет удалён с помощью `shift` в `run_rc_command`, так что то, что было `$2` в оригинальной командной строке, будет представлено как `$1` в методе, и так далее.

Чтобы проиллюстрировать эту возможность, давайте изменим примитивный скрипт-заглушку так, чтобы его сообщения зависели от дополнительных переданных аргументов. Вот как это выглядит:

```
#!/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
```

Какие основные изменения мы можем заметить в скрипте?

① Все аргументы, которые вы вводите после `start`, могут стать позиционными параметрами для соответствующего метода. Мы можем использовать их любым способом в соответствии с нашей задачей, навыками и предпочтениями. В текущем примере мы просто передаем все их в echo(1) как одну строку в следующей строке — обратите внимание на `$*` в двойных кавычках. Вот как теперь можно вызывать этот скрипт:

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

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

② То же самое относится к любому методу, который предоставляет наш скрипт, не только к стандартному. Мы добавили пользовательский метод с именем `kiss`, и он может использовать дополнительные аргументы не меньше, чем `start`. Например:

```
# /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...
```

③ Если мы хотим просто передать все дополнительные аргументы любому методу, мы можем просто заменить `"$@"` на `"$1"` в последней строке нашего скрипта, где мы вызываем `run_rc_command`.

Программист sh(1) должен понимать тонкую разницу между `$*` и `$@` как способами обозначения всех позиционных параметров. Для детального обсуждения обратитесь к хорошему руководству по написанию скриптов на sh(1). *Не используйте* эти выражения, пока полностью не поймёте их, так как их неправильное применение приведёт к созданию ненадёжных и

небезопасных скриптов.

В настоящее время в `run_rc_command` может присутствовать ошибка, которая мешает ему сохранять исходные границы между аргументами. То есть аргументы с встроенными пробелами могут обрабатываться некорректно. Ошибка возникает из-за неправильного использования `$*`.

9. Подготовка скрипта для сервисных клеток

Скрипты, запускающие долго работающую службу, подходят для служебных клеток и должны поставляться с соответствующей конфигурацией сервисной клетки.

Некоторые примеры скриптов, которые не подходят для запуска в сервисной клетке:

- любой скрипт, который в команде start только изменяет настройки времени выполнения для программ или ядра,
- или пытается что-то смонтировать,
- или находит и удаляет файлы

Необходимо предотвратить использование внутри сервисных клеток скриптов, не предназначенных для запуска в сервисной клетке.

Скрипт с долго работающей службой, которому необходимо выполнить одно из перечисленных выше действий перед запуском или после остановки, может быть разделён на два скрипта с зависимостями или использовать части `precommand` и `postcommand` скрипта для выполнения этого действия.

По умолчанию только части `start` и `stop` скрипта выполняются внутри сервисной клетки, остальное выполняется вне клетки. Таким образом, любые настройки, используемые в частях `start`/`stop` скрипта, не могут быть заданы, например, из `precommand`.

Чтобы сделать скрипт готовым к использованию с Сервисными Клетками, необходимо добавить всего лишь одну дополнительную конфигурационную строку:

```
#!/bin/sh

. /etc/rc.subr

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

: ${dummy_svcj_options:=""} ①

dummy_start()
{
```

```
        echo "Nothing started."
}

load_rc_config $name
run_rc_command "$1"
```

 Если имеет смысл, чтобы скрипт выполнялся в клетке, он должен иметь переопределяемую конфигурацию сервисных клеток. Если ему не требуется доступ к сети или любым другим ресурсам, которые ограничены в клетках, достаточно пустой конфигурации, как показано.

Строго говоря, пустая конфигурация не обязательна, но она явно указывает, что скрипт готов к работе с сервисными клетками и не требует дополнительных разрешений для клеток. Поэтому настоятельно рекомендуется добавить такую пустую конфигурацию в таком случае. Наиболее распространённая опция — "net_basic", которая позволяет использовать IPv4 и IPv6 адреса хоста. Все возможные опции описаны в rc.conf(5).

Если настройка запуска/остановки зависит от переменных из rc-фреймворка (например, заданных в rc.conf(5)), это должно обрабатываться с помощью `load_rc_config` и `run_rc_command`, а не внутри precommand.

Если по какой-то причине скрипт не может быть запущен внутри сервисной клетки, например, потому что его невозможно запустить или нет смысла запускать его в клетке, используйте следующее:

```
#!/bin/sh

. /etc/rc.subr

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

dummy_start()
{
        echo "Nothing started."
}

load_rc_config $name
dummy_svcj="NO"     # does not make sense to run in a svcj ①
run_rc_command "$1"
```

 Отключение должно происходить после вызова `load_rc_config`, иначе параметр из rc.conf(5) может переопределить его.

