Red Hat Enterprise Linux 8

打包和分发软件

使用 RPM 软件包管理系统打包软件

Last Updated: 2024-06-27

Red Hat Enterprise Linux 8 打包和分发软件

使用 RPM 软件包管理系统打包软件

法律通告

摘要

使用 RPM 软件包管理器将软件打包到 RPM 软件包中。准备打包的源代码、打包软件并调查高级打包场景，如打包 Python 项目或将 RubyGems 打包到 RPM 软件包中。

目录

对红帽文档提供反馈

我们感谢您对我们文档的反馈。帮助我们如何进行改进。

通过 Jira 提交反馈（需要帐户）

1. 登录到 Jira 网站。
2. 单击顶部导航栏中的 **Create**。
3. 在 **Summary** 字段中输入描述性标题。
4. 在 **Description** 字段中输入您的建议以改进。包括文档相关部分的链接。
5. 点对话框底部的 **Create**。

第 1 章 RPM 简介

RPM Package Manager (RPM)是一个运行在 Red Hat Enterprise Linux (RHEL)、CentOS 和 Fedora 上的软件包管理系统。您可以使用 RPM 为任何这些操作系统所创建的软件进行分发、管理和更新。

与在传统存档文件中分发软件相比，RPM 软件包管理系统有以下优点：

- RPM 以可独立安装、更新或删除软件包的形式管理软件，从而更轻松地维护操作系统。
- RPM 简化了软件的分发，因为 RPM 软件包是独立的二进制文件，类似于压缩的存档。这些软件包是为特定的操作系统和硬件架构而构建的。RPM 包含诸如这样的文件，如在软件包安装时放到文件系统上适当路径下的已编译的可执行文件和库。

使用 RPM，您可以执行以下任务：

- 安装、升级和删除打包的软件。
- 查询关于打包的软件的详细信息。
- 验证打包的软件的完整性。
- 从软件源构建您自己的软件包，并完成构建说明。
- 使用 GNU Privacy Guard (GPG)工具对您的软件包进行数字签名。
- 在 YUM 存储库中发布您的软件包。

在 Red Hat Enterprise Linux 中，RPM 完全集成到更高级别的软件包管理软件中，如 YUM 或 PackageKit。虽然 RPM 提供了自己的命令行界面，但大多数用户只需要通过这个软件与 RPM 进行交互。但是，在构建 RPM 软件包时，您必须使用 RPM 工具，如 **rpmbuild (8)** 。

1.1. RPM 软件包

RPM 软件包由用于安装和删除这些文件的文件和元数据的存档组成。具体来说，RPM 软件包包含以下部分：

GPG 签名

GPG 签名用于验证软件包的完整性。

标头（软件包元数据）

RPM 软件包管理器使用此元数据来确定软件包依赖项、安装文件的位置及其他信息。

payload

有效负载是一个 **cpio** 归档，其中包含要安装到系统的文件。

RPM 软件包有两种类型。这两种类型都共享文件格式和工具，但内容不同，并实现不同的目的：

- 源 RPM（SRPM）
  SRPM 包含源代码和 **spec** 文件，它描述了如何将源代码构建为二进制 RPM。另外，SRPM 可以包含源代码的补丁。

  二进制 RPM

  一个二进制 RPM 包含了根据源代码和补丁构建的二进制文件。

1.2. 列出 RPM 打包工具

除了用于构建软件包的 **rpmbuild (8)** 程序外，RPM 还提供其他工具，以便更轻松地创建软件包。您可以在 **rpmdevtools** 软件包中找到这些程序。

先决条件

- **rpmdevtools** 软件包已安装：

```
# yum install rpmdevtools
```

步骤

- 使用以下方法之一列出 RPM 打包工具：
  - 要列出 **rpmdevtools** 软件包提供的某些工具及其简短描述，请输入：

```
$ rpm -qi rpmdevtools
```

  - 要列出所有工具，请输入：

```
$ rpm -ql rpmdevtools | grep ^/usr/bin
```

其他资源

- RPM 工具手册页

第 2 章 为 RPM 打包创建软件

要为 RPM 打包准备软件，您必须了解什么是源代码以及如何创建软件。

2.1. 什么是源代码

源代码是人类可读的计算机指令，其描述了如何执行计算。源代码是使用编程语言表达的。

以下三种不同编程语言编写的 **Hello World** 程序版本涵盖了主要的 RPM Package Manager 用例：

- 使用 Bash 编写的 **hello World**
  bello 项目使用 Bash 实现 **Hello World**。该实现仅包含 **bello** shell 脚本。此程序的目的是在命令行上输出 **Hello World**。

  bello 文件包含以下内容：

  ```
  #!/bin/bash

  printf "Hello World\n"
  ```

- 使用 Python 编写的 **hello World**
  pello 项目使用 Python 实现 **Hello World**。该实现仅包含 **pello.py** 程序。程序的目的是在命令行中输出 **Hello World**。

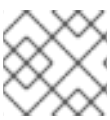
  pello.py 文件包含以下内容：

  ```
  #!/usr/bin/python3

  print("Hello World")
  ```

- 使用 C 编写的 **hello World**
  cello 项目使用 C 实现 **Hello World**。实现仅包含 **cello.c** 和 **Makefile** 文件。因此，生成的 **tar.gz** 存档除了 **LICENSE** 文件外还有两个文件。程序的目的是在命令行中输出 **Hello World**。

  cello.c 文件包含以下内容：

  ```
  #include <stdio.h>

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



注意

对于 **Hello World** 程序的每个版本，打包过程会是不同的。

2.2. 创建软件的方法

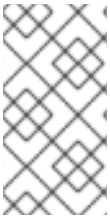
您可以使用以下方法之一将人类可读的源代码转换为机器码：

- 原生编译软件。

- 使用语言解释器或语言虚拟机解释软件。您可以使用原始解释或字节编译软件。

2.2.1. 原生编译的软件

原生编译的软件是使用编程语言编写的软件，使用生成的二进制可执行文件编译到机器代码中。原生编译的软件是独立软件。



注意

原生编译的 RPM 软件包是特定于架构的。

如果您在使用 64 位(x86_64) AMD 或 Intel 处理器的计算机上编译此类软件，则它不能在 32 位(x86) AMD 或 Intel 处理器上运行。生成的软件包在其名称中指定了架构。

2.2.2. 解释的软件

有些编程语言（如 Bash 或 Python ）不会编译成机器码。相反，语言解释器或语言虚拟机会逐步执行程序的源代码步骤，而无需之前进行转换。

注意

完全使用解释编程语言编写的软件不是特定于架构的。因此，生成的 RPM 软件包在其名称中有 **noarch** 字符串。

您可以使用解释语言编写的原始解释或字节编译软件：

- 原始解释的软件
  您不需要编译这类软件。原始解释的软件由解释器直接执行。
- 字节编译的软件
  您必须首先将这类软件编译成字节码，然后由语言虚拟机执行。

注意

有些字节编译的语言可以是原始解释的或字节编译的。

请注意，对于这两个软件类型，使用 RPM 构建和打包软件的方式是不同的。

2.3. 从源构建软件

在软件构建过程中，源代码被转换为可以使用 RPM 打包的软件工件。

2.3.1. 从原生编译的代码构建软件

您可以使用以下方法之一将使用编译语言编写的软件构建成可执行文件：

- 手动构建
- 自动化构建

2.3.1.1. 手动构建示例 C 程序

您可以使用手动构建来构建使用编译语言编写的软件。

使用 C 编写的 **Hello World** 程序示例(**cello.c**)具有以下内容：

```
#include <stdio.h>

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

步骤

1. 从 GNU Compiler Collection 调用 C 编译器，将源代码编译到二进制中：

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

2. 运行生成的二进制文件 **cello** ：

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

2.3.1.2. 为示例 C 程序设置自动构建

大规模软件通常使用自动构建。您可以通过创建 **Makefile** 文件，然后运行 GNU **make** 工具来设置自动构建。

流程

1. 在与 **cello.c** 相同的目录中，创建包含以下内容的 **Makefile** 文件：

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

请注意，**cello:** 和 **clean:** 下的行必须以制表符(tab)开头。

2. 构建软件：

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

3. 因为构建已在当前目录中可用，因此请输入 **make clean** 命令，然后再次输入 **make** 命令：

```
$ make clean
rm cello

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

请注意，此时尝试再次构建程序无效，因为 GNU **make** 系统检测到现有的二进制文件：

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

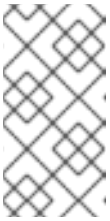
4. 运行程序：

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

2.3.2. 解释源代码

您可以使用以下方法之一将解释编程语言编写的源代码转换为机器码：

- 字节编译
  字节软件的流程因以下因素而异：
  - 编程语言
  - 语言虚拟机
  - 与该语言一起使用的工具和流程



注意

您可以对例如 Python 编写的软件进行字节编译。用于分发的 Python 软件通常是字节编译的，但这不是以本文档中介绍的方法。上述流程目的不是为了符合社区标准，而是为了简单。有关实际工作环境中的 Python 指南，请参阅打包和发布。

  您还可以原始解释 Python 源代码。但是，字节编译的版本速度更快。因此，RPM 软件包程序更喜欢将字节编译的版本打包，来分发给最终用户。

- 原始解释
  使用 shell 脚本语言（如 Bash ）编写的软件总是由原始解释执行。

2.3.2.1. 字节编译示例 Python 程序

通过对 Python 源代码选择字节编译而不是原始解释，您可以创建更快的软件。

使用 Python 编程语言编写的 **Hello World** 程序示例(**pello.py**)具有以下内容：

```
print("Hello World")
```

流程

1. 字节编译 **pello.py** 文件：

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

2. 验证是否已创建了文件的字节编译版本：

```
$ ls __pycache__
pello.cpython-311.pyc
```

请注意，输出中的软件包版本可能会因安装的 Python 版本而有所不同。

3. 在 **pello.py** 中运行程序：

```
$ python pello.py
Hello World
```

2.3.2.2. 原始解释示例 Bash 程序

使用 Bash shell 内置语言(**bello**)编写的 **Hello World** 程序示例具有以下内容：

```
#!/bin/bash

printf "Hello World\n"
```

注意

bello 文件顶部的 **shebang** (**#!**)符号不是编程语言源代码的一部分。

使用 **shebang** 将文本文件转换为可执行文件。系统程序加载程序解析包含 **shebang** 的行，以获取二进制可执行文件的路径，其然后用作编程语言解释器。

流程

1. 使源代码文件可执行：

```
$ chmod +x bello
```

2. 运行创建的文件：

```
$ ./bello
Hello World
```

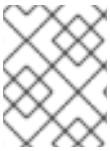
第 3 章 为 RPM 打包准备软件

要准备使用 RPM 打包的软件，您可以首先修补软件，为其创建一个 LICENSE 文件，并将其归档为 tarball。

3.1. 修复软件

在打包软件时，您可能需要对原始源代码进行某些更改，如修复 bug 或更改配置文件。在 RPM 打包中，您可以将原始源代码保持不变，并在其上应用补丁。

补丁是更新源代码文件的一段文本。补丁具有 *diff* 格式，因为它代表文本的两个版本之间的区别。您可以使用 **diff** 实用程序创建补丁，然后使用 patch 实用程序将补丁应用到源代码。



注意

软件开发人员通常使用 Git 等版本控制系统来管理其代码库。这些工具提供了自己的创建区别或给软件打补丁的方法。

3.1.1. 为示例 C 程序创建补丁文件

您可以使用 **diff** 工具从原始源代码创建一个补丁。例如，要修补使用 C 编写 **Hello world** 程序(**cello.c**)，请完成以下步骤。

先决条件

- 在您的系统上安装了 **diff** 工具：

```
# yum install diffutils
```

流程

1. 备份原始源代码：

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

-p 选项保留模式、所有权和时间戳。

2. 根据需要修改 **cello.c** ：

