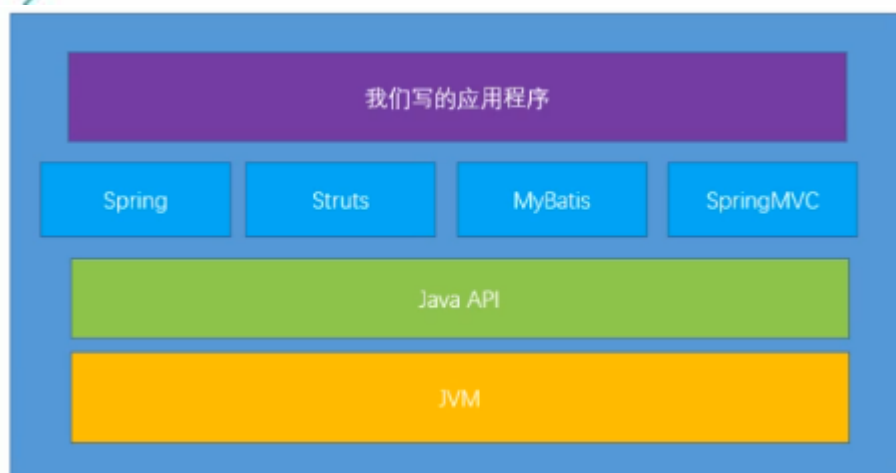


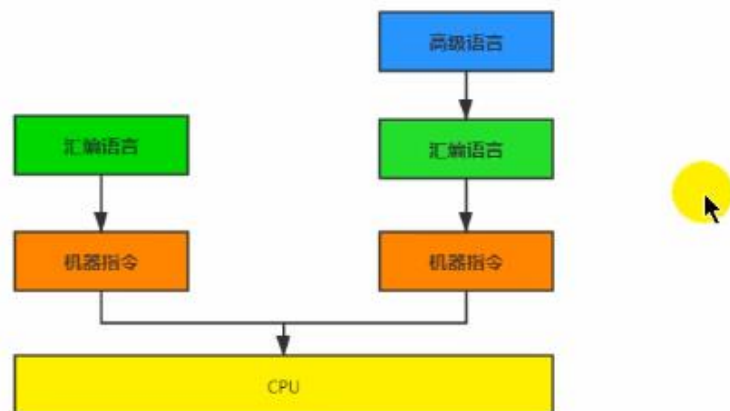
名称	修改
尚硅谷_宋红康_01_JVM与Java体系结构.pptx	2020
尚硅谷_宋红康_02_添加子系统.pptx	2020
尚硅谷_宋红康_03_运行时数据区概述及编程.pptx	2020
尚硅谷_宋红康_04_程序计数器.pptx	2020
尚硅谷_宋红康_05_虚拟机栈.pptx	2020
尚硅谷_宋红康_06_本地方法接口.pptx	2020
尚硅谷_宋红康_07_本地方法栈.pptx	2020
尚硅谷_宋红康_08_堆.pptx	2020
尚硅谷_宋红康_09_方法区.pptx	2020
尚硅谷_宋红康_10_直接内存.pptx	2020
尚硅谷_宋红康_11_执行引擎.pptx	2020
尚硅谷_宋红康_12_StringTable.pptx	2020
尚硅谷_宋红康_13_垃圾回收概述.pptx	2020
尚硅谷_宋红康_14_垃圾回收相关算法.pptx	2020
尚硅谷_宋红康_15_垃圾回收相关概念.pptx	2020
尚硅谷_宋红康_16_垃圾回收器.pptx	2020

6	JVM的整体结构
7	Java代码执行流程
8	JVM的架构模型
9	JVM的生命周期
X	JVM的发展历程



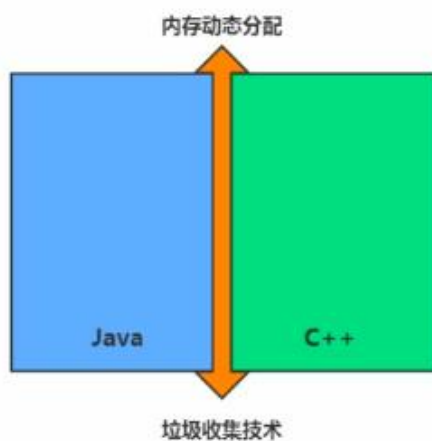
开发人员如何看待上层框架

- 一些有一定工作经验的开发人员，打心眼儿里觉得SSM、微服务等上层技术才是重点，基础技术并不重要，这其实是一种本末倒置的“病态”。
- 如果我们把核心类库的 API 比做数学公式的话，那么 Java 虚拟机的知识就好比公式的推导过程。



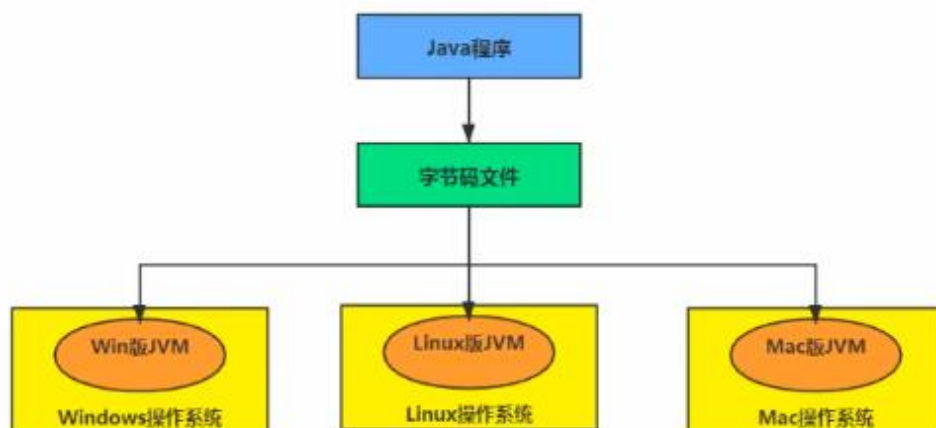
计算机系统体系对我们来说越来越远，在不了解底层实现方式的前提下，通过高级语言很容易编写程序代码。但事实上计算机并不认识高级语言

