



openEuler

20.03 LTS

应用开发指南

发布日期 2020-03-23

目 录

法律声明.....	iv
前言.....	v
1 开发环境准备.....	7
1.1 环境要求	7
1.2 配置 repo 源	8
1.3 安装软件包	10
1.3.1 安装 JDK 软件包.....	10
1.3.2 安装 rpm-build 软件包	11
1.4 使用 IDE 进行 Java 开发.....	11
1.4.1 简介	11
1.4.2 使用 MobaXterm 登录服务器	12
1.4.3 设置 JDK 环境	12
1.4.4 下载安装 GTK 库	12
1.4.5 设置 X11 Forwarding	12
1.4.6 下载并运行 IntelliJ IDEA.....	13
2 使用 GCC 编译	14
2.1 简介	14
2.2 基本规则	14
2.2.1 文件类型	14
2.2.2 编译流程	15
2.2.3 编译选项	15
2.2.4 多源文件编译	17
2.3 库	17
2.3.1 动态链接库	18
2.3.2 静态链接库	19
2.4 示例	19
2.4.1 使用 GCC 编译 C 程序示例.....	19
2.4.2 使用 GCC 创建和使用动态链接库示例	20
2.4.3 使用 GCC 创建和使用静态链接库示例	21

3 使用 make 编译.....	24
3.1 简介	24
3.2 基本规则	24
3.2.1 文件类型	24
3.2.2 make 工作流程.....	25
3.2.3 make 选项.....	25
3.3 Makefile.....	27
3.4 示例	28
3.4.1 使用 Makefile 实现编译的示例	28
4 使用 JDK 编译.....	30
4.1 简介	30
4.2 基本规则	30
4.2.1 文件类型及工具	30
4.2.2 java 程序生成流程.....	31
4.2.3 JDK 常用工具选项.....	31
4.3 类库	35
4.4 示例	36
4.4.1 编译不带包的 java 程序示例	36
4.4.2 编译带包的 java 程序示例	37
5 构建 RPM 包	39
5.1 打包说明	39
5.2 本地构建	43
5.2.1 搭建开发环境	43
5.2.2 创建 Hello World RPM 包	43
5.2.2.1 下载源码	43
5.2.2.2 编辑 SPEC 文件.....	43
5.2.2.3 构建 RPM 包.....	45
5.3 使用 OBS 构建.....	45
5.3.1 OBS 简介	45
5.3.2 在线构建软件包	45
5.3.2.1 构建已有软件包	46
5.3.2.2 新增软件包	47
5.3.2.3 获取软件包	50
5.3.3 使用 osc 构建软件包	50
5.3.3.1 安装并配置 osc	51
5.3.3.2 构建已有软件包	51
5.3.3.3 新增软件包	52
5.3.3.4 获取软件包	53

法律声明

版权所有 © 2020 华为技术有限公司。

您对“本文档”的复制、使用、修改及分发受知识共享(Creative Commons)署名—相同方式共享 4.0 国际公共许可协议(以下简称“CC BY-SA 4.0”)的约束。为了方便用户理解，您可以通过访问 <https://creativecommons.org/licenses/by-sa/4.0/> 了解 CC BY-SA 4.0 的概要 (但不是替代)。CC BY-SA 4.0 的完整协议内容您可以访问如下网址获取：
<https://creativecommons.org/licenses/by-sa/4.0/legalcode>。

商标声明

openEuler 为华为技术有限公司的商标。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

免责声明

本文档仅作为使用指导，除非适用法强制规定或者双方有明确书面约定，华为技术有限公司对本文档中的所有陈述、信息和建议不做任何明示或默示的声明或保证，包括但不限于不侵权，时效性或满足特定目的的担保。

前言

概述

本文档主要介绍如下四部分内容，以指导用户使用 openEuler 并基于 openEuler 进行代码开发。

- 在 openEuler 系统中安装和使用 GCC 编译器，并完成一个简单代码的开发、编译和执行。
- 在 openEuler 系统中使用 JDK 自带工具完成代码的编译和执行。
- 在 openEuler 系统中安装 IntelliJ IDEA 进行 Java 开发。
- 在本地或使用 OBS（Open Build Service）创建 RPM（The RPM Package Manager）软件包的方法。



读者对象

本文档适用于所有使用 openEuler 操作系统进行代码开发的用户。用户需要具备如下经验或能力：

- 具备 Linux 操作系统基础知识
- 了解 Linux 命令行的基本使用方法

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 注意	用于传递设备或环境安全警示信息，若不可避免，可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “注意”不涉及人身伤害。
 说明	用于突出重要/关键信息、最佳实践和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

命令行格式约定

表1 命令行格式的约定

格式	含义
粗体	命令行关键字，即命令中保持不变、必须照输的部分，采用 加粗 字体表示。
<i>斜体</i>	命令行参数，即命令中必须由实际值进行替代的部分，采用 <i>斜体</i> 表示。
[]	用“[]”括起来的部分表示在命令配置时是可选的。
{ x y ... }	表示从两个或多个选项中选择 <code>一个</code> 。
[x y ...]	表示从两个或多个选项中选择 <code>一个或者不选</code> 。
{ x y ... }*	表示从两个或多个选项中选择 <code>多个</code> ，最少选取一个，最多选取所有选项。
[x y ...]*	表示从两个或多个选项中选择 <code>一个、多个或者不选</code> 。

1 开发环境准备

- 1.1 环境要求
- 1.2 配置 repo 源
- 1.3 安装软件包
- 1.4 使用 IDE 进行 Java 开发

1.1 环境要求

- 若使用的是物理机，则开发环境所需的最小硬件要求如表 1-1 所示。

表1-1 最小硬件要求

部件名称	最小硬件要求	说明
架构	<ul style="list-style-type: none">• AArch64• x86	<ul style="list-style-type: none">• 支持 Arm 的 64 位架构。• 支持 Intel 的 x86 64 位架构。
CPU	<ul style="list-style-type: none">• 华为鲲鹏 920 系列 CPU• Intel® Xeon®处理器	-
内存	不小于 4GB（为了获得更好的应用体验，建议不小于 8GB）	-
硬盘	为了获得更好的应用体验，建议不小于 120GB）	支持 IDE、SATA、SAS 等接口的硬盘。

- 若使用的是虚拟机，则开发环境所需的小虚拟化空间要求如表 1-2 所示。

表1-2 最小虚拟化空间要求

部件名称	最小虚拟化空间要求	说明
架构	<ul style="list-style-type: none">• AArch64• x86	-
CPU	2 个 CPU	-
内存	不小于 4GB（为了获得更好的应用体验，建议不小于 8GB）	-
硬盘	不小于 1032GB（为了获得更好的应用体验，建议不小于 120GB）	-

操作系统要求

操作系统要求为 openEuler 操作系统。

openEuler 操作系统具体安装方法请参考《openEuler 1.0 安装指南》，其中“软件选择”页面的“已选环境的附加选项”中将“开发工具”勾选。

1.2 配置 repo 源

📖 说明

本章节中以 openEuler-20.03-LTS-aarch64-dvd.iso 镜像文件和 openEuler-20.03-LTS-aarch64-dvd.iso.sha256sum 校验文件为例，请根据实际需要的镜像文件和校验文件进行修改。

下载 ISO 镜像

- 通过跨平台文件传输工具下载 ISO 镜像
 - a. 登录 openEuler 社区，网址为：<https://openeuler.org>。
 - b. 单击“下载”，进入下载页面。
 - c. 单击“获取 ISO: ”后面的“Link”，显示下载列表。
 - d. 单击“openEuler-20.03-LTS-aarch64-dvd.iso”将 openEuler 发布包下载到本地。
 - e. 单击“openEuler-20.03-LTS-aarch64-dvd.iso.sha256sum”将 openEuler 校验文件下载到本地。
 - f. 登录 openEuler 操作系统，新建用于存放发布包和检验文件的目录,如“/home/iso”。

```
mkdir /home/iso
```

 - g. 使用跨平台文件传输工具（如 WinSCP）将本地的 openEuler 发布包和校验文件上传到 openEuler 操作系统。

- 通过 `wget` 命令下载 ISO 镜像
 - a. 登录 openEuler 社区，网址为：<https://openeuler.org>。
 - b. 单击“下载”，进入下载页面。
 - c. 单击“获取 ISO:”后面的“Link”，显示下载列表。
 - d. 右键单击“openEuler-20.03-LTS-aarch64-dvd.iso”，单击“复制链接地址”，将 openEuler 发布包地址记录好。
 - e. 右键单击“openEuler-20.03-LTS-aarch64-dvd.iso.sha256sum”，单击“复制链接地址”，将 openEuler 校验文件地址记录好。
 - f. 登录 openEuler 操作系统，新建用于存放发布包和检验文件的目录，如“/home/iso”，并切换到该目录。

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

- g. 使用 `wget` 命令远程下载发布包和检验文件，命令中的 `ipaddriso` 和 `ipaddrisum` 分别为 **d** 和 **e** 中记录的地址。

```
wget ipaddriso
wget ipaddrisum
```

