Red Hat Enterprise Linux 9

打包和分发软件

在 Red Hat Enterprise Linux 9 中打包和发布软件的指南

Last Updated: 2022-06-15

Red Hat Enterprise Linux 9 打包和分发软件

在 Red Hat Enterprise Linux 9 中打包和发布软件的指南

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.

法律通告

摘要

本文档论述了如何将软件打包到 RPM 中。它还演示了如何准备源代码以打包，并解释所选的高级打包场景，如将 Python 项目打包或 RubyGems 打包到 RPM 中。

目录

让开源更具包容性

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。详情请查看 CTO Chris Wright 的信息。

对红帽文档提供反馈

我们感谢您对文档提供反馈信息。请让我们了解如何改进文档。

- 关于特定内容的简单评论：
  1. 请确定您使用 *Multi-page HTML* 格式查看文档。另外，确定 **Feedback** 按钮出现在文档页的右上方。
  2. 用鼠标指针高亮显示您想评论的文本部分。
  3. 点在高亮文本上弹出的 **Add Feedback**。
  4. 按照显示的步骤操作。
- 要通过 Bugzilla 提交反馈，请创建一个新的问题单：
  1. 进入 Bugzilla 网站。
  2. 在 Component 中选择 **Documentation**。
  3. 在 **Description** 中输入您要提供的信息。包括文档相关部分的链接。
  4. 点 **Submit Bug**。

第 1 章 RPM 打包入门

RPM Package Manager(RPM)是一个在 Red Hat Enterprise Linux、CentOS 和 Fedora 上运行的软件包管理系统。您可以使用 RPM 为上述任何所述操作系统分发、管理和更新创建的软件。

与传统存档文件中软件分发相比，RPM 软件包管理系统带来了一些优势。

RPM 可让您：

- 使用标准软件包管理工具（如 DNF 或 PackageKit）安装、重新安装、删除、升级和验证软件包。
- 使用已安装软件包的数据库查询和验证软件包。
- 使用元数据描述软件包、安装说明和其他软件包参数。
- 将软件源、补丁和完成构建指令打包为源代码和二进制软件包。
- 在 DNF 软件仓库中添加软件包。
- 使用 GNU Privacy Guard(GPG)签名密钥来数字签名您的软件包。

第 2 章 为 RPM 打包准备软件

本节介绍如何为 RPM 打包准备软件。为此，了解如何执行代码。但是，您需要了解基本概念，例如 什么是源代码 以及程序是如何 制作的程序。

2.1. 什么是源代码

这部分解释了什么是源代码，并显示了使用三种不同编程语言编写的程序的源代码示例。

源代码是对计算机的人类可读指令，描述如何执行计算。源代码使用编程语言表达。

本文档提供三个使用三种不同编程语言编写的 **Hello World** 程序版本：

- 使用 bash 编写的 hello World
- 使用 Python 编写 hello World
- 使用 C 语言编写的 hello World

每个版本都以不同的方式进行打包。

这些版本的 **Hello World** 程序涵盖了 RPM 软件包器的三个主要用例。

例 2.1. 使用 bash 编写的 hello World

bello 项目在 bash 中实施 **Hello World**。该实施仅包含 **bello** shell 脚本。程序的目的是在命令行中输出 **Hello World**。

bello 文件使用以下语法：

```
#!/bin/bash

printf "Hello World\n"
```

例 2.2. 使用 Python 编写 hello World

pello 项目使用 Python 实施 **Hello World**。该实施仅包含 **pello.py** 程序。程序的目的是在命令行中输出 **Hello World**。

pello.py 文件的语法如下：

```
#!/usr/bin/python3

print("Hello World")
```

例 2.3. 使用 C 语言编写的 hello World

cello 项目使用 C 实施 **Hello World**。实施仅包含 **cello.c** 和 **Makefile** 文件，因此生成的 **tar.gz** 存档除了 **LICENSE** 文件外有两个文件。

程序的目的是在命令行中输出 **Hello World**。

unito.c 文件使用以下语法：

```
#include <stdio.h>

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

2.2. 如何提交程序

从人类可读源代码转换为机器代码（计算机遵循执行程序）的方法包括：

- 程序被原生编译。
- 程序由原始解释器进行解释。
- 程序通过字节编译进行解释。

2.2.1. 原生编译代码

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。这些软件可以独立运行。

以这种方式构建的 RPM 软件包是特定于架构的。

如果您在使用 64 位(x86_64)AMD 或 Intel 处理器的计算机中编译此类软件，则无法在 32 位(x86)AMD 或 Intel 处理器上执行。生成的软件包的名称指定了架构。

2.2.2. 解释代码

有些编程语言（如 bash 或 Python ）不编译到机器代码中。相反，其程序的源代码是在无需进行提前处理的情况下，按步骤直接执行源代码。这是通过相关语言的解析器或一个语言虚拟机实现的。

完全使用解释编程语言编写的软件特定于架构。因此，生成的 RPM 软件包的名称中包含 **noarch** 字符串。

解释语言为 Raw-interpreted 程序，也可以是 编译程序。这两种类型的程序构建过程和打包过程会有所不同。

2.2.2.1. Raw-interpreted 程序

原始解释的语言程序不需要编译，并由解释器直接执行。

2.2.2.2. comp-compiled 程序

字节编译型语言需要编译成字节代码，然后由语言虚拟机执行。

注意

有些语言提供了一个选择：它们可以是原始解释的形式或字节编译的形式。

2.3. 从源构建软件

对于使用编译语言编写的软件，源代码将通过构建流程生成机器代码。这个过程通常称为编译或转换，不同的语言会有所不同。所生成的构建软件可以被运行，使计算机执行程序员指定的任务。

对于使用原始解释语言编写的软件，源代码不会被构建，而是直接执行。

对于以字节编译的解释语言编写的软件，源代码将编译成字节代码，然后由语言虚拟机执行。

以下子章节描述了如何从源代码构建软件。

2.4. 从原生编译的代码构建软件

本节演示了如何将 C 语言编写的 **cello.c** 程序构建成可执行文件。

cello.c

```
#include <stdio.h>

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

2.4.1. 手动构建

如果要手动构建 **cello.c** 程序，请使用此流程：

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. 执行生成的输出二进制 **cello** ：

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

2.4.2. 自动化构建

大规模软件通常使用自动化构建，方法是创建 **Makefile** 文件，然后运行 GNU **make** 实用程序。

如果要使用自动构建来构建 **cello.c** 程序，请使用以下步骤：

步骤

1. 要设置自动化构建，请在与 **cello.c** 相同的目录中使用以下内容创建 **Makefile** 文件。
Makefile

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

```
    rm cello
```

请注意，**cello:** 和 **clean:** 下的行必须以一个 tab 空间开头。

2. 要构建软件，请运行 **make** 命令：

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

3. 因为已有可用的构建，请运行 **make clean** 命令，然后再次运行 **make** 命令：

```
$ make clean
rm cello

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

注意

在另一个构建之后尝试构建程序无效。

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

4. 执行程序：

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

您现在已手动编译程序并使用构建工具。

2.5. 解释代码

本节演示了如何对 Python 编写的程序进行字节编译，以及使用 bash 编写的程序的原始解析。

注意

在下面的两个示例中，文件顶部的 **#!** 行称为 **shebang**，不是编程语言源代码的一部分。

shebang 启用文本文件作为可执行文件：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，然后用作编程语言解释器。功能要求将文本文件标记为可执行文件。

2.5.1. 字节编译代码

本节演示了如何将 Python 编写的 **pello.py** 程序编译成字节代码，然后由 Python 语言虚拟机执行。

Python 源代码也可以是原始解释器，但编译的版本速度更快。因此，RPM Packagers 更喜欢将字节版本打包为最终用户发布。

pello.py

