



openEuler
20.03 LTS

Application Development Guide

Date **2020-04-10**

Contents

Terms of Use	iv
About This Document.....	v
1 Preparation	7
1.1 Configuring the Development Environment.....	7
1.2 Configuring a Repo Source.....	8
1.3 Installing the Software Package.....	11
1.3.1 Installing the JDK Software Package	11
1.3.2 Installing the rpm-build Software Package	12
1.4 Using the IDE for Java Development	12
1.4.1 Overview	12
1.4.2 Logging In to the Server Using MobaXterm	12
1.4.3 Setting the JDK Environment	13
1.4.4 Downloading and Installing the GTK Library	13
1.4.5 Setting X11 Forwarding.....	13
1.4.6 Downloading and Running IntelliJ IDEA.....	14
2 Using GCC for Compilation	15
2.1 Overview	15
2.2 Basics.....	15
2.2.1 File Type	15
2.2.2 Compilation Process	16
2.2.3 Compilation Options.....	16
2.2.4 Multi-file Compilation.....	18
2.3 Libraries.....	18
2.3.1 Dynamic Link Library	19
2.3.2 Static Link Library.....	20
2.4 Examples	20
2.4.1 Example for Using GCC to Compile C Programs	20
2.4.2 Example for Creating and Using a DLL Using GCC.....	21
2.4.3 Example for Creating and Using an SLL Using GCC	23
3 Using Make for Compilation.....	25
3.1 Overview	25

3.2 Basics.....	25
3.2.1 File Type	25
3.2.2 make Work Process.....	26
3.2.3 make Options	26
3.3 Makefiles	28
3.4 Examples	29
3.4.1 Example of Using Makefile to Implement Compilation	29
4 Using JDK for Compilation	31
4.1 Overview	31
4.2 Basics.....	31
4.2.1 File Type and Tool	31
4.2.2 Java Program Generation Process.....	32
4.2.3 Common JDK Options.....	32
4.3 Class Library.....	36
4.4 Examples	37
4.4.1 Compiling a Java Program Without a Package	37
4.4.2 Compiling a Java Program with a Package.....	38
5 Building an RPM Package.....	40
5.1 Packaging Description	40
5.2 Building an RPM Package Locally.....	44
5.2.1 Setting Up the Development Environment	44
5.2.2 Creating a Hello World RPM Package.....	45
5.2.2.1 Obtaining the Source Code	45
5.2.2.2 Editing the SPEC File	45
5.2.2.3 Building an RPM Package	46
5.3 Building an RPM Package Using the OBS.....	47
5.3.1 OBS Overview.....	47
5.3.2 Building an RPM Software Package Online.....	47
5.3.2.1 Building an Existing Software Package	47
5.3.2.2 Adding a Software Package	49
5.3.2.3 Obtaining the Software Package.....	51
5.3.3 Building a Software Package Using OSC.....	52
5.3.3.1 Installing and Configuring the OSC	52
5.3.3.2 Building an Existing Software Package.....	53
5.3.3.3 Adding a Software Package	54
5.3.3.4 Obtaining the Software Package.....	55

Terms of Use

Copyright © Huawei Technologies Co., Ltd. 2020. All rights reserved.

Your replication, use, modification, and distribution of this document are governed by the Creative Commons License Attribution-ShareAlike 4.0 International Public License (CC BY-SA 4.0). You can visit <https://creativecommons.org/licenses/by-sa/4.0/> to view a human-readable summary of (and not a substitute for) CC BY-SA 4.0. For the complete CC BY-SA 4.0, visit <https://creativecommons.org/licenses/by-sa/4.0/legalcode>.

Trademarks and Permissions

openEuler is a trademark or registered trademark of Huawei Technologies Co., Ltd. All other trademarks and registered trademarks mentioned in this document are the property of their respective holders.

Disclaimer

This document is used only as a guide. Unless otherwise specified by applicable laws or agreed by both parties in written form, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, including but not limited to non-infringement, timeliness, and specific purposes.

About This Document

Overview

This document describes the following four parts to guide users to use openEuler and develop code based on openEuler.

- Install and use the GCC compiler in the openEuler operating system (OS), and complete the development, compilation, and execution of simple code.
- In the openEuler OS, use the JDK built-in tool to compile and execute code.
- Install IntelliJ IDEA in the openEuler OS for Java development.
- Create an RPM package locally or using the Open Build Service (OBS).


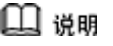
Intended Audience

This document is intended for all users who use the openEuler OS for code development. You are expected to have the following experience or capabilities:

- Have basic knowledge of the Linux OS.
- Know how to use Linux command lines.

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Command Conventions

Table 1 Command conventions

Format	Description
Boldface	Command keywords, which remain unchanged in the commands, are in boldface .
<i>Italic</i>	Command parameters, which are replaced with actual values in the commands, are in <i>italic</i> .
[]	Items in square brackets are optional.
{ x y ... }	Optional items are grouped in braces and separated by vertical bars. One item is selected.
[x y ...]	Optional items are grouped in brackets and separated by vertical bars. One item is selected or no item is selected.
{ x y ... }*	Optional items are grouped in brackets and separated by vertical bars. A minimum of one or a maximum of all can be selected.
[x y ...]*	Optional items are grouped in brackets and separated by vertical bars. One or more items are selected or no item is selected.

1 Preparation

- [1.1 Configuring the Development Environment](#)
- [1.2 Configuring a Repo Source](#)
- [1.3 Installing the Software Package](#)
- [1.4 Using the IDE for Java Development](#)

1.1 Configuring the Development Environment

- If physical machines (PMs) are used, the minimum hardware requirements of the development environment are described in Table 1-1.

Table 1-1 Minimum hardware specifications

Component	Minimum Hardware Specification	Description
Architecture	<ul style="list-style-type: none">• AArch64• x86_64	<ul style="list-style-type: none">• 64-bit Arm architecture• 64-bit Intel x86 architecture
CPU	<ul style="list-style-type: none">• Huawei Kunpeng 920 series• Intel® Xeon® processor	-
Memory	≥ 4 GB (8 GB or higher recommended for better user experience)	-
Hard disk	≥ 120 GB (for better user experience)	IDE, SATA, SAS interfaces are supported.

- If virtual machines (VMs) are used, the minimum virtualization space required for the development environment is described in Table 1-2.

Table 1-2 Minimum virtualization space

Component	Minimum Virtualization Space	Description
Architecture	<ul style="list-style-type: none">AArch64x86_64	-
CPU	Two CPUs	-
Memory	≥ 4 GB (8 GB or higher recommended for better user experience)	-
Hard disk	≥ 32 GB (120 GB or higher recommended for better user experience)	-

OS Requirements

The openEuler OS is required.

For details about how to install the openEuler OS, see the *openEuler 20.03 LTS Installation Guide*. On the **SOFTWARE SELECTION** page, select **Development Tools** in the **Add-Ons for Selected Environment** area.

1.2 Configuring a Repo Source

You can configure a repo source by directly obtaining the repo source file or by mounting an ISO file.

Configuring a Repo Source by Directly Obtaining the Repo Source File

NOTE

openEuler provides multiple repo source files. This section uses the OS repo source file of the AArch64 architecture as an example.

Step 1 Go to the yum source directory.

```
cd /etc/yum.repos.d
```

Step 2 Create and edit the **local.repo** file. Configure the repo source file as the yum source.

```
vi local.repo
```

Edit the **local.repo** file as follows:

```
[basiclocal]
```

```
name=basiclocal
```

```
baseurl=http://repo.openeuler.org/openEuler-20.03-LTS/OS/aarch64/
```

```
enabled=1
```



```
gpgcheck=0
```

----End

Configuring a Repo Source by Mounting an ISO File

NOTE

This section uses the **openEuler-20.03-LTS-aarch64-dvd.iso** image file and **openEuler-20.03-LTS-aarch64-dvd.iso.sha256sum** verification file as examples. Modify them based on the actual requirements.

Step 1 Download the ISO image.

- Download an ISO image using a cross-platform file transfer tool.
 - a. Log in to the openEuler community at <https://openeuler.org>.
 - b. Click **Download**.
 - c. Click the link provided after **Download ISO**. The download list is displayed.
 - d. Select the version to be downloaded, for example, openEuler 20.03 LTS. Then, click **openEuler-20.03-LTS**. The download list is displayed.
 - e. Click **ISO**. The ISO download list is displayed.
 - **aarch64**: ISO image file of the AArch64 architecture
 - **x86_64**: ISO image file of the x86_64 architecture
 - **source**: ISO image file of the openEuler source code
 - f. Click **aarch64**.
 - g. Click **openEuler-20.03-LTS-aarch64-dvd.iso** to download the openEuler release package to the local host.
 - h. Click **openEuler-20.03-LTS-aarch64-dvd.iso.sha256sum** to download the openEuler verification file to the local host.
 - i. Log in to the openEuler OS and create a directory for storing the release package and verification file, for example, **/home/iso**.

```
mkdir /home/iso
```
 - j. Use a cross-platform file transfer tool (such as WinSCP) to upload the local openEuler release package and verification file to the target openEuler OS.
- Run the **wget** command to download the ISO image.
 - a. Log in to the openEuler community at <https://openeuler.org>.
 - b. Click **Download**.
 - c. Click the link provided after **Download ISO**. The download list is displayed.
 - d. Select the version to be downloaded, for example, openEuler 20.03 LTS. Then, click **openEuler-20.03-LTS**. The download list is displayed.
 - e. Click **ISO**. The ISO download list is displayed.
 - **aarch64**: ISO image file of the AArch64 architecture
 - **x86_64**: ISO image file of the x86_64 architecture
 - **source**: ISO image file of the openEuler source code
 - f. Click **aarch64**.
 - g. Right-click **openEuler-20.03-LTS-aarch64-dvd.iso** and choose **Copy URL** from the shortcut menu to copy the address of the openEuler release package.