发布包完整性校验

1. 获取校验文件中的校验值。

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

2. 计算 openEuler 发布包的 sha256 校验值。

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

命令执行完成后，输出校验值。

3. 对比步骤 1 和步骤 2 计算的校验值是否一致。

如果校验值一致说明 iso 文件完整性没有破坏，如果校验值不一致则可以确认文件完整性已被破坏，需要重新获取。

挂载 ISO 并配置为 repo 源

使用 `mount` 命令挂载镜像文件。

示例如下：

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

挂载好的 `mnt` 目录如下：

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

其中，`Packages` 为 rpm 包所在的目录，`repodata` 为 repo 源元数据所在的目录，`RPM-GPG-KEY-openEuler` 为 openEuler 的签名公钥。。

挂载后的目录可以配置为 yum 源使用，在 `/etc/yum.repos.d/` 目录下创建 `***.repo` 的配置文件（必须以 `.repo` 为扩展名）。

示例如下：

在 `/etc/yum.repos.d` 目录下创建 `openEuler.repo` 文件，使用本地镜像挂载目录作为 yum 源，`openEuler.repo` 的内容如下：

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

📖 说明

- `gpgcheck` 可设置为 1 或 0，1 表示进行 `gpg`（GNU Private Guard）校验，0 表示不进行 `gpg` 校验，`gpgcheck` 可以确定 rpm 包的来源是有效和安全的。
- `gpgkey` 为签名公钥的存放路径。

1.3 安装软件包

安装开发过程中需要用到的软件。不同的开发需要的软件不一样，但安装方法相同，本章以安装常用的几个软件包（JDK，`rpm-build`）为例。有些开发软件 openEuler 操作系统已默认自带，如 `GCC`、`GNU make`。

1.3.1 安装 JDK 软件包

步骤 1 执行 `dnf list installed | grep jdk` 查询 JDK 软件是否已安装。

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

查看命令打印信息，若打印信息中包含“`jdk`”，表示该软件已经安装了，则不需要再安装。若无任何打印信息，则表示该软件未安装。

步骤 2 清除缓存。

```
dnf clean all
```

步骤 3 创建缓存。

```
dnf makecache
```

步骤 4 查询可安装的 JDK 软件包。

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

查看命令打印信息，选择安装 `java-x.x.x-openjdk-devel.aarch64` 软件包。其中 `x.x.x` 为版本号。

步骤 5 安装 JDK 软件包，以安装 java-1.8.0-openjdk-devel 软件包为例。

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

步骤 6 查询 JDK 软件版本。

```
java -version
```

查看打印信息，若打印信息中包括“openjdk version "1.8.0_232"”信息，表示已正确安装，其中 1.8.0_232 为版本号。

----结束

1.3.2 安装 rpm-build 软件包

步骤 1 执行 `dnf list installed | grep rpm-build` 查询 rpm-build 软件是否已安装。

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

查看命令打印信息，若打印信息中包含“rpm-build”，表示该软件已经安装了，则不需要再安装。若无任何打印信息，则表示该软件未安装。

步骤 2 清除缓存。

```
dnf clean all
```

步骤 3 创建缓存。

```
dnf makecache
```

步骤 4 安装 rpm-build 软件包。

```
dnf install rpm-build
```

步骤 5 查询 rpm-build 软件版本。

```
rpm-build --version
```

----结束

1.4 使用 IDE 进行 Java 开发

对于小型的 Java 程序，可以直接使用 JDK 编译得到可运行 Java 应用。但是对于大中型 Java 应用，这种方式已经无法满足开发者的需求。因此您可以参考如下步骤安装 IDE 并进行使用，以方便您在 openEuler 系统上的 Java 开发工作。

1.4.1 简介

IntelliJ IDEA 是一款非常流行的 Java IDE，其社区版可以免费下载使用。目前 openEuler 支持使用 IntelliJ IDEA 集成开发环境（IDE）进行 Java 程序的开发，从而可以提升开发人员的工作效率。

1.4.2 使用 MobaXterm 登录服务器

MobaXterm 是一款非常优秀的 SSH 客户端，其自带 X Server，可以轻松解决远程 GUI 显示问题。

您需要提前下载安装好 MobaXterm 并打开，然后 SSH 登录您的服务器并进行以下操作。

1.4.3 设置 JDK 环境

在设置 JAVA_HOME 之前您需要先找到 JDK 的安装路径。在 1.3.1 安装 JDK 软件包中您已经学会了如何安装 JDK，如果您还没安装好 JDK，请提前安好。

查看 java 路径，命令如下：

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

查看软链接的实际指向目录，命令如下：

```
# 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
```

发现 JDK 的真实路径为/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.232.b09-1.h2.aarch64，设置 JAVA_HOME 和 PATH，命令如下：

```
# 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 下载安装 GTK 库

请确认您具有 root 权限后，运行如下命令：

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

如果显示 gtk2 或者 gtk3，则表示您已安装该库，可以直接跳过进入下一步，否则运行如下命令自动下载安装 gtk 库。

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

1.4.5 设置 X11 Forwarding

切换到 sshd 配置目录

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

如果该目录不存在，则创建目录后再进行切换，创建目录命令如下：

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

然后在.ssh 目录下编辑 config 文件并保存：

1. 使用 vim 打卡 config 文件

