目录

[Java基本知识点](#_Toc1485102369_WPSOffice_Level1) [2](#_Toc1485102369_WPSOffice_Level1)

[1 Java的类加载过程](#_Toc1637959682_WPSOffice_Level1) [2](#_Toc1637959682_WPSOffice_Level1)

[2 基础知识点](#_Toc595504481_WPSOffice_Level1) [2](#_Toc595504481_WPSOffice_Level1)

[3 JVM内存结构](#_Toc1370017147_WPSOffice_Level1) [4](#_Toc1370017147_WPSOffice_Level1)

[4 Java对象](#_Toc558526495_WPSOffice_Level1) [8](#_Toc558526495_WPSOffice_Level1)

[5 类加载器](#_Toc503780428_WPSOffice_Level1) [12](#_Toc503780428_WPSOffice_Level1)

[6、集合](#_Toc1657116922_WPSOffice_Level1) [15](#_Toc1657116922_WPSOffice_Level1)

[7 常量池](#_Toc448690111_WPSOffice_Level1) [18](#_Toc448690111_WPSOffice_Level1)

[8、泛型](#_Toc1319610960_WPSOffice_Level1) [19](#_Toc1319610960_WPSOffice_Level1)

[9、反射](#_Toc1637782151_WPSOffice_Level1) [20](#_Toc1637782151_WPSOffice_Level1)

[10、代理](#_Toc1906708258_WPSOffice_Level1) [21](#_Toc1906708258_WPSOffice_Level1)

[11 注解](#_Toc1294711672_WPSOffice_Level1) [22](#_Toc1294711672_WPSOffice_Level1)

[12 锁](#_Toc1914759900_WPSOffice_Level1) [22](#_Toc1914759900_WPSOffice_Level1)

[13 线程](#_Toc1327189005_WPSOffice_Level1) [26](#_Toc1327189005_WPSOffice_Level1)

# Java基本知识点

## 1 Java的类加载过程

jvm将.class类文件信息加载到内存并解析成对应的class对象的过程，**注意：**jvm并不是一开始就把所有的类加载进内存中，只是在第一次遇到某个需要运行的类才会加载，并且只加载一次

主要分为三部分：1、加载，2、链接（1.验证，2.准备，3.解析），3、初始化

1：加载

类加载器包括 BootClassLoader、ExtClassLoader、APPClassLoader

2：链接

验证：（验证class文件的字节流是否符合jvm规范）

准备：为类变量分配内存，并且进行赋初值

解析：将常量池里面的符号引用（变量名）替换成直接引用（内存地址）过程，**在解析阶段，jvm会把所有的类名、方法名、字段名、这些符号引用替换成具体的内存地址或者偏移量。**

3：初始化

主要对类变量进行初始化，执行类构造器的过程，换句话说，只对static修饰的变量或者语句进行初始化。

**范例：Person person = new Person();为例进行说明。**

**注意：Java编程思想中的类的初始化过程**

1. 找到People.class文件，将它加载到内存
2. 在堆内存中分配内存地址
3. 初始化
4. 将堆内存地址指给栈内存中的p变量

## 2 基础知识点

### 2.1 String、StringBuilder、StringBuffer

StringBuffer里面的很多方法添加了synchronized关键字，是可以表征线程安全的，所以多线程情况下使用它。

**执行速度：**

StringBuilder > StringBuffer > String

StringBuilder牺牲了性能来换取速度的，**这两个是可以直接在原对象上面进行修改，省去了创建新对象和回收老对象的过程**，而String是字符串常量（final）修试，另外两个是字符串变量，**常量对象一旦创建就不可以修改，变量是可以进行修改的，所以对于String字符串的操作包含下面三个步骤：**

1. 创建一个新对象，名字和原来的一样
2. 在新对象上面进行修改
3. 原对象被垃圾回收掉

### 2.2 抽象类和接口

**抽象类：**

1. 抽象类使用abstract修饰
2. 抽象类不能实例化即不可以通过new来实例化对象
3. 含有抽象方法的类是抽象类，必须用abstract来修饰
4. 抽象类可以有抽象方法，也可以不包含抽象方法，可以有具体实现的方法
5. 抽象类中的抽象方法只有方法体没有具体实现

**接口：**

1. 接口使用interface修饰
2. 接口也不能被实例化
3. 接口中的方法都是抽象方法
4. 接口中不能包含实例域和静态方法（静态方法必须实现，接口中方法是抽象不能实现）

**区别：**

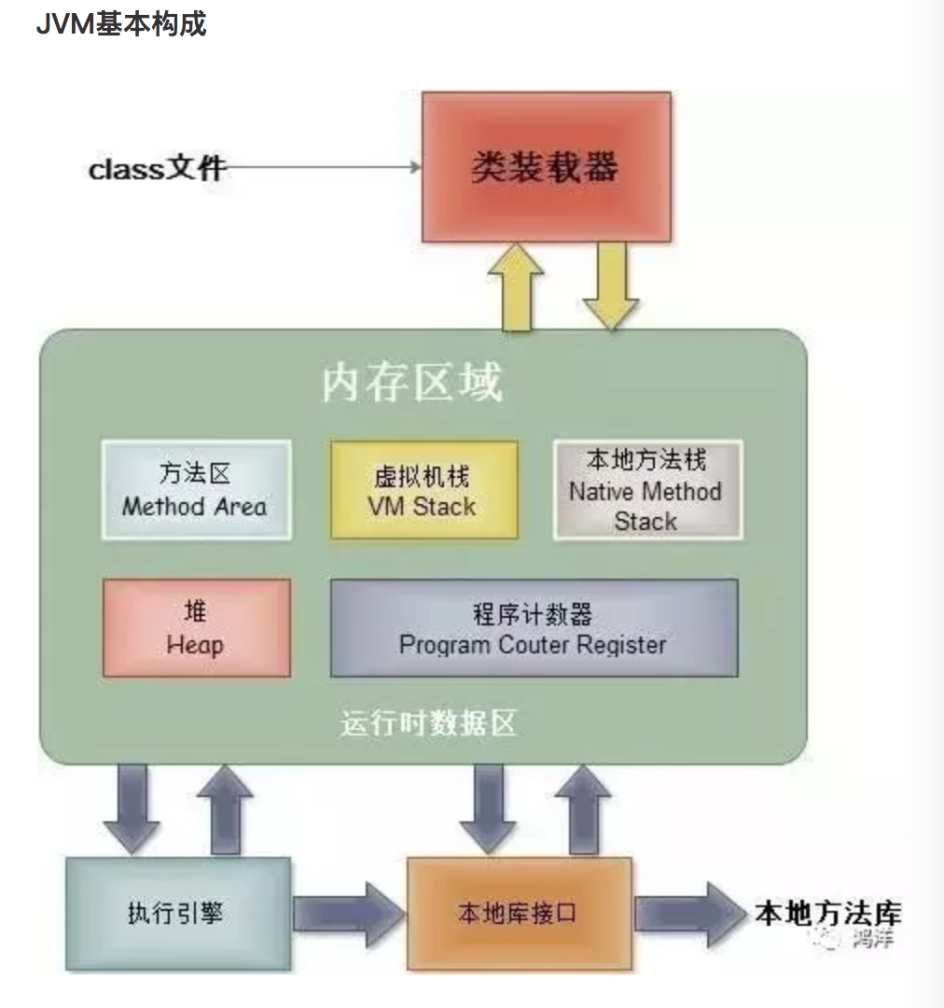
1. 抽象类只能继承一次，但是可以实现多个接口
2. 抽象类中的成员变量可以是任何类型修饰符，而接口中的成员变量只能是public static final 类型。
3. 接口中不能含有静态代码块和静态方法，抽象类中可以包含。

### **2.3 equal和==区别**

## 3 JVM内存结构

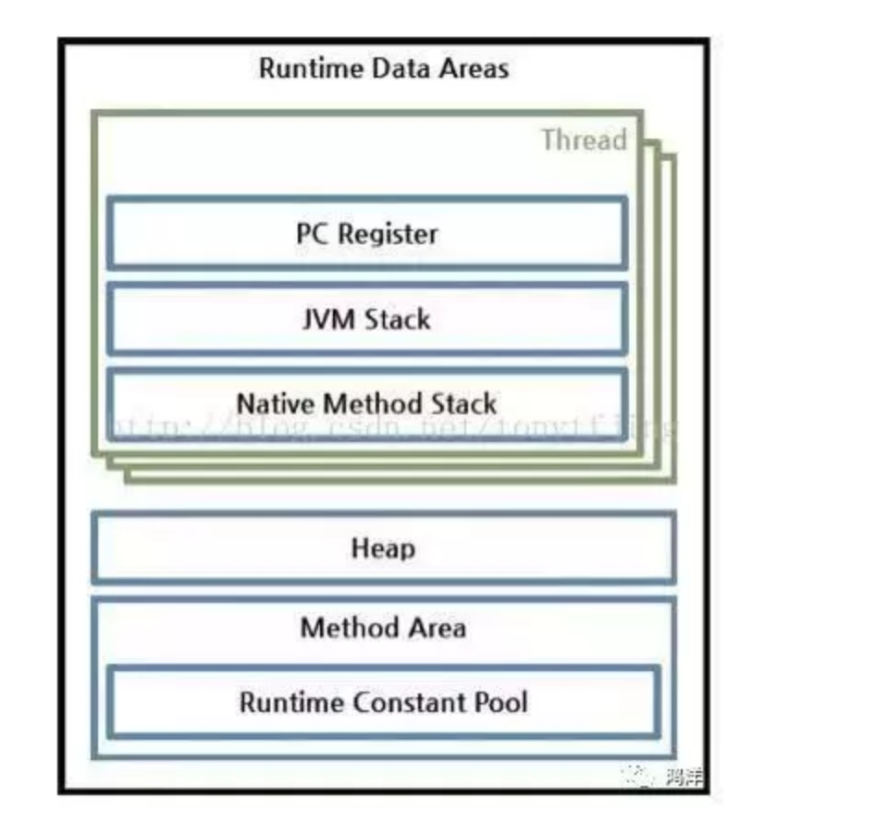
Java对象实例化过程中，主要使用到虚拟机栈、Java堆和方法区。**Java文件经过编译之后首先会被加载到jvm方法区中，jvm方法区中很重的一个部分是运行时常量池，用以存储class文件类的版本、字段、方法、接口等描述信息和编译期间的常量和静态常量。**

### 3.1 JVM基本结构



1. **类加载器classLoader，**在JVM启动时或者类运行时将需要的.class文件加载到内存中。
2. **执行引擎，**负责执行class文件中包含的字节码指令。
3. **本地方法接口，**主要是调用C/C++实现的本地方法及返回结果。
4. **内存区域（运行时数据区），**是在JVM运行的时候操作所分配的内存区，

主要分为以下五个部分：

****

1. **方法区：**用于存储类结构信息的地方，包括常量池、静态变量、构造函数等。
2. **Java堆（heap）：**存储Java实例或者对象的地方。这块是gc的主要区域。
3. **Java栈（stack）：**Java栈总是和线程关联的，每当创建一个线程时，JVM就会为这个线程创建一个对应的Java栈。在这个java栈中又会包含多个栈帧，每运行一个方法就创建一个栈帧，用于存储局部变量表、操作栈、方法返回值等。每一个方法从调用直至执行完成的过程，就对应一个栈帧在java栈中入栈到出栈的过程。所以java栈是线程私有的。
4. **程序计数器：**用于保存当前线程执行的内存地址，由于JVM是多线程执行的，所以为了保证线程切换回来后还能恢复到原先状态，就需要一个独立的计数器，记录之前中断的地方，可见程序计数器也是线程私有的。
5. **本地方法栈：**和Java栈的作用差不多，只不过是为JVM使用到的native方法服务的。

### 3.2 JVM源码分析

<https://www.jianshu.com/nb/12554212>

### 3.3 对象存活计算算法

1. **引用计数法**

给对象添加一个引用计数器，当有一个地方引用它时，计数器值就加1；当引用失效时，计数器就减1；任何时候计数器都为0的对象就是不可能再被使用的。引用计数器算法（Reference Counting）实现简单，判定效率也很高，在大部分情况下他都是一个不错的算法。但是，Java语言中没有选用引用计数算法来管理内存，其中最主要的原因是他很难解决对象之间的互相循环引用的问题。

1. **可达性分析算法：**

根搜索算法（GC Roots Tracing）的基本思路是通过一系列名为“GC Roots”的对象作为起始点，从这个节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain）,当一个对象到GC Roots没有任何引用链相连（用图论术语描述就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。在主流的商用程序语言中（Java、C#）,都是使用根搜索算法判定对象是否存活的。

