Red Hat Enterprise Linux 8

ソフトウェアのパッケージ化および配布

Red Hat Enterprise Linux 8 でのソフトウェアのパッケージ化と配布のガイド

Last Updated: 2022-09-07

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

Red Hat Enterprise Linux 8 でのソフトウェアのパッケージ化と配布のガイド

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.

法律上の通知

概要

本ガイドでは、ソフトウェアを RPM でパッケージ化する方法を説明します。また、パッケージ化用のソースコードを準備する方法と、いくつかのパッケージ作成の詳細なシナリオ (Python プロジェクトや RubyGems の RPM へのパッケージ化など) について説明します。

目次

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社の CTO、Chris Wright のメッセージ を参照してください。

RED HAT ドキュメントへのフィードバックの提供

弊社の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。ドキュメントの改善点があればお知らせください。

特定の文章へのコメント送信

1. **Multi-page HTML** 形式でドキュメントを表示し、ページが完全にロードされた後に右上隅の **Feedback** ボタンが表示されていることを確認します。
2. カーソルを使用して、コメントを追加するテキストの部分を強調表示します。
3. 強調表示されたテキストの近くに表示される **Add Feedback** ボタンをクリックします。
4. フィードバックを追加し、**Submit** をクリックします。

Bugzilla によるフィードバックの送信（アカウントが必要）

1. Bugzilla の Web サイトにログインします。
2. **Version** メニューから正しいバージョンを選択します。
3. **Summary** フィールドに説明的なタイトルを入力します。
4. **Description** フィールドに改善の提案を入力します。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. **Submit Bug** をクリックします。

第1章 RPM のパッケージ化の使用

RPM Package Manager (RPM) は、Red Hat Enterprise Linux、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 を使用することで、上記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は、従来のアーカイブファイルでのソフトウェア配布に比べて、いくつかの利点があります。

RPM を使用すると、以下が可能になります。

- YUM、PackageKit などの標準パッケージ管理ツールを使用した、パッケージのインストール、再インストール、削除、アップグレード、および検証。
- インストール済みのパッケージのデータベースを使用した、パッケージのクエリーおよび検証。
- メタデータを使用した、パッケージ、インストール手順、その他のパッケージパラメーターの記述。
- ソフトウェアソース、パッチ、完全なビルド命令の、ソースパッケージおよびバイナリーパッケージへのパッケージ化。
- Yum リポジトリーへのパッケージの追加。
- GNU Privacy Guard (GPG) 署名鍵を使用した、パッケージへのデジタル署名。

第2章 RPM パッケージ化を行うためのソフトウェアの準備

本セクションでは、RPM パッケージ化のソフトウェアを準備する方法を説明します。この準備には、プログラミングの知識は必要ありません。ただし、「ソースコードとは」、「プログラムが作られる仕組み」など、基本的な概念を理解しておく必要があります。

2.1. ソースコードとは

ここでは、ソースコードの概要を説明し、3 種類のプログラミング言語で書かれたプログラムのソースコード例を紹介します。

ソースコードとは、人間が読むことの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ています。ソースコードは、プログラミング言語で書かれています。

本書では、3 つのプログラミング言語で書かれた **Hello World** プログラムが紹介されています。

- bash で書かれた Hello World
- Python で書かれた Hello World
- C で書かれた Hello World

各言語で、パッケージ化が異なります。

各言語の **Hello World** プログラムは、RPM パッケージャーの主要な 3 つのユースケースをカバーしています。

例2.1 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 ファイルの構文は以下のようになります。

```
#!/bin/bash

printf "Hello World\n"
```

例2.2 Python で書かれた Hello World

pello プロジェクトは、Python に **Hello World** を実装します。この実装には、**pello.py** プログラムのみが含まれます。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。

pello.py ファイルの構文は以下のようになります。

```
#!/usr/bin/python3

print("Hello World")
```

例2.3 C で書かれた Hello World

> **cello** プロジェクトは、C の **Hello World** を実装します。この実装には、**cello.c** および **Makefile** ファイルだけが含まれます。そのため、生成される **tar.gz** アーカイブには、**LICENSE** ファイル以外にファイルが 2 つ含まれます。
>
> 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** ファイルの構文は以下のようになります。
>
> ```
> #include <stdio.h>
>
> int main(void) {
>     printf("Hello World\n");
>     return 0;
> }
> ```

2.2. プログラムが作られる仕組み

人間が判読できるソースコードからマシンコード (コンピューターがプログラムを実行するために従う命令) に変換する方法は、以下のとおりです。

- プログラムが ネイティブにコンパイル される。
- プログラムが、マシンコードの解釈により解釈される。
- プログラムがバイトコンパイルにより解釈される。

2.2.1. ネイティブにコンパイルされたコード

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。このようなソフトウェアは、スタンドアロンで実行できます。

この方法でビルドした 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. 解釈されたコード

bash や Python などの一部のプログラミング言語は、マシンのコードにコンパイルしません。代わりに、プログラムのソースコードは、言語インタープリターまたは言語仮想マシンにより、事前の変換なしで順を追って実行されます。

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。

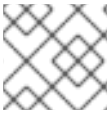
インタープリター言語は、raw インタープリタープログラム または バイトコンパイル言語です。この 2 つは、パッケージ化作業のプログラムビルドプロセスにおいて異なります。

2.2.2.1. raw インタープリタープログラム

raw インタープリター言語プログラムはコンパイルする必要はなく、インタープリターにより直接実行されます。

2.2.2.2. バイトコンパイルプログラム

バイトコンパイル言語は、バイトコードにコンパイルして、その言語の仮想マシンにより実行される必要があります。



注記

一部の言語では、raw インタープリターとバイトコンパイルを選ぶことができます。

2.3. ソースからのソフトウェア構築

コンパイル言語で書かれたソフトウェアの場合、ソースコードはビルドプロセスを経て、マシンコードを生成します。このプロセスは、コンパイルまたはトランスレートと呼ばれ、言語によって異なります。その結果構築されるソフトウェアは実行できるようになります。これにより、プログラマーが指定したタスクをコンピューターが実行するようになります。

raw インタープリター言語で書かれたソフトウェアの場合、ソースコードは構築されず、直接実行します。

バイトコンパイル型のインタープリター言語で書かれたソフトウェアの場合、ソースコードがバイトコードにコンパイルされ、その言語の仮想マシンによって実行されます。

本章では、ソースコードからソフトウェアを構築する方法を説明します。

2.4. 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構築

本セクションでは、C 言語で書かれた **cello.c** プログラムの実行可能なファイルへの構築方法を紹介します。

cello.c

```
#include <stdio.h>

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

2.4.1. 手動による構築

cello.c プログラムを手動で構築する場合は、以下の手順を使用します。

手順

1. GNU コンパイラコレクションから 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. 生成された出力バイナリー **cello** を実行します。

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

2.4.2. 自動化ビルド

大規模なソフトウェアは通常、**Makefile** ファイルを作成し、GNU **make** ユーティリティーを実行して自動ビルドを使用します。

自動ビルドを使用して **cello.c** プログラムを構築する場合は、以下の手順を使用します。

手順

1. 自動ビルドを設定するには、次の内容の **Makefile** ファイルを **cello.c**と同じディレクトリーに作成します。
makefile

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

cello: および **clean:** の行は、タブスペースで始まる必要があります。

2. ソフトウェアを構築するには、**make** コマンドを実行します。

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

3. ビルドはすでに利用できるため、**make clean** コマンドを実行してから、**make** コマンドを再び実行します。

