Scripts rc.d práticos no BSD

Resumo

Os iniciantes podem achar difícil relacionar os fatos da documentação formal do framework rc.d do BSD com as tarefas práticas do script rc.d. Neste artigo, consideramos alguns casos típicos de complexidade crescente, vamos mostrar os recursos do rc.d adequados para cada caso e vamos discutir como eles funcionam. Esse exame deve fornecer pontos de referência para um estudo mais aprofundado do design e da aplicação eficiente do rc.d.

Índice

1. Introdução . . . 1
2. Esboçando a tarefa . . . 3
3. Um script fictício . . . 3
4. Um script fictício configurável . . . 5
5. Inicialização e desligamento de um daemon simples . . . 7
6. Inicialização e desligamento de um daemon avançado . . . 9
7. Conectando um script ao framework rc.d . . . 13
8. Dando mais flexibilidade a um script rc.d . . . 15
9. Leitura adicional . . . 18

1. Introdução

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

O problema real com a abordagem monolítica era que ela não fornecia nenhum 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 daemon manualmente, matá-lo, esperar até que ele realmente finalizasse, então procurar pelas flags no /etc/rc, 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 de reinicialização consistisse em mais de um daemon ou exigisse ações adicionais. Em poucas palavras, o único script não cumpriu o objetivo dos scripts: 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 configurar a rede. Ele permitia acessar a rede a partir do modo single-user, 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 mudou para /etc/rc.network. Este último não era mais um script comum; ele era composto por um emaranhado de funções sh(1) chamadas pelo /etc/rc em diferentes estágios da inicialização do sistema. No entanto, a medida que as tarefas de inicialização cresciam variadas e sofisticadas, a abordagem "quase modular" tornou-se ainda mais engessada do que o monolítico /etc/rc.

Sem um framework limpo e bem projetado, os scripts de inicialização tiveram que se curvar para satisfazer as necessidades de desenvolvimento rápido dos sistemas operacionais baseados no BSD. Tornou-se óbvio, finalmente, que mais passos eram necessários no caminho para construção de um sistema rc extensível e customizável. Assim nasceu o BSD rc.d. Seus pais reconhecidos foram o Luke Mewburn e a comunidade do NetBSD. Mais tarde ele foi importado para o FreeBSD. Seu nome se refere à localização dos scripts do sistema para serviços individuais, que é o /etc/rc.d. Em breve, vamos aprender sobre mais componentes do sistema rc.d e vamos ver como os scripts individuais são invocados.

As idéias básicas por trás do BSD rc.d 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, obtém seu próprio script sh() capaz de iniciar o serviço, pará-lo, recarregá-lo e verificar seu status. Uma ação específica é escolhida pelo argumento da linha de comando para o script. O script /etc/rc ainda comanda a inicialização do sistema, mas agora ele simplesmente invoca os scripts menores um por um com o argumento `start`. É fácil executar tarefas de desligamento executando o mesmo conjunto de scripts com o argumento `stop`, o que é feito pelo /etc/rc.shutdown. Observe como isso segue de perto o modo Unix de ter um conjunto de pequenas ferramentas especializadas, cada uma cumprindo sua tarefa da melhor forma 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 conter apenas algumas linhas de código sh(1). Finalmente, uma parte importante do framework do rc.d é rcorder(8), o qual ajuda o /etc/rc a executar os pequenos scripts ordenadamente em relação às dependências entre eles. Ele também pode ajudar o /etc/rc.shutdown, porque a ordem apropriada para a sequência de encerramento é oposta à da inicialização.

O design do BSD rc.d é descrito no the original article by Luke Mewburn, e os componentes do rc.d são documentados em grande detalhe nas the respective manual pages. No entanto, pode não parecer óbvio para um novato em rc.d como amarrar os inúmeros pedaços juntos para criar um script bem estilizado para uma tarefa específica. Portanto, este artigo tentará uma abordagem diferente para descrever o rc.d. Ele mostrará quais recursos devem ser usados em vários casos típicos e por quê. Note que este não é um documento explicativo porque nosso objetivo não é fornecer receitas prontas, mas mostrar algumas entradas fáceis no domínio do rc.d. Nem este artigo é um substituto para as páginas de manual relevantes. Não hesite em consultá-los para obter uma documentação mais formal e completa ao ler este artigo.