垃圾回收算法、JIT、底层原理

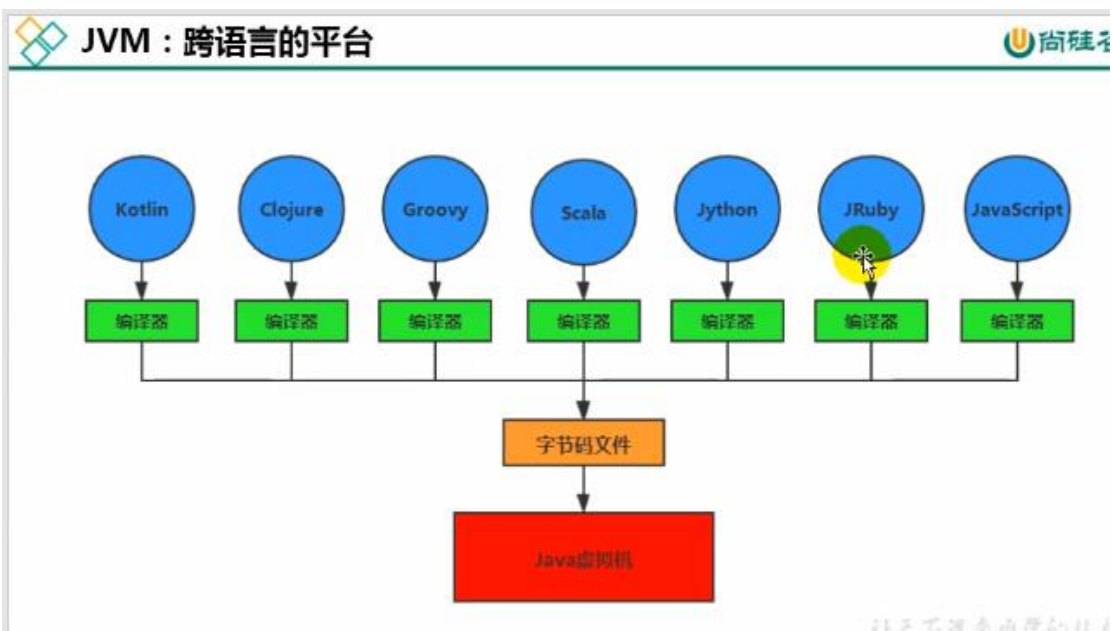


垃圾收集机制为我们打理了很多繁琐的工作，大大提高了开发的效率，但是，垃圾收集也不是万能的，懂得JVM内部的内存结构、工作机制，是设计高扩展性应用和诊断运行时问题的基础，也是Java工程师进阶的必备能力。

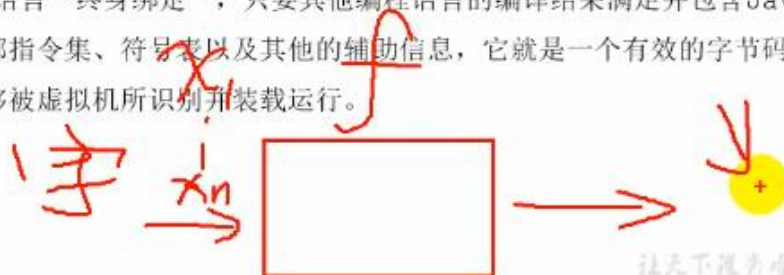




"write once, run anywhere."



- 随着Java7的正式发布，Java虚拟机的设计者们通过JSR-292规范基本实现在Java虚拟机平台上运行非Java语言编写的程序。
- Java虚拟机根本不关心运行在其内部的程序到底使用何种编程语言编写的，它只关心“字节码”文件。也就是说Java虚拟机拥有语言无关性，并不会单纯地与Java语言“终身绑定”，只要其他编程语言的编译结果满足并包含Java虚拟机的内部指令集、符号表以及其他的辅助信息，它就是一个有效的字节码文件，就能够被虚拟机所识别并装载运行。



- 我们平时说的java字节码，指的是用java语言编译成的字节码。准确的说任何能在jvm平台上执行的字节码格式都是一样的。所以应该统称为：**jvm字节码**。
- 不同的编译器，可以编译出相同的字节码文件，字节码文件也可以在不同的JVM上运行。
- Java 虚拟机与 Java 语言并没有必然的联系，它只与特定的二进制文件格式—Class文件格式所关联，Class 文件中包含了 Java 虚拟机指令集（或者称为字节码、Bytecodes）和符号表，还有一些其他辅助信息。

多语言混合编程

Java平台上的多语言混合编程正成为主流，通过特定领域的语言去解决特定领域的问题是当前软件开发应对日趋复杂的项目需求的一个方向。

试想一下，在一个项目之中，并行处理用Clojure语言编写，展示层使用JRuby/Rails，中间层则是Java，每个应用层都将使用不同的编程语言来完成，而且，接口对每一层的开发者都是透明的，各种语言之间的交互不存在任何困难，就像使用自己语言的原生API一样方便，因为它们最终都运行在一个虚拟机之上。

