Red Hat Enterprise Linux 8

Software de embalagem e distribuição

Um guia para embalagem e distribuição de software no Red Hat Enterprise Linux 8

Last Updated: 2021-03-08

Red Hat Enterprise Linux 8 Software de embalagem e distribuição

Um guia para embalagem e distribuição de software no Red Hat Enterprise Linux 8

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

Nota Legal

Resumo

Este documento descreve como empacotar software em um RPM. Ele também mostra como preparar o código fonte para empacotamento e explica alguns cenários avançados de empacotamento, como empacotar projetos Python ou RubyGems em RPM.

Índice

TORNANDO O CÓDIGO ABERTO MAIS INCLUSIVO

A Red Hat tem o compromisso de substituir a linguagem problemática em nosso código, documentação e propriedades da web. Estamos começando com estes quatro termos: master, slave, blacklist e whitelist. Por causa da enormidade deste esforço, estas mudanças serão implementadas gradualmente ao longo de vários lançamentos futuros. Para mais detalhes, veja a mensagem de nosso CTO Chris Wright.

FORNECENDO FEEDBACK SOBRE A DOCUMENTAÇÃO DA RED HAT

Agradecemos sua contribuição em nossa documentação. Por favor, diga-nos como podemos melhorá-la. Para fazer isso:

- Para comentários simples sobre passagens específicas:
  1. Certifique-se de que você está visualizando a documentação no formato *Multi-page HTML*. Além disso, certifique-se de ver o botão **Feedback** no canto superior direito do documento.
  2. Use o cursor do mouse para destacar a parte do texto que você deseja comentar.
  3. Clique no pop-up **Add Feedback** que aparece abaixo do texto destacado.
  4. Siga as instruções apresentadas.
- Para enviar comentários mais complexos, crie um bilhete Bugzilla:
  1. Ir para o site da Bugzilla.
  2. Como Componente, use **Documentation**.
  3. Preencha o campo **Description** com sua sugestão de melhoria. Inclua um link para a(s) parte(s) relevante(s) da documentação.
  4. Clique em **Submit Bug**.

CAPÍTULO 1. COMEÇANDO COM A EMBALAGEM RPM

A seção seguinte introduz o conceito de embalagem RPM e suas principais vantagens.

1.1. INTRODUÇÃO À EMBALAGEM RPM

O RPM (RPM) é um sistema de gerenciamento de pacotes que roda no Red Hat Enterprise Linux, CentOS, e Fedora. Você pode usar o RPM para distribuir, gerenciar e atualizar o software que você criar para qualquer um dos sistemas operacionais mencionados acima.

1.1.1. Vantagens do RPM

O sistema de gerenciamento de pacotes RPM traz várias vantagens em relação à distribuição de software em arquivos convencionais.

RPM permite que você o faça:

- Instalar, reinstalar, remover, atualizar e verificar pacotes com ferramentas padrão de gerenciamento de pacotes, tais como Yum ou PackageKit.
- Use um banco de dados de pacotes instalados para consultar e verificar os pacotes.
- Use metadados para descrever os pacotes, suas instruções de instalação e outros parâmetros de pacotes.
- Pacote de fontes de software, correções e instruções completas de compilação em pacotes fonte e binários.
- Adicionar pacotes aos repositórios Yum.
- Assine digitalmente seus pacotes usando as chaves de assinatura do GNU Privacy Guard (GPG).

CAPÍTULO 2. PREPARANDO SOFTWARE PARA EMBALAGEM RPM

Esta seção explica como preparar o software para embalagem RPM. Para fazer isso, não é necessário saber codificar. Entretanto, você precisa entender os conceitos básicos, tais como O que é código fonte e Como os programas são feitos.

2.1. O QUE É CÓDIGO FONTE

Esta parte explica o que é código fonte e mostra exemplos de códigos fonte de um programa escrito em três linguagens de programação diferentes.

O código fonte é uma instrução legível para o computador, que descreve como fazer um cálculo. O código fonte é expresso usando uma linguagem de programação.

2.1.1. Exemplos de código fonte

Este documento apresenta três versões do programa **Hello World** escritas em três linguagens de programação diferentes:

- Seção 2.1.1.1, “Olá Mundo escrito em bash”
- Seção 2.1.1.2, “Olá Mundo escrito em Python”
- Seção 2.1.1.3, “Olá Mundo escrito em C”

Cada versão é embalada de forma diferente.

Estas versões do programa **Hello World** cobrem os três principais casos de uso de um empacotador RPM.

2.1.1.1. Olá Mundo escrito em bash

O projeto *bello* implementa **Hello World** em bash. A implementação contém apenas o script em shell **bello**. O objetivo do programa é produzir **Hello World** na linha de comando.

O arquivo **bello** tem a seguinte sintaxe:

```
#!/bin/bash

printf "Hello World\n"
```

2.1.1.2. Olá Mundo escrito em Python

O projeto *pello* implementa **Hello World** em Python. A implementação contém apenas o programa **pello.py**. O objetivo do programa é produzir **Hello World** na linha de comando.

O arquivo **pello.py** tem a seguinte sintaxe:

```
#!/usr/bin/python3

print("Hello World")
```

2.1.1.3. Olá Mundo escrito em C

O projeto *cello* implementa **Hello World** em C. A implementação contém apenas os arquivos **cello.c** e **Makefile**, portanto o arquivo **tar.gz** resultante terá dois arquivos além do arquivo **LICENSE**.

O objetivo do programa é sair **Hello World** na linha de comando.

O arquivo **cello.c** tem a seguinte sintaxe:

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

2.2. COMO OS PROGRAMAS SÃO FEITOS

Os métodos de conversão de código fonte legível por humanos para código de máquina (instruções que o computador segue para executar o programa) incluem o seguinte:

- O programa é compilado nativamente.
- O programa é interpretado por meio de interpretação em bruto.
- O programa é interpretado pela compilação de bytes.

2.2.1. Código Nativamente Compilado

Software compilado nativamente é um software escrito em uma linguagem de programação que se compila em código de máquina com um arquivo executável binário resultante. Tal software pode ser executado de forma autônoma.

Os pacotes de RPM construídos desta forma são específicos da arquitetura.

Se você compila tal software em um computador que usa um processador AMD ou Intel de 64 bits (x86_64), ele não é executado em um processador AMD ou Intel de 32 bits (x86). O pacote resultante tem a arquitetura especificada em seu nome.

2.2.2. Código Interpretado

Algumas linguagens de programação, tais como bash ou Python, não compilam para código de máquina. Ao invés disso, o código fonte de seus programas é executado passo a passo, sem transformações prévias, por um Intérprete de Linguagem ou por uma Máquina Virtual de Linguagem.

O software escrito inteiramente em linguagens de programação interpretadas não é específico para arquitetura. Portanto, o pacote RPM resultante tem a string **noarch** em seu nome.

As línguas interpretadas são programas de interpretação em bruto ou programas compilados por Byte. Estes dois tipos diferem no processo de construção do programa e no procedimento de embalagem.

2.2.2.1. Programas de interpretação em bruto

Os programas de linguagem de interpretação em bruto não precisam ser compilados e são executados diretamente pelo intérprete.

2.2.2.2. Programas compilados por bytes

Os idiomas compilados em byte precisam ser compilados em código byte, que é então executado pela máquina virtual do idioma.

NOTA

Alguns idiomas oferecem uma escolha: eles podem ser interpretados em bruto ou compilados por bytes.

2.3. CONSTRUINDO SOFTWARE A PARTIR DA FONTE

Esta parte descreve como construir software a partir do código fonte.

Para software escrito em idiomas compilados, o código fonte passa por um processo de compilação, produzindo código de máquina. Este processo, comumente chamado de compilação ou tradução, varia para diferentes idiomas. O software construído resultante pode ser executado, o que faz com que o computador execute a tarefa especificada pelo programador.

Para software escrito em idiomas de interpretação bruta, o código fonte não é construído, mas executado diretamente.

Para software escrito em idiomas interpretados por bytes, o código fonte é compilado em código byte, que é então executado pela máquina virtual do idioma.

2.3.1. Código Nativamente Compilado

Esta seção mostra como construir o programa **cello.c** escrito na linguagem C em um executável.

cello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

2.3.1.1. Construção manual

Se você quiser construir o programa **cello.c** manualmente, use este procedimento:

Procedimento

1. Invocar o compilador C da Coleção de Compiladores GNU para compilar o código fonte em binário:

```
gcc -g -o violoncelo cello.c
```

2. Executar o binário de saída resultante **cello**:

```
$ ./cello
Hello World
```

2.3.1.2. Construção automatizada

O software de grande escala geralmente usa a construção automatizada que é feita através da criação do arquivo **Makefile** e, em seguida, executando o utilitário GNU **make**.

Se você quiser usar o edifício automatizado para construir o programa **cello.c**, use este procedimento:

Procedimento

1. Para criar um edifício automatizado, crie o arquivo **Makefile** com o seguinte conteúdo no mesmo diretório que **cello.c**.
   Makefile

   ```
   cello:
    gcc -g -o cello cello.c
   clean:
    rm cello
   ```

   Observe que as linhas sob **cello:** e **clean:** devem começar com um espaço de tabulação.

2. Para construir o software, execute o comando **make**:

   ```
   $ make
   make: 'cello' is up to date.
   ```

3. Como já existe uma construção disponível, execute o comando **make clean**, e depois execute novamente o comando **make**:

   ```
   $ make clean
   rm cello

   $ make
   gcc -g -o cello cello.c
   ```

   NOTA

   Tentar construir o programa depois de outra construção não tem nenhum efeito.

   ```
   $ make
   make: 'cello' is up to date.
   ```

4. Executar o programa:

   ```
   $ ./cello
   Hello World
   ```

Agora você compilou um programa tanto manualmente quanto usando uma ferramenta de construção.

2.3.2. Código interpretativo

Esta seção mostra como byte-compilar um programa escrito em Python e interpretar em bruto um programa escrito em bash.

NOTA

Nos dois exemplos abaixo, a linha **#!** no topo do arquivo é conhecida como **shebang**, e não faz parte da linguagem de programação código fonte.

O **shebang** permite utilizar um arquivo texto como executável: o carregador de programas do sistema analisa a linha contendo o **shebang** para obter um caminho para o executável binário, que é então utilizado como o intérprete da linguagem de programação. A funcionalidade exige que o arquivo de texto seja marcado como executável.

2.3.2.1. Byte-código de compilação

Esta seção mostra como compilar o programa **pello.py** escrito em Python em código byte, que é então executado pela máquina virtual em linguagem Python.

O código fonte Python também pode ser interpretado em bruto, mas a versão compilada por bytes é mais rápida. Portanto, os empacotadores RPM preferem empacotar a versão compilada por bytes para distribuição aos usuários finais.

pello.py

```
#!/usr/bin/python3

print("Hello World")
```

O procedimento para programas de compilação de bytes varia de acordo com os seguintes fatores:

- Linguagem de programação
- Máquina virtual do idioma
- Ferramentas e processos utilizados com essa linguagem

NOTA

Python é freqüentemente compilado por bytes, mas não da maneira descrita aqui. O procedimento a seguir visa não estar em conformidade com os padrões da comunidade, mas ser simples. Para as diretrizes Python do mundo real, veja Software Packaging and Distribution.

Use este procedimento para compilar **pello.py** em código byte:

Procedimento

1. Byte-compilar o arquivo **pello.py**:

```
$ python -m compileall pello.py

$ file pello.pyc
pello.pyc: python 2.7 byte-compiled
```

2. Executar o código de bytes em **pello.pyc**:

```
$ python pello.pyc
Hello World
```

2.3.2.2. Código de interpretação em bruto

Esta seção mostra como interpretar em bruto o programa **bello** escrito na linguagem bash shell embutida.

bello

```
#!/bin/bash

printf "Hello World\n"
```

Os programas escritos em linguagem de script de concha, como *bash*, são interpretados em bruto.

Procedimento

- Faça o arquivo com o código fonte executável e execute-o:

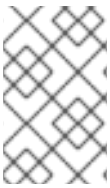
```
$ chmod +x bello
$ ./bello
Hello World
```

2.4. SOFTWARE DE REMENDO

Esta seção explica como consertar o software.

Na embalagem RPM, em vez de modificar o código fonte original, nós o mantemos e usamos patches nele.

Um patch é um código fonte que atualiza outro código fonte. Ele é formatado como um *diff*, porque representa o que há de diferente entre duas versões do texto. Um *diff* é criado usando o utilitário **diff**, que é então aplicado ao código-fonte usando o utilitário de patch.



NOTA

Os desenvolvedores de software freqüentemente usam Sistemas de Controle de Versão como o git para gerenciar sua base de códigos. Tais ferramentas fornecem seus próprios métodos de criação de diffs ou de software de correção.

O exemplo a seguir mostra como criar um patch a partir do código fonte original usando **diff**, e como aplicar o patch usando **patch**. O patch é usado em uma seção posterior ao criar um RPM; ver Seção 3.2, “Trabalhando com arquivos SPEC”.

Este procedimento mostra como criar um patch a partir do código fonte original para **cello.c**.

Procedimento

1. Preservar o código fonte original:

```
$ cp -p cello.c cello.c.origina
```

A opção **-p** é usada para preservar o modo, a propriedade e os carimbos de data e hora.

2. Modifique o site **cello.c** conforme necessário:

```
#include <stdio.h>

int main(void) {
    printf("Hello World from my very first patch!\n");
    return 0;
}
```

3. Gerar um patch usando o utilitário **diff**:

```
$ diff -Naur cello.c.orig cello.c
--- cello.c.orig        2016-05-26 17:21:30.478523360 -0500
+ cello.c     2016-05-27 14:53:20.668588245 -0500
@@ -1,6 +1,6 @@
 #include<stdio.h>

 int main(void){
-    printf("Hello World!\n");
+    printf("Hello World from my very first patch!\n");
     return 0;
 }
\ No newline at end of file
```

As linhas que começam com um **-** são removidas do código fonte original e substituídas pelas linhas que começam com .

O uso das opções **Naur** com o comando **diff** é recomendado porque se encaixa na maioria dos casos de uso habitual. Entretanto, neste caso em particular, somente a opção **-u** é necessária. As opções particulares garantem o seguinte:

- **-N** (ou **--new-file**) - Manipula arquivos ausentes como se fossem arquivos vazios.
- **-a** (ou **--text**) - Trata todos os arquivos como texto. Como resultado, os arquivos que **diff** classifica como binários não são ignorados.
- **-u** (ou **-U NUM** ou **--unified[=NUM]**) - Retorna a saída sob a forma de linhas NUM (padrão 3) de contexto unificado. Este é um formato de fácil leitura que permite uma correspondência difusa ao aplicar o patch a uma árvore de origem alterada.
- **-r** (ou **--recursive**) - Compara recursivamente quaisquer subdiretórios que sejam encontrados.
  Para mais informações sobre os argumentos comuns para a utilidade **diff**, consulte a página do manual **diff**.

4. Salvar o patch em um arquivo:

```
$ diff -Naur cello.c.origina cello.c > cello-output-first-patch.patch
```

5. Restaurar o original **cello.c**:

```
$ cp cello.c.origina cello.c
```

O original **cello.c** deve ser mantido, pois quando se constrói um RPM, é utilizado o arquivo original e não o modificado. Para maiores informações, veja Seção 3.2, "Trabalhando com arquivos SPEC".

O procedimento a seguir mostra como remendar **cello.c** usando **cello-output-first-patch.patch**, construir o programa remendado e executá-lo.