```
# vim config
```

2. 将以下内容添加到文件末尾并保存:

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

1.4.6 下载并运行 IntelliJ IDEA

在执行如上环境配置后，您就可以下载使用 IntelliJ IDEA 了。鉴于最新版的 IntelliJ IDEA 和 openEuler 系统在部分功能上有兼容性问题，建议您从此[链接](#)下载 2018 版本 linux 压缩包。下载好后把压缩包移到您想要安装该软件的目录，对压缩包进行解压：

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

解压后切换到 IntelliJ IDEA 的目录下并运行。

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

2 使用 GCC 编译

本章介绍 GCC 编译的一些基本知识，并通过示例进行实际演示。更多的 GCC 知识请通过 `man gcc` 命令查询。

2.1 简介

2.2 基本规则

2.3 库

2.4 示例

2.1 简介

GCC (GNU Compiler Collection) 是 GNU 推出的功能强大、性能优越的多平台编译器。GCC 编译器能将 C、C++ 语言源程序、汇编程序和目标程序编译、连接成可执行文件。openEuler 操作系统中已默认安装了 GCC 软件包。

2.2 基本规则

2.2.1 文件类型

对于任何给定的输入文件，文件类型决定进行何种编译。GCC 常用的文件类型如表 2-1 所示。

表2-1 GCC 常用的文件类型

扩展名 (后缀)	说明
.c	C 语言源代码文件。
.C, .cc 或 .cxx	C++ 源代码文件。
.m	Objective-C 源代码文件。
.s	汇编语言源代码文件。

扩展名（后缀）	说明
.i	已经预处理的 C 源代码文件。
.ii	已经预处理的 C++源代码文件。
.S	已经预处理的汇编语言源代码文件。
.h	程序所包含的头文件。
.o	编译后的目标文件。
.so	动态链接库，它是一种特殊的目标文件。
.a	静态链接库。
.out	可执行文件，但可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。如果没有给出可执行文件的名称，GCC 将生成一个名为 <code>a.out</code> 的文件。

2.2.2 编译流程

使用 GCC 将源代码文件生成可执行文件，需要经过预处理、编译、汇编和链接。

1. 预处理：将源程序（如.c 文件）预处理，生成.i 文件。
2. 编译：将预处理后的.i 文件编译成为汇编语言，生成.s 文件。
3. 汇编：将汇编语言文件经过汇编，生成目标文件.o 文件。
4. 链接：将各个模块的.o 文件链接起来生成一个可执行程序文件。

其中.i 文件、.s 文件、.o 文件是中间文件或临时文件，如果使用 GCC 一次性完成 C 语言程序的编译，则这些文件会被删除。

2.2.3 编译选项

GCC 编译的命令格式为：`gcc [options] [filenames]`

其中：

options：编译选项。

filenames：文件名称。

GCC 是一个功能强大的编译器，其 *options* 参数取值很多，但有些大部分并不常用，常用的 *options* 取值如表 2-2 所示。

表2-2 GCC 常用的编译选项

<i>options</i> 取值	说明	示例
-------------------	----	----

<i>options</i> 取值	说明	示例
-c	编译、汇编指定的源文件生成目标文件，但不进行链接。通常用于编译不包含主程序的子程序文件。	#使用-c 选项编译 test1.c、test2.c 源文件 gcc -c test1.c test2.c
-S	编译指定的源文件生成以.s 作为后缀的汇编语言文件，但不进行汇编。	#编译器预处理 circle.c，将其翻译成汇编语言，并将结果存储在 circle.s 文件中。 gcc -S circle.c
-E	预处理指定的源文件，但不进行编译。 默认情况下，预处理器的输出会被导入到标准输出流（如显示器），可以利用-o 选项把它导入到某个输出文件。	#预处理的结果导出到 circle.i 文件。 gcc -E circle.c -o circle.i
-o <i>file</i>	用在生成可执行文件时，生成指定的输出文件 <i>file</i> 。同时该名称不能和源文件同名。如果不给出这个选项，gcc 就给出预设的可执行文件 a.out。	#将源文件作为输入文件，将可执行文件作为输出文件，也即完整地编译整个程序。 gcc main.c func.c -o app.out
-g	在可执行程序中包含标准调试信息。	-
-L <i>library_path</i>	在库文件的搜索路径列表中添加 <i>library_path</i> 路径。	-
-l <i>library</i>	链接时搜索指定的函数库 <i>library</i> 。 使用 GCC 编译和链接程序时，GCC 默认会链接 libc.a 或者 libc.so，但是对于其他的库（例如非标准库、第三方库等），就需要手动添加。	#使用-l 选项，以链接数学库。 gcc main.c -o main.out -lm 说明 数学库的文件名是 libm.a。前缀 lib 和后缀.a 是标准的，m 是基本名称，GCC 会在-l 选项后紧跟着的基本名称的基础上自动添加这些前缀、后缀，本例中，基本名称为 m。
-I <i>head_path</i>	在头文件的搜索路径列表中添加 <i>head_path</i> 路径。	-
-static	进行静态编译，及链接静态库，禁止链接动态库。	-

<i>options</i> 取值	说明	示例
-shared	默认选项，可省略。 <ul style="list-style-type: none">可以生成动态库文件。进行动态编译，优先链接动态库，只有没有动态库是才会链接同名的静态库。	-
-fPIC（或-fpic）	生成使用相对地址的位置无关的目标代码。通常使用-static 选项从该 PIC 目标文件生成动态库文件。	-

2.2.4 多源文件编译

多个源文件的编译方法有 2 种。

- 多个源文件一起编译。编译时需要所有文件重新编译。

示例：将 test1.c 和 tes2.c 分别编译后链接成 test 可执行文件。

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

- 分别编译各个源文件，之后对编译后输出的目标文件链接。编译时只重新编译修改的文件，未修改的文件不用重新编译。

示例：分别编译 test1.c, test2.c, 在将二者的目标文件 test1.o, test2.o 链接成 test 可执行文件。

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

2.3 库

库是写好的、现有的、成熟的、可以复用的代码。每个程序都要依赖很多基础的底层库。

库文件在命名时约定，以 lib 为前缀，以.so（动态库）或.a（静态库）为后缀，中间为库文件名。如 libfoo.so 或 libfoo.a。由于所有的库文件都遵循了同样的规范，因此当在链接库时，-l 选项指定链接的库文件名时可以省去 lib 前缀，即 GCC 在对-lfoo 进行处理时，会自动去链接名为 libfoo.so 或 libfoo.a 的库文件。而当在创建库时，必须指定完整文件名 libfoo.so 或 libfoo.a。

根据链接时期的不同，库分为静态库和动态库。静态库是在链接阶段，将汇编生成的目标文件.o 与引用到的库一起链接打包到可执行文件中；而动态库是在程序编译时并不会被链接到目标代码中，而是在程序运行时才被载入。二者有如下差异。

- 资源利用不一样。

静态库为生成的可执行文件的一部分，而动态库为单独的文件。所以使用静态库和动态库的可执行文件大小和占用的磁盘空间大小不一样，导致资源利用不一样。

- 扩展性与兼容性不一样

静态库中某个函数的实现变了，那么可执行文件必须重新编译，而对于动态链接生成的可执行文件，只需要更新动态库本身即可，不需要重新编译可执行文件。

- 依赖不一样

静态库的可执行文件不需要依赖其他的内容即可运行，而动态库的可执行文件必须依赖动态库的存在。所以静态库更方便移植。

- 加载速度不一样

静态库在链接时就和可执行文件在一块了，而动态库在加载或者运行时才链接，因此，对于同样的程序，静态链接的要比动态链接加载更快。

2.3.1 动态链接库

使用 `-shared` 选项 和 `-fPIC` 选项，可直接使用源文件、汇编文件或者目标文件创建一个动态库。其中 `-fPIC` 选项作用于编译阶段，在生成目标文件时就需要使用该选项，以生成位置无关的代码。

示例 1：从源文件生成动态链接库。

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

示例 2：从目标文件生成动态链接库。

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

将一个动态库链接到可执行文件，需要在命令行中列出动态库的名称。

示例：将 `main.c` 和 `libtest.so` 一起编译成 `app.out`，当 `app.out` 运行时，会动态地加载链接库 `libtest.so`。

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

这种方式是直接指定使用当前目录下的 `libtest.so` 文件。

若使用下面搜索动态库的方式，则为了确保程序在运行时能够链接到动态库，需要通过如下三种方法中的任一种实现。

- 将动态库保存在标准目录下，例如 `/usr/lib`。
- 把动态库所在路径 `libraryDIR` 增加到环境变量 `LD_LIBRARY_PATH` 中
`export LD_LIBRARY_PATH=libraryDIR:$LD_LIBRARY_PATH`

📖 说明

`LD_LIBRARY_PATH` 为动态库的环境变量。当运行动态库时，若动态库不在缺省文件夹（`/lib` 和 `/usr/lib`）下，则需要指定环境变量 `LD_LIBRARY_PATH`。

- 把动态库所在路径 `libraryDIR` 增加 `/etc/ld.so.conf` 中然后执行 `ldconfig` 或者以动态库所在路径 `libraryDIR` 为参数执行 `ldconfig`。

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

2.3.2 静态链接库

创建一个静态链接库，需要先将源文件编译为目标文件，然后在使用 `ar` 命令将目标文件打包成静态链接库。

示例：将源文件 `test1.c`，`test2.c`，`test3.c` 编译并打包成静态库。

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

其中 `ar` 是一个备份压缩命令，可以将多个文件打包成一个备份文件（也叫归档文件），也可以从备份文件中提取成员文件。`ar` 最常见的用法是将目标文件打包为静态链接库。

`ar` 将目标文件打包成静态链接库的命令格式为：

`ar rcs` 静态库文件名 目标文件列表

- `r`: 替换库中已有的目标文件，或者加入新的目标文件。
- `c`: 创建一个库，不管库是否存在，都将创建。
- `s`: 创建目标文件索引，在创建较大的库时能提高速度。

示例：创建一个 `main.c` 文件来使用静态库

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

其中 `libraryDIR` 为 `libtest.a` 库的路径。

2.4 示例

2.4.1 使用 GCC 编译 C 程序示例

步骤 1 `cd` 到代码目录，此处以用户“`/home/code`”进行举例。如下所示：

```
cd /home/code
```

步骤 2 编写 Hello World 程序，保存为 `helloworld.c`，此处以编译 Hello World 程序进行举例说明。示例如下：

```
vi helloworld.c
```

代码内容示例：

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

步骤 3 在代码目录，执行编译，使用命令：

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

编译执行未报错，表明执行通过。

步骤 4 编译完成后，会生成 helloworld 文件，查看编译结果，示例如下：

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

----结束

2.4.2 使用 GCC 创建和使用动态链接库示例

步骤 1 cd 到代码目录，此处以用户“/home/code”进行举例。并在该目录下创建 src，lib，include 子目录，分别用于存放源文件，动态库文件和头文件。

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

步骤 2 cd 到/home/code/src 目录，创建 2 个函数 add.c、sub.c，分别实现加、减。

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

add.c 代码内容示例：

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

sub.c 代码内容示例：

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

步骤 3 将 add.c、sub.c 源文件创建为动态库 libmath.so，并将该动态库存放在/home/code/lib 目录。

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

步骤 4 cd 到/home/code/include 目录，创建 1 个头文件 math.h，声明函数的头文件。

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

math.h 代码内容示例：

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

步骤 5 cd 到/home/code/src 目录，创建一个调用 add()和 sub()的 main.c 函数。

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

math.c 代码内容示例:

```
#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;  
}
```

步骤 6 将 main.c 和 libmath.so 一起编译成 math.out。

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

步骤 7 将动态链接库所在的路径加入到环境变量中。

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

步骤 8 执行 math.out。

```
./math.out
```

执行结果如下所示:

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

----结束

2.4.3 使用 GCC 创建和使用静态链接库示例

步骤 1 cd 到代码目录，此处以用户“/home/code”进行举例。并在该目录下创建 src，lib，include 子目录，分别用于存放源文件，动态库文件和头文件。

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

步骤 2 cd 到/home/code/src 目录，创建 2 个函数 add.c、sub.c，分别实现加、减。

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

add.c 代码内容示例:

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

```
{  
    return a+b;  
}
```

sub.c 代码内容示例:

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

步骤 3 将 add.c、sub.c 源文件编译为目标文件 add.o、sub.o。

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

步骤 4 将 add.o、sub.o 目标文件通过 ar 命令打包成静态库 libmath.a，并将该静态库存放在 /home/code/lib 目录。

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

步骤 5 cd 到/home/code/include 目录，创建 1 个头文件 math.h，声明函数的头文件。

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

math.h 代码内容示例:

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

步骤 6 cd 到/home/code/src 目录，创建一个调用 add()和 sub()的 main.c 函数。

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

main.c 代码内容示例:

```
#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;  
}
```

步骤 7 将 main.c 和 libmath.a 一起编译成 math.out。

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

步骤 8 执行 math.out。