对这些运行于Java虚拟机之上、Java之外的语言，来自系统级的、底层的支持正在迅速增强，以JSR-292为核心的一系列项目和功能改进（如Da Vinci Machine项目、Nashorn引擎、InvokeDynamic指令、java.lang.invoke包等），推动Java虚拟机从“Java语言的虚拟机”向“多语言虚拟机”的方向发展。

- ✓ 1990年，在 Sun 计算机公司中，由 Patrick Naughton、Mike Sheridan 及 James Gosling 领导的小组Green Team，开发出的新的程序语言，命名为Oak，后期命名为Java
- ✓ 1995年，Sun正式发布Java和HotJava产品，Java首次公开亮相。
- ✓ 1996年1月23日Sun Microsystems发布了JDK 1.0。
- ✓ 1998年，JDK 1.2版本发布。同时，Sun发布了 JSP/Servlet、EJB规范，以及将Java分成了 J2EE、J2SE和J2ME。这表明了 Java开始向企业、桌面应用和移动设备应用3大领域挺进。
- ✓ 2000年，JDK 1.3发布，Java HotSpot Virtual Machine正式发布，成为Java的默认虚拟机。
- ✓ 2002年，JDK 1.4发布，古老的Classic虚拟机退出历史舞台。
- ✓ 2003年年底，Java平台的Scala正式发布，同年Groovy也加入了 Java阵营。
- ✓ 2004年，JDK 1.5发布。同时JDK 1.5改名为JavaSE 5.0。
- ✓ 2006年，JDK 6发布。同年，Java开源并建立了 OpenJDK。顺理成章，Hotspot虚拟机也成为了 OpenJDK中的默认虚拟机。

-



- 大名鼎鼎的Visual Box, VMware就属于系统虚拟机, 它们完全是对物理计算机的仿真, 提供了一个可运行完整操作系统的软件平台。
- 程序虚拟机的典型代表就是Java虚拟机, 它专门为执行单个计算机程序而设计, 在Java虚拟机中执行的指令我们称为Java字节码指令。

还是程序虚拟机，在上面

Java虚拟机

- Java虚拟机是一台执行Java字节码的虚拟计算机，它拥有独立的运行机制，其运行的Java字节码也未必由Java语言编译而成。
- JVM平台的各种语言可以共享Java虚拟机带来的跨平台性、优秀的垃圾回收器，以及可靠的即时编译器。
- **Java技术的核心就是Java虚拟机**（JVM，Java Virtual Machine），因为所有的Java程序都运行在Java虚拟机内部。



让天下没有难学的

• 作用

Java虚拟机就是二进制字节码的运行环境，负责装载字节码到其内部，解释/编译为对应平台上的机器指令执行。每一条Java指令，Java虚拟机规范中都有详细定义，如怎么取操作数，怎么处理操作数，处理结果放在哪里。