  在Java语言里，可作为GC Roots的对象包括下面几种：

**（1）虚拟机栈（栈帧中的本地变量表）中的引用的对象。**

**（2）方法区中的类静态属性引用的对象。**

**（3）方法区中的常量引用的对象。**

**（4）本地方法栈中JNI（即一般说的Native方法）的引用的对象。**

### 3.4 四种引用

在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）,这四种引用强度依次逐渐减弱。

1. **强引用**

就是指在程序代码之中普遍存在，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

1. **软引用**

用来描述一些还有用，但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中并进行第二次回收。如果这次回收还是没有足够的内存，才会抛出内存溢出异常。在JDK1.2之后提供了SoftReference类来实现软引用。

1. **弱引用**

也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的的对象。在JDK1.2之后提供了WeakReference类来实现弱引用。

1. **虚引用（幽灵引用、幻影引用），**

他是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。在JDK1.2之后，提供了PhantomReference类来实现虚引用。

### 3.5 四种GC机制

垃圾收集器一般完成两件事

1. 检测出垃圾
2. 回收垃圾

Link:<https://blog.csdn.net/clover_lily/article/details/80160726>

#### 3.5.0 新生代老年代概念

Java 中的堆是 JVM 所管理的最大的一块内存空间，主要用于存放各种类的实例对象。

在 Java 中，堆被划分成两个不同的区域：新生代 ( Young )、老年代 ( Old )。

新生代 ( Young ) 又被划分为三个区域：Eden、From Survivor、To Survivor。

这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。

**堆的内存模型：**



默认的，新生代 ( Young ) 与老年代 ( Old ) 的比例的值为**1:2**( 该值可以通过参数 –XX:NewRatio 来指定 )，

即：**新生代 ( Young ) = 1/3 的堆空间大小。老年代 ( Old ) = 2/3 的堆空间大小**。

其中，新生代 ( Young ) 被细分为 Eden 和 两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分

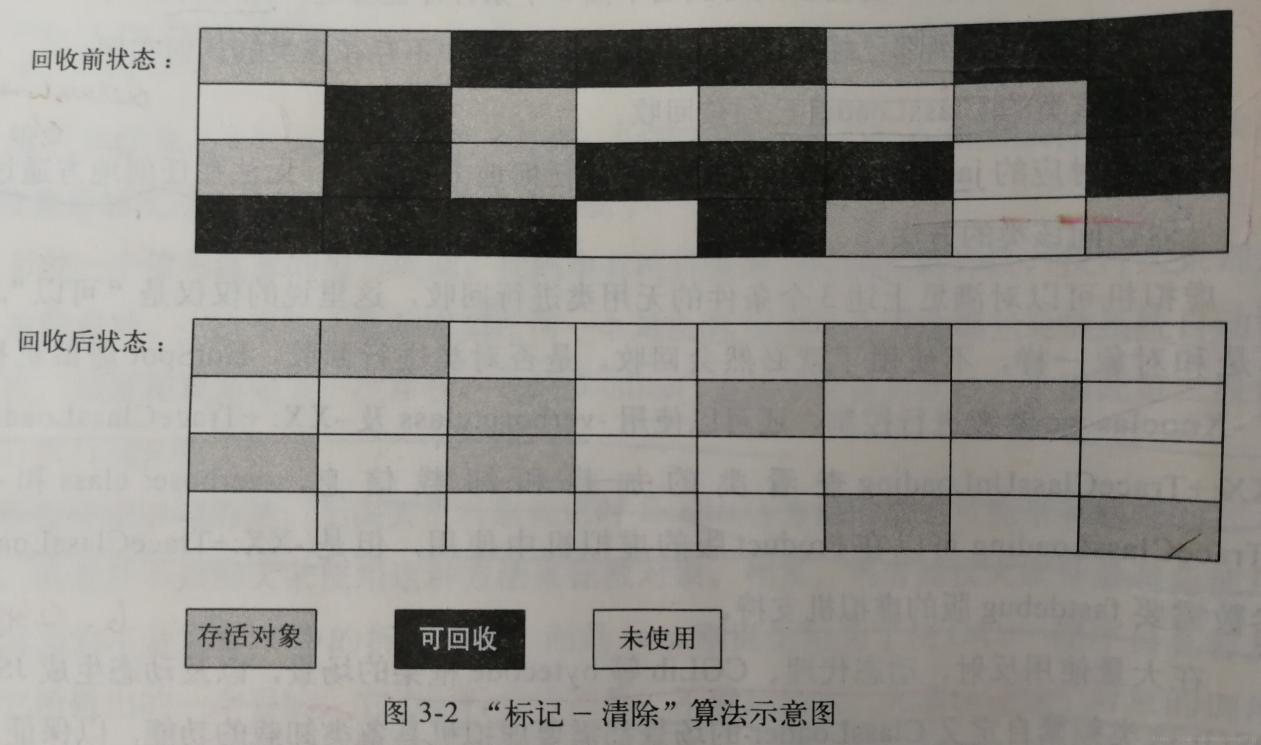
默认的，**Edem : from : to = 8 :1 : 1**( 可以通过参数**–XX:SurvivorRatio** 来设定 )，即：**Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小**。

JVM **每次只会使用 Eden 和其中的一块 Survivor 区域**来为对象服务，所以**无论什么时候，总是有一块Survivor区域是空闲着的**。

因此，新生代实际可用的内存空间为 9/10 ( 即90% )的新生代空间。

#### 3.5.1 标记-清除算法（Mark-Sweep）

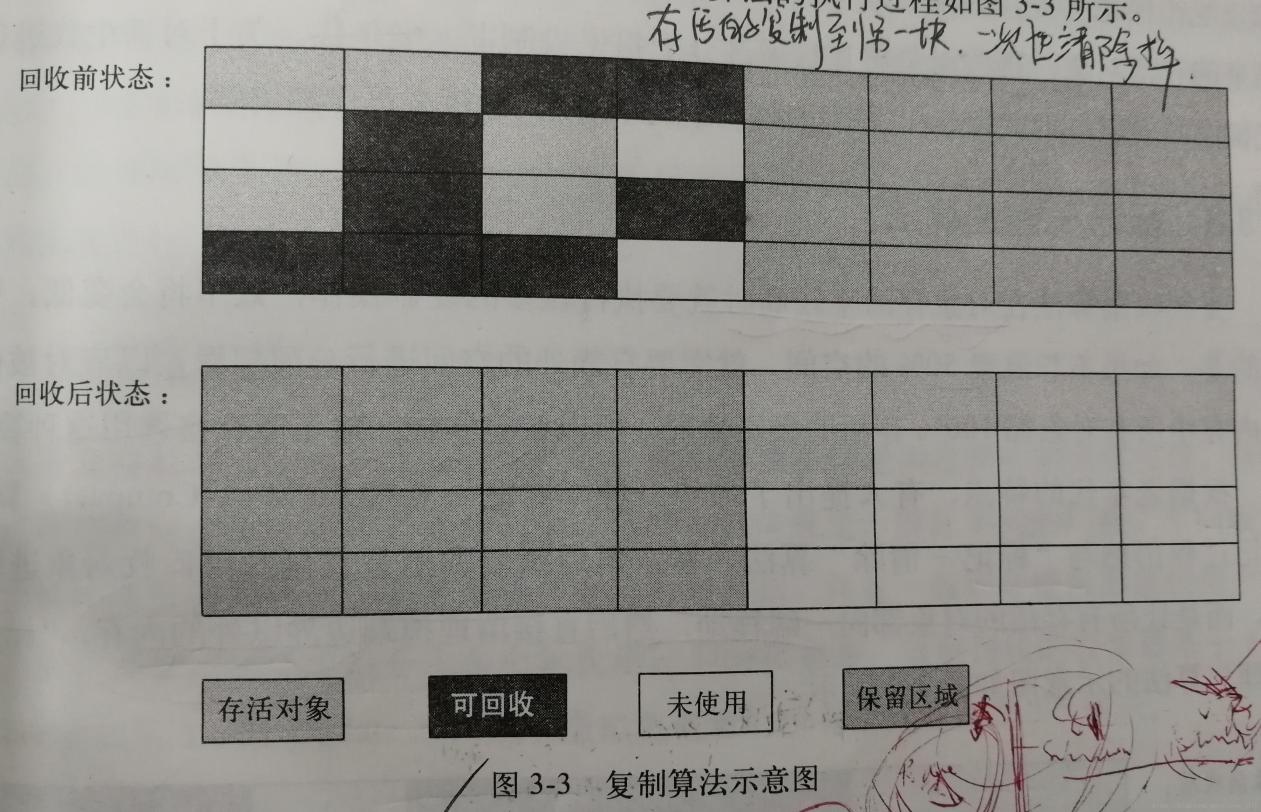
“标记-清除”算法是最基础的算法，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。它主要由两个缺点：一个是效率问题，标记和清除过程的效率都不高；另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。



#### 3.5.2 复制算法（Copying）（针对新生代）

为了解决标记清除算法的效率问题，出现了复制算法，它将可用内存按容量划分为大小相等的两块，每次使用其中的一块。当这块的内存用完了，就将还存活着的对象复制到另一块上面，然后再把已使用过的内存空间一次清理掉。优点是每次都是对其中的一块进行内存回收，内存分配时就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。缺点是将内存缩小为原来的一半，代价太高了一点。

**现在的商业虚拟机都采用复制收集算法来回收新生代，有研究表明，新生代中的对象98%是朝生夕死的，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中的一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地拷贝到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存是会被“浪费”的。当然，并不能保证每次回收都只有10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。即如果另外一块Survivor空间没有足够的空间存放上一次新生代收集下来的存活对象，这些对象将直接通过分配担保机制进入老年代。**



#### 3.5.3 标记-整理算法（Mark-Compact）（针对老年代）

      复制收集算法在对象存活率较高时就需要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用复制收集算法。

     根据老年代的特点提出了“标记-整理”算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

**标记-整理的步骤：**

标记阶段

整理阶段：移动存活对象，同时更新存活对象中所有指向被移动对象的指针

**整理的顺序**

    不同算法中，堆遍历的次数，整理的顺序，对象的迁移方式都有所不同。而整理顺序又会影响到程序的局部性。主要有以下3种顺序：

     1. 任意顺序：对象的移动方式和它们初始的对象排列及引用关系无关

       任意顺序整理实现简单，且执行速度快，但任意顺序可能会将原本相邻的对象打乱到不同的高速缓存行或者是虚拟内存页中，会降低赋值器的局部性。任意顺序算法只能处理单一大小的对象，或者针对大小不同的对象需要分批处理；

     2. 线性顺序：将具有关联关系的对象排列在一起

     3. 滑动顺序：将对象“滑动”到堆的一端，从而“挤出”垃圾，可以保持对象在堆中原有的顺序

    所有现代的标记-整理回收器均使用滑动整理，它不会改变对象的相对顺序，也就不会影响赋值器的空间局部性。复制式回收器甚至可以通过改变对象布局的方式，将对象与其父节点或者兄弟节点排列的更近以提高赋值器的空间局部性。

