



Red Hat Enterprise Linux 9

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

Red Hat Enterprise Linux 9 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

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

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)	3
第1章 RPM の概要	4
1.1. RPM パッケージ	4
1.2. RPM パッケージ化ユーティリティーのリスト	5
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	6
2.1. ソースコードとは	6
2.2. ソフトウェアの作成方法	7
2.3. ソースからのソフトウェアのビルド	8
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	12
3.1. ソフトウェアへのパッチの適用	12
3.2. LICENSE ファイルの作成	14
3.3. 配布用ソースコードアーカイブの作成	15
第4章 ソフトウェアのパッケージ化	19
4.1. RPM パッケージ化ワークスペースのセットアップ	19
4.2. SPEC ファイルについて	20
4.3. BUILDRROOTS	24
4.4. RPM マクロ	24
4.5. SPEC ファイルでの作業	24
4.6. RPM のビルド	32
4.7. RPM のサニティーチェック	36
4.8. RPM アクティビティーの SYSLOG へのロギング	40
4.9. RPM コンテンツの抽出	40
第5章 高度なトピック	42
5.1. RPM パッケージへの署名	42
5.2. マクロの詳細	43
5.3. EPOCH、SCRIPTLETS、TRIGGERS	48
5.4. RPM 条件	53
5.5. PYTHON 3 RPM のパッケージ化	55
5.6. PYTHON スクリプトでのインタープリターディレクティブの処理	59
5.7. RUBYGEMS パッケージ	61
5.8. PERL スクリプトで RPM パッケージを処理する方法	66
第6章 RHEL 9 の新機能	69
6.1. 動的ビルドの依存関係	69
6.2. パッチ宣言の改善	69
6.3. その他の機能	70
第7章 関連情報	71

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せん。

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 DNF リポジトリに公開する。

Red Hat Enterprise Linux では、RPM は DNF 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、ソースコードをバイナリー RPM にビルドする方法を記述した **spec** ファイルが含まれています。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。

```
# dnf 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")
```

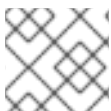
- C で書かれた **Hello World**

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 プログラムを手動ビルド

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**) 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;
}
```

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。

- ソフトウェアをビルドします。

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

- 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.
```

- 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せんが、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で書かれたサンプルの **Hello World** プログラム (`pello.py`) 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ります。ご注意ください。

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 プログラムの直接解釈

`Bash` シェルの組み込み言語で書かれたサンプルの **Hello World** プログラム (`bello`) 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 形式でアーカイブします。

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変更せずにそのままにして、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** ユーティリティを使用してパッチを作成し、**patch** ユーティリティ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の作成

diff ユーティリティを使用して、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 で書かれた **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** ユーティリティをインストールした。

```
# dnf install diffutils
```

手順

- 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。

- 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;
}
```

- 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。

- **diff(1)** man ページ

3.1.2. サンプル C プログラムへのパッチ適用

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

前提条件

- システムに **patch** ユーティリティをインストールした。

```
# dnf install patch
```

- 元のソースコードからパッチを作成した。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](#) 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

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。ソースコードのアーカイブは、後で配布用にパッケージ化するソフトウェアのリリース方法として一般的です。

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の作成

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移動します。

```
$ 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の作成

`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移動します。

```
$ 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/
```

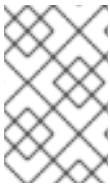
関連情報

- [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の作成

`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移動します。

```
$ 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

4.1. RPM パッケージ化ワークスペースのセットアップ

RPM パッケージをビルドするには、まず、特別なワークスペースを作成する必要があります。このワークスペースは、さまざまなパッケージ化のために使用するディレクトリーで構成されています。

4.1.1. RPM パッケージ化ワークスペースの設定

RPM パッケージ化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できます。

前提条件

- RPM をパッケージ化するためのユーティリティーを提供する **rpmdevtools** パッケージをインストールした。

```
# dnf 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 パッケージ化ワークスペースディレクトリー](#)

4.1.2. RPM パッケージ化ワークスペースディレクトリー

以下は、**rpmdev-setuptree** ユーティリティーで作成される RPM パッケージ化ワークスペースディレクトリーです。

表4.1 RPM パッケージ化ワークスペースディレクトリー

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格納されます。

ディレクトリー	目的
RPMS	バイナリー RPM は、さまざまなアーキテクチャー用のサブディレクトリーの RPMS ディレクトリー配下に作成されます。たとえば、 x86_64 や noarch サブディレクトリーです。
SOURCES	圧縮されたソースコードのアーカイブとパッチが格納されます。 rpmbuild コマンドによって、このディレクトリー内のアーカイブとパッチが検索されます。
SPECS	パッケージャーによって作成された spec ファイルが格納されます。このファイルはパッケージのビルドに使用されます。
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配下に作成されます。