```
$ make clean
rm cello

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

注記

別のビルドに影響がなければ、プログラムの構築を試行します。

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

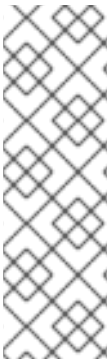
4. プログラムを実行します。

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

これで、ビルドツールを使用した手動によるプログラムのコンパイルが完了しました。

2.5. コードの解釈

本セクションでは、Python で書かれたプログラムをバイトコンパイルして、bash で書かれたプログラムをそのまま解釈する方法を示しています。



注記

以下の 2 つの例では、ファイルの一番上の **#!** 行は、**シバン (shebang)** と呼ばれるもので、プログラミング言語ソースコードの一部ではありません。

シバン により、実行ファイルとしてテキストファイルを使用できるようになります。システムプログラムローダーは、**シバン**を含む行を解析して、バイナリーの実行ファイルへのパスを取得します。これは、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。この場合は、テキストファイルを実行ファイルとしてマークする必要があります。

2.5.1. コードのバイトコンパイル

本セクションでは、Python で書かれた **pello.py** プログラムをバイトコードにコンパイルし、Python 言語の仮想マシンで実行する方法を説明します。

Python のソースコードは、そのまま解釈することもできますが、バイトにコンパイルした方が高速です。したがって、RPM パッケージャーは、エンドユーザーが配布するバージョンにはバイトコンパイルのパッケージ化を推奨しています。

pello.py

```
#!/usr/bin/python3

print("Hello World")
```

プログラムのバイトコンパイル手順は、以下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

Python は、多くの場合バイトコンパイルが行われますが、ここでは説明しません。以下の手順は、コミュニティーの標準に準拠するのではなく、簡潔さを重視しています。実際の Python ガイドラインは「Software Packaging and Distribution」を参照してください。

この手順に従って **pello.py** をバイトコードにコンパイルします。

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
```

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

2. **pello.pyc** のバイトコードを実行します。

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

2.5.2. raw-interpreting code

本セクションでは、bash シェルの組み込み言語で書かれた **bello** プログラムをそのまま解釈する方法を示しています。

bello

```
#!/bin/bash

printf "Hello World\n"
```

bashなどのシェルスクリプト言語で書かれたプログラムはそのまま解釈されます。

手順

- ソースコードの実行ファイルでファイルを作成して実行します。

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

2.6. ソフトウェアへのパッチの適用

RPM のパッケージ化では、元のソースコードを変更するのではなく、コードを維持し、コードにパッチを使用します。

パッチは、ソースコードを更新するソースコードです。これは、2 つのバージョンのテキストの差を示すため、**diff** としてフォーマットされます。**diff** は、**diff** ユーティリティーを使用して作成されます。これは、パッチ ユーティリティーを使用してソースコードに適用されます。

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。

本セクションでは、ソフトウェアにパッチを適用する方法を説明します。

以下の例は、**diff** を使用して元のソースコードからパッチを作成する方法と、**パッチ** でパッチを適用する方法を示しています。後半のセクションで RPM を作成するときにパッチを適用します。

この手順では、**cello.c** の元のソースコードからパッチを作成する方法を説明します。

手順

1. 元のソースコードを保持します。

```
$ 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するために使用されます。

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. **diff** ユーティリティーを使用して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、元のソースコードから削除され、**+** で始まる行に置き換えられます。

通常のユースケースの多くに適切なため、**diff** コマンドに **Naur** オプションを指定することが推奨されます。ただし、この場合は、**-u** オプションのみが必要になります。特定のオプションにより、以下が確保されます。

- **-N** (または **--new-file**) - 存在しないファイルを、空のファイルであるかのように処理します。
- **-a** (または **--text**) - すべてのファイルをテキストとして処理します。そのため、**diff** がバイナリーとして分類するファイルは無視されません。
- **-u** (もしくは **-U NUM** または **--unified[=NUM]**) - 統一されたコンテキストの出力の NUM (デフォルトは 3) 行の形式で、出力を返します。これは、変更したソースツリーにパッチを適用する際に、あいまい一致を可能にする読みやすい形式です。
- **-r** (または **--recursive**) - 検出されたサブディレクトリーを再帰的に比較します。

diff ユーティリティーの一般的な引数は、man ページの **diff** を参照してください。

4. ファイルにパッチを保存します。

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

5. 元の **cello.c** を復元します。

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

RPM を構築するときは変更後のファイルではなく、元のファイルが使用されるため、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。

以下の手順は、**output-first-patch.patch** を使用して **cello.c** にパッチを適用して、パッチプログラムを構築し、これを実行する方法を示しています。

手順

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

2. **cello.c** の内容がパッチを反映し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;
}
```

3. パッチが適用された **cello.c** を構築して実行します。

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c

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

2.7. 任意のアーティファクト

UNIX のようなシステムでは、ファイルシステム階層標準 (FHS) を使用して、特定のファイルに適したディレクトリーを指定します。

RPM パッケージからインストールしたファイルは、FHS に従って配置されます。たとえば、実行ファイルは、システム **$PATH** 変数のディレクトリーに置く必要があります。

このドキュメントのコンテキストでは、**任意アーティファクト** は RPM からシステムにインストールされたものを意味します。RPM およびシステムの場合は、スクリプト、パッケージのソースコードからコンパイルしたバイナリー、事前にコンパイルしたバイナリー、またはその他のファイルを意味します。

以下のセクションでは、システムに **任意アーティファクト** を配置する一般的な 2 つの方法を説明します。

- install コマンドの使用

- make install コマンドの使用

2.8. INSTALL コマンドを使用したシステムに任意のアーティファクトを配置する

パッケージャーは、GNU make などのビルド自動化ツールが最適ではない場合に **install** コマンドを使用することがよくあります。たとえば、パッケージ化したプログラムに余分なオーバーヘッドが必要ない場合などが考えられます。

coreutils により、**install** コマンドをシステムで利用できます。このコマンドは、指定のパーミッションセットで、ファイルシステム内の指定したディレクトリーにアーティファクトを配置します。

以下の手順では、このインストール方法に、任意アーティファクトとして以前に作成された **bello** ファイルを使用します。

手順

1. **install** コマンドを実行して、実行可能スクリプトに共通のパーミッションを持つ **/usr/bin** ディレクトリーに **bello** ファイルを配置します。

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

これにより、**bello** は、**$PATH** 変数に一覧表示されているディレクトリーに置かれます。

2. 完全パスを指定せずに、任意のディレクトリーから **bello** を実行します。