```
#include <stdio.h>

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

3. 生成补丁：

```
$ 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
```

以 **+** 开头的行替换以 **-** 开头的行。

注意

建议使用 **diff** 命令的 **Naur** 选项，因为它适合大多数用例：

- **-N** (**--new-file**)
  -N 选项将缺少的文件处理为空文件。
- **-a** (**--text**)
  -a 选项将所有文件视为文本。因此，**diff** 工具不会忽略它归类为二进制文件的文件。
- **-u** (**-U NUM** 或 **--unified[=NUM]**)
  -u 选项以统一上下文的输出 NUM （默认 3）行的形式返回输出。这是一个在补丁文件中常用的紧凑的，易读的格式。
- **-r** (**--recursive**)
  -r 选项递归比较 **diff** 工具找到的任何子目录。

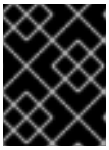
但请注意，在这种特殊情况下，只需要 **-u** 选项。

4. 将补丁保存到文件中：

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

5. 恢复原始 **cello.c** ：

```
$ mv cello.c.orig cello.c
```



重要

您必须保留原始 **cello.c**，因为 RPM 软件包管理器在构建 RPM 软件包时使用原始文件，而不是修改后的文件。如需更多信息，请参阅使用 spec 文件。

其他资源

- **diff (1)** 手册页

3.1.2. 修补示例 C 程序

要对软件应用代码补丁，您可以使用 **patch** 工具。

先决条件

- 在您的系统上安装了补丁程序：

```
# yum install patch
```

- 您从原始源代码创建了补丁。具体步骤请参阅 为 C 程序创建补丁文件。

流程

以下步骤在 cello.c 文件中应用之前创建的 cello.patch 文件。

1. 将补丁文件重定向到 patch 命令：

```
$ patch < cello.patch
patching file cello.c
```

2. 检查 cello.c 的内容现在是否反映了所需的更改：

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

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

验证

1. 构建打了补丁的 cello.c 程序：

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

2. 运行构建的 cello.c 程序：

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

3.2. 创建 LICENSE 文件

建议您使用软件许可证发布软件。

软件许可证文件告知用户他们可以使用源代码做什么和不能做什么。没有源代码许可证意味着您保留此代码的所有权限，任何人都不能从源代码复制、分发或创建衍生工作。

流程

- 使用所需的许可证语句创建 LICENSE 文件：

```
$ vim LICENSE
```

例 3.1. GPLv3 LICENSE 文件文本示例

```
$ 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/.
```

其他资源

- 源代码示例

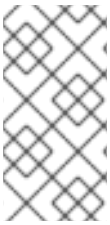
3.3. 为分发创建源代码存档

归档文件是带有 .tar.gz 或 .tgz 后缀的文件。将源代码放入存档中是发布软件以稍后打包为分发的常用方法。

3.3.1. 为 Bash 程序创建源代码存档

bello 项目是 Bash 中的 Hello World 文件。

以下示例仅包含 bello shell 脚本。因此，生成的 tar.gz 存档除了 LICENSE 文件只有一个文件。



注意

补丁 文件不随程序一起在存档中分发。构建 RPM 时，RPM 软件包管理器应用补丁。补丁将与 tar.gz 存档一起放在 ~/rpmbuild/SOURCES/ 目录中。

先决条件

- 假定使用 bello 程序的 0.1 版本。
- 您创建了 LICENSE 文件。具体步骤请参阅创建 LICENSE 文件。

流程

1. 将所有需要的文件移动到单个目录中：

```
$ mkdir bello-0.1

$ mv ~/bello bello-0.1/

$ mv LICENSE bello-0.1/
```

2. 为分发创建存档：

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

3. 将创建的存档移到 ~/rpmbuild/SOURCES/ 目录中，这是 rpmbuild 命令存储用于构建软件包的文件的默认目录：

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

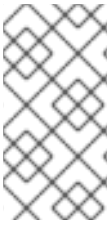
其他资源

- 使用 bash 编写的 hello World

3.3.2. 为示例 Python 程序创建源代码存档

pello 项目是 Python 中的 Hello World 文件。

以下示例仅包含 pello.py 程序。因此，生成的 tar.gz 存档除了 LICENSE 文件只有一个文件。



注意

补丁 文件不随程序一起在存档中分发。构建 RPM 时，RPM 软件包管理器应用补丁。补丁将与 tar.gz 存档一起放在 ~/rpmbuild/SOURCES/ 目录中。

先决条件

- 假定使用 pello 程序的 0.1.1 版本。
- 您创建了 LICENSE 文件。具体步骤请参阅创建 LICENSE 文件。

流程

1. 将所有需要的文件移动到单个目录中：

```
$ mkdir pello-0.1.1

$ mv pello.py pello-0.1.1/

$ mv LICENSE pello-0.1.1/
```

2. 为分发创建存档：

```
$ tar -cvzf pello-0.1.1.tar.gz pello-0.1.1
pello-0.1.1/
pello-0.1.1/LICENSE
pello-0.1.1/pello.py
```

3. 将创建的存档移到 ~/rpmbuild/SOURCES/ 目录中，这是 rpmbuild 命令存储用于构建软件包的文件的默认目录：

```
$ mv pello-0.1.1.tar.gz ~/rpmbuild/SOURCES/
```

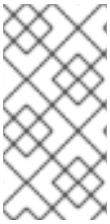
其他资源

- 使用 Python 编写 hello World

3.3.3. 为示例 C 程序创建源代码存档

cello 项目是 C 中的 Hello World 文件。

以下示例仅包含 cello.c 和 Makefile 文件。因此，生成的 tar.gz 存档除了 LICENSE 文件有两个文件。



注意

补丁 文件不随程序一起在存档中分发。构建 RPM 时，RPM 软件包管理器应用补丁。补丁将与 tar.gz 存档一起放在 ~/rpmbuild/SOURCES/ 目录中。

先决条件

- 假定使用 cello 程序的 1.0 版本。
- 您创建了 LICENSE 文件。具体步骤请参阅创建 LICENSE 文件。

流程

1. 将所有需要的文件移动到单个目录中：

```
$ mkdir cello-1.0

$ mv cello.c cello-1.0/

$ mv Makefile cello-1.0/

$ mv LICENSE cello-1.0/
```

2. 为分发创建存档：

```
$ 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
```

3. 将创建的存档移到 ~/rpmbuild/SOURCES/ 目录中，这是 rpmbuild 命令存储用于构建软件包的文件的默认目录：

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

其他资源

- 使用 C 语言编写的 hello World

第 4 章 打包软件

在以下部分中，了解使用 RPM 软件包管理器的打包过程的基础知识。

4.1. 设置 RPM 打包工作区

要构建 RPM 软件包，您必须首先创建一个特殊的工作区，其中包含用于不同打包目的的目录。

4.1.1. 配置 RPM 打包工作区

要配置 RPM 打包工作区，您可以使用 rpmdev-setuptree 程序设置目录布局。

先决条件

- 已安装 rpmdevtools 软件包，它为打包 RPM 提供工具：

```
# yum install rpmdevtools
```

流程

- 运行 rpmdev-setuptree 程序：

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

其他资源

- RPM 打包工作区目录

4.1.2. RPM 打包工作区目录

以下是使用 rpmdev-setuptree 工具创建的 RPM 打包工作区目录：

表 4.1. RPM 打包工作区目录

目录	目的
BUILD	包含从 **SOURCES** 目录中的源文件编译的构建工件。
RPMS	二进制 RPM 在用于不同架构的子目录中的 **RPMS** 目录下创建。例如，在 **x86_64** 或 **noarch** 子目录中。
源	包含压缩的源代码存档和补丁。然后，**rpmbuild** 命令会在这个目录中搜索这些存档和补丁。
SPECS	包含由打包程序创建的 **spec** 文件。然后，使用这些文件来构建软件包。
SRPMS	当您使用 **rpmbuild** 命令构建 SRPM 而不是二进制 RPM 时，会在该目录下创建生成的 SRPM。

4.2. 关于 SPEC 文件

spec 文件是一个文件，其中包含 rpmbuild 实用程序用来构建 RPM 软件包的指令。此文件通过在一系列部分中定义指令，为构建系统提供必要信息。这些部分在 spec 文件的 *Preamble* 和 *Body* 部分中定义：

- *Preamble* 部分包含一系列在 *Body* 部分中使用的元数据项。
- *Body* 部分代表指令的主要部分。

4.2.1. Preamble 项

以下是您可以在 RPM spec 文件的 *Preamble* 部分中使用的一些指令。

表 4.2. *Preamble* 部分指令

指令	定义
名称	必须与 **spec** 文件名匹配的软件包的基本名称。

指令	定义
版本	软件的上游版本。
Release	发布软件包版本的次数。 将初始值设为 **1%{?dist}**，并随着软件包的每个新版本增加。当软件的一个新版本 **构建时**，将重置为 **1**。
概述	软件包的一个简短摘要。
许可证	被打包的软件许可证。 如何在 **spec** 文件中标记 **License** 的具体格式因您遵循的基于 RPM 的 Linux 发行版准则而不同，例如 GPLv3+。
URL	有关软件的更多信息的完整 URL，例如，用于打包软件的上游项目网站。
源	未修补上游源代码的压缩存档的路径或 URL。此链接必须指向该存档的可访问且可靠的存储，例如上游页面，而不是打包程序的本地存储。 您可以在指令名称的末尾应用 **Source** 指令或不带数字。如果未指定数字，则会在内部为条目分配数字。您还可以显式提供数字，例如 **Source** 0、**Source1**、 **Source** 2、**Source3** 等。
Patch	应用到源代码的第一个补丁名称（如有必要）。 您可以在指令名称末尾应用 **Patch** 指令或不带数字。如果未指定数字，则会在内部为条目分配数字。您还可以明确给出数字，例如 **Patch0**,**Patch1**,**Patch2**,**Patch3** 等。 您可以使用 %patch0、**%patch1**、 **%patch 2** 宏等单独应用补丁。宏在 RPM **spec** 文件的 *Body* 部分中的 **%prep** 指令中应用。或者，您可以使用 **%autopatch** 宏，按照 **spec** 文件中给出的顺序自动应用所有补丁。
BuildArch	将构建软件的架构。 如果软件不是独立于架构，例如，如果您完全使用解释编程语言编写软件，请将值设为 **BuildArch: noarch**。如果没有设置这个值，软件会自动继承构建它的机器的架构，如 **x86_64**。
BuildRequires	使用编译语言构建程序所需的以逗号分隔的软件包列表。**BuildRequires** 可以有多个条目，每个条目都在 SPEC 文件中的独立的行中。
Requires	安装之后，软件需要以逗号或空格分开的软件包列表。**Requires** 可以有多个条目，每个条目都在 **spec** 文件中自己的行中。
ExcludeArch	如果某一软件无法在特定的处理器架构上运行，您可以在 **ExcludeArch** 指令中排除此架构。

指令	定义
Conflicts	逗号或空格分开的软件包列表，不能在系统中安装这些软件包，以便您的软件在安装时可以正常工作。可能存在多个 **Conflicts** 条目，每个条目都在 **spec** 文件中自己的行中。
Obsoletes	**Obsoletes** 指令根据以下因素更改更新工作的方式： ● 如果您直接在命令行中使用 **rpm** 命令，它会删除与正在安装的软件包过时的所有软件包，或者更新是由更新或依赖项解析器执行的。 ● 如果您使用更新或依赖项解析器(YUM)，包含匹配 **Obsoletes:** 的软件包会添加为更新并替换匹配的软件包。
Provides	如果您在软件包中添加 **Provides** 指令，则这个软件包可以通过名称以外的依赖项引用。

Name、Version 和 Release (NVR)指令以 name-version-release 格式包含 RPM 软件包的文件名。

您可以使用 rpm 命令查询 RPM 数据库来显示特定软件包的 NVR 信息，例如：

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

在这里，bash 是软件包名称，4.4.19 是版本，7el8 是发行版本。x86_64 标记是软件包架构。与 NVR 不同，架构标记不是直接控制 RPM 软件包器，而是由 rpmbuild 构建环境定义。这种情况的例外是独立于架构的 noarch 软件包。

4.2.2. 正文项

以下是 RPM spec 文件的 *Body* 部分中使用的项目。

表 4.3. Body 部分项

指令	定义
%description	RPM 中打包的软件的完整描述。此描述可跨越多行，并且可以分为几个段落。
%prep	用于为构建准备软件的命令或一系列命令，例如，用于在 **Source** 指令中解压缩存档。**%prep** 指令可以包含 shell 脚本。