4.2. SPEC 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 パッケージをビルドするために使用する指示が記述されたファイルです。このファイルは、一連のセクションで指示を定義することにより、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、**spec** ファイルの **Preamble** 部分と **Body** 部分で定義されます。

- **Preamble** セクションには、**Body** 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。
- **Body** セクションは、指示の主要部分を表します。

4.2.1. Preamble 項目

以下は、RPM **spec** 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 Preamble セクションのディレクティブ

ディレクティブ	定義
Name	パッケージのベース名。 spec ファイル名と一致する必要があります。
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	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値を増やします。新しい Version のソフトウェアをビルドするときに 1 にリセットします。

ディレクティブ	定義
Summary	パッケージの簡単な一行の概要。
License	<p>パッケージ化するソフトウェアのライセンス。</p> <p>spec ファイルで License にラベルを付ける方法は、GPLv3+ など、準拠する RPM ベースの Linux ディストリビューションのガイドラインによって異なります。</p>
URL	ソフトウェアの詳細情報の完全な URL (パッケージ化するソフトウェアのアップストリームプロジェクトの Web サイトなど)。
Source	<p>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。このリンクの参照先は、パッケージャーのローカルストレージではなく、アクセスしやすく信頼性の高いアーカイブ保存場所 (アップストリームのページなど) である必要があります。</p> <p>Source ディレクティブは、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。</p>
Patch	<p>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。</p> <p>Patch ディレクティブは、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。</p> <p>%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します。または、%autopatch マクロを使用できます。このマクロは、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。</p>
BuildArch	<p>ソフトウェアのビルド対象アーキテクチャー。</p> <p>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ー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、x86_64 など、ビルド対象のマシンのアーキテクチャーをソフトウェアが自動的に継承します。</p>
BuildRequires	コンパイル言語で記述されたプログラムをビルドするために必要なパッケージのコンマまたは空白区切りのリスト。複数の BuildRequires のエントリーを SPEC ファイル内の別の行にそれぞれ含めることができます。
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。複数の Requires のエントリーを spec ファイル内の別の行にそれぞれ含めることができます。

ディレクティブ	定義
ExcludeArch	ソフトウェアが特定のプロセッサアーキテクチャーで動作しない場合は、 ExcludeArch ディレクティブでそ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区切りのリスト。複数の Conflicts のエントリーを spec ファイル内の別の行にそれぞれ含めることができます。
Obsoletes	<p>Obsoletes ディレクティブを指定すると、次の要因に応じて更新の動作が変わります。</p> <ul style="list-style-type: none"> ● rpm コマンドをコマンドラインで直接使用した場合、インストールされているパッケージの古いバージョン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ソルバー (DNF) を使用した場合、一致する Obsoletes: を含んでいるパッケージが更新対象として追加され、一致するパッケージが置き換えられます。
Provides	パッケージに Provides ディレクティブを追加すると、そのパッケージが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、**name-version-release** という形式の RPM パッケージのファイル名で構成されます。

rpm コマンドを使用して RPM データベースをクエリーすることにより、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** は、パッケージアーキテクチャーを示すマーカーです。アーキテクチャーマーカーは、**NVR** とは異なり、RPM パッケージャーが直接制御するものではなく、**rpmbuild** ビルド環境によって定義されます。アーキテクチャーに依存しない **noarch** パッケージは例外です。

4.2.2. Body 項目

以下は、RPM **spec** ファイルの **Body** セクションで使用される項目です。

表4.3 Body 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。

ディレクティブ	定義
%prep	ソフトウェアのビルドを準備するための1つまたは一連のコマンド。たとえば、 Source ディレクティブでのアーカイブの展開などです。 %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	<p>ソフトウェアのビルド後に、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際に使用する1つまたは一連のコマンド。これらのコマンドにより、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と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がコピーされます。これには、~/rpmbuild/BUILD から ~/rpmbuild/BUILDROOT へのファイルのコピーと、~/rpmbuild/BUILDROOT での必要なディレクトリーの作成が含まれます。</p> <p>%install ディレクトリーは、エンドユーザーの root ディレクトリーに似た空の chroot ベースディレクトリーです。ここに、インストールされるファイルを格納する任意の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 マクロを使用できます。</p> <p>%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。</p>
%check	ソフトウェアをテストするための1つまたは一連のコマンド(ユニットテストなど)。
%files	<p>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のリストと、システム上におけるファイル配置場所のフルパスのリスト。</p> <p>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が存在する可能性があるという警告が表示されます。</p> <p>%files セクション内には、組み込みマクロを使用してさまざまなファイルの役割を指定できます。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。</p>
%changelog	異なる Version または Release ビルドの間でパッケージに行われた変更の記録。この変更には、パッケージの各 Version-Release の日付入りエントリーのリストを含めます。これらのエントリーには、ソフトウェアの変更ではなく、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、パッケージの変更を記録します。