```
./math.out
```

执行结果如下所示：

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

----结束

3 使用 make 编译

本章介绍 make 编译的一些基本知识，并通过示例进行实际演示。更多的 make 知识请通过 `man make` 命令查询。

3.1 简介

3.2 基本规则

3.3 Makefile

3.4 示例

3.1 简介

GNU make 实用程序（通常缩写为 make）是一种用于控制从源文件生成可执行文件的工具。make 会自动确定复杂程序的哪些部分已更改并需要重新编译。make 使用称为 Makefiles 的配置文件来控制程序的构建方式。

3.2 基本规则

3.2.1 文件类型

makefile 文件中可能用到的文件类型如表 3-1 所示。

表3-1 GCC 常用的文件类型

扩展名（后缀）	说明
.c	C 语言源代码文件。
.C, .cc 或.cxx	C++源代码文件。
.m	Objective-C 源代码文件。
.s	汇编语言源代码文件。

扩展名（后缀）	说明
.i	已经预处理的 C 源代码文件。
.ii	已经预处理的 C++源代码文件。
.S	已经预处理的汇编语言源代码文件。
.h	程序所包含的头文件。
.o	编译后的目标文件。
.so	动态链接库，它是一种特殊的目标文件。
.a	静态链接库。
.out	可执行文件，但可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。如果没有给出可执行文件的名称，GCC 将生成一个名为 a.out 的文件。

3.2.2 make 工作流程

使用 make 由源代码文件生成可执行文件，需要经过如下步骤。

1. make 命令会读入 Makefile 文件，包括当前目录下命名为"GNUmakefile"、"makefile"、"Makefile"的文件、被 include 的 makefile 文件、参数-f、--file、--makefile 指定的规则文件。
2. 初始化变量。
3. 推导隐含规则，分析依赖关系，并创建依赖关系链。
4. 根据依赖关系链，决定哪些目标需要重新生成。
5. 执行生成命令，最终输出终极文件。

3.2.3 make 选项

make 命令格式为：**make** [option]... [target]...

其中：

option: 参数选项。

ftarget: Makefile 中指定的目标。

常用 make 的 *option* 取值如表 3-2 所示。

表3-2 常用的 make 选项

<i>options</i> 取值	说明
-------------------	----

<i>options</i> 取值	说明
<code>-C dir, --directory=dir</code>	指定 <code>make</code> 在开始运行后的工作目录为 <code>dir</code> 。 当存在多个 <code>-C</code> 选项的时候， <code>make</code> 的最终工作目录是第一个目录的相对路径。
<code>-d</code>	<code>make</code> 在执行的过程中打印出所有的调试信息。使用 <code>-d</code> 选项可以显示 <code>make</code> 构造依赖关系链、重建目标过程中的所有信息。
<code>-e, --enveronment-overrides</code>	使用环境变量定义覆盖 Makefile 中的同名变量定义。
<code>-f file, --file=file, --makefile=file</code>	指定 <code>file</code> 文件为 <code>make</code> 执行的 Makefile 文件。
<code>-p, --help</code>	打印帮助信息。
<code>-i, --ignore-errors</code>	执行过程中忽略规则命令执行的错误。
<code>-k, --keep-going</code>	执行命令错误时不终止 <code>make</code> 的执行， <code>make</code> 尽最大可能执行所有的命令，直至出现知名的错误才终止。
<code>-n, --just-print, --dry-run</code>	按实际运行时的执行顺序模拟执行命令(包括用 <code>@</code> 开头的命令)，没有实际执行效果，仅仅用于显示执行过程。
<code>-o file, --old-file=file, --assume-old=file</code>	指定 <code>file</code> 文件不需要重建，即使它的依赖已经过期，同时不重建此依赖文件的任何目标。
<code>-p, --print-date-base</code>	命令执行之前，打印出 <code>make</code> 读取的 Makefile 的所有数据，同时打印出 <code>make</code> 的版本信息。如果只需要打印这些数据信息，可以使用 “ <code>make -qp</code> ” 命令，查看 <code>make</code> 执行之前预设的规则和变量，可使用命令 “ <code>make -p -f /dev/null</code> ”。
<code>-r, --no-builtin-rules</code>	忽略内嵌的隐含规则的使用，同时忽略所有后缀规则的隐含后缀列表。
<code>-R, --no-builtin-variables</code>	忽略内嵌的隐含变量。
<code>-s, --silent, --quiet</code>	取消命令执行过程中的打印。
<code>-S, --no-keep-going, --stop</code>	取消 “ <code>-k</code> ” 的选项在递归的 <code>make</code> 过程中子 <code>make</code> 通过 “ <code>MAKEFLAGS</code> ” 变量继承了上层的命令行选项那个。我们可以在子 <code>make</code> 中使用 “ <code>-S</code> ” 选项取消上层传递的 “ <code>-k</code> ” 选项，或者取消系统环境变量

<i>options</i> 取值	说明
	量 "MAKEFLAGS" 中 "-k"选项。
-t, --touch	更新所有的目标文件的时间戳到当前系统时间。防止 make 对所有过时目标文件的重建。
-v, version	查看 make 的版本信息。

3.3 Makefile

make 是通过 Makefile 文件获取如何编译、链接和安装、清理的方法，从而实现将源代码文件生成可执行文件和其他相关文件的工具。因此，Makefile 中描述了整个工程的编译和链接等规则，其中包含了哪些文件需要编译，哪些文件不需要编译，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重建等等。Makefile 文件让工程编译实现了自动化，不需要每次都手动输入一堆源文件和参数。

本章简单介绍 Makefile 文件的结构和主要内容，更多 Makefile 的内容请通过 **info make** 命令查询

Makefile 结构

Makefile 文件结构如下所示：

```
targets:prerequisites
```

```
command
```

或者是：

```
targets:prerequisites;command
```

```
command
```

其中：

- *targets*: 目标，可以是目标文件、可执行文件或标签。
- *prerequisites*: 依赖文件，生成 *targets* 需要的文件或者是目标。可以是多个，也可以是没有。
- *command*: make 需要执行的命令（任意的 shell 命令）。可以有多条命令，每一条命令占一行。
- 目标和依赖文件之间要使用“:”分隔，命令的开始一定要按“Tab”。

Makefile 文件结构表明了输出的目标，输出目标的依赖对象和生成目标需要执行的命令。

Makefile 主要内容

一个 Makefile 文件主要由以下内容组成。

- 显式规则
明确写出来的依赖关系，如要生成的文件，文件的依赖文件，生成的命令。
- 隐含规则
由 `make` 自动推导的规则，`make` 命令支持自动推导功能。
- 变量的定义
- 文件指示
文件指示包括三部分：
 - `include` 其他 Makefile，如 `include xx.md`。
 - 选择执行，如 `#ifdef`。
 - 定义多行命令，如 `define...endif`。(define ... endif)
- 注释
以 “#” 开头。

3.4 示例

3.4.1 使用 Makefile 实现编译的示例

步骤 1 `cd` 到代码目录，此处以用户 “/home/code” 进行举例。

```
cd /home/code
```

步骤 2 创建 1 个头文件 `hello.h` 和 2 个函数 `hello.c`、`main.c`。

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

`hello.h` 代码内容示例：

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

`hello.c` 代码内容示例：

```
#include "hello.h"
void hello()
{
    int i=1;
    while(i<5)
    {
        printf("The %dth say hello.\n", i);
        i++;
    }
}
```

`main.c` 代码内容示例：

```
#include "hello.h"
#include <stdio.h>
int main()
```

```
{  
    hello();  
    return 0;  
}
```

步骤 3 创建 Makefile 文件。

```
vi Makefile
```

Makefile 文件内容示例:

```
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
```

步骤 4 执行 make 命令。

```
make
```

命令执行后，会打印 Makefile 中执行的命令。如果不需要打印该信息，可以在执行 make 命令是加上参数-s。

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

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

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

步骤 5 执行 ./main 目标。

```
./main
```

命令执行后，打印如下信息:

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

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

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

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