指令	定义
%build	将软件构建到机器代码（用于编译的语言）或字节代码（用于某些解释语言）的命令或一系列命令。
%install	在软件构建后，**rpmbuild** 实用程序将使用的命令或一系列命令将软件安装到 **BUILDROOT** 目录中。这些命令将所需的构建工件从 **%_builddir** 目录（其中发生构建）复制到包含要打包文件的目录结构的 **%buildroot** 目录。这包括将文件从 **~/rpmbuild/BUILD** 复制到 **~/rpmbuild/BUILDROOT**，并在 **~/rpmbuild/BUILDROOT** 中创建必要的目录。 **%install** 目录是一个空的 **chroot** 基础目录，类似于最终用户 **的根目录**。您可以在此处创建包含安装文件的目录。要创建这样的目录，您可以使用 RPM 宏，而无需硬编码路径。 请注意，**%install** 仅在创建软件包时运行，而不在安装该软件包时运行。如需更多信息，请参阅使用 spec 文件。
%check	用于测试软件的命令或一系列命令，如单元测试。
%files	由 RPM 软件包提供的文件列表，安装用户系统及其完整路径位置。 在构建期间，如果 **%buildroot** 目录中没有列出 **的文件**，您将收到有关可能的未打包文件的警告。 在 **%files** 部分中，您可以使用内置宏来指示各种文件的角色。这可用于使用 **rpm** 命令查询软件包文件清单元数据。例如，要指示 **LICENSE** 文件是一个软件许可证文件，请使用 **%license** 宏。
%changelog	在不同 **Version** 或 **Release** 构建之间软件包发生的更改记录。这些更改包括软件包的每个 Version-Release 的 date-stamped 条目列表。这些条目日志打包更改，而不是软件更改，例如，在 **%build** 部分中添加补丁或更改构建流程。

4.2.3. 高级 items

spec 文件可以包含高级项目，如 Scriptlets 或 Triggers。

Scriptlets 和 Triggers 在安装过程中在最终用户系统的不同点上生效，而不是构建过程。

4.3. BUILDROOTS

在 RPM 打包上下文中，buildroot 是 chroot 环境。构建工件通过使用与最终用户系统中将来层次结构相同的文件系统层次结构放在此，buildroot 充当根目录。构建工件的放置必须遵循最终用户系统的文件系统层次结构标准。

buildroot 中的文件稍后放入 cpio 存档，后者成为 RPM 的主要部分。当在最终用户的系统中安装 RPM 时，这些文件将提取到 root 目录中，保留正确的层次结构。

注意

rpmbuild 程序有自己的默认值。覆盖这些默认值可能会导致某些问题。因此，请避免定义您自己的 buildroot 宏值。改为使用默认的 %{buildroot} 宏。

4.4. RPM 宏

rpm 宏 是一种直接文本替换，在使用特定内置功能时，可以根据声明的可选评估来有条件地分配。因此，RPM 可以为您执行文本替换。

例如，您可以在 %{version} 宏中定义打包软件的 *Version* 一次，并在 spec 文件中使用此宏。每次发生时都会自动替换为您在宏中定义的 *Version*。

注意

如果您看到不熟悉的宏，您可以使用以下命令评估它：

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

例如，要评估 %{_bindir} 和 %{_libexecdir} 宏，请输入：

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

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

其他资源

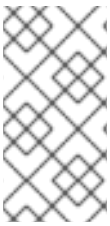
- 有关宏的更多内容

4.5. 使用 SPEC 文件

要打包新软件，您必须创建一个 spec 文件。您可以通过以下任一方法创建 spec 文件：

- 从头开始手动编写新的 spec 文件。

- 使用 rpmdev-newspec 工具。这个工具会创建一个未填充的 spec 文件，其中会填写必要的指令和字段。



注意

某些以编程为导向的文本编辑器使用自己的 spec 模板预先填充新的 spec 文件。rpmdev-newspec 实用程序提供了一个与编辑器无关的方法。

4.5.1. 为 Bash、C 和 Python 程序创建新规格文件

您可以使用 rpmdev-newspec 程序为 Hello World! 程序的三个实现创建一个 spec 文件。

先决条件

- 以下 Hello World! 程序实现被放入 ~/rpmbuild/SOURCES 目录中：
  - bello-0.1.tar.gz
  - pello-0.1.2.tar.gz
  - cello-1.0.tar.gz (cello-output-first-patch.patch)

流程

1. 进入 ~/rpmbuild/SPECS 目录：

```
$ cd ~/rpmbuild/SPECS
```

2. 为 Hello World! 程序的三个实现创建一个 spec 文件：

```
$ 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.
```

~/rpmbuild/SPECS/ 目录现在包含三个名为 bello. spec、cello.spec 和 pello.spec 的 spec 文件。

3. 检查创建的文件。

文件中的指令代表了 关于 spec 文件 中描述的指令。在以下部分中，您将在 rpmdev-newspec 的输出文件中填充特定的部分。

4.5.2. 修改原始 spec 文件

rpmdev-newspec 实用程序生成的原始输出 spec 文件代表一个模板，您必须修改该模板，以提供 rpmbuild 实用程序的必要说明。然后 rpmbuild 使用这些指令来构建 RPM 软件包。

先决条件

- 未填充的 ~/rpmbuild/SPECS/<name>. spec spec 文件是使用 rpmdev-newspec 实用程序创建的。如需更多信息，请参阅 为 Bash、C 和 Python 程序创建新 spec 文件。

流程

1. 打开 rpmdev-newspec 程序提供的 ~/rpmbuild/SPECS/<name>.spec 文件。

2. 填充 spec 文件 *Preamble* 部分的以下指令：

名称

name 已指定为 rpmdev-newspec 的参数。

版本

将 Version 设置为与源代码的上游版本匹配。

Release

Release 会自动设置为 1%{?dist}，它最初是 1。

概述

输入软件包的单行说明。

许可证

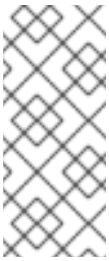
输入与源代码关联的软件许可证。

URL

输入上游软件网站的 URL。为实现一致性，请使用 %{name} RPM 宏变量，并使用 https://example.com/%{name} 格式。

源

输入上游软件源代码的 URL。直接链接到被打包的软件版本。



注意

本文档中的示例 URL 包括可能会在以后更改的硬编码值。同样，发行版本也可以更改。要简化这些潜在的更改，请使用 %{name} 和 %{version} 宏。通过使用这些宏，您只需要更新 spec 文件中的一个字段。

BuildRequires

指定软件包的 build-time 依赖项。

Requires

指定软件包的运行时依赖项。

BuildArch

指定软件架构。

3.

填充 spec 文件 *Body* 部分中的以下指令：您可以将这些指令视为部分标题，因为这些指令可以定义多行、多结构或脚本化任务。

%description

输入软件的完整描述。

%prep

输入命令或一系列命令来准备软件以进行构建。

%build

输入用于构建软件的命令或一系列命令。

%install

输入命令或一系列命令，以指示 rpmbuild 命令如何将软件安装到 BUILDROOT 目录中。

%files

指定要在您的系统上安装的 RPM 软件包提供的文件列表。

%changelog

输入软件包的每个 Version-Release 的 datetamped 条目列表。

从 %changelog 部分的第一行开始，带有星号(*)字符，后跟 Day-of-Week Month Day Year Name Surname <email> - Version-Release。

对于实际更改条目，请遵循这些规则：

- 每个更改条目都可以包含多个项目，每个代表一个改变。
- 每个项目在新行中开始。
- 每个项目以连字符(-)字符开头。

您现在已为所需程序编写了整个 spec 文件。

其他资源

- Preamble 项
- 正文项
- Bash 程序的 spec 文件示例
- Python 程序的 spec 文件示例
- C 程序的 spec 文件示例
- 构建 RPM

4.5.3. Bash 程序的 spec 文件示例

您可以将以下 spec 文件示例用于 bash 中编写的 bello 程序供您参考。

使用 bash 编写的 bello 程序的 spec 文件示例

```
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
```

- BuildRequires 指令指定软件包的 build-time 依赖项已被删除，因为没有可用于 bello 的构建步骤。Bash 是原始解释编程语言，文件仅安装到其系统上的位置。
- Requires 指令指定软件包的运行时依赖项，它只包括 bash，因为 bello 脚本只需要 bash shell 环境才能执行。
- %build 部分指定如何构建软件为空，因为不需要构建 bash 脚本。

注意

要安装 bello，您必须创建目标目录并在其中安装可执行的 bash 脚本文件。因此，您可以在 %install 部分中使用 install 命令。您可以使用 RPM 宏来执行此操作，而无需硬编码路径。

其他资源

- 什么是源代码

4.5.4. Python 程序的 spec 文件示例

您可以对使用 Python 编程语言编写的 pello 程序使用以下示例 spec 文件供您参考。

使用 Python 编写的 pello 程序的 spec 文件示例

```
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
```

- Requires 指令指定软件包的运行时依赖项，其中包括两个软件包：
  - 在运行时执行字节代码所需的 python 软件包。
  - 执行小入口点脚本所需的 bash 软件包。
- BuildRequires 指令指定软件包的 build-time 依赖项，它只包括 python 软件包。pello 程序需要 python 执行字节型构建流程。
- %build 部分指定如何构建软件，创建脚本的字节版本。请注意，在实际打包中，通常会根据所使用的发行版自动执行。
- %install 部分与这个事实对应，您必须将字节文件安装到系统上的库目录中，以便可以访问它。

在 spec 文件中，创建打包程序脚本的示例显示了 spec 文件本身可以脚本化。这个打包程序脚本使用 此处文档 执行 Python 字节编译的代码。

其他资源

- 什么是源代码

4.5.5. C 程序的 spec 文件示例

您可以将以下示例 spec 文件用于使用 C 编程语言编写的 cello 程序供您参考。

使用 C 语言编写的 cello 程序的 spec 文件示例

```
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
```

- BuildRequires 指令指定软件包的 build-time 依赖项，其中包括执行编译构建过程所需的以下软件包：
  - gcc
  - make
- 本例中省略了该软件包的运行时依赖项 Requires 指令。所有运行时要求都由 rpmbuild 进行处理，而 cello 程序不需要核心 C 标准库之外的任何内容。
- %build 部分反映了在这个示例中编写了 cello 程序的 Makefile 文件的事实。因此，您可以使用 GNU make 命令。但是，您必须删除对 %configure 的调用，因为您没有提供配置脚本。

您可以使用 %make_install 宏安装 cello 程序。这是因为 cello 程序的 Makefile 文件可用。

其他资源

- 什么是源代码

4.6. 构建 RPM

您可以使用 rpmbuild 命令构建 RPM 软件包。使用此命令时，应该有一个特定的目录和文件结构，它与 rpmdev-setuptree 程序设置的结构相同。

不同的用例和所需结果需要不同的参数组合到 rpmbuild 命令。以下是主要用例：

- 构建源 RPM.
- 构建二进制 RPM：
  - 从源 RPM 重建二进制 RPM。
  - 从 spec 文件构建二进制 RPM。

4.6.1. 构建源 RPM

构建源 RPM (SRPM)具有以下优点：

- 您可以保留部署到环境中的 RPM 文件的特定 Name-Version-Release 的确切源。这包括确切的 spec 文件、源代码以及所有相关补丁。这可用于跟踪和调试目的。
- 您可以在不同的硬件平台或构架中构建二进制 RPM。

先决条件

- 您已在系统中安装了 rpmbuild 工具：

```
# yum install rpm-build
```

- 以下 Hello World! 实现放置在 ~/rpmbuild/SOURCES/ 目录中：
  - bello-0.1.tar.gz
  - pello-0.1.2.tar.gz
  - cello-1.0.tar.gz (cello-output-first-patch.patch)
- 您要软件包的程序的 spec 文件已存在。

