Scripting rc.d na prática no BSD

Resumo

Os iniciantes podem achar difícil relacionar os fatos da documentação formal sobre o framework rc.d do BSD com as tarefas práticas de escrever scripts rc.d. Neste artigo, consideramos alguns casos típicos de crescente complexidade, mostramos recursos do rc.d adequados para cada caso e discutimos como eles funcionam. Essa análise deve fornecer pontos de referência para estudos posteriores do design e aplicação eficiente do rc.d.

Índice

1. Introdução

No histórico BSD, havia um script de inicialização monolítico, /etc/rc. Ele era invocado pelo init(8) no momento da inicialização do sistema e realizava todas as tarefas de usuário necessárias para a operação multiusuário: verificação e montagem de sistemas de arquivos, configuração da rede, inicialização de daemons e assim por diante. A lista precisa de tarefas não era a mesma em todos os sistemas; os administradores precisavam personalizá-la. Com poucas exceções, o /etc/rc tinha que ser modificado, e os verdadeiros hackers gostavam disso.

O problema real com a abordagem monolítica era que ela não fornecia controle sobre os componentes individuais iniciados a partir do /etc/rc. Por exemplo, o /etc/rc não podia reiniciar um único daemon. O administrador do sistema tinha que encontrar o processo do daemon manualmente, matá-lo, aguardar até que ele realmente finalizasse, navegar por /etc/rc em busca das flags e, finalmente, digitar a linha de comando completa para iniciar o daemon novamente. A tarefa se tornaria ainda mais difícil e propensa a erros se o serviço a ser reiniciado consistisse em mais de um daemon ou exigisse ações adicionais. Em poucas palavras, o script único falhou em cumprir o objetivo pelo qual um script é criado: tornar a vida do administrador do sistema mais fácil.

Mais tarde, houve uma tentativa de dividir algumas partes do /etc/rc para iniciar os subsistemas

mais importantes separadamente. O exemplo notório foi o /etc/netstart para iniciar a rede. Isso permitiu o acesso à rede no modo de usuário único, mas não se integrou bem ao processo de inicialização automática porque partes de seu código precisavam intercalar com ações essencialmente não relacionadas à rede. Foi por isso que o /etc/netstart se transformou em /etc/rc.network. Este último não era mais um script comum; era composto de grandes funções sh(1) complexas chamadas pelo /etc/rc em diferentes estágios da inicialização do sistema. No entanto, à medida que as tarefas de inicialização ficaram mais diversas e sofisticadas, a abordagem "quase modular" tornou-se ainda mais pesada do que o monolítico /etc/rc tinha sido.

Sem um framework limpo e bem projetado, os scripts de inicialização tiveram que se dobrar para atender às necessidades dos sistemas operacionais baseados em BSD que estavam em rápida evolução. Tornou-se evidente, finalmente, que mais etapas eram necessárias para se chegar a um sistema rc refinado, granular e extensível. Assim nasceu o rc.d do BSD. Seus pais reconhecidos foram Luke Mewburn e a comunidade NetBSD. Mais tarde, foi importado para o FreeBSD. Seu nome se refere ao local dos scripts do sistema para serviços individuais, que está em /etc/rc.d. Em breve, aprenderemos mais sobre os componentes do sistema rc.d e veremos como os scripts individuais são invocados.

As ideias básicas por trás do rc.d do BSD são *modularidade fina* e *reutilização de código*. *Modularidade fina* significa que cada "serviço" básico, como um daemon do sistema ou uma tarefa de inicialização primitiva, possui seu próprio script sh(1) capaz de iniciar o serviço, pará-lo, recarregá-lo e verificar seu status. Uma ação específica é escolhida pelo argumento da linha de comando do script. O script /etc/rc ainda conduz a inicialização do sistema, mas agora ele apenas invoca os scripts menores um por um com o argumento `start`. Também é fácil realizar tarefas de desligamento executando o mesmo conjunto de scripts com o argumento `stop`, que é feito por /etc/rc.shutdown. Observe como isso segue de perto a maneira Unix de ter um conjunto de ferramentas especializadas pequenas, cada uma cumprindo sua tarefa da melhor maneira possível. *Reutilização de código* significa que operações comuns são implementadas como funções sh(1) e coletadas em /etc/rc.subr. Agora, um script típico pode ser composto apenas de algumas linhas de código sh(1). Finalmente, uma parte importante do framework rc.d é o rcorder(8), que ajuda o /etc/rc a executar os pequenos scripts de maneira ordenada com respeito às dependências entre eles. Isso também pode ajudar o /etc/rc.shutdown, porque a ordem apropriada para a sequência de desligamento é oposta à de inicialização.