```
#!/usr/bin/python3

print("Hello World")
```

字节程序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

Python 通常进行字节编译，但不采用这里描述的方式。以下过程的目的并不是满足社区标准，而是为了简单过程。有关实际工作环境中的 Python 指南，请参阅打包和发布。

使用这个步骤将 **pello.py** 编译成字节代码：

步骤

1. 字节编译 **pello.py** 文件：

```
$ python -m compileall pello.py

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

2. 在 **pello.pyc** 中执行字节代码：

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

2.5.2. 原始解析代码

本节介绍如何使用 bash shell 内置语言编写的 **bello** 程序。

bello

```
#!/bin/bash

printf "Hello World\n"
```

使用 shell 脚本语言（如 *bash* ）编写的程序是原始的解释。

步骤

- 使含有源代码的文件可执行并运行它：

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

2.6. 修复软件

在 RPM 打包中，而不是修改原始源代码，我们保留它，并在上面使用补丁。

补丁（patch）是用于更新其他源代码的源代码。它被格式化为 *diff*，因为它代表文本的两个版本之间的区别。使用 *diff* 实用程序创建 **diff**，然后使用 patch 实用程序应用到源代码。

注意

软件开发人员通常使用版本控制系统（如 git ）来管理其代码库。这些工具提供自己创建 diffs 或 patching 软件的方法。

本节介绍如何修补软件。

以下示例演示了如何使用 **diff** 从原始源代码创建补丁，以及如何使用 **patch** 应用补丁。创建 RPM 时，将在后面的章节中使用补丁。

此流程演示了如何从原始源代码为 **cello.c** 创建补丁。

步骤

1. 保留原始源代码：

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

-p 选项用于保留模式、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. 使用 **diff** 实用程序生成补丁：

```
$ 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 **-** 开头的行将从原始源代码中删除，并替换为以 **+** 开头的行。

建议将 **Naur** 选项与 **diff** 命令一起使用，因为它符合大多数常见用例。然而，在这种情况下，只需要 **-u** 选项。具体选项可确保：

- **-N** （或 **--new-file**）- 处理缺少的文件，就像它们是空文件一样。
- **-a** （或 **--text**）- 将所有文件作为文本文件。因此，被 **diff** 认为是二进制的文件不会被忽略。
- **-u** (or **-U NUM** 或 **--unified[=NUM]**) - 以统一上下文的输出 NUM（默认为 3）行返回输出。这是一个易于阅读的格式，允许在将补丁应用到更改的源树时进行模糊匹配。
- **-r** (or **--recursive**) - 递归比较任何找到的子目录。
  有关 **diff** 实用程序通用参数的更多信息，请参阅 **diff** 手册页。

4. 将补丁保存到文件中：

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

5. 恢复原始 **cello.c** ：

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

必须保留原始 **cello.c**，因为当构建 RPM 时，将使用原始文件，而不是修改的原始文件。如需更多信息，请参阅使用 SPEC 文件。

以下步骤演示了如何使用 **cello-output-first-patch.patch** 对 **cello.c** 进行补丁，构建补丁的程序并运行它。

步骤

1. 将补丁文件重定向到 **patch** 命令：

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

2. 检查 **cello.c** 的内容现在是否反映了这个补丁：

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

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

3. 构建并运行补丁的 **cello.c** ：

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c

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

2.7. 任意工件

与 UNIX 类似的系统使用文件系统层次结构标准(FHS)指定适合特定文件的目录。

从 RPM 软件包安装的文件按照 FHS 放置在其中。例如，可执行文件应进入一个位于系统 **$PATH** 变量中的目录。

在本文档的上下文中，*Arbitrary Artifact（任意工件）* 代表从 RPM 安装到系统的项。对于 RPM 和系统，它可以是脚本、从软件包的源代码编译的二进制代码、预编译二进制文件或任何其他文件。

以下小节描述了将 *Arbitrary Artifacts* 放置到系统的两个常用方法：

- 使用 install 命令
- 使用 make install 命令

2.8. 使用 INSTALL 命令在系统中放置任意工件

在构建自动化工具（如 GNU make ）进行构建自动化工具时，打包程序通常使用 **install** 命令；例如，打包程序不需要额外的开销。

install 命令由 coreutils 向系统提供，后者将工件放置在文件系统中具有指定权限集的文件系统中。

以下流程使用之前作为此安装方法创建的任意工件的 **bello** 文件。

步骤

1. 运行 **install** 命令，将 **bello** 文件放入 **/usr/bin** 目录中，权限为可执行脚本：

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

现在，**stllo** 位于 **$PATH** 变量中列出的目录中。

2. 从任何目录中执行 **bello**，而不指定其完整路径：

```
$ cd ~

$ bello
Hello World
```

2.9. 使用 MAKE INSTALL 命令在系统中放置任意工件

使用 **make install** 命令是自动安装构建软件到系统的方法。在这种情况下，您需要指定如何在由开发人员编写的 **Makefile** 中向系统安装任意工件。

此流程演示了如何将构建工件安装到系统中所选位置。

步骤

1. 将 **install** 部分添加到 **Makefile** 中：
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
```

请注意，**cello:**, **clean:**, and **install:** 下的行需要以一个 tab 空间开头。

注意

$(DESTDIR) 变量是一个 GNU make 内置变量，通常用于将安装指定到与根目录不同的目录中。

现在，您只能使用 **Makefile** 来构建软件，也可以将其安装到目标系统。

2. 构建并安装 **cello.c** 程序：

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

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

因此，**cello** 现在位于 **$PATH** 变量中列出的目录中。

3. 从任何目录中执行 **cello**，而不指定其完整路径：

```
$ cd ~

$ cello
Hello World
```

2.10. 为打包准备源代码

开发人员通常会将软件作为源代码的压缩存档分发，然后用于创建软件包。RPM 软件包程序与可用的源代码存档配合使用。

软件应通过软件许可证发布。

此流程使用 GPLv3 许可证文本，作为 **LICENSE** 文件示例内容。

步骤

1. **创建** LICENSE 文件，并确保它包含以下内容：

```
$ 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

- 本节中创建的代码

2.11. 将源代码放入 TARBALL

这部分论述了如何将引入的三个 **Hello World** 程序都放在 gzip 的 gzip- 压缩 tarball 中，这也是以后发布软件的常用方法。???

例 2.4. 将 bello 项目放入 tarball

bello 项目在 bash 中实施 **Hello World**。该实施仅包含 **bello** shell 脚本，因此生成的 **tar.gz** 存档除 **LICENSE** 文件外仅有一个文件。

此流程演示了如何准备要分发的 *bello* 项目。

先决条件

在这里，这是计划的 **0.1** 版。

步骤

1. 将所有需要的文件放入一个目录中：

```
$ mkdir /tmp/bello-0.1

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

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

2. 为分发创建存档并将其移动到 **~/rpmbuild/SOURCES/** 目录，这是 **rpmbuild** 命令存储构建软件包的默认目录：

```
$ 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/
```

有关 bash 编写的示例源代码的更多信息，请参阅使用 bash 编写 Hello World。

例 2.5. 将 pello 项目放入 tarball

pello 项目使用 Python 实施 **Hello World**。该实施仅包含 **pello.py** 程序，因此生成的 **tar.gz** 存档除 **LICENSE** 文件外有一个文件。

此流程演示了如何准备用于分发的 *pello* 项目。

先决条件

在这里，这是计划的 **0.1.1** 版本。

步骤

1. 将所有需要的文件放入一个目录中：

```
$ mkdir /tmp/pello-0.1.2

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

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

2. 为分发创建存档并将其移动到 **~/rpmbuild/SOURCES/** 目录，这是 **rpmbuild** 命令存储构建软件包的默认目录：

