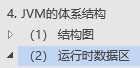
# 一些（面试）问题及答案

<https://www.lubojian.cn/post/112.html>

<https://xie.infoq.cn/article/83777ef09f349643cee6b5cd7>

## jvm运行时数据区哪些是线程私有/共享的？

参考：下面

线程私有/隔离：程序计数器，虚拟机栈，本地方法栈

线程公有/共享：方法区，堆

## （2） 是否所有对象都在堆上分配？

参考：享学vip 3期-对象的分配及垃圾回收机制下的19:30

答：几乎所有的对象都在堆中进行分配，也能在栈上分配。

## （3） JVM的内存划分

<https://juejin.cn/post/6935417287990050846>

<https://blog.csdn.net/CSDN_Terence/article/details/77771429>

<https://blog.csdn.net/ljd2038/article/details/83146816>

<https://segmentfault.com/a/1190000022080301>

就是JVM的“运行时数据区”。

# JVM是什么

## 参1：

<https://blog.csdn.net/sinat_35512245/article/details/54744815>

JVM是Java Virtual Machine的缩写，是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。Java虚拟机包括一套字节码指令集，一组寄存器，一个栈，一个垃圾回收堆和一个存储方法域。JVM屏蔽了与具体操作系统平台相关的信息，使Java程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。JVM在执行字节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行。

Java语言的一个非常重要的特点就是与平台无关。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少要编译成不同的目标代码。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息。使得Java语言编译程序只需生成Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。这就是Java能够“一次编译，到处运行”的原因。

## 参2：享学vip-2021.8.10 JVM内存管理

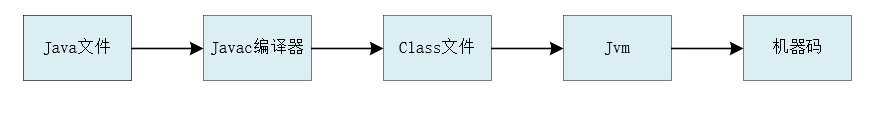
图示

描述已自动生成

# 2. JVM原理

## 参1：<https://segmentfault.com/a/1190000022080301>

想要运行Java文件，必须先通过一个叫javac的编译器，将代码编译成class文件，然后通过JVM把class文件解释成各个平台可以识别的机器码，最终实现跨平台运行代码。



## 参2：

<https://blog.csdn.net/sinat_35512245/article/details/54744815>

图示

描述已自动生成

Java编译器只要面向JVM，生成JVM能理解的代码或字节码文件。Java源文件经编译成字节码程序，通过JVM将每条指令翻译成不同平台机器码，通过特定平台运行。

# 3. 查看JVM源码

<https://zhuanlan.zhihu.com/p/146919894?utm_id=0#:~:text=%E5%A6%82%E4%BD%95%E9%98%85%E8%AF%BBJVM%20%E6%BA%90%E7%A0%81>,[JDK](javascript:volid(0))%E4%B8%ADJVM%EF%BC%88%E5%AE%89%E8%A3%85%E5%9C%A8%E6%9C%AC%E5%9C%B0C%3AProgram%20FilesJavajdk1.8.0\_121jrebinserver%E4%B8%8Bjvm.[dll](javascript:volid(0))%EF%BC%89%E6%9C%AC%E8%BA%AB%E5%B9%B6%E4%B8%8D%E5%BC%80%E6%BA%90%EF%BC%8C%E5%8F%AA%E8%83%BD%E6%89%BE%E6%9D%A5openJDK%E6%9D%A5%E7%9C%8B%EF%BC%88%E8%AF%B4%E6%98%AF%E5%92%8CJDK%E7%9B%B8%E4%BC%BC%E5%BA%A6%E5%BE%88%E9%AB%98%EF%BC%89

<https://zhuanlan.zhihu.com/p/295074294?utm_id=0>

<https://blog.csdn.net/chl87783255/article/details/111551781>

# 4. JVM的体系结构

## （1） 结构图

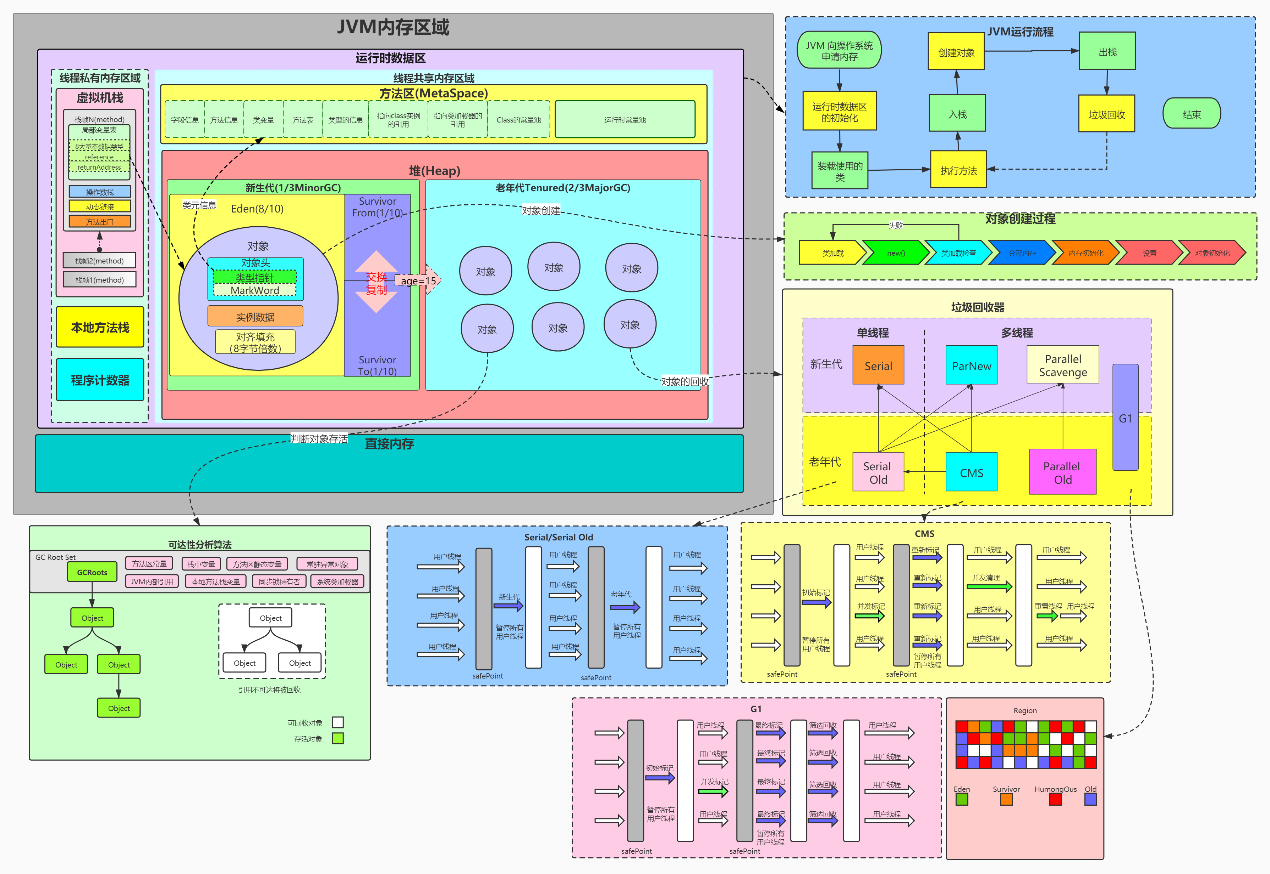
### 参1：

<https://blog.csdn.net/sinat_35512245/article/details/54744815>

图示

描述已自动生成

### 参2：享学vip3期-jvm的3节课



## （2） 运行时数据区

参考：

1. 深入理解Java虚拟机（第二版）2.2
2. 深入理解Java虚拟机（第三版）2.2

Java虚拟机在执行Java程序的过程中会把它所管理的内存分为若干个不同的数据区。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有的区域则依赖用户线程的启动和结束而建立和销毁。根据《Java虚拟机规范（Java SE7版）》的规定，Java虚拟机所管理的内存将会包括以下几个**运行时数据区**，如图

图示

描述已自动生成

### 【1】 程序计数器

这是一块较小的内存空间。它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支，循环，跳转，异常处理，线程恢复等基础功能都需要依赖该计数器。