流程

1. 导航到 ~/rpmbuild/SPECS/ 指令，其中包含创建的 spec 文件：

```
$ cd ~/rpmbuild/SPECS/
```

2. 使用指定 spec 文件输入 rpmbuild 命令构建源 RPM：

```
$ rpmbuild -bs <specfile>
```

-bs 选项代表 *构建源*。

例如，要为 bello、pello 和 cello 程序构建源 RPM，请输入：

```
$ 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
```

验证步骤

- 验证 rpmbuild/SRPMS 目录是否包含生成的源 RPM。该目录是 rpmbuild 所期望的结构的一部分。

其他资源

- 使用 spec 文件
- 为 Bash、C 和 Python 程序创建新规格文件
- 修改原始 spec 文件

4.6.2. 从源 RPM 重建二进制 RPM

要从源 RPM (SRPM)重建二进制 RPM，请使用 rpmbuild 命令和 --rebuild 选项。

创建二进制 RPM 时生成的输出非常详细，这对于调试非常有用。输出因不同的示例而异，并对应于其 spec 文件。

生成的二进制 RPM 位于 ~/rpmbuild/RPMS/YOURARCH 目录中，其中 YOURARCH 是您的架构，或者在 ~/rpmbuild/RPMS/noarch/ 目录中（如果该软件包不特定于架构）。

先决条件

- 您已在系统中安装了 rpmbuild 工具：

```
# yum install rpm-build
```

流程

1. 进入 ~/rpmbuild/SRPMS/ 指令，其中包含源 RPM：

```
$ cd ~/rpmbuild/SRPMS/
```

2. 从源 RPM 重建二进制 RPM：

```
$ rpmbuild --rebuild <srpm>
```

将 *srpm* 替换为源 RPM 文件的名称。

例如，要从 SRPMs 重建 bello、pello 和 cello，请输入：

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

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

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

注意

调用 rpmbuild --rebuild 涉及以下进程：

- 将 SRPM (spec 文件和源代码)的内容安装到 ~/rpmbuild/ 目录中。
- 使用安装的内容构建一个 RPM。
- 删除 spec 文件和源代码。

您可以在构建以下任一方法后保留 spec 文件和源代码：

- 在构建 RPM 时，请使用 rpmbuild 命令和 --recompile 选项，而不是 --rebuild 选项。
- 为 bello、pello 和 cello 安装 SRPMs：

```
$ 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%]
```

4.6.3. 从 spec 文件构建二进制 RPM

要从其 spec 文件构建二进制 RPM，请使用 rpmbuild 命令和 -bb 选项。

先决条件

- 您已在系统中安装了 rpmbuild 工具：

```
# yum install rpm-build
```

流程

1. 导航到 ~/rpmbuild/SPECS/ 指令，其中包含 spec 文件：

```
$ cd ~/rpmbuild/SPECS/
```

2. 从其 规格 构建二进制 RPM：

```
$ rpmbuild -bb <spec_file>
```

例如，要从其 spec 文件构建 bello、pello 和 cello 二进制 RPM，请输入：

```
$ rpmbuild -bb bello.spec
$ rpmbuild -bb pello.spec
$ rpmbuild -bb cello.spec
```

4.7. 检查 RPM 健全性

创建软件包后，您可能希望检查软件包的质量。检查软件包质量的主要工具是 rpmlint。

使用 rpmlint 工具，您可以执行以下操作：

- 提高 RPM 可维护性。
- 通过对 RPM 进行静态分析来启用完整性检查。
- 通过对 RPM 进行静态分析来启用错误检查。

您可以使用 rpmlint 检查二进制 RPM、源 RPM (SRPMs)和 spec 文件。因此，这个工具对打包的所有阶段都很有用。

请注意，rpmlint 有严格的准则。因此，有时可以接受跳过其某些错误和警告，如以下部分所示。

注意

在以下部分中描述的示例中，rpmlint 会不带任何选项运行，这会产生一个非详细的输出。有关每个错误或警告的详细说明，请运行 rpmlint -i。

4.7.1. 检查示例 Bash 程序是否有健全性

在以下部分中，调查在检查 bello spec 文件示例和 bello 二进制 RPM 时可能出现的警告和错误。

4.7.1.1. 检查 bello spec 文件是否有健全性

检查以下示例的输出，了解如何检查 bello spec 文件是否有 sanity。

在 bello spec 文件中运行 rpmlint 命令的输出

```
$ 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.
```

对于 bello.spec，只有一个 invalid-url Source0 警告。此警告意味着 Source0 指令中列出的 URL 不可访问。这是正常的，因为指定的 example.com URL 不存在。假设此 URL 在以后有效，您可以忽略此警告。

在 bello SRPM 上运行 rpmlint 命令的输出

```
$ 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.
```

对于 bello SRPM，有一个新的 invalid-url URL 警告，表示 URL 指令中指定的 URL 不可访问。假设此 URL 在以后有效，您可以忽略此警告。

4.7.1.2. 检查 bello 二进制 RPM 的健全性

在检查二进制 RPM 时，rpmlint 命令会检查以下项目：

- Documentation
- man page
- 致地使用文件系统层次结构标准

检查以下示例的输出，来了解如何检查 bello 二进制 RPM 的健全性。

在 bello 二进制 RPM 上运行 rpmlint 命令的输出

```
$ 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.
```

no-documentation 和 no-manual-page-for-binary 警告意味着 RPM 没有文档或手册页，因为您没有提供任何一个。除了输出警告外，RPM 通过了 rpmlint 检查。

4.7.2. 检查示例 Python 程序是否有健全性

在以下部分中，调查在 pello spec 文件和 pello 二进制 RPM 示例中检查 RPM 健全时可能会出现的警告和错误。

4.7.2.1. 检查 pello spec 文件是否有健全性

检查以下示例的输出，了解如何检查 pello spec 文件是否有 sanity。

在 pello spec 文件上运行 rpmlint 命令的输出

```
$ 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.
```

- invalid-url Source0 警告表示 Source0 指令中列出的 URL 不可访问。这是正常的，因为指定的 example.com URL 不存在。假设此 URL 在以后有效，您可以忽略此警告。
- hardcoded-library-path 错误建议使用 %{_libdir} 宏而不是硬编码库路径。在本例中，可以安全地忽略这些错误。但是，对于要进入生产环境的软件包，请仔细检查所有错误。

在 SRPM for pello 上运行 rpmlint 命令的输出

```
$ 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.
```

invalid-url URL 错误表示 URL 指令中提到的 URL 不可访问。假设此 URL 在以后有效，您可以忽略此警告。

4.7.2.2. 检查 pello 二进制 RPM 的健全性

在检查二进制 RPM 时，rpmlint 命令会检查以下项目：

- Documentation
- man page
- 致地使用文件系统层次结构标准

检查以下示例的输出，来了解如何检查 pello 二进制 RPM 的健全性。

在 pello 二进制 RPM 上运行 rpmlint 命令的输出

```
$ 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.
```

- no-documentation 和 no-manual-page-for-binary 警告意味着 RPM 没有文档或 man page，因为您没有提供任何文档。
- only-non-binary-in-usr-lib 警告意味着您在 /usr/lib/ 目录中只提供了非二进制工件。此目录通常用于共享目标文件，这些文件是二进制文件。因此，rpmlint 期望 /usr/lib/ 中至少有一个或者多个文件是二进制文件。

  这是 rpmlint 检查的一个示例，它是否符合文件系统层次结构标准。要确保正确放置文件，

请使用 RPM 宏。在本例中，可以安全地忽略这个警告。

- non-executable-script 错误表示 /usr/lib/pello/pello.py 文件没有执行权限。rpmlint 工具预期文件可以执行，因为文件包含 shebang (#!)。在本例中，您可以保留此文件而不具有执行权限，并忽略此错误。

除了输出警告和错误外，RPM 通过了 rpmlint 检查。

4.7.3. 检查示例 C 程序是否有健全性

在以下部分中，调查在 cello spec 文件和 cello 二进制 RPM 示例中检查 RPM 健全时可能会出现的警告和错误。

4.7.3.1. 检查 cello spec 文件是否有健全性

检查以下示例的输出，了解如何检查 cello spec 文件是否有 sanity。

在 cello spec 文件中运行 rpmlint 命令的输出

```
$ 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.
```

对于 cello.spec，只有一个 invalid-url Source0 警告。此警告意味着 Source0 指令中列出的 URL 不可访问。这是预期的，因为指定的 example.com URL 不存在。假设此 URL 在以后有效，您可以忽略此警告。

在 cello SRPM 上运行 rpmlint 命令的输出

```
$ 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.
```

对于 cello SRPM，有一个新的 invalid-url URL 警告。此警告意味着 URL 指令中指定的 URL 不可访问。假设此 URL 在以后有效，您可以忽略此警告。

4.7.3.2. 检查 cello 二进制 RPM 的健全性

在检查二进制 RPM 时，rpmlint 命令会检查以下项目：

- Documentation
- man page
- 致地使用文件系统层次结构标准

检查以下示例的输出，来了解如何检查 cello 二进制 RPM 的健全性。

在 cello 二进制 RPM 上运行 rpmlint 命令的输出

```
$ 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.
```

no-documentation 和 no-manual-page-for-binary 警告意味着 RPM 没有文档或 man page，因为您没有提供任何文档。

除了输出警告外，RPM 通过了 rpmlint 检查。

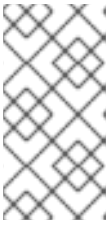
4.8. 将 RPM 活动记录到 SYSLOG

您可以使用系统日志协议(syslog)记录任何 RPM 活动或事务。

先决条件

- syslog 插件已安装在系统上：

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



注意

syslog 消息的默认位置是 /var/log/messages 文件。但是，您可以将 syslog 配置为使用另一个位置来存储消息。

流程

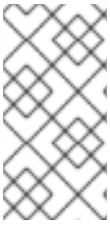
1. 打开您配置为存储 syslog 信息的文件。

   或者，如果您使用默认 syslog 配置，请打开 /var/log/messages 文件。

2. 搜索包括 [RPM] 字符串的新行。

4.9. 提取 RPM 内容

在某些情况下，例如，如果 RPM 所需的软件包被损坏，您可能需要提取软件包的内容。在这种情况下，如果 RPM 安装仍正常工作，您可以使用 rpm2archive 实用程序将 .rpm 文件转换为 tar 存档以使用软件包的内容。



注意

如果 RPM 安装严重损坏，您可以使用 rpm2cpio 工具将 RPM 软件包文件转换为 cpio 存档。

流程

- **将 RPM 文件转换为 tar 归档：**

```
$ rpm2archive <filename>.rpm
```

生成的文件具有 .tgz 后缀。例如，要从 bash 软件包创建存档，请输入：

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

第 5 章 高级主题

本节涵盖超出入门教程范围但对真实 RPM 打包很有用的主题。

5.1. 签名 RPM 软件包

您可以签署 RPM 软件包，以确保没有第三方可以更改其内容。要添加额外的安全层，请在下载软件包时使用 HTTPS 协议。

您可以使用 rpm-sign 软件包提供的 --addsign 选项为软件包签名。

先决条件

- 您已创建了 GNU Privacy Guard (GPG)密钥，如 创建 GPG 密钥 中所述。

5.1.1. 创建 GPG 密钥

使用以下步骤创建签名软件包所需的 GNU Privacy Guard (GPG)密钥。

流程

1. 生成 GPG 密钥对：

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

2. 检查生成的密钥对：

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

3. 导出公钥：

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

将 <Key_name> 替换为您选择的实际密钥名称。

4. 将导出的公钥导入到 RPM 数据库中：

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

5.1.2. 配置 RPM 为软件包签名

要能够签署 RPM 软件包，您需要指定 %_gpg_name RPM 宏。

以下流程描述了如何配置 RPM 以签名软件包。

流程

- 在 $HOME/.rpmmacros 文件中定义 %_gpg_name 宏，如下所示：

```
%_gpg_name Key ID
```

使用您要用来签署软件包的 GNU Privacy Guard (GPG)密钥 ID 替换 *Key ID*。有效的 GPG 密钥 ID 值是创建密钥的用户的完整名称或电子邮件地址。

5.1.3. 在 RPM 软件包中添加签名

最常见的情况是在没有签名的情况下构建软件包。签名仅在软件包发布前添加。

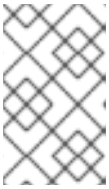
要在 RPM 软件包中添加签名，请使用 rpm-sign 软件包提供的 --addsign 选项。

流程

- 在软件包中添加签名：

```
$ rpm --addsign package-name.rpm
```

使用您要签名的 RPM 软件包的名称替换 *package-name*。



注意

您必须输入密码来解锁签名的 secret 密钥。

5.2. 有关宏的更多内容

本节介绍所选内置 RPM Macros。有关此类宏的详细列表，请参阅 RPM 文档。

5.2.1. 定义您自己的宏

下面的部分论述了如何创建自定义宏。

流程

- 在 RPM spec 文件中包括以下行：

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

删除 <body> 周围的空白。名称可以是字母数字字符，字符 _，长度必须至少为 3 个字符。包含 (opts) 字段是可选的：

- Simple 宏不包含 (opts) 字段。在这种情况下，只执行递归宏扩展。
- Parametrized 宏包含 (opts) 字段。在宏调用开始时传递括号之间的 opts 字符串可得到 argc/argv 处理的 getopt(3)。

注意

旧的 RPM spec 文件使用 %define <name> <body& gt; 宏模式。%define 和 %global 宏之间的差异如下：

- %define 是本地范围的。它适用于 spec 文件的特定部分。%define 宏的主体部分在使用时会被扩展。
- %global 有全局范围。它适用于整个 spec 文件。在定义时扩展 %global 宏的正文。

重要

宏会被评估，即使它们被注释掉，或者宏的名称被赋予到 spec 文件的 %changelog 部分。要注释掉宏，请使用 %%。例如 %%global.

