Red Hat Enterprise Linux 8

Empaquetado y distribución de software

Una guía para empaquetar y distribuir software en Red Hat Enterprise Linux 8

Last Updated: 2021-03-08

Red Hat Enterprise Linux 8 Empaquetado y distribución de software

Una guía para empaquetar y distribuir software en 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.

Legal Notice

Resumen

Este documento describe cómo empaquetar software en un RPM. También muestra cómo preparar el código fuente para el empaquetado, y explica algunos escenarios de empaquetado avanzados, como el empaquetado de proyectos Python o RubyGems en RPM.

Table of Contents

HACER QUE EL CÓDIGO ABIERTO SEA MÁS INCLUSIVO

Red Hat se compromete a sustituir el lenguaje problemático en nuestro código, documentación y propiedades web. Estamos empezando con estos cuatro términos: maestro, esclavo, lista negra y lista blanca. Debido a la enormidad de este esfuerzo, estos cambios se implementarán gradualmente a lo largo de varias versiones próximas. Para más detalles, consulte el mensaje de nuestro CTO Chris Wright .

PROPORCIONAR COMENTARIOS SOBRE LA DOCUMENTACIÓN DE RED HAT

Agradecemos su opinión sobre nuestra documentación. Por favor, díganos cómo podemos mejorarla. Para ello:

- Para comentarios sencillos sobre pasajes concretos:
  1. Asegúrese de que está viendo la documentación en el formato *Multi-page HTML*. Además, asegúrese de ver el botón **Feedback** en la esquina superior derecha del documento.
  2. Utilice el cursor del ratón para resaltar la parte del texto que desea comentar.
  3. Haga clic en la ventana emergente **Add Feedback** que aparece debajo del texto resaltado.
  4. Siga las instrucciones mostradas.
- Para enviar comentarios más complejos, cree un ticket de Bugzilla:
  1. Vaya al sitio web de Bugzilla.
  2. Como componente, utilice **Documentation**.
  3. Rellene el campo **Description** con su sugerencia de mejora. Incluya un enlace a la(s) parte(s) pertinente(s) de la documentación.
  4. Haga clic en **Submit Bug**.

CAPÍTULO 1. INTRODUCCIÓN AL EMPAQUETADO RPM

La siguiente sección presenta el concepto de envase RPM y sus principales ventajas.

1.1. INTRODUCCIÓN AL EMBALAJE RPM

El gestor de paquetes RPM (RPM) es un sistema de gestión de paquetes que se ejecuta en Red Hat Enterprise Linux, CentOS y Fedora. Puede utilizar RPM para distribuir, gestionar y actualizar el software que cree para cualquiera de los sistemas operativos mencionados anteriormente.

1.1.1. Ventajas de las RPM

El sistema de gestión de paquetes RPM aporta varias ventajas respecto a la distribución de software en archivos convencionales.

RPM le permite:

- Instale, reinstale, elimine, actualice y verifique los paquetes con herramientas estándar de gestión de paquetes, como Yum o PackageKit.
- Utilice una base de datos de paquetes instalados para consultar y verificar los paquetes.
- Utilice los metadatos para describir los paquetes, sus instrucciones de instalación y otros parámetros del paquete.
- Empaquetar fuentes de software, parches e instrucciones completas de construcción en paquetes fuente y binarios.
- Añadir paquetes a los repositorios Yum.
- Firme digitalmente sus paquetes utilizando las claves de firma de GNU Privacy Guard (GPG).

CAPÍTULO 2. PREPARACIÓN DEL SOFTWARE PARA SU EMPAQUETADO EN RPM

Esta sección explica cómo preparar el software para el empaquetado RPM. Para ello, no es necesario saber codificar. Sin embargo, es necesario entender los conceptos básicos, como qué es el código fuente y cómo se hacen los programas .

2.1. QUÉ ES EL CÓDIGO FUENTE

Esta parte explica qué es el código fuente y muestra ejemplos de códigos fuente de un programa escrito en tres lenguajes de programación diferentes.

El código fuente son instrucciones legibles para el ordenador, que describen cómo realizar un cálculo. El código fuente se expresa mediante un lenguaje de programación.

2.1.1. Ejemplos de código fuente

Este documento presenta tres versiones del programa **Hello World** escritas en tres lenguajes de programación diferentes:

- Sección 2.1.1.1, “Hola Mundo escrito en bash”
- Sección 2.1.1.2, “Hola Mundo escrito en Python”
- Sección 2.1.1.3, “Hola Mundo escrito en C”

Cada versión está empaquetada de forma diferente.

Estas versiones del programa **Hello World** cubren los tres principales casos de uso de un empaquetador RPM.

2.1.1.1. Hola Mundo escrito en bash

El proyecto *bello* implementa **Hello World** en bash. La implementación sólo contiene el script de shell **bello**. El propósito del programa es dar salida a **Hello World** en la línea de comandos.

El archivo **bello** tiene la siguiente sintaxis:

```
#!/bin/bash

printf "Hello World\n"
```

2.1.1.2. Hola Mundo escrito en Python

El proyecto *pello* implementa **Hello World** en Python. La implementación sólo contiene el programa **pello.py**. El propósito del programa es dar salida a **Hello World** en la línea de comandos.

El archivo **pello.py** tiene la siguiente sintaxis:

```
#!/usr/bin/python3

print("Hello World")
```

2.1.1.3. Hola Mundo escrito en C

El proyecto *cello* implementa **Hello World** en C. La implementación sólo contiene los archivos **cello.c** y **Makefile**, por lo que el archivo **tar.gz** resultante tendrá dos archivos además del archivo **LICENSE**.

El propósito del programa es dar salida a **Hello World** en la línea de comandos.

El archivo **cello.c** tiene la siguiente sintaxis:

```
#include <stdio.h>

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

2.2. CÓMO SE HACEN LOS PROGRAMAS

Los métodos de conversión de código fuente legible por humanos a código máquina (instrucciones que el ordenador sigue para ejecutar el programa) incluyen los siguientes:

- El programa está compilado de forma nativa.
- El programa se interpreta por medio de la interpretación en bruto.
- El programa se interpreta mediante compilación de bytes.

2.2.1. Código compilado de forma nativa

El software compilado de forma nativa es un software escrito en un lenguaje de programación que se compila a código máquina con un archivo binario ejecutable resultante. Este tipo de software puede ejecutarse de forma autónoma.

Los paquetes RPM construidos de esta manera son específicos para cada arquitectura.

Si compila dicho software en un ordenador que utiliza un procesador AMD o Intel de 64 bits (x86_64), no se ejecuta en un procesador AMD o Intel de 32 bits (x86). El paquete resultante tiene la arquitectura especificada en su nombre.

2.2.2. Código interpretado

Algunos lenguajes de programación, como bash o Python, no compilan a código máquina. En su lugar, el código fuente de sus programas es ejecutado paso a paso, sin transformaciones previas, por un intérprete de lenguaje o una máquina virtual de lenguaje.

El software escrito completamente en lenguajes de programación interpretados no es específico de la arquitectura. De ahí que el paquete RPM resultante tenga la cadena **noarch** en su nombre.

Los lenguajes interpretados son programas interpretados en bruto o programas compilados en bytes. Estos dos tipos difieren en el proceso de construcción del programa y en el procedimiento de empaquetado.

2.2.2.1. Programas interpretados en bruto

Los programas en lenguaje sin interpretar no necesitan ser compilados y son ejecutados directamente por el intérprete.

2.2.2.2. Programas compilados en bytes

Los lenguajes compilados en bytes necesitan ser compilados en código de bytes, que luego es ejecutado por la máquina virtual del lenguaje.

NOTA

Algunos lenguajes ofrecen una opción: pueden ser interpretados en bruto o compilados en bytes.

2.3. CREACIÓN DE SOFTWARE A PARTIR DEL CÓDIGO FUENTE

Esta parte describe cómo construir software a partir del código fuente.

En el caso del software escrito en lenguajes compilados, el código fuente pasa por un proceso de construcción que produce el código máquina. Este proceso, llamado comúnmente compilación o traducción, varía según los distintos lenguajes. El software construido resultante puede ejecutarse, lo que hace que el ordenador realice la tarea especificada por el programador.

En el caso del software escrito en lenguajes interpretados en bruto, el código fuente no se construye, sino que se ejecuta directamente.

En el caso del software escrito en lenguajes interpretados compilados en bytes, el código fuente se compila en código de bytes, que luego es ejecutado por la máquina virtual del lenguaje.

2.3.1. Código compilado de forma nativa

Esta sección muestra cómo construir el programa **cello.c** escrito en el lenguaje C en un ejecutable.

cello.c

```
#include <stdio.h>

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

2.3.1.1. Construcción manual

Si quiere construir el programa **cello.c** manualmente, utilice este procedimiento:

Procedimiento

1. Invoca el compilador C de la Colección de Compiladores GNU para compilar el código fuente en binario:

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

2. Ejecuta el binario de salida resultante **cello**:

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

2.3.1.2. Edificio automatizado

El software a gran escala suele utilizar la construcción automatizada que se realiza creando el archivo **Makefile** y ejecutando después la utilidad GNU **make**.

Si desea utilizar la construcción automatizada para construir el programa **cello.c**, utilice este procedimiento:

Procedimiento

1. Para configurar la construcción automática, cree el archivo **Makefile** con el siguiente contenido en el mismo directorio que **cello.c**.
   Makefile

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

   Tenga en cuenta que las líneas bajo **cello:** y **clean:** deben comenzar con un espacio de tabulación.

2. Para construir el software, ejecute el comando **make**:

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

3. Como ya hay una compilación disponible, ejecute el comando **make clean**, y después ejecute de nuevo el comando **make**:

   ```
   $ make clean
   rm cello

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

   NOTA

   Intentar construir el programa después de otra construcción no tiene ningún efecto.

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

4. Ejecuta el programa:

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

Ahora ha compilado un programa tanto manualmente como utilizando una herramienta de compilación.

2.3.2. Interpretación del código

Esta sección muestra cómo compilar en bytes un programa escrito en Python y cómo interpretar en crudo un programa escrito en bash.

NOTA

En los dos ejemplos siguientes, la línea **#!** al principio del archivo se conoce como **shebang**, y no forma parte del código fuente del lenguaje de programación.

El **shebang** permite utilizar un archivo de texto como ejecutable: el cargador de programas del sistema analiza la línea que contiene el **shebang** para obtener una ruta al ejecutable binario, que se utiliza entonces como intérprete del lenguaje de programación. La funcionalidad requiere que el archivo de texto esté marcado como ejecutable.

2.3.2.1. Código de compilación de bytes

Esta sección muestra cómo compilar el programa **pello.py** escrito en Python en código de bytes, que luego es ejecutado por la máquina virtual del lenguaje Python.

El código fuente de Python también puede ser interpretado en bruto, pero la versión compilada en bytes es más rápida. Por ello, los empaquetadores de RPM prefieren empaquetar la versión compilada en bytes para su distribución a los usuarios finales.

pello.py

```
#!/usr/bin/python3

print("Hello World")
```

El procedimiento para compilar programas en bytes varía en función de los siguientes factores:

- Lenguaje de programación
- Máquina virtual del lenguaje
- Herramientas y procesos utilizados con ese lenguaje

NOTA

Python suele compilarse en bytes, pero no de la manera descrita aquí. El siguiente procedimiento no pretende ajustarse a los estándares de la comunidad, sino ser sencillo. Para conocer las directrices del mundo real de Python, consulte Empaquetado y distribución de software.

Utilice este procedimiento para compilar **pello.py** en código de bytes:

Procedimiento

1. Compilación de bytes del archivo **pello.py**:

```
$ python -m compileall pello.py
```

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

2. Ejecuta el código de bytes en **pello.pyc**:

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

2.3.2.2. Código de interpretación en bruto

Esta sección muestra cómo interpretar en bruto el programa **bello** escrito en el lenguaje incorporado del shell bash.

bello

```
#!/bin/bash

printf "Hello World\n"
```

Los programas escritos en lenguajes de scripting de shell, como *bash*, se interpretan en bruto.

Procedimiento

- Haz que el archivo con el código fuente sea ejecutable y ejecútalo:

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

2.4. SOFTWARE DE PARCHEO

En esta sección se explica cómo parchear el software.

En el empaquetado RPM, en lugar de modificar el código fuente original, lo conservamos y utilizamos parches sobre él.

Un parche es un código fuente que actualiza otro código fuente. Se formatea como un *diff*, porque representa lo que es diferente entre dos versiones del texto. Un *diff* se crea con la utilidad **diff**, que luego se aplica al código fuente con la utilidad de parche.

NOTA

Los desarrolladores de software suelen utilizar sistemas de control de versiones como git para gestionar su base de código. Estas herramientas proporcionan sus propios métodos para crear diffs o parchear el software.

El siguiente ejemplo muestra cómo crear un parche a partir del código fuente original utilizando **diff**, y cómo aplicar el parche utilizando **patch**. El parcheado se utiliza en una sección posterior al crear un RPM; véase Sección 3.2, “Trabajar con archivos SPEC” .

Este procedimiento muestra cómo crear un parche a partir del código fuente original de **cello.c**.

Procedimiento

1. Conservar el código fuente original:

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

   La opción **-p** se utiliza para conservar el modo, la propiedad y las marcas de tiempo.

2. Modifique **cello.c** según sea necesario:

```
#include <stdio.h>

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

3. Genere un parche utilizando la utilidad **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
```

   Las líneas que empiezan por **-** se eliminan del código fuente original y se sustituyen por las líneas que empiezan por .

   Se recomienda utilizar las opciones **Naur** con el comando **diff** porque se ajusta a la mayoría de los casos de uso habituales. Sin embargo, en este caso particular, sólo es necesaria la opción **-u**. Las opciones particulares garantizan lo siguiente:

   - **-N** (o **--new-file**) - Maneja los archivos ausentes como si fueran archivos vacíos.
   - **-a** (o **--text**) - Trata todos los archivos como texto. Como resultado, los archivos que **diff** clasifica como binarios no son ignorados.
   - **-u** (o **-U NUM** o **--unified[=NUM]**) - Devuelve el resultado en forma de salida NUM (por defecto 3) líneas de contexto unificado. Este es un formato fácilmente legible que permite una coincidencia difusa al aplicar el parche a un árbol de fuentes modificado.
   - **-r** (o **--recursive**) - Compara de forma recursiva los subdirectorios encontrados.
     Para más información sobre los argumentos habituales de la utilidad **diff**, consulte la página del manual **diff**.

4. Guarda el parche en un archivo:

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

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

```
$ cp cello.c.orig cello.c
```

El original **cello.c** debe conservarse, porque cuando se construye un RPM, se utiliza el archivo original, no el modificado. Para más información, consulte Sección 3.2, "Trabajar con archivos SPEC".

El siguiente procedimiento muestra cómo parchear **cello.c** utilizando **cello-output-first-patch.patch**, construir el programa parcheado y ejecutarlo.

1. Redirige el archivo de parches al comando **patch**:

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

2. Compruebe que el contenido de **cello.c** refleja ahora el parche:

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

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

3. Construya y ejecute el parche **cello.c**:

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c

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

2.5. INSTALACIÓN DE ARTEFACTOS ARBITRARIOS

Los sistemas tipo Unix utilizan el estándar de jerarquía del sistema de archivos (FHS) para especificar un directorio adecuado para un archivo concreto.

Los archivos instalados a partir de los paquetes RPM se colocan según FHS. Por ejemplo, un archivo ejecutable debe ir a un directorio que esté en la variable del sistema **$PATH**.

En el contexto de esta documentación, un *Arbitrary Artifact* es cualquier cosa instalada desde un RPM al sistema. Para el RPM y para el sistema puede ser un script, un binario compilado desde el código fuente del paquete, un binario precompilado o cualquier otro archivo.

Esta sección describe dos formas habituales de colocar *Arbitrary Artifacts* en el sistema:

- Sección 2.5.1, "Utilizando el comando de instalación"
- Sección 2.5.2, "Utilizando el comando make install"

2.5.1. Utilizando el comando de instalación

Los empaquetadores suelen utilizar el comando **install** en los casos en que las herramientas de automatización de la construcción como GNU make no son óptimas; por ejemplo, si el programa empaquetado no necesita una sobrecarga adicional.

El comando **install** es proporcionado al sistema por coreutils, que coloca el artefacto en el directorio especificado en el sistema de archivos con un conjunto especificado de permisos.

El siguiente procedimiento utiliza el archivo **bello** que se creó previamente como artefacto arbitrario como objeto de este método de instalación.

Procedimiento

1. Ejecute el comando **install** para colocar el archivo **bello** en el directorio **/usr/bin** con los permisos habituales para los scripts ejecutables:

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

   Como resultado, **bello** se encuentra ahora en el directorio que aparece en la variable **$PATH**.

2. Ejecuta **bello** desde cualquier directorio sin especificar su ruta completa:

```
$ cd ~

$ bello
Hello World
```

2.5.2. Utilizando el comando make install

El uso del comando **make install** es una forma automatizada de instalar en el sistema el software construido. En este caso, es necesario especificar cómo instalar los artefactos arbitrarios en el sistema en el **Makefile** que suele ser escrito por el desarrollador.

Este procedimiento muestra cómo instalar un artefacto de construcción en una ubicación elegida en el sistema.

Procedimiento

1. Añada la sección **install** a la página web **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
```

   Tenga en cuenta que las líneas bajo **cello:**, **clean:**, y **install:** deben comenzar con un espacio de tabulación.