```
$ cd ~

$ bello
Hello World
```

2.9. MAKE INSTALL コマンドを使用したシステムに任意のアーティファクトを配置する

make install コマンドを使用することで、ビルドしたソフトウェアをシステムに自動的にインストールできます。この場合、開発者が作成する **Makefile** 内のシステムにおいて、任意アーティファクトをシステムにインストールする方法を指定する必要があります。

この手順では、システム上の任意の場所にビルドアーティクトをインストールする方法を説明します。

手順

1. **Makefile** に**インストール**セクションを追加します。
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
```

cello:、**clean:**、および **install:** の行は、行頭にタブスペースを追加する必要があります。

注記

$(DESTDIR) 変数は GNU make の組み込みで、一般的には、root ディレクトリーとは異なるディレクトリーへのインストールを指定するために使用されます。

これで、**Makefile** を使用してソフトウェアを構築するだけでなく、ターゲットシステムへのインストールも可能になります。

2. **cello.c** プログラムを構築してインストールします。

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

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

これにより、**cello** 変数に記載されているディレクトリーに **cello** が置かれます。

3. 完全パスを指定せずに、任意のディレクトリーから **cello** を実行します。

```
$ cd ~

$ cello
Hello World
```

2.10. パッケージ化を行うためのソースコードの準備

開発者は、ソースコードの圧縮アーカイブとしてソフトウェアを配布するため、パッケージの作成に使用されます。RPM パッケージャーは、準備の整ったソースコードアーカイブと連携します。

ソフトウェアは、ソフトウェアライセンスとともに配布する必要があります。

この手順では、**LICENSE** ファイルのサンプルコンテンツとして GPLv3 ライセンステキストを使用します。

手順

1. **LICENSE** ファイルを作成し、以下の内容が含まれることを確認します。

```
$ 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

- このセクションで作成したコード

2.11. ソースコードを TARBALL へ追加

本セクションでは、ソースコードとは セクションで紹介されている 3 つの **Hello World** プログラムをそれぞれ gzip で圧縮した tarball にまとめる方法を説明します。これは、後で配布用にパッケージ化するソフトウェアをリリースする一般的な方法です。

例2.4 bello プロジェクトを tarball へ追加

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るため、生成される **tar.gz** アーカイブには **LICENSE** ファイルとは別のファイルのみが含まれます。

この手順では、配布用に **bello** を準備する方法を示しています。

前提条件

このプログラムのバージョンが **0.1** であることを考慮してください。

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 /tmp/bello-0.1

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

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

2. 配布用のアーカイブを作成し、これを **~/rpmbuild/SOURCES/** ディレクトリーに移動します。このディレクトリーは、**rpmbuild** コマンドがパッケージを構築するファイルを保存するデフォルトディレクトリーです。

```
$ 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/
```

bash で書かれたサンプルソースコードの詳細は「bash で書かれた Hello World」を参照してください。

例2.5 pello プロジェクトを tarball へ追加

pello プロジェクトは、Python に **Hello World** を実装します。この実装には **pello.py** プログラムのみが含まれるため、生成された **tar.gz** アーカイブには **LICENSE** ファイルとは異なる 1 つのファイルのみが含まれます。

この手順では、**pello** プロジェクトを配布するために準備する方法を示します。

前提条件

このプログラムのバージョンが **0.1.1** であることを考慮する。

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 /tmp/pello-0.1.2

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

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

2. 配布用のアーカイブを作成し、これを **~/rpmbuild/SOURCES/** ディレクトリーに移動します。このディレクトリーは、**rpmbuild** コマンドがパッケージを構築するファイルを保存するデフォルトディレクトリーです。

```
$ 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/
```

Python で書かれたサンプルソースコードの詳細は「Python で書かれた Hello World」を参照してください。

例2.6 cello プロジェクトを tarball へ追加

cello プロジェクトは、C の **Hello World** を実装します。この実装には、**cello.c** および **Makefile** ファイルだけが含まれます。そのため、生成される **tar.gz** アーカイブには、**LICENSE** ファイル以外にファイルが 2 つ含まれます。

パッチ ファイルは、プログラムとともにアーカイブで配布されないことに注意してください。RPM Packager は、RPM を構築する際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。

この手順では、**cello** プロジェクトを配布するために準備する方法を示します。

前提条件

このプログラムのバージョンが **1.0** であることを考慮してください。

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 /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. 配布用のアーカイブを作成し、これを **~/rpmbuild/SOURCES/** ディレクトリーに移動します。このディレクトリーは、**rpmbuild** コマンドがパッケージを構築するファイルを保存するデフォルトディレクトリーです。

```
$ 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. パッチを追加します。

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

C で書かれたサンプルソースコードの詳細は「C で書かれた Hello World」を参照してください。

第3章 ソフトウェアのパッケージ化

このセクションでは、RPM のパッケージ化の基本を説明します。

3.1. RPM パッケージ

RPM パッケージは、他のファイルとそのメタデータ (システムが必要とするファイルに関する情報) を含むファイルです。

特に、RPM パッケージは **cpio** アーカイブで構成されています。

cpio アーカイブには以下が含まれます。

- ファイル
- 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依存関係、ファイルのインストール先、およびその他の情報を決定します。

RPM パッケージの種類
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、ソースコードへのパッチも含まれます。
- 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。

3.2. RPM パッケージ化ツールのユーティリティーの一覧表示

以下の手順では、**rpmdevtools** パッケージが提供するユーティリティーの一覧を表示する方法を説明します。

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

- RPM パッケージ化ツールのユーティリティーを一覧表示します。

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

上記のユーティリティーの詳細は、各マニュアルページまたはヘルプダイアログを参照してください。

3.3. RPM パッケージ化を行うためのワークスペースの設定

本セクションでは、**rpmdev-setuptree** ユーティリティーを使用して、RPM のパッケージ化ワークスペースとなるディレクトリーレイアウトを設定する方法を説明します。