1. Redirecionar o arquivo de patch para o comando **patch**:

```
$ patch < cello-output-first-patch.patch
patching file cello.c
```

2. Verifique se o conteúdo de **cello.c** agora reflete o adesivo:

```
$ cat cello.c
#include<stdio.h>

int main(void){
    printf("Hello World from my very first patch!\n");
    return 1;
}
```

3. Construir e executar o remendado **cello.c**:

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c

$ ./cello
Hello World from my very first patch!
```

2.5. INSTALAÇÃO DE ARTEFATOS ARBITRÁRIOS

Sistemas do tipo Unix usam o Filesystem Hierarchy Standard (FHS) para especificar um diretório adequado para um determinado arquivo.

Os arquivos instalados a partir dos pacotes RPM são colocados de acordo com FHS. Por exemplo, um arquivo executável deve ir para um diretório que está na variável **$PATH** do sistema.

No contexto desta documentação, um *Arbitrary Artifact* é qualquer coisa instalada desde um RPM até o sistema. Para o RPM e para o sistema pode ser um script, um binário compilado a partir do código fonte do pacote, um binário pré-compilado, ou qualquer outro arquivo.

Esta seção descreve duas formas comuns de colocar o *Arbitrary Artifacts* no sistema:

- Seção 2.5.1, "Usando o comando de instalação"
- Seção 2.5.2, "Usando o comando make install"

2.5.1. Usando o comando de instalação

Os empacotadores freqüentemente usam o comando **install** nos casos em que a construção de ferramentas de automação como o GNU make não é ótima; por exemplo, se o programa empacotado não precisar de despesas extras.

O comando **install** é fornecido ao sistema por coreutils, que coloca o artefato no diretório especificado no sistema de arquivos com um conjunto específico de permissões.

O procedimento seguinte utiliza o arquivo **bello** que foi criado anteriormente como o artefato arbitrário como um objeto sujeito a este método de instalação.

Procedimento

1. Execute o comando **install** para colocar o arquivo **bello** no diretório **/usr/bin** com as permissões comuns para scripts executáveis:

```
$ sudo install -m 0755 bello /usr/bin/bello
```

   Como resultado, **bello** está agora localizado no diretório que está listado na variável **$PATH**.

2. Executar **bello** a partir de qualquer diretório sem especificar seu caminho completo:

```
$ cd ~

$ bello
Hello World
```

2.5.2. Usando o comando make install

O uso do comando **make install** é uma forma automatizada de instalar o software construído no sistema. Neste caso, é necessário especificar como instalar os artefatos arbitrários ao sistema no **Makefile** que é normalmente escrito pelo desenvolvedor.

Este procedimento mostra como instalar um artefato de construção em um local escolhido no sistema.

Procedimento

1. Adicione a seção **install** ao site **Makefile**:
   Makefile

```
cello:
 gcc -g -o cello cello.c

clean:
 rm cello

install:
 mkdir -p $(DESTDIR)/usr/bin
 install -m 0755 cello $(DESTDIR)/usr/bin/cello
```

   Observe que as linhas sob **cello:**, **clean:** e **install:** devem começar com um espaço de tabulação.

NOTA

A variável $(DESTDIR) é uma variável do GNU e é comumente usada para especificar a instalação em um diretório diferente do diretório raiz.

Agora você pode usar **Makefile** não apenas para construir software, mas também para instalá-lo no sistema alvo.

2. Construir e instalar o programa **cello.c**:

```
$ make
gcc -g -o cello cello.c

$ sudo make install
install -m 0755 cello /usr/bin/cello
```

Como resultado, **cello** está agora localizado no diretório que está listado na variável **$PATH**.

3. Executar **cello** a partir de qualquer diretório sem especificar seu caminho completo:

```
$ cd ~

$ cello
Hello World
```

2.6. PREPARANDO O CÓDIGO FONTE PARA EMBALAGEM

Os desenvolvedores freqüentemente distribuem software como arquivos compactados de código fonte, que depois são usados para criar pacotes. Os empacotadores de RPM trabalham com um arquivo de código fonte pronto.

O software deve ser distribuído com uma licença de software.

Este procedimento utiliza o texto da licença GPLv3 como um exemplo do conteúdo do arquivo **LICENSE**.

Procedimento

- Crie um arquivo **LICENSE**, e certifique-se de que ele inclua o seguinte conteúdo:

```
$ cat /tmp/LICENSE
This program is free software: you can redistribute it and/or modify it under the terms of the
GNU General Public License as published by the Free Software Foundation, either version 3
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this
program. If not, see http://www.gnu.org/licenses/.
```

Recursos adicionais

- O código criado nesta seção pode ser encontrado aqui.

2.7. COLOCANDO O CÓDIGO FONTE NO TARBALL

Esta seção descreve como colocar cada um dos três programas **Hello World** introduzidos em Seção 2.1.1, "Exemplos de código fonte" em um tarball gzip-comprimido, que é uma forma comum de lançar o software a ser posteriormente empacotado para distribuição.

2.7.1. Colocando o projeto bello em tarball

O projeto *bello* implementa **Hello World** em bash. A implementação contém apenas o script da shell **bello**, portanto, o arquivo **tar.gz** resultante terá apenas um arquivo à parte do arquivo **LICENSE**.

Este procedimento mostra como preparar o projeto *bello* para distribuição.

Pré-requisitos

Considerando que esta é a versão **0.1** do programa.

Procedimento

1. Coloque todos os arquivos necessários em um único diretório:

```
$ mkdir /tmp/bello-0.1

$ mv ~/bello /tmp/bello-0.1/

$ cp /tmp/LICENSE /tmp/bello-0.1/
```

2. Crie o arquivo para distribuição e mova-o para o diretório **~/rpmbuild/SOURCES/**, que é o diretório padrão onde o comando **rpmbuild** armazena os arquivos para a construção de pacotes:

```
$ cd /tmp/

$ tar -cvzf bello-0.1.tar.gz bello-0.1
bello-0.1/
bello-0.1/LICENSE
bello-0.1/bello

$ mv /tmp/bello-0.1.tar.gz ~/rpmbuild/SOURCES/
```

Para mais informações sobre o exemplo de código fonte escrito em bash, veja Seção 2.1.1.1, "Olá Mundo escrito em bash".

2.7.2. Colocando o projeto pello em tarball

O projeto *pello* implementa **Hello World** em Python. A implementação contém apenas o programa **pello.py**, portanto o arquivo **tar.gz** resultante terá apenas um arquivo à parte do arquivo **LICENSE**.

Este procedimento mostra como preparar o projeto *pello* para distribuição.

Pré-requisitos

Considerando que esta é a versão **0.1.1** do programa.

Procedimento

1. Coloque todos os arquivos necessários em um único diretório:

```
$ mkdir /tmp/pello-0.1.2

$ mv ~/pello.py /tmp/pello-0.1.2/

$ cp /tmp/LICENSE /tmp/pello-0.1.2/
```

2. Crie o arquivo para distribuição e mova-o para o diretório **~/rpmbuild/SOURCES/**, que é o diretório padrão onde o comando **rpmbuild** armazena os arquivos para a construção de pacotes:

```
$ cd /tmp/

$ tar -cvzf pello-0.1.2.tar.gz pello-0.1.2
pello-0.1.2/
pello-0.1.2/LICENSE
pello-0.1.2/pello.py

$ mv /tmp/pello-0.1.2.tar.gz ~/rpmbuild/SOURCES/
```

Para mais informações sobre o exemplo de código fonte escrito em Python, veja Seção 2.1.1.2, “Olá Mundo escrito em Python”.

2.7.3. Colocando o projeto do violoncelo em tarball

O projeto *cello* implementa **Hello World** em C. A implementação contém apenas os arquivos **cello.c** e **Makefile**, portanto o arquivo **tar.gz** resultante terá dois arquivos além do arquivo **LICENSE**.

NOTA

O arquivo **patch** não é distribuído no arquivo com o programa. O RPM Packager aplica o patch quando o RPM é construído. O patch será colocado no diretório **~/rpmbuild/SOURCES/** junto com o arquivo **.tar.gz**.

Este procedimento mostra como preparar o projeto *cello* para distribuição.

Pré-requisitos

Considerando que esta é a versão **1.0** do programa.

Procedimento

1. Coloque todos os arquivos necessários em um único diretório:

```
$ mkdir /tmp/cello-1.0

$ mv ~/cello.c /tmp/cello-1.0/

$ mv ~/Makefile /tmp/cello-1.0/

$ cp /tmp/LICENSE /tmp/cello-1.0/
```

2. Crie o arquivo para distribuição e mova-o para o diretório **~/rpmbuild/SOURCES/**, que é o diretório padrão onde o comando **rpmbuild** armazena os arquivos para a construção de pacotes:

```
$ cd /tmp/

$ tar -cvzf cello-1.0.tar.gz cello-1.0
cello-1.0/
cello-1.0/Makefile
cello-1.0/cello.c
cello-1.0/LICENSE

$ mv /tmp/cello-1.0.tar.gz ~/rpmbuild/SOURCES/
```

3. Adicione o adesivo:

```
$ mv ~/cello-output-first-patch.patch ~/rpmbuild/SOURCES/
```

Para mais informações sobre o exemplo de código fonte escrito em C, veja Seção 2.1.1.3, “Olá Mundo escrito em C”.

CAPÍTULO 3. SOFTWARE DE EMBALAGEM

3.1. PACOTES DE RPM

Esta seção cobre os conceitos básicos do formato de embalagem RPM.

3.1.1. O que é um RPM

Um pacote RPM é um arquivo contendo outros arquivos e seus metadados (informações sobre os arquivos que são necessários ao sistema).

Especificamente, um pacote de RPM consiste no arquivo **cpio**.

O arquivo **cpio** contém:

- Arquivos
- Cabeçalho RPM (metadados do pacote)
  O gerenciador de pacotes **rpm** utiliza estes metadados para determinar as dependências, onde instalar arquivos e outras informações.

Tipos de pacotes de RPM
Há dois tipos de pacotes de RPM. Ambos os tipos compartilham o formato do arquivo e as ferramentas, mas têm conteúdos diferentes e servem a propósitos diferentes:

- Fonte RPM (SRPM)
  Um SRPM contém o código fonte e um arquivo SPEC, que descreve como construir o código fonte em um RPM binário. Opcionalmente, os patches para o código-fonte também são incluídos.
- RPM Binário
  Um RPM binário contém os binários construídos a partir das fontes e remendos.

3.1.2. Listagem das utilidades da ferramenta de embalagem RPM

Os seguintes procedimentos mostram como listar as utilidades fornecidas pelo pacote **rpmdevtools**.

Pré-requisitos

Para poder utilizar as ferramentas de embalagem RPM, você precisa instalar o pacote **rpmdevtools**, que fornece várias utilidades para embalagem de RPMs.

```
# yum instalar rpmdevtools
```

Procedimento

- Liste as utilidades da ferramenta de embalagem RPM:

  ```
  $ rpm -ql rpmdevtools | grep bin
  ```

Informações adicionais

- Para mais informações sobre as utilidades acima, consulte suas páginas de manual ou diálogos de ajuda.

3.1.3. Criação de espaço de trabalho de embalagem RPM

Esta seção descreve como configurar um layout de diretório que é o espaço de trabalho de embalagem RPM, usando o utilitário **rpmdev-setuptree**.

Pré-requisitos

O pacote **rpmdevtools** deve ser instalado em seu sistema:

```
# yum instalar rpmdevtools
```

Procedimento

- Execute o utilitário **rpmdev-setuptree**:

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/user/rpmbuild/
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS

5 directories, 0 files
```

Os diretórios criados servem a estes propósitos:

Diretório	Objetivo
CONSTRUÇÃO	Quando as embalagens são construídas, vários diretórios **%buildroot** são criados aqui. Isto é útil para investigar uma construção falhada se a saída dos logs não fornecer informações suficientes.
RPMS	Os RPMs binários são criados aqui, em subdiretórios para diferentes arquiteturas, por exemplo, nos subdiretórios **x86_64** e **noarch**.
FONTES	Aqui, o empacotador coloca arquivos de código fonte comprimido e patches. O comando **rpmbuild** procura por eles aqui.
SPECS	O embalador coloca aqui os arquivos SPEC.
SRPMS	Quando **rpmbuild** é usado para construir um SRPM em vez de um RPM binário, o SRPM resultante é criado aqui.

3.1.4. O que é um arquivo SPEC

Você pode entender um arquivo SPEC como uma receita que o utilitário **rpmbuild** usa para construir um

RPM. Um arquivo SPEC fornece informações necessárias para o sistema de construção definindo instruções em uma série de seções. As seções são definidas no *Preamble* e na parte *Body*. A parte *Preamble* contém uma série de itens de metadados que são usados na parte *Body*. A parte *Body* representa a parte principal das instruções.

3.1.4.1. Preâmbulo Itens

A tabela abaixo apresenta algumas das diretrizes que são usadas com freqüência na seção *Preamble* do arquivo SPEC da RPM.

Tabela 3.1. Itens utilizados no *Preamble* seção do arquivo SPEC da RPM

Diretriz SPEC	Definição
Name	O nome base do pacote, que deve combinar com o nome do arquivo SPEC.
Version	O número da versão upstream do software.
Release	O número de vezes que esta versão do software foi lançada. Normalmente, defina o valor inicial para 1%{?dist}, e aumente-o a cada novo lançamento do pacote. Redefinir para 1 quando um novo **Version** do software for construído.
Summary	Um breve resumo de uma linha do pacote.
License	A licença do software que está sendo embalado.
URL	A URL completa para mais informações sobre o programa. Na maioria das vezes este é o site do projeto upstream para o software que está sendo empacotado.
Source0	Caminho ou URL para o arquivo comprimido do código-fonte a montante (não corrigido, as correções são tratadas em outro lugar). Isto deve apontar para um armazenamento acessível e confiável do arquivo, por exemplo, a página a montante e não o armazenamento local do empacotador. Se necessário, mais diretrizes do SourceX podem ser adicionadas, incrementando o número a cada vez, por exemplo: Fonte1, Fonte2, Fonte3, e assim por diante.
Patch	O nome do primeiro patch a ser aplicado ao código fonte, se necessário. A diretiva pode ser aplicada de duas maneiras: com ou sem números no final do Patch. Se nenhum número for dado, um é atribuído à entrada internamente. Também é possível dar os números explicitamente usando Patch0, Patch1, Patch2, Patch3, e assim por diante. Estes remendos podem ser aplicados um a um utilizando a macro %patch0, %patch1, %patch2 e assim por diante. As macros são aplicadas dentro da diretiva %prep na seção *Body* do arquivo SPEC da RPM. Alternativamente, é possível usar a macro topatch que aplica automaticamente todos os patches na ordem em que são dados no arquivo SPEC.

Diretriz SPEC	Definição
BuildArch	Se o pacote não for dependente da arquitetura, por exemplo, se for escrito inteiramente em uma linguagem de programação interpretada, defina isso para **BuildArch: noarch**. Se não for definido, o pacote herda automaticamente a Arquitetura da máquina sobre a qual é construído, por exemplo **x86_64**.
BuildRequires	Uma lista separada por vírgula ou espaço em branco dos pacotes necessários para construir o programa escrito em uma linguagem compilada. Pode haver múltiplas entradas de **BuildRequires**, cada uma em sua própria linha no arquivo SPEC.
Requires	Uma lista separada por vírgula ou espaço em branco dos pacotes requeridos pelo software para ser executado uma vez instalado. Pode haver múltiplas entradas de **Requires**, cada uma em sua própria linha no arquivo SPEC.
ExcludeArch	Se um software não puder operar em uma arquitetura de processador específica, você pode excluir essa arquitetura aqui.
Conflicts	**Conflicts** são inversos para **Requires**. Se houver um pacote compatível **Conflicts**, o pacote não pode ser instalado independentemente se a tag **Conflict** está no pacote que já foi instalado ou em um pacote que vai ser instalado.
Obsoletes	Esta diretiva altera a forma de trabalho das atualizações dependendo se o comando **rpm** é usado diretamente na linha de comando ou se a atualização é realizada por um solucionador de atualizações ou dependência. Quando usado em uma linha de comando, o RPM remove todos os pacotes que correspondem a pacotes obsoletos que estão sendo instalados. Ao usar uma atualização ou resolvedor de dependência, pacotes contendo **Obsoletes:** correspondente são adicionados como atualizações e substituem os pacotes correspondentes.
Provides	Se **Provides** for adicionado a um pacote, o pacote pode ser referido por outras dependências além de seu nome.

As diretrizes **Name**, **Version**, e **Release** compreendem o nome do arquivo do pacote RPM. Os mantenedores do pacote RPM e os administradores do sistema freqüentemente chamam estas três diretivas de **N-V-R** ou **NVR**, porque os nomes dos arquivos do pacote RPM têm o formato **NAME-VERSION-RELEASE**.

O exemplo a seguir mostra como obter a informação **NVR** para um pacote específico consultando o comando **rpm**.

Exemplo 3.1. Consultar as rpm para fornecer as informações NVR para o pacote bash

```
# rpm -q bash
bash-4.4.19-7.el8.x86_64
```

-

Aqui, **bash** é o nome do pacote, **4.4.19** é a versão, e **7.el8** é o lançamento. O marcador final é **x86_64**, que sinaliza a arquitetura. Ao contrário do **NVR**, o marcador de arquitetura não está sob controle direto do empacotador RPM, mas é definido pelo ambiente de construção **rpmbuild**. A exceção a isto é o pacote **noarch** independente da arquitetura.

3.1.4.2. Itens do corpo

Os itens usados no arquivo **Body section** do RPM SPEC estão listados na tabela abaixo.

Tabela 3.2. Itens utilizados na seção Corpo do arquivo SPEC da RPM

Diretriz SPEC	Definição
%description	Uma descrição completa do software embalado no RPM. Esta descrição pode abranger várias linhas e pode ser dividida em parágrafos.
%prep	Comando ou série de comandos para preparar o software a ser construído, por exemplo, desempacotar o arquivo em **Source0**. Esta diretiva pode conter um script de shell.
%build	Comando ou série de comandos para construir o software em código de máquina (para idiomas compilados) ou código de byte (para alguns idiomas interpretados).
%install	Comando ou série de comandos para copiar os artefatos de construção desejados do diretório **%builddir** (onde a construção acontece) para o diretório**%buildroot** (que contém a estrutura do diretório com os arquivos a serem empacotados). Isto geralmente significa copiar os arquivos de **~/rpmbuild/BUILD** para **~/rpmbuild/BUILDROOT** e criar os diretórios necessários em **~/rpmbuild/BUILDROOT**. Isto só é executado quando se cria um pacote, e não quando o usuário final instala o pacote. Veja Seção 3.2, "Trabalhando com arquivos SPEC" para detalhes.
%check	Comando ou série de comandos para testar o software. Isto normalmente inclui coisas tais como testes unitários.
%files	A lista de arquivos que serão instalados no sistema do usuário final.
%changelog	Um registro das mudanças que aconteceram com o pacote entre as diferentes construções do **Version** ou **Release**.