• 特点

- 一次编译，到处运行
- 自动内存管理
- 自动垃圾回收功能

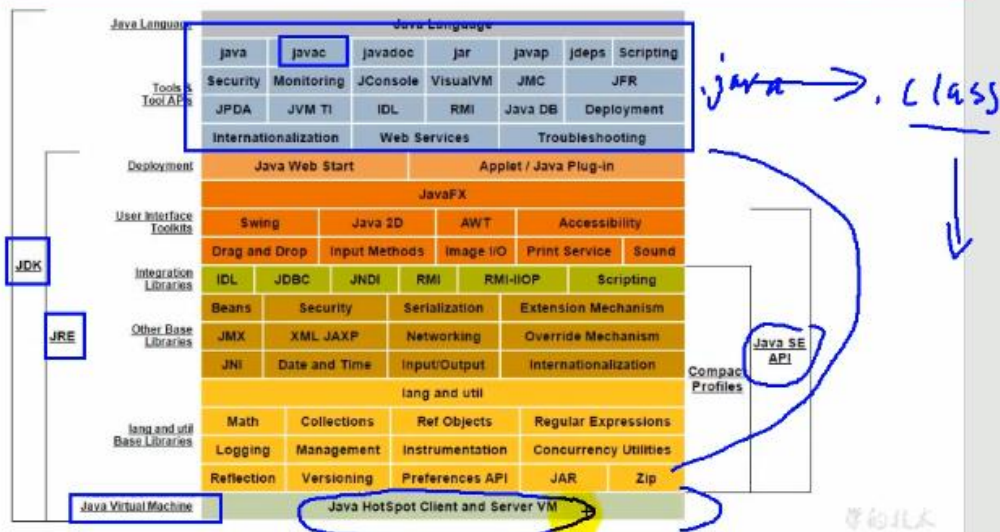


JVM的位置

尚硅谷



JVM是运行在操作系统之上的，它与硬件没有直接的交互。

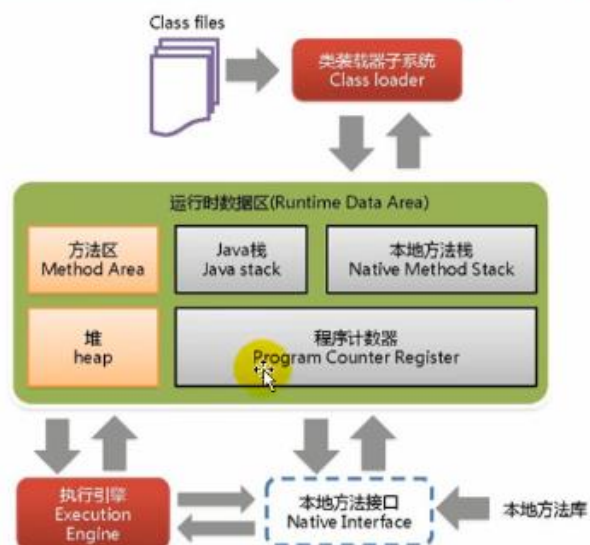


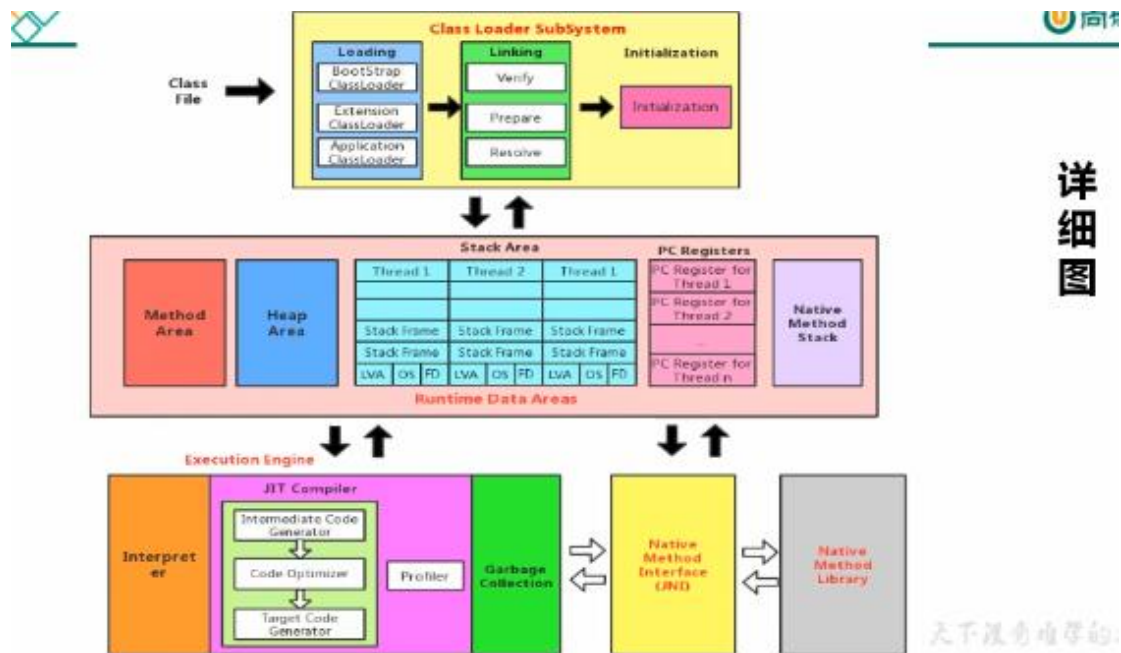
Google的Android系统结构



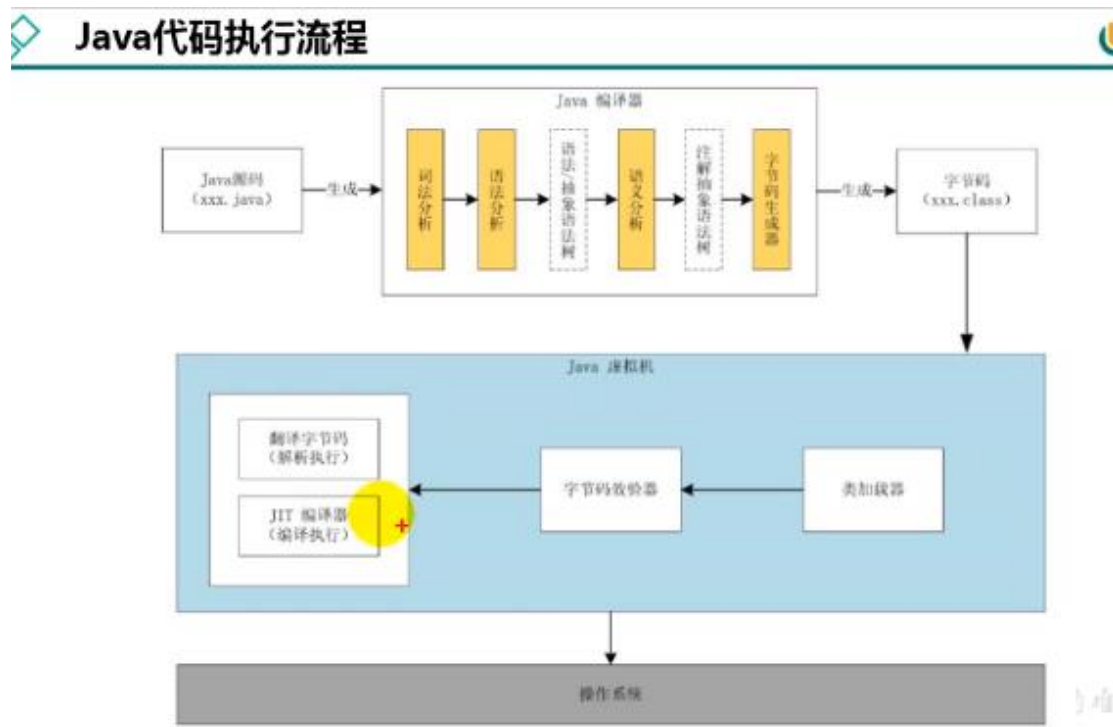
JVM的整体结构

- HotSpot VM是目前市面上高性能虚拟机的代表作之一。
- 它采用解释器与即时编译器并存的架构。
- 在今天，Java程序的运行性能早已脱胎换骨，已经达到了可以和C/C++程序一较高下的地步。





The JIT compiler is the rear-end compiler



bytecode translation makes sure the translating will be completed on time, so as soon as it gets bytecode, it will start translating line by line .