```
$ 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/
```

有关 Python 编写的示例源代码的更多信息，请参阅 使用 Python 编写 Hello World。

例 2.6. 将 cello 项目放入 tarball

cello 项目使用 C 实施 **Hello World**。实施仅包含 **cello.c** 和 **Makefile** 文件，因此生成的 **tar.gz** 存档除了 **LICENSE** 文件外有两个文件。

请注意，**patch** 文件并没有在程序的归档中发布。构建 RPM 软件包时，RPM 软件包程序应用补丁。补丁和 **.tar.gz** 存档一起放置在 **~/rpmbuild/SOURCES/** 目录中。

此步骤演示了如何准备用于分发的 *cello* 项目。

先决条件

认为这是程序版本 **1.0**。

步骤

1. 将所有需要的文件放入一个目录中：

```
$ 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. 为分发创建存档并将其移动到 **~/rpmbuild/SOURCES/** 目录，这是 **rpmbuild** 命令存储构建软件包的默认目录：

```
$ 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. 添加补丁：

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

有关 C 编写的示例源代码的更多信息，请参阅使用 C 编写 Hello World。

第 3 章 打包软件

本节介绍 RPM 打包的基础知识。

3.1. RPM 软件包

RPM 软件包是包含其它文件和元数据的文件（系统所需文件的信息）。

特别是，RPM 软件包由 **cpio** 归档组成。

cpio 归档包含：

- 文件
- RPM 标头（软件包元数据）
  rpm 软件包管理器使用此元数据来确定依赖项、安装文件的位置和其他信息。

RPM 软件包的类型
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，这些文件描述了如何将源代码构建为二进制 RPM。另外，也可以选择包括源代码的补丁。
- 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。

3.2. 列出 RPM 打包工具的工具

以下流程描述了如何列出 **rpmdevtools** 软件包提供的工具。

先决条件

- 已安装 **rpmdevtools** 软件包，它提供多个用于打包 RPM 的实用程序：

```
# dnf install rpmdevtools
```

步骤

- 列出 RPM 打包工具的工具：

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

有关以上实用程序的更多信息，请参阅其手册页或帮助对话框。

3.3. 设置 RPM 打包工作区

这部分论述了如何使用 **rpmdev-setuptree** 程序设置属于 RPM 打包工作区的目录布局。

先决条件

- 已安装 **rpmdevtools** 软件包，它提供多个用于打包 RPM 的实用程序：

```
# dnf install rpmdevtools
```

步骤

- 运行 **rpmdev-setuptree** 程序：

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

创建的目录用于以下目的：

目录	目的
BUILD	构建软件包时，会创建各种 **%buildroot** 目录。如果日志输出没有足够的信息，这可用于调查失败的构建。
RPMS	此处创建了二进制 RPM，在用于不同架构的子目录中创建，例如在子目录 **x86_64** 和 **noarch** 中。
源	此处，打包商放置了压缩源代码存档和补丁。**rpmbuild** 命令将在此处查找它们。
SPECS	软件包程序在此放置 SPEC 文件。
SRPMS	当 **rpmbuild** 用于构建 SRPM 而不是二进制 RPM 时，会创建生成的 SRPM。

3.4. 什么是 SPEC 文件

您可以将 SPEC 文件作为 **rpmbuild** 实用程序用来构建 RPM 的配方。SPEC 文件通过定义一系列部分中的说明，为构建系统提供必要信息。这些部分在 *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* 部分代表说明的主要部分。

以下小节描述了 SPEC 文件的每个部分。

3.4.1. Preamble 项

下表介绍了 RPM SPEC 文件的 *Preamble* 部分中经常使用的一些指令。

表 3.1. RPM SPEC 文件的*Preamble*部分中使用的项目

SPEC 指令	定义
名称	软件包的基本名称，应该与 SPEC 文件名匹配。
版本	软件的上游版本。
Release	发布此软件版本的次数。通常，将初始值设为 1%{?dist}，并在每个新版软件包中递增。当软件的一个新**版本**构建时，将重置为 1。
概述	软件包的一个简短总结.
许可证	所打包的软件许可证。
URL	有关程序的更多信息的完整 URL。大多数情况下，这是所打包软件的上游项目网站。
Source0	上游源代码的压缩存档的路径或 URL（未修补，补丁会在其他位置处理）。这应该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。如果需要，可以添加更多 SourceX 指令，每次递增数字，例如：Source1、Source2、Source3 等。
Patch	应用于源代码的第一个补丁的名称（如有必要）。 该指令可以通过两种方式应用：带有或不带补丁末尾的数字。 如果没有指定数字，则会在内部分配一个条目。也可以使用 Patch0, Patch1, Patch2, Patch3 明确提供数字。 这些补丁可以通过使用 %patch0、%patch1、%patch2 宏等应用。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，以 SPEC 文件中提供的顺序自动应用所有补丁。
BuildArch	如果软件包没有架构依赖，例如，如果完全使用解释编程语言编写，则将其设置为 **BuildArch: noarch**。如果没有设置，软件包会自动继承构建机器的架构，如 **x86_64**。
BuildRequires	使用编译语言构建程序所需的逗号或空格分开的软件包列表。**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 文件中的独立的行中。
ExcludeArch	如果某一软件不能在特定处理器架构上运行，您可以在此处排除该架构。
Conflicts	**Conflicts** 与 **Requires** 相反。如果存在与 **Conflicts** 匹配的软件包，则软件包是否可以安装取决于，带有 **Conflict** 标签的软件包是否位于已安装的软件包中，还是准备要被安装到的软件包中。

SPEC 指令	定义
Obsoletes	这个指令会改变更新的工作方式，具体取决于 **rpm** 命令是否直接在命令行中使用，或者更新是由更新还是依赖项解析程序执行。当在命令行中使用时，RPM 会删除与正在安装的软件包的过时匹配的所有软件包。当使用更新或依赖项解析器时，包含匹配 **Obsoletes:** 的软件包会作为更新添加并替换匹配的软件包。
Provides	如果向软件包添加了 **Provides**，则软件包可以通过名称以外的依赖项引用。

Name、**Version** 和 **Release** 指令包含 RPM 软件包的文件名。RPM 软件包维护者和系统管理员经常调用这三个指令 **N-V-R** 或 **NVR**，因为 RPM 软件包文件名具有 **NAME-VERSION-RELEASE** 格式。

以下示例演示了如何通过查询 **rpm** 命令获取特定软件包的 **NVR** 信息。

例 3.1. 查询 rpm 为 bash 软件包提供 NVR 信息

```
# 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。最后的标记是 **x86_64**，它向架构发出信号。与 **NVR** 不同，架构标记不直接控制 RPM 打包程序，而是由 **rpmbuild** 构建环境进行定义。这种情况的例外是独立于架构的 **noarch** 软件包。

3.4.2. 正文项

下表列出 RPM SPEC 文件的 **Body 部分**中使用的项目。

表 3.2. RPM SPEC 文件的 Body 部分中使用的项目

SPEC 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	用于准备要构建的软件的命令或一系列命令，例如，在 **Source0** 中解压缩存档。此指令可以包含 shell 脚本。
%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	命令或一系列命令，用于将所需的构建工件从 **%builddir**（构建发生位置）复制到 **%buildroot** 目录（其中包含要打包文件的目录结构）。这通常意味着将文件从 **~/rpmbuild/BUILD** 复制到 **~/rpmbuild/BUILDROOT**，并在 **~/rpmbuild/BUILDROOT** 中创建必要的目录。这仅在创建软件包时运行，而不是当最终用户安装软件包时。有关详细信息，请参阅使用 SPEC 文件。
%check	用于测试软件的命令或一系列命令。这通常包括单元测试等内容。