由于Java虚拟机的多线程是通过线程轮转并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”内存。

如果线程正在执行的是一个Java方法，该计数器记录的是正在执行的虚拟机字节码指令的地址。如果线程正在执行的是Native方法，这个计数器值则为空（Undefined）。此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

### 【2】 Java虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stack）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

经常有人把Java内存区域笼统地分为堆内存和栈内存，这种分法比较粗糙，Java内存区域的划分实际上远比这个复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。其中所指的“堆”在后面会讲，而所指的“栈”就是现在这个虚拟机栈，或者说是虚拟机栈中局部变量表部分。

局部变量表存放了编译期可知的各种基本数据类型（boolean，byte，

char，short，int，float，long，double），对象引用（reference类型，它

不等同于对象本身，可能是一个指向对象起始地址的引用指针，也可能

是指向一个代表对象的句柄或其他与此对象相关的位置）和

returnAddress类型（指向了一条字节码指令的地址）。

其中64位长度的long和double类型的数据会占用2个局部变量空

间（Slot），其余的数据类型只占用1个。局部变量表所需的内存空间在

编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大

的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的

大小。

在Java虚拟机规范中，对这个区域规定了两种异常状况：如果线程

请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机栈容量可以动态扩展，当栈扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

### 【3】 本地方法栈

本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是很相似的，它们之间的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。在虚拟机规范中对本地方法栈中方法使用的语言，使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

### 【4】 Java堆

#### 参1：深入理解Java虚拟机（第二版）2.2

对于大多数应用来说，Java堆（Java Heap）是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例。几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译的发展与逃逸分析技术逐渐成熟，栈上分配，标量替换等优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不那么绝对了。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为GC堆。从内存回收的角度看，由于现在收集器基本都采用分代收集算法，所以Java堆中还可以细分为：新生代和老年代；再细致点有Eden空间，From Survivor空间，To Survivor空间等。从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer）。不过无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例；进一步划分的目的是为了更好地回收内存，或者更快地分配内存。在本章中，我们仅仅针对内存区域的作用进行讨论。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上连续即可。就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx和-Xms控制）。如果在堆中没有内存能完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

#### 参2：

<https://www.jianshu.com/p/2f4a8e04c657>

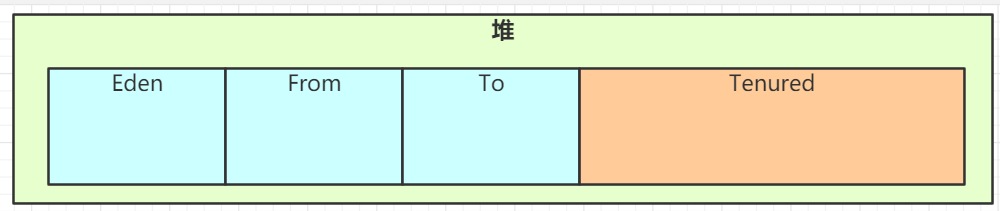
享学vip3期-jvm内存管理下的33min41s

享学vip3期-对象的分配及垃圾回收机制下的27min50s+48min9s

##### 1） 定义

JVM堆是Java对象的活动空间，程序中的类的对象从中分配空间，其存储着正在运行着的应用程序用到的所有对象。这些对象的建立方式就是new，当对象无用后，是GC来负责处理。

##### 2） JVM堆分类/分代模型



###### 1》 新生代

存储所有新生成的对象，被分为3部分：

1. Eden区
2. Survivor区：From Space/s0，To Space/s1

###### 2》 老年代

新生代中的对象，经过了一定次数的GC循环后未被回收掉，将被移入老年代。

### 【5】 方法区

#### 《1》 概述

##### 参1：深入理解Java虚拟机（第三版）2.2

方法区（Method Area） 与Java堆一样， 是各个线程共享的内存区域， 它用于存储已被虚拟机加载的类型信息、 常量、 静态变量、 即时编译器编译后的代码缓存等数据。 虽然《Java虚拟机规范》 中把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫作“非堆”（Non-Heap），目的是与Java堆区分开来。

说到方法区， 不得不提一下“永久代”这个概念， 尤其是在JDK 8以前， 许多Java程序员都习惯在HotSpot虚拟机上开发、 部署程序， 很多人都更愿意把方法区称呼为**“永久代”（Permanent Generation）** ， 或将两者混为一谈。 本质上这两者并不是等价的， 因为仅仅是当时的HotSpot虚拟机设计团队选择把收集器的分代设计扩展至方法区， 或者说使用永久代来实现方法区而已， 这样使得HotSpot的垃圾收集器能够像管理Java堆一样管理这部分内存， 省去专门为方法区编写内存管理代码的工作。 但是对于其他虚拟机实现， 譬如BEA JRockit、 IBM J9等来说， 是不存在永久代的概念的。 原则上如何实现方法区属于虚拟机实现细节， 不受《Java虚拟机规范》 管束， 并不要求统一。 但现在回头来看， 当年使用永久代来实现方法区的决定并不是一个好主意， 这种设计导致了Java应用更容易遇到内存溢出的问题（永久代有-XX：MaxPermSize的上限， 即使不设置也有默认大小， 而J9和JRockit只要没有触碰到进程可用内存的上限， 例如32位系统中的4GB限制， 就不会出问题） ， 而且有极少数方法（例如String::intern()） 会因永久代的原因而导致不同虚拟机下有不同的表现。 当Oracle收购BEA获得了JRockit的所有权后， 准备把JRockit中的优秀功能， 譬如Java Mission Control管理工具， 移植到HotSpot虚拟机时， 但因为两者对方法区实现的差异而面临诸多困难。 考虑到HotSpot未来的发展， 在JDK 6的时候HotSpot开发团队就有放弃永久代， 逐步改为采用本地内存（Native Memory） 来实现方法区的计划了[1]； 到了JDK 7的HotSpot， 已经把原本放在永久代的字符串常量池、 静态变量等移出； 而到了JDK 8， 终于完全废弃了永久代的概念， 改用与JRockit、 J9一样在本地内存中实现的**元空间（Metaspace）** 来代替， 把JDK 7中永久代还剩余的内容（主要是类型信息） 全部移到元空间中。

《Java虚拟机规范》 对方法区的约束是非常宽松的， 除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外， 甚至还可以选择不实现垃圾收集。 相对而言， 垃圾收集行为在这个区域的确是比较少出现的， 但并非数据进入了方法区就如永久代的名字一样“永久”存在了。 这区域的内存回收目标主要是针对常量池的回收和对类型的卸载， 一般来说这个区域的回收效果比较难令人满意， 尤其是类型的卸载， 条件相当苛刻， 但是这部分区域的回收有时又确实是必要的。 以前Sun公司的Bug列表中， 曾出现过的若干个严重的Bug就是由于低版本的HotSpot虚拟机对此区域未完全回收而导致内存泄漏。

根据《Java虚拟机规范》 的规定， 如果方法区无法满足新的内存分配需求时， 将抛出OutOfMemoryError异常。

[1] JEP 122-Remove the Permanent Generation：

<http://openjdk.java.net/jeps/122>。

##### 参2：享学vip3期

###### 参2-1：

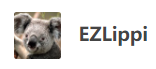
<https://ke.qq.com/webcourse/index.html?r=1677559851652#cid=347420&term_id=103755197&taid=12286518454734108&type=3072&source=PC_COURSE_DETAIL&vid=3701925922409846272> 35:26

**永久代**【<=jdk1.7】和**元空间**【jdk1.8】是HotSpot对**方法区**的实现的称呼，在java虚拟机规范中只有**方法区**的说法。

###### 参2-2：

<https://ke.qq.com/webcourse/index.html?r=1677559851652#cid=347420&term_id=103755197&taid=12286518454734108&type=3072&source=PC_COURSE_DETAIL&vid=3701925922409846272> 和（含示例）

##### 参3：<https://www.zhihu.com/question/38496907>

和

##### 参4：中搜“方法区”

##### 参5：<https://segmentfault.com/a/1190000022080301>

##### 参6：

<https://blog.csdn.net/CSDN_Terence/article/details/77771429>

#### 《2》 运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class文件中除了有类的版本，字段，方法，接口等描述信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量（literal）和符号引用。这部分内容将在类加载后进入方法区的运行时常量池中存放。

Java虚拟机对Class文件每一部分（自然也包括常量池）的格式都有严格规定，每一个字节用于存储哪些数据都必须符合规范上的要求才会被虚拟机认可，装载和执行。但对于运行时常量池，Java虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存Class文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只能编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得较多的便是String类的intern（）方法。

