# .InnoDB支持缓存和数据，支持事物和外键，对内存大小，性能有影响，表空间比myIsAM要大。MyISAM偏向于读

# [Java虚拟机：类加载机制详解](http://www.cnblogs.com/fangfuhai/p/7230179.html)

大家知道，我们的Java程序被编译器编译成class文件，在class文件中描述的各种信息，最终都需要加载到虚拟机内存才能运行和使用，那么虚拟机是如何加载这些class文件的呢？在加载class文件的过程中虚拟机又干了哪些事呢？今天我们来解密虚拟机的类加载机制。

       虚拟机把class文件加载到内存，并对数据进行校验、解析和初始化，最终形成可以被虚拟机直接使用的Java类型（Class对象），这就是虚拟机的类加载机制。

       类从被加载到虚拟机内存开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载7个阶段。，其中验证、准备和解析3个阶段统称为连接阶段。如

  总结一下，加

载阶段的主要工作就是，把class二进制文件加载到内存后，将类中定义的静态变量、常量、类信息等数据存放到方法区，并在堆内存中创建一个代表这个类的Class对象，作为方法区中这个类的数据信息的访问入口，程序猿可以持有这个Class对象。

**双亲委派模型**

      前面提到过，在类加载的整个过程中，除了加载阶段我们可以通过自定义的类加载器参与之外，其他的阶段都是虚拟机帮我们完成的。虚拟机设计团队把加载这个动作放到Java虚拟机外部去实现，实现这个动作的代码模块称为“类加载器”。这样做的目的是让应用程序自己去决定如何获取所需要的类。

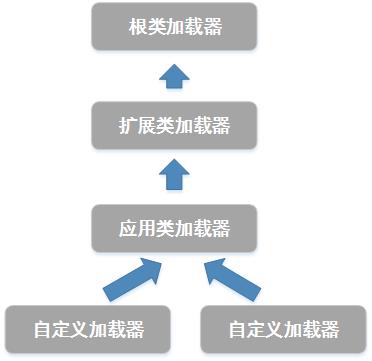
      除了我们自己可以定义类加载器，Java虚拟机也为我们提供了系统自带的类加载器。主要可以分为以下三种：

      根类加载器（Bootstrap ClassLoader）：这个类加载器负责加载存放在<JAVA\_HOME>\lib目录中的，或者通过参数-Xbootclasspath所指定的路径中的类。

      扩展类加载器（Extension ClassLoader）：这个加载器负责加载<JAVA\_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库。

      应用类加载器（Application ClassLoader）：它负责加载用户设置的ClassPath路径上所指定的类库。如果应用程序中没有自定义的类加载器，一般情况下这个就是程序默认的类加载器。

      我们的应用程序都是由这3种类加载器相互配合进行加载的，如果有必要，还可以定义自己的类加载器。这些类加载器之间的关系如下：



上图中展示的类加载器之间的层次关系，称为类加载器的双亲委派模型。双亲委派模型要求除了顶层的根类加载器外，其余的类加载器都应当有自己的父类加载器。双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的类加载请求最终都应该传送到根类加载器中，只有当父加载器反馈自己无法完成这个加载请求（搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

       类加载器虽然只用于实现类的加载动作，但是它在Java程序中起的作用却不仅仅是进行类加载。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立它在Java虚拟机中的唯一性。简单来说就是，一个类的class文件被不同的两个类加载器加载，那么加载后的这两个类就不“相等”，不是相同的类。

       使用双亲委派模型，有一个显而易见的好处就是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。例如java.lang.Object类，它存放在rt.jar中，无论哪一个类加载器要加载这个类，最终都会委派给模型最顶端的根类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类（始终被根类加载器加载）。相反，如果不使用双亲委派模型，由各个类加载器自己去加载的话，假如用户编写了一个称为java.lang.Object的类，并放在ClassPath中，那系统中会出现多个不同的Object类，应用程序也会变的一片混乱。

       以上内容总结了Java类加载机制的整个过程以及双亲委派模型的原理，欢迎交流。

内存回收机制算法：

内存回收就是释放掉在内存中已经没用的对象。

首先，要判断怎样的对象是没用的对象。这里有2种方法：

1.采用标记计数的方法：

给内存中的对象给打上标记，对象被引用一次，计数就加1，引用被释放了，计数就减一，当这个计数为0的时候，这个对象就可以被回收了。当然，这也就引发了一个问题：循环引用的对象是无法被识别出来并且被回收的。所以就有了第二种方法：

