# Java学习笔记

## 线程与进程

### 线程和进程的对比：

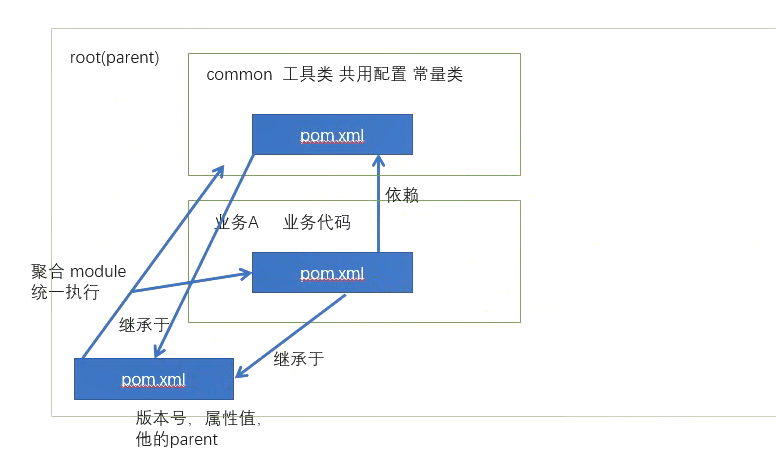
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 进程 | 资源分配的基本单位 | 每一个进程具有独立的代码和数据空间（内存） | 一个进程具有1~n个线程 | 进程之间的通信和数据交互非常耗时 |
| 线程 | CPU调度的最小单位 | 同一类线程共享代码和数据空间 | 线程，只能获取所属进程的内存等资源 | 线程之间的通信和数据交换都非常的快 |

举例子：

在笔记本上打开word就是一个进程，winWord.ext ，打开word写文档，会有一个容灾备份线程，还有语法检测线程，分别是backup Thread，checkStyle Thread线程。

## Maven的使用

继承，依赖，聚合三者之间的关系



对于写在<dependencyManagement>标签中的包并不会实际的下载，只是一个声明，并不会实际的从网上下载这些包，这就好像是对于一个类的声明，并没有产生实际的对象。

在<dependencyManagement> 标签下有一个版本子标签，凡是继承了该pom文件对应的项目的项目的pom文件，只要前面的groupId和artifact一致，它会读取下面的这个版本标签，那么就意味着在子工程中不用在定义版本号了。

<groupId>org.apache.spark</groupId>  
<artifactId>spark-core\_2.11</artifactId>

<version>${spark.version}</version>

Version标签的值，还可以提取出来，然后统一读取值。

## Java的面向对象

Java是一门面向对象的语言，但是Java中的静态方法不是面对对象的，Java中的静态方法是通过类来调用的。

## 日志级别，（偏）

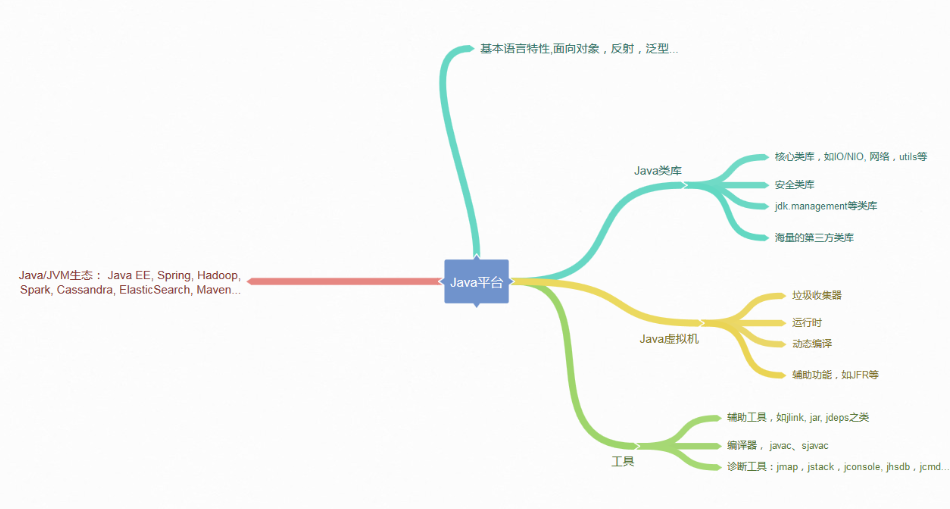
越是靠后，输出的信息越少

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |
| Trace | **Debug** | **Info** | **Warn** | *Error* | Fatal |