```
----结束
```

4 使用 JDK 编译

- 4.1 简介
- 4.2 基本规则
- 4.3 类库
- 4.4 示例

4.1 简介

JDK (Java Development Kit) 是 Java 开发者进行 Java 开发所必须的软件包，包含 JRE (Java Runtime Environment) 和编译、调测工具。openEuler 在 OpenJDK 的基础上进行了 GC 优化、并发稳定性增强、安全性增强等修改，提高了 Java 应用程序在 ARM 上的性能和稳定性。

4.2 基本规则

4.2.1 文件类型及工具

对于任何给定的输入文件，文件类型决定采用何种工具进行处理。JDK 常用的文件类型如表 4-1 所示，JDK 常用的工具如表 4-2 所示。

表4-1 JDK 常用的文件类型

扩展名 (后缀)	说明
.java	java 语言源代码文件。
.class	java 的字节码文件，是一种和任何具体机器环境及操作系统环境无关的中间代码。它是一种二进制文件，是 Java 源文件由 Java 编译器编译后生成的目标代码文件。

扩展名（后缀）	说明
.jar	java 的 jar 压缩文件。

表4-2 JDK 常用的工具

工具名称	说明
java	Java 运行工具，用于运行.class 字节码文件或.jar 文件。
javac	Java 编程语言的编译器，将.java 的源代码文件编译成.class 的字节码文件。
jar	创建和管理 Jar 文件。

4.2.2 java 程序生成流程

通过 JDK 将 java 源代码文件生成并运行 Java 程序，需要经过编译和运行。

1. 编译：是指使用 Java 编译器（javac）将 java 源代码文件（.java 文件）编译为.class 的字节码文件。
2. 运行：是指在 Java 虚拟机上执行字节码文件。

4.2.3 JDK 常用工具选项

javac 编译选项

javac 编译的命令格式为：**javac** [*options*] [*sourcefiles*] [*classes*] [*@argfiles*]

其中：

options：命令选项。

sourcefiles：一个或多个需要编译的源文件。

classes：一个或多个要为注释处理的类。

@argfiles：一个或多个列出选项和源文件的文件。这些文件中不允许有-J 选项。

javac 是 java 编译器，其 *options* 参数取值很多，但有些大部分并不常用，常用的 *options* 取值如表 4-3 所示。

表4-3 javac 常用的编译选项

<i>options</i> 取值	说明	示例
-d <i>path</i>	指定存放生成的类文件的路径。	#使用-d 选项将所有类文件输出到 bin 路径下

<i>options</i> 取值	说明	示例
	默认情况下，编译生成的类文件与源文件在同一路径下。使用-d选项可以将类文件输出到指定路径。	<code>javac /src/*.java -d /bin</code>
<code>-s path</code>	指定存放生成的源文件的路径。	-
<code>-cp path</code> 或 <code>-classpath path</code>	搜索编译所需的 class 文件，指出编译所用到的 class 文件的位置。	#在 Demo 中要调用 GetStringDemo 类中的 getLine()方法，而 GetStringDemo 类编译后的文件，即.class 文件在 bin 目录下。 <code>javac -cp bin Demo.java -d bin</code>
<code>-verbose</code>	输出关于编译器正在执行的操作的消息，如加载的类信息和编译的源文件信息。	#输出关于编译器正在执行的操作的消息。 <code>javac -verbose -cp bin Demo.java</code>
<code>-source sourceversion</code>	指定查找输入源文件的位置。	-
<code>-sourcepath path</code>	用于搜索编译所需的源文件（即 java 文件），指定要搜索的源文件的位置，如 jar、zip 或其他包含 java 文件的目录。	-
<code>-target targetversion</code>	生成特定 JVM 版本的类文件。取值为 1.1, 1.2, 1.3, 1.4, 1.5（或 5），1.6（或 6），1.7（或 7），1.8（或 8）。 <i>targetversion</i> 的默认取值与 <code>-source</code> 选项的 <i>sourceversion</i> 有关。 <i>sourceversion</i> 取值： <ul style="list-style-type: none"> • 1.2, <i>targetversion</i> 为 1.4; • 1.3, <i>targetversion</i> 为 1.4; • 1.5、1.6、1.7、未指定, <i>targetversion</i> 为 1.8。 • 其他值, <i>targetversion</i> 	-

<i>options</i> 取值	说明	示例
	与 <i>sourceversion</i> 取值相同。	

java 运行选项

java 运行的格式为:

运行类文件: **java** [*options*] *classname* [*args*]

运行 jar 文件: **java** [*options*] -jar *filename* [*args*]

其中:

options: 命令选项, 选项之间用空格分隔。

classname: 运行的.class 文件名。

filename: 运行的.jar 文件名。

args: 传递给 main()函数的参数, 参数之间用空格分隔。

java 是运行 java 应用程序的工具, 其 *options* 参数取值很多, 但有些大部分并不常用, 常用的 *options* 取值如表 4-4 所示。

表4-4 java 常用的运行选项

<i>options</i> 取值	说明	示例
-cp <i>path</i> 或 -classpath <i>path</i>	指定要运行的文件所在的位置以及需要用到的类路径, 包括 jar、zip 和 class 文件目录。 当路径有多个是, 使用“.”分隔。	-
-verbose	输出关于编译器正在执行的操作的消息, 如加载的类信息和编译的源文件信息。	#输出关于编译器正在执行的操作的消息。 java -verbose -cp bin Demo.java

jar 打包选项

jar 的命令格式为: **jar** {c|t|x|u}[vfm0M] [*jarfile*] [*manifest*] [-C *dir*] *file...*

jar 命令参数说明如表 4-5 所示。

表4-5 jar 命令参数说明

参数	说明	示例
c	创建 jar 文件包。	#把当前目录的 hello.class 文件打包到 Hello.jar, 且不显示打包的过程。如果 Hello.jar 文件还不存在, 就创建它, 否则首先清空它。 jar cf Hello.jar hello.class
t	列出 jar 文件包的内容列表。	#列出 Hello.jar 包含的文件清单。 jar tf Hello.jar
x	展开 jar 文件包的指定文件或者所有文件。	#解压 Hello.jar 到当前目录, 不显示任何信息。 jar xf Hello.jar
u	更新已存在的 jar 文件包, 如添加文件到 jar 文件包中。	-
v	生成详细报告并打印到标准输出。	#把当前目录的 hello.class 文件打包到 Hello.jar, 并显示打包的过程。如果 Hello.jar 文件还不存在, 就创建它, 否则首先清空它。 jar cvf Hello.jar hello.class
f	指定 jar 文件名, 通常这个参数是必须的。	-
m	指定需要包含的 manifest 清单文件。	-
0	只存储, 不压缩, 这样产生的 jar 文件包会比不用该参数产生的体积大, 但速度更快。	-
M	不产生所有项的 manifest 清单文件, 此参数会忽略 m 参数	#把当前目录的 hello.class 文件打包到 Hello.jar, 并显示打包的过程。如果 Hello.jar 文件还不存在, 就创建它, 否则首先清空它。但在创建 Hello.jar 时不产生 manifest 文件。 jar cvfM Hello.jar

参数	说明	示例
		hello.class
<i>jarfile</i>	.jar 文件包，它是 f 参数的附属参数。	-
<i>manifest</i>	.mf 的 manifest 清单文件，它是 m 参数的附属参数	-
<i>-C dir</i>	转到指定 <i>dir</i> 下执行 jar 命令，只能配合参数 c、t 使用。	-
<i>file</i>	指定文件/路径列表，文件或路径下的所有文件（包括递归路径下的）都会被打入 jar 文件包中，或解压 jar 文件到路径下。	#把当前目录的所有 class 文件打包到 Hello.jar，并显示打包的过程。如果 Hello.jar 文件还不存在，就创建它，否则首先清空它。 jar cvf Hello.jar *.class

4.3 类库

java 类库是以包的形式实现的，包是类和接口的集合。java 编译器为每个类生成一个字节码文件，且文件名与类名相同，因此同名的类之间就有可能发生冲突。java 语言中，把一组类和接口封装在一个包内，包可以有效地管理类名空间，位于不同包中的类即使同名也不会冲突，从而解决了同名类之间可能发生的冲突问题，为管理大量的类和接口提供了方便，也有利于类和接口的安全。

除 java 提供的许多包外，开发者也可以自定义包，把自己编写的类和接口等组成程序包的形式，以便后续使用。

自定义包需要先声明包，然后在使用包。

包的声明

包的声明格式为：`package pkg1[.pkg2[.pkg3...]];`

为了声明一个包，首先必须建立一个相应的目录结构，子目录与包名一致，然后在需要放入该包的类文件开头声明包，表示该文件的全部类都属于这个包。包声明中的“.”指明了目录的层次。如果源程序文件中没有 `package` 语句，则指定为无名包。无名包没有路径，一般情况下，java 仍然会把源文件中的类存储在当前工作目录（即存放 java 源文件的目录）下。

包声明语句必须被加到源程序文件的起始部分，而且前面不能有注释和空格。如果在不同源程序文件中使用相同的包声明语句，就可以将不同源程序文件中的类都包含在相同的包中。

包的引用

在 Java 中，为了能使用 java 提供的包中的公用类，或者使用自定义的包中的类，有两种方法。

- 在要引用的类名前带上包名。

如：`name.A obj=new name.A ();`

其中，`name` 为包名，`A` 为类名，`obj` 为对象。表示程序中用 `name` 包中的 `A` 类定义一个对象 `obj`。

示例：新建一个 `example` 包中 `Test` 类的 `test` 对象。

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

- 在文件开头使用 `import` 来导入包中的类。

`import` 语句的格式为：`import pkg1[.pkg2[.pkg3...]].(classname | *);`

其中，`pkg1[.pkg2[.pkg3...]]` 表明包的层次，`classname` 为所要导入的类。如果要从一个包中导入多个类，则可以使用通配符“*”来替代。

示例：导入 `example` 包中的 `Test` 类。

```
import example.Test;
```

示例：将 `example` 整个包导入。

```
import example.*;
```

4.4 示例

4.4.1 编译不带包的 java 程序示例

步骤 1 cd 到代码目录，此处以用户“/home/code”进行举例。如下所示：

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

步骤 2 编写 Hello World 程序，保存为 `HelloWorld.java`，此处以编译 Hello World 程序进行举例说明。示例如下：

```
# vi HelloWorld.java
```

代码内容示例：

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

步骤 3 在代码目录，执行编译，使用命令：

```
# javac HelloWorld.java
```

编译执行未报错，表明执行通过。

步骤 4 编译完成后，会生成 `HelloWorld.class` 文件，通过 `java` 命令可执行查看结果，示例如下：

```
# java HelloWorld  
Hello World
```

----结束

4.4.2 编译带包的 java 程序示例

步骤 1 cd 到代码目录，此处以用户“/home/code”进行举例。并在该目录下创建“/home/code/Test/my/example”、“/home/code/Hello/world/developers”、“/home/code/Hi/openos/openeuler”子目录，分别用于存放源文件。

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

步骤 2 cd 到/home/code/Test/my/example 目录，创建 Test.java。

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

Test.java 代码内容示例：