其他资源

- 宏语法

5.2.2. 使用 %setup 宏

这部分论述了如何使用 %setup 宏的不同变体构建带有源代码 tarball 的软件包。请注意，宏变体可以合并。rpmbuild 输出说明了 %setup 宏的标准行为。在每个阶段开始时，宏输出 Executing(%...)，如下例所示。

例 5.1. %setup 宏输出示例

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

shell 输出启用了 set -x。要查看 /var/tmp/rpm-tmp.DhddsG 的内容，请使用 --debug 选项，因为 rpmbuild 在成功构建后删除临时文件。这将显示环境变量的设置，后跟：

```
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 .
```

%setup 宏：

- 确保我们在正确的目录中工作。
- 删除之前构建的恢复。
- 解包源 tarball。
- 设置一些默认特权。

5.2.2.1. 使用 %setup -q 宏

-q 选项限制 %setup 宏的详细程度。仅执行 tar -xof 而不是 tar -xvvof。使用这个选项作为第一个选项。

5.2.2.2. 使用 %setup -n 宏

-n 选项指定已展开 tarball 中的目录名称。

当来自扩展 tarball 的目录与预期内容不同时，会使用这个情况(%{name}-%{version})，这可能会导致 %setup 宏的错误。

例如，如果软件包名称是 cello，但源代码在 hello-1.0.tgz 中存档，并且包含 hello/ 目录，则 spec 文件内容需要如下：

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

5.2.2.3. 使用 %setup -c 宏

如果源代码 tarball 不包含任何子目录，并在解压缩后的文件会填充当前目录，则使用 -c 选项。

然后，-c 选项会在归档扩展中创建目录和步骤，如下所示：

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

归档扩展后不会更改该目录。

5.2.2.4. 使用 %setup -D 和 %setup -T 宏

-D 选项会禁用删除源代码目录，在使用 %setup 宏时特别有用。使用 -D 选项时，不会使用以下行：

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

-T 选项通过从脚本中删除以下行来禁用源代码 tarball 的扩展：

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

5.2.2.5. 使用 %setup -a 和 %setup -b 宏

-a 和 -b 选项可以扩展特定的源：

- -b 选项代表 before。这个选项在进入工作目录前扩展特定的源。
- -a 选项代表 after。这个选项在进入后扩展这些源。它们的参数是 spec 文件中的源号。

在以下示例中，cello-1.0.tar.gz 存档包含一个空 examples 目录。示例以单独的 example.tar.gz tarball 中提供，它们被扩展到同一名称的目录中。在这种情况下，如果您在进入工作目录后想扩展 Source1，请使用 -a 1 ：

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

```
%prep
%setup -a 1
```

在以下示例中，在单独的 cello-1.0-examples.tar.gz tarball 中提供了示例，它扩展至 cello-1.0/examples。在这种情况下，在进入工作目录前，使用 -b 1 扩展 Source1 ：

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

5.2.3. %files 部分中的常见 RPM 宏

下表列出了 spec 文件的 %files 部分中需要的高级 RPM Macros。

表 5.1. %files 部分中的高级 RPM Macros

Macro	定义
%license	**%license** 宏识别列为 **LICENSE** 文件的文件，它将被安装，并被 RPM 标记成这样 。示例：**%license LICENSE**.
%doc	**%doc** 宏识别列为文档的文件，它将被安装，并被 RPM 标记成这样。%doc 宏用于有关打包的软件的文档，以及用于代码示例和各种附带项。如果包含代码示例，则必须小心地从文件中删除可执行模式。示例：**%doc README**
%dir	**%dir** 宏确保路径是由此 RPM 拥有的目录。这一点很重要，因此 RPM 文件清单准确知道在卸载时要清理哪些目录。示例： **%dir %{_libdir}/%{name}**
%config(noreplace)	**%config (noreplace)** 宏可确保以下文件是一个配置文件，因此如果已从原始安装校验和修改了该文件，则其不应在软件包安装或更新软件包时被覆盖（或替换）。如果有更改，则会在升级或安装时使用 **.rpmnew** 创建该文件，以便不修改目标系统上的预先存在的或修改的文件。示例：**%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

5.2.4. 显示内置宏

Red Hat Enterprise Linux 提供多个内置 RPM 宏。

流程

1. 要显示所有内置 RPM 宏，请运行：

```
rpm --showrc
```

注意

输出很长。要缩小结果范围，请在 grep 命令中使用上述命令。

2. 要查找有关您系统 RPM 版本 RPM 宏的信息，请运行：

```
rpm -ql rpm
```

注意

RPM 宏是在输出目录结构中标题为 macros 的文件。

5.2.5. RPM 发布宏

不同的发行版根据被打包的软件语言或发布的具体准则，提供不同的推荐 RPM 宏集合。

推荐的 RPM 宏集合通常以 RPM 软件包形式提供，可以使用 yum 软件包管理器安装。

安装后，宏文件可在 /usr/lib/rpm/macros.d/ 目录中找到。

流程

- 要显示原始 RPM 宏定义，请运行：

```
rpm --showrc
```

以上输出显示原始 RPM 宏定义。

- 要确定宏的作用以及在打包 RPM 时如何有帮助，使用宏名称作为其参数运行 rpm --eval 命令：

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

其他资源

- RPM man page

5.2.6. 创建自定义宏

您可以使用自定义宏覆盖 ~/.rpmmacros 文件中的发布宏。您所做的任何更改都会影响您计算机上的每个构建。

警告

不建议在 ~/.rpmmacros 文件中定义任何新宏。其他机器上不会包括此类宏，因为用户可能想要重新构建您的软件包。

流程

- 要覆盖宏，请运行：

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

您可以从上面示例中创建 目录，包括通过 rpmdev-setuptree 实用程序的所有子目录。此宏的值默认为 ~/rpmbuild。

```
%_smp_mflags -l3
```

以上宏通常用于传递 Makefile，如 make %{?_smp_mflags}，并在构建阶段设置多个并发进程。默认情况下，它被设置为 -jX，其中 X 是内核数。如果您更改了内核数量，您可以加快或减慢软件包构建速度或减慢速度。

5.3. EPOCH, SCRIPTLETS 和 TRIGGERS

本节介绍 Epoch、Scriptlets 和 Triggers，它们代表 RMP spec 文件的高级指令。

所有这些指令都影响不仅影响 spec 文件，还影响到安装结果 RPM 的末尾机器。

5.3.1. Epoch 指令

Epoch 指令支持根据版本号定义权重的依赖关系。

如果 RPM spec 文件中没有列出此指令，则不会设置 Epoch 指令。这与常规的理解不同：不设置 Epoch 的结果是 Epoch 为 0。但是，出于解算的目的，yum 工具会将未设置的 Epoch 视为与 Epoch 等于 0 一样。

但是，在 spec 文件中列出 Epoch 通常会省略，因为大多数情况下，在引入 Epoch 值时，在比较软件包版本时会出现预期的 RPM 行为。

> **例 5.2. 使用 Epoch**
>
> 如果您使用 Epoch: 1 和 Version: 1.0 安装 foobar 软件包，且其他人使用 Version: 2.0 打包了 foobar，但没有 Epoch 指令，新版本永远不会被视为更新。原因是，在签发 RPM 软件包版本是首选使用 Epoch 版本而不是传统的 Name-Version-Release marker。

使用 Epoch 比较罕见。但是，Epoch 通常用于解决升级排序问题。在软件版本号方案或带有字母字符的版本中，这个问题可能会出现上游变化的影响，这些字符不能始终根据编码进行可靠地进行比较。

5.3.2. scriptlets 指令

Scriptlets 是一组在安装或删除软件包之前或之后执行的 RPM 指令。

使用 Scriptlets 仅在构建时或启动脚本中无法完成的任务。

存在一组常用 Scriptlet 指令。它们与 spec 文件部分标头类似，如 %build 或 %install。它们由多行代码段定义，这些片段通常写为标准的 POSIX shell 脚本。但是，它们也可以使用其他适用于目标机器分布接受的 RPM 编程语言编写。RPM 文档包括可用语言的详尽列表。

下表包含 Scriptlet 指令，按其执行顺序列出。请注意，包含脚本的软件包会在 %pre 和 %post 指令之间安装，并在 %preun 和 %postun 指令之间卸载。

表 5.2. Scriptlet 指令

指令	定义
%pretrans	Scriptlet 在安装或删除任何软件包之前执行。
%pre	Scriptlet 在目标系统上安装软件包之前执行。
%post	Scriptlet 仅在目标系统上安装软件包后执行。
%preun	在从目标系统卸载软件包前执行的 Scriptlet。
%postun	Scriptlet 在软件包从目标系统卸载后执行。
%posttrans	在事务结束时执行的 Scriptlet。

5.3.3. 关闭 scriptlet 执行

下面的步骤描述了如何使用 rpm 命令和 --no_scriptlet_name_ 选项一起关闭任何 scriptlet 的执行。

流程

- **例如，要关闭 %pretrans scriptlets 的执行，请运行：**

```
# rpm --nopretrans
```

  您还可以使用 -- noscripts 选项，它等同于以下所有：

  - **--nopre**
  - **--nopost**
  - **--nopreun**
  - **--nopostun**

- --nopretrans

- --noposttrans

其他资源

- rpm(8) 手册页.