## 谈谈你的Java平台的理解：

1. 书写一次，到处运行（这里的编译时java源代码编译成为字节码，而不是机器码）
2. 垃圾回收机制
3. 还可以谈谈Java的语言特点，泛型，Lambda表达式的特性，
4. 还可以谈谈集合，IO / NIO，网络，并发，安全基础类库
5. 还可以谈谈JVM的基础概念和一些机制，比如Java的类加载机制，类加载的过程（加载，验证，链接，初始化）
6. 还可以谈谈GC的种类，各自使用于什么样的工作环境。

回答的思路如下：



JRE，包含了JVM，和Java类库，JDK包含了JRE 和一些工具（如编译工具，各种诊断工具）

### Java是解释执行的么？

我们将java程序分为编译期，和运行期，编译期，是将java源代码编译成字节码而不是像C或者是C++ 那样将源代码编译成为机器码。

在运行的时候，JVM会通过类加载器加载字节码，**解释或者是编译执行**，比如JDK1.8实际上是属于混合编译的一种类型的。

绝大多数我们使用的JVM都提供了JIT（Just in Time） 编译器，也即动态编译器，JIT能够在运行时将热点代码编译成机器码，这个时候部分热点代码就属于编译执行，而不是解释执行。

## Exception和Error有什么区别：

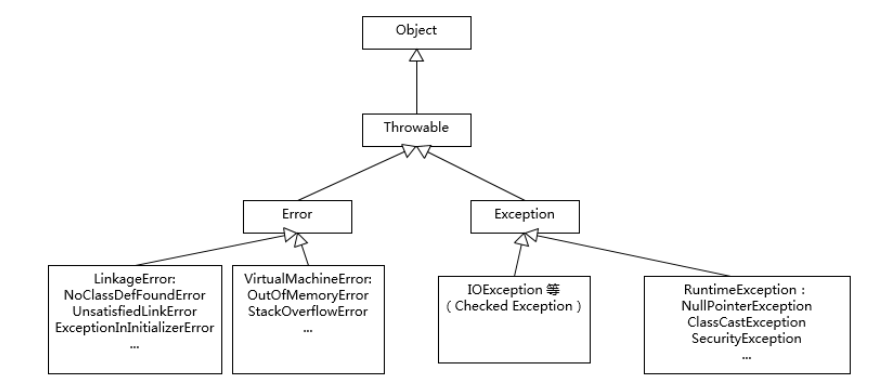
相同点：都继承了Throwable类，

### 不同点：

Error是指在正常的情况下不太可能出现的情况，一旦发生，程序就处于非正常的，不可恢复的状态。

Exception 分为可检查异常，和不可检查异常（也即运行时异常），可检查异常在代码中必须进行显示的捕获和声明。而不可检查异常，并不需要。

回答思路：



#### NoClassDefFoundError 和 ClassNotFoundException的区别

类的装载分为显示和隐式两种方式，

**显示装载类**：当调用下面的方法：

1.类 Class 中的 forName() 方法。

2.类 ClassLoader 中的 findSystemClass() 方法。

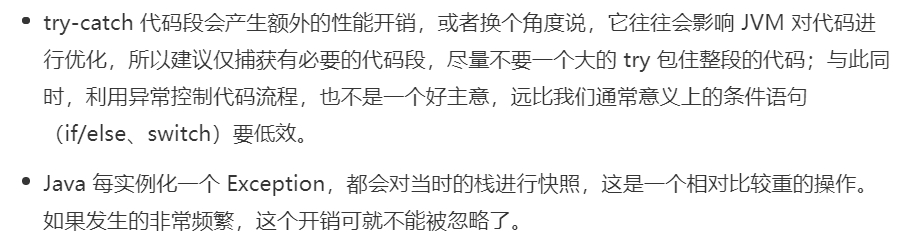
3.类 ClassLoader 中的 loadClass() 方法。

指定类（以类名作为参数）有类装载器装入的时候，就是显示的装载类，如果类应装入，那么会返回一个引用，如果没有，则装载器会通过委托模式来装载类。

以上如果没有找到这个类，就会发生ClassNotFoundException；

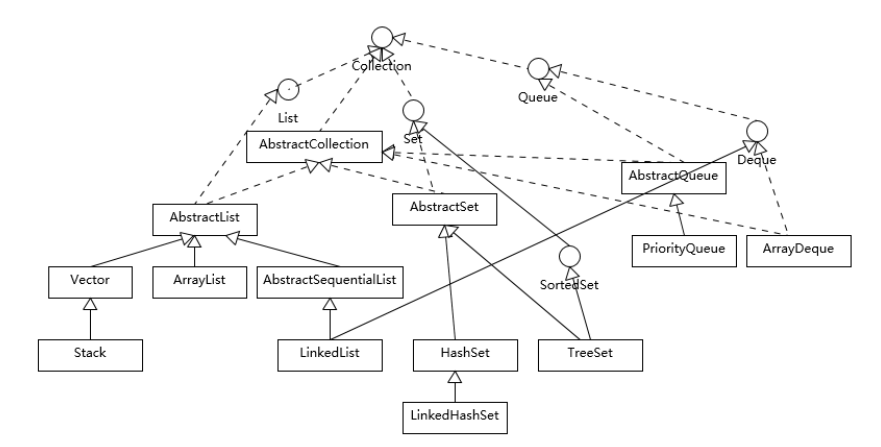
**隐式装载类：**发生在由于引用，实例化或继承导致装载类的时候，在这样的情况下，装入都是在幕后进行，JVM会解析必要的引用并装入类。如果引用的类在类路径中没有找到，就会发生NoClassDefFoundError。