3.1.4.3. Itens avançados

O arquivo SPEC também pode conter itens avançados, tais como Scriptlets ou Triggers. Eles têm efeito em diferentes pontos durante o processo de instalação no sistema do usuário final, não no processo de construção.

3.1.5. BuildRoots

No contexto da embalagem RPM, **buildroot** é um ambiente chroot. Isto significa que os artefatos de construção são colocados aqui usando a mesma hierarquia do sistema de arquivos que a hierarquia futura no sistema do usuário final, com **buildroot** atuando como diretório raiz. A colocação dos

artefatos de construção deve estar de acordo com o padrão de hierarquia do sistema de arquivos do sistema do usuário final.

Os arquivos em **buildroot** são posteriormente colocados em um arquivo **cpio**, que se torna a parte principal do RPM. Quando o RPM é instalado no sistema do usuário final, estes arquivos são extraídos no diretório **root**, preservando a hierarquia correta.

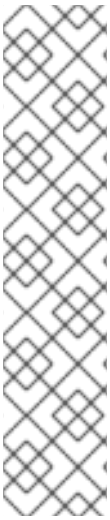
NOTA

A partir do Red Hat Enterprise Linux 6, o programa **rpmbuild** tem seus próprios padrões. Substituir estes padrões leva a vários problemas; portanto, a Red Hat não recomenda que você defina seu próprio valor desta macro. Você pode usar a macro **%{buildroot}** com os defaults do diretório **rpmbuild**.

3.1.6. Macros RPM

Uma macro rpm é uma substituição de texto reto que pode ser atribuída condicionalmente com base na avaliação opcional de uma declaração quando determinada funcionalidade embutida é utilizada. Portanto, o RPM pode realizar substituições de texto para você.

Um exemplo de uso é a referência ao software embalado *Version* várias vezes em um arquivo SPEC. Você define *Version* apenas uma vez na macro **%{version}**, e usa esta macro em todo o arquivo SPEC. Cada ocorrência será automaticamente substituída por *Version* que você definiu anteriormente.



NOTA

Se você vir uma macro não familiar, você pode avaliá-la com o seguinte comando:

```
$ rpm --eval %{_MACRO}
```

Avaliando as macros %{_bindir} e %{_libexecdir}

```
$ rpm --eval %{_bindir}
/usr/bin

$ rpm --eval %{_libexecdir}
/usr/libexec
```

Uma das macros mais usadas é a macro **%{?dist}**, que sinaliza qual distribuição é usada para a construção (etiqueta de distribuição).

```
# On a RHEL 8.x machine
$ rpm --eval %{?dist}
.el8
```

3.2. TRABALHANDO COM ARQUIVOS SPEC

Esta seção descreve como criar e modificar um arquivo SPEC.

Pré-requisitos

Esta seção utiliza os três exemplos de implementações do programa **Hello World!** que foram descritos em Seção 2.1.1, “Exemplos de código fonte” .

Cada um dos programas também está totalmente descrito na tabela abaixo.

Nome do software	Explicação do exemplo
bello	Um programa escrito em uma linguagem de programação interpretada em bruto. Ele demonstra quando o código fonte não precisa ser construído, mas apenas instalado. Se um binário pré-compilado precisar ser empacotado, você também pode usar este método, já que o binário também seria apenas um arquivo.
pello	Um programa escrito em uma linguagem de programação interpretada por bytes. Ele demonstra o byte-compilando o código fonte e instalando o bytecode - os arquivos pré-optimizados resultantes.
violoncelo	Um programa escrito em uma linguagem de programação nativamente compilada. Ele demonstra um processo comum de compilação do código fonte em código de máquina e instalação dos executáveis resultantes.

As implementações de **Hello World!** são:

- bello-0.1.tar.gz
- pello-0.1.2.tar.gz
- violoncelo-1.0.tar.gz
  - cello-output-first-patch.patch

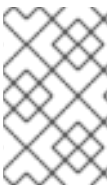
Como pré-requisito, estas implementações precisam ser colocadas no diretório **~/rpmbuild/SOURCES**.

3.2.1. Formas de criar um novo arquivo SPEC

Para empacotar novos softwares, você precisa criar um novo arquivo SPEC.

Há dois para conseguir isso:

- Escrever o novo arquivo SPEC manualmente a partir do zero
- Use o utilitário **rpmdev-newspec**
  Esta utilidade cria um arquivo SPEC despovoado, e você preenche as diretrizes e os campos necessários.



NOTA

Alguns editores de texto com foco no programador pré-popularam um novo arquivo **.spec** com seu próprio modelo SPEC. O utilitário **rpmdev-newspec** fornece um método de editor-agnóstico.

3.2.2. Criação de um novo arquivo SPEC com rpmdev-newspec

O procedimento a seguir mostra como criar um arquivo SPEC para cada um dos três programas acima mencionados **Hello World!** usando o utilitário **rpmdev-newspec**.

Procedimento

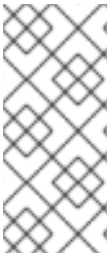
1. Mude para o diretório **~/rpmbuild/SPECS** e use o utilitário **rpmdev-newspec**:

```
$ cd ~/rpmbuild/SPECS

$ rpmdev-newspec bello
bello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec cello
cello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec pello
pello.spec created; type minimal, rpm version >= 4.11.
```

   O diretório **~/rpmbuild/SPECS/** contém agora três arquivos SPEC chamados **bello.spec**, **cello.spec**, e **pello.spec**.

fd. Examine os arquivos:

As diretrizes contidas nos arquivos representam as descritas na seção Seção 3.1.4, "O que é um arquivo SPEC". Nas seções seguintes, você preencherá uma seção específica nos arquivos de saída de **rpmdev-newspec**.



NOTA

O utilitário **rpmdev-newspec** não utiliza diretrizes ou convenções específicas para nenhuma distribuição Linux em particular. Entretanto, este documento tem como alvo o Red Hat Enterprise Linux, portanto a notação **%{buildroot}** é preferida em relação à notação **$RPM_BUILD_ROOT** ao referenciar o Buildroot do RPM para consistência com todas as outras macros definidas ou fornecidas em todo o arquivo SPEC.

3.2.3. Modificando um arquivo SPEC original para criar RPMs

O procedimento a seguir mostra como modificar o arquivo SPEC de saída fornecido por **rpmdev-newspec** para a criação das RPMs.

Pré-requisitos

Certifique-se disso:

- O código fonte do programa em particular foi colocado no diretório **~/rpmbuild/SOURCES/**.
- O arquivo não-povoado da SPEC **~/rpmbuild/SPECS/<name>.spec** foi criado pelo utilitário **rpmdev-newspec**.

Procedimento

1. Abra o modelo de saída do arquivo **~/rpmbuild/SPECS/<name>.spec** fornecido pelo utilitário **rpmdev-newspec**:

2. Povoar a primeira seção do arquivo da SPEC:
   A primeira seção inclui estas diretrizes que foram agrupadas em **rpmdev-newspec**:

   - **Name**

- **Version**
- **Release**
- **Summary**
  O **Name** já foi especificado como um argumento para **rpmdev-newspec**.

  Defina o **Version** para combinar com a versão de lançamento upstream do código fonte.

  O **Release** é automaticamente ajustado para **1%{?dist}**, que é inicialmente **1**. Aumente o valor inicial sempre que atualizar o pacote sem uma mudança no lançamento a montante **Version** - como quando se inclui um patch. Redefinir **Release** para **1** quando um novo lançamento upstream acontecer.

  O **Summary** é uma breve explicação de uma linha do que é este software.

3. Popular as diretrizes **License**, **URL**, e **Source0**:
   O campo **License** é a Licença de Software associada com o código fonte da versão upstream. O formato exato de como rotular o **License** em seu arquivo SPEC variará dependendo de quais diretrizes específicas de distribuição Linux baseadas em RPM você está seguindo.

   Por exemplo, você pode usar a GPLv3 .

   O campo **URL** fornece o URL para o site do software upstream. Para maior consistência, utilize a macro variável RPM de **%{name}**, e use o campo **https://example.com/%{name}**.

   O campo **Source0** fornece o URL para o código fonte do software upstream. Ele deve ser ligado diretamente à versão específica do software que está sendo empacotado. Note que as URLs de exemplo dadas nesta documentação incluem valores codificados que são possíveis sujeitos a mudanças no futuro. Da mesma forma, a versão de lançamento também pode mudar. Para simplificar estas possíveis mudanças futuras, use as macros **%{name}** e **%{version}**. Ao usá-las, você precisa atualizar apenas um campo no arquivo SPEC.

4. Povoar as diretrizes **BuildRequires**, **Requires** e **BuildArch**:
   BuildRequires especifica as dependências de tempo de construção para o pacote.

   Requires especifica as dependências de tempo de execução para o pacote.

   Este é um software escrito em uma linguagem de programação interpretada, sem extensões compiladas nativamente. Portanto, acrescente a diretiva **BuildArch** com o valor **noarch**. Isto diz ao RPM que este pacote não precisa estar vinculado à arquitetura do processador sobre o qual é construído.

5. Povoar as diretrizes **%description**, **%prep**, **%build**, **%install**, **%files**, e **%license**:
   Essas diretrizes podem ser consideradas como títulos de seção, pois são diretrizes que podem definir tarefas de múltiplas linhas, multi-instrução ou tarefas com scripts a serem realizadas.

   O **scription** é uma descrição mais longa e completa do software do que **Summary**, contendo um ou mais parágrafos.

   A seção **%prep** especifica como preparar o ambiente de construção. Isto geralmente envolve a expansão de arquivos comprimidos do código fonte, aplicação de correções e, potencialmente, análise das informações fornecidas no código fonte para uso em uma parte posterior do arquivo SPEC. Nesta seção, você pode usar a macro **%setup -q** embutida.

   A seção **%build** especifica como construir o software.

A seção **%install** contém instruções para **rpmbuild** sobre como instalar o software, uma vez que tenha sido construído, no diretório **BUILDROOT**.

Este diretório é um diretório de base chroot vazio, que se assemelha ao diretório raiz do usuário final. Aqui você pode criar quaisquer diretórios que contenham os arquivos instalados. Para criar tais diretórios, você pode usar as macros RPM sem ter que codificar os caminhos de forma rígida.

A seção **%files** especifica a lista de arquivos fornecidos por esta RPM e sua localização completa no sistema do usuário final.

Dentro desta seção, você pode indicar o papel de vários arquivos usando macros embutidos. Isto é útil para consultar os metadados do manifesto do arquivo de pacotes usando o comando[]**rpm**. Por exemplo, para indicar que o arquivo LICENSE é um arquivo de licença de software, use a macro **%license**.

6. A última seção, **%changelog**, é uma lista de entradas marcadas com dados para cada Versão-Release do pacote. Eles registram as mudanças de embalagem, não as mudanças de software. Exemplos de mudanças de embalagem: adicionar um patch, mudar o procedimento de construção na seção **%build**.
Siga este formato para a primeira linha:

   Comece com um caracter * seguido por **Day-of-Week Month Day Year Name Surname <email> - Version-Release**

   Siga este formato para a entrada da mudança propriamente dita:

   - Cada entrada de mudança pode conter vários itens, um para cada mudança.
   - Cada item começa em uma nova linha.
   - Cada item começa com um caracter **-**.

Agora você escreveu um arquivo completo da SPEC para o programa necessário.

Para exemplos de arquivos SPEC escritos em diferentes linguagens de programação, veja:

- Um arquivo SPEC de exemplo para um programa escrito em bash
- Um arquivo SPEC de exemplo para um programa escrito em Python
- Um exemplo de arquivo SPEC para um programa escrito em C

A construção do RPM a partir do arquivo da SPEC está descrita em Seção 3.3, “RPMs de construção” .

3.2.4. Um arquivo SPEC de exemplo para um programa escrito em bash

Esta seção mostra um arquivo SPEC de exemplo para o programa **bello** que foi escrito em bash. Para maiores informações sobre **bello**, veja Seção 2.1.1, “Exemplos de código fonte” .

Um arquivo SPEC de exemplo para o programa bello escrito em bash

```
Name:       bello
Version:    0.1
Release:    1%{?dist}
Summary:    Hello World example implemented in bash script
```

```
License:        GPLv3+
URL:            https://www.example.com/%{name}
Source0:        https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Requires:       bash

BuildArch:      noarch

%description
The long-tail description for our Hello World Example implemented in
bash script.

%prep
%setup -q

%build

%install

mkdir -p %{buildroot}/%{_bindir}

install -m 0755 %{name} %{buildroot}/%{_bindir}/%{name}

%files
%license LICENSE
%{_bindir}/%{name}

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First bello package
- Example second item in the changelog for version-release 0.1-1
```

A diretiva **BuildRequires**, que especifica as dependências de tempo de construção do pacote, foi excluída porque não há etapa de construção para **bello**. Bash é uma linguagem de programação interpretada em bruto, e os arquivos são apenas instalados até sua localização no sistema.

A diretiva **Requires**, que especifica as dependências de tempo de execução do pacote, inclui apenas **bash**, pois o script **bello** requer apenas o ambiente shell **bash** para ser executado.

A seção **%build**, que especifica como construir o software, está em branco, porque um **bash** não precisa ser construído.

Para instalar **bello** você só precisa criar o diretório de destino e instalar lá o arquivo de script executável **bash**. Portanto, você pode usar o comando **install** na seção **%install**. As macros RPM permitem fazer isso sem caminhos de codificação rígidos.

3.2.5. Um arquivo SPEC de exemplo para um programa escrito em Python

Esta seção mostra um arquivo SPEC de exemplo para o programa **pello** escrito na linguagem de programação Python. Para maiores informações sobre **pello**, veja Seção 2.1.1, “Exemplos de código fonte”.

Um arquivo SPEC de exemplo para o programa pello escrito em Python

```
Name:          pello
Version:       0.1.1
Release:       1%{?dist}
Summary:       Hello World example implemented in Python

License:       GPLv3+
URL:           https://www.example.com/%{name}
Source0:       https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

BuildRequires:  python
Requires:      python
Requires:      bash

BuildArch:     noarch

%description
The long-tail description for our Hello World Example implemented in Python.

%prep
%setup -q

%build

python -m compileall %{name}.py

%install

mkdir -p %{buildroot}/%{_bindir}
mkdir -p %{buildroot}/usr/lib/%{name}

cat > %{buildroot}/%{_bindir}/%{name} <← EOF
#!/bin/bash
/usr/bin/python /usr/lib/%{name}/%{name}.pyc
EOF

chmod 0755 %{buildroot}/%{_bindir}/%{name}

install -m 0644 %{name}.py* %{buildroot}/usr/lib/%{name}/

%files
%license LICENSE
%dir /usr/lib/%{name}/
%{_bindir}/%{name}
/usr/lib/%{name}/%{name}.py*

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1.1-1
  - First pello package
```

IMPORTANTE

O programa **pello** é escrito em uma linguagem interpretada por bytes. Portanto, o shebang não é aplicável porque o arquivo resultante não contém a entrada.

Como o shebang não é aplicável, você pode querer aplicar uma das seguintes abordagens:

- Criar um script de shell não compilado em bytes que chamará o executável.
- Fornecer um pouco do código Python que não seja compilado por bytes como ponto de entrada para a execução do programa.

Estas abordagens são úteis especialmente para grandes projetos de software com muitos milhares de linhas de código, onde o aumento de desempenho do código pré-compilado é considerável.

A diretiva **BuildRequires**, que especifica as dependências de tempo de construção para o pacote, inclui dois pacotes:

- O pacote **python** é necessário para realizar o processo de compilação de byte-compilação
- O pacote **bash** é necessário para executar o pequeno roteiro do ponto de entrada

A diretiva **Requires**, que especifica as dependências de tempo de execução para o pacote, inclui apenas o pacote **python**. O programa **pello** requer o pacote **python** para executar o código compilado pelo byte em tempo de execução.

A seção **%build**, que especifica como construir o software, corresponde ao fato de que o software é compilado por bytes.

Para instalar **pello**, você precisa criar um roteiro de embalagem porque o shebang não é aplicável em linguagens compiladas por bytes. Existem múltiplas opções para realizar isto, como por exemplo:

- Fazendo um roteiro separado e usando-o como uma diretriz separada **SourceX**.
- Criação do arquivo em linha no arquivo da SPEC.

Este exemplo mostra a criação de um roteiro de embalagem em linha no arquivo SPEC para demonstrar que o próprio arquivo SPEC é scriptable. Este script de invólucro executará o código Python compilado por bytes, usando um documento **here**.

A seção **%install** neste exemplo também corresponde ao fato de que você precisará instalar o arquivo compilado pelo byte em um diretório de biblioteca no sistema, de modo que ele possa ser acessado.

3.2.6. Um exemplo de arquivo SPEC para um programa escrito em C

Esta seção mostra um arquivo SPEC de exemplo para o programa **cello** que foi escrito na linguagem de programação C. Para maiores informações sobre **cello**, veja Seção 2.1.1, “Exemplos de código fonte” .