5.3.4. scriptlets 宏

Scriptlets 指令也适用于 RPM 宏。

以下示例显示了使用 systemd scriptlet 宏，这样可确保 systemd 会收到有关新单元文件的通知。

```
$ 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
```

5.3.5. Triggers 指令

Triggers 是 RPM 指令，可提供在软件包安装和卸载期间交互的方法。

警告

Triggers 可能会在意外执行，例如在更新包含软件包时执行。很难调试 Triggers，因此需要以可靠的方式实施它们，以便在意外执行时不会中断任何操作。因此，红帽建议尽可能减少使用 Triggers。

下面列出了一次软件包升级的顺序以及每个现有 Triggers 的详情：

```
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
```

以上项目位于 /usr/share/doc/rpm-4.*/triggers 文件中。

5.3.6. 在 spec 文件中使用非 shell 脚本

spec 文件中的 -p scriptlet 选项允许用户调用特定的解释器，而不是默认的 shell 脚本解释器(-p /bin/sh)。

下面的步骤描述了如何创建脚本，它会在安装 pello.py 程序后输出信息：

流程

1. 打开 pello.spec 文件。
2. 找到以下行：

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

3. 在上面的行下，插入：

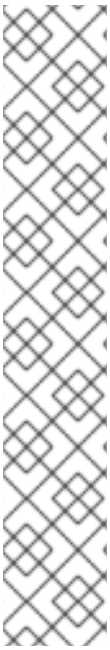
```
%post -p /usr/bin/python3
print("This is {} code".format("python"))
```

4. 按照构建 RPM 中所述构建您的软件包。
5. 安装软件包：

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

6. 安装后检查输出信息：

```
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
```



注意

要使用 Python 3 脚本，请在 spec 文件中 install -m 下包含以下行：

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

要使用 Lua 脚本，在 SPEC 文件中的 install -m 下包含以下行：

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

这样，您可以在 spec 文件中指定任何解释器。

5.4. RPM 条件

RPM 条件可启用 spec 文件的各种部分的条件。

条件包括通常会处理：

- 特定于架构的部分
- 特定于操作系统的部分
- 不同操作系统版本之间的兼容性问题
- 宏的存在和定义

5.4.1. RPM 条件语法

RPM 条件使用以下语法：

如果 *expression* 为 true，则执行一些操作：

```
%if expression
…
%endif
```

如果 *expression*为 true，则执行一些操作，在其他情况下执行另一个操作：

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

5.4.2. %if 条件

以下示例显示了 %if RPM 条件的用法。

例 5.3. 使用 %if 条件来处理 Red Hat Enterprise Linux 8 和其他操作系统间的兼容性

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

这个条件在支持 AS_FUNCTION_DESCRIBE 宏时处理 RHEL 8 和其他操作系统间的兼容性。如果为 RHEL 构建软件包，则会定义 %rhel 宏，并将其扩展到 RHEL 版本。如果它的值是 8，表示软件包是为 RHEL 8 构建的。然后对 AS_FUNCTION_DESCRIBE 的引用（不被 RHEL 8 支持）会从 autoconfig 脚本中删除。

例 5.4. 使用 %if 条件句处理宏定义

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

这个条件处理宏的定义。如果设置了 %milestone 或 %revision 宏，则会重新定义用于定义上游 tarball 名称的 %ruby_archive 宏。

5.4.3. %if 条件的专用变体

%ifarch 条件、%ifnarch 条件和 %ifos 条件是 %if 条件的专用变体。这些变体常被使用，因此它们有自己的宏。

%ifarch 条件

%ifarch 条件用于开始特定于架构的 spec 文件的块。它后接一个或多个架构说明符，各自以逗号或空格分开。

例 5.5. 使用 %ifarch 条件的示例

```
%ifarch i386 sparc
…
%endif
```

%ifarch 和 %endif 之间的 spec 文件的内容仅在 32 位 AMD 和 Intel 架构或基于 Sun SPARC 的系统上进行处理。

%ifnarch 条件

%ifnarch 条件的逻辑与 %ifarch 条件的逻辑相反。

例 5.6. 使用 %ifnarch 条件的示例

```
%ifnarch alpha
…
%endif
```

只有在基于数字 Alpha/AXP 的系统上未执行时，才会处理 %ifnarch 和 %endif 之间的 spec 文件的内容。

%ifos 条件

%ifos 条件用于根据构建的操作系统控制处理。其后可以使用一个或多个操作系统名称。

例 5.7. 使用 %ifos 条件的示例

```
%ifos linux
…
%endif
```

只有在 Linux 系统上完成构建时，才会处理 %ifos 和 %endif 之间的 spec 文件的内容。

5.5. 打包 PYTHON 3 RPM

大多数 Python 项目使用 Setuptools 打包，并在 setup.py 文件中定义软件包信息。有关 Setuptools 打包的详情，请查看 Setuptools 文档。

您还可以将 Python 项目打包到 RPM 软件包中，它与 Setuptools 包装相比有以下优点：

- 有关软件包在其他 RPM 中的依赖软件包（甚至非 Python）
- 加密签名

  通过加密签名，RPM 软件包的内容可以验证、集成并测试操作系统的其余部分。

5.5.1. Python 软件包的 spec 文件描述

spec 文件包含 rpmbuild 实用程序用于构建 RPM 的说明。这些指令包含在不同的部分。spec 文件有两个主要部分来定义：

- Preamble（包含一系列在 Body 中使用的元数据项）
- Body（包含指令的主要部分）

与非 Python RPM SPEC 文件相比，Python 项目的 RPM SPEC 文件有一些特定信息。最值得注意的是，Python 库的任何 RPM 软件包的名称必须始终包含确定版本的前缀，例如： Python 3.6 的 python 3、Python 3.8 的 python38、Python 3.9 的 python39、Python 3.11 的 python3.11 或 Python 3.12 的 python3.12。

其他具体信息显示在 python3-detox 软件包的以下 spec 文件示例 中。有关此类特定描述，请查看示例中的备注。

```
%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

modname 宏包含 Python 项目的名称。在本例中，它是 detox。

2

将 Python 项目打包到 RPM 时，需要将 python3 前缀添加到项目的原始名称。这里的原始名称为 detox，RPM 的名称为 python3-detox。

3

BuildRequires 指定了构建和测试此软件包所需的软件包。在 BuildRequires 中，始终包括提供构建 Python 软件包所需的工具的项目： python36-devel 和 python3-setuptools。需要 python36-rpm-macros 软件包，以便 /usr/bin/python3 解释器指令的文件会自动改为 /usr/bin/python3.6。

4

每个 Python 软件包都需要一些其他软件包才能正常工作。这些软件包还需要在 spec 文件中指定。要指定依赖项，您可以使用 %python_enable_dependency_generator 宏自动使用 setup.py 文件中定义的依赖项。如果软件包的依赖软件包没有使用 Setuptools 指定，请在附加 Requires 指令中指定它们。

5

%py3_build 和 %py3_install 宏会分别运行 setup.py build 和 setup.py install 命令，使用附加参数来指定安装位置、要使用的解释器以及其他详情。

6

检查 部分提供了运行正确 Python 版本的宏。%{__python3} 宏包含 Python 3 解释器的路径，如 /usr/bin/python3。我们建议始终使用宏而不是字面上的路径。

5.5.2. Python 3 RPM 的常见宏

在 spec 文件中，始终使用用于 Python 3 RPM 的 Macros 的 Macros 表中的宏，而不是硬编码其值。

在宏名称中，总是使用 python3 或 python2，而不是未指定版本的 python。在 SPEC 文件的 BuildRequires 部分中，将特定的 Python 3 版本配置为 python36-rpm-macros,python38-rpm-macros,python39-rpm-macros,python3.11-rpm-macros, 或 python3.12-rpm-macros。

表 5.3. Python 3 RPM 宏

Macro	常规定义	描述
%{__python3}	/usr/bin/python3	Python 3 解释器
%{python3_version}	3.6	Python 3 解释器的完整版本。
%{python3_sitelib}	/usr/lib/python3.6/site-packages	安装纯 Python 模块的位置。
%{python3_sitearch}	/usr/lib64/python3.6/site-packages	安装包含特定架构扩展的模块。
%py3_build		使用适合系统软件包的参数运行 **setup.py build** 命令。
%py3_install		使用适合系统软件包的参数运行 **setup.py install** 命令。

5.5.3. 自动为 Python RPM 提供

在打包 Python 项目时，请确保如果这些目录存在，请确保生成的 RPM 中包含以下目录：

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

从这些目录中，RPM 构建过程会自动生成虚拟 pythonX.Ydist，如 python3.6dist(detox)。这些虚拟提供由 %python_enable_dependency_generator 宏指定的软件包使用。

5.6. 在 PYTHON 脚本中处理解释器指令

在 Red Hat Enterprise Linux 8 中，可执行的 Python 脚本应使用解释器指令（也称为 hashbangs 或 shebangs），它们至少指定了主要 Python 版本。例如：

```
#!/usr/bin/python3
#!/usr/bin/python3.6
#!/usr/bin/python3.8
#!/usr/bin/python3.9
```

```
#!/usr/bin/python3.11
#!/usr/bin/python3.12
#!/usr/bin/python2
```

在构建任何 RPM 软件包时，/usr/lib/rpm/redhat/brp-mangle-shebangs buildroot 策略 (BRP) 脚本会自动运行，并尝试在所有可执行文件中更正解释器指令。

当遇到带有模糊的解释解释器指令的 Python 脚本时，BRP 脚本会生成错误，例如：

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

或者

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

5.6.1. 修改 Python 脚本中的解释器指令

修改 Python 脚本中的解释器指令，以便 RPM 构建时出现构建错误。

先决条件

- Python 脚本中的一些解释器指令会导致构建错误。

流程

要修改解释器指令，请完成以下任务之一：

- 应用 platform-python-devel 软件包中的 pathfix.py 脚本：

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

请注意，可以指定多个 *PATH*。如果 *PATH* 是一个目录，则 pathfix.py 会递归扫描与模式 ^[a-zA-Z0-9_]+\.py$ 匹配的 Python 脚本，而不仅仅是具有模糊的解释器指令。将此命令添加到 %prep 部分，或者在 %install 部分的末尾。

- 修改打包的 Python 脚本，以便它们符合预期格式。为此，pathfix.py 也可以在 RPM 构建进程之外使用。当在 RPM 构建之外运行 pathfix.py 时，将上例中的 *%{_python3}* 替换为解释器指

令的路径，如 /usr/bin/python3。

如果打包的 Python 脚本需要 Python 3.6 以外的版本，请将前面的命令调整为包含所需的版本。

5.6.2. 在自定义软件包中更改 /usr/bin/python3 解释器指令

默认情况下，以 /usr/bin/python3 形式形式的解释器指令被替换为从 platform-python 软件包中指向 Python 的解释器指令，该指令用于使用 Red Hat Enterprise Linux 的系统工具。您可以将自定义软件包中的 /usr/bin/python3 解释器指令更改为指向您从 AppStream 软件仓库安装的特定版本的 Python。

流程

- 要为特定版本的 Python 构建软件包，请将相应 python 软件包的*python*-rpm-macros 子软件包添加到 spec 文件的 BuildRequires 部分。例如，在 Python 3.6 中包括以下行：

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

因此，自定义软件包中的 /usr/bin/python3 解释器指令会自动转换为 /usr/bin/python3.6。

注意

要防止 BRP 脚本检查和修改解释器指令，请使用以下 RPM 指令：

```
%undefine __brp_mangle_shebangs
```

5.7. RUBYGEMS 软件包

本节介绍 RubyGems 软件包是什么，以及如何将它们打包到 RPM 中。

5.7.1. RubyGems 是什么

Ruby 是一个动态、解释、反射、面向对象的通用编程语言。

使用 Ruby 编写的程序通常使用 RubyGems 项目打包，该项目提供了特定的 Ruby 打包格式。

RubyGems 创建的软件包名为 gems，也可以将其重新打包到 RPM 中。

注意

本文档指的是与 gem 前缀相关的 RubyGems 概念，如 .gemspec 用于 gem 规范，且与 RPM 相关的术语无效。

5.7.2. RubyGems 与 RPM 的关系

RubyGems 代表 Ruby 自己的打包格式。但是，RubyGems 包含 RPM 所需的元数据，它启用了从 RubyGems 转换到 RPM。

根据 Ruby 打包指南，可以以这种方式将 RubyGems 软件包重新打包到 RPM 中：

- 这些 RPM 适合其余发行版。
- 最终用户可以通过安装适当的 RPM 软件包 gem 来满足 gem 的依赖项。

RubyGems 使用类似于 RPM 的术语，如 spec 文件、软件包名称、依赖项和其他项目。

要适应 RHEL RPM 的其他发行版本，由 RubyGems 创建的软件包必须遵循以下列出的约定：

- gems 的名称必须遵循此模式：

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

- 要实现 shebang 行，必须使用以下字符串：

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

5.7.3. 从 RubyGems 软件包创建 RPM 软件包

要为 RubyGems 软件包创建源 RPM，需要以下文件：

- gem 文件
- RPM 规格文件

下面的部分描述了如何从 RubyGems 创建软件包中创建 RPM 软件包。

5.7.3.1. RubyGems spec 文件惯例

RubyGems spec 文件必须满足以下条件：

- 包含 %{gem_name} 的定义，这是 gem 规范中的名称。
- 软件包的来源必须是发布的 gem 归档的完整 URL；软件包的版本必须是 gem 的版本。
- 包含 BuildRequires： 一个定义的指令，可以拉取(pull)构建所需的宏。