Java的异常处理机制，会消耗一定的性能：



## 集合

下图没有将java.util.concurrent包下的线程安全容器添加到进去，也么有列出Map容器，但是Map也是作为集合框架的一部分，再但是本身Map并不是真正的Collection。



实线：继承关系

虚线：实现关系

|  |  |
| --- | --- |
| List | 有序集合，可以重复 |
| Set | 不允许重复元素 |
| Queue/Deque | Java提供的标准队列的实现，还支持FIFO， |

|  |  |
| --- | --- |
| TreeSet | 支持自然顺序访问 |
| HashSet | 没有顺序 |
| LinkedHahSet | 保持插入的顺序 |

如何实现线程不安全的集合支持并发编程的场景：

**List list = Collections.synchronizedList(new ArrayList());** 该方法的实现，就是在get，set，add等基本方法前面添加了**synchronizd**关键字。

Java底层默认的排序：对于基本数据类型，（见数据结构和算法之美）

\*\*Set的底层实现是\*\*Map，具体的value的值是虚拟对象Dummy对象，PRESENT。

在Java9 中，Java标准库提供了一系列的静态工厂方法，比如List.of() ， Set.of()，创建不可变的集合。

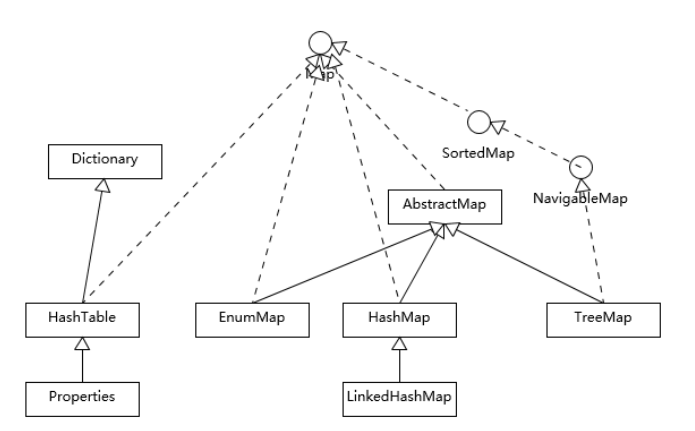
### Vector，ArrayList，LinkedList的区别

**三者都是集合框架中的List，也就是所谓的有序集合**

|  |  |
| --- | --- |
| Vector | Java早起提供的**线程安全的动态数组**，Vector内部使用动态数组来保存数据；扩容时变为原来的2倍 |
| ArrayList | 应用更加广泛的**动态数组**，线程不安全，扩容时为原来的1.5倍 |
| LinkedList | Java提供的**双向链表**，线程不安全 |

### Hashtable，HashMap，TreeMap有什么不同

|  |  |
| --- | --- |
| Hashtable | 早期提供的哈希表的实现，本身是同步的，不支持null键和值，不推荐用 |
| HashMap | 不是同步的，支持null键和值，  （并发编程中产生，无限占用CPUsize不准等异常） |
| TreeMap | 基于红黑树 的一种提供**顺序访问的Map**，它的get，put，remove等方法都是log（N）的时间复杂度，顺序由比较器（comparator）或自然顺序（comparable）来决定。 |
| LinkedHashMap | 遍历顺序符合插入顺序（也维护了双向链表）场景：比如释放空间占用敏感的资源池中，将最不长访问的对象释放掉。如LRU。 |



HashMap的是依赖于hashCode和equal方法的，所以：

1. equals相等，hashCode一定相等
2. 重写了hashCode就要重写equals

#### HashMap的源码：

putVal 方法本身逻辑非常集中，从初始化、扩容到树化，全都和该方法有关

// HashMap默认的初始大小是16  
static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; // aka 16  
  
// 默认装载因子，当HashMap中的元素超过了0.75 \* capacity的时候，扩容为原来的两倍  
static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;  
  