- h. Right-click **openEuler-20.03-LTS-aarch64-dvd.iso.sha256sum** and choose **Copy URL** from the shortcut menu to copy the address of the openEuler verification file.
- i. Log in to the openEuler OS, create a directory (for example, **/home/iso**) for storing the release package and verification file, and switch to the directory.

```
mkdir /home/iso  
cd /home/iso
```

- j. Run the **wget** command to remotely download the release package and verification file. In the command, **ipaddriso** and **ipaddrisosum** are the addresses copied in [Step 1.g](#) and [Step 1.h](#).

```
wget ipaddriso  
wget ipaddrisosum
```

Step 2 Release Package Integrity Check

1. Obtain the verification value in the verification file.

```
cat openEuler-20.03-LTS-aarch64-dvd.iso.sha256sum
```

2. Calculate the SHA256 verification value of the openEuler release package.

```
sha256sum openEuler-20.03-LTS-aarch64-dvd.iso
```

After the command is run, the verification value is displayed.

3. Check whether the values calculated in step 1 and step 2 are consistent.

If the verification values are consistent, the .iso file is not damaged. If they are inconsistent, the file is damaged and you need to obtain the file again.

Step 3 Mount the ISO image file and configure it as a repo source.

Run the **mount** command to mount the image file.

The following is an example:

```
# mount /home/iso/openEuler-20.03-LTS-aarch64-dvd.iso /mnt/
```

The mounted **mnt** directory is as follows:

```
.  
├─ boot.catalog  
├─ docs  
├─ EFI  
├─ images  
├─ Packages  
├─ repodata  
├─ TRANS.TBL  
└─ RPM-GPG-KEY-openEuler
```

In the preceding command, **Packages** indicates the directory where the RPM package is stored, **repodata** indicates the directory where the repo source metadata is stored, and **RPM-GPG-KEY-openEuler** indicates the public key for signing openEuler.

The mounted directory can be configured as the yum source. Create the *****.repo** configuration file (with the extension .repo) in the **/etc/yum.repos.d/** directory.

The following is an example:

Create the openEuler.repo file in the **/etc/yum.repos.d** directory and use the local image mounting directory as the yum source. The content of the openEuler.repo file is as follows:

```
[base]
name=base
baseurl=file:///mnt
enabled=1
gpgcheck=1
gpgkey=file:///mnt/RPM-GPG-KEY-openEuler
```

NOTE

- **gpgcheck** indicates whether to enable the GNU privacy guard (GPG) to check the validity and security of the source of RPM packages. **1** indicates that the GPG check is enabled. **0** indicates that the GPG check is disabled. If this option is not specified, the GPG check is enabled by default.
- **gpgkey** is the storage path of the signed public key.

----End

1.3 Installing the Software Package

Install the software required for development. The software required varies in different development environments. However, the installation methods are the same. This section describes how to install common software packages (such as JDK and rpm-build). Some development software, such as GCC and GNU make, is provided by the openEuler OS by default.

1.3.1 Installing the JDK Software Package

Step 1 Run the **dnf list installed | grep jdk** command to check whether the JDK software is installed.

```
dnf list installed | grep jdk
```

Check the command output. If the command output contains "jdk", the JDK has been installed. If no such information is displayed, the software is not installed.

Step 2 Clear the cache.

```
dnf clean all
```

Step 3 Create a cache.

```
dnf makecache
```

Step 4 Query the JDK software package that can be installed.

```
dnf search jdk | grep jdk
```

View the command output and install the **java-x.x.x-openjdk-devel.aarch64** software package. **x.x.x** indicates the version number.

Step 5 Install the JDK software package. The following uses the **java-1.8.0-openjdk-devel** software package as an example.

```
dnf install java-1.8.0-openjdk-devel.aarch64
```

Step 6 Query information about the JDK software.

```
java -version
```

Check the command output. If the command output contains "openjdk version "1.8.0_232"", the JDK has been correctly installed. In the command output, **1.8.0_232** indicates the JDK version.

----End

1.3.2 Installing the rpm-build Software Package

Step 1 Run the **dnf list installed | grep rpm-build** command to check whether the rpm-build software is installed.

```
dnf list installed | grep rpm-build
```

Check the command output. If the command output contains "rpm-build", the software has been installed. If no such information is displayed, the software is not installed.

Step 2 Clear the cache.

```
dnf clean all
```

Step 3 Create a cache.

```
dnf makecache
```

Step 4 Install the rpm-build package.

```
dnf install rpm-build
```

Step 5 Query the rpm-build software version.

```
rpmbuild --version
```

----End

1.4 Using the IDE for Java Development

For small-sized Java applications, you can directly use JDK to compile them to run Java applications. However, for medium- and large-sized Java applications, this method cannot meet the development requirements. You can perform the following steps to install and use the IDE to facilitate Java development on the openEuler OS.

1.4.1 Overview

IntelliJ IDEA is a popular Java IDE. You can download the community edition of IntelliJ IDEA for free. Currently, openEuler supports Java development in the IntelliJ IDEA integrated development environment (IDE), improving the work efficiency of developers.

1.4.2 Logging In to the Server Using MobaXterm

MobaXterm is an excellent SSH client. It has an X Server and can easily solve remote GUI display problems.

You need to download, install, and start MobaXterm in advance, and then log in to your server in SSH mode to perform the following operations:

1.4.3 Setting the JDK Environment

Before setting `JAVA_HOME`, you need to find the installation path of the JDK. You are supported to have installed the JDK. If you have not installed the JDK, install it by referring to Preparation > Installing the Software Package > Installing the JDK Software Package.

Run the following command to view the Java path:

```
# which java
/usr/bin/java
```

Run the following command to check the directory to which the soft link points:

```
# ls -la /usr/bin/java
lrwxrwxrwx. 1 root root 22 Mar 6 20:28 /usr/bin/java -> /etc/alternatives/java
# ls -la /etc/alternatives/java
lrwxrwxrwx. 1 root root 83 Mar 6 20:28 /etc/alternatives/java ->
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.232.b09-1.h2.aarch64/jre/bin/java
```

The actual path is `/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.232.b09-1.h2.aarch64`. Run the following command to set `JAVA_HOME` and `PATH`:

```
# export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.232.b09-1.h2.aarch64
# export PATH=$JAVA_HOME/bin:$PATH
```

1.4.4 Downloading and Installing the GTK Library

Ensure that you have the **root** permission and run the following command:

```
# dnf list installed | grep gtk
```

If **gtk2** or **gtk3** is displayed, the GTK library has been installed. In this case, skip this step. Otherwise, run the following command to automatically download and install the GTK library:

```
# dnf -y install gtk2 libXtst libXrender xauth
```

1.4.5 Setting X11 Forwarding

Switch to the SSHD configuration directory.

```
# cd ~/.ssh
```

If the directory does not exist, run the following command to create the directory and then switch to the directory:

```
# mkdir ~/.ssh
```

Edit the configuration file in the `.ssh` directory and save the file.

1. Run the **vim** command to open the configuration file.

```
# vim config
```

2. Add the following content to the end of the file and save the file:

```
Host *
    ForwardAgent yes
    ForwardX11 yes
```

1.4.6 Downloading and Running IntelliJ IDEA

After the preceding environment configuration is complete, you can download and run the IntelliJ IDEA. The latest version of IntelliJ IDEA is incompatible with openEuler in some functions. You are advised to click [here](#) and download the Linux package of the 2018 version. Move the downloaded package to the directory where you want to install the software and decompress the package.

```
# tar xf ideaIC-2018.3.tar.gz
```

Decompress the package, switch to the IntelliJ IDEA directory, and run the IntelliJ IDEA.

```
# cd ./idea-IC-183.4284.148  
# bin/idea.sh &
```

2 Using GCC for Compilation

This chapter describes the basic knowledge of GCC compilation and provides examples for demonstration. For more information about GCC, run the **man gcc** command.

[2.1 Overview](#)

[2.2 Basics](#)

[2.3 Libraries](#)

[2.4 Examples](#)

2.1 Overview

The GNU Compiler Collection (GCC) is a powerful and high-performance multi-platform compiler developed by GNU. The GCC compiler can compile and link source programs, assemblers, and target programs of C and C++ into executable files. By default, the GCC software package is installed in the openEuler OS.

2.2 Basics

2.2.1 File Type

For any given input file, the file type determines which compilation to perform. Table 2-1 describes the common GCC file types.

Table 2-1 Common GCC file types

Extension (Suffix)	Description
.c	C source code file.
.C, .cc, or .cxx	C++ source code file.
.m	Objective-C source code file.
.s	Assembly language source code file.

Extension (Suffix)	Description
.i	Preprocessed C source code file.
.ii	Preprocessed C++ source code file.
.S	Pre-processed assembly language source code file.
.h	Header file contained in the program.
.o	Target file after compilation.
.so	Dynamic link library, which is a special target file.
.a	Static link library.
.out	Executable files, which do not have a fixed suffix. The system distinguishes executable files from unexecutable files based on file attributes. If the name of an executable file is not given, GCC generates a file named a.out .

2.2.2 Compilation Process

Using GCC to generate executable files from source code files requires preprocessing, compilation, assembly, and linking.

1. Preprocessing: Preprocess the source program (such as a .c file) to generate an .i file.
2. Compilation: Compile the preprocessed .i file into an assembly language to generate an .s file.
3. Assemble: Assemble the assembly language file to generate the target file .o.
4. Linking: Link the .o files of each module to generate an executable program file.

The .i, .s, and .o files are intermediate or temporary files. If the GCC is used to compile programs in C language at a time, these files will be deleted.

2.2.3 Compilation Options

GCC compilation command format: **gcc** [*options*] [*filenames*]

In the preceding information:

options: compilation options.

filenames: file name.

GCC is a powerful compiler. It has many *options*, but most of them are not commonly used. Table 2-2 describes the common *options*.

