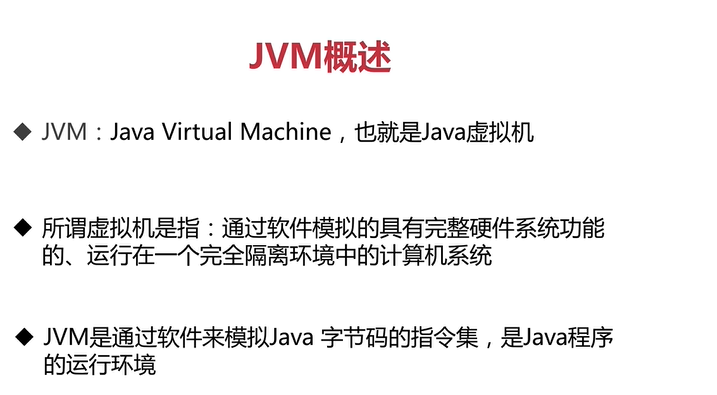
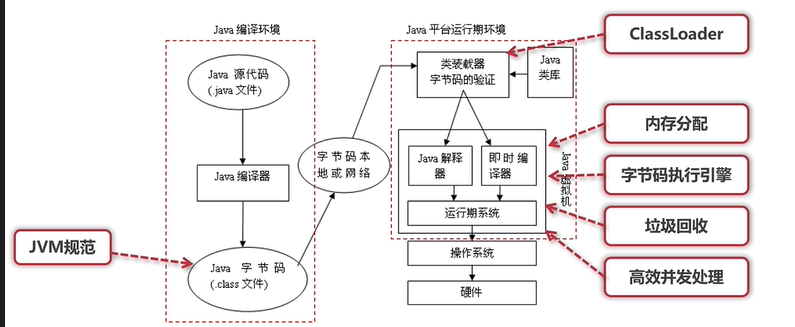
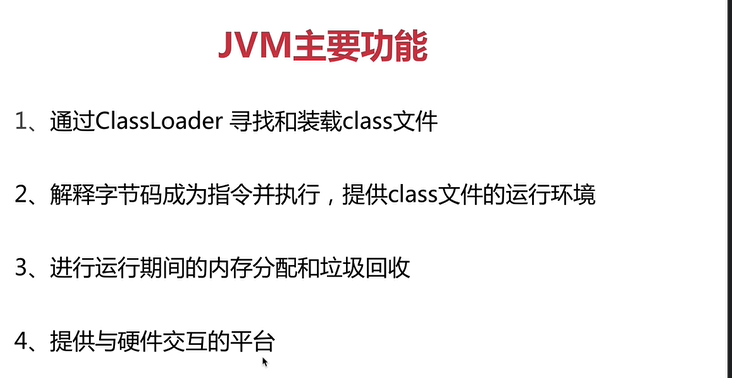
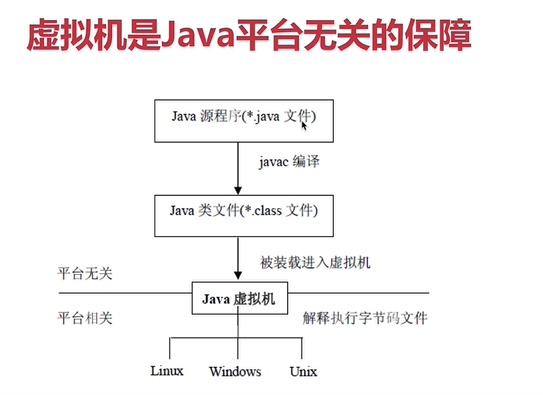
# JVM概述

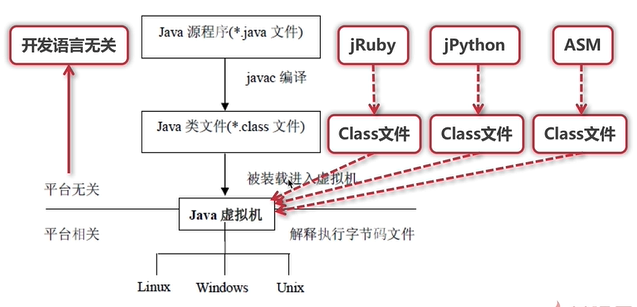




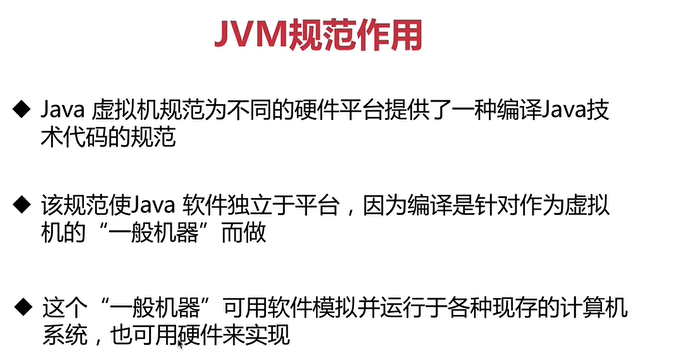




**jvm只认\*.class文件，与开发语言无关**



# JVM规范作用



# ASM开发

## ASM概述：

\* ASM是一个java字节码操纵框架，它能被用来动态生成类或者增强既有类的功能

\* ASM可以直接产生二进制class文件，也可以在类被加载入虚拟机之前动态改变类的行为，ASM从类文件中读入信息后，能够改变类行为，分析类信息，甚至能根据要求生成新类

\* 目前许多框架如cglib，hibernate，spring都直接或间接的使用ASM操作字节码

## ASM编程模型：

\* core api :提供了基于事件形式的变成模型。该模型不需要一次性将整个类的结构读取到内存中，因此这种方式更快，需要更少的内存，但这种编程方式难度较大

\* tree api：提供了基于树形的编程模型。该模型需要一次性将一个类的完整结构全部读取到内存当中，所以这种方法需要更多的内存，这种编程方式较简单

## ASM的Core API-1

\* ASM Core API中操纵字节码的功能基于CLassVisitor接口，这个接口中的每个方法对应了class文件中的每一项

\* ASM提供了三个基于ClassVisitor接口的类来实现class文件的生成和转换

1：classReader解析一个类的class字节码

2：ClassAdapter：ClassAdapeter是ClassVisitor的实现类

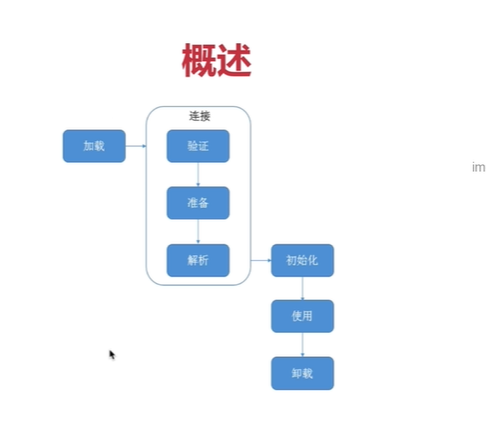
3：ClassWriter：ClassWriter也是ClassVisitor的实现类，可以用来输出变化后的字节码