4.2.3. 高度な項目

spec ファイルには、[Scriptlets](#) や [Triggers](#) などの高度な項目を含めることができます。

Scriptlets と Triggers は、ビルドプロセスではなく、エンドユーザーのシステムにおける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. BUILROOTS

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設定があります。このデフォルト設定をオーバーライドすると、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、ユーザーによって変わってテキストの置換を行うことができます。

たとえば、パッケージ化するソフトウェアの **Version** を **%{version}** マクロで1回だけ定義し、このマクロを **spec** ファイル全体で使用することができます。該当するすべてのテキストが、マクロで定義した **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}
```

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。

SPEC ファイルは次の方法で作成できます。

- 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トリタイプ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。

次のセクションでは、**Hello World!** プログラムの 3 つの実装例を使用します。

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。ソースコードをバイトコンパイルし、バイトコード (生成される、事前に最適化されたファイル) をインストールする方法を示します。
cello	ネイティブコンパイル言語で書かれたプログラム。ソースコードをマシンコードにコンパイルし、生成される実行可能ファイルをインストールする一般的なプロセスを示します。

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` ディレクトリに置く必要があります。

Hello World! プログラムの実装の詳細は、[ソースコードとは](#) を参照してください。

次のセクションでは、SPEC ファイルの使用方法を説明します。

- [rpmdev-newspec](#) を使用して新しい SPEC ファイルを作成します。
- [RPM](#) を作成するための元の SPEC ファイルを変更します。
- [bash](#)、[Python](#)、および [C](#) で書かれたプログラムの spec ファイルの例を調べます。

4.5.1. サンプルの Bash、C、および Python プログラム用の新しい spec 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 つの実装それぞれに **spec** ファイルを作成できます。

前提条件

- 次の **Hello World!** プログラム実装が、`~/rpmbuild/SOURCES` ディレクトリーに配置されている。
 - [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!** プログラムの 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**、**pello.spec** という 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、[spec ファイルについて](#) で説明されているものです。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. 元の spec 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る元の **spec** 出力ファイルは、テンプレートです。このテンプレートを変更して、**rpmbuild** ユーティリティーに必要な指示を提供する必要があります。**rpmbuild** は、その指示を使用して RPM パッケージをビルドします。

前提条件

- 未入力の `~/rpmbuild/SPECS/<name>.spec` **spec** ファイルを、**rpmdev-newspec** ユーティリティーを使用して作成した。詳細は、[サンプルの Bash、C、および Python プログラム用の新しい spec 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. **spec** ファイルの **Preamble** 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。

Version

Version は、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。

Summary

パッケージの説明を1行で入力します。

License

ソースコードに関連付けるソフトウェアライセンスを入力します。

URL

アップストリームのソフトウェア 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。

Source

アップストリームのソフトウェアソースコードへの URL を入力します。パッケージ化す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、ハードコードされた値が含まれています。この値は将来変更される可能性があります。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、更新する必要がある箇所が、**spec** ファイル内の1つのフィールドだけになります。

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. **spec** 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、実行する複数行、複数指示のタスクやスクリプト処理タスクを定義できるため、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1つまたは一連のコマンドを入力します。

%build

ソフトウェアをビルドするための1つまたは一連のコマンドを入力します。

%install

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 **rpmbuild** コマンドに指示する1つ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 **Version-Release** の日付入り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、アスタリスク (*) 文字で始め、その後に **Day-of-Week Month Day Year Name Surname <email> - Version-Release** を入力します。

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。
- 各項目の先頭にハイフン (-) 文字を付けます。

これで、必要なプログラムの完全な **spec** ファイルが作成されました。

関連情報

- [Preamble 項目](#)
- [Body 項目](#)
- [サンプル Bash プログラムの spec ファイルの例](#)
- [サンプル Python プログラムの spec ファイルの例](#)
- [サンプル C プログラムの spec 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 spec ファイルの例

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 で書かれた bello プログラムの 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。
- パッケージの実行時の依存関係を指定する **Requires** ディレクティブには、**bash** のみが含まれています。これは、**bello** スクリプトの実行には **bash** シェル環境のみが必要であるためです。
- ソフトウェアのビルド方法を指定する **%build** セクションは空白です。これは **bash** スクリプトはビルドする必要がないためです。



注記

bello をインストールするには、インストール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. Python で書かれたプログラムの spec ファイルの例

Python プログラミング言語で書かれた **pello** プログラムの spec ファイルの例を次に示します。

Python で書かれた pello プログラムの spec ファイルの例

```
%global python3_pkgversion 3.11
```

1

```
Name:      python-pello
```

2

```

Version:      1.0.2
Release:      1%{?dist}
Summary:      Example Python library

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

BuildArch:    noarch
BuildRequires: python%{python3_pkgversion}-devel 3

# Build dependencies needed to be specified manually
BuildRequires: python%{python3_pkgversion}-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires: python%{python3_pkgversion}-pytest >= 3