整理算法的限制，如整理过程需要2次或者3次遍历堆空间；对象头部可能需要一个额外的槽来保存迁移的信息。

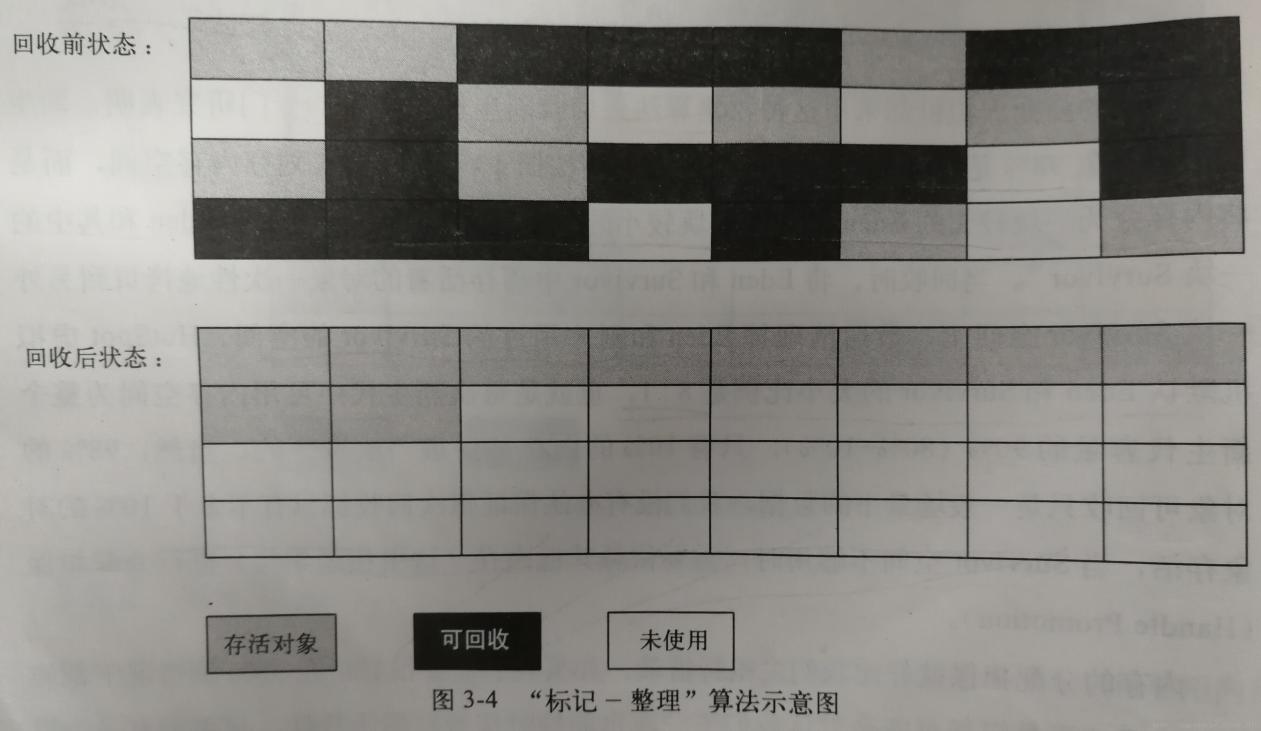
**部分整理算法：**

双指针回收算法：实现简单且速度快，但会打乱对象的原有布局

Lisp2算法（滑动回收算法）：需要在对象头用一个额外的槽来保存迁移完的地址

引线整理算法：可以在不引入额外空间开销的情况下实现滑动整理，但需要2次遍历堆，且遍历成本较高

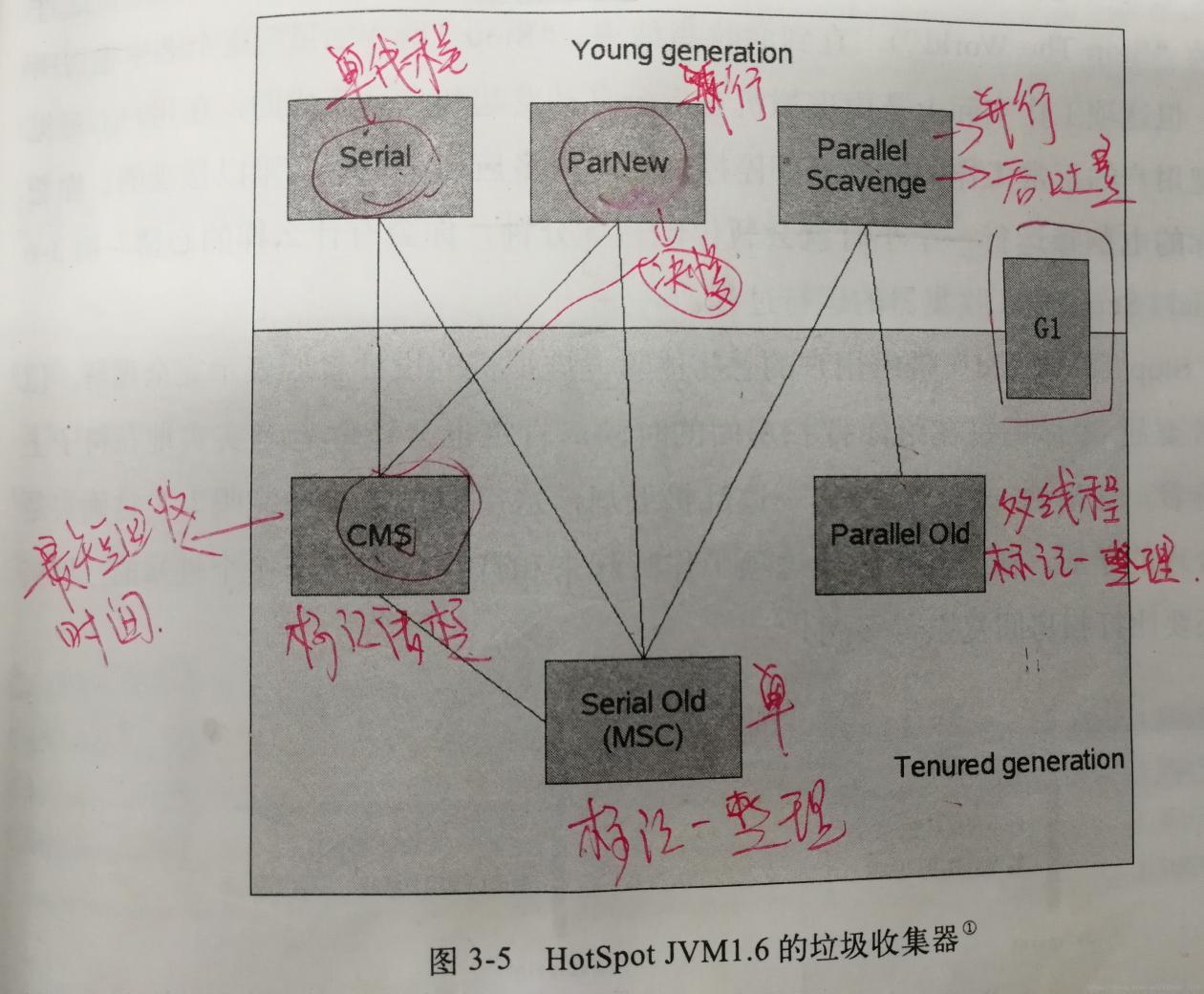
单次遍历算法：滑动回收，实时计算出对象的转发地址而不需要额外的开销



#### 3.5.4 分代收集算法（Generational Collection）

      当前商业虚拟机的垃圾收集都采用“分代收集”算法，这种算法并无新的方法，只是根据对象的存活周期的不同将内存划分为几块，一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

### **3.6 垃圾收集器**

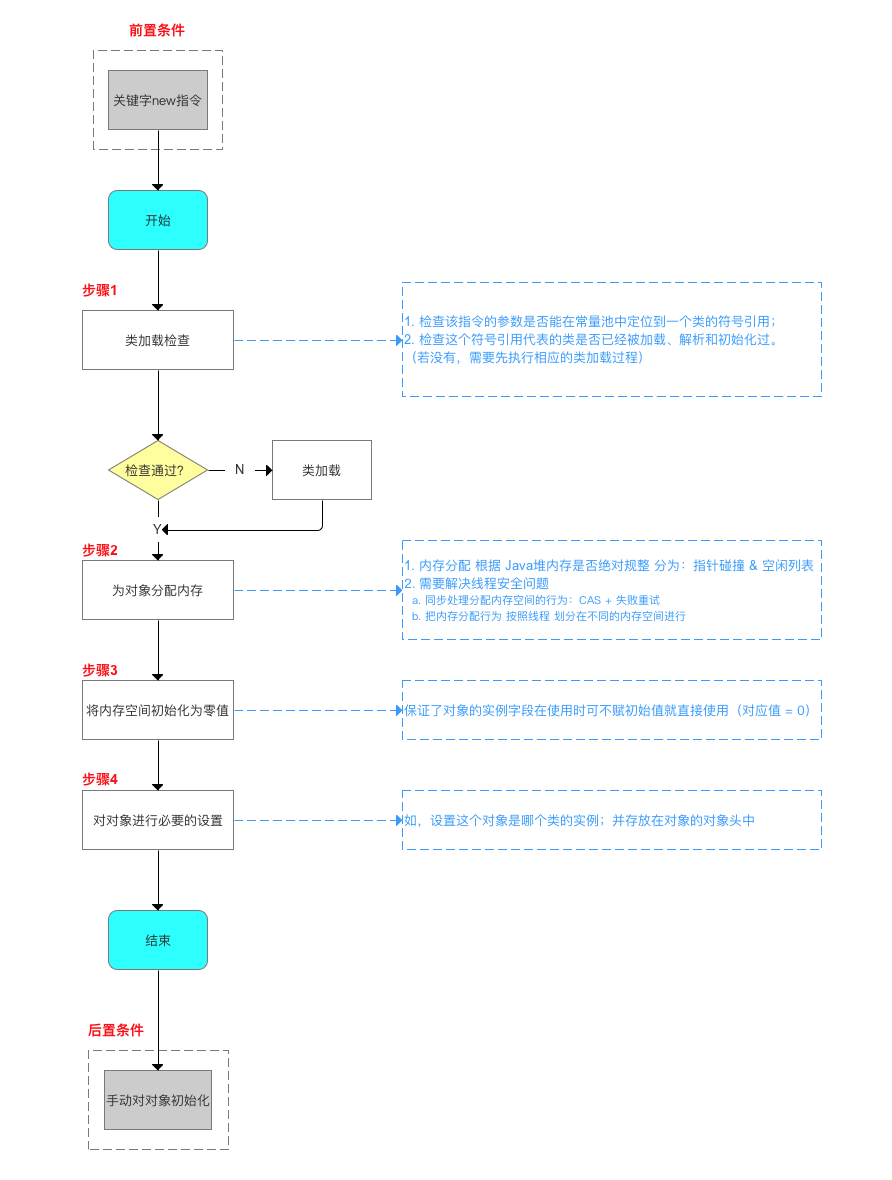


## 4 Java对象

### 4.0 对象组成

1. 对象头
2. 实例数据
3. 对齐填充字节

### 4.1 对象创建过程



**步骤1：类加载检查**

1. 检查 该new指令的参数 是否能在 常量池中 定位到一个类的符号引用
2. 检查 该类符号引用 代表的类是否已被加载、解析和初始化过

**步骤2：为对象分配内存**

虚拟机将为对象分配内存，即把一块确定大小的内存从 Java 堆中划分出来（对象所需内存的大小在类加载完成后便可完全确定）

内存分配 根据 **Java堆内存是否绝对规整** 分为两种方式：指针碰撞 & 空闲列表

1. Java堆内存 规整：已使用的内存在一边，未使用内存在另一边
2. Java堆内存 不规整：已使用的内存和未使用内存相互交错

**方式1：指针碰撞**

（1）假设Java堆内存绝对规整，内存分配将采用指针碰撞

（2）分配形式：已使用内存在一边，未使用内存在另一边，中间放一个作为分界点的指示器

那么，分配对象内存 = 把指针向 未使用内存 移动一段 与对象大小相等的距离

**方式2：空闲列表**

（1）假设Java堆内存不规整，内存分配将采用 **空闲列表**

（2）分配形式：虚拟机维护着一个 记录可用内存块 的列表，在分配时从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录

**额外知识**

（1）分配方式的选择 取决于 Java堆内存是否规整；

（2）而 Java堆是否规整 由所采用的垃圾收集器是否带有压缩整理功能决定。因此：

1. 使用带 Compact 过程的垃圾收集器时，采用指针碰撞；

如Serial、ParNew垃圾收集器

2. 使用基于 Mark\_sweep算法的垃圾收集器时，采用空闲列表。

如 CMS垃圾收集器

**特别注意**

对象创建在虚拟机中是非常频繁的操作，即使仅仅修改一个指针所指向的位置，在并发情况下也会引起线程不安全

如，正在给对象A分配内存，指针还没有来得及修改，对象B又同时使用了原来的指针来分配内存

**所以，给对象分配内存会存在线程不安全的问题。**

解决 线程不安全 有两种方案：

1. 同步处理分配内存空间的行为

虚拟机采用 **CAS + 失败重试的方式** 保证更新操作的原子性

2. 把内存分配行为 按照线程 划分在不同的内存空间进行

1. 即每个线程在 Java堆中预先分配一小块内存（本地线程分配缓冲（Thread Local Allocation Buffer ，TLAB）），哪个线程要分配内存，就在哪个线程的TLAB上分配，只有TLAB用完并分配新的TLAB时才需要同步锁。
2. 虚拟机是否使用TLAB，可以通过-XX:+/-UseTLAB参数来设定。

**步骤3： 将内存空间初始化为零值**

内存分配完成后，虚拟机需要将分配到的内存空间初始化为零（不包括对象头）

1. 保证了对象的实例字段在使用时可不赋初始值就直接使用（对应值 = 0）
2. 如使用本地线程分配缓冲（TLAB），这一工作过程也可以提前至TLAB分配时进行。