Table 2-2 Common GCC compilation options

<i>options</i> Value	Description	Example
----------------------	-------------	---------

<i>options Value</i>	Description	Example
-c	Compiles and assembles specified source files to generate target files without linking them. It is usually used to compile subprogram files.	# Use the -c option to compile the source files test1.c and test2.c . gcc -c test1.c test2.c
-S	Compiles the specified source file to generate an assembly language file with the .s suffix but without assembling it.	# Use the compiler to preprocess circle.c , translate it into assembly language, and store the result in circle.s. gcc -S circle.c
-E	Preprocesses specified source files without compiling them. By default, the output of the preprocessor is imported to a standard output stream, such as a display. You can use the -o option to import it to an output file.	# Export the preprocessing result to the circle.i file. gcc -E circle.c -o circle.i
-o <i>file</i>	Generates a specified output <i>file</i> when an executable file is generated. The name must be different from that of the source file. If this option is not given, GCC generates the preset executable file a.out .	# Use the source file as the input file and the executable file as the output file. That is, compile the entire program. gcc main.c func.c -o app.out
-g	Contains standard debugging information in executable programs.	-
-L <i>library_path</i>	Adds the <i>library_path</i> to the library file search path list.	-
-l <i>library</i>	Searches for the specified function <i>library</i> during linking. When GCC is used to compile and link programs, GCC links libc.a or libc.so by default. However, other libraries (such as non-standard libraries and third-party libraries) need to be manually added.	# Use the -l option to link the math library. gcc main.c -o main.out -lm NOTE The file name of the math library is libm.a . The prefix lib and suffix .a are standard, and m is the basic name. GCC automatically adds these prefixes and suffixes to the basic name following the -l option. In this example, the basic name is m .

<i>options Value</i>	Description	Example
<code>-I head_path</code>	Adds the <i>head_path</i> to the search path list of the header file.	-
<code>-static</code>	Performs static compilation and links static libraries. Do not link dynamic libraries.	-
<code>-shared</code>	Default option, which can be omitted. <ul style="list-style-type: none">• A dynamic library file can be generated.• During dynamic compilation, the dynamic library is preferentially linked. The static library with the same name is linked only when there is no dynamic library.	-
<code>-fPIC</code> (or <code>-fpic</code>)	Generates location-independent target code that uses a relative address. Generally, the -static option is used to generate a dynamic library file from the PIC target file.	-

2.2.4 Multi-file Compilation

There are two methods provided for compiling multiple source files.

- Multiple source files are compiled at the same time. All files need to be recompiled during compilation.

Example: Compile **test1.c** and **test2.c** and link them to the executable file **test**.

```
gcc test1.c test2.c -o test
```

- Compile each source file, and then link the target files generated after compilation. During compilation, only modified files need to be recompiled.

For example, compile **test1.c** and **test2.c**, and link the target files **test1.o** and **test2.o** to the executable file **test**.

```
gcc -c test1.c
gcc -c test2.c
gcc -o test1.o test2.o -o test
```

2.3 Libraries

A library is mature and reusable code that has been written for use. Each program depends on many basic underlying libraries.

The library file name is prefixed with **lib** and suffixed with **.so** (dynamic library) or **.a** (static library). The middle part is the user-defined library file name, for example, **libfoo.so** or **libfoo.a**. Because all library files comply with the same specifications, the **lib** prefix can be omitted when the **-l** option specifies the name of the linked library file. That is, when GCC processes **-lfoo**, the library file **libfoo.so** or **libfoo.a** is automatically linked. When creating a library, you must specify the full file name **libfoo.so** or **libfoo.a**.

Libraries are classified into static libraries and dynamic libraries based on the linking time. The static library links and packs the target file **.o** generated by assembly and the referenced library into an executable file in the linking phase. The dynamic library is not linked to the target code when the program is compiled, but is loaded when the program is run. The differences are as follows:

- The resource usage is different.
The static library is a part of the generated executable file, while the dynamic library is a separate file. Therefore, the sizes and occupied disk space of the executable files of the static library and dynamic library are different, which leads to different resource usage.
- The scalability and compatibility are different.
If the implementation of a function in the static library changes, the executable file must be recompiled. For the executable file generated by dynamic linking, only the dynamic library needs to be updated, and the executable file does not need to be recompiled.
- The dependency is different.
The executable file of the static library can run without depending on any other contents, while the executable file of the dynamic library must depend on the dynamic library. Therefore, the static library is convenient to migrate.
- The loading speeds are different.
Static libraries are linked together with executable files, while dynamic libraries are linked only when they are loaded or run. Therefore, for the same program, static linking is faster than dynamic linking.

2.3.1 Dynamic Link Library

You can use the **-shared** and **-fPIC** options to create a dynamic link library (DLL) with the source file, assembly file, or target file. The **-fPIC** option is used in the compilation phase. This option is used when the target file is generated, so as to generate location-independent code.

Example 1: Generate a DLL from the source file.

```
gcc -fPIC -shared test.c -o libtest.so
```

Example 2: Generate a DLL from the target file.

```
gcc -fPIC -c test.c -o test.o  
gcc -shared test.o -o libtest.so
```

To link a DLL to an executable file, you need to list the name of the DLL in the command line.

Example: Compile **main.c** and **libtest.so** into **app.out**. When **app.out** is running, the link library **libtest.so** is dynamically loaded.

```
gcc main.c libtest.so -o app.out
```

In this mode, the **libtest.so** file in the current directory is used.

If you choose to search for a DLL, to ensure that the DLL can be linked when the program is running, you must implement by using one of the following methods:

- Save the DLL to a standard directory, for example, **/usr/lib**.
- Add the DLL path **libraryDIR** to the environment variable **LD_LIBRARY_PATH**.
`export LD_LIBRARY_PATH=libraryDIR:$LD_LIBRARY_PATH`

NOTE

LD_LIBRARY_PATH is an environment variable of the DLL. If the DLL is not in the default directories (**/lib** and **/usr/lib**), you need to specify the environment variable **LD_LIBRARY_PATH**.

- Add the DLL path **libraryDIR** to **/etc/ld.so.conf** and run **ldconfig**, or use the DLL path **libraryDIR** as a parameter to run **ldconfig**.

```
gcc main.c -L libraryDIR -ltest -o app.out
export LD_LIBRARY_PATH=libraryDIR:$LD_LIBRARY_PATH
```

2.3.2 Static Link Library

To create a static link library (SLL), you need to compile the source file to the target file, and then run the **ar** command to compress the target file into an SLL.

Example: Compile and compress source files **test1.c**, **test2.c**, and **test3.c** into an SLL.

```
gcc -c test1.c test2.c test3.c
ar rcs libtest.a test1.o test2.o test3.o
```

The **ar** command is a backup compression command. You can compress multiple files into a backup file (also called an archive file) or extract member files from the backup file. The most common use of **ar** is to compress the target files into an SLL.

The format of the **ar** command to compress the target files into an SLL is as follows:

`ar rcs Sllfilename Targetfilelist`

- *Sllfilename*: Name of the static library file.
- *Targetfilelist*: Target file list.
- **r**: replaces the existing target file in the library or adds a new target file.
- **c**: creates a library regardless of whether the library exists.
- **s**: creates the index of the target file. The speed can be improved when a large library is created.

Example: Create a **main.c** file to use the SLL.

```
gcc main.c -L libraryDIR -ltest -o test.out
```

In the preceding command, **libraryDIR** indicates the path of the **libtest.a** library.

2.4 Examples

2.4.1 Example for Using GCC to Compile C Programs

Step 1 Run the **cd** command to go to the code directory. The **/home/code** directory is used as an example. The command is as follows:

```
cd /home/code
```

Step 2 Compile the Hello World program and save it as **helloworld.c**. The following uses the Hello World program as an example. The command is as follows:

```
vi helloworld.c
```

Code example:

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Step 3 Run the following command to compile the code in the code directory:

```
gcc helloworld.c -o helloworld
```

If no error is reported, the execution is successful.

Step 4 After the compilation is complete, the helloworld file is generated. Check the compilation result. The following is an example:

```
# ./helloworld
Hello World!
```

----End

2.4.2 Example for Creating and Using a DLL Using GCC

Step 1 Run the **cd** command to go to the code directory. The **/home/code** directory is used as an example. Create the **src**, **lib**, and **include** subdirectories in the directory to store the source file, DLL file, and header file, respectively.

```
cd /home/code
mkdir src
mkdir lib
mkdir include
```

Step 2 Run the **cd** command to go to the **/home/code/src** directory and create two functions **add.c** and **sub.c** to implement addition and subtraction, respectively.

```
cd /home/code/src
vi add.c
vi sub.c
```

The following is an example of the **add.c** code:

```
#include "math.h"
int add(int a, int b)
{
    return a+b;
}
```

The following is an example of the **sub.c** code:

```
#include "math.h"
int sub(int a, int b)
{
```

```
        return a-b;
    }
```

Step 3 Compile the source files `add.c` and `sub.c` into the DLL `libmath.so`, and store the DLL in the `/home/code/lib` directory.

```
gcc -fPIC -shared add.c sub.c -o /home/code/lib/libmath.so
```

Step 4 Go to the `/home/code/include` directory, create a header file `math.h`, and declare the header file of the function.

```
cd /home/code/include
vi math.h
```

The following is an example of the `math.h` code:

```
#ifndef    MATH H
#define    MATH H
int add(int a, int b);
int sub(int a, int b);
#endif
```

Step 5 Run the `cd` command to go to the `/home/code/src` directory and create a `main.c` function that invokes `add()` and `sub()`.

```
cd /home/code/src
vi main.c
```

The following is an example of the `math.c` code:

```
#include <stdio.h>
#include "math.h"
int main()
{
    int a, b;
    printf("Please input a and b:\n");
    scanf("%d %d", &a, &b);
    printf("The add: %d\n", add(a,b));
    printf("The sub: %d\n", sub(a,b));
    return 0;
}
```

Step 6 Compile `main.c` and `libmath.so` into `math.out`.

```
gcc main.c -I /home/code/include -L /home/code/lib -lmath -o math.out
```

Step 7 Add the path of the DLL to the environment variable.

```
export LD_LIBRARY_PATH=/home/code/lib:$LD_LIBRARY_PATH
```

Step 8 Run the following command to execute `math.out`:

```
./math.out
```

The command output is as follows:

```
Please input a and b:
9 2
The add: 11
The sub: 7
```