1. 根搜索算法

根搜索算法的基本思路就是通过一系列名为”GC Roots”的对象作为起始点，从这些节点开始向下搜索，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时，则证明此对象是不可用的。

内存回收就是释放掉在内存中已经没用的对象。

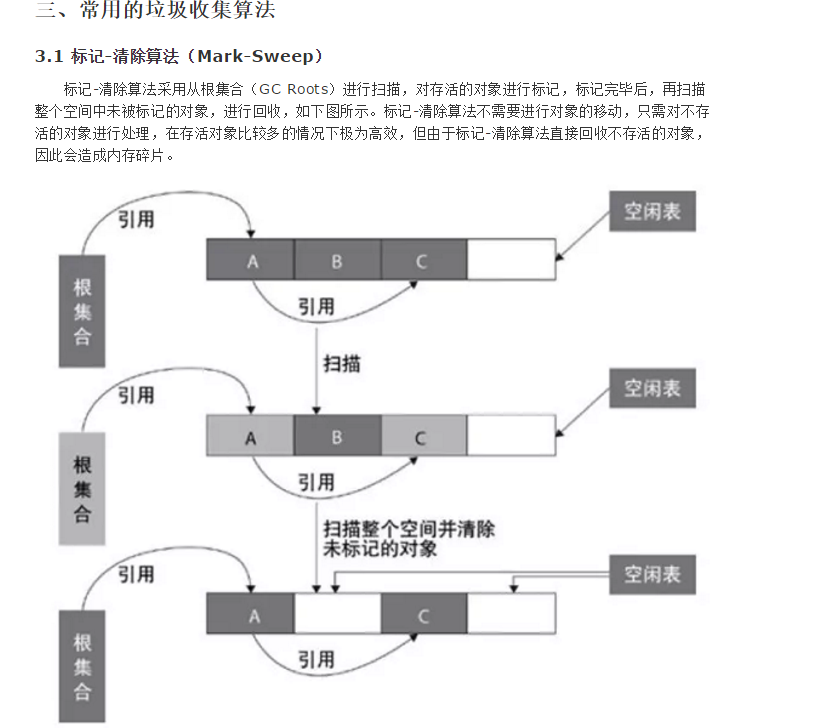
首先，要判断怎样的对象是没用的对象。这里有2种方法：

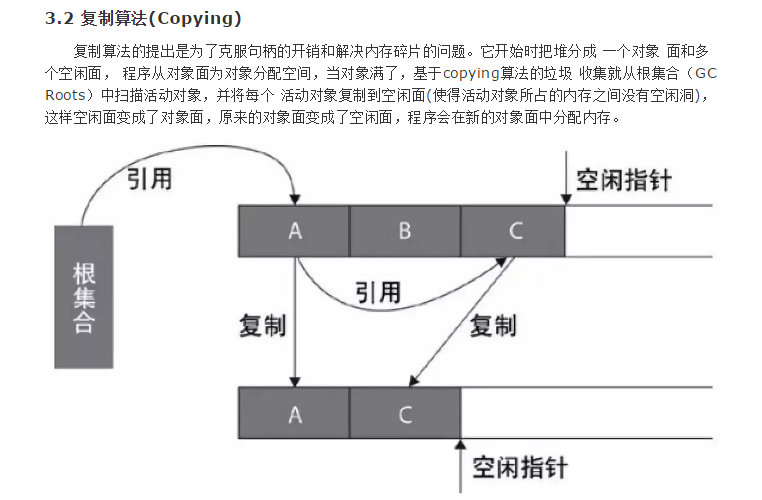
判断gcroots引用链（）和这个对象是否有关联 有关联就不回收 没有关联的就标记一次 再判断这个对象的final()方法是否使用过？这个方法