O design do BSD rc.d é descrito no artigo original de Luke Mewburn, e os componentes do rc.d são documentados em grande detalhe nas respectivas páginas do manual. No entanto, pode não ser óbvio para um iniciante no rc.d como ele deve unir as numerosas partes para criar um script bem estruturado para uma tarefa específica. Portanto, este artigo tentará abordar o rc.d de forma diferente. Mostrará quais recursos devem ser usados em vários casos típicos e por que. Observe que este não é um documento de "como fazer", porque nosso objetivo não é fornecer receitas prontas, mas mostrar algumas entradas fáceis no mundo do rc.d. Este artigo também não substitui as páginas do manual relevantes. Não hesite em consultá-las para obter documentação mais formal e completa enquanto lê este artigo.

Existem pré-requisitos para entender este artigo. Em primeiro lugar, você deve estar familiarizado com a linguagem de script sh(1) para dominar o rc.d. Além disso, você deve saber como o sistema realiza tarefas de inicialização e desligamento do espaço do usuário, o que é descrito em rc(8).

Este artigo foca no ramo do FreeBSD do rc.d. No entanto, ele pode ser útil também para

desenvolvedores do NetBSD, pois os dois ramos do rc.d não apenas compartilham o mesmo design, mas também permanecem similares em seus aspectos visíveis aos autores de scripts.

2. Esboçando a tarefa

Uma pequena reflexão antes de iniciar o `$EDITOR` não fará mal. Para escrever um script rc.d bem elaborado para um serviço do sistema, devemos ser capazes de responder às seguintes perguntas primeiro:

- O serviço é obrigatório ou opcional?
- O script servirá um único programa, por exemplo, um daemon, ou realizará ações mais complexas?
- De quais outros serviços nosso serviço dependerá e vice-versa?

A partir dos exemplos que se seguem, veremos o porque é importante conhecer as respostas a essas perguntas.

3. Um script fictício

O script a seguir apenas emite uma mensagem toda vez que o sistema é inicializado:

```
#!/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
```

Os pontos a serem observados são:

 Um script interpretado deve começar com a linha mágica "shebang". Essa linha especifica o programa interpretador para o script. Devido à linha shebang, o script pode ser invocado exatamente como um programa binário, desde que tenha o bit de execução definido. (Veja chmod(1).) Por exemplo, um administrador do sistema pode executar nosso script manualmente, a partir da linha de comando:

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

Para ser gerenciado corretamente pelo framework rc.d, os scripts devem ser escritos na linguagem sh(1). Se você tiver um serviço ou port que usa um utilitário de controle binário ou uma rotina de inicialização escrita em outra linguagem, instale esse elemento em /usr/sbin (para o sistema) ou /usr/local/sbin (para ports) e chame-o a partir de um script sh(1) no diretório rc.d apropriado.

Se você gostaria de aprender os detalhes de por que os scripts rc.d devem ser escritos na linguagem sh(1), veja como o /etc/rc os invoca por meio de `run_rc_script` e, em seguida, estude a implementação de `run_rc_script` em /etc/rc.subr.

 Em /etc/rc.subr, uma série de funções sh(1) estão definidas para serem utilizadas por um script rc.d. As funções estão documentadas em rc.subr(8). Embora seja teoricamente possível escrever um script rc.d sem nunca usar o rc.subr(8), suas funções provam ser extremamente úteis e tornam o trabalho uma ordem de magnitude mais fácil. Portanto, não é surpresa que todo mundo recorra ao rc.subr(8) em scripts rc.d. Não seremos uma exceção.

Um script rc.d deve fazer o "source" do /etc/rc.subr (inclua-o usando “.”) *antes* de chamar as funções rc.subr(8) para que o sh(1) tenha a oportunidade de aprender as funções. O estilo preferido é incluir o /etc/rc.subr antes de tudo.

Algumas funções úteis relacionadas à rede são fornecidas por outro arquivo de inclusão, o /etc/network.subr.

 A variável obrigatória `name` especifica o nome do nosso script. Ela é exigida pelo rc.subr(8). Isso significa que cada script rc.d *deve* definir `name` antes de chamar funções do rc.subr(8).

Agora é o momento certo para escolher um nome exclusivo para o nosso script de uma vez por todas. Vamos usá-lo em vários lugares enquanto desenvolvemos o script. Para começar, também vamos dar o mesmo nome ao arquivo de script.

O estilo atual de escrita de scripts rc.d é envolver os valores atribuídos às variáveis em aspas duplas. Tenha em mente que isso é apenas uma questão de estilo que nem sempre é aplicável. Você pode seguramente omitir as aspas ao redor de palavras simples sem metacaracteres sh(1), enquanto em certos casos você precisará de aspas simples para evitar qualquer interpretação do valor por sh(1). Um programador deve ser capaz de distinguir a sintaxe da linguagem das convenções de estilo e usá-las sabiamente.