SPEC 指令	定义
%files	包括在最终用户系统中的文件列表。
%changelog	在不同 **Version** 或 **Release** 之间软件包所发生的更改记录。

3.4.3. 高级 items

SPEC 文件还可以包含高级项目，如 Scriptlets 或 Triggers。

它们在安装过程中对最终用户系统而不是构建过程的不同点生效。

3.5. BUILDROOTS

在 RPM 打包上下文中，**buildroot** 是 chroot 环境。这意味着，构建工件被放在使用与最终用户系统中未来层次结构相同的文件系统层次结构，并将 **buildroot** 用作根目录。构建工件的放置应遵守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。

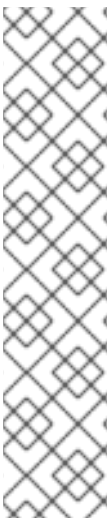
注意

从 Red Hat Enterprise Linux 6 开始，**rpmbuild** 程序有自己的默认值。覆盖这些默认设置会导致几个问题，因此红帽不推荐定义您对该宏的值。您可以在 **rpmbuild** 目录中使用 **%{buildroot}** 宏。

3.6. 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。

示例用法是在 SPEC 文件中多次引用打包软件 *Version*。您仅在 **%{version}** 宏中定义 *Version* 一次，并在 SPEC 文件中使用此宏。每次出现时都会自动替换为您之前定义的 *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}
```

评估 %{_bindir} 和 %{_libexecdir} 宏

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

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

常用的一个宏是 **%{?dist}** 宏，它表示哪个发行版用于构建（分配标签）。

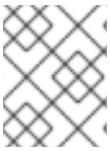
```
# On a RHEL 9.x machine
$ rpm --eval %{?dist}
.el8
```

3.7. 使用 SPEC 文件

要打包新软件，您需要创建新的 SPEC 文件。

实现这一点的方法有两种：

- 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，并填写所需的指令和字段。



注意

某些以编程为导向的文本编辑器，预先使用其自身 SPEC 模板填充新的 **.spec** 文件。**rpmdev-newspec** 实用程序提供了一个与编辑器无关的方法。

以下部分使用 **Hello World!** 程序的三个示例实现，它们描述了 什么源代码。

下表介绍了每个程序。

软件名称	示例说明
bello	程序使用原始解释编程语言编写。它演示了，当不需要构建源代码时，只需要安装源代码。如果需要打包预编译的二进制代码，您也可以使用此方法，因为二进制文件也只是一个文件。
pello	程序以字节编译的解释语言编写。它演示了源代码的字节，并安装字节代码 - 生成的预优化文件。
cello	程序使用原生编译的编程语言编写。它演示了将源代码编译到机器代码中的常见流程，并安装生成的可执行文件。

Hello World 的实现是 ：

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

作为前提条件，这些实施需要放入 **~/rpmbuild/SOURCES** 目录中。

3.8. 使用 RPMDEV-NEWSPEC 创建新的 SPEC 文件

以下步骤演示了如何为上述三个 **Hello World!** 程序（使用 **rpmdev-newspec** 程序）为每一个创建 SPEC 文件。

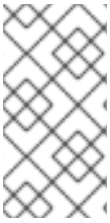
步骤

1. 进入 **~/rpmbuild/SPECS** 目录并使用 **rpmdev-newspec** 实用程序：

```
$ 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.
```

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

2. 检查文件：
   文件中的指令代表 在 什么是 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。



注意

rpmdev-newspec 实用程序不使用特定于任何特定 Linux 发行版的指南或约定。但是，本文档以 Red Hat Enterprise Linux 为目标，因此当引用 RPM 的 Buildroot 时，在 SPEC 文件中与所有其他定义或提供宏一致时，最好使用 **%{buildroot}** 表示法替代 **$RPM_BUILD_ROOT** 表示法。

3.9. 修改原始的 SPEC 文件以创建 RPM

以下步骤演示了如何修改 **rpmdev-newspec** 提供的输出 SPEC 文件以创建 RPM。

先决条件

- 特定程序的源代码已放入 **~/rpmbuild/SOURCES/** 目录中。
- 未填充的 SPEC 文件 **~/rpmbuild/SPECS/<name>.spec** 已被 **rpmdev-newspec** 创建。

步骤

1. 打开 **rpmdev-newspec** 程序提供的 **~/rpmbuild/SPECS/<name>.spec** 文件的输出模板：
2. 填充 SPEC 文件的第一个部分：
   第一部分包括 **rpmdev-newspec** 分组在一起的这些指令：

   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 自动设置为 **1%{?dist}**，它最初是 **1**。每当更新软件包而上游发行版本的 **Version** 没有更改时（例如当包含补丁时），增加初始的值。当出现新的上游版本时，**Release** 被重置为 **1**。

概述

Summary 是该软件的简短说明。

3. 填充 **License**、**URL** 和 **Source0** 指令：
License 字段是与上游发行版本中源代码关联的软件许可证。如何在 SPEC 文件中标记 **License** 的具体格式将有所不同，具体取决于您遵循的基于哪个基于 RPM 的 Linux 发行版准则。

例如，您可以使用 GPLv3+。

URL 字段提供上游软件网站的 URL。为实现一致性，请使用 **%{name}** 的 RPM 宏变量，并使用 **https://example.com/%{name}**。

Source0 字段提供上游软件源代码的 URL。它应直接链接到被打包的特定版本。请注意，本文档中给出的示例 URL 包括可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。通过使用以上，您仅需要在 SPEC 文件中更新一个字段。

4. 填充 **BuildRequires**、**Requires** 和 **BuildArch** 指令：
BuildRequires 指定软件包的构建时依赖项。

Requires 指定软件包的运行时依赖项。

这是使用没有原生编译扩展的解释编程语言编写的软件。因此，使用 **noarch** 值添加 **BuildArch** 指令。这告知 RPM 不需要将这个软件包绑定到构建它的处理器架构。

5. 填充 **%description**、**%prep**、**%build**、**%install**、**%files** 和 **%license** 指令：
这些指令可被视为部分标题，因为它们是可以定义多行、多结构或脚本化任务的指令。

%description 是一个比 **Summary** 更长的软件的信息，其中包含一个或多个段落。

%prep 部分指定如何准备构建环境。这通常涉及对源代码的压缩存档、补丁应用程序以及可能解析源代码中提供的信息的扩展，以便在 SPEC 文件以后的部分中使用。在本节中，您可以使用内置的 **%setup -q** 宏。

%build 部分指定如何构建软件。

%install 部分包含在 **BUILDROOT** 目录中构建软件后如何安装软件的 **rpmbuild** 指令。

该目录是一个空的 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。

%files 部分指定此 RPM 提供的文件列表及其终端用户系统的完整路径位置。

在本节中，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，要表示 LICENSE 文件是软件许可证文件，请使用 **%license** 宏。

6. 最后一个部分(**%changelog**)是软件包的每个 Version-Release 的带有日期戳的条目列表。它们记录打包更改，而非软件更改。打包更改示例：添加补丁，更改 **%build** 部分中的构建流程。
在第一行使用此格式：

以一个 * 字符开头，后跟 **Day-of-Week Month Day Year Name Surname <email> - Version-Release**

使用以下格式进行实际更改条目：

- 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 - 字符开头。

您已为所需的程序编写了整个 SPEC 文件。

其他资源

- 使用 bash 编写的程序的 SPEC 文件示例
- 使用 Python 编写的程序的 SPEC 文件示例
- 使用 C 语言编写的程序的 SPEC 文件示例
- 构建 RPM

3.10. 使用 BASH 编写的程序的 SPEC 文件示例

这部分显示了在 bash 中编写的 **bello** 程序的示例 SPEC 文件。

在 bash 中编写的 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**。

要安装 **bello**，您只需要创建目标目录并在其中安装可执行的 **bash** 脚本文件。因此，您可以使用 **%install** 部分中的 **install** 命令。RPM 宏允许在没有硬编码路径的情况下执行此操作。

其他资源

- 什么是源代码

3.11. 使用 PYTHON 编写的程序的 SPEC 文件示例

本节介绍使用 Python 编程语言编写的 **pello** 程序的示例 SPEC 文件。

使用 Python 编写的 pello 程序的 SPEC 文件示例