10. Продвинутые сценарии rc: запуск нескольких экземпляров

Иногда полезно запускать несколько экземпляров службы. Обычно требуется иметь возможность независимо запускать/останавливать такие экземпляры, а также иметь отдельный файл конфигурации для каждого из них. Каждый экземпляр должен запускаться при загрузке, после обновления каждый экземпляр должен оставаться, и при этом должен обновиться.

Вот пример rc-скрипта, который поддерживает это:

```
#!/bin/sh

#
# PROVIDE: dummy
# REQUIRE: NETWORKING SERVERS
# KEYWORD: shutdown
#
# Add these following line to /etc/rc.conf.local or /etc/rc.conf
# to enable this service:
#
# dummy_enable (bool):  Set it to YES to enable dummy on startup.
#           Default: NO
# dummy_user (string):  User account to run with.
#           Default: www
#

. /etc/rc.subr

case $0 in ①
/etc/rc*)
    # during boot (shutdown) $0 is /etc/rc (/etc/rc.shutdown),
    # so get the name of the script from $_file
    name=$_file
    ;;
*)
    name=$0
    ;;
esac

name=${name##*/} ②
rcvar="${name}_enable" ③
desc="Short description of this service"
command="/usr/local/sbin/dummy"

load_rc_config "$name"

eval "${rcvar}=\${${rcvar}:-'NO'}" ④
```

```
eval "${name}_svcj_options=\${${name}_svcj_options:-'net_basic'}" ⑤
eval "_dummy_user=\${${name}_user:-'www'}" ⑥

_dummy_configname=/usr/local/etc/${name}.cfg ⑦
pidfile=/var/run/dummy/${name}.pid
required_files ${_dummy_configname}
command_args="-u ${_dummy_user} -c ${_dummy_configfile} -p ${pidfile}"

run_rc_command "$1"
```

 и убедитесь, что переменная name установлена в значение basename(1) имени скрипта. Если имя файла — /usr/local/etc/rc.d/dummy, то name будет установлено в dummy. Таким образом, изменение имени rc-скрипта автоматически изменит содержимое переменной name.

 указывает имя переменной, которая используется в rc.conf для включения этой службы на основе имени файла этого скрипта. В данном примере это преобразуется в dummy_enable.

 убеждается, что значение по умолчанию для переменной _enable установлено в NO.

 Вот пример установки некоторых значений по умолчанию для переменных фреймворка, специфичных для службы, в данном случае — опций клетки службы.

 и устанавливают переменные, внутренние для скрипта (обратите внимание на подчёркивание в начале _dummy_user, чтобы отличать её от dummy_user, которая может быть задана в rc.conf).

Часть в предназначена для переменных, которые не используются внутри самого скрипта, но используются в рамках rc. Все переменные, которые используются как параметры в скрипте, присваиваются общей переменной, как в , чтобы упростить их использование (нет необходимости выполнять eval при каждом обращении).

Этот скрипт теперь будет вести себя по-другому, если скрипт запуска имеет другое имя. Это позволяет создавать символьные ссылки на него:

```
# ln -s dummy /usr/local/etc/rc.d/dummy_foo
# sysrc dummy_foo_enable=YES
# service dummy_foo start
```

Вышеприведённое создаёт экземпляр службы dummy с именем dummy_foo. Он использует не файл конфигурации /usr/local/etc/dummy.cfg, а файл конфигурации /usr/local/etc/dummy_foo.cfg (), и использует PID-файл /var/run/dummy/dummy_foo.pid вместо /var/run/dummy/dummy.pid.

Сервисы dummy и dummy_foo могут управляться независимо друг от друга, при этом скрипт запуска обновляется автоматически при обновлении пакета (благодаря символьной ссылке). Это не обновляет строку REQUIRE, поэтому нет простого способа зависеть от конкретного экземпляра. Чтобы зависеть от конкретного экземпляра в порядке запуска, необходимо создать копию вместо использования символьной ссылки. Это предотвращает

автоматическое применение изменений в скрипте запуска при установке обновления.

11. Дополнительная литература

Оригинальная статья Люка Мьюберна предлагает общий обзор rc.d и подробное обоснование принятых при его проектировании решений. В ней представлено понимание всего фреймворка rc.d и его места в современной BSD-системе.

Руководства rc(8), rc.subr(8) и rcorder(8) подробно описывают компоненты rc.d. Без изучения этих руководств и обращения к ним при написании собственных скриптов невозможно в полной мере использовать возможности rc.d.

Основным источником рабочих, жизненных примеров является /etc/rc.d в работающей системе. Его содержимое легко и приятно читать, поскольку большинство сложных моментов скрыто глубоко в rc.subr(8). Однако помните, что скрипты в /etc/rc.d были написаны не ангелами, поэтому они могут содержать ошибки и неоптимальные решения. Теперь вы можете их улучшить!