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

%description %_description

%package -n python%{python3_pkgversion}-pello 4
Summary:      %{summary}

%description -n python%{python3_pkgversion}-pello %_description

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

%build
# The macro only supported projects with setup.py
%py3_build 5

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

%check 6
%{pytest}

# Note that there is no %%files section for the unversioned python module
%files -n python%{python3_pkgversion}-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

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

```

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

- 1 **python3_pkgversion** マクロを定義することで、このパッケージがビルドされる Python バージョンを設定します。デフォルトの Python バージョン 3.9 用にビルドするには、マクロをデフォルト値 **3** に設定するか、その行を完全に削除します。
- 2 Python プロジェクトを RPM にパッケージ化するとき、常に **python-** 接頭辞をプロジェクトの元の名前に追加してください。ここでの元の名前は **pello** であるため、ソース RPM (SRPM) の名前は、**python-pello** になります。
- 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** には、Python パッケージのビルドに必要なツールを提供するアイテム **python3-devel** (もしくは **python3.11-devel** または **python3.12-devel**) と、パッケージ化する特定のソフトウェアに必要な関連プロジェクト **python3-setuptools** (もしくは **python3.11-setuptools** または **python3.12-setuptools**)、あるいは **%check** セクションでテストを実行するために必要なランタイムとテストの依存関係を常に含めます。
- 4 バイナリー RPM (ユーザーがインストールできるパッケージ) の名前を選択する際には、バージョン管理された Python 接頭辞を追加します。デフォルトの Python 3.9 の場合は **python3-** 接頭辞、Python 3.11 の場合は **python3.11-** 接頭辞、Python 3.12 の場合は **python3.12-** 接頭辞を使用します。**%{python3_pkgversion}** マクロを使用できます。これは、明示的なバージョン (**3.11** など) に設定しない限り、デフォルトの Python バージョン 3.9 の場合は **3** と評価されます(脚注1を参照)。
- 5 **%py3_build** マクロおよび **%py3_install** マクロは、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追加の引数を使用して、**setup.py build** コマンドおよび **setup.py install** コマンドをそれぞれ実行します。
- 6 **%check** セクションは、パッケージ化されたプロジェクトのテストを実行する必要があります。正確なコマンドはプロジェクト自体に依存しますが、**%pytest** マクロを使用して、RPM に適した方法で **pytest** コマンドを実行することができます。

4.5.5. サンプル C プログラムの spec 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 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ています。
 - **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 **cello** プログラム用の **Makefile** ファイルを作成したことを反映しています。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** への呼び出しを削除する必要があります。

cello プログラムは、**%make_install** マクロを使用してインストールできます。これが可能なのは、**cello** プログラムの **Makefile** を利用できるためです。

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求められます。

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 をビルドします。

- バイナリー RPM をビルドします。
 - ソース RPM からバイナリー RPM を再ビルドします。
 - **spec** ファイルからバイナリー RPM をビルドします。

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特定の **Name-Version-Release** を持つ RPM ファイルの正確なソースを保持できます。これには、正確な **spec** ファイル、ソースコード、および関連するすべての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る。

```
# dnf install rpm-build
```

- 次の **Hello World!** 実装が `~/rpmbuild/SOURCES/` ディレクトリーに配置されている。
 - [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 **spec** ファイルが存在する。

手順

1. 作成した **spec** ファイルが含まれている `~/rpmbuild/SPECS/` ディレクトリーに移動します。

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

2. **rpmbuild** コマンドを入力し、**spec** ファイルを指定して、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/admilller/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
```

```
$ rpmbuild -bs pello.spec  
Wrote: /home/admilller/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
```

```
$ rpmbuild -bs cello.spec
Wrote: /home/admilller/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](#)
- [サンプルの Bash、C、および Python プログラム用の新しい spec ファイルを作成する](#)
- [元の spec 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 を再ビルドするには、**--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、デバッグに役立ちます。出力は例によって異なり、それぞれの **spec** ファイルに対応したものになります。

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る。

```
# dnf install rpm-build
```

手順

1. ソース RPM が含まれている **~/rpmbuild/SRPMS/** ディレクトリーに移動します。

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

2. ソース RPM からバイナリー RPM を再ビルドします。

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

srpm は、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 から再ビルドするには、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** ファイルとソースコード) を **~/rpmbuild/** ディレクトリーにインストールします。
- 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すると、**spec** ファイルとソースコードを保持できます。

- RPM をビルドするときに、**--rebuild** オプションではなく **--recompile** オプションを指定した **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

spec 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る。

```
# dnf install rpm-build
```

手順

1. **spec** ファイルが含まれている **~/rpmbuild/SPECS/** ディレクトリーに移動します。

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

2. **spec** 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ら、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 の保守性を向上する。
- RPM の静的分析を実行してサニティーチェックを可能にする。
- RPM の静的分析を実行してエラーチェックを可能にする。

rpmlint では、バイナリー RPM、ソース RPM (SRPM)、および **spec** 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、次のセクションに示すように、一部のエラーと警告をスキップすることが許容される場合もあります。



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 プログラムのサニティーチェック

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 spec ファイルで **rpmlint** コマンドを実行した場合の出力