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请内存时会抛出OutOfMemoryError异常。

#### 自总结：

方法区用于存储已被虚拟机加载【即类加载机制】的class文件数据，这些数据被分为不同类别存储在不同数据结构中，包括：类型信息【类/接口的名称、字段信息、方法信息】、常量【static final】，静态变量，运行时常量池等。

### 【6】 直接内存（不属于运行时数据区）

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError异常出现，所以我们放到这里一起讲。

在JDK1.4中新加入了NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来复制数据。

显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存（包括RAM以及SWAP区域或者分页文件）大小以及处理器寻址空间的限制。服务器管理员在配置虚拟机参数时，会根据实际内存设置-Xmx等参数信息，但经常忽略直接内存，使得各个内存区域总和大于物理内存限制（包括物理和操作系统级的限制），从而导致动态扩展时出现OutOfMemoryError异常。

# JVM中的常量池

<https://cloud.tencent.com/developer/article/1450501>

<https://zhuanlan.zhihu.com/p/107776367>

<https://ke.qq.com/webcourse/index.html?r=1677559851652#cid=347420&term_id=103755197&taid=12286527044668700&type=3072&source=PC_COURSE_DETAIL&vid=3701925922460860215> 

JVM常量池主要分为【注：参考的资料中说的最多的是前三种】：

1. **Class(文件)常量池**
2. **是编译期生成的class文件的一部分**
3. **包括直接/字面量和符号引用。**
4. **运行时常量池**
5. **是方法区的一部分**
6. **包括Class文件常量池中的直接量和符号引用，同时经类加载中的解析阶段后****符号引用被替换为直接引用（直接指向实例对象的指针，内存地址）。**
7. **(全局)字符串常量池**
8. **《Java虚拟机规范》中未定义，属于一种优化技术。**
9. **在HotSpot VM中，jdk1.7开始存的是指向String对象的引用表，叫StringTable，其在堆区；jdk1.6及以前在方法区，存的是对象。**
10. **String.intern()与此常量池关联，作用：对于jdk1.7开始的HotSpot VM，若字符串常量池中的StringTable里有该字符串值对象的引用就复用并返回引用，没有就添加进去然后返回引用。**
11. **基本类型包装类对象常量池**

# JVM发展

1. JDK1.0发布时自带的虚拟机->Sun Classic VM

Sun Classic VM是最早的商用虚拟机,但是由于技术所限他的运行速度特别的慢.

（2） JDK1.2发布时,发布一个名为Exact VM

由于一开始的虚拟机运行编译代码太慢,java的开发团队一直优化,并且有了Exact VM这个高性能虚拟机的模型

（3） JDK1.3时 Sun HotSpot VM变成了默认的虚拟机

Sun HotSpot VM是我们见得最多的虚拟机,它是前身是一家小公司以开发一个相当于c 50%以上性能的虚拟机为目的开发的优秀的虚拟机,最终这家公司被Sun公司收购,Sun HotSpot VM集合了前两个虚拟机的优点

（4） 直到现在(JDk1.8)

目前为止JDk发布到了1.8版本,默认使用的虚拟机都是HotSpot

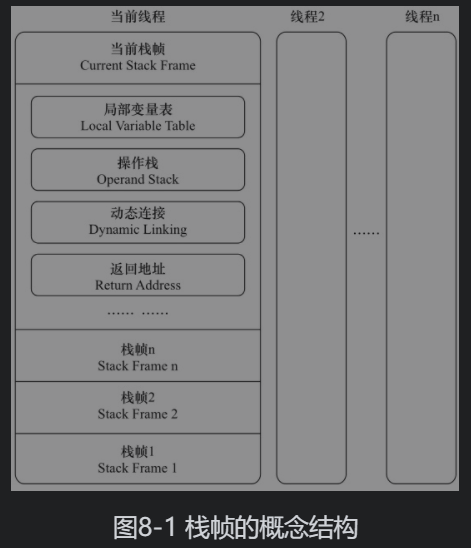
# Java运行时栈帧

参考：《深入理解java虚拟机v3》 8.2

**栈帧（Stack Frame）**是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）（详见2.2节）的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

每一个栈帧都包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的Code属性之中（详细见6.3.7节），因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

一个线程中的方法调用链可能会很长，很多方法都同时处于执行状态。对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为**当前栈帧（Current Stack Frame）**，与这个栈帧相关联的方法称为**当前方法（Current Method）**。执行引擎运行的所有字节码指令都只针对当前栈帧进行操作，在概念模型上，典型的栈帧结构如图8-1所示。



接下来详细讲解一下栈帧中的局部变量表、操作数栈、动态连接、方法返回地址等各个部分的作用和数据结构。

## 局部变量表

1. 定义

**局部变量表（Local Variable Table）**是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在Java程序编译为class文件时，就在方法的Code属性的max\_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。

Xx

## 操作数栈

**操作数栈（Operand Stack）**也常称为操作栈，它是一个后入先出（Last In First Out, LIFO）栈。同局部变量表一样，操作数栈的最大深度也在编译的时候写入到Code属性的max\_stacks数据项中。

Xx<https://v.douyu.com/show/a4Jj7lzlZXjWDk01>

## 动态连接

每个栈帧都包含一个指向运行时常量池（详见第2章）中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接（Dynamic Linking）。

Xx

## 方法返回地址

当一个方法开始执行后，只有两种方式可以退出这个方法：

* 方式1：执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者（调用当前方法的方法称为调用者），是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式称为**正常完成出口（Normal Method Invocation Completion）**。
* 方式2：在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，无论是Java虚拟机内部产生的异常，还是代码中使用athrow字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为**异常完成出口（Abrupt Method Invocation Completion）**。一个方法使用异常完成出口的方式退出，是不会给它的上层调用者产生任何返回值的。

确定继续执行地址：

无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的PC计数器的值可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中一般不会保存这部分信息。

退栈时可能执行的操作：

方法退出的过程实际上就等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈中，调整PC计数器的值以指向方法调用指令后面的一条指令等。

## 附加信息

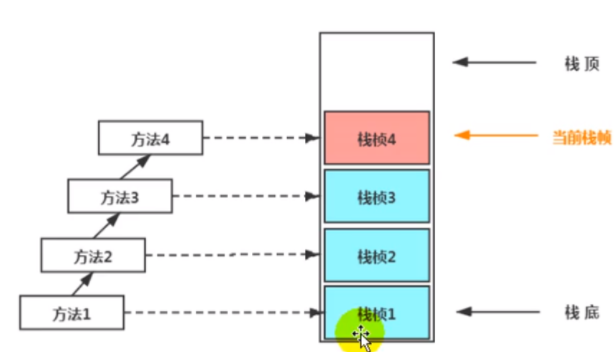
虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧之中，例如与调试相关的信息，这部分信息完全取决于具体的虚拟机实现，这里不再详述。在实际开发中，一般会把动态连接、方法返回地址与其他附加信息全部归为一类，称为栈帧信息。

# 虚拟机栈执行过程

参考：

1. <https://www.jianshu.com/p/ecfcc9fb1de7>
2. <https://blog.csdn.net/weixin_44624375/article/details/113922929>
3. <https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fazhegps%2Farticle%2Fdetails%2F54092466>
4. 《数据结构-严蔚敏-v1》电子版 P66

* 每个方法从调用到执行完成的过程，就对应着一个栈帧在虚拟机栈中的入栈（压栈）到出栈（弹栈）的过程。
* JVM直接对Java栈的操作只有两个，就是对栈帧的**压栈**和**出栈**，遵循“**先进后出**”/"**后进先出**"原则
* 在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧被称为**当前栈帧**，与当前栈帧相对应的方法称为**当前方法**，定义这个方法的类就是**当前类**。
* **执行引擎运行的所有字节码指令只针对当前栈帧进行操作**
* 如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，放在栈的顶端，成为新的当前栈
* 不同线程中所包含的栈帧是不允许存在相互引用的，即不可能在一个栈帧中引用另外一个线程的栈帧。
* 如果当前方法调用了其他方法，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧。



# JVM与进程/线程

参考1：<https://blog.csdn.net/ghj1976/article/details/5481038>

Java编写的程序都运行在在Java虚拟机（JVM）中，每用java命令启动一个java应用程序【怎么定义？】，就会启动一个JVM进程。

在同一个JVM进程中，有且只有一个进程，就是它自己。在这个JVM环境中，所有程序代码的运行都是以线程来运行的。JVM找到程序的入口点main()，

在这个JVM环境中，所有程序代码的运行都是以线程来运行的。

JVM找到程序程序的入口点main()，然后运行main()方法，这样就产生了一个线程，这个线程称之为主线程。当main方法结束后，主线程运行完成。JVM进程也随即退出。