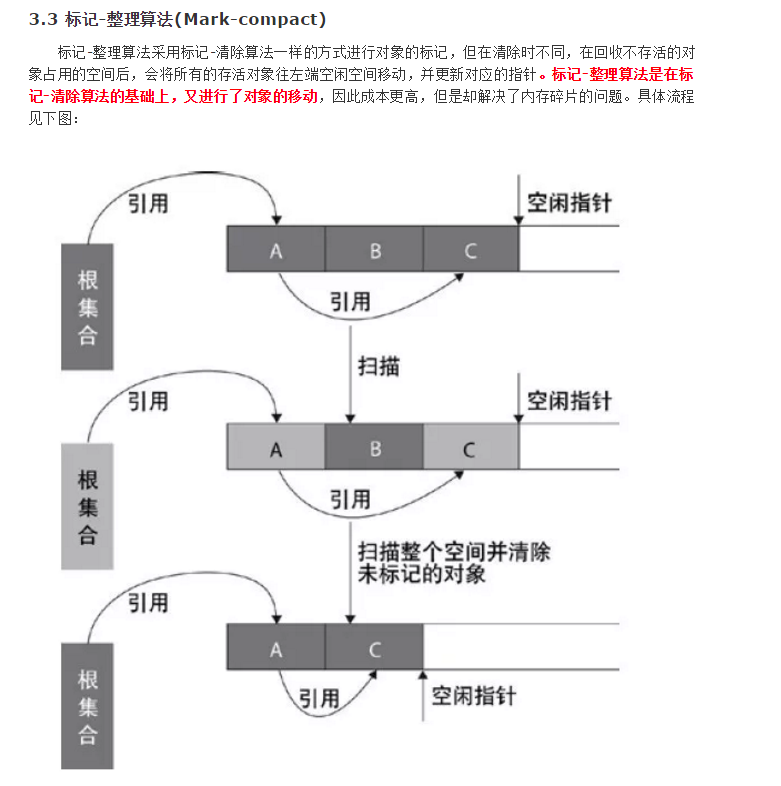
针对这个问题，虚拟机的做法是**进行两次标记**，即第一次标记不在“关系网”中的对象。**第二次的话就要先判断该对象有没有实现finalize()方法了**，如果没有实现就直接判断该对象可回收；如果实现了就会先放在一个队列中，并由虚拟机建立的一个低优先级的线程去执行它，随后就会进行第二次的小规模标记，**在这次被标记的对象就会真正的被回收了。**

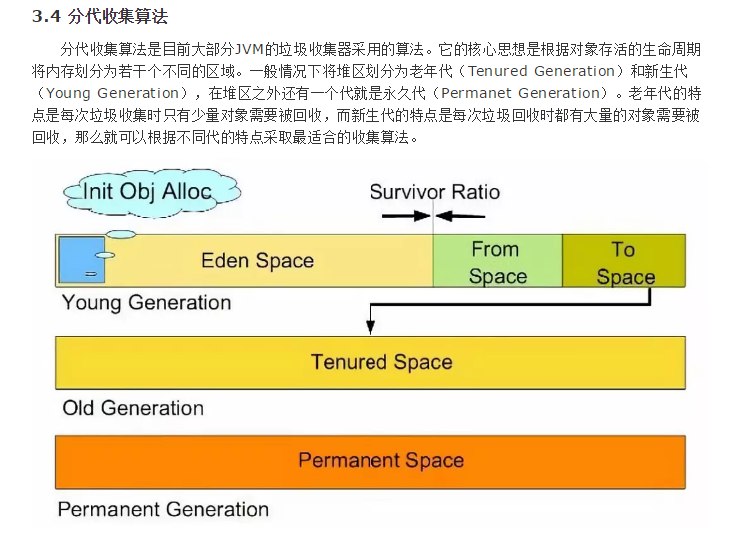
### java中垃圾收集的方法有哪些?

1. **标记-清除:**  
   这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是**标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会产生大量不连续的内存碎片，**导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作。
2. **复制算法:**  
   为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。  
   于是将该算法进行了改进，内存区域不再是按照1：1去划分，而是将内存划分为8:1:1三部分，较大那份内存交Eden区，其余是两块较小的内存区叫Survior区。每次都会优先使用Eden区，若Eden区满，就将对象复制到第二块内存区上，然后清除Eden区，如果此时存活的对象太多，以至于Survivor不够时，会将这些对象通过分配担保机制复制到老年代中。(java堆又分为新生代和老年代)
3. **标记-整理**  
   该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在**清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。**
4. **分代收集**   
   现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用**复制**算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用**标记-整理** 或者 **标记-清除**。





