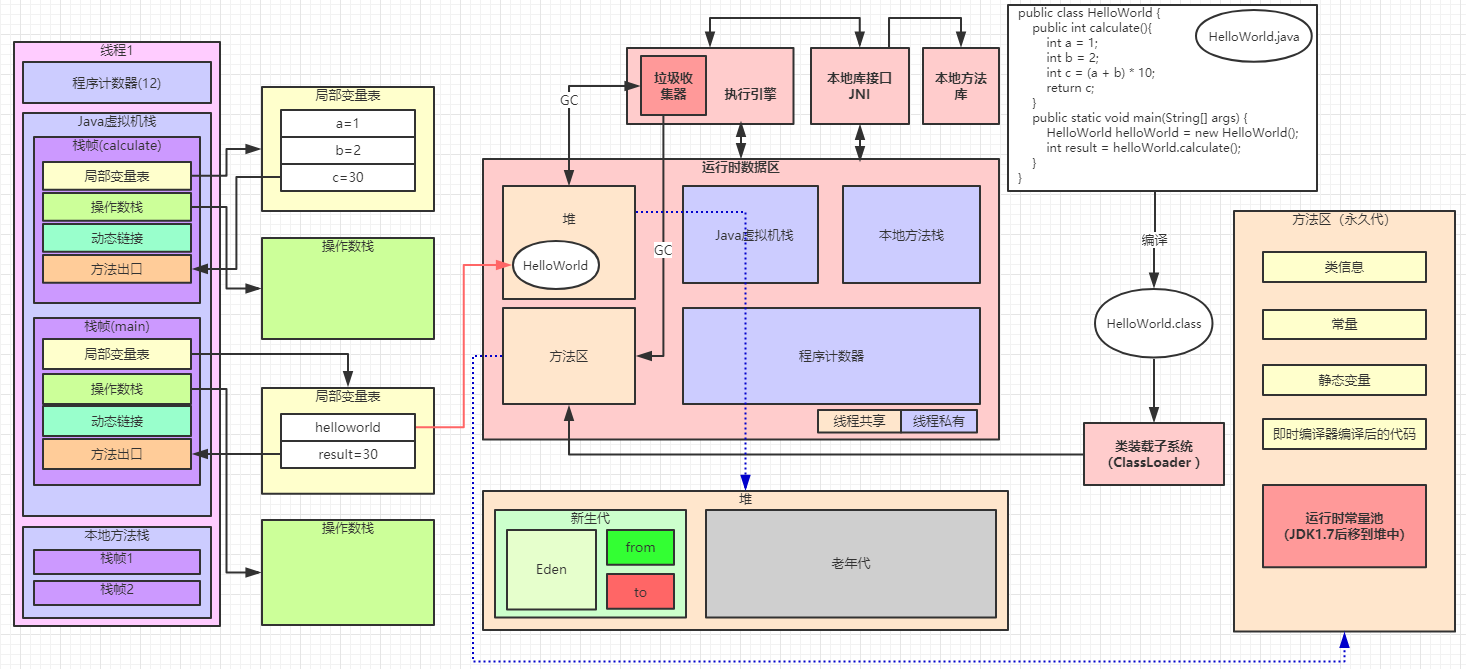
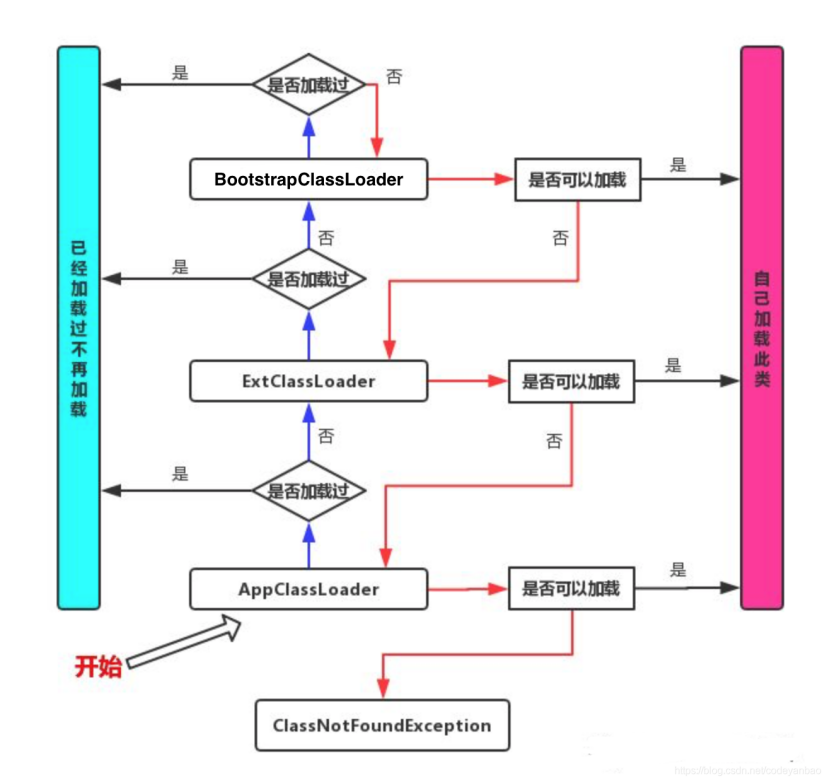
# JVM