Existem pré-requisitos para entender este artigo. Primeiro de tudo, você deve estar familiarizado com a linguagem de script sh(1) para poder dominar o rc.d. Além disso, você deve saber como o sistema executa as tarefas de inicialização e encerramento do userland, o que está descrito em rc(8).

Este artigo foca no branch rc.d do FreeBSD. No entanto, ele também pode ser útil para os desenvolvedores do NetBSD, porque os dois branchs rc.d do BSD não apenas compartilham o

mesmo design, mas também permanecem similares em seus aspectos visíveis aos autores do script.

2. Esboçando a tarefa

Um pouco de consideração antes de iniciar o `$EDITOR` não irá prejudicar. Para escrever um script rc.d corretamente customizado para um serviço do sistema, devemos poder responder as seguintes questões 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 observadas são:

 Um script interpretado deve começar com a linha mágica "shebang". Essa linha especifica o programa interpretador para o script. Devido a 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 adequadamente gerenciado pelo framework do rc.d, seus scripts precisam ser escritos na linguagem sh(1). Se você tiver um serviço ou port que use um utilitário de controle binário ou uma rotina de inicialização escrita em outra linguagem, instale este elemento em /usr/sbin (para o sistema) ou em /usr/local/sbin (para um port) e invoque-o por meio de um script sh(1) no diretório apropriado do rc.d.

Caso você queira aprender os detalhes do porque os scripts rc.d devem ser escritos na linguagem sh(1), veja como o /etc/rc invoca-os por meio de `run_rc_script`, e então estude a implementação de `run_rc_script` em /etc/rc. subr.

 Em /etc/rc.subr, várias 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 usar o rc.subr(8), as suas funções são extremamente úteis e tornam o trabalho mais fácil. Portanto, não é de surpreender que todos recorram a scripts rc.subr(8) em rc.d. Nós não vamos ser uma exceção.

Um script rc.d deve "incluir" o /etc/rc.subr (isto por ser feito usando o comando “.”) *antes* que ele chame as funções do rc.subr(8) para que o sh(1) tenha a oportunidade para aprender as funções. O estilo preferido é incluir o /etc/rc.subr antes de tudo.

Algumas funções úteis relacionadas a rede são fornecidas por outro arquivo include, o /etc/network.subr.

 A variável obrigatória `name` especifica o nome do nosso script. Ela é exigida pelo rc.subr(8). Ou seja, cada script rc.d *deve* definir a variável `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 do script rc.d é incluir valores atribuídos as variáveis entre aspas duplas. Tenha em mente que é apenas um problema de estilo que nem sempre pode ser aplicável. Você pode omitir com segurança as aspas das palavras simples sem os metacaracteres do sh(1) nelas, enquanto em certos casos você precisará de aspas simples para evitar qualquer interpretação do valor pelo sh(1). Um programador deve ser capaz de dizer a sintaxe da linguagem a partir das convenções de estilo e bem como de usá-las sabiamente.

 A idéia principal por trás do rc.subr(8) é que um 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 tratados desta maneira. Um método é uma expressão sh(1) armazenada em uma variável denominada `argument_cmd`, no qual *argument* corresponde ao que pode ser especificado na linha de comando do script. Vamos ver mais adiante como o rc.subr(8) fornece métodos default para os argumentos padrão.

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

 Devemos ter em mente que o rc.subr(8) fornece métodos default para os argumentos padrões. Consequentemente, devemos sobrescrever um método default com uma expressão no-op sh() se desejarmos que ele não faça nada.

 O corpo de um método sofisticado pode ser implementado como uma função. É uma boa ideia tornar o nome da função significativo.

