

## 为什么是JDK17

### 语法层面新特性

- 1、文本块
- 2、Switch 表达式增强
- 3、instanceof的模式匹配
- 4、var 局部变量推导

### 模块化及类封装

- 1、记录类 record
- 2、隐藏类 Hidden Classes
- 3、密封类 Sealed Classes
- 4、模块化 Module System
  - 1、什么是模块化
  - 2、声明一个module
  - 3、require 声明module依赖
  - 4、exports 和 opens 声明对外的 API
  - 5、uses 服务开放机制
  - 6、构建模块化 Jar 包
  - 7、类加载机制调整
- 5、GC调整

### GraalVM 虚拟机

- 1、关于 Graal
- 2、使用 GraalVM

## JDK17 新特性梳理

-- 楼兰

## 为什么是JDK17

“你发任你发，我用 Java8”。虽然业界现在对于 JDK8 后每半年更新一次的 JDK 新版本都保持着比较谨慎的态度，但是 JDK17是一个 Java 程序员不得不去关注的新版本。最直观的原因是，作为现代 Java 应用基石的 Spring 和 SpringBoot，都在新版本中走出了抛弃 JDK8，支持 JDK17 的这一步。

## Spring Framework Overview

Spring makes it easy to create Java enterprise applications. It provides everything you need to embrace the Java language in an enterprise environment, with support for Groovy and Kotlin as alternative languages on the JVM, and with the flexibility to create many kinds of architectures depending on an application's needs. **As of Spring Framework 6.0, Spring requires Java 17+.**

Spring supports a wide range of application scenarios. In a large enterprise, applications often exist for a long time and have to run on a JDK and application server whose upgrade cycle is beyond developer control. Others may run as a single jar with the server embedded, possibly in a cloud environment. Yet others may be standalone applications (such as batch or integration workloads) that do not need a server.

Spring is open source. It has a large and active community that provides continuous feedback based on a diverse range of real-world use cases. This has helped Spring to successfully evolve over a very long time.

## 2. System Requirements

Spring Boot 3.2.0 requires [Java 17](#) and is compatible up to and including Java 21. [Spring Framework 6.1.1](#) or above is also required.

Explicit build support is provided for the following build tools:

Build Tool	Version
Maven	3.6.3 or later
Gradle	7.x (7.5 or later) and 8.x

你或许不一定需要像技术极客一样去紧追 JDK 各种令人眼花缭乱的最新特性，但是 JDK17 却是每个 Java 程序员必须走出的下一个里程碑。不光是因为 JDK17 是 JDK8 后一个重要的 LTS 长期支持版本，更是因为在应用生态构建方面，JDK17 会比之前的 JDK11 更加成熟。

个人觉得，跳过 JDK11，直接入手 JDK17，对 JDK8 时代的程序员来说是一个比较实惠的选择。而至于后续的 JDK21 版本，除了虚拟线程比较亮眼外，其他特性相比 JDK17，感觉不痛不痒。因此，接下来，楼兰将在 JDK8 的基础上，带你全面认识一下 JDK17。

## 语法层面新特性

先从一些无关痛痒的小的语法增强带你来走进 JDK17。

### 1、文本块

文本块功能，文本块指多行的字符串，使用连续的三个双引号来包围一段带换行的文字，它避免了换行转义的需要，并支持 `String.format`。

同时添加了两个新的转义字符：

- `\:` 置于行尾，用来将两行连接为一行
- `\s`: 单个空白字符

示例代码：

```
String query =
    """
    SELECT `EMP_ID`, `LAST_NAME` FROM `EMPLOYEE_TB` \s
    WHERE `CITY` = '%s' \
    ORDER BY `EMP_ID`, `LAST_NAME`;
    """;

System.out.println("==== query start =====");
System.out.println(String.format(query, "合肥"));
System.out.println("==== query stop =====");
```

打印结果：

```
SELECT `EMP_ID`, `LAST_NAME` FROM `EMPLOYEE_TB`
WHERE `CITY` = '合肥' ORDER BY `EMP_ID`, `LAST_NAME`;
```

## 2、Switch 表达式增强

从JDK8 到JDK17, Switch 表达式做了很大的增强, 再也不是简单的if-else的替代品了。

扩展switch语句, 使其既可以作为语句使用, 也可以作为表达式使用, 并且两种形式都可以用“传统”或“简化”的作用域和控制流行为。同时添加了yield关键字, 提供break 与switch返回值的功能。

示例 1: 可以将多个匹配写到一起。

```
switch (name) {  
    case "李白", "杜甫", "白居易" -> System.out.println("唐代诗人");  
    case "苏轼", "辛弃疾" -> System.out.println("宋代诗人");  
    default -> System.out.println("其他朝代诗人");  
}
```

示例 2: 每个分支直接返回一个值。

```
int tmp = switch (name) {  
    case "李白", "杜甫", "白居易" -> 1;  
    case "苏轼", "辛弃疾" -> 2;  
    default -> {  
        System.out.println("其他朝代诗人");  
        yield 3;  
    }  
};
```

## 3、instanceof的模式匹配

instances 增加了模式匹配的功能, 如果变量类型经过instances判断能够匹配目标类型, 则对应分支中无需再做类型强转。

示例代码:

```
if (o instanceof Integer i && i > 0) {  
    System.out.println(i.intValue());  
} else if (o instanceof String s && s.startsWith("t")) {  
    System.out.println(s.charAt(0));  
}
```

## 4、var 局部变量推导

对于某些可以直接推导出类型的局部变量, 可以使用var进行声明。

```
var nums = new int[] {1, 2, 3, 4, 5};  
var sum = Arrays.stream(nums).sum();  
System.out.println("数组之和为: " + sum);
```

这个特性仁者见仁, 智者见智。Java 的强类型语法更能保护代码安全。

# 模块化及类封装

从JDK8开始，JDK中陆续更新了很多应用相关的新特性。这些新特性里明显能够看出借鉴了很多新兴的动态语言的特征，让Java变得更年轻有活力了。

## 1、记录类 record

在JDK17中，可以声明一种特殊的类，record。被record定义的类代表的是一种不可变的常量，只能用来描述一种简单的不可变的数据结构。这样我们未来或许就不用再定义一大堆的BO、VO、DTO这些只用来进行值传递的复杂对象了。