```
Name:           python-pello
Version:        1.0.2
Release:        1%{?dist}
Summary:        Example Python library

License:        MIT
URL:            https://github.com/fedora-python/Pello
Source:         %{url}/archive/v%{version}/Pello-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python3-devel

# Build dependencies needed to be specified manually
BuildRequires:  python3-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires:  python3-pytest >= 3


%global _description %{expand:
Pello is an example package with an executable that prints Hello World! on the command line.}

%description %_description

%package -n python3-pello
Summary:        %{summary}

%description -n python3-pello %_description
```

```
%prep
%autosetup -p1 -n Pello-%{version}


%build
# The macro only supported projects with setup.py
%py3_build


%install
# The macro only supported projects with setup.py
%py3_install


%check
%{pytest}


# Note that there is no %%files section for the unversioned python module
%files -n python3-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

# The library files needed to be listed manually
%{python3_sitelib}/pello/

# The metadata files needed to be listed manually
%{python3_sitelib}/Pello-*.egg-info/
```

其他资源

- Python 软件包的 SPEC 文件描述
- 什么是源代码

3.12. 使用 C 语言编写的程序的 SPEC 文件示例

本节介绍使用 C 编程语言编写的 **cello** 程序的示例 SPEC 文件。

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，其中包含执行编译构建过程需要的两个软件包：

- **gcc** 软件包
- **make** 软件包

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 部分反映了编写了 **cello** 程序的 **Makefile** 的事实，因此可以使用 **rpmdev-newspec** 程序提供的 GNU make 命令。但是，您需要删除对 **%configure** 的调用，因为您没有提供配置脚本。

可使用 **rpmdev-newspec** 命令提供的 **%make_install** 宏来完成 **cello** 程序安装。这是因为 **cello** 程序的 **Makefile** 可用。

其他资源

- 什么是源代码

3.13. 构建 RPM

RPM 使用 **rpmbuild** 命令构建。此命令需要特定的目录和文件结构，这与 **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。主要用例有：

- 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
- 从源 RPM 构建二进制 RPM

下面的部分论述了如何在为程序创建 SPEC 文件后构建 RPM。

3.14. 构建源 RPM

以下流程描述了如何构建源 RPM。

先决条件

- 我们要打包的程序的 SPEC 文件必须已经存在。

步骤

- 使用指定的 SPEC 文件运行 **rpmbuild** 命令：

```
$ rpmbuild -bs SPECFILE
```

使用 SPECfile 替换 *SPECFILE*。**-bs** 选项代表构建源。

以下示例显示了为 **bello**、**pello** 和 **cello** 项目构建源 RPM。

为 **bello**、**pello 和 cello** 构**建源 RPM.**

```
$ 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
```

验证步骤

- 确保 **rpmbuild/SRPMS** 目录包含生成的源 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。
- 使用 rpmdev-newspec 创建新的 SPEC 文件
- 修改原始的 SPEC 文件以创建 RPM

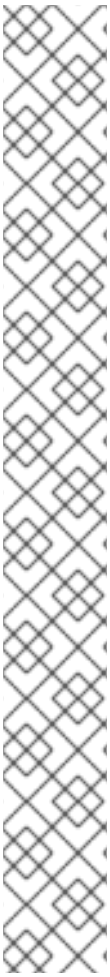
3.15. 从源 RPM 重建二进制 RPM

以下流程演示了如何从源 RPM(SRPM)重建二进制 RPM。

步骤

- 要从 SRPMs 中重建 **bello**、**pello** 和 **cello**，请运行：

```
$ 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]
```



注意

调用 **rpmbuild --rebuild** 涉及：

- 在 **~/rpmbuild/** 目录中安装 SRPM – SPEC 文件和源代码 –。
- 使用安装的内容进行构建.
- 删除 SPEC 文件和源代码。

要在构建后保留 SPEC 文件和源代码，您可以：

- 构建时，使用带有 **--recompile** 选项而非 **--rebuild** 选项的 **rpmbuild** 命令。
- 使用以下命令安装 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%]
```

创建二进制 RPM 时生成的输出是详细的，这对调试非常有用。输出因不同示例而异，并对应于其 SPEC 文件。

如果软件包没有特定架构，生成的二进制 RPM 位于 **~/rpmbuild/RPMS/YOURARCH** 目录中（其中 **YOURARCH** 是您的架构），或位于 **~/rpmbuild/RPMS/noarch/** 目录中。

3.16. 从 SPEC 文件构建二进制 RPM

以下步骤演示了如何从 SPEC 文件构建 **bello**、**pello** 和 **cello** 二进制 RPM。

步骤

- 使用 **bb** 选项运行 **rpmbuild** 命令：

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

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

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

3.17. 从源 RPM 构建二进制 RPM

也可以从源 RPM 构建任何类型的 RPM。要做到这一点，请使用以下步骤。

步骤

- 使用以下选项之一运行 **rpmbuild** 命令，并使用指定的源软件包：

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

其他资源

- **rpmbuild(8)** 手册页

3.18. 检查 RPM 健全性

在创建了软件包后，需要检查软件包的质量。

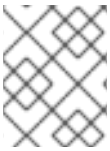
检查软件包质量的主要工具是 rpmlint。

rpmlint 工具执行以下操作：

- 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。

rpmlint 工具可以检查二进制 RPM、源 RPM(SRPMs)和 SPEC 文件，因此它对打包的所有阶段都很有用，如以下部分所示。

请注意，**rpmlint** 有非常严格的准则，因此有时可以接受跳过其中的一些错误和警告，如下例所示。



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。如需了解每个错误或警告的详细说明，您可以运行 **rpmlint -i**。

3.19. 检查 BELLO FOR SANITY

本节介绍在检查 bello SPEC 文件示例和 bello 二进制 RPM 时可能发生的警告和错误。

3.19.1. 检查 bello SPEC 文件

例 3.2. 在适用于 bello 的 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。假设我们预期此 URL 在未来工作，我们可以忽略这个警告。

例 3.3. 在 SRPM forllo 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，有一个新的警告，表示 URL 指令中指定的 **URL** 不可访问。假设链接将在以后工作，我们可以忽略此警告。

3.19.2. 检查 bello 二进制 RPM

在检查二进制 RPM 时，**rpmlint** 会检查以下项目：

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

例 3.4. 在 bello 的二进制 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** 警告表示他 RPM 没有文档或 man page，因为我们没有提供任何文档。除以上警告外，RPM 会传递 **rpmlint** 检查。

3.20. 检查 PELLO FOR SANITY

本节显示在 pello SPEC 文件和 pello 二进制 RPM 示例中检查 RPM 健全时可能出现的警告和错误。

3.20.1. 检查 pello SPEC 文件

例 3.5. 在 pello 的 SPEC 文件中运行 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 将在以后正常工作，您可以忽略这个警告。

hardcoded-library-path 错误建议，使用 **%{_libdir}** 宏而不是使用硬编码的库路径。在本例中，可以安全地忽略这些错误。但是，对于将它们进行生产而言，请确保仔细检查所有错误。