```
BuildRequires:rubygems-devel
```

- 不包含任何 RubyGems Requires 或 Provides，因为它们是自动生成的。
- 除非要明确指定 Ruby 版本兼容性，否则请不要包含如下定义的 BuildRequires: 指令：

```
Requires: ruby(release)
```

自动生成的对 RubyGems 的依赖关系（Requires: ruby(rubygems)）就足够了。

5.7.3.2. RubyGems macros

下表列出了对于 RubyGems 创建的软件包有用的宏。这些宏由 rubygems-devel 软件包提供。

表 5.4. RubyGems 的宏

宏名称	扩展路径	使用
%{gem_dir}	/usr/share/gems	gem 结构的顶级目录。
%{gem_instdir}	%{gem_dir}/gems/%{gem_name}-%{version}	包含 gem 的实际内容的目录。
%{gem_libdir}	%{gem_instdir}/lib	gem 的库目录。
%{gem_cache}	%{gem_dir}/cache/%{gem_name}-%{version}.gem	缓存的 gem。
%{gem_spec}	%{gem_dir}/specifications/%{gem_name}-%{version}.gemspec	gem 规范文件。
%{gem_docdir}	%{gem_dir}/doc/%{gem_name}-%{version}	gem 的 RDoc 文档。
%{gem_extdir_mri}	%{_libdir}/gems/ruby/%{gem_name}-%{version}	gem 扩展的目录。

5.7.3.3. RubyGems spec 文件示例

构建 gems 的 spec 文件示例以及其特定部分的说明如下。

RubyGems spec 文件示例

```
%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}/
```

下表解释 RubyGems spec 文件中特定项目的细节：

表 5.5. 特定于 RubyGems 的 spec 指令

指令	RubyGems 特定
%prep	RPM 可以直接解包 gem 归档，以便您可以运行 **gem unpack** 命令来从 gem 中提取源。**%setup -n %{gem_name}-%{version}** 宏提供 gem 已解压缩的目录。在同一目录级别，会自动创建 **%{gem_name}-%{version}.gemspec** 文件，该文件可用于重新构建 gem，以修改 **.gemspec** 或将补丁应用到代码。
%build	此指令包括将软件构建到机器代码的命令或一系列命令。**%gem_install** 宏只在 gem 归档上运行，而 gem 可使用下一个 gem 构建重新创建。然后，**%gem_install** 创建的 gem 文件会被用于构建代码并安装到临时目录中，默认为 **./%{gem_dir}**。**%gem_install** 宏构建并安装代码。在安装之前，构建的源会被放入自动创建的临时目录中。 **%gem_install** 宏接受两个附加选项：**-n <gem_file>**，它可以覆盖用于安装的 gem，**-d <install_dir>**，它可能会覆盖 gem 安装目的地；不建议使用这个选项。 **%gem_install** 宏不能用于安装到 **%{buildroot}** 中。
%install	安装将在 **%{buildroot}** 层次结构中执行。您可以创建需要的目录，然后将临时目录中安装的内容复制到 **%{buildroot}** 层次结构中。如果这个 gem 创建共享对象，则会移到特定于构架的 **%{gem_extdir_mri}** 路径中。

其他资源

- Ruby 打包指南

5.7.3.4. 使用 gem2rpm 将 RubyGems 软件包转换为 RPM spec 文件

gem2rpm 实用程序将 RubyGems 软件包转换为 RPM 规格文件。

以下小节描述了如何进行：

- 安装 gem2rpm 工具
- 显示所有 gem2rpm 选项
- 使用 gem2rpm 将 RubyGems 软件包覆盖到 RPM spec 文件
- 编辑 gem2rpm 模板

5.7.3.4.1. 安装 gem2rpm

以下流程描述了如何安装 gem2rpm 工具。

流程

- 要从 RubyGems.org 安装 gem2rpm，请运行：

```
$ gem install gem2rpm
```

5.7.3.4.2. 显示 gem2rpm 的所有选项

下面的步骤描述了如何显示 gem2rpm 工具的所有选项。

流程

- 要查看 gem2rpm 的所有选项，请运行：

```
$ gem2rpm --help
```

5.7.3.4.3. 使用 gem2rpm 将 RubyGems 软件包覆盖到 RPM spec 文件

以下流程描述了如何使用 gem2rpm 实用程序将 RubyGems 软件包覆盖到 RPM spec 文件。

流程

- 在其最新版本中下载 gem，并为这个 gem 生成 RPM spec 文件：

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

描述的步骤根据 gem 元数据中提供的信息创建 RPM spec 文件。但是 gem 丢失了通常在 RPM 中提供的一些重要信息，如许可证和更改日志。因此，生成的 spec 文件需要编辑。

5.7.3.4.4. gem2rpm 模板

gem2rpm 模板是一个标准嵌入式 Ruby(ERB)文件，其中包含下表中列出的变量。

表 5.6. gem2rpm 模板中的变量

变量	解释
package	gem 的 **Gem::Package** 变量。
spec	gem 的 **Gem::Specification** 变量（与 format.spec 相同）。
config	**Gem2Rpm::Configuration** 变量，可以重新定义 spec 模板帮助程序中使用的默认宏或规则。
runtime_dependencies	**Gem2Rpm::RpmDependencyList** 变量提供软件包运行时依赖项列表。
development_dependencies	**Gem2Rpm::RpmDependencyList** 变量提供软件包开发依赖项列表。
测试	**Gem2Rpm::TestSuite** 变量提供允许执行测试框架的列表。
files	**Gem2Rpm::RpmFileList** 变量提供软件包中未过滤的文件列表。
main_files	**Gem2Rpm::RpmFileList** 变量提供适合主软件包的文件列表。
doc_files	**Gem2Rpm::RpmFileList** 变量提供适合 **-doc** 子软件包的文件列表。
格式	gem 的 **Gem::Format** 变量。请注意，此变量现已弃用。

5.7.3.4.5. 列出可用的 gem2rpm 模板

使用以下步骤列出所有可用的 gem2rpm 模板。

流程

- 要查看所有可用的模板，请运行：

```
$ gem2rpm --templates
```

5.7.3.4.6. 编辑 gem2rpm 模板

您可以编辑生成 RPM spec 文件而不是编辑生成的 spec 文件的模板。

使用以下步骤编辑 gem2rpm 模板。

流程

1. 保存默认模板：

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

2. 根据需要编辑模板。

3. 使用编辑的模板生成 spec 文件：

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

现在，您可以使用编辑的模板构建一个 RPM 软件包，如 构建 RPM 中所述。

5.8. 如何使用 PERLS 脚本处理 RPM 软件包

从 RHEL 8 开始，默认 buildroot 中不包含 Perl 编程语言。因此，包含 Perl 脚本的 RPM 软件包必须使用 RPM spec 文件中的 BuildRequires: 指令明确指示 Perl 的依赖项。

5.8.1. 与 Perl 相关的常见依赖项

BuildRequires 中使用的与 Perl 相关的构建依赖项是 ：

- perl-generators

  为已安装的 Perl 文件自动生成运行时 Requires 和 Provides。安装 Perl 脚本或 Perl 模块时，必须包含针对这个软件包的构建依赖项。

- perl-interpreter

  如果以任何方式（通过 perl 软件包或 %__perl 宏），或作为软件包构建系统的一部分，则必须将 Perl 解释器列为构建依赖项。

- perl-devel

  提供 Perl 的 header 文件。如果构建特定于架构的代码，该代码链接到 libperl.so 库，如 XS Perl 模块，则必须包括 BuildRequires: perl-devel。

5.8.2. 使用特定的 Perl 模块

如果构建时需要特定的 Perl 模块，请使用以下步骤：

流程

- 在您的 RPM spec 文件中应用以下语法：

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

注意

另外，将此语法应用到 Perl 核心模块，因为它们可能会随时间推移和移出 perl 软件包。

5.8.3. 将软件包限制为特定的 Perl 版本

要将软件包限制为特定的 Perl 版本，请按照以下步骤执行：

流程

- 在 RPM spec 文件中，将 perl (:VERSION) 依赖项与所需版本约束一起使用：

  例如，要将软件包限制为 Perl 版本 5.22 及更新的版本，请使用：

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

警告

不要使用与 perl 软件包版本的比较，因为它会包括 epoch 号。

5.8.4. 确保软件包使用正确的 Perl 解释器

红帽提供了多个 Perl 解释器，它们不完全兼容。因此，任何提供 Perl 模块的软件包都必须在运行时使用在构建时所用的 Perl 解释器。

要确定这一点，请按照以下步骤执行：

流程

- 对于提供 Perl 模块的任何软件包，在 RPM spec 文件中包括版本化的 MODULE_COMPAT Requires ：

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

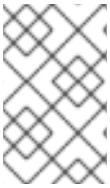
第 6 章 RHEL 8 的新功能

这部分记录了 Red Hat Enterprise Linux 7 和 8 之间的 RPM 打包的最显著变化。

6.1. 支持弱依赖项

弱依赖项 是 Requires 指令的不同版本。这些变体与 virtual Provides: 和使用 Epoch-Version-Release 范围比较的软件包名称匹配。

弱依赖项 有两个强（弱 和提示）和两个方向（正向 向），如下表中所述。



注意

前进 方向与 Requires 类似：在以前的依赖项系统中，后 向 没有模拟。

表 6.1. 弱依赖关系的强度和指示可能的组合

优势/总监	向前	向后
weak	建议：	补充：
提示	建议：	加强：

弱依赖项 策略的主要优点是：

- 它允许最小安装，同时保持丰富的默认安装功能。
- 软件包可为特定供应商指定首选项，同时保持虚拟提供的灵活性。

6.1.1. Weak 依赖项简介

默认情况下，弱依赖项 与常规 Requires 相似：匹配的软件包包含在 YUM 事务中。如果添加软件包会导致错误，YUM 默认会忽略依赖项。因此，用户可以排除由 弱依赖项 添加的软件包，或在以后删除它们。

使用条件

只有在软件包在没有 依赖项 的情况下正常工作时才可以使用 Weak 依赖项。

注意

可以接受创建功能非常有限的软件包，而无需添加任何弱要求。

使用案例

使用 Weak 依赖项，特别是对于合理用例，尽量减小安装，例如构建具有单一目的的虚拟机或容器，且不需要软件包的完整功能集。

Weak 依赖项 的典型用例是：

- Documentation
  - 文档查看器，如果缺少它们，则文档查看器
- 例子
- 插件或附加组件
  - 支持文件格式
  - 支持协议

6.1.2. Hints 强度

YUM 默认忽略 提示。GUI 工具可以使用它们提供默认安装的附加软件包，但与安装的软件包结合使用时非常有用。

请勿对软件包的主用例的要求使用 Hints。将这些要求包括在强或 弱依赖项 中。

软件包首选项

YUM 使用 Weak 依赖项，提示 决定在多个有效软件包之间进行选择是否选择哪些软件包。首选由依赖项指向安装的软件包或被安装的软件包。

请注意，这个功能不会影响依赖项解析的普通规则。例如，弱依赖项 无法强制选择旧版本的软件包。

如果依赖项有多个供应商，则需要软件包可以添加 Suggests：为选择哪个选项的依赖项解析器提供提示。

增强：仅在主软件包和其他供应商同意将提示添加到所需软件包时，才会出于某种原因被使用。

> **例 6.1. 使用 Hint 首选一个软件包**
>
> > Package A: Requires: mysql
> >
> > Package mariadb: Provides: mysql
> >
> > Package community-mysql: Provides: mysql
>
> 如果您希望使用 community-mysql 软件包 → 使用 community-mysql 软件包的 mariadb 软件包：
>
> > Suggests: mariadb to Package A.

6.1.3. forward 和 Backward 依赖项

转发依赖项 与 Requires 类似，对要安装的软件包进行了评估。还安装了匹配软件包的最佳版本。

通常，最好 转发依赖项。在获取添加到系统中的其他软件包时，将依赖项添加到软件包。

对于 向后移植依赖关系，如果也安装了匹配的软件包，则会安装包含依赖项的软件包。

向后兼容性 主要是为向分发或其他第三方软件包附加插件、附加组件或扩展的第三方供应商设计的。

6.2. 支持布尔值依赖项

从版本 4.13 开始,RPM 可以在以下依赖关系中处理布尔表达式：

- Requires
- 建议
- 推荐
- 补充
- 加强
- Conflicts

以下小节描述了 布尔值依赖项语法，它提供了 布尔值运算符 的列表，并解释了 布尔值依赖项嵌套 以及 布尔值依赖项语义。

6.2.1. 布尔值依赖关系语法

布尔表达式始终用括号括起。

它们不是普通依赖项的构建：

- 只使用名称或名称
- 比较
- 版本描述

6.2.2. 布尔值运算符

RPM 4.13 引入了以下布尔值运算符：

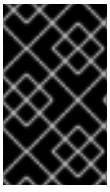
表 6.2. RPM 4.13 中引入的布尔值 operator

布尔值运算符	描述	使用示例
和	需要满足所有操作对象，才能实现这一术语。	冲突：（pkgA 和 pkgB）
或者	需要满足其中一个操作对象，才能使术语变为 true。	Requires:（pkgA >= 3.2 或 pkgB）
if	如果第二个对象需要满足第一个操作对象。（覆盖）	建议（myPkg-langCZ，如果 langsupportCZ）
如果出现其他情况	与 **if** operator 相同，如果第二个操作不是，则要求完成第三个操作对象。	Requires：如果 mariaDB 其他 sqlite，则 myPkg-backend-mariaDB

RPM 4.14 引入了以下额外布尔值运算符：

表 6.3. RPM 4.14 中引入的布尔值运算符

布尔值运算符	描述	使用示例
with	需要同一软件包实现所有操作对象才能使术语为 true。	Requires:(pkgA-foo with pkgA-bar)
without	需要一个满足第一个操作对象但不能满足第二个操作（设置的减法）的单个软件包	Requires:(pkgA-foo without pkgA-bar)
unless	如果第二个对象不是，则需要达到第一个操作对象。（覆盖了负数表示）	Conflicts: (myPkg-driverA unless driverB)
除非另有	与 unless **运算符** 相同，另外，如果第二个操作就会达到第三个操作数。	冲突：(myPkg-backend-SDL1, unless myPkg-backend-SDL2 else SDL2)



重要

if operator 无法与 或 运算在同一上下文中使用，除非 运算符无法用于 和。

6.2.3. 嵌套

操作对象本身可用作布尔值表达式，如下例所示。

请注意，在这种情况下，操作对象也需要由圆括号包围起来。您可以将 和 或 运算符串联在一起，使同一运算符只与一个周围的括号一起重复。

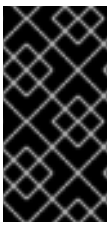
例 6.2. 将操作对象用作布尔值表达式的示例