record在JDK14中引入，到JDK16才转正。

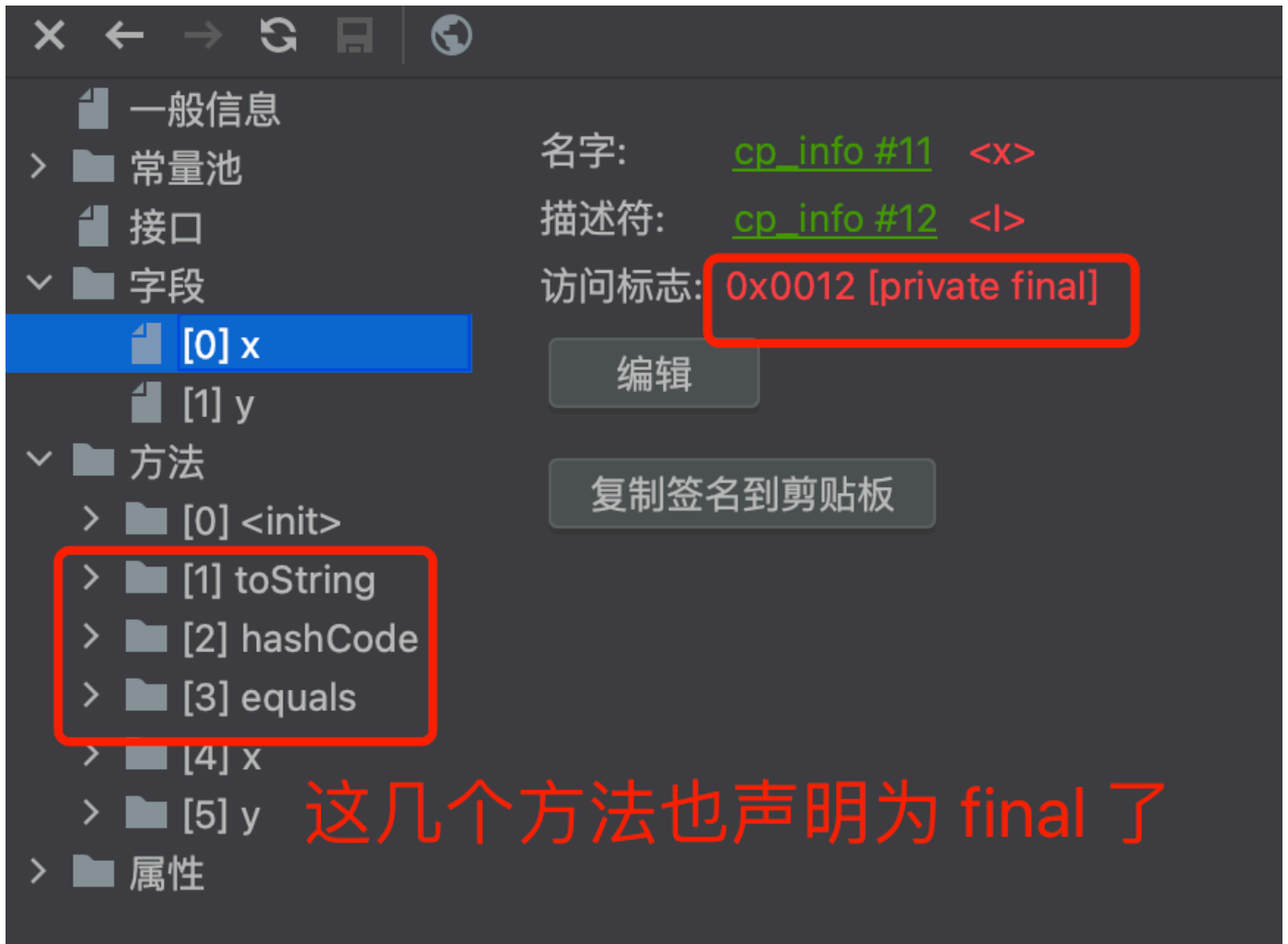
例如可以这样声明一个带有x,y两个属性的Point类。

```
public record Point(int x, int y) {  
  
}
```

然后，这个类只能初始化设置属性值，初始化后，不允许修改属性值，用反射也不行。唯一和我们自己写的POJO有点不同的是，获取属性的方法，与属性同名，而不再是getXXX这样的了。

```
public class RecordTest {  
  
    @Test  
    public void getPoint() throws IllegalAccessException {  
        Point p = new Point(10,20);  
        for (Method method : p.getClass().getMethods()) {  
            System.out.println(method);  
        }  
        for (Field field : p.getClass().getDeclaredFields()) {  
            System.out.println(field);  
            // 不允许通过反射修改值。  
            // field.setAccessible(true);  
            // field.set(p,30);  
        }  
        System.out.println(p.x()+"===="+p.y());  
    }  
}
```

record记录类的实现原理，其实大致相当于给每个属性添加了private final声明。这样就不允许修改。另外，从字节码也能看到，对于record类，同时还实现了toString，hashCode，equals方法，而这些方法都被声明成了final，进一步阻止应用定制record相关的业务逻辑。



## 2、隐藏类 Hidden Classes

从JDK15开始，JDK引入了一个很有意思的特性，隐藏类。隐藏类是一种不能被其他类直接使用的类。隐藏类不再依赖于类加载器，而是通过读取目标类字节码的方式，创建一个对其他类字节码隐藏的class对象，然后通过反射的方式创建对象，调用方法。

我们先来一个示例理解一下什么是隐藏类，再来思考隐藏类有什么用处。

比如先编写一个普通的测试类

```
public class HiddenClass {  
    public String sayHello(String name) {  
        return "Hello, " + name;  
    }  
  
    public static void printHello(String name) {  
        System.out.printf(""  
            Hello, %s !  
            Hello, HiddenClass !  
            %n""", name);  
    }  
}
```

传统方式下，要使用这个类，就需要经过编译，然后类加载的整个过程。但是隐藏类机制允许直接从编译后的class字节码入手，直接使用这个类。

比如，我们可以使用下面的方法获取class字节数组：

```
public void printHiddenClassBytesInBase64(){
    //编译后的 class 文件地址
    String classPath =
"/Users/roykingw/DevCode/JDK17Demo/demoModule/target/classes/com/roy/hidden/HiddenClass.cl
ass";

    try {
        byte[] bytes = Files.readAllBytes(Paths.get(classPath));
        System.out.println(Base64.getEncoder().encodeToString(bytes));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

这样就可以拿到一串编码后的class文件的字节码。接下来，就可以用这个字节码直接生成这个类。例如：

```
public void testInvokeHiddenClass() throws Throwable {
    //class文件的字节码
    String CLASS_INFO =
        "yv66vgAAAD0ANgoAAGADBWAEDAAFAAYBABBBqYXZhL2xhbmcvT2JqZWNOAQAGPgluaXQ+AQADKClWEgAAAAgMAAkAC
        gEAF21ha2VDb25jYXRxaXRoQ29uc3RhbnRzAQAMKEqxYXZhL2xhbmcvU3RyaW5nOy1MamF2YS9sYW5nLlN0cmduZzs
        JAAwADQcADgwAdwAQQAQAmF2YS9sYW5nLlN5c3RlbQEAA29ldAEAFUxqYXZhL2lvL1Byaw50U3RyZWftOwgAEgEAI
        0hlbGxvLCAlcYAhCkhlbGxvLCBIAWRkZW5DbGFzczyAhCiVuCgAUABUHABYMABcAGAEAE2phdmEvaW8vUHJpbmRTdHJ
        lYW0BAAZwcmludGYBADwoTGphdmEvbGFuZy9TdHJpbmc7W0xqYXZhL2xhbmcvT2JqZWNOOy1MamF2YS9pbY9Qcmdu
        FN0cmVhbTsHABoBABpj20vcmluZS12hpZGRlbi9IaWRkZW5DbGFzcwEABENvZGUBAA9MaW51TnVtYmVyVGfibGUBABJ
        Mb2NhbfZhcmlhYmxlVGfibGUBAAR0aGlzaQAQTGNvbS9yb3kvaGlkZGVuL0hpZGRlbkNsYXNzOWEACHNheUhlbGxva
        QAEbmFtZQEAEkxqYXZhL2xhbmcvU3RyaW5nOwEAcnByaw50SGVsbg8BABUoTGphdmEvbGFuZy9TdHJpbmc7KVYBAAp
        Tb3VyY2VGaWxlAQASGLkZGVuQ2xhc3MuamF2YQEAEJEjb3RzdHJhcElldGhvZHMPBgApCgAqACSHACwMAAKALQEAJ
        GphdmEvbGFuZy9pbmZva2UvU3RyaW5nQ29uY2F0RmFjdG9yeQEAmChMamF2YS9sYW5nL2ludm9rZS9NZXRob2RIYW5
        kbGVzJExxb2t1cDtmamF2YS9sYW5nLlN0cmduZztmamF2YS9sYW5nL2ludm9rZS9NZXRob2RUeXB1O0xqYXZhL2xhb
        mcvU3RyaW5nO1tmamF2YS9sYW5nL09iamVjdDspTGphdmEvbGFuZy9pbmZva2UvQ2FsbnFpdGU7CAAvaQAISGVsbG8
        sIAEBAAxJbm5lcNnsYXNzZXMHADIbACVqYXZhL2xhbmcvaW52b2t1L01ldGhvZEhhbmRsZXMKtG9va3VwBwA0AQaea
        mF2YS9sYW5nL2ludm9rZS9NZXRob2RIYW5kbGVzaQAGTG9va3VwACEAGQACAAAAAADAAEAQBAGAAEAGwAAC8AAQA
        BAAAABSq3AAGxAaaaaAgAcAAAABgABAAAAAwadAAAAADAABAAAABQAeAB8AAAABACAACgABABsAAAA7AAEAagAAAAcru
        gAHAACwAAAAAgAcAAAABgABAAAABQAdAAAAFgACAAAABwAeAB8AAAAAAacAIQAIaAEACQajACQAAQAbAAAAQAAGAAE
        AAAASsgALEhEEvQACWQMqu7YAElexAAAAAgAcAAAACgACAAAACQARAA0AHQAAAAwAAQAAABIATQAIaAAAAAwAlAAAAA
        gAmAccAAAAIAAEAkaABAC4MAAAAAAoAAQAxADMANQAZ";

    byte[] classInBytes = Base64.getDecoder().decode(CLASS_INFO);
    Class<?> proxy = MethodHandles.lookup()
        .defineHiddenClass(classInBytes, true,
            MethodHandles.Lookup.ClassOption.NESTMATE)
        .lookupClass();