----End

2.4.3 Example for Creating and Using an SLL Using GCC

- Step 1** Run the **cd** command to go to the code directory. The **/home/code** directory is used as an example. Create the **src**, **lib**, and **include** subdirectories in the directory to store the source file, SLL file, and header file respectively.

```
cd /home/code
mkdir src
mkdir lib
mkdir include
```

- Step 2** Run the **cd** command to go to the **/home/code/src** directory and create two functions **add.c** and **sub.c** to implement addition and subtraction, respectively.

```
cd /home/code/src
vi add.c
vi sub.c
```

The following is an example of the **add.c** code:

```
#include "math.h"
int add(int a, int b)
{
    return a+b;
}
```

The following is an example of the **sub.c** code:

```
#include "math.h"
int sub(int a, int b)
{
    return a-b;
}
```

- Step 3** Compile the source files **add.c** and **sub.c** into the target files **add.o** and **sub.o**.

```
gcc -c add.c sub.c
```

- Step 4** Run the **ar** command to compress the **add.o** and **sub.o** target files into the SLL **libmath.a** and save the SLL to the **/home/code/lib** directory.

```
ar rcs /home/code/lib/libmath.a add.o sub.o
```

- Step 5** Go to the **/home/code/include** directory, create a header file **math.h**, and declare the header file of the function.

```
cd /home/code/include
vi math.h
```

The following is an example of the **math.h** code:

```
#ifndef MATH H
#define MATH H
int add(int a, int b);
int sub(int a, int b);
#endif
```

- Step 6** Run the **cd** command to go to the **/home/code/src** directory and create a **main.c** function that invokes **add()** and **sub()**.

```
cd /home/code/src  
vi main.c
```

The following is an example of the **math.c** code:

```
#include <stdio.h>  
#include "math.h"  
int main()  
{  
    int a, b;  
    printf("Please input a and b:\n");  
    scanf("%d %d", &a, &b);  
    printf("The add: %d\n", add(a,b));  
    printf("The sub: %d\n", sub(a,b));  
    return 0;  
}
```

Step 7 Compile **main.c** and **libmath.a** into **math.out**.

```
gcc main.c -I /home/code/include -L /home/code/lib -lmath -o math.out
```

Step 8 Run the following command to execute **math.out**:

```
./math.out
```

The command output is as follows:

```
Please input a and b:  
9 2  
The add: 11  
The sub: 7
```

----End

3 Using Make for Compilation

This chapter describes the basic knowledge of make compilation and provides examples for demonstration. For more information about Make, run the **man make** command.

[3.1 Overview](#)

[3.2 Basics](#)

[3.3 Makefiles](#)

[3.4 Examples](#)

3.1 Overview

The GNU make utility (usually abbreviated as make) is a tool for controlling the generation of executable files from source files. make automatically identifies which parts of the complex program have changed and need to be recompiled. Make uses a configuration file called makefiles to control how the program is built.

3.2 Basics

3.2.1 File Type

Table 3-1 describes the file types that may be used in the makefiles file.

Table 3-1 文件类型

Extension (Suffix)	Description
.c	C source code file.
.C, .cc, or .cxx	C++ source code file.
.m	Objective-C source code file.
.s	Assembly language source code file.
.i	Preprocessed C source code file.

Extension (Suffix)	Description
.ii	Preprocessed C++ source code file.
.S	Pre-processed assembly language source code file.
.h	Header file contained in the program.
.o	Target file after compilation.
.so	Dynamic link library, which is a special target file.
.a	Static link library.
.out	Executable files, which do not have a fixed suffix. The system distinguishes executable files from unexecutable files based on file attributes. If the name of an executable file is not given, GCC generates a file named a.out .

3.2.2 make Work Process

The process of deploying make to generate an executable file from the source code file is described as follows:

1. The make command reads the Makefiles, including the files named GNUmakefile, makefile, and Makefile in the current directory, the included makefile, and the rule files specified by the **-f**, **--file**, and **--makefile** options.
2. Initialize variables.
3. Derive implicit rules, analyze dependencies, and create a dependency chain.
4. Determine which targets need to be regenerated based on the dependency chain.
5. Run a command to generate the final file.

3.2.3 make Options

make command format: **make** [*option*]... [*target*]...

In the preceding command:

option: parameter option.

target: target specified in Makefile.

Table 3-2 describes the common make options.

Table 3-2 Common make options

options Value	Description
-C <i>dir</i> , --directory = <i>dir</i>	Specifies <i>dir</i> as the working directory after the make command starts to run. When there are multiple -C options, the

<i>options</i> Value	Description
	final working directory of make is the relative path of the first directory.
-d	Displays all debugging information during execution of the make command. You can use the -d option to display all the information during the construction of the dependency chain and the reconstruction of the target.
-e, --environment-overrides	Overwrites the variable definition with the same name in Makefile with the environment variable definition.
-f <i>file</i> , --file= <i>file</i> , --makefile= <i>file</i>	Specifies the <i>file</i> as the Makefile for the make command.
-p, --help	Displays help information.
-i, --ignore-errors	Ignores the errors occurred during the execution.
-k, --keep-going	When an error occurs during command execution, the make command is not terminated. The make command executes all commands as many as possible until a known error occurs.
-n, --just-print, --dry-run	Simulates the execution of commands (including the commands starting with @) in the actual execution sequence. This command is used only to display the execution process and has no actual execution effect.
-o <i>file</i> , --old-file= <i>file</i> , --assume-old= <i>file</i>	The specified <i>file</i> does not need to be rebuilt even if its dependency has expired, and no target of this dependency file is rebuilt.
-p, --print-date-base	Before the command is executed, all data of Makefile read by make and the version information of make are printed. If you only need to print the data, run the make -qp command to view the preset rules and variables before the make command is executed. You can run the make -p -f /dev/null command.
-r, --no-builtin-rules	Ignores the use of embedded implicit rules and the implicit suffix list of all suffix rules.
-R, --no-builtin-variables	Ignores embedded hidden variables.
-s, --silent, --quiet	Cancels the printing during the command execution.

<i>options Value</i>	Description
-S, --no-keep-going, --stop	Cancels the -k option. In the recursive make process, the sub-make inherits the upper-layer command line option through the MAKEFLAGS variable. You can use the -S option in the sub-make to cancel the -k option transferred by the upper-layer command, or cancel the -k option in the system environment variable MAKEFLAGS .
-t, --touch	Updates the timestamp of all target files to the current system time. Prevents make from rebuilding all outdated target files.
-v, version	Displays the make version.

3.3 Makefiles

Make is a tool that uses makefiles for compilation, linking, installation, and cleanup, so as to generate executable files and other related files from source code files. Therefore, makefiles describe the compilation and linking rules of the entire project, including which files need to be compiled, which files do not need to be compiled, which files need to be compiled first, which files need to be compiled later, and which files need to be rebuilt. The makefiles automate project compilation. You do not need to manually enter a large number of source files and parameters each time.

This chapter describes the structure and main contents of makefiles. For more information about makefiles, run the **info make** command.

Makefile Structure

The makefile file structure is as follows:

targets:prerequisites

command

or

targets:prerequisites;command

command

In the preceding information:

- *targets*: targets, which can be target files, executable files, or tags.
- *prerequisites*: dependency files, which are the files or targets required for generating the *targets*. There can be multiple or none of them.
- *command*: command (any shell command) to be executed by make. Multiple commands are allowed, and each command occupies a line.

- Use colons (:) to separate the target files from the dependency files. Press **Tab** at the beginning of each command line.

The makefile file structure indicates the output target, the object on which the output target depends, and the command to be executed for generating the target.

Makefile Contents

A makefile file consists of the following contents:

- Explicit rule
Specify the dependency, such as the file to be generated, dependency file, and generated command.
- Implicit rule
Specify the rule that is automatically derived by make. The make command supports the automatic derivation function.
- Variable definition
- File indicator
The file indicator consists of three parts:
 - Inclusion of other makefiles, for example, include xx.md
 - Selective execution, for example, #ifdef
 - Definition of multiple command lines, for example, define...endef. (define ... endef)
- Comment
The comment starts with a number sign (#).

3.4 Examples

3.4.1 Example of Using Makefile to Implement Compilation

Step 1 Run the **cd** command to go to the code directory. The **/home/code** directory is used as an example.

```
cd /home/code
```

Step 2 Create a header file **hello.h** and two functions **hello.c** and **main.c**.

```
cd /home/code/
```

The following is an example of the **hello.h** code:

```
#pragma once
#include <stdio.h>
void hello();
```

The following is an example of the **hello.c** code:

```
#include "hello.h"
void hello()
{
    int i=1;
    while(i<5)
    {
```

```
        printf("The %dth say hello.\n", i);  
        i++;  
    }  
}
```

The following is an example of the **main.c** code:

```
#include "hello.h"  
#include <stdio.h>  
int main()  
{  
    hello();  
    return 0;  
}
```

Step 3 Create the makefile.

```
vi Makefile
```

The following provides an example of the makefile content:

```
main:main.o hello.o  
    gcc -o main main.o hello.o  
main.o:main.c  
    gcc -c main.c  
hello.o:hello.c  
    gcc -c hello.c  
clean:  
    rm -f hello.o main.o main
```

Step 4 Run the **make** command.

```
make
```

After the command is executed, the commands executed in makefile are printed. If you do not need to print the information, add the **-s** option to the **make** command.

```
gcc -c main.c
```

```
gcc -c hello.c
```

```
gcc -o main main.o hello.o
```

Step 5 Execute the **./main** target.

```
./main
```

After the command is executed, the following information is displayed:

```
The 1th say hello.
```

```
The 2th say hello.
```

```
The 3th say hello.
```

```
The 4th say hello.
```

```
----End
```

4 Using JDK for Compilation

- [4.1 Overview](#)
- [4.2 Basics](#)
- [4.3 Class Library](#)
- [4.4 Examples](#)

4.1 Overview

A Java Development Kit (JDK) is a software package required for Java development. It contains the Java Runtime Environment (JRE) and compilation and commissioning tools. On the basis of OpenJDK, openEuler optimizes GC, enhances concurrency stability, and enhances security, improving the performance and stability of Java applications on ARM.