É altamente recomendado adicionar o prefixo `${name}` aos nomes de todas as funções definidas em nosso script, para que eles nunca entrem em conflito com as funções do rc.subr(8) ou outro arquivo de inclusão comum.

 Essa chamada ao rc.subr(8) carrega as variáveis do rc.conf(5). Nosso script não faz uso delas ainda, mas ainda assim é recomendado carregar o rc.conf(5) pois podem haver variáveis rc.conf(5) controlando o rc.subr(8) propriamente dito.

 Geralmente este é o último comando em um script rc.d. Ele invoca o maquinário rc.subr(8) para executar a ação solicitada usando as variáveis e métodos que nosso script forneceu.

4. Um script fictício configurável

Agora vamos adicionar alguns controles ao nosso script fictício. Como você deve saber, os scripts rc.d são controlados pelo rc.conf(5). Felizmente, o rc.subr(8) esconde todas as complicações de nós. O script a seguir usa o rc.conf(5) via rc.subr(8) para ver se ele está habilitado em primeiro lugar, e buscar uma mensagem para mostrar no momento da inicialização. Estas duas tarefas são de fato independentes. Por um lado, um script rc.d pode apenas suportar a ativação e desativação de seu serviço. Por outro lado, um script rc.d obrigatório pode ter variáveis de configuração. Nós vamos fazer 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 ON/OFF.

② Agora o `load_rc_config` é invocado anteriormente no script, antes que qualquer variável do rc.conf(5) seja acessada.

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

③ Um aviso será emitido pelo `run_rc_command` se o próprio `rcvar` estiver definido, mas a variável de knob indicada não estiver definida. Se o seu script rc.d for para o sistema base, você deve adicionar uma configuração padrão para o knob no /etc/defaults/rc.conf e documentá-lo em rc.conf(5). Caso contrário, será o seu script que deverá 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 knob fosse definido como `ON`, independentemente da sua configuração atual, prefixando o argumento para o script com `one` ou `force`, como em `onestart` ou `forcestop`. Tenha em mente que o `force` tem outros efeitos perigosos que mencionaremos abaixo, enquanto `one` apenas sobrescreve o knob ON/OFF. Por exemplo, suponha que `dummy_enable` seja `OFF`. O comando a seguir executará o método `start` apesar da configuração:

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

 Agora, a mensagem a ser mostrada no momento da inicialização não é mais codificada no script. Ela é especificada por uma variável do rc.conf(5) chamada `dummy_msg`. Este é um exemplo trivial de como as variáveis do rc.conf(5) podem controlar um script rc.d.

Os nomes de todas as variáveis do rc.conf(5) usadas exclusivamente pelo nosso script *devem* possuir 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 exclusivo `${name}_` a todos os nomes globais introduzidos pelo nosso script nos salvará de possíveis colisões com o nome das funções existentes no rc.subr(8).

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

 Aqui usamos `dummy_msg` para realmente controlar nosso script, ou seja, para emitir uma mensagem variável. O uso de uma função de shell é um exagero aqui, já que ele só executa um único comando; uma alternativa igualmente válida seria:

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

5. Inicialização e desligamento de um daemon simples

Dissemos anteriormente que o rc.subr(8) poderia fornecer métodos padrão. Obviamente, estes padrões não podem ser muito gerais. Eles são adequados para o caso comum de iniciar e encerrar um programa 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 o rc.subr(8). Se estiver definido, o rc.subr(8) agirá de acordo com o cenário de servir um daemon convencional. Em particular, os métodos padrão serão fornecidos para tais argumentos: `start`, `stop`, `restart`, `poll`, e `status`.

O daemon será iniciado executando `$command` com os sinalizadores de linha de comando especificados por `$mumbled_flags`. Assim, todos os dados de entrada para o método padrão `start` estão disponíveis nas variáveis configuradas pelo nosso script. Ao contrário do `start`, outros métodos podem requerer informações adicionais sobre o processo iniciado. Por exemplo, `stop` deve conhecer o PID do processo para terminá-lo. No presente caso, rc.subr(8)varrerá a lista de todos os processos, procurando por um processo com seu nome igual a `$procname`. Esta última é outra variável de significado para rc.subr(8), e seu valor é padronizado para `command`. Em outras palavras, quando definimos o `command`, `procname` é efetivamente definido para o mesmo valor. Isso permite que nosso script mate o daemon e verifique se ele está sendo executado em primeiro lugar.

