链滴

JVM | 运行时数据区的 JVM 规范

作者：xiaodaojava
原文链接：https://ld246.com/article/1580464922557
来源网站：链滴

本文知识点

- JVM 虚拟机制定的规范
- 方法区，永久代，元空间的区别

参考文档

- jvm 官方文档 https://docs.oracle.com/javase/specs/jvms/se11/jvms11.pdf 2.5 Run-Time Data Areas
- 《深入理解 Java 虚拟机-jvm 高级特性与最佳实践》

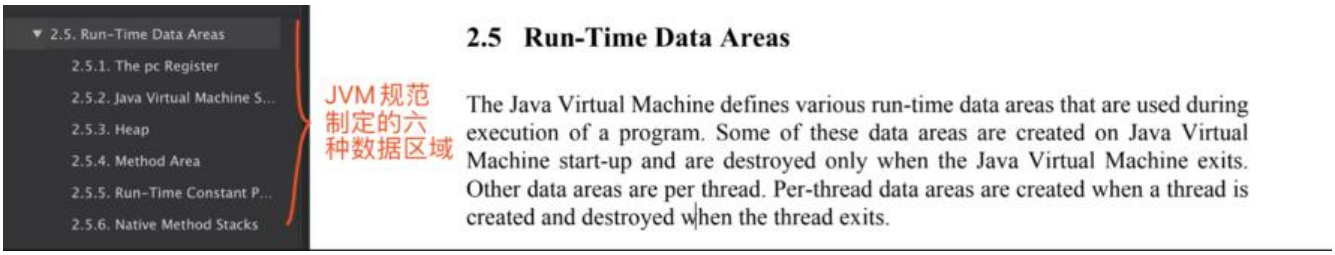
总述

在这一块的学习时， 我们容易陷入一个误区，就是一上来就直接搜索运行时数据区， 网上有些文章虚拟机规范和 HOTSPOT 实现没有区分开，导致有时候大家看的两篇文章解释尽不一样。 自己也容糊涂。 所以本篇特地将两个拆开讲。 且尽量以官方文档为准

我们可以把 jvm 规范理解成接口。 就是要这些东西， 然后不同的虚拟机厂商有不同的实现方案。 如法区，hotspot 用了 1.7 及以前用了永久代， 1.8 及以后用了元数据区。 别的虚拟机如 JRocket,J9 没有永久代的概念。

JVM 运行时数据区制定的虚拟机规范

如上参考 pdf 中及下图所示，主要有六大数据区域：



The pc Register | program Counter Register | 程序计器

程序计数器为线程私有的，每个线程都有自己独立的程序计数器

如果当前线程执行的是 Java 方法，则程序计数器中是当前执行虚所机字节码指令的地址，如果正在行是 native 方法，这个计数器的值是空的

我们假设有以下场景。 此时有两个线程 A,B 正在执行。

CPU 执行线程 A 的 Ia1 指令时， Ta 的程序计数器存的是 Ia1 指令的地址， 执行完指令 Ia1 后，转执行线程 B 的 Ib1 指令， 再回到线程 A 时， 从程序计数器中取出上次执行到了 Ia1, 然后继续往下行。