\*ASM给我们提供了ASMifier工具来帮助开发，可使用ASMigier工具生成ASM结构来对比

# 类加载，连接和初始化

## 类加载和类加载器

类从被加载到jvm开始到卸载出内存，整个生命周期如图



\* 加载：查找并加载类文件的二进制数据

\* 连接：将已经读入内存的类的二进制数据合并到jvm运行时环境中去，包含如下几个步骤

1验证：确保被加载类的正确性

2准备：为类的静态变量分配内存，并初始化他们

3 解析：把常量池中的符号引用转换成直接引用

\* 初始化：为类的静态变量赋初始值

### 类加载要完成的功能

1 通过类的全限定名来获取该类的二进制字节流

2 把二进制字节流转化为方法区的运行时数据结构

3 在堆上创建一个java.lang.Class对象，用来封装类在方法区内的数据结构，并向外提供了访问方法区内数据结构的接口

### 加载类的方式

\* 最常见的方式：本地文件系统中加载，从jar等归档文件中加载

\* 动态的方式：将java源文件动态编译成class

\* 其他方式：网络下载，从专有数据库中加载等等

### 类加载器

java虚拟机自带的加载器包括如下几种：

启动类加载器（BootstrapClassLoader）

扩展类加载器 （ExtensionClassLoader）

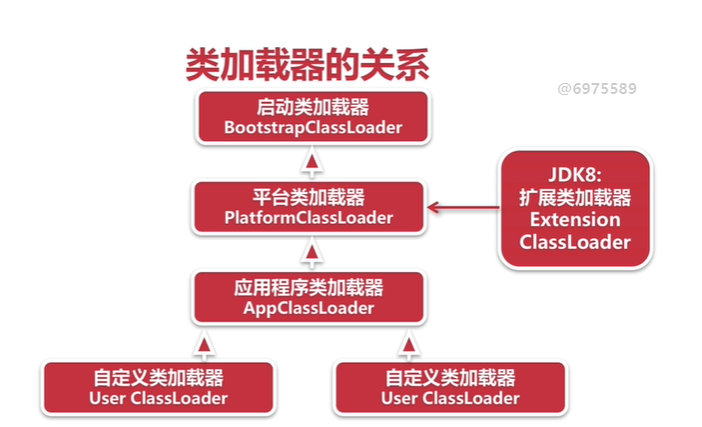
应用程序类加载器（AppClassLoader）

java9之后为了支持“模块化”将扩展类加载器改为平台类加载器（PlatformClassLoader）

用户自定义的加载器，是java.lang.ClassLoader的子类

用户可以定制类的加载方式；只不过自定义类加载器是在所有系统类加载器的最后

### 类加载器的关系



\*启动类加载器：用于加载启动的基础模块类，比如：java.base，java.management,java.xml等等

\*平台类加载器：用于加载一些平台相关的模块，比如：java.scripting,java.compiler\*,java.corba\*等等

\*应用程序加载器：用于加载应用级别的模块，比如：jdk.compiler,jdk.jartool,jdk.jshell等等；还加载classpath路径中的所有类库

\*JDK8：启动类加载器：负责将<JAVA\_HOME>/lib，或者Xbootclasspath参数指定的路径中的，且是虚拟机识别的类库加载到内存中（按照名字识别，比如rt.jar，对于不能识别的文件不予装载）

\*JDK8：扩展类加载器：负责加载<JRE\_HOME>/lib/ext,或java.ext.dirs系统变量所指定路径中的所有类库

\*JDK8：应用程序类加载器：负责加载classpath路径中的所有类库

\*java程序不能直接引用启动类加载器，直接设置classloader为null，默认就使用启动类加载器

\*类加载器并不需要等到某个类“首次主动使用”的时候才加载它，jvm规范允许类加载器在预料到某个类将要被使用的时候就预先加载它

\*如果在加载的时候.class文件缺失，会在改类首次主动使用的时候报告linkageError错误，如果一直没有被使用，就不会报错

## 双亲委派模型

### 双亲委派模型的原理

双亲委派模式是在Java 1.2后引入的，其工作原理的是，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的加载器去执行，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

### 双亲委派模型的优势

· 采用双亲委派模式的好处是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。

· 其次是考虑到安全因素，java核心api中定义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，通过双亲委托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，发现该类已被加载，并不会重新加载网络传递的过来的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。

· 可能你会想，如果我们在classpath路径下自定义一个名为java.lang.SingleInterge类(该类是胡编的)呢？该类并不存在java.lang中，经过双亲委托模式，传递到启动类加载器中，由于父类加载器路径下并没有该类，所以不会加载，将反向委托给子类加载器加载，最终会通过系统类加载器加载该类。但是这样做是不允许，因为java.lang是核心API包，需要访问权限，强制加载将会报出异常

### 双亲委派模型说明

1 双亲委派模型对于保证java程序的稳定运作很重要

2 实现双亲委派的代码在java.lang.ClassLoader的loadClass()方法中，如果自定义类加载器的话，推荐覆盖实现findClass()

3 如果有一个类加载器能加载某个类，称为定义类加载器，所有能成功返回该类的Classs的类加载器都被称为初始类加载器

4 如果没有指定父加载器，默认就是启动加载器

5 每个类加载器都有自己的命名空间，命名空间由该加载器及其所有父加载器所加载的类构成，不同的命名空间，可以出现类的全路径名相同的情况

6 运行时包由同一个类加载器的类构成，决定两个类是否属于同一个运行时包，不仅要看全路径名是否一样，还要看定义类加载器是否相同。只有属于同一个运行时包的类才能实现相互包内可见

### 破坏双亲委派模型

\* 双亲模型有个问题：父加载器无法向下识别子加载器加载的资源

\* 为了解决这个问题，引入了线程上下文类加载器，可以通过Thread的setContextClassLoader（）进行设置

\* 另外一种典型情况就是实现热替换，比如OSGI的模块化热部署，它的类加载器就不再是严格按照双亲委派模型，很多可能就在平级的类加载器中执行了

## 类连接和初始化

### 类验证的主要验证的内容

\* 类文件结构检查：按照jvm规范规定的类文件结构进行

\* 元数据验证： 对字节码描述的信息进行语义分析，保证其符合java语言规范要求

\* 字节码验证： 通过对数据流和控制流进行分析，确保程序语义是合法和符合逻辑的。这里主要对方法进行校验

\* 符号引用验证：对类自身以外的信息，也就是常量池中的各种符号引用，进行匹配校验

### 类连接中的解析

\* 所谓解析就是把常量池中的符号引用转换成直接引用的过程，包括:符号引用：以一组无歧义的符号来描述锁引用的目标，与虚拟机的实现无关

\* 直接引用：直接指向目标的指针，相对偏移量，或是能间接定位到目标的句柄，是和虚拟机实现相关的

\* 主要针对：类，接口，字段，类方法，接口方法，方法类型，方法句柄，调用点限定符

### 类的初始化

\* 类的初始化就是为类的静态变量赋初始值，或者说是执行类构造器<clinit>方法的过程

1 如果类还没有加载和连接，就先加载和连接

2 如果类存在父类，且父类没有初始化，就先初始化父类

3 如果类中存在初始化语句，就依次执行这些初始化语句

4 如果是接口的话：

1. 初始化一个类的时候，并不会先初始化它实现的接口
2. 初始化一个接口时，并不会初始化它的父接口
3. 只有程序首次使用接口里面的变量或者是调用接口方法的时候，才会导致接口初始化