```
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();  
    }  
}
```

步骤 3 cd 到/home/code/Hello/world/developers 目录，创建 Hello.java。

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

Hello.java 代码内容示例：

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

步骤 4 cd 到/home/code/Hi/openos/openeuler 目录，创建 Hi.java。

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

Hi.java 代码内容示例：

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

步骤 5 cd 到/home/code，使用 javac 编译源文件。

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

执行完命令后，会在“/home/code/Test/my/example”、“/home/code/Hello/world/developers”、“/home/code/Hi/openos/openeuler”目录下分别生成 Test.class、Hello.class、Hi.class 文件。

步骤 6 cd 到/home/code，使用 java 运行 Test 程序。

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

执行结果如下所示：

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

----结束

5 构建 RPM 包

本章介绍通过本地或 OBS 构建 RPM 软件包的方法。详细的打包规则请参见《openEuler 1.0 打包规则》。

- 5.1 打包说明
- 5.2 本地构建
- 5.3 使用 OBS 构建

5.1 打包说明

原理介绍

RPM 打包的时候需要编译源码，需要把编译好的配置文件、二进制命令文件等放到合适的位置，还要根据需要对 RPM 的包进行测试，这些都需要先有一个“工作空间”。`rpmbuild` 命令使用一套标准化的“工作空间”：

```
# rpmddev-setuptree
```

`rpmddev-setuptree` 这个命令就是安装 `rpmddevtools` 带来的。可以看到运行了这个命令之后，在“/root”目录（非 root 用户为“/home/用户名”目录）下多了一个 `rpmbuild` 的文件夹，目录结构如下：

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

内容相关的说明如下：

目录	宏代码	名称	功能
~/rpmbuild/BUILD	%_builddir	构建目录	源码包被解压至此，并在该目录的子目录完成编译

目录	宏代码	名称	功能
~/rpmbuild/RPMS	\$_rpmdir	标准 RPM 包目录	生成/保存二进制 RPM 包
~/rpmbuild/SOURCE S	\$_sourcedir	源代码目录	保存源码包（如 .tar 包）和所有 patch 补丁
~/rpmbuild/SPECS	\$_specdir	Spec 文件目录	保存 RPM 包配置（.spec）文件
~/rpmbuild/SRPMS	\$_srcrpmdir	源代码 RPM 包目录	生成/保存源码 RPM 包 (SRPM)

SPECS 下是 RPM 包的配置文件，是 RPM 打包的“图纸”，这个文件会告诉 rpmbuild 命令如何去打包。“宏代码”这一列就可以在 SPEC 文件中用来代指所对应的目录，类似于编程语言中的宏或全局变量。

打包流程

打包的过程主要分为如下步骤：

1. 把源代码放到\$_sourcedir 中。
2. 进行编译，编译的过程是在\$_builddir 中完成的，一般情况下，源代码是压缩包格式，需要先进行解压。
3. 进行“安装”，类似于预先组装软件包，把软件包应该包含的内容（比如二进制文件、配置文件、man 文档等）复制到\$_buildrootdir 中，并按照实际安装后的目录结构组装，比如二进制命令可能会放在/usr/bin 下，那么就在\$_buildrootdir 下也按照同样的目录结构放置。
4. 做一些必要的配置，比如在实际安装前的准备，安装后的清理等等。这些都是通过配置在 SPEC 文件中来告诉 rpmbuild 命令。
5. 检查软件是否正常运行。
6. 生成的 RPM 包放置到\$_rpmdir，源码包放置到\$_srcrpmdir 下。

在 SPEC 文件中的，各个阶段说明如下：

阶段	读取的目录	写入的目录	具体动作
%prep	\$_sourcedir	\$_builddir	读取位于 \$_sourcedir 目录的源代码和 patch。之后，解压源代码至 \$_builddir 的子目录并应用所有 patch。
%build	\$_builddir	\$_builddir	编译位于 \$_builddir 构建目录下的文件。通过执行类似 ./configure && make 的命令实现。
%install	\$_builddir	\$_buildrootdir	读取位于 \$_builddir 构建目录下的文件并将其安装至 \$_buildrootdir 目录。这些文件就是用户安装 RPM 后，最终得到的文件。

阶段	读取的目录	写入的目录	具体动作
%check	%_builddir	%_builddir	检查软件是否正常运行。通过执行类似 <code>make test</code> 的命令实现。
bin	%_buildrootdir	%_rpmmdir	读取位于 %_buildrootdir 最终安装目录下的文件，以便最终在 %_rpmmdir 目录下创建 RPM 包。在该目录下，不同架构的 RPM 包会分别保存至不同子目录，noarch 目录保存适用于所有架构的 RPM 包。这些 RPM 文件就是用户最终安装的 RPM 包。
src	%_source_dir	%_srcrpmdir	创建源码 RPM 包（简称 SRPM，以 <code>src.rpm</code> 作为后缀名），并保存至 %_srcrpmdir 目录。SRPM 包通常用于审核和升级软件包。

打包选项

通过 `rpmbuild` 命令构建软件包。`rpmbuild` 构建软件包一般可以通过构建 SPEC 文件、tar 文件、source 文件实现。

`rpmbuild` 命令格式为：`rpmbuild [option...]`

常用的 `rpmbuild` 打包选项如表 5-1 所示。

表5-1 `rpmbuild` 打包选项

<i>option</i> 取值	说明
<code>-bp specfile</code>	从 <i>specfile</i> 文件的%prep 段开始构建（解开源码包并打补丁）。
<code>-bc specfile</code>	从 <i>specfile</i> 文件的%build 段开始构建。
<code>-bi specfile</code>	从 <i>specfile</i> 文件的%install 段开始构建。
<code>-bl specfile</code>	从 <i>specfile</i> 文件的%file 段开始检查。
<code>-ba specfile</code>	通过 <i>specfile</i> 文件构建源码包和二进制包。
<code>-bb specfile</code>	通过 <i>specfile</i> 文件构建二进制包。
<code>-bs specfile</code>	通过 <i>specfile</i> 文件构建源码包。
<code>-rp sourcefile</code>	从 <i>sourcefile</i> 文件的%prep 段开始构建（解开源码包并打补丁）。
<code>-rc sourcefile</code>	从 <i>sourcefile</i> 文件的%build 段开始构建。
<code>-ri sourcefile</code>	从 <i>sourcefile</i> 文件的%install 段开始构

<i>option</i> 取值	说明
	建。
<i>-rl sourcefile</i>	从 <i>sourcefile</i> 文件的%file 段开始检查。
<i>-ra sourcefile</i>	通过 <i>sourcefile</i> 文件构建源码包和二进制包。
<i>-rb sourcefile</i>	通过 <i>sourcefile</i> 文件构建二进制包。
<i>-rs sourcefile</i>	通过 <i>sourcefile</i> 文件构建源码包。
<i>-tp tarfile</i>	从 <i>tarfile</i> 文件的%prep 段开始构建（解开源码包并打补丁）。
<i>-tc tarfile</i>	从 <i>tarfile</i> 文件的%build 段开始构建。
<i>-ti tarfile</i>	从 <i>tarfile</i> 文件的%install 段开始构建。
<i>-ta tarfile</i>	通过 <i>tarfile</i> 文件构建源码包和二进制包。
<i>-tb tarfile</i>	通过 <i>tarfile</i> 文件构建二进制包。
<i>-ts tarfile</i>	通过 <i>tarfile</i> 文件构建源码包。
<i>--buildroot=DIRECTORY</i>	在构建时，使用 <i>DIRECTORY</i> 目录覆盖默认的/root 目录。
<i>--clean</i>	完成打包后清除 BUILD 目录下的文件。
<i>--nobuild</i>	不执行任何实际的构建步骤。可用于测试 spec 文件。
<i>--noclean</i>	不执行 spec 文件的"%clean"阶段(即使它确实存在)。
<i>--nocheck</i>	不执行 spec 文件的"%check"阶段(即使它确实存在)。
<i>--dbpath DIRECTORY</i>	使用 <i>DIRECTORY</i> 中的数据库，而不是默认的 /var/lib/rpm。
<i>--root DIRECTORY</i>	使 <i>DIRECTORY</i> 为最高级别的路径，默认为 "/" 为最高路径。
<i>--rebuild sourcefile</i>	将安装指定的源代码包 <i>sourcefile</i> ，然后进行准备、编译、安装。
<i>--recompile sourcefile</i>	在 <i>--recompile</i> 的基础上额外构建一个新的二进制包。在构建结束时，构建目录、源代码和 spec 文件都将被删除。将被删除(就好像用了 <i>--clean</i>)，
<i>-?, --help</i>	打印详细的帮助信息。

<i>option</i> 取值	说明
--version	打印详细的版本信息。

5.2 本地构建

本章通过一个简单的示例介绍如何在本地构建 RPM 软件包的方法。

5.2.1 搭建开发环境

前提条件

需要 root 权限，已设置 openEuler 的 repo 软件源。

操作步骤

用户可以直接使用 DNF 工具安装 rpmdevtools，其中包含 rpm-build 等命令以及相关依赖（例如 make、gdb）。使用如下命令：

```
# dnf install rpmdevtools*
```

5.2.2 创建 Hello World RPM 包

这里以 GNU “Hello World” 项目的打包过程作为示例，包含了典型的 FOSS（Free and Open Source Software）软件项目相关的最常用的外围组件，其中包括配置/编译/安装环境、文档、国际化等等。

5.2.2.1 下载源码

我们直接下载官方例子的源码，使用如下命令：

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

5.2.2.2 编辑 SPEC 文件

新建 spec 文件，参考命令如下：

```
# vi hello.spec
```

在文件中写入对应内容后保存文件。文件内容示例如下，请根据实际情况修改相应字段。

```
Name:      hello  
Version:   2.10  
Release:   1%{?dist}  
Summary:   The "Hello World" program from GNU  
Summary(zh_CN): GNU "Hello World" 程序  
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
"Hello World" 程序, 包含 FOSS 项目所需的所有部分, 包括配置, 构建, 国际化, 帮助文件等.