如下图所示：

图示, 文本

描述已自动生成

参考2：<https://blog.csdn.net/datadev_sh/article/details/79192552>

 java程序（怎么定义）是跑在JVM上的，严格来讲，是跑在JVM实例上的，一个JVM实例其实就是JVM跑起来的进程，二者合起来称之为一个JAVA进程。

各个JVM实例之间是相互隔离的。

参考3：https://zhuanlan.zhihu.com/p/25713880

JVM在Java程序开始执行的时候，它才运行，程序结束的时它就停止。

一个Java程序会开启一个JVM进程，如果一台机器上运行三个程序，那么就会有三个运行中的JVM进程。

# JIT技术

<https://chinese.freecodecamp.org/news/just-in-time-compilation-explained/>

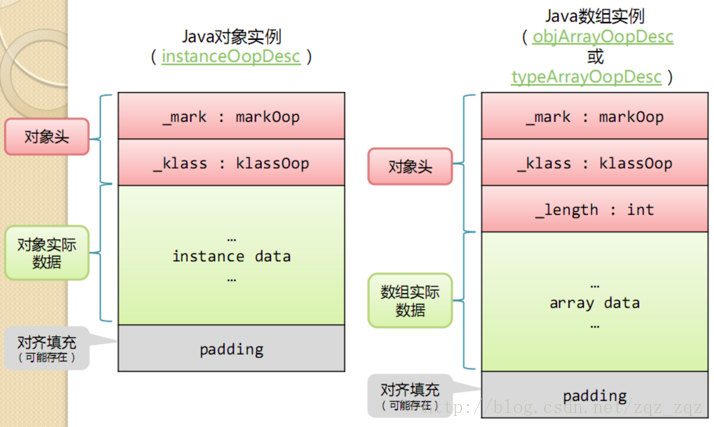
# Java对象结构

## 参1：

<https://blog.csdn.net/zqz_zqz/article/details/70246212>

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头（[Header](https://so.csdn.net/so/search?q=Header&spm=1001.2101.3001.7020)）、实例数据（Instance Data）和对齐填充（Padding）。

下图是普通对象实例与数组对象实例的数据结构：



## 参2：享学vip3期-对象的分配及垃圾回收机制上的23：55

# 12. 对象的分配策略

参考：

1. 主：享学vip 3期-对象的分配及垃圾回收机制下的19:10
2. 辅：

<https://www.cnblogs.com/little-YTMM/p/5613642.html>

<https://www.jb51.net/article/229409.htm>

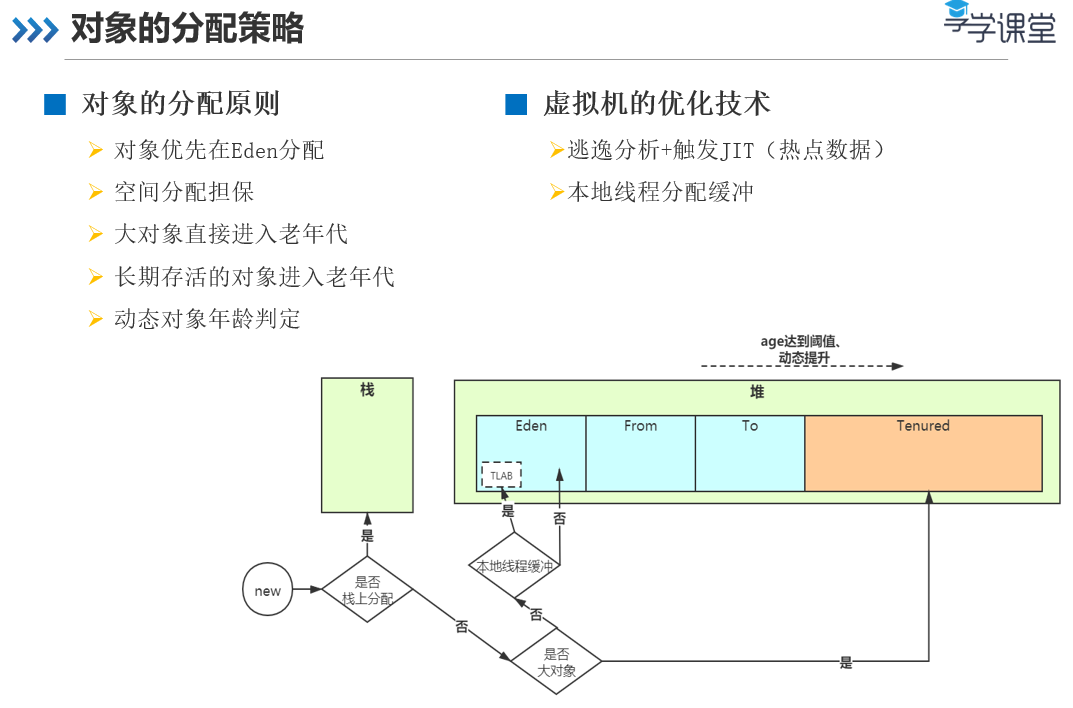
<https://www.cnblogs.com/ZJOE80/p/12931989.html>

## （1） 概述

### 参1：<https://www.cnblogs.com/little-YTMM/p/5613642.html>

Java技术体系中所提倡的自动内存管理最终可以归结为自动化地解决2个问题：**给对象分配内存**【即这里的内容】以及**回收分配给对象的内存**【即GC】。

### 参2：享学vip 3期-对象的分配及垃圾回收机制下的19:10+40:14



其中：“对象的分配原则”为大的分配原则；“虚拟机的优化技术”为基于大的分配原则的优化技术，里面的“逃逸分析+触发JIT（热点数据）”是对象在栈上分配的2个条件。

需要记住上面这张图，即对象的分配策略流程。

## （2） 详解

### 【1】 逃逸分析+触发JIT（热点数据）

见：享学vip 3期-对象的分配及垃圾回收机制下的19:30

### 对象优先在Eden分配（+本地线程分配缓冲）

参考：

1. <https://www.cnblogs.com/little-YTMM/p/5613642.html>
2. 享学vip 3期-对象的分配及垃圾回收机制下的27:44+31:23

大多数情况下，（新）对象在新生代Eden区分配。

### 大对象直接进老年代

#### 参1：<https://www.cnblogs.com/little-YTMM/p/5613642.html>

大对象是指需要大量连续内存空间的Java对象（例如很长的字符串/数组）。

#### 参2：享学vip 3期-对象的分配及垃圾回收机制下的28:28

### 长期存活的对象将进入老年代+动态对象年龄判断/定

参考：

1. <https://www.cnblogs.com/little-YTMM/p/5613642.html>
2. 享学vip 3期-对象的分配及垃圾回收机制下的32:02+33:57
3. 动态对象年龄判定

<https://www.jianshu.com/p/989d3b06a49d>

<https://www.cnblogs.com/zjfjava/p/13861797.html>

<https://github.com/fenixsoft/jvm_book>

JVM为每个对象定义一个对象年龄计数器：

* 1. 若对象在Eden出生并经历过第一次Minor GC后仍然存活，并且能够被Survivor容纳，则应该被移动到Survivor空间中，并且对象年龄设置为1。
  2. 对象在Survivor区中每熬过一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认15岁，可通过参数-XX:MaxTenuringThreshold设置），就会被晋升到老年代中。
  3. 要注意的是：为了能更好地适应不同程序的内存状况，JVM并不是永远的要求对象的年龄必须达到MaxTenuingThreshold才能晋升老年代，如果在Survivor空间中小于等于某年龄所有对象大小的总和大于Survivor空间的一半，年龄大于等于该年龄的对象就可以直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄。【动态对象年龄判定】

### 空间分配担保

<https://www.cnblogs.com/little-YTMM/p/5613642.html>

<https://ke.qq.com/webcourse/index.html?r=1678119046269#cid=347420&term_id=103755197&taid=12286522749701404&type=3072&source=PC_COURSE_DETAIL&vid=3701925922528318833> 2:02:00