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
```

作成されるディレクトリーの目的は、以下のとおりです。

ディレクトリー	目的
BUILD	パッケージを構築すると、ここにさまざまな **%buildroot** ディレクトリーが作成されます。これは、ログ出力で十分な情報を得られない場合に、失敗したビルドを調べるのに場合に便利です。
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 (例: **x86_64** および **noarch**) に作成されます。
SOURCES	ここでは、このパッケージャーは、圧縮したソースコードアーカイブとパッチを配置します。**rpmbuild** コマンドは、これらを検索します。
SPECS	パッケージャーは、SPEC ファイルをここに配置します。
SRPMS	**rpmbuild** を使用してバイナリー RPM の代わりに SRPM を構築すると、生成される SRPM がここに作成されます。

3.4. SPEC ファイルとは

SPEC ファイルには、RPM を構築するのに **rpmbuild** ユーティリティーが使用するレシピが含まれています。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。このセクションは、**Preamble** と **Body** で定義されます。**Preamble** では、**Body** 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。**Body** は、命令の主要部分を示しています。

次のセクションでは、SPEC ファイルの各セクションを説明します。

3.4.1. Preamble 項目

以下の表では、RPM SPEC ファイルの **Preamble** セクションで頻繁に使用されるディレクティブの一部を示しています。

表3.1 RPM SPEC ファイルの Preamble セクションで使用される項目

SPEC ディレクティブ	定義
Name	SPEC ファイル名と一致する必要があるパッケージのベース名。
Version	ソフトウェアのアップストリームのバージョン番号。
Release	このバージョンのソフトウェアがリリースされた回数。通常、初期値は 1%{?dist} に設定し、パッケージの新規リリースごとに増加させます。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
License	パッケージ化しているソフトウェアのライセンス。
URL	プログラムに関する詳細情報の完全な URL。多くの場合、この URL は、パッケージ化しているソフトウェアのアップストリームプロジェクトの Web サイトです。
Source0	アップストリームのソースコードの圧縮アーカイブへのパスまたは URL (パッチを適用していないものや、パッチは別の場所で処理されます)。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。必要に応じて、SourceX ディレクティブを追加して、たとえば、Source1、Source2、Source3 など、毎回数を増やすことができます。
Patch	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。 ディレクティブは、パッチの末尾に数字を付けて、または付けずに適用できます。 数値を指定しないと、内部的にエントリーに割り当てられます。Patch0、Patch1、Patch2、Patch3 などを使用して、明示的に数字を指定することもできます。 このパッチは、%patch0、%patch1、%patch2 といったマクロを使用して、1 つずつ適用できます。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。

SPEC ディレクティブ	定義
BuildArch	パッケージがアーキテクチャーに依存していない場合は (たとえば、インタープリター型のプログラミング言語ですべて書かれた場合など)、これを **BuildArch: noarch** に設定します。設定しないと、パッケージは構築されるマシンのアーキテクチャー (**x86_64**など) を自動的に継承します。
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	ソフトウェアの一部が特定のプロセッサーアーキテクチャーで動作しない場合には、そのアーキテクチャーを除外できます。
Conflicts	**Conflicts** は **Requires** と逆の意味を持ちます。**Conflicts** に一致するパッケージが存在すると、既にインストールされているパッケージに **Conflict** タグがあるか、インストールされるパッケージにある場合は、そのパッケージを独立してインストールすることができません。
Obsoletes	このディレクティブでは、**rpm** コマンドが直接コマンドラインで使用されるか、更新が更新または依存関係リゾルバーにより実行されるかによって、更新の方法が変更されます。コマンドラインで使用すると、RPM により、インストールしているパッケージに一致するすべての古いパッケージが削除されます。更新または依存関係リゾルバーを使用する場合は、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を置き換えます。
Provides	**Provides** がパッケージに追加されると、名前以外の依存関係でパッケージを参照できます。

Name のディレクティブ、**Version** のディレクティブ、および **Release** のディレクティブは、RPM パッケージのファイル名から構成されます。RPM パッケージの担当者やシステム管理者は、これら 3 つのディレクティブを **N-V-R** または **NVR** と呼びます。これは、RPM パッケージのファイル名に **NAME-VERSION-RELEASE** 形式が含まれるためです。

以下の例は、**rpm** コマンドを実行して、特定のパッケージの **NVR** 情報を取得する方法を示しています。

例3.1 bash パッケージの NVR 情報を出力する rpm のクエリー

```
# 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。最後のマーカーの **x86_64** は、アーキテクチャーを意味しています。**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。

3.4.2. Body 項目

RPM SPEC ファイルの **Body セクション** の項目を以下の表に一覧表示します。

表3.2 RPM SPEC ファイルの Body セクションで使用される項目

SPEC 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	**Source0**でアーカイブを展開するなど、構築するソフトウェアを準備する単一または一連のコマンド。この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 (コンパイル言語用) またはバイトコード (インタープリター言語) に構築するための 1 つまたは一連のコマンド。
%install	**%builddir** (ビルドが行われた場所) から、パッケージ化するファイルのディレクトリー構造を含む **%buildroot** ディレクトリーに、希望のビルドアーティファクトをコピーする単一または一連のコマンド。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。これは、エンドユーザーがパッケージをインストールするときではなく、パッケージを作成する時にのみ実行されます。詳細は SPEC ファイルの使用 を参照してください。
%check	ソフトウェアをテストする単一または一連のコマンド。これには通常、ユニットテストなどが含まれます。
%files	エンドユーザーのシステムにインストールされるファイルの一覧。
%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。

3.4.3. 高度な項目

SPEC ファイルには、Scriptlets や Triggers などの高度な項目を追加することもできます。

これは、ビルドプロセスではなく、エンドユーザーのシステムのインストールプロセスのさまざまな地点で有効になります。

3.5. 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。つまり、ビルドのアーティファクトが、エンドユーザーシステムの今後の階層と同じファイルシステム階層を使用して配置され、**buildroot** が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基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。

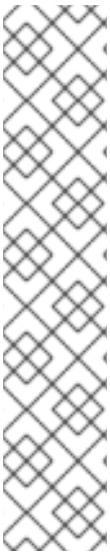
注記

Red Hat Enterprise Linux 6 以降では、**rpmbuild** プログラムには独自のデフォルトが設定されています。このデフォルト値を上書きすると、問題が発生することがあります。Red Hat では、このマクロの値を自身で定義することを推奨していません。**%{buildroot}** マクロは、**rpmbuild** ディレクトリーのデフォルトで使用できます。

3.6. 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、ユーザーに変わってテキストの置換を行うことができます。

使用例では、SPEC ファイルでパッケージ化されたソフトウェアの **Version** を複数回参照しています。**%{version}** マクロで 1 回だけ **Version** を定義し、SPEC ファイル全体でこのマクロを使用します。すべては、以前に定義した **Version** 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}
```

%{_bindir} マクロおよび %{_libexecdir} マクロの評価

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

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

一般的に使用されるマクロの 1 つに **%{?dist}** マクロがあります。これは、ビルドに使用されるディストリビューション (ディストリビューションタグ) を示します。

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

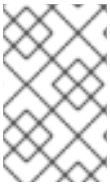
3.7. SPEC ファイルでの作業

新しいソフトウェアをパッケージ化するには、新しい SPEC ファイルを作成する必要があります。

これを行うには、以下の 2 つの方法があります。

- 手動による SPEC ファイルの新規作成
- **rpmdev-newspec** ユーティリティーの使用

このユーティリティーは、空の SPEC ファイルを作成し、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。

以下のセクションでは、ソースコードとはで説明されている **Hello World!** プログラムの 3 つの実装例を使用します。

各プログラムは、以下の表で詳細に説明しています。

ソフトウェアの名前	例の説明
bello	raw インタープリタープログラミング言語で書かれたプログラム。ソースコードを構築する必要はなく、インストールのみが必要である場合を示しています。事前にコンパイル済みのバイナリーをパッケージ化する必要がある場合、バイナリーは単なるファイルであるため、この方法を使用することもできます。
pello	バイトコンパイルインタプリタープログラム言語で書かれたプログラム。これは、ソースコードのバイトコンパイルと、生成される事前処理ファイルのバイトコードのインストールを示しています。
cello	ネイティブコンパイル言語で書かれたプログラム。これは、ソースコードをマシンコードにコンパイルし、生成される実行ファイルをインストールする一般的なプロセスを示しています。

Hello World! の実装は次のとおりです。

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

前提条件として、これらの実装は、**~/rpmbuild/SOURCES** ディレクトリーに置く必要があります。

3.8. RPMDEV-NEWSPEC を使用した新規 SPEC ファイルの作成

以下の手順は、**rpmdev -newspec** ユーティリティーを使用して、上記の 3 つの **Hello World!** プログラムごとに SPEC ファイルを作成する方法を示しています。

手順

1. **~/rpmbuild/specs** ディレクトリーに移動し、**rpmdev -newspec** ユーティリティーを使用します。

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

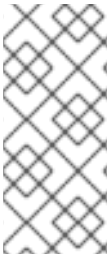
```
$ rpmdev-newspec bello
bello.spec created; type minimal, rpm version >= 4.11.