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

Para cada script rc.d, existe uma variável rc.conf() que tem precedência sobre `command`. Seu nome é construído da seguinte forma: `${name}_program`, onde `name` é a variável obrigatória que discutimos earlier. Por exemplo, neste caso, será `mumbled_program`. É rc.subr(8) que organiza `${name}_program` para substituir o comando.

Obviamente, o sh(1) permitirá que você defina `${name}_program` a partir do rc.conf(5) ou o próprio script, mesmo que o `command` esteja indefinido. Nesse caso, as propriedades especiais de `${name}_program` são perdidas e se tornam uma variável comum que seu script pode usar para seus próprios propósitos. No entanto, o uso exclusivo de `${name}_program` é desencorajado porque usá-lo junto com o `command` tornou-se um idioma na escrita de scripts rc.d.

Para obter informações mais detalhadas sobre 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/${name}.conf /usr/shared/misc/${name}.rules" ③

sig_reload="USR1" ④

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

extra_commands="reload plugh xyzzy" ⑦

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

mumbled_prestart()
{
    if checkyesno mumbled_smart; then ⑨
        rc_flags="-o smart ${rc_flags}" ⑩
    fi
    case "$mumbled_mode" in
    foo)
        rc_flags="-frotz ${rc_flags}"
        ;;
    bar)
        rc_flags="-baz ${rc_flags}"
        ;;
    *)
        warn "Invalid value for mumbled_mode" ⑪
```

```
        return 1 ⑫
        ;;
    esac
    run_rc_command xyzzy ⑬
    return 0
}