<https://www.jb51.net/article/229409.htm>

#### 是什么：

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间：

若该条件成立，则进行Minor GC是安全的。

若不成立，则虚拟机会查看-XX：HandlePromotionFailure参数的设置值是否允许担保失败。若允许，则立即检查老年代最大可用连续空间是否大于历代晋升到老年代对象的平均大小：

若大于，将尝试着进行一次Minor GC，尽管它是有风险的。

若小于或者HandlePromotionFailure设置为不允许冒险，则这时要改为进行一次Full GC。

#### “冒险”是冒了什么风险：

前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况——最极端的情况就是内存回收后新生代中所有对象都存活，需要老年代进行分配担保，把Survivor无法容纳的对象直接送入老年代，这与生活中贷款担保类似。老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，但一共有多少对象会在这次回收中活下来在实际完成内存回收之前是无法明确知道的，所以只能取之前每一次回收晋升到老年代对象容量的平均大小作为经验值，与老年代的剩余空间进行比较，决定是否进行Full GC来让老年代腾出更多空间。

#### 那我们需要把担保【即-XX：HandlePromotionFailure参数】打开吗：

取历史平均值来比较其实仍然是一种赌概率的解决办法，也就是说假如某次Minor GC存活后的对象突增，远远高于历史平均值的话，依然会导致担保失败。如果出现了担保失败，那就只好老老实实地重新发起一次Full GC，这样停顿时间就很长了。虽然担保失败时绕的圈子是最大的，但通常情况下都还是会将-XX：HandlePromotionFailure开关打开，避免Full GC过于频繁。

# 跨代（引用）

## 是什么

<https://www.jianshu.com/p/f1ff4ab0fed7>

红色的线表示由虚拟机栈中发出的引用。显然B--->A、E--->F都是跨代引用。

图示

描述已自动生成

# Java内存模型：Java Memory Model<JMM>

## （1） 产生背景

<https://zhuanlan.zhihu.com/p/51613784>

在介绍 Java 内存模型之前，我们先了解一下物理计算机中的并发问题，理解这些问题可以搞清楚内存模型产生的背景。

物理机遇到的并发问题与虚拟机中的情况有不少相似之处，物理机的解决方案对虚拟机的实现有相当的参考意义。

### 物理机的并发问题

1. 硬件的效率问题【待去掉】

使用“缓存作为内存和处理器之间的缓冲”

【2】 缓存一致性问题

【3】 代码乱序执行优化问题

## （2） 是什么

<https://zhuanlan.zhihu.com/p/51613784>

### 【1】 内存模型通用概念

为了更好解决上面提到的系列问题，内存模型被总结提出，我们可以把内存模型理解为在特定操作协议下，对特定的内存或高速缓存进行读写访问的过程抽象。

不同架构的物理计算机可以有不一样的内存模型，Java 虚拟机也有自己的内存模型。

### 【2】 Java内存模型

Java 虚拟机规范中试图定义一种 **Java 内存模型（Java Memory Model，简称 JMM）**来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果，不必因为不同平台上的物理机的内存模型的差异，对各平台定制化开发程序。

更具体一点说，Java 内存模型提出目标在于：定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。

此处的变量(Variables)与 Java 编程中所说的变量有所区别，它包括了实例字段、静态字段和构成数值对象的元素，但不包括局部变量与方法参数，因为后者是线程私有的。

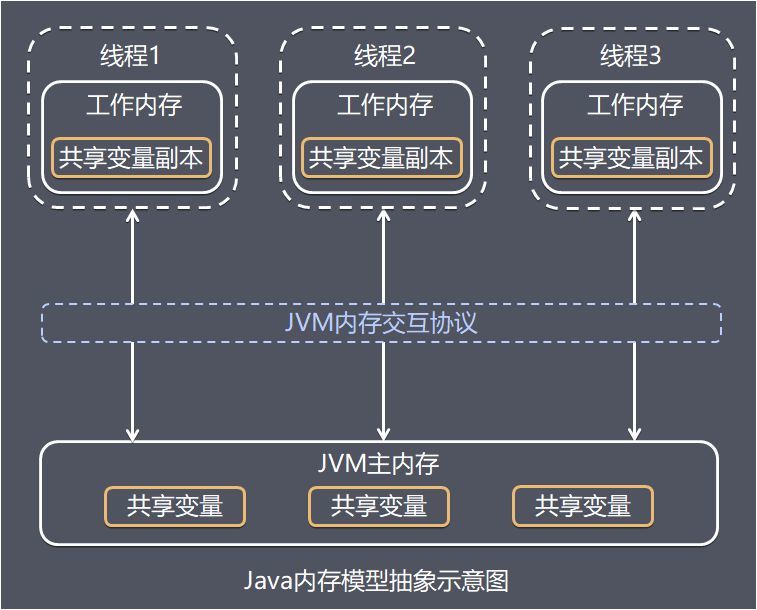
注：如果局部变量是一个 reference 类型，它引用的对象在 Java 堆中可被各个线程共享，但是 reference 本身在 Java 栈的局部变量表中，它是线程私有的。

## （3） Java内存模型的组成

参考：

1. <https://zhuanlan.zhihu.com/p/51613784>
2. 《Java并发编程之美》2.4
3. <https://zhuanlan.zhihu.com/p/455820119?utm_id=0#:~:text=%E8%80%8CJMM%E5%86%85%E5%AD%98%E6%A8%A1%E5%9E%8B,%E5%88%B7%E5%9B%9E%E5%88%B0%E4%B8%BB%E5%86%85%E5%AD%98%E4%B8%AD%E3%80%82>
4. <https://blog.csdn.net/xujingyiss/article/details/122243996?spm=1001.2101.3001.6650.8&utm_medium=distribute.wap_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-122243996-blog-107197067.wap_relevant_featuresortv7&depth_1-utm_source=distribute.wap_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-122243996-blog-107197067.wap_relevant_featuresortv7>

Java 内存模型抽象示意图如下：



**主内存【线程共享】：**

Java 内存模型规定了所有变量都存储在主内存(Main Memory)中（此处的主内存与介绍物理硬件的主内存RAM名字一样，两者可以互相类比，但此处仅是虚拟机内存的一部分）。

**工作内存【线程私有】：**

每条线程都有自己的工作内存(Working Memory，又称本地内存)，线程的工作内存中保存了该线程使用到的主内存中的共享变量的副本拷贝。

工作内存是 JMM 的一个抽象概念，并不真实存在。它对应到物理机涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

### 工作内存和ThreadLocal的关系

<https://blog.csdn.net/weixin_43821273/article/details/102458329>

<https://ask.csdn.net/questions/7428343>

## （4） Java内存操作的并发问题

<https://zhuanlan.zhihu.com/p/51613784>

结合前面介绍的物理机的处理器处理内存的问题，可以类比总结出 JVM 内存操作的问题，下面介绍的 Java 内存模型的执行处理将围绕解决这两个问题展开。

### 【1】 工作内存数据一致性

### 【2】 指令重排序优化

## （5） 共享变量的内存可见性问题

参考：《Java并发编程之美》2.4

Java内存模型是一个抽象的概念，那么在实际实现中线程的工作内存是什么呢？请看图2-5。

图形用户界面

描述已自动生成

图中所示是一个双核CPU系统架构，每个核有自己的控制器和运算器，其中控制器包含一组寄存器和操作控制器，运算器执行算术逻辑运算。每个核都有自己的一级缓存，在有些架构里面还有一个所有CPU都共享的二级缓存。那么Java内存模型里面的**工作内存**，就对应这里的L1或者L2缓存或者CPU的寄存器。

当一个线程操作共享变量时，它首先从主内存复制共享变量到自己的工作内存，然后对工作内存里的变量进行处理，处理完后将变量值更新到主内存。

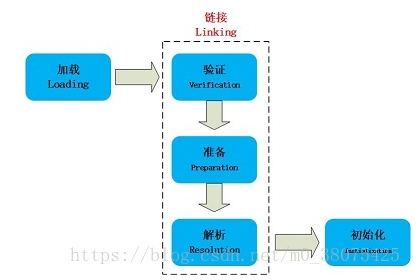
那么假如线程A和线程B同时处理一个共享变量，会出现什么情况？我们使用图2-5所示CPU架构，假设线程A和线程B使用不同CPU执行，并且当前两级Cache都为空，那么这时候由于Cache的存在，将会导致**内存不可见问题**，具体看下面的分析：

