小马哥的 Java 项目实战营

云原生项目 - 第三十节 Java Native

小马哥（mercyblitz）

我是谁？

小马哥（mercyblitz）

- 父亲
- Java 劝退师
- Apache Dubbo PMC
- Spring Cloud Alibaba 架构师
- 《Spring Boot 编程思想》作者

议题

- Hotspot VM
- GraalVM
- 问答互动

Hotspot VM

- 简介

作为Java Hotspot 性能引擎发布，是一个用于桌面和服务器计算机的 Java 虚拟机，由 Oracle 公司维护和分发。它通过实时编译（just-in-time compilation）和自适应优化（adaptive optimization）等方法提高了性能。

Hotspot VM

- 历史

Java HotSpot 性能引擎于1999年4月27日发布，基于编程语言 Smalltalk 的实现，名为 Strongtalk，最初由 Longview technologies 开发，商业名称为 Animorphic 。

1997年， SUN 购买了Animorphic 在收购Animorphic后不久，SUN 决定为Java虚拟机编写一个新的实时（JIT）编译器。这个新的编译器将产生HotSpot这个名字，源于软件的行为：当它运行Java字节码时，就像运行自VM一样，HotSpot不断分析经常或重复执行的热点的程序性能。然后，这些目标是进行优化，从而以最小的开销实现高性能执行，从而获得性能要求较低的代码。最初作为Java1.2的附加组件提供，HotSpot 成为 Java1.3 中默认的 SUN JVM。

Hotspot VM

- 特性

JRE 具有两个虚拟机，一个称为客户机，另一个称为服务器。客户机版本经过了快速加载的调整。它利用解释执行。服务器版本的加载速度较慢，在生成高度优化的JIT编译时投入了更多的精力，以获得更高的性能。两个vm都只编译经常运行的方法，使用可配置的调用计数阈值来决定要编译的方法。

分层编译是Java 7中引入的一个选项，它同时使用客户机和服务器编译器，以提供比服务器编译器更快的启动时间，但具有类似或更好的峰值性能。从 Java 8 开始，分层编译是服务器VM的默认值。

Hotspot VM

- 特性

  Hotspot 提供:

  - Java Classloader
  - Java 字节码解释器
  - 客户端和服务端虚拟机
  - 垃圾回收器
  - 运行时类库

Hotspot VM

- JIT 编译

实时（JIT）编译（也称为动态翻译或运行时编译）是一种执行计算机代码的方法，它涉及在程序执行期间（运行时）而不是在执行之前进行编译。通常，这包括源代码或更常见的字节码转译为机器代码，然后直接执行。实现JIT编译器的系统通常会连续分析正在执行的代码，并识别代码中编译或重新编译所获得的加速比会超过编译代码开销的部分。

JIT编译是两种传统的机器代码翻译方法（提前编译（AOT）和解释执行）的结合，并结合了两者的一些优点和缺点。

Hotspot VM

- JIT 编译

由于加载和编译字节码所花费的时间，JIT在应用程序的初始执行中会导致轻微到明显的延迟。有时这种延迟被称为“启动时间延迟”或“预热时间”。一般来说，JIT执行的优化越多，它生成的代码就越好，但是初始延迟也会增加。因此，JIT编译器必须在编译时间和希望生成的代码质量之间进行权衡。除了JIT编译之外，启动时间还可以包括更多的IO绑定操作：例如，Java虚拟机（JVM）的rt.jar类数据文件是40mb，JVM必须在这个上下文巨大的文件中查找大量数据。

Hotspot VM

- JIT 编译

SUN Hotspot 虚拟机就是将解释和JIT编译结合起来。应用程序代码最初是被解释的，但是JVM监视哪些字节码序列经常被执行，并将它们转换成机器代码以便在硬件上直接执行。对于只执行几次的字节码，这节省了编译时间并减少了初始延迟；对于频繁执行的字节码，JIT编译用于在缓慢解释的初始阶段之后以高速运行。此外，由于程序花费大部分时间执行少数代码，因此减少的编译时间非常重要。最后，在初始代码解释期间，可以在编译之前收集执行统计信息，这有助于执行更好的优化。还有一些Java实现将AOT（提前）编译器与JIT编译器（Excelsior JET）或解释器（GNU Compiler for Java）结合起来。

Hotspot VM

- JIT 编译

SUN Hotspot 虚拟机有两种主要模式客户端和服务器。在客户机模式下，执行最小的编译和优化，以减少启动时间。在服务器模式下，执行广泛的编译和优化，以牺牲启动时间，从而在应用程序运行后最大限度地提高性能。其他Java实时编译器使用运行时测量的方法执行次数与方法的字节码大小相结合，作为决定何时编译的启发式方法。还有一种编译器使用执行次数与循环检测相结合。一般来说，在短时间运行的应用程序中准确预测要优化的方法比在长时间运行的应用程序中要困难得多。