NOTA

La variable $(DESTDIR) es un built-in de GNU make y se utiliza comúnmente para especificar la instalación en un directorio diferente al directorio raíz.

Ahora puede utilizar **Makefile** no sólo para crear software, sino también para instalarlo en el sistema de destino.

2. Construya e instale el programa **cello.c**:

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

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

Como resultado, **cello** se encuentra ahora en el directorio que aparece en la variable **$PATH**.

3. Ejecuta **cello** desde cualquier directorio sin especificar su ruta completa:

```
$ cd ~

$ cello
Hello World
```

2.6. PREPARACIÓN DEL CÓDIGO FUENTE PARA SU EMPAQUETADO

Los desarrolladores suelen distribuir el software como archivos comprimidos de código fuente, que luego se utilizan para crear paquetes. Los empaquetadores RPM trabajan con un archivo de código fuente listo.

El software debe distribuirse con una licencia de software.

Este procedimiento utiliza el texto de la licencia GPLv3 como contenido de ejemplo del archivo **LICENSE**.

Procedimiento

- Cree un archivo **LICENSE** y asegúrese de que incluye el siguiente contenido:

```
$ 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 adicionales

- El código creado en esta sección se puede encontrar aquí.

2.7. PONER EL CÓDIGO FUENTE EN EL TARBALL

Esta sección describe cómo poner cada uno de los tres programas de **Hello World** introducidos en Sección 2.1.1, "Ejemplos de código fuente" en un tarball comprimido con gzip, que es una forma común de liberar el software para ser empaquetado posteriormente para su distribución.

2.7.1. Poner el proyecto bello en el tarball

El proyecto *bello* implementa **Hello World** en bash. La implementación sólo contiene el script de shell **bello**, por lo que el archivo resultante **tar.gz** sólo tendrá un archivo aparte del archivo **LICENSE**.

Este procedimiento muestra cómo preparar el proyecto *bello* para su distribución.

Requisitos previos

Teniendo en cuenta que esta es la versión **0.1** del programa.

Procedimiento

1. Ponga todos los archivos necesarios en un solo directorio:

```
$ mkdir /tmp/bello-0.1

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

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

2. Cree el archivo para su distribución y muévalo al directorio **~/rpmbuild/SOURCES/**, que es el directorio por defecto donde el comando **rpmbuild** almacena los archivos para la construcción de paquetes:

```
$ 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 más información sobre el código fuente de ejemplo escrito en bash, consulte Sección 2.1.1.1, "Hola Mundo escrito en bash".

2.7.2. Poner el proyecto pello en el tarball

El proyecto *pello* implementa **Hello World** en Python. La implementación sólo contiene el programa **pello.py**, por lo que el archivo resultante **tar.gz** sólo tendrá un archivo aparte del archivo **LICENSE**.

Este procedimiento muestra cómo preparar el proyecto *pello* para su distribución.

Requisitos previos

Teniendo en cuenta que esta es la versión **0.1.1** del programa.

Procedimiento

1. Ponga todos los archivos necesarios en un solo directorio:

```
$ mkdir /tmp/pello-0.1.2

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

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

2. Cree el archivo para su distribución y muévalo al directorio **~/rpmbuild/SOURCES/**, que es el directorio por defecto donde el comando **rpmbuild** almacena los archivos para la construcción de paquetes:

```
$ 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 más información sobre el código fuente de ejemplo escrito en Python, consulte Sección 2.1.1.2, "Hola Mundo escrito en Python".

2.7.3. Poner el proyecto cello en el tarball

El proyecto *cello* implementa **Hello World** en C. La implementación sólo contiene los archivos **cello.c** y **Makefile**, por lo que el archivo **tar.gz** resultante tendrá dos archivos además del archivo **LICENSE**.

NOTA

El archivo **patch** no se distribuye en el archivo con el programa. El empaquetador de RPM aplica el parche cuando se construye el RPM. El parche se colocará en el directorio **~/rpmbuild/SOURCES/** junto al archivo **.tar.gz**.

Este procedimiento muestra cómo preparar el proyecto *cello* para su distribución.

Requisitos previos

Teniendo en cuenta que esta es la versión **1.0** del programa.

Procedimiento

1. Ponga todos los archivos necesarios en un solo directorio:

```
$ 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. Cree el archivo para su distribución y muévalo al directorio **~/rpmbuild/SOURCES/**, que es el directorio por defecto donde el comando **rpmbuild** almacena los archivos para la construcción de paquetes:

```
$ 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. Añade el parche:

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

Para más información sobre el código fuente de ejemplo escrito en C, consulte Sección 2.1.1.3, “Hola Mundo escrito en C”.

CAPÍTULO 3. SOFTWARE DE ENVASADO

3.1. PAQUETES RPM

Esta sección cubre los fundamentos del formato de embalaje RPM.

3.1.1. Qué es un RPM

Un paquete RPM es un archivo que contiene otros archivos y sus metadatos (información sobre los archivos que necesita el sistema).

En concreto, un paquete RPM consiste en el archivo **cpio**.

El archivo **cpio** contiene:

- Archivos
- Cabecera del RPM (metadatos del paquete)
  El gestor de paquetes **rpm** utiliza estos metadatos para determinar las dependencias, dónde instalar los archivos y otra información.

Tipos de paquetes RPM
Hay dos tipos de paquetes RPM. Ambos tipos comparten el formato de archivo y las herramientas, pero tienen contenidos diferentes y sirven para fines distintos:

- Fuente RPM (SRPM)
  Un SRPM contiene el código fuente y un archivo SPEC, que describe cómo construir el código fuente en un RPM binario. Opcionalmente, también se incluyen los parches del código fuente.
- RPM binario
  Un RPM binario contiene los binarios construidos a partir de las fuentes y los parches.

3.1.2. Listado de utilidades de la herramienta de empaquetado RPM

Los siguientes procedimientos muestran cómo listar las utilidades proporcionadas por el paquete **rpmdevtools**.

Requisitos previos

Para poder utilizar las herramientas de empaquetado de RPM, es necesario instalar el paquete **rpmdevtools**, que proporciona varias utilidades para empaquetar RPMs.

```
# yum install rpmdevtools
```

Procedimiento

- Lista de utilidades de la herramienta de empaquetado RPM:

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

Información adicional

- Para más información sobre las utilidades anteriores, consulte sus páginas de manual o los diálogos de ayuda.

3.1.3. Configuración del espacio de trabajo de empaquetado RPM

Esta sección describe cómo configurar una distribución de directorios que es el espacio de trabajo de empaquetado de RPM utilizando la utilidad **rpmdev-setuptree**.

Requisitos previos

El paquete **rpmdevtools** debe estar instalado en su sistema:

```
# yum install rpmdevtools
```

Procedimiento

- Ejecute la utilidad **rpmdev-setuptree**:

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

Los directorios creados sirven para estos fines:

Directorio	Propósito
CONSTRUIR	Cuando se construyen los paquetes, se crean aquí varios directorios **%buildroot**. Esto es útil para investigar una construcción fallida si la salida de los registros no proporciona suficiente información.
RPMS	Los RPM binarios se crean aquí, en subdirectorios para diferentes arquitecturas, por ejemplo en los subdirectorios **x86_64** y **noarch**.
FUENTES	Aquí, el empaquetador coloca los archivos de código fuente comprimidos y los parches. El comando **rpmbuild** los busca aquí.
SPECS	El empaquetador pone aquí los archivos SPEC.
SRPMS	Cuando se utiliza **rpmbuild** para construir un SRPM en lugar de un RPM binario, el SRPM resultante se crea aquí.

3.1.4. Qué es un archivo SPEC

Puede entender un archivo SPEC como una receta que la utilidad **rpmbuild** utiliza para construir un

RPM. Un archivo SPEC proporciona la información necesaria al sistema de compilación definiendo instrucciones en una serie de secciones. Las secciones se definen en las partes *Preamble* y *Body*. La parte *Preamble* contiene una serie de metadatos que se utilizan en la parte *Body*. La parte *Body* representa la parte principal de las instrucciones.

3.1.4.1. Artículos del preámbulo

La siguiente tabla presenta algunas de las directivas que se utilizan frecuentemente en la sección *Preamble* del archivo RPM SPEC.

Tabla 3.1. Los elementos utilizados en la sección *Preamble* del archivo RPM SPEC

Directiva SPEC	Definición
Name	El nombre base del paquete, que debe coincidir con el nombre del archivo SPEC.
Version	El número de versión del software.
Release	El número de veces que se ha publicado esta versión del software. Normalmente, se establece el valor inicial en 1%{?dist}, y se incrementa con cada nueva versión del paquete. Se restablece a 1 cuando se construye un nuevo **Version** del software.
Summary	Un breve resumen de una línea del paquete.
License	La licencia del software que se empaqueta.
URL	La URL completa para obtener más información sobre el programa. En la mayoría de los casos se trata del sitio web del proyecto de origen del software empaquetado.
Source0	Ruta o URL al archivo comprimido del código fuente upstream (sin parches, los parches se gestionan en otro lugar). Esto debería apuntar a un almacenamiento accesible y fiable del archivo, por ejemplo, la página de upstream y no el almacenamiento local del empaquetador. Si es necesario, se pueden añadir más directivas SourceX, incrementando el número cada vez, por ejemplo Source1, Source2, Source3, y así sucesivamente.

Directiva SPEC	Definición
Patch	El nombre del primer parche que se aplicará al código fuente si es necesario. La directiva puede aplicarse de dos maneras: con o sin números al final de Patch. Si no se da ningún número, se asigna uno a la entrada internamente. También es posible dar los números explícitamente utilizando Parche0, Parche1, Parche2, Parche3, etc. Estos parches se pueden aplicar uno a uno utilizando la macro %patch0, %patch1, %patch2 y así sucesivamente. Las macros se aplican dentro de la directiva %prep en la sección *Body* del archivo RPM SPEC. Alternativamente, puede utilizar la macro topatch que aplica automáticamente todos los parches en el orden en que se dan en el archivo SPEC.
BuildArch	Si el paquete no depende de la arquitectura, por ejemplo, si está escrito enteramente en un lenguaje de programación interpretado, configúrelo como **BuildArch: noarch**. Si no se establece, el paquete hereda automáticamente la Arquitectura de la máquina en la que se construye, por ejemplo **x86_64**.
BuildRequires	Una lista separada por comas o espacios en blanco de los paquetes necesarios para construir el programa escrito en un lenguaje compilado. Puede haber varias entradas de **BuildRequires**, cada una en su propia línea en el archivo SPEC.
Requires	Una lista separada por comas o espacios en blanco de los paquetes necesarios para que el software funcione una vez instalado. Puede haber varias entradas de **Requires**, cada una en su propia línea en el archivo SPEC.
ExcludeArch	Si un software no puede funcionar en una arquitectura de procesador específica, puede excluir esa arquitectura aquí.
Conflicts	**Conflicts** son inversas a **Requires**. Si hay un paquete que coincide con **Conflicts**, el paquete no puede ser instalado independientemente de si la etiqueta **Conflict** está en el paquete que ya ha sido instalado o en un paquete que va a ser instalado.
Obsoletes	Esta directiva altera la forma en que funcionan las actualizaciones dependiendo de si el comando **rpm** se utiliza directamente en la línea de comandos o la actualización es realizada por un solucionador de actualizaciones o dependencias. Cuando se utiliza en la línea de comandos, RPM elimina todos los paquetes que coinciden con los obsoletos de los paquetes que se están instalando. Cuando se utiliza un solucionador de actualizaciones o dependencias, los paquetes que coinciden con **Obsoletes:** se añaden como actualizaciones y sustituyen a los paquetes que coinciden.

Directiva SPEC	Definición
Provides	Si se añade **Provides** a un paquete, se puede hacer referencia al paquete por otras dependencias que no sean su nombre.

Las directivas **Name**, **Version**, y **Release** comprenden el nombre de archivo del paquete RPM. Los mantenedores de paquetes RPM y los administradores de sistemas suelen llamar a estas tres directivas **N-V-R** o **NVR**, porque los nombres de archivo de los paquetes RPM tienen el formato **NAME-VERSION-RELEASE**.

El siguiente ejemplo muestra cómo obtener la información de **NVR** para un paquete específico consultando el comando **rpm**.

Ejemplo 3.1. Consulta de rpm para proporcionar la información de NVR para el paquete bash

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

Aquí, **bash** es el nombre del paquete, **4.4.19** es la versión, y **7.el8** es el lanzamiento. El último marcador es **x86_64**, que indica la arquitectura. A diferencia de **NVR**, el marcador de arquitectura no está bajo el control directo del empaquetador RPM, sino que está definido por el entorno de compilación **rpmbuild**. La excepción a esto es el paquete **noarch**, independiente de la arquitectura.

3.1.4.2. Artículos del cuerpo

Los elementos utilizados en el **Body section** del archivo RPM SPEC se enumeran en la siguiente tabla.

Tabla 3.2. Elementos utilizados en la sección del cuerpo del archivo RPM SPEC

Directiva SPEC	Definición
%description	Una descripción completa del software empaquetado en el RPM. Esta descripción puede abarcar varias líneas y puede dividirse en párrafos.
%prep	Comando o serie de comandos para preparar el software a construir, por ejemplo, desempaquetando el archivo en **Source0**. Esta directiva puede contener un script de shell.
%build	Comando o serie de comandos para construir el software en código de máquina (para lenguajes compilados) o código de bytes (para algunos lenguajes interpretados).
%install	Comando o serie de comandos para copiar los artefactos de compilación deseados desde **%builddir** (donde se realiza la compilación) al directorio**%buildroot** (que contiene la estructura de directorios con los archivos a empaquetar). Esto normalmente significa copiar los archivos de **~/rpmbuild/BUILD** a **~/rpmbuild/BUILDROOT** y crear los directorios necesarios en **~/rpmbuild/BUILDROOT**. Esto sólo se ejecuta cuando se crea un paquete, no cuando el usuario final instala el paquete. Véase Sección 3.2, "Trabajar con archivos SPEC" para más detalles.

Directiva SPEC	Definición
%check	Comando o serie de comandos para probar el software. Esto incluye normalmente cosas como las pruebas unitarias.
%files	La lista de archivos que se instalarán en el sistema del usuario final.
%changelog	Un registro de los cambios que se han producido en el paquete entre diferentes construcciones de **Version** o **Release**.

3.1.4.3. Artículos avanzados

El archivo SPEC también puede contener elementos avanzados, como Scriptlets o Triggers. Tienen efecto en diferentes puntos durante el proceso de instalación en el sistema del usuario final, no en el proceso de construcción.

3.1.5. BuildRoots

En el contexto del empaquetado RPM, **buildroot** es un entorno chroot. Esto significa que los artefactos de construcción se colocan aquí usando la misma jerarquía del sistema de archivos que la futura jerarquía en el sistema del usuario final, con **buildroot** actuando como el directorio raíz. La colocación de los artefactos de construcción debe cumplir con el estándar de la jerarquía del sistema de archivos del sistema del usuario final.

Los archivos de **buildroot** se colocan posteriormente en un archivo **cpio**, que se convierte en la parte principal del RPM. Cuando el RPM se instala en el sistema del usuario final, estos archivos se extraen en el directorio **root**, conservando la jerarquía correcta.

NOTA

A partir de Red Hat Enterprise Linux 6, el programa **rpmbuild** tiene sus propios valores por defecto. Anular estos valores predeterminados conduce a varios problemas; por lo tanto, Red Hat no recomienda definir su propio valor de esta macro. Puede utilizar la macro **%{buildroot}** con los valores por defecto del directorio **rpmbuild**.

3.1.6. Macros RPM

Una macro de RPM es una sustitución de texto directa que puede ser asignada condicionalmente basada en la evaluación opcional de una sentencia cuando se utiliza cierta funcionalidad incorporada. Por lo tanto, RPM puede realizar sustituciones de texto por usted.

Un ejemplo de uso es hacer referencia al software empaquetado *Version* varias veces en un archivo SPEC. Defina *Version* sólo una vez en la macro **%{version}** y utilice esta macro en todo el archivo SPEC. Cada ocurrencia será automáticamente sustituida por *Version* que usted definió previamente.

NOTA

Si ve una macro desconocida, puede evaluarla con el siguiente comando:

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

Evaluación de las macros %{_bindir} y %{_libexecdir}

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

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

Una de las macros más utilizadas es la macro **%{?dist}**, que indica qué distribución se utiliza para la compilación (etiqueta de distribución).

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

3.2. TRABAJAR CON ARCHIVOS SPEC

Esta sección describe cómo crear y modificar un archivo SPEC.

Requisitos previos

Esta sección utiliza las tres implementaciones de ejemplo del programa **Hello World!** que se describieron en Sección 2.1.1, “Ejemplos de código fuente” .

Cada uno de los programas se describe con detalle en el cuadro siguiente.

Nombre del software	Explicación del ejemplo
bello	Un programa escrito en un lenguaje de programación interpretado en bruto. Se demuestra cuando el código fuente no necesita ser construido, sino que sólo necesita ser instalado. Si se necesita empaquetar un binario precompilado, también se puede utilizar este método, ya que el binario también sería sólo un archivo.
pello	Un programa escrito en un lenguaje de programación interpretado compilado en bytes. Demuestra la compilación de bytes del código fuente y la instalación del bytecode - los archivos preoptimizados resultantes.
cello	Un programa escrito en un lenguaje de programación compilado de forma nativa. Demuestra un proceso común de compilación del código fuente en código máquina y la instalación de los ejecutables resultantes.

Las implementaciones de **Hello World!** son:

- bello-0.1.tar.gz

- pello-0.1.2.tar.gz
- cello-1.0.tar.gz
  - cello-output-first-patch.patch

Como requisito previo, estas implementaciones deben colocarse en el directorio **~/rpmbuild/SOURCES**.

3.2.1. Formas de crear un nuevo archivo SPEC

Para empaquetar un nuevo software, es necesario crear un nuevo archivo SPEC.

Hay dos maneras de conseguirlo:

- Escribir el nuevo archivo SPEC manualmente desde cero
- Utilice la utilidad **rpmdev-newspec**
  Esta utilidad crea un archivo SPEC sin rellenar, y usted rellena las directivas y campos necesarios.

NOTA

Algunos editores de texto centrados en la programación rellenan previamente un nuevo archivo **.spec** con su propia plantilla SPEC. La utilidad **rpmdev-newspec** proporciona un método independiente del editor.

3.2.2. Creación de un nuevo archivo SPEC con rpmdev-newspec

El siguiente procedimiento muestra cómo crear un archivo SPEC para cada uno de los tres programas mencionados de **Hello World!** utilizando la utilidad **rpmdev-newspec**.

Procedimiento

1. Cambie al directorio **~/rpmbuild/SPECS** y utilice la utilidad **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.
```