    // 输出类名
    System.out.println(proxy.getName());
    // 输出类有哪些函数
    for (Method method : proxy.getDeclaredMethods()) {
```

```

        System.out.println(method.getName());
    }
    // 2. 调用对应的方法
    MethodHandle mhPrintHello = MethodHandles.lookup().findStatic(proxy, "printHello",
MethodType.methodType(void.class, String.class));
    mhPrintHello.invokeExact("loulan");
    Object proxyObj = proxy.getConstructors()[0].newInstance();
    MethodHandle mhSayHello = MethodHandles.lookup().findVirtual(proxy, "sayHello",
MethodType.methodType(String.class, String.class));
    System.out.println(mhSayHello.invoke(proxyObj, "loulan"));
}

```

示例很简单，看似花里胡哨，但是其实，这将是以后开发框架非常重要的一个特性。因为这些隐藏类直接操作字节码，可以极大提高 Java 的动态语言能力。

实际上这也是 Java 吸收其他语言优点的一种表现。近年来有很多基于 JVM 的语言都在强调动态语言。比如 Scala, Kotlin 中大量运用匿名函数，Java 自己的 Lambda 表达式，本质上也是一种匿名函数。这些匿名函数在语法层面并不需要提前声明，只要在运行时拿来用就可以了。但是在 JVM 中，Java 一切皆对象，这些匿名函数也必须经过类加载的繁琐过程，并且类的卸载也非常受限制。所以在 Spring 框架中，大量的运用了 ASM 这样的直接操作字节码的技术，就是为了加快这些动态对象的生命周期。但是这些技术方案的实现即麻烦又低效，而 JDK 中引入了隐藏类机制，就可以作为生成动态类的新标准。

## 3、密封类 Sealed Classes

密封类在 JDK15 引入，到 JDK17 中正式转正。

在 JDK8 中，每一个类都可以被任意多个子类继承，并修改其中的内置功能。比如 JDK8 中最重要的类加载双亲委派机制，在应用当中，程序员可以随意挑选一个内置的类加载器，继承出新的类加载器实现，随意打破双亲委派机制。这其实是不太安全的，意味着很多内置的行为得不到保护。而密封类就是用来限制每一个父类可以被哪些子类继承或者实现。

首先在声明类或方法时，如果增加 sealed 修饰，那么还需要同时增加 permits 指定这个类可以被哪些类来继承或实现。例如：

```

public sealed abstract class Shape permits Circle, Rectangle, Square {

    public abstract int lines();
}

```

接下来，Shape 的子类也会要收到限制。在声明类时，需要声明自己的密封属性。可以有三个选项：

- final，表示这个子类不能再被继承了。
- non-sealed 表示这个子类没有密封特性，可以随意继承。
- sealed 表示这个子类有密封特性。再按照之前的方式声明他的子类。

例如针对 Shape，就可以声明这样的一些子类：

```

// 非密封子类，可以随意继承
public non-sealed class Square extends Shape{
    @Override

```

```

    public int lines() {
        return 4;
    }
}
//final 子类, 不可再被继承
public final class Circle extends Shape{
    @Override
    public int lines() {
        return 0;
    }
}
// 密封子类, 继续声明他所允许的子类。
public sealed class Rectangle extends Shape permits FilledRectangle {
    @Override
    public int lines() {
        return 3;
    }
}

public final class FilledRectangle extends Rectangle{
    @Override
    public int lines() {
        return 0;
    }
}

```

比如对 JDK8 的类加载体系, 就可以通过密封类机制, 让子类只能从 SecureClassLoader 或者 URLClassLoader 往下继承, 这样就可以保护 SecureClassLoader 和 URLClassLoader 中的安全行为。

当然, 这只是假设, JDK 的类加载机制并没有这么做。

密封类能够保护父类的安全行为, 但是也有一些限制。父类和指定的子类必须在同一个显式命名的 module 下, 并且子类必须直接继承父类。

## 4、模块化 Module System

这是从 JDK9 之后引入的一个重要机制, 对于熟悉 JDK8 的开发者, 这是一个颠覆性的大变革。如果你真的有升级 JDK 版本的打算, 那么对于这个模块化机制, 你一定不能只是简简单单的看下网上的介绍贴, 必须要做好颠覆三观的准备。

### 1、什么是模块化





































JDK8 中, 我们写的 Java 代码是在一个一个的 package 下面的, 模块化在包之上增加了更高级别的聚合, 它包括一组密切相关的包和资源以及一个新的模块描述符文件。简单点说, module 是 java 中 package 包的上一层抽象。通过 module, java 可以更精确的分配对象的生效范围。

比如, 在 JDK8 的安装目录下, JDK 预设的功能是以一个一个 jar 包的形式存在的, 但是在 JDK17 的安装目录下, 你就看不到那些 jar 包了, 取而代之的, 是一系列以 jmod 后缀的文件, 这些就是一个一个的模块。

< > jmods





jmods				
名称	^	修改日期	大小	种类
 java.base.jmod		2023年6月14日 19:24	17.9 MB	文稿
 java.compiler.jmod		2023年6月14日 19:24	125 KB	文稿
 java.datatransfer.jmod		2023年6月14日 19:24	54 KB	文稿
 java.desktop.jmod		2023年6月14日 19:24	13.6 MB	文稿
 java.instrument.jmod		2023年6月14日 19:24	43 KB	文稿
 java.logging.jmod		2023年6月14日 19:24	117 KB	文稿
 java.management.jmod		2023年6月14日 19:24	901 KB	文稿
 java.management.rmi.jmod		2023年6月14日 19:24	95 KB	文稿
 java.naming.jmod		2023年6月14日 19:24	469 KB	文稿
 java.net.http.jmod		2023年6月14日 19:24	741 KB	文稿
 java.prefs.jmod		2023年6月14日 19:24	86 KB	文稿
 java.rmi.jmod		2023年6月14日 19:24	271 KB	文稿
 java.scripting.jmod		2023年6月14日 19:24	45 KB	文稿
 java.se.jmod		2023年6月14日 19:24	5 KB	文稿
 java.security.jgss.jmod		2023年6月14日 19:24	628 KB	文稿
 java.security.sasl.jmod		2023年6月14日 19:24	84 KB	文稿
 java.smartcardio.jmod		2023年6月14日 19:24	62 KB	文稿
 java.sql.jmod		2023年6月14日 19:24	78 KB	文稿
 java.sql.rowset.jmod		2023年6月14日 19:24	196 KB	文稿
 java.transaction.xa.jmod		2023年6月14日 19:24	6 KB	文稿
 java.xml.crypto.jmod		2023年6月14日 19:24	672 KB	文稿
 java.xml.jmod		2023年6月14日 19:24	4.7 MB	文稿
 jdk.accessibility.jmod		2023年6月14日 19:24	53 KB	文稿
 jdk.attach.jmod		2023年6月14日 19:24	37 KB	文稿
 jdk.charsets.jmod		2023年6月14日 19:24	1.7 MB	文稿
 jdk.compiler.jmod		2023年6月14日 19:24	9.6 MB	文稿
 jdk.crypto.cryptoki.jmod		2023年6月14日 19:24	372 KB	文稿
 jdk.crypto.ec.jmod		2023年6月14日 19:24	134 KB	文稿
 jdk.dynalink.jmod		2023年6月14日 19:24	161 KB	文稿
 jdk.editpad.jmod		2023年6月14日 19:24	10 KB	文稿
 jdk.hotspot.agent.jmod		2023年6月14日 19:24	2.3 MB	文稿
 jdk.httpserver.jmod		2023年6月14日 19:24	109 KB	文稿
 jdk.incubator.foreign.jmod		2023年6月14日 19:24	324 KB	文稿
 jdk.incubator.vector.jmod		2023年6月14日 19:24	708 KB	文稿
 jdk.internal.ed.jmod		2023年6月14日 19:24	10 KB	文稿
 jdk.internal.jvmstat.jmod		2023年6月14日 19:24	93 KB	文稿

 jdk.internal.le.jmod	2023年6月14日 19:24	437 KB	文稿
 jdk.internal.opt.jmod	2023年6月14日 19:24	84 KB	文稿
 jdk.internal.vm.ci.jmod	2023年6月14日 19:24	449 KB	文稿
 jdk.internal.vm.compiler.jmod	2023年6月14日 19:24	4 KB	文稿
 jdk.internal.v...nagement.jmod	2023年6月14日 19:24	4 KB	文稿
 jdk.jartool.jmod	2023年6月14日 19:24	203 KB	文稿

这些jmod文件可以认为是一种特殊的jar包。JMOD设计为在编译时间和链接时间使用，但不在运行时间使用。

也就是说，应用中可以通过只保留需要用的jmod文件来定制自己的JRE。但是这些jmod文件不能配合java -cp/-m 等机制使用。

安装JDK17后，也可以使用java --list-modules 查看到所有的系统模块。