// 当链表的长度超过了8的时候，就需要treeify  
static final int *TREEIFY\_THRESHOLD* = 8

HashMap的也是Lazy-load的模式，在首次使用的时候，才别初始化，new的时候只是被声明（特殊的构造除外）

扩容的门限值：装载因子 \* 容量 ，当达到了门限值的时候，就需要给Map扩容。

扩容的时候，由于Map是成倍扩容的，门限值也成倍扩容的。

## 集合的线程安全

并发容器，**ConcurrentHashMap**，**CopyOnWriteArrayList**，**ArrayBlockingQueue**。

### ConcurrentHashMap

##### 为什么需要ConcurrentHashMap？

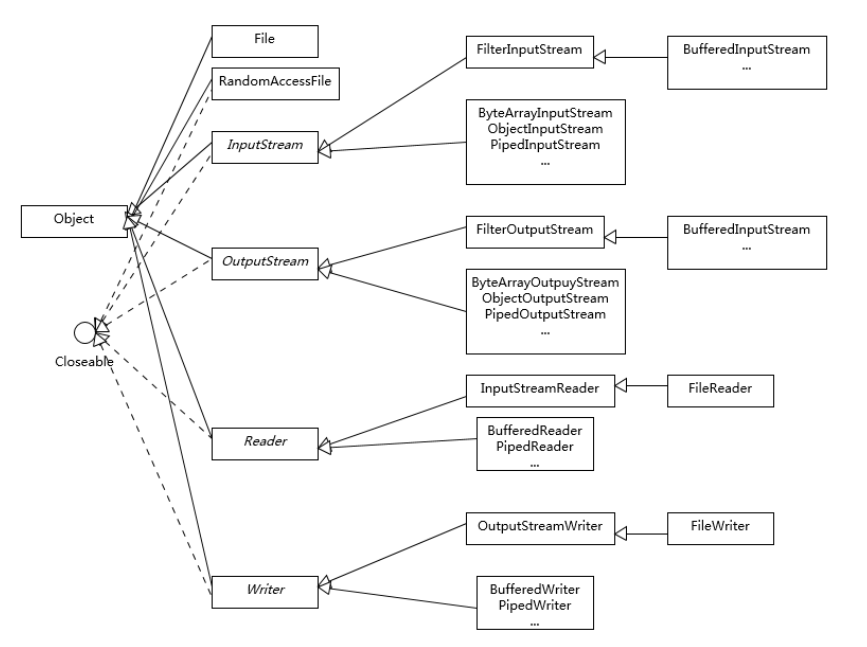
Hashtable本身比较低效，因为它的基本实现就是将put，get，size添加上synchronized，这就导致了**所有并发的操作都要竞争同一把锁，一个线程在进行同步操作的时候，其他的线程只能等待，大大降低了并发操作的效率**。即便使用了同步包装版本，都只是适合非高度并发的场景下。

悲观锁认为，并发编程中，每次访问共享的资源都会发现冲突，所以要加锁；乐观锁认为并发编程中，每次访问共享的资源都不会发生冲突，线程可以不停的执行，无需加锁，一旦发现冲突，无锁策略就会采用一种称为CAS的技术来保证线程执行的安全性。

## IO

IO一是软件开发的核心部分，随着海量数据的增长和分布式系统的发展，IO扩展能力变得相对重要。在java中，最早出现的是java.io包，该包中对应着Java的IO操作。

BufferOutputStream等带有缓冲区的实现，利用了缓冲区，实现对数据进行批量操作，可以避免频繁的磁盘读写，进而提高IO处理的效率，但在使用中，别忘了flush。



### BIO

**阻塞式IO**，在读取输入流或者是写入输出流时，读写动作完成之前，线程会一直阻塞在哪里，他们之间的调用是可靠的线性顺序。

比如Tomcat就是一个阻塞式IO，当一个请求达到Tomcat的时候，Tomcat会生成一个线程，来处理这个请求，这个线程只能用来处理这个请求，不能干别的事情，也就是说该线程和请求绑定在了一起。这样一来效率是比较低的，所以一般来说，会使用一个线程池来提高效率。

网络通信也是IO行为，所以，人们把Java.net下面提供的部分网络API，比如Socket，ServerSocket，HTTPURLConnection也归类到同步阻塞IO类库。

(select \*, rank() over(partition by area order by clickcount desc) rk from result2) tmp

Select area,click\_product\_id, clickcount from (select \*, rank() over(partition by area order by clickcount desc) rk from result2) tmp

where rk <= 3

### NIO

非阻塞式IO，Java1.4中引入了NIO框架（Java.nio包）提供了**Channel**，**Selector**，**Buffer**等新的抽象，**可以构建多路复用**，同步非阻塞IO，提供了更接近操作系统底层的高性能数据操作方式