   El directorio **~/rpmbuild/SPECS/** contiene ahora tres archivos SPEC llamados **bello.spec**, **cello.spec** y **pello.spec**.

fd. Examina los archivos:

Las directivas de los archivos representan las descritas en la sección Sección 3.1.4, “Qué es un archivo SPEC”. En las siguientes secciones, se rellenará una sección particular en los archivos de salida de **rpmdev-newspec**.

NOTA

La utilidad **rpmdev-newspec** no utiliza directrices o convenciones específicas para ninguna distribución de Linux en particular. Sin embargo, este documento está dirigido a Red Hat Enterprise Linux, por lo que se prefiere la notación **%{buildroot}** sobre la notación **$RPM_BUILD_ROOT** cuando se hace referencia a Buildroot de RPM para que sea coherente con todas las demás macros definidas o proporcionadas en todo el archivo SPEC.

3.2.3. Modificación de un archivo SPEC original para crear RPMs

El siguiente procedimiento muestra cómo modificar el archivo SPEC de salida proporcionado por **rpmdev-newspec** para crear los RPM.

Requisitos previos

Asegúrate de que:

- El código fuente del programa concreto se ha colocado en el directorio **~/rpmbuild/SOURCES/**.
- El archivo SPEC despoblado **~/rpmbuild/SPECS/<name>.spec** ha sido creado por la utilidad **rpmdev-newspec**.

Procedimiento

1. Abra la plantilla de salida del archivo **~/rpmbuild/SPECS/<name>.spec** proporcionada por la utilidad **rpmdev-newspec**:
2. Rellene la primera sección del archivo SPEC:
   La primera sección incluye estas directivas que **rpmdev-newspec** agrupa:
   - **Name**
   - **Version**
   - **Release**
   - **Summary**
     La dirección **Name** ya se había especificado como argumento de **rpmdev-newspec**.

     Establezca el **Version** para que coincida con la versión de lanzamiento del código fuente.

     El **Release** se establece automáticamente en **1%{?dist}**, que es inicialmente **1**. Incrementa el valor inicial cada vez que se actualiza el paquete sin que haya un cambio en la versión upstream **Version** - como cuando se incluye un parche. Restablezca **Release** a **1** cuando se produzca una nueva liberación del upstream.

     El sitio web **Summary** es una breve explicación de una línea de lo que es este software.
3. Rellene las directivas **License**, **URL**, y **Source0**:
   El campo **License** es la licencia de software asociada con el código fuente de la versión anterior. El formato exacto de cómo etiquetar el **License** en su archivo SPEC variará dependiendo de las directrices específicas de la distribución de Linux basada en RPM que esté siguiendo.

   Por ejemplo, puede utilizar la GPLv3 .

El campo **URL** proporciona la URL del sitio web del software de origen. Por coherencia, utilice la macrovariable RPM de **%{name}**, y use **https://example.com/%{name}**.

El campo **Source0** proporciona la URL del código fuente del software de origen. Debe enlazar directamente con la versión específica del software que se está empaquetando. Tenga en cuenta que los ejemplos de URL que se ofrecen en esta documentación incluyen valores codificados que pueden cambiar en el futuro. Del mismo modo, la versión de lanzamiento también puede cambiar. Para simplificar estos posibles cambios futuros, utilice las macros **%{name}** y **%{version}**. Al utilizarlas, sólo es necesario actualizar un campo en el archivo SPEC.

4. Rellene las directivas **BuildRequires**, **Requires** y **BuildArch**:
   BuildRequires especifica las dependencias en tiempo de compilación para el paquete.

   Requires especifica las dependencias en tiempo de ejecución del paquete.

   Se trata de un software escrito en un lenguaje de programación interpretado sin extensiones compiladas de forma nativa. Por lo tanto, añade la directiva **BuildArch** con el valor **noarch**. Esto le dice a RPM que este paquete no necesita estar ligado a la arquitectura del procesador en el que se construye.

5. Rellene las directivas **%description**, **%prep**, **%build**, **%install**, **%files**, y **%license**:
   Estas directivas pueden ser consideradas como encabezados de sección, ya que son directivas que pueden definir tareas de varias líneas, de varias instrucciones o de secuencias de comandos.

   El **scription** es una descripción más larga y completa del software que el **Summary**, que contiene uno o más párrafos.

   La sección **%prep** especifica cómo preparar el entorno de construcción. Esto normalmente implica la expansión de archivos comprimidos del código fuente, la aplicación de parches y, potencialmente, el análisis de la información proporcionada en el código fuente para su uso en una parte posterior del archivo SPEC. En esta sección se puede utilizar la macro incorporada **%setup -q**.

   La sección **%build** especifica cómo construir el software.

   La sección **%install** contiene instrucciones para **rpmbuild** sobre cómo instalar el software, una vez construido, en el directorio **BUILDROOT**.

   Este directorio es un directorio base chroot vacío, que se asemeja al directorio raíz del usuario final. Aquí puede crear cualquier directorio que contenga los archivos instalados. Para crear estos directorios, puede utilizar las macros RPM sin tener que codificar las rutas.

   La sección **%files** especifica la lista de archivos proporcionados por este RPM y su ubicación completa en el sistema del usuario final.

   Dentro de esta sección, puede indicar el papel de varios archivos utilizando macros incorporadas. Esto es útil para consultar los metadatos del manifiesto del archivo del paquete utilizando el comando[]**rpm**. Por ejemplo, para indicar que el archivo LICENSE es un archivo de licencia de software, utilice la macro **%license**.

6. La última sección, **%changelog**, es una lista de entradas con fecha para cada versión del paquete. Registran los cambios de empaquetado, no los cambios de software. Ejemplos de cambios en el paquete: añadir un parche, cambiar el procedimiento de construcción en la sección **%build**.
   Siga este formato para la primera línea:

Comienza con un carácter * seguido de **Day-of-Week Month Day Year Name Surname <email> - Version-Release**

Siga este formato para la entrada del cambio real:

- Cada entrada de modificación puede contener varios elementos, uno por cada modificación.
- Cada elemento comienza en una nueva línea.
- Cada elemento comienza con un carácter **-**.

Ahora ha escrito un archivo SPEC completo para el programa requerido.

Para ver ejemplos de archivos SPEC escritos en diferentes lenguajes de programación, consulte:

- Un ejemplo de archivo SPEC para un programa escrito en bash
- Un ejemplo de archivo SPEC para un programa escrito en Python
- Un ejemplo de archivo SPEC para un programa escrito en C

La construcción del RPM a partir del archivo SPEC se describe en Sección 3.3, “Construir RPMs”.

3.2.4. Un ejemplo de archivo SPEC para un programa escrito en bash

Esta sección muestra un archivo SPEC de ejemplo para el programa **bello** que fue escrito en bash. Para más información sobre **bello**, consulte Sección 2.1.1, “Ejemplos de código fuente” .

Un archivo SPEC de ejemplo para el programa bello escrito en 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
```

La directiva **BuildRequires**, que especifica las dependencias en tiempo de compilación para el paquete, fue eliminada porque no hay ningún paso de compilación para **bello**. Bash es un lenguaje de programación interpretado en bruto, y los archivos sólo se instalan en su ubicación en el sistema.

La directiva **Requires**, que especifica las dependencias en tiempo de ejecución para el paquete, incluye sólo **bash**, porque el script **bello** sólo requiere el entorno de shell **bash** para ejecutarse.

La sección **%build**, que especifica cómo construir el software, está en blanco, porque no es necesario construir un **bash**.

Para instalar **bello** sólo es necesario crear el directorio de destino e instalar allí el archivo de script ejecutable **bash**. Por lo tanto, puede utilizar el comando **install** en la sección **%install**. Las macros RPM permiten hacer esto sin codificar las rutas.

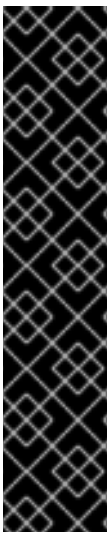
3.2.5. Un ejemplo de archivo SPEC para un programa escrito en Python

Esta sección muestra un archivo SPEC de ejemplo para el programa **pello** escrito en el lenguaje de programación Python. Para más información sobre **pello**, véase Sección 2.1.1, “Ejemplos de código fuente”.

Un archivo SPEC de ejemplo para el programa pello escrito en 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

El programa **pello** está escrito en un lenguaje interpretado compilado en bytes. Por lo tanto, el shebang no es aplicable porque el archivo resultante no contiene la entrada.

Como el tinglado no es aplicable, puede aplicar uno de los siguientes enfoques:

- Cree un script de shell no compilado que llame al ejecutable.
- Proporcionar un pequeño trozo de código Python que no esté compilado en bytes como punto de entrada a la ejecución del programa.

Estos enfoques son útiles sobre todo para grandes proyectos de software con muchos miles de líneas de código, donde el aumento de rendimiento del código precompilado es considerable.

La directiva **BuildRequires**, que especifica las dependencias en tiempo de compilación del paquete, incluye dos paquetes:

- El paquete **python** es necesario para realizar el proceso de compilación de bytes
- El paquete **bash** es necesario para ejecutar el pequeño script de entrada

La directiva **Requires**, que especifica las dependencias en tiempo de ejecución del paquete, incluye sólo el paquete **python**. El programa **pello** requiere el paquete **python** para ejecutar el código compilado en bytes en tiempo de ejecución.

La sección **%build**, que especifica cómo construir el software, corresponde al hecho de que el software está compilado en bytes.

Para instalar **pello**, es necesario crear un script envolvente porque el shebang no es aplicable en los lenguajes compilados en bytes. Hay múltiples opciones para lograr esto, como:

- Hacer un script separado y utilizarlo como una directiva separada **SourceX**.
- Creación del archivo en línea en el archivo SPEC.

Este ejemplo muestra la creación de un script envolvente en línea en el archivo SPEC para demostrar que el archivo SPEC en sí mismo es scriptable. Este script envolvente ejecutará el código compilado en bytes de Python utilizando un documento **here**.

La sección **%install** en este ejemplo también corresponde al hecho de que tendrá que instalar el archivo compilado en bytes en un directorio de la biblioteca en el sistema de tal manera que se pueda acceder a él.

3.2.6. Un ejemplo de archivo SPEC para un programa escrito en C

Esta sección muestra un archivo SPEC de ejemplo para el programa **cello** que fue escrito en el lenguaje de programación C. Para más información sobre **cello**, véase Sección 2.1.1, “Ejemplos de código fuente” .

Un archivo SPEC de ejemplo para el programa cello escrito en 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
```

La directiva **BuildRequires**, que especifica las dependencias en tiempo de compilación para el paquete, incluye dos paquetes que son necesarios para realizar el proceso de compilación:

- El paquete **gcc**
- El paquete **make**

La directiva **Requires**, que especifica las dependencias en tiempo de ejecución para el paquete, se omite en este ejemplo. Todos los requisitos de tiempo de ejecución son manejados por **rpmbuild**, y el programa **cello** no requiere nada fuera de las bibliotecas estándar del núcleo de C.