```
[(base) roykingw@roykingwdeMacBook-Pro ~ % java --list-modules
java.base@17.0.8
java.compiler@17.0.8
java.datatransfer@17.0.8
java.desktop@17.0.8
java.instrument@17.0.8
java.logging@17.0.8
java.management@17.0.8
java.management.rmi@17.0.8
java.naming@17.0.8
java.net.http@17.0.8
java.prefs@17.0.8
java.rmi@17.0.8
java.scripting@17.0.8
java.se@17.0.8
java.security.jgss@17.0.8
java.security.sasl@17.0.8
java.smartcardio@17.0.8
java.sql@17.0.8
java.sql.rowset@17.0.8
java.transaction.xa@17.0.8
java.xml@17.0.8
.
```

可以看到，整个JDK都已经用模块化的方式进行了重组，并且，在JDK的安装目录下，也已经取消了JRE目录。这意味着，在新版本的JDK中，应用程序可以定制自己的JRE，只选择应用所需要的模块，而不再需要引入整个JDK庞大的后台功能。

比如，我们如果只需要使用java.base模块中的类型，那么随时可以用一下指令打包出一个可以在服务器上运行的JRE：

```
jlink -p $JAVA_HOME/jmods --add-modules java.base --output basejre
```

这个basejre就可以像安装 JDK 一样部署到一台新的服务器上运行了。

## 2、声明一个module

引入模块化机制后，应用需要在每个模块的根目录下创建一个module-info.java文件，用来声明一个模块。然后在这个文件中，用module关键字，声明一个模块。例如：

```
module roy.demomodule {  
}
```

这样，当前目录下的所有package下的代码，都将属于同一个module。module名字必须全局唯一。至于具体的格式，没有强制要求，不过通常的惯例是类似于包结构，全部用小写，用.连接。

接下来就需要在roy.demomodule中声明module的一些补充信息。这些补充信息主要包括：

- 对其他module的依赖关系
- 当前module对外开放的 API
- 使用和服务

## 3、require 声明module依赖

在module-info.java中首先要声明当前module需要依赖哪些外部模块。比如，如果你要使用junit，那么除了要在pom.xml中引入junit对应的依赖外，还需要在module-info.java中添加配置，否则项目编译就会报错。

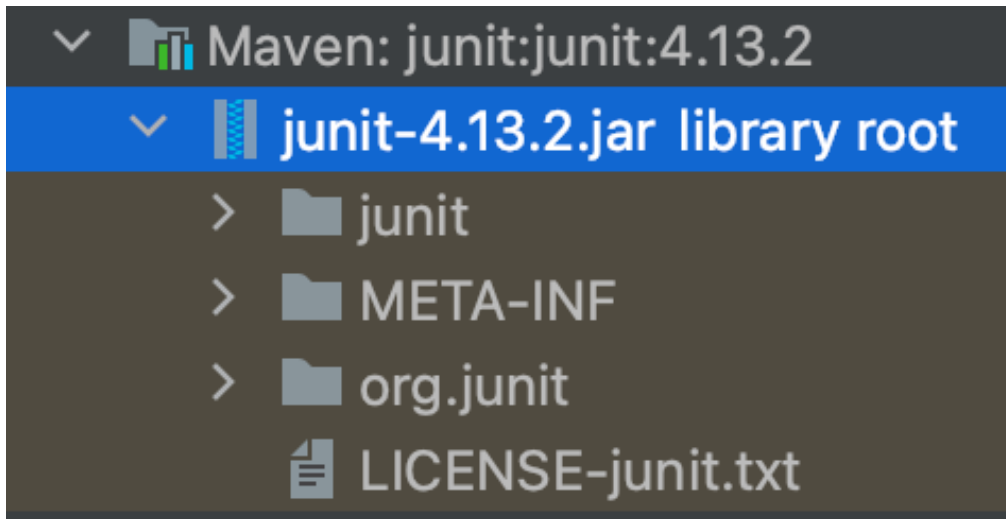
```
requires junit;
```

这里要注意，对于显式声明了module-info.java的模块来说，模块名是显而易见的。但是对于没有声明module-info.java的非模块化jar包来说，默认就会创建具有jar包名称的模块。而这个名称还去掉版本号之后的标准包名。

比如，在demoModule1中，我引入了如下的junit依赖

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.13.2</version>  
</dependency>
```

那么此时，从 Maven仓库下载下来的jar包是junit-4.13.2.jar。那么此时，junit 的模块名就是junit。



另外，从JDK9开始，JDK所有的内置基础代码都已经按照模块化进行了重组，所以，要使用JDK内置的功能，也同样需要通过requires声明所需要的依赖。比如，如果你要使用JDBC功能，那么就需要单独引入java.sql这个模块。

```
requires java.sql
```

当一个module需要另一个module的类型进行编译，但不想在运行时依赖它时，可以使用requires static进行声明。如果module A requires static module B，那么此时，A就只需要B参与模块编译，在运行时，可以没有B模块。类似于Maven当中的compile配置。

## 4、exports 和 opens 声明对外的 API

接下来，当要进行跨模块的功能访问时，需要在模块上声明模块对外的API。

例如，当我们想要使用junit构建一个单元案例时，如果直接执行就会看到下面的报错提示：

```
package com.roy.language;
import org.junit.Test;
public class SwitchDemo {

    @Test
    public void demo1(){
        String name="李白";
        switch (name) {
            case "李白", "杜甫", "白居易" -> System.out.println("唐代诗人");
            case "苏轼", "辛弃疾" -> System.out.println("宋代诗人");
            default -> System.out.println("其他朝代诗人");
        }
    }
}
```

Tests failed: 1 of 1 test - 3 ms

/Users/roykingw/tools/jdk/jdk-17.0.8.jdk/Contents/Home/bin/java ...

```
java.lang.reflect.InaccessibleObjectException: Unable to make public void com.roy.language.SwitchDemo.demo1()
    accessible: module roy.demomodule does not "exports com.roy.language" to module junit
```

这就是因为这个单元案例实际上需要经过junit模块调用到当前的模块。当出现这种跨模块的调用时，就需要在模块中声明有哪些 API 是其他模块可以调用的。而声明的方式，就是提示当中的exports关键字。使用exports关键字可以对外开放哪些package。