JIT is the second translation of the whole program. It is supposed to translate the bytecode to machine-code, and the next step is to save the most frequent machine-


code in the method area so it can be called quickly next time。

JVM的架构模型

Java编译器输入的指令流基本上是一种基于**栈的指令集架构**，另外一种指令集架构则是基于**寄存器的指令集架构**。

具体来说：这两种架构之间的区别：

- **基于栈式架构的特点**
 - 设计和实现更简单，适用于资源受限的系统；
 - 避开了寄存器的分配难题：使用零地址指令方式分配。
 - 指令流中的指令大部分是零地址指令，其执行过程依赖于操作栈。指令集更小，编译器容易实现。
 - 不需要硬件支持，可移植性更好，更好实现跨平台
- **基于寄存器架构的特点**
 - 典型的应用是x86的二进制指令集：比如传统的PC以及Android的Davlik虚拟机。
 - 指令集架构则完全依赖硬件，可移植性差
 - 性能优秀和执行更高效；
 - 花费更少的指令去完成一项操作。
 - 在大部分情况下，基于寄存器架构的指令集往往都以一地址指令、二地址指令和三地址指令为主，而基于栈式架构的指令集却是以零地址指令为主。



zero-address means zero address and one operate number.

To complete the same mission, Stack 指令集小但是指令多；register 指令少

What't more,stack is 8 bites,while register is 16 bites

举例1:

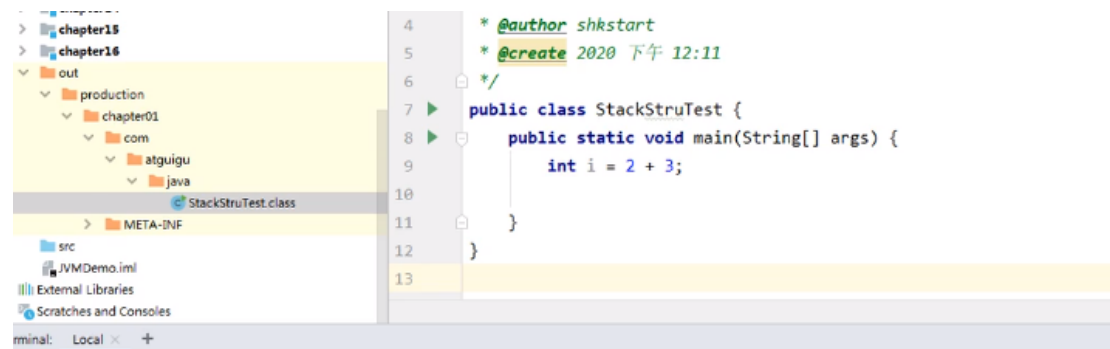
同样执行2+3这种逻辑操作，其指令分别如下：

基于栈的计算流程（以Java虚拟机为例）：

```
1  iconst_2 //常量2入栈
2  istore_1
3  iconst_3 //常量3入栈
4  istore_2
5  iload_1
6  iload_2
7  iadd //常量2、3出栈，执行相加
8  istore_0 //结果5入栈
```

而基于寄存器的计算流程：

```
1  mov eax,2 //将eax寄存器的值设为1
2  add eax,3 //使eax寄存器的值加3
```

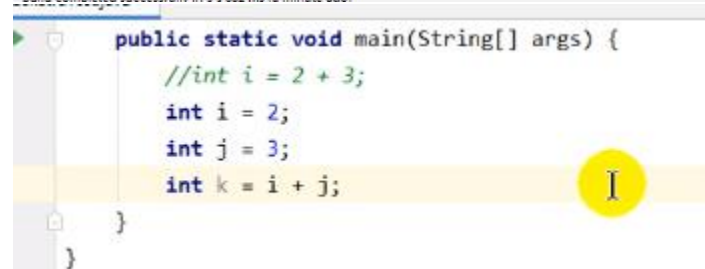
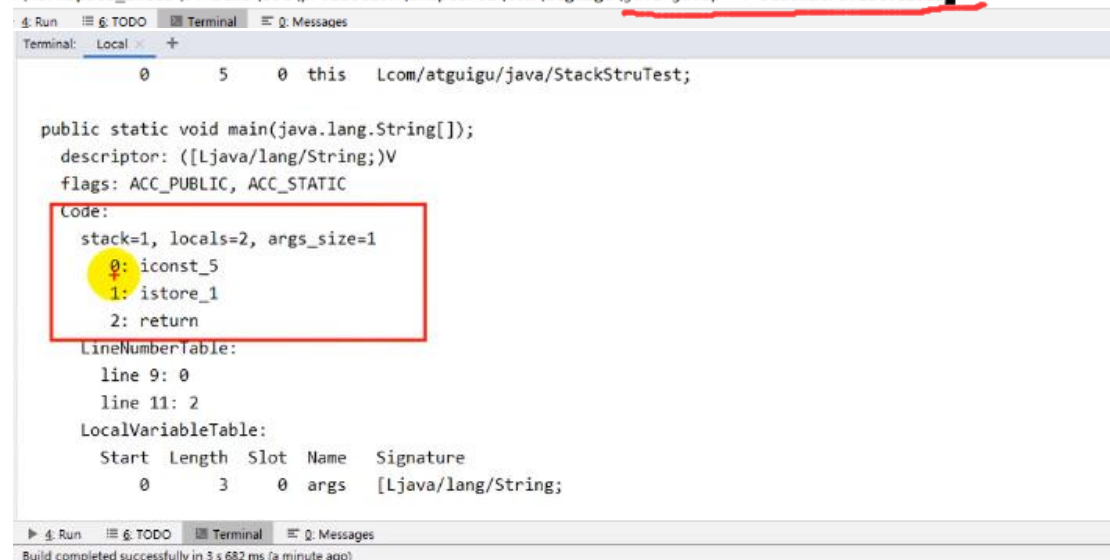


: \workspace_idea5\JVMDemo\out>cd..