Um arquivo SPEC de exemplo para o programa cello escrito em C

```
Name:       cello
Version:    1.0
Release:    1%{?dist}
Summary:    Hello World example implemented in C
```

```
License:        GPLv3+
URL:            https://www.example.com/%{name}
Source0:        https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Patch0:         cello-output-first-patch.patch

BuildRequires:  gcc
BuildRequires:  make

%description
The long-tail description for our Hello World Example implemented in
C.

%prep
%setup -q

%patch0

%build
make %{?_smp_mflags}

%install
%make_install

%files
%license LICENSE
%{_bindir}/%{name}

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 1.0-1
- First cello package
```

A diretiva **BuildRequires**, que especifica as dependências de tempo de compilação para o pacote, inclui dois pacotes que são necessários para realizar o processo de compilação:

- O pacote **gcc**
- O pacote **make**

A diretiva **Requires**, que especifica as dependências de tempo de execução para o pacote, é omitida neste exemplo. Todos os requisitos de tempo de execução são tratados por **rpmbuild**, e o programa **cello** não requer nada fora das bibliotecas centrais do padrão C.

A seção **%build** reflete o fato de que neste exemplo foi escrito um **Makefile** para o programa **cello**, daí que o comando GNU make fornecido pelo utilitário **rpmdev-newspec** pode ser usado. Entretanto, é necessário remover a chamada para **%configure** porque não foi fornecido um script de configuração.

A instalação do programa **cello** pode ser realizada utilizando a macro **%make_install** que foi fornecida pelo comando **rpmdev-newspec**. Isto é possível porque o **Makefile** para o programa **cello** está disponível.

3.3. RPMS DE CONSTRUÇÃO

Esta seção descreve como construir um RPM após a criação de um arquivo SPEC para um programa.

As RPMs são construídas com o comando **rpmbuild**. Este comando espera um determinado diretório e estrutura de arquivos, que é o mesmo que a estrutura que foi criada pelo utilitário **rpmdev-setuptree**.

Diferentes casos de uso e resultados desejados requerem diferentes combinações de argumentos para o comando **rpmbuild**. Esta seção descreve os dois principais casos de uso:

- Fonte de construção RPMs
- Construindo RPMs binários

3.3.1. Fonte de construção RPMs

Este parágrafo é a introdução do módulo de procedimento: uma breve descrição do procedimento.

Pré-requisitos

Um arquivo SPEC para o programa que queremos empacotar já deve existir. Para mais informações sobre como criar arquivos SPEC, consulte Trabalhando com arquivos SPEC.

Procedimento

O procedimento a seguir descreve como construir uma fonte RPM.

- Execute o comando **rpmbuild** com o arquivo SPEC especificado:

  ```
  rpmbuild -bs SPECFILE
  ```

  Substituir *SPECFILE* pelo arquivo da SPEC. A opção **-bs** representa a fonte de construção.

O exemplo a seguir mostra fontes de construção RPMs para os projetos **bello**, **pello**, e **cello**.

Fonte de construção RPMs para bello, pello e violoncelo.

```
$ cd ~/rpmbuild/SPECS/

8$ rpmbuild -bs bello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm

$ rpmbuild -bs pello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm

$ rpmbuild -bs cello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
```

Etapas de verificação

- Certifique-se de que o diretório **rpmbuild/SRPMS** inclua as RPMs de origem resultantes. O diretório é uma parte da estrutura esperada por **rpmbuild**.

3.3.2. Construindo RPMs binários

Os seguintes métodos são viáveis para a construção de RPMs binários:

- Reconstruindo um RPM binário a partir de um RPM fonte
- Construção de um RPM binário a partir do arquivo da SPEC

- Construção de um RPM binário a partir de um RPM de origem

3.3.2.1. Reconstruindo um RPM binário a partir de um RPM fonte

O procedimento a seguir mostra como reconstruir um RPM binário a partir de um RPM fonte (SRPM).

Procedimento

- Para reconstruir **bello**, **pello** e **cello** a partir de suas SRPMs, executar:

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
[output truncated]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
[output truncated]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
[output truncated]
```

NOTA

A invocação de **rpmbuild --rebuild** envolve:

- Instalando o conteúdo do SRPM - o arquivo SPEC e o código fonte - no diretório **~/rpmbuild/**.
- Construção utilizando o conteúdo instalado.
- Remoção do arquivo SPEC e do código fonte.

Para reter o arquivo SPEC e o código fonte após a construção, você pode:

- Ao construir, use o comando **rpmbuild** com a opção **--recompile** ao invés da opção **--rebuild**.
- Instale os SRPMs usando estes comandos:

```
$ rpm -Uvh ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
Updating / installing…
   1:bello-0.1-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
Updating / installing…
…1:pello-0.1.2-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
Updating / installing…
…1:cello-1.0-1.el8              [100%]
```

A saída gerada ao criar um RPM binário é verbosa, o que é útil para a depuração. A saída varia para diferentes exemplos e corresponde a seus arquivos SPEC.

Os RPMs binários resultantes estão no diretório **~/rpmbuild/RPMS/YOURARCH** onde **YOURARCH** é sua arquitetura ou no diretório **~/rpmbuild/RPMS/noarch/**, se o pacote não for específico da arquitetura.

3.3.2.2. Construção de um RPM binário a partir do arquivo da SPEC

O seguinte procedimento mostra como construir **bello**, **pello**, e **cello** RPMs binários a partir de seus arquivos SPEC.

Procedimento

- Execute o comando **rpmbuild** com a opção **bb**:

```
$ rpmbuild -bb ~/rpmbuild/SPECS/bello.spec

$ rpmbuild -bb ~/rpmbuild/SPECS/pello.spec

$ rpmbuild -bb ~/rpmbuild/SPECS/cello.spec
```

3.3.2.3. Construindo RPMs a partir de RPMs de origem

Também é possível construir qualquer tipo de RPM a partir de uma fonte RPM. Para fazer isso, use o seguinte procedimento.

Procedimento

- Execute o comando **rpmbuild** com uma das opções abaixo e com o pacote fonte especificado:

```
# rpmbuild {-ra|-rb|-rp|-rc|-ri|-rl|-rs} [rpmbuild-options] SOURCEPACKAGE
```

Recursos adicionais

Para mais detalhes sobre a construção de RPMs a partir de RPMs de origem, consulte a seção **BUILDING PACKAGES** na página de manual **rpmbuild(8)**.

3.4. VERIFICAÇÃO DA SANIDADE DOS RPMS

Depois de criar uma embalagem, verifique a qualidade da embalagem.

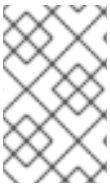
A principal ferramenta para verificar a qualidade da embalagem é o rpmlint.

A ferramenta **rpmlint** faz o seguinte:

- Melhora a capacidade de manutenção do RPM.
- Permite a verificação da sanidade através da análise estática do RPM.
- Possibilita a verificação de erros através da análise estática da RPM.

A ferramenta **rpmlint** pode verificar RPMs binários, RPMs de origem (SRPMs) e arquivos SPEC, portanto é útil para todas as etapas de embalagem, como mostrado nos exemplos a seguir.

Note que **rpmlint** tem diretrizes muito rigorosas; por isso, às vezes é aceitável pular alguns de seus erros e avisos, como mostrado nos exemplos a seguir.



NOTA

Nos exemplos a seguir, **rpmlint** é executado sem nenhuma opção, o que produz uma saída não-verbose. Para explicações detalhadas de cada erro ou aviso, você pode rodar **rpmlint -i** em seu lugar.

3.4.1. Verificando a sanidade do Bello

Esta seção mostra possíveis avisos e erros que podem ocorrer ao verificar a sanidade RPM no exemplo do arquivo bello SPEC e bello binário RPM.

3.4.1.1. Verificação do Arquivo Bello SPEC

Exemplo 3.2. Saída da execução do comando rpmlint no arquivo SPEC para o bello

```
$ rpmlint bello.spec
bello.spec: W: invalid-url Source0: https://www.example.com/bello/releases/bello-0.1.tar.gz HTTP
Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

Para **bello.spec**, há apenas um aviso, que diz que a URL listada na diretiva **Source0** é inalcançável. Isto é esperado, pois o URL especificado **example.com** não existe. Presumindo que esperamos que este URL funcione no futuro, podemos ignorar esta advertência.

Exemplo 3.3. Saída da execução do comando rpmlint no SRPM para o bello

```
$ rpmlint ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
bello.src: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.src: W: invalid-url Source0: https://www.example.com/bello/releases/bello-0.1.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

Para a SRPM **bello**, há um novo aviso, que diz que a URL especificada na diretiva **URL** é inalcançável. Supondo que o link estará funcionando no futuro, podemos ignorar esta advertência.

3.4.1.2. Verificação do RPM binário bello

Ao verificar os RPMs binários, **rpmlint** verifica os seguintes itens:

- Documentação
- Páginas do manual
- Uso consistente do padrão de hierarquia do sistema de arquivos

Exemplo 3.4. Saída da execução do comando rpmlint no RPM binário para bello

```
$ rpmlint ~/rpmbuild/RPMS/noarch/bello-0.1-1.el8.noarch.rpm
bello.noarch: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.noarch: W: no-documentation
```

```
bello.noarch: W: no-manual-page-for-binary bello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

Os avisos **no-documentation** e **no-manual-page-for-binary** dizem que o RPM não tem documentação ou páginas de manual, porque não fornecemos nenhuma. Além das advertências acima, o RPM passou por **rpmlint** verificações.

3.4.2. Verificação da sanidade do pello

Esta seção mostra possíveis avisos e erros que podem ocorrer ao verificar a sanidade RPM no exemplo do arquivo SPEC pello e RPM pello binário.

3.4.2.1. Verificação do arquivo da SPEC Pello

Exemplo 3.5. Saída da execução do comando rpmlint no arquivo SPEC para pello

```
$ rpmlint pello.spec
pello.spec:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.spec:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.spec:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.spec:43: E: hardcoded-library-path in /usr/lib/%{name}/
pello.spec:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.spec: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.2.tar.gz
HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 5 errors, 1 warnings.
```

O aviso **invalid-url Source0** diz que a URL listada na diretiva **Source0** é inalcançável. Isto é esperado, porque o URL especificado **example.com** não existe. Presumindo que este URL irá funcionar no futuro, você pode ignorar este aviso.

Os erros de **hardcoded-library-path** sugerem o uso da macro **%{_libdir}** em vez de codificar rigidamente o caminho da biblioteca. Para este exemplo, você pode ignorar estes erros com segurança. Entretanto, para pacotes que entram em produção, certifique-se de verificar todos os erros cuidadosamente.

Exemplo 3.6. Saída da execução do comando rpmlint no SRPM para pello

```
$ rpmlint ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
pello.src: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.src:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.src:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.src:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.src:43: E: hardcoded-library-path in /usr/lib/%{name}/
pello.src:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.src: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.2.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 5 errors, 2 warnings.
```

O novo erro **invalid-url URL** aqui é sobre a diretiva **URL**, que é inalcançável. Assumindo que a URL será válida no futuro, você pode ignorar este erro com segurança.

3.4.2.2. Verificação do RPM binário do pello

Ao verificar os RPMs binários, **rpmlint** verifica os seguintes itens:

- Documentação
- Páginas do manual
- Uso consistente da Hierarquia de Sistemas de Arquivos Padrão

Exemplo 3.7. Saída da execução do comando rpmlint no RPM binário para pello

```
$ rpmlint ~/rpmbuild/RPMS/noarch/pello-0.1.2-1.el8.noarch.rpm
pello.noarch: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.noarch: W: only-non-binary-in-usr-lib
pello.noarch: W: no-documentation
pello.noarch: E: non-executable-script /usr/lib/pello/pello.py 0644L /usr/bin/env
pello.noarch: W: no-manual-page-for-binary pello
1 packages and 0 specfiles checked; 1 errors, 4 warnings.
```

Os avisos **no-documentation** e **no-manual-page-for-binary** dizem que o RPM não tem documentação ou páginas de manual, porque você não forneceu nenhuma.

O aviso **only-non-binary-in-usr-lib** diz que você forneceu apenas artefatos não binários em **/usr/lib/**. Este diretório é normalmente reservado para arquivos de objetos compartilhados, que são arquivos binários. Portanto, **rpmlint** espera que pelo menos um ou mais arquivos no diretório **/usr/lib/** sejam binários.

Este é um exemplo de uma verificação de conformidade com a norma da Hierarquia de Sistemas de Arquivos (Filesystem Hierarchy Standard) em **rpmlint**. Normalmente, use macros RPM para garantir a correta colocação dos arquivos. Por causa deste exemplo, você pode ignorar com segurança este aviso.

O erro **non-executable-script** avisa que o arquivo **/usr/lib/pello/pello.py** não tem permissões de execução. A ferramenta **rpmlint** espera que o arquivo seja executável, pois o arquivo contém o shebang. Para o propósito deste exemplo, você pode deixar este arquivo sem executar as permissões e ignorar este erro.

Além das advertências e erros acima, o RPM passou no site **rpmlint**.

3.4.3. Verificação da sanidade do violoncelo

Esta seção mostra possíveis avisos e erros que podem ocorrer ao verificar a sanidade RPM no exemplo do arquivo SPEC do violoncelo e RPM do pello binário.

3.4.3.1. Verificação do arquivo SPEC do violoncelo

Exemplo 3.8. Saída da execução do comando rpmlint no arquivo SPEC para violoncelo

```
$ rpmlint ~/rpmbuild/SPECS/cello.spec
/home/admiller/rpmbuild/SPECS/cello.spec: W: invalid-url Source0:
https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

Para **cello.spec**, há apenas um aviso, que diz que a URL listada na diretiva **Source0** é inalcançável. Isto é esperado, pois o URL especificado **example.com** não existe. Presumindo que este URL irá funcionar no futuro, você pode ignorar este aviso.

Exemplo 3.9. Saída da execução do comando rpmlint no SRPM para violoncelo

```
$ rpmlint ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
cello.src: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.src: W: invalid-url Source0: https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

Para a SRPM **cello**, há um novo aviso, que diz que a URL especificada na diretiva **URL** é inalcançável. Supondo que o link estará funcionando no futuro, você pode ignorar esta advertência.

3.4.3.2. Verificação do RPM binário do violoncelo

Ao verificar os RPMs binários, **rpmlint** verifica os seguintes itens:

- Documentação
- Páginas do manual
- Uso consistente do padrão de hierarquia do sistema de arquivos

Exemplo 3.10. Saída da execução do comando rpmlint no RPM binário para violoncelo

```
$ rpmlint ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el8.x86_64.rpm
cello.x86_64: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.x86_64: W: no-documentation
cello.x86_64: W: no-manual-page-for-binary cello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

Os avisos **no-documentation** e **no-manual-page-for-binary** dizem que ele RPM não tem documentação ou páginas de manual, porque você não forneceu nenhuma. Além das advertências acima, o RPM passou por **rpmlint** verificações.

3.5. REGISTRO DA ATIVIDADE RPM NO SYSLOG

Qualquer atividade ou transação RPM pode ser registrada pelo protocolo de registro do sistema (syslog).

Pré-requisitos

- Para permitir o registro das transações RPM no syslog, certifique-se de que o plug-in **syslog** esteja instalado no sistema.