 Cada script rc.d fornece manipuladores, ou métodos, para o rc.subr(8) invocar. Em particular, `start`, `stop`, e outros argumentos para um script rc.d são manipulados desta forma. Um método é uma expressão sh(1) armazenada em uma variável chamada `argument_cmd`, onde *argument* corresponde ao que pode ser especificado na linha de comando do script. Veremos mais tarde como o rc.subr(8) fornece métodos padrão para os argumentos padrão.

Para tornar o código em rc.d mais uniforme, é comum usar `${name}` sempre que apropriado. Assim, várias linhas podem ser simplesmente copiadas de um script para outro.

 Devemos ter em mente que o rc.subr(8) fornece métodos padrões para os argumentos padrões. Consequentemente, devemos substituir um método padrão por uma expressão sh(1) sem efeito se quisermos que ele não faça nada.

 O corpo de um método sofisticado pode ser implementado como uma função. É uma boa ideia dar um nome significativo à função.

Recomenda-se fortemente adicionar o prefixo `${name}` aos nomes de todas as funções definidas no nosso script para que nunca entrem em conflito com as funções de rc.subr(8) ou outro arquivo de inclusão comum.

 Essa chamada para o rc.subr(8) carrega as variáveis do rc.conf(5). Nosso script ainda não as usa, mas ainda é recomendável carregar o rc.conf(5) porque pode haver variáveis do rc.conf(5) controlando o rc.subr(8) em si.

 Geralmente, esse é o último comando em um script rc.d. Ele invoca a maquinaria do rc.subr(8) para realizar a ação solicitada usando as variáveis e métodos que o nosso script forneceu.

4. Um script fictício configurável

Agora vamos adicionar alguns controles ao nosso script fictício. Como você deve saber, scripts rc.d são controlados com o rc.conf(5). Felizmente, o rc.subr(8) oculta todas as complicações para nós. O script a seguir usa o rc.conf(5) por meio do rc.subr(8) para verificar se está habilitado em primeiro lugar e para buscar uma mensagem para ser exibida na inicialização. Essas duas tarefas, na verdade, são independentes. Por um lado, um script rc.d pode apenas suportar a habilitação e desabilitação do seu serviço. Por outro lado, um script rc.d obrigatório pode ter variáveis de configuração. No entanto, faremos as duas coisas no mesmo script:

```
#!/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"
```

O que mudou neste exemplo?

① A variável `rcvar` especifica o nome da variável do botão LIGA/DESLIGA.

② Agora, o `load_rc_config` é invocado mais cedo no script, antes que quaisquer variáveis do rc.conf(5) sejam acessadas.

Enquanto examina os scripts rc.d, tenha em mente que o sh(1) adia a avaliação de expressões em uma função até que esta seja chamada. Portanto, não é um erro invocar `load_rc_config` tão tarde quanto logo antes de `run_rc_command` e ainda assim acessar as variáveis rc.conf(5) das funções de método exportadas para `run_rc_command`. Isso ocorre porque as funções de método devem ser chamadas por `run_rc_command`, que é invocado *após* `load_rc_config`.

③ Aviso será emitido pelo `run_rc_command` se o `rcvar` em si estiver configurado, mas a variável de controle indicada estiver desativada. Se o seu script rc.d é para o sistema base, você deve adicionar uma configuração padrão para o knob em /etc/defaults/rc.conf e documentá-lo em rc.conf(5). Caso contrário, é seu script que deve fornecer uma configuração padrão para o knob. A abordagem canônica para o último caso é mostrada no exemplo.

Você pode fazer o rc.subr(8) agir como se o botão estivesse definido como `ON`, independentemente de sua configuração atual, prefixando o argumento do script com `one` ou `force`, como em `onestart` ou `forcestop`. No entanto, tenha em mente que `force` tem outros efeitos perigosos que abordaremos abaixo, enquanto `one` apenas substitui o botão ON/OFF. Por exemplo, suponha que `dummy_enable` esteja definido como `OFF`. O seguinte comando executará o método `start` apesar da configuração:

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

④ Agora a mensagem a ser exibida na inicialização não é mais codificada no script. É especificada por uma variável rc.conf(5) chamada `dummy_msg`. Este é um exemplo trivial de como as variáveis rc.conf(5) podem controlar um script rc.d.

Os nomes de todas as variáveis rc.conf(5) usadas exclusivamente pelo nosso script *devem* ter o mesmo prefixo: `${name}_`. Por exemplo: `dummy_mode`, `dummy_state_file`, e assim por diante.

Embora seja possível usar um nome mais curto internamente, por exemplo, apenas `msg`, adicionar o prefixo único `${name}_` a todos os nomes globais introduzidos pelo nosso script nos salvará de possíveis colisões com o namespace do rc.subr(8).

> Como regra geral, os scripts rc.d do sistema base não precisam fornecer valores padrão para suas variáveis rc.conf(5), pois os valores padrão devem ser definidos em /etc/defaults/rc.conf. Por outro lado, os scripts rc.d para ports devem fornecer os valores padrão conforme mostrado no exemplo.