5 调用Classloader类的loadClass方法来装载一个类，并不会初始化这个类，不是对类的主动使用

### 类的初始化时机

\* java 程序对类的使用方式分成：主动使用和别动使用，JVM必须在每个类或接口“首次主动使用”时才初始化他们；被动使用类不会导致类的初始化，主动使用的情况：

1 创建类实例

2 访问某个类或接口的静态变量

3 调用类的静态方法

4 反射某个类

5 初始化某个类的子类，而父类还没有初始化

6 JMV启动的时候运行的主类

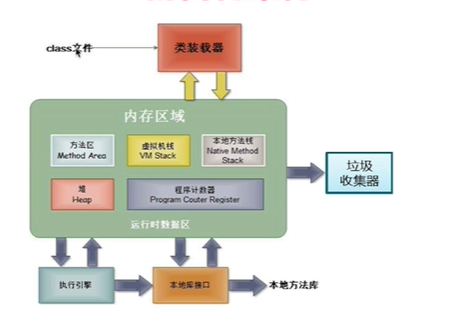
7 定义了default方法的接口，当接口实现类初始化时

### 类的卸载

\* 当代表一个类的Class对象不再被引用，那么Class对象的生命周期就结束了，对应的在方法区中的数据也会被卸载

\* JVM自带的类加载器装载的类，是不会卸载的，由用户自定义的类加载器加载的类是可以卸载的

# JVM内存分配



## JVM的简化架构和运行时数据区

### 运行时数据区

包括：PC寄存器，Java虚拟机栈，Java堆，方法区，运行时常量池，本地方法栈等

\* PC(Program Counter)寄存器说明：

1 每个线程拥有一个PC寄存器，是线程私有的，用来存储指向下一条指令的地址

2 在创建线程的时候，创建响应的PC寄存器

3 执行本地方法是，PC寄存器的值为undefined

4 是一块较小的内存空间，是唯一一个在JVM规范中没有规定OutOfMemoryError的内存区域（不会内存溢出）

\* Java栈

1 栈是由一系列帧组成（因此Java栈也叫做帧栈），是线程私有的

2 帧用来保存一个方法的局部变量，操作数栈（java没有寄存器，所有参数传递使用操作数栈），常量池指针，动态链接，方法返回值等

3 每一次方法调用创建一个帧，并压栈，退出方法的时候，修改栈顶指针就可以把栈帧中的内容销毁

4 局部变量表存放了编译期可知的各种基本数据类型和引用类型，每个slot存放32位的数据，long,double占两个槽位

5 栈的优点：存取速度比堆快，仅次于寄存器

6 栈的缺点：存在栈中的数据大小，生存期是编译期决定的缺乏灵活性

\*Java 堆

1 用来存放应用系统创建的对象和数组，所有线程共享java堆

2 GC主要就管理堆空间，对分代GC来说，堆也是分代的

3 堆的优点：运行期动态分配内存大小，自动进行垃圾回收；

堆的缺点：效率相对较慢

\* 方法区

1 方法区是线程共享的，通常用来保存装载的类的结构信息

2 通常和元空间关联在一起，但具体的跟JVM实现和版本有关

3 JVM规范把方法区描述为堆的一个逻辑部分，但它有个别名为Non-heap（非堆），应是为了与java堆区分开

\* 运行时常量池

1 是class文件中每个类或接口的常量池表，在运行期间的表示形式，通常包括：类的版本，字段，方法，接口等信息

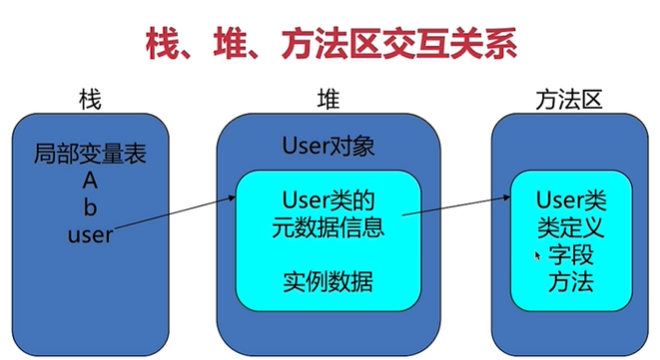
2 在方法区中分配

3 通常在加载类和接口到JVM后，就创建响应的运行时常量池

\* 本地方法栈

1 在JVM中用来支持native方法执行的栈就是本地方法栈

### 栈 堆 方法区交互关系



## Java堆内存模型和分配

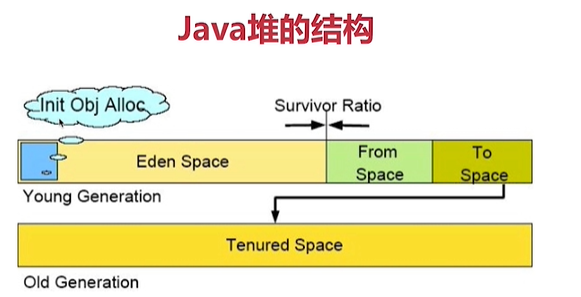
### java堆内存概述

1 java堆用来存放应用系统创建的对象和数组，所有线程共享java堆

2 java堆是在运行期动态分配内存大小，自动进行垃圾回收

3 java垃圾回收（GC）主要就是回收堆内存，对分代GC来说，堆也是分代的

### java堆的结构



\* 新生代用来放新分配的对象；新生代中经过垃圾回收，没有回收掉的对象，被复制到老年代

\* 老年代存储对象比新生代存储对象的年龄大得多

\* 老年代存储一些大对象

\* 整个堆大小 = 新生代+老年代

\* 新生代= eden+存活区

\* 从前的持久代，用来存放Class，Method等元信息的区域，从JDK8开始去掉了，取而代之的是元空间，元空间并不在虚拟机里面，而是直接使用本地内存

### 对象的内存布局

\* 对象的内存中存储的布局（这里以Hotspot虚拟机为例来说明），分为：对象头，实例数据和对齐填充

\* 对象头包含两部分：

1 Mark Word：存储对象自身的运行数据，如：HashCode,GC分代年龄，锁状态标志等

2 类型指针：对象指向它的类元数据的指针

\* 实例数据：真正存放对象示例数据的地方

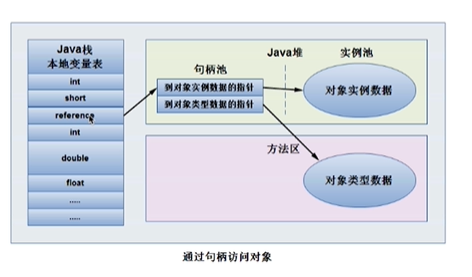
\* 对齐填充：这部分不一定存在，也没有什么特别的含义，仅仅是占位符，因为HotSpot要求对象起始地址都是8字节的整数倍，如果不是，就对齐