**步骤4： 对对象进行必要的设置**

如，设置 这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。

**这些信息存放在对象的对象头中**。

* 至此，从 Java 虚拟机的角度来看，一个新的 Java对象创建完毕
* 但从 Java 程序开发来说，对象创建才刚开始，需要进行一些初始化操作。

### **4.2 对象布局**

**问题：在 Java 对象创建后，到底是如何被存储在Java内存里的呢？**

答：在Java虚拟机（HotSpot）中，对象在 Java 内存中的 存储布局 可分为三块：

1. 对象头 存储区域
2. 实例数据 存储区域
3. 对齐填充 存储区域

****

### 4.3 对象访问定位

问：建立对象后，该如何访问对象呢？

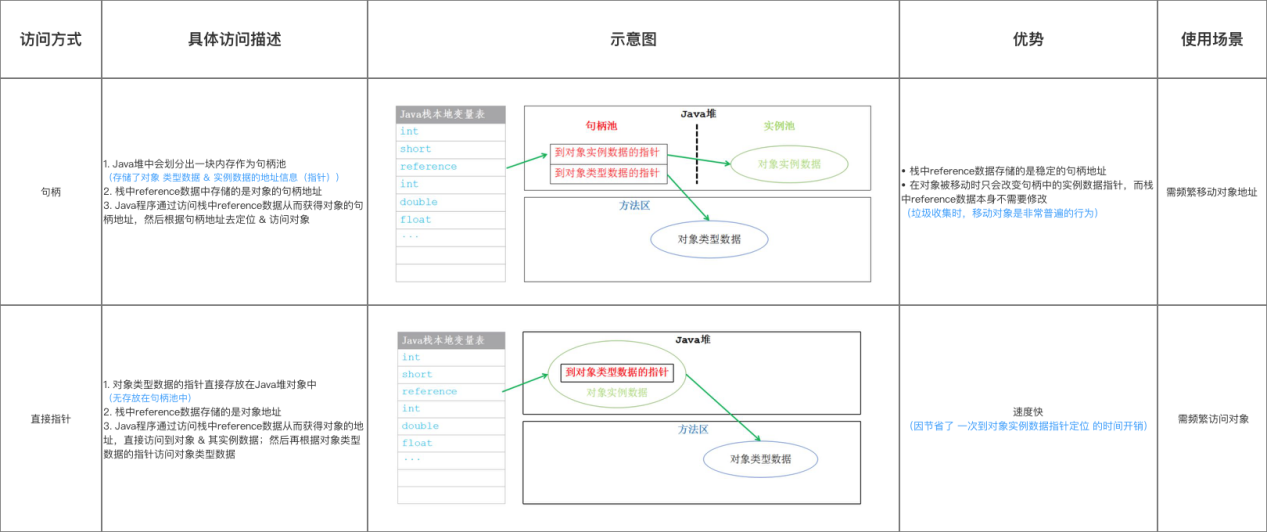
实际上需访问的是 对象类型数据 & 对象实例数据

答：Java程序 通过 栈上的引用类型数据（reference） 来访问Java堆上的对象。

由于引用类型数据（reference）在 Java虚拟机中只规定了一个指向对象的引用，但没定义该引用应该通过何种方式去定位、访问堆中的对象的具体位置

所以对象访问方式取决于虚拟机实现。目前主流的对象访问方式有两种：

* 句柄 访问
* 直接指针 访问

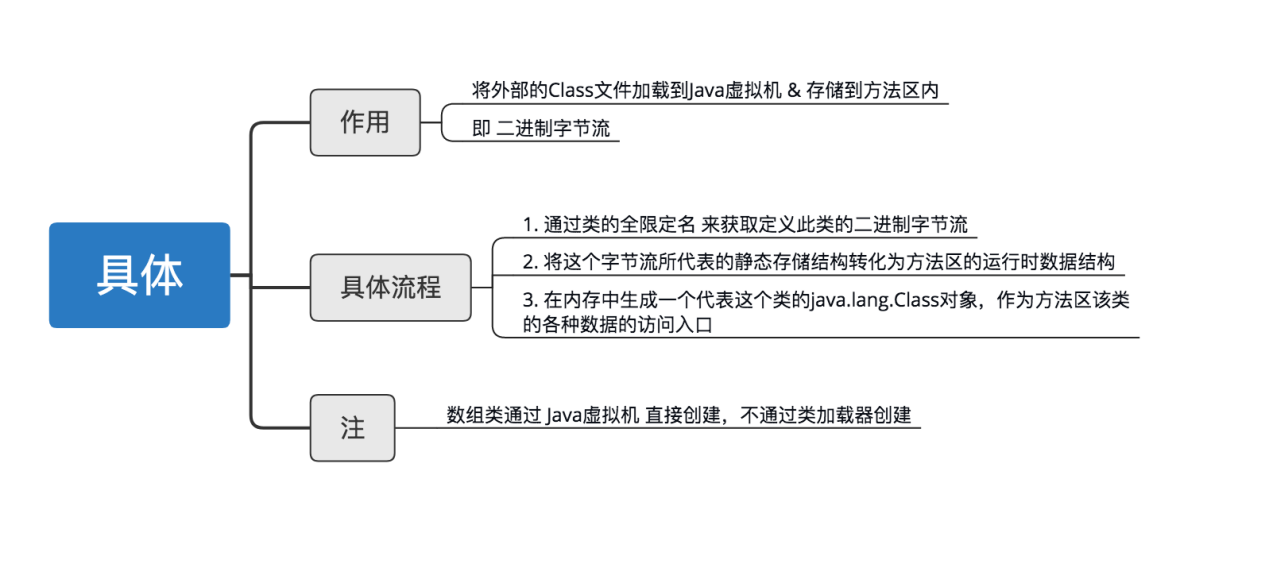
****

## 5 类加载器

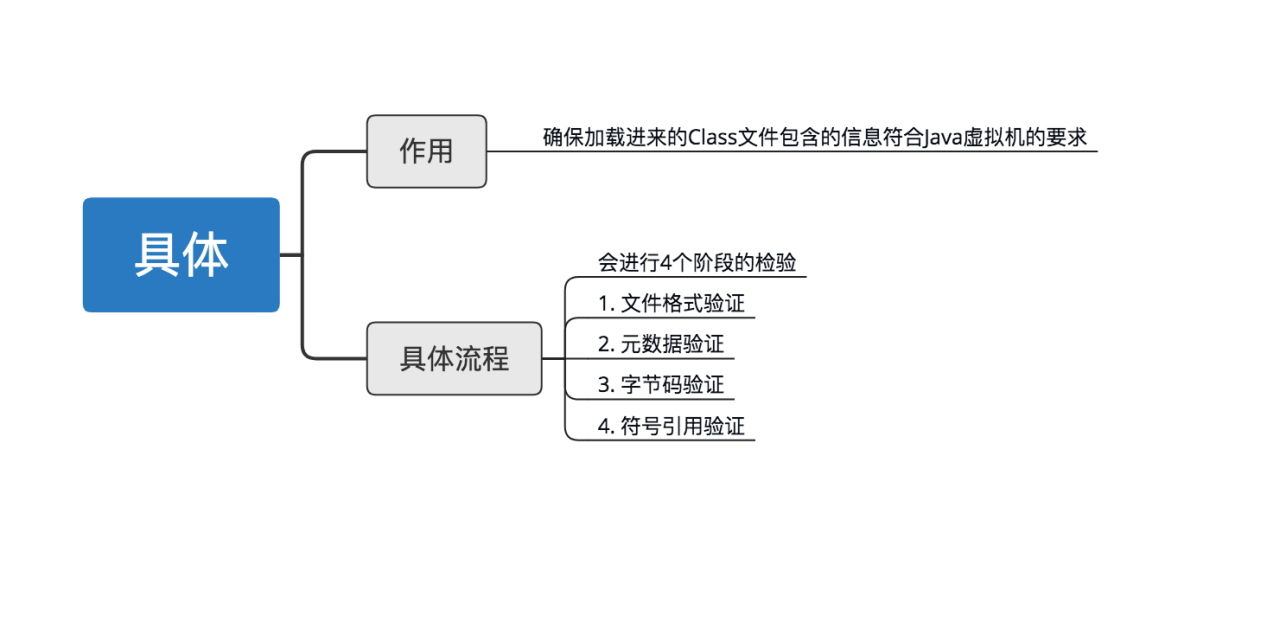
程序在启动的时候，并不会一次性加载程序所要用的所有class文件，而是根据程序的需要，通过Java的类加载机制（ClassLoader）来动态加载某个class文件到内存当中的，从而只有class文件被载入到了内存之后，才能被其它class所引用。所以ClassLoader就是用来动态加载class文件到内存当中用的。