4.2 Basics

4.2.1 File Type and Tool

For any given input file, the file type determines which tool to use for processing. The common file types and tools are described in Table 4-1 and Table 4-2.

Table 4-1 Common JDK file types

Extension (Suffix)	Description
.java	Java source code file.
.class	Java bytecode file, which is intermediate code irrelevant to any specific machine or OS environment. It is a binary file, which is the target code file generated after the Java source file is compiled by the Java compiler.
.jar	JAR package of Java files.

Table 4-2 Common JDK tools

Name	Description
java	Java running tool, which is used to run .class bytecode files or .jar files.
javac	Compiles Java source code files into .class bytecode files.
jar	Creates and manages JAR files.

4.2.2 Java Program Generation Process

To generate a program from Java source code files and run the program using Java, compilation and run are required.

1. Compilation: Use the Java compiler (javac) to compile Java source code files (.java files) into .class bytecode files.
2. Run: Execute the bytecode files on the Java virtual machine (JVM).

4.2.3 Common JDK Options

Javac Compilation Options

The command format for javac compilation is as follows: **javac** [*options*] [*sourcefiles*] [*classes*] [*@argfiles*]

In the preceding information:

options: command options.

sourcefiles: one or more source files to be compiled.

classes: one or more classes to be processed as comments.

@argfiles: one or more files that list options and source files. The **-J** option is not allowed in these files.

Javac is a Java compiler. It has many *options*, but most of them are not commonly used. Table 4-3 describes the common options values.

Table 4-3 Common javac options

<i>options</i> Value	Description	Example
-d <i>path</i>	Path for storing the generated class files. By default, the class files generated after compilation are in the same path as the source file. You can use the -d option to export the class	# Use the -d option to export all class files to the bin directory. javac /src/*.java -d /bin

<i>options Value</i>	Description	Example
	files to the specified path.	
<i>-s path</i>	Path for storing the generated source files.	-
<i>-cp path</i> or <i>-classpath path</i>	Searches for the class files required for compilation and specifies the location of the class files.	# In the Demo, the <code>getLine()</code> method in the <code>GetStringDemo</code> class needs to be invoked. The <code>.class</code> file compiled by the <code>GetStringDemo</code> class is stored in the <code>bin</code> directory. <code>javac -cp bin Demo.java -d bin</code>
<i>-verbose</i>	Outputs information about the operations being performed by the compiler, such as loaded class information and compiled source file information.	# Display information about the operations that are being performed by the compiler. <code>javac -verbose -cp bin Demo.java</code>
<i>-source sourceversion</i>	Specifies the location of the input source files to be searched for.	-
<i>-sourcepath path</i>	Searches for source files (Java files) required for compilation and specifies the location of the source files to be searched for, for example, JAR, ZIP, or other directories that contain Java files.	-
<i>-target targetversion</i>	Generates class files of a specific JVM version. The value can be 1.1, 1.2, 1.3, 1.4, 1.5 (or 5), 1.6 (or 6), 1.7 (or 7), or 1.8 (or 8). The default value of <i>targetversion</i> is related to <i>sourceversion</i> of the -source option. The options of <i>sourceversion</i> are as follows: <ul style="list-style-type: none"> 1.2, corresponding to target version 1.4 1.3, corresponding to target version 1.4 1.5, 1.6, 1.7, and unspecified, corresponding to target 	-

<i>options</i> Value	Description	Example
	version 1.8 <ul style="list-style-type: none"> For other values, the values of <i>targetversion</i> and <i>sourceversion</i> are the same. 	

Java Running Options

The Java running format is as follows:

Running class file: **java** [*options*] *classname* [*args*]

Running Java file: **java** [*options*] -jar *filename* [*args*]

In the preceding information:

options: command options, which are separated by spaces.

classname: name of the running .class file.

filename: name of the running .jar file.

args: parameters transferred to the main() function. The parameters are separated by spaces.

Java is a tool for running Java applications. It has many *options*, but most of them are not commonly used. Table 4-4 describes the common options.

Table 4-4 Common Java running options

<i>options</i> Value	Description	Example
-cp <i>path</i> or -classpath <i>path</i>	Specifies the location of the file to be run and the class path to be used, including the .jar, .zip, and class file directories. If there are multiple paths, separate them with colons (:).	-
-verbose	Outputs information about the operations being performed by the compiler, such as loaded class information and compiled source file information.	# Display information about the operations that are being performed by the compiler. java -verbose -cp bin Demo.java

JAR Options

The JAR command format is as follows: **jar** {c | t | x | u}[vfm0M] [*jarfile*] [*manifest*] [-C *dir*] *file...*

Table 4-5 describes the parameters in the **jar** command.

Table 4-5 JAR parameter description

Parameter	Description	Example
c	Creates a JAR package.	# Compress the hello.class files in the current directory into Hello.jar. The compression process is not displayed. If the Hello.jar files do not exist, create them. Otherwise, clear the directory. jar cf Hello.jar hello.class
t	Lists the contents of a JAR package.	# List the files contained in Hello.jar. jar tf Hello.jar
x	Decompresses a JAR package.	# Decompress Hello.jar to the current directory. No information is displayed. jar xf Hello.jar
u	Updates the existing JAR package, for example, add files to the JAR package.	-
v	Generates a detailed report and prints it to the standard output.	# Compress the hello.class files in the current directory into Hello.jar and display the compression process. If the Hello.jar files do not exist, create them. Otherwise, clear the directory. jar cvf Hello.jar hello.class
f	Specifies the name of a JAR package. This parameter is mandatory.	-
m	Specifies the manifest file to be contained.	-
0	If this parameter is not set, the generated JAR package is larger but faster than that generated when this parameter is not set.	-
M	If the manifest file of all items is not generated, this parameter will be ignored.	# Compress the hello.class files in the current directory into Hello.jar and display the compression process. If

Parameter	Description	Example
		the Hello.jar files do not exist, create them. Otherwise, clear the directory. However, the manifest file is not generated when Hello.jar is created. jar cvfM Hello.jar hello.class
<i>jarfile</i>	JAR package, which is an auxiliary parameter of the f parameter.	-
<i>manifest</i>	Manifest file in .mf format, which is an auxiliary parameter of the m parameter.	-
<i>-C dir</i>	Runs the jar command in the specified <i>dir</i> . This command can be used only with parameters c and t .	-
<i>file</i>	Specifies the file or path list. All files in the file or path (including those in the recursive path) are compressed into the JAR package or the JAR package is decompressed to the path.	# Compress all class files in the current directory into Hello.jar and display the compression process. If the Hello.jar files do not exist, create them. Otherwise, clear the directory. jar cvf Hello.jar *.class

4.3 Class Library

The Java class library is implemented as a package, which is a collection of classes and interfaces. The Java compiler generates a bytecode file for each class, and the file name is the same as the class name. Therefore, conflicts may occur between classes with the same name. In the Java language, a group of classes and interfaces are encapsulated in a package. Class namespaces can be effectively managed by package. Classes in different packages do not conflict even if they have the same name. This solves the problem of conflicts between classes with the same name and facilitates the management of a large number of classes and interfaces. It also ensures the security of classes and interfaces.

In addition to many packages provided by Java, developers can customize packages by collecting compiled classes and interfaces into a package for future use.

Before using a custom package, you need to declare the package.

Package Declaration

The declaration format of a package is `package pkg1[.pkg2[.pkg3...]]`.

To declare a package, you must create a directory. The subdirectory name must be the same as the package name. Then declare the package at the beginning of the class file that needs to be placed in the package, indicating that all classes of the file belong to the package. The dot (.) in the package declaration indicates the directory hierarchy. If the source program file does not contain the package statement, the package is specified as an anonymous package. An anonymous package does not have a path. Generally, Java still stores the classes in the source file in the current working directory (that is, the directory where the Java source files are stored).

The package declaration statement must be added to the beginning of the source program file and cannot be preceded by comments or spaces. If you use the same package declaration statement in different source program files, you can include the classes in different source program files in the same package.

Package Reference

In Java, there are two methods to use the common classes in the package provided by Java or the classes in the custom package.

- Add the package name before the name of the class to be referenced.

For example, `name.A obj=new name.A ()`

name indicates the package name, **A** indicates the class name, and **obj** indicates the object. This string indicates that class **A** in the **name** package is used to define an object **obj** in the program.

Example: Create a test object of the **Test** class in the **example** package.

```
example.Test test = new example.Test();
```

- Use **import** at the beginning of the file to import the classes in the package.

The format of the **import** statement is `import pkg1[.pkg2[.pkg3...]].(classname | *)`.

pkg1[.pkg2[.pkg3...]] indicates the package level, and **classname** indicates the class to be imported. If you want to import multiple classes from a package, you can use the wildcard (*) instead.

Example: Import the **Test** class in the **example** package.

```
import example.Test;
```

Example: Import the entire **example** package.

```
import example.*;
```

4.4 Examples

4.4.1 Compiling a Java Program Without a Package

- Step 1** Run the **cd** command to go to the code directory. The **/home/code** directory is used as an example. The command is as follows:

```
# cd /home/code
```

- Step 2** Compile the Hello World program and save it as **HelloWorld.java**. The following uses the Hello World program as an example. The command is as follows:

```
# vi HelloWorld.java
```

Code example:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

Step 3 Run the following command to compile the code in the code directory:

```
# javac HelloWorld.java
```

If no error is reported, the execution is successful.