### 对象的访问定位

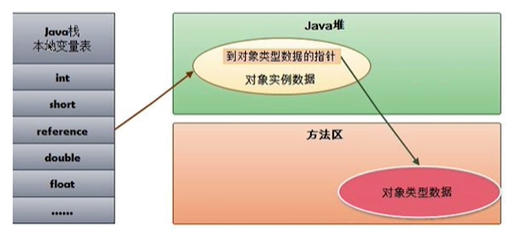
\* 对象的访问定位:在jvm规范中只规定了reference类型是一个指向对象的引用，但没有规定这个引用具体如何去定位，访问堆中对象的具体位置

\* 因此对象的访问方式取决于JVM的实现，目前主流的有：使用句柄或使用指针两种方式

\* 使用句柄：java堆中会划分出一块内存来作为句柄池，reference中存储句柄的地址句柄中存储对象的实例数据和类元数据的地址，如下图所示：



\* 使用指针：java堆中会存放访问类元数据的地址，reference存储的就直接是对象的地址，如下图所示



\* 使用指针的方式比使用句柄的方式要更快

## Trace跟踪和Java堆的参数配置

### Trace跟踪参数

\* 可以打印GC的简要信息： -Xlog:gc

\* 打印GC详细信息： -Xlog:gc\*

\* 指定GC log的位置，以文件输出 -Xlog:gc:garbage-collection.log

\* 每一次GC后，都打印堆信息：-Xlog:gc+heap=debug

### GC日志格式

\* GC发生的时间，也就是JVM从启动以来经过的秒数

\* 日志级别信息和日志类型标记

\* GC识别号

\* GC类型和说明GC的原因

\* 容量：GC前容量->GC后容量（该区域总容量）

\* GC持续时间，单位秒。有的收集器会有更详细的描述，比如：user表示应用程序消耗的时间，sys表示系统内核消耗的时间，real表示操作从开始到结束的时间

## Java堆参数

\* Xms：初始堆大小，默认物理内存的1/64

\* Xmx: 最大堆大小，默认为物理内存的1/4

\* Xmn: 新生代大小，默认整个堆的3/8

\* -XX:+HeapDumpOnOutOfMemoryError:OOM时导出堆到文件

\* -XX:+HeapDumpPath:导出OOM的路径

\* -XX:OnOutOfMemoryError:在OOM时，执行一个脚本

\*-XX:NewRatio:老年代和新生代的比值

如果xms-xmx，且设置了xmn的情况下，该参数不用设置

\*-XX:SurvivorRatio:Eden区和Survivor区的大小比值，设置为8，则两个Survivor去与一个Eden区的比值为2:8，一个Survivor占整个新生的1/10

