Red Hat Enterprise Linux 8

ソフトウェアのパッケージ化および配布

RPM パッケージ管理システムを使用したソフトウェアのパッケージ化

Last Updated: 2024-06-26

Red Hat Enterprise Linux 8 ソフトウェアのパッケージ化および配布

RPM パッケージ管理システムを使用したソフトウェアのパッケージ化

法律上の通知

概要

RPM パッケージマネージャーを使用して、ソフトウェアを RPM パッケージにパッケージ化します。パッケージ化用のソースコードを準備し、ソフトウェアをパッケージ化して、Python プロジェクトや RubyGems を RPM パッケージにパッケージ化するなど、高度なパッケージ化シナリオを調査します。

目次

RED HAT ドキュメントへのフィードバック (英語のみ)

Red Hat ドキュメントに関するご意見やご感想をお寄せください。また、改善点があればお知らせください。

Jira からのフィードバック送信 (アカウントが必要)

1. Jira の Web サイトにログインします。
2. 上部のナビゲーションバーで **Create** をクリックします。
3. **Summary** フィールドにわかりやすいタイトルを入力します。
4. **Description** フィールドに、ドキュメントの改善に関するご意見を記入してください。ドキュメントの該当部分へのリンクも追加してください。
5. ダイアログの下部にある **Create** をクリックします。

第1章 RPM の概要

RPM Package Manager (RPM) は、Red Hat Enterprise Linux (RHEL)、CentOS、および Fedora で実行できるパッケージ管理システムです。RPM を使用すると、これらのオペレーティングシステム用に作成したソフトウェアを配布、管理、および更新できます。

RPM パッケージ管理システムには、従来のアーカイブファイルでソフトウェアを配布する場合と比べて、次のような利点があります。

- RPM は、ソフトウェアを個別にインストール、更新、または削除できるパッケージの形式で管理されるため、オペレーティングシステムのメンテナンスが容易になります。
- RPM パッケージは圧縮アーカイブと同様にスタンドアロンのバイナリーファイルであるため、RPM によりソフトウェアの配布が簡素化されます。これらのパッケージは、特定のオペレーティングシステムとハードウェアアーキテクチャー向けにビルドされています。RPM には、コンパイルされた実行可能ファイルやライブラリーなどのファイルが含まれています。これらのファイルは、パッケージのインストール時にファイルシステム上の適切なパスに配置されます。

RPM を使用すると、次のタスクを実行できます。

- パッケージ化されたソフトウェアをインストール、アップグレード、削除する。
- パッケージソフトウェアの詳細情報を問い合わせる。
- パッケージ化されたソフトウェアの整合性を検証する。
- ソフトウェアソースから独自のパッケージをビルドし、ビルド手順を完了する。
- GNU Privacy Guard (GPG) ユーティリティーを使用して、パッケージにデジタル署名する。
- パッケージを YUM リポジトリーに公開する。

Red Hat Enterprise Linux では、RPM は YUM や PackageKit などの上位レベルのパッケージ管理ソフトウェアに完全に統合されています。RPM には独自のコマンドラインインターフェイスが用意されていますが、ほとんどのユーザーはこのようなソフトウェアを通じて RPM を操作するだけで済みます。ただし、RPM パッケージをビルドする場合は、**rpmbuild(8)** などの RPM ユーティリティーを使用する必要があります。

1.1. RPM パッケージ

RPM パッケージは、ファイルのアーカイブと、これらのファイルのインストールと消去に使用されるメタデータで構成されます。具体的には、RPM パッケージには次の要素が含まれています。

GPG 署名

GPG 署名は、パッケージの整合性を検証するために使用されます。

RPM ヘッダー (パッケージのメタデータ)

RPM パッケージマネージャーは、このメタデータを使用して、パッケージの依存関係、ファイルのインストール先、その他の情報を確認します。

ペイロード

ペイロードは、システムにインストールするファイルを含む **cpio** アーカイブです。

RPM パッケージには 2 つの種類があります。いずれも、同じファイル形式とツールを使用しますが、コンテンツが異なるため、目的が異なります。

- ソース RPM (SRPM)
  SRPM には、ソースコードと SPEC ファイルが含まれます。これには、ソースコードをバイナリー RPM にビルドする方法が書かれています。必要に応じて、SRPM にはソースコードへのパッチを含めることができます。

  バイナリー RPM

  バイナリー RPM には、ソースおよびパッチから構築されたバイナリーが含まれます。

1.2. RPM パッケージ化ユーティリティーのリスト

パッケージを構築するための **rpmbuild(8)** プログラムに加えて、RPM はパッケージの作成プロセスを容易にするその他のユーティリティーを提供します。これらのプログラムは **rpmdevtools** パッケージにあります。

前提条件

- **rpmdevtools** パッケージがインストールされている。

```
# yum install rpmdevtools
```

手順

- RPM パッケージ化ユーティリティーをリスト表示するには、次のいずれかの方法を使用します。
  - **rpmdevtools** パッケージによって提供される特定のユーティリティーとその簡単な説明をリストするには、次のように入力します。

    ```
    $ rpm -qi rpmdevtools
    ```

  - すべてのユーティリティーをリストするには、次のように入力します。

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

関連情報

- RPM ユーティリティーの man ページ

第2章 RPM パッケージ化を行うためのソフトウェアの作成

RPM パッケージ化用のソフトウェアを準備するには、ソースコードとは何か、およびソフトウェアの作成方法を理解する必要があります。

2.1. ソースコードとは

ソースコードとは、人間が判読できるコンピューターへの命令で、計算の実行方法を記述したものです。ソースコードは、プログラミング言語で書かれています。

3 つの異なるプログラミング言語で書かれた次のバージョンの **Hello World** プログラムにより、RPM パッケージマネージャーの主な使用例を説明します。

- Bash で書かれた **Hello World**
  bello プロジェクトは、Bash で **Hello World** を実装しています。この実装には **bello** シェルスクリプトのみが含まれています。このプログラムの目的は、コマンドラインで **Hello World** を出力することです。

  bello ファイルには次の内容が含まれています。

  ```
  #!/bin/bash

  printf "Hello World\n"
  ```

- Python で書かれた **Hello World**
  pello プロジェクトは、Python で **Hello World** を実装しています。この実装には **pello.py** プログラムのみが含まれています。このプログラムの目的は、コマンドラインで **Hello World** を出力することです。

  pello.py ファイルには次の内容が含まれています。

  ```
  #!/usr/bin/python3

  print("Hello World")
  ```

- C で書かれた **Hello World**
  cello プロジェクトは、C で **Hello World** を実装しています。この実装には **cello.c** と **Makefile** ファイルのみが含まれています。したがって、作成される **tar.gz** アーカイブには、**LICENSE** ファイルに加えて 2 つのファイルが含まれます。このプログラムの目的は、コマンドラインで **Hello World** を出力することです。

  cello.c ファイルには次の内容が含まれています。

  ```
  #include <stdio.h>

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

注記

パッケージ化プロセスは、**Hello World** プログラムのバージョンごとに異なります。

2.2. ソフトウェアの作成方法

次のいずれかの方法を使用して、人間が判読できるソースコードをマシンコードに変換できます。

- ソフトウェアをネイティブにコンパイル します。
- 言語インタープリターまたは言語仮想マシンを使用して ソフトウェアを解釈 します。ソフトウェアは、そのまま解釈することも、バイトコンパイルすることもできます。

2.2.1. ネイティブにコンパイルされたソフトウェア

ネイティブにコンパイルされたソフトウェアは、生成されたバイナリーの実行ファイルでマシンコードにコンパイルされるプログラミング言語で書かれたソフトウェアです。ネイティブにコンパイルされたソフトウェアはスタンドアロンソフトウェアです。

注記

ネイティブにコンパイルされた RPM パッケージは、アーキテクチャー固有のパッケージです。

64 ビット (x86_64) AMD または Intel のプロセッサーを使用するコンピューターでこのようなソフトウェアをコンパイルすると、32 ビット (x86) AMD または Intel プロセッサーでは実行できません。生成されるパッケージには、名前でアーキテクチャーが指定されています。

2.2.2. インタープリター型ソフトウェア

Bash や Python などの一部のプログラミング言語は、マシンコードにコンパイルできません。代わりに、言語インタープリターまたは言語仮想マシンが、事前の変換を行わずにプログラムのソースコードを逐次実行します。

注記

インタープリター型のプログラミング言語でのみ書かれたソフトウェアは、アーキテクチャーに依存しません。そのため、作成される RPM パッケージの名前には **noarch** 文字列が付きます。

インタープリター言語で書かれたソフトウェアは、そのまま解釈することも、バイトコンパイルすることもできます。

- そのまま解釈されるソフトウェア
  このタイプのソフトウェアをコンパイルする必要はありません。そのまま解釈されるソフトウェアは、インタープリターによって直接実行されます。
- バイトコンパイルされるソフトウェア
  このタイプのソフトウェアはまずバイトコードにコンパイルする必要があり、その後言語仮想マシンによって実行されます。

注記

一部のバイトコンパイル言語は、そのまま解釈することも、バイトコンパイルすることもできます。

RPM を使用してソフトウェアをビルドおよびパッケージ化する方法は、これら 2 つのソフトウェアタイプで異なることに注意してください。

2.3. ソースからのソフトウェアのビルド

ソフトウェア構築プロセス中に、ソースコードは、RPM を使用してパッケージ化できるソフトウェアアーティファクトに変換されます。

2.3.1. ネイティブにコンパイルされたコードからのソフトウェアのビルド

次のいずれかの方法を使用して、コンパイル言語で書かれたソフトウェアを実行可能ファイルにビルドできます。

- 手動ビルド
- 自動ビルド

2.3.1.1. サンプル C プログラムを手動で構築する

手動ビルドを使用して、コンパイル言語で書かれたソフトウェアをビルドできます。

C で書かれたサンプルの **Hello World** プログラム (**cello.c**) の内容は次のとおりです。

```
#include <stdio.h>

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

手順

1. GNU コンパイラーコレクション から C コンパイラーを起動し、ソースコードをバイナリーにコンパイルします。

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

2. 作成されたバイナリー **cello** を実行します。

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

2.3.1.2. サンプル C プログラムの自動ビルドの設定

大規模なソフトウェアでは、自動ビルドが一般的に使用されます。**Makefile** ファイルを作成し、GNU **make** ユーティリティーを実行することで、自動ビルドを設定できます。

手順

1. 次の内容を含む **Makefile** ファイルを **cello.c** と同じディレクトリーに作成します。

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

```
clean:
 rm cello
```

cello: と **clean:** の下の行は、行頭にタブ文字 (タブ) を追加する必要があることに注意してください。

2. ソフトウェアをビルドします。

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

3. ビルドが現在のディレクトリーですでに利用可能であるため、**make clean** コマンドを入力してから、**make** コマンドを再度入力します。

```
$ make clean
rm cello

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

GNU **make** システムが既存のバイナリーを検出するため、この時点でプログラムを再度ビルドしても効果がないことに注意してください。

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

4. プログラムを実行します。

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

2.3.2. ソースコードの解釈

次のいずれかの方法を使用して、インタープリター型プログラミング言語で記述されたソースコードをマシンコードに変換できます。

- バイトコンパイル
  ソフトウェアのバイトコンパイル手順は、次の要素によって異なります。
  - プログラミング言語
  - 言語の仮想マシン
  - その言語で使用するツールおよびプロセス

注記

たとえば Python で書かれたソフトウェアをバイトコンパイルできます。配布を目的とした Python ソフトウェアは多くの場合バイトコンパイルされますが、このドキュメントで説明されている方法では行われません。説明されている手順は、コミュニティー標準に準拠することではなく、簡素化することを目的としたものです。実際の Python ガイドラインは Software Packaging and Distribution を参照してください。

Python ソースコードは、そのまま解釈することもできます。ただし、バイトコンパイルされたバージョンの方が高速です。したがって、RPM パッケージの作成者は、エンドユーザーに配布するために、バイトコンパイルされたバージョンをパッケージ化する傾向があります。

- 直接解釈
  Bash などのシェルスクリプト言語で書かれたソフトウェアは、常に直接解釈によって実行されます。

2.3.2.1. サンプル Python プログラムのバイトコンパイル

Python ソースコードの直接解釈ではなくバイトコンパイルを選択すると、より高速なソフトウェアを作成できます。

Python プログラミング言語 (**pello.py**) で記述された **Hello World** プログラムには、次の内容が含まれています。

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

手順

1. **pello.py** ファイルをバイトコンパイルします。

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

2. ファイルのバイトコンパイルされたバージョンが作成されたことを確認します。

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

   出力内のパッケージのバージョンは、インストールされている Python のバージョンによって異なる場合があることに注意してください。

3. **pello.py** でプログラムを実行します。

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

2.3.2.2. サンプル Bash プログラムの生の解釈

Bash シェルの組み込み言語 (**bello**) で記述された **Hello World** プログラムには、次の内容が含まれています。

```
#!/bin/bash

printf "Hello World\n"
```

注記

bello ファイルの先頭にある **シバン** (**#!**) 記号は、プログラミング言語のソースコードの一部ではありません。

シバン を使用して、テキストファイルを実行可能ファイルに変換します。システムのプログラムローダーが、**シバン** を含む行を解析してバイナリー実行可能ファイルへのパスを取得し、それがプログラミング言語インタープリターとして使用されます。

手順

1. ソースコードを含むファイルを実行可能ファイルにします。

```
$ chmod +x bello
```

2. 作成したファイルを実行します。

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

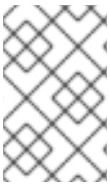
第3章 RPM パッケージ化を行うためのソフトウェアの準備

RPM を使用してソフトウェアをパッケージ化する準備をするには、まずソフトウェアにパッチを適用し、LICENSE ファイルを作成し、それを tarball としてアーカイブします。

3.1. ソフトウェアへのパッチの適用

ソフトウェアをパッケージ化する場合、バグの修正や設定ファイルの変更など、元のソースコードに特定の変更を加えることが必要になる場合があります。RPM パッケージでは、元のソースコードをそのまま残し、パッチを適用することができます。

パッチは、ソースコードファイルを更新するテキストです。パッチは 2 つのバージョンのテキストの差を示すものであるため、**diff** 形式を使用します。**diff** ユーティリティーを使用してパッチを作成し、patch ユーティリティーを使用してそのパッチをソースコードに適用できます。



注記

ソフトウェア開発者は多くの場合、Git などのバージョン管理システムを使用してコードベースを管理します。このようなツールでは、diff やパッチソフトウェアを独自の方法で作成できます。

3.1.1. サンプル C プログラムのパッチファイルを作成する

diff ユーティリティーを使用して、元のソースコードからパッチを作成できます。たとえば、C で記述された **Hello world** プログラム (**cello.c**) にパッチを適用するには、次の手順を実行します。

前提条件

- システムに **diff** ユーティリティーをインストールしました:

```
# yum install diffutils
```

手順

1. 元のソースコードをバックアップします。

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

-p オプションは、モード、所有権、およびタイムスタンプを保持します。

2. 必要に応じて **cello.c** を変更します。

```
#include <stdio.h>

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

3. パッチを生成します。

```
$ 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
```

+ で始まる行は、**-** で始まる行を置き換えます。

注記

diff コマンドには **Naur** オプションを指定することを推奨します。ほとんどのユースケースに適しているためです。

- **-N** (**--new-file**)
  -N オプションは、存在しないファイルを空のファイルとして処理します。
- **-a** (**--text**)
  -a オプションは、すべてのファイルをテキストとして扱います。その結果、**diff** ユーティリティーがバイナリーとして分類されたファイルを無視しなくなります。
- **-u** (**-U NUM** または **--unified[=NUM]**)
  -u オプションは、統一されたコンテキストの出力の NUM 行 (デフォルトは 3 行) の形式で出力を返します。これは、パッチファイルで一般的に使用される、コンパクトで読みやすい形式です。
- **-r** (**--recursive**)
  -r オプションは、**diff** ユーティリティーが検出したサブディレクトリーを再帰的に比較します。

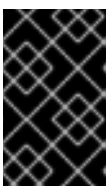
ただし、この場合は、**-u** オプションのみが必要であることに注意してください。

4. ファイルにパッチを保存します。

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

5. 元の **cello.c** を復元します。

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



重要

RPM パッケージマネージャーは RPM パッケージをビルドするときに、変更されたファイルではなく元のファイルを使用するため、元の **cello.c** を保持する必要があります。詳細は、SPEC ファイルでの作業 を参照してください。

関連情報

- **diff(1)** man ページ

3.1.2. サンプル C プログラムのパッチ適用

ソフトウェアにコードパッチを適用するには、**パッチ** ユーティリティーを使用できます。

前提条件

- システムに **パッチ** ユーティリティーをインストールしました:

```
# yum install patch
```

- 元のソースコードからパッチを作成しました。手順については、サンプル C プログラムのパッチファイルの作成を 参照してください。

手順

次の手順では、以前に作成した **cello.patch** ファイルを **cello.c** ファイルに適用します。

1. パッチファイルの出力先を **patch** コマンド変更します。

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

2. **cello.c** の内容に必要な変更が反映されていることを確認します。

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

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

検証

1. パッチを適用した **cello.c** プログラムをビルドします。

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

2. ビルドした **cello.c** プログラムを実行します。

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

3.2. LICENSE ファイルの作成

作成したソフトウェア は、ソフトウェアライセンスを使用して配布することを推奨します。

ソフトウェアライセンスファイルは、ソースコードで何ができるか、何ができないかをユーザーに通知するものです。ソースコードに対するライセンスがないということは、そのコードに対するすべての権利を作成者が保持し、誰もソースコードから複製、配布、または派生著作物を作成できないことを意味

します。

手順

- 必要なライセンスステートメントを含む **LICENSE** ファイルを作成します。

```
$ vim LICENSE
```

例3.1 GPLv3 LICENSE ファイルのテキストの例

```
$ 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/.
```

関連情報

- ソースコードの例

3.3. 配布用ソースコードアーカイブの作成

アーカイブファイルは、**.tar.gz** または **.tgz** 接尾辞を持つファイルです。tarball へのソースコードの追加は、後でパッケージ化して配布するソフトウェアをリリースする一般的な方法です。

3.3.1. サンプル Bash プログラムのソースコードアーカイブを作成する

bello プロジェクトは、Bash の **Hello World** ファイルです。

次の例には、**bello** シェルスクリプトのみが含まれています。したがって、作成される **tar.gz** アーカイブには、**LICENSE** ファイルのほかにファイルが 1 つだけ含まれます。

注記

patch ファイルは、プログラムとともにアーカイブで配布されません。RPM パッケージマネージャーは、RPM のビルド時にパッチを適用します。パッチは、**tar.gz** アーカイブとともに ~/**rpmbuild**/**SOURCES**/ ディレクトリーに配置されます。

前提条件

- **bello** プログラムのバージョン **0.1** を使用する。
- **LICENSE** ファイルを作成しました。手順については、LICENSE ファイルの作成を 参照してください。

手順

1. 必要なファイルをすべて 1 つのディレクトリーに追加します。

```
$ mkdir bello-0.1

$ mv ~/bello bello-0.1/

$ mv LICENSE bello-0.1/
```

2. 配布用のアーカイブを作成します。

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

3. 作成したアーカイブを **~/rpmbuild/SOURCES/** ディレクトリーに移動します。これは、**rpmbuild** コマンドがパッケージをビルドするためのファイルを保存するデフォルトのディレクトリーです。

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

関連情報

- bash で書かれた Hello World

3.3.2. サンプル Python プログラムのソースコードアーカイブを作成する

pello プロジェクトは、Python の **Hello World** ファイルです。

次の例には **pello.py** プログラムのみが含まれています。したがって、作成される **tar.gz** アーカイブには、**LICENSE** ファイルのほかにファイルが 1 つだけ含まれます。

注記

patch ファイルは、プログラムとともにアーカイブで配布されません。RPM パッケージマネージャーは、RPM のビルド時にパッチを適用します。パッチは、**tar.gz** アーカイブとともに **~/rpmbuild/SOURCES/** ディレクトリーに配置されます。

前提条件

- **pello** プログラムのバージョン **0.1.1** を使用する。
- **LICENSE** ファイルを作成しました。手順については、LICENSE ファイルの作成を 参照してください。

手順

1. 必要なファイルをすべて 1 つのディレクトリーに追加します。

```
$ mkdir pello-0.1.1

$ mv pello.py pello-0.1.1/
```

```
$ mv LICENSE pello-0.1.1/
```

2. 配布用のアーカイブを作成します。

```
$ 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. 作成したアーカイブを **~/rpmbuild/SOURCES/** ディレクトリーに移動します。これは、**rpmbuild** コマンドがパッケージをビルドするためのファイルを保存するデフォルトのディレクトリーです。

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

関連情報

- Python で書かれた Hello World

3.3.3. サンプル C プログラムのソースコードアーカイブを作成する

cello プロジェクトは C 言語の **Hello World** ファイルです。

次の例には、**cello.c** と **Makefile** ファイルのみが含まれています。したがって、作成される **tar.gz** アーカイブには、**LICENSE** ファイルに加えて 2 つのファイルが含まれます。

注記

patch ファイルは、プログラムとともにアーカイブで配布されません。RPM パッケージマネージャーは、RPM のビルド時にパッチを適用します。パッチは、**tar.gz** アーカイブとともに **~/rpmbuild/SOURCES/** ディレクトリーに配置されます。

前提条件

- **cello** プログラムのバージョン **1.0** を使用する。
- **LICENSE** ファイルを作成しました。手順については、LICENSE ファイルの作成を 参照してください。

手順

1. 必要なファイルをすべて 1 つのディレクトリーに追加します。

```
$ mkdir cello-1.0

$ mv cello.c cello-1.0/

$ mv Makefile cello-1.0/

$ mv LICENSE cello-1.0/
```

2. 配布用のアーカイブを作成します。

```
$ 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. 作成したアーカイブを ~/**rpmbuild**/**SOURCES**/ ディレクトリーに移動します。これは、**rpmbuild** コマンドがパッケージをビルドするためのファイルを保存するデフォルトのディレクトリーです。

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

関連情報

- C で書かれた Hello World

第4章 ソフトウェアのパッケージ化

次のセクションでは、RPM パッケージマネージャーを使用したパッケージ化プロセスの基本を説明します。

4.1. RPM パッケージ化を行うためのワークスペースの設定

RPM パッケージをビルドするには、まず、さまざまなパッケージ化目的で使用されるディレクトリーで設定される特別なワークスペースを作成する必要があります。

4.1.1. RPM パッケージングワークスペースの設定

RPM パッケージングワークスペースを設定するには、**rpmdev-setuptree** ユーティリティーを使用してディレクトリーレイアウトを設定します。

前提条件

- **rpmdevtools** パッケージをインストールしている。これにより、RPM をパッケージ化するためのユーティリティーがいくつか提供されます。

```
# yum install rpmdevtools
```

手順

- **rpmdev-setuptree** ユーティリティーを実行します。

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

関連情報

- RPM パッケージングワークスペースディレクトリー

4.1.2. RPM パッケージングワークスペースディレクトリー

以下は、**rpmdev-setuptree** ユーティリティーを使用して作成された RPM パッケージワークスペースディレクトリーです。

表4.1 RPM パッケージングワークスペースディレクトリー

ディレクトリー	目的

ディレクトリー	目的
BUILD	**SOURCES** ディレクトリーのソースファイルからコンパイルされたビルドアーティファクトが含まれます。
RPMS	バイナリー RPM は、さまざまなアーキテクチャーのサブディレクトリーの **RPMS** ディレクトリーの下に作成されます。たとえば、**x86_64** または **noarch** サブディレクトリー内。
SOURCES	圧縮されたソースコードアーカイブとパッチが含まれています。次に、**rpmbuild** コマンドはこのディレクトリー内でこれらのアーカイブとパッチを検索します。
SPECS	パッケージャーによって作成された **仕様** ファイルが含まれます。これらのファイルはパッケージの構築に使用されます。
SRPMS	**rpmbuild** コマンドを使用してバイナリー RPM ではなく SRPM をビルドすると、結果の SRPM がこのディレクトリーの下に作成されます。

4.2. スペックファイルについて

spec ファイルは、**rpmbuild** ユーティリティーが RPM パッケージを構築するために使用する指示が記述されたファイルです。SPEC ファイルは、一連のセクションで命令を定義することで、ビルドシステムに必要な情報を提供します。これらのセクションは、**仕様** ファイルの **プリアンブル** と **本文** 部分で定義されています。

- **プリアンブル** セクションには、**本文** セクションで使用される一連のメタデータ項目が含まれています。
- **本文** セクションは、手順の主要部分を表します。

4.2.1. Preamble 項目

以下は、RPM **仕様** ファイルの Preamble セクションで使用できるディレクティブの一部です。

表4.2 前文 セクションの指示

ディレクティブ	定義
Name	**仕様** ファイル名と一致する必要があるパッケージの基本名。
バージョン	ソフトウェアのアップストリームのバージョン番号。

ディレクティブ	定義
リリース	パッケージのバージョンがリリースされた回数。 初期値を **1%{?dist}** に設定し、パッケージの新しいリリースごとに増加します。新しい Version のソフトウェアを構築するときに、1にリセットされます。
Summary	パッケージの1行の概要
ライセンス	パッケージ化しているソフトウェアのライセンス。 SPEC ファイルで **License** にラベルを付ける方法は、使用する RPM ベースの特定の Linux ディストリビューションガイドラインによって異なります。
URL	ソフトウェアの詳細情報の完全な URL (パッケージ化されるソフトウェアのアップストリームプロジェクト Web サイトなど)。
ソース	パッチが適用されていないアップストリームソースコードの圧縮アーカイブへのパスまたは URL。これは、たとえば、パッケージャーのローカルストレージではなく、アップストリームページなどのアーカイブの、アクセス可能で信頼できるストレージを参照している必要があります。 ディレクティブ名の末尾に数字を付けても付けなくても、**Source** ディレクティブを適用できます。番号が指定されていない場合は、エントリーに内部的に番号が割り当てられます。**Source0**、**Source1**、**Source2**、**Source3** などのように、番号を明示的に指定することもできます。
パッチ	必要に応じて、ソースコードに適用する最初のパッチの名前。 ディレクティブ名の末尾に数字を付けても付けなくても、**Patch** ディレクティブを適用できます。番号が指定されていない場合は、エントリーに内部的に番号が割り当てられます。**Patch0**、**Patch1**、**Patch2**、**Patch3** などのように番号を明示的に指定することもできます。 **%patch0**、**%patch1**、**%patch2** マクロなどを使用して、パッチを個別に適用できます。マクロは、RPM SPEC ファイルの Body セクションの %prep ディレクティブ内で適用されます。または、%autounconfined マクロを使用できます。これは、SPEC ファイルに指定されている順序ですべてのパッチを自動的に適用します。
BuildArch	ソフトウェアが構築されるアーキテクチャー。 ソフトウェアがアーキテクチャーに依存しない場合、たとえば、ソフトウェア全体をインタープリタ型プログラミング言語で記述した場合は、値を **BuildArch: noarch** に設定します。この値を設定しないと、ソフトウェアは、**x86_64** など、構築されたマシンのアーキテクチャーを自動的に継承します。

ディレクティブ	定義
BuildRequires	コンパイル言語で書かれたプログラムを構築するのに必要なコンマ区切りまたは空白区切りのリスト。**BuildRequires** のエントリーは複数になる場合があります。各エントリーに対する行が、SPEC ファイル行に含まれます。
Requires	インストール後のソフトウェアの実行に必要なパッケージのコンマ区切りまたは空白区切りのリスト。**Requires** のエントリーは複数ある場合があります。これらは、SPEC ファイル行に独自の行を持ちます。
ExcludeArch	ソフトウェアが特定のプロセッサーアーキテクチャーで動作できない場合は、**ExcludeArch** ディレクティブでこのアーキテクチャーを除外できます。
Conflicts	ソフトウェアをインストールしたときに正常に機能させるために、システムにインストールしてはならないパッケージの、コンマまたは空白で区切られたリスト。**Conflicts** のエントリーは、**仕様** ファイル内の各行に複数存在できます。
Obsoletes	**Obsoletes** ディレクティブは、次の要因に応じて更新の動作方法を変更します。 ● **rpm** コマンドをコマンドラインで直接使用すると、インストールされているパッケージの古いパッケージに一致するすべてのパッケージが削除されるか、更新または依存関係ソルバーによって更新が実行されます。 ● 更新または依存関係リゾルバー (YUM) を使用すると、一致する **Obsoletes:** を含むパッケージが更新として追加され、一致するパッケージが置き換えられます。
提供する項目	パッケージに **Provides** ディレクティブを追加すると、このパッケージは名前以外の依存関係によって参照できるようになります。