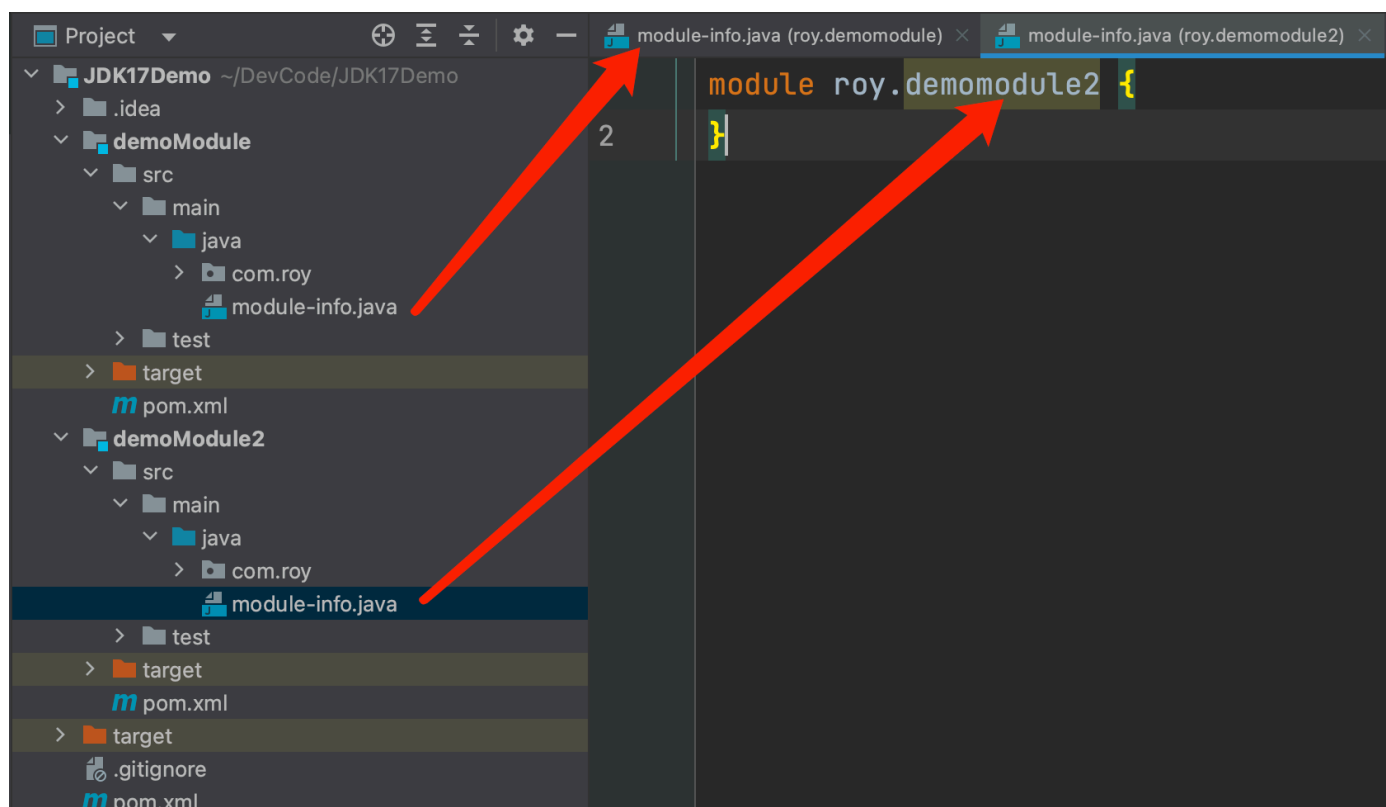
另外，需要补充一下的是，exports关键字开放的成员在编译和运行时都可以访问，但是不能用反射的方式访问。如果想要通过反射的方式访问一个模块内的成员，需要改为使用opens关键字声明。声明方式和exports一样。

## 5、uses 服务开放机制

基于模块化机制，JDK 还重新定制了 SPI 机制，来实现接口与服务实现类的解耦。

JDK8中的SPI机制还记得吧。

这里，我们就需要增加一个自定义模块来演示这种 SPI机制。比如，在下面这个示例中，就用 Maven 创建了两个模块，demoModule的模块名为roy.demomodule，demoModule2 的模块名为roy.demomodule2。在roy.demomodule模块中引入了roy.demomodule2模块。



接下来，在demoModule2 模块中，添加一个接口以及两个不同的实现类。

```
package com.roy.service;

public interface HelloService {
    String sayHello(String name);
}
```

```
package com.roy.service.impl;

import com.roy.service.HelloService;

public class MorningHello implements HelloService {
    @Override
    public String sayHello(String name) {
        return "good morning " + name;
    }
}
```

```
package com.roy.service.impl;

import com.roy.service.HelloService;

public class EveningHello implements HelloService {
    @Override
    public String sayHello(String name) {
        return "good evening " + name;
    }
}
```

这时，就可以在 demoModule2 模块中对外暴露一个HelloService的服务接口，并且选择对外暴露这个服务接口的一个或多个服务实现类。

```
module roy.demomodule2 {
    exports com.roy.service;
    provides com.roy.service.HelloService with
        com.roy.service.impl.MorningHello,
        com.roy.service.impl.EveningHello;
}
```

而在调用方，也就是demoModule模块中，可以在module-info.java文件中使用uses关键字声明要使用的服务。

```
module roy.demomodule {
    requires roy.demomodule2;
    uses com.roy.service.HelloService;
}
```

接下来，就可以在demoModule模块中，使用 SPI 机制直接调用另一个模块的服务。

```
public class ServiceDemo {

    public static void main(String[] args) {
        ServiceLoader<HelloService> services = ServiceLoader.load(HelloService.class);
        for (HelloService service : services) {
            System.out.println(service.sayHello("loulou"));
        }
    }
}
```

通过这种机制，可以实现服务之间的解耦。未来demoModule2可以通过调整module-info.java，快速替换新的服务实现类，而对调用方没有任何影响。

其实可以看到，这种机制就已经有了很多微服务的影子了。

## 6、构建模块化 Jar 包

当这些模块代码构建好了之后，就可以放到服务器上运行了。与模块化机制配套的，在java指令中，也增加了使用模块的参数。

例如，将我们之前演示的两个模块导出成jar包后，就可以通过以下参数在服务器执行：

```
(base) % java --module-path demoModule/demoModule.jar:demoModule2_jar/demoModule2.jar -m
roy.demomodule/com.roy.spi.ServiceDemo
good morning loulou
good evening loulou
```

当然，也可以快速检索这些目录下的模块情况

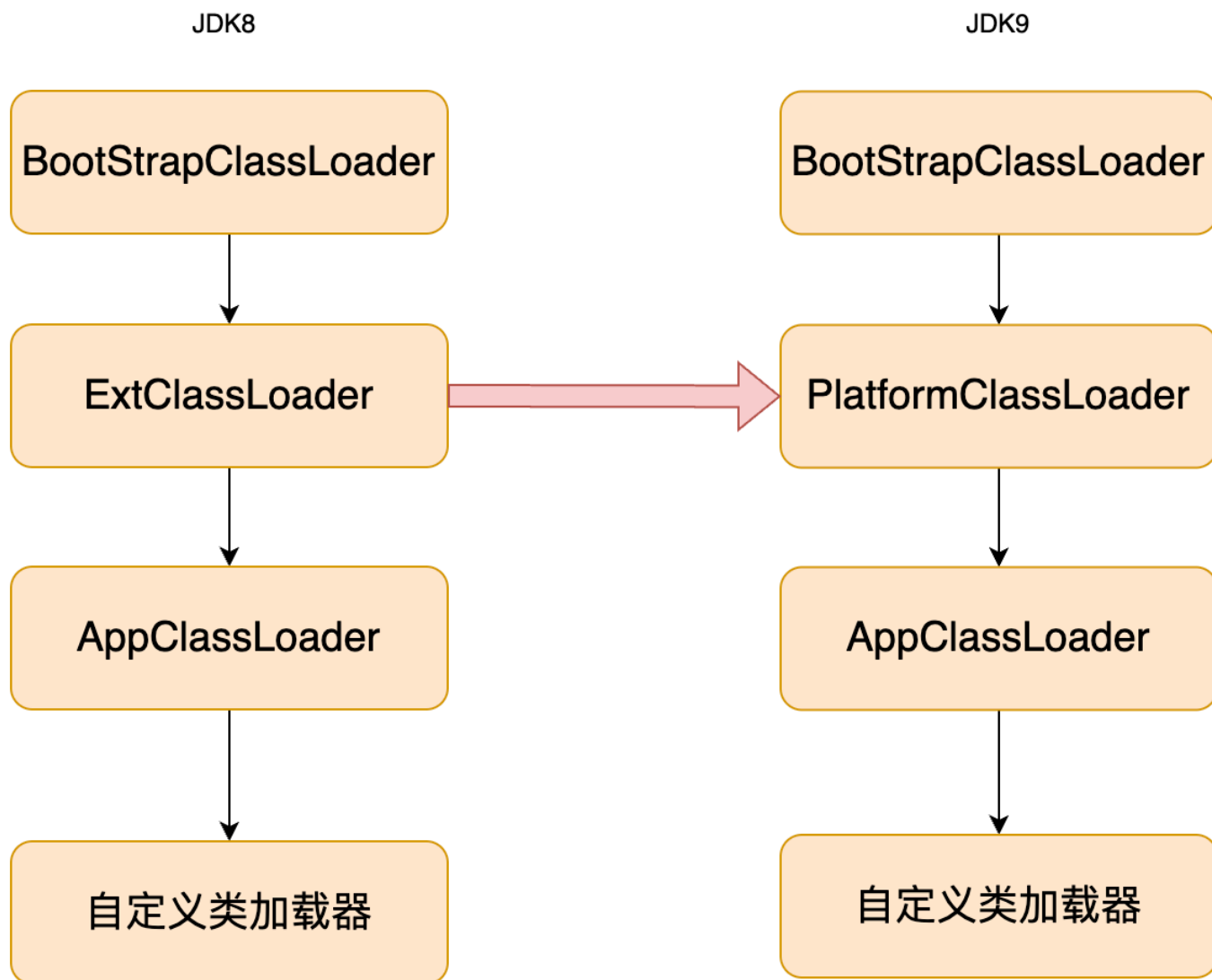
```
(base) % java --module-path demoModule:demoModule2_jar --list-modules

jdk.unsupported@17.0.8
jdk.unsupported.desktop@17.0.8
jdk.xml.dom@17.0.8
jdk.zipfs@17.0.8
roy.demomodule
file:///Users/roykingw/DevCode/JDK17Demo/out/artifacts/demoModule/demoModule.jar
roy.demomodule2
file:///Users/roykingw/DevCode/JDK17Demo/out/artifacts/demoModule2_jar/demoModule2.jar
```