 Aqui usamos `dummy_msg` para controlar nosso script, ou seja, para emitir uma mensagem variável. O uso de uma função shell é excessivo aqui, uma vez que ela executa apenas um único comando; uma alternativa igualmente válida é:

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

5. Inicialização e desligamento de um daemon simples

Dissemos anteriormente que o rc.subr(8) pode fornecer métodos padrões. Obviamente, tais padrões não podem ser muito gerais. Eles são adequados para o caso comum de iniciar e desligar um programa de daemon simples. Vamos supor agora que precisamos escrever um script rc.d para um daemon chamado `mumbled`. Aqui está:

```
#!/bin/sh

. /etc/rc.subr

name=mumbled
rcvar=mumbled_enable

command="/usr/sbin/${name}" ①

load_rc_config $name
run_rc_command "$1"
```

Agradavelmente simples, não é? Vamos examinar nosso pequeno script. A única coisa nova a observar é o seguinte:

 A variável `command` é significativa para rc.subr(8). Se ela estiver definida, rc.subr(8) agirá de acordo com o cenário de servir a um daemon convencional. Em particular, os métodos padrão serão fornecidos para esses argumentos: `start`, `stop`, `restart`, `poll` e `status`.

O daemon será iniciado executando `$command` com as flags de linha de comando especificadas por `$mumbled_flags`. Assim, todos os dados de entrada para o método `start` padrão estão disponíveis nas variáveis definidas pelo nosso script. Ao contrário de `start`, outros métodos podem exigir informações adicionais sobre o processo iniciado. Por exemplo, `stop` deve saber o PID do processo para terminá-lo. No caso presente, rc.subr(8) irá pesquisar a lista de todos os processos, procurando por um processo com o nome igual a `procname`. Este último é outra variável com significado para rc.subr(8), e seu valor padrão é o de `command`. Em outras palavras, quando definimos `command`,

`procname` é efetivamente definido para o mesmo valor. Isso permite que nosso script mate o daemon e verifique se ele está em execução.

Algumas vezes, programas são de fato scripts executáveis. O sistema executa esse script iniciando o seu interpretador e passando o nome do script como um argumento na linha de comando. Isso é refletido na lista de processos, o que pode confundir rc.subr(8). Você deve adicionalmente definir `command_interpreter` para que rc.subr(8) saiba o nome real do processo se `$command` for um script.

A cada script rc.d, há uma variável opcional do rc.conf(5) que tem precedência sobre `command`. Seu nome é construído da seguinte forma: `${name}_program`, onde `name` é a variável obrigatória discutida anteriormente. Por exemplo, neste caso, será `mumbled_program`. É o rc.subr(8) que arruma `${name}_program` para substituir `command`.

Claro que o sh(1) permite definir `${name}_program` a partir do rc.conf(5) ou do próprio script, mesmo se `command` não estiver definido. Nesse caso, as propriedades especiais de `${name}_program` são perdidas, e ela se torna uma variável comum que o script pode usar para seus próprios fins. No entanto, o uso exclusivo de `${name}_program` é desencorajado, pois usá-la em conjunto com `command` se tornou idiomático em rc.d scripting.

Para obter informações mais detalhadas sobre os métodos padrões, consulte rc.subr(8).

6. Inicialização e desligamento de um daemon avançado

Vamos adicionar um pouco de carne aos ossos do script anterior e torná-lo mais complexo e cheio de funcionalidades. Os métodos padrões podem fazer um bom trabalho para nós, mas podemos precisar ajustar alguns dos seus aspectos. Agora vamos aprender como ajustar os métodos padrões para as nossas necessidades.

```
#!/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"
```

 Argumentos adicionais para `$command` podem ser passados em `command_args`. Eles serão adicionados à linha de comando após `$mumbled_flags`. Como a linha de comando final é passada para `eval` para sua execução real, redirecionamentos de entrada e saída podem ser especificados em `command_args`.

Nunca inclua opções com hífen, como `-X` ou `--foo`, em `command_args`. O conteúdo de `command_args` aparecerá no final da linha de comando final, portanto, é provável que sigam argumentos presentes em `${name}_flags`; mas a maioria dos comandos não reconhecerá opções com hífen após argumentos comuns. Uma maneira melhor de passar opções adicionais para `$command` é adicioná-las no início de `${name}_flags`. Outra maneira é modificar `rc_flags` conforme mostrado posteriormente.

 Um daemon bem comportado deve criar um *pidfile* para que seu processo possa ser encontrado de forma mais fácil e confiável. A variável `pidfile`, se definida, informa ao rc.subr(8) onde ele pode encontrar o pidfile para ser utilizado em seus métodos padrões.