比如Redis中的多路IO复用（底层使用的就是**Linux中的EPOl l方法**）

目前Linux只能使用NIO，无法使用AIO，Windows可以做AIO

##### NIO的组成部分：

Buffer，高效的数据容器，所有的基本数据类型都有对应的Buffer实现

Channel，是NIO中被用来支持批量式IO操作的一种抽象,File或者是Socket通常被认为是较高层次的抽象，而channel则是操作系统底层的一种抽象。

Selector，**是NIO实现多路复用的基础**，可以检测到注册在selector上的多个channel中，是否有channel处于就绪状态，进而实现了单线程对于多channel的高效管理。

### AIO

异步非阻塞式IO，Java7中，NIO有了进一步的改进，即NIO2，**异步非阻塞方式**。异步IO操作基于时间和回调机制，应用操作之后，不会阻塞，当后台处理完成之后，操作系统会通知相应的线程进行后续的工作。

比如吃饭的时候，只能在饭店等着，在饭店吃饭的时候，先去买点东西（但是需要不断的轮询，没有耦合性），订餐（存在着耦合性，获知了地址）。

**Netty**是一个IO框架，刚开始的时候使用的是AIO，后来使用的是NIO。

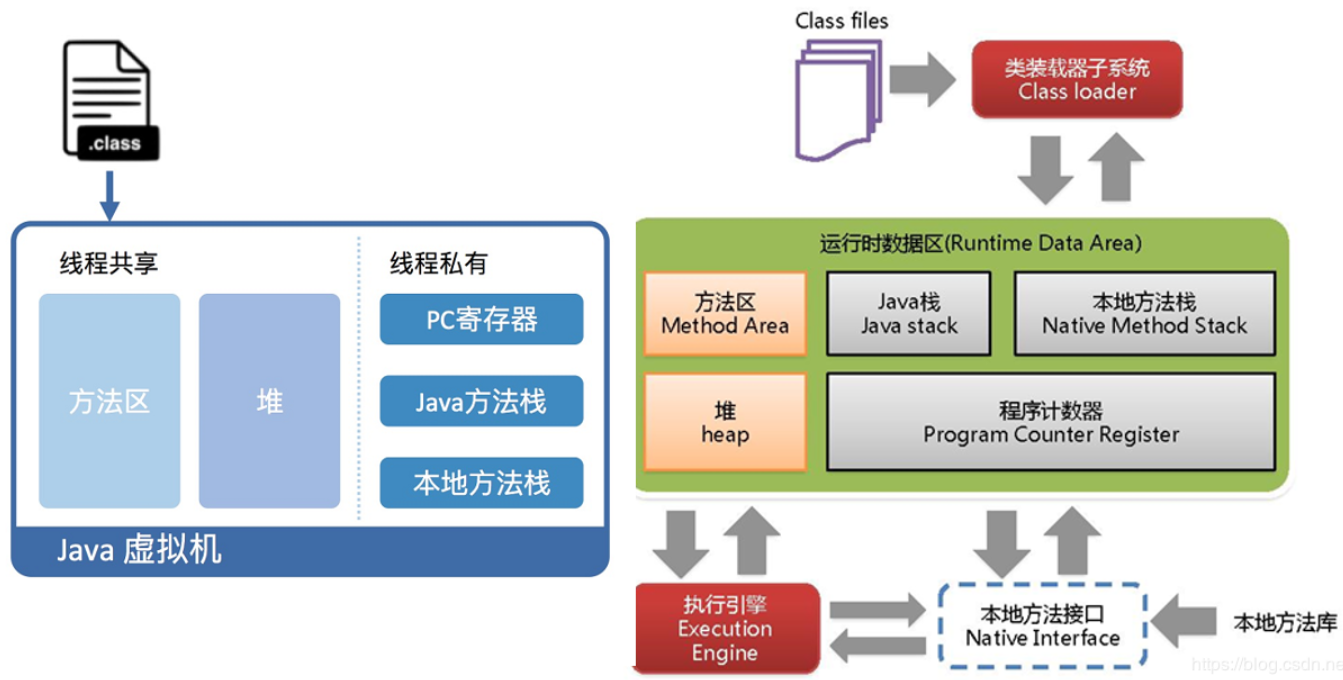
## Final，finally，finalize的区别

|  |  |
| --- | --- |
| Final | 修饰类，不可继承；  修饰方法，不可重写；  修饰变量，不可修改（在并发编程中，保护只读数据） |
| finally | 保证重点代码一定会被执行的机制，比如保证JDBC的关闭链接  更推荐try-with-resources（Java7） |
| finalize | Object的一个方法，保证对象在垃圾回收是完成特定资源的回收  现在已经不再推荐使用，Java9中已经deprecated  推荐使用的是：java.lang.ref.Cleaner，Cleaner 的实现利用了幻象引用（PhantomRefence.） |

|  |
| --- |
| try {  // do something  System.exit(1);  } finally{  System.out.println(“Print from finally”);  }  **Finally中的代码不会被执行。**  Java9中，of方法，创建出来的是immutable的List  final List<String> strList = new ArrayList<>();  strList.add("Hello");  strList.add("world");  List<String> unmodifiableStrList = List.of("hello", "world");  unmodifiableStrList.add("again"); |