Name、**Version**、および **Release** (NVR) ディレクティブは 、**名前 - バージョン - リリース** 形式の RPM パッケージのファイル名を設定します。

rpm コマンドを使用して RPM データベースを照会することにより、特定のパッケージの **NVR** 情報を表示できます。次に例を示します。

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

ここでは、**bash** がパッケージ名で、**4.4.19** がバージョン番号を示し、**7.el8** がリリースを意味しています。**x86_64** マーカーはパッケージアーキテクチャーです。NVR とは異なり、アーキテクチャーのマーカーは RPM パッケージャーで直接管理されていませんが、rpmbuild ビルド環境で定義されます。ただし、これはアーキテクチャーに依存しない **noarch** パッケージです。

4.2.2. Body 項目

RPM SPEC ファイルの **Body section** で使用される項目は次のとおりです。

表4.3 本文セクションの項目

ディレクティブ	定義
%description	RPM でパッケージ化されているソフトウェアの完全な説明。この説明は、複数の行や、複数の段落にまでわたることがあります。
%prep	たとえば、**Source** ディレクティブでアーカイブを展開するなど、ソフトウェアをビルド用に準備するためのコマンドまたは一連のコマンド。**%prep** ディレクティブにはシェルスクリプトを含めることができます。
%build	ソフトウェアをマシンコード (コンパイル言語の場合) またはバイトコード (一部のインタープリター言語の場合) にビルドするための 1 つまたは一連のコマンド。
%install	ソフトウェアがビルドされた後、**rpmbuild** ユーティリティーがソフトウェアを **BUILDROOT** ディレクトリーにインストールするために使用するコマンドまたは一連のコマンド。これらのコマンドは、ビルドが行われる **%_builddir** ディレクトリーから、パッケージ化されるファイルを含むディレクトリー構造を含む **%buildroot** ディレクトリーに、必要なビルドアーティファクトをコピーします。これは通常、ファイルを ~/rpmbuild/BUILD から /rpmbuild/buildroot にコピーして、必要なディレクトリーを /rpmbuild/buildroot に作成することを意味します。 このディレクトリーは空の chroot ベースディレクトリーで、エンドユーザーの root ディレクトリーに似ています。ここでは、インストールしたファイルを格納するディレクトリーを作成できます。このようなディレクトリーを作成するには、パスをハードコードせずに RPM マクロを使用します。 **%install は** パッケージのインストール時ではなく、パッケージの作成時にのみ実行されることに注意してください。詳細は、SPEC ファイルでの作業 を参照してください。
%check	ソフトウェアをテストするためのコマンドまたは一連のコマンド (ユニットテストなど)。
%files	RPM パッケージによって提供され、ユーザーのシステムにインストールされるファイルと、システム上の完全なパスの場所のリスト。 ビルド中に、**%buildroot** ディレクトリーに **%files** にリストされていないファイルがある場合、パッケージ化されていないファイルの可能性があるという警告が表示されます。 **%files** セクション内では、組み込みマクロを使用してさまざまなファイルのロールを示すことができます。これは、**rpm** コマンドを使用してパッケージファイルのマニフェストメタデータをクエリーする場合に便利です。たとえば、**LICENSE** ファイルがソフトウェアライセンスファイルであることを示すには、**%license** マクロを使用します。

ディレクティブ	定義
%changelog	異なる **Version** または **Release** ビルド間でパッケージに行われた変更の記録。これらの変更には、パッケージの各バージョンリリースの日付スタンプ付きエントリーのリストが含まれます。これらのエントリーは、パッチの追加や **%build** セクションでのビルド手順の変更など、ソフトウェアの変更ではなくパッケージの変更を記録します。

4.2.3. 高度な項目

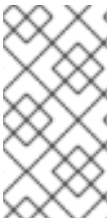
仕様 ファイルには、スクリプトレット や トリガー などの高度な項目を含めることができます。

スクリプトレットとトリガーは、ビルドプロセスではなく、エンドユーザーのシステムでのインストールプロセス中のさまざまな時点で有効になります。

4.3. BUILDROOTS

RPM のパッケージ化のコンテキストでは、**buildroot** が chroot 環境となります。ビルドアーティファクトは、エンドユーザーシステムの将来の階層と同じファイルシステム階層を使用してここに配置され、**buildroot** はルートディレクトリーとして機能します。ビルドアーティファクトの配置は、エンドユーザーシステムのファイルシステム階層の標準に準拠する必要があります。

buildroot のファイルは、後で **dhcpd** アーカイブに置かれ、RPM の主要部分になります。RPM がエンドユーザーのシステムにインストールされている場合、これらのファイルは **root** ディレクトリーに抽出され、階層が正しく保持されます。



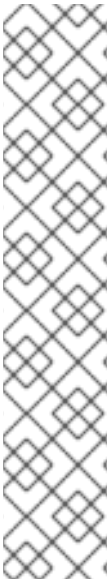
注記

rpmbuild プログラムには独自のデフォルトがあります。これらのデフォルトを上書きすると、特定の問題が発生する可能性があります。したがって、**buildroot** マクロの独自の値を定義することは避けてください。代わりにデフォルトの **%{buildroot}** マクロを使用してください。

4.4. RPM マクロ

rpm マクロ は、特定の組み込み機能が使用されている場合に、ステートメントのオプションの評価に基づいて、条件付きで割り当てられる直接的なテキスト置換です。したがって、RPM はテキストの置換を実行できます。

たとえば、パッケージ化されたソフトウェアの **バージョン** を **%{version}** マクロで 1 回だけ定義し、このマクロを **仕様** ファイル全体で使用することができます。すべての出現は、マクロで定義した **バージョン** に自動的に置き換えられます。



注記

見たことのないマクロが表示されている場合は、次のコマンドを使用してマクロを評価できます。

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

たとえば、**%{_bindir}** マクロと **%{_libexecdir}** マクロを評価するには、次のように入力します。

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

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

関連情報

- マクロの詳細

4.5. SPEC ファイルでの作業

新しいソフトウェアをパッケージ化するには、SPEC ファイルを作成する必要があります。次のいずれかの方法で **spec** ファイルを作成できます。

- 新しい SPEC ファイルを最初から手動で作成します。
- **rpmdev-newspec** ユーティリティーを使用します。このユーティリティーは、空の SPEC ファイルを作成します。このファイルに必要なディレクティブとフィールドを入力します。

注記

プログラマーに焦点を合わせたテキストエディターの中には、独自の SPEC テンプレートで新しい **.spec** ファイルを事前に準備しているものもあります。**rpmdev-newspec** ユーティリティーでは、エディターに依存しないアプローチを利用できます。

4.5.1. サンプル Bash、C、Python プログラム用の新しい仕様ファイルを作成する

rpmdev-newspec ユーティリティーを使用して、**Hello World!** プログラムの 3 つの実装ごとに **spec** ファイルを作成できます。

前提条件

- 次の **Hello World!** プログラム実装は 、**~/rpmbuild/SOURCES** ディレクトリーに配置されました。
  - bello-0.1.tar.gz
  - pello-0.1.2.tar.gz
  - cello-1.0.tar.gz (cello-output-first-patch.patch)

手順

1. **~/rpmbuild/SPECS** ディレクトリーに移動します。

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

2. Hello World! の 3 つの実装それぞれに SPEC ファイルを作成します。

```
$ 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/ ディレクトリーに、**bello.spec**、**cello.spec**、および **pello.spec** という名前の 3 つの SPEC ファイルが追加されます。

3. 作成されたファイルを調べます。
ファイル内のディレクティブは 、仕様ファイルについて で説明されているものを表します。次のセクションでは、**rpmdev -newspec** の出力ファイルの特定のセクションを作成します。

4.5.2. オリジナルのスペックファイルの変更

rpmdev-newspec ユーティリティーによって生成された元の出力 **仕様** ファイルは、**rpmbuild** ユーティリティーに必要な指示を提供するために変更する必要があるテンプレートを表します。**rpmbuild は** これらの命令を使用して RPM パッケージを構築します。

前提条件

- 未入力の **~/rpmbuild/SPECS/<name>.spec仕様** ファイルは 、**rpmdev-newspec** ユーティリティーを使用して作成されました。詳細は、サンプルの Bash、C、および Python プログラム用の新しい仕様ファイルを作成する を参照してください。

手順

1. **rpmdev-newspec** ユーティリティーによって提供される **~/rpmbuild/SPECS/<name>.spec** ファイルを開きます。

2. **仕様** ファイルの **プリアンブル** セクションに次のディレクティブを入力します。

Name

Name はすでに **rpmdev-newspec** の引数として指定されています。

バージョン

Version を、ソースコードのアップストリームのリリースバージョンと一致するように設定します。

リリース

Release は、**1%{?dist}** に自動的に設定されます。最初は **1** となります。

Summary

パッケージの 1 行の説明を入力します。

ライセンス

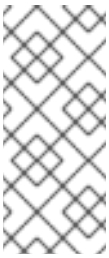
ソースコードに関連付けられているソフトウェアライセンスを入力します。

URL

アップストリームソフトウェア Web サイトの URL を入力します。一貫性を保つために、**%{name}** RPM マクロ変数を利用し、**https://example.com/%{name}** 形式を使用します。

ソース

アップストリームソフトウェアソースコードへの URL を入力します。パッケージ化されているソフトウェアバージョンに直接リンクします。



注記

このドキュメントのサンプル URL には、将来変更される可能性のあるハードコードされた値が含まれています。同様に、リリースのバージョンも変更される可能性があります。今後の変更を簡素化するには、**%{name}** マクロと **%{version}** マクロを使用します。これらのマクロを使用すると、**仕様** ファイル内の 1 つのフィールドのみを更新する必要があります。

BuildRequires

パッケージのビルド時の依存関係を指定します。

Requires

パッケージの実行時の依存関係を指定します。

BuildArch

ソフトウェアアーキテクチャーを指定します。

3. **仕様** ファイルの **Body** セクションに次のディレクティブを入力します。これらのディレクティブは、マルチライン、マルチインストラクション、または実行するスクリプト処理タスクを定義することができるため、セクションの見出しと考えることができます。

%description

ソフトウェアの詳細な説明を入力します。

%prep

ソフトウェアのビルドを準備するためのコマンドまたは一連のコマンドを入力します。

%build

ソフトウェアを構築するためのコマンドまたは一連のコマンドを入力します。

%install

rpmbuild コマンドにソフトウェアを **BUILDROOT** ディレクトリーにインストールする方法を指示するコマンドまたは一連のコマンドを入力します。

%files

システムにインストールする、RPM パッケージによって提供されるファイルのリストを指定します。

%changelog

パッケージの各 **バージョンリリース** の日付スタンプ付きエントリーのリストを入力します。
%changelog セクションの最初の行は * 文字で始まり、その後に **Day-of-Week Month Day Year Name Surname <email> - Version-Release**。

実際の変更エントリーについては、次のルールに従ってください。

- 各変更エントリーには、変更ごとに複数の項目を含めることができます。
- 各項目は新しい行で始まります。

- 各項目はハイフン (-) 文字で始まります。

これで、必要なプログラム用に SPEC ファイル全体を作成できるようになりました。

関連情報

- Preamble 項目
- Body 項目
- サンプル Bash プログラムのサンプル仕様ファイル
- サンプル Python プログラムのサンプル仕様ファイル
- サンプル C プログラムの仕様ファイルの例
- RPM のビルド

4.5.3. サンプル Bash プログラムのサンプル仕様ファイル

以下に示す、bash で書かれた **bello** プログラムの SPEC ファイルの例を参考として使用できます。

bash で記載された bello の SPEC ファイルの例

```
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
```

- **bello** のビルドステップがないため、パッケージのビルドタイム依存関係を指定する **BuildRequires** ディレクティブが削除されました。bash は、raw インタープリタープログラミング言語で、ファイルはシステム上のその場所にインストールされます。
- パッケージのランタイム依存関係を指定する **Requires** ディレクティブは、**bash** のみを含めます。これは、**bello** スクリプトを実行するには **bash** シェル環境のみが必要なためです。
- ソフトウェアのビルド方法を指定する **%build** セクションは、**bash** スクリプトをビルドする必要がないため空白です。

注記

bello をインストールするには、宛先ディレクトリーを作成し、そこに実行可能な **bash** スクリプトファイルをインストールする必要があります。したがって、**%install** セクションで **install** コマンドを使用できます。RPM マクロを使用すると、パスをハードコーディングせずにこれを行うことができます。

関連情報

- ソースコードとは

4.5.4. サンプル Python プログラムのサンプル仕様ファイル

以下に示す、Python プログラミング言語で書かれた **pello** プログラムの SPEC ファイルの例を参考として使用できます。

Python で書かれた pello プログラムの SPEC ファイルサンプル

```
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** ディレクティブ (パッケージにランタイム依存関係を指定) には、以下の 2 つのパッケージが含まれます。
  - 実行時にバイトコンパイルされたコードを実行するために必要な **Python** パッケージ。
  - 小さなエントリーポイントスクリプトを実行するために必要な **bash** パッケージ。
- **BuildRequires** ディレクティブは、パッケージのビルド時の依存関係を指定し、**python** パッケージのみを含みます。**pello** プログラムでは、バイトコンパイルビルドプロセスを実行するために **Python が** 必要です。
- ソフトウェアのビルド方法を指定する **%build** セクションでは、スクリプトのバイトコンパイルバージョンを作成します。実際のパッケージングでは、使用されるディストリビューションに応じて、通常は自動的に実行されることに注意してください。
- **%install** セクションは、バイトコンパイルされたファイルをシステム上のライブラリーディレクトリーにインストールしてアクセスできるようにする必要があるという事実に対応しています。

仕様 ファイル内にインラインでラッパースクリプトを作成するこの例は、**仕様** ファイル自体がスクリプト可能であることを示しています。このラッパースクリプトは、**ヒアドキュメント** を使用して Python バイトコンパイルコードを実行します。

関連情報

- ソースコードとは

4.5.5. サンプル C プログラムの仕様ファイルの例

以下に示す、C プログラミング言語で書かれた **cello** プログラムの SPEC ファイルの例を参考として使用できます。

C 言語で書かれた cello の SPEC ファイルの例

```
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
```

- パッケージのビルド時の依存関係を指定する **BuildRequires** ディレクティブには、コンパイルビルドプロセスを実行するために必要な次のパッケージが含まれます。
  - **gcc**
  - **make**
- この例では、パッケージにランタイム依存関係を指定する **Requires** ディレクティブは省略されています。すべてのランタイム要件は **rpmbuild** により処理されます。**cello** プログラムはコア C 標準ライブラリー以外のものは必要としません。
- **%build** セクションは、この例では **cello** プログラムの **Makefile** ファイルが作成されたという事実を反映しています。したがって、GNU make コマンドを使用できます。ただし、設定スクリプトを指定していないため、%configure に対する呼び出しを削除する必要があります。