## Java栈的参数

\*-Xss:通常只有几百k，决定了函数调用的深度

## 元空间的参数

-XX:MetaspaceSize:初始空间大小

-XX:MaxMetaspaceSize:最大空间，默认是没有限制的

-XX:MinMetaspaceFreeRatio:在GC之后，最小的Metaspace剩余空间容量的百分比

-XX:MaxMetaspaceFreeRatio:在GC之后，最大的Metaspace剩余空间容量的百分比

# 字节码执行引擎

## 字节码执行引擎概述

\* JVM的字节码执行引擎，功能基本就是输入字节码文件，然后对字节码进行解析并处理，最后输入执行的结果。

\*实现方式可能有通过解释器直接解释执行字节码，或者是通过即时编译器产生本地代码，也就是编译执行，当然也可能两者皆有

## 栈帧概述

\*栈帧是用于支持JVM进行方法调用和方法执行的数据结构

\*栈帧随着方法调用而创建，随着方法结束而销毁

\*栈帧里面存储了方法的局部变量，操作数栈，动态连接，方法返回地址等信息

#### 栈帧的概念结构



#### 局部变量表

\* 局部变量表：用来存放方法参数和方法内部定义的局部变量的存储空间

1 以变量槽slot为单位，目前一个slot存放32位以内的数据类型

2 对于64位的数据栈两个slot

3 对于实例方法，第0位slot存放的是this，然后从1到n，依次分配给参数列表

4 根据方法体内部定义的变量顺序和作用域来分配slot  
· 5 slot是复用的，以节省栈帧的空间，这种设计可能会影响到系统的垃圾收集行为

#### 操作数栈

\*操作数栈：用来存放方法运行期间，各个指令操作的数据

1 操作数栈中的元素的数据类型必须和字节码指令的顺序严格匹配

2 虚拟机在实现栈帧的时候可能会做一些优化，让两个栈帧出现部分重叠区域，以存放公用的数据

#### 动态连接

\* 动态连接：每个栈帧持有一个指向运行时常量池中该栈帧所属方法的引用，以支持方法调用过程的动态连接

1 静态解析：类加载的时候，符号引用就转化成直接引用

2 动态连接：运行期间转换为直接引用

#### 方法返回地址

\* 方法返回地址：方法执行后返回的地址

#### 方法调用

\* 方法调用：方法调用就是确定具体调用哪一个方法，并不涉及方法内部的执行过程

1 部分方法是直接在类加载的解析阶段，就确定了直接引用关系

2 但是对于实例方法，也成虚方法，因为重载和多态，需要运行期动态委派

#### 分派

\*分派：又分为静态分派和动态分派

1 静态分派：所有依赖静态类型来定位方法执行版本的分派方式。比如：重载方法

2 动态分派：根据运行期的实际类型来定位方法执行版本的分派方式，比如：覆盖方法

\*单分派和多分派：就是按照分派思考的维度，多余一个的就算多分派，只有一个的成为单分派

\*如何执行方法中的字节码指令：JVM通过基于栈的字节码解释执行引擎来执行指令，JVM的指令集也是基于栈的

# 垃圾回收基础和根搜索算法

## 垃圾回收概述

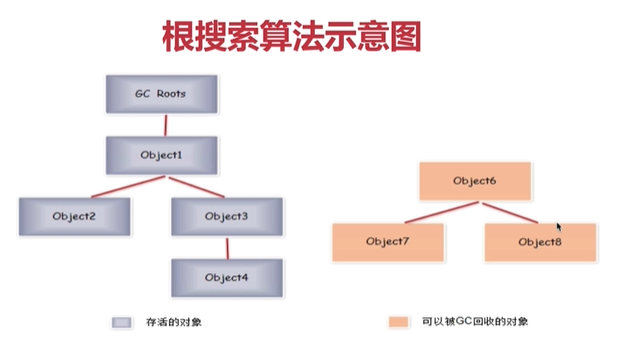
\* 什么是垃圾：简单说就是内存中已经不再被使用到的内存空间就是垃圾

\* 引用计数法：给对象添加一个引用计数器，有访问就加1，引用失效就减1

\* 优点：实现简单，效率高；缺点：不能解决对象之间循环引用的问题

## 根搜索算法

\* 从根(GC Roots) 节点向下搜索对象节点，搜索走过的路径称为引用链，当一个对象到根之间没有连通的话，则该对象不可用



\* 可作为GC Roots的对象包括：虚拟机栈（栈帧局部变量）中引用的对象，方法区类静态属性引用的对象，方法区中常量引用的对象，本地方法栈中JNI引用的对象

\* HotSpot使用了一组叫做OopMapde数据结构达到准确式GC的目的

\*在OopMap的协助下，JVM可以很快的做完GC Roots枚举。但是JVM并没有为每一条指令生成一个OopMap

\*记录OopMap的这些“特定位置”被称为安全点，即当前线程执行到安全点后才允许暂停进行GC

\*如果一段代码中，对象引用关系不会发生变化，这个区域中任何地方开始GC都是安全的，那么这个区域成为安全区域

## 引用分类

\* 强引用：类似于Object a = new A()这样的，不会被回收

\* 软引用：还有用单并不必须的对象，用SoftReference来实现软引用

\* 弱引用：非必须对象，比软引用还要弱，垃圾回收时会回收掉。用WeakReference来实现弱引用

\* 虚引用：也称为幽灵引用和幻影引用，是最弱的引用。垃圾回收时会回收掉，用PhantomReference来实现虚引用

## 跨代引用

跨代引用：也就是一个代中的对象引用另一个代中的对象

跨代引用假说：跨代引用相对于同代引用来说只是极少数

隐含推论：存在互相引用关系的两个对象，是应该倾向于同时生存或同时消亡的

## 记忆集

记忆集：一种用于记录从非收集区域指向收集区域的指针集合的抽象数据结构

字长精度：每个记录精确到一个机器字长，该字包含跨代指针

对象精度：每个记录精确到一个对象，该对象里有字段含有跨代指针

卡精度：每个记录精确到一块内存区域，该区域内有对象含有跨代指针

卡表：是记忆集的一种具体实现，定义了记忆集的记录精度和与堆内存的映射关系等

卡表的每个元素都对应着标识的内存区域中一块特定大小的内存块，这内存块称为卡页

## 写屏障

写屏障可以看成是JVM对“引用类型字段赋值”这个动作的AOP

通过写屏障来实现当对象状态改变后，维护卡表状态

## 判断是否垃圾的步骤

1 跟搜索算法判断不可用

2 看是否有必要执行finalize方法

3 两个步骤走完后对象仍然没有人使用，那就属于垃圾

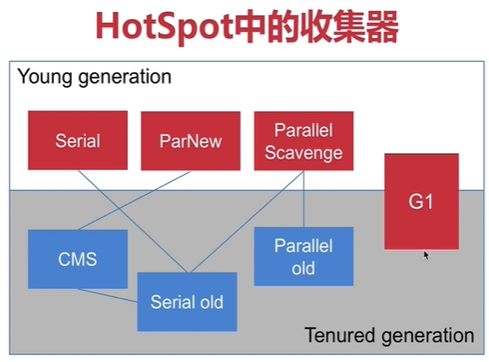
## 垃圾收集器基础和串行收集器

\*垃圾收集器：串行收集器，并行收集器，新生代Parallel Scavenge收集器，CMS,G1

\*GC性能指标和JVM内存配置原则

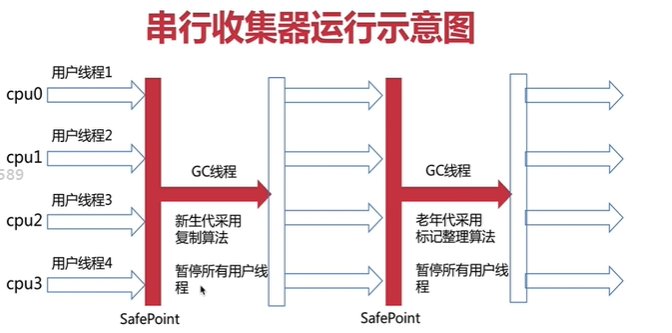
\*垃圾收集算法只是内存回收的方法，垃圾收集器就来具体实现这些算法并实现内存回收

\*不同厂商，不同版本的虚拟机实现差别很大，HotSpot中包含的收集器如下图所示



### 串行收集器

\* Serial（串行）收集器/Serial Old收集器，是一个单线程的收集器，在垃圾收集时，会Stop-the-World



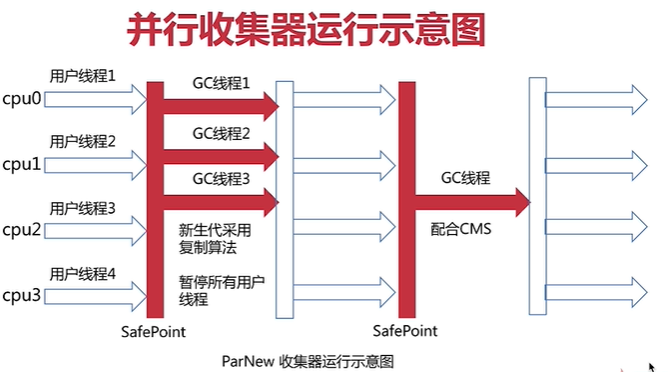
\* 优点是简单，对于单cpu，由于没有多线程的交互开销，可能更高效，是默认的Client模式下的新生代收集器

\* 使用—XX:+UseSerialGC来开启，会使用：Serial+Serial Old的收集器组合

\* 新生代使用复制算法，老年代使用标记整理算法

### 并行收集器

\*ParNew(并行)收集器：使用多线程进行垃圾回收，在垃圾收集时，会Stop-the-World



\* 在并发能力好的CPU环境中，它停顿的时间要比串行收集器短；但对于单cpu或并发能力较弱的CPU，由于多线程的交互开销，可能比串行收集器更差

\*是Server模式下首选的新生代收集器，且能和CMS收集器配合使用

\*不再使用-XX:+UserParNewGC来单独开启

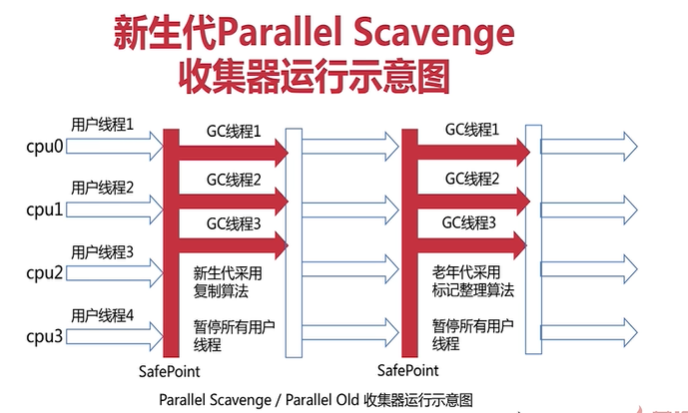
\*-XX:ParallelGCThreads:指定线程数，最好与CPU数量一致