De fato, o rc.subr(8) também usa o pidfile para ver se o daemon já está em execução antes de iniciá-lo. Essa verificação pode ser ignorada usando o argumento `faststart`.

 Se o daemon não puder ser executado a menos que certos arquivos existam, basta listá-los em `required_files`, e o rc.subr(8) verificará se esses arquivos existem antes de iniciar o daemon. Existem também `required_dirs` e `required_vars` para diretórios e variáveis de ambiente, respectivamente. Todos eles são descritos em detalhes no rc.subr(8).

O método padrão do rc.subr(8) pode ser forçado a pular as verificações de pré-requisito usando `forcestart` como argumento para o script.

 Podemos personalizar sinais a serem enviados ao daemon caso eles difiram dos sinais conhecidos. Em particular, `sig_reload` especifica o sinal que faz com que o daemon recarregue sua configuração; por padrão, é o SIGHUP. Outro sinal é enviado para interromper o processo do daemon; o padrão é o SIGTERM, mas isso pode ser alterado configurando `sig_stop` adequadamente.

As sinalizações devem ser especificadas para o rc.subr(8) sem o prefixo `SIG`, como é mostrado no exemplo. A versão do FreeBSD do kill(1) pode reconhecer o prefixo `SIG`, mas as versões de outros sistemas operacionais podem não reconhecer.

 Realizar tarefas adicionais antes ou depois dos métodos padrões é fácil. Para cada argumento de comando suportado por nosso script, podemos definir `argument_precmd` e `argument_postcmd`. Esses comandos sh(1) são invocados antes e depois do respectivo método, como é evidente pelos seus nomes.

Sobrescrever um método padrão com um `argument_cmd` personalizado ainda não nos impede de usar `argument_precmd` ou `argument_postcmd` se precisarmos. Em particular, o primeiro é bom para verificar condições personalizadas e sofisticadas que devem ser atendidas antes de executar o próprio comando. Usar `argument_precmd` juntamente com `argument_cmd` nos permite separar logicamente as verificações da ação.

Não se esqueça de que você pode colocar qualquer expressão válida do sh(1) nos métodos, pre e pós comandos que você define. Invocar uma função que realiza o trabalho real é um bom estilo na maioria dos casos, mas nunca deixe o estilo limitar sua compreensão do que está acontecendo nos bastidores.

 Se quisermos implementar argumentos personalizados, que também podem ser considerados como *comandos* para o nosso script, precisamos listá-los em `extra_commands` e fornecer métodos para lidar com eles.

O comando `reload` é especial. Por um lado, ele possui um método predefinido em rc.subr(8). Por outro lado, `reload` não é oferecido por padrão. A razão é que nem

todos os daemons usam o mesmo mecanismo de recarga e alguns não têm nada para recarregar. Portanto, precisamos pedir explicitamente que a funcionalidade integrada seja fornecida. Podemos fazer isso por meio de `extra_commands`.

O que recebemos do método padrão para `reload`? Muitas vezes, os daemons recarregam sua configuração após receber um sinal - geralmente, SIGHUP. Portanto, o rc.subr(8) tenta recarregar o daemon enviando um sinal para ele. O sinal é pré-definido como SIGHUP, mas pode ser personalizado via `sig_reload`, se necessário.

O nosso script suporta dois comandos não padrão, `plugh` e `xyzzy`. Nós os vimos listados em `extra_commands`, e agora é hora de fornecer métodos para eles. O método para `xyzzy` é apenas inserido em linha enquanto que para `plugh` é implementado como a função `mumbled_plugh`.

Comandos não padrão não são invocados durante a inicialização ou desligamento. Geralmente, eles são para a conveniência do administrador do sistema. Eles também podem ser usados em outros subsistemas, por exemplo, o devd(8) se especificados em devd.conf(5).

A lista completa de comandos disponíveis pode ser encontrada na linha de uso impressa pelo rc.subr(8) quando o script é invocado sem argumentos. Por exemplo, aqui está a linha de uso do script em estudo:

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

Um script pode invocar seus próprios comandos padrão ou não-padrão, se necessário. Isso pode parecer semelhante a chamar funções, mas sabemos que comandos e funções do shell nem sempre são a mesma coisa. Por exemplo, `xyzzy` não é implementado como uma função aqui. Além disso, pode haver um pré-comando e um pós-comando, que devem ser invocados ordenadamente. Portanto, a maneira adequada para um script executar seu próprio comando é por meio do rc.subr(8), como mostrado no exemplo.

Uma função útil chamada `checkyesno` é fornecida pelo rc.subr(8). Ela recebe um nome de variável como argumento e retorna um código de saída zero se e somente se a variável estiver definida como `YES`, ou `TRUE`, ou `ON`, ou `1`, insensível a maiúsculas e minúsculas; um código de saída não-zero é retornado caso contrário. Neste último caso, a função testa se a variável está definida como `NO`, `FALSE`, `OFF` ou `0`, insensível a maiúsculas e minúsculas; ela imprime uma mensagem de aviso se a variável contiver qualquer outra coisa, ou seja, lixo.