### 5.1 类加载五个过程

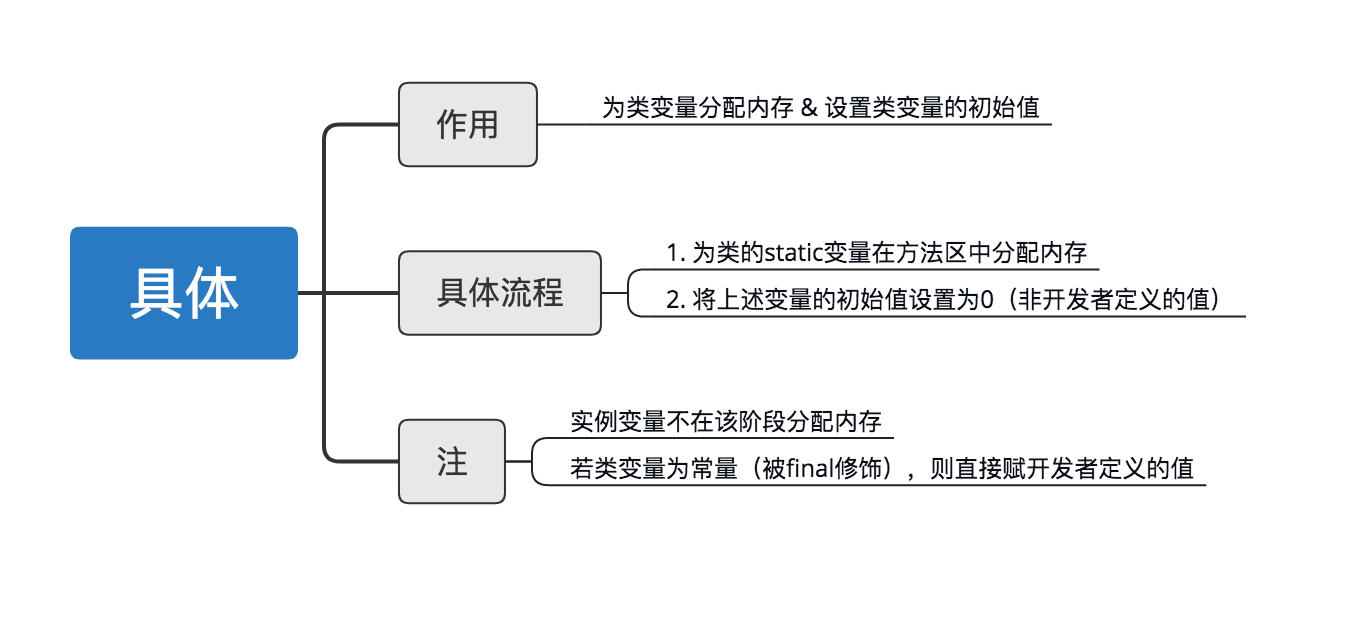
**步骤1：加载**



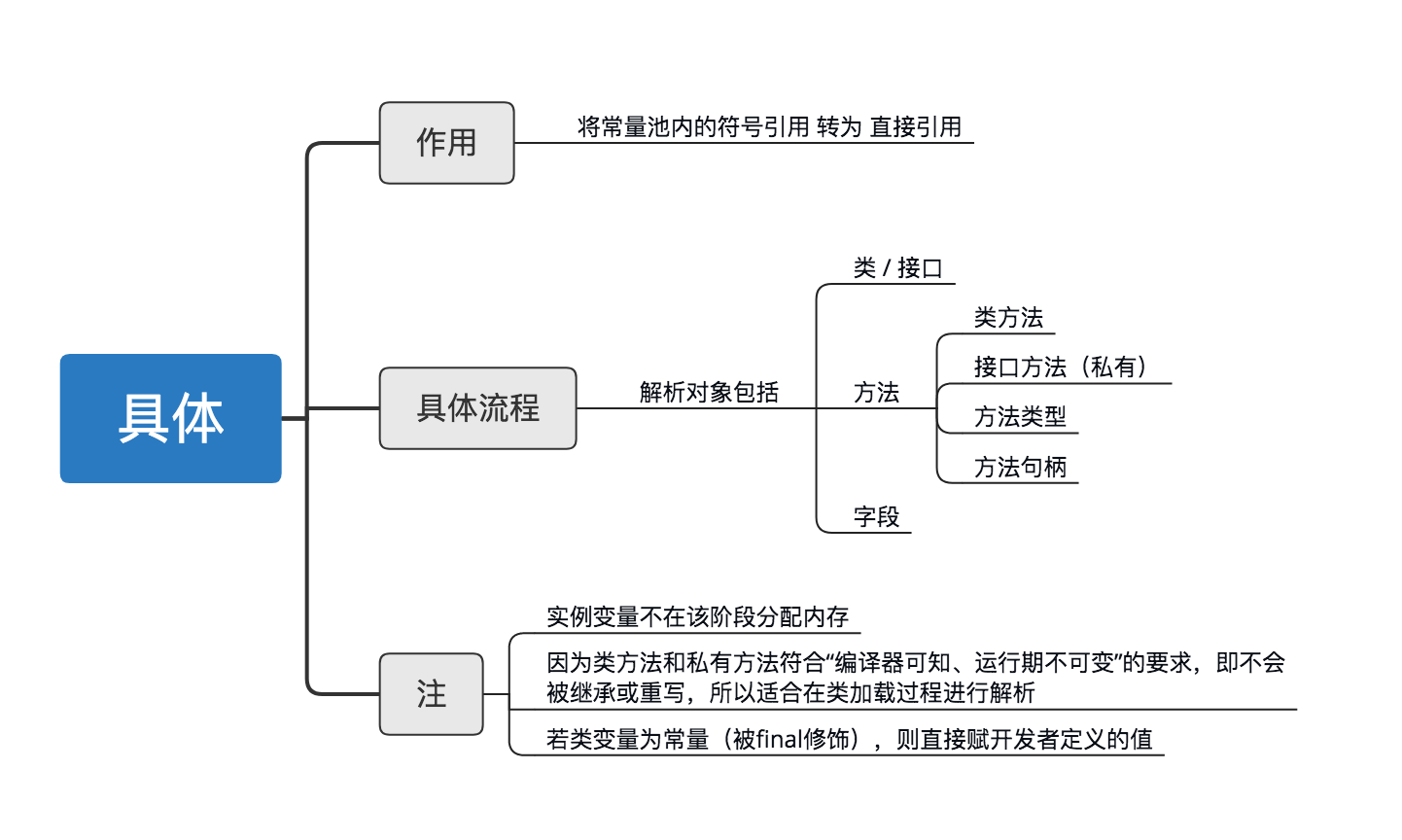
**步骤2：验证**



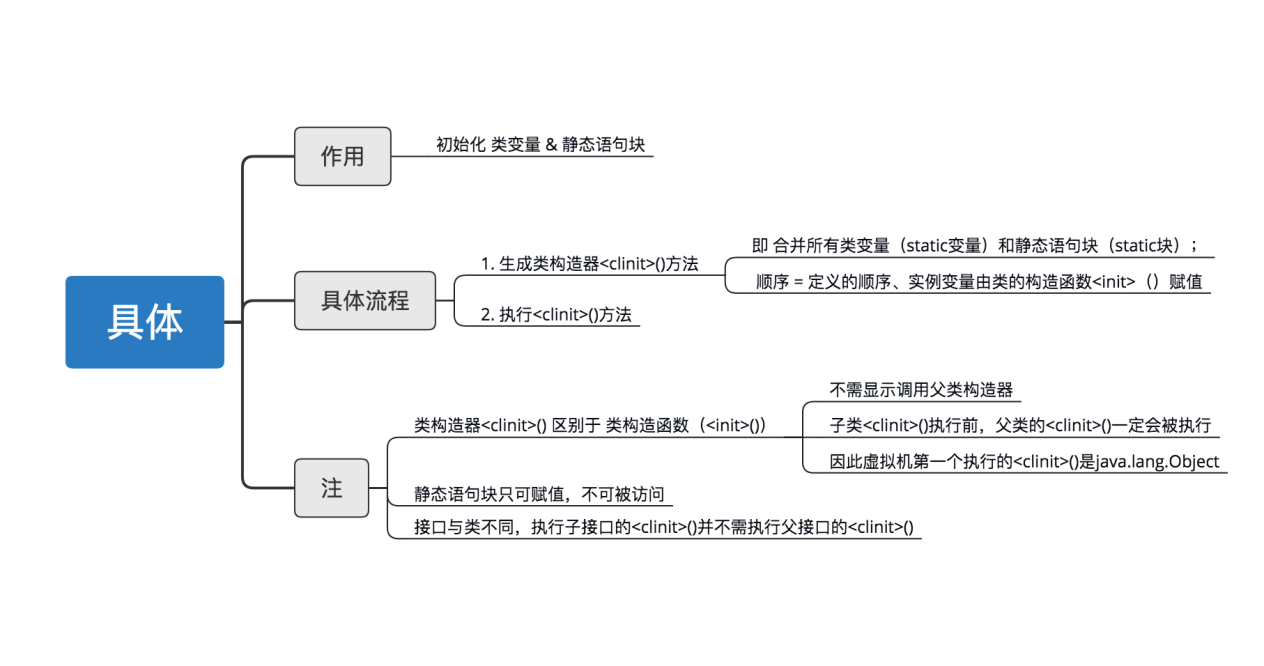
**步骤3：准备**



**步骤4：解析**

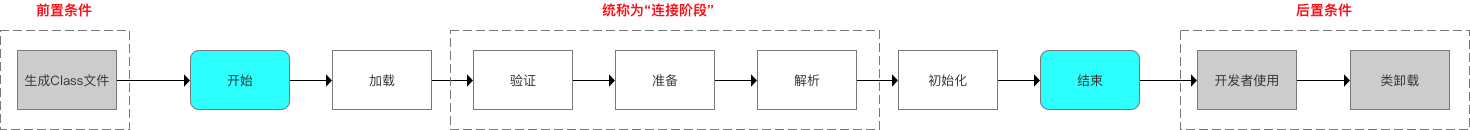


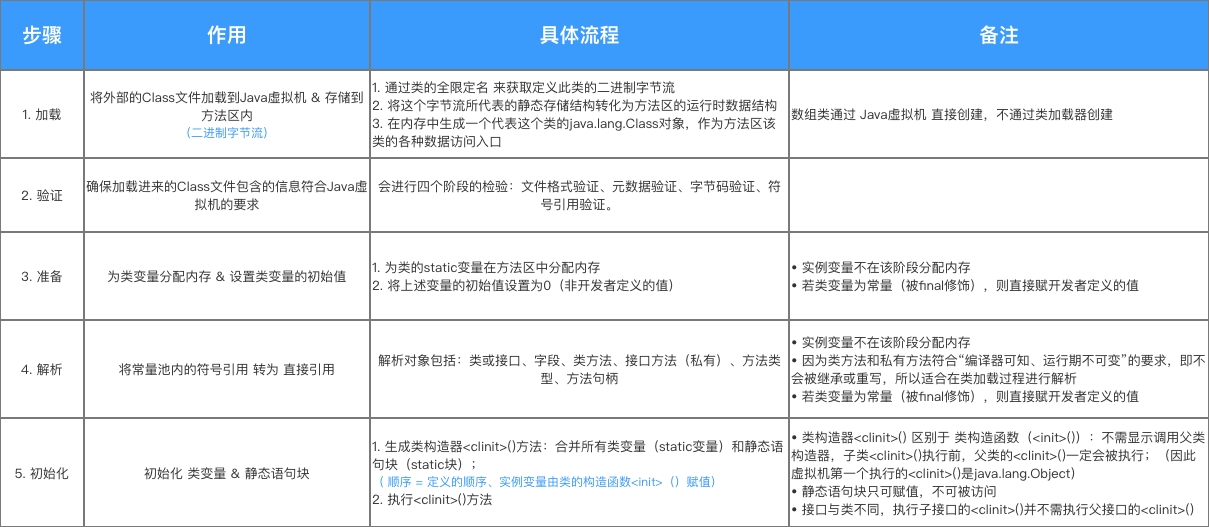
**步骤5：初始化**



**总结**

本文全面讲解类加载过程的5个步骤，总结如下





### 5.2、双亲委派原理

每个ClassLoader实例都有一个父类加载器的引用（不是继承关系，是一个包含的关系），虚拟机内置的类加载器（Bootstrap ClassLoader）本身没有父类加载器，但是可以用做其他ClassLoader实例的父类加载器。

当一个ClassLoader 实例需要加载某个类时，它会试图在亲自搜索这个类之前先把这个任务委托给它的父类加载器，这个过程是由上而下依次检查的，首先由顶层的类加载器Bootstrap CLassLoader进行加载，如果没有加载到，则把任务转交给Extension CLassLoader视图加载，如果也没有找到，则转交给AppCLassLoader进行加载，还是没有的话，则交给委托的发起者，由它到指定的文件系统或者网络等URL中进行加载类。还没有找到的话，则会抛出CLassNotFoundException异常。**否则将这个类生成一个类的定义，并将它加载到内存中，最后返回这个类在内存中的Class实例对象。**

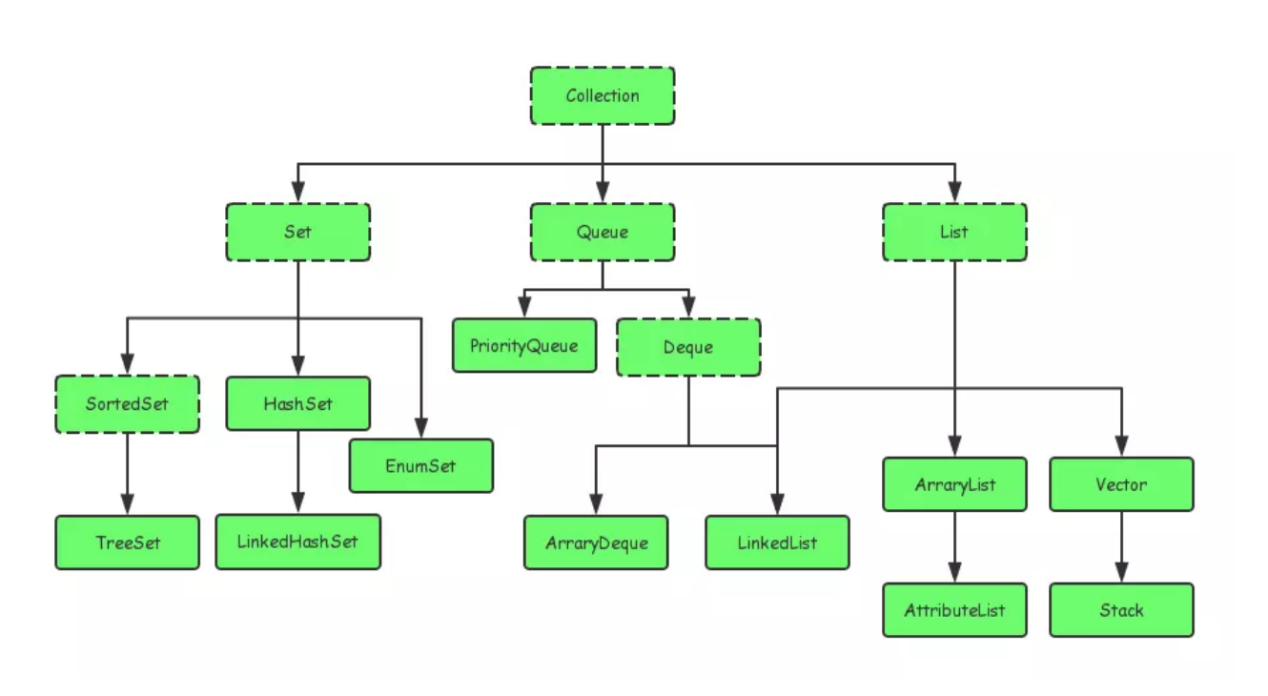
### 5.3 为什么使用双亲委托模型

JVM在判断两个class是否相同时，不仅要判断两个类名是否相同，还要判断是否是同一个类加载器加载的。

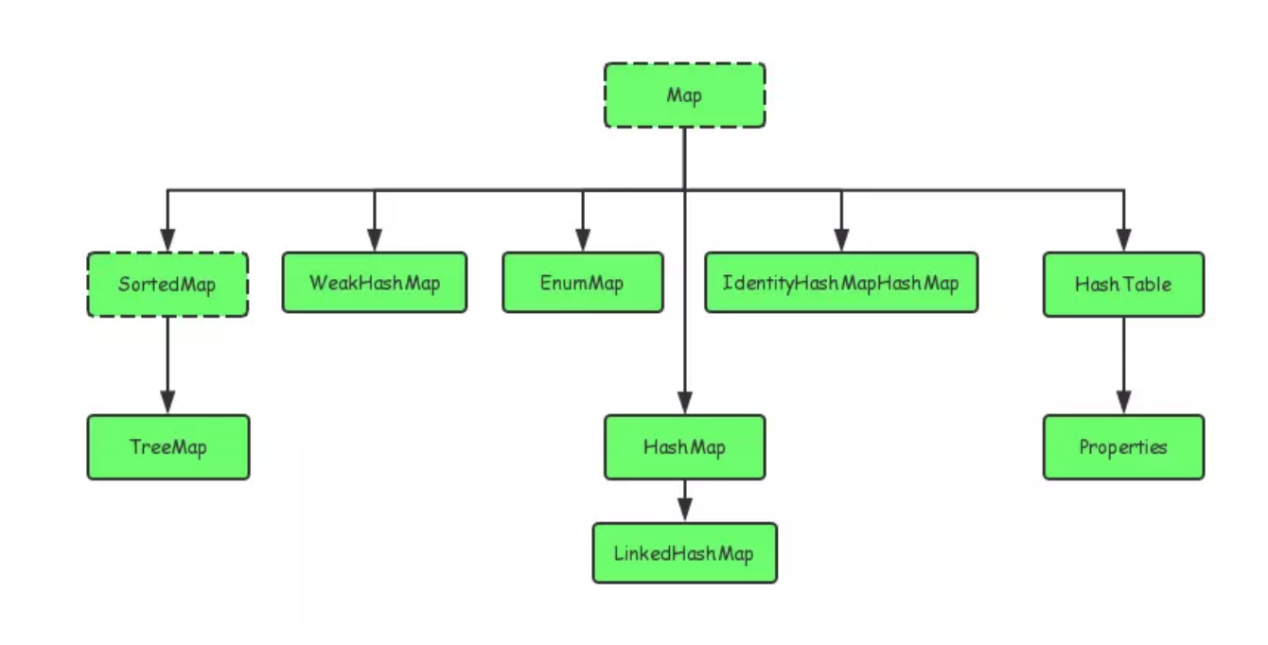
1. **避免重复加载**，父类已经加载了，则子CLassLoader没有必要再次加载。
2. **考虑安全因素**，假设自定义一个String类，除非改变JDK中CLassLoader的搜索类的默认算法，否则用户自定义的CLassLoader如法加载一个自己写的String类，因为String类在启动时就被引导类加载器Bootstrap CLassLoader加载了。