$ 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/ ディレクトリーには、**bello.spec**、**cello.spec**、および **pello.spec** という名前の 3 つの SPEC ファイルが含まれています。

2. ファイルを調べます。
ファイル内のディレクティブは、SPEC ファイルとは で説明されているディレクティブを表します。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。



注記

rpmdev -newspec ユーティリティーは、特定の Linux ディストリビューションに固有のガイドラインや規則を使用しません。ただし、本ドキュメントは Red Hat Enterprise Linux を対象にしています。そのため、、SPEC ファイル全体にわたり定義または提供したその他のすべてのマクロとの一貫性を確立するために、RPM のビルドルートを参照する際には、**$RPM_BUILD_ROOT** において **%{buildroot}** の記述が推奨されます。

3.9. RPM を作成するための、元の SPEC ファイルの変更

以下の手順では、RPM を作成する **rpmdev- newspec** による SPEC 出力ファイルを修正する方法を示しています。

前提条件

- 特定のプログラムのソースコードが、**~/rpmbuild/SOURCES/** ディレクトリーに置かれている。
- 空の SPEC ファイル (**~/rpmbuild/specs/<name>.spec** ファイル) が、**rpmdev -newspec** ユーティリティーで作成されている。

手順

1. **rpmdev -newspec** ユーティリティーで生成される **~/rpmbuild/specs/<name>.spec** ファイルの出力テンプレートを開きます。
2. SPEC ファイルの最初のセクションを作成します。
最初のセクションには、**rpmdev -newspec** がグループ化される以下のディレクティブが含まれます。

Name

Name は既に **rpmdev -newspec** の引数として指定されています。

Version

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。

Release

Release は、**1%{?dist}** に自動的に設定されます。最初は **1** となります。パッチを追加する場合など、アップストリームリリースの **Version** を変更せずにパッケージを更新するたびに、初期値を増やします。新しいアップストリームリリースが行われた際に、**Release** が **1** にリセットされます。

Summary

Summary は、ソフトウェアに関する 1 行の短い説明です。

3. **License**、**URL**、および **Source0** ディレクティブを入力します。
License フィールドは、アップストリームリリースのソースコードに関連するソフトウェアライセンスです。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。

たとえば、GPLv3+ を使用できます。

URL フィールドは、アップストリームのソフトウェア Web サイトへの URL を指定します。一貫性を保つために、**%{name}** の RPM マクロ変数を利用して、 **https://example.com/% {name}** を使用します。

Source0 フィールドは、アップストリームのソフトウェアソースコードへの URL を指定します。これは、パッケージ化している特定のバージョンのソフトウェアに直接リンクする必要があります。本ドキュメントの URL の例には、将来変更される可能性があるハードコーディングし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。これらを使用して、SPEC ファイルの 1 つのフィールドのみを更新する必要があります。

4. **BuildRequires** ディレクティブ、**Requires** ディレクティブ、および **BuildArch** ディレクティブを入力します。
BuildRequires は、パッケージのビルドタイム依存関係を指定します。

Requires は、パッケージのランタイム依存関係を指定します。

これは、ネイティブにコンパイルされた拡張機能がない、インタープリター型プログラミング言語で書かれたソフトウェアです。したがって、**noarch** 値とともに **BuildArch** ディレクティブを追加します。これは、このパッケージを構築するプロセッサーアーキテクチャーに制限する必要がないことを RPM に指定します。

5. **%description**、**%prep**、**%build**、**%install**、**%files**、**%license** 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 は、ソフトウェアの完全な説明で **Summary** よりも長く、複数の段落が含まれています。

% prep セクションでは、ビルド環境の準備方法を指定します。通常、これには、ソースコードの圧縮アーカイブの展開、パッチの適用、および SPEC ファイルの後半で使用するためにソースコードでによる情報の解析が含まれます。このセクションでは、ビルトインの **% setup -q** マクロを使用できます。

%build セクション では、ソフトウェアを構築する方法を指定します。

%install セクションには、ソフトウェアを構築してから **BUILDROOT** ディレクトリーにインストールする方法に関する **rpmbuild** の説明が記載されています。

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。

%files セクションは、この RPM によるファイルのリストと、エンドユーザーシステム上のファイルの完全なパス場所を指定します。

このセクションでは、組み込みのマクロを使用して、さまざまなファイルの役割を示すことができます。これは、**rpm** コマンドを使用したパッケージファイルマニフェストのメタデータの照会に役立ちます。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。

6. 最後のセクションの **%changelog** は、パッケージの各 Version-Release に対する日付入りのエントリーの一覧です。これらは、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。パッケージ変更の例: パッチの追加、**%build** セクションのビルド手順の変更。
最初の行は、以下の形式に従います。

   * 文字でで始まり、**Day-of-Week Month Day Year Name Surname <email> - Version-Release** が続きます。

   実際の変更エントリーには、以下の形式に従います。

   - 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 **-** 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

- bash で書かれたプログラム用の SPEC ファイルサンプル
- Python で書かれたプログラムの SPEC ファイルサンプル
- C で書かれたプログラムの SPEC ファイルサンプル
- RPM のビルド

3.10. BASH で書かれたプログラム用の SPEC ファイルサンプル

このセクションでは、bash 書かれた **bello** プログラムの SPEC ファイルの例を示しています。

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。

bash はビルド不要のため、ソフトウェアの構築方法を示す **%build** セクションは空白です。

bello をインストールする場合は、インストール先の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。よって、**%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実行できます。