## 类加载器

### 双亲委派机制



从上图中我们就更容易理解了，当一个Hello.class这样的文件要被加载时。不考虑我们自定义类加载器，首先会在AppClassLoader中检查是否加载过，如果有那就无需再加载了。如果没有，那么会拿到父加载器，然后调用父加载器的loadClass方法。父类中同理也会先检查自己是否已经加载过，如果没有再往上。注意这个类似递归的过程，直到到达Bootstrap classLoader之前，都是在检查是否加载过，并不会选择自己去加载。直到BootstrapClassLoader，已经没有父加载器了，这时候开始考虑自己是否能加载了，如果自己无法加载，会下沉到子加载器去加载，一直到最底层，如果没有任何加载器能加载，就会抛出ClassNotFoundException。

## native

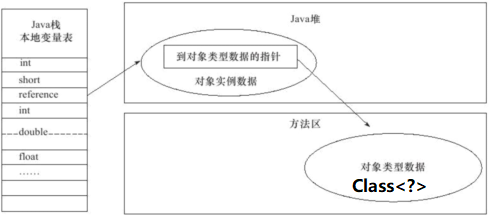
执行底层c语言方法，会进入本地方法栈，调用本地方法接口(JNI).扩展java的使用，融合不同编程语言。

## 程序计数器

是一块较小的内存空间，可以看作当前线程执行字节码的行号指示器。

## 方法区（非堆）

与堆一样是线程共享的区域，用于存储类信息(Class<?>>，常量(final),静态变量(static)，运行时常量池等数据。



## 栈**(线程私有)**

8大基本类型，对象引用，方法出口。每执行的方法都会产生一个栈帧。(见顶图)程序正在执行的方法一定在栈的顶部。

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。每一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入栈到出栈的过程。

## 堆**(线程共享)**

垃圾回收重点区域

### 新生代：轻gc。

### 老年代：full gc。

**大对象直接进入老年代。大对象是指,需要大量连续内存空间的Java对象,最典型的大对象就是那种很长的字符串以及数组。**控制Full GC频率的关键是看应用中绝大多数对象能否符合“朝生夕灭”的原则，即大多数对象的生存时间不应太长，尤其是不能有成批量的、长生存时间的大对象产生，这样才能保障老年代空间的稳定。

### 元空间(方法区)：见四

## GC常用算法：

计数器，复制算法(坏处：需要额外空间-幸存区from、幸存区to)，压缩清除算法(坏处：内存条形化-碎片)

jvm内存调节宗旨：

1. 频繁gc则可能是相应的内存区域分配太小
2. 在gc时频繁扩容，则考虑将最大和最小值设为相等，这样启动后直接分配最大值，不需要运行时扩容了
3. GC后内存无变化，则肯能是程序手动调用System.gc(),加入参数-XX：

+DisableExplicitGC屏蔽掉System.gc（）

可以使用JProfiler分析OOM，运行是使用jvm参数-XX:HeapDumpOnOutOfMemoryError

## JMM - Java Memory Model

内存间交互操作

lock（锁定）：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。

unlock（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放

后的变量才可以被其他线程锁定。

read（读取）：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内

存中，以便随后的load动作使用。

load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工

作内存的变量副本中。

use（使用）：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引

擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。

assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内

存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存

中，以便随后的write操作使用。

write（写入）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入

主内存的变量中。

如果要把一个变量从主内存复制到工作内存，那就要顺序地执行read和load操作，如果

要把变量从工作内存同步回主内存，就要顺序地执行store和write操作。注意，Java内存模型

只要求上述两个操作必须按顺序执行，而没有保证是连续执行。也就是说，read与load之

间、store与write之间是可插入其他指令的，如对主内存中的变量a、b进行访问时，一种可能

出现顺序是read a、read b、load b、load a。除此之外，Java内存模型还规定了在执行上述8种

基本操作时必须满足如下规则

当一个变量定义为volatile之后，它将具备两种特性，第一是保证此变量对所有线程的可

见性，这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以

立即得知的。而普通变量不能做到这一点，普通变量的值在线程间传递均需要通过主内存来

完成，例如，线程A修改一个普通变量的值，然后向主内存进行回写，另外一条线程B在线

程A回写完成了之后再从主内存进行读取操作，新变量值才会对线程B可见

由于volatile变量只能保证可见性，我们仍然要通

过加锁（使用synchronized或java.util.concurrent中的原子类）来保证原子性。

使用volatile变量的第二个语义是禁止指令重排序优化，普通的变量仅仅会保证在该方法

的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的

顺序与程序代码中的执行顺序一致。

除了volatile之外，Java还有两个关键字能实现可见性，即synchronized和final。

Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性

线程的状态

1.新建(New)

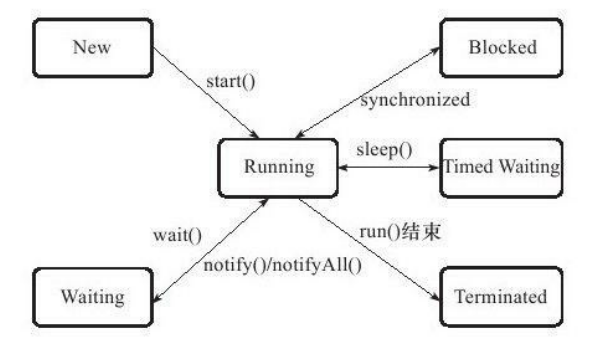
2.运行(Runable)

3.无限期等待(Waiting)

4.限期等待(Timed Waiting)

5.阻塞(Blocked)

6.结束(Terminated)



线程安全

当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交

替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象

的行为都可以获得正确的结果，那这个对象是线程安全的

安全类型：

1. 不可变类型

如 被final修饰的基本变量类型

String和枚举类型

1. 相对线程安全类

例如Vector、HashTable、

Collections的synchronizedCollection（）方法包装的集合等，因为它的add（）、get（）和size（）这类方法都是被synchronized修饰的