Tenha em mente que para o sh(1), um código de saída zero significa verdadeiro e um código de saída não-zero significa falso.

A função `checkyesno` recebe um *nome de variável*. Não passe o *valor expandido* de uma variável para ela; isso não funcionará como esperado.

Aqui está o uso correto de `checkyesno`:

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

Ao contrário, chamar `checkyesno` como mostrado abaixo não funcionará - pelo menos não como esperado:

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

 Podemos afetar as flags que serão passadas para `$command` modificando `rc_flags` em `$start_precmd`.

 Em certos casos, podemos precisar emitir uma mensagem importante que também deve ser registrada no `syslog`. Isso pode ser feito facilmente com as seguintes funções do rc.subr(8): `debug`, `info`, `warn` e `err`. Esta última função finaliza o script com o código especificado.

 Os códigos de saída dos métodos e seus pre-comandos não são apenas ignorados por padrão. Se `argument_precmd` retornar um código de saída diferente de zero, o método principal não será executado. Por sua vez, `argument_postcmd` não será chamado, a menos que o método principal retorne um código de saída igual a zero.

Entretanto, é possível instruir o rc.subr(8) a ignorar esses códigos de saída e executar todos os comandos mesmo assim, adicionando um argumento com o prefixo `force`, como em `forcestart`.

7. Conectando um script ao framework rc.d

Depois que um script é escrito, ele precisa ser integrado ao rc.d. O passo crucial é instalar o script em /etc/rc.d (para o sistema base) ou /usr/local/etc/rc.d (para o ports). Tanto o bsd.prog.mk quanto o bsd.port.mk fornecem ganchos convenientes para isso, e geralmente você não precisa se preocupar com a propriedade e o modo adequados. Os scripts do sistema devem ser instalados a partir de src/libexec/rc/rc.d através do Makefile encontrado lá. Scripts de ports podem ser instalados usando `USE_RC_SUBR` como descrito no Porter's Handbook.

No entanto, devemos considerar antecipadamente o local do nosso script na sequência de inicialização do sistema. O serviço manipulado pelo nosso script provavelmente depende de outros serviços. Por exemplo, um daemon de rede não pode funcionar sem as interfaces de rede e o roteamento em funcionamento. Mesmo que um serviço pareça não exigir nada, dificilmente pode ser iniciado antes que os sistemas de arquivos básicos tenham sido verificados e montados.

Nós já mencionamos o rcorder(8). Agora é hora de dar uma olhada mais de perto nele. Em poucas palavras, o rcorder(8) pega um conjunto de arquivos, examina seus conteúdos e imprime uma lista ordenada por dependência dos arquivos do conjunto no `stdout`. O objetivo é manter as informações de dependência *dentro* dos arquivos, de modo que cada arquivo possa falar apenas por si mesmo. Um arquivo pode especificar as seguintes informações:

- os nomes das "condições" (ou seja, serviços para nós) que ele *fornece*;
- os nomes das "condições" que ele *requer*;
- os nomes das "condições" que este arquivo deve ser executado *antes*;
- palavras-chave adicionais que podem ser usadas para selecionar um subconjunto do conjunto completo de arquivos (rcorder(8) pode ser instruído via opções para incluir ou omitir os arquivos que possuem determinadas palavras-chave listadas.)

Não é surpresa que o rcorder(8) possa lidar apenas com arquivos de texto com uma sintaxe próxima à do sh(1). Ou seja, as linhas especiais entendidas pelo rcorder(8) se parecem com comentários do sh(1). A sintaxe dessas linhas especiais é bastante rígida para simplificar seu processamento. Consulte rcorder(8) para obter detalhes.

Além de usar as linhas especiais entendidas pelo rcorder(8), um script pode exigir sua dependência de outro serviço simplesmente iniciando-o forçadamente. Isso pode ser necessário quando o outro serviço é opcional e não iniciará por si só porque o administrador do sistema o desativou por engano em rc.conf(5).

Com este conhecimento geral em mente, vamos considerar o simples script daemon aprimorado com coisas de dependência:

```
#!/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"
```

Como antes, a análise detalhada segue:

 Essa linha declara os nomes das "condições" que nosso script fornece. Agora, outros scripts podem registrar uma dependência em nosso script por esses nomes.

Geralmente, um script especifica uma única condição fornecida. No entanto, nada nos impede de listar várias condições, por exemplo, por razões de compatibilidade.

Em qualquer caso, o nome da condição principal, ou única, `PROVIDE:` deve ser o mesmo que `${name}`.

 Então, nosso script indica em quais "condições" fornecidas por outros scripts ele depende. De acordo com as linhas, nosso script pede para o rcorder(8) colocá-lo após o(s) script(s) fornecendo o DAEMON e o cleanvar, mas antes daquele que fornece LOGIN.