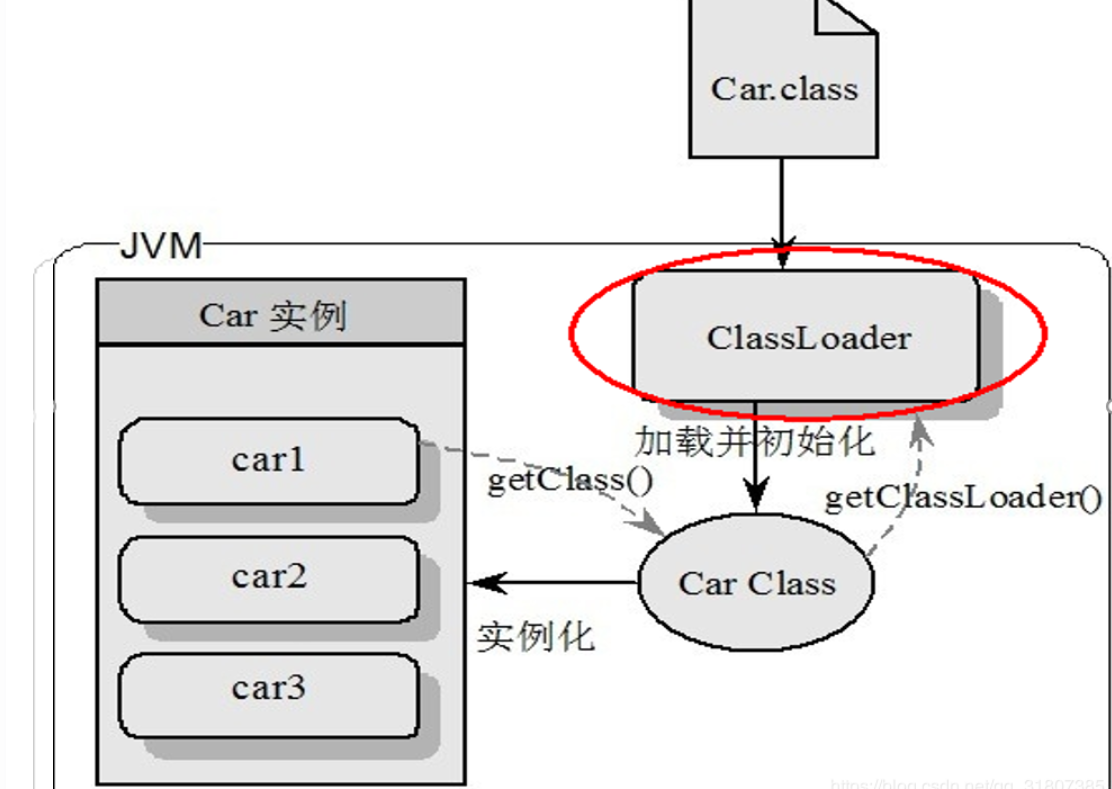
## JVM

### JVM的内存结构图：



### 类加载器

#### 结构图



#### 双亲委托模式

|  |
| --- |
| Java允许创建和JDK自带类库相同名称类  但是JVM不会加载我们自己定义的类，而是加载JDK提供的类  JVM的实现方式为：双亲委派机制  1） 如果应用类加载器加载一个类，不是马上加载  2）委托父加载器（扩展类加载器）加载  3）扩展类加载器加载类的时候，也不是马上加载  4） 委托父加载器（启动类加载器）加载  5）启动类加载器开始加载这个类，如果加载成功，那么直接使用。  6）如果加载失败，那么会返回到扩展类加载器，由扩展类加载器加载这个类  7) 扩展类加载器开始加载这个类，如果加载成功，那么直接使用。  8）如果加载失败，**会抛出异常给**子加载器**（应用类加载器）**  9）应用类加载器捕捉异常后开始加载这个类，如果加载成功，那么直接使用。  10）如果加载失败，会将异常抛出给JVM    JVM自身提供了3个类加载器，每一个类加载器会加载不同位置的类  1）启动类加载器：加载JDK核心类库，由C++语言实现,  加载位置： $JRE\_HOME/**lib**  2）扩展类加载器：是java类，可以加载JDK扩展类库  加载位置： $JRE\_HOME**/lib/ext**  3）应用类加载器：是java类，可以加载环境变量classpath中的类  加载位置： **$classpath**  JAVA\_HOME, PATH, CLASSPATH |

