36 SubstrateVM：AOT编译框架

今天我们来聊聊 GraalVM 中的 Ahead-Of-Time（AOT）编译框架 SubstrateVM。

先来介绍一下 AOT 编译，所谓 AOT 编译，是与即时编译相对立的一个概念。我们知道，即时编译指的是在程序的运行过程中，将字节码转换为可在硬件上直接运行的机器码，并部署至托管环境中的过程。

而 AOT 编译指的则是，在**程序运行之前**，便将字节码转换为机器码的过程。它的成果可以是需要链接至托管环境中的动态共享库，也可以是独立运行的可执行文件。

狭义的 AOT 编译针对的目标代码需要与即时编译的一致，也就是针对那些原本可以被即时编译的代码。不过，我们也可以简单地将 AOT 编译理解为类似于 GCC 的静态编译器。

AOT 编译的优点显而易见：我们无须在运行过程中耗费 CPU 资源来进行即时编译，而程序也能够在启动伊始就达到理想的性能。

然而，与即时编译相比，AOT 编译无法得知程序运行时的信息，因此也无法进行基于类层次分析的完全虚方法内联，或者基于程序 profile 的投机性优化（并非硬性限制，我们可以通过限制运行范围，或者利用上一次运行的程序 profile 来绕开这两个限制）。这两者都会影响程序的峰值性能。

Java 9 引入了实验性 AOT 编译工具jaotc。它借助了 Graal 编译器，将所输入的 Java 类文件转换为机器码，并存放至生成的动态共享库之中。

在启动过程中，Java 虚拟机将加载参数 `-XX:AOTLibrary` 所指定的动态共享库，并部署其中的机器码。这些机器码的作用机理和即时编译生成的机器码作用机理一样，都是在方法调用时切入，并能够去优化至解释执行。

由于 Java 虚拟机可能通过 Java agent 或者 C agent 改动所加载的字节码，或者这份 AOT 编译生成的机器码针对的是旧版本的 Java 类，因此它需要额外的验证机制，来保证即将链接的机器码的语义与对应的 Java 类的语义是一致的。

jaotc 使用的机制便是类指纹（class fingerprinting）。它会在动态共享库中保存被 AOT 编译的 Java 类的摘要信息。在运行过程中，Java 虚拟机负责将该摘要信息与已加载的 Java

类相比较，一旦不匹配，则直接舍弃这份 AOT 编译的机器码。

jaotc 的一大应用便是编译 java.base module，也就是 Java 核心类库中最为基础的类。这些类很有可能会被应用程序所调用，但调用频率未必高到能够触发即时编译。

因此，如果 Java 虚拟机能够使用 AOT 编译技术，将它们提前编译为机器码，那么将避免在执行即时编译生成的机器码时，因为“不小心”调用到这些基础类，而需要切换至解释执行的性能惩罚。

不过，今天要介绍的主角并非 jaotc，而是同样使用了 Graal 编译器的 AOT 编译框架 SubstrateVM。

SubstrateVM 的设计与实现

SubstrateVM 的设计初衷是提供一个高启动性能、低内存开销，并且能够无缝衔接 C 代码的 Java 运行时。它与 jaotc 的区别主要有两处。

第一，SubstrateVM 脱离了 HotSpot 虚拟机，并拥有独立的运行时，包含异常处理，同步，线程管理，内存管理（垃圾回收）和 JNI 等组件。

第二，SubstrateVM 要求目标程序是封闭的，即不能动态加载其他类库等。基于这个假设，SubstrateVM 将探索整个编译空间，并通过静态分析推算出所有虚方法调用的目标方法。最终，SubstrateVM 会将所有可能执行到的方法都纳入编译范围之中，从而免于实现额外的解释执行器。

> 有关 SubstrateVM 的其他限制，你可以参考这篇文档。

从执行时间上来划分，SubstrateVM 可分为两部分：native image generator 以及 SubstrateVM 运行时。后者 SubstrateVM 运行时便是前面提到的精简运行时，经过 AOT 编译的目标程序将跑在该运行时之上。

native image generator 则包含了真正的 AOT 编译逻辑。它本身是一个 Java 程序，将使用 Graal 编译器将 Java 类文件编译为可执行文件或者动态链接库。

在进行编译之前，native image generator 将采用指针分析（points-to analysis），从用户提供的程序入口出发，探索所有可达的代码。在探索的同时，它还将执行初始化代码，并在最终生成可执行文件时，将已初始化的堆保存至一个堆快照之中。这样一来，SubstrateVM 将直接从目标程序开始运行，而无须重复进行 Java 虚拟机的初始化。

SubstrateVM 主要用于 Java 虚拟机语言的 AOT 编译，例如 Java、Scala 以及 Kotlin。

Truffle 语言实现本质上就是 Java 程序，而且它所有用到的类都是编译时已知的，因此也适合在 SubstrateVM 上运行。不过，它并不会 AOT 编译用 Truffle 语言写就的程序。

SubstrateVM 的启动时间与内存开销

SubstrateVM 的启动时间和内存开销非常少。我们曾比较过用 C 和用 Java 两种语言写就的 Hello World 程序。C 程序的执行时间在 10ms 以下，内存开销在 500KB 以下。在 HotSpot 虚拟机上运行的 Java 程序则需要 40ms，内存开销为 24MB。