## 6、集合

Java集合类主要由两个接口派生出：Collection和Map，这两个接口是Java集合的根接口。



Collection接口是集合类的根接口，**[Java](https://link.juejin.im/?target=http%3A%2F%2Flib.csdn.net%2Fbase%2Fjavase" \t "_blank" \o "Java SE知识库)**中没有提供这个接口的直接的实现类。但是却让其被继承产生了两个接口，就是 Set和List。Set中不能包含重复的元素。List是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式。



Map是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value。

### 6.1、区别

1. List,Set都是继承自Collection接口，Map则不是;
2. **List特点：**元素有放入顺序，元素可重复; **Set特点：**元素无放入顺序，元素不可重复，重复元素会覆盖掉，（注意：元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的，加入Set 的Object必须定义equals()方法;
3. LinkedList、ArrayList、HashSet是非线程安全的，Vector是线程安全的;
4. HashMap是非线程安全的，HashTable是线程安全的;

### 6.2、List和Vector比较

1. Vector是多线程安全的，线程安全就是说多线程访问同一代码，不会产生不确定的结果。而ArrayList不是，这个可以从源码中看出，Vector类中的方法很多有synchronized进行修饰，这样就导致了Vector在效率上无法与ArrayList相比；
2. 两个都是采用的线性连续空间存储元素，但是当空间不足的时候，两个类的增加方式是不同。
3. Vector可以设置增长因子，而ArrayList不可以。
4. Vector是一种老的动态数组，是线程同步的，效率很低，一般不赞成使用。

### 6.3、HashSet如何保证不重复

HashSet底层通过HashMap来实现的，在往HashSet中添加元素是

public boolean add(E e) {  
 return map.put(e, *PRESENT*)==null;  
}

// Dummy value to associate with an Object in the backing Map  
private static final Object *PRESENT* = new Object();

在HashMap中进行查找是否存在这个key，value始终是一样的，PRESENT

1. **如果hash码值不相同，说明是一个新元素，存；**
2. **如果hash码值相同，且equles判断相等，说明元素已经存在，不存；**
3. **如果hash码值相同，且equles判断不相等，说明元素不存在，存；**

如果有元素和传入对象的hash值相等，那么，继续进行equles()判断，如果仍然相等，那么就认为传入元素已经存在，不再添加，结束，否则仍然添加；

### 6.4、HashSet与Treeset的适用场景

1. HashSet是基于Hash算法实现的，其性能通常都优于TreeSet。为快速查找而设计的Set，我们通常都应该使用HashSet，在我们需要排序的功能时，我们才使用TreeSet。
2. TreeSet 是二叉树（红黑树的树据结构）实现的,Treeset中的数据是自动排好序的，不允许放入null值
3. HashSet 是哈希表实现的,HashSet中的数据是无序的，可以放入null，但只能放入一个null，两者中的值都不能重复，就如数据库中唯一约束。
4. **HashSet是基于Hash算法实现的，其性能通常都优于TreeSet。为快速查找而设计的Set，我们通常都应该使用HashSet，在我们需要排序的功能时，我们才使用TreeSet。**

### 6.5、HashMap与TreeMap、HashTable的区别及适用场景

1. **HashMap 非线程安全，**基于哈希表(散列表)实现。使用HashMap要求添加的键类明确定义了hashCode()和equals()[可以重写hashCode()和equals()]，为了优化HashMap空间的使用，您可以调优初始容量和负载因子。其中散列表的冲突处理主要分两种，一种是开放定址法，另一种是链表法。HashMap的实现中采用的是链表法。
2. TreeMap：**非线程安全**基于红黑树实现，TreeMap没有调优选项，因为该树总处于平衡状态

### 6.6 ArrayList

至此，我们彻底明白了ArrayList的扩容机制了。首先创建一个空数组***elementData***，第一次插入数据时直接扩充至10，然后如果***elementData***的长度不足，就扩充1.5倍，如果扩充完还不够，就使用需要的长度作为***elementData***的长度。

这样的方式显然比我们例子中好一些，但是在遇到大量数据时还是会频繁的拷贝数据。那么如何缓解这种问题呢，ArrayList为我们提供了两种可行的方案：

1. 使用ArrayList(int initialCapacity)这个有参构造，在创建时就声明一个较大的大小，这样解决了频繁拷贝问题，但是需要我们提前预知数据的数量级，也会一直占有较大的内存。

（2）除了添加数据时可以自动扩容外，我们还可以在插入前先进行一次扩容。只要提前预知数据的数量级，就可以在需要时直接一次扩充到位，与ArrayList(int initialCapacity)相比的好处在于不必一直占有较大内存，同时数据拷贝的次数也大大减少了。这个方法就是**ensureCapacity(int minCapacity)**，其内部就是调用了ensureCapacityInternal(int minCapacity)。

### 6.7 HashMap

Hashmap底层为什么是线程不安全的？

并发场景下使用时容易出现死循环，在 HashMap 扩容的时候会调用 resize() 方法，就是这里的并发操作容易在一个桶上形成环形链表；这样当获取一个不存在的 key 时，计算出的 index 正好是环形链表的下标就会出现死循环；

在 1.7 中 hash 冲突采用的头插法形成的链表，在并发条件下会形成循环链表，一旦有查询落到了这个链表上，当获取不到值时就会死循环。

## 7 常量池

### 7.1、Interger中的128(-128~127)

1. 当数值范围为-128~127时：如果两个new出来Integer对象，即使值相同，通过“==”比较结果为false，但两个对象直接赋值，则通过“==”比较结果为“true，这一点与String非常相似。
2. 当数值不在-128~127时，无论通过哪种方式，即使两个对象的值相等，通过“==”比较，其结果为false；
3. 当一个Integer对象直接与一个int基本数据类型通过“==”比较，其结果与第一点相同；
4. Integer对象的hash值为数值本身；

@Override  
public int hashCode() {  
 return Integer.*hashCode*(value);  
}

### 7.2、为什么是-128-127?

在Integer类中有一个静态内部类IntegerCache，在IntegerCache类中有一个Integer数组，用以缓存当数值范围为-128~127时的Integer对象。

## 8、泛型

泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。 Java语言引入泛型的好处是安全简单。

**泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，提高代码的重用率。**

它提供了编译期的类型安全，确保你只能把正确类型的对象放入 集合中，避免了在运行时出现ClassCastException。

1. 泛型的类型参数只能是类类型（包括自定义类），不能是简单类型。
2. 同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的泛型类实例是不兼容的。
3. 泛型的类型参数可以有多个。
4. 泛型的参数类型可以使用extends语句，例如<T extends superclass>。习惯上称为“有界类型”。
5. 泛型的参数类型还可以是通配符类型。例如Class<?> classType = Class.forName("java.lang.String");

### 8.1、泛型擦除

Java中的泛型基本上都是在编译器这个层次来实现的。在生成的Java字节码中是不包含泛型中的类型信息的。**使用泛型的时候加上的类型参数，会在编译器在编译的时候去掉。这个过程就称为类型擦除**。

泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如 List<String>在运行时仅用一个List来表示。这样做的目的，是确保能和Java 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。

### 8.2、限定通配符

限定通配符对类型进行了限制。

1. 一种是<? extends T>它通过确保类型必须是T的子类来设定类型的上界，
2. 另一种是<? super T>它通过确保类型必须是T的父类来设定类型的下界。
3. 另一方面<?>表 示了非限定通配符，因为<?>可以用任意类型来替代。

例如List<? extends Number>可以接受List<Integer>或List<Float>。

### 8.3、泛型面试题

* **你可以把List<String>传递给一个接受List<Object>参数的方法吗？**

对任何一个不太熟悉泛型的人来说，这个Java泛型题目看起来令人疑惑，因为乍看起来String是一种Object，所以 List<String>应当可以用在需要List<Object>的地方，但是事实并非如此。真这样做的话会导致编译错误。如 果你再深一步考虑，你会发现Java这样做是有意义的，因为List<Object>可以存储任何类型的对象包括String, Integer等等，而List<String>却只能用来存储Strings。

* **Array中可以用泛型吗?**

Array事实上并不支持泛型，这也是为什么Joshua Bloch在Effective Java一书中建议使用List来代替Array，因为List可以提供编译期的类型安全保证，而Array却不能

* **Java中List<Object>和原始类型List之间的区别?**

原始类型和带参数类型<Object>之间的主要区别是，在编译时编译器不会对原始类型进行类型安全检查，却会对带参数的类型进行检 查，通过使用Object作为类型，可以告知编译器该方法可以接受任何类型的对象，比如String或Integer。这道题的考察点在于对泛型中原始类 型的正确理解。它们之间的第二点区别是，你可以把任何带参数的类型传递给原始类型List，但却不能把List<String>传递给接受 List<Object>的方法，因为会产生编译错误。

**List<?> 是一个未知类型的List，而List<Object> 其实是任意类型的List。你可以把List<String>, List<Integer>赋值给List<?>，却不能把List<String>赋值给 List<Object>。**

## 9、反射

### 9.1、概念

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

### 9.2、作用

Java反射机制主要提供了以下功能： 在运行时判断任意一个对象所属的类；在运行时构造任意一个类的对象；在运行时判断任意一个类所具有的成员变量和方法；在运行时调用任意一个对象的方法；生成动态代理。

### 9.3、反射效率低的原因

（1）Method#invoke 方法会对参数做封装和解封操作

我们可以看到，invoke 方法的参数是 Object[] 类型，也就是说，如果方法参数是简单类型的话，需要在此转化成 Object 类型，例如 long ,在 javac compile 的时候 用了Long.valueOf() 转型，也就大量了生成了Long 的 Object, 同时 传入的参数是Object[]数值,那还需要额外封装object数组。  
而在上面 MethodAccessorGenerator#emitInvoke 方法里我们看到，生成的字节码时，会把参数数组拆解开来，把参数恢复到没有被 Object[] 包装前的样子，同时还要对参数做校验，这里就涉及到了解封操作。  
因此，在反射调用的时候，因为封装和解封，产生了额外的不必要的内存浪费，当调用次数达到一定量的时候，还会导致 GC。

（2）需要检查方法可见性

通过上面的源码分析，我们会发现，反射时每次调用都必须检查方法的可见性（在 Method.invoke 里）

（3）需要校验参数

反射时也必须检查每个实际参数与形式参数的类型匹配性（在NativeMethodAccessorImpl.invoke0 里或者生成的 Java 版 MethodAccessor.invoke 里）；

1. 反射方法难以内联  
   Method invoke 就像是个独木桥一样，各处的反射调用都要挤过去，在调用点上收集到的类型信息就会很乱，影响内联程序的判断，使得 Method.invoke() 自身难以被内联到调用方。参见 [www.iteye.com/blog/rednax…](https://links.jianshu.com/go?to=https://www.iteye.com/blog/rednaxelafx-548536" \t "/Users/admin/Documents\\x/_blank)

（5）JIT 无法优化  
因为反射涉及到动态加载的类型，所以无法进行优化。

## 10、代理

代理这个词大家肯定已经非常熟悉，因为现实中接触的很多，其实现实中的东西恰恰可以非常形象和直观地反映出模式的抽象过程以及本质。现在房子不是吵得热火朝天吗？我们就以房子为例，来拨开代理的面纱。

假设你有一套房子要卖，一种方法是你直接去网上发布出售信息，然后直接带要买房子的人来看房子、过户等一直到房子卖出去，但是可能你很忙，你没有时间去处理这些事情，所以你可以去找中介，让中介帮你处理这些琐碎事情，中介实际上就是你的代理。本来是你要做的事情，现在中介帮助你一一处理，对于买方来说跟你直接交易跟同中介直接交易没有任何差异，买方甚至可能觉察不到你的存在，这实际上就是代理的一个最大好处。