### 方法区

多线程中，**每一个线程都会产生一个Java栈**，**线程独享的内存是栈区，线程共享的内存是方法区**。

方法区中，有**静态变量+常量+类信息（**构造方法、接口的定义）+运行时的**常量池放置在方法区**中。但是实例变量不在方法区中。

### Stack栈

**栈内存管理Java程序的运行， 伴随着线程的整个生命周期，所以栈不存在GC。栈中有：8种基本类型的变量+对象的引用变量+实例方法**

#### 栈溢出和栈内存溢出的区别

|  |  |  |
| --- | --- | --- |
| 栈溢出 | StackOverflowError | 栈帧太多，导致的原因是栈太深 |
| 栈内存溢出 | OutOfMemoryError  OOM | 来一个线程，创建一个栈，最后导致内存不够 |

出现OOM Java Heap space的具体原因：

1. Java虚拟机的堆内存设置不够，可以通过调节参数-Xms，-Xmx
2. 代码中创建大量的大对象，并且长时间不能被垃圾回收器回收

### 堆

一个JVM中只存在一个堆内存，堆内存的大小是可以调节的，类加载器读取了文件后，需要把类，方法，常变量都放到堆内存中，保存所有引用类型的真实信息，**这里的堆内存在逻辑上是包含方法区的**。

|  |
| --- |
| Young Generation Space 新生区 Young/ New |
| Tenure Generation Space 养老区 Old/ Tenure 单词Tenure是任期的意思 |
| Permanent Space 永久区 Perm 这里的永久区其实就是方法区 |

#### 堆内存的逻辑结构



#### 新生区：

新生区是对象，诞生，成长，消亡的地方，一个类在这里产生，应用，最后被GC回收，新生区分为两个部分，伊甸园区（Eden space）和幸存者区（Survivor space），所有的类都是在伊甸园区被new出来的。幸存者有两个区，幸存者0区（From区）和幸存者1区（To区），当伊甸园区的空间快要用完的时候，程序还要创建新的对象，JVM的垃圾回收器将会对伊甸园区进行垃圾回收（**Minor** GC最小化的GC），将伊甸园区的其他对象进行销毁，然后将伊甸园区幸存的对象放置到幸存者0区，0区满了，再次GC，移动到1区，1区满了，再次GC，移动到养老区，若养老区也满了，这个时候将会产生**Major** GC，（FullGC），进行养老区的内存清理，若养老区FullGC之后，依然发现无法保存对象，就会产生OOM

**OOM只有在Full GC之后才会出现。**

GC中的几个算法：

如果我们创建对象，但是发现在伊甸园区中再也容纳不下的时候，将会启动Minor GC 对伊甸园区中的对象进行回收，那么将对哪些对象进行回收呢？

此时将会使用

1. 可达性算法，Minor GC会从ROOT对象进行查找，找到能关联到的对象，如果找不到关联，那么这个对象就可以被标记，**这里的ROOT对象指的是栈中的引用变量，方法区存储的静态变量**。也即如果我们发现，伊甸园中的某个对象没有被上述的变量引用，就**可能**会被Minor GC回收掉（还需要进一步标记）
2. 标记算法，对于没有进行关联的对象，JVM会对标记的对象调用finalized 方法，判断是否真的要被清除
3. 清除：对于确定要删除的对象，Minnor GC将会把他清除掉，这时候会产生内存碎片。
4. 复制算法：为了解决伊甸园区内存碎片的问题，会将幸存的对象复制一份到S1区，然后对伊甸园区的数据进行清除。同时复制到S1区的对象，标记上年龄的信息，表示经历过了多少次的垃圾回收

S1总为空

如果伊甸园区的又要满了，又要触发垃圾回收，此时，Minor GC将会对伊甸园区和S0区的对象进行回收，伊甸园区和S0区幸存的对象，将会被复制一份到S1区，然后清除掉伊甸园区和S0区的对象，幸存下来的对象的年龄也要+1，此时S0 和 S1交换编号。