%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
```

- **Name** 标签是软件名, **Version** 标签是版本号, 而 **Release** 标签是发布编号。
- **Summary** 标签是简要说明, 英文的话第一个字母应大写, 以避免 rpmlint 工具 (打包检查工具) 警告。
- **License** 标签说明软件包的协议版本, 审查软件的 License 状态是打包者的职责, 这可以通过检查源码或 LICENSE 文件, 或与作者沟通来完成。
- **Group** 标签过去用于按照 /usr/share/doc/rpm-/GROUPS 分类软件包。目前该标记已丢弃, vim 的模板还有这一条, 删掉即可, 不过添加该标记也不会有任何影响。%changelog 标签应包含每个 Release 所做的更改日志, 尤其应包含上游的安

全/漏洞补丁的说明。`%changelog` 条目应包含版本字符串，以避免 `rpmlint` 工具警告。

- 多行的部分，如 `%changelog` 或 `%description` 由指令下一行开始，空行结束。
- 一些不需要的行 (如 `BuildRequires` 和 `Requires`) 可在行首使用 ‘#’ 注释。
- `%prep`、`%build`、`%install`、`%file` 暂时用默认的，未做任何修改。

5.2.2.3 构建 RPM 包

构建源码、二进制和包含调试信息的软件包，在 `spec` 文件所在目录执行如下命令：

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

执行成功后，查看结果，使用如下命令：

```
# 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 使用 OBS 构建

本章介绍通过网页和 `osc` 构建 RPM 软件包的方法。主要包括如下两类：

- 修改已有软件包：修改已有软件包源代码，然后将修改后的源代码构建成一个 RPM 软件包。
- 新增软件包：从无到有全新开发一个新的软件源文件，并将新开发的源文件构建成一个 RPM 软件包。

5.3.1 OBS 简介

OBS 是基于 openSUSE 发行版的通用编译框架，用于将源码包构建为 RPM 软件包或 Linux 镜像。OBS 采用自动化的分布式编译方式，支持多种 Linux 操作系统发行版（openEuler、SUSE、Debian 等）镜像和安装包的编译，且支持在多种架构平台（x86、ARM64 等）上编译。

OBS 由后端和前端组成，后端实现所有核心功能，前端提供了网页应用和 API，用于与后端进行交互。此外，OBS 还有一个 API 命令行客户端 `osc`，`osc` 是在一个单独的存储库中开发的。

OBS 使用工程组织软件包。基础的权限控制、相关的存仓库和构建目标（操作系统和架构）都可以在工程中定义。一个工程可以包含多个子工程，各个子工程可以独立配置，共同完成任务。

5.3.2 在线构建软件包

本章介绍通过 OBS 网页端在线构建 RPM 软件包的方法。

5.3.2.1 构建已有软件包

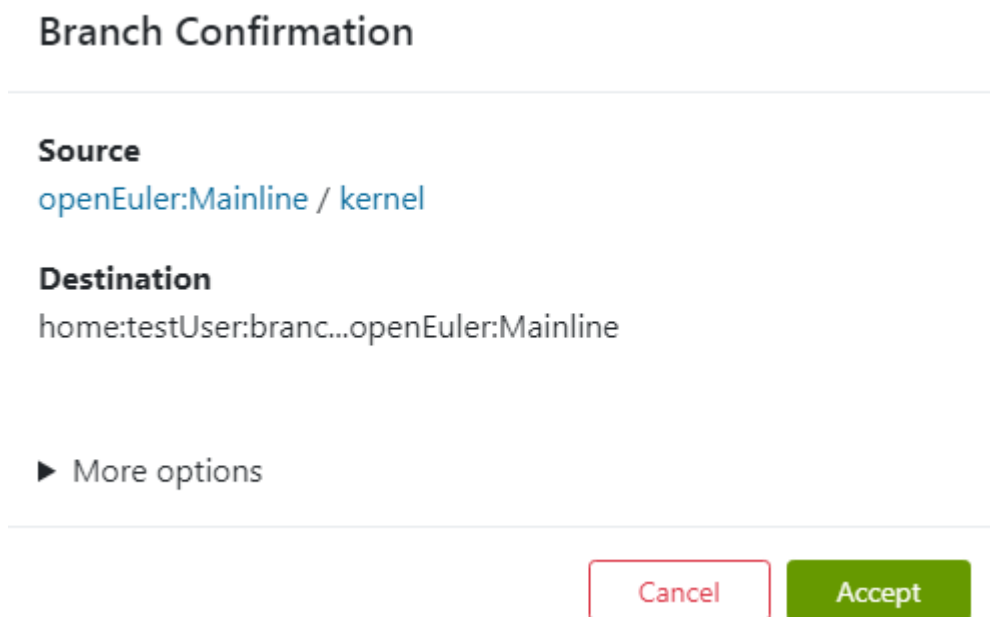
📖 说明

- 若为首次使用，请首先在 OBS 网页注册个人账号。
- 该方法需要拷贝修改后的代码，因此，请在执行下述操作前完成代码修改并提交到正确的代码路径。代码路径会在 `_service` 文件中指定。

使用 OBS 网页端，修改已有软件的源代码，并将修改后的源文件构建为 RPM 软件包的操作方法如下：

1. 登录 OBS 界面，地址为：<http://117.78.1.88/>。
2. 单击“**All Projects**”进入所有工程页面。
3. 单击需要修改的对应工程，进入该工程的详情页面，例如单击“**openEuler:Mainline**”。
4. 在工程详情页面的搜索框查找需要修改的软件包，然后单击该软件包包名，进入该软件包详情页。
5. 单击“**Branch package**”，在弹出的确认页面单击“**Accept**”确认创建子工程，如图 5-1 所示。

图5-1 Branch Confirmation 页面



6. 单击“`_service`”文件进入编辑页面，修改 `_service` 内容后单击“**Save**”保存该文件。`_service` 内容示例如下，其中 `userCodeURL`、`userCommitID` 分别为用户代码托管路径、用户代码提交版本号或分支。

```
<services>
  <service name="tar_scm_kernel">
    <param name="scm">git</param>
    <param name="url">userCodeURL</param>
```

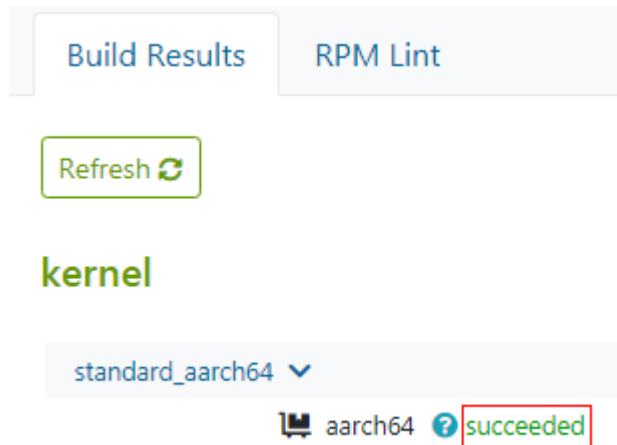
```
<param name="revision">userCommitID</param>
</service>
<service name="recompress">
  <param name="compression">bz2</param>
  <param name="file">*.tar</param>
</service>
</services>
```

说明

单击“Save”保存_service文件后，OBS服务会根据_service文件描述，从指定的url下载源码到OBS对应工程的软件目录下并替换原有文件，例如上述例子中“openEuler:Mainline”工程的kernel目录。

7. 文件拷贝并替换完成后，OBS会自动开始构建RPM软件包。等待构建完成，并查看右侧状态栏的构建状态。
 - succeeded: 构建成功。用户可以单击“succeeded”查看构建日志，如图5-2所示。

图5-2 succeeded 的页面



- failed: 构建失败。请单击“failed”查看错误日志进行问题定位后重新构建。
- unresolvable: 未进行构建，可能由于缺失依赖。
- disabled: 构建被手动关闭或正在排队构建。
- excluded: 构建被禁止，可能由于缺少spec文件或者spec文件中禁止了目标架构的编译。

5.3.2.2 新增软件包

使用OBS网页端，新增一个全新软件包的操作方法如下：

1. 登录OBS界面。
2. 根据新增软件包的依赖情况，选择合适的工程，即单击“All Projects”选择对应工程，例如“openEuler:Mainline”。
3. 单击工程下任一软件包，进入该软件包的详情页面。
4. 单击“Branch package”，在弹出的确认页面单击“Accept”确认创建子工程。

5. 单击“Delete package”，删除新创建子工程中的软件包，如图 5-3 所示。

图5-3 删除子工程中软件包



📖 说明

通过已有软件创建新工程是为了继承环境等依赖，而不需要实际的文件，所以这里需要把这些文件删除。

6. 单击“Create Package”，在弹出的页面输入软件包名称、标题和软件包描述，然后单击“Create”创建软件包，分别如图 5-4、图 5-5 所示。