Step 4 After the compilation is complete, the HelloWorld.class file is generated. You can run the **java** command to view the result. The following is an example:

```
# java HelloWorld  
Hello World
```

----End

4.4.2 Compiling a Java Program with a Package

Step 1 Run the **cd** command to go to the code directory. The **/home/code** directory is used as an example. Create the **/home/code/Test/my/example**, **/home/code/Hello/world/developers**, and **/home/code/Hi/openos/openeuler** subdirectories in the directory to store source files.

```
cd /home/code  
  
mkdir -p Test/my/example  
mkdir -p Hello/world/developers  
mkdir -p Hi/openos/openeuler
```

Step 2 Run the **cd** command to go to the **/home/code/Test/my/example** directory and create **Test.java**.

```
cd /home/code/Test/my/example  
vi Test.java
```

The following is an example of the Test.java code:

```
package my.example;  
import world.developers.Hello;  
import openos.openeuler.Hi;  
public class Test {  
    public static void main(String[] args) {  
        Hello me = new Hello();  
        me.hello();  
        Hi you = new Hi();  
        you.hi();  
    }  
}
```

Step 3 Run the **cd** command to go to the **/home/code/Hello/world/developers** directory and create **Hello.java**.

```
cd /home/code/Hello/world/developers
vi Hello.java
```

The following is an example of the Hello.java code:

```
package world.developers;
public class Hello {
    public void hello(){
        System.out.println("Hello, openEuler.");
    }
}
```

Step 4 Run the **cd** command to go to the **/home/code/Hi/openos/openeuler** directory and create **Hi.java**.

```
cd /home/code/Hi/openos/openeuler
vi Hi.java
```

The following is an example of the Hi.java code:

```
package openos.openeuler;
public class Hi {
    public void hi(){
        System.out.println("Hi, the global developers.");
    }
}
```

Step 5 Run the **cd** command to go to the **/home/code** directory and use **javac** to compile the source file.

```
cd /home/code
javac -classpath Hello:Hi Test/my/example/Test.java
```

After the command is executed, the **Test.class**, **Hello.class**, and **Hi.class** files are generated in the **/home/code/Test/my/example**, **/home/code/Hello/world/developers**, and **/home/code/Hi/openos/openeuler** directories.

Step 6 Run the **cd** command to go to the **/home/code** directory and run the **Test** program using Java.

```
cd /home/code
java -classpath Test:Hello:Hi my/example/Test
```

The command output is as follows:

```
Hello, openEuler.
Hi, the global developers.
```

----End

5 Building an RPM Package

This section describes how to build an RPM software package on a local PC or using OBS. For details, see <https://gitee.com/openeuler/community/blob/master/zh/contributors/packaging.md>.

[5.1 Packaging Description](#)

[5.2 Building an RPM Package Locally](#)

[5.3 Building an RPM Package Using the OBS](#)

5.1 Packaging Description

Principles

During RPM packaging, the source code needs to be compiled. The compiled configuration files and binary command files need to be placed in proper positions. The RPM packages need to be tested as required. A workspace is required for these operations. The **rpmbuild** command uses a set of standard workspaces.

```
# rpmdev-setuptree
```

The **rpmdev-setuptree** command is used to install rpmdevtools. After the command is executed, the **rpmbuild** folder is generated in the **/root** directory (or the **/home/username** directory for non-root users). The directory structure is as follows:

```
# tree rpmbuild
rpmbuild
├── BUILD
├── RPMS
├── SOURCES
├── SPECS
└── SRPMS
```

The content is described as follows:

Content	Macro Code	Name	Function
~/rpmbuild/BUILD	%_builddir	Build directory.	The source code package is decompressed and compiled in

Content	Macro Code	Name	Function
			a subdirectory of the directory.
~/rpmbuild/RPMS	%_rpmdir	Standard RPM package directory.	The binary RPM package is generated and stored in this directory.
~/rpmbuild/SOURCES	%_sourcedir	Source code directory.	The source code package (for example, .tar package) and all patches are stored in this directory.
~/rpmbuild/SPECS	%_specdir	Spec file directory.	The RPM package configuration file (.spec) is stored in this directory.
~/rpmbuild/SRPMS	%_srcrpmdir	Source code RPM package directory.	The source code RPM package (SRPM) is stored in this directory.

The **~/rpmbuild/SPECS** directory contains the configuration file of the RPM package, which is the drawing of the RPM package. This file tells the **rpmbuild** command how to build the RPM package. The **Macro Code** column contains the corresponding directories in the .spec file, which is similar to the macro or global variable in the programming language.

Packaging Process

The packaging process is as follows:

1. Place the source code in **%_sourcedir**.
2. Compile the source code in **%_builddir**. Generally, the source code is compressed and needs to be decompressed first.
3. Install the RPM package. The installation is similar to pre-assembling the software package. Copy the contents (such as binary files, configuration files, and man files) that should be contained in the software package to **%_buildrootdir** and assemble the contents based on the actual directory structure after installation. For example, if binary commands are stored in **/usr/bin**, copy the directory structure to **%_buildrootdir**.
4. Perform necessary configurations, such as preparations before installation and cleanup after installation. These are configured in the SPEC file to tell the **rpmbuild** command how to build.
5. Check whether the software is running properly.
6. The generated RPM package is stored in **%_rpmdir**, and the source code package is stored in **%_srcrpmdir**.

In the SPEC file, each phase is described as follows:

Phase	Directory to Be Read	Directory to Which Data Is Written	Action
%prep	%_sourcedir	%_builddir	Read the source code and patch in the %_sourcedir directory. Then, decompress the source code to the %_builddir subdirectory and apply all patches.
%build	%_builddir	%_builddir	Compile files in the %_builddir build directory. Run a command similar to ./configure && make .
%install	%_builddir	%_buildrootdir	Read files in the %_builddir build directory and install them to the %_buildrootdir directory. These files are generated after the RPM is installed.
%check	%_builddir	%_builddir	Check whether the software is running properly. Run a command similar to make test .
bin	%_buildrootdir	%_rpmmdir	Read files in the %_buildrootdir final installation directory to create RPM packages in the %_rpmmdir directory. In this directory, RPM packages of different architectures are stored in different subdirectories. The noarch directory stores RPM packages applicable to all architectures. These RPM files are the RPM packages that are finally installed by users.
src	%_sourcedir	%_srcrpmdir	Create the source code RPM package (SRPM for short, with the file name extension .src.rpm) and save it to the %_srcrpmdir directory. The SRPM package is usually used to review and upgrade software packages.

Packaging Options

Run the **rpmbuild** command to build the software package. The **rpmbuild** command can be used to build software packages by building .spec, .tar, and source files.

The format of the **rpmbuild** command is **rpmbuild [option...]**

Table 5-1 describes the common **rpmbuild** packaging options.

Table 5-1 rpmbuild Packaging Options

option Value	Description
-bp <i>specfile</i>	Starts build from the %prep phase of the <i>specfile</i> (decompress the source code package and install the patch).
-bc <i>specfile</i>	Starts build from the %build phase of the <i>specfile</i> .

<i>option Value</i>	Description
-bi <i>specfile</i>	Starts build from the %install phase of the <i>specfile</i> .
-bl <i>specfile</i>	Starts check from the %file phase of the <i>specfile</i> .
-ba <i>specfile</i>	Uses the <i>specfile</i> to build the source code package and binary package.
-bb <i>specfile</i>	Uses the <i>specfile</i> to build the binary package.
-bs <i>specfile</i>	Uses the <i>specfile</i> to build the source code package.
-rp <i>sourcefile</i>	Starts build from the %prep phase of the <i>sourcefile</i> (decompress the source code package and install the patch).
-rc <i>sourcefile</i>	Starts build from the %build phase of the <i>sourcefile</i> .
-ri <i>sourcefile</i>	Starts build from the %install phase of the <i>sourcefile</i> .
-rl <i>sourcefile</i>	Starts build from the %file phase of the <i>sourcefile</i> .
-ra <i>sourcefile</i>	Uses the <i>sourcefile</i> to build the source code package and binary package.
-rb <i>sourcefile</i>	Uses the <i>sourcefile</i> to build the binary package.
-rs <i>sourcefile</i>	Uses the <i>sourcefile</i> to build the source code package.
-tp <i>tarfile</i>	Starts build from the %prep phase of the <i>tarfile</i> (decompress the source code package and install the patch).
-tc <i>tarfile</i>	Starts build from the %build phase of the <i>tarfile</i> .
-ti <i>tarfile</i>	Starts build from the %install phase of the <i>tarfile</i> .
-ta <i>tarfile</i>	Uses the <i>tarfile</i> to build the source code package and binary package.
-tb <i>tarfile</i>	Uses the <i>tarfile</i> to build the binary package.
-ts <i>tarfile</i>	Uses the <i>tarfile</i> to build the source code package.
--buildroot= <i>DIRECTORY</i>	During the build, uses <i>DIRECTORY</i> to overwrite the default /root directory.

<i>option Value</i>	Description
--clean	Deletes the files in the BUILD directory.
--nobuild	No actual build steps are performed. It can be used to test the .spec file.
--noclean	Skips the %clean phase of the .spec file (even if it does exist).
--nocheck	Skips the %check phase of the .spec file (even if it does exist).
--dbpath <i>DIRECTORY</i>	Uses the database in DIRECTORY instead of the default directory /var/lib/rpm .
--root <i>DIRECTORY</i>	Sets <i>DIRECTORY</i> to the highest level. The default value is / , indicating the highest level.
--rebuild <i>sourcefile</i>	Installs the specified source code package <i>sourcefile</i> , that is, start preparation, compilation, and installation of the source code package.
--recompile <i>sourcefile</i>	Builds a new binary package based on --recompile . When the build is complete, the build directory, source code, and .spec file are deleted. The deletion effect is the same as that of --clean .
-, --help	Displays detailed help information.
--version	Displays detailed version information.

5.2 Building an RPM Package Locally

This section uses an example to describe how to build an RPM software package locally.

5.2.1 Setting Up the Development Environment

Prerequisites

You have obtained the **root** permission, and have configured a repo source for openEuler.

Procedure

You can use the DNF tool to install rpmdevtools, including the **rpm-build** command and related dependencies (such as make and gdb). Run the following command:

```
# dnf install rpmdevtools*
```

5.2.2 Creating a Hello World RPM Package

The following uses the packaging process of the GNU Hello World project as an example. The package contains the most common peripheral components related to the typical Free and Open Source Software (FOSS) project, including the configuration, compilation, and installation environments, documents, and internationalization (i18n) information.