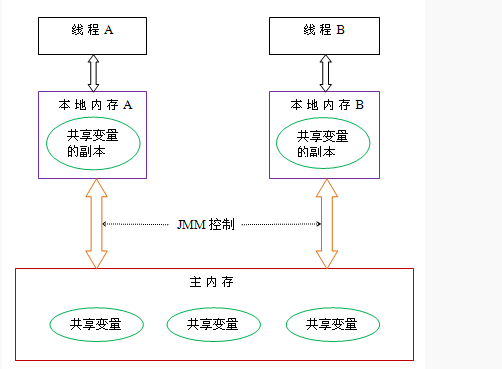




总结一下，加载阶段的主要工作就是，把class二进制文件加载到内存后，将类中定义的静态变量、常量、类信息等数据存放到方法区，并在堆内存中创建一个代表这个类的Class对象，作为方法区中这个类的数据信息的访问入口，程序猿可以持有这个Class对象。

### 5.java内存模型

java内存模型(JMM)是线程间通信的控制机制.JMM定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java内存模型的抽象示意图如下：



线程私有的：程序计数器，虚拟机栈，本地方法栈

线程共享的：堆，方法区

#### 2.1 程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时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、线程恢复等功能都需要依赖这个计数器来完。

**当前线程**所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。

注意：程序计数器是唯不会出现 OutOfMemoryError 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

#### 2.2栈分为java虚拟机栈和本地方法栈

重点是Java虚拟机栈，它是线程私有的，生命周期与线程相同。

**每个方法执行都会创建一个栈帧，用于存放局部变量表，操作栈，动态链接，方法出口等**。**每个方法从被调用，直到被执行完。对应着一个栈帧在虚拟机中从入栈到出栈的过程**。

通常说的栈就是指局部变量表部分，存放编译期间可知的8种基本数据类型，及对象引用和指令地址。局部变量表是在编译期间完成分配，当进入一个方法时，这个栈中的局部变量分配内存大小是确定的。

会有两种异常StackOverFlowError和 OutOfMemoneyError。当线程请求栈深度大于虚拟机所允许的深度就会抛出StackOverFlowError错误；虚拟机栈动态扩展，当扩展无法申请到足够的内存空间时候，抛出OutOfMemoneyError。

本地方法栈为虚拟机使用到本地方法服务（native）

#### 2.3堆区：

堆被所有线程共享区域，在虚拟机启动时创建，**唯一目的存放对象实例**。

堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。**更细一点年轻代又分为Eden区最要放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比 8:1:1**。

不过很多文章介绍分为3个区块，把**方法区算着为永久代**。这大概是基于Hotspot虚拟机划分，然后比如IBM j9就不存在永久代概论。不管怎么分区，都是存放对象实例。

会有异常OutOfMemoneyError

#### 2.4方法区：

被所有线程共享区域，用于存放已被虚拟机**加载的类信息，常量，静态变量**等数据。被Java虚拟机描述为**堆的一个逻辑部分**。习惯是也叫它永久代（permanment generation）

垃圾回收很少光顾这个区域，不过也是需要回收的，**主要针对常量池回收，类型卸载**。

**常量池**用于存放编译期生成的**各种字节码和符号引用**，常量池具有一定的动态性，**里面可以存放编译期生成的常量**；运行期间的常量也可以添加进入常量池中，比如string的intern()方法。

#### 2. 堆里面的分区：Eden，survivalfrom to，老年代，各自的特点。

1.JVM中堆空间可以分成三个大区，**新生代、老年代、永久代**。

2.新生代可以划分为三个区，Eden区，两个幸存区。

在JVM运行时，可以通过配置以下参数改变整个JVM堆的配置比例

1.JVM运行时堆的大小

-Xms堆的最小值 Xms 是指设定程序启动时**占用内存大小**。一般来讲，大点，程序会启动的快一点，但是也可能会导致机器暂时间变慢。

-Xmx堆空间的最大值 是指设定程序运行期间最大可占用的内存大小。如果程序运行需要占用更多的内存，超出了这个设置值，就会抛出OutOfMemory异常 **Xmx 为jvm运行过程中分配的最大内存，比如-Xms500m，表示jvm进程最多只能够占用500M内存**

2.新生代堆空间大小调整

　　-XX:NewSize新生代的最小值

　　-XX:MaxNewSize新生代的最大值

　　-XX:NewRatio设置新生代与老年代在堆空间的大小

　　-XX:SurvivorRatio新生代中Eden所占区域的大小

3.永久代大小调整

　-XX:MaxPermSize

4.其他

-XX:MaxTenuringThreshold,设置将新生代对象转到老年代时需要经过多少次垃圾回收，但是仍然没有被回收