## 7、类加载机制调整

与模块化机制对应，JDK9 往后的类加载机制也做了不小的调整。





我们知道，在 JDK8 中的类加载体系可以简单概括为三点：

- JDK 中的类加载器分为BootstrapClassLoader,ExtClassLoader和AppClassLoader，通过parent属性构成亲属关系。
- 每个类加载器对应一个独立的加载目录，并对加载过的类保留一个缓存。
- 双亲委派机制，也就是加载一个类时，要向上委托查找，向下委托加载。

而使用模块化机制后，虽然 JDK 也依然在尽力兼容传统的类加载体系，但是，为了兼容模块化机制，JDK 还是对类加载体系做了几个非常明显的调整。

#### 1、用平台类加载器PlatformClassloader代替扩展类加载器 ExtClassLoader。

这是一个很自然的变化。以往保留ExtClassLoader是为了在 JDK 的标准实现之外，引入一些具有额外扩展功能的 Jar 包。而使用模块化机制后，整个 JDK 都基于模块化进行了构建。由一系列 JMOD 文件构成的 JDK 已经天生就具备了可扩展的能力，自然也就不需要扩展类加载器了。

#### 2、调整类加载器的实现

以往ExtClassLoader和AppClassLoader都继承自URLClassLoader，现在PlatformClassLoader和AppClassLoader 都改为继承自BuildinClassLoader。在BuildinClassLoader中实现了新的模块化架构下类如何从模块中加载的逻辑，以及模块中资源可访问性的处理。



另外，以往在JDK中看不到的BootstrapClassLoader，在新的架构下也已经有了明确的类来进行描述。只不过，为了保持与之前代码的兼容性，所有获取BootstrapClassLoader的场景(比如String.class.getClassLoader())还是会返回null，而不是BootstrapClassLoader实例。

```
public class ClassLoaders {

    private ClassLoaders() { }

    private static final JavaLangAccess JLA = SharedSecrets.getJavaLangAccess();

    // the built-in class loaders
    private static final BootClassLoader BOOT_LOADER;
    private static final PlatformClassLoader PLATFORM_LOADER;
    private static final AppClassLoader APP_LOADER;
```

### 3、调整双亲委派机制

在Java模块化系统中明确规定了三个类加载器各自负责加载哪些系统模块。当PlatformClassLoader和AppClassLoader收到类加载请求时，在委派给父类加载器加载前，会先判断该类是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责加载那个模块的加载器完成加载。

但是为了与之前的代码兼容，在自定义实现类加载器时，还是保持按照以前的双亲委派机制进行工作。

## 5、GC调整

另外，从JDK8到JDK17，还涉及到了非常多的调整，这些调整让Java具备了更多现代语言的特性，并且执行效率也在不断提高。

比如，重写了Socket底层的API代码，让这写代码使用更简单，实现也更易于维护和替换。

默认禁用偏向锁。从JDK1.6开始，对synchronized关键字的实现就形成了无锁->偏向锁->轻量级锁->重量级锁的锁升级过程。而在JDK15中，默认就废弃了偏向锁。当然，目前还是可以通过添加参数-XX:+UseBiasedLocking手动开启偏向锁。

不过JDK在升级过程中，一直保持着良好的向下兼容性，因此，有很多优化调整对开发工作影响都还不是太大。但是在这些调整当中，有一部分调整是大家需要额外关心一下的，那就是对于GC的调整。

### 1、ZGC转正

ZGC在之前已经做过介绍，是现在最为强大的一个垃圾回收器，自JDK11开始引入，从JDK15开始正式投入了使用。现在使用-XX:+UseZGC参数就可以快速使用ZGC。

另外，ZGC的具体实现其实也在版本升级过程中不断优化。在JDK17中使用指令java -XX:+PrintFlagsFinal -version可以简单看到，与ZGC相关的系统不稳定参数已经基本没有了。G1的还有一大堆这也说明ZGC的算法优化已经相当成熟了。

随ZGC登场的，还有RedHat推出的Shenandoah垃圾回收器。尽管Oracle一直比较抵触这个非官方推出的垃圾回收器，但是最终也还是将Shennandoah垃圾回收器以可选的方案集成了进来。现在可以使用-XX:+UseShenandoahGC参数手动选择shennandoah。

### 2、废除CMS

虽然 CMS 作为 G1 之前唯一的一款并发的垃圾回收器，在相当长的时间里，都扮演者非常重要的角色。在最为经典的 JDK8 时代，尽管 CMS 一直没有作为默认垃圾回收器登场过，但是关于 G1 和 CMS 的比较以及取舍，一直都是业界津津乐道的话题。但是，随着 G1 垃圾回收器发展得更为完善，以及后续 ZGC，shenandoah 等现代垃圾回收器开始登场，过于复杂的 CMS 垃圾回收器还是退出了历史舞台。

在 JDK14 中，就彻底删除了 CMS 垃圾回收器。与 CMS 一起退场的，还有 Serial 垃圾回收器。SerialOld 这个最早的垃圾回收器其实早就应该退出历史舞台了，只不过由于他一直作为 CMS 的补充方案而一直保留。这次也终于随着 CMS 一起退出了。

# GraalVM 虚拟机

Graal 编译器以及由此诞生的 GraalVM，虽然目前还处在实验阶段，但是也是 Java 程序员们必须要了解的，因为他未来极有可能替代 HotSpot，成为 Java 生态的下一代技术基础。

## 1、关于 Graal

Graal 编译器最早是作为 HotSpot 的 C1 编译器的下一代编译器设计的，使用 Java 语言进行编写。2012 年，Graal 编译器才发展成为一个独立的 Java 编译项目。而早期的 Graal 其实也和 C1，C2 一样，需要与 HotSpot 虚拟机配合工作。但是随着 JDK9 开始推出 JVMCI (Java 虚拟机编译器接口)，才让 Graal 可以从 HotSpot 中独立出来，并逐渐形成了现在的 GraalVM。

虽然你可能对 Graal 了解不多，但是，Graal 其实一直深受业界关注。Oracle 公司希望它能够发展成为一个更完美的编译器，高编译效率、高输出质量、支持提前编译和即时编译，同时支持应用于包括 HotSpot 在内的不同虚拟机。而使用 C/C++ 编写的 C1 和 C2 编译器，也逐渐变得越来越臃肿，维护和更新都更加困难。这时使用 Java 语言编写的 Graal 自然就成了首选。

另外，在业务层面，Java 也急需一种更高效的编译器来迎合现在越来越火爆的云原生架构。现在作为 Java 主流的服务端版本总体上是面向大规模，长时间运行的系统设计的。像即时编译器 (JIT)、性能优化、垃圾回收等代表性的特征都是面向程序长时间运行设计的，需要一段时间预热才能达到最佳性能，才能享受硬件规模提升带来的红利。但是现在的微服务背景下，对服务的规模以及稳定性要求在逐渐降低，反而对容器化、启动速度、预热时间等方面提出了新的要求。而这些方面都是 Java 的弱项。因此 Java 语言也需要这样一款新的虚拟机，来提升与很多新出来的现代语言，比如 golang 的竞争优势。

## 2、使用 GraalVM

接下来我们来使用一下 GraalVM。在 GraalVM 的官方文档中，首先有一段对于 GraalVM 的整体描述：

### GraalVM Overview

GraalVM compiles your Java applications ahead of time into standalone binaries. These binaries are smaller, start up to 100x faster, provide peak performance with no warmup, and use less memory and CPU than applications running on a Java Virtual Machine (JVM).