```
$ 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 の場合、**invalid-url Source0** という警告が1つだけ出力されます。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。

belo 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
belo.src: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
belo.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.
```

belo SRPM の場合、新しい **invalid-url URL** 警告が出力されます。これは、**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。

4.7.1.2. belo 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、**belo** 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。

belo バイナリー RPM で rpmlint コマンドを実行した結果の出力

```
$ rpmlint ~/rpmbuild/RPMS/noarch/bello-0.1-1.el8.noarch.rpm
belo.noarch: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
belo.noarch: W: no-documentation
belo.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 プログラムのサニティーチェック

次のセクションでは、**pello 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。

4.7.2.1. 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 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**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場合の出力

```
$ 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** という警告は、ドキュメントや man ページが指定されていないため、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以外の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。ファイルを正しく配置するために、RPM マクロを使用してください。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。**rpmlint** ツールは、ファイルにシバン (`#!`) が含まれているため、ファイルが実行可能であると想定します。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 プログラムのサニティーチェック

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てください。

cello spec ファイルで **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 の場合、**invalid-url Source0** という警告が1つだけ出力されます。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、**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、**rpm lint** 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 rpm lint コマンドを実行した場合の出力

```
$ rpm lint ~/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、ドキュメントや man ページが指定されていないため、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 **rpm lint** チェックに合格しています。

4.8. RPM アクティビティの SYSLOG へのロギング

System Logging プロトコル (**syslog**) を使用して、あらゆる RPM アクティビティ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。

```
# dnf 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、メッセージを別の場所に保存するように **syslog** を設定することもできます。

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きます。または、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

場合によってはパッケージのコンテンツを抽出する必要があります。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です。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** ファイルに次の行を含めます。

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

<body> の周りの空白すべてが削除されます。名前には英数字と文字 `_` を使用できます。長さが 3 文字以上である必要があります。**(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** ファイルでは、代わりに **%define <name> <body>** マクロパターンが使用されています。**%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ている場合や、マクロの名前が **spec** ファイルの **%changelog** セクション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エーションを使用して、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。マクロバリエーション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、マクロは以下の例で示すように **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** であるが、ソースコードが **hello-1.0.tgz** にアーカイブされていて、**hello/** ディレクトリーを含んでいる場合、**spec** ファイルの内容を次のようにする必要があります。

```
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 -xvof -
```

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。引数は、**spec** ファイルの Preamble 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}

マクロ	定義
<code>%config(noreplace)</code>	%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

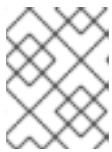
5.2.4. ビルトインマクロの表示

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
```

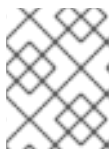


注記

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、**dnf** 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。コア数を変えると、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、RPM `spec` ファイルの高度なディレクティブを表す **Epoch**、**Scriptlets**、および **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、生成される RPM がインストールされ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の**spec**ファイルにない場合、**Epoch** ディレクティブは一切設定されません。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、**dnf** 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ときにRPMの動作が予想と異なるものになるためです。

例5.2 Epoch の使用

Epoch: 1と**Version:1.0**でfoobarパッケージをインストールし、別のユーザーがfoobarを**Version: 2.0**でインストールした場合、**Epoch**ディレクティブがないと、新しいバージョンが更新と見なされることはありません。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。これらのディレクティブは、**%build**や**%install**などの**spec**ファイルのセクションヘッダーに似ています。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。

ディレクティブ	定義
%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 --noprereans
```

--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**
- **--noprereans**
- **--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せん。**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使用すると、ユーザーはデフォルトのシェルスクリプトインタプリター (`-p /bin/sh`) の代わりに特定のインタプリターを呼び出すことができます。

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。

```
# dnf 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** の下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>
```

このようにして、**spec** ファイル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、**spec** ファイルのさまざまなセクション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` 条件の特殊なバリエーション

`%ifarch` 条件、`%ifnarch` 条件、`%ifos` 条件は、`%if` 条件の特殊なバリエーションです。これらのバリエーション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内容は、32 ビット AMD および Intel アーキテクチャーまたは Sun SPARC 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` 条件よりもリバーズ論理があります。

例5.6 `%ifnarch` 条件の使用例

-

```
%ifnarch alpha
...
%endif
```

%ifnarch と **%endif** の間にある **spec** ファイルのすべての内容は、Digital Alpha/AXP ベース以外のシステムでのみ処理されます。

%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
```

%ifos と **%endif** の間にある **spec** ファイルのすべての内容は、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

Python パッケージは、**pip** インストーラーを使用してアップストリームの PyPI リポジトリから、または DNF パッケージマネージャーを使用してシステムにインストールできます。DNF は RPM パッケージ形式を使用します。これにより、ソフトウェアのダウストリーム制御が強化されます。