接下来我们再深入考虑一下为什么你不直接买房子而需要中介？其实一个问题恰恰解答了什么时候该用代理模式的问题。

**原因一：你可能在外地上班，买房子的人没法找到你直接交易。**

对应到我们程序设计的时候就是：客户端无法直接操作实际对象。那么为什么无法直接操作？一种情况是你需要调用的对象在另外一台机器上，你需要跨越网络才能访问，如果让你直接coding去调用，你需要处理网络连接、处理打包、解包等等非常复杂的步骤，所以为了简化客户端的处理，我们使用代理模式，在客户端建立一个远程对象的代理，客户端就象调用本地对象一样调用该代理，再由代理去跟实际对象联系，对于客户端来说可能根本没有感觉到调用的东西在网络另外一端，这实际上就是Web Service的工作原理。另一种情况虽然你所要调用的对象就在本地，但是由于调用非常耗时，你怕影响你正常的操作，所以特意找个代理来处理这种耗时情况，一个最容易理解的就是Word里面装了很大一张图片，在word被打开的时候我们肯定要加载里面的内容一起打开，但是如果等加载完这个大图片再打开Word用户等得可能早已经跳脚了，所以我们可以为这个图片设置一个代理，让代理慢慢打开这个图片而不影响Word本来的打开的功能。申明一下我只是猜可能Word是这么做的，具体到底怎么做的，俺也不知道。

**原因二：你不知道怎么办过户手续，或者说除了你现在会干的事情外，还需要做其他的事情才能达成目的。**

对应到我们程序设计的时候就是：除了当前类能够提供的功能外，我们还需要补充一些其他功能。最容易想到的情况就是权限过滤，我有一个类做某项业务，但是由于安全原因只有某些用户才可以调用这个类，此时我们就可以做一个该类的代理类，要求所有请求必须通过该代理类，由该代理类做权限判断，如果安全则调用实际类的业务开始处理。可能有人说为什么我要多加个代理类？我只需要在原来类的方法里面加上权限过滤不就完了吗？在程序设计中有一个类的单一性原则问题，这个原则很简单，就是每个类的功能尽可能单一。为什么要单一，因为只有功能单一这个类被改动的可能性才会最小，就拿刚才的例子来说，如果你将权限判断放在当前类里面，当前这个类就既要负责自己本身业务逻辑、又要负责权限判断，那么就有两个导致该类变化的原因，现在如果权限规则一旦变化，这个类就必需得改，显然这不是一个好的设计。

好了，原理的东西已经讲得差不多了，要是再讲个没完可能大家要扔砖头了。呵呵，接下来就看看怎么来实现代理。

## 11 注解

注解的本质就是一个继承了 Annotation 接口的接口。有关这一点，你可以去反编译任意一个注解类，你会得到结果的。**一个注解准确意义上来说，只不过是一种特殊的注释而已，如果没有解析它的代码，它可能连注释都不如。**

## 12 线程内存模型和锁

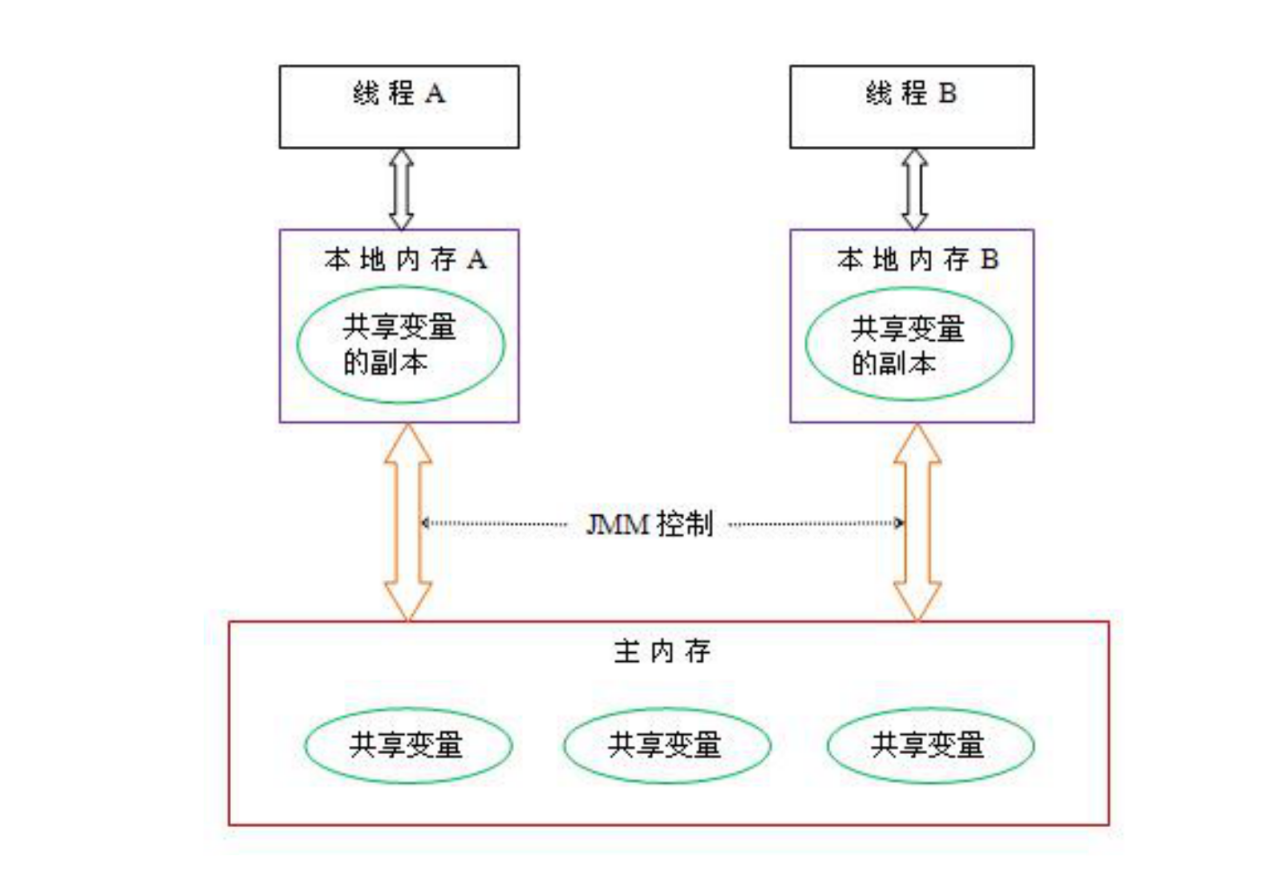
### 12.0 线程状态

1. 新建、初始状态（New） ：线程对象被创建后就进入了新建状态，Thread thread = new Thread();

2. 就绪（Runnable）：也被称之为“可执行状态”，当线程被new出来后，其他的线程调用了该对象的start()方法，即thread.start()，此时线程位于“可运行线程池”中，只等待获取CPU的使用权，随时可以被CPU调用。进入就绪状态的进程除CPU之外，其他运行所需的资源都已经全部获得。

3. 运行（Running）：线程获取CPU权限开始执行。注意：线程只能从就绪状态进入到运行状态。

4. 阻塞（Bloacked）：阻塞状态是线程因为某种原因放弃CPU的使用权，暂时停止运行，知道线程进入就绪状态后才能有机会转到运行状态。



首先，所有的变量都是储存在主内存中的，每个线程都有自己的独立的工作内存， 里面保存的是该线程使用到的变量的副本（它们是内存这些变量的一份拷贝，而 变量的原件是保存在主内存的）。 线程修改的变量的值是修改的该线程的工作 内存中的变量的副本，然后再将修改后的值刷新到主内存中。 此时，其他线程 能否及时地看到主内存中该变量的更新的值，这时就涉及到了“可见性”问题。

（1）线程对共享变量的所有操作都必须在自己的工作内存中进行，不能直接在主内存中读写；

（2）不同的线程之间不能直接访问其他线程工作内存的变量， 线程之间变量值得传递只能通过主内存来传递。

**实现共享变量的可见性，必须保证两点：**

1. 线程修改后的共享变量值能够及时从线程工作内存中刷新到主内存中；

（2）其他线程能够及时把共享变量的最新值从主内存更新到自己的工作内存中。

### 12.1 线程安全

**线程安全**：

就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。 线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

**如何保证**：

（1）使用线程安全的类；

（2）使用synchronized同步代码块，或者用Lock锁；

（3）多线程并发情况下，线程共享的变量改为方法局部级变量；

### 12.2 死锁

**产生死锁的四个条件**

**（1）互斥**：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。

**（2）占有且等待**：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。

**（3）不可抢占**：别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。

**（4）循环等待**：存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源。

### 12.3 volatile属性

Link:<https://cloud.tencent.com/developer/article/1446555>

**指令重排序：重排序不会对单线程带来内存可见性问题。但是多线程交互时，指令重排序可能会造成 内存可见性问题。**

1. **实现可见性原理**

如何实现内存可见性：通过**加入内存屏障和禁止重排序**来优化实现的。

（1）对 volatile 变量执行写操作时，会在写操作后加入一条 store 屏障指令；

（2）对 volatile 变量执行读操作时，会在读操作前加入一条 load 屏障指令。

**（1）线程写 volatile 变量的过程：**

1.改变线程工作内存中 volatile 变量副本的值

1. 将改变后的副本的值从工作内存刷新的主内存
2. **线程读 volatile 变量的过程：**

1.从主内存中读取 volatile 变量的最新值到线程的工作内存中

2.从工作内存中读取 volatile 变量的副本

**（2）保证有序性**

处理器为了提高运行效率，在JVM中的及时编译存在指令重排序的优化，它会改变各个语句的执行顺序，但是不改变运行结果，

**（3）不保证原子性**

原子性就是指操作不可分割，比如i=1这个就不可分割，i++就包含三个操作：读取i的值，进行加1操作，写入新的值,这就可分割不是原子操作。如果将i换成volatile int i，将i=1换成i++，当线程一执行到对i加1操作时，还没有来得及写入新值时，线程二执行j=i,线程获取的i值任然为0；可以通过synchronized和Lock来实现更大范围操作的原子性。

### 12.4 synchronized 实现原理

**synchronized 能够保证同步方法或同步代码块中变量的复合操作的原子性； 也能保证变量的可见性。**

**作用：**确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性。

**synchronized 如何实现内存原子性**：

持有同一个锁的两个同步块只能串行地进入

**synchronized 如何实现内存可见性：**

（1）线程加锁时，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值。

（2）线程解锁时，必须把共享变量的最新值刷 新到主内存中。

**synchronized 线程执行互斥代码的六个过程：**

1. 获得互斥锁；
2. 清空工作内存；

（3）从主内存中拷贝变量的最新值到工作内存；

1. 执行代码；

（5）将更改后的共享变量的值刷新到主内存；

（6）释放互斥锁。

**volatile 和 synchronized 的区别**

1. volatile 本质是在告诉 jvm 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

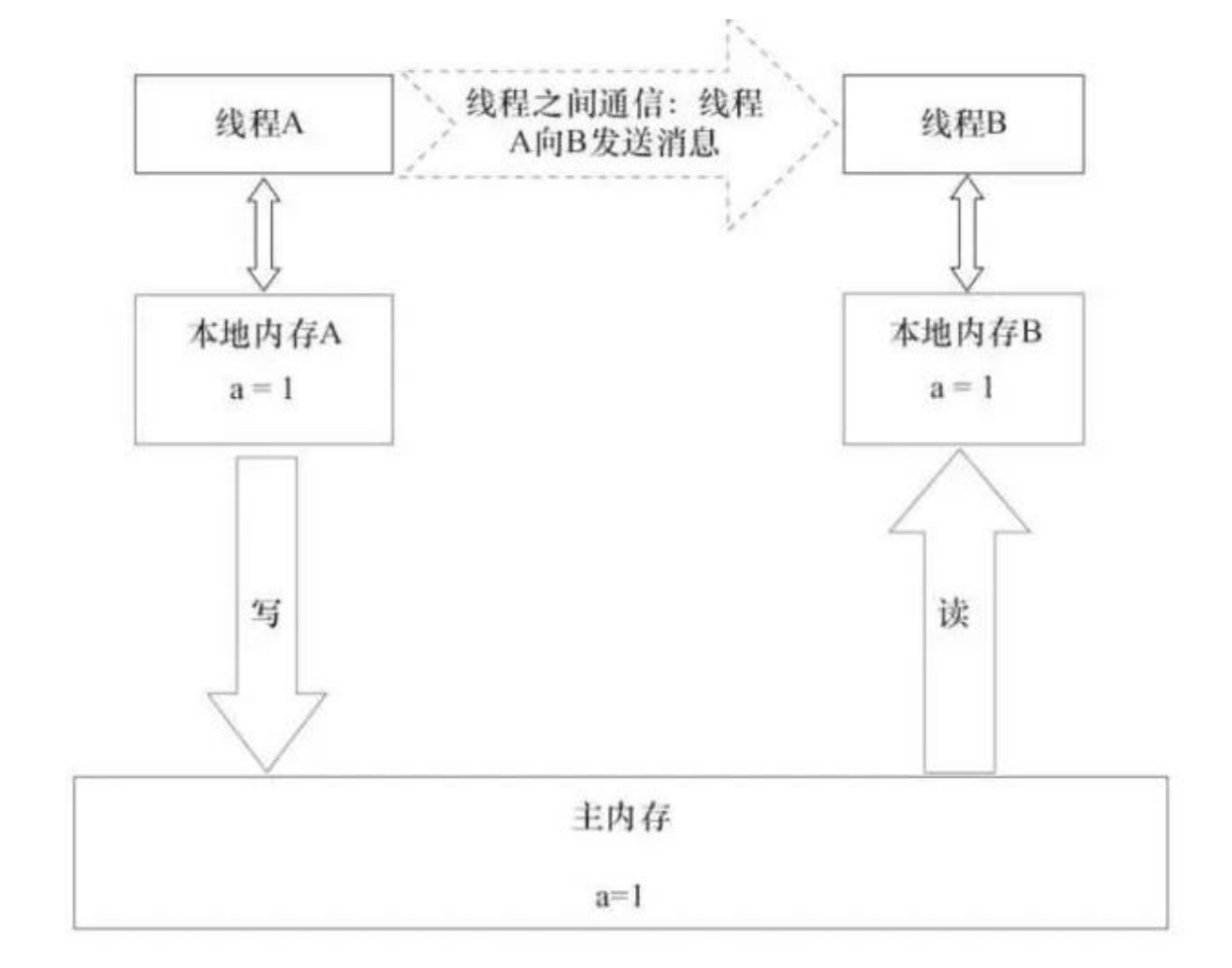
2. volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的

3. volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized则可以保证变量的修改可见性和原子性。

4. volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

5. volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

**线程间通信**

****

**锁的内存语义：**

(1)当线程释放锁时，JMM 会把该线程对应的本地内存中的共享变量刷新到主内存中

(2) 当线程获取锁时，JMM 会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须从主内存中读取共享变量

**锁释放和锁获取的内存语义：**

(1)线程 A 释放一个锁，实质上是线程 A 向接下来将要获取这个锁的某个线程发出了（线程 A 对共享变量所做修改的）消息。

(2)线程 B 获取一个锁，实质上是线程 B 接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。

(3)线程 A 释放锁，随后线程 B 获取这个锁，这个过程实质上是线程 A 通过主内存向线程 B 发送消息

### 12.5 ThreadLocal与Synchronized

ThreadLocal<T>其实是与线程绑定的一个变量，主要实现相同线程共享数据，不同线程隔离数据的工具集合。它的本质就是一个内部的静态map，key是当前线程的一个句柄，value是需要保存的值。

ThreadLocal和Synchonized都用于解决多线程并发访问。但是ThreadLocal与synchronized有本质的区别。

1. Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。
2. Synchronized是利用锁的机制，使变量或代码块在某一时该只能被一个线程访问。而ThreadLocal为每一个线程都提供了变量的副本，使得每个线程在某一时间访问到的并不是同一个对象，这样就隔离了多个线程对数据的数据共享。而Synchronized却正好相反，它用于在多个线程间通信时能够获得数据共享。

当需要使用多线程时，有个变量恰巧不需要共享，此时就不必使用 synchronized 这么麻烦的关键字来锁住，每个线程都相当于在堆内存中开辟一个空间，线程中带有对共享变量的缓冲区，通过缓冲区将堆内存中的共享变量进行读取和操作，ThreadLocal 相当于线程内的内存，一个局部变量。每次可以对线程自身的数据读取和操作，并不需要通过缓冲区与 主内存中的变量进行交互。并不会像 synchronized 那样修改主内存的数据，再将主内存的数据复制到线程内的工作内存。ThreadLocal 可以让线程独占资源，存储于线程内部，避免线程堵塞造成 CPU 吞吐下降。在每个 Thread 中包含一个 ThreadLocalMap，ThreadLocalMap 的 key 是 ThreadLocal 的对象，value 是独

享数据。

## 13 线程

Link:<https://blog.csdn.net/programmer_at/article/details/79799267>

### 13.1 线程创建方式

java中有三种创建线程的方式，或者说四种

（1）继承Thread类实现多线程

（2）实现Runnable接口

（3）实现Callable接口

（4）通过线程池

线程状态：

1. NEW，创建线程未start之前
2. RUNNABLE，执行中，可能在等待CPU可能在执行中，可分为READY和RUNNING状态
3. BLOCKED，
4. WAITING
5. TIMED\_WAITING
6. TERMINATED

### 13.2 线程池原理

**缺点**

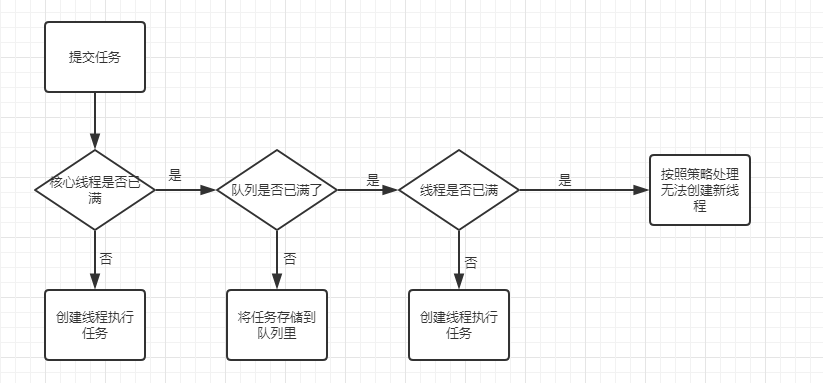
a、每次new Thread新建对象，性能差。

b、缺乏统一管理，可能无限制的新建线程，过多占用系统资源导致死机或OOM

**优点**

a、重用存在的线程，减少对象创建，消亡的开销

b、有效控制最大并发线程数，提高系统资源利用率



**当一个任务提交至线程池之后：**

1. 线程池首先判断核心线程池里的线程是否已经满了。如果不是，则创建一个新的工作线程来执行任务。否则进入2.

2. 判断工作队列是否已经满了，倘若还没有满，将线程放入工作队列。否则进入3.

3. 判断线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行。如果线程池满了，则交给饱和策略来处理任务。

**ThreadPoolExecutor执行execute()流程：**

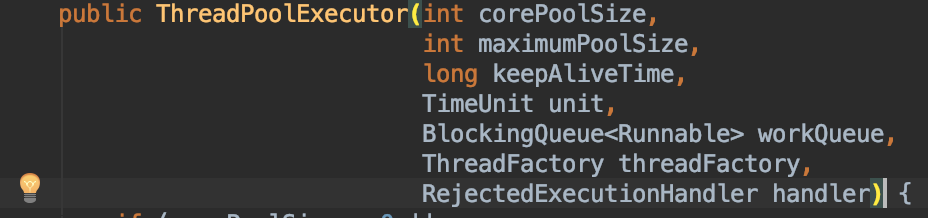
当一个任务提交至线程池之后，

1. 线程池首先当前运行的线程数量是否少于corePoolSize。如果是，则创建一个新的工作线程来执行任务。如果都在执行任务，则进入2.

2. 判断BlockingQueue是否已经满了，倘若还没有满，则将线程放入BlockingQueue。否则进入3.

3. 如果创建一个新的工作线程将使当前运行的线程数量超过maximumPoolSize，则交给RejectedExecutionHandler来处理任务。

当ThreadPoolExecutor创建新线程时，通过CAS来更新线程池的状态ctl.



**corePoolSize**

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize, 即使有其他空闲线程能够执行新来的任务, 也会继续创建线程；如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

**maximumPoolSize**

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize；当阻塞队列是无界队列, 则maximumPoolSize则不起作用, 因为无法提交至核心线程池的线程会一直持续地放入workQueue.

**keepAliveTime**

线程空闲时的存活时间，即当线程没有任务执行时，该线程继续存活的时间；默认情况下，该参数只在线程数大于corePoolSize时才有用, 超过这个时间的空闲线程将被终止；

**unit**

keepAliveTime的单位

**threadFactory**

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名。默认为DefaultThreadFactory

**workQueue**

用来保存等待被执行的任务的阻塞队列. 在JDK中提供了如下阻塞队列：

(1) ArrayBlockingQueue：基于数组结构的有界阻塞队列，按FIFO排序任务；

(2) LinkedBlockingQuene：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于ArrayBlockingQuene；

(3) SynchronousQuene：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene；

(4) priorityBlockingQuene：具有优先级的无界阻塞队列；

**handler**

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

AbortPolicy：直接抛出异常，默认策略；

CallerRunsPolicy：用调用者所在的线程来执行任务；

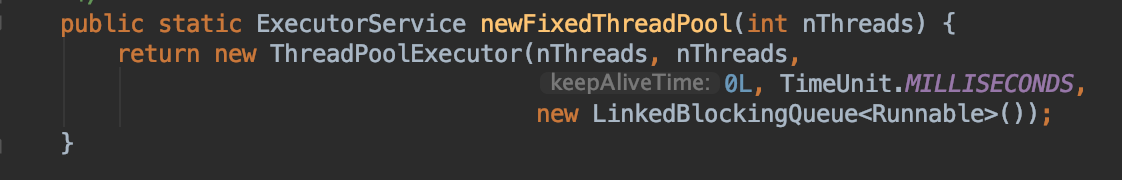
DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

### 13.3 线程池分类

**（1）newFixedThreadPool**



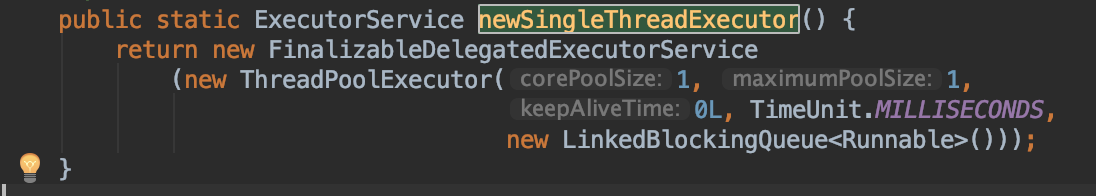
**线程池的线程数量达corePoolSize后，即使线程池没有可执行任务时，也不会释放线程。**

**FixedThreadPool的工作队列为无界队列LinkedBlockingQueue(队列容量为Integer.MAX\_VALUE), 这会导致以下问题:**

**- 线程池里的线程数量不超过corePoolSize,这导致了maximumPoolSize和keepAliveTime将会是个无用参数**

**- 由于使用了无界队列, 所以FixedThreadPool永远不会拒绝, 即饱和策略失效**

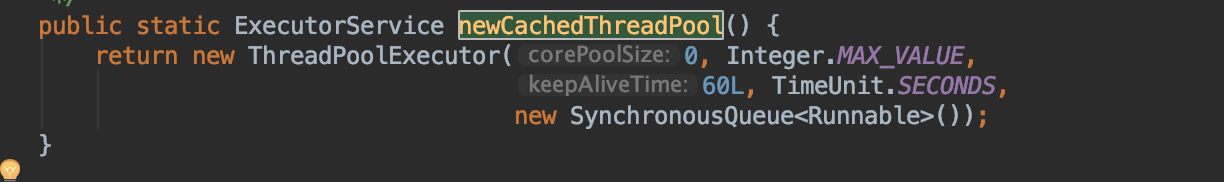
（2）**newSingleThreadExecutor**



初始化的线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行.

由于使用了无界队列, 所以SingleThreadPool永远不会拒绝, 即饱和策略失效

（3）**newCachedThreadPool**



线程池的线程数可达到Integer.MAX\_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；

和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；

执行过程与前两种稍微不同:

(1) 主线程调用SynchronousQueue的offer()方法放入task, 倘若此时线程池中有空闲的线程尝试读取 SynchronousQueue的task, 即调用了SynchronousQueue的poll(), 那么主线程将该task交给空闲线程. 否则执行(2)

(2) 当线程池为空或者没有空闲的线程, 则创建新的线程执行任务.

(3) 执行完任务的线程倘若在60s内仍空闲, 则会被终止. 因此长时间空闲的CachedThreadPool不会持有任何线程资源.

### 13.4 关闭线程池

**原理**

遍历线程池中的所有线程，然后逐个调用线程的interrupt方法来中断线程.

**关闭方式**

**（1）shutdown**

将线程池里的线程状态设置成SHUTDOWN状态, 然后中断所有没有正在执行任务的线程.

**（2）shutdownNow**

将线程池里的线程状态设置成STOP状态, 然后停止所有正在执行或暂停任务的线程.

只要调用这两个关闭方法中的任意一个, isShutDown() 返回true.

当所有任务都成功关闭了, isTerminated()返回true.