例 3.6. 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.
```

此处新的 **invalid-url URL** 错误是关于 **URL** 指令的，它无法被访问。假设该 URL 将在以后有效，您可以安全地忽略此错误。

3.20.2. 检查 pello 二进制 RPM

在检查二进制 RPM 时，**rpmlint** 会检查以下项目：

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

例 3.7. 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** 警告表示他 RPM 没有文档或 man page，因为没有提供任何文档。

only-non-binary-in-usr-lib 警告表示您在 **/usr/lib/** 中只提供了非二进制工件。该目录通常为共享对象文件保留，它们是二进制文件。因此，**rpmlint** 预期 **/usr/lib/** 目录中的至少一个或者多个文件是二进制的。

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。通常，使用 RPM 宏来确保文件正确放置。在本例中，可以安全地忽略这个警告。

non-executable-script 错误警告 **/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。

除以上警告和错误外，RPM 传递 **rpmlint** 检查。

3.21. 检查完整性的单元格

本节显示在 pello SPEC 文件和 cello 二进制 RPM 示例中检查 RPM 健全时可能出现的警告和错误。

3.21.1. 检查 cello SPEC 文件

例 3.8. 在 SPEC 文件中为 cello 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 将在以后正常工作，您可以忽略这个警告。

例 3.9. 在 SRPM for cello 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，有一个新的警告，表示 URL 指令中指定的 **URL** 不可访问。假设链接将在以后工作，您可以忽略此警告。

3.21.2. 检查 cello 二进制 RPM

在检查二进制 RPM 时，**rpmlint** 会检查以下项目：

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

例 3.10. 在用于 cello 的二进制 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** 警告表示他 RPM 没有文档或 man page，因为您没有提供任何信息。除以上警告外，RPM 会传递 **rpmlint** 检查。

3.22. 将 RPM 活动记录到 SYSLOG

任何 RPM 活动或事务都可以由系统日志协议(syslog)记录。

先决条件

- 要启用 RPM 事务记录到 syslog，请确定在系统中安装了 **syslog** 插件：

```
# dnf 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信息。

要查看 RPM 活动的更新，请按照上述步骤操作。

步骤

1. 打开您配置为存储 syslog 消息的文件，或者使用默认的 syslog 配置，打开 **/var/log/messages** 文件。
2. 搜索包括 **[RPM]** 字符串的新行。

3.23. 提取 RPM 内容

在某些情况下，比如，如果 RPM 所需的软件包被损坏，则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 安装被严重损坏，您可以使用 **rpm2cpio** 实用程序将 RPM 软件包文件转换为 cpio 存档。

以下流程描述了如何使用 **rpm2archive** 实用程序将 rpm 有效负载转换为 tar 归档。

步骤

- 运行以下命令：

```
$ rpm2archive filename.rpm
```

将 *filename* 替换为 **.rpm** 文件的名称。

生成的文件具有 **.tgz** 后缀。例如，要归档 **bash** 软件包：

```
$ 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
```

第 4 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。

4.1. 签名软件包

软件包经过签名，以确保没有第三方可以更改其内容。在下载软件包时，用户可以使用 HTTPS 协议添加额外的安全层。

对软件包进行签名有两种：

- 向已经存在的软件包中添加签名。
- 在已经存在的软件包中替换签名。

先决条件

- 要能够签署软件包，您需要创建 GNU Privacy Guard(GPG)密钥，如创建 GPG 密钥 中所述。

4.1.1. 创建 GPG 密钥

以下流程描述了如何创建签名软件包所需的 GNU Privacy Guard(GPG)密钥。

步骤

1. 生成 GNU Privacy Guard(GPG)密钥对：

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

2. 确认并查看生成的密钥：

```
# 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名称。

4. 将导出的公钥导入到 RPM 数据库中：

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

4.1.2. 配置 RPM 为软件包签名

要能够签署软件包，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。

步骤

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

```
%_gpg_name Key ID
```

使用 GNU Privacy Guard(GPG)密钥 ID 替换 *Key ID*，以用于签名。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。

4.1.3. 在已经存在的软件包中添加签名

这部分论述了在没有签名的情况下构建软件包时最常见的情况。签名仅在软件包发布前添加。

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

通过多个签名，可以将软件包构建器的所有权路径记录到最终用户。

步骤

- 在软件包中添加签名：

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

注意

您需要输入密码来解锁签名的机密密钥。

4.1.4. 在已经存在的软件包中添加签名的实际示例

这部分论述了在现有软件包中添加签名可能会很有用的示例。

公司的一个部门创建了软件包并使用部门的密钥对其进行签名。然后，公司总部检查软件包的签名，并将企业签名添加到软件包中，说明已签名软件包是验证的。

使用两个签名时，该软件包可让其为零售商采用方法。零售商会检查签名，如果匹配，也会添加其签名。

现在，这个软件包已成为希望部署该软件包的公司。检查软件包中的每个签名后，它们知道它是真实的副本。根据部署公司的内部控制，他们可以选择添加自己的签名，以通知其员工收到其公司批准。

4.1.5. 在已经存在的软件包中替换签名

以下流程描述了如何更改公钥而无需重建每个软件包。

步骤

- 要更改公钥，请运行以下命令：

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

注意

您需要输入密码来解锁签名的机密密钥。

--resign 选项还允许您更改多个软件包的公钥，如以下步骤所示。

步骤

- 要更改多个软件包的公钥，请执行：

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

注意

您需要输入密码来解锁签名的机密密钥。

4.2. 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。

4.2.1. 定义您自己的宏

下面的部分论述了如何创建自定义宏。

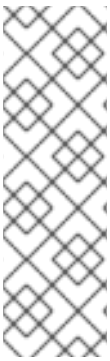
步骤

- 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>** 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，即使被注释掉或者宏的名称被指定到 SPEC 文件的 **%changelog** 部分中。要注释掉宏，请使用 **%%**。例如 **%%global**.

其他资源

- 宏语法

4.2.2. 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(%...)**, 如以下示例所示。

例 4.1. %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。

4.2.2.1. 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。

4.2.2.2. 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**，但源代码以 **hello-1.0.tgz** 中存档，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
```

4.2.2.3. 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。

4.2.2.4. 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 -
```

4.2.2.5. 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**之前（before）**，在进入工作目录前扩展特定源。**-a** 选项代表 **之后，在** 输入后会扩展这些源。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，如果在输入工作目录后扩展 **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，使用 **-b 1**，在进入工作目录前扩展 **Source1** ：

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

4.2.3. %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 4.1. %files 部分中的高级 RPM Macros

Macro	定义
%license	宏识别列为 LICENSE 文件的文件，该文件将被 RPM 安装和标记（例如）。示例：**%license LICENSE**
%doc	宏识别列出为文档的文件，还将安装并标记 RPM。宏用于有关打包软件的文档，以及用于代码示例和各种附带项的文档。在包括事件代码示例时，应谨慎地从文件中删除可执行模式。示例：**%doc README**
%dir	宏可确保路径是此 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)	宏可确保以下文件是一个配置文件，因此如果从原始安装校验和中修改了该文件，则不应在软件包安装或更新包时覆盖（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**

4.2.4. 显示内置宏

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

步骤

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。

4.2.5. 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 宏集合通常作为 RPM 软件包提供，可以使用 **dnf** 软件包管理器进行安装。

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

步骤

- 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

4.2.6. 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。

步骤

- 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。

4.3. 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 的末尾计算机。

4.3.1. 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 文件中未列出此指令，则完全不设置 **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，**dnf** 工具会把一个未设置的 **Epoch** 视为 **Epoch** 为 0 用于处理。

但是，在 SPEC 文件中列出 **Epoch** 时通常会被省略，因为在大多数情况下，如果使用 **Epoch** 值，则在进行软件包版本比较时会 skews 预期的 RPM 行为。

> **例 4.2. 使用 Epoch**
>
> 如果您安装了 **foobar** 软件包，带有 **Epoch:1** 和 **Version:1.0**，以及其它软件包 **foobar**，带有 **Version:2.0** 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。

4.3.2. 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。它们和 SPEC 文件部分标题类似，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 4.2. 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。

4.3.3. 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。

步骤

- 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.

4.3.4. 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
```