ネイティブ Python パッケージのパッケージ形式は、[Python Packaging Authority \(PyPA\) 仕様](#) によって定義されています。ほとんどの Python プロジェクトでは、パッケージ化に **distutils** または **setuptools** ユーティリティーを使用し、**setup.py** ファイルでパッケージ情報を定義しています。ただし、ネイティブ Python パッケージ作成の可能性は、時代とともに進化してきています。新しいパッケージング標準の詳細は、[pyproject-rpm-macros](#) を参照してください。

この章では、**setup.py** を使用する Python プロジェクトを RPM パッケージにパッケージ化する方法を説明します。このアプローチには、ネイティブ Python パッケージと比較して次の利点があります。

- Python および Python 以外のパッケージへの依存が可能です。依存関係は **DNF** パッケージマネージャーによって厳密に適用されます。
- パッケージに暗号で署名できます。暗号化署名を使用すると、RPM パッケージのコンテンツを、オペレーティングシステムの他の部分を使用して検証、統合、およびテストできます。
- ビルドプロセス中にテストを実行できます。

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、**rpmbuild** ユーティリティーが RPM をビルドする際に使用する指示を記述します。指示は一連のセクションに記述します。SPEC ファイルには、次の2つの主要な部分があります。各部分に複数のセクションを定義します。

- Preamble (Body で使用される一連のメタデータ項目を記述)

- Body (指示の主要部分を記述)

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。



重要

Python ライブラリーの RPM パッケージの名前には、常に **python3-**、**python3.11-**、または **python3.12-** の接頭辞が含まれている必要があります。

その他の詳細は、以下の SPEC ファイルの **python3*-pello** 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。

Python で書かれた pello プログラムの spec ファイルの例

```
%global python3_pkgversion 3.11 1

Name:      python-pello 2
Version:   1.0.2
Release:   1%{?dist}
Summary:   Example Python library

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

BuildArch: noarch
BuildRequires: python%{python3_pkgversion}-devel 3

# Build dependencies needed to be specified manually
BuildRequires: python%{python3_pkgversion}-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires: python%{python3_pkgversion}-pytest >= 3

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

%description %_description

%package -n python%{python3_pkgversion}-pello 4
Summary:    %{summary}

%description -n python%{python3_pkgversion}-pello %_description

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

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

```
%py3_build
```

5

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

```
%check
%{pytest}
```

6

```
# Note that there is no %%files section for the unversioned python module
%files -n python%{python3_pkgversion}-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

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

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

- 1 **python3_pkgversion** マクロを定義することで、このパッケージがビルドされる Python バージョンを設定します。デフォルトの Python バージョン 3.9 用にビルドするには、マクロをデフォルト値 **3** に設定するか、その行を完全に削除します。
- 2 Python プロジェクトを RPM にパッケージ化するときは、常に **python-** 接頭辞をプロジェクトの元の名前に追加してください。ここでの元の名前は **pello** であるため、ソース RPM (SRPM) の名前は、**python-pello** になります。
- 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** には、Python パッケージのビルドに必要なツールを提供するアイテム **python3-devel** (もしくは **python3.11-devel** または **python3.12-devel**) と、パッケージ化する特定のソフトウェアに必要な関連プロジェクト **python3-setuptools** (もしくは **python3.11-setuptools** または **python3.12-setuptools**)、あるいは **%check** セクションでテストを実行するために必要なランタイムとテストの依存関係を常に含めます。
- 4 バイナリー RPM (ユーザーがインストールできるパッケージ) の名前を選択する際には、バージョン管理された Python 接頭辞を追加します。デフォルトの Python 3.9 の場合は **python3-** 接頭辞、Python 3.11 の場合は **python3.11-** 接頭辞、Python 3.12 の場合は **python3.12-** 接頭辞を使用します。**%{python3_pkgversion}** マクロを使用できます。これは、明示的なバージョン (**3.11** など) に設定しない限り、デフォルトの Python バージョン 3.9 の場合は **3** と評価されます(脚注1を参照)。
- 5 **%py3_build** マクロおよび **%py3_install** マクロは、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追加の引数を使用して、**setup.py build** コマンドおよび **setup.py install** コマンドをそれぞれ実行します。
- 6 **%check** セクションは、パッケージ化されたプロジェクトのテストを実行する必要があります。正確なコマンドはプロジェクト自体に依存しますが、**%pytest** マクロを使用して、RPM に適した方法で **pytest** コマンドを実行することができます。

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。SPEC ファイルの上に `python3_pkgversion` マクロを定義することで、これらのマクロで使用する Python 3 バージョンを再定義できます(「[Python パッケージ用の SPEC ファイルの説明](#)」を参照)。`python3_pkgversion` マクロを定義すると、以下の表で説明されているマクロの値は、指定された Python 3 バージョンを反映します。