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

load_rc_config $name
run_rc_command "$1"
```

① Argumentos adicionais para `$command` podem ser passados em `command_args`. Eles serão adicionados a linha de comando após `$mumbled_flags`. Como a linha de comando final é passada para `eval` para sua execução real, os redirecionamentos de entrada e saída podem ser especificados em `command_args`.

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

⑫ Um daemon de boas maneiras deve criar um *pidfile* para que seu processo possa ser encontrado com mais facilidade e confiabilidade. A variável `pidfile`, se configurada, informa ao rc.subr(8) onde pode encontrar o pidfile para seus métodos padrão possam usar.

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

⑬ Se o daemon não puder ser executado a menos que existam certos arquivos, apenas liste-os em `required_files`, e rc.subr(8) irá verificar se esses arquivos existem antes de iniciar o daemon. Também existem `required_dirs` e `required_vars` para diretórios e variáveis de ambiente, respectivamente. Todos eles são descritos em detalhes em rc.subr(8).

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

⑭ Podemos personalizar sinais para enviar para o daemon caso eles sejam diferentes dos mais conhecidos. Em particular, `sig_reload` especifica o sinal que faz o daemon recarregar sua configuração; é `SIGHUP` por padrão. Outro sinal é enviado para parar o processo do daemon; o padrão é `SIGTERM`, mas isso pode ser alterado definindo `sig_stop` apropriadamente.

Os nomes dos sinais devem ser especificados 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 tipos de sistema operacional não.

Realizar tarefas adicionais antes ou depois dos métodos padrão é fácil. Para cada argumento de comando suportado pelo nosso script, podemos definir o argumento `_precmd` e `_postcmd`. Esses comandos no sh(1) são invocados antes e depois do respectivo método, como é evidente em seus nomes.

Sobrescrever um método padrão com um `argumento _cmd` personalizado ainda não nos impede de fazer uso do `argumento _precmd` ou `argumento _postcmd` se precisarmos. Em particular, o primeiro é bom para verificar condições personalizadas e sofisticadas que devem ser atendidas antes de executar o comando em si. Usar o `argumento _precmd` junto com o `argumento _cmd` nos permite separar logicamente as verificações da ação.

Não se esqueça de que você pode amontoar qualquer expressão válida do sh(1) nos métodos, pré e pós-comandos definidos por você. Apenas invocar uma função que faz com que o trabalho real seja um bom estilo na maioria dos casos, mas nunca deixe o estilo limitar sua compreensão do que está acontecendo por trás da cortina.

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

O comando `reload` é especial. Por um lado, tem 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 solicitar explicitamente que a funcionalidade incorporada seja fornecida. Podemos fazer isso via `extra_commands`.

O que obtemos do método padrão para `reload`? Muitas vezes, os daemons recarregam sua configuração na recepção de um sinal - normalmente, `SIGHUP`. Portanto, o rc.subr(8) tenta recarregar o daemon enviando um sinal para ele. O sinal é predefinido para `SIGHUP`, mas pode ser personalizado via `sig_reload`, caso necessário.

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 embutido, enquanto que para `plugh` é implementado como a função `mumbled_plugh`.

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

A lista completa de comandos disponíveis pode ser encontrada na linha de uso impressa por 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
Uso: /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. Isto pode parecer semelhante as funções de chamada, mas sabemos que comandos e funções de 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 chamados ordenadamente. Portanto, a maneira correta de um script executar seu próprio comando é por meio de rc.subr(8), conforme mostrado no exemplo.

 Uma função útil chamada `checkyesno` é fornecida por rc.subr(8). Ele usa um nome de variável como argumento e retorna um código de saída zero se, e somente se, a variável estiver configurada como `YES`, ou `TRUE`, ou `ON`, ou `1`, sem distinção entre maiúsculas e minúsculas; um código de saída diferente de zero é retornado de outra forma. No último caso, a função testa a variável como sendo definida como `NO`,`FALSE`,`OFF` ou `0` insensível a maiúsculas e minúsculas; 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 diferente de zero significa falso.

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

O uso correto de `checkyesno` é:

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

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

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

 Podemos afetar os sinalizadores a serem passados para `$command` modificando `rc_flags` em `$start_precmd`.

