


在Ubuntu中编译和调试OpenJDK

OpenJDK Ubuntu CLion

 2020年 07月19日

构建编译环境

安装GCC编译器：

```
sudo apt install build-essential
```

安装OpenJDK依赖库：

工具	库名称	安装命令
FreeType	The FreeType Project	<code>sudo apt install libfreetype6-dev</code>
CUPS	Common UNIX Printing System	<code>sudo apt install libcups2-dev</code>
X11	X Window System	<code>sudo apt install libx11-dev libxext-dev libxrender-dev libxrandr-dev libxtst-dev libxt-dev</code>
ALSA	Advanced Linux Sound Architecture	<code>sudo apt install libasound2-dev</code>
libffi	Portable Foreign Function Interface	<code>sudo apt install libffi-dev</code>
Autoconf	Extensible Package of M4 Macros	<code>sudo apt install autoconf</code>
zip/unzip	unzip	<code>sudo apt install zip unzip</code>
fontconfig	fontconfig	<code>sudo apt install libfontconfig1-dev</code>

假设要编译大版本号为N的JDK，我们还要安装一个大版本号**至少为N-1**的、已经编译好的JDK作为“Bootstrap JDK”：

```
sudo apt install openjdk-11-jdk
```

获取源码

可以直接访问准备下载的JDK版本的仓库页面（譬如本例中OpenJDK 11的页面为<https://hg.openjdk.java.net/jdk-updates/jdk11u/>），然后点击左边菜单中的“Browse”，再点击左边的“zip”链接即可下载当前版本打包好的源码，到本地直接解压即可。

也可以从Github的镜像Repositories中获取 (<https://github.com/openjdk>)，进入所需版本的JDK的页面，点击Clone按钮下的**Download ZIP**按钮下载打包好的源码，到本地直接解压即可。

进行编译

首先进入解压后的源代码目录，本例解压到的目录为 `~/openjdk/`：

```
cd ~/openjdk
```

要想带着调试、定制化的目的去编译，就要使用OpenJDK提供的编译参数，可以使用 `bash configure --help` 查看。本例要编译SlowDebug版、仅含Server模式的HotSpot虚拟机，同时我们还可以禁止压缩生成的调试符号信息，方便gdb调试获取当前正在执行的源代码和行号等调试信息。对应命令如下：

```
bash configure --with-debug-level=slowdebug --with-jvm-variants=server --disable-zip-debug-info
```

对于版本较低的OpenJDK，编译过程中可能会出现源码**deprecated**的错误，这是因为 ≥ 2.24 版本的glibc中，`readdir_r`等方法被标记为deprecated。若读者也出现了该问题，请在 `configure` 命令加上 `--disable-warnings-as-errors` 参数，如下：

```
bash configure --with-debug-level=slowdebug --with-jvm-variants=server --disable-zip-debug-info --disable-warn
```

此外，若要重新编译，请先执行 `make dist-clean`

执行 `make` 命令进行编译：

```
make
```

生成的JDK在 `build/配置名称/jdk` 中，测试一下，如：

```
cd build/linux-x86_64-normal-server-slowdebug/jdk/bin
./java -version
```

生成Compilation Database

CLion 可以通过 `Compilation Database` 来导入项目。在 OpenJDK 11u 及之后版本中，OpenJDK 官方提供了对于 IDE 的支持，可以使用 `make compile-commands` 命令生成 `Compilation Database`：

```
make compile-commands
```

对于版本较低的OpenJDK，可以使用一些工具来生成 `Compilation Database`，比如：

- [Bear](#)
- [scan-build](#)
- [compiled](#)

然后检查一下 `build/配置名称/` 下是否生成了 `compile_commands.json`。

```
cd build/linux-x86_64-normal-server-slowdebug
ls -l
```

导入项目至CLion

优化CLion索引速度

提高 Inotify 监视文件句柄上限，以优化CLion索引速度：

1. 在 `/etc/sysctl.conf` 中或 `/etc/sysctl.d/` 目录下新建一个 `*.conf` 文件，添加以下内容：

```
fs.inotify.max_user_watches = 524288
```

2. 应用更改：

```
sudo sysctl -p --system
```

3. 重新启动CLion

导入项目

打开 CLion ，选择 `Open Or Import` ，选择上文生成的 `build/配置名称/compile_commands.json` 文件，弹出框选择 `Open as Project` ，等待文件索引完成。

接着，修改项目的根目录，通过 `Tools -> Compilation Database -> Change Project Root` 功能，选中你的源码目录。

为了减少CLion索引文件数，提高CLion效率，建议将非必要的文件夹排除： `Mark Directory as -> Excluded` 。大部分情况下，我们只需要索引以下文件夹下的源码：

- `src/hotspot`
- `src/java.base`

配置调试选项

创建自定义Build Target

点击 `File` 菜单栏， `Settings -> Build, Execution, Deployment -> Custom Build Targets` ，点击 `+` 新建一个 `Target` ，配置如下：