```
# yum instalar rpm-plugin-syslog
```

NOTA

O local padrão para as mensagens do syslog é o arquivo /**var**/**log**/**messages**. Entretanto, você pode configurar o syslog para usar outro local para armazenar as mensagens.

Para ver as atualizações sobre a atividade de RPM, siga este procedimento.

Procedimento

1. Abra o arquivo que você configurou para armazenar as mensagens do syslog, ou se você usar a configuração padrão do syslog, abra o arquivo /**var**/**log**/**messages**.
2. Procure novas linhas incluindo a corda **[RPM]**.

assembly_archiving-rpms.adoc :contexto de arquivamento-pai-arquivamento-rpms: embalagem-software

3.6. EXTRAÇÃO DO CONTEÚDO RPM

Em casos particulares, por exemplo, se uma embalagem exigida pelo RPM for danificada, é necessário extrair o conteúdo da embalagem. Nestes casos, se uma instalação RPM ainda estiver funcionando apesar dos danos, você pode usar o utilitário **rpm2archive** para converter um arquivo **.rpm** em um arquivo tar para usar o conteúdo do pacote.

Esta seção descreve como converter uma carga útil de rpm para um arquivo de alcatrão.

NOTA

Se a instalação do RPM estiver gravemente danificada, você pode usar o utilitário **rpm2cpio** para converter o arquivo do pacote RPM para um arquivo cpio.

3.6.1. Conversão de RPMs em arquivos de alcatrão

Para converter pacotes de RPM para tar achives, você pode usar o utilitário **rpm2archive**.

Procedimento

- Execute o seguinte comando:

```
$ rpm2archive file.rpm
```

O arquivo resultante tem o sufixo **.tgz**. Por exemplo, para arquivar o pacote **bash**:

```
$ rpm2archive bash-4.4.19-6.el8.x86_64.rpm
$ bash-4.4.19-6.el8.x86_64.rpm.tgz
bash-4.4.19-6.el8.x86_64.rpm.tgz
```

CAPÍTULO 4. TÓPICOS AVANÇADOS

Esta seção cobre tópicos que estão além do escopo do tutorial introdutório, mas que são úteis em embalagens RPM do mundo real.

4.1. ASSINATURA DE PACOTES

Os pacotes são assinados para garantir que nenhum terceiro possa alterar seu conteúdo. Um usuário pode adicionar uma camada adicional de segurança usando o protocolo HTTPS ao fazer o download do pacote.

Há três maneiras de assinar um pacote:

- Seção 4.1.2, “Acrescentar uma assinatura a um pacote já existente” .
- Seção 4.1.5, “Substituindo a assinatura em um pacote já existente” .
- Seção 4.1.6, “Assinatura de um pacote no momento da construção” .

Para poder assinar um pacote, você precisa criar uma chave GNU Privacy Guard (GPG), conforme descrito em Seção 4.1.1, “Criando uma chave GPG” .

4.1.1. Criando uma chave GPG

Procedimento

1. Gerar um par de chaves GNU Privacy Guard (GPG):

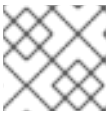
```
# gpg --gen-key
```

2. Confirmar e ver a chave gerada:

```
# gpg --list-keys
```

3. Exportar a chave pública:

```
# gpg - export -a '<Key_name>' > RPM-GPG-KEY-pmanager
```



NOTA

Inclua o nome real que você selecionou para a chave em vez de <Key_name>.

4. Importar a chave pública exportada para um banco de dados RPM:

```
# rpm --importar RPM-GPG-KEY-pmanager
```

4.1.2. Acrescentar uma assinatura a um pacote já existente

Esta seção descreve o caso mais usual quando um pacote é construído sem uma assinatura. A assinatura é adicionada imediatamente antes do lançamento do pacote.

Para adicionar uma assinatura a um pacote, use a opção **--addsign** fornecida pelo pacote **rpm-sign**.

Ter mais de uma assinatura permite registrar o caminho de propriedade do pacote desde o construtor do pacote até o usuário final.

Procedimento

- Adicione uma assinatura a um pacote:

```
$ rpm --blather-addsign-7.9-1.x86_64.rpm
```

NOTA

Você deve inserir a senha para desbloquear a chave secreta para a assinatura.

4.1.3. Verificação das assinaturas de um pacote com múltiplas assinaturas

Procedimento

- Para verificar as assinaturas de um pacote com múltiplas assinaturas, execute o seguinte:

```
$ rpm --checksig blather-7.9-1.x86_64.rpm
blather-7.9-1.x86_64.rpm: size pgp pgp md5 OK
```

As duas cordas **pgp** na saída do comando **rpm --checksig** mostram que o pacote foi assinado duas vezes.

4.1.4. Um exemplo prático de como adicionar uma assinatura a um pacote já existente

Esta seção descreve um exemplo de situação em que adicionar uma assinatura a um pacote já existente pode ser útil.

Uma divisão de uma empresa cria um pacote e o assina com a chave da divisão. A sede da empresa então verifica a assinatura do pacote e adiciona a assinatura corporativa ao pacote, declarando que o pacote assinado é autêntico.

Com duas assinaturas, a embalagem chega a um varejista. O varejista verifica as assinaturas e, se corresponderem, adiciona também sua assinatura.

O pacote agora chega a uma empresa que quer implantar o pacote. Após verificar cada assinatura no pacote, eles sabem que se trata de uma cópia autêntica. Dependendo dos controles internos da empresa implantadora, eles podem optar por adicionar sua própria assinatura, para informar seus funcionários que o pacote recebeu sua aprovação corporativa

4.1.5. Substituindo a assinatura em um pacote já existente

Este procedimento descreve como mudar a chave pública sem ter que reconstruir cada pacote.

Procedimento

- Para mudar a chave pública, execute o seguinte:

```
$ rpm --reinscrição blather-7.9-1.x86_64.rpm
```

NOTA

Você deve inserir a senha para desbloquear a chave secreta para a assinatura.

A opção **--resign** também permite alterar a chave pública para vários pacotes, como mostrado no procedimento a seguir.

Procedimento

- Para mudar a chave pública para vários pacotes, execute:

```
$ rpm - resignar b*.rpm
```

NOTA

Você deve inserir a senha para desbloquear a chave secreta para a assinatura.

4.1.6. Assinatura de um pacote no momento da construção

Procedimento

1. Construa o pacote com o comando **rpmbuild**:

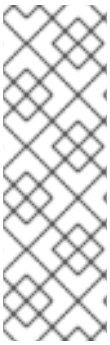
```
R$ rpmbuild blather-7.9.spec
```

2. Assine o pacote com o comando **rpmsign** usando a opção **--addsign**:

```
$ rpmsign --addsign blather-7.9-1.x86_64.rpm
```

3. Opcionalmente, verificar a assinatura de um pacote:

```
$ rpm --checksig blather-7.9-1.x86_64.rpm
blather-7.9-1.x86_64.rpm: size pgp md5 OK
```



NOTA

Ao construir e assinar vários pacotes, use a seguinte sintaxe para evitar a entrada da senha Pretty Good Privacy (PGP) várias vezes.

```
$ rpmbuild -ba --sign b*.spec
```

Observe que você deve digitar a senha para desbloquear a chave secreta para a assinatura.

4.2. MAIS SOBRE MACROS

Esta seção cobre Macros RPM embutidos selecionados. Para uma lista exaustiva de tais macros, consulte a Documentação RPM.

4.2.1. Definindo suas próprias macros

A seção seguinte descreve como criar uma macro personalizada.

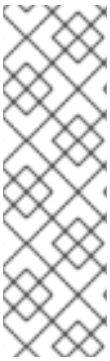
Procedimento

- Inclua a seguinte linha no arquivo RPM SPEC:

```
%global <nome INTERNGREGUNA-[(opts)] <corpo>[(opts)] <corpo-
```

Todo o espaço em branco ao redor de \ é removido. O nome pode ser composto de caracteres alfanuméricos e o caractere _ e deve ter pelo menos 3 caracteres de comprimento. A inclusão do campo **(opts)** é opcional:

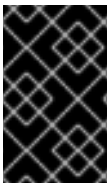
- **Simple** macros não contém o campo **(opts)**. Neste caso, apenas a expansão recursiva das macros é realizada.
- **Parametrized** macros contém o campo **(opts)**. A seqüência **opts** entre parênteses é passada para **getopt(3)** para **argc**/**argv** processamento no início de uma invocação macro.



NOTA

Os arquivos RPM SPEC mais antigos usam o padrão macro **fine <name> <body>**. As diferenças entre as macros **fine** e **%global** são as seguintes:

- **fine** tem alcance local. Ela se aplica a uma parte específica de um arquivo da SPEC. O corpo de uma macro **fine** é expandido quando usada.
- **%global** tem escopo global. Aplica-se a todo um arquivo da SPEC. O corpo de uma macro **%global** é expandido no momento da definição.



IMPORTANTE

As macros são avaliadas mesmo se forem comentadas ou se o nome da macro for dado na seção **%changelog** do arquivo da SPEC. Para comentar uma macro, use **%%**. Por exemplo: **%%global**.

Recursos adicionais

Para informações abrangentes sobre as capacidades de macros, consulte a Documentação RPM.

4.2.2. Usando a macro %setup

Esta seção descreve como construir pacotes com tarballs de código fonte usando diferentes variantes da macro **%setup**. Note que as variantes da macro podem ser combinadas A saída **rpmbuild** ilustra o comportamento padrão da macro **%setup**. No início de cada fase, a saída da macro **Executing(%...)**, como mostrado no exemplo abaixo.

Exemplo 4.1. Exemplo %setup macro output

```
Executando(%prep): /bin/sh -e /var/tmp/rpm-tmp.DhddsG
```

A saída do invólucro é definida com **set -x** habilitado. Para ver o conteúdo de **/var/tmp/rpm-tmp.DhddsG**, use a opção **--debug** porque **rpmbuild** apaga arquivos temporários após uma construção bem sucedida. Isto exibe a configuração das variáveis de ambiente seguida, por exemplo:

```
cd '/builddir/build/BUILD'
rm -rf 'cello-1.0'
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xof -
STATUS=$?
if [ $STATUS -ne 0 ]; then
  exit $STATUS
fi
cd 'cello-1.0'
/usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
```

A macro. **%setup**:

- Assegura que estamos trabalhando no diretório correto.
- Remove os resíduos de construções anteriores.
- Desembala o tarball de origem.
- Estabelece alguns privilégios padrão.

4.2.2.1. Usando a macro %setup -q

A opção **-q** limita a verbosidade da macro **%setup**. Somente **tar -xof** é executado ao invés de **tar -xvvof**. Use esta opção como a primeira opção.

4.2.2.2. Usando a %setup -n macro

A opção **-n** é usada para especificar o nome do diretório do tarball expandido.

Isto é usado em casos em que o diretório de tarball expandido tem um nome diferente do esperado (**%{name}-%{version}**), o que pode levar a um erro da macro **%setup**.

Por exemplo, se o nome do pacote é **cello**, mas o código fonte está arquivado em **hello-1.0.tgz** e contém o diretório **hello**/, o conteúdo do arquivo SPEC precisa ser o seguinte:

```
Name: cello
Source0: https://example.com/%{name}/release/hello-%{version}.tar.gz
…
%prep
%setup -n hello
```

4.2.2.3. Usando a %setup -c macro

A opção **-c** é utilizada se o código fonte tarball não contiver nenhum subdiretório e, após desempacotar, os arquivos de um arquivo preenchem o diretório atual.

A opção **-c** cria então o diretório e entra na expansão do arquivo, como mostrado abaixo:

```
/usr/bin/mkdir -p cello-1.0
cd 'cello-1.0'
```

O diretório não é alterado após a expansão do arquivo.

4.2.2.4. Usando a %setup -D e %setup -T macros

A opção **-D** desativa a exclusão do diretório do código fonte, e é particularmente útil se a macro **%setup** for usada várias vezes. Com a opção **-D**, as seguintes linhas não são utilizadas:

```
rm -rf 'cello-1.0'
```

A opção **-T** desativa a expansão do tarball do código fonte, removendo a seguinte linha do script:

```
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xvvof -
```

4.2.2.5. Usando as %setup -a e %setup -b macros

As opções **-a** e **-b** expandem fontes específicas:

A opção **-b** significa **before**, e expande fontes específicas antes de entrar no diretório de trabalho. A opção **-a** significa **after**, e expande essas fontes após a entrada. Seus argumentos são os números das fontes do preâmbulo do arquivo SPEC.

No exemplo a seguir, o arquivo **cello-1.0.tar.gz** contém um diretório **examples** vazio. Os exemplos são enviados em um tarball **examples.tar.gz** separado e se expandem para o diretório com o mesmo nome. Neste caso, use **-a 1**, se você quiser expandir **Source1** após entrar no diretório de trabalho:

```
Source0: https://example.com/%{name}/release/%{name}-%{version}.tar.gz
Source1: examples.tar.gz
…
%prep
%setup -a 1
```

No exemplo a seguir, exemplos são fornecidos em um tarball **cello-1.0-examples.tar.gz** separado, que se expande para **cello-1.0/examples**. Neste caso, use **-b 1**, para expandir **Source1** antes de entrar no diretório de trabalho:

```
Source0: https://example.com/%{name}/release/%{name}-%{version}.tar.gz
Source1: %{name}-%{version}-examples.tar.gz
…
%prep
%setup -b 1
```

4.2.3. Macros RPM comuns na seção les

Esta seção lista as Macros RPM avançadas que são necessárias na seção **%files** de um arquivo SPEC.

Tabela 4.1. Macros RPM avançados na seção%files

Macro	Definição
%license	A macro identifica o arquivo listado como um arquivo LICENSE e ele será instalado e rotulado como tal pelo RPM. Exemplo **%license LICENSE**

Macro	Definição
%doc	A macro identifica um arquivo listado como documentação e ele será instalado e rotulado como tal pelo RPM. A macro é usada para documentação sobre o software embalado e também para exemplos de código e vários itens que o acompanham. Caso sejam incluídos exemplos de código, deve-se ter o cuidado de remover o modo executável do arquivo. Exemplo **%doc README**
%dir	A macro assegura que o caminho é um diretório de propriedade desta RPM. Isto é importante para que o arquivo RPM se manifeste com precisão sobre quais diretórios devem ser limpos na desinstalação. Exemplo **%dir %{_libdir}/%{name}**
%config(noreplace)	A macro assegura que o seguinte arquivo é um arquivo de configuração e, portanto, não deve ser sobrescrito (ou substituído) em uma instalação ou atualização de pacote se o arquivo tiver sido modificado a partir do checksum da instalação original. Se houver uma mudança, o arquivo será criado com **.rpmnew** anexado ao final do nome do arquivo na atualização ou instalação para que o arquivo pré-existente ou modificado no sistema alvo não seja modificado. Exemplo **%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

4.2.4. Exibição das macros embutidas

O Red Hat Enterprise Linux fornece múltiplas macros RPM embutidas.

Procedimento

1. Para exibir todas as macros RPM embutidas, execute:

```
rpm --showrc
```

NOTA

A produção é bastante grande. Para reduzir o resultado, use o comando acima com o comando **grep**.

2. Para encontrar informações sobre as macros RPMs para a versão de seu sistema de RPM, execute:

```
rpm -ql rpm
```

NOTA

As macros RPM são os arquivos intitulados **macros** na estrutura do diretório de saída.

4.2.5. Macros de distribuição RPM

Diferentes distribuições fornecem diferentes conjuntos de macros RPM recomendadas com base na implementação da linguagem do software a ser empacotado ou nas diretrizes específicas da distribuição.

Os conjuntos de macros RPM recomendados são freqüentemente fornecidos como pacotes RPM, prontos para serem instalados com o gerenciador de pacotes **yum**.

Uma vez instalados, os arquivos de macro podem ser encontrados no diretório **/usr/lib/rpm/macros.d/**.

Para exibir as definições de macro RPM em bruto, execute:

```
rpm --showrc
```

A saída acima mostra as definições de macro de RPM bruto.

Para determinar o que uma macro faz e como pode ser útil ao embalar RPMs, execute o comando **rpm --eval** com o nome da macro usada como seu argumento:

```
rpm --eval %{_MACRO}
```

Para mais informações, consulte a página de manual **rpm**.

4.2.5.1. Criando macros personalizadas

Você pode substituir as macros de distribuição no arquivo **~/.rpmmacros** com suas macros personalizadas. Qualquer mudança que você fizer afeta cada construção em sua máquina.

ATENÇÃO

A definição de qualquer nova macros no arquivo **~/.rpmmacros** não é recomendada. Tais macros não estariam presentes em outras máquinas, onde os usuários podem querer tentar reconstruir seu pacote.

Para sobrepor uma macro, execute :

```
%_topdir /opt/some/trabalho/diretório/rpmbuild
```

Você pode criar o diretório a partir do exemplo acima, incluindo todos os subdiretórios através do utilitário **rpmdev-setuptree**. O valor desta macro é, por padrão, **~/rpmbuild**.

```
%_smp_mflags -l3
```

A macro acima é freqüentemente usada para passar para Makefile, por exemplo **make %{?_smp_mflags}**, e para definir uma série de processos simultâneos durante a fase de construção. Por padrão, ela é definida para **-jX**, onde **X** é um número de núcleos. Se você alterar o número de núcleos, você pode acelerar ou retardar uma construção de pacotes.

4.3. EPOCH, SCRIPTLETS E TRIGGERS

Esta seção abrange **Epoch**, **Scriptlets**, e **Triggers**, que representam diretrizes avançadas para arquivos RMP SPEC.

Todas essas diretrizes influenciam não apenas o arquivo SPEC, mas também a máquina final na qual a RPM resultante é instalada.

4.3.1. A Diretiva Epoch

A diretiva **Epoch** permite definir as dependências ponderadas com base nos números de versão.

Se esta diretiva não estiver listada no arquivo RPM SPEC, a diretiva **Epoch** não está definida de forma alguma. Isto é contrário à crença comum de que não definir **Epoch** resulta em um **Epoch** de 0. Entretanto, o utilitário YUM trata um **Epoch** não definido como o mesmo que um **Epoch** de 0 para fins de depsolução.

Entretanto, a lista **Epoch** em um arquivo SPEC é geralmente omitida porque na maioria dos casos a introdução de um valor **Epoch** distorce o comportamento esperado do RPM ao comparar versões de pacotes.

> **Exemplo 4.2. Usando a Epoch**
>
> Se você instalar o pacote **foobar** com **Epoch: 1** e **Version: 1.0**, e outro pacote **foobar** com **Version: 2.0** mas sem a diretiva **Epoch**, a nova versão nunca será considerada uma atualização. A razão é que a versão **Epoch** é preferida ao tradicional marcador **Name-Version-Release** que significa a versão para pacotes RPM.

O uso do site **Epoch** é, portanto, bastante raro. No entanto, **Epoch** é normalmente usado para resolver um problema de pedidos de atualização. A questão pode aparecer como um efeito colateral da mudança a montante nos esquemas de números de versão do software ou versões incorporando caracteres alfabéticos que nem sempre podem ser comparados de forma confiável com base na codificação.

4.3.2. Scriptlets

Scriptlets são uma série de diretrizes RPM que são executadas antes ou depois que os pacotes são instalados ou excluídos.

Use **Scriptlets** somente para tarefas que não podem ser feitas no momento da construção ou em um roteiro de inicialização.

4.3.2.1. Diretrizes dos Scriptlets

Existe um conjunto de diretrizes comuns no site **Scriptlet**. Elas são semelhantes aos cabeçalhos da seção de arquivos da SPEC, tais como **%build** ou **%install**. Elas são definidas por segmentos de código de múltiplas linhas, que muitas vezes são escritas como um script padrão POSIX shell. Entretanto, elas também podem ser escritas em outras linguagens de programação que o RPM para a distribuição da máquina alvo aceita. A documentação do RPM inclui uma lista exaustiva das linguagens disponíveis.

A tabela a seguir inclui as diretrizes **Scriptlet** listadas em sua ordem de execução. Note que um pacote contendo os scripts é instalado entre a diretiva **%pre** e **%post**, e é desinstalado entre as diretivas **%preun** e **%postun**.

Tabela 4.2. Diretrizes do Scriptlet

Diretiva	Definição
%pretrans	Scriptlet que é executado imediatamente antes da instalação ou remoção de qualquer pacote.
%pre	Scriptlet que é executado imediatamente antes da instalação do pacote no sistema alvo.
%post	Scriptlet que é executado logo após a instalação do pacote no sistema alvo.
%preun	Scriptlet que é executado pouco antes de desinstalar o pacote do sistema alvo.
%postun	Scriptlet que é executado logo após a desinstalação do pacote do sistema alvo.
%posttrans	Scriptlet que é executado no final da transação.

4.3.2.2. Desligando a execução de um scriptlet

Para desativar a execução de qualquer scriptlet, use o comando **rpm** junto com a opção **--no_scriptlet_name_**.

Procedimento

- Por exemplo, para desligar a execução dos scriptlets **%pretrans**, execute:

```
# rpm --nopretrans
```

  Você também pode usar a opção **-- noscripts**, que é equivalente a todas as opções a seguir:

  - **--nopre**
  - **--nopost**
  - **--nopreun**
  - **--nopostun**
  - **--nopretrans**
  - **--noposttrans**

Recursos adicionais

- Para mais detalhes, consulte a página de manual **rpm(8)**.

4.3.2.3. Macros de Scriptlets

As diretrizes **Scriptlets** também trabalham com macros RPM.

O exemplo seguinte mostra o uso da macro systemd scriptlet, que garante que o systemd seja notificado sobre um novo arquivo unitário.