: \workspace_idea5\JVMDemo>cd out/production/chapter01

: \workspace_idea5\JVMDemo\out\production\chapter01>cd com/atguigu/java

: \workspace_idea5\JVMDemo\out\production\chapter01\com\atguigu\java>javap -v StackStruTest.class



```

public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=4, args_size=1
      0: iconst_2
      1: istore_1
      2: iconst_3
      3: istore_2+
      4: iload_1
      5: iload_2
      6: iadd
      7: istore_3
      8: return

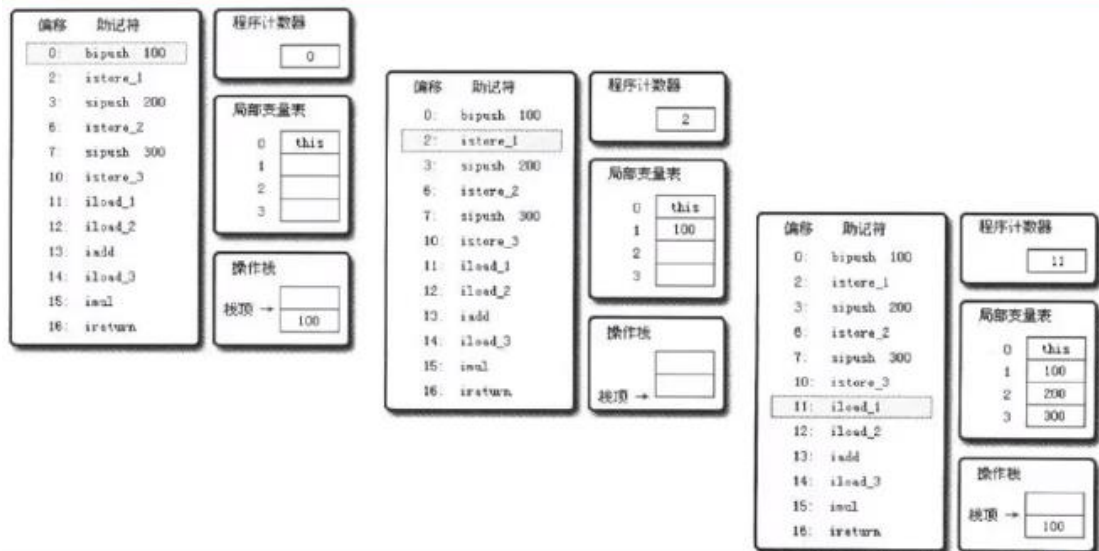
```

LineNumberTable:

```

line 10: 0
line 11: 2
line 12: 4

```



举例2:

```

public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}

```

```

public int calc();
  Code:
    Stack=2, Locals=4, Args_size=1
    0: bipush 100
    2: istore_1
    3: sipush 200
    6: istore_2
    7: sipush 300
    10: istore_3
    11: iload_1
    12: iload_2
    13: iadd
    14: iload_3
    15: imul
    16: ireturn

```

总结:

由于跨平台性的设计, Java的指令都是根据栈来设计的。不同平台CPU架构不同, 所以不能设计为基于寄存器的。优点是跨平台, 指令集小, 编译器容易实现, 缺点是性能下降, 实现同样的功能需要更多的指令。

时至今日, 尽管嵌入式平台已经不是Java程序的主流运行平台了(准确来说应该是HotSpotVM 的宿主环境已经不局限于嵌入式平台了), 那么为什么不将架构更换为基于寄存器的架构呢?

栈:

跨平台性、指令集小、指令多; 执行性能比寄存器差

9- JVM的生命周期

虚拟机的启动

Java虚拟机的启动是通过引导类加载器(bootstrap class loader)创建一个初始类(initial class)来完成的, 这个类是由虚拟机的具体实现指定的。

虚拟机的执行

- 一个运行中的Java虚拟机有着一个清晰的任务: 执行Java程序。
- 程序开始执行时他才运行, 程序结束时他就停止。
- 执行一个所谓的Java程序的时候, 真真正正在执行的是一个叫做Java虚拟机的进程。

虚拟机的生命周期



虚拟机的退出

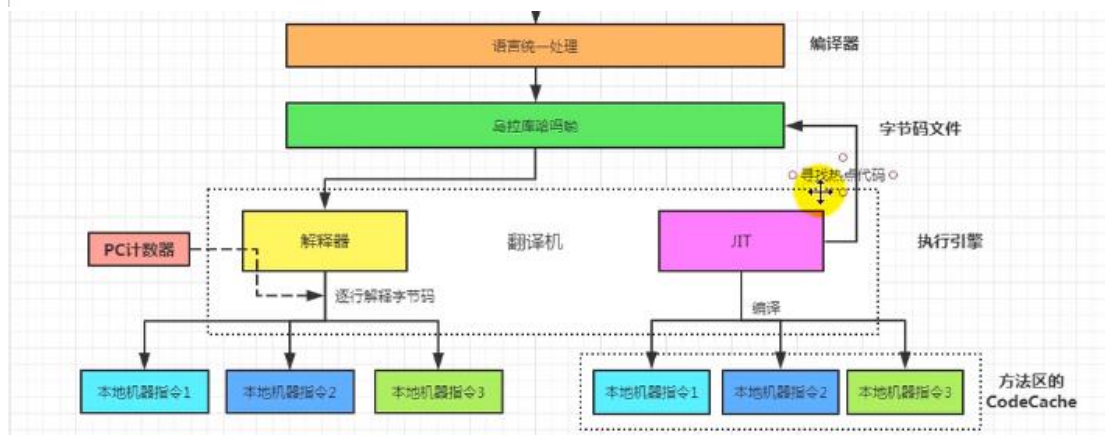
有如下的几种情况:

- 程序正常执行结束
- 程序在执行过程中遇到了异常或错误而异常终止
- 由于操作系统出现错误而导致Java虚拟机进程终止
- 某线程调用Runtime类或System类的exit方法, 或 Runtime类的halt方法, 并且Java安全管理器也允许这次exit或halt操作。
- 除此之外, JNI (Java Native Interface) 规范描述了用JNI Invocation API来加载或卸载 Java虚拟机时, Java虚拟机的退出情况。



Sun Classic VM

- 早在1996年Java1.0版本的时候，Sun公司发布了一款名为Sun Classic VM的Java虚拟机，它同时也是**世界上第一款商用Java虚拟机**，JDK1.4时完全被淘汰。
- 这款虚拟机内部只提供解释器。
- 如果使用JIT编译器，就需要进行外挂。但是一旦使用了JIT编译器，JIT就会接管虚拟机的执行系统。解释器就不再工作。解释器和编译器不能配合工作。
- 现在hotspot内置了此虚拟机。



classical VM only has interpreter but no JIT, although it does work without JIT, the efficiency is really low, so finally we have JIT.

If there is only interpreter, we will have to wait every start of a segment, so now we use the interpreter in combination with JIT, and it is like, bus\work\bus\work...

Exact VM

- 为了解决上一个虚拟机问题，jdk1.2时，sun提供了此虚拟机。
- Exact Memory Management: 准确式内存管理
 - 也可以叫Non-Conservative/Accurate Memory Management
 - 虚拟机可以知道内存中某个位置的数据具体是什么类型。I
- 具备现代高性能虚拟机的雏形
 - 热点探测
 - 编译器与解释器混合工作模式
- 只在Solaris平台短暂使用，其他平台上还是classic vm
 - 英雄气短，终被Hotspot虚拟机替换



JVM发展历程



尚硅谷

SUN公司的 HotSpot VM

- HotSpot历史
 - 最初由一家名为“Longview Technologies”的小公司设计
 - 1997年，此公司被Sun收购；2009年，Sun公司被甲骨文收购。
 - JDK1.3时，HotSpot VM成为默认虚拟机
- 目前Hotspot占有绝对的市场地位，称霸武林。
 - 不管是现在仍在广泛使用的JDK6，还是使用比例较多的JDK8中，默认的虚拟机都是HotSpot
 - Sun/Oracle JDK 和 OpenJDK的默认虚拟机
 - 因此本课程中默认介绍的虚拟机都是HotSpot，相关机制也主要是指HotSpot的GC机制。（比如其他两个商用虚拟机都没有方法区的概念）
- 从服务器、桌面到移动端、嵌入式都有应用。
- 名称中的HotSpot指的就是它的热点代码探测技术。
 - 通过计数器找到最具编译价值代码，触发即时编译或栈上替换
 - 通过编译器与解释器协同工作，在最优化的程序响应时间与最佳执行性能中取得平衡



JVM发展历程



尚硅谷

BEA 的 JRockit

- 专注于服务器端应用 I
 - 它可以不太关注程序启动速度，因此JRockit内部不包含解析器实现，全部代码都靠即时编译器编译后执行。
- 大量的行业基准测试显示，JRockit JVM是世界上最快的JVM。
 - 使用JRockit产品，客户已经体验到了显著的性能提高（一些超过了70%）和硬件成本的减少（达50%）。
- 优势：全面的Java运行时解决方案组合
 - JRockit面向延迟敏感型应用的解决方案JRockit Real Time提供以毫秒或微秒级的JVM响应时间，适合财务、军事指挥、电信网络的需要
 - MissionControl服务套件，它是一组以极低的开销来监控、管理和分析生产环境中的应用程序的工具。
- 2008年，BEA被Oracle收购。
- Oracle表达了整合两大优秀虚拟机的工作，大致在JDK 8中完成。整合的方式是在HotSpot的基础上，移植JRockit的优秀特性。
- 高斯林：目前就职于谷歌，研究人工智能和水下机器人

让天下没有难学的技术

MissionControl is used to monitor memory leaks,SUN company had tried a lot to combine the missionControl with JDK,cause these two parts have so many differences,the combination is really difficult.

IBM 的 J9

- 全称: IBM Technology for Java Virtual Machine, 简称IT4J, 内部代号: J9
- 市场定位与HotSpot接近, 服务器端、桌面应用、嵌入式等多用途VM
- 广泛用于IBM的各种Java产品。
- 目前, 有影响力的三大商用虚拟机之一, 也号称是世界上最快的Java虚拟机。
- 2017年左右, IBM发布了开源J9 VM, 命名为OpenJ9, 交给Eclipse基金会管理, 也称为 Eclipse OpenJ9

KVM和CDC/CLDC Hotspot

- Oracle在Java ME产品线上的两款虚拟机为: CDC/CLDC HotSpot Implementation VM
- KVM (Kilobyte) 是CLDC-HI早期产品
- 目前移动领域地位尴尬, 智能手机被Android和iOS二分天下。
- KVM简单、轻量、高度可移植, 面向更低端的设备上还维持自己的一片市场
 - 智能控制器、传感器
 - 老人手机、经济欠发达地区的功能手机
- 所有的虚拟机的原则: 一次编译, 到处运行。