- `Name` ： `Target` 的名字，之后在创建 `Run/Debug` 配置的时候会看到这个名字
- 点击 `Build` 或者 `clean` 右边的三点，弹出框中点击 `+` 新建两个 `External Tool` 配置如下：

```
# 第一个配置如下，用来指定构建指令
# Program 和 Arguments 共同构成了所要执行的命令 "make"
Name: make
Program: make
Arguments:
Working directory: {项目的根目录}

# 第二个配置如下，用来清理构建输出
# Program 和 Arguments 共同构成了所要执行的命令 "make clean"
Name: make clean
Program: make
Arguments: clean
Working directory: {项目的根目录}
```

- `ToolChain` 选择 `Default` ； `Build` 选择 `make` （上面创建的第一个 `External Tool` ）； `clean` 选择 `make clean` （上面创建的第二个 `External Tool` ）

创建自定义的Run/Debug configuration

点击 `Run` 菜单栏， `Edit Configurations` ，点击 `+` ，选择 `Custom Build Application` ，配置如下：

```
# Executable 和 Program arguments 可以根据需要调试的信息自行选择

# Name: Configure 的名称
Name: OpenJDK
# Target: 选择上一步创建的 “Custom Build Target”
Target: { 上一步创建的 “Custom Build Target” }
# Executable: 程序执行入口, 也就是需要调试的程序
Executable: 这里我们调试`java`, 选择`{source_root}/build/{build_name}/jdk/bin/java`。
# Program arguments: 与 “Executable” 配合使用, 指定其参数
Program arguments: 这里我们选择`-version`, 简单打印一下`java`版本。
# Before launch: 这个下面的Build可去可不去, 去掉就不会每次执行都去Build, 节省时间, 但其实OpenJDK增量编译的方式, 每次Build都很快,
```

配置GDB

由于HotSpot JVM内部使用了SEGV等信号来实现一些功能（如 `NullPointerException` 、 `safepoints` 等），所以调试过程中，GDB 可能会误报 `Signal: SIGSEGV (Segmentation fault)`。解决办法是，在用户目录下创建 `.gdbinit`，让 GDB 捕获 SEGV 等信号：

```
vi ~/.gdbinit
```

将以下内容追加到文件中并保存：

```
handle SIGSEGV nostop noprint pass
```

开始调试

使用CLion调试C++层面的代码

完成以上配置之后，一个可修改、编译、调试的HotSpot工程就完全建立起来了。HotSpot虚拟机启动器的执行入口是 `${source_root}/src/java.base/share/native/libjli/java.c` 的 `JavaMain()` 方法，读者可以设置断点后点击 `Debug` 可开始调试。

使用GDB调试汇编层面的代码

这里提供两个方法，一个是使用 `-XX:StopInterpreterAt=<n>` 虚拟机参数来实现中断，缺点是需要找到你所感兴趣的字节码在程序中的序号；第二个方法是直接去寻找记录生成的机器指令的入口(EntryPoint)的表，即

```
Interpreter::_normal_table
```

，在对应的字节码入口地址打断点，但是这需要读者对模板解释器有一定了解。

使用虚拟机参数进行中断

对于汇编级别的调试，我们可以手动使用GDB进行调试：

```
gdb build/linux-x86_64-normal-server-slowdebug/jdk/bin/java
```

由于目前HotSpot在主流的操作系统上，都采用模板解释器来执行字节码，它与即时编译器一样，最终执行的汇编代码都是运行期间产生的，无法直接设置断点，所以HotSpot增加了一些参数来方便开发人员调试解释器。

我们可以先使用参数 `-XX:+TraceBytecodes`，打印并找出你所感兴趣的字节码位置，中途可以使用 `ctrl + c` 退出：

```
set args -XX:+TraceBytecodes
run
```

然后，再使用参数 `-XX:StopInterpreterAt=<n>`，当遇到程序的第n条字节码指令时，便会进入 `${source_root}/src/os/linux/vm/os_linux.cpp` 中的空函数 `breakpoint()`：

```
set args -XX:+TraceBytecodes -XX:StopInterpreterAt=<n>
```

再通过GDB在 `${source_root}/src/hotspot/os/linux/os_linux.cpp` 中的 `breakpoint()` 函数上打上断点:

break breakpoint

为什么要将断点打在这里？

去看 `${source_root}/src/hotspot/share/interpreter/templateInterpreterGenerator.cpp` 里, 函数 `TemplateInterpreterGenerator::generate_and_dispatch` 中对 `stop_interpreter_at()` 的调用就知道了.