Hotspot VM

- JIT 编译

  HotSpot 虚拟机包含多个即时编译器：

  - C1：-client，-XX:CompileThreshold 默认 1500

  - C2：-server，-XX:CompileThreshold 默认 10000

  - Graal：-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler 启用

Hotspot VM

- JIT 编译

JDK 7 引入分层编译（Tiered compiling），JVM 参数：-XX:+TieredCompilation，JDK 8 默认开启。当分层编译（Tiered compiling）激活时，-XX:CompileThreshold 将失效，采用自适应动态阈值。

在 64 位 JVM 中，默认情况下编译线程的总数目是根据处理器数量来调整的，-XX:+CICompilerCountPerCPU 默认为true，当需要指定数量时，需设置 -XX:+CICompilerCount=N。

Hotspot VM

- JIT 编译

  JVM JIT 编译层次有五（参数 -XX:TieredStopAtLevel=N，N 的取值范围 [0,4]）：

  - level 0 - interpreter
  - level 1 - C1 with full optimization (no profiling)
  - level 2 - C1 with invocation and backedge counters
  - level 3 - C1 with full profiling (level 2 + MDO)
  - level 4 - C2

Hotspot VM

- JIT 编译

  编译信息输出参数 -XX:+PrintCompilation

  161 2 3 java.lang.String::equals (81 bytes)

  161 4 n 0 java.lang.System::arraycopy (native) (static)

  162 11 % 4 java.lang.String::indexOf @ 37 (70 bytes)

  第一列：编译时间

  第二列：编译 ID

  n：native 方法

  %：OSR 编译

Hotspot VM

- JIT 编译

  编译信息输出参数 -XX:+PrintCompilation

  165 29 s 4 java.lang.StringBuffer::append (13 bytes)

  168 74 ! 3 java.util.zip.ZipFile::getEntry (101 bytes)

  180 20 3 java.lang.String::startsWith (72 bytes) made not entrant

  s：同步方法

  !：异常处理器

Hotspot VM

- JIT 编译

  编译信息输出参数 -XX:+PrintCompilation

  161 2 3 java.lang.String::equals (81 bytes)

  161 4 n 0 java.lang.System::arraycopy (native) (static)

  162 11 % 4 java.lang.String::indexOf @ 37 (70 bytes)

  165 29 s 4 java.lang.StringBuffer::append (13 bytes)

  168 74 ! 3 java.util.zip.ZipFile::getEntry (101 bytes)

  180 20 3 java.lang.String::startsWith (72 bytes) made not entrant

Hotspot VM

- AOT 编译

提前编译（AOT compilation）是在程序执行之前（通常在构建时）将一种（通常）高级编程语言编译成一种（通常）低级语言的行为，以减少在运行时需要执行的工作量。简单地说，这将中间代码编译成机器代码，以便在执行中间代码时进行本机运行，这可能会降低应用程序的性能。提前编译通过在执行之前而不是在执行期间执行，消除了对这一步骤的需要。

- 实现
  - Excelsior JET
  - GNU Compiler for Java
  - GraalVM

GraalVM

- 简介（https://www.graalvm.org/）

GraalVM 是一个高性能的 JDK 发行版，旨在加速用 Java 和其他 JVM 语言编写的应用程序的执行，同时支持JavaScript、Ruby、Python和许多其他流行语言。GraalVM 的 polyglot 功能使得在一个应用程序中混合多种编程语言成为可能，同时消除了外语调用成本。