関連情報

- ソースコードとは

3.11. PYTHON で書かれたプログラムの SPEC ファイルサンプル

このセクションでは、Python プログラミング言語で書かれた **pello** プログラムの SPEC ファイルの例を示します。

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
```

重要

pello バイトコンパイルインタープリター言語で書かれたプログラムです。よって、生成されるファイルにはエントリーが含まれていないため、シバンは使いません。

シバンは使わないため、以下のアプローチのいずれかを適用する必要があります。

- 実行ファイルを呼び出す、バイトコンパイル以外のシェルスクリプトを作成します。
- プログラムの実行にエントリーポイントとしてバイトコンパイルされない小規模の Python コードを提供します。

これらのアプローチは特に、事前にコンパイルされたコードのパフォーマンス向上が大きい、数千行ものコードを含む大規模ソフトウェアプロジェクトに便利です。

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コードを実行するには、**python** パッケージが必要です。
- 小規模なエントリーポイントスクリプトを実行するには、**bash** パッケージが必要。

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** パッケージが必要です。

%build セクションは、ソフトウェアを構築する方法を指定します。つまり、ソフトウェアがバイトコンパイルされるということになります。

pello をインストールするには、ラッパースクリプトを作成する必要があります。これは、シバンがバイトコンパイル言語で該当しないためです。これを行うには、以下のような複数のオプションを利用できます。

- 個別のスクリプトを作成し、それを個別の **SourceX** ディレクティブとして使用します。
- SPEC ファイルにファイルをインラインで作成。

この例では、SPEC ファイルにラッパースクリプトのインラインを作成し、SPEC 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、**こちら**の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。

この例の **%install** セクションは、アクセスできるように、バイトコンパイル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する必要があるという事実に一致します。

関連情報

- ソースコードとは

3.12. C で書かれたプログラムの SPEC ファイルサンプル

このセクションでは、C プログラミング言語で書かれた **cello** プログラム用の SPEC ファイルの例を示します。

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 2 つのパッケージが含まれます。

- **gcc** パッケージ
- **make** パッケージ

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** が書かれているため、**rpmdev-newspec** ユーティリティーによる 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。

cello プログラムのインストールは、**rpmdev-newspec** コマンドによる **%make_install** マクロを使用して行うことができます。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

3.13. RPM のビルド

RPM は、**rpmbuild** コマンドで構築されます。このコマンドは、特定のディレクトリーと **rpmdev-setuptree** ユーティリティーで設定された構造と同じファイル構造を想定します。

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。主なユースケースは以下の2 つです。

- 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
  - ソース RPM からのバイナリー RPM のビルド

次のセクションでは、プログラムの SPEC ファイルを作成した後に RPM をビルドする方法を説明します。

3.14. ソース RPM のビルド

次の手順では、ソース RPM のビルド方法を説明します。

前提条件

- パッケージ化するプログラムの SPEC ファイルが既に存在している必要があります。

手順

- 指定の SPEC ファイルを使用して **rpmbuild** コマンドを実行します。

  ```
  $ rpmbuild -bs SPECFILE
  ```

  SPECFILE を SPEC ファイルに置き換えます。**-bs** オプションは、ビルドソースを表します。

以下の例は、**bello** プロジェクト、**pello** プロジェクト、および **cello** プロジェクトのソース RPM のビルドを示しています。

bello、pello、および cello のソース RPM のビルド。

```
$ 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
```

検証手順

- 生成されたソース RPM が **rpmbuild/SRPMS** ディレクトリーに含まれていることを確認してください。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
- rpmdev-newspec を使用した新規 SPEC ファイルの作成
- RPM を作成するための、元の SPEC ファイルの変更

3.15. ソース RPM からのバイナリー RPM の再ビルド

以下の手順は、ソース RPM (SRPM) からバイナリー RPM を再構築する方法を示しています。

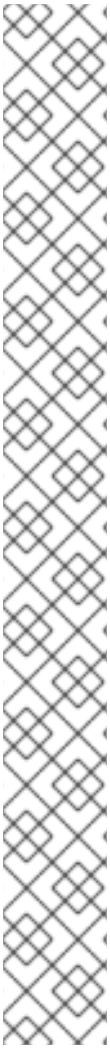
手順

- SRPMS から **bello**、**pello**、および **cello** を再構築するには、以下を実行します。

```
$ 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]
```



注記

rpmbuild --rebuild を起動すると、以下が関係します。

- 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。
- インストール済みコンテンツを使用したビルド。
- SPEC ファイルとソースコードの削除

SPEC ファイルとソースコードをビルド後も維持するには、以下を行います。

- ビルド時には、**--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。
- 以下のコマンドを使用して 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%]
```

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、**YOURARCH** がアーキテクチャーとなる ~/**rpmbuild**/**RPMS**/**YOURARCH** ディレクトリーか、パッケージがアーキテクチャー固有でなければ、~/**rpmbuild**/**RPMS**/**noarch**/ ディレクトリーに位置します。

3.16. SPEC ファイルからのバイナリー RPM のビルド

以下の手順では、SPEC ファイルから **bello**、**pello**、および **cello** バイナリー RPM のビルド方法を示しています。

手順

- **bb** オプションを指定して、**rpmbuild** コマンドを実行します。

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

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

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

3.17. ソース RPM からのバイナリー RPM の構築

ソース RPM からあらゆる種類の RPM をビルドすることもできます。これを行うには、以下の手順を行います。

手順

- 以下のオプションのいずれかと、ソースパッケージを指定して、**rpmbuild** コマンドを実行します。

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

関連情報

- **rpmbuild(8)** man ページ

3.18. RPM のサニティーチェック

パッケージを作成したら、パッケージの品質を確認する必要があります。

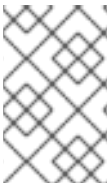
パッケージの品質をチェックする主要なツールは、rpmlint です。

rpmlint ツールは、以下のことを行います。

- RPM の保守性の向上。
- 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 ツールはバイナリー RPM、ソース RPM (SRPMS) 、SPEC ファイルをチェックできるため、以下のセクションで示すように、パッケージ化のすべての段階で役に立ちます。

rpmlint には非常に厳密なガイドラインがあるため、以下の例にあるように、一部のエラーや警告をスキップできる場合もあることに注意してください。



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。それぞれのエラーや警告の詳細な説明は、**rpmlint -i** を実行してください。

3.19. BELLO によるサニティーチェック

本セクションでは、bello SPEC ファイルおよび bello バイナリー RPM の例で RPM のサニティーチェックを行う際に発生する可能性のある警告およびエラーを示します。

3.19.1. cello の SPEC ファイルの確認

例3.2 bello の SPEC ファイルでのrpmlint コマンド実行の出力

```
$ 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 には、**Source0** ディレクティブに一覧表示される URL に到達できないことを示す警告が 1 つのみあります。**example.com** URL は存在しないため、この出力は当然です。今後、この URL が機能すると仮定して、この警告を無視します。

例3.3 cello の SRPM で rpmlint コマンドを実行した場合の出力

```
$ 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 については、**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示す新しい警告が表示されます。今後、リンクが機能すると仮定して、この警告を無視します。

3.19.2. bello バイナリー RPM の確認

バイナリー RPM をチェックする場合、**rpmlint** は以下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

例3.4 bello のバイナリー RPM での rpmlint コマンドの実行の出力

```
$ 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** の警告では、RPM にドキュメントや man ページがないことが表示されます。これは指定しないため当然です。上記の警告とは別に、RPM は **rpmlint** チェックに合格しています。

3.20. PELLO のサニティーチェック

本セクションでは、pello の SPEC ファイルおよび pello のバイナリー RPM の例で RPM のサニティーチェックを行う際に発生する可能性のある警告およびエラーを示します。

3.20.1. cello の SPEC ファイルの確認

例3.5 pello の SPEC ファイルで rpmlint コマンドを実行した場合の出力