%make_install マクロを使用して **cello** プログラムをインストールできます。これは、cello プログラムの Makefile が利用できるため可能です。

関連情報

- ソースコードとは

4.6. RPM のビルド

rpmbuild コマンドを使用して RPM パッケージをビルドできます。このコマンドを使用する場合、**rpmdev-setuptree** ユーティリティーによって設定された構造と同じ特定のディレクトリーおよびファイル構造が想定されます。

rpmbuild コマンドでは、ユースケースや期待する結果によって組み合わせる引数が異なります。主な使用例は次のとおりです。

- ソース RPM のビルド
- バイナリー RPM のビルド
  - ソース RPM からのバイナリー RPM の再ビルド
  - SPEC ファイルからのバイナリー RPM のビルド

4.6.1. ソース RPM のビルド

ソース RPM (SRPM) をビルドすると、次の利点があります。

- 環境にデプロイされた RPM ファイルの特定の **名前、バージョン、リリース** の正確なソースを保持できます。これには、正しい SPEC ファイル、ソースコード、およびすべての関連パッチが含まれます。これは追跡とデバッグに役立ちます。
- 異なるハードウェアプラットフォームまたはアーキテクチャー上でバイナリー RPM をビルドできます。

前提条件

- システムに **rpmbuild** ユーティリティーがインストールされています:

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

- 次の **Hello World!** 実装は **~/rpmbuild/SOURCES/** ディレクトリーに配置されました。
  - bello-0.1.tar.gz
  - pello-0.1.2.tar.gz
  - cello-1.0.tar.gz (cello-output-first-patch.patch)
- パッケージ化するプログラムの **仕様** ファイルが存在します。

手順

1. **~/rpmbuild/SPECS/** ディレクティブに移動します。このディレクティブには、作成した SPEC ファイルが含まれています。

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

2. 指定された **仕様** ファイルを使用して **rpmbuild** コマンドを入力して、ソース RPM をビルドします。

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

-bs オプションは、ビルドソースを表します。

たとえば、**bello**、**pello**、**cello** プログラムのソース RPM をビルドするには、次のように入力します。

```
$ 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
```

検証手順

- **rpmbuild/SRPMS** ディレクトリーに、生成されたソース RPM が含まれていることを確認します。ディレクトリーは、**rpmbuild** で必要な構造の一部です。

関連情報

- SPEC ファイルでの作業
- サンプル Bash、C、Python プログラム用の新しい仕様ファイルを作成する
- オリジナルのスペックファイルの変更

4.6.2. ソース RPM からのバイナリー RPM の再ビルド

ソース RPM (SRPM) からバイナリー RPM を再構築するには、**--rebuild** オプションを指定した **rpmbuild** コマンドを使用します。

バイナリー RPM の作成時に生成される出力は詳細なもので、これはデバッグに役立ちます。この出力は各種例によって異なり、SPEC ファイルに一致します。

生成されるバイナリー RPM は、**~/rpmbuild/RPMS/YOURARCH** ディレクトリー (**YOURARCH** はアーキテクチャー) に配置されます。パッケージがアーキテクチャー固有でない場合は **~/rpmbuild/RPMS/noarch/** ディレクトリーに配置されます。

前提条件

- システムに **rpmbuild** ユーティリティーがインストールされています:

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

手順

1. ソース RPM が含まれている ~/**rpmbuild/SRPMS/** ディレクティブに移動します。

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

2. ソース RPM からバイナリー RPM を再構築します。

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

srpm を ソース RPM ファイルの名前に置き換えます。

たとえば、**bello**、**pello**、**cello を** SRPM から再構築するには、次のように入力します。

```
$ 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]
```

注記

rpmbuild --rebuild を呼び出すと、次のプロセスが実行されます。

- SRPM の内容 (SPEC ファイルおよびソースコード) の、~/**rpmbuild/** ディレクトリーへのインストール。
- インストールされたコンテンツを使用して RPM をビルドします。
- SPEC ファイルとソースコードを削除します。

次のいずれかの方法でビルドした後、**仕様** ファイルとソースコードを保持できます。

- RPM をビルドするときは、**--rebuild** オプションの代わりに **--recompile** オプションを指定して **rpmbuild** コマンドを使用します。
- **bello**、**pello**、**cello** の SRPM をインストールします。

```
$ 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. SPEC ファイルからのバイナリー RPM のビルド

仕様 ファイルからバイナリー RPM をビルドするには、**-bb** オプションを指定した **rpmbuild** コマンドを使用します。

前提条件

- システムに **rpmbuild** ユーティリティーがインストールされています:

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

手順

1. **仕様** ファイルが含まれている **~/rpmbuild/SPECS/** ディレクティブに移動します。

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

2. **仕様** からバイナリー RPM をビルドします。

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

たとえば、**spec** ファイルから **bello**、**pello**、**cello** バイナリー RPM をビルドするには、次のように入力します。

```
$ rpmbuild -bb bello.spec

$ rpmbuild -bb pello.spec

$ rpmbuild -bb cello.spec
```

4.7. RPM のサニティーチェック

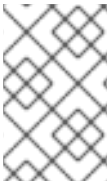
パッケージを作成した後、パッケージの品質を確認することを推奨します。パッケージの品質をチェックするための主なツールは **rpmlint** です。

rpmlint ツールを使用すると、次のアクションを実行できます。

- RPM の保守性を向上します。
- RPM の静的な分析の実行によるサニティーチェック。
- RPM の静的な分析の実行による、エラーチェック。

rpmlint を使用して、バイナリー RPM、ソース RPM (SRPM)、および **仕様** ファイルをチェックできます。したがって、このツールはパッケージ化のすべての段階で役立ちます。

rpmlint のガイドラインは厳密なものです。したがって、次の例に示すように、エラーや警告の一部をスキップしてもよい場合があります。



注記

以下のセクションで説明する例では、**rpmlint** にオプションを指定せずに実行しており、詳細な出力が得られません。各エラーまたは警告の詳細な説明は、代わりに **rpmlint -i** を実行してください。

4.7.1. サンプル Bash プログラムの健全性をチェックする

次のセクションでは、**bello** SPEC ファイルと **bello** バイナリー RPM を例に、RPM のサニティーチェックを行うときに発生する可能性のある警告とエラーを調べます。

4.7.1.1. bello SPEC ファイルのサニティーチェック

次の例の出力を調べて、**bello** SPEC ファイルのサニティーチェックを行う方法を確認してください。

belloスペック ファイルで rpmlint コマンドを実行した出力

```
$ 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.
```

bello.spec の場合、**invalid-url Source0** 警告は 1 つだけです。この警告は、**Source0** ディレクティブにリストされている URL にアクセスできないことを意味します。**example.com** URL は存在しないため、この出力は当然です。この URL が今後有効になると仮定して、この警告を無視します。

bello SRPM で rpmlint コマンドを実行した出力

```
$ 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.
```

bello SRPM には、**URL** ディレクティブで指定された URL にアクセスできないことを意味する新しい **invalid-url URL** 警告があります。この URL が今後有効になると仮定して、この警告を無視します。

4.7.1.2. bello バイナリー RPM のサニティーチェック

バイナリー RPM をチェックする場合、**rpmlint** コマンドは次の項目をチェックします。

- ドキュメント
- man ページ
- ファイルシステム階層規格の一貫した使用

次の例の出力を調べて、**bello** バイナリー RPM のサニティーチェックを行う方法を確認してください。

bello バイナリー RPM で rpmlint コマンドを実行した出力

```
$ 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 および **no-manual-page-for-binary** という警告は、ユーザーがドキュメントや man ページを提供しなかったため、RPM にドキュメントや man ページがないことを意味します。出力の警告を別にすれば、RPM は **rpmlint** チェックに合格しています。

4.7.2. サンプルの Python プログラムの妥当性をチェックする

次のセクションでは、**pello** SPEC ファイルと **pello** バイナリー RPM を例に、RPM のサニティーチェックを行うときに発生する可能性のある警告とエラーを調べます。

4.7.2.1. pello SPEC ファイルのサニティーチェック

次の例の出力を調べて、**pello** SPEC ファイルのサニティーチェックを行う方法を確認してください。

pelloスペック ファイルで rpmlint コマンドを実行した出力

```
$ 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**という警告は、**Source0** ディレクティブにリストされている URL に到達できないことを意味します。**example.com** URL は存在しないため、この出力は当然です。この URL が今後有効になると仮定して、この警告を無視します。
- **hardcoded-library-path** というエラーは、ライブラリーパスをハードコーディングする代わりに **%{_libdir}** マクロを使用することを提案しています。この例では、これらのエラーは無視しても問題はありません。ただし、実稼働環境に導入するパッケージの場合は、すべてのエラーを慎重に確認してください。

pello の SRPM で rpmlint コマンドを実行した場合の出力

```
$ 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 エラーは、**URL** ディレクティブで指定された URL にアクセスできないことを意味します。この URL が今後有効になると仮定して、この警告を無視します。

4.7.2.2. pello バイナリー RPM のサニティーチェック

バイナリー RPM をチェックする場合、**rpmlint** コマンドは次の項目をチェックします。

- ドキュメント
- man ページ
- ファイルシステム階層規格の一貫した使用

次の例の出力を調べて、**pello** バイナリー RPM のサニティーチェックを行う方法を確認してください。

pello バイナリー RPM で rpmlint コマンドを実行した出力

```
$ 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** および **no-manual-page-for-binary** という警告は、ユーザーがドキュメントや man ページを提供しなかったため、RPM にドキュメントや man ページがないことを意味します。

- **Only-non-binary-in-usr-lib** という警告は、**/usr/lib/** ディレクトリーにバイナリーでないアーティファクトしかないことを意味します。このディレクトリーは通常、バイナリーファイルである共有オブジェクトファイルに使用されます。したがって、**rpmlint** は、**/usr/lib/** 内の少なくとも 1 つ以上のファイルがバイナリーであることを想定します。
  これは、ファイルシステム階層規格への準拠についての **rpmlint** チェック例です。ファイルの正しい配置を確保するには、RPM マクロを使用します。この例では、この警告は無視しても問題はありません。

- **non-executable-script** というエラーは、**/usr/lib/pello/pello.py** ファイルに実行権限がないことを意味します。ファイルにシバンが含まれているため、rpmlint ツールは、ファイルが実行ファイルであること想定します。この例では、このファイルは実行権限なしのままにし、このエラーを無視します。

出力の警告とエラーを別にすれば、RPM は **rpmlint** チェックに合格しています。

4.7.3. サンプル C プログラムの妥当性チェック

次のセクションでは、**cello** SPEC ファイルと **cello** バイナリー RPM を例に、RPM のサニティーチェックを行うときに発生する可能性のある警告とエラーを調べます。

4.7.3.1. cello SPEC ファイルのサニティーチェック

次の例の出力を調べて、**cello** SPEC ファイルのサニティーチェックを行う方法を確認します。

celloスペック ファイルで rpmlint コマンドを実行した出力

```
$ 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.
```

cello.spec の場合、**invalid-url Source0** 警告が 1 つだけあります。この警告は、**Source0** ディレクティブにリストされている URL にアクセスできないことを意味します。example.com URL は存在しないため、この出力は当然です。この URL が今後有効になると仮定して、この警告を無視します。

cello SRPM で rpmlint コマンドを実行した出力

```
$ 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.
```

cello SRPM については、**invalid-url URL** という新しい警告があります。この警告は、**URL** ディレクティブで指定された URL に到達できないことを意味します。この URL が今後有効になると仮定して、この警告を無視します。

4.7.3.2. cello バイナリー RPM のサニティーチェック

バイナリー RPM をチェックする場合、**rpmlint** コマンドは次の項目をチェックします。

- ドキュメント
- man ページ
- ファイルシステム階層規格の一貫した使用

次の例の出力を調べて、**cello** バイナリー RPM のサニティーチェックを行う方法を確認してください。

cello バイナリー RPM で rpmlint コマンドを実行した出力

```
$ 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 および **no-manual-page-for-binary** という警告は、ユーザーがドキュメントや man ページを提供しなかったため、RPM にドキュメントや man ページがないことを意味します。

出力の警告を別にすれば、RPM は **rpmlint** チェックに合格しています。

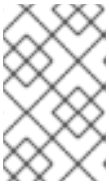
4.8. RPM アクティビティーの SYSLOG へのロギング

System ロギングプロトコル (**syslog**) を使用して、あらゆる RPM アクティビティーまたはトランザクションをログに記録できます。

前提条件

- **syslog** プラグインがシステムにインストールされている。

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



注記

syslog メッセージのデフォルトの場所は **/var/log/messages** ファイルです。ただし、別の場所を使用してメッセージを格納するように syslog を設定できます。

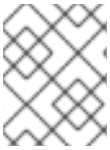
手順

1. **syslog** メッセージを保存するために設定したファイルを開きます。
   あるいは、デフォルトの **syslog** 設定を使用する場合は、**/var/log/messages** ファイルを開きます。

2. **[RPM]** 文字列を含む新しい行を検索します。

4.9. RPM コンテンツの抽出

たとえば、RPM に必要なパッケージが破損している場合など、場合によってはパッケージの内容を抽出する必要があります。この場合、RPM インストールが破損しているにもかかわらず機能している場合は、**rpm2archive** ユーティリティーを使用して、**.rpm** ファイルを tar アーカイブに変換し、パッケージのコンテンツを使用できます。



注記

RPM インストールが著しく破損している場合は、**rpm2cpio** ユーティリティーを使用して RPM パッケージファイルを **cpio** アーカイブに変換できます。

手順

- RPM ファイルを tar アーカイブに変換します。

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

作成されたファイルには **.tgz** 接尾辞が付きます。たとえば、**bash** パッケージからアーカイブを作成するには、次のように入力します。

```
$ 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 高度なトピック

本セクションでは、入門的なチュートリアルの範囲外のトピックについて説明しますが、実際の RPM パッケージ化で役に立ちます。

5.1. RPM パッケージへの署名

RPM パッケージに署名して、第三者がコンテンツを変更できないようにすることができます。セキュリティーのレイヤーを追加するには、パッケージをダウンロードするときに HTTPS プロトコルを使用します。

rpm-sign パッケージで提供される **--addsign** オプションを使用して、パッケージに署名できます。

前提条件

- GPG キーの作成 の説明に従って、GNU Privacy Guard (GPG) キーを作成しました。

5.1.1. GPG キーの作成

パッケージの署名に必要な GNU Privacy Guard (GPG) キーを作成するには、次の手順を使用します。

手順

1. GPG キーペアを生成します。

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

2. 生成されたキーペアを確認します。

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

3. 公開鍵をエクスポートします。

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

<Key_name> を、選択した実際の鍵の名前に置き換えます。

4. エクスポートした公開鍵を RPM データベースにインポートします。

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

5.1.2. パッケージに署名するための RPM の設定

パッケージに署名するには、**%_gpg_name** RPM マクロを指定する必要があります。

以下の手順では、パッケージの署名に使用する RPM を設定する方法を説明します。

手順

- **$HOME/.rpmmacros** で **%_gpg_name** を定義するには、以下のコマンドを実行します。

