14

PACKAGE MANAGEMENT

If we spend any time in the Linux community, we hear many opinions as to which of the many Linux distributions is "best." Often, these discussions get really silly, focusing on such things as the prettiness of the desktop background (some people won't use Ubuntu because of its default color scheme!) and other trivial matters.

The most important determinant of distribution quality is the *packaging system* and the vitality of the distribution's support community. As we spend more time with Linux, we will see that its software landscape is extremely dynamic. Things are constantly changing. Most of the top-tier Linux distributions release new versions every six months and many individual program updates every day. To keep up with this blizzard of software, we need good tools for *package management*.

Package management is a method of installing and maintaining software on the system. Today, most people can satisfy all of their software needs by installing *packages* from their Linux distributor. This contrasts with the early days of Linux, when one had to download and compile *source code* to install software. There isn't anything wrong with compiling source code; in fact, having access to source code is the great wonder of Linux. It gives us (and everybody else) the ability to examine and improve the system. It's just that having a precompiled package is faster and easier to deal with.

In this chapter, we will look at some of the command line tools used for package management. While all the major distributions provide powerful and sophisticated graphical programs for maintaining the system, it is important to learn about the command line programs, too. They can perform many tasks that are difficult (or impossible) to do with their graphical counterparts.

Packaging Systems

Different distributions use different packaging systems, and as a general rule, a package intended for one distribution is not compatible with another distribution. Most distributions fall into one of two camps of packaging technologies: the Debian *.deb* camp and the Red Hat *.rpm* camp. There are some important exceptions such as Gentoo, Slackware, and Arch, but most others use one of these two basic systems, as shown in Table 14-1.

Table 14-1: Major Packaging System Families

Packaging system	Distributions (partial listing)
Debian-style (.deb)	Debian, Ubuntu, Linux Mint, Raspbian
Red Hat–style (.rpm)	Fedora, CentOS, Red Hat Enterprise Linux, OpenSUSE

How a Package System Works

The method of software distribution found in the proprietary software industry usually entails buying a piece of installation media such as an "install disk" or visiting each vendor's website and downloading the product and then running an "installation wizard" to install a new application on the system.

Linux doesn't work that way. Virtually all software for a Linux system will be found on the Internet. Most of it will be provided by the distribution vendor in the form of *package files*, and the rest will be available in source code form that can be installed manually. We'll talk about how to install software by compiling source code in Chapter 23.

Package Files

The basic unit of software in a packaging system is the *package file*. A package file is a compressed collection of files that comprise the software package. A package may consist of numerous programs and data files that support the programs. In addition to the files to be installed, the package file also includes metadata about the package, such as a text description of the package and its contents. Additionally, many packages contain pre- and post-installation scripts that perform configuration tasks before and after the package installation.

Package files are created by a person known as a *package maintainer*, often (but not always) an employee of the distribution vendor. The package maintainer gets the software in source code form from the *upstream provider* (the author of the program), compiles it, and creates the package metadata and any necessary installation scripts. Often, the package maintainer will apply modifications to the original source code to improve the program's integration with the other parts of the Linux distribution.

Repositories

While some software projects choose to perform their own packaging and distribution, most packages today are created by the distribution vendors and interested third parties. Packages are made available to the users of a distribution in central repositories that may contain many thousands of packages, each specially built and maintained for the distribution.

A distribution may maintain several different repositories for different stages of the software development life cycle. For example, there will usually be a "testing" repository that contains packages that have just been built and are intended for use by brave souls who are looking for bugs be-

fore the packages are released for general distribution. A distribution will often have a “development” repository where work-in-progress packages destined for inclusion in the distribution’s next major release are kept.

A distribution may also have related third-party repositories. These are often needed to supply software that, for legal reasons such as patents or DRM anti-circumvention issues, cannot be included with the distribution. Perhaps the best known case is that of encrypted DVD support, which is not legal in the United States. The third-party repositories operate in countries where software patents and anti-circumvention laws do not apply. These repositories are usually wholly independent of the distribution they support, and to use them, one must know about them and manually include them in the configuration files for the package management system.