La sección **%build** refleja el hecho de que en este ejemplo se escribió un **Makefile** para el programa **cello**, por lo que se puede utilizar el comando make de GNU proporcionado por la utilidad **rpmdev-newspec**. Sin embargo, es necesario eliminar la llamada a **%configure** porque no se proporcionó un script de configuración.

La instalación del programa **cello** puede realizarse utilizando la macro **%make_install** que fue proporcionada por el comando **rpmdev-newspec**. Esto es posible porque el **Makefile** para el programa **cello** está disponible.

3.3. CONSTRUIR RPMS

Esta sección describe cómo construir un RPM después de haber creado un archivo SPEC para un programa.

Los RPM se construyen con el comando **rpmbuild**. Este comando espera una determinada estructura de directorios y archivos, que es la misma que la estructura que se configuró con la utilidad **rpmdev-setuptree**.

Diferentes casos de uso y resultados deseados requieren diferentes combinaciones de argumentos para el comando **rpmbuild**. Esta sección describe los dos casos de uso principales:

- Creación de RPMs de origen
- Creación de RPM binarios

3.3.1. Creación de RPMs de origen

Este párrafo es la introducción del módulo del procedimiento: una breve descripción del procedimiento.

Requisitos previos

Debe existir un archivo SPEC para el programa que queremos empaquetar. Para obtener más información sobre la creación de archivos SPEC, consulte Trabajar con archivos SPEC .

Procedimiento

El siguiente procedimiento describe cómo construir un RPM de origen.

- Ejecute el comando **rpmbuild** con el archivo SPEC especificado:
  -

```
$ rpmbuild -bs SPECFILE
```

Sustituya *SPECFILE* por el archivo SPEC. La opción **-bs** representa la fuente de construcción.

El siguiente ejemplo muestra la construcción de RPMs fuente para los proyectos **bello**, **pello**, y **cello**.

Construyendo RPMs de origen para bello, pello y cello.

```
$ 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
```

Pasos de verificación

- Asegúrese de que el directorio **rpmbuild/SRPMS** incluye los RPMs fuente resultantes. El directorio es una parte de la estructura esperada por **rpmbuild**.

3.3.2. Creación de RPM binarios

Los siguientes métodos están disponibles para construir RPMs binarios:

- Reconstrucción de un RPM binario a partir de un RPM fuente
- Construir un RPM binario a partir del archivo SPEC
- Construir un RPM binario a partir de un RPM fuente

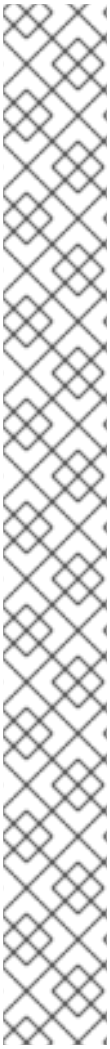
3.3.2.1. Reconstrucción de un RPM binario a partir de un RPM fuente

El siguiente procedimiento muestra cómo reconstruir un RPM binario a partir de un RPM fuente (SRPM).

Procedimiento

- Para reconstruir **bello**, **pello**, y **cello** desde sus SRPMs, ejecute:

```
$ 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

Invocar **rpmbuild --rebuild** implica:

- Instalar el contenido del SRPM -el archivo SPEC y el código fuente- en el directorio ~/**rpmbuild**/.
- Construir utilizando los contenidos instalados.
- Eliminación del archivo SPEC y del código fuente.

Para conservar el archivo SPEC y el código fuente después de la construcción, puede:

- Al construir, utilice el comando **rpmbuild** con la opción **--recompile** en lugar de la opción **--rebuild**.
- Instale los SRPMs utilizando estos 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%]
```

La salida generada al crear un RPM binario es verbosa, lo cual es útil para la depuración. La salida varía para diferentes ejemplos y corresponde a sus archivos SPEC.

Los RPM binarios resultantes se encuentran en el directorio ~/**rpmbuild**/**RPMS**/**YOURARCH** donde **YOURARCH** es su arquitectura o en el directorio ~/**rpmbuild**/**RPMS**/**noarch**/, si el paquete no es específico de la arquitectura.

3.3.2.2. Construir un RPM binario a partir del archivo SPEC

El siguiente procedimiento muestra cómo construir **bello**, **pello**, y **cello** RPMs binarios desde sus archivos SPEC.

Procedimiento

- Ejecute el comando **rpmbuild** con la opción **bb**:

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

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

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

3.3.2.3. Construcción de RPMs a partir de RPMs fuente

También es posible construir cualquier tipo de RPM a partir de un RPM fuente. Para ello, utilice el siguiente procedimiento.

Procedimiento

- Ejecute el comando **rpmbuild** con una de las siguientes opciones y con el paquete fuente especificado:

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

Recursos adicionales

Para más detalles sobre la construcción de RPMs a partir de RPMs fuente, consulte la sección **BUILDING PACKAGES** en la página man **rpmbuild(8)**.

3.4. COMPROBACIÓN DE LOS RPM PARA LA SANIDAD

Después de crear un paquete, compruebe la calidad del mismo.

La principal herramienta para comprobar la calidad de los paquetes es rpmlint.

La herramienta **rpmlint** hace lo siguiente:

- Mejora la capacidad de mantenimiento de las RPM.
- Permite la comprobación de la sanidad realizando un análisis estático del RPM.
- Activa la comprobación de errores realizando un análisis estático del RPM.

La herramienta **rpmlint** puede comprobar los RPM binarios, los RPM fuente (SRPM) y los archivos SPEC, por lo que es útil para todas las etapas del empaquetado, como se muestra en los siguientes ejemplos.

Tenga en cuenta que **rpmlint** tiene unas directrices muy estrictas, por lo que a veces es aceptable saltarse algunos de sus errores y advertencias, como se muestra en los siguientes ejemplos.

NOTA

En los siguientes ejemplos, **rpmlint** se ejecuta sin ninguna opción, lo que produce una salida no verbosa. Para obtener explicaciones detalladas de cada error o advertencia, puede ejecutar **rpmlint -i** en su lugar.

3.4.1. Comprobando la cordura de Bello

Esta sección muestra las posibles advertencias y errores que pueden producirse al comprobar la sanidad del RPM en el ejemplo del archivo SPEC de bello y el RPM binario de bello.

3.4.1.1. Comprobación del archivo bello SPEC

Ejemplo 3.2. Resultado de la ejecución del comando rpmlint en el archivo SPEC para 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.
```

En el caso de **bello.spec**, sólo hay una advertencia, que dice que la URL indicada en la directiva **Source0** es inalcanzable. Esto es esperado, porque la URL **example.com** especificada no existe. Suponiendo que esperamos que esta URL funcione en el futuro, podemos ignorar esta advertencia.

Ejemplo 3.3. Resultado de la ejecución del comando rpmlint en el SRPM para 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 el **bello** SRPM, hay una nueva advertencia, que dice que la URL especificada en la directiva **URL** es inalcanzable. Asumiendo que el enlace funcionará en el futuro, podemos ignorar esta advertencia.

3.4.1.2. Comprobación del RPM binario de bello

Al comprobar los RPM binarios, **rpmlint** comprueba los siguientes elementos:

- Documentación
- Páginas del manual
- Uso coherente del estándar de la jerarquía del sistema de archivos

Ejemplo 3.4. Resultado de la ejecución del comando rpmlint en el RPM binario de 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.
```

Las advertencias de **no-documentation** y **no-manual-page-for-binary** dicen que el RPM no tiene documentación o páginas de manual, porque no proporcionamos ninguna. Aparte de las advertencias anteriores, el RPM pasó las comprobaciones de **rpmlint**.

3.4.2. Comprobando la cordura de Pello

Esta sección muestra las posibles advertencias y errores que pueden ocurrir al comprobar la sanidad del RPM en el ejemplo del archivo pello SPEC y el RPM binario pello.

3.4.2.1. Comprobación del archivo pello SPEC

Ejemplo 3.5. Resultado de la ejecución del comando rpmlint en el archivo 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.
```

La advertencia **invalid-url Source0** dice que la URL indicada en la directiva **Source0** es inalcanzable. Esto es esperado, porque la URL **example.com** especificada no existe. Suponiendo que esta URL funcionará en el futuro, puede ignorar esta advertencia.

Los errores de **hardcoded-library-path** sugieren utilizar la macro **%{_libdir}** en lugar de codificar la ruta de la biblioteca. Para este ejemplo, puede ignorar estos errores. Sin embargo, para los paquetes que van a producción, asegúrese de comprobar todos los errores cuidadosamente.

Ejemplo 3.6. Resultado de la ejecución del comando rpmlint en el 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.
```

El nuevo error de **invalid-url URL** se refiere a la directiva **URL**, que es inalcanzable. Asumiendo que la URL será válida en el futuro, puedes ignorar este error con seguridad.

3.4.2.2. Comprobación del RPM binario de pello

Al comprobar los RPM binarios, **rpmlint** comprueba los siguientes elementos:

- Documentación
- Páginas del manual
- Uso coherente de la norma de jerarquía de sistemas de archivos

Ejemplo 3.7. Resultado de la ejecución del comando rpmlint en el RPM binario 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.
```

Las advertencias de **no-documentation** y **no-manual-page-for-binary** dicen que el RPM no tiene documentación o páginas de manual, porque no ha proporcionado ninguna.

La advertencia de **only-non-binary-in-usr-lib** dice que usted proporcionó sólo artefactos no binarios en **/usr/lib/**. Este directorio está normalmente reservado para archivos de objetos compartidos, que son archivos binarios. Por lo tanto, **rpmlint** espera que al menos uno o más archivos del directorio **/usr/lib/** sean binarios.

Este es un ejemplo de una comprobación de **rpmlint** para el cumplimiento de la norma de jerarquía del sistema de archivos. Normalmente, utilice las macros de RPM para asegurar la colocación correcta de los archivos. Por el bien de este ejemplo, puede ignorar con seguridad esta advertencia.

El error **non-executable-script** advierte que el archivo **/usr/lib/pello/pello.py** no tiene permisos de ejecución. La herramienta **rpmlint** espera que el archivo sea ejecutable, porque el archivo contiene el shebang. Para el propósito de este ejemplo, puede dejar este archivo sin permisos de ejecución e ignorar este error.

Aparte de las advertencias y errores mencionados, el RPM pasó las comprobaciones de **rpmlint**.

3.4.3. Comprobación de la cordura del chelo

Esta sección muestra las posibles advertencias y errores que pueden ocurrir al comprobar la sanidad del RPM en el ejemplo del archivo SPEC de cello y el RPM binario de pello.

3.4.3.1. Comprobación del archivo SPEC de cello

Ejemplo 3.8. Resultado de la ejecución del comando rpmlint en el archivo SPEC para cello

```
$ 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.
```

En el caso de **cello.spec**, sólo hay una advertencia, que dice que la URL indicada en la directiva **Source0** es inalcanzable. Esto es de esperar, porque la URL **example.com** especificada no existe. Suponiendo que esta URL funcionará en el futuro, puede ignorar esta advertencia.

Ejemplo 3.9. Resultado de la ejecución del comando rpmlint en el SRPM para cello

```
$ 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 el **cello** SRPM, hay una nueva advertencia, que dice que la URL especificada en la directiva **URL** es inalcanzable. Asumiendo que el enlace funcionará en el futuro, puedes ignorar esta advertencia.

3.4.3.2. Comprobación de las RPM binarias del chelo

Al comprobar los RPM binarios, **rpmlint** comprueba los siguientes elementos:

- Documentación
- Páginas del manual
- Uso coherente del estándar de la jerarquía del sistema de archivos

Ejemplo 3.10. Resultado de la ejecución del comando rpmlint en el RPM binario para cello

```
$ 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.
```

Las advertencias de **no-documentation** y **no-manual-page-for-binary** dicen que el RPM no tiene documentación o páginas de manual, porque no has proporcionado ninguna. Aparte de las advertencias anteriores, el RPM pasó las comprobaciones de **rpmlint**.

3.5. REGISTRO DE LA ACTIVIDAD DE RPM EN SYSLOG

Cualquier actividad o transacción de RPM puede ser registrada por el protocolo de registro del sistema (syslog).

Requisitos previos

- Para habilitar el registro de las transacciones de RPM en syslog, asegúrese de que el plug-in **syslog** está instalado en el sistema.

```
# yum install rpm-plugin-syslog
```

NOTA

La ubicación por defecto de los mensajes syslog es el archivo **/var/log/messages**. Sin embargo, puede configurar syslog para que utilice otra ubicación para almacenar los mensajes.

Para ver las actualizaciones de la actividad de RPM, siga este procedimiento.

Procedimiento

1. Abra el archivo que configuró para almacenar los mensajes syslog, o si utiliza la configuración syslog por defecto, abra el archivo **/var/log/messages**.
2. Busca nuevas líneas que incluyan la cadena **[RPM]**.

assembly_archiving-rpms.adoc :parent-context-of-archiving-rpms: packaging-software

3.6. EXTRAER EL CONTENIDO DEL RPM

En casos particulares, por ejemplo, si un paquete requerido por RPM está dañado, es necesario extraer

el contenido del paquete. En estos casos, si una instalación de RPM sigue funcionando a pesar del daño, puede utilizar la utilidad **rpm2archive** para convertir un archivo **.rpm** en un archivo tar para utilizar el contenido del paquete.

Esta sección describe cómo convertir una carga útil rpm en un archivo tar.

NOTA

Si la instalación RPM está muy dañada, puede utilizar la utilidad **rpm2cpio** para convertir el archivo del paquete RPM en un archivo cpio.

3.6.1. Convertir los RPM en archivos tar

Para convertir los paquetes RPM en archivos tar, puede utilizar la utilidad **rpm2archive**.

Procedimiento

- Ejecute el siguiente comando:

```
$ rpm2archive file.rpm
```

El archivo resultante tiene el sufijo **.tgz**. Por ejemplo, para archivar el paquete **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. TEMAS AVANZADOS

Esta sección cubre temas que están más allá del alcance del tutorial introductorio pero que son útiles en el empaquetado de RPM en el mundo real.