```
%_gpg_name Key ID
```

Key ID を、パッケージの署名に使用する GNU Privacy Guard (GPG) キー ID に置き換えます。有効な GPG キー ID の値は、鍵を作成したユーザーの氏名またはメールアドレスです。

5.1.3. RPM パッケージへの署名の追加

一般的に、パッケージは署名なしでビルドされます。署名はパッケージのリリース直前に追加されます。

RPM パッケージに署名を追加するには、**rpm -sign** パッケージで使用できる **--addsign** を指定します。

手順

- パッケージに署名を追加します。

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

package-name を、署名する RPM パッケージの名前に置き換えます。

注記

署名の秘密鍵のロックを解除するには、パスワードを入力する必要があります。

5.2. マクロの詳細

本セクションでは、選択したビルトイン RPM マクロについて説明します。そのようなマクロの完全なリストは、RPM ドキュメンテーション を参照してください。

5.2.1. 独自のマクロの定義する

次のセクションでは、カスタムマクロの作成方法を説明します。

手順

- RPM SPEC ファイルに以下の行を含めます。

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

<body> の周りの空白すべてが削除されます。名前は英数字と _ で設定できます。最低でも 3 文字で指定する必要があります。**(opts)** フィールドの指定は任意です。

- **Simple** マクロには、**(opts)** フィールドは含まれません。この場合、再帰的なマクロ拡張のみが実行されます。
- **Parametrized** マクロには、**(opts)** フィールドが含まれます。括弧で囲まれている **opts** 文字列は、マクロ呼び出しの開始時に **argc/argv** 処理の **getopt (3)** に渡されます。

注記

古い RPM SPEC ファイルは、代わりに % **define <name> <body>** マクロパターンを使用します。**%define** マクロと **%global** マクロの違いは次のとおりです。

- **%define** にはローカルスコープがあります。これは、SPEC ファイルの特定の部分に適用されます。使用時に、**%define** マクロの本文が展開されます。
- **%global** にはグローバルスコープがあります。これは SPEC ファイル全体に適用されます。**%global** マクロの本文は、定義時に展開されます。

重要

マクロは、コメントアウトされた場合でも、マクロ名が SPEC ファイルの **%changelog** に指定されている場合でも評価されます。マクロをコメントアウトするには **%%** を使用します。例: **%%global**

関連情報

- マクロ構文

5.2.2. %setup マクロの使用

このセクションでは、**%setup** マクロの異なるバリアントを使用して、ソースコード tarball でパッケージを構築する方法を説明します。マクロバリアントは組み合わせることができることに注意してください。**rpmbuild** の出力は、**%setup** マクロにおける標準的な挙動を示しています。各フェーズの開始時に、マクロは以下の例のように **Executing(%...)** を出力します。

例5.1 %setup マクロの出力例

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

シェルの出力は、**set -x enabled** で設定されます。**/var/tmp/rpm-tmp.DhddsG** の内容を表示するには、**--debug** オプションを指定します。これは、**rpmbuild** により、ビルドの作成後に一時ファイルが削除されるためです。環境変数の設定の後に、以下のような設定が表示されます。

```
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 マクロ:

- 正しいディレクトリーで作業していることを確認します。
- 以前のビルドで残ったファイルを削除します。
- ソース tarball をデプロイメントします。

- 一部のデフォルト権限を設定します。

5.2.2.1. %setup -q マクロの使用

-q オプションでは、**%setup** マクロの冗長性が制限されます。**tar -xvof** の代わりに **tar -xof** のみが実行されます。このオプションは、最初のオプションとして使用します。

5.2.2.2. %setup -n マクロの使用

- n オプションは、拡張 tarball からディレクトリー名を指定します。

展開した tarball のディレクトリーの名前が、想定される名前 (**%{name}-%{version}** と異なる場合に、これを使用すると、**%setup** マクロのエラーが発生することがあります。

たとえば、パッケージ名が **cello** で、ソースコードが **hello-1.0.tgz** でアーカイブされ、**hello**/ ディレクトリーが含まれている場合、SPEC ファイルのコンテンツは次のようになります。

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

5.2.2.3. %setup -c マクロの使用

-c オプションは、ソースコード tarball にサブディレクトリーが含まれておらず、デプロイメント後に、アーカイブのファイルで現在のディレクトリーを埋める場合に使用されます。

次に、**-c** オプションによりディレクトリーが作成され、以下のようにアーカイブデプロイメント手順に映ります。

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

このディレクトリーは、アーカイブ拡張後も変更されません。

5.2.2.4. %setup -D マクロおよび %setup -T マクロの使用

- D オプションは、ソースコードのディレクトリーの削除を無効するため、**%setup** マクロを複数回使用する場合に特に便利です。**-D** オプションでは、次の行は使用されません。

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

- T オプションは、スクリプトから以下の行を削除して、ソースコード tarball の拡張を無効にします。

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

5.2.2.5. %setup -a マクロおよび %setup -b マクロの使用

-a オプションおよび **-b** オプションは、特定のソースを拡張します。

- **-b** オプションは **before** を表します。このオプションは、作業ディレクトリーに移動する前に特定のソースを展開します。

- **-a** オプションは **after** を表します。このオプションは、移動した後にそれらのソースを展開します。これらの引数は、SPEC ファイルのプリアンブルからのソース番号です。

以下の例では、**cello-1.0.tar.gz** アーカイブに空の **example** ディレクトリーが含まれています。サンプルは、別の **example.tar.gz** tarball に同梱されており、同じ名前のディレクトリーに展開されます。この場合、作業ディレクトリーに移動してから **Source1** を展開する場合は、**-a 1** を指定します。

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

次の例では、サンプルは別の **cello-1.0-examples.tar.gz** tarball にあります。これは **cello-1.0/examples** に展開されます。この場合、作業ディレクトリーに移動する前に、**-b 1** を指定して **Source1** を展開します。

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

5.2.3. %files セクション共通の RPM マクロ

次の表は、SPEC ファイルの **%files** セクションに必要な高度な RPM マクロのリストを示しています。

表5.1 %files セクションの高度な RPM マクロ

マクロ	定義
%license	**%license** マクロは、**LICENSE** ファイルとしてリストされているファイルを識別します。このファイルは、RPM によってインストールされ、適切にラベル付けされます。例: **%license LICENSE**.
%doc	**%doc** マクロは、ドキュメントとしてリストされているファイルを識別します。このファイルは、RPM によってインストールされ、適切にラベル付けされます。%doc マクロは、パッケージ化するソフトウェアに関するドキュメントのほか、コード例やさまざまな付随項目にも使用されます。コード例が含まれている場合は、ファイルから実行可能モードを削除するように注意する必要があります。例: **%doc README**
%dir	**%dir** マクロは、パスがこの RPM によって所有されているディレクトリーであることを確認します。これは、RPM ファイルマニフェストが、アンインストール時にどのディレクトリーをクリーンアップするかを正確に認識できるようにするために重要です。例: **%dir %{_libdir}/%{name}**
%config(noreplace)	**%config(noreplace)** マクロは、後続のファイルが設定ファイルであることを確認し、ファイルが元のインストールチェックサムから変更されている場合、パッケージのインストールまたは更新時にファイルを上書き (または置換) しないようにします。変更がある場合は、アップグレード時またはインストール時にファイル名の末尾に **.rpmnew** を追加してファイルが作成され、ターゲットシステム上の既存ファイルまたは変更されたファイルが変更されないようにします。例: **%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

5.2.4. ビルトインマクロの表示

Red Hat Enterprise Linux では、複数のビルトイン RPM マクロを提供しています。

手順

1. ビルトイン RPM マクロをすべて表示するには、以下のコマンドを実行します。

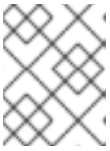
```
rpm --showrc
```

注記

出力のサイズは非常に大きくなります。結果を絞り込むには、**grep** コマンドとともに上記のコマンドを使用します。

2. システムの RPM バージョン用の RPM マクロに関する情報を確認するには、以下のコマンドを実行します。

```
rpm -ql rpm
```



注記

RPM マクロは、出力ディレクトリー構造の **macros** というタイトルのファイルです。

5.2.5. RPM ディストリビューションマクロ

パッケージ化しているソフトウェアの言語実装や、ディストリビューションの特定のガイドラインに基づいて提供する推奨 RPM マクロセットは、ディストリビューションによって異なります。

多くの場合、推奨される RPM マクロセットは RPM パッケージとして提供され、**yum** パッケージマネージャーでインストールできます。

インストールすると、マクロファイルは、**/usr/lib/rpm/macros.d/** ディレクトリーに配置されます。

手順

- raw RPM マクロ定義を表示するには、以下のコマンドを実行します。

```
rpm --showrc
```

上記の出力では、raw RPM マクロ定義が表示されます。

- RPM のパッケージ化を行う際のマクロの機能や、マクロがどう役立つかを確認するには、**rpm --eval** コマンドに、引数として使用するマクロの名前を付けて実行します。

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

関連情報

- **rpm** man ページ

5.2.6. カスタムマクロの作成

~/.rpmmacros ファイル内のディストリビューションマクロは、カスタムマクロで上書きできます。加えた変更は、マシン上のすべてのビルドに影響します。

警告

~/.rpmmacros ファイルで新しいマクロを定義することは推奨されません。このようなマクロは、ユーザーがパッケージを再構築する可能性がある他のマシンには存在しません。

手順

- マクロを上書きするには、次のコマンドを実行します。

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

上記の例から、**rpmde-setuptree** ユーティリティーを使用して、すべてのサブディレクトリーを含むディレクトリーを作成できます。このマクロの値は、デフォルトでは **~/rpmbuild** です。

```
%_smp_mflags -l3
```

上記のマクロは、Makefile に渡すためによく使用されます。たとえば、**make %{?_smp_mflags}** と、ビルドフェーズ時に多数の同時プロセスを設定します。デフォルトでは、**-jX** に設定されています。**X** は多数のコアです。コア数を変すると、パッケージビルドの速度アップまたはダウンを行うことができます。

5.3. EPOCH、SCRIPTLETS、TRIGGERS

このセクションでは、RMP SPEC ファイルの高度なディレクティブを表す **Epoch**、**Scriptlet**、**Triggers** について説明します。

これらのディレクティブはすべて、SPEC ファイルだけでなく、生成された RPM がインストールされているエンドマシンにも影響します。

5.3.1. Epoch ディレクティブ

Epoch ディレクティブでは、バージョン番号に基づいて加重依存関係を定義できます。

このディレクティブが RPM SPEC ファイルにない場合、**Epoch** ディレクティブは全く設定されません。これは、**Epoch** を設定しないと **Epoch** が 0 になるという一般的な考え方に反しています。ただし、**yum** ユーティリティーは、depsolve の目的で、0 の **Epoch** と同様に設定されていない **Epoch** を処理します。

ただし、SPEC ファイルでの **Epoch** の リストは通常省略されます。これは、多くの場合、**Epoch** 値を導入すると、パッケージのバージョンを比較する際に、想定される RPM 動作がスキューされるためです。

例5.2 Epoch の使用

> **Epoch: 1** および **Version: 1.0** で **foobar** パッケージをインストールし、他のユーザーが **Version 2.0** で **foobar** をパッケージ化します。ただし、**Epoch** ディレクティブがない場合、新しいバージョンは更新とはみなされません。RPM パッケージ用のバージョン管理を示す従来の **Name-Version-Release** ラッパーよりも、**Epoch** バージョンが推奨されている理由。

Epoch を使用することはほとんどありません。ただし、**Epoch** は 、通常、アップグレードの順序の問題を解決するために使用されます。この問題は、ソフトウェアバージョン番号のスキームや、エンコードに基づいて確実に比較できないアルファベット文字を組み込んだバージョンにおける、アップストリームによる変更の副次的効果として見られる場合があります。

5.3.2. Scriptlets ディレクティブ

Scriptlets は、パッケージがインストールまたは削除される前または後に実行される一連の RPM ディレクティブです。

Scriptlets は、ビルド時またはスタートアップスクリプト内で実行できないタスクにのみ使用します。

共通の Scriptlet ディレクティブのセットがあります。これは、SPEC ファイルセクションのヘッダー (%build、**%install** など) と似ています。これは、標準の POSIX シェルスクリプトとしてよく書かれる、マルチラインのコードセグメントによって定義されます。ただし、ターゲットマシンのディストリビューションの RPM が対応する他のプログラミング言語で書くこともできます。RPM ドキュメントには、利用可能な言語の完全なリストが含まれます。

以下の表には、実行順の Scriptlet ディレクティブのリストが含まれます。スクリプトを含むパッケージは、**%pre** と **%post** ディレクティブの間にインストールされ、**%preun** ディレクティブと **%postun** ディレクティブ間でアンインストールされることに注意してください。

表5.2 Scriptlet ディレクティブ

ディレクティブ	定義
%pretrans	パッケージのインストールまたは削除の直前に実行されるスクリプトレット。
%pre	ターゲットシステムにパッケージをインストールする直前に実行されるスクリプトレット。
%post	ターゲットシステムにパッケージがインストールされた直後に実行されるスクリプトレット。
%preun	ターゲットシステムからパッケージをアンインストールする直前に実行されるスクリプトレット。
%postun	ターゲットシステムからパッケージをアンインストールした直後に実行されるスクリプトレット。
%posttrans	トランザクションの最後に実行されるスクリプトレット。

5.3.3. スクリプトレット実行の無効化

以下の手順では、**rpm** コマンドと **--no_scriptlet_name_** オプションを使用して、スクリプトレットの実行を無効にする方法を説明します。

手順

- たとえば、**%pretrans** スクリプトレットの実行を無効にするには、次のコマンドを実行します。

  ```
  # rpm --nopretrans
  ```