\*新生代使用复制算法

### 新生代Parallel Scavenge收集器

\* 新生代Parallel Scavenge收集器/Parallel Old收集器：是一个应用于新生代的，使用复制算法的，并行的收集器

\* 跟ParNew很类似，但更关注吞吐量，能最高效率的利用CPU，适合运行后台应用



\* 使用-XX:+UseParallelGC来开启

\* 使用-XX:+UseParallelOldGC来开启老年代使用ParallelOld收集器，使用Parallel Scavenge + Parallel Old的收集器组合

\* -XX:MaxGCPauserMillis:设置GC的最大停顿时间

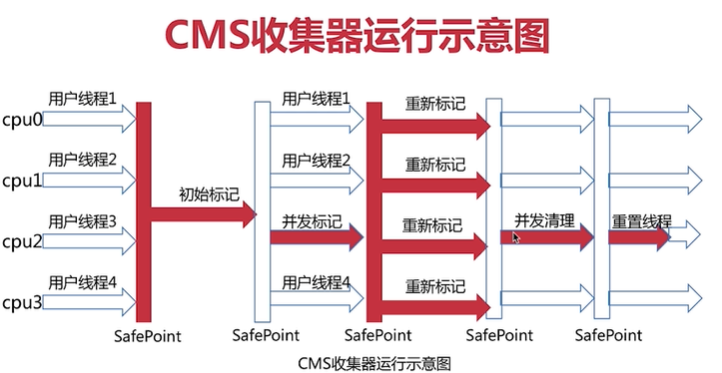
\* 新生代使用复制算法，老年代使用标记整理算法

### CMS收集器

\* CMS（Concurrent Mark and Sweep并发标记清除）收集器分为：初始标记：只标记GC Roots能直接关联到的对象； 并发标记：进行GC Roots Tracing的过程

\* 重新标记：修正并发标记期间，因程序运行导致标记发生变化的那一部分对象

\* 并发清除：并发回收垃圾对象



\* 在初始标记和重新标记两个阶段还是会发生Stop-the-world

\* 使用标记清除算法，多线程并发收集的垃圾收集器

\* 最后的重置线程，指的是清空跟收集相关的数据并重置，为下一次收集做准备

\* 优点：低停顿，并发执行

\* 缺点：

并发执行，对CPU资源压力大

无法处理 在处理过程中产生的垃圾，可能导致FullGC

采用的标记清除算法会导致大量碎片，从而在分配大对象时可能触发FullGC

\* 开启：-XX:UseConcMarkSweepGC:使用ParNew+CMS+Serial Old的收集器组合，Serial Old将作为CMS出错的后备收集器

\* -XX:CMSInitiatingOccupancyFraction:设置CMS收集器在老年代空间被使用多少后触发回收，默认80%

### G1收集器

\* G1（Garbage-First）收集器：是一款面向服务端应用的收集器，与其他收集器相比，具有如下特点：

1 G1把内存划分为多个独立的区域（Region）

2 G1仍采用分代思想，保留了新生代和老年代，但他们不再是物理隔离的，而是一部分Region的集合，且不需要Region是连续的

3 G1能充分利用多CPU,多核环境硬件优势，尽量缩短STW

4 G1整体上采用标记整理算法，局部是通过复制算法，不会产生内存碎片

5 G1的停顿可预测，能明确指定在一个时间段内，消耗在垃圾收集上的时间不能超过多长时间

6 G1跟踪各个Region里面垃圾堆的价值大小，在后台维护一个优先列表，每次根据允许的时间来回收价值最大的区域，从而保证在有限的时间内的高效收集

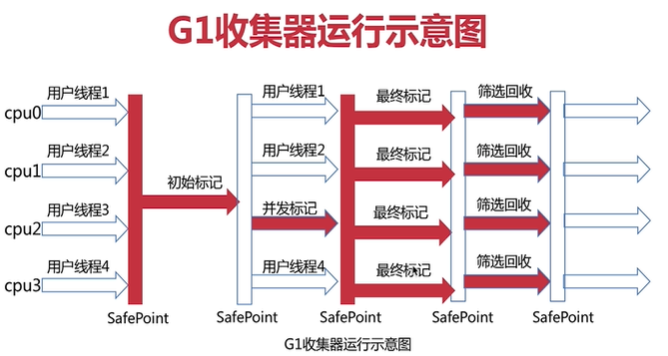
\* 跟CMS类似，也分为四个阶段：

初始标记：只标记GC Roots能直接关联到的对象

并发标记：进行GC Roots Tracing的过程

最终标记：修正并发标记期间，因程序运行导致标记发生变化的那一部分对象

筛选回收：根据时间来进行价值最大化的回收



\* 使用和配置G1: -XX:+UseG1GC:开启G1，默认就是G1

\* -XX:MaxGCPauseMillis=n:最大GC停顿时间，这是个软目标，JVM将尽可能（但不保证）停顿小于这个时间

\* -XX:InitiatingHeapOccupancyPercent=n:堆占用了多少的时候就触发GC，默认为45

\* -XX:NewRatio=n:默认为2

\* -XX:SurvivorRatio=n: 默认为8

\* -XX:MaxTenuringThreshold=n:新生代到老年代的岁数，默认为15

\* -XX:ParallelGCThreads=n:并发GC的线程数，默认值会根据平台不同而不同

\* -XX:ConcGCThreads=n:并发GC使用的线程数

\* -XX:G1ReservePercent=n:设置作为空闲空间的预留内存百分比，以降低目标空间溢出的风险，默认值是10%

\* -XX:G1HeapRegionSize=n:设置的G1区域的大小。值是2的幂，范围是1MB到32MB。目标是根据最小的java堆大小划分出约2048个区域