4.1. PAQUETES DE FIRMAS

Los paquetes están firmados para garantizar que ningún tercero pueda alterar su contenido. Un usuario puede añadir una capa adicional de seguridad utilizando el protocolo HTTPS al descargar el paquete.

Hay tres formas de firmar un paquete:

- Sección 4.1.2, "Añadir una firma a un paquete ya existente" .
- Sección 4.1.5, "Sustitución de la firma en un paquete ya existente" .
- Sección 4.1.6, "Firmar un paquete en el momento de la compilación" .

Para poder firmar un paquete, es necesario crear una clave GNU Privacy Guard (GPG) como se describe en Sección 4.1.1, "Creación de una clave GPG" .

4.1.1. Creación de una clave GPG

Procedimiento

1. Generar un par de claves GNU Privacy Guard (GPG):

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

2. Confirme y vea la clave generada:

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

3. Exportar la clave pública:

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

NOTA

Incluya el nombre real que ha seleccionado para la clave en lugar de <Nombre_de_la_clave>.

4. Importar la clave pública exportada a una base de datos RPM:

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

4.1.2. Añadir una firma a un paquete ya existente

Esta sección describe el caso más habitual cuando un paquete se construye sin firma. La firma se añade justo antes de la publicación del paquete.

Para añadir una firma a un paquete, utilice la opción **--addsign** que ofrece el paquete **rpm-sign**.

Tener más de una firma permite registrar la ruta de propiedad del paquete desde el constructor del paquete hasta el usuario final.

Procedimiento

- Añadir una firma a un paquete:

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

NOTA

Se supone que debes introducir la contraseña para desbloquear la clave secreta de la firma.

4.1.3. Comprobación de las firmas de un paquete con múltiples firmas

Procedimiento

- Para comprobar las firmas de un paquete con múltiples firmas, ejecute lo siguiente:

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

Las dos cadenas **pgp** en la salida del comando **rpm --checksig** muestran que el paquete ha sido firmado dos veces.

4.1.4. Un ejemplo práctico de cómo añadir una firma a un paquete ya existente

Esta sección describe una situación de ejemplo en la que puede ser útil añadir una firma a un paquete ya existente.

Una división de una empresa crea un paquete y lo firma con la clave de la división. A continuación, la sede central de la empresa comprueba la firma del paquete y añade la firma corporativa al paquete, indicando que el paquete firmado es auténtico.

Con dos firmas, el paquete llega a un minorista. El minorista comprueba las firmas y, si coinciden, añade también la suya.

El paquete llega ahora a una empresa que quiere desplegarlo. Tras comprobar todas las firmas del paquete, saben que se trata de una copia auténtica. Dependiendo de los controles internos de la empresa que lo despliega, puede optar por añadir su propia firma para informar a sus empleados de que el paquete ha recibido su aprobación corporativa

4.1.5. Sustitución de la firma en un paquete ya existente

Este procedimiento describe cómo cambiar la clave pública sin tener que reconstruir cada paquete.

Procedimiento

- Para cambiar la clave pública, ejecute lo siguiente:

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

NOTA

Se supone que debes introducir la contraseña para desbloquear la clave secreta de la firma.

La opción **--resign** también permite cambiar la clave pública de varios paquetes, como se muestra en el siguiente procedimiento.

Procedimiento

- Para cambiar la clave pública de varios paquetes, ejecute:

```
$ rpm --resign b*.rpm
```

NOTA

Se supone que debes introducir la contraseña para desbloquear la clave secreta de la firma.

4.1.6. Firmar un paquete en el momento de la compilación

Procedimiento

1. Construya el paquete con el comando **rpmbuild**:

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

2. Firme el paquete con el comando **rpmsign** utilizando la opción **--addsign**:

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

3. Opcionalmente, verificar la firma de un paquete:

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

NOTA

Cuando construya y firme varios paquetes, utilice la siguiente sintaxis para evitar introducir la frase de contraseña Pretty Good Privacy (PGP) varias veces.

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

Tenga en cuenta que debe introducir la contraseña para desbloquear la clave secreta de la firma.

4.2. MÁS SOBRE LAS MACROS

Esta sección cubre algunas macros incorporadas a RPM. Para una lista exhaustiva de dichas macros, consulte la documentación de RPM.

4.2.1. Definir sus propias macros

La siguiente sección describe cómo crear una macro personalizada.

Procedimiento

- Incluya la siguiente línea en el archivo RPM SPEC:

```
%global <name>[(opts)] <body>
```

Se eliminan todos los espacios en blanco que rodean a \. El nombre puede estar compuesto por caracteres alfanuméricos y el carácter _ y debe tener una longitud mínima de 3 caracteres. La inclusión del campo **(opts)** es opcional:

- **Simple** las macros no contienen el campo **(opts)**. En este caso, sólo se realiza la expansión recursiva de las macros.
- las macros**Parametrized** contienen el campo **(opts)**. La cadena **opts** entre paréntesis se pasa a **getopt(3)** para el procesamiento de **argc/argv** al principio de la invocación de una macro.

NOTA

Los archivos RPM SPEC más antiguos utilizan el patrón de macros **fine <name> <body>** en su lugar. Las diferencias entre las macros **fine** y **%global** son las siguientes:

- **fine** tiene alcance local. Se aplica a una parte específica de un archivo SPEC. El cuerpo de una macro **fine** se expande cuando se utiliza.
- **%global** tiene un alcance global. Se aplica a todo un archivo SPEC. El cuerpo de una macro **%global** se expande en el momento de la definición.

IMPORTANTE

Las macros se evalúan incluso si se comentan o el nombre de la macro se indica en la sección **%changelog** del archivo SPEC. Para comentar una macro, utilice **%%**. Por ejemplo: **%%global**.

Recursos adicionales

Para obtener información completa sobre las capacidades de las macros, consulte la documentación de RPM.

4.2.2. Uso de la macro %setup

Esta sección describe cómo construir paquetes con los tarballs de código fuente utilizando diferentes variantes de la macro **%setup**. Tenga en cuenta que las variantes de la macro pueden combinarse La salida **rpmbuild** ilustra el comportamiento estándar de la macro **%setup**. Al principio de cada fase, la macro da salida a **Executing(%...)**, como se muestra en el siguiente ejemplo.

Ejemplo 4.1. Ejemplo %setup salida de la macro

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

La salida del shell se establece con **set -x** activado. Para ver el contenido de **/var/tmp/rpm-**

tmp.DhddsG, utilice la opción **--debug** porque **rpmbuild** borra los archivos temporales después de una construcción exitosa. Esto muestra la configuración de las variables de entorno seguidas, por ejemplo:

```
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 .
```

La macro **%setup**:

- Asegura que estamos trabajando en el directorio correcto.
- Elimina los residuos de construcciones anteriores.
- Descomprime el tarball fuente.
- Establece algunos privilegios por defecto.

4.2.2.1. Utilizando la macro %setup -q

La opción **-q** limita la verbosidad de la macro **%setup**. Sólo se ejecuta **tar -xof** en lugar de **tar -xvvof**. Utilice esta opción como primera opción.

4.2.2.2. Utilizando la macro %setup -n

La opción **-n** se utiliza para especificar el nombre del directorio del tarball expandido.

Se utiliza en los casos en que el directorio del tarball expandido tiene un nombre diferente al esperado (**%{name}-%{version}**), lo que puede provocar un error de la macro **%setup**.

Por ejemplo, si el nombre del paquete es **cello**, pero el código fuente está archivado en **hello-1.0.tgz** y contiene el directorio **hello**/, el contenido del archivo SPEC debe ser el siguiente:

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

4.2.2.3. Utilizando la macro %setup -c

La opción **-c** se utiliza si el tarball del código fuente no contiene ningún subdirectorio y después de desempaquetarlo, los ficheros de un archivo llenan el directorio actual.

La opción **-c** crea entonces el directorio y los pasos en la expansión del archivo como se muestra a continuación:

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

El directorio no se modifica tras la ampliación del archivo.

4.2.2.4. Uso de las macros %setup -D y %setup -T

La opción **-D** desactiva el borrado del directorio del código fuente, y es particularmente útil si la macro **%setup** se utiliza varias veces. Con la opción **-D**, no se utilizan las siguientes líneas:

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

La opción **-T** desactiva la expansión del tarball del código fuente eliminando la siguiente línea del script:

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

4.2.2.5. Uso de las macros %setup -a y %setup -b

Las opciones **-a** y **-b** amplían fuentes específicas:

La opción **-b** significa **before**, y expande fuentes específicas antes de entrar en el directorio de trabajo. La opción **-a** significa **after**, y expande esas fuentes después de entrar. Sus argumentos son números de fuentes del preámbulo del archivo SPEC.

En el siguiente ejemplo, el archivo **cello-1.0.tar.gz** contiene un directorio vacío **examples**. Los ejemplos se envían en un tarball separado **examples.tar.gz** y se expanden en el directorio del mismo nombre. En este caso, utilice **-a 1**, si desea expandir **Source1** después de entrar en el directorio de trabajo:

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

En el siguiente ejemplo, los ejemplos se proporcionan en un tarball separado **cello-1.0-examples.tar.gz**, que se expande en **cello-1.0/examples**. En este caso, utilice **-b 1**, para expandir **Source1** antes de entrar en el directorio de trabajo:

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

4.2.3. Macros comunes de RPM en la sección les

Esta sección enumera las macros RPM avanzadas que se necesitan en la sección **%files** de un archivo SPEC.

Tabla 4.1. Macros RPM avanzadas en la sección%files

Macro	Definición
%license	La macro identifica el archivo listado como un archivo LICENSE y será instalado y etiquetado como tal por RPM. Ejemplo **%license LICENSE**
%doc	La macro identifica un archivo listado como documentación y será instalado y etiquetado como tal por RPM. La macro se utiliza para la documentación sobre el software empaquetado y también para los ejemplos de código y diversos elementos de acompañamiento. En el caso de que se incluyan ejemplos de código, se debe tener cuidado de eliminar el modo ejecutable del archivo. Ejemplo **%doc README**
%dir	La macro asegura que la ruta es un directorio propiedad de este RPM. Esto es importante para que el manifiesto del archivo RPM sepa con precisión qué directorios debe limpiar al desinstalar. Ejemplo **%dir %{_libdir}/%{name}**
%config(noreplace)	La macro asegura que el siguiente archivo es un archivo de configuración y por lo tanto no debe ser sobrescrito (o reemplazado) en una instalación o actualización de paquete si el archivo ha sido modificado desde la suma de control de la instalación original. Si hay un cambio, el archivo se creará con **.rpmnew** añadido al final del nombre del archivo en el momento de la actualización o instalación para que no se modifique el archivo preexistente o modificado en el sistema de destino. Ejemplo **%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

4.2.4. Visualización de las macros incorporadas

Red Hat Enterprise Linux proporciona múltiples macros RPM incorporadas.

Procedimiento

1. Para mostrar todas las macros RPM incorporadas, ejecute:

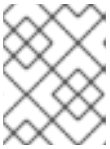
```
rpm --showrc
```

NOTA

El resultado es bastante grande. Para reducir el resultado, utilice el comando anterior con el comando **grep**.

2. Para encontrar información sobre las macros RPMs para la versión de RPM de su sistema, ejecute

```
rpm -ql rpm
```



NOTA

Las macros RPM son los archivos titulados **macros** en la estructura del directorio de salida.

4.2.5. Macros de distribución RPM

Diferentes distribuciones proporcionan diferentes conjuntos de macros RPM recomendados basados en la implementación del lenguaje del software que se está empaquetando o en las directrices específicas de la distribución.

Los conjuntos de macros RPM recomendados suelen proporcionarse como paquetes RPM, listos para ser instalados con el gestor de paquetes **yum**.

Una vez instalados, los archivos de macros se encuentran en el directorio **/usr/lib/rpm/macros.d/**.

Para mostrar las definiciones de las macros RPM en bruto, ejecute:

```
rpm --showrc
```

La salida anterior muestra las definiciones de macros RPM en bruto.

Para determinar qué hace una macro y cómo puede ser útil al empaquetar RPMs, ejecute el comando **rpm --eval** con el nombre de la macro utilizada como argumento:

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

Para más información, consulte la página de manual **rpm**.

4.2.5.1. Creación de macros personalizadas

Puede anular las macros de distribución en el archivo **~/.rpmmacros** con sus macros personalizadas. Cualquier cambio que realices afectará a todas las compilaciones de tu máquina.

AVISO

No se recomienda definir nuevas macros en el archivo **~/.rpmmacros**. Dichas macros no estarían presentes en otras máquinas, donde los usuarios podrían intentar reconstruir su paquete.

Para anular una macro, ejecute :

```
%_topdir /opt/some/working/directory/rpmbuild
```

Puede crear el directorio del ejemplo anterior, incluyendo todos los subdirectorios a través de la utilidad **rpmdev-setuptree**. El valor de esta macro es por defecto **~/rpmbuild**.

```
%_smp_mflags -l3
```

La macro anterior se utiliza a menudo para pasar a Makefile, por ejemplo **make %{?_smp_mflags}**, y para establecer un número de procesos concurrentes durante la fase de construcción. Por defecto, se establece en **-jX**, donde **X** es un número de núcleos. Si se altera el número de núcleos, se puede acelerar o ralentizar la construcción de paquetes.

4.3. EPOCH, SCRIPTLETS Y TRIGGERS

Esta sección cubre **Epoch**, **Scriptlets**, y **Triggers**, que representan directivas avanzadas para los archivos RMP SPEC.

Todas estas directivas influyen no sólo en el archivo SPEC, sino también en la máquina final en la que se instala el RPM resultante.

4.3.1. La directiva de la época

La directiva **Epoch** permite definir dependencias ponderadas en función del número de versión.

Si esta directiva no aparece en el archivo RPM SPEC, la directiva **Epoch** no se establece en absoluto. Esto es contrario a la creencia común de que no establecer **Epoch** resulta en un **Epoch** de 0. Sin embargo, la utilidad YUM trata un **Epoch** no establecido como lo mismo que un **Epoch** de 0 para los propósitos de depsolución.

Sin embargo, el listado de **Epoch** en un archivo SPEC suele omitirse porque en la mayoría de los casos la introducción de un valor **Epoch** sesga el comportamiento esperado de RPM al comparar versiones de paquetes.

> **Ejemplo 4.2. Uso de Epoch**
>
> Si usted instala el paquete **foobar** con **Epoch: 1** y **Version: 1.0**, y alguien más empaqueta **foobar** con **Version: 2.0** pero sin la directiva **Epoch**, la nueva versión nunca será considerada una actualización. La razón es que se prefiere la versión **Epoch** sobre el marcador tradicional **Name-Version-Release** que significa el versionado de los paquetes RPM.

Por lo tanto, el uso de **Epoch** es bastante raro. Sin embargo, **Epoch** se suele utilizar para resolver un problema de ordenación de actualizaciones. El problema puede aparecer como efecto secundario de un cambio en los esquemas de números de versión del software o de versiones que incorporan caracteres alfabéticos que no siempre pueden compararse de forma fiable basándose en la codificación.