 Em certos casos, podemos precisar emitir uma mensagem importante que também deve ser enviada para o `syslog`. Isto pode ser feito facilmente com as seguintes funções rc.subr(8): `debug`, `info`, `warn` e `err`. A última função, em seguida, sai do script com o código especificado.

 Os códigos de saída dos métodos e seus pré-comandos não são apenas ignorados por padrão. Se o argumento `_precmd` retornar um código de saída diferente de zero, o método principal não será executado. Por sua vez, o `argumento_postcmd` não será invocado a menos que o método principal retorne um código de saída zero.

No entanto, o rc.subr(8) pode ser instruído a partir da linha de comando para ignorar esses códigos de saída e invocar todos os comandos, prefixando um argumento com `force`, como em `forcestart`.

7. Conectando um script ao framework rc.d

Depois que um script foi escrito, ele precisa ser integrado em rc.d>. O passo crucial é instalar o script em /etc/rc.d (para o sistema base) ou /usr/local/etc/rc.d (para ports). Ambos bsd.prog.mk e bsd.port.mk fornecer ganchos convenientes para isso, e geralmente você não precisa se preocupar com a propriedade e o modo adequado. Os scripts do sistema devem ser instalados a partir do src /etc/rc.d através do Makefile encontrado lá. Os scripts de porta podem ser instalados usando `USE_RC_SUBR` conforme descrito em no Manual do Porter.

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 de perto. Em poucas palavras, o rcorder(8) pega um conjunto de arquivos, examina seu conteúdo e imprime uma lista ordenada por dependência de arquivos do conjunto para `stdout`. O objetivo é manter as informações de dependência *dentro* dos arquivos para que cada arquivo possa falar por si só. Um arquivo pode especificar as seguintes informações:

- os nomes das "condições" (o que significa serviços para nós) que ele *fornece*;
- os nomes das "condições" que ele *requer*;
- os nomes das "condições" deste arquivo devem ser executados *antes*;
- *palavras-chave adicionais* que podem ser usadas para selecionar um subconjunto de todo o conjunto de arquivos (rcorder(8) podem ser instruídos através de opções para incluir ou omitir os arquivos com determinadas palavras-chave listadas.)

Não é surpresa que rcorder(8) possa manipular apenas arquivos de texto com uma sintaxe próxima a de sh(1). Ou seja, linhas especiais compreendidas por rcorder(8) se parecem com comentários sh(1). A sintaxe de tais linhas especiais é bastante rígida para simplificar seu processamento. Veja rcorder(8) para detalhes.

Além de usar linhas especiais do rcorder(8), um script pode insistir em sua dependência de outro serviço apenas iniciando-o forçadamente. Isso pode ser necessário quando o outro serviço é opcional e não será iniciado automaticamente porque o administrador do sistema o desativou por engano no 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:

 Esta linha declara os nomes das "condições" que nosso script fornece. Agora, outros scripts podem registrar uma dependência em nosso script por estes 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 a única, `PROVIDE:` deve ser o mesmo que `${name}`.

 Portanto, nosso script indica quais condições "" são fornecidas por outros scripts dos quais depende. De acordo com as linhas, nosso script pede ao rcorder(8) para colocá-lo após o(s) script(s) fornecendo DAEMON e cleanvar, mas antes disso prover LOGIN.

A linha `BEFORE:` não deve ser abusada para contornar uma lista de dependências incompleta no outro script. O caso apropriado para usar o `BEFORE:` é quando o outro script não se importa com o nosso, mas nosso script pode fazer sua tarefa melhor se for executado antes do outro. Um típico exemplo da vida real são as interfaces de rede versus o firewall: embora as interfaces não dependam do firewall em realizar seu trabalho, a segurança do sistema se beneficiará do firewall estar pronto antes que haja qualquer tráfego de rede.

Além das condições correspondentes a um único serviço, existem meta-condições e seus scripts "placeholder" usados para garantir que determinados grupos de operações sejam executados antes dos outros. Estes são denotados pelos nomes UPPERCASE. Sua lista e finalidades podem ser encontradas em rc(8).

Tenha em mente que colocar um nome de serviço na linha `REQUIRE:` não garante que o serviço estará realmente em execução no momento em que nosso script for iniciado. O serviço necessário pode falhar ao iniciar ou simplesmente ser desativado em rc.conf(5). Obviamente, o rcorder(8) não pode controlar tais detalhes, e o rc(8) também não fará isso. Consequentemente, o aplicativo iniciado por nosso script deve ser capaz de lidar com quaisquer serviços necessários indisponíveis. Em certos casos, podemos ajudá-lo conforme discutido below

 Como lembramos do texto acima, as palavras-chave do rcorder(8) podem ser usadas para selecionar ou deixar alguns scripts. Ou seja, qualquer consumidor rcorder(8) pode especificar através das opções `-k` e `-s` que as palavras-chave estão na "keep list" e na "skip list", respectivamente. De todos os arquivos a serem classificados, o rcorder(8) selecionará apenas aqueles que tiverem uma palavra-chave da lista de manutenção (a menos que vazia) e não uma palavra-chave da lista de itens ignorados.