1. 线程A首先获取共享变量X的值，由于两级Cache都没有命中，所以加载主内存中X的值，假如为0。然后把X=0的值缓存到两级缓存，线程A修改X的值为1，然后将其写入两级Cache，并且刷新到主内存，并且刷新到主内存。线程A操作完毕后，线程A所在的CPU的两级Cache内和主内存里面的X的值都是1；
2. 线程B获取X的值，首先一级缓存没有命中，然后看二级缓存，二级缓存命中了，所以返回X=1；到这里一切都正常的，因为这时候主内存中也是X=1。然后线程B修改X的值为2，并将其存放到线程2所在的一级Cache和共享二级Cache中，最后更新主内存中X的值为2；到这里一切都是好的；
3. 线程A这次又需要修改X的值，获取时一级缓存命中，并且X=1，到这里问题出现了，明明线程B已经把X的值修改为了2，为何线程A获取的还是1呢？这就是共享变量的内存不可见问题，即线程B写入的值对线程A不可见。

# 16. Java类加载详解

参考：<https://blog.csdn.net/m0_38075425/article/details/81627349>

当程序主动使用某个类时，如果该类还未被加载到内存中，则JVM会通过加载、链接、初始化3个步骤来对该类进行初始化。如果没有意外，JVM将会连续完成3个步骤，所以有时也把这3个步骤统称为类加载或类初始化。



## 类加载过程

### 加载

#### 概述：

加载指的是将类的class文件读入到内存，并为之创建一个java.lang.Class对象，也就是说，当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。

类的加载由类加载器完成，类加载器通常由JVM提供，这些类加载器也是前面所有程序【根据上下文推测，这篇blog是从一篇大文章中摘的关于类加载的部分，在类加载部分前有些代码示例】运行的基础，JVM提供的这些类加载器通常被称为系统类加载器。除此之外，开发者可以通过继承ClassLoader基类创建自己的类加载器。

通过使用不同的类加载器，可以从不同来源加载类的二进制数据，通常有如下几种来源：

1. 从本地文件系统加载class文件，这是前面绝大部分示例程序的类加载方式。
2. 从JAR包加载class文件，这种方式也很常见，前面介绍JDBC编程时用到的数据库驱动类就放在JAR文件中，JVM可以从JAR文件中直接加载该class文件。
3. 通过网络加载class文件。
4. 把一个Java源文件动态编译，并执行加载。

类加载器通常无须等到“首次使用”该类时才加载该类，Java虚拟机规范允许系统预先加载某些类。

#### 详解：

见下面

### 链接

当类被加载之后，系统为之生成一个对应的Class对象，接着将会进入链接阶段，链接阶段负责把类的二进制数据合并到JRE中。类链接又可分为如下3个阶段：

#### 验证

验证阶段用于检验被加载的类是否有正确的内部结构，并和其他类协调一致。Java相对C++语言是安全的语言，例如它有C++不具有的数组越界检查。这本身就是对自身安全的一种保护。验证阶段是Java非常重要的一个阶段，它会直接保证应用是否会被恶意入侵的一道重要的防线，越是严谨的验证机制越安全。验证的目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全。其中包括四种验证：**文件格式验证，元数据验证，字节码验证，符号引用验证。**

下面一一说明：

##### 文件格式验证

主要验证字节流是否符合Class文件格式规范，并且能被当前的虚拟机加载处理。例如：主，次版本号是否在当前虚拟机处理的范围之内。常量池中是否有不被支持的常量类型。指向常量中的索引值是否有不存在的常量或不符合类型的常量。

##### 元数据验证

对字节码描述的信息进行语义的分析，分析是否符合java的语言语法规范。

##### 字节码验证

最重要的验证环节，分析数据流和控制，确定语义是合法的，符合逻辑的。主要针对元数据验证后对方法体的验证。保证类方法在运行时不会有危害出现。

##### 符号引用验证

主要是针对符号引用转换为直接引用的时候，是会延伸到第三解析阶段，主要去确定访问类型等涉及到引用的情况，主要是保证引用一定会被访问到，不会出现类无法访问的问题。

#### 准备

类准备阶段负责为类的静态变量分配内存，并设置默认初始值。

#### 解析

将类的二进制数据中的符号引用替换成直接引用。

说明：

* 符号引用：是以一组符号来描述所引用的目标，符号可以是任何的字面形式的字面量，只要不会出现冲突能够定位到就行。布局和内存无关。
* 直接引用：是指向目标的指针，偏移量或者能够直接定位的句柄。该引用是和内存中的布局有关的，并且一定加载进来的。

### 初始化

初始化是为类的静态变量赋予正确的初始值，准备阶段和初始化阶段看似有点矛盾，其实不然，如果类中有语句：private static int a=10，它的执行过程是这样的，首先字节码文件被加载到内存后，先进行链接的验证这一步骤，验证通过后进入准备阶段，给a分配内存，因为a是static的，所以此时a等于int类型的默认初始值0，即a=0，然后到解析（后面再说），再到初始化这一步骤时，才把a的真正值赋给a，此时a=10。

## 类加载时机

<https://blog.csdn.net/m0_38075425/article/details/81627349>

<https://juejin.cn/post/6844903621000216589>

<https://cloud.tencent.com/developer/article/1651442>

<https://juejin.cn/post/7032903220238385165>

<https://www.cnblogs.com/fnlingnzb-learner/p/11990943.html>

<https://developer.aliyun.com/article/666294>

### 主动引用【会触发类的加载】

在什么情况下需要开始类加载过程的第一个阶段“**加载**”，《Java虚拟机规范》中并没有进行强制约束，但是对于初始化阶段《Java虚拟机规范》则是严格规定了有且只有以下六种情况必须立即对类进行“**初始化**”（加载、链接都会随着发生）：

1. 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则必须先触发其初始化。

涉及到这四条指令的典型场景有：

* 使用new关键字实例化对象的时候
* 读取或设置一个类的静态字段

注：被 final 修饰，已在编译期把结果放入常量池的静态字段除外【见下面文本, 信件

描述已自动生成】。

* 调用一个类的静态方法的时候

1. 使用 java.lang.reflect包的方法对类进行反射调用的时候【如Class.forName("com.lyj.load")】，如果类没有进行过初始化，则需要先触发其初始化。
2. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

注：关于接口初始化，其在初始化的时候和类的初始化不同，其并不需要父接口的初始化，只有真正用到了父接口的时候（例如引用接口中定义的常量）才会初始化。

1. 当虚拟机启动时，用户需要指定一个要执行的主/启动类（包含 main() 方法的那个类），虚拟机会先初始化这个主类。
2. 当使用 JDK7 的动态语言支持时，如果一个

java.lang.invoke.MethodHandle 实例最后的解析结果为 REF\_getStatic, REF\_putStatic, REF\_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

1. 一个接口定义了JDK8新加入的默认方法(被**default**关键字修饰的接口方法)，在初始化该接口的实现类的时候，需要先初始化该接口。

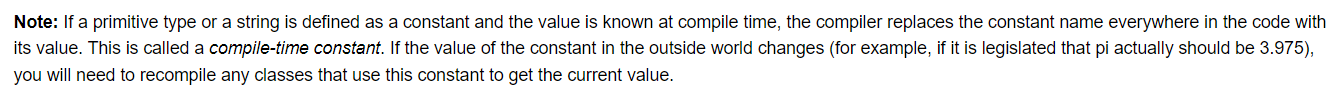
### 被动引用【不会触发类的加载】

以上6种场景中的行为称为对一个类进行**主动引用**。除此之外，所有引用类的方式都不会触发初始化，称为**被动引用**。

被动引用的常见例子包括：

1. 通过子类引用父类的静态字段，不会导致子类初始化，**只有父类被初始化**
2. 通过数组定义来引用类，不会触发此类的初始化
3. static final修饰的常量：对于一个常量，如果它的值在编译时就可以确定下来，那么这个常量相当于“宏变量”。Java编译器会在编译时直接把这个常量出现的地方替换成它的值，因此即使程序使用该常量，也不会导致该类的初始化。反之，如果常量的值不能在编译时确定下来，则必须等到运行时才可以确定该常量的值，如果通过该类来访问它的常量，则会导致该类被初始化。

<https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html>



## “加载”详解

### 1.3.1 类加载器

#### （1） 概述

参考：

1. <https://blog.csdn.net/m0_38075425/article/details/81627349>
2. 图形用户界面, 应用程序

   描述已自动生成
3. <https://www.51cto.com/article/658174.html>
4. <https://blog.csdn.net/javazejian/article/details/73413292>
5. <https://blog.csdn.net/lijianqingfeng/article/details/104863552>