```
Requires: (pkgA or pkgB or pkgC)

Requires: (pkgA or (pkgB and pkgC))

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

Requires: (pkgA with capB) or (pkgB without capA)

Supplements: ((driverA and driverA-tools) unless driverB)

Recommends: myPkg-langCZ and (font1-langCZ or font2-langCZ) if langsupportCZ
```

6.2.4. 语义

使用 布尔值依赖项 不会更改常规依赖项的语义。

如果使用 布尔值依赖项，则检查所有名称与所有名称匹配的布尔值，并且有匹配项的布尔值则会被聚合在布尔值运算符上。



重要

对于除 Conflicts: 之外的所有依赖项，结果必须是 True 才可以阻止安装。对于 Conflicts : ，结果必须是 False 不可阻止安装。

警告

提供 不是依赖项，不能包含布尔值表达式。

6.2.5. 了解 if operator 的输出

如果 Operator 也返回布尔值，这通常接近直观的理解。然而，以下示例显示，在某些情况下直观理解，如果 具有误导。

例 6.3. if operator 误导输出

如果没有安装 pkgB，则此语句为 true。但是，如果使用此语句时，默认结果为 false，则情况会变得复杂：

```
Requires: (pkgA if pkgB)
```

除非安装了 pkgB，且 pkgA 没有被安装，否则此声明会发生冲突：

```
Conflicts: (pkgA if pkgB)
```

因此，您可能要使用：

```
Conflicts: (pkgA and pkgB)
```

如果 if 运算符嵌套在 or 术语中，则同样如此：

```
Requires: ((pkgA if pkgB) or pkgC or pkg)
```

这也可使整个术语为 true，因为如果 pkgB 尚未安装 pkgB，if term为 true。如果 pkgB 只是安装 pkgB 时的帮助，请使用 和 ：

```
Requires: ((pkgA and pkgB) or pkgC or pkg)
```

6.3. 支持文件触发器

文件触发器 是 RPM scriptlets 的一种类型，

在软件包的 spec 文件中定义。

与 触发器 类似，它们在一个软件包中声明，但在安装或删除包含匹配文件的其他软件包时执行。

文件触发器的常见用法 是更新 registry 或缓存。在这种用例中，包含或管理注册表或缓存的软件包还应包含一个或多个文件 触发器。与软件包控制更新本身的情况相比，包括 文件触发器 可节省时间。

6.3.1. 文件触发器语法

文件触发器 使用以下语法：

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

其中：

file_trigger_tag 定义文件触发器类型。允许的类型是：

- filetriggerin
- filetriggerun
- filetriggerpostun
- transfiletriggerin
- transfiletriggerun

- transfiletriggerpostun

FILE_TRIGGER_OPTIONS 的用途与 RPM 脚本设置选项相同，但 -P 选项除外。

触发器的优先级由数字定义。文件触发器脚本的越大，执行文件触发器脚本的时间就越早。优先级大于 100000 的触发器在标准脚本允许之前执行，其他触发器在标准脚本允许后执行。默认优先级设置为 1000000。

每个类型的文件触发器都必须包含一个或多个路径前缀和脚本。

6.3.2. 文件触发器语法示例

以下示例显示了 文件触发器 语法：

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

在安装包含以 /usr/lib 或 /lib 开头的文件的软件包安装后，此文件触发器直接执行 /usr/bin/ldconfig。即使软件包包含多个以 /usr/lib 或 /lib 开头的路径，文件触发器也会执行一次。但是，以 /usr/lib 或 /lib 开头的所有文件名都传递给触发器脚本的标准输入，以便您可以在脚本内过滤，如下所示：

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

此文件触发器对每个包含 /usr/lib 文件的软件包执行 /usr/bin/ldconfig，同时包含 foo。请注意，前缀匹配的文件包括所有类型的文件，包括常规文件、目录、符号链接等。

6.3.3. 文件触发器类型

文件触发器 有两个主要类型：

- 每个软件包执行一次文件触发器
- 每个事务执行一次的文件触发器

文件触发器 会根据执行时间进一步划分，如下所示：

- 软件包的之前或后，或之后
- 事务前后

6.3.3.1. 每个软件包文件触发器执行一次

每个软件包执行一次文件触发器 为：

- %filetriggerin
- %filetriggerun
- %filetriggerpostun

%filetriggerin

如果此软件包包含与此触发器前缀匹配的一个或多个文件，则该文件触发器会在软件包安装后执行。它还在安装包含此文件触发器的软件包后执行，并且与 rpmdb 数据库中此文件的前缀匹配的一个或多个文件。

%filetriggerun

如果此软件包包含与此触发器前缀匹配的一个或多个文件，则在卸载软件包前执行此文件触发器。它还在卸载包含此文件触发器的软件包之前执行，且与 rpmdb 中此文件的前缀匹配有一个或多个文件。

%filetriggerpostun

如果此软件包包含与此触发器前缀匹配的一个或多个文件，则该文件触发器会在卸载软件包后执行。

6.3.3.2. 每个事务文件触发器执行一次

每个事务执行一次的文件 触发器 为：

- %transfiletriggerin
- %transfiletriggerun
- %transfiletriggerpostun

%transfiletriggerin

对于包含此触发器前缀的一个或多个文件，在事务后执行此文件触发器。如果该事务中有包含此文件触发器的软件包，并且 rpmdb 中此触发器的前缀有一个或多个与这个触发器的前缀匹配的文件，它也在事务后执行。

%transfiletriggerun

此文件触发器会在满足以下条件的所有软件包的事务前执行一次：

- 这个事务中会卸载软件包
- 软件包包含一个或多个与这个触发器前缀匹配的文件

如果该事务中有包含此文件触发器的软件包，且 rpmdb 中与此触发器的前缀匹配了一个或多个文件，则它也会在事务前执行。

%transfiletriggerpostun

对于包含此触发器前缀的一个或多个文件的所有卸载软件包的事务后，会执行一次此文件触发器。

注意

此触发器类型中没有触发文件的列表。

因此，如果您安装或卸载包含库的多个软件包，则会在整个事务结束时更新 ldconfig 缓存。与 RHEL 7 相比，与为每个软件包单独更新缓存时，这可以显著提高性能。另外，不再需要在每个软件包的 spec 文件中称为 ldconfig 中的 scriptlet 和 %postun。

6.3.4. 在 glibc 中使用文件触发器示例

以下示例显示了 glibc 软件包中 文件触发器 的实际用法。

在 RHEL 8 中，文件触发器 在 glibc 中实现，以在安装或卸载事务结束时调用 ldconfig 命令。

通过在 glibc 的 SPEC 文件中包括以下 scriptlets 来确保这一点：

```
%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
```

因此，如果您安装或卸载多个软件包，则会在整个事务完成后为所有已安装的库更新 ldconfig 缓存。因此，不再需要将 scriptlets 调用 ldconfig 包含在各个软件包的 RPM spec 文件中。与 RHEL 7 相比，这提高了性能，其中为每个软件包单独更新缓存。

6.4. 更严格的 SPEC 解析器

SPEC 解析器现在已合并了一些更改。因此，它可以识别之前被忽略的新问题。

6.5. 支持超过 4 GB 的文件

在 Red Hat Enterprise Linux 8 中，RPM 可以使用 64 位变量和标签，它允许在大于 4 GB 的文件和软件包中运行。

6.5.1. 64 位 RPM 标签

64 位版本和之前的 32 位版本中都存在几个 RPM 标签。请注意，64 位版本的名称前带有 LONG 字符串。

表 6.4. 32 位和 64 位版本中可用的 RPM 标签

32 位变体标签名称	62-bit 变体标签名称	标签描述
RPMTAG_SIGSIZE	RPMTAG_LONGSIGSIZE	标头和压缩的有效负载大小。
RPMTAG_ARCHIVESIZE	RPMTAG_LONGARCHIVESIZE	未压缩的有效负载大小。

32 位变体标签名称	62-bit 变体标签名称	标签描述
RPMTAG_FILESIZES	RPMTAG_LONGFILESIZES	文件大小数组。
RPMTAG_SIZE	RPMTAG_LONGSIZE	所有文件大小的总和。

6.5.2. 在命令行中使用 64 位标签

LONG 扩展总是在命令行上启用。如果您之前使用包含 rpm -q --qf 命令的脚本，您可以在此类标签名称中添加 很长时间 ：

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

6.6. 其他功能

与 Red Hat Enterprise Linux 8 中 RPM 打包的其他新功能有：

- 在非详细模式中简化签名检查输出
- 支持强制有效负载验证
- 支持 enforcing 签名检查模式
- 在宏中添加和弃用

其他资源

请参阅以下与 RPM、RPM 打包和 RPM 构建相关的各种主题的参考资料。其中一些是高级的，并扩展了本文档中包含的简介资料。

Red Hat Software Collections Overview - Red Hat Software Collections 产品在最新稳定版本中提供持续更新的开发工具。

Red Hat Software Collections - 打包指南介绍了 Software Collections 以及如何构建和打包它们。具有基本了解 RPM 的软件打包的开发人员和系统管理员可以使用本指南来启动 Software Collections。

Mock - Mock 为各种架构及 Fedora 或 RHEL 版本相比具有构建主机的不同架构提供社区支持的软件包构建解决方案。

RPM 文档 - 官方 RPM 文档.

Fedora 打包指南 - Fedora 的官方打包指南，适用于所有基于 RPM 的发行版。