A linha `BEFORE:` não deve ser usada para contornar uma lista de dependências incompleta em outro script. O caso apropriado para usar `BEFORE:` é quando o outro script não se importa com o nosso, mas nosso script pode executar sua tarefa melhor se for executado antes do outro. Um exemplo típico da vida real é a relação entre as interfaces de rede e o firewall: embora as interfaces não dependam do firewall para fazer o trabalho delas, a segurança do sistema se beneficiará se o firewall estiver pronto antes de haver qualquer tráfego de rede.

Além das condições correspondentes a um único serviço, existem meta-condições e seus scripts "placeholder" usados para garantir que certos grupos de operações sejam executados antes de outros. Estes são denotados por nomes em UPPERCASE. Sua lista e propósitos podem ser encontrados no manual rc(8).

Lembre-se de que colocar o nome de um serviço na linha `REQUIRE:` não garante que o serviço realmente estará em execução no momento em que o script começar. O serviço necessário pode falhar ao iniciar ou simplesmente ser desativado em rc.conf(5). Obviamente, o rcorder(8) não pode controlar esses detalhes e o rc(8) também não fará isso. Consequentemente, a aplicação iniciada pelo nosso script deve ser capaz de lidar com qualquer serviço necessário que esteja indisponível. Em certos casos, podemos ajudá-lo conforme discutido abaixo

 Como lembramos do texto acima, as palavras-chave do rcorder(8) podem ser usadas para selecionar ou deixar de fora alguns scripts. Qualquer consumidor do rcorder(8) pode especificar, através das opções `-k` e `-s`, quais palavras-chave estarão na "lista de manutenção" e na "lista de exclusão", respectivamente. De todos os arquivos a serem ordenados por dependência, rcorder(8) escolherá apenas aqueles que tiverem uma palavra-chave da lista de manutenção (a menos que esteja vazia) e que não tiverem uma palavra-chave da lista de exclusão.

No FreeBSD, o rcorder(8) é usado por /etc/rc e /etc/rc.shutdown. Esses dois scripts definem a lista padrão de palavras-chaves do FreeBSD rc.d e seus significados da seguinte forma:

nojail

O serviço não é para ambiente jail(8). Os procedimentos automáticos de inicialização e desligamento ignorarão o script se estiver dentro de uma jail.

nostart

O serviço deve ser iniciado manualmente ou não iniciado de forma alguma. O procedimento de inicialização automático ignorará o script. Em conjunto com a palavra-chave shutdown, isso pode ser usado para escrever scripts que fazem algo apenas no desligamento do sistema.

shutdown

Este palavra-chave deve ser listada de forma *explícita* se o serviço precisa ser parado antes do desligamento do sistema.

Quando o sistema está prestes a desligar, o arquivo /etc/rc.shutdown é executado. Ele assume que a maioria dos scripts rc.d não tem nada a fazer nesse momento. Portanto, o /etc/rc.shutdown invoca seletivamente scripts rc.d com a palavra-chave shutdown, ignorando efetivamente o restante dos scripts. Para desligamento ainda mais rápido, o /etc/rc.shutdown passa o comando faststop para os scripts que executa para que eles pulem verificações preliminares, como a verificação do pidfile. Como os serviços dependentes devem ser interrompidos antes de suas dependências, o /etc/rc.shutdown executa os scripts em ordem de dependência reversa. Se você está escrevendo um script rc.d real, deve considerar se ele é relevante no momento do desligamento do sistema. Por exemplo, se o seu script faz seu trabalho apenas em resposta ao comando start, então você não precisa incluir essa palavra-chave. No entanto, se o seu script gerencia um serviço, é provavelmente uma boa ideia pará-lo antes que o sistema prossiga para o estágio final de sua sequência de desligamento descrita em halt(8). Em particular, um serviço deve ser interrompido explicitamente se precisar de tempo considerável ou ações especiais para ser desligado corretamente. Um exemplo típico desse tipo de serviço é um mecanismo de banco de dados.

 Em primeiro lugar, `force_depend` deve ser usado com muito cuidado. Geralmente, é melhor revisar a hierarquia das variáveis de configuração para seus scripts rc.d se eles são interdependentes.

Se ainda assim você não puder abrir mão do `force_depend`, o exemplo oferece um exemplo de como invocá-lo condicionalmente. No exemplo, nosso daemon `mumbled` requer que outro daemon, `frotz`, seja iniciado antecipadamente. No entanto, `frotz` também é opcional; e o rcorder(8) não conhece esses detalhes. Felizmente, nosso script tem acesso a todas as variáveis de rc.conf(5). Se `frotz_enable` for verdadeiro, esperamos o melhor e confiamos no rc.d para ter iniciado `frotz`. Caso contrário, verificamos forçadamente o status de `frotz`. Finalmente, impomos nossa dependência em `frotz` se ele não estiver em execução. Uma mensagem de aviso será emitida por `force_depend`, pois ele só deve ser invocado se uma configuração incorreta for detectada.