GraalVM reduces the attack surface of your application. It excludes unused classes, methods, and fields from the application binary. It restricts reflection and other dynamic Java language features to build time only. It does not load any unknown code at run time.

Popular microservices frameworks such as Spring Boot, Micronaut, Helidon, and Quarkus, and cloud platforms such as Oracle Cloud Infrastructure, Amazon Web Services, Google Cloud Platform, and Microsoft Azure all support GraalVM.

With profile-guided optimization and the G1 (Garbage-First) garbage collector, you can get lower latency and on-par or better peak performance and throughput compared to applications running on a Java Virtual Machine (JVM).

You can use the GraalVM JDK just like any other Java Development Kit in your IDE.

从这段整体描述就能看到，使用 GraalVM 和使用其他的 JDK，没有什么大的区别。所以，使用 GraalVM 的方式也是 下载-》配置环境变量-》编译-》执行 几个步骤。

GraalVM 的官网地址是：<https://www.graalvm.org>。官网上目前就可以下载对应版本的产品。当前有 Java17 和 Java21 两个版本。



JDK版本选择17，我当前的服务器是AArch64的Linux，选择对应的信息后即可下载。

下载下来后是一个tar包压缩文件。接下来跟安装jdk一样，解压，配置JAVA\_HOME 环境变量，就可以用java -version进行测试了。这部分就略过了。比如我安装后的结果是这样的

```
[oper@localhost ~]$ java -version
java version "17.0.9" 2023-10-17 LTS
Java(TM) SE Runtime Environment Oracle GraalVM 17.0.9+11.1 (build 17.0.9+11-LTS-jvmdi-23.0-b21)
Java HotSpot(TM) 64-Bit Server VM Oracle GraalVM 17.0.9+11.1 (build 17.0.9+11-LTS-jvmdi-23.0-b21, mixed mode, sharing)
```

另外，在 GraalVM 中还提供了一个管理指令 gu

```
[oper@localhost ~]$ gu list
ComponentId      Version      Component name      Stability
Origin
-----
-----
graalvm          23.0.2      GraalVM Core        Supported
native-image     23.0.2      Native Image        Early adopter
```

接下来，写一个简单的Java 代码进行测试。Hello.java

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

使用 javac 编译成 Hello.class 文件。然后使用 java Hello 进行执行。使用 time 指令看到执行的时间如下：

```
[oper@localhost ~]$ time java Hello  
Hello World!  
  
real    0m0.062s  
user    0m0.067s  
sys     0m0.011s
```

接下来，切换到 Oracle 的 JDK17，也同样执行这个程序，做个简单的对比。结果如下：

```
[oper@localhost ~]$ time java Hello  
Hello World!  
  
real    0m0.059s  
user    0m0.027s  
sys     0m0.036s
```

然后，GraalVM 还提供了一个功能，可以将 Class 文件直接编译成本地镜像，这些本地镜像不需要 JVM 虚拟机也能直接运行。这就是 Graal 的 AOT 提前编译。

关于 AOT，之前已经介绍过，这里就不多说了。

但是我在编译这个简单的 Hello 时，却遇到了意想不到的错误：

```
[oper@localhost ~]$ native-image Hello  
=====
```

GraalVM Native Image: Generating 'hello' (executable)...

```
=====
```

[1/8] Initializing...  
(3.4s @ 0.09GB)  
Java version: 17.0.9+11-LTS, vendor version: Oracle GraalVM 17.0.9+11.1  
Graal compiler: optimization level: 2, target machine: armv8-a, PGO: ML-inferred  
C compiler: gcc (redhat, aarch64, 11.4.1)  
Garbage collector: Serial GC (max heap size: 80% of RAM)