当伊甸园区和S0区中幸存的对象太多，S1放不下的时候，会将对象放置到养老区，当age达到一定的程度的时候，也会被放置到老年代（默认是15）

如果JVM发现，老年代也放置不下对象了，将会启动Full GC，对全部的内存做垃圾回收，这个是时候效率是比较低的。

**做GC的时候，是阻塞写操作的**。

#### 永久区

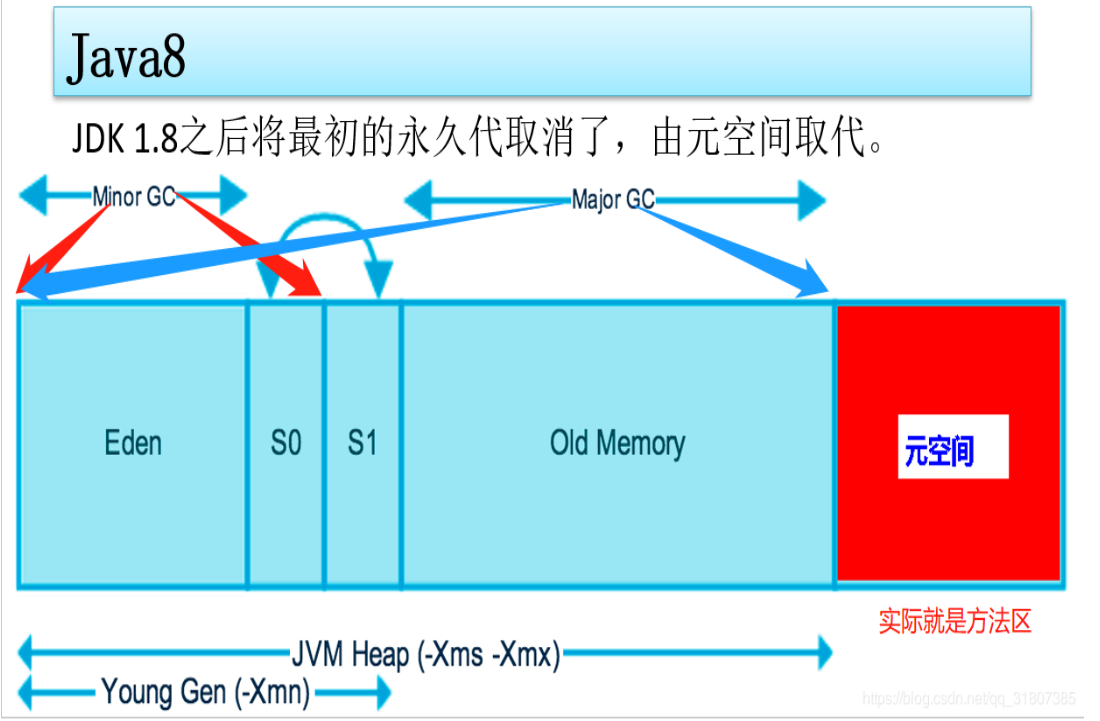
也叫非堆内存，因为永久区也叫方法区，是堆的逻辑部分，而且GC不对其进行回收（低概率），因此就称之为非堆内存。

Jdk1.6及之前： 有永久代, 常量池1.6在方法区

Jdk1.7： 有永久代，但已经逐步“去永久代”，常量池1.7在堆

**Jdk1.8及之后： 无永久代，常量池1.8在元空间**

#### 堆参数调优



-Xms 默认为内存的1 / 64 Xmx默认为内存的 1 / 4

垃圾回收器：CMS和G1

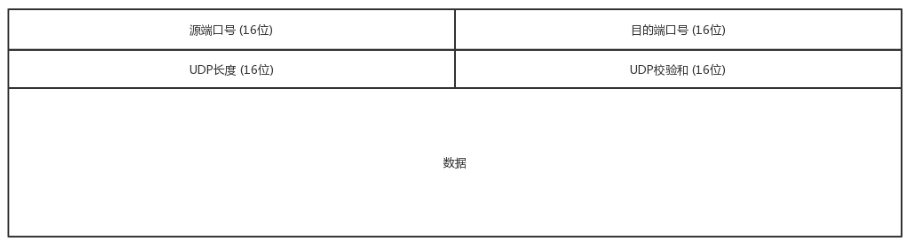
以hotspot中的CMS回收器为例，CMS是基于Mark-Sweep实现的，对于对像的回收效率很高，而对于碎片问题，CMS采用基于Mark-Compact算法的Serial Old回收器做为补偿措施：当内存回收不佳（碎片导致的Concurrent Mode Failure时），将采用Serial Old执行Full GC以达到对老年代内存的整理。 从JDK1.9后不再推荐使用CMS垃圾回收器， 推荐采用 Garbage-First（G1）垃圾回收器， Garbage-First（G1）是专门针对多核处理器的，虽然是整理，但是非常是多核完成的，因此较快。

# 网络

## UDP

UDP是面向无连接的，不保证不丢失，不保证数据按照顺序到达。UDP是无状态服务，相对于TCP没有拥塞控制。

UDP的包头格式



## TCP

TCP是面向连接的，TCP会三次握手，建立可靠的连接，从而保证可靠的数据交付，做到数据的无差错，不丢失，不重复，而且按序到达；TCP有拥塞控制，TCP面向字节流。