表5.3 Python 3 RPM 用のマクロ

マクロ	一般的な定義	説明
<code>%{python3_pkgversion}</code>	3	他のすべてのマクロで使用される Python バージョン。Python 3.11 を使用するには 3.11 に再定義し、Python 3.12 を使用するには 3.12 に再定義できます。
<code>%{python3}</code>	<code>/usr/bin/python3</code>	Python3 インタープリター
<code>%{python3_version}</code>	3.9	Python3 インタープリターの major.minor バージョン
<code>%{python3_sitelib}</code>	<code>/usr/lib/python3.9/site-packages</code>	pure-Python モジュールがインストールされている場所
<code>%{python3_sitelib64}</code>	<code>/usr/lib64/python3.9/site-packages</code>	アーキテクチャー固有の拡張モジュールを含むモジュールがインストールされている場所
<code>%py3_build</code>		RPM 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。
<code>%py3_install</code>		RPM 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。
<code>%{py3_shebang_flags}</code>	s	Python インタープリターディレクティブマクロのデフォルトのフラグセット %py3_shebang_fix
<code>%py3_shebang_fix</code>		Python インタープリターディレクティブを #! %{python3} に変更すると、既存のフラグ (見つかった場合) を保持し、 %{py3_shebang_flags} マクロで定義されたフラグを追加します。

関連情報

- [アップストリームドキュメントの Python マクロ](#)

5.5.3. Python RPM の自動生成された依存関係の使用

次の手順では、Python プロジェクトを RPM としてパッケージ化するときに自動生成された依存関係を使用する方法を説明します。

前提条件

- RPM の SPEC ファイルが存在する。詳細は、[Python パッケージの SPEC ファイルの説明](#) を参照してください。

手順

1. アップストリームで提供されるメタデータを含む次のディレクトリーのいずれかが、結果の RPM に含まれていることを確認します。

- **.dist-info**

- **.egg-info**

RPM ビルドプロセスは、これらのディレクトリーから仮想 **pythonX.Ydist Provides** を自動的に生成します。次に例を示します。

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

次に、Python 依存関係ジェネレーターはアップストリームメタデータを読み取り、生成された **pythonX.Ydist** 仮想 Provides を使用して各 RPM パッケージのランタイム要件を生成します。たとえば、生成された要件タグは次のようになります。

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

2. 生成された require を検査します。
3. 生成された require の一部を削除するには、次のいずれかの方法を使用します。
 - a. SPEC ファイルの **%prep** セクションでアップストリーム提供のメタデータを変更します。
 - b. [アップストリームドキュメント](#) で説明されている依存関係の自動フィルタリングを使用します。
4. 自動依存関係ジェネレーターを無効にするには、メインパッケージの **%description** 宣言の上に **%{?python_disable_dependency_generator}** マクロを含めます。

関連情報

- [Automatically generated dependencies](#)

5.6. PYTHON スクリプトでのインタープリターディレクティブの処理

Red Hat Enterprise Linux 9 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/python3.9
#!/usr/bin/python3.11
#!/usr/bin/python3.12
```

/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 スクリプトでのインタープリターディレクティブの変更

次の手順を使用して、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。
 - SPEC ファイルの `%prep` セクションで次のマクロを使用します。

```
# %py3_shebang_fix SCRIPTNAME ...
```

`SCRIPTNAME` には、任意のファイル、ディレクトリー、またはファイルおよびディレクトリーのリストを指定できます。

結果として、リストしたすべてのファイルと、リストしたディレクトリー内のすべての `.py` ファイルのインタープリターディレクティブが、`%{python3}` を指すように変更されます。元のインタープリターディレクティブの既存のフラグは保持され、`%{py3_shebang_flags}` マクロで定義された追加のフラグが追加されます。SPEC ファイルの `%{py3_shebang_flags}` マクロを再定義すると、追加されるフラグを変更できます。

- `python3-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 ...
```

複数のパスを指定できます。`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。上記のコマンドを `%prep` セクションまたは `%install` セクションの最後に追加します。

- 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、RPM ビルドプロセスの外部で `pathfix.py` スクリプトを使用することもできます。`pathfix.py` を RPM ビルド以外で実行する場合は、前述の例の `%{python3}` を、`/usr/bin/python3` または `/usr/bin/python3.11` 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。

関連情報

- [Interpreter invocation](#)

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 では、**spec** ファイル、パッケージ名、依存関係、その他の項目など、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 ファイルの規則

RubyGems **spec** ファイルは、次の規則を満たしている必要があります。

- 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。

マクロ名	拡張パス	用途
% {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

以下に、gem をビルドするための **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** ファイルに含まれる特定の項目の詳細を説明しています。

表5.5 RubyGems の spec のディレクティブ詳細

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	<p>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。</p> <p>%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。</p> <p>%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。</p>
%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 を使用して RubyGems パッケージを RPM spec ファイルに変換する

gem2rpm ユーティリティは、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** ファイルに変換する
- **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 ファイルに変換する