接着我们开始运行hotspot:

run

当命中断点时，我们再跳出 `breakpoint()` 函数：

finish

这样就会返回到真正的字节码的执行了。

不过，我们还要跳过函数 `TemplateInterpreterGenerator::generate_and_dispatch` 中插入到字节码真正逻辑前的一些用于debug的逻辑：

```

if (PrintBytecodeHistogram)                histogram_bytecode(t);
// debugging code
if (CountBytecodes || TraceBytecodes || StopInterpreterAt > 0) count_bytecode();
if (PrintBytecodePairHistogram)            histogram_bytecode_pair(t);
if (TraceBytecodes)                       trace_bytecode(t);
if (StopInterpreterAt > 0)                 stop_interpreter_at();

```

比如开启了参数 `-XX:+TraceBytecodes` 和 `-XX:StopInterpreterAt=<n>`，应该跳过的指令如下：

count_bytecode()对应指令:

```
0x7fffe07e8261: incl    0x16901039(%rip)      # 0x7ffff70e92a0 <BytecodeCounter::_counter_value>
```

```
# trace bytecode(t)对应指令:
```

```
0x7fffe07e8267: mov    %rsp,%r12
```

```
0x7fffe07e826a: and    $0xfffffffffffffffff0,%rsp
```

```
0x7fffe07e826e: callq 0x7fffe07c5edf
```

```
0x7fffe07e8273: mov    %r12,%rsp
```

```
0x7fffe07e8276: xor    %r12,%r12
```

stop_interpreter_at()对应指令:

```
0x7fffe07e8279: cmpl    $0x66,0x1690101d(%rip)      # 0x7ffff70e92a0 <BytecodeCounter::_counter_value>
```

```
0x7fffe07e8283: ine    0x7fffe07e828e
```

```
0x7ffffe07e8289: callq 0x7ffff606281a <os::breakpoint()>
```

.....

```
# .....真正的字节码逻辑.....
```

.....

```
# dispatch epilog(tos out, step)对应指令, 用来取下一条指令执行...
```

进入真正的字节码逻辑后，我们就可以使用指令级别的 `stepi` , `nexti` 命令来进行跟踪调试了。（由于汇编代码都是运行期产生的，GDB中没有与源代码的对应符号信息，所以不能用C++源码行级命令 `step` 以及 `next` ）

寻找字节码机器指令的入口手动打断点

关于模板解释器相关知识，可以阅读：[JVM之模板解释器](#).

还是一样，运行GDB：

```
gdb build/linux-x86_64-normal-server-slowdebug/jdk/bin/java
start
break JavaMain
continue
```

我们先在 `${source_root}/src/hotspot/share/interpreter/templateInterpreter.cpp` 的 `DispatchTable::set_entry(...)` 函数上打条件断点，条件是函数实参 `i == <字节码对应十六进制>` ，字节码对应的十六进制见：

`${source_root}/src/hotspot/share/interpreter/bytecodes.hpp` 的 `Bytecodes::Code` .

```
break DispatchTable::set_entry if i==<字节码对应十六进制>
```

然后继续运行

```
continue
```

命中断点后，查看函数实参 `entry` 所指向的内存地址

```
print entry
```

在这个地址上打断点。

```
break *<内存地址>
```

然后继续运行

```
continue
```

命中断点后，就跟前一个方法一样可以直接使用指令级别的 `stepi` , `nexti` 命令来进行跟踪调试了。

配置IDEA

为项目的绑定JDK源码路径

打开IDEA，新建一个项目。然后选择 `File -> Project Structure` ，选到 `SDKs` 选项，新添加上自己刚刚编译生成的JDK， `JDK home path` 为 `${source_root}/build/配置名称/jdk` . 然后在 `Sourcepath` 下移除原本的源码路径（如果有），并添加为前面的源代码，如 `${source_root}/src/java.base/share/classes` 等. 这样以来，我们就可以在IDEA中编辑JDK的JAVA代码，添加自己的注释了。

重新编译JDK的JAVA代码

在添加中文注释后，再编译JDK时会报错：

```
error: unmappable character (0x??) for encoding ascii
```

我们可以在 `${source_root}/make/common/SetupJavaCompilers.gmk` 中，修改两处编码方式的设置，替换原内容：

```
-encoding ascii
```

为：

```
-encoding utf-8
```

这样编译就不会报错了。

而且，如果我们只修改了JAVA代码，无需使用 `make` 命令重新编译整个OpenJDK，而只需要使用以下命令仅编译JAVA模块：

```
make java
```

使用IDEA的Step Into跟踪调试源码

我们发现，在IDEA调试JDK源码时，无法使用 `Step Into` (F7)跟进JDK中的相关函数，这是因为IDEA默认设置不步入这些内置的源码。可以在 `File -> Settings -> Build, Execution, Deployment -> Debugger -> Stepping` 中，取消勾选 `Do not step into the classes` 来取消限制。

参考文章

- [Tips & Tricks: Develop OpenJDK in CLion with Pleasure](#)
- [OpenJDK 编译调试指南\(Ubuntu 16.04 + MacOS 10.15\)](#)
- [JVM-在MacOS系统上使用CLion编译并调试OpenJDK12](#)
- [深入理解Java虚拟机：JVM高级特性与最佳实践\(第3版\)](#)
- [编译JDK源码踩坑纪实](#)
- [How to to debug the HotSpot interpreter](#)
- [JVM之模板解释器](#)