Java程序被编译器编译之后成为字节码文件(.class文件)，当程序需要某个类时，虚拟机便会通过**类加载器**将对应的class文件加载进内存的JVM中，创建出对应的java.lang.Class对象。【即对应类加载过程中的“加载”阶段】

一旦一个类被加载进JVM中，同一个类就不会被再次载入了。正如一个对象有一个唯一的标识一样，一个载入JVM的类也有一个唯一的标识。在Java中，一个类用其全限定类名（包名+类名）作为标识；但在JVM中，一个类用其全限定类名和其类加载器作为唯一标识。例如，如果在pg的包中有一个名为Person的类，被类加载器ClassLoader的实例kl负责加载，则该Person类对应的Class对象在JVM中表示为（Person.pg.kl）。这意味着两个类加载器加载的同名类：（Person.pg.kl）和（Person.pg.kl2）是不同的，它们所加载的类也是完全不同，互不兼容的。

#### （2） 类别

##### 【1】 三种预定义类加载器

JVM预定义有三种类加载器，当一个JVM启动的时候，Java开始使用如下三种类加载器：

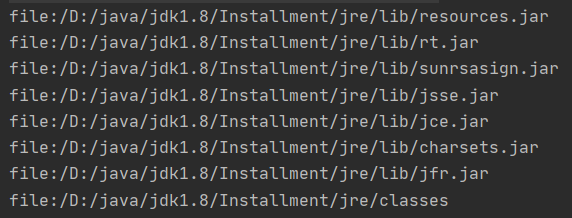
###### 《1》 根/启动/引导类加载器（bootstrap class loader）

参1：<https://blog.csdn.net/m0_38075425/article/details/81627349>

它用来加载 Java 的核心类【核心类库中的类？】，是用原生代码来实现的，并不继承自 java.lang.ClassLoader（负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。

下面程序可以获得根类加载器所加载的核心类库，并会看到本机安装的Java环境变量指定的jdk中提供的核心jar包路径：

|  |
| --- |
| public class ClassLoaderTest1 {   public static void main(String[] args) {  URL[] urls = sun.misc.Launcher.*getBootstrapClassPath*().getURLs();  for (URL url : urls) {  System.*out*.println(url.toExternalForm());  }  } } |

运行结果：

参2：<https://blog.csdn.net/javazejian/article/details/73413292>

启动类加载器主要加载的是JVM自身需要的类，这个类加载使用C++语言实现的，是虚拟机自身的一部分，它负责将 <JAVA\_HOME>/lib路径下的核心类库或-Xbootclasspath参数指定的路径下的jar包加载到内存中，注意由于虚拟机是按照文件名识别加载jar包的，如rt.jar，如果文件名不被虚拟机识别，即使把jar包丢到lib目录下也是没有作用的(出于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类)。

###### 《2》 扩/拓展类加载器（extension class loader）

参1：<https://blog.csdn.net/m0_38075425/article/details/81627349>

它负责加载jre的扩展目录：lib/ext或者由java.ext.dirs系统属性指定的目录中的JAR包的类。由java语言实现，父类加载器为null（就是启动类加载器）。

参2：<https://blog.csdn.net/javazejian/article/details/73413292>

扩展类加载器是指Sun公司(已被Oracle收购)实现的sun.misc.Launcher$ExtClassLoader类，由Java语言实现的，是Launcher的静态内部类，它负责加载<JAVA\_HOME>/lib/ext目录下或者由系统变量-Djava.ext.dir指定路径中的类库，开发者可以直接使用标准扩展类加载器。

|  |
| --- |
| //ExtClassLoader类中获取路径的代码  private static File[] getExtDirs() {  //加载<JAVA\_HOME>/lib/ext目录中的类库  String s = System.getProperty("java.ext.dirs");  File[] dirs;  if (s != null) {  StringTokenizer st =  new StringTokenizer(s, File.pathSeparator);  int count = st.countTokens();  dirs = new File[count];  for (int i = 0; i < count; i++) {  dirs[i] = new File(st.nextToken());  }  } else {  dirs = new File[0];  }  return dirs;  } |

###### 《3》 系统类加载器（system class loader）

参1：<https://blog.csdn.net/m0_38075425/article/details/81627349>

被称为系统（也称为应用）类加载器，它负责在JVM启动时加载来自java命令的-classpath选项、java.class.path系统属性，或者CLASSPATH环境变量所指定的JAR包和类路径。程序可以通过ClassLoader的静态方法getSystemClassLoader()来获取系统类加载器。如果没有特别指定，则用户自定义的类加载器都以此类加载器作为父加载器。由Java语言实现，父类加载器为ExtClassLoader。

参2：<https://blog.csdn.net/javazejian/article/details/73413292>

也称应用程序加载器是指 Sun公司实现的sun.misc.Launcher$AppClassLoader。它负责加载系统类路径java -classpath或-D java.class.path 指定路径下的类库，也就是我们经常用到的classpath路径，开发者可以直接使用系统类加载器，一般情况下该类加载是程序中默认的类加载器，通过ClassLoader#getSystemClassLoader()方法可以获取到该类加载器。

##### 【2】 自定义类加载器

<https://blog.csdn.net/javazejian/article/details/73413292>

###### 《1》 如何自定义

实现自定义类加载器需要继承ClassLoader或者URLClassLoader，继承ClassLoader则需要自己重写findClass()方法并编写加载逻辑，继承URLClassLoader则可以省去编写findClass()方法以及class文件加载转换成字节码流的代码。

###### 《2》 为什么

那么编写自定义类加载器的意义何在呢？

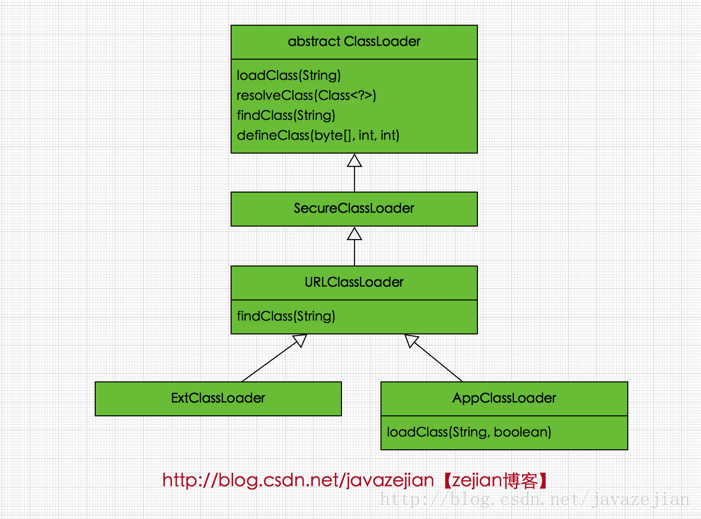
* 当class文件不在ClassPath路径下，默认系统类加载器无法找到该class文件，在这种情况下我们需要实现一个自定义的ClassLoader来加载特定路径下的class文件生成class对象。
* 当一个class文件是通过网络传输并且可能会进行相应的加密操作时，需要先对class文件进行相应的解密后再加载到JVM内存中，这种情况下也需要编写自定义的ClassLoader并实现相应的逻辑。
* 当需要实现热部署功能时(一个class文件通过不同的类加载器产生不同class对象从而实现热部署功能)，需要实现自定义ClassLoader的逻辑。

###### 《3》 示例

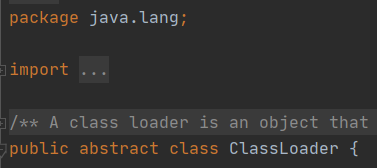
#### （3） 相互间的关系

##### 【1】 类继承关系

<https://blog.csdn.net/javazejian/article/details/73413292>



###### 《1》 ClassLoader：



作用

从上图可以看出顶层的类加载器是ClassLoader类，它是一个抽象类，其后所有的类加载器都继承自ClassLoader（不包括启动类加载器）。

重要的方法

loadClass（String）

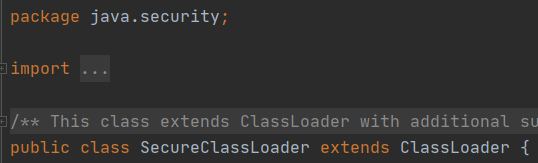
体现了class文件加载流程，见下面。

findClass（String）

defineClass(byte[] b, int off, int len)

resolveClass(Class<?> c)