图5-4 Create Package 页面

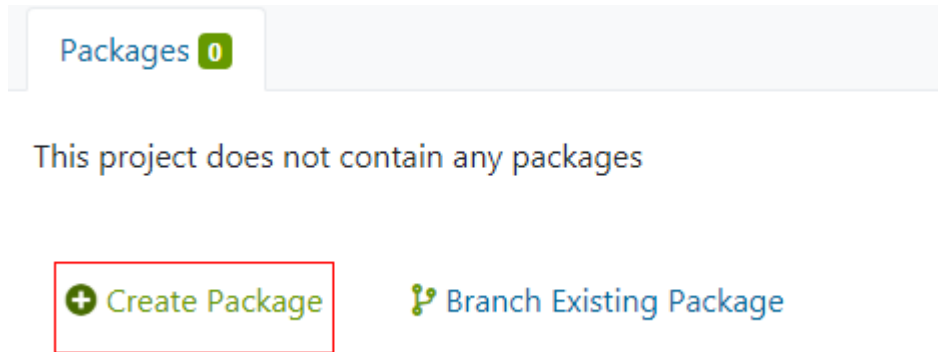


图5-5 创建软件包信息填写页面

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

Name:

Title:

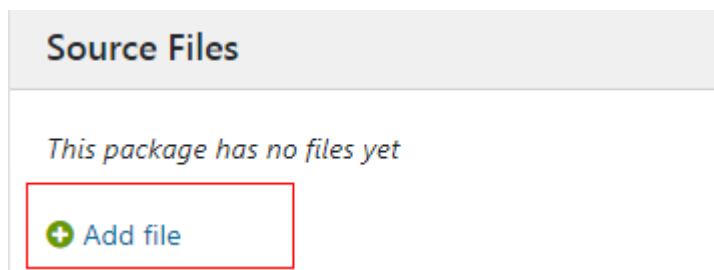
Description:

Deny access to source of package

Disable build results publishing

7. 在页面中单击“Add file”上传 spec 文件和需要编译的文件（在 spec 文件中指定），如图 5-6 所示。

图5-6 Add file 页面



8. 文件上传成功后，OBS 会自动开始构建 RPM 软件包。等待构建完成，并查看右侧状态栏的构建状态。
 - **succeed:** 构建成功。用户可以单击“succeeded”查看构建日志。
 - **failed:** 构建失败。请单击“failed”查看错误日志进行问题定位后重新构建。
 - **unresolvable:** 未进行构建，可能由于缺失依赖。
 - **disabled:** 构建被手动关闭或正在排队构建。

- **excluded:** 构建被禁止，可能由于缺少 spec 文件或者 spec 文件中禁止了目标架构的编译。

5.3.2.3 获取软件包

RPM 软件包构建完成后，通过网页端获取对应 RPM 软件包的方法如下：



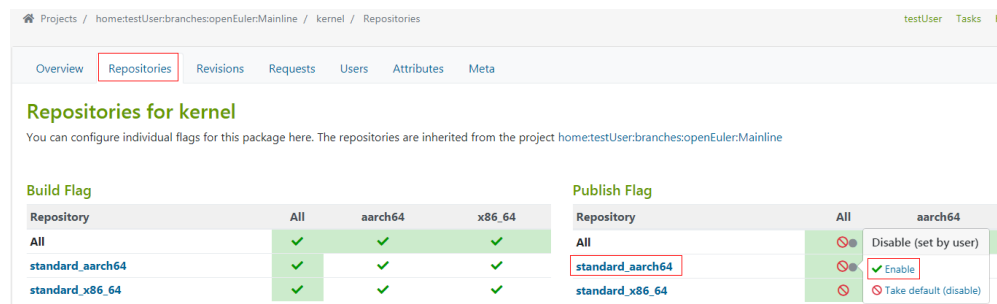
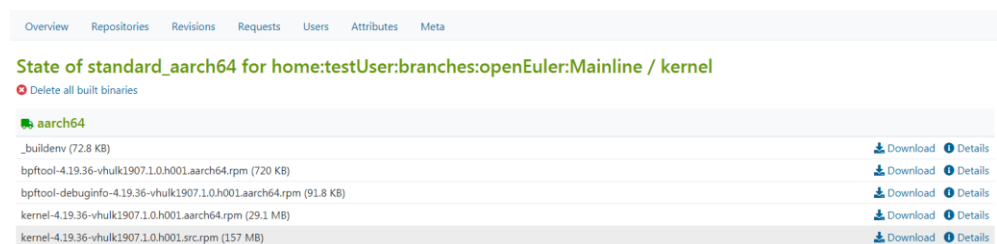
1. 登录 OBS 界面。
2. 单击“**All Projects**”找到所需软件包的对应工程，例如“**openEuler:Mainline**”。
3. 在工程下单击所需软件包的包名，进入该软件包详情页。例如上述例子中的 **kernel** 页面。
4. 选择 **Repositories** 页签进入软件包的软件仓库管理页面，在 **Publish Flag** 中通过单击选择“**Enable**”开启（状态由  变为 ）对应的 RPM 软件包下载功能，如图 5-7 所示。

图5-7 Repositories 页面



5. 单击 **Repository** 列的构建工程名称，进入 RPM 软件包下载页面，单击 RPM 软件包右侧的“**Download**”即可下载对应 RPM 软件包，如图 5-8 所示。

图5-8 RPM 软件包下载页面



5.3.3 使用 osc 构建软件包

本章介绍使用 OBS 的命令行工具 **osc** 创建工程并构建 RPM 软件包的方法。

5.3.3.1 安装并配置 osc

前提条件

需要 root 权限，已设置 openEuler 的 repo 软件源。

操作步骤

步骤 1 使用 root 用户安装 osc 命令行工具及依赖。

```
# dnf install osc build
```

📖 说明

编译 RPM 软件包的过程中会依赖 build。

步骤 2 配置 osc。

1. 打开 ~/.osrcrc，命令如下：

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

2. 在 ~/.osrcrc 中添加 user 和 pass 字段，如下所示，它们的取值 *userName* 和 *passWord* 分别是用户在 OBS 网页 (<https://build.openeuler.org/>) 上已经注册的账号和密码。

```
[general]
apiurl = https://build.openeuler.org
no_verify = 1
[https://build.openeuler.org]
user=userName
pass=passWord
```

3. 如果域名 build.openeuler.org 无法解析，则可以在 /etc/hosts 文件中手动添加如下一行。其中， *ip-address* 是 obs 的 ip 地址，为 http://117.78.1.88/。

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

----结束

5.3.3.2 构建已有软件包

创建工程

1. 通过拷贝已有工程，创建属于用户自己的子工程。例如将 openEuler:Mainline 工程下的 zlib 软件包到新分支，参考命令如下：

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

回显如下所示，说明在用户 testUser 下创建了新的分支工程 home:testUser:branches:openEuler:Mainline。

```
A working copy of the branched package can be checked out with:
osc co home:testUser:branches:openEuler:Mainline/zlib
```

2. 将需要修改软件包的相关配置文件（例如 *_service*）下载到本地当前路径。其中 *testUser* 为 ~/.osrcrc 配置文件中配置的账户名称，请根据实际情况修改。

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

回显如下所示：

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

3. 进入本地子工程目录，并将软件包远程代码同步到本地。

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

回显如下所示：

```
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
```

构建 RPM 包

4. 重命名源文件，然后将重命名后的源文件添加到 OBS 暂存中。

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

5. 修改源代码和 spec 文件，并将对应软件包的所有修改同步到 OBS 服务器。参考命令如下，-m 参数后的信息为提交记录。

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

6. 获取当前工程的仓库名称和架构，参考命令如下：

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

7. 修改提交成功后，OBS 会自动开始编译软件包。可以通过如下命令，查看对应仓库的编译日志，其中 *standard_aarch64*、*aarch64* 分别为查询所得仓库名称和架构。

```
# osc buildlog standard_aarch64 aarch64
```

📖 说明

用户也可以通过网页端打开自己创建的对应用工程，查看构建日志。

5.3.3.3 新增软件包

使用 OBS 的 osc 工具新增一个全新软件包的操作方法如下：

创建工程

1. 根据新增软件包的依赖情况，基于合适的工程，创建属于用户自己的个人工程。例如基于 *openEuler:Mainline* 工程的 *zlib* 创建工程的参考命令如下，*zlib* 为工程下的任一软件包。

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

2. 删除创建工程时新增的无用软件包。例如删除 *zlib* 软件包的参考命令如下：

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

3. 在个人工程下创建新增的软件包。例如新增软件包 `my-first-obs-package` 命令如下：

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

构建 RPM 包

4. 添加准备的源文件和 `spec` 文件到软件包目录。
5. 修改源代码和 `spec` 文件，并将对应软件包的所有文件上传到 OBS 服务器。参考命令如下，`-m` 参数后的信息为提交记录。

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

6. 获取当前工程的仓库名称和架构，参考命令如下：

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

7. 修改提交成功后，OBS 会自动开始编译软件包。可以通过如下命令，查看对应仓库的编译日志，其中 `standard_aarch64`、`aarch64` 分别为查询所得仓库名称和架构。

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

📖 说明

用户也可以通过网页端打开自己创建的对应用工程，查看构建日志。

5.3.3.4 获取软件包

RPM 软件包构建完成后，使用 `osc` 获取对应 RPM 软件包的命令如下：

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

命令中的各参数含义如下，请用户根据实际情况修改：

- `home:testUser:branches:openEuler:Mainline`：软件包所在工程名称
- `my-first-obs-package`：软件包名称
- `standard_aarch64`：仓库名称
- `aarch64`：仓库架构名称

📖 说明

使用 `osc` 构建的软件包也可以在网页端获取，获取方式请参见网页端获取软件包相关内容。