Dependencies

Programs are seldom “stand alone”; rather, they rely on the presence of other software components to get their work done. Common activities, such as input/output, for example, are handled by routines shared by many programs. These routines are stored in *shared libraries*, which provide essential services to more than one program. If a package requires a shared resource such as a shared library, it is said to have a *dependency*. Modern package management systems all provide some method of *dependency resolution* to ensure that when a package is installed, all of its dependencies are installed, too.

High- and Low-Level Package Tools

Package management systems usually consist of two types of tools.

- Low-level tools that handle tasks such as installing and removing package files
- High-level tools that perform metadata searching and dependency resolution

In this chapter, we will look at the tools supplied with Debian-style systems (such as Ubuntu and many others) and those used by Red Hat products. While all Red Hat–style distributions rely on the same low-level program (`rpm`), they use different high-level tools. For our discussion, we will cover the high-level program `yum`, used by Red Hat Enterprise Linux and CentOS. Other Red Hat–style distributions provide high-level tools with comparable features (see Table 14-2).

Table 14-2: Packaging System Tools

Distributions	Low-level tools	High-level tools
Debian-style	`dpkg`	`apt-get`, `apt`, `aptitude`
Fedora, Red Hat Enterprise Linux, CentOS	`rpm`	`yum`, `dnf`

Common Package Management Tasks

Many operations can be performed with the command line package management tools. We will look at the most common. Be aware that the low-level tools also support the creation of package files, an activity outside the scope of this book.

In the discussion that follows, the term *`package_name`* refers to the actual name of a package rather than the term *`package_file`*, which is the name of the file that contains the package.

Finding a Package in a Repository

Using the high-level tools to search repository metadata, a package can be located based on its name or description (see Table 14-3).

Table 14-3: Package Search Commands

Style	Command(s)
Debian	`apt-get update` `apt-cache search` *`search_string`*
Red Hat	`yum search` *`search_string`*

For example, to search a `yum` repository for the `emacs` text editor, we can use this command:

```
yum search emacs
```

Installing a Package from a Repository

High-level tools permit a package to be downloaded from a repository and installed with full dependency resolution (see Table 14-4).

Table 14-4: Package Installation Commands

Style	Command(s)
Debian	`apt-get update` `apt-get install` *`package_name`*
Red Hat	`yum install` *`package_name`*

For example, to install the `emacs` text editor from an `apt` repository on a Debian system, we can use this command:

```
apt-get update; apt-get install emacs
```

Installing a Package from a Package File

If a package file has been downloaded from a source other than a repository, it can be installed directly (though without dependency resolution)

using a low-level tool (see Table 14-5).

Table 14-5: Low-Level Package Installation Commands

Style	Command(s)
Debian	`dpkg -i` *`package_file`*
Red Hat	`rpm -i` *`package_file`*

For example, if the *emacs-22.1-7.fc7-i386.rpm* package file had been downloaded from a non-repository site, it would be installed this way:

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

NOTE

Because this technique uses the low-level `rpm` program to perform the installation, no dependency resolution is performed. If `rpm` discovers a missing dependency, `rpm` will exit with an error.

Removing a Package

Packages can be uninstalled using either the high-level or low-level tools. Table 14-6 lists the high-level tools.

Table 14-6: Package Removal Commands

Style	Command(s)
Debian	`apt-get remove` *`package_name`*
Red Hat	`yum erase` *`package_name`*

For example, to uninstall the `emacs` package from a Debian-style system, we can use this command:

```
apt-get remove emacs
```

Updating Packages from a Repository

The most common package management task is keeping the system up-to-date with the latest versions of packages. The high-level tools can perform this vital task in a single step (see Table 14-7).

Table 14-7: Package Update Commands

Style	Command(s)
Debian	`apt-get update; apt-get upgrade`
Red Hat	`yum update`

For example, to apply all available updates to the installed packages on a Debian-style system, we can use this command:

```
apt-get update; apt-get upgrade
```

Upgrading a Package from a Package File

If an updated version of a package has been downloaded from a non-repository source, it can be installed, replacing the previous version (see Table 14-8).