## GC类型

\* MinorGC/YoungGC:发生在新生代的收集动作

\*MajorGC/OldGC:发生在老年代的GC，目前只有CMS收集器会有单独收集老年代的行为

\*MixedGC:收集整个新生代以及部分老年代，目前只有G1收集器会有这种行为

\*FullGC：收集整个Java堆和方法区的GC

## Stop-The-World

\* STW是java中一种全局暂停的现象，多半由于GC引起。所谓全局停顿，就是所有java代码停止运行，native代码可以执行，但不能和jvm交互

\* 其危害是长时间服务停止，没有相应；对于HA系统，可能引起主备切换，严重危害生产环境

## 垃圾收集类型

\*串行收集：GC单线程内存回收,会暂停所有的用户线程，如：Serial

\*并行收集：多个GC线程并发工作，此时用户线程是暂停的如：Parallel

\*并发收集：用户线程和GC线程同时执行（不一定是并行，可能交替执行），不需要停顿用户线程。如：CMS

## 判断类无用的条件

\*JVM中该类的所有实例都已经被回收

\*加载该类的ClassLoader已经别回收

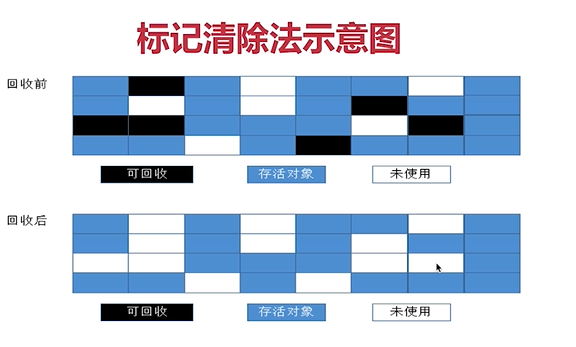
\*没有任何地方引用该类的Class对象

\*无法在任何地方通过反射访问这个类

## 垃圾回收算法

### 标记清除法

\*标记清除法（Mark-Sweep）算法分成标记和清除两个阶段，先标记出要回收的对象，然后统一回收这些对象



\* 优点是简单

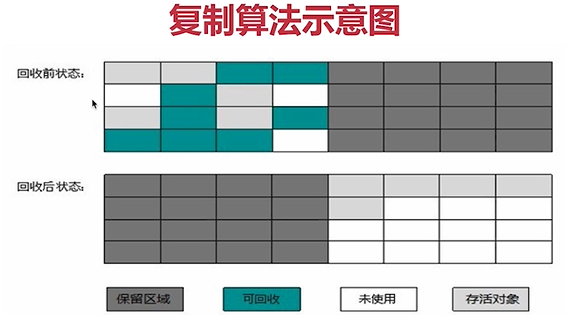
\* 缺点是：

1 效率不高，标记和清除的效率都不高

2 标记清除后会产生大量不连续的内存碎片，从而导致在分配大对象时触发GC

### 复制算法

\*复制算法（Copying）:把内存分成两块完全相同的区域，每次使用其中一块儿，当一块使用完了，就把这块上还存活的对象拷贝到另外一块，然后把这块清除掉



\* 优点是：实现简单，运行高效，不用考虑内存碎片问题

\* 缺点是：内存有些浪费

\* JVM实际实现中，是将内存分为一块较大的Eden区和两块较小的Survivor空间，每次使用Eden和一块Survivor，回收时，把存活的对象复制到另一块Survivor

\* HotSpot默认的Eden和Survivor比是8:1，也就是每次能用90%的新生代空间

\* 如果Survivor空间不够，就要依赖老年代进行分配担保，把放不下的对象直接进入老年代

### 分配担保

\*分配担保是：当新生代进行垃圾回收后，新生代的存活区放不下，那么需要把这些对象放置到老年代去的策略，也就时老年代为新生代的GC做空间分配担保，步骤如下：

1 在发生MinorGC前，JVM会检查老年代的最大可用的连续空间，是否大于新生代所有对象的总空间，如果大于，可以确保MinorGC是安全的

2 如果小于，那么JVM会检查是否设置了允许担保失败，如果允许，则继续检查老年代最大可用的连续空间，是否大于历次晋升到老年代对象的平均大小  
 3 如果大于，则尝试进行一次MinorGC

4 如果不大于，则改做一次Full GC

### 标记整理法

\* 标记整理算法(Mark-Compact):由于复制算法在存活对象比较多的时候，效率较低，且有空间浪费，因此老年代一般不会选用复制算法，老年代多选用标记整理算法

\* 标记过程跟标记清除一样，但后续不是直接清除可回收对象，而是让所有存活对象都向一端移动，然后直接清除边界以外的内存



## ZGC收集器

\* ZGC收集器：JDK11加入的具有实验性质的低延迟收集器

\* ZGC的设计目标是：支持TB级内存容量，暂停时间低（<10ms），对整个程序吞吐量的影响小于15%

\*ZGC里面的新技术：着色指针和读屏障

## GC性能指标

\* 吞吐量=应用代码执行的时间/运行的总时间

\* GC负荷，与吞吐量相反，是GC时间/运行的总时间

\* 暂停时间，就是发生Stop-the-World的总时间

\* GC性能指标:GC频率，就是GC在一个时间段发生的次数

\* 反应速度，就是从对象成为垃圾到被回收的时间

\* 交互式应用通常希望暂停时间越少越好

## JVM内存配置原则

\*新生代尽可能设置大点，如果太小会导致：

1.YGC次数更加频繁

2.可能导致YGC后的对象进入老年代，如果此时老年代满了，会触发FGC

\*对老年代，针对响应时间优先的应用：由于老年代通常采用并发收集器，因此其大小要综合考虑并发量和并发持续时间等参数

\*如果设置小了，可能会造成内存碎片，高回收频率会导致应用暂停

\*如果设置大了，会需要较长的回收时间

\*对老年代，针对吞吐量优先的应用：通常设置较大的新生代和较小的老年代，这样可以尽可能回收大部分短期对象，减少中期对象，而老年代尽量存放长期存活的对象

\*依据对象的存活周期进行分类，对象优先在新生代分配，长时间存活的对象进入老年代

\*根据不同的特点，选取合适的收集算法：少量对象存活，适合复制算法：大量对象存活，适合标记清除或者标记整理

# 高效并发

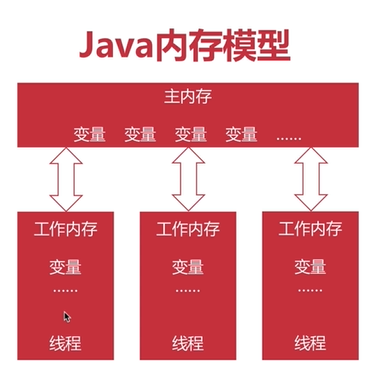
## Java内存模型和内存间的交互操作

### java内存模型

\*JCP定义了一种java内存模型，以前是在JVM规范中，后来独立出来成为JSR-133（java内存模型和线程规范修订）

\*内存模型：在特定的操作协议下，对特定的内存或高速缓存进行读写访问的过程抽象

\*java内存模型主要关注JVM中把变量存储到内存和从内存中取出变量值这样的底层细节



\*所有变量（共享的）都存储在主内存中，每个线程都有自己的工作内存；工作内存中保存该线程使用到的变量的主内存副本拷贝

\* 线程对变量的所有操作（读，写）都应该在工作内存中完成

\* 不同线程不能相互访问工作内存，交互数据要通过主内存

### 内存间的交互操作

\* java内存模型规定了一些操作来实现内存间交互，JVM会保证他们是原子的

\* lock：锁定，把变量标识为线程独占，作用于主内存变量

\* unlock:解锁，把锁定的变量释放，别的线程才能使用，作用于主内存变量

\* read:读取，把变量值从主内存读取到工作内存

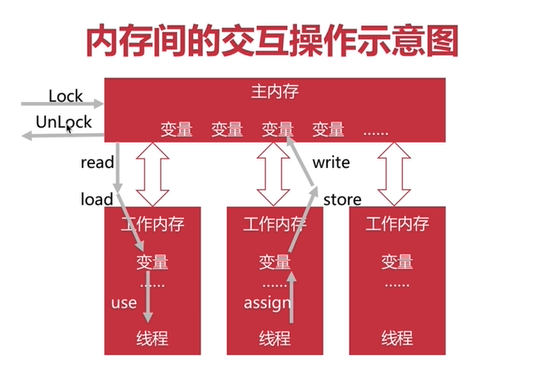
\* load：载入，把read读取到的值放入工作内存的变量副本中

\* use: 使用，把工作内存中一个变量的值传递给执行引擎

\* assign :赋值，把从执行引擎接收到的值赋给工作内存里面的变量

\* store:存储，把工作内存中一个变量的值传递到内存中

\* write:写入，把store进来的数据存放入主内存的变量中



### 内存间交互操作的规则

\* 不允许read和load，store和write操作之一单独出现，以上两个操作必须按顺序执行，但不保证连续执行，也就是说，read和load之间，store与write之间是可插入其他指令的