4.3.2. Scriptlets

Scriptlets son una serie de directivas RPM que se ejecutan antes o después de instalar o eliminar paquetes.

Utilice **Scriptlets** sólo para las tareas que no pueden realizarse en tiempo de construcción o en un script de inicio.

4.3.2.1. Directivas Scriptlets

Existe un conjunto de directivas comunes de **Scriptlet**. Son similares a las cabeceras de sección de los archivos SPEC, como **%build** o **%install**. Están definidas por segmentos de código de varias líneas, que a menudo se escriben como un script de shell POSIX estándar. Sin embargo, también pueden escribirse en otros lenguajes de programación que RPM para la distribución de la máquina de destino acepte. La documentación de RPM incluye una lista exhaustiva de los lenguajes disponibles.

La siguiente tabla incluye las directivas **Scriptlet** listadas en su orden de ejecución. Tenga en cuenta que un paquete que contiene los scripts se instala entre la directiva **%pre** y **%post**, y se desinstala entre la directiva **%preun** y **%postun**.

Tabla 4.2. Directivas Scriptlet

Directiva	Definición
%pretrans	Scriptlet que se ejecuta justo antes de instalar o eliminar cualquier paquete.
%pre	Scriptlet que se ejecuta justo antes de instalar el paquete en el sistema de destino.
%post	Scriptlet que se ejecuta justo después de instalar el paquete en el sistema de destino.
%preun	Scriptlet que se ejecuta justo antes de desinstalar el paquete del sistema de destino.
%postun	Scriptlet que se ejecuta justo después de desinstalar el paquete del sistema de destino.
%posttrans	Scriptlet que se ejecuta al final de la transacción.

4.3.2.2. Desactivación de la ejecución de un scriptlet

Para desactivar la ejecución de cualquier scriptlet, utilice el comando **rpm** junto con la opción **--no_scriptlet_name_**.

Procedimiento

- Por ejemplo, para desactivar la ejecución de los scriptlets de **%pretrans**, ejecute

```
# rpm --nopretrans
```

  También puede utilizar la opción **-- noscripts**, que equivale a todo lo siguiente:

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

Recursos adicionales

- Para más detalles, consulte la página de manual **rpm(8)**.

4.3.2.3. Macros Scriptlets

Las directivas **Scriptlets** también funcionan con las macros RPM.

El siguiente ejemplo muestra el uso de la macro scriptlet systemd, que asegura que systemd sea notificado sobre un nuevo archivo de unidad.

```
$ 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. Las directivas Triggers

Triggers son directivas de RPM que proporcionan un método de interacción durante la instalación y desinstalación de paquetes.

AVISO

Triggers puede ser ejecutado en un momento inesperado, por ejemplo en la actualización del paquete que lo contiene. **Triggers** son difíciles de depurar, por lo tanto necesitan ser implementados de una manera robusta para que no rompan nada cuando se ejecuten inesperadamente. Por estas razones, Red Hat recomienda minimizar el uso de **Triggers**.

El orden de ejecución y los detalles de cada uno de los **Triggers** existentes se enumeran a continuación:

```
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
```

Los elementos anteriores se encuentran en el archivo **/usr/share/doc/rpm-4.*/triggers**.

4.3.4. Uso de scripts que no son de Shell en un archivo SPEC

La opción **-p** scriptlet en un archivo SPEC permite al usuario invocar un intérprete específico en lugar del intérprete de scripts de shell por defecto (**-p /bin/sh**).

El siguiente procedimiento describe cómo crear una secuencia de comandos que imprima un mensaje tras la instalación del programa **pello.py**:

Procedimiento

1. Abra el archivo **pello.spec**.
2. Encuentra la siguiente línea:

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

3. Debajo de la línea anterior, insértese:

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

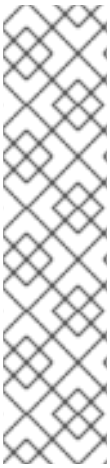
4. Construya su paquete como se describe en Sección 3.3, “Construir RPMs”.

5. Instala tu paquete:

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

6. Compruebe el mensaje de salida después de la instalación:

```
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 utilizar un script de Python 3, incluya la siguiente línea en **install -m** en un archivo SPEC:

```
%post -p /usr/bin/python3
```

Para utilizar un script Lua, incluya la siguiente línea en **install -m** en un archivo SPEC:

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

De esta manera, puede especificar cualquier intérprete en un archivo SPEC.

4.4. CONDICIONALES DE RPM

Los condicionales RPM permiten la inclusión condicional de varias secciones del archivo SPEC.

Las inclusiones condicionales suelen tratar:

- Secciones específicas de la arquitectura
- Secciones específicas del sistema operativo
- Problemas de compatibilidad entre varias versiones de sistemas operativos
- Existencia y definición de macros

4.4.1. Sintaxis de los condicionales RPM

Los condicionales RPM utilizan la siguiente sintaxis:

Si *expression* es cierto, entonces haz alguna acción:

```
%if expression
…
%endif
```

Si *expression* es verdadero, entonces haz alguna acción, en otro caso, haz otra acción:

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

4.4.2. Ejemplos de condicionales RPM

Esta sección proporciona múltiples ejemplos de condicionales RPM.

4.4.2.1. Los condicionales %if

Ejemplo 4.3. Uso del condicional %if para manejar la compatibilidad entre Red Hat Enterprise Linux 8 y otros sistemas operativos

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

Este condicional maneja la compatibilidad entre RHEL 8 y otros sistemas operativos en términos de soporte de la macro AS_FUNCTION_DESCRIBE. Si el paquete está construido para RHEL, la macro **%rhel** está definida, y se expande a la versión de RHEL. Si su valor es 8, lo que significa que el paquete está construido para RHEL 8, entonces las referencias a AS_FUNCTION_DESCRIBE, que no está soportada por RHEL 8, se eliminan de los scripts de autoconfiguración.

Ejemplo 4.4. Uso del condicional %if para manejar la definición 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 maneja la definición de las macros. Si se definen las macros **%milestone** o **%revision**, se redefine la macro **%ruby_archive**, que define el nombre del tarball ascendente.

4.4.2.2. Variantes especializadas de los condicionales %if

Las condicionales **%ifarch**, **%ifnarch** y **%ifos** son variantes especializadas de las condicionales **%if**. Estas variantes son de uso común, por lo que tienen sus propias macros.

4.4.2.2.1. El condicional %ifarch

El condicional **%ifarch** se utiliza para comenzar un bloque del archivo SPEC que es específico de la arquitectura. Va seguido de uno o más especificadores de arquitectura, cada uno separado por comas o espacios en blanco.

Ejemplo 4.5. Un ejemplo de uso de la condicional %ifarch

```
%ifarch i386 sparc
…
%endif
```

Todo el contenido del archivo SPEC entre **%ifarch** y **%endif** se procesa sólo en las arquitecturas AMD e Intel de 32 bits o en los sistemas basados en Sun SPARC.

4.4.2.2.2. El condicional %ifnarch

El condicional **%ifnarch** tiene una lógica inversa al condicional **%ifarch**.

Ejemplo 4.6. Un ejemplo de uso de la condicional %ifnarch

```
%ifnarch alpha
…
%endif
```

Todo el contenido del archivo SPEC entre **%ifnarch** y **%endif** se procesa sólo si no se hace en un sistema basado en Digital Alpha/AXP.

4.4.2.2.3. El condicional %ifos

El condicional **%ifos** se utiliza para controlar el procesamiento basado en el sistema operativo de la compilación. Puede ir seguido de uno o más nombres de sistemas operativos.

Ejemplo 4.7. Un ejemplo de uso de la condicional %ifos

```
%ifos linux
…
%endif
```

Todo el contenido del archivo SPEC entre **%ifos** y **%endif** se procesa sólo si la construcción se hizo en un sistema Linux.

4.5. EMPAQUETADO DE RPMS DE PYTHON 3

La mayoría de los proyectos de Python utilizan Setuptools para el empaquetado, y definen la información del paquete en el archivo **setup.py**. Para más información sobre el empaquetado de Setuptools, consulte la documentación de Setuptools.

También puedes empaquetar tu proyecto Python en un paquete RPM, que proporciona las siguientes ventajas en comparación con el empaquetado de Setuptools:

- Especificación de las dependencias de un paquete con otros RPM (incluso los que no son de Python)
- Firma criptográfica
  Con la firma criptográfica, el contenido de los paquetes RPM puede ser verificado, integrado y probado con el resto del sistema operativo.

4.5.1. Descripción típica del archivo SPEC para un paquete RPM de Python

Un archivo RPM SPEC para proyectos de Python tiene algunas especificidades en comparación con los archivos RPM SPEC que no son de Python. En particular, el nombre de cualquier paquete RPM de una biblioteca de Python debe incluir siempre el prefijo **python3**.

En el siguiente archivo SPEC se muestran otros datos específicos **example for thepython3-detox package**. Para la descripción de estos detalles, consulte las notas debajo del ejemplo.

```
%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. La macro **modname** contiene el nombre del proyecto Python. En este ejemplo es **detox**.
2. Cuando se empaqueta un proyecto Python en RPM, el prefijo **python3** siempre debe añadirse al nombre original del proyecto. El nombre original aquí es **detox** y el **name of the RPM** es **python3-detox**.
3. **BuildRequires** especifica qué paquetes son necesarios para construir y probar este paquete. En BuildRequires, incluya siempre los elementos que proporcionan las herramientas necesarias para construir paquetes de Python: **python36-devel** y **python3-setuptools**. El paquete **python36-rpm-macros** es necesario para que los archivos con **/usr/bin/python3** shebangs se cambien automáticamente a **/usr/bin/python3.6**. Para más información, consulte Sección 4.5.4, "Manejo de hashbangs en scripts de Python".
4. Cada paquete de Python requiere algunos otros paquetes para funcionar correctamente. Dichos paquetes deben especificarse también en el archivo SPEC. Para especificar el **dependencies**, puede utilizar la macro **%python_enable_dependency_generator** para utilizar automáticamente las dependencias definidas en el archivo **setup.py**. Si un paquete tiene dependencias que no se especifican usando Setuptools, especifíquelas dentro de las directivas adicionales **Requires**.
5. Las macros **%py3_build** y **%py3_install** ejecutan los comandos **setup.py build** y **setup.py install**, respectivamente, con argumentos adicionales para especificar las ubicaciones de instalación, el intérprete a utilizar y otros detalles.
6. La sección **check** proporciona una macro que ejecuta la versión correcta de Python. La macro **%{__python3}** contiene una ruta para el intérprete de Python 3, por ejemplo **/usr/bin/python3**. Recomendamos utilizar siempre la macro en lugar de una ruta literal.

4.5.2. Macros comunes para paquetes RPM de Python 3

En un archivo SPEC, utilice siempre las siguientes macros en lugar de codificar sus valores.

En los nombres de las macros, utilice siempre **python3** o **python2** en lugar de **python** sin versionar.

Macro	Definición normal	Descripción
%{__python3}	/usr/bin/python3	Intérprete de Python 3
%{python3_version}	3.6	La versión completa del intérprete de Python 3.
%{python3_sitelib}	/usr/lib/python3.6/paquetes-sitio	Donde se instalan los módulos de Python puro.
%{python3_sitearch}	/usr/lib64/python3.6/site-packages	Donde se instalan los módulos que contienen extensiones específicas de la arquitectura.

Macro	Definición normal	Descripción
%py3_build		Ejecuta el comando **setup.py build** con argumentos adecuados para un paquete del sistema.
%py3_install		Ejecuta el comando **setup.py install** con argumentos adecuados para un paquete del sistema.

4.5.3. Proporciona automáticamente los paquetes RPM de Python

Al empaquetar un proyecto Python, asegúrese de que, si están presentes, los siguientes directorios se incluyan en el RPM resultante:

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

A partir de estos directorios, el proceso de compilación de RPM genera automáticamente los suministros virtuales **pythonX.Ydist**, por ejemplo **python3.6dist(detox)**. Estas provisiones virtuales son utilizadas por los paquetes especificados por la macro **%python_enable_dependency_generator**.

4.5.4. Manejo de hashbangs en scripts de Python

En Red Hat Enterprise Linux 8, se espera que los scripts ejecutables de Python usen hashbangs (shebangs) especificando explícitamente al menos la versión principal de Python.

El script **/usr/lib/rpm/redhat/brp-mangle-shebangs** buildroot policy (BRP) se ejecuta automáticamente al construir cualquier paquete RPM, e intenta corregir los hashbangs en todos los archivos ejecutables. El script BRP generará errores cuando encuentre un script de Python con un hashbang ambiguo, como:

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

o

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

Para modificar los hashbangs en los scripts de Python que causan estos errores de compilación en el momento de construir el RPM, utilice el script **pathfix.py** del paquete **platform-python-devel**:

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

Se pueden especificar múltiples ***PATH*s** pueden ser especificados. Si a ***PATH*** es un directorio,

pathfix.py busca recursivamente cualquier script de Python que coincida con el patrón **^[a-zA-Z0-9_]\.py$**, no sólo los que tengan un hashbang ambiguo. Añade este comando a la sección **%prep** o al final de la sección **%install**.

Alternativamente, modifique los scripts de Python empaquetados para que se ajusten al formato esperado. Para este propósito, **pathfix.py** puede ser usado fuera del proceso de construcción del RPM, también. Cuando ejecute **pathfix.py** fuera de una compilación RPM, sustituya ***__python3*** del ejemplo anterior por una ruta para el hashbang, como **/usr/bin/python3**.

Si los scripts de Python empaquetados requieren la versión 2 de Python, sustituya el número 3 por el 2 en los comandos anteriores.

Además, los hashbangs en la forma **/usr/bin/python3** son reemplazados por defecto con hashbangs que apuntan a Python desde el paquete **platform-python** utilizado para las herramientas del sistema con Red Hat Enterprise Linux.

Para cambiar los hashbangs **/usr/bin/python3** en sus paquetes personalizados para que apunten a una versión de Python instalada desde el flujo de aplicaciones, en la forma **/usr/bin/python3.6**, añada el paquete **python36-rpm-macros** en la sección **BuildRequires** del archivo SPEC:

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

NOTA

Para evitar la comprobación del hashbang y su modificación por el script BRP, utilice la siguiente directiva RPM:

```
%undefine p_mangle_shebangs
```

4.6. PAQUETES RUBYGEMS

Esta sección explica qué son los paquetes RubyGems y cómo reempaquetarlos en RPM.

4.6.1. Qué son las RubyGems

Ruby es un lenguaje de programación dinámico, interpretado, reflexivo, orientado a objetos y de propósito general.

Los programas escritos en Ruby suelen empaquetarse utilizando el proyecto RubyGems, que proporciona un formato de empaquetado específico para Ruby.

Los paquetes creados por RubyGems se llaman gemas, y también pueden ser reempaquetados en RPM.

NOTA

Esta documentación se refiere a los términos relacionados con el concepto de RubyGems con el prefijo **gem**, por ejemplo .gemspec se utiliza para el **gem specification**, y los términos relacionados con RPM no se califican.

4.6.2. Cómo se relacionan las RubyGems con el RPM

Las RubyGems representan el formato de empaquetado propio de Ruby. Sin embargo, las RubyGems contienen metadatos similares a los que necesita RPM, lo que permite la conversión de RubyGems a RPM.

Según las directrices de empaquetado de Ruby , es posible volver a empaquetar los paquetes de RubyGems en RPM de esta manera:

- Dichas RPMs encajan con el resto de la distribución.
- Los usuarios finales pueden satisfacer las dependencias de una gema instalando la gema adecuada empaquetada en RPM.

RubyGems utiliza una terminología similar a la de RPM, como archivos SPEC, nombres de paquetes, dependencias y otros elementos.

Para encajar en el resto de la distribución RPM de RHEL, los paquetes creados por RubyGems deben seguir las convenciones indicadas a continuación:

- Los nombres de las gemas deben seguir este patrón:

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

- Para implementar una línea shebang, se debe utilizar la siguiente cadena:

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

4.6.3. Creación de paquetes RPM a partir de paquetes RubyGems

Esta sección describe cómo crear paquetes RPM a partir de paquetes creados por RubyGems.

Para crear un RPM fuente para un paquete RubyGems, se necesitan dos archivos:

- Un archivo de gemas
- Un archivo RPM SPEC

4.6.3.1. Convenciones de los archivos SPEC de RubyGems

Un archivo SPEC de RubyGems debe cumplir las siguientes convenciones:

- Contiene una definición de **%{gem_name}**, que es el nombre de la especificación de la gema.
- El origen del paquete debe ser la URL completa del archivo de la gema liberada; la versión del paquete debe ser la versión de la gema.
- Contiene el **BuildRequires:** una directiva definida de la siguiente manera para poder tirar de las macros necesarias para construir.

