



Red Hat Enterprise Linux 7

RPM パッケージングガイド

RPM パッケージマネージャーを使用した基本および高度なソフトウェアパッケージングシナリオ

Red Hat Enterprise Linux 7 RPM パッケージングガイド

RPM パッケージマネージャーを使用した基本および高度なソフトウェアパッケージングシナリオ

Marie Doleželová

Red Hat Customer Content Services

mdolezel@redhat.com

Maxim Svistunov

Red Hat Customer Content Services

Adam Miller

Red Hat

Adam Kvítek

Red Hat Customer Content Services

Petr Kovář

Red Hat Customer Content Services

Miroslav Suchý

Red Hat

Customer Content Services rhel-notes@redhat.com

法律上の通知

Copyright © 2023 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 パッケージングガイドでは、RPM へのソフトウェアパッケージングについて説明します。また、パッケージ化するソースコードを準備する方法も説明します。最後に、このガイドでは、選択した高度なパッケージ化シナリオについて説明しています。

目次

第1章 RPM のパッケージ化の使用	3
1.1. RPM のパッケージ化の概要	3
1.2. RPM の利点	3
1.3. 最初の RPM パッケージ作成	3
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	5
2.1. ソースコードとは	5
2.2. プログラムが作られる仕組み	6
2.3. ソースからのソフトウェア構築	7
2.4. ソフトウェアへのパッチの適用	10
2.5. 任意のアーティファクトのインストール	12
2.6. パッケージ化を行うためのソースコードの準備	14
2.7. ソースコードを TARBALL へ追加	15
第3章 ソフトウェアのパッケージ化	18
3.1. RPM パッケージ	18
3.2. SPEC ファイルでの作業	23
3.3. RPM のビルド	31
3.4. RPM のサニティーチェック	34
第4章 高度なトピック	39
4.1. パッケージの署名	39
4.2. マクロの詳細	41
4.3. EPOCH、SCRIPTLETS、TRIGGERS	46
4.4. RPM 条件	51
付録A RHEL 7 の RPM の新機能	54
第5章 RPM のパッケージ化の関連情報	55

第1章 RPM のパッケージ化の使用

次のセクションでは、RPM パッケージ化の概念とその主な利点を説明します。

1.1. RPM のパッケージ化の概要

RPM Package Manager (RPM) は、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。

1.2. RPM の利点

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パッケージへのパッケージ化。
- **Yum** リポジトリ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。

1.3. 最初の RPM パッケージ作成

RPM パッケージの作成は複雑になる可能性があります。以下では、複数のことをスキップし、簡素化した完全で利用できる RPM Spec ファイルです。

```
Name:    hello-world
Version:  1
Release:  1
Summary:  Most simple RPM package
License:  FIXME

%description
This is my first RPM package, which does nothing.

%prep
# we have no source, so nothing here

%build
cat > hello-world.sh <<EOF
#!/usr/bin/bash
echo Hello world
```

```
EOF
```

```
%install
mkdir -p %{buildroot}/usr/bin/
install -m 755 hello-world.sh %{buildroot}/usr/bin/hello-world.sh

%files
/usr/bin/hello-world.sh

%changelog
# let's skip this for now
```

このファイルを **hello-world.spec** として保存します。

ここで、以下のコマンドを使用します。

```
$ rpmdev-setuptree
$ rpmbuild -ba hello-world.spec
```

コマンド **rpmdev-setuptree** は、複数の作業ディレクトリーを作成します。これらのディレクトリーは \$HOME に永続的に格納されているため、このコマンドを再び使用する必要はありません。

コマンド **rpmbuild** は、実際の rpm パッケージを作成します。このコマンドの出力は、以下のようになります。

```
... [SNIP]
Wrote: /home/<username>/rpmbuild/SRPMS/hello-world-1-1.src.rpm
Wrote: /home/<username>/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.wgaJzv
+ umask 022
+ cd /home/<username>/rpmbuild/BUILD
+ /usr/bin/rm -rf /home/<username>/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
+ exit 0
```

/home/<username>/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm ファイルが、最初の RPM パッケージです。これはシステムにインストールしてテストできます。

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。

2.1.1. ソースコードの例

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。

2.1.1.1. 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"
```

2.1.1.2. 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")
```

2.1.1.3. 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って解釈される。

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、名前が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

このセクションでは、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。

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。

2.3.1. ネイティブにコンパイルされたコード

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합니다。

cello.c

```
#include <stdio.h>

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

2.3.1.1. 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. 生成された出力バイナリー **cello** を実行します。

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

2.3.1.2. 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。

2.3.2. 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。

2.3.2.1. 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
```

2.3.2.2. 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
```

2.4. ソフトウェアへのパッチの適用

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。

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。

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。後続のセクションで RPM を作成するときにパッチを適用します。[「SPEC ファイルでの作業」](#) を参照してください。

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。

- 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;
}
```