\* 不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变之后必须把该变化同步回主内存

\* 不允许一个线程无原因的（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中

\* 一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化的变量，也就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作

\* 一个变量在同一时刻只允许一条线程对其执行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁

\* 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作化变量的值

\* 如果一个变量没有被lock操作锁定，则不允许对它执行unlock操作，也不能unlock一个被其他线程锁定的变量

\* 对一个变量执行unlock操作之前，必须先把此变量同步回主内存（执行store和write操作）

## 多线程中的可见性

\* 可见性：就是一个线程修改了变量，其他线程可以知道

\* 保证可见性的常见方法：volatile，synchronized,final(一旦初始化完成，其他线程就可见)

## volatile

\* volatile基本上是JVM提供的最轻量级的同步机制，用volatile修饰的变量，对所有线程可见，即对volatile变量所做的写操作能立即反映到其他线程中

\* 用volatile修饰的变量，在多线程环境下仍然是不安全的

\* 用volatile修饰的变量，是禁止指令重排优化的

\* 适合使用volatile的场景：

运算结果不依赖变量的当前值

能确保只有一个线程去修改变量的值

## 指令重排

\*指令重排：指的是JVM为了优化，在条件允许的情况下，对指令进行一定的重新排列，直接运行当前能够立即执行的后续指令，避开获取下一条指令所需数据造成的等待

\* 线程内串行语义，不考虑多线程间的语义

\* 不是所有的指令都能重排，比如：

1 写后读a=1;b=a;写一个变量之后，再读这个位置

2 写后写a =1;a=2;写一个变量之后，再写这个变量

3 读后写a=b;b=1;读一个变量之后，再写这个变量

\* 以上语句不可重排，但是a=1;b=2;是可以重排的

\*程序顺序原则：一个线程内保证语义的串行性

\*volatile规则：volatile变量的写，先发生与读

\*锁规则:解锁（unlock）必然发生在随后的加锁（lock）前

\*传递性:A先于B,B先于C 那么A必然先于C

\*线程的start方法先于它的每一个动作

\*线程的所有操作先于线程的终结（Thread.join()）

\*线程的终端（interrupt（））先于被中断线程的代码

\*对象的构造函数执行结束先于finalize（）方法

## 多线程中的有序性

\* 在本线程内，操作都是有序的

\* 在线程外观察，操作都是无序的，因为存在指令重排或主内存同步延时

## Java线程安全的处理方法

\*不可变是线程安全的

\*互斥同步（阻塞同步）:synchronized，java.util.concurrent.ReentrantLock.目前这两个方法性能已经差不多了，建议优先选用synchronized,ReentrantLock增加了如下特性：

1 等待可中断：当持有锁的线程长时间不释放锁，正在等待的线程可以选择放弃等待

2 公平锁：多个线程等待同一个锁时,须严格按照申请锁的时间顺序来获得锁

3 锁绑定多个条件：一个ReentrantLock对象可以绑定多个condition对象，而synchronized是针对一个条件的，如果要多个，就得有多个锁

\*非阻塞同步：是一种基于冲突检查的乐观锁定策略，通常是先操作，如果没有冲突，操作就成功了，有冲突再采取其他方式进行补偿处理

\*无同步方案：其实就是在多线程中，方法并不涉及共享数据，自然也就无需同步了

## 锁优化

### 锁优化之自旋锁与自适应自旋

\*自旋：如果线程可以很快获得锁，那么可以不再OS层挂起线程，而是让线程做几个忙循环，这就是自旋

\*自适应自旋：自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间和锁的拥有者状态来决定

\*如果锁被占用时间很短，自旋成功，那么能节省线程挂起，以及切换时间，从而提升系统性能

\*如果锁被占用时间很长，自旋失败，会白白耗费处理器资源，降低系统性能

### 锁优化之锁消除

\*在编译代码的时候，检测到根本不存在共享数据竞争，自然也就无需同步加锁了;通过-XX：+EliminateLocks来开启

\*同时要使用-XX:+DoEscapeAnalysis开启逃逸分析，所谓逃逸分析：

1. 如果一个方法中定义的一个对象，可能被外部方法引用，称为方法逃逸
2. 如果对象可能被其他外部线程访问，称为线程逃逸，比如赋值给类变量或者可以在其他线程中访问的实例变量

### 锁优化之锁粗化

\*通常我们都要求同步块要小，但一系列连续的操作导致对一个对象反复的加锁和解锁，这会导致不必要的性能损耗，这种情况建议把锁同步的范围加大到整个操作序列

### 锁优化之轻量级锁

\* 轻量级是相对于传统锁机制而言，本意是没有多线程竞争的情况下，减少传统锁机制使用OS实现互斥所产生的性能损耗

\* 其实现原理很简单，就是类似乐观锁的方式

\* 如果轻量级锁失败，表示存在竞争，升级为重量级锁，导致性能下降

### 锁优化之偏向锁

\* 偏向锁是在无竞争情况下，直接把整个同步消除了，连乐观锁都不用，从而提高性能；所谓的偏向，就是偏心，即锁会偏向于当前已经占有锁的线程

\* 只要没有竞争，获得偏向锁的线程，在将来进入同步块，也不需要做同步

\* 当有其他线程请求相同的锁时，偏向模式结束

\* 如果程序中大多数锁总是被多个线程访问的时候，也就是竞争比较激烈，偏向锁反而会降低性能

\* 使用-XX:-UseBiasedLocking来禁用偏向锁，默认开启

## JVM中获取锁的步骤

\* 会先尝试偏向锁；然后尝试轻量级锁

\* 再然后尝试自旋锁

\* 最后尝试普通锁，使用OS互斥量在操作系统层挂起

## 同步代码的基本规则

\* 尽量减少锁持有的时间

\* 尽量减小锁的粒度

# 性能监控与故障处理工具

\*命令行工具：jps,jinfo,jstack,jmap,jstat,jstatd,jcmd

\*图形化工具：jconsole,jmc,visualvm

\*两种连接方式：JMX，jstatd

## JVM监控工具的作用

\*对jvm运行期间的内部情况进行监控，比如：对jvm参数，CPU,内存，堆等信息的查看

\*辅助进行性能调优

\*辅助解决应用运行时的一些问题，比如：OutOfMemoryError,内存泄漏，线程死锁，锁争用，java进程消耗CPU过高等等

### jps

jps（JVM Process Status Tool）：主要用来输出JVM中运行的进程状态信息，语法格式如下：jps[options][hostid]

hostid字符串的语法与URI的语法基本一致：[protocol:][[//]hostname][:port][/servername]，如果不指定hostid，默认为当前主机或服务器