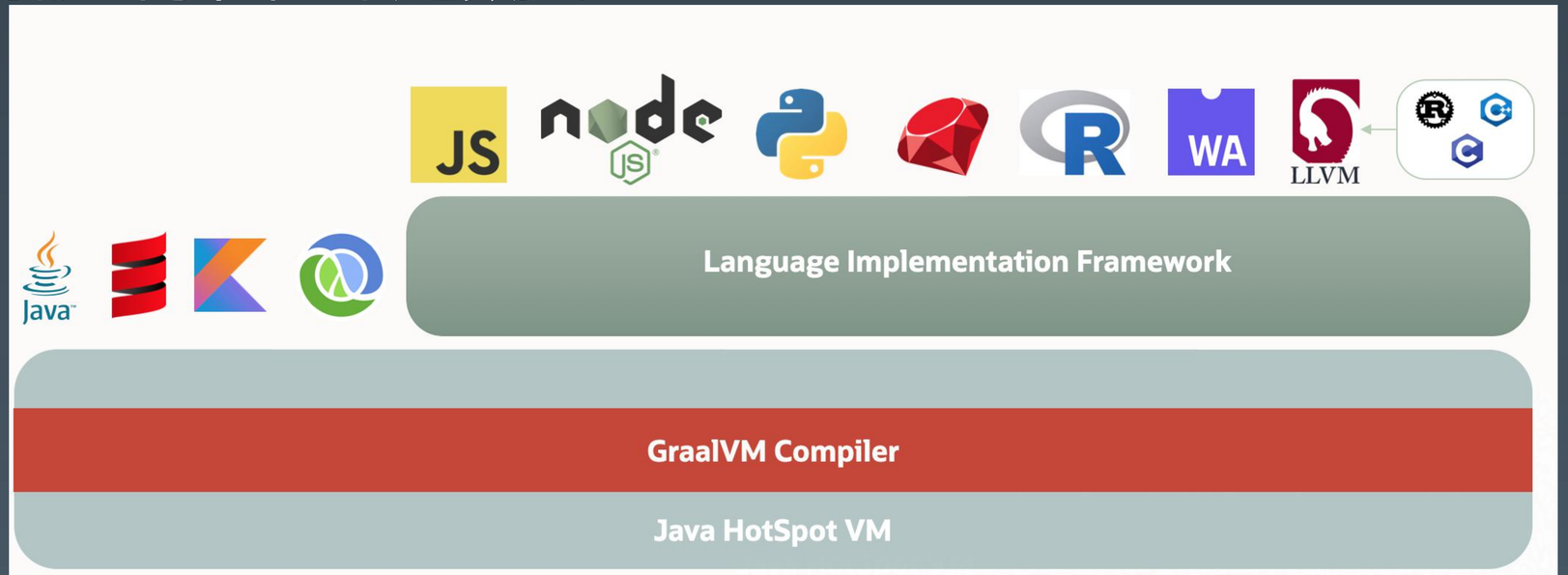
GraalVM 以 GraalVM 企业版和 GraalVM 社区版的形式提供，包括对 Java 8、Java 11和 Java 16的支持。GraalVM企业版基于 Oracle JDK，而 GraalVM 社区版基于 OpenJDK。

GraalVM 社区版是从 GitHub 上可用的源代码构建的开放源码软件，在GNU通用公共许可证的第2版下发布，但有“Classpath”例外，这与Java的术语相同。

GraalVM

- 架构

GraalVM 向 HotSpot Java 虚拟机添加了一个用 Java 编写的高级实时（JIT）优化编译器。GraalVM 的 Truffle 语言实现框架还可以在JVM上运行 JavaScript、Ruby、Python 和许多其他流行语言。通过 GraalVM Truffle，Java和其他受支持的语言可以直接互操作，并在同一内存空间中来回传递数据。



GraalVM

- 运行时模式
  - JVM Runtime：GraalVM默认使用GraalVM编译器作为顶级JIT编译器。在运行时，应用程序通常在JVM上加载和执行

  - Native Image：将Java代码编译成独立的二进制可执行文件或本机共享库。在本机映像构建期间处理的Java字节码包括所有应用程序类、依赖项、第三方依赖库以及所需的任何JDK类。

  - Java on Truffle：一个完整的 JVM，包括所有核心组件，实现与Java运行时环境库相同的API，并重用GraalVM中的所有jar和本机库。

GraalVM

- 核心组件
  - 运行时
    - Java HotSpot VM
    - JavaScript runtime
    - LLVM runtime
  - 类库
    - GraalVM compiler
    - Polyglot API

GraalVM

- 核心组件
  - 工具
    - JavaScript REPL with the JavaScript interpreter
    - lli tool to directly execute programs from LLVM bitcode
    - GraalVM Updater to install additional functionalities

GraalVM

- 特性支持

GraalVM技术以生产可用（支持）和实验性的形式分发：

Feature	Linux AMD64	Linux ARM64	MacOS	Windows
Native Image	supported	experimental	supported	supported
LLVM runtime	supported	experimental	supported	not available
LLVM toolchain	supported	experimental	supported	not available
JavaScript	supported	experimental	supported	supported
Node.js	supported	experimental	supported	supported
Java on Truffle	experimental	not available	experimental	experimental
Python	experimental	not available	experimental	not available
Ruby	experimental	not available	experimental	not available
R	experimental	not available	experimental	not available
WebAssembly	experimental	experimental	experimental	experimental

GraalVM

- 核心组件
  - 工具
    - JavaScript REPL with the JavaScript interpreter
    - lli tool to directly execute programs from LLVM bitcode
    - GraalVM Updater to install additional functionalities

THANKS!

极客大学