8. Dando mais flexibilidade a um script rc.d

Quando invocado durante a inicialização ou desligamento, um script rc.d deve agir em todo o subsistema pelo qual é responsável. Por exemplo, o /etc/rc.d/netif deve iniciar ou parar todas as interfaces de rede descritas em rc.conf(5). Cada tarefa pode ser indicada por um único argumento de comando, como `start` ou `stop`. Entre a inicialização e o desligamento, os scripts rc.d ajudam o administrador a controlar o sistema em execução e é quando surge a necessidade de mais flexibilidade e precisão. Por exemplo, o administrador pode querer adicionar as configurações de

uma nova interface de rede em rc.conf(5) e, em seguida, iniciá-la sem interferir na operação das interfaces existentes. Na próxima vez, o administrador pode precisar desligar uma única interface de rede. Para facilitar o uso na linha de comando, o respectivo script rc.d pede um argumento extra, o nome da interface.

Felizmente, o rc.subr(8) permite passar qualquer número de argumentos para os métodos do script (dentro dos limites do sistema). Devido a isso, as mudanças no próprio script podem ser mínimas.

Como o rc.subr(8) pode ter acesso aos argumentos adicionais da linha de comando? Ele deve simplesmente pegá-los diretamente? De maneira alguma! Em primeiro lugar, uma função do sh(1) não tem acesso aos parâmetros posicionais de quem a chamou, mas o rc.subr(8) é apenas um conjunto dessas funções. Em segundo lugar, a boa prática do rc.d dita que é responsabilidade do script principal decidir quais argumentos devem ser passados para seus métodos.

Portanto, a abordagem adotada pelo rc.subr(8) é a seguinte: `run_rc_command` passa todos os seus argumentos, exceto o primeiro, ao respectivo método sem alterações. O primeiro argumento omitido é o nome do método em si: `start`, `stop`, etc. Ele será removido por `run_rc_command`, então o que é `$2` na linha de comando original será apresentado como `$1` para o método, e assim por diante.

Para ilustrar essa oportunidade, vamos modificar o script fictício primitivo para que suas mensagens dependam dos argumentos adicionais fornecidos. Aqui vamos nós:

```
#!/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
```

Quais mudanças essenciais podemos notar no script?

 Todos os argumentos que você digita após `start` podem acabar como parâmetros posicionais para o respectivo método. Podemos usá-los de qualquer maneira de acordo com nossa tarefa, habilidades e gosto. No exemplo atual, simplesmente passamos todos eles para o echo(1) como uma única string na próxima linha - observe o `$*` dentro das aspas duplas. Aqui está como o script pode ser invocado agora:

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

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

 O mesmo se aplica a qualquer método que nosso script ofereça, não apenas a um padrão. Adicionamos um método personalizado chamado `kiss`, e ele pode tirar proveito dos argumentos extras da mesma forma que o `start` pode. Por exemplo:

```
# /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...
```

 Se quisermos apenas passar todos os argumentos extras para qualquer método, podemos simplesmente substituir `"$1"` por `"$@"` na última linha do nosso script, onde invocamos `run_rc_command`.

Um programador em sh(1) deve entender a diferença sutil entre `$*` e `$@` como formas de designar todos os parâmetros posicionais. Para uma discussão aprofundada, consulte um bom manual de sh(1). Não use essas expressões até entender completamente o seu uso, pois o uso incorreto pode resultar em scripts com bugs e inseguros.

Atualmente, o `run_rc_command` pode ter um bug que o impede de manter as fronteiras originais entre os argumentos. Ou seja, argumentos com espaços em

> branco embutidos podem não ser processados corretamente. O bug decorre do uso inadequado de `$*`.

9. Leitura adicional

O artigo original de Luke Mewburn oferece uma visão geral do rc.d e uma justificativa detalhada para suas decisões de design. Ele fornece uma compreensão do quadro geral do rc.d e seu lugar em um sistema operacional BSD moderno.

As páginas do manual para rc(8), rc.subr(8) e rcorder(8) documentam em detalhes os componentes do sistema rc.d. Você não pode aproveitar completamente o poder do rc.d sem estudar as páginas do manual e consultá-las ao escrever seus próprios scripts.

A principal fonte de exemplos práticos e funcionais é o diretório /etc/rc.d em um sistema em operação. O seu conteúdo é fácil e agradável de ler, pois a maioria das partes difíceis está escondida profundamente em rc.subr(8). No entanto, tenha em mente que os scripts em /etc/rc.d não foram escritos por anjos, então eles podem conter bugs e decisões de design subótimas. Agora você pode melhorá-los!