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

 Para começar, `force_depend` deve ser usado com muito cuidado. Geralmente é melhor revisar a hierarquia de variáveis de configuração para seus scripts rc. se eles forem interdependentes.

Se você ainda não pode fazer sem `force_depend`, o exemplo oferece uma expressão de como invocá-lo condicionalmente. No exemplo, nosso daemon `mumbled` requer que outro, `frotz`, seja iniciado antecipadamente. No entanto, `frotz` é opcional também; e rcorder(8) não sabe nada sobre esses detalhes. Felizmente, nosso script tem acesso a todas as variáveis rc.conf(5). Se `frotz_enable` estiver como true, esperamos pelo melhor e dependemos de rc.d para iniciar `frotz`. Caso contrário, nós forçadamente verificaremos o status de `frotz`. Finalmente, impomos nossa dependência ao `frotz` se ele não estiver sendo executado. Uma mensagem de aviso será emitida por `force_depend` porque ele deve ser chamado apenas se um erro de configuração for detectado.

8. Dando mais flexibilidade a um script rc.d

Quando chamado durante a inicialização ou desligamento, um script rc.d deve agir em todo o subsistema pelo qual é responsável. Por exemplo, /etc/rc.d/netif deve iniciar ou parar todas as interfaces de rede descritas por rc.conf(5). Qualquer tarefa pode ser indicada exclusivamente 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 ao rc.conf(5) e então iniciá-lo sem interferir o funcionamento das interfaces existentes. Da próxima vez, o administrador pode precisar desligar uma única interface de rede. No espírito da linha de comando, o respectivo script rc.d solicita um argumento extra, o nome da interface.

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

Como o rc.subr(8) pode obter acesso aos argumentos de linha de comando extra. Deveria pegá-los diretamente? Não por qualquer meio. Primeiro, uma função sh(1) não tem acesso aos parâmetros posicionais de seu chamador, mas o rc.subr(8) é apenas uma despedida de tais funções. Em segundo lugar, a boa maneira de rc.d determina que é para o script principal decidir quais argumentos devem ser passados para seus métodos.

Portanto, a abordagem adotada por rc.subr(8) é a seguinte: `run_rc_command` transmite todos os seus argumentos, mas o primeiro um para o respectivo método na íntegra. O primeiro, omitido, argumento é o nome do próprio método: `start`,`stop`, etc. Ele será deslocado por `run_rc_command`, então o que é `$2` na linha de comando original será apresentado como `$1` ao 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 digitados após `start` podem terminar como parâmetros posicionais para o respectivo método. Podemos usá-los de qualquer maneira de acordo com nossa tarefa, habilidades e fantasia. No exemplo atual, apenas passamos todos eles para echo(1) como uma cadeia na linha seguinte - note `$*` entre aspas duplas. Aqui está como o script pode ser chamado 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 forneça, não apenas a um método padrão. Nós adicionamos um método customizado chamado `kiss`, e ele pode tirar proveito dos argumentos extras da mesma forma que o `start` tira. 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 `"$@"` por `"$ 1"` na última linha do nosso script, onde invocamos o `run_rc_command`.

Um programador sh(1) deve entender a diferença sutil entre `$*` e `$@` como as formas de designar todos os parâmetros posicionais. Para sua discussão aprofundada, consulte um bom manual sobre programação de scripts sh(1). *Não* use estas expressões até que você as compreenda completamente, porque o uso incorreto delas resultará em scripts inseguros e contendo bugs.

Atualmente, o `run_rc_command` pode ter um bug que o impede de manter os limites originais entre os argumentos. Ou seja, argumentos com espaços em branco incorporados podem não ser processados corretamente. O bug deriva do uso inadequado de `$*`.

9. Leitura adicional

O artigo original de Luke Mewburn oferece uma visão geral do rc.d e o raciocínio detalhado que o levou a suas decisões de design. Ele fornece informações sobre toda o framework do rc.d e o seu lugar em um moderno sistema operacional BSD.

As páginas de manual rc(8), rc.subr(8) e rcorder(8) documentam os componentes do rc.d com grande detalhe. Você não pode usar totalmente o poder do rc.d sem estudar as páginas de manual e se referir a elas enquanto escreve seus próprios scripts.

A sua principal fonte de inspiração são os exemplos da vida real, existentes em no /etc/rc.d de um sistema vivo. Seu conteúdo é fácil e agradável de ler, porque a maioria dos cantos ásperos estão escondidos fundo no rc.subr(8). Tenha em mente que os scripts /etc/rc.d não foram escritos por anjos, então eles podem sofrer de bugs e decisões sub-ótimas de design. Agora você pode melhorá-los!