- 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。

- 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
```

- 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。

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!
```

2.5. 任意のアーティファクトのインストール

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。

本セクションでは、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」](#)

2.5.1. install コマンドの使用

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
```

2.5.2. make install コマンドの使用

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ファ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
```

2.6. 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

- **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は、[こちら](#)にあります。

2.7. ソースコードを **TARBALL** へ追加

本セクションでは、「[ソースコードの例](#)」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。

2.7.1. 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。

2.7.2. 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。

2.7.3. 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。



注記

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ません。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

3.1. RPM パッケージ

このセクションでは、RPM パッケージ形式の基本を説明します。

3.1.1. RPM とは

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** アーカイブで設定されています。

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。

3.1.2. 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示しています。

前提条件

RPM パッケージ化ツールを使用できるようにするには、**rpmdevtools** パッケージをインストールする必要があります。

```
# 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
```

追加情報

- 上記のユーティリティの詳細は、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。

3.1.3. 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 パッケージがシステムにインストールされている必要があります。

```
# yum install rpmdevtools
```

手順

- **rpmdev-setuptree** ユーティリティを実行します。

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/<username>/rpmbuild/
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS

5 directories, 0 files
```

作成されたディレクトリーは、以下の目的で使します。

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。

3.1.4. SPEC ファイルの概要

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。

3.1.4.1. 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	<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>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。</p> <p>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。</p> <p>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。</p>

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設定されます。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-4.2.46-34.el7.x86_64
```

ここでは、**bash** がパッケージ名で **4.2.46** がバージョン、**34.el7** がリリースです。最後のマーカの **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。

3.1.4.2. 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 ファイルでの作業 」を参照してください。
%check	ソフトウェアをテストする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。

3.1.4.3. 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。

3.1.5. 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 **dhcpcd** 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

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。{RH} 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。

3.1.6. 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には、**%{?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
```

3.2. SPEC ファイルでの作業

このセクションでは、SPEC ファイルを作成して変更する方法を説明します。

前提条件

このセクションでは、「**ソースコードの例**」で説明された **Hello World!** 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。

3.2.1. 新しい SPEC ファイルを作成する方法

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。

これをアーカイブするには、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。

3.2.2. 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。

fd。ファイルを調べます。

+



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。

3.2.3. 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

以下の点を確認してください。

- 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**
 - **Version**

- **Release**

- **Summary**

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

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 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 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ロールを示すことができます。これは、`command[]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。

- 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。

異なるプログラミング言語で書かれた SPEC ファイルの例は、以下を参照してください。

3.2.4. 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 で書かれた **bello** プログラムの SPEC ファイルの例を示しています。**bello** の詳細は「[ソースコードの例](#)」を参照してください。

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。

3.2.5. 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。**pello** の詳細は「[ソースコードの例](#)」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。

パッケージのビルドタイム依存関係を指定する **BuildRequires** ディレクティブには、以下の 2 つのパッケージが含まれます。

- **python** パッケージが、バイトコンパイルのビルドプロセスを実行する必要がある。
- 小規模なエンタ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。

パッケージのランタイム依存関係を指定する **Requires** ディレクティブには、**python** パッケージのみが含まれます。**pello** プログラムは、実行時にバイトコンパイルコードを実行するために **python** パッケージを必要とします。

%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と一致します。

3.2.6. 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。**cello** の詳細は「ソースコードの例」を参照してください。

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-newspect** 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-newspect** 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。

3.3. RPM のビルド

本セクションでは、プログラム用の SPEC ファイルを作成した後に RPM を構築する方法を説明します。

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。本セクションでは、主なユースケースを2つ説明します。

- ソース RPM のビルド
- バイナリー RPM のビルド

3.3.1. ソース RPM のビルド

この段落は、手順モジュールの紹介 (手順の簡単な説明) です。

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。SPEC ファイルの作成方法は [SPEC ファイルでの作業](#) を参照してください。

手順

次の手順では、ソース RPM のビルド方法を説明します。

- 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/<username>/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm

$ rpmbuild -bs pello.spec
Wrote: /home/<username>/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm

$ rpmbuild -bs cello.spec
Wrote: /home/<username>/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。

3.3.2. バイナリー RPM のビルド

バイナリー RPM のビルドには、以下の方法を使用できます。

- 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

3.3.2.1. 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。

3.3.2.2. 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
```

3.3.2.3. ソース RPM からの RPM のビルド

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

ソース RPM から RPM をビルドする方法は、man ページの **rpmbuild(8)** の **BUILDING PACKAGES** セクションを参照してください。

3.4. 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します。

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、以下の例で示すように、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

以下の例では、**rpmlint** はオプションを指定せずに実行し、詳細でない出力を生成します。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。