5.2.2.1 Obtaining the Source Code

Run the following command to download the source code of the official example:

```
# cd ~/rpmbuild/SOURCES
# wget http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
```

5.2.2.2 Editing the SPEC File

Run the following command to create the .spec file:

```
# vi hello.spec
```

Write the corresponding content to the file and save the file. The following is an example of the file content. Modify the corresponding fields based on the actual requirements.

```
Name:      hello
Version:   2.10
Release:   1%{?dist}
Summary:   The "Hello World" program from GNU
Summary(zh CN): GNU Hello World program
License:   GPLv3+
URL:       http://ftp.gnu.org/gnu/hello
Source0:   http://ftp.gnu.org/gnu/hello/%{name}-%{version}.tar.gz

BuildRequires: gettext
Requires(post): info
Requires(preun): info

%description
The "Hello World" program, done with all bells and whistles of a proper FOSS
project, including configuration, build, internationalization, help files, etc.

%description -l zh_CN
The Hello World program contains all parts required by the FOSS project, including
configuration, build, i18n, and help files.

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
make install DESTDIR=%{buildroot}
%find_lang %{name}
rm -f %{buildroot}/%{_infodir}/dir

%post
```

```
/sbin/install-info %[_infodir]/%{name}.info %[_infodir]/dir || :

%preun
if [ $1 = 0 ] ; then
/sbin/install-info --delete %[_infodir]/%{name}.info %[_infodir]/dir || :
fi

%files -f %{name}.lang
%doc AUTHORS ChangeLog NEWS README THANKS TODO
%license COPYING
%{_mandir}/man1/hello.1.*
%[_infodir]/hello.info.*
%[_bindir]/hello

%changelog
* Thu Dec 26 2019 Your Name <youremail@xxx.xxx> - 2.10-1
- Update to 2.10
* Sat Dec 3 2016 Your Name <youremail@xxx.xxx> - 2.9-1
- Update to 2.9
```

- The **Name** tag indicates the software name, the **Version** tag indicates the version number, and the **Release** tag indicates the release number.
- The **Summary** tag is a brief description. The first letter of the tag must be capitalized to prevent the rpmlint tool (packaging check tool) from generating alarms.
- The **License** tag describes the protocol version of the software package. The packager is responsible for checking the license status of the software, which can be implemented by checking the source code or license file or communicating with the author.
- The **Group** tag is used to classify software packages by `/usr/share/doc/rpm-/GROUPS`. Currently, this tag has been discarded. However, the VIM template still has this tag. You can delete it. However, adding this tag does not affect the system. The **%changelog** tag should contain the log of changes made for each release, especially the description of the upstream security/vulnerability patches. The **%changelog** tag should contain the version string to avoid the rpmlint tool from generating alarms.
- If multiple lines are involved, such as %changelog or %description, start from the next line of the instruction and end with a blank line.
- Some unnecessary lines (such as BuildRequires and Requires) can be commented out with a number sign (#) at the beginning of the lines.
- The default values of **%prep**, **%build**, **%install**, and **%file** are retained.

5.2.2.3 Building an RPM Package

Run the following command in the directory where the .spec file is located to build the source code, binary files, and software packages that contain debugging information:

```
# rpmbuild -ba hello.spec
```

Run the following command to view the execution result:

```
# tree ~/rpmbuild/*RPMS

/home/testUser/rpmbuild/RPMS
├─ aarch64
│   ├── hello-2.10-1.aarch64.rpm
│   ├── hello-debuginfo-2.10-1.aarch64.rpm
│   └─ hello-debugsource-2.10-1.aarch64.rpm
```

```
/home/testUser/rpmbuild/SRPMs  
└─ hello-2.10-1.src.rpm
```

5.3 Building an RPM Package Using the OBS

This section describes how to build RPM software packages using the OBS on the web page or with OSC. There are two methods:

- **Modifying an existing software package:** Modify the source code of an existing software package and build the modified source code into an RPM software package.
- **Adding a software package:** A new software source file is developed from scratch, and the newly developed source file is used to build an RPM software package.

5.3.1 OBS Overview

OBS is a general compilation framework based on the openSUSE distribution. It is used to build source code packages into RPM software packages or Linux images. OBS uses the automatic distributed compilation mode and supports the compilation of images and installation packages of multiple Linux OS distributions (such as openEuler, SUSE, and Debian) on multiple architecture platforms (such as x86 and ARM64).

OBS consists of the backend and frontend. The backend implements all core functions. The frontend provides web applications and APIs for interaction with the backend. In addition, OBS provides an API command line client OSC, which is developed in an independent repository.

OBS uses the project organization software package. Basic permission control, related repository, and build targets (OS and architecture) can be defined in the project. A project can contain multiple subprojects. Each subproject can be configured independently to complete a task.

5.3.2 Building an RPM Software Package Online

This section describes how to build an RPM software package online on OBS.

5.3.2.1 Building an Existing Software Package

NOTE

- If you use OBS for the first time, register an individual account on the OBS web page.
- With this method, you must copy the modified code and commit it to the code directory before performing the following operations. The code directory is specified in the `_service` file.

To modify the source code of the existing software and build the modified source file into an RPM software package on the OBS web client, perform the following steps:

1. Log in to OBS at <http://openeuler-build.huawei.com/>.
2. Click **All Projects**. The **All Projects** page is displayed.
3. Click the project to be modified. The project details page is displayed. For example, click **openEuler:Mainline**.
4. On the project details page, search for the software package to be modified and click the software package name. The software package details page is displayed.
5. Click **Branch package**. In the displayed dialog box, click **Accept**, as shown in Figure 5-1.

Figure 5-1 Branch Confirmation page

Branch Confirmation

Source

openEuler:Mainline / kernel

Destination

home:testUser:branc...openEuler:Mainline

► More options

Cancel

Accept

- Click the **_service** file to go to the editing page, modify the file content, and click **Save**. An example of the **_service** file content is as follows. **userCodeURL** and **userCommitID** indicate the user code path and commission version number or branch, respectively.

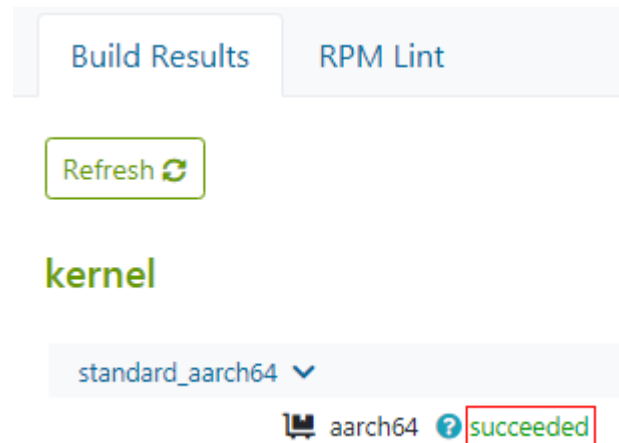
```
<services>
  <service name="tar_scm_kernel">
    <param name="scm">git</param>
    <param name="url">userCodeURL</param>
    <param name="revision">userCommitID</param>
  </service>
  <service name="recompress">
    <param name="compression">bz2</param>
    <param name="file">*.tar</param>
  </service>
</services>
```

NOTE

Click **Save** to save the **_service** file. OBS downloads the source code from the specified URL to the software directory of the corresponding OBS project based on the **_service** file description and replaces the original file. For example, the **kernel** directory of the **openEuler:Mainline** project in the preceding example.

- After the files are copied and replaced, OBS automatically starts to build the RPM software package. Wait until the build is complete and view the build status in the status bar on the right.
 - succeeded**: The build is successful. You can click **succeeded** to view the build logs, as shown in Figure 5-2.

Figure 5-2 Succeeded page



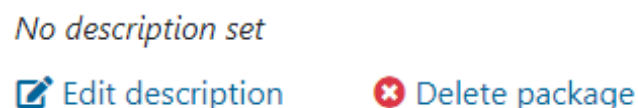
- **failed:** The build failed. Click **failed** to view error logs, locate the fault, and rebuild again.
- **unresolvable:** The build is not performed. The possible cause is that the dependency is missing.
- **disabled:** The build is manually closed or is queuing for build.
- **excluded:** The build is prohibited. The possible cause is that the .spec file is missing or the compilation of the target architecture is prohibited in the .spec file.

5.3.2.2 Adding a Software Package

To add a new software package on the OBS web page, perform the following steps:

1. Log in to the OBS console.
2. Select a project based on the dependency of the new software package. That is, click **All Projects** and select the corresponding project, for example, **openEuler:Mainline**.
3. Click a software package in the project. The software package details page is displayed.
4. Click **Branch package**. On the confirmation page that is displayed, click **Accept**.
5. Click **Delete package** to delete the software package in the new subproject, as shown in Figure 5-3.

Figure 5-3 Deleting a software package from a subproject



NOTE

The purpose of creating a project by using existing software is to inherit the dependency such as the environment. Therefore, you need to delete these files.

6. Click **Create Package**. On the page that is displayed, enter the software package name, title, and description, and click **Create** to create a software package, as shown in Figure 5-4 and Figure 5-5.

Figure 5-4 Create Package page

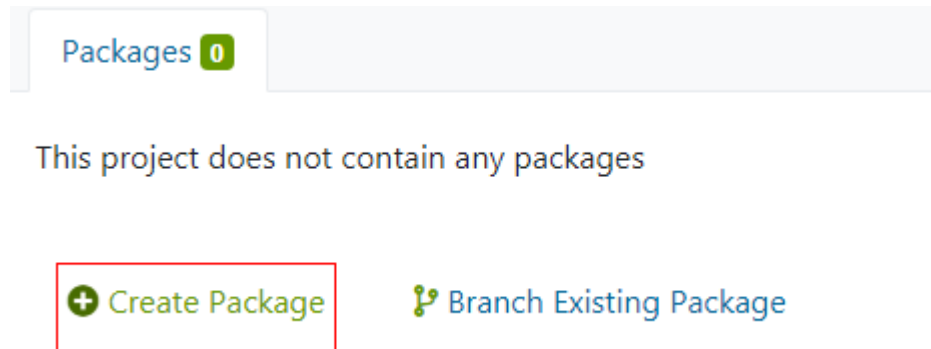


Figure 5-5 Creating a software package

Create Package for home:testUser:branches:openEuler:Mainline

Name:

Title:

Description:

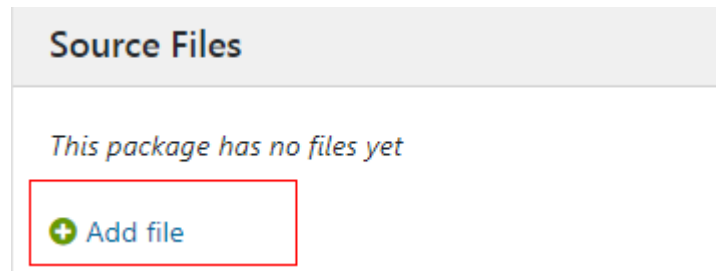
☐ Deny access to source of package

☐ Disable build results publishing

Cancel Create

7. Click **Add file** to upload the .spec file and the file to be compiled (specified in the .spec file), as shown in Figure 5-6.