```
$ rpm --showrc | grep systemd
-14: transaction_systemd_inhibit %{plugindir}/systemd_inhibit.so
-14: _journalcatalogdir /usr/lib/systemd/catalog
-14: _presetdir /usr/lib/systemd/system-preset
-14: _unitdir   /usr/lib/systemd/system
-14: _userunitdir       /usr/lib/systemd/user
/usr/lib/systemd/systemd-binfmt %{?} >/dev/null 2>&1 || : /usr/lib/systemd/systemd-sysctl %{?}
>/dev/null 2>&1 || :
-14: systemd_post
-14: systemd_postun
-14: systemd_postun_with_restart
-14: systemd_preun
-14: systemd_requires
Requires(post): systemd
Requires(preun): systemd
Requires(postun): systemd
-14: systemd_user_post  %systemd_post --user --global %{?} -14: systemd_user_postun %{nil} -
14: systemd_user_postun_with_restart %{nil} -14: systemd_user_preun systemd-sysusers %
{?} >/dev/null 2>&1 || :
echo %{?} | systemd-sysusers - >/dev/null 2>&1 || : systemd-tmpfiles --create %{?} >/dev/null
2>&1 || :

$ rpm --eval %{systemd_post}

if [ $1 -eq 1 ] ; then
        # Initial installation
        systemctl preset  >/dev/null 2>&1 || :
fi

$ rpm --eval %{systemd_postun}

systemctl daemon-reload >/dev/null 2>&1 || :

$ rpm --eval %{systemd_preun}

if [ $1 -eq 0 ] ; then
        # Package removal, not upgrade
        systemctl --no-reload disable  > /dev/null 2>&1 || :
        systemctl stop  > /dev/null 2>&1 || :
fi
```

4.3.3. As diretrizes de Gatilhos

Triggers são diretrizes RPM que fornecem um método para interação durante a instalação e desinstalação de pacotes.

ATENÇÃO

Triggers pode ser executado em um momento inesperado, por exemplo, na atualização do pacote contendo. **Triggers** são difíceis de depurar, portanto precisam ser implementados de forma robusta para que não quebrem nada quando executados inesperadamente. Por estas razões, a Red Hat recomenda minimizar o uso do **Triggers**.

A ordem de execução e os detalhes para cada uma das existentes **Triggers** estão listados abaixo:

```
all-%pretrans
…
any-%triggerprein (%triggerprein from other packages set off by new install)
new-%triggerprein
new-%pre      for new version of package being installed
…             (all new files are installed)
new-%post     for new version of package being installed

any-%triggerin (%triggerin from other packages set off by new install)
new-%triggerin
old-%triggerun
any-%triggerun (%triggerun from other packages set off by old uninstall)

old-%preun    for old version of package being removed
…             (all old files are removed)
old-%postun   for old version of package being removed

old-%triggerpostun
any-%triggerpostun (%triggerpostun from other packages set off by old un
          install)
…
all-%posttrans
```

Os itens acima podem ser encontrados no arquivo **/usr/share/doc/rpm-4.*/triggers**.

4.3.4. Utilização de scripts sem casca em um arquivo SPEC

A opção **-p** scriptlet em um arquivo SPEC permite que o usuário invoque um intérprete específico em vez do intérprete shell scripts padrão (**-p /bin/sh**).

O procedimento a seguir descreve como criar um roteiro, que imprime uma mensagem após a instalação do programa **pello.py**:

Procedimento

1. Abra o arquivo **pello.spec**.
2. Encontre a seguinte linha:

```
install -m 0644 %{nome}.py* %{buildroot}/usr/lib/%{nome}/
```

3. Sob a linha acima, inserir:

```
%post -p /usr/bin/python3
print("This is {} code".format("python"))
```

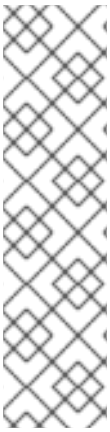
4. Construa seu pacote conforme descrito em Seção 3.3, “RPMs de construção” .

5. Instale seu pacote:

```
# yum install /home/<username>/rpmbuild/RPMS/noarch/pello-0.1.2-1.el8.noarch.rpm
```

6. Verifique a mensagem de saída após a instalação:

```
Installing       : pello-0.1.2-1.el8.noarch                    1/1
Running scriptlet: pello-0.1.2-1.el8.noarch                    1/1
This is python code
```



NOTA

Para usar um script Python 3, inclua a seguinte linha em **install -m** em um arquivo SPEC:

```
% poste -p /usr/bin/pithon3
```

Para usar um roteiro Lua, inclua a seguinte linha em **install -m** em um arquivo SPEC:

```
%post -p <lua>
```

Desta forma, você pode especificar qualquer intérprete em um arquivo da SPEC.

4.4. CONDICIONADORES RPM

Os RPM Conditionals permitem a inclusão condicional de várias seções do arquivo da SPEC.

As inclusões condicionais geralmente tratam de inclusões:

- Seções específicas de arquitetura
- Seções específicas do sistema operacional
- Problemas de compatibilidade entre várias versões de sistemas operacionais
- Existência e definição de macros

4.4.1. Sintaxe dos condicionadores RPM

Os condicionadores RPM utilizam a seguinte sintaxe:

Se *expression* for verdade, então faça alguma ação:

```
%if expression
…
%endif
```

Se *expression* for verdade, então faça alguma ação, em outro caso, faça outra ação:

```
%if expression
…
%else
…
%endif
```

4.4.2. Exemplos de condições RPM

Esta seção fornece múltiplos exemplos de condições de RPM.

4.4.2.1. As %if condicionais

Exemplo 4.3. Usando a %if condicional para lidar com a compatibilidade entre o Red Hat Enterprise Linux 8 e outros sistemas operacionais

```
%if 0%{?rhel} == 8
sed -i '/AS_FUNCTION_DESCRIBE/ s/^//' configure.in sed -i '/AS_FUNCTION_DESCRIBE/ s/^//'
acinclude.m4
%endif
```

Este condicional trata da compatibilidade entre a RHEL 8 e outros sistemas operacionais em termos de suporte da macro AS_FUNCTION_DESCRIBE. Se o pacote for construído para a RHEL, a macro **%rhel** é definida, e é expandida para a versão RHEL. Se seu valor é 8, significando que o pacote é construído para o RHEL 8, então as referências à AS_FUNCTION_DESCRIBE, que não é suportada pela RHEL 8, são excluídas dos scripts autoconfig.

Exemplo 4.4. Usando o %if condicional para lidar com a definição de macros

```
%define ruby_archive %{name}-%{ruby_version}
%if 0%{?milestone:1}%{?revision:1} != 0
%define ruby_archive %{ruby_archive}-%{?milestone}%{?!milestone:%{?revision:r%{revision}}}
%endif
```

Este condicional trata da definição de macros. Se as macros **%milestone** ou **%revision** forem definidas, a macro **%ruby_archive**, que define o nome do tarball a montante, é redefinida.

4.4.2.2. Variantes especializadas de %if condicionais

Os condicionais **%ifarch**, **%ifnarch** condicional e **%ifos** condicional são variantes especializadas dos condicionais **%if**. Estas variantes são comumente usadas, portanto, têm suas próprias macros.

4.4.2.2.1. A %ifarch condicional

O condicional **%ifarch** é usado para iniciar um bloco do arquivo SPEC que é específico da arquitetura. Ele é seguido por um ou mais especificadores de arquitetura, cada um separado por vírgulas ou espaço em branco.

Exemplo 4.5. Um exemplo de uso da %ifarch condicional

```
%ifarch i386 sparc
…
%endif
```

Todo o conteúdo do arquivo SPEC entre **%ifarch** e **%endif** é processado somente nas arquiteturas 32-bit AMD e Intel ou nos sistemas baseados na Sun SPARC.

4.4.2.2.2. A %ifnarch condicional

O condicional **%ifnarch** tem uma lógica inversa à do condicional **%ifarch**.

Exemplo 4.6. Um exemplo do uso do %ifnarch condicional

```
%ifnarch alpha
…
%endif
```

Todo o conteúdo do arquivo SPEC entre **%ifnarch** e **%endif** é processado somente se não for feito em um sistema baseado em Digital Alpha/AXP.

4.4.2.2.3. A %ifos condicional

O condicional **%ifos** é usado para controlar o processamento com base no sistema operacional do edifício. Ele pode ser seguido por um ou mais nomes de sistemas operacionais.

Exemplo 4.7. Um exemplo de uso do %ifos condicional

```
%ifos linux
…
%endif
```

Todo o conteúdo do arquivo SPEC entre **%ifos** e **%endif** é processado somente se a construção foi feita em um sistema Linux.

4.5. EMBALAGEM DE PYTHON 3 RPMS

A maioria dos projetos Python usa Setuptools para embalagem, e define as informações de embalagem no arquivo **setup.py**. Para mais informações sobre embalagens do Setuptools, consulte a documentação do Setuptools.

Você também pode embalar seu projeto Python em um pacote RPM, que oferece as seguintes vantagens em comparação com o pacote Setuptools:

- Especificação das dependências de um pacote em outras RPMs (mesmo não-Python)
- Assinatura criptográfica
  Com a assinatura criptográfica, o conteúdo dos pacotes de RPM pode ser verificado, integrado e testado com o resto do sistema operacional.

4.5.1. Descrição típica do arquivo SPEC para um pacote Python RPM

Um arquivo RPM SPEC para projetos Python tem algumas especificidades em comparação com arquivos SPEC não-Python RPM. Mais notavelmente, um nome de qualquer pacote RPM de uma biblioteca Python deve sempre incluir o prefixo **python3**.

Outras especificações são mostradas no seguinte arquivo da SPEC **example for thepython3-detox package**. Para a descrição de tais especificidades, veja as notas abaixo do exemplo.

```
%global modname detox                                                    1

Name:          python3-detox                                             2
Version:       0.12
Release:       4%{?dist}
Summary:       Distributing activities of the tox tool
License:       MIT
URL:           https://pypi.io/project/detox
Source0:       https://pypi.io/packages/source/d/%{modname}/%{modname}-%{version}.tar.gz

BuildArch:     noarch

BuildRequires: python36-devel                                            3
BuildRequires: python3-setuptools
BuildRequires: python36-rpm-macros
BuildRequires: python3-six
BuildRequires: python3-tox
BuildRequires: python3-py
BuildRequires: python3-eventlet

%?python_enable_dependency_generator                                     4

%description

Detox is the distributed version of the tox python testing tool. It makes efficient use of multiple CPUs
by running all possible activities in parallel.
Detox has the same options and configuration that tox has, so after installation you can run it in the
same way and with the same options that you use for tox.

   $ detox

%prep
%autosetup -n %{modname}-%{version}

%build
%py3_build                                                               5

%install
%py3_install

%check
%{__python3} setup.py test                                               6

%files -n python3-%{modname}
%doc CHANGELOG
%license LICENSE
```

```
%{_bindir}/detox
%{python3_sitelib}/%{modname}/
%{python3_sitelib}/%{modname}-%{version}*

%changelog
...
```

1. A macro **modname** contém o nome do projeto Python. Neste exemplo, ele é **detox**.
2. Ao embalar um projeto Python em RPM, o prefixo **python3** sempre precisa ser adicionado ao nome original do projeto. O nome original aqui é **detox** e o **name of the RPM** é **python3-detox**.
3. **BuildRequires** especifica quais pacotes são necessários para construir e testar este pacote. No BuildRequires, inclua sempre itens fornecendo ferramentas necessárias para a construção de pacotes Python: **python36-devel** e **python3-setuptools**. O pacote **python36-rpm-macros** é necessário para que os arquivos com **/usr/bin/python3** shebangs sejam automaticamente alterados para **/usr/bin/python3.6**. Para maiores informações, veja Seção 4.5.4, "Manuseio de hashbangs em scripts Python".
4. Cada pacote Python requer alguns outros pacotes para funcionar corretamente. Tais pacotes também precisam ser especificados no arquivo da SPEC. Para especificar o **dependencies**, você pode usar a macro **%python_enable_dependency_generator** para usar automaticamente as dependências definidas no arquivo **setup.py**. Se um pacote tem dependências que não são especificadas usando Setuptools, especifique-as dentro das diretivas adicionais **Requires**.
5. As macros **%py3_build** e **%py3_install** executam os comandos **setup.py build** e **setup.py install**, respectivamente, com argumentos adicionais para especificar os locais de instalação, o intérprete a utilizar e outros detalhes.
6. A seção **check** fornece uma macro que executa a versão correta do Python. A macro **%{__python3}** contém um caminho para o intérprete Python 3, por exemplo **/usr/bin/python3**. Recomendamos usar sempre a macro em vez de um caminho literal.

4.5.2. Macros comuns para pacotes Python 3 RPM

Em um arquivo SPEC, utilize sempre as macros abaixo, em vez de codificar rigidamente seus valores.

Em nomes macro, use sempre **python3** ou **python2** em vez de **python** não versionado.

Macro	Definição normal	Descrição
%{__pitão3}	/usr/bin/python3	Intérprete Python 3
%{python3_version}	3.6	A versão completa do intérprete Python 3.
%{python3_sitelib}	/usr/lib/python3.6/site-packages	Onde são instalados módulos Python puros.
%{python3_sitearch}	/usr/lib64/pithon3.6/site-packages	Onde módulos contendo extensões específicas de arquitetura são instalados.

Macro	Definição normal	Descrição
%py3_build		Executa o comando **setup.py build** com argumentos adequados para um pacote de sistema.
%py3_instalar		Executa o comando **setup.py install** com argumentos adequados para um pacote de sistema.

4.5.3. Fornece automaticamente os pacotes Python RPM

Ao embalar um projeto Python, certifique-se de que, se presente, os seguintes diretórios estejam incluídos na RPM resultante:

- **.dist-info**
- **.egg-info**
- **.egg-link**

A partir destes diretórios, o processo de construção do RPM gera automaticamente o **pythonX.Ydist** virtual, por exemplo **python3.6dist(detox)**. Estes provimentos virtuais são utilizados por pacotes que são especificados pela macro **%python_enable_dependency_generator**.

4.5.4. Manuseio de hashbangs em scripts Python

No Red Hat Enterprise Linux 8, espera-se que scripts Python executáveis usem hashbangs (shebangs) especificando explicitamente pelo menos a versão principal do Python.

O script **/usr/lib/rpm/redhat/brp-mangle-shebangs** buildroot policy (BRP) é executado automaticamente ao construir qualquer pacote RPM, e tenta corrigir hashbangs em todos os arquivos executáveis. O script BRP irá gerar erros ao encontrar um script Python com um hashbang ambíguo, como por exemplo:

```
#! /usr/bin/pithon
```

ou

```
#! /usr/bin/env python
```

Para modificar hashbangs nos scripts Python causando estes erros de construção em tempo de construção RPM, use o script **pathfix.py** do pacote **platform-python-devel**:

```
pathfix.py -pn -i %{__python3} PATH...
```

Múltiplos ***PATH*s** pode ser especificado. Se um ***PATH*** é um diretório, **pathfix.py** escaneia recursivamente qualquer script Python que corresponda ao padrão **^[a-zA-Z0-9_] \.py$**, não apenas aqueles com um hashbang ambíguo. Adicione este comando à seção **%prep** ou ao final da seção

%install.

Alternativamente, modifique os scripts Python empacotados para que eles estejam de acordo com o formato esperado. Para este fim, **pathfix.py** também pode ser usado fora do processo de construção RPM. Ao rodar **pathfix.py** fora de uma construção RPM, substitua ***__python3*** do exemplo acima com um caminho para o hashbang, tal como /**usr**/**bin**/**python3**.

Se os scripts Python empacotados exigirem a versão Python 2, substitua o número 3 por 2 nos comandos acima.

Adicionalmente, hashbangs no formulário /**usr**/**bin**/**python3** são por default substituídos por hashbangs apontando para Python do pacote **platform-python** usado para ferramentas de sistema com o Red Hat Enterprise Linux.

Para alterar o hashbangs /**usr**/**bin**/**python3** em seus pacotes personalizados para apontar para uma versão do Python instalada a partir do Application Stream, no formulário /**usr**/**bin**/**python3.6**, adicione o pacote **python36-rpm-macros** na seção **BuildRequires** do arquivo SPEC:

```
BuildRequires: python36-rpm-macros
```

NOTA

Para evitar verificação de hashbang e modificação pelo roteiro do BRP, use a seguinte diretiva RPM:

```
%undefine p_mangle_shebangs
```

4.6. PACOTES RUBYGEMS

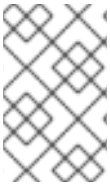
Esta seção explica o que são os pacotes RubyGems, e como re-embalá-los em RPM.

4.6.1. O que são os RubyGems

O Ruby é uma linguagem de programação dinâmica, interpretada, reflexiva, orientada a objetos, de uso geral.

Os programas escritos em Ruby são tipicamente embalados utilizando o projeto RubyGems, que fornece um formato específico de embalagem Ruby.

Os pacotes criados pela RubyGems são chamados gemas, e também podem ser reembalados em RPM.



NOTA

Esta documentação refere-se aos termos relacionados ao conceito RubyGems com o prefixo **gem**, por exemplo .gemspec é usado para o **gem specification**, e os termos relacionados ao RPM são inqualificáveis.

4.6.2. Como os RubyGems se relacionam com o RPM

Os RubyGems representam o próprio formato de embalagem da Ruby. Entretanto, os RubyGems contêm metadados similares aos necessários ao RPM, o que permite a conversão de RubyGems para RPM.

De acordo com as Diretrizes para Embalagens Ruby , é possível reembalar as embalagens RubyGems em RPM desta forma:

- Tais RPMs se ajustam ao resto da distribuição.
- Os usuários finais são capazes de satisfazer as dependências de uma gema, instalando a gema empacotada com o RPM apropriado.

Os RubyGems usam terminologia semelhante à RPM, como arquivos SPEC, nomes de pacotes, dependências e outros itens.

Para se encaixar no resto da distribuição RHEL RPM, os pacotes criados pela RubyGems devem seguir as convenções listadas abaixo:

- Os nomes das gemas devem seguir este padrão:

```
rubygem-%{gem_name}
```

- Para implementar uma linha de shebang, deve ser utilizado o seguinte cordel:

```
#!/usr/bin/ruby
```

4.6.3. Criação de pacotes RPM a partir de pacotes RubyGems

Esta seção descreve como criar pacotes RPM a partir de pacotes criados pela RubyGems.

Para criar um RPM fonte para um pacote RubyGems, são necessários dois arquivos:

- Um arquivo gema
- Um arquivo RPM SPEC

4.6.3.1. RubyGems convenções de arquivos SPEC

Um arquivo da RubyGems SPEC deve atender as seguintes convenções:

- Contém uma definição de **%{gem_name}**, que é o nome da especificação da gema.
- A fonte do pacote deve ser o URL completo do arquivo de gemas lançado; a versão do pacote deve ser a versão da gema.
- Conter o **BuildRequires:** uma diretiva definida como segue para poder puxar as macros necessárias para construir.

```
BuildRequires:rubygems-devel
```

- Não contém nenhum RubyGems **Requires** ou **Provides**, pois estes são autogerados.
- Não contém a diretriz **BuildRequires:** definida como segue, a menos que você queira especificar explicitamente a compatibilidade da versão Ruby:

```
Requer: rubi(liberação)
```

A dependência gerada automaticamente pelo RubyGems (**Requires: ruby(rubygems)**) é suficiente.

Macros
Macros úteis para pacotes criados pela RubyGems são fornecidos pelos pacotes **rubygems-devel**.

Tabela 4.3. Macros da RubyGems

Macro nome	Caminho estendido	Utilização
%{gem_dir}	/usr/share/gems	Diretório superior para a estrutura da jóia.
%{gem_instdir}	%{gem_dir}/gems/%{gem_name}-%{version}	Diretório com o conteúdo real da jóia.
%{gem_libdir}	%{gem_instdir}/lib	O diretório da biblioteca da jóia.
%{gem_cache}	%{gem_dir}/cache/%{gem_name}-%{versão}.gem	A jóia em cache.
%{gem_spec}	%{gem_dir}/specifications/%{gem_name}-%{version}.gemspec	O arquivo de especificação da jóia.
%{gem_docdir}	%{gem_dir}/doc/%{gem_name}-%{version}	A documentação RDoc da jóia.
%{gem_extdir_mri}	%{_libdir}/gems/ruby/%{gem_name}-%{versão}	O diretório para extensão de jóias.

4.6.3.2. RubyGems SPEC exemplo de arquivo

Esta seção fornece um arquivo SPEC de exemplo para a construção de gemas, juntamente com uma explicação de suas seções particulares.

Um exemplo de arquivo SPEC RubyGems

```
%prep
%setup -q -n  %{gem_name}-%{version}

# Modify the gemspec if necessary
# Also apply patches to code if necessary
%patch0 -p1

%build
# Create the gem as gem install only works on a gem file
gem build ../%{gem_name}-%{version}.gemspec

# %%gem_install compiles any C extensions and installs the gem into ./%%gem_dir
# by default, so that we can move it into the buildroot in %%install
%gem_install

%install
mkdir -p %{buildroot}%{gem_dir}
cp -a ./%{gem_dir}/* %{buildroot}%{gem_dir}/
```

```
# If there were programs installed:
mkdir -p %{buildroot}%{_bindir}
cp -a ./%{_bindir}/* %{buildroot}%{_bindir}

# If there are C extensions, copy them to the extdir.
mkdir -p %{buildroot}%{gem_extdir_mri}
cp -a .%{gem_extdir_mri}/{gem.build_complete,*.so} %{buildroot}%{gem_extdir_mri}/
```

A tabela a seguir explica as especificidades de itens específicos em um arquivo da RubyGems SPEC:

Tabela 4.4. As diretrizes específicas da SPEC da RubyGems

Diretriz SPEC	RubyGems específicos
%prep	O RPM pode desempacotar diretamente os arquivos de gemas, para que você possa executar o comando **gem unpack** para extrair a fonte da gema. A macro **%setup -n %{gem_name}-%{version}** fornece o diretório no qual a gema foi desempacotada. No mesmo nível de diretório, o arquivo **%{gem_name}-%{version}.gemspec** é criado automaticamente, o qual pode ser usado para reconstruir a gema mais tarde, para modificar o **.gemspec**, ou para aplicar correções ao código.
%build	Esta diretiva inclui comandos ou séries de comandos para construir o software em código de máquina. A macro **%gem_install** opera somente em arquivos de gemas, e a gema é recriada com a próxima construção de gema. O arquivo gem que é criado é então usado por **%gem_install** para construir e instalar o código no diretório temporário, que é **./%{gem_dir}** por padrão. A macro **%gem_install** tanto constrói como instala o código em uma única etapa. Antes de serem instalados, os fontes construídos são colocados em um diretório temporário que é criado automaticamente. A macro **%gem_install** aceita duas opções adicionais: **-n <gem_file>**, que permite anular a gema usada para instalação, e **-d <install_dir>**, que pode anular o destino da instalação da gema; não é recomendável usar esta opção. A macro **%gem_install** não deve ser utilizada para instalar no **%{buildroot}**.

Diretriz SPEC	RubyGems específicos
%instalar	A instalação é realizada dentro da hierarquia **%{buildroot}**. Você pode criar os diretórios que precisa e depois copiar o que foi instalado nos diretórios temporários para a hierarquia **%{buildroot}**. Se esta jóia cria objetos compartilhados, eles são movidos para o caminho específico da arquitetura **%{gem_extdir_mri}**.

Para mais informações sobre os arquivos SPEC da RubyGems, consulte as Diretrizes de embalagem Ruby.

4.6.3.3. Conversão de pacotes RubyGems em arquivos RPM SPEC com gem2rpm

O utilitário **gem2rpm** converte os pacotes RubyGems para arquivos RPM SPEC.

4.6.3.3.1. Instalando gem2rpm

Procedimento

- Para instalar **gem2rpm** de RubyGems.org, execute:

```
$ gem instalar gem2rpm
```

4.6.3.3.2. Exibindo todas as opções de gem2rpm

Procedimento

- Para ver todas as opções do site **gem2rpm**, execute:

```
gem2rpm --ajuda
```

4.6.3.3.3. Usando gem2rpm para cobrir pacotes RubyGems para arquivos RPM SPEC

Procedimento

- Baixe uma jóia em sua última versão e gere o arquivo RPM SPEC para esta jóia:

```
$ gem2rpm --fetch <gem_name> > <gem_name>.spec
```

O procedimento descrito cria um arquivo SPEC RPM com base nas informações fornecidas nos metadados da gema. Entretanto, a gema perde algumas informações importantes que normalmente são fornecidas nos RPMs, tais como a licença e o changelog. Portanto, o arquivo SPEC gerado precisa ser editado.

4.6.3.3.4. Edição de modelos gem2rpm

Recomenda-se editar o modelo a partir do qual o arquivo RPM SPEC é gerado em vez do próprio arquivo SPEC gerado.

O modelo é um arquivo padrão Embedded Ruby (ERB), que inclui variáveis listadas na tabela a seguir.

Tabela 4.5. Variáveis no modelo gem2rpm

Variável	Explicação
pacote	A variável **Gem::Package** para a jóia.
spec	A variável **Gem::Specification** para a gema (o mesmo que format.spec).
config	A variável **Gem2Rpm::Configuration** que pode redefinir macros padrão ou regras usadas em ajudantes de modelos de especificações.
tempo de execução_dependências	A variável **Gem2Rpm::RpmDependencyList** fornece uma lista de dependências de tempo de execução de pacotes.
dependências_desenvolvimento	A variável **Gem2Rpm::RpmDependencyList** fornece uma lista de dependências de desenvolvimento de pacotes.
testes	A variável **Gem2Rpm::TestSuite** fornece uma lista de estruturas de teste que permitem sua execução.
arquivos	A variável **Gem2Rpm::RpmFileList** fornece uma lista não filtrada de arquivos em um pacote.
main_files	A variável **Gem2Rpm::RpmFileList** fornece uma lista de arquivos adequados para o pacote principal.
doc_files	A variável **Gem2Rpm::RpmFileList** fornece uma lista de arquivos adequados para o subpacote **-doc**.
formato	A variável **Gem::Format** para a jóia. Note que esta variável está agora depreciada.

Procedimento

- Para ver todos os modelos disponíveis, execute:

```
$ gem2rpm -- modelos
```

Para editar os modelos **gem2rpm**, siga este procedimento:

Procedimento

1. Salvar o modelo padrão:

```
$ gem2rpm -T > rubygem-<gem_name>.spec.template
```

2. Edite o modelo conforme necessário.

3. Gerar o arquivo SPEC usando o modelo editado:

```
$ gem2rpm -t rubygem-<gem_name>.spec.template <gem_name>-<latest_version.gem >
<gem_name>-GEM.spec
```

Agora você pode construir um pacote RPM usando o modelo editado, conforme descrito em Seção 3.3, "RPMs de construção".

4.7. COMO LIDAR COM PACOTES RPM COM SCRIPTS PERLS

No RHEL 8, a linguagem de programação Perl não está incluída no buildroot padrão. Portanto, os pacotes RPM que incluem scripts Perl devem indicar explicitamente a dependência do Perl usando a diretiva **BuildRequires:** no arquivo SPEC do RPM.

4.7.1. Dependências comuns relacionadas ao Perl

As dependências de construção relacionadas com o Perl que ocorrem com mais frequência utilizadas em **BuildRequires:** são :

- **perl-generators**
  Gera automaticamente tempo de execução **Requires** e **Provides** para arquivos Perl instalados. Quando você instala um script Perl ou um módulo Perl, você deve incluir uma dependência de construção neste pacote.

- **perl-interpreter**
  O interpretador Perl deve ser listado como uma dependência de construção se for chamado de alguma forma, seja explicitamente através do pacote **perl** ou da macro **%__perl**, ou como parte do sistema de construção do seu pacote.

- **perl-devel**
  Fornece arquivos de cabeçalho Perl. Se construir um código específico da arquitetura que se ligue à biblioteca **libperl.so**, como um módulo XS Perl, você deve incluir **BuildRequires: perl-devel**.

4.7.2. Usando um módulo Perl específico

Se um módulo Perl específico for necessário no momento da construção, use o seguinte procedimento:

Procedimento

- Aplique a seguinte sintaxe em seu arquivo RPM SPEC:

```
BuildRequires: perl(MÓDULO)
```

NOTA

Aplique esta sintaxe também aos módulos do núcleo Perl, pois eles podem entrar e sair do pacote **perl** ao longo do tempo.

4.7.3. Limitando um pacote a uma versão Perl específica

Para limitar seu pacote a uma versão Perl específica, siga este procedimento:

Procedimento

- Use a dependência **perl(:VERSION)** com a restrição de versão desejada em seu arquivo SPEC RPM:
  Por exemplo, para limitar um pacote à versão Perl 5.22 e superior, use:

```
BuildRequires: perl(:VERSION) >= 5,22
```

ATENÇÃO

Não utilize uma comparação com a versão do pacote **perl**, pois ele inclui um número de época.

4.7.4. Assegurar que um pacote utilize o intérprete Perl correto

A Red Hat fornece múltiplos intérpretes Perl, que não são totalmente compatíveis. Portanto, qualquer pacote que forneça um módulo Perl deve usar em tempo de execução o mesmo intérprete Perl que foi usado em tempo de construção.

Para garantir isso, siga o procedimento abaixo:

Procedimento

- Incluir a versão **MODULE_COMPAT Requires** no arquivo RPM SPEC para qualquer pacote que forneça um módulo Perl:

```
Requires:  perl(:MODULE_COMPAT_%(eval `perl -V:version`; echo $version))
```

CAPÍTULO 5. NOVAS CARACTERÍSTICAS NO RHEL 8

Esta seção documenta as mudanças mais notáveis na embalagem RPM entre o Red Hat Enterprise Linux 7 e 8.

5.1. APOIO PARA DEPENDÊNCIAS FRACAS

5.1.1. Introdução à política de dependências fracas

Weak dependencies são variantes da diretiva **Requires**. Estas variantes são comparadas com o virtual **Provides:** e nomes de pacotes usando **Epoch-Version-Release** comparações de faixa.

Weak dependencies tem dois pontos fortes (**weak** e **hint**) e duas direções (**forward** e **backward**), conforme resumido na tabela a seguir.

NOTA

A direção **forward** é análoga a **Requires:**. O **backward** não tem análogo no sistema de dependência anterior.

Tabela 5.1. Possíveis combinações de forças e direções de dependências fracas

Força/Direção	Avançar	Para trás
Fraco	Recomenda:	Suplementos:
Dica	Sugere:	Aprimoramentos:

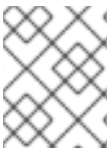
As principais vantagens da política **Weak dependencies** são:

- Permite instalações mínimas menores, mantendo o recurso de instalação padrão rico.
- Os pacotes podem especificar preferências para fornecedores específicos, mantendo a flexibilidade das ofertas virtuais.

5.1.1.1. Dependências fracas

Por padrão, **Weak dependencies** são tratados de forma semelhante ao normal **Requires:**. Os pacotes de correspondência estão incluídos no **YUM** transação. Se a adição do pacote levar a um erro, **YUM** por padrão, ignora a dependência. Portanto, os usuários podem excluir pacotes que seriam adicionados por **Weak dependencies** ou removê-los posteriormente.

Condições de uso
Você pode usar **Weak dependencies** somente se o pacote ainda funcionar sem a dependência.



NOTA

É aceitável criar pacotes com funcionalidade muito limitada sem acrescentar nenhuma de suas fracas exigências.

Casos de uso

Use **Weak dependencies** especialmente onde é possível minimizar a instalação para casos de uso razoável, como a construção de máquinas virtuais ou containers que têm um único propósito e não requerem o conjunto completo de recursos do pacote.

Os casos típicos de uso para **Weak dependencies** são:

- Documentação
  - Os telespectadores de documentação, se faltarem, são tratados com elegância
- Exemplos
- Plug-ins ou add-ons
  - Suporte para formatos de arquivo
  - Apoio a protocolos

5.1.1.2. Dicas

Hints são, por padrão, ignorados por **YUM**. Eles podem ser usados por ferramentas GUI para oferecer pacotes adicionais que não são instalados por padrão, mas podem ser úteis em combinação com os pacotes instalados.

Não utilize **Hints** para as exigências dos casos de uso principal de um pacote. Inclua tais requisitos no forte ou **Weak dependencies** em seu lugar.

Preferência de pacote
YUM usa **Weak dependencies** e **Hints** para decidir qual pacote usar se há uma escolha entre vários pacotes igualmente válidos. Os pacotes que são apontados por dependências de pacotes instalados ou a serem instalados são preferidos.

Observe que as regras normais de resolução de dependência não são influenciadas por esta característica. Por exemplo, **Weak dependencies** não pode impor uma versão mais antiga de um pacote a ser escolhido.

Se houver vários fornecedores para uma dependência, o pacote necessário pode adicionar um **Suggests:** para fornecer uma dica ao resolvedor de dependência sobre qual opção é preferida.

Enhances: só é usado quando o pacote principal e outros fornecedores concordam que adicionar a dica ao pacote necessário é, por alguma razão, a solução mais limpa.

Exemplo 5.1. Usando Dicas para preferir um pacote em vez de outro