4.3.5. 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。

4.3.6. 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：

步骤

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：

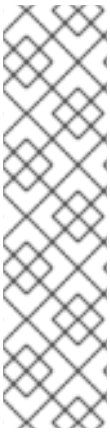
```
%post -p /usr/bin/python3
print("This is {} code".format("python"))
```

4. 按照构建 RPM 中所述构建您的软件包。
5. 安装软件包：

```
# dnf 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，在 SPEC 文件中的 **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。

4.4. 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

4.4.1. 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
```

4.4.2. %if 条件

这部分提供了使用 **%if** RPM 条件的示例。

例 4.3. 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 4.4. 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。

4.4.3. %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体系结构的 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 4.5. 使用 %ifarch 条件的示例

```
%ifarch i386 sparc
…
%endif
```

在 **%ifarch** 和 **%endif** if 之间所有 SPEC 文件的内容都仅在 32 位 AMD 和 Intel 构架或 Sun SPARC 的系统中处理。

%ifnarch 条件

%ifnarch 条件的逻辑与 **%ifarch** 条件的逻辑相反。

例 4.6. 使用 %ifnarch 条件的示例

```
%ifnarch alpha
…
%endif
```

只有在基于 Digital Alpha/AXP 的系统上的数字 Alpha/AXP 系统上执行时，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 4.7. 使用 %ifos 条件的示例

```
%ifos linux
…
%endif
```

只有 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。

4.5. 打包 PYTHON 3 RPM

您可以使用 **pip** 安装程序，或使用 DNF 软件包管理器在系统中安装 Python 软件包。DNF 使用 RPM 软件包格式，它提供对软件的下游控制。

原生 Python 软件包的打包格式由 Python 打包授权机构(PyPA)规范定义。大多数 Python 项目使用 **distutils** 或 **setuptools** 实用程序进行打包，并在 **setup.py** 文件中定义的软件包信息。然而，创建原生 Python 软件包的可能性随着时间推移而演进。有关新兴打包标准的更多信息，请参阅 pyproject-rpm-macros。

本章论述了如何将 **setup.py** 的 Python 项目打包到一个 RPM 软件包中。与原生 Python 软件包相比，此方法提供以下优点：

- 可以对 Python 和非 Python 软件包的依赖项，并严格由 **DNF** 软件包管理器强制执行。
- 您可以用加密的方式为软件包签名。使用加密签名，您可以验证、集成和测试 RPM 软件包的内容与操作系统的其余部分。
- 您可以在构建过程中执行测试。

4.5.1. 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 的指令。这些说明包含在一系列部分中。SPEC 文件有两个主要部分，它们定义了该部分：

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

与非 Python RPM SPEC 文件相比，适用于 Python 项目的 RPM SPEC 文件有一些特定信息。

重要

Python 库的任何 RPM 软件包的名称必须始终包含 **python3-** 前缀。

其他具体信息可在以下 **适用于 python3-pello 软件包 的 SPEC 文件示例**中显示。有关此类特定描述，请查看示例中的备注。

```
Name:           python-pello                                    1
Version:        1.0.2
Release:        1%{?dist}
Summary:        Example Python library

License:        MIT
URL:            https://github.com/fedora-python/Pello
Source:         %{url}/archive/v%{version}/Pello-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python3-devel                                   2

# Build dependencies needed to be specified manually
BuildRequires:  python3-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires:  python3-pytest >= 3


%global _description %{expand:
Pello is an example package with an executable that prints Hello World! on the command line.}

%description %_description

%package -n python3-pello                                       3
Summary:        %{summary}
```

```
%description -n python3-pello %_description


%prep
%autosetup -p1 -n Pello-%{version}


%build
# The macro only supported projects with setup.py
%py3_build


%install
# The macro only supported projects with setup.py
%py3_install


%check
%{pytest}


# Note that there is no %%files section for the unversioned python module
%files -n python3-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

# The library files needed to be listed manually
%{python3_sitelib}/pello/

# The metadata files needed to be listed manually
%{python3_sitelib}/Pello-*.egg-info/
```

1. 将 Python 项目打包到 RPM 中时，始终将 **python-** 前缀添加到项目的原始名称。这里的原始名称为 **pello**，因此 **源 RPM(SRPM)** 的名称是 **python-pello**。

2. **BuildRequires** 指定构建并测试此软件包所需的软件包。在 **BuildRequires** 中，始终包括提供构建 Python 软件包所需工具的项目： **python3-devel** 和您软件包所需的相关项目，如 **python3-setuptools** 或在 **%check** 部分中运行测试所需的运行时和测试依赖关系。

3. 当为二进制 RPM 选择名称时（用户可以安装的软件包）时，添加版本化的 Python 前缀，即当前 **python3-**。因此，生成的二进制 RPM 将命名为 **python3-pello**。

4. **%py3_build** 和 **%py3_install** macros 宏分别运行 **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。

5. **%check** 部分应该运行打包项目的测试。确切的命令取决于项目本身，但可以使用 **%pytest** 宏以 RPM 友好的方式运行 **pytest** 命令。**%{python3}** 宏包含 Python 3 解释器的路径，即 **/usr/bin/python3**。我们建议使用宏，而不是字面上的路径。

4.5.2. Python 3 RPM 的常见宏

在 SPEC 文件中，始终使用以下 Macros *用于 Python 3 RPM* 表而不是硬编码其值的宏。

表 4.3. Python 3 RPM 宏

Macro	常规定义	描述
%{python3}	/usr/bin/python3	Python 3 解释器
%{python3_version}	3.9	Python 3 解释器的 major.minor 版本
%{python3_sitelib}	/usr/lib/python3.9/site-packages	安装纯 Python 模块的位置
%{python3_sitearch}	/usr/lib64/python3.9/site-packages	安装包含特定于架构扩展模块的模块的位置
%py3_build		使用适用于 RPM 软件包的参数运行 **setup.py build** 命令
%py3_install		使用适用于 RPM 软件包的参数运行 **setup.py install** 命令
%{py3_shebang_flags}	s	Python 解释器指令宏的默认标记集，**%py3_shebang_fix**
%py3_shebang_fix		将 Python 解释器指令改为 **#! %{python3}**，保留任何现有标志（如果找到），并添加在 **%{py3_shebang_flags}** 宏中定义的标记

其他资源

- 上游文档中的 Python 宏

4.5.3. 为 Python RPM 使用自动生成的依赖项

以下流程描述了如何在将 Python 项目打包为 RPM 时使用自动生成的依赖项。

先决条件

- RPM 的 SPEC 文件存在。如需更多信息，请参阅 Python 软件包的 SPEC 文件描述。

步骤

1. 确保以下包含上游提供元数据的目录之一包含在生成的 RPM 中：
   - **.dist-info**
   - **.egg-info**
     RPM 构建过程会自动从这些目录中生成虚拟 **pythonX.Ydist**，例如：

     ```
     python3.9dist(pello)
     ```

     然后，Python 依赖项生成器读取上游元数据，并使用生成的 **pythonX.Ydist** 虚拟提供为每个 RPM 软件包生成运行时要求。例如，生成的要求标签可能如下所示：

```
Requires: python3.9dist(requests)
```