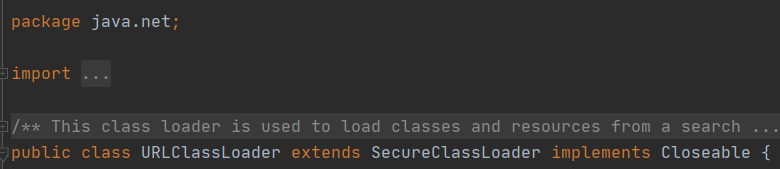
###### 《2》 SecureClassLoader：



作用

SecureClassLoader扩展了 ClassLoader，新增了几个与使用相关的代码源(对代码源的位置及其证书的验证)和权限定义类验证(主要指对class源码的访问权限)的方法。

###### 《3》 URLClassLoader：



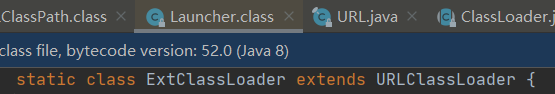
作用

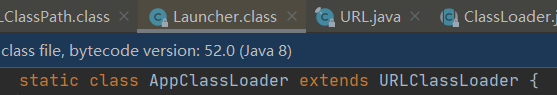
一般我们不会直接跟SecureClassLoader类打交道，更多是与它的子类URLClassLoader有所关联。

前面说过，ClassLoader是一个抽象类，很多方法是空的没有实现，比如 findClass()、findResource()等。而URLClassLoader这个实现类为这些方法提供了具体的实现，并新增了URLClassPath类协助取得Class字节码流等功能。

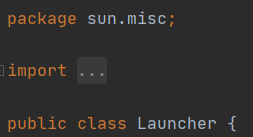
###### 《4》 Launcher.ExtClassLoader+Launcher.AppClassLoader：

拓展类加载器ExtClassLoader和系统类加载器AppClassLoader，





这两个类都继承自URLClassLoader，是sun.misc.Launcher

的静态内部类。

ExtClassLoader和AppClassLoader都是由sun.misc.Launcher创建的。

##### 【2】 父子关系

<https://blog.csdn.net/javazejian/article/details/73413292>

类加载器间的父子关系(不是类继承关系)，主要可以分为以下4点：

* 启动类加载器：由C++实现，没有父类。
* 拓展类加载器(ExtClassLoader)：由Java语言实现，父类加载器为null（其实就是启动类加载器）

参1：<https://blog.csdn.net/justloveyou_/article/details/72217806>

事实上，由于启动类加载器无法被Java程序直接引用，因此JVM默认直接使用 null 代表启动类加载器。

参2：<https://juejin.cn/post/7038627461713788958>

因为**启动类加载器并非java代码实现, 已经脱离了java体系**, 所以如果尝试调用扩展类加载器的getParent()方法获取父类加载器会得到**null**的返回, 然而**它的父类加载器确实是BootStrap ClassLoader。**

* 系统类加载器(AppClassLoader)：由Java语言实现，父类加载器为ExtClassLoader
* 自定义类加载器：父类加载器肯定为AppClassLoader。

### JVM的类加载机制

<https://blog.csdn.net/m0_38075425/article/details/81627349>

<https://cloud.tencent.com/developer/article/1353281>

主要有如下3种【在类加载过程中均用到了】：

#### 全盘负责

<https://blog.csdn.net/m0_38075425/article/details/81627349>

<https://blog.csdn.net/zhangzeyuaaa/article/details/42499839>

<https://cloud.tencent.com/developer/article/1353281>

即当一个类加载器负责加载某个Class时，该Class所依赖和引用的其他Class也将由该类加载器负责载入，除非显式使用另外一个类加载器来载入。

例如，系统[类加载](https://so.csdn.net/so/search?q=%E7%B1%BB%E5%8A%A0%E8%BD%BD&spm=1001.2101.3001.7020)器AppClassLoader加载入口类（含有main方法的类）时，会把main方法所依赖的类及引用的类也载入，依此类推。“全盘负责”机制也可称为当前类加载器负责机制。显然，入口类所依赖的类及引用的类的当前类加载器就是入口类的类加载器。

以上步骤只是调用了ClassLoader.loadClass(name)方法，并没有真正定义类。真正加载class字节码文件生成Class对象由“双亲委派”机制完成。

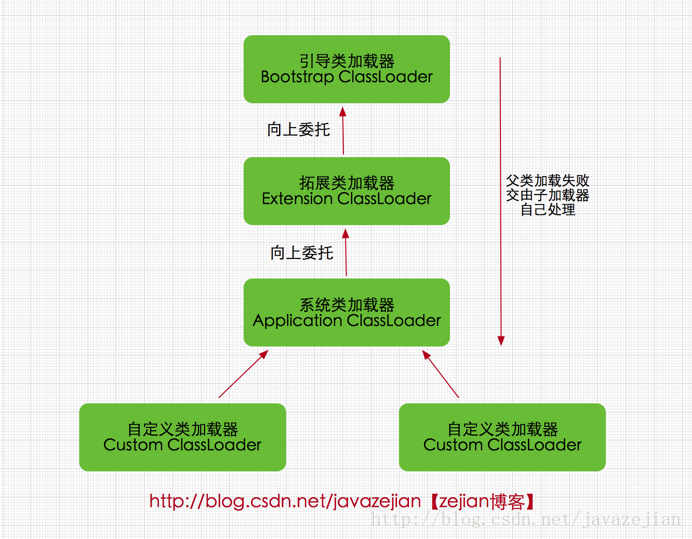
#### 双亲委派/托

##### 【1】 工作原理

<https://blog.csdn.net/m0_38075425/article/details/81627349>

<https://blog.csdn.net/javazejian/article/details/73413292>

双亲委派模式要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器，请注意双亲委派模式中的父子关系并非通常所说的类继承关系，而是采用组合关系来复用父类加载器的相关代码，类加载器间的关系如下：



双亲委派模式是在Java 1.2后引入的，其工作原理是：如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器；如果父类加载器可以完成类加载任务，就成功返回；倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式。

##### 【2】 优势

采用双亲委派模式的好处：

1. Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关系可以避免类的重复加载，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。
2. 考虑到安全因素，java核心api中定义的类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委派模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类已经被加载，并不会重新加载网络传递过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。

##### 【3】 源码分析

<https://blog.csdn.net/javazejian/article/details/73413292>

<https://blog.csdn.net/u014798614/article/details/106758883>

ClassLoader类的loadClass()方法中的逻辑就是双亲委派模式的实现。

#### 缓存机制

##### 【1】 是什么

<https://blog.csdn.net/m0_38075425/article/details/81627349>

<https://cloud.tencent.com/developer/article/1353281>

缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区搜寻该Class，只有当缓存区中不存在该Class对象时，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区中。这就是为什么修改了class文件后，必须重新启动JVM，程序所做的修改才会生效的原因。

##### 【2】 源码分析

<https://blog.csdn.net/javazejian/article/details/73413292>

### 1.3.3 类加载器加载class文件过程

参考：

1. 主：<https://blog.csdn.net/zhangzeyuaaa/article/details/42499839>
2. 辅：

<https://blog.csdn.net/m0_38075425/article/details/81627349>

<https://blog.csdn.net/javazejian/article/details/73413292> <https://cloud.tencent.com/developer/article/1353281>

该过程即ClassLoader.loadClass()代码逻辑。

类加载器加载class文件大致要经过如下步骤：

1. 源ClassLoader先（从缓存）判断该Class是否已加载，如果已加载，则返回Class对象；如果没有则委托给父类加载器。
2. 父类加载器（从缓存）判断是否加载过该Class，如果已加载，则返回Class对象；如果没有则委托给祖父类加载器。
3. 依此类推，直到始祖类加载器（引导类加载器）。
4. 始祖类加载器（从缓存）判断是否加载过该Class，如果已加载，则返回Class对象；如果没有则尝试从其对应的类路径下寻找class字节码文件并载入。如果载入成功，则返回Class对象；如果载入失败，则委托给始祖类加载器的子类加载器。
5. 始祖类加载器的子类加载器尝试从其对应的类路径下寻找class字节码文件并载入。如果载入成功，则返回Class对象；如果载入失败，则委托给始祖类加载器的孙类加载器。
6. 依此类推，直到源ClassLoader。
7. 源ClassLoader尝试从其对应的类路径下寻找class字节码文件并载入。如果载入成功，则返回Class对象；如果载入失败，源ClassLoader不会再委托其子类加载器，而是抛出ClassNotFountException异常。