Table 14-8: Low-Level Package Upgrade Commands

Style	Command(s)
Debian	`dpkg -i` *`package_file`*

Style	Command(s)
Red Hat	`rpm -U package_file`

For example, to update an existing installation of `emacs` to the version contained in the package file *emacs-22.1-7.fc7-i386.rpm* on a Red Hat system, we can use this command:

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

> **NOTE**
>
> *`dpkg` does not have a specific option for upgrading a package versus installing one as `rpm` does.*

Listing Installed Packages

Table 14-9 lists the commands we can use to display a list of all the packages installed on the system.

Table 14-9: Package Listing Commands

Style	Command(s)
Debian	`dpkg -l`
Red Hat	`rpm -qa`

Determining Whether a Package Is Installed

Table 14-10 lists the low-level tools we can use to display whether a specified package is installed.

Table 14-10: Package Status Commands

Style	Command(s)
Debian	`dpkg -s` *package_name*
Red Hat	`rpm -q` *package_name*

For example, to determine whether the `emacs` package is installed on a Debian-style system, we can use this:

```
dpkg -s emacs
```

Displaying Information About an Installed Package

If the name of an installed package is known, we can use the commands in Table 14-11 to display a description of the package.

Table 14-11: Package Information Commands

Style	Command(s)
Debian	`apt-cache show` *package_name*
Red Hat	`yum info` *package_name*

For example, to see a description of the `emacs` package on a Debian-style system, we can use the following:

```
apt-cache show emacs
```

Finding Which Package Installed a File

To determine what package is responsible for the installation of a particular file, we can use the commands in Table 14-12.

Table 14-12: Package File Identification Commands

Style	Command(s)
Debian	`dpkg -S file_name`
Red Hat	`rpm -qf file_name`

To see what package installed the `/usr/bin/vim` file on a Red Hat system, we can use the following:

```
rpm -qf /usr/bin/vim
```

Summing Up

In the chapters that follow, we will explore many different programs covering a wide range of application areas. While most of these programs are commonly installed by default, we may need to install additional packages if the necessary programs are not already installed on our system. With our newfound knowledge (and appreciation) of package management, we should have no problem installing and managing the programs we need.

THE LINUX SOFTWARE INSTALLATION MYTH

People migrating from other platforms sometimes fall victim to the myth that software is somehow difficult to install under Linux and that the variety of packaging schemes used by different distributions is a hindrance. Well, it is a hindrance, but only to proprietary software vendors that want to distribute binary-only versions of their secret software.

The Linux software ecosystem is based on the idea of open source code. If a program developer releases source code for a program, it is likely that a person associated with a distribution will package the program and include it in their repository. This method ensures that the program is well integrated into

the distribution, and the user is given the convenience of "one-stop shopping" for software, rather than having to search for each program's website. Recently, major proprietary platform vendors have begun building application stores that mimic this idea.

Device drivers are handled in much the same way, except that instead of being separate items in a distribution's repository, they become part of the Linux kernel. Generally speaking, there is no such thing as a "driver disk" in Linux. Either the kernel supports a device or it doesn't, and the Linux kernel supports a lot of devices—many more, in fact, than Windows does. Of course, this is of no consolation if the particular device you need is not supported. When that happens, you need to look at the cause. A lack of driver support is usually caused by one of three things.

- **The device is too new.** Since many hardware vendors don't actively support Linux development, it falls upon a member of the Linux community to write the kernel driver code. This takes time.
- **The device is too exotic.** Not all distributions include every possible device driver. Each distribution builds its own kernels, and since kernels are configurable (which is what makes it possible to run Linux on everything from wristwatches to mainframes), they may have overlooked a particular device. By locating and downloading the source code for the driver, it is possible for you (yes, you) to compile and install the driver yourself. This process is not overly difficult, but it is rather involved. We'll talk about compiling software in a later chapter.
- **The hardware vendor is hiding something.** It has neither released source code for a Linux driver nor has it released the technical documentation for somebody to create one for them. This means the hardware vendor is trying to keep the programming interfaces to the device a secret. Because we don't want secret devices in our computers, it is best that you avoid such products.