```
$ 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 警告では、**Source0** ディレクティブに一覧表示される URL にアクセスできないことが書かれています。**example.com** URL は存在しないため、この出力は当然です。この URL が今後機能すると仮定して、この警告を無視します。

hardcoded-library-path エラーでは、ライブラリーパスをハードコーディングするのではなく、**%{_libdir}** マクロを使用することが推奨されます。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、実際のパッケージの場合は、すべてのエラーが慎重にチェックされていることを確認してください。

例3.6 cello 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.
```

ここでの新しい **invalid-url URL** エラーは、到達できない **URL** ディレクティブに関するものです。今後、この URL が有効であると仮定して、この警告を無視しても問題はありません。

3.20.2. cello バイナリー RPM の確認

バイナリー RPM をチェックする場合、**rpmlint** は以下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

例3.7 pello のバイナリー RPM でのrpmlint コマンドの実行の出力

```
$ 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** の警告では、RPM にドキュメントや man ページがないことが表示されます。これは指定しないため当然です。

only-non-binary-in-usr-lib 警告では、**/usr/lib/** にバイナリーでないアーティクトのみを提供していることが表示されます。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用に予約されています。したがって、**rpmlint** は、**/usr/lib/** ディレクトリー内の少なくとも 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。通常、ファイルを正しく配置するには 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 エラーは、**/usr/lib/pello/pello.py** ファイルに実行権限がないことを警告します。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。

上記の警告およびエラーとは別に、RPM は **rpmlint** チェックに合格しています。

3.21. CELLO のサニティーチェック

本セクションでは、cello の SPEC ファイルおよび pello のバイナリー RPM の例で RPM のサニティーチェックを行う際に発生する可能性のある警告およびエラーを示します。

3.21.1. cello の SPEC ファイルの確認

例3.8 cello の SRPM で rpmlint コマンドを実行した場合の出力

```
$ 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 には、**Source0** ディレクティブに一覧表示される URL に到達できないことを示す警告が 1 つのみあります。**example.com** URL は存在しないため、この出力は当然です。この URL が今後機能すると仮定して、この警告を無視します。

例3.9 cello の SRPM で rpmlint コマンドを実行した場合の出力

```
$ 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、**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示す新しい警告が表示されます。今後、リンクが機能すると仮定して、この警告を無視することができます。

3.21.2. cello バイナリー RPM の確認

バイナリー RPM をチェックする場合、**rpmlint** は以下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

例3.10 cello のバイナリー RPM で rpmlint コマンドを実行した場合の出力

```
$ 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** の警告では、RPM にドキュメントや man ページがないことが表示されます。これは指定しないため当然です。上記の警告とは別に、RPM は **rpmlint** チェックに合格しています。

3.22. RPM アクティビティーの SYSLOG へのロギング

RPM アクティビティーまたはトランザクションはすべて、システムロギングプロトコル (syslog) によりログに記録できます。

前提条件

- RPM トランザクションの syslog へのロギングを有効にするには、**syslog** プラグインがシステムにインストールされていることを確認します。

```
# 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。

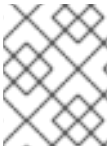
RPM アクティビティーの更新を表示するには、説明されている手順に従います。

手順

1. syslog メッセージを保存するように設定したファイルを開くか、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。

3.23. RPM コンテンツの抽出

特定の状況 (RPM に必要なパッケージが破損している場合など) では、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。

以下の手順では、**rpm2archive** ユーティリティーを使用して、rpm ペイロードを tar アーカイブに変換する方法を説明します。

手順

- 以下のコマンドを実行します。

```
$ rpm2archive filename.rpm
```

filename を、**.rpm** ファイルの名前に置き換えます。

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** パッケージのアーカイブを作成するには、次のコマンドを実行します。

```
$ 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。

4.1. パッケージの署名

サードパーティーがそのコンテンツを変更できないようにパッケージに署名を行います。ユーザーは、パッケージをダウンロードする際に HTTPS プロトコルを使用して、セキュリティーをさらに強化できます。

パッケージの署名には、以下の 3 つの方法があります。

- 既存パッケージへの署名の追加
- 既存のパッケージの署名の置き換え
- ビルド時のパッケージの署名

前提条件

- パッケージに署名するには、GPG キーの作成 の説明に従って、GNU プライバシーガード (GPG) 鍵を作成する必要があります。

4.1.1. GPG キーの作成

以下の手順では、署名パッケージに必要な GNU Privacy Guard (GPG) キーを作成する方法を説明します。

手順

1. GNU Privacy Guard (GPG) キーペアを生成します。

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

2. 生成したキーを確認し、表示します。

```
# 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
```

4.1.2. 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、署名に使用する GNU プライバシーガード (GPG) 鍵 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。

4.1.3. 既存パッケージへの署名の追加

このセクションでは、署名なしでパッケージを構築する場合に最も役立つケースを説明します。この署名は、パッケージのリリースの直前に追加されます。

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。

複数の署名があると、パッケージ作成者からエンドユーザーに、パッケージの所有権のパスを記録できます。

手順

- パッケージに署名を追加します。

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

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。

4.1.4. 複数の署名のあるパッケージの署名の確認

以下の手順では、複数の署名のあるパッケージの署名を確認する方法を説明します。

手順

- 複数の署名のあるパッケージの署名を確認するには、以下を実行します。

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

rpm --checksig コマンドの出力の 2 つの **pgp** 文字列は、パッケージが回署名されていることを示しています。

4.1.5. 既存のパッケージに署名を追加する実用的な例

本セクションでは、既存のパッケージへの署名の追加が役立つ状況の例を示します。

ある会社の部門が、パッケージを作成し、その部門のキーで署名を行います。次に、本社がパッケージの署名を確認します。次に、そのパッケージにコーポレート署名を追加し、その署名されたパッケージが本物であることを表明します。

これら 2 つの署名が付いた状態で、パッケージが小売商に送られます。この小売商は、署名をチェックし、一致を確認して自身の署名も追加します。

そして、このパッケージは、このパッケージを展開したいと思う会社へと向かいます。パッケージ上の署名をすべて確認すれば、その署名が正式コピーであることが分かります。パッケージが企業の承認を受けたことを従業員に通知するために、その会社独自の署名を追加するかどうかは、パッケージ導入を行う会社の内部管理によって決まります。

4.1.6. 既存のパッケージの署名の置き換え

以下の手順では、各パッケージを再構築せずに公開鍵を変更する方法を説明します。

手順

- 公開鍵を変更するには、次のコマンドを実行します。

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

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。

また、以下の手順で示しているように、**--resign** オプションを指定すると、複数のパッケージの公開鍵を変更できます。

手順

- 複数のパッケージの公開鍵を変更するには、以下のコマンドを実行します。

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

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。

4.1.7. ビルド時のパッケージの署名

以下の手順では、ビルド時にパッケージに署名する方法を説明します。

手順

1. **rpmbuild** コマンドを使用して、パッケージを構築します。

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

2. **--addsign** オプションを指定して、**rpmsign** コマンドでパッケージに署名します。

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

3. 必要に応じて、パッケージの署名を確認します。

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

注記

複数のパッケージのビルドと署名を行う場合は、以下の構文を使用して Pretty good Privacy (PGP) パスフレーズを複数回入力しないようにします。

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

署名の秘密鍵のロックを解除にはパスワードを入力する必要があることに注意してください。

4.2. 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。

4.2.1. 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。

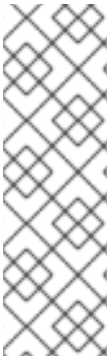
手順

- 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 _ で構成できます。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。

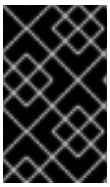
- **Simple** マクロには、**(opts)** フィールドは含まれません。この場合、再帰的なマクロ拡張のみが実行されます。
- **Parametrized** マクロには、**(opts)** フィールドが含まれます。括弧で囲まれている **opts** 文字列は、マクロ呼び出しの開始時に **argc/argv** 処理の **getopt (3)** に渡されます。



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

4.2.2. %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4.1 %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 を展開します。
- 一部のデフォルト権限を設定します。

4.2.2.1. %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。

4.2.2.2. %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
```