2. 检查生成的要求。
3. 要删除其中的一些生成的需要，请使用以下方法之一：
   a. 在 SPEC 文件的 **%prep** 部分中修改上游提供的元数据。
   b. 使用 上游文档 中描述的依赖项自动过滤。
4. 要禁用自动依赖项生成器，请在主软件包的 **%description** 声明中包含 **%{? python_disable_dependency_generator}** 宏。

其他资源

- 自动生成的依赖项

4.6. 在 PYTHON 脚本中处理解释器指令

在 Red Hat Enterprise Linux 9 中，可执行 Python 脚本应该使用解析程序指令（也称为 hashbangs 或 shebangs），至少指定主要 Python 版本。例如：

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

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
```

4.6.1. 修改 Python 脚本中的解释器指令

使用以下步骤修改 Python 脚本中的解释器指令，以便在 RPM 构建时出现错误。

先决条件

- Python 脚本中的一些解释器指令会导致构建错误。

步骤

- 要修改解释器指令，请完成以下任务之一：
  - 在您的 SPEC 文件的 **%prep** 部分中使用以下宏：

    ```
    # %py3_shebang_fix SCRIPTNAME …
    ```

    SCRIPTNAME 可以是任何文件、目录或文件和目录列表。

因此，列出的所有文件以及列出目录中所有 **.py** 文件都会修改其解释器指令以指向 **%{python3}**。将保留原始解释器指令的现有标记，并将添加 **%{py3_shebang_flags}** 宏中定义的其他标志。您可以在 SPEC 文件中重新定义 **%{py3_shebang_flags}** 宏，以更改将要添加的标志。

- 从 **python3-devel** 软件包应用 **pathfix.py** 脚本：

```
# pathfix.py -pn -i %{python3} PATH ...
```

您可以指定多个路径。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。将上述命令添加到 **%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，您也可以使用 RPM 构建进程之外的 **pathfix.py** 脚本。在 RPM 构建之外运行 **pathfix.py** 时，将上面的示例中的 **%{python3}** 替换为解释器指令的路径，如 **/usr/bin/python3**。

其他资源

- 解释器调用

4.7. 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。

4.7.1. 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。

4.7.2. 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 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
```

4.7.3. 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 SPEC 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。

4.7.3.1. 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。

4.7.3.2. 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 4.4. 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。

宏名称	扩展路径	使用
%{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。

4.7.3.3. RubyGems SPEC 文件示例

这部分提供了将 gem 一起构建 gem 的示例，以及其特定部分的内容。

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 文件中特定项的具体内容：

表 4.5. 特定于 RubyGems 的 SPEC 指令

SPEC 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

4.7.3.4. 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 SPEC 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

4.7.3.4.1. 安装 gem2rpm

以下流程描述了如何安装 **gem2rpm** 工具。

步骤

- 要从 RubyGems.org 安装 **gem2rpm**，请运行：

```
$ gem install gem2rpm
```

4.7.3.4.2. 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。

步骤

- 要查看 **gem2rpm** 的所有选项，请运行：

```
gem2rpm --help
```

4.7.3.4.3. 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 软件包到 RPM SPEC 文件。

步骤

- 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。

4.7.3.4.4. 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 4.6. 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。

变量	解释
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。

4.7.3.4.5. 列出可用的 gem2rpm 模板

使用以下步骤列出所有可用的 **gem2rpm** 模板。

步骤

- 要查看所有可用的模板，请运行：

```
$ gem2rpm --templates
```

4.7.3.4.6. 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。

步骤

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 RPM 软件包，如构建 RPM 所述。

4.8. 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:** 指令明确指明 Perl 的依赖项。

4.8.1. 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**。

4.8.2. 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：

步骤

- 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。

4.8.3. 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：

步骤

- 使用 RPM SPEC 文件中的 **perl(:VERSION)** 依赖项与所需的版本约束：
  例如，要将软件包限制为 Perl 版本 5.30 及更高版本，请使用：

```
BuildRequires: perl(:VERSION) >= 5.30
```

警告

不要使用与 **perl** 软件包版本的比较，因为它会包括 epoch 号。

4.8.4. 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：

步骤

- 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 5 章 RHEL 9 中的新功能

这部分记录了 Red Hat Enterprise Linux 8 和 9 之间的 RPM 打包中最重要的变化。

5.1. 动态构建依赖项

Red Hat Enterprise Linux 9 引进了 **%generate_buildrequires** 部分，它可生成动态构建依赖项。

现在，可以使用新可用的 **%generate_buildrequires** 脚本，以编程方式生成额外的构建依赖项。这在使用特殊实用程序编写的语言打包软件时很有用，它用于确定运行时或构建运行时依赖项，如 Rust、Node.js、Ruby、Python 或 Haskell。

您可以使用 **%generate_build requires** 脚本来动态确定在构建时将哪些 **BuildRequires** 指令添加到 SPEC 文件中。如果存在，**%generate_buildrequires** 在 **%prep** 部分后执行，并可以访问解压缩并修补的源文件。脚本必须使用与常规 **BuildRequires** 指令相同的语法将找到的构建依赖项打印到标准输出。

然后，**rpmbuild** 实用程序会在继续构建前检查是否满足依赖关系。

如果缺少一些依赖项，则会创建带有 **.buildreqs.nosrc.rpm** 后缀的软件包，其中包含找到的 **BuildRequires**，且没有源文件。在重启构建前，您可以使用此软件包在安装 **dnf builddep** 命令中缺少的构建依赖项。

有关更多信息，请参阅 **rpmbuild(8)** man page 中的 **DYNAMIC BUILD DEPENDENCIES** 部分。

其他资源

- **rpmbuild(8)** 手册页
- **yum-builddep(1)** man page

5.2. 改进了补丁声明

5.2.1. 可选的自动补丁和源编号

Patch: 和 **Source:** 标签现在会根据列出它们的顺序自动为没有数字编号。

numbering 由 **rpmbuild** 实用程序从上一次手动编号的条目在内部运行，如果没有此类条目，则为 **0**。

例如：

```
Patch: one.patch
Patch: another.patch
Patch: yet-another.patch
```

5.2.2. %patchlist 和 %sourcelist 部分

现在，可以使用新添加的 **%patchlist** 和 **%sourcelist** 部分，列出补丁和源文件，而无需在每个项目前面加上相应的 **补丁：** 和 **Source:** 标签。

例如，以下条目：

```
Patch0: one.patch
Patch1: another.patch
Patch2: yet-another.patch
```

现在可以使用以下内容替换：

```
%patchlist
one.patch
another.patch
yet-another.patch
```

5.2.3. %autopatch 现在接受补丁范围

%autopatch 宏现在接受 **-m** 和 **-M** 参数，以分别限制要应用的最小和最大补丁号：

- **m** 参数指定应用补丁号（含）时要开始的补丁号（含）。
- **-M** 参数指定应用补丁号（含）时要停止的补丁号（含）。

当需要在特定补丁集合之间执行某个操作时，此功能很有用。

5.3. 其他功能

与 Red Hat Enterprise Linux 9 中 RPM 打包的其他新功能包括：

- 快速基于宏的依赖关系生成器
- 强大的宏和 **%if** 表达式，包括 ternary operator 和原生版本比较
- 元（未排序）依赖项
- caret 版本 operator(**^**)，可用于表达高于基本版本的版本。此运算符补充了波形符(**~**)运算符，其采用相对语义。
- **%elif**、**%elifos** 和 **%elifarch** 语句

第 6 章 其他资源

本节介绍了与 RPM、RPM 打包和 RPM 构建相关的各种主题。

- 模拟
- RPM 文档
- RPM 4.15.0 发行注记
- RPM 4.16.0 发行注记
- Fedora 打包指南