次の手順では、**gem2rpm** ユーティリティを使用して RubyGems パッケージを RPM **spec** ファイルに変換する方法について説明します。

手順

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

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

上記の手順では、gem のメタデータで提供される情報に基づいて 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 Gem2RPM::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。

変数	説明
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** ファイルの生成元の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。したがって、Perl スクリプトを含む RPM パッケージでは、RPM **spec** ファイルの **BuildRequires:** ディレクティブを使用して、Perl への依存関係を明示的に示す必要があります。

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** ファイルに次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** ファイルで、必要なバージョン制約を指定した **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 5.30 以上に制限するには、以下を使用します。

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



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パッケージの RPM **spec** ファイルに、バージョン管理された **MODULE_COMPAT Requires** を含めます。

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


第6章 RHEL 9 の新機能

本セクションでは、Red Hat Enterprise Linux 8 および 9 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. 動的ビルドの依存関係

Red Hat Enterprise Linux 9 では、動的ビルド依存関係の生成を可能にする `%generate_buildrequires` セクションが導入されています。

追加のビルド依存関係は、RPM のビルド時に、新しく利用可能になった `%generate_buildrequires` スクリプトを使用してプログラムで生成できるようになりました。これは、特殊なユーティリティーが、Rust、Golang、Node.js、Ruby、Python、Haskell などのランタイム依存関係またはビルド時依存関係を判断するために、一般的に使用される言語で記述されたソフトウェアをパッケージ化する場合に役立ちます。

`%generate_buildrequires` スクリプトを使用して、ビルド時に SPEC ファイルに追加される **BuildRequires** ディレクティブを動的に判別できます。存在する場合、`%generate_buildrequires` は `%prep` セクションの後に実行され、解凍およびパッチされたソースファイルにアクセスできます。スクリプトは、通常の **BuildRequires** ディレクティブと同じ構文を使用して、見つかったビルドの依存関係を標準出力に出力する必要があります。

次に、`rpmbuild` ユーティリティーは、ビルドを続行する前に、依存関係が満たされているかどうかを確認します。

一部の依存関係が欠落している場合は、`.buildreqs.nosrc.rpm` 接尾辞が付いたパッケージが作成されます。このパッケージには、見つかった **BuildRequires** が含まれ、ソースファイルは含まれていません。このパッケージを使用して、ビルドを再開する前に、`dnf builddep` コマンドで不足しているビルドの依存関係をインストールできます。

詳細は、`rpmbuild(8)` の man ページの **DYNAMIC BUILD DEPENDENCIES** セクションを参照してください。

関連情報

- `rpmbuild(8)` man ページ
- `yum-builddep(1)` の man ページ

6.2. パッチ宣言の改善

6.2.1. オプションの自動パッチとソースのナンバリング

番号のない **Patch:** タグおよび **Source:** タグは、リスト表示されている順序に基づいて自動的に番号が付けられるようになりました。

番号付けは、最後に手動で番号が付けられたエントリーから開始して、`rpmbuild` ユーティリティーによって内部的に実行します。そのようなエントリーがない場合は **0** になります。

以下に例を示します。

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

6.2.2. %patchlist および %sourcelist セクション

新しく追加した **%patchlist** セクションおよび **%sourcelist** セクションを使用して、各項目の前に各 **Patch:** タグおよび **Source:** タグを付けずに、パッチファイルおよびソースファイルのリストを表示できるようになりました。

たとえば、次のエントリは、

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

次のように置き換えることができるようになりました。

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

6.2.3. %autopatch がパッチの範囲を受け入れる

%autopatch マクロで、適用する最小パッチ番号と最大パッチ番号をそれぞれ制限する **-m** パラメーターと **-M** パラメーターが使用できるようになりました。

- **-m** パラメーターは、パッチを適用するときに開始するパッチ番号 (両端を含む) を指定します。
- **-M** パラメーターは、パッチを適用するときに停止するパッチ番号 (両端を含む) を指定します。

この機能は、特定のパッチセット間でアクションを実行する必要がある場合に役立ちます。

6.3. その他の機能

Red Hat Enterprise Linux 9 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。

- 高速なマクロベースの依存関係ジェネレータ
- 三項演算子とネイティブバージョンの比較を含む、強力なマクロ式および **%if** 式
- メタ (順不同) な依存関係
- キャレットバージョン演算子 (^)。これは、ベースバージョンよりも高いバージョンを表すのに使用できます。この演算子は、反対のセマンティクスを持つチルダ (~) 演算子を補完します。
- **%elif**、**%elifos**、および **%elifarch** ステートメント

第7章 関連情報

ここでは、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の参考資料を紹介します。

- [Mock](#)
- [RPM ドキュメント](#)
- [RPM4.15.0 リリースノート](#)
- [RPM4.16.0 リリースノート](#)
- [Fedora パッケージングガイドライン](#)