3.4.1. 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。

3.4.1.1. bello 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 bello 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。

3.4.1.2. 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。

3.4.2. 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。

3.4.2.1. pello 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 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.
```

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。

3.4.2.2. pello 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。

3.4.3. 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。

3.4.3.1. cello の SPEC ファイルの確認

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

```
$ rpmlint ~/rpmbuild/SPECS/cello.spec
/home/<username>/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。

3.4.3.2. 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。

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。

4.1.1. GPG キーの作成

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
```



注記

<Key_name> の代わりにキーに対して選択した実際の名前を含めます。

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. 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。

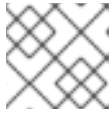
パッケージに署名を追加するには、**rpm -sign** パッケージで利用できる **--addsign** を指定します。

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。

4.1.3. 複数の署名のあるパッケージの署名の確認

手順

- 複数の署名を持つパッケージの署名を確認するには、以下のコマンドを実行します。

```
$ 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。

4.1.4. 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。

4.1.5. 既存のパッケージの署名の置き換え

この手順では、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。

4.1.6. ビルド時のパッケージの署名

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>
```

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

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



注記

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

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



重要

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

関連情報

マクロ機能に関する包括的な情報は、[RPM ドキュメント](#) を参照してください。

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。マクロの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` マクロの標準的な動作を示しています。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 -xvof -
```

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

このセクションでは、SPEC ファイルの **%files** セクションで必要となる高度な RPM マクロを紹介します。

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}

マクロ	定義
%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

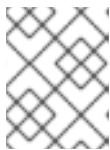
4.2.4. ビルトインマクロの表示

複数のビルトイン RPM マクロを指定します。

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。

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]
```

詳細は **rpm** の man ページを参照してください。

4.2.5.1. 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。

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。

4.3. 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**、**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

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。

4.3.2.1. Scriptlets ディレクティブ

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。

ディレクティブ	定義
%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。

4.3.2.2. スクリプトレット実行の無効化

スクリプトレットの実行を無効にするには、**rpm** コマンドに **--no_scriptlet_name_** オプションを指定して使用します。

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

- 詳細は、man ページの **rpm(8)** を参照してください。

4.3.2.3. 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

```

4.3.3. 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、of Triggers の使用は最小限に抑えることを推奨します。

実行の順序と、既存の各 Triggers の詳細を以下に示します。

```

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。

4.3.4. 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プリター (**-p /bin/sh**) の代わりに特定のインタ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. 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
```

5. 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. RPM 条件例

このセクションでは、RPM 条件の例を複数示します。

4.4.2.1. %if 条件

例4.3 RHEL 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 条件を使用

```
%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

```
%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。

4.4.2.2. %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。

4.4.2.2.1. %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。

4.4.2.2.2. %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。

4.4.2.2.3. %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。

付録A RHEL 7 の RPM の新機能

このリストでは、Red Hat Enterprise Linux 6 と 7 の間の RPM パッケージ化における最も重要な変更が記載されています。

- キーリングのインポートおよび署名検証に使用される新しいコマンド **rpmkeys** が追加されました。
- **spec** クエリーおよび解析出力に使用される新しいコマンド **rpmspec** が追加されました。
- パッケージ署名に使用される新しいコマンド **rpmsign** が追加されました。
- **posix.fork()** スクリプトレットで作成された子プロセスから呼び出されない限り、**%{lua:...}** スクリプトに埋め込まれた **posix.exec()** および **os.exit()** エクステンションはスクリプトに失敗します。
- **%pretrans** スクリプトレットの失敗により、パッケージのインストールはスキップされます。
- スクリプトレットは、ランタイム時に展開されたマクロおよびクエリー形式が考えられます。
- トランザクション前およびトランザクション後のスクリプトレット依存関係は、**Requires(pretrans)** および **Requires(posttrans)** スクリプトレットで正確に表記されるようになっています。
- 追加の順序付けヒントを提供するための **OrderWithRequires** タグが追加されました。タグは **Requires** タグ構文に従いますが、実際の依存関係を生成しません。順序付けヒントは、関係するパッケージが同じトランザクションに存在する場合のみ、トランザクションの順序を計算する際に **Requires** であるかのように処理されます。
- **%license** フラグは、**%files** セクションで使用できます。このフラグは、**%doc** フラグと同様に使用して、ファイルをライセンスとしてマークできます。これは、**--nodocs** オプションを指定してもインストールする必要があります。
- パッチアプリケーションの自動化用の **%autosetup** マクロが、オプションの分散バージョン制御システム統合とともに追加されました。
- 自動依存関係ジェネレーターは、ビルトインフィルター付きで、拡張可能かつカスタマイズ可能なルールベースのシステムに書き換えられました。
- OpenPGP V3 公開鍵に対応しなくなりました。

第5章 RPM のパッケージ化の関連情報

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。