```
Package A: Requires: mysql

Package mariadb: Provides: mysql

Package community-mysql: Provides: mysql
```

Se você quiser preferir o pacote **mariadb** ao invés do pacote **community-mysql** utilize →:

```
Sugere: mariadb para o Pacote A.
```

5.1.1.3. Dependências para frente e para trás

Forward dependencies são, de forma semelhante a **Requires**, avaliados para pacotes que estão sendo instalados. Os melhores dos pacotes correspondentes também são instalados.

Em geral, prefira **Forward dependencies**. Adicione a dependência ao pacote quando conseguir que o outro pacote seja adicionado ao sistema.

Para **Backward dependencies**, os pacotes contendo a dependência são instalados se um pacote correspondente for instalado também.

Backward dependencies são projetados principalmente para fornecedores terceiros que podem anexar seus plug-ins, add-ons ou extensões à distribuição ou outros pacotes de terceiros.

5.2. APOIO ÀS DEPENDÊNCIAS BOOLEANAS

A partir da versão 4.13, o RPM é capaz de processar expressões booleanas nas seguintes dependências:

- **Requires**
- **Recommends**
- **Suggests**
- **Supplements**
- **Enhances**
- **Conflicts**

Esta seção descreve a sintaxe das dependências booleanas, fornece uma lista de operadores booleanos e explica as dependências booleanas de aninhamento, bem como a semântica das dependências booleanas.

5.2.1. Sintaxe das dependências booleanas

Expressões booleanas são sempre fechadas com parênteses.

Eles são construídos a partir de dependências normais:

- Somente nome ou nome
- Comparação
- Descrição da versão

5.2.2. Operadores booleanos

A RPM 4.13 introduziu os seguintes operadores booleanos:

Tabela 5.2. Operadores booleanos introduzidos com RPM 4.13

Operador booleano	Descrição	Exemplo de uso

Operador booleano	Descrição	Exemplo de uso
and	Requer que todos os operandos sejam cumpridos para que o termo seja verdadeiro.	Conflitos: (pkgA e pkgB)
or	Requer um dos operandos a ser cumprido para que o termo seja verdadeiro.	Requer: (pkgA >= 3.2 ou pkgB)
if	Exige que a primeira operação seja cumprida se a segunda for. (implicação inversa)	Recomenda: (myPkg-langCZ se langsupportCZ)
if else	O mesmo que o operador **if**, além de exigir que o terceiro operando seja cumprido se o segundo não for.	Requer: myPkg-backend-mariaDB se mariaDB mais sqlite

A RPM 4.14 introduziu os seguintes operadores booleanos adicionais:

Tabela 5.3. Operadores booleanos introduzidos com RPM 4.14

Operador booleano	Descrição	Exemplo de uso
with	Requer que todos os operandos sejam cumpridos pelo mesmo pacote para que o termo seja verdadeiro.	Requer: (pkgA-foo com pkgA-bar)
without	Requer um único pacote que satisfaça o primeiro operando, mas não o segundo. (subtração do conjunto)	Requer: (pkgA-foo sem pkgA-bar)
unless	Exige que a primeira operação seja cumprida se a segunda não o for. (implicação negativa inversa)	Conflitos: (myPkg-driverA unless driverB)
unless else	O mesmo que o operador **unless**, além de exigir que o terceiro operando seja cumprido se o segundo for.	Conflitos: (myPkg-backend-SDL1 a não ser que myPkg-backend-SDL2 seja SDL2)

IMPORTANTE

O operador **if** não pode ser usado no mesmo contexto com o operador **or** e o operador **unless** não pode ser usado no mesmo contexto com **and**.

5.2.3. Aninhamento

Os próprios operandos podem ser usados como expressões booleanas, como mostrado nos exemplos abaixo.

Note que, nesse caso, os operandos também precisam ser cercados por parênteses. Você pode encadear os operadores **and** e **or** repetindo juntos o mesmo operador com apenas um conjunto de parênteses ao redor.

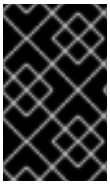
Exemplo 5.2. Exemplo de uso de operandos aplicados como expressões booleanas

```
Requer: (pkgA ou pkgB ou pkgC)

Requer: (pkgA ou (pkgB e pkgC))

Suplementos: (foo e (lang-support-cz ou lang-support-all))

Requer: (pkgA com capB) ou (pkgB sem capA)

Suplementos: ((driverA e driverA-tools) a menos que driverB)

Recomenda: myPkg-langCZ e (font1-langCZ ou font2-langCZ) se langsupportCZ
```

5.2.4. Semântica

Usar **Boolean dependencies** não muda a semântica das dependências regulares.

Se for utilizado o site **Boolean dependencies**, a verificação de uma correspondência é feita em todos os nomes e o valor booleano da correspondência é então agregado sobre os operadores booleanos.



IMPORTANTE

Para todas as dependências, com exceção de **Conflicts:**, o resultado tem que ser **True** para não impedir uma instalação. Para **Conflicts:**, o resultado tem que ser **False** para não impedir uma instalação.

ATENÇÃO

Provides não são dependências e não podem conter expressões booleanas.

5.2.4.1. Compreender a saída do se operador

O operador **if** também está devolvendo um valor booleano, que normalmente está próximo do que é a compreensão intuitiva. Entretanto, os exemplos abaixo mostram que, em alguns casos, a compreensão intuitiva do **if** pode ser enganosa.

Exemplo 5.3. Resultados enganosos do se operador

Esta afirmação é verdadeira se o pkgB não for instalado. Entretanto, se esta declaração for usada onde o resultado padrão for falso, as coisas se tornam complicadas:

```
Requer: (pkgA se pkgB)
```

Esta declaração é um conflito, a menos que o pkgB esteja instalado e o pkgA não esteja:

```
Conflitos: (pkgA se pkgB)
```

Portanto, talvez você prefira usar:

```
Conflitos: (pkgA e pkgB)
```

O mesmo é verdade se o operador **if** estiver aninhado em termos de **or**:

```
Requer: ((pkgA se pkgB) ou pkgC ou pkgC)
```

Isto também torna todo o termo verdadeiro, pois o termo **if** é verdadeiro se o pkgB não for instalado. Se o pkgA só ajuda se o pkgB estiver instalado, use **and** em seu lugar:

```
Requer: ((pkgA e pkgB) ou pkgC ou pkgP)
```

5.3. SUPORTE PARA ACIONADORES DE ARQUIVOS

File triggers são uma espécie de RPM scriptlets, que são definidos em um arquivo SPEC de um pacote.

Similar a **Triggers**, eles são declarados em um pacote, mas executados quando outro pacote que contém os arquivos correspondentes é instalado ou removido.

Um uso comum do **File triggers** é a atualização de registros ou caches. Nesse caso, o pacote contendo ou gerenciando o registro ou cache deve conter também um ou mais **File triggers**. A inclusão de **File triggers** economiza tempo em comparação com a situação em que o pacote controla a atualização em si.

5.3.1. O arquivo aciona a sintaxe

File triggers têm a seguinte sintaxe:

```
%file_trigger_tag [FILE_TRIGGER_OPTIONS] — PATHPREFIX…
body_of_script
```

Onde:

file_trigger_tag define um tipo de gatilho de arquivo. Os tipos permitidos são:

- **filetriggerin**
- **filetriggerun**
- **filetriggerpostun**
- **transfiletriggerin**

- **transfiletriggerun**
- **transfiletriggerpostun**

FILE_TRIGGER_OPTIONS têm a mesma finalidade que as opções de RPM scriptlets, exceto pela opção **-P**.

A prioridade de um gatilho é definida por um número. Quanto maior o número, mais cedo o script de disparo do arquivo é executado. Gatilhos com prioridade maior que 100000 são executados antes dos scriptlets padrão, e os outros gatilhos são executados após os scriptlets padrão. A prioridade padrão é definida para 1000000.

Cada disparo de arquivo de cada tipo deve conter um ou mais prefixos de caminho e scripts.

5.3.2. Exemplos de arquivo que aciona a sintaxe

Esta seção mostra exemplos concretos da sintaxe **File triggers**:

```
%filetriggerin — /lib, /lib64, /usr/lib, /usr/lib64
/usr/sbin/ldconfig
```

Este gatilho de arquivo executa **/usr/bin/ldconfig** diretamente após a instalação de um pacote que contém um arquivo com um caminho que começa com **/usr/lib** ou **/lib**. O gatilho de arquivo é executado apenas uma vez, mesmo que o pacote inclua vários arquivos com o caminho começando por **/usr/lib** ou **/lib**. Entretanto, todos os nomes de arquivos começando com **/usr/lib** ou **/lib** são passados para a entrada padrão do script de gatilho para que você possa filtrar dentro de seu script, como mostrado abaixo:

```
%filetriggerin — /lib, /lib64, /usr/lib, /usr/lib64
grep "foo" && /usr/sbin/ldconfig
```

Este gatilho de arquivo executa **/usr/bin/ldconfig** para cada pacote contendo arquivos começando com **/usr/lib** e contendo **foo** ao mesmo tempo. Observe que os arquivos com prefixos incluem todos os tipos de arquivos, incluindo arquivos regulares, diretórios, links simbólicos e outros.

5.3.3. Tipos de acionadores de arquivos

File triggers têm dois tipos principais:

- Acionadores de arquivo executados uma vez por pacote
- Acionadores de arquivos executados uma vez por transação

File triggers são ainda divididos com base no tempo de execução da seguinte forma:

- Antes ou depois da instalação ou do apagamento de um pacote
- Antes ou depois de uma transação

5.3.3.1. Executado uma vez por pacote Acionadores de arquivo

File triggers executados uma vez por pacote são:

- letriggerin

- letriggerun
- letriggerpostun

letriggerin
Este gatilho de arquivo é executado após a instalação de um pacote se este pacote contiver um ou mais arquivos que correspondam ao prefixo deste gatilho. Também é executado após a instalação de um pacote que contenha este gatilho de arquivo e houver um ou mais arquivos que correspondam ao prefixo deste gatilho de arquivo no banco de dados **rpmdb**.

letriggerun
Este gatilho de arquivo é executado antes da desinstalação de um pacote se este pacote contiver um ou mais arquivos que correspondam ao prefixo deste gatilho. Também é executado antes da desinstalação de um pacote que contém este gatilho de arquivo e há um ou mais arquivos que combinam com o prefixo deste gatilho de arquivo em **rpmdb**.

letriggerpostun
Este gatilho de arquivo é executado após a desinstalação de um pacote se este pacote contiver um ou mais arquivos que correspondam ao prefixo deste gatilho.

5.3.3.2. Executado uma vez por transação Acionamento do arquivo

File triggers executados uma vez por transação são:

- %transfiletriggerin
- %transfiletriggerun
- %transfiletriggerpostun

%transfiletriggerin
Este gatilho de arquivo é executado uma vez após uma transação para todos os pacotes instalados que contenham um ou mais arquivos que correspondam ao prefixo deste gatilho. Também é executado após uma transação se houver um pacote contendo este gatilho de arquivo naquela transação e houver um ou mais arquivos que correspondam ao prefixo deste gatilho em **rpmdb**.

%transfiletriggerun
Este gatilho de arquivo é executado uma vez antes de uma transação para todos os pacotes que atendam as seguintes condições:

- O pacote será desinstalado nesta transação
- O pacote contém um ou mais arquivos que correspondem ao prefixo deste gatilho

Ele também é executado antes de uma transação se houver um pacote contendo este gatilho de arquivo nessa transação e houver um ou mais arquivos que correspondam ao prefixo deste gatilho em **rpmdb**.

%transfiletriggerpostun
Este gatilho de arquivo é executado uma vez após uma transação para todos os pacotes desinstalados que contenham um ou mais arquivos que correspondam ao prefixo deste gatilho.

NOTA

A lista de arquivos de ativação não está disponível neste tipo de ativação.

Portanto, se você instalar ou desinstalar vários pacotes que contenham bibliotecas, o cache ldconfig é atualizado no final de toda a transação. Isto melhora significativamente o desempenho comparado ao RHEL 7, onde o cache foi atualizado para cada pacote separadamente. Também os scriptlets que chamavam ldconfig em %post e %postun em arquivo SPEC de cada pacote não são mais necessários.

5.3.4. Exemplo de uso de gatilhos de arquivo em glibc

Esta seção mostra um exemplo real de uso do **File triggers** dentro do pacote **glibc**.

Na RHEL 8, **File triggers** são implementadas em **glibc** para chamar o comando **ldconfig** ao final de uma transação de instalação ou desinstalação.

Isto é assegurado pela inclusão dos seguintes roteiros no arquivo **glibc's** SPEC:

```
%transfiletriggerin common -P 2000000 – /lib /usr/lib /lib64 /usr/lib64
/sbin/ldconfig
%end
%transfiletriggerpostun common -P 2000000 – /lib /usr/lib /lib64 /usr/lib64
/sbin/ldconfig
%end
```

Portanto, se você instalar ou desinstalar vários pacotes, o cache **ldconfig** é atualizado para todas as bibliotecas instaladas após a conclusão de toda a transação. Conseqüentemente, não é mais necessário incluir os scriptlets chamando **ldconfig** nos arquivos RPM SPEC de pacotes individuais. Isto melhora o desempenho comparado ao RHEL 7, onde o cache foi atualizado para cada pacote separadamente.

5.4. ANALISADOR MAIS RIGOROSO DA SPEC

O analisador da SPEC tem agora algumas mudanças incorporadas. Assim, ele pode identificar novas questões que antes eram ignoradas.

5.5. SUPORTE PARA ARQUIVOS ACIMA DE 4 GB

No Red Hat Enterprise Linux 8, **RPM** pode usar variáveis e tags de 64 bits, o que permite operar em arquivos e pacotes maiores que 4 GB.

5.5.1. Etiquetas de 64 bits RPM

Existem várias etiquetas RPM tanto nas versões de 64 bits quanto nas versões anteriores de 32 bits. Note que as versões de 64 bits têm a string **LONG** na frente de seu nome.

Tabela 5.4. Tags RPM disponíveis tanto nas versões 32-bit como 64-bit

Nome da etiqueta da variante de 32 bits	62-bit variant tag name	Descrição da etiqueta
RPMTAG_SIGSIZE	RPMTAG_LONGSIGSIZE	Tamanho do cabeçalho e da carga útil comprimida.
RPMTAG_ARCHIVESIZE	RPMTAG_LONGARCHIVESIZE	Tamanho de carga útil sem compressão.

Nome da etiqueta da variante de 32 bits	62-bit variant tag name	Descrição da etiqueta
RPMTAG_FILESIZES	RPMTAG_LONGFILESIZES	Conjunto de tamanhos de arquivos.
RPMTAG_SIZE	RPMTAG_LONGSIZE	Soma de todos os tamanhos de arquivos.

5.5.1.1. Usando tags de 64 bits na linha de comando

As extensões **LONG** estão sempre habilitadas na linha de comando. Se você usou anteriormente scripts contendo o comando **rpm -q --qf**, você pode adicionar **long** ao nome de tais tags:

```
rpm -qp --qf="[%{filenames} %{longfilesizes}n]}"
```

5.6. OUTRAS CARACTERÍSTICAS

Outras novas características relacionadas à embalagem RPM no Red Hat Enterprise Linux 8 são:

- Saída de verificação de assinatura simplificada em modo não-verbose
- Apoio para a verificação da carga útil forçada
- Suporte para o modo de verificação da assinatura
- Adições e depreciações em macros

CAPÍTULO 6. RECURSOS ADICIONAIS SOBRE EMBALAGENS RPM

Esta seção fornece referências a vários tópicos relacionados a RPMs, embalagem RPM e edifício RPM. Alguns deles são avançados e ampliam o material introdutório incluído nesta documentação.

Red Hat Software Collections Overview - A oferta da Red Hat Software Collections fornece ferramentas de desenvolvimento continuamente atualizadas nas últimas versões estáveis.

Red Hat Software Collections - O Guia de Embalagem fornece uma explicação das Coleções de Software e detalhes de como construí-las e embalá-las. Desenvolvedores e administradores de sistemas com entendimento básico de empacotamento de software com RPM podem usar este Guia para começar a trabalhar com o Software Collections.

Mock - Mock fornece uma solução de construção de pacotes apoiada pela comunidade para várias arquiteturas e diferentes versões Fedora ou RHEL do que tem o anfitrião de construção.

Documentação RPM - A documentação oficial da RPM.

Diretrizes de embalagem Fedora - As diretrizes oficiais de embalagem para Fedora, úteis para todas as distribuições baseadas em RPM.