  -- noscripts オプションも使用できます。これは、以下のすべてと同等になります。

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

関連情報

- **rpm(8)** man ページ

5.3.4. スクリプトレットマクロ

Scriptlets ディレクティブは、RPM マクロでも機能します。

以下の例は、systemd スクリプトレットマクロの使用を示しています。これにより、systemd は新しいユニットファイルについて通知されるようになります。

```
$ 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 ディレクティブ

Triggers は、パッケージのインストールおよびアンインストール時に対話できる手段を提供する RPM ディレクティブです。

警告

Triggers は、含まれるパッケージの更新など、予期できないタイミングで実行できます。**Triggers** はデバッグが難しいため、予期せず実行されたときに破損しないように、安定したな方法で実装する必要があります。このため、Red Hat では、**Trigger** の使用は最小限に抑えることを推奨します。

1 つのパッケージアップグレードの実行順序と、既存の各 **Triggers** の詳細は、以下のとおりです。

```
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
```

上記の項目は、**/usr/share/doc/rpm-4.*/triggers** ファイルにあります。

5.3.6. SPEC ファイルでのシェルスクリプト以外のスクリプトの使用

SPEC ファイルの **-p** スクリプトレットオプションを指定すると、ユーザーはデフォルトのシェルスクリプトインタープリター (**-p /bin/sh**) の代わりに特定のインタープリターを起動することができます。

次の手順では、**pello.py** プログラムのインストール後にメッセージを出力するスクリプトの作成方法を説明します。

手順

1. **pello.spec** ファイルを開きます。
2. 以下の行を見つけます。

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

3. 上記の行の下に、以下を挿入します。

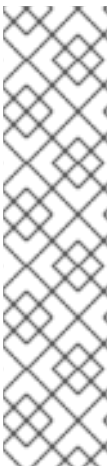
```
%post -p /usr/bin/python3
print("This is {} code".format("python"))
```

4. RPM のビルド の説明に従ってパッケージをビルドします。
5. パッケージをインストールします。

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

6. インストール後に出力メッセージを確認します。

```
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
```



注記

Python 3 スクリプトを使用するには、SPEC ファイルの **install -m** に次の行を含めます。

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

Lua スクリプトを使用するには、SPEC ファイルの **install -m** に次の行を含めます。

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

これにより、SPEC ファイル内で任意のインタープリターを指定できます。

5.4. RPM 条件

RPM 条件により、さまざまなバージョンの SPEC ファイルを条件付きで含めることができます。

条件を含めるには通常、次を処理します。

- アーキテクチャー固有のセクション
- オペレーティングシステム固有のセクション
- さまざまなバージョンのオペレーティング間の互換性の問題
- マクロの存在と定義

5.4.1. RPM 条件構文

RPM 条件では、次の構文を使用します。

expression が真であれば、以下のアクションを実行します。

```
%if expression
…
%endif
```

expression が真であれば、別のアクションを実行し、別の場合には別のアクションを実行します。

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

5.4.2. %if 条件

次の例は、**%if** RPM 条件の使用法を示しています。

例5.3 Red Hat Enterprise Linux 8 と他のオペレーティングシステム間の互換性を処理するために %if を使用

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

この条件では、AS_FUNCTION_DESCRIBE マクロのサポート上、RHEL 8 と他のオペレーティングシステム間の互換性が処理されます。パッケージが RHEL 用にビルドされている場合、**%rhel** マクロが定義され、RHEL バージョンに展開されます。値が 8 の場合、パッケージは RHEL 8 用にビルドされ、RHEL 8 で対応していない AS_FUNCTION_DESCRIBE への参照が autoconfig スクリプトから削除されます。

例5.4 %if 条件を使用したマクロの定義の処理

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

この条件では、マクロの定義を処理します。**%milestone** マクロまたは **%revision** マクロが設定されている場合は、アップストリームの tarball の名前を定義する **%ruby_archive** マクロが再定義されます。

5.4.3. %if 条件の特殊なバリアント

%ifarch 条件、**%ifnarch** 条件、**%ifos** 条件は、**%if** 条件の特殊なバリアントです。これらのバリアントは一般的に使用されるため、独自のマクロがあります。

%ifarch 条件

%ifarch 条件は、アーキテクチャー固有の SPEC ファイルのブロックを開始するために使用されます。この後に、アーキテクチャー指定子が続きます。これらは、それぞれコンマまたは空白で区切ります。

例5.5 %ifarch 条件の使用例

```
%ifarch i386 sparc
…
%endif
```

%ifarch と **%endif** の間にある SPEC ファイルのすべてのコンテンツは、32 ビット AMD および Intel のアーキテクチャー、または SunMAJOROS ベースのシステムでのみ処理されます。

%ifnarch 条件

% ifnarch 条件には、**%ifarch** 条件よりもリバース論理があります。

例5.6 %ifnarch 条件の使用例

```
%ifnarch alpha
…
%endif
```

SPEC ファイルの **% ifnarch** と **% endif** との間のすべてのコンテンツは、Digital Alpha/AXP ベースのシステムで処理されない場合に限り処理されます。

%ifos 条件

%ifos 条件は、ビルドのオペレーティングシステムに基づいて処理を制御するために使用されます。その後に複数のオペレーティングシステム名を指定できます。

例5.7 %ifos 条件の使用例

```
%ifos linux
…
%endif
```

SPEC ファイルの **%ifos** と **%endif** と の間のすべてのコンテンツは、ビルドが Linux システムで実行された場合にのみ処理されます。

5.5. PYTHON 3 RPM のパッケージ化

ほとんどの Python プロジェクトは、パッケージ化に Setuptools を使用して、**setup.py** ファイルにパッケージ情報を定義します。Setuptools パッケージ化の詳細は Setuptools ドキュメント を参照してください。

Python プロジェクトを RPM パッケージにパッケージ化することもできます。これには、Setuptools パッケージ化と比較して以下の利点があります。

- その他の RPM のパッケージの依存関係の指定 (Python 以外も含む)
- 電子署名
  電子署名を使用すると、RPM パッケージの内容は、オペレーティングシステムのその他の部分とともに検証、統合、およびテストできます。

5.5.1. Python パッケージ用の SPEC ファイルの説明

SPEC ファイルには、RPM のビルドに **rpmbuild** ユーティリティーを使用する命令が含まれています。命令は、一連のセクションに含まれています。SPEC ファイルには、セクションが定義されている 2 つの主要部分があります。

- プリアンブル (ボディーに使用されている一連のメタデータ項目が含まれています)
- ボディー (命令の主要部分が含まれています)

Python プロジェクトの RPM SPEC ファイルには、非 Python RPM SPEC ファイルと比較していくつかの詳細があります。特に注目すべきは、Python ライブラリーの RPM パッケージ名に、Python 3.6 の場合は **python3**、Python 3.8 の場合は **python38**、Python 3.9 の場合は **python39**、Python 3.11 の場合は **python3.11** など、バージョンを判別する接頭辞を常に含める必要があります。

その他の詳細は、次の SPEC ファイルの **python3-detox パッケージの例** に記載されています。その詳細の説明は、例の下に記載されている注意事項を参照してください。

```
%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** マクロには、Python プロジェクトの名前が含まれます。この例では **detox** となります。

2. Python プロジェクトを RPM にパッケージ化する場合は、常にプロジェクトの元の名前に接頭辞 **python3** を追加する必要があります。ここでの元の名前は **detox** で、**RPM の名前** は **python3-detox** です。

3. **BuildRequires** は、このパッケージのビルドおよびテストに必要なパッケージを指定します。BuildRequires では、Python パッケージをビルドするのに必要なツールを提供する項目 (**python36-devel** および **python3-setuptools**) が常に含まれます。**/usr/bin/python3** インタープリターディレクティブを持つファイルが自動的に **/usr/bin/python3.6** に変更されるように、**python36-rpm-macros** パッケージが必要です。

4. すべての Python パッケージが正しく動作するためには、その他のパッケージがいくつか必要です。このようなパッケージも、SPEC ファイルで指定する必要があります。**依存関係**を指定するには、**%python_enable_dependency_generator** マクロを使用して、**setup.py** ファイルに定義した依存関係を自動的に使用できます。パッケージに、Setuptools で指定していない依存関係がある場合は、追加の **Requires** ディレクティブ内に指定します。

5. **%py3_build** マクロおよび **%py3_install** マクロは、**setup.py build** コマンドおよび **setup.py install** コマンドを実行します。それぞれには、インストール場所、使用するインタープリター、その他の詳細を指定する引数を用います。

その他の詳細を指定する引数を用います。

6 **check** セクションは、Python の正しいバージョンを実行するマクロを提供します。**%{__python3}** マクロには、Python 3 インタープリターのパス (**/usr/bin/python3** など) が含まれます。リテラルパスではなく、マクロを使用することが常に推奨されます。

5.5.2. Python 3 RPM の一般的なマクロ

SPEC ファイルでは、値をハードコーディングするのではなく、以下の **Python 3 RPM のマクロ** の表で説明されているマクロを常に使用します。

マクロ名では、バージョンを指定しない **python** ではなく、**python3** または **python2** を使用してください。SPEC ファイルの BuildRequires セクションで特定の Python 3 バージョンを python36-rpm-macros、python38-rpm-macros、**python39-rpm-macros**、または **python3.11-rpm-macros** に設定します。

表5.3 Python 3 RPM 用のマクロ

マクロ	一般的な定義	説明
%{__python3}	/usr/bin/python3	Python 3 のインタープリター
%{python3_version}	3.6	Python 3 インタープリターのフルバージョン
%{python3_sitelib}	/usr/lib/python3.6/site-packages	pure-Python モジュールのインストール先
%{python3_sitearch}	/usr/lib64/python3.6/site-packages	アーキテクチャー固有の拡張を含むモジュールがインストールされている場合
%py3_build		システムパッケージに適した引数で **setup.py build** コマンドを実行します。
%py3_install		システムパッケージに適した引数で **setup.py install** コマンドを実行します。

5.5.3. Python RPM の自動 Provides

Python プロジェクトをパッケージ化する際、以下のディレクトリーが存在する場合は、作成される RPM に以下のディレクトリーが含まれていることを確認してください。

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

このディレクトリーから、RPM ビルドプロセスは自動的に仮想 **pythonX.Ydist** Provides

(**python3.6dist(detox)** など) を生成します。この仮想 Provides は、**%python_enable_dependency_generator** マクロにより指定されるパッケージにより提供されます。

5.6. PYTHON スクリプトでインタープリターディレクティブの処理

Red Hat Enterprise Linux 8 では、実行可能な Python スクリプトは、少なくとも主要な Python バージョンを明示的に指定するインタープリターディレクティブ (別名 hashbangs または shebangs) を使用することが想定されます。以下に例を示します。

```
#!/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
```

/usr/lib/rpm/redhat/brp-mangle-shebangs BRP (buildroot policy) スクリプトは、RPM パッケージをビルドする際に自動的に実行され、実行可能なすべてのファイルでインタープリターディレクティブを修正しようとします。

BRP スクリプトは、以下のようにあいまいなインタープリターディレクティブを含む Python スクリプトを検出すると、エラーを生成します。

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

または

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

5.6.1. Python スクリプトでインタープリターディレクティブの変更

RPM ビルド時にビルドエラーが発生する Python スクリプト内のインタープリターディレクティブを変更します。

前提条件

- Python スクリプトのインタープリターディレクティブの一部でビルドエラーが発生する。

手順

インタープリターディレクティブを変更するには、以下のタスクのいずれかを実行します。

- **platform-python-devel** パッケージから **pathfix.py** スクリプトを適用します。

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

  複数の **PATH** を指定できます。**PATH** がディレクトリーの場合、**pathfix.py** はあいまいなインタープリターディレクティブを持つスクリプトだけでなく、**^[a-zA-Z0-9_]+\.py$** のパターンに一致する Python スクリプトを再帰的にスキャンします。このコマンドを **%prep** セクション、または **%install** セクションに追加します。

- パッケージ化した Python スクリプトを、想定される形式に準拠するように変更します。この

目的のために、**pathfix.py** は、RPM ビルドプロセス以外でも使用できます。**pathfix.py** を RPM ビルド以外で実行する場合は、上記の例の **%{__python3}** を、**/usr/bin/python3** などのインタープリターディレクティブのパスに置き換えます。

パッケージ化された Python スクリプトに Python 3.6 以外のバージョンが必要な場合は、上記のコマンドを調整して必要なバージョンを含めます。

5.6.2. カスタムパッケージの /usr/bin/python3 インタープリターディレクティブの変更

デフォルトでは、**/usr/bin/python3** の形式でのインタープリターディレクティブは、Red Hat Enterprise Linux のシステムツールに使用される **platform-python** パッケージから Python を参照するインタープリターディレクティブに置き換えられます。カスタムパッケージの **/usr/bin/python3** インタープリターディレクティブを変更して、AppStream リポジトリーからインストールした特定バージョンの Python を参照できます。

手順

- Python の特定バージョンのパッケージを構築するには、対応する **python** パッケージの **python*-rpm-macros** サブパッケージを SPEC ファイルの **BuildRequires** セクションに追加します。たとえば、Python 3.6 の場合は、以下の行を追加します。

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

これにより、カスタムパッケージの **/usr/bin/python3** インタープリターディレクティブは、自動的に **/usr/bin/python3.6** に変換されます。

注記

BRP スクリプトがインタープリターディレクティブを確認したり、変更したりしないようにするには、以下の RPM ディレクティブを使用します。

```
%undefine __brp_mangle_shebangs
```

5.7. RUBYGEMS パッケージ

本セクションでは、RubyGems パッケージの概要と、RPM への再パッケージ化方法を説明します。

5.7.1. RubyGems の概要

Ruby は、ダイナミックなインタープリター言語で、反映的なオブジェクト指向の汎用プログラミング言語です。

Ruby で書かれたプログラムは、特定の Ruby パッケージ形式を提供する RubyGems プロジェクトを使用してパッケージ化されます。

RubyGems で作成したパッケージは gems と呼ばれ、RPM に再パッケージ化することもできます。

注記

本書は、**gem** 接頭辞とともに RubyGems の概念に関する用語を参照します。たとえば、.gemspec は **gem specification** に使用され、RPM に関連する用語は非修飾になります。

5.7.2. RubyGems が RPM に関連している仕組み

RubyGems は、Ruby 独自のパッケージ形式を表します。ただし、RubyGems には RPM が必要とするメタデータと同様のものが含まれ、RubyGems から RPM への変換が可能になります。

Ruby Packaging Guidelines では、以下の方法で RubyGems パッケージを RPM に再パッケージ化できます。

- このような RPM は、残りすべてのディストリビューションに適合します。
- RPM パッケージ化された正しい gem をインストールすると、エンドユーザーで gem の依存関係を満たすことができます。