Figure 5-6 Add file page



8. After the file is uploaded, OBS automatically starts to build the RPM software package. Wait until the build is complete and view the build status in the status bar on the right.
 - **succeeded**: The build is successful. You can click **succeeded** to view the build logs.
 - **failed**: The build failed. Click **failed** to view error logs, locate the fault, and rebuild again.
 - **unresolvable**: The build is not performed. The possible cause is that the dependency is missing.
 - **disabled**: The build is manually closed or is queuing for build.
 - **excluded**: The build is prohibited. The possible cause is that the .spec file is missing or the compilation of the target architecture is prohibited in the .spec file.

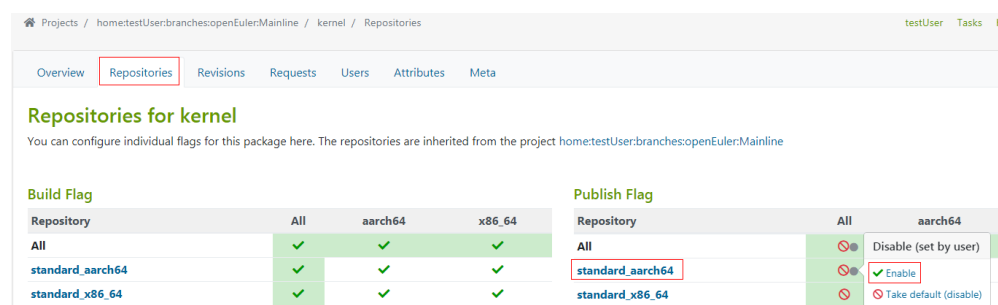
5.3.2.3 Obtaining the Software Package

After the RPM software package is built, perform the following operations to obtain the RPM software package on the web page:

1. Log in to the OBS console.
2. Click **All Projects** and find the project corresponding to the required software package, for example, **openEuler:Mainline**.
3. Click the name of the required software package in the project. The software package details page is displayed, for example, the **kernel** page in the preceding example.
4. Click the **Repositories** tab. On the software repository management page that is displayed, click **Enable** in **Publish Flag** to enable the RPM software package download

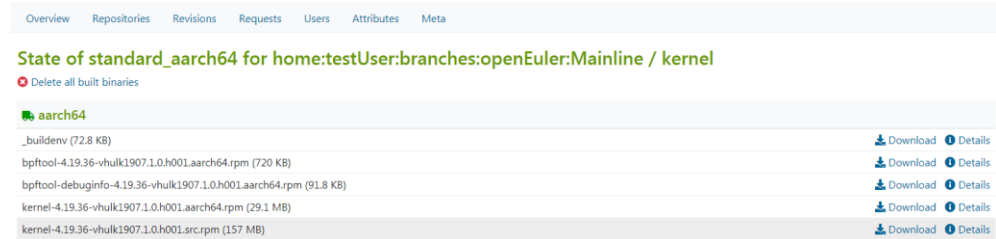
function (the status changes from  to ), as shown in Figure 5-7.

Figure 5-7 Repositories page



- Click the project name in the **Repository** column. On the RPM software package download page that is displayed, click **Download** on the right of the RPM software package to download the RPM software package, as shown in Figure 5-8.

Figure 5-8 RPM software package download page



5.3.3 Building a Software Package Using OSC

This section describes how to use the OBS command line tool OSC to create a project and build an RPM software package.

5.3.3.1 Installing and Configuring the OSC

Prerequisites

You have obtained the **root** permission, and have configured a repo source for openEuler.

Procedure

- Step 1** Install the OSC command line tool and its dependency as the **root** user.

```
# dnf install osc build
```

NOTE

The compilation of RPM software packages depends on build.

- Step 2** Configure the OSC.

- Run the following command to open the `~/.osrcrc` file:

```
# vi ~/.osrcrc
```

- Add the **user** and **pass** fields to `~/.osrcrc`. The values of *userName* and *passWord* are the account and password registered on the OBS website (<http://openeuler-build.huawei.com/>).

```
[general]
apiurl = http://openeuler-build.huawei.com/
no verify = 1
[http://openeuler-build.huawei.com/]
user=userName
pass=passWord
```

- If the domain name **openeuler-build.openeuler.org** cannot be resolved, manually add the following line to the `/etc/hosts` file: *ip-address* indicates the IP address of OBS, which is `http://117.78.1.88/`.

```
ip-address openeuler-build.openeuler.org
```

----End

5.3.3.2 Building an Existing Software Package

Creating a Project

1. You can copy an existing project to create a subproject of your own. For example, to copy the **zlib** software package in the **openEuler:Mainline** project to the new branch, run the following command:

```
# osc branch openEuler:Mainline zlib
```

If the following information is displayed, a new branch project **home:testUser:branches:openEuler:Mainline** is created for user **testUser**.

A working copy of the branched package can be checked out with:

```
osc co home:testUser:branches:openEuler:Mainline/zlib
```

2. Download the configuration file (for example, **_service**) of the software package to be modified to the local directory. In the preceding command, *testUser* indicates the account name configured in the **~/.osrc** configuration file. Change it based on the actual requirements.

```
# osc co home:testUser:branches:openEuler:Mainline/zlib
```

Information similar to the following is displayed:

```
A home:testUser:branches:openEuler:Mainline
A home:testUser:branches:openEuler:Mainline/zlib
A home:testUser:branches:openEuler:Mainline/zlib/_service
```

3. Go to the local subproject directory and synchronize the remote code of the software package to the local host.

```
# cd home:testUser:branches:openEuler:Mainline/zlib
# osc up -S
```

Information similar to the following is displayed:

```
A service:tar scm kernel repo:0001-Neon-Optimized-hash-chain-rebase.patch
A service:tar scm kernel repo:0002-Porting-optimized-longest match.patch
A service:tar scm kernel repo:0003-arm64-specific-build-patch.patch
A service:tar scm kernel repo:zlib-1.2.11-optimized-s390.patch
A service:tar scm kernel repo:zlib-1.2.11.tar.xz
A service:tar scm kernel repo:zlib-1.2.5-minizip-fixuncrypt.patch
A _service:tar scm kernel repo:zlib.spec
```

Building an RPM Package

4. Rename the source file and add the renamed source file to the temporary storage of OBS.

```
# rm -f service;for file in `ls | grep -v .osc`;do new file=${file##*};mv $file
$new file;done
# osc addremove *
```

5. Modify the source code and .spec file, and synchronize all modifications of the corresponding software package to the OBS server. The following is a command example. The information after the **-m** parameter is the commission record.

```
# osc ci -m "commit log"
```

6. Run the following command to obtain the repository name and architecture of the current project:

```
# osc repos home:testUser:branches:openEuler:Mainline
```

7. After the modification is committed, OBS automatically compiles the software package. You can run the following command to view the compilation logs of the corresponding repository. In the command, *standard_aarch64* and *aarch64* indicate the repository name and architecture obtained in the command output.

```
# osc buildlog standard_aarch64 aarch64
```

NOTE

You can also open the created project on the web client to view the build logs.

5.3.3.3 Adding a Software Package

To use the OSC tool of OBS to add a new software package, perform the following steps:

Creating a Project

1. Create a project based on the dependency of the new software package and a proper project. For example, to create a project based on **zlib** of the **openEuler:Mainline** project, run the following command (**zlib** is any software package in the project):

```
# osc branch openEuler:Mainline zlib
```

2. Delete unnecessary software packages added during project creation. For example, to delete the **zlib** software package, run the following command:

```
# cd home:testUser:branches:openEuler:Mainline
# osc rm zlib
# osc commit -m "commit log"
```

3. Create a software package in your own project. For example, to add the **my-first-obs-package** software package, run the following command:

```
# mkdir my-first-obs-package
# cd my-first-obs-package
```

Building an RPM Package

4. Add the prepared source file and .spec file to the software package directory.
5. Modify the source code and .spec file, and upload all files of the corresponding software package to the OBS server. The following is a command example. The information after the **-m** parameter is the commission record.

```
# cd home:testUser:branches:openEuler:Mainline
# osc add my-first-obs-package
# osc ci -m "commit log"
```

6. Run the following command to obtain the repository name and architecture of the current project:

```
# osc repos home:testUser:branches:openEuler:Mainline
```

7. After the modification is committed, OBS automatically compiles the software package. You can run the following command to view the compilation logs of the corresponding repository. In the command, *standard_aarch64* and *aarch64* indicate the repository name and architecture obtained in the command output.

```
# cd home:testUser:branches:openEuler:Mainline/my-first-obs-package
# osc buildlog standard_aarch64 aarch64
```

NOTE

You can also open the created project on the web client to view the build logs.

5.3.3.4 Obtaining the Software Package

After the RPM software package is built, run the following command to obtain the RPM software package using the OSC:

```
# osc getbinaries home:testUser:branches:openEuler:Mainline my-first-obs-package  
standard_aarch64 aarch64
```

The parameters in the command are described as follows. You can modify the parameters according to the actual situation.

- *home:testUser:branches:openEuler:Mainline*: name of the project to which the software package belongs.
- *my-first-obs-package*: name of the software package.
- *standard_aarch64*: repository name.
- *aarch64*: repository architecture name.

NOTE

You can also obtain the software package built using OSC from the web page. For details, see 5.2.2.3 Building an RPM Package.