```
BuildRequires:rubygems-devel
```

- No contener ninguna RubyGems **Requires** o **Provides**, porque son autogeneradas.
- No contener la directiva **BuildRequires:** definida como sigue, a menos que quiera especificar explícitamente la compatibilidad con la versión de Ruby:

```
Requiere: ruby(release)
```

La dependencia generada automáticamente de RubyGems (**Requires: ruby(rubygems)**) es suficiente.

Macros
Las macros útiles para los paquetes creados por RubyGems son proporcionadas por los paquetes **rubygems-devel**.

Tabla 4.3. Macros de RubyGems

Nombre de la macro	Ruta ampliada	Uso
%{gem_dir}	/usr/share/gems	Directorio superior para la estructura de la gema.
%{gem_instdir}	%{gem_dir}/gems/%{gem_name}-%{version}	Directorio con el contenido real de la gema.
%{gem_libdir}	%{gem_instdir}/lib	El directorio de la biblioteca de la gema.
%{gem_cache}	%{gem_dir}/cache/%{gem_name}-%{version}.gem	La gema en caché.
%{gem_spec}	%{gem_dir}/especificaciones/%{gem_name}-%{version}.gemspec	El archivo de especificación de gemas.
%{gem_docdir}	%{gem_dir}/doc/%{gem_name}-%{version}	La documentación RDoc de la gema.
%{gem_extdir_mri}	%{_libdir}/gems/ruby/%{gem_name}-%{version}	El directorio para la extensión de la gema.

4.6.3.2. Ejemplo de archivo RubyGems SPEC

Esta sección proporciona un archivo SPEC de ejemplo para construir gemas junto con una explicación de sus secciones particulares.

Un ejemplo de archivo SPEC de 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}/
```

En la siguiente tabla se explican los elementos específicos de un archivo SPEC de RubyGems:

Tabla 4.4. Especificaciones de las directivas SPEC de RubyGems

Directiva SPEC	Especificaciones de RubyGems
%prep	RPM puede desempaquetar directamente archivos de gemas, por lo que puede ejecutar la comamnd **gem unpack** para extraer el código fuente de la gema. La macro **%setup -n %{gem_name}-%{version}** proporciona el directorio en el que se ha desempaquetado la gema. En el mismo nivel de directorio, se crea automáticamente el archivo **%{gem_name}-%{version}.gemspec**, que puede utilizarse para reconstruir la gema más tarde, para modificar el **.gemspec**, o para aplicar parches al código.

Directiva SPEC	Especificaciones de RubyGems
%build	Esta directiva incluye comandos o series de comandos para construir el software en código máquina. La macro **%gem_install** opera sólo en archivos gem, y la gema se recrea con la siguiente construcción de gem. El archivo gem que se crea es utilizado por **%gem_install** para construir e instalar el código en el directorio temporal, que es **./%{gem_dir}** por defecto. La macro**%gem_install** construye e instala el código en un solo paso. Antes de ser instalado, las fuentes construidas se colocan en un directorio temporal que se crea automáticamente. La macro **%gem_install** acepta dos opciones adicionales: **-n <gem_file>**, que permite anular la gema utilizada para la instalación, y **-d <install_dir>**, que podría anular el destino de instalación de la gema; no se recomienda utilizar esta opción. La macro **%gem_install** no debe utilizarse para instalar en el **%{buildroot}**.
%instalación	La instalación se realiza en la jerarquía **%{buildroot}**. Puedes crear los directorios que necesites y luego copiar lo que se instaló en los directorios temporales en la jerarquía de **%{buildroot}**. Si esta gema crea objetos compartidos, se mueven a la ruta específica de la arquitectura **%{gem_extdir_mri}**.

Para más información sobre los archivos SPEC de RubyGems, véase Ruby Packaging Guidelines.

4.6.3.3. Conversión de paquetes RubyGems a archivos RPM SPEC con gem2rpm

La utilidad **gem2rpm** convierte los paquetes RubyGems en archivos RPM SPEC.

4.6.3.3.1. Instalación de gem2rpm

Procedimiento

- Para instalar **gem2rpm** desde RubyGems.org, ejecute:

```
$ gem install gem2rpm
```

4.6.3.3.2. Mostrar todas las opciones de gem2rpm

Procedimiento

- Para ver todas las opciones de **gem2rpm**, ejecute:

```
gem2rpm --help
```

4.6.3.3.3. Uso de gem2rpm para convertir paquetes RubyGems en archivos RPM SPEC

Procedimiento

- Descargar una gema en su última versión, y generar el archivo RPM SPEC para esta gema:

```
$ gem2rpm --fetch <nombre-del-gem> > <nombre-del-gem>.spec
```

El procedimiento descrito crea un archivo RPM SPEC basado en la información proporcionada en los metadatos de la gema. Sin embargo, la gema omite alguna información importante que se suele proporcionar en los RPM, como la licencia y el registro de cambios. Por lo tanto, el archivo SPEC generado debe ser editado.

4.6.3.3.4. Edición de plantillas de gem2rpm

Se recomienda editar la plantilla a partir de la cual se genera el archivo RPM SPEC en lugar del propio archivo SPEC generado.

La plantilla es un archivo estándar de Embedded Ruby (ERB), que incluye las variables enumeradas en la siguiente tabla.

Tabla 4.5. Variables en la plantilla gem2rpm

Variable	Explicación
paquete	La variable **Gem::Package** para la gema.
especificación	La variable **Gem::Specification** para la gema (la misma que format.spec).
config	La variable **Gem2Rpm::Configuration** que puede redefinir las macros o reglas por defecto utilizadas en los ayudantes de las plantillas de especificaciones.
dependencias en tiempo de ejecución	La variable **Gem2Rpm::RpmDependencyList** que proporciona una lista de dependencias de tiempo de ejecución del paquete.
dependencias_de_desarrollo	La variable **Gem2Rpm::RpmDependencyList** proporciona una lista de dependencias de desarrollo de paquetes.
pruebas	La variable **Gem2Rpm::TestSuite** proporciona una lista de marcos de pruebas que permiten su ejecución.
archivos	La variable **Gem2Rpm::RpmFileList** proporciona una lista no filtrada de archivos en un paquete.
archivos_principales	La variable **Gem2Rpm::RpmFileList** proporciona una lista de archivos adecuados para el paquete principal.

Variable	Explicación
archivos_doc	La variable **Gem2Rpm::RpmFileList** proporciona una lista de archivos adecuados para el subpaquete **-doc**.
formato	La variable **Gem::Format** para la gema. Tenga en cuenta que esta variable es ahora obsoleta.

Procedimiento

- Para ver todas las plantillas disponibles, ejecute:

```
$ gem2rpm --templates
```

Para editar las plantillas de **gem2rpm**, siga este procedimiento:

Procedimiento

1. Guarde la plantilla por defecto:

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

2. Edite la plantilla según sea necesario.
3. Genere el archivo SPEC utilizando la plantilla editada:

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

Ahora puede construir un paquete RPM utilizando la plantilla editada como se describe en Sección 3.3, “Construir RPMs”.

4.7. CÓMO MANEJAR PAQUETES RPM CON SCRIPTS PERLS

En RHEL 8, el lenguaje de programación Perl no está incluido en el buildroot por defecto. Por lo tanto, los paquetes RPM que incluyen scripts de Perl deben indicar explícitamente la dependencia de Perl utilizando la directiva **BuildRequires:** en el archivo RPM SPEC.

4.7.1. Dependencias comunes relacionadas con Perl

Las dependencias de compilación relacionadas con Perl que se utilizan con más frecuencia en **BuildRequires:** son :

- **perl-generators**
  Genera automáticamente **Requires** y **Provides** en tiempo de ejecución para los archivos Perl instalados. Cuando se instala un script de Perl o un módulo de Perl, se debe incluir una dependencia de construcción de este paquete.
- **perl-interpreter**

El intérprete de Perl debe ser listado como una dependencia de construcción si es llamado de alguna manera, ya sea explícitamente a través del paquete **perl** o la macro **%__perl**, o como parte del sistema de construcción de su paquete.

- **perl-devel**
  Proporciona archivos de cabecera de Perl. Si se construye código específico de la arquitectura que se enlaza con la biblioteca **libperl.so**, como un módulo XS Perl, debe incluirse **BuildRequires: perl-devel**.

4.7.2. Utilización de un módulo Perl específico

Si se requiere un módulo Perl específico en el momento de la construcción, utilice el siguiente procedimiento:

Procedimiento

- Aplique la siguiente sintaxis en su archivo RPM SPEC:

```
BuildRequires: perl(MODULE)
```

NOTA

Aplique esta sintaxis también a los módulos del núcleo de Perl, porque pueden entrar y salir del paquete **perl** con el tiempo.

4.7.3. Limitación de un paquete a una versión específica de Perl

Para limitar su paquete a una versión específica de Perl, siga este procedimiento:

Procedimiento

- Utilice la dependencia **perl(:VERSION)** con la restricción de versión deseada en su archivo RPM SPEC:
  Por ejemplo, para limitar un paquete a la versión 5.22 o superior de Perl, utilice

```
BuildRequires: perl(:VERSION) >= 5.22
```

AVISO

No utilice una comparación con la versión del paquete **perl** porque incluye un número de época.

4.7.4. Garantizar que un paquete utiliza el intérprete de Perl correcto

Red Hat proporciona múltiples intérpretes de Perl, que no son totalmente compatibles. Por lo tanto, cualquier paquete que entregue un módulo Perl debe utilizar en tiempo de ejecución el mismo intérprete Perl que se utilizó en tiempo de compilación.

Para ello, siga el siguiente procedimiento:

Procedimiento

- Incluya la versión **MODULE_COMPAT Requires** en el archivo RPM SPEC para cualquier paquete que entregue un módulo Perl:

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

CAPÍTULO 5. NUEVAS CARACTERÍSTICAS DE RHEL 8

Esta sección documenta los cambios más notables en el empaquetado RPM entre Red Hat Enterprise Linux 7 y 8.

5.1. APOYO A LAS DEPENDENCIAS DÉBILES

5.1.1. Introducción a la política de dependencia débil

Weak dependencies son variantes de la directiva **Requires**. Estas variantes se comparan con los nombres de paquetes y **Provides:** virtuales utilizando comparaciones de rango de **Epoch-Version-Release**.

Weak dependencies tiene dos fuerzas (**weak** y **hint**) y dos direcciones (**forward** y **backward**), como se resume en la siguiente tabla.

NOTA

La dirección **forward** es análoga a **Requires:**. La dirección **backward** no tiene ningún análogo en el sistema de dependencia anterior.

Tabla 5.1. Posibles combinaciones de puntos fuertes y direcciones de las dependencias débiles

Fuerza/Dirección	Adelante	Hacia atrás
Débil	Recomienda:	Suplementos:
Sugerencia	Sugiere:	Mejora:

Las principales ventajas de la política **Weak dependencies** son:

- Permite instalaciones mínimas más pequeñas, manteniendo la característica de la instalación por defecto.
- Los paquetes pueden especificar las preferencias por proveedores específicos, manteniendo la flexibilidad de los proveedores virtuales.

5.1.1.1. Dependencias débiles

Por defecto, **Weak dependencies** se trata de forma similar a la regular **Requires:**. Los paquetes coincidentes se incluyen en la **YUM** transacción. Si al añadir el paquete se produce un error **YUM** por defecto ignora la dependencia. Por lo tanto, los usuarios pueden excluir los paquetes que serían añadidos por **Weak dependencies** o eliminarlos posteriormente.

Condiciones de uso
Puede utilizar **Weak dependencies** sólo si el paquete sigue funcionando sin la dependencia.

NOTA

Es aceptable crear paquetes con una funcionalidad muy limitada sin añadir ninguno de sus requisitos débiles.

Casos de uso
Utilice **Weak dependencies** especialmente cuando sea posible minimizar la instalación para casos de uso razonables, como la construcción de máquinas virtuales o contenedores que tienen un único propósito y no requieren el conjunto completo de características del paquete.

Los casos típicos de uso de **Weak dependencies** son:

- Documentación
  - Los espectadores de la documentación si se pierden se maneja con gracia
- Ejemplos
- Plug-ins o complementos
  - Soporte para formatos de archivo
  - Apoyo a los protocolos

5.1.1.2. Consejos

Hints son ignorados por defecto por **YUM**. Pueden ser utilizados por las herramientas de la GUI para ofrecer paquetes adicionales que no se instalan por defecto pero que pueden ser útiles en combinación con los paquetes instalados.

No utilice **Hints** para los requisitos de los principales casos de uso de un paquete. Incluya en su lugar dichos requisitos en strong o **Weak dependencies**.

Preferencia de paquetes
YUM utiliza **Weak dependencies** y **Hints** para decidir qué paquete utilizar si hay que elegir entre varios paquetes igualmente válidos. Se prefieren los paquetes a los que apuntan las dependencias de los paquetes instalados o por instalar.

Tenga en cuenta que las reglas normales de resolución de dependencias no se ven afectadas por esta característica. Por ejemplo, **Weak dependencies** no puede obligar a elegir una versión más antigua de un paquete.

Si hay varios proveedores para una dependencia, el paquete solicitante puede añadir un **Suggests:** para proporcionar una pista al resolvedor de la dependencia sobre qué opción es la preferida.

Enhances: sólo se utiliza cuando el paquete principal y otros proveedores están de acuerdo en que añadir la pista al paquete requerido es, por alguna razón, la solución más limpia.

Ejemplo 5.1. Utilizar los consejos para preferir un paquete a otro

```
Package A: Requires: mysql

Package mariadb: Provides: mysql

Package community-mysql: Provides: mysql
```

Si quiere preferir el paquete **mariadb** al paquete **community-mysql** → utilice:

```
Sugiere: mariadb al paquete A.
```

5.1.1.3. Dependencias hacia delante y hacia atrás

Forward dependencies son, de forma similar a **Requires**, evaluados para los paquetes que se están instalando. También se instalan los mejores paquetes que coinciden.