使用 SubstrateVM 的 Java 程序的执行时间则与 C 程序持平，内存开销在 850KB 左右。这得益于 SubstrateVM 所保存的堆快照，以及无须额外初始化，直接执行目标代码的特性。

同样，我们还比较了用 JavaScript 编写的 Hello World 程序。这里的测试对象是 Google 的 V8 以及基于 Truffle 的 Graal.js。这两个执行引擎都涉及了大量的解析代码以及执行代码，因此可以当作大型应用程序来看待。

V8 的执行效率非常高，能够与 C 程序的 Hello World 相媲美，但是它使用了约 18MB 的内存。运行在 HotSpot 虚拟机上的 Graal.js 则需要 650ms 方能执行完这段 JavaScript 的 Hello World 程序，而且内存开销在 120MB 左右。

运行在 SubstrateVM 上的 Graal.js 无论是执行时间还是内存开销都十分优越，分别为 10ms 以下以及 4.2MB。我们可以看到，它在运行时间与 V8 持平的情况下，内存开销远小于 V8。

由于 SubstrateVM 的轻量特性，它十分适合于嵌入至其他系统之中。Oracle Labs 的另一个团队便是将 Truffle 语言实现嵌入至 Oracle 数据库之中，这样就可以在数据库中运行任意语言的预储程序（stored procedure）。如果你感兴趣的话，可以搜索 Oracle Database Multilingual Engine（MLE），或者参阅这个网址。我们团队也在与 MySQL 合作，开发 MySQL MLE，详情可留意我们在今年 Oracle Code One 的[讲座](https://oracle.rainfocus.com/widget/oracle/oow18/catalogcodeone18?search=MySQL JavaScript)。

Metropolis 项目

去年 OpenJDK 推出了Metropolis 项目，他们希望可以实现“Java-on-Java”的远大目标。

我们知道，目前 HotSpot 虚拟机的绝大部分代码都是用 C++ 写的。这也造就了一个非常有趣的现象，那便是对 Java 语言本身的贡献需要精通 C++。此外，随着 HotSpot 项目日渐庞大，维护难度也逐渐上升。

由于上述种种原因，使用 Java 来开发 Java 虚拟机的呼声越来越高。Oracle 的架构师 John Rose 便提出了使用 Java 开发 Java 虚拟机的四大好处：

1. 能够完全控制编译 Java 虚拟机时所使用的优化技术；
2. 能够与 C++ 语言的更新解耦合；
3. 能够减轻开发人员以及维护人员的负担；
4. 能够以更为敏捷的方式实现 Java 的新功能。

当然，Metropolis 项目并非第一个提出 Java-on-Java 概念的项目。实际上，JikesRVM 项目和Maxine VM 项目都已用 Java 完整地实现了一套 Java 虚拟机（后者的即时编译器 C1X 便是 Graal 编译器的前身）。

然而，Java-on-Java 技术通常会干扰应用程序的垃圾回收、即时编译优化，从而严重影响 Java 虚拟机的启动性能。

举例来说，目前使用了 Graal 编译器的 HotSpot 虚拟机会在即时编译过程中生成大量的 Java 对象，这些 Java 对象同样会占据应用程序的堆空间，从而使得垃圾回收更加频繁。

另外，Graal 编译器本身也会触发即时编译，并与应用程序的即时编译竞争编译线程的 CPU 资源。这将造成应用程序从解释执行切换至即时编译生成的机器码的时间大大地增长，从而降低应用程序的启动性能。

Metropolis 项目的第一个子项目便是探索部署已 AOT 编译的 Graal 编译器的可能性。这个子项目将借助 SubstrateVM 技术，把整个 Graal 编译器 AOT 编译为机器码。

这样一来，在运行过程中，Graal 编译器不再需要被即时编译，因此也不会再占据可用于即时编译应用程序的 CPU 资源，使用 Graal 编译器的 HotSpot 虚拟机的启动性能将得到大幅度地提升。

此外，由于 SubstrateVM 编译得到的 Graal 编译器将使用独立的堆空间，因此 Graal 编译器在即时编译过程中生成的 Java 对象将不再干扰应用程序所使用的堆空间。

目前 Metropolis 项目仍处于前期验证阶段，如果你感兴趣的话，可以关注之后的发展情况。

总结与实践

今天我介绍了 GraalVM 中的 AOT 编译框架 SubstrateVM。

SubstrateVM 的设计初衷是提供一个高启动性能、低内存开销，和能够无缝衔接 C 代码的 Java 运行时。它是一个独立的运行时，拥有自己的内存管理等组件。

SubstrateVM 要求所要 AOT 编译的目标程序是封闭的，即不能动态加载其他类库等。在进行 AOT 编译时，它会探索所有可能运行到的方法，并全部纳入编译范围之内。

SubstrateVM 的启动时间和内存开销都非常少，这主要得益于在 AOT 编译时便已保存了已初始化好的堆快照，并支持从程序入口直接开始运行。作为对比，HotSpot 虚拟机在执行 main 方法前需要执行一系列的初始化操作，因此启动时间和内存开销都要远大于运行在 SubstrateVM 上的程序。

Metropolis 项目将运用 SubstrateVM 项目，逐步地将 HotSpot 虚拟机中的 C++ 代码替换成 Java 代码，从而提升 HotSpot 虚拟机的可维护性，也加快新 Java 功能的开发效率。

今天的实践环节，请你参考我们官网的SubstrateVM 教程，AOT 编译一段 Java-Kotlin 代码。

上一页 下一页