[2/8] Performing analysis... [\*\*\*\*]  
(7.6s @ 0.24GB)  
1,831 (59.26%) of 3,090 types reachable  
1,733 (46.69%) of 3,712 fields reachable  
7,726 (35.98%) of 21,471 methods reachable

```
623 types,      0 fields, and 285 methods registered for reflection
49 types,      32 fields, and 48 methods registered for JNI access
4 native libraries: dl, pthread, rt, z
[3/8] Building universe...
      (1.0s @ 0.25GB)
[4/8] Parsing methods...    [**]
      (2.6s @ 0.22GB)
[5/8] Inlining methods...   [***]
      (0.6s @ 0.21GB)
[6/8] Compiling methods...  [****]
      (16.6s @ 0.25GB)
[7/8] Layouting methods...  [*]
      (0.4s @ 0.39GB)
[8/8] Creating image...     [*
]                                                                    (0.0s @ 0.27GB)
-----
-----
2.7s (8.1% of total time) in 146 GCs | Peak RSS: 0.95GB | CPU
load: 1.95
-----
-----
Produced artifacts:
/home/oper/svm_err_b_20231129T171758.715_pid2504.md (build_info)
=====
=====
Failed generating 'hello' after 33.1s.

The build process encountered an unexpected error:

> java.lang.RuntimeException: There was an error linking the native image: Linker command
exited with 1

Linker command executed:
/usr/bin/gcc -z noexecstack -Wl,--gc-sections -Wl,--version-script,/tmp/SVM-
6698742675696986223/exported_symbols.list -Wl,-x -o /home/oper/hello hello.o
/home/oper/graalvm-jdk-17.0.9/lib/svm/clibraries/linux-aarch64/liblibchelper.a
/home/oper/graalvm-jdk-17.0.9/lib/static/linux-aarch64/glibc/libnet.a /home/oper/graalvm-
jdk-17.0.9/lib/static/linux-aarch64/glibc/libnio.a /home/oper/graalvm-jdk-
17.0.9/lib/static/linux-aarch64/glibc/libjava.a /home/oper/graalvm-jdk-
17.0.9/lib/static/linux-aarch64/glibc/libfdlibm.a /home/oper/graalvm-jdk-
17.0.9/lib/svm/clibraries/linux-aarch64/libjvm.a -Wl,--export-dynamic -v -L/tmp/SVM-
6698742675696986223 -L/home/oper/graalvm-jdk-17.0.9/lib/static/linux-aarch64/glibc -
L/home/oper/graalvm-jdk-17.0.9/lib/svm/clibraries/linux-aarch64 -lz -lpthread -ldl -lrt

Linker command output:
使用内建 specs。
COLLECT_GCC=/usr/bin/gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/aarch64-redhat-linux/11/lto-wrapper
目标: aarch64-redhat-linux
```

```
配置为: ../configure --enable-bootstrap --enable-host-pie --enable-host-bind-now --enable-
languages=c,c++,fortran,lto --prefix=/usr --mandir=/usr/share/man --
infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-shared
--enable-threads=posix --enable-checking=release --with-system-zlib --enable-__cxa_atexit
--disable-libunwind-exceptions --enable-gnu-unique-object --enable-linker-build-id --with-
gcc-major-version-only --enable-plugin --enable-initfini-array --without-isl --enable-
multilib --with-linker-hash-style=gnu --enable-gnu-indirect-function --build=aarch64-
redhat-linux --with-build-config=bootstrap-lto --enable-link-serialization=1
线程模型: posix
Supported LTO compression algorithms: zlib zstd
gcc 版本 11.4.1 20230605 (Red Hat 11.4.1-2) (GCC)
COMPILER_PATH=/usr/libexec/gcc/aarch64-redhat-linux/11:/usr/libexec/gcc/aarch64-redhat-
linux/11:/usr/libexec/gcc/aarch64-redhat-linux:/usr/lib/gcc/aarch64-redhat-
linux/11:/usr/lib/gcc/aarch64-redhat-linux/
LIBRARY_PATH=/usr/lib/gcc/aarch64-redhat-linux/11:/usr/lib/gcc/aarch64-redhat-
linux/11/../../../../lib64:/lib/../../lib64:/usr/lib/../../lib64:/usr/lib/gcc/aarch64-redhat-
linux/11/../../../../lib:/usr/lib/
COLLECT_GCC_OPTIONS='-z' 'noexecstack' '-o' '/home/oper/hello' '-v' '-L/tmp/SVM-
6698742675696986223' '-L/home/oper/graalvm-jdk-17.0.9/lib/static/linux-aarch64/glibc' '-
L/home/oper/graalvm-jdk-17.0.9/lib/svm/clibraries/linux-aarch64' '-mlittle-endian' '-
mabi=lp64' '-dumpdir' '/home/oper/hello.'
/usr/libexec/gcc/aarch64-redhat-linux/11/collect2 -plugin /usr/libexec/gcc/aarch64-
redhat-linux/11/liblto_plugin.so -plugin-opt=/usr/libexec/gcc/aarch64-redhat-linux/11/lto-
wrapper -plugin-opt=-fresolution=/tmp/cc7AWuPB.res -plugin-opt=-pass-through=-lgcc -
plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-
lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --no-add-needed --eh-frame-hdr --hash-
style=gnu -dynamic-linker /lib/ld-linux-aarch64.so.1 -X -EL -maarch64linux -o
/home/oper/hello -z noexecstack /usr/lib/gcc/aarch64-redhat-
linux/11/../../../../lib64/crt1.o /usr/lib/gcc/aarch64-redhat-
linux/11/../../../../lib64/crti.o /usr/lib/gcc/aarch64-redhat-linux/11/crtbegin.o -
L/tmp/SVM-6698742675696986223 -L/home/oper/graalvm-jdk-17.0.9/lib/static/linux-
aarch64/glibc -L/home/oper/graalvm-jdk-17.0.9/lib/svm/clibraries/linux-aarch64 -
L/usr/lib/gcc/aarch64-redhat-linux/11 -L/usr/lib/gcc/aarch64-redhat-
linux/11/../../../../lib64 -L/lib/../../lib64 -L/usr/lib/../../lib64 -L/usr/lib/gcc/aarch64-
redhat-linux/11/../../../../ --gc-sections --version-script /tmp/SVM-
6698742675696986223/exported_symbols.list -x hello.o /home/oper/graalvm-jdk-
17.0.9/lib/svm/clibraries/linux-aarch64/liblibchelper.a /home/oper/graalvm-jdk-
17.0.9/lib/static/linux-aarch64/glibc/libnet.a /home/oper/graalvm-jdk-
17.0.9/lib/static/linux-aarch64/glibc/libnio.a /home/oper/graalvm-jdk-
17.0.9/lib/static/linux-aarch64/glibc/libjava.a /home/oper/graalvm-jdk-
17.0.9/lib/static/linux-aarch64/glibc/libfdlibm.a /home/oper/graalvm-jdk-
17.0.9/lib/svm/clibraries/linux-aarch64/libjvm.a --export-dynamic -lz -lpthread -ldl -lrt
-lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -
lgcc_s --pop-state /usr/lib/gcc/aarch64-redhat-linux/11/crtend.o /usr/lib/gcc/aarch64-
redhat-linux/11/../../../../lib64/crtn.o
/usr/bin/ld: 找不到 -lz
collect2: 错误: ld 返回 1
```

Please inspect the generated error report at:  
/home/oper/svm\_err\_b\_20231129T171758.715\_pid2504.md

If you are unable to resolve this problem, please file an issue with the error report at:



<https://graalvm.org/support>

经分析，这是因为我当前服务器上缺少zlib库导致。所以需要先安装zlib库

```
# root权限安装zlib库
sudo yum install zlib-devel
```

之后再重新编译，就可以编译出一个可以执行的hello应用程序。这个应用程序不需要JDK也能正常运行，并且执行速度也更快。

```
[oper@localhost ~]$ java -version
bash: java: command not found...
Install package 'java-11-openjdk-headless' to provide command 'java'? [N/y] n

[oper@localhost ~]$ time ./hello
Hello World!

real    0m0.006s
user    0m0.000s
sys     0m0.006s
```

稍有曲折，完成了第一次GraalVM的体验。从这个过程中可以简单看出，GraalVM的这种AOT编译模式，能够极大提升Java程序的执行速度，更贴合现在的微服务，云原生的技术环境。所以，或许不久的将来，深入理解GraalVM有可能成为每个java程序员的必修课。

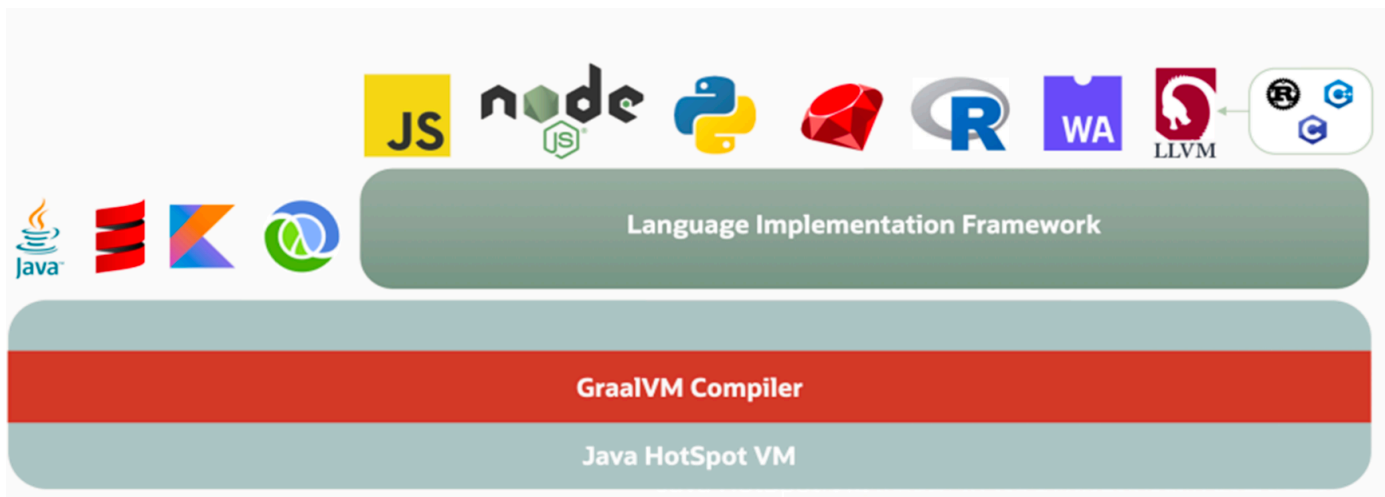
另外，从当前的官网文档中，可以看到 GraalVM 更为强大之处。

## GraalVM as a Platform

GraalVM is an open ecosystem and allows users to implement a custom language or tool on top of it with the [Truffle language implementation framework](#) which offers APIs for writing interpreters for programming languages in the form of Java programs.

GraalVM loads and runs the Truffle framework, which itself is a Java program – a collection of JAR files – together with interpreters. These get optimized at runtime into efficient machine code for executing loaded programs.

基于这个 Truffle 框架，未来完全可以开发出各种语言的翻译器，这样，其他一些常用的语言也可以在 GraalVM 上执行。想象一下，未来js,python,php, lua等等这些语言都可以在 GraalVM 上执行，再加上这种本地镜像的执行方式，会是一种什么样的景象？



所以，不要再说 Java 没落了，Java 未来可期，你我共同期待!!!

有道云笔记链接: <https://note.youdao.com/s/6cCF3AsL>