4.2.2.3. %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、展開後に、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展開手順に映ります。

```
/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。

4.2.2.4. %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 -
```

4.2.2.5. %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** を意味し、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** を意味し、移動後にそ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
```

以下の例では、サンプルは **cello-1.0-examples.tar.gz** tarball にあり、**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
```

4.2.3. %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 マクロの一覧を示しています。

表4.1 %files セクションの高度な RPM マクロ

マクロ	定義
%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。そしてインストールされ、RPM などとしてラベルが付けられます。例: **%license LICENSE**
%doc	マクロは、ドキュメントとしてリストされるファイルを識別して、RPM によりインストールされ、ラベル付けされます。このマクロは、パッケージソフトウェアに関するドキュメントや、コード例や、付随するさまざまなアイテムに使用されます。コードの例が含まれる場合は、実行ファイルから実行可能モードを削除するように注意してください。例: **%doc README**
%dir	このマクロは、そのパスが、この RPM が所有するディレクトリーとなるようにします。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)	このマクロにより、次のファイルが設定ファイルであることが保証されます。そのため、ファイルを元のインストールチェックサムから修正しても、パッケージのインストールまたは更新で上書き (または置き換え) しないでください。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**

4.2.4. 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。

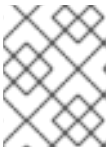
```
rpm --showrc
```

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。

4.2.5. 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

4.2.6. 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。

4.3. 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。

4.3.1. 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 一覧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4.2 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。

4.3.2. 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 ディレクティブの一覧が含まれます。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4.2 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。

4.3.3. 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

4.3.4. 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
```

4.3.5. 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。

4.3.6. 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。

4.4. 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

4.4.1. 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
```

4.4.2. %if 条件

本セクションでは、**%if** RPM 条件の使用例を説明します。

例4.3 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 用に構築されている場合は、**%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4.4 %if 条件を使用したマクロの定義の処理す

```
%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。

4.4.3. %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4.5 %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4.6 %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4.7 %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。

4.5. 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。

4.5.1. Python パッケージ用の SPEC ファイルの説明

SPEC ファイルには、RPM の構築に **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** 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。

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。

4.5.2. 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 のマクロのテーブル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** で、特定の Python 3 バージョンを **python36-rpm-macros**、**python38-rpm-macros**、または **python39-rpm-macros** に設定します。

表4.3 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。

4.5.3. 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。

4.6. 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/python2
```

/usr/lib/rpm/redhat/brp-mangle-shebangs BRP (buildroot policy) スクリプト は、RPM パッケージを構築する際に自動的に実行し、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 スクリプトが発生すると、エラーを作成します。

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

または

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

4.6.1. 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します。

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。

4.6.2. 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

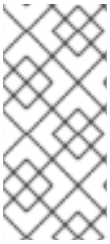
デフォルトでは、**/usr/bin/python3** の形式でのインタープリターディレクティブは、Red Hat Enterprise Linux のシステムツールに使用される**platform-python** パッケージから Python を参照するインタープリターディレクティブに置き換えられます。カスタムパッケージの **/usr/bin/python3** インタープリターディレクティブを変更して、AppStream リポジトリーからインストールした特定バージョンの Python を参照できます。

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
```

4.7. 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。

4.7.1. 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。

4.7.2. 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
```

4.7.3. 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。

4.7.3.1. 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。

4.7.3.2. 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一覧表示します。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4.4 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。

マクロ名	拡張パス	用途
%{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。

4.7.3.3. RubyGems SPEC ファイルの例

本セクションでは、特定のセクションの説明とともに、gem を構築する SPEC ファイルの例を示します。

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4.5 RubyGems' SPEC ディレクティブの詳細

SPEC ディレクティブ	RubyGems の詳細
%prep	RPM は gem アーカイブを直接展開できるため、**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 が展開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。
%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

4.7.3.4. 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

4.7.3.4.1. 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
```

4.7.3.4.2. 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
```

4.7.3.4.3. 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。

4.7.3.4.4. 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4.6 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)。

変数	説明
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一覧を示す **Gem2RPM::RpmDependencyList** 変数。
development_dependencies	パッケージ開発依存関係の一覧を示す **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一覧を示します。
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一覧を提供する **Gem2RPM::RpmFileList** 変数。
doc_files	**-doc** サブパッケージに適したファイルの一覧を提供する **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。

4.7.3.4.5. 利用可能な gem2rpm テンプレートの一覧表示

以下の手順では、利用可能な **gem2rpm** テンプレートの一覧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
```

4.7.3.4.6. 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
```

RPM の構築 の説明に従って、編集したテンプレートを使用し、RPM パッケージを作成できるようになりました。

4.8. 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。

4.8.1. 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、または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。

4.8.2. 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。

4.8.3. 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 以上に制限するには、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。

4.8.4. 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5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。

5.1. 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5.1 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。

5.1.1. 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

5.1.2. 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5.1 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.
> ```

5.1.3. 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パッケージにアタッチできるサードパーティベンダー向けに設計されています。

5.2. 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、ブール値演算子の一覧を紹介して、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。

5.2.1. 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

5.2.2. 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5.2 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5.3 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。

5.2.3. 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5.2 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

5.2.4. 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。

5.2.5. 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5.3 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)
```

5.3. 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、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。

5.3.1. 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つ以上のパスプレフィックスとスクリプトが含まれている必要があります。

5.3.2. ファイルトリガー構文の例

このセクションでは、**ファイルトリガー**構文の具体例を示します。

```
%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。

5.3.3. 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

5.3.3.1. 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。

5.3.3.2. 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。

5.3.4. glibc でのファイルトリガーの使用例

このセクションでは、**glibc** パッケージ内での**ファイルトリガー**の使用の実際の例を示しています。

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。

5.4. 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。

5.5. 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。

5.5.1. 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5.4 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。
RPMTAG_ARCHIVESIZE	RPMTAG_LONGARCHIVESIZE	非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_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。

5.5.2. 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]"
```

5.6. 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

本セクションでは、RPM、RPM のパッケージ化、RPM ビルドに関連するさまざまなトピックの参考資料を紹介します。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。