Java Virtual Machine Stacks | Java 虚拟机栈

Java 虚拟机栈也是线程私有的， 该线程每调用一个方法，都用创建一个栈帧(Frame).栈帧中有局部量表，操作数栈，动态链接，方法出口等信息。

开发中遇到和虚拟机栈相关的问题：

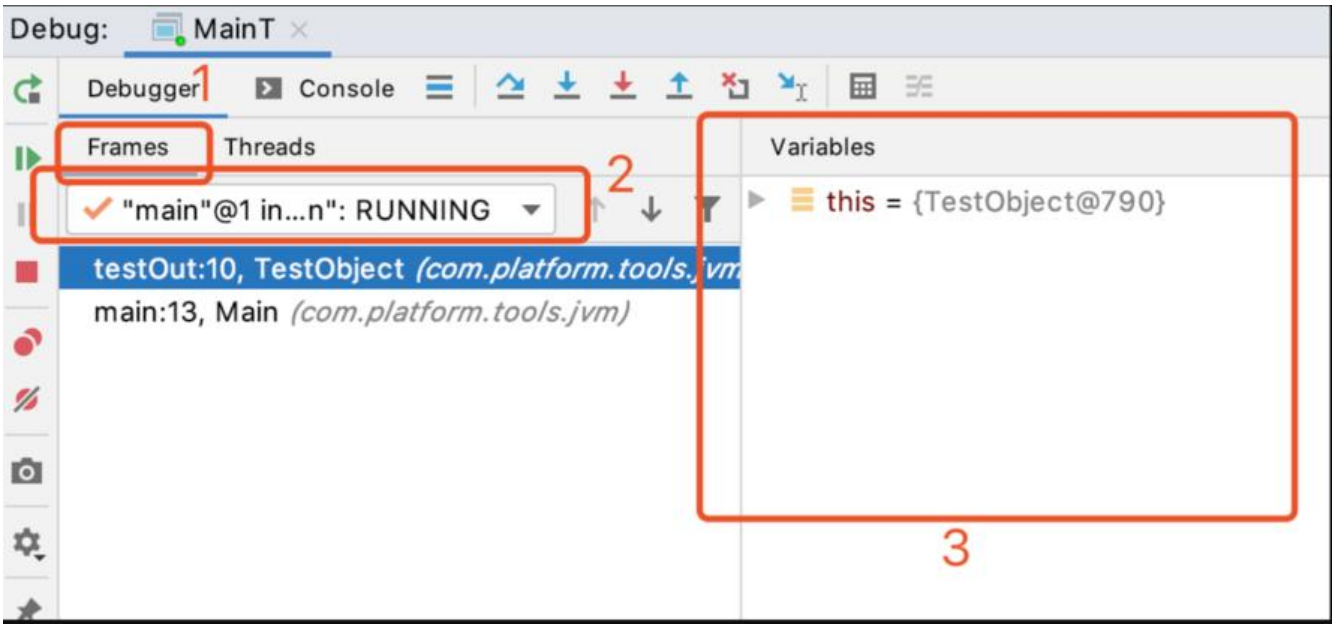
1. i++ 线程不安全

我们常说的 i++ 线程不安全问题， 其根本就在于栈帧中的局部变量表，操作数栈这两个结构。

2. 递归太多 StackOverFlow

递归就是自己调用自己，每调用一次，就是创建一个栈帧添加到虚拟机栈中， 添加的多了，超过了容，就会报如上 StackOverFlow 的错误

我们在 idea 的 debug 界面也可以看到关于栈和栈帧相关界面，如下图所示：



1:栈帧列表

2:可以切换不同的线程，看对应的栈帧

3:当前栈帧中的用到的变量

Heap 堆

所以线程共享的一块区域，几乎所以有 Java 对象都在堆里面进行分配，这里要注意以下几个问题

1. 并不是所有的对象都在堆中分配

这是一个很容易被忽略的点，jdk1.8 之后，虚拟机默认开启子逃逸分析，如果变量 A 只在本方法中用，则可以不在堆中分其分配，可以在栈中为其分配。 这样随着方法调用结束，栈帧销毁，对角也跟销毁，就不用调用 GC 了

2. 虚拟机规范并没有对堆进行分代划分

如我们现在常说的年轻代，老年代等是 HotSpot 的实现， JVM 规范只是制定了堆，没有制定分代的准。

Run-Time Constant Pool | 运行时常量池

运行时常量池是方法区的一部分，与之对应是。class 文件中的静态常量信息，如下图所示：

```
Terminal:  Local  +
$ javap -v Main
Warning: File ./Main.class does not contain class Main
Classfile /Users/lixiang/360jk/code/shenzhen/tools/tools-java/out/production/classes/com/platform/tools/jvm/Ma
  Last modified Jan 31, 2020; size 747 bytes
  MD5 checksum c002b239e1948aa1ba7299d53697b422
  Compiled from "Main.java"
public class com.platform.tools.jvm.Main
  minor version: 0
  major version: 52
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #7                          // com/platform/tools/jvm/Main
  super_class: #8                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:   .class文件中的静态常量池
   #1 = Methodref          #8.#26         // java/lang/Object."<init>":()V
   #2 = Class              #27            // com/platform/tools/jvm/TestObject
   #3 = Methodref          #2.#26         // com/platform/tools/jvm/TestObject."<init>":()V
   #4 = Methodref          #2.#28         // com/platform/tools/jvm/TestObject.testOut:()Ljava/lang/String;
   #5 = Fieldref           #29.#30        // java/lang/System.out:Ljava/io/PrintStream;
   #6 = Methodref          #31.#32        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #7 = Class              #33            // com/platform/tools/jvm/Main
```

在 class 文件加载的链接步骤中的解析阶段，会把静态的常量池和运行时常量池关联起来，把符号引变成直接引用。

Method Area | 方法区

方法区也是被线程所共享的，其实是从堆里面划出来的一片区域(这里不要钻是从哪个代里面划出来的 如上据说，JVM 规范并没有规定分代的，由各个实际的虚机机去实现的，可自己去看怎么划分)

里面存放的有：已被虚拟机加载的类信息， 常量，静态变量，即时编译器编译后的代码缓存

Native Method Stacks | 本地方法栈

这个和上面的 Java 虚拟机栈没太大的差别， 在 jvm 规范层面，把本地方法栈描述为在 Java 调用其语言写的方法时创建，在 HotSpot 实现层面， 直接把本地方法栈和虚拟机栈合二为一。 (所以说规和实现要分开学习)

方法区，永久代，元空间(MetaSpace)的区别

总的来说， 方法区是接口， 永久代和元空间是实现

在 HotSpot 中，1.7 及以前的版本以永久代做为方法区的实现， 1.8 及以后版本的 jdk 以 MetaSpace 做方法区的实现。

永久代在堆里面， MetaSpace 直接使用了直接(本地)内存。

相应的 1.8 及以后。 移除了永久代，以 MetaSpace 做方法区实现，常量池中的字符串常量池，直接在了堆中。 其他的如类信息，静态信息留在了 MetaSpace 中，也跟着去了直接(本地)内存

总结

本文以 JVM 规范为主，部分区域给出了 HotSpot 的实现，先学规范，再学实现，两者切记一定要分学，不然就学着学着就混乱了！

原文链接：JVM | 运行时数据区的 JVM 规范