En general, prefiera **Forward dependencies**. Añade la dependencia al paquete cuando consigas añadir el otro paquete al sistema.

En el caso de **Backward dependencies**, los paquetes que contienen la dependencia se instalan si también se instala un paquete coincidente.

Backward dependencies está diseñado principalmente para los proveedores de terceros que pueden adjuntar sus plug-ins, complementos o extensiones a la distribución o a otros paquetes de terceros.

5.2. APOYO A LAS RELACIONES BOOLEANAS

A partir de la versión 4.13, RPM es capaz de procesar expresiones booleanas en las siguientes dependencias:

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

Esta sección describe la sintaxis de las relaciones booleanas, proporciona una lista de operadores booleanos y explica el anidamiento de las relaciones booleanas, así como la semántica de las mismas.

5.2.1. Sintaxis de las relaciones booleanas

Las expresiones booleanas siempre van entre paréntesis.

Se construyen a partir de las dependencias normales:

- Sólo nombre o nombre
- Comparación
- Descripción de la versión

5.2.2. Operadores booleanos

RPM 4.13 introdujo los siguientes operadores booleanos:

Tabla 5.2. Operadores booleanos introducidos con RPM 4.13

Operador booleano	Descripción	Ejemplo de uso

Operador booleano	Descripción	Ejemplo de uso
and	Requiere que se cumplan todos los operandos para que el término sea verdadero.	Conflictos: (pkgA y pkgB)
or	Requiere que uno de los operandos se cumpla para que el término sea verdadero.	Requiere: (pkgA >= 3.2 o pkgB)
if	Requiere que el primer operando se cumpla si el segundo lo hace. (implicación inversa)	Recomienda: (myPkg-langCZ si langsupportCZ)
if else	Igual que el operador **if**, el plus requiere que el tercer operando se cumpla si el segundo no lo hace.	Requiere: myPkg-backend-mariaDB if mariaDB else sqlite

RPM 4.14 introdujo los siguientes operadores booleanos adicionales:

Tabla 5.3. Operadores booleanos introducidos con RPM 4.14

Operador booleano	Descripción	Ejemplo de uso
with	Requiere que todos los operandos sean cumplidos por el mismo paquete para que el término sea verdadero.	Requiere: (pkgA-foo con pkgA-bar)
without	Requiere un único paquete que satisfaga el primer operando pero no el segundo. (sustracción de conjuntos)	Requiere: (pkgA-foo sin pkgA-bar)
unless	Requiere que el primer operando se cumpla si el segundo no lo hace. (implicación negativa inversa)	Conflictos: (myPkg-driverA a menos que driverB)
unless else	Igual que el operador **unless**, más requiere que el tercer operando se cumpla si el segundo lo es.	Conflictos: (myPkg-backend-SDL1 unless myPkg-backend-SDL2 else SDL2)

IMPORTANTE

El operador **if** no puede utilizarse en el mismo contexto que el operador **or**, y el operador **unless** no puede utilizarse en el mismo contexto que **and**.

5.2.3. Nido

Los propios operandos pueden utilizarse como expresiones booleanas, como se muestra en los siguientes ejemplos.

Tenga en cuenta que en este caso, los operandos también deben estar rodeados de paréntesis. Puede encadenar los operadores **and** y **or** repitiendo el mismo operador con un solo conjunto de paréntesis circundantes.

Ejemplo 5.2. Ejemplo de uso de operandos aplicados como expresiones booleanas

```
Requiere: (pkgA o pkgB o pkgC)

Requiere: (pkgA o (pkgB y pkgC))

Suplementos: (foo and (lang-support-cz or lang-support-all))

Requiere: (pkgA con capB) o (pkgB sin capA)

Suplementos: ((conductorA y conductorA-herramientas) a menos que conductorB)

Recomienda: myPkg-langCZ y (font1-langCZ o font2-langCZ) si langsupportCZ
```

5.2.4. Semántica

El uso de **Boolean dependencies** no cambia la semántica de las relaciones regulares.

Si se utiliza **Boolean dependencies**, al comprobar una coincidencia se comprueban todos los nombres y el valor booleano de que haya una coincidencia se agrega sobre los operadores booleanos.

IMPORTANTE

Para todas las dependencias, a excepción de **Conflicts:**, el resultado debe ser **True** para no impedir la instalación. Para **Conflicts:**, el resultado debe ser **False** para no impedir la instalación.

AVISO

Provides no son relaciones y no pueden contener expresiones booleanas.

5.2.4.1. Comprender la salida del operador if

El operador **if** también devuelve un valor booleano, que suele acercarse a lo que se entiende intuitivamente. Sin embargo, los siguientes ejemplos muestran que en algunos casos la comprensión intuitiva de **if** puede ser engañosa.

Ejemplo 5.3. Salidas erróneas del operador if

Esta sentencia es verdadera si pkgB no está instalado. Sin embargo, si esta declaración se utiliza cuando el resultado por defecto es falso, las cosas se complican:

```
Requiere: (pkgA si pkgB)
```

Esta declaración es un conflicto a menos que pkgB esté instalado y pkgA no:

```
Conflictos: (pkgA si pkgB)
```

Así que es mejor que uses:

```
Conflictos: (pkgA y pkgB)
```

Lo mismo ocurre si el operador **if** está anidado en términos de **or**:

```
Requiere: ((pkgA si pkgB) o pkgC o pkg)
```

Esto también hace que el término completo sea verdadero, porque el término **if** es verdadero si pkgB no está instalado. Si pkgA sólo ayuda si pkgB está instalado, utilice **and** en su lugar:

```
Requiere: ((pkgA y pkgB) o pkgC o pkg)
```

5.3. APOYO A LOS ACTIVADORES DE ARCHIVOS

File triggers son un tipo de scriptlets de RPM, que se definen en un archivo SPEC de un paquete.

Al igual que **Triggers**, se declaran en un paquete pero se ejecutan cuando se instala o elimina otro paquete que contiene los archivos correspondientes.

Un uso común de **File triggers** es la actualización de registros o cachés. En este caso, el paquete que contiene o gestiona el registro o la caché debe contener también uno o más **File triggers**. Incluir **File triggers** ahorra tiempo en comparación con la situación en la que el paquete controla la actualización por sí mismo.

5.3.1. Sintaxis de los activadores de archivos

File triggers tienen la siguiente sintaxis:

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

Dónde:

file_trigger_tag define un tipo de activación de archivo. Los tipos permitidos son:

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

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

FILE_TRIGGER_OPTIONS tienen el mismo propósito que las opciones de los scriptlets de RPM, excepto la opción **-P**.

La prioridad de un disparador se define mediante un número. Cuanto mayor sea el número, antes se ejecutará el script de activación de archivos. Los disparadores con prioridad superior a 100000 se ejecutan antes que los scriptlets estándar, y los demás disparadores se ejecutan después de los scriptlets estándar. La prioridad por defecto se establece en 1000000.

Cada disparador de archivos de cada tipo debe contener uno o más prefijos de ruta y guiones.

5.3.2. Ejemplos de sintaxis de activadores de archivos

Esta sección muestra ejemplos concretos de la sintaxis de **File triggers**:

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

Este activador de archivos ejecuta **/usr/bin/ldconfig** directamente después de la instalación de un paquete que contiene un archivo cuya ruta comienza con **/usr/lib** o **/lib**. El activador de archivos se ejecuta sólo una vez, aunque el paquete incluya varios archivos cuya ruta comience por **/usr/lib** o **/lib**. Sin embargo, todos los nombres de archivos que comienzan con **/usr/lib** o **/lib** se pasan a la entrada estándar del script de activación para que pueda filtrar dentro de su script como se muestra a continuación:

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

Este disparador de archivos ejecuta **/usr/bin/ldconfig** para cada paquete que contenga archivos que empiecen por **/usr/lib** y que contengan **foo** al mismo tiempo. Tenga en cuenta que los archivos que coinciden con el prefijo incluyen todo tipo de archivos, incluyendo archivos regulares, directorios, enlaces simbólicos y otros.

5.3.3. Tipos de activadores de archivos

File triggers tiene dos tipos principales:

- Los activadores de archivos se ejecutan una vez por paquete
- Los activadores de archivos se ejecutan una vez por transacción

File triggers se dividen a su vez en función del tiempo de ejecución como sigue:

- Antes o después de la instalación o el borrado de un paquete
- Antes o después de una transacción

5.3.3.1. Se ejecuta una vez por paquete Activadores de archivos

File triggers ejecutado una vez por paquete son:

- letriggerin

- letriggerun
- letriggerpostun

letriggerin
Este activador de archivos se ejecuta tras la instalación de un paquete si éste contiene uno o más archivos que coinciden con el prefijo de este activador. También se ejecuta tras la instalación de un paquete que contiene este activador de archivos y hay uno o más archivos que coinciden con el prefijo de este activador de archivos en la base de datos **rpmdb**.

letriggerun
Este activador de archivos se ejecuta antes de la desinstalación de un paquete si éste contiene uno o más archivos que coinciden con el prefijo de este activador. También se ejecuta antes de la desinstalación de un paquete que contiene este activador de archivos y hay uno o más archivos que coinciden con el prefijo de este activador de archivos en **rpmdb**.

letriggerpostun
Este activador de archivos se ejecuta tras la desinstalación de un paquete si éste contiene uno o más archivos que coinciden con el prefijo de este activador.

5.3.3.2. Se ejecuta una vez por transacción Activadores de archivos

File triggers ejecutado una vez por transacción son:

- %transfiletriggerin
- %transfiletriggerun
- %transfiletriggerpostun

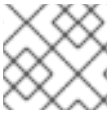
%transfiletriggerin
Este disparador de archivos se ejecuta una vez después de una transacción para todos los paquetes instalados que contengan uno o más archivos que coincidan con el prefijo de este disparador. También se ejecuta después de una transacción si había un paquete que contenía este disparador de archivos en esa transacción y hay uno o más archivos que coinciden con el prefijo de este disparador en **rpmdb**.

%transfiletriggerun
Este disparador de archivos se ejecuta una vez antes de una transacción para todos los paquetes que cumplen las siguientes condiciones:

- El paquete será desinstalado en esta transacción
- El paquete contiene uno o más archivos que coinciden con el prefijo de este disparador

También se ejecuta antes de una transacción si hay un paquete que contiene este disparador de archivos en esa transacción y hay uno o más archivos que coinciden con el prefijo de este disparador en **rpmdb**.

%transfiletriggerpostun
Este disparador de archivos se ejecuta una vez después de una transacción para todos los paquetes desinstalados que contengan uno o más archivos que coincidan con el prefijo de este disparador.



NOTA

La lista de archivos de activación no está disponible en este tipo de activación.

Por lo tanto, si instala o desinstala varios paquetes que contienen bibliotecas, la caché de ldconfig se actualiza al final de toda la transacción. Esto mejora significativamente el rendimiento en comparación con RHEL 7, donde la caché se actualizaba para cada paquete por separado. También los scriptlets que llamaban a ldconfig en %post y %postun en el archivo SPEC de cada paquete ya no son necesarios.

5.3.4. Ejemplo de uso de los activadores de archivos en glibc

Esta sección muestra un ejemplo real de uso de **File triggers** dentro del paquete **glibc**.

En RHEL 8, **File triggers** se implementa en **glibc** para llamar al comando **ldconfig** al final de una transacción de instalación o desinstalación.

Esto se garantiza incluyendo los siguientes scriptlets en el archivo **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
```

Por lo tanto, si se instalan o desinstalan varios paquetes, la caché de **ldconfig** se actualiza para todas las bibliotecas instaladas una vez finalizada toda la transacción. En consecuencia, ya no es necesario incluir los scriptlets que llaman a **ldconfig** en los archivos RPM SPEC de los paquetes individuales. Esto mejora el rendimiento en comparación con RHEL 7, donde la caché se actualizaba para cada paquete por separado.

5.4. PARSER SPEC MÁS ESTRICTO

El analizador SPEC ha incorporado ahora algunos cambios. Por lo tanto, puede identificar nuevos problemas que antes se ignoraban.

5.5. SOPORTE PARA ARCHIVOS DE MÁS DE 4 GB

En Red Hat Enterprise Linux 8, **RPM** puede utilizar variables y etiquetas de 64 bits, lo que permite operar con archivos y paquetes de más de 4 GB.

5.5.1. Etiquetas RPM de 64 bits

Existen varias etiquetas RPM tanto en las versiones de 64 bits como en las versiones anteriores de 32 bits. Tenga en cuenta que las versiones de 64 bits tienen la cadena **LONG** delante de su nombre.

Tabla 5.4. Etiquetas RPM disponibles en versiones de 32 y 64 bits

Nombre de la etiqueta variante de 32 bits	Nombre de la etiqueta variante de 62 bits	Descripción de la etiqueta
RPMTAG_SIGSIZE	RPMTAG_LONGSIGSIZE	Tamaño de la cabecera y de la carga útil comprimida.
RPMTAG_ARCHIVESIZE	RPMTAG_LONGARCHIVESIZE	Tamaño de la carga útil sin comprimir.

Nombre de la etiqueta variante de 32 bits	Nombre de la etiqueta variante de 62 bits	Descripción de la etiqueta
RPMTAG_FILESIZES	RPMTAG_LONGFILESIZES	Conjunto de tamaños de archivos.
RPMTAG_SIZE	RPMTAG_LONGSIZE	Suma de todos los tamaños de archivos.

5.5.1.1. Uso de etiquetas de 64 bits en la línea de comandos

Las extensiones **LONG** están siempre activadas en la línea de comandos. Si anteriormente utilizaba scripts que contenían el comando **rpm -q --qf**, puede añadir **long** al nombre de dichas etiquetas:

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

5.6. OTRAS CARACTERÍSTICAS

Otras nuevas características relacionadas con el empaquetado RPM en Red Hat Enterprise Linux 8 son:

- Salida de comprobación de firmas simplificada en modo no verboso
- Apoyo a la verificación forzada de la carga útil
- Soporte para el modo de comprobación de la firma
- Adiciones y desapariciones en las macros

CAPÍTULO 6. RECURSOS ADICIONALES SOBRE EL EMBALAJE RPM

Esta sección proporciona referencias a varios temas relacionados con los RPMs, el empaquetado de RPMs y la construcción de RPMs. Algunos de ellos son avanzados y amplían el material introductorio incluido en esta documentación.

Visión general de Red Hat Software Coll ections - La oferta de Red Hat Software Collections proporciona herramientas de desarrollo continuamente actualizadas en las últimas versiones estables.

Red Hat Software Collections - El Manual de Empaquetamiento proporciona una explicación de las Colecciones de Software y detalla cómo construirlas y empaquetarlas. Los desarrolladores y administradores de sistemas con conocimientos básicos de empaquetado de software con RPM pueden utilizar este Manual para empezar a utilizar las Colecciones de Software.

Mock - Mock proporciona una solución de construcción de paquetes apoyada por la comunidad para varias arquitecturas y diferentes versiones de Fedora o RHEL que tiene el host de construcción.

Documentación de R PM - La documentación oficial de RPM.

Directrices de empaquetado de Fedora - Las directrices oficiales de empaquetado de Fedora, útiles para todas las distribuciones basadas en RPM.