RubyGems は、SPEC ファイル、パッケージ名、依存関係などの RPM と同様の用語を使用します。

残りの RHEL RPM ディストリビューションに合わせるには、RubyGems で作成したパッケージが以下の規則に従う必要があります。

- gems の名前は以下のパターンに従います。

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

- シバンの行を実装するには、以下の文字列を使用する必要があります。

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

5.7.3. RubyGems パッケージからの RPM パッケージの作成

RubyGems パッケージのソース RPM を作成するには、以下のファイルが必要です。

- gem ファイル
- RPM SPEC ファイル

次のセクションでは、RubyGems が作成したパッケージから RPM パッケージを作成する方法を説明します。

5.7.3.1. RubyGems SPEC ファイル規則

RubyGems SPEC ファイルは、以下の規則を満たす必要があります。

- gem の仕様の名前である **%{gem_name}** の定義が含まれる。
- パッケージのソースは、リリースされた gem アーカイブの完全な URL であること。パッケージのバージョンは、gem のバージョンであること。
- ビルドに必要なマクロをプルできるように、以下のように定義された **BuildRequires:** ディレクティブが含まれる。

```
BuildRequires:rubygems-devel
```

- RubyGems **Requires** または **Provides** は自動生成されるため、含まれません。
- Ruby バージョンの互換性を明示的に指定しない限り、以下のように定義された **BuildRequires:** ディレクティブは含まれません。

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

RubyGems で自動生成された依存関係 (**Requires:ruby (rubygems)**) で十分です。

5.7.3.2. RubyGems マクロ

以下の表は、RubyGems で作成したパッケージで役に立つマクロをリスト表示します。これらのマクロは、**rubygems-devel** パッケージで提供されています。

表5.4 RubyGems マクロ

マクロ名	拡張パス	用途
%{gem_dir}	/usr/share/gems	gem 構造のトップディレクトリー。
%{gem_instdir}	%{gem_dir}/gems/%{gem_name}-%{version}	gem の実際のコンテンツが含まれるディレクトリー。
%{gem_libdir}	%{gem_instdir}/lib	gem のライブラリーディレクトリー。
%{gem_cache}	%{gem_dir}/cache/%{gem_name}-%{version}.gem	キャッシュした gem。
%{gem_spec}	%{gem_dir}/specifications/%{gem_name}-%{version}.gemspec	gem 仕様ファイル。
%{gem_docdir}	%{gem_dir}/doc/%{gem_name}-%{version}	gem の RDoc ドキュメンテーション。
%{gem_extdir_mri}	%{_libdir}/gems/ruby/%{gem_name}-%{version}	gem 拡張のディレクトリー。

5.7.3.3. RubyGems SPEC ファイルの例

gem を構築するための SPEC ファイルの例と、その特定のセクションの説明を次に示します。

RubyGems SPEC ファイルの例

```
%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}/
```

次の表は、RubyGems SPEC ファイルの特定項目の詳細を説明します。

表5.5 RubyGems' SPEC ディレクティブの詳細

ディレクティブ	RubyGems の詳細
%prep	RPM は gem アーカイブを直接デプロイメントできるため、**gem unpack** コマンドを実行して gem からソースを抽出できます。**%setup -n %{gem_name}-%{version}** マクロは、gem がデプロイメントされたディレクトリーを提供します。同じディレクトリーレベルでは、**%{gem_name}-%{version}.gemspec** ファイルが自動的に作成されます。このファイルは、後で gem を再構築したり、**.gemspec** を変更したり、コードにパッチを適用したりするために使用されます。
%build	このディレクティブには、ソフトウェアをマシンコードに構築するためのコマンドまたは一連のコマンドが含まれます。**%gem_install** マクロは gem アーカイブでのみ動作し、gem は次の gem ビルドで再作成されます。作成した gem ファイルは、**%gem_install** により使用され、一時ディレクトリー (**デフォルトでは /%{gem_dir}**) にコードを構築してインストールします。**%gem_install** マクロは両者とも、コードを 1 つのステップで構築してインストールします。ビルドしたソースはインストール前に、自動的に作成される一時ディレクトリーに配置されます。 **%gem_install** マクロは、2 つの追加オプションを受け付けます。そのうちの 1 つは**-n <gem_file>** で、インストールに使用される gem を上書きできます。もうひとつは、**- d <install_dir>** で、gem インストール先を上書きできます。なお、このオプションの使用は推奨されません。 **%gem_install** マクロは、**%{buildroot}** へのインストールに使用することはできません。

ディレクティブ	RubyGems の詳細
%install	インストールは、**%{buildroot}** 階層で実行されます。必要なディレクトリーを作成し、一時ディレクトリーにインストールされているものを、**%{buildroot}** 階層にコピーできます。この gem が共有オブジェクトを作成すると、これらはアーキテクチャー固有の **%{gem_extdir_MRI}** パスに移動されます。

関連情報

- Ruby パッケージ化のガイドライン

5.7.3.4. gem2rpm を使用した RubyGems パッケージの RPM SPEC ファイルへの変換

gem2rpm ユーティリティーは、RubyGems パッケージを RPM SPEC ファイルに変換します。

以下のセクションでは、次の方法を説明します。

- **gem2rpm** ユーティリティーのインストール
- すべての **gem2rpm** オプションの表示
- **gem2rpm** を使用して RubyGems パッケージを RPM SPEC ファイルへ変換する
- **gem2rpm** テンプレートの変更

5.7.3.4.1. GFS2 のインストール

以下の手順では、**gem2rpm** ユーティリティーのインストール方法を説明します。

手順

- RubyGems.org から **gem2rpm** にインストールするには、以下のコマンドを実行します。

```
$ gem install gem2rpm
```

5.7.3.4.2. gem2rpm のすべてのオプションの表示

以下の手順では、**gem2rpm** ユーティリティーのすべてのオプションを表示する方法を説明します。

手順

- **gem2rpm** のすべてのオプションを表示するには、以下を実行してください。

```
$ gem2rpm --help
```

5.7.3.4.3. gem2rpm を使用して RubyGems パッケージを RPM SPEC ファイルへ変換

以下の手順では、**gem2rpm** ユーティリティーを使用して、RubyGems パッケージを RPM SPEC ファイルに変換する方法を説明します。

手順

- 最新バージョンの gem ダウンロードし、この gem 用の RPM SPEC ファイルを生成します。

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

説明した手順では、gem のメタデータの情報に基づいて RPM SPEC ファイルを作成します。ただし、gem は、通常 RPM (ライセンスや変更ログなど) で提供される重要な情報に欠けています。したがって、生成された SPEC ファイルを編集する必要があります。

5.7.3.4.4. gem2rpm テンプレート

gem2rpm テンプレートとは、次の表に示す変数を含む標準の埋め込み Ruby (ERB) ファイルです。

表5.6 gem2rpm テンプレート内の変数

変数	説明
package	gem の **Gem::Package** 変数。
spec	gem の **Gem ::Specification** 変数 (format.spec と同じ)。
config	仕様のテンプレートヘルパーで使用されるデフォルトのマクロまたはルールを再定義できる **Gem 2RPM::Configuration** 変数。
runtime_dependencies	パッケージランタイム依存関係のリストを示す **Gem2RPM::RpmDependencyList** 変数。
development_dependencies	パッケージ開発依存関係のリストを示す **Gem2RPM::RpmDependencyList** 変数。
テスト	**Gem 2RPM::testsuite** 変数は、実行を許可するテストフレームワークのリストを示します。
files	パッケージ内のファイルにフィルターが適用されていないリストを示す **Gem 2RPM::RpmFileList** 変数。
main_files	メインパッケージに適したファイルのリストを提供する **Gem2RPM::RpmFileList** 変数。
doc_files	**-doc** サブパッケージに適したファイルのリストを提供する **Gem 2RPM::RpmFileList** 変数。
format	gem の **Gem::Format** 変数。この変数は現在非推奨になっています。

5.7.3.4.5. 利用可能な gem2rpm テンプレートのリスト表示

以下の手順では、利用可能な **gem2rpm** テンプレートのリストを表示する方法を説明します。

手順

- 利用可能なテンプレートをすべて表示するには、以下を実行します。

```
$ gem2rpm --templates
```

5.7.3.4.6. gem2rpm テンプレートの編集

生成された SPEC ファイルを編集する代わりに、RPM SPEC ファイルの生成元となるテンプレートを編集できます。

gem2rpm のテンプレートを変更する場合は、以下の手順を行います。

手順

1. デフォルトのテンプレートを保存します。

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

2. 必要に応じてテンプレートを編集します。
3. 編集したテンプレートを使用して SPEC ファイルを生成します。

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

これで、RPM のビルド の説明に従って、編集したテンプレートを使用して RPM パッケージをビルドできるようになりました。

5.8. PERL スクリプトで RPM パッケージを処理する方法

RHEL 8 以降、Perl プログラミング言語はデフォルトの buildroot に含まれていません。そのため、Perl スクリプトを含む RPM パッケージは、RPM SPEC ファイルの **BuildRequires:** ディレクティブを使用して、Perl の依存関係を明示的に指定する必要があります。

5.8.1. 一般的な Perl 関連の依存関係

BuildRequires: で使用される Perl 関連のビルドの最も頻繁に発生する依存関係は、以下の通りです。

- **perl-generators**
  インストールした Perl ファイルのランタイム **Requires** と **Provides** を自動的に生成します。Perl スクリプトまたは Perl モジュールをインストールする場合は、このパッケージにビルドの依存関係を含める必要があります。

- **perl-interpreter**
  Perl インタープリターは、**perl** パッケージまたは **%__perl** マクロから明示的に呼び出されるか、パッケージのビルドシステムの一部としてビルド依存関係として記載する必要があります。

- **perl-devel**
  Perl ヘッダーファイルを提供します。XS Perl モジュールなどの **libperl.so** ライブラリーにリンクしているアーキテクチャー固有のコードを構築する場合は、**BuildRequires: perl-devel** を含める必要があります。

5.8.2. 特定の Perl モジュールの使用

特定の Perl モジュールがビルド時に必要な場合は、以下の手順に従います。

手順

- RPM SPEC ファイルに以下の構文を適用します。

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

注記

この構文は Perl コアモジュールにも適用します。これは、**perl** パッケージを同時に移動し、タイムアウトするためです。

5.8.3. 特定の Perl バージョンへのパッケージの制限

パッケージを特定の Perl バージョンに限定するには、以下の手順に従います。

手順

- RPM SPEC ファイルの希望のバージョン制約で **perl (:VERSION)** 依存関係を使用します。たとえば、パッケージを Perl バージョン 5.22 以降に制限するには、以下を使用します。

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

警告

perl パッケージのバージョンには、エポック番号が含まれるため、バージョンに対する比較は行わないでください。

5.8.4. パッケージが正しい Perl インタープリターを使用することを確認

Red Hat は、完全に互換性がない複数の Perl インタープリターを提供しています。そのため、Perl モジュールを提供するすべてのパッケージは、ビルド時に使用されたものと同じ Perl インタープリターをランタイムで使用する必要があります。

これを確認するには、以下の手順に従います。

手順

- Perl モジュールを提供するすべてのパッケージについては、バージョン化された **MODULE_compat Requires** を RPM SPEC ファイルに含めます。

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

第6章 RHEL 8 の新機能

本セクションでは、Red Hat Enterprise Linux 7 と 8 における RPM パッケージ化の主な変更点を説明します。

6.1. WEAK 依存関係のサポート

Weak 依存関係 は、**Requires** ディレクティブのバリアントです。これらのバリアントは、**Epoch-Version-Release** 範囲比較で仮想 **Provides:** とパッケージ名に対して一致します。

Weak 依存関係 には、以下の表でまとめているように、2 つの強み (**weak** と **hint**) と 2 つの方向 (**forward** と **backward**) があります。

注記

forward 方向は **Requires:** に類似しています。**backward** には、以前の依存関係システムには類似していません。

表6.1 Weak の依存関係の強みと方向の組み合わせが可能

強み/方向	Forward	Backward
Weak	推奨:	補助:
Hint	提案:	強化:

Weak 依存関係 ポリシーの主な利点は以下のとおりです。

- デフォルトのインストール機能の豊富さを維持しつつ、最小限のインストールが可能です。
- パッケージは、仮想の柔軟性を維持しながら、特定のプロバイダーの設定を指定できます。

6.1.1. Weak 依存関係の概要

デフォルトでは、**Weak 依存関係** は、通常の **Requires:** と同様に扱われます。一致するパッケージが YUM トランザクションに含まれます。パッケージの追加でとエラーが発生する場合、デフォルトでは YUM は依存関係を無視します。そのため、ユーザーは **Weak 依存関係** により追加されるパッケージを除外したり、後で削除したりできます。

使用の条件

Weak 依存関係 は、パッケージが依然として依存関係なしで機能している場合に限り使用できます。

注記

Weak の要件を追加することなく、非常に限定的な機能を持つパッケージを作成できます。

使用例

Weak の依存関係 は特に、目的が単一で、パッケージの全機能セットを必要としない仮想マシンやコンテナーを構築するなど、合理的なユースケースのためにインストールを最小限に抑えることができる場合に使用します。

Weak 依存関係 の一般的なユースケースは、以下のとおりです。

- ドキュメント
  - ドキュメントビューアーがない場合でも正常に処理されるドキュメントビューアー
- 例
- プラグインまたはアドオン
  - 対応ファイル形式
  - 対応プロトコル

6.1.2. Hints の強度

Hints はデフォルトで、**YUM** で無視されます。これは、GUI ツールで使用することで、デフォルトでインストールされていないアドオンパッケージを利用できます。ただし、インストールされているパッケージと組み合わせることで役に立ちます。

パッケージの主なユースケースの要件には、**Hints** を使用しないでください。代わりに、このような要件を強固な依存関係または **Weak 依存関係** に含めます。

パッケージ設定

YUM は **Weak の依存関係** と **Hints** を使用して、複数の同等に有効なパッケージ間の選択肢がある場合に使用するパッケージを決定します。インストールされているパッケージまたはインストールされるパッケージの依存関係で指示されるパッケージが推奨されます。

依存関係の解決に関する通常のルールは、この機能の影響を受けないことに注意してください。たとえば、**Weak 依存関係** は、古いバージョンのパッケージが強制的に選ばれるようにすることはできません。

依存関係に複数のプロバイダーがある場合は、必須となるパッケージで **Suggests:** を追加して、どのオプションが優先されるかについて依存関係リゾルバーにヒントを提供することができます。

メインパッケージおよび他のプロバイダーが、必要なパッケージにヒントを追加することが、より簡素化されたソリューションであるということに同意する場合にのみ **Enhances:** が使用されます。