Azul VM

- 前面三大“高性能Java虚拟机”使用在通用硬件平台上
- 这里Azul VM和BEA Liquid VM是与特定硬件平台绑定、软硬件配合的专有虚拟机
 - 高性能Java虚拟机中的战斗机。
- Azul VM是Azul Systems公司在HotSpot基础上进行大量改进, 运行于Azul Systems公司的专有硬件Vega系统上的Java虚拟机。
- 每个Azul VM实例都可以管理至少数十个CPU和数百GB内存的硬件资源, 并提供在巨大内存范围内实现可控的GC时间的垃圾收集器、专有硬件优化的线程调度等优秀特性。
- 2010年, Azul Systems公司开始从硬件转向软件, 发布了自己的Zing JVM, 可以在通用x86平台上提供接近于Vega系统的特性。

Liquid VM

- 高性能Java虚拟机中的战斗机。
- BEA公司开发的，直接运行在自家Hypervisor系统上
- Liquid VM即是现在的JRockit VE (Virtual Edition), **Liquid VM不需要操作系统的支持，或者说它自己本身实现了一个专用操作系统的必要功能，如线程调度、文件系统、网络支持等。**
- 随着JRockit虚拟机终止开发，Liquid VM项目也停止了。

Apache Harmony

- Apache也曾经推出过与JDK 1.5和JDK 1.6兼容的Java运行平台 Apache Harmony。
- 它是IBM和Intel联合开发的开源JVM，受到同样开源的OpenJDK的压制，Sun坚决不让Harmony获得JCP认证，最终于2011年退役，IBM转而参与OpenJDK
- 虽然目前并没有Apache Harmony被大规模商用的案例，但是它的Java类库代码吸纳进了Android SDK。
- JCP

Microsoft JVM

- 微软为了在IE3浏览器中支持Java Applets，开发了Microsoft JVM。
- 只能在window平台下运行。但确是当时Windows下性能最好的Java VM。
- 1997年，Sun以侵犯商标、不正当竞争罪名指控微软成功，赔了sun很多钱。微软在WindowsXP SP3中抹掉了其VM。现在windows上安装的jdk都是HotSpot。

TaobaoJVM

- 由AliJVM 团队发布。阿里，国内使用Java最强大的公司，覆盖云计算、金融、物流、电商等众多领域，需要解决高并发、高可用、分布式的复合问题。有大量的开源产品。
- **基于OpenJDK 开发了自己的定制版本AlibabaJDK，简称AJDK。是整个阿里Java体系的基石。**
- 基于OpenJDK HotSpot VM 发布的国内第一个优化、**深度定制且开源的高性能服务器版Java虚拟机。**
 - 创新的GCIH (GC invisible heap) 技术实现了off-heap，**即将生命周期较长的Java对象从heap中移到heap之外，并且GC不能管理GCIH内部的Java 对象，以此达到降低GC 的回收频率和提升GC 的回收效率的目的。**
 - GCIH 中的**对象还能够**在多个Java 虚拟机进程中实现共享
 - 使用crc32 指令实现 JVM intrinsic 降低JNI 的调用开销
 - PMU hardware 的Java profiling tool 和诊断协助功能
 - 针对大数据场景的ZenGC
- taobao vm应用在阿里产品上性能高，硬件严重依赖intel的cpu，损失了兼容性，但提高了性能
 - 目前已经在淘宝、天猫上线，把Oracle 官方JVM 版本全部替换了。

Dalvik VM:

- 谷歌开发的，应用于Android系统，并在Android2.2中提供了JIT，发展迅猛。
- Dalvik VM 只能称作虚拟机，而不能称作“Java 虚拟机”，它没有遵循 Java 虚拟机规范
- 不能直接执行 Java 的 Class 文件
- 基于寄存器架构，不是jvm的栈架构。
- 执行的是编译以后的dex (Dalvik Executable) 文件。执行效率比较高。
 - 它执行的dex (Dalvik Executable) 文件可以通过Class文件转化而来，使用Java语法编写应用程序，可以直接使用大部分的Java API等。
- Android 5.0 使用支持提前编译 (Ahead Of Time Compilation, AOT) 的 ART VM替换Dalvik VM。

AOT:files skip the step of translating into bytecode,they turn into machine code directly.

其他JVM:

Java Card VM、 Squawk VM、 JavaInJava、 Maxine VM、 Jikes RVM、 IKVM.NET、 Jam VM、 Cacao VM、 Sable VM、 Kaffe、 Jelatine JVM、 Nano VM、 MRP、 Moxie JVM

具体JVM的内存结构，其实取决于其实现，不同厂商的JVM，或者同一厂商发布的不同版本，都有可能存在一定差异。本套课程主要以Oracle HotSpot VM为默认虚拟机。

JVM发展历程

Graal VM

- 2018年4月，Oracle Labs公开了Graal VM，号称“Run Programs Faster Anywhere”，勃勃野心。与1995年java的“write once, run anywhere”遥相呼应。
- Graal VM在HotSpot VM基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”的运行平台使用。语言包括：Java、Scala、Groovy、Kotlin；C、C++、JavaScript、Ruby、Python、R等
- 支持不同语言中混用对方的接口和对象，支持这些语言使用已经编写好的本地库文件
- 工作原理是将这些语言的源代码或源代码编译后的中间格式，通过解释器转换为能被Graal VM接受的中间表示。Graal VM 提供Truffle工具集快速构建面向一种新语言的解释器。在运行时还能进行即时编译优化，获得比原生编译器更优秀的执行效率。
- 如果说HotSpot有一天真的被取代，Graal VM希望最大。但是Java的软件生态没有丝毫变化。