例6.1 Hints を使用した、ある特定のパッケージの優先

```
Package A: Requires: mysql

Package mariadb: Provides: mysql

Package community-mysql: Provides: mysql
```

community-mysql パッケージよりも **mariadb** パッケージを優先する場合は、以下を使用します。

```
Suggests: mariadb to Package A.
```

6.1.3. Forwared と Backward の依存関係

Forward 依存関係 は **Requires** と同様に、インストールするパッケージに対して評価されます。最も一致するパッケージもインストールされます。

一般的には、**Forward 依存関係** が優先されます。システムに追加された別のパッケージを取得する場合は、この依存関係をパッケージに追加します。

Backward 依存関係 の場合、一致するパッケージもインストールされていると、その依存関係を含むパッケージがインストールされます。

Backward 依存関係 は主に、そのプラグイン、アドオン、エクステンション機能をディストリビューションやその他サードパーティーパッケージにアタッチできるサードパーティーベンダー向けに設計されています。

6.2. ブール型依存関係のサポート

バージョン 4.13 以降では、RPM は以下の依存関係でブール式を処理できます。

- **Requires**
- **Recommends**
- **Suggests**
- **supplements**
- **Enhances**
- **Conflicts**

このセクションでは、ブール値の依存関係構文 を説明し、ブール値演算子 のリストを紹介して、ブール値の依存関係のセマンティクス およびブール値の 依存関係のセマンティクス について説明します。

6.2.1. ブール値の依存関係構文

ブール値は、常に括弧で囲まれています。

これは、通常の依存関係から構築されます。

- 名前のみまたは名前
- 比較
- バージョンの説明

6.2.2. ブール値の演算子

RPM 4.13 では、以下のブール値演算子が導入されました。

表6.2 RPM 4.13 で導入されたブール値演算子

ブール値演算子	説明	使用例

ブール値演算子	説明	使用例
and	用語が真となるには、すべてのオペランドを満たす必要があります。	Conflicts: (pkgA and pkgB)
または	用語が真となるには、いずれかのオペランドを満たす必要があります。	Requires: (pkgA >= 3.2 or pkgB)
if	第 2 のオペランドが満たされる場合、第 1 オペランドも満たされる必要があります (リバースインプリケーション)	Recommends: (myPkg-langCZ if langsupportCZ)
if else	**if** 演算子と同じで、2 番目のオペランドが一致しない場合は、第 3 オペランドが満たされる必要があります。	Requires: myPkg-backend-mariaDB if mariaDB else sqlite

RPM 4.14 では、以下のブール値演算子がさらに導入されています。

表6.3 RPM 4.14 で導入されたブール値演算子

ブール値演算子	説明	使用例
with	用語が真となるには、すべてのオペランドが同じパッケージで満たされる必要があります。	Requires: (pkgA-foo with pkgA-bar)
without	最初のオペランドを満たすが、2 番目のオペランドを満たさない単一のパッケージが必要	Requires: (pkgA-foo without pkgA-bar)
unless	2 番目のオペランドが満たされない場合、最初のオペランドを満たす必要があります (リバースネガティブインプリケーション)。	Conflicts: (myPkg-driverA unless driverB)
unless else	**unless** 演算子と同じで、2 番目のオペランドが一致しない場合は、第 3 オペランドが満たされる必要があります。	Conflicts: (myPkg-backend-SDL1 unless myPkg-backend-SDL2 else SDL2)

重要

if 演算子と **or** 演算子は同じコンテキストで使用できず、**unless** 演算子は **and** と同じコンテキストで使用できません。

6.2.3. ネスト化

以下の例のように、オペランド自体をブール式として使用できます。

このような場合は、オペランドを括弧で囲む必要もあります。**and** と **or** 演算子を組み合わせて、括弧で囲った同じ 1 セットと同じ演算子を繰り返すことができます。

例6.2 ブール式として適用されるオペランドの使用例

```
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. セマンティクス

ブール値の依存関係 を使用しても、通常の依存関係のセマンティックは変更されません。

ブール値の依存関係 が使用されている場合は、名前が一致するものをすべてチェックし、一致する名前のブール値をブール値演算子で集計します。

重要

Conflicts: を除くすべての依存関係では、インストールを防ぐために、結果が **True** である必要があります。**Conflicts:** については、インストールを防ぐために、結果が **False** である必要があります。

警告

Provides は依存関係ではないため、ブール式を含めることはできません。

6.2.5. if 演算子の出力の理解

if 演算子もブール値を返します。これは、直感的に理解しやすいものです。ただし、以下の例では、**if** の直感的な使用が誤解を招く可能性がある場合を示しています。

例6.3 if 演算子の出力の誤解

このステートメントは、pkgB がインストールされていない場合には真になります。ただし、このステートメントを、デフォルトの結果が偽であるところで使用すると、事が複雑になります。

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

このステートメントは、pkgB がインストールされていて、pkgA がインストールされていない場合に競合します。

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

そのため、以下を使用することが推奨されます。

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

if 演算子が **or** にネストされている場合も同様です。

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

pkgB がインストールされていない場合に **if** が真となるため、用語が完全に真となります。pkgB がインストールされている場合にのみ pkgA が役立つ場合は、代わりに **and** を使用します。

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

6.3. ファイルトリガーのサポート

File triggers は RPM スクリプトレット の一種です。

これらはパッケージの SPEC ファイルで定義されます。

Triggers と同様、これらはあるパッケージで宣言されますが、一致するファイルを含む別のパッケージをインストールまたは削除したときに実行されます。

ファイルトリガー の一般的な用途は、レジストリーまたはキャッシュを更新することです。このようなユースケースでは、レジストリーまたはキャッシュを含む、または管理するパッケージにも、単一または複数の **ファイルトリガー** が含まれている必要があります。**ファイルトリガー** を含めると、パッケージがそれ自体の更新を制御する場合と比べて時間を短縮できます。

6.3.1. ファイルトリガー構文

ファイルトリガー の構文は以下のとおりです。

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

詳細は以下のようになります。

file_trigger_tag は、ファイルトリガーのタイプを定義します。使用可能なタイプは次のとおりです。

- **filetriggerin**
- **filetriggerun**

- **filetriggerpostun**
- **transfiletriggerin**
- **transfiletriggerun**
- **transfiletriggerpostun**

FILE_TRIGGER_OPTIONS は、**-P** オプションを除き、RPM スクリプトレットオプションと同じ目的で使用されます。

トリガーの優先度は数字で定義されます。この数字が大きいほど、ファイルトリガースクリプトの実行優先度が高くなります。優先度が 100000 を超えるトリガーは、標準のスクリプトレットの前に実行され、その他のトリガーは標準のスクリプトレットの後に実行されます。デフォルトの優先度は 1000000 に設定されています。

各タイプのすべてのファイルトリガーには、1 つ以上のパス接頭辞とスクリプトが含まれている必要があります。

6.3.2. ファイルトリガー構文の例

次の例は、**File triggers** の構文を示しています。

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

このファイルトリガーは、**/usr/lib** または **/lib** で始まるパスを含むパッケージのインストール後に、**/usr/bin/ldconfig** を直接実行します。**/usr/lib** または **/lib** で始まるパスを持つ複数のファイルがパッケージに含まれている場合でも、ファイルトリガーは 1 度のみ実行されます。ただし、**/usr/lib** または **/lib** で始まるファイル名はすべて、以下のようにスクリプト内でフィルタリングできるように、トリガースクリプトの標準入力に渡されます。

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

このファイルトリガーは、**/usr/lib** で始まり、**foo** を同時に含むファイルがある各パッケージに対して **/usr/bin/ldconfig** を実行します。接頭辞に一致するファイルには、通常のファイル、ディレクトリー、シンボリックリンクなど、すべての種類のファイルが含まれることに注意してください。

6.3.3. ファイルトリガータイプ

ファイルトリガー には、以下の 2 つの主要タイプがあります。

- パッケージごとに 1 回実行されるファイルトリガー
- トランザクションごとに 1 回実行されるファイルトリガー

ファイルトリガー は、以下のように、実行時間に基づいてさらに分割されます。

- パッケージのインストールまたは消去の前または後
- トランザクションの前または後

6.3.3.1. パッケージファイルトリガーごとの実行

パッケージごとに1回実行される **ファイルトリガー**:

- %filetriggerin
- %filetriggerun
- %filetriggerpostun

%filetriggerin
このファイルトリガーは、このトリガーの接頭辞に一致するファイルが1つ以上パッケージに含まれている場合にパッケージのインストール後に実行されます。また、これはこのファイルトリガーを含むパッケージのインストール後に実行されます。**rpmdb** データベースにこのファイルトリガーの接頭辞に一致するファイルが1つ以上存在します。

%filetriggerun
このトリガーの接頭辞に一致するファイルが1つ以上このパッケージに含まれている場合、このファイルトリガーはパッケージのアンインストールの前に実行されます。また、これはこのファイルトリガーを含むパッケージをアンインストールする前に実行されます。**rpmdb** には、このファイルトリガーの接頭辞に一致するファイルが1つ以上あります。

%filetriggerpostun
このファイルトリガーは、このトリガーの接頭辞に一致するファイルが1つ以上含まれている場合に、パッケージのアンインストール後に実行されます。

6.3.3.2. トランザクションファイルトリガーごとに1の回実行

トランザクションごとに1回実行される **ファイルトリガー**:

- %transfiletriggerin
- %transfiletriggerun
- %transfiletriggerpostun

%transfiletriggerin
このファイルトリガーはトランザクション後に、このトリガーの接頭辞に一致する1つ以上のファイルを含むすべてのインストール済みパッケージに対して実行されます。このトランザクションにこのファイルトリガーを含むパッケージがあり、**rpmdb** にこのトリガーの接頭辞に一致するファイルがつ以上ある場合は、トランザクションの後にも実行されます。

%transfiletriggerun
このファイルトリガーは、以下の条件を満たすすべてのパッケージのトランザクションの前に1度だけ実行されます。

- このトランザクションでパッケージがアンインストールされます。
- パッケージには、このトリガーの接頭辞に一致する1つ以上のファイルが含まれています。

このトランザクションにこのファイルトリガーを含むパッケージがあり、**rpmdb** にこのトリガーの接頭辞に一致するファイルが1つ以上ある場合は、トランザクションの前にも実行されます。

%transfiletriggerpostun
このファイルトリガーは、トランザクション後に、このトリガーの接頭辞に一致する1つ以上のファイルを含むすべてのアンインストール済みパッケージに対して実行されます。

注記

このトリガータイプでは、トリガーファイルのリストは利用できません。

そのため、ライブラリーを含む複数のパッケージをインストールまたはアンインストールすると、トランザクション全体の最後で ldconfig キャッシュが更新されます。これにより、キャッシュが各パッケージに対して個別に更新されていた RHEL 7 に比べて、パフォーマンスが大幅に改善されます。また、すべてのパッケージの SPEC ファイルで ldconfig と %postun を呼び出していたスクリプレットが必要ではなくなりました。

6.3.4. glibc でのファイルトリガーの使用例

次の例は、**glibc** パッケージ内での **File triggers** の実際の使用法を示しています。

RHEL 8 では、インストールまたはアンインストールトランザクションの最後に **ldconfig** コマンドを呼び出すために、**ファイルトリガー** が **glibc** に実装されています。

これは、**glibc** SPEC ファイルに以下のスクリプトレットを含めることで実現されました。

```
%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
```

そのため、複数のパッケージのインストールまたはアンインストールを行うと、トランザクション全体が終了してから、インストールしたすべてのライブラリーに対して **ldconfig** キャッシュが更新されます。そのため、個別のパッケージの RPM SPEC ファイルに ldconfig を呼び出すスクリプトレットを含める必要はありません。これにより、RHEL 7 に比べてパフォーマンスが改善され、各パッケージに対してキャッシュが別途更新されるようになりました。

6.4. より厳密な SPEC パーサー

SPEC パーサーに、いくつかの変更が行われました。したがって、以前は無視されていた新しい問題を特定できます。

6.5. 4 GB を超えるファイルのサポート

Red Hat Enterprise Linux 8 では、**RPM** は 64 ビットの変数とタグを使用できます。これにより、4 GB を超えるファイルやパッケージで動作可能になりました。

6.5.1. 64 ビット RPM タグ

64 ビットバージョンとそれ以前の 32 ビットバージョンには、複数の RPM タグがあります。64 ビットバージョンの名前の前には **LONG** 文字列があることに注意してください。

表6.4 32 ビットバージョンと 64 ビットバージョンの両方で利用可能な RPM タグ

32 ビットバリアントタグ名	62 ビットバリアントタグ名	タグ説明
RPMTAG_SIGSIZE	RPMTAG_LONGSIGSIZE	ヘッダーおよび圧縮ペイロードサイズ。

32 ビットバリアントタグ名	62 ビットバリアントタグ名	タグ説明
RPMTAG_ARCHIVESIZE	RPMTAG_LONGARCHIVESIZE	非圧縮ペイロードサイズ。
RPMTAG_FILESIZES	RPMTAG_LONGFILESIZES	ファイルサイズの配列。
RPMTAG_SIZE	RPMTAG_LONGSIZE	すべてのファイルサイズの合計。

6.5.2. コマンドラインでの 64 ビットタグの使用

LONG 拡張機能は、コマンドラインで常に有効になります。**rpm -q --QF** コマンドを含むスクリプトを以前使用していた場合は、これらのタグ名に **long** を追加できます。

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

6.6. その他の機能

Red Hat Enterprise Linux 8 の RPM のパッケージ化に関連するその他の新機能は、以下のとおりです。

- 非冗長モードで出力を確認する簡易署名
- 強制ペイロード検証のサポート
- 署名チェックの強制モードのサポート
- マクロの追加と廃止事項

関連情報

RPM、RPM のパッケージ化、RPM のビルドに関連するさまざまなトピックについては、以下を参照してください。これらの一部は高度なもので、本書に記載されている入門資料の発展となります。

Red Hat Software Collections Overview - Red Hat Software Collections は、最新の安定したバージョンで継続的に更新される開発ツールを提供します。

Red Hat Software Collections - Packaging Guide は、Software Collections の概要と、これらの構築およびパッケージする方法について説明しています。RPM を使用したソフトウェアパッケージングの基本知識がある開発者およびシステム管理者は、このガイドを使用して Software Collections を開始できます。

Mock - Mock は、さまざまなアーキテクチャー向けのコミュニティー対応パッケージビルドソリューションと、ビルドホストと異なる Fedora または RHEL バージョンを提供します。

RPM ドキュメント - 公式の RPM ドキュメント

Fedora Packaging Guidelines - Fedora の公式パッケージングガイドラインで、RPM ベースのすべてのディストリビューションに役に立ちます。