**JVM内存管理的机制**

内存空间划分为：SunJDK在实现时遵照JVM规范，将内存空间划分为**堆、JVM方法栈、方法区、本地方法栈、PC寄存器**。

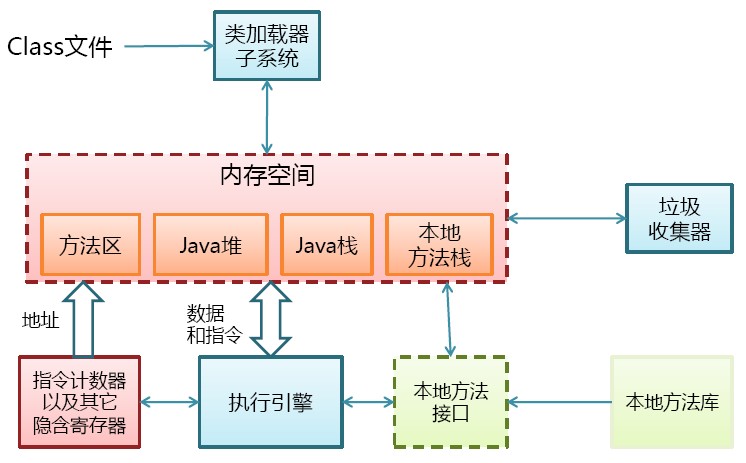
堆：堆用于存储对象实例及数组值，可以认为Java中所有通过new创建的对象的内存都在此分配，Heap中对象所占用的内存由GC进行回收，在32位操作系统上最大为2GB，在64位操作系统上则没有限制，其大小可通过-Xms和-Xmx来控制，-Xms为JVM启动时申请的最小Heap内存，默认为物理内存的1/64但小于1GB；-Xmx为JVM可申请的最大Heap内存，默认为物理内存的1/4但小于1GB，默认当空余堆内存小于40%时，JVM会增大Heap到-Xmx指定的大小，可通过-XX:MinHeapFreeRatio=来指定这个比例；当空余堆内存大于70%时，JVM会减小Heap的大小到-Xms指定的大小，可通过-XX:MaxHeapFreeRatio=来指定这个比例，对于运行系统而言，为避免在运行时频繁调整Heap的大小，通常将-Xms和-Xmx的值设成一样。

JVM方法栈（Java栈）：为线程私有，其在内存分配上非常高效。当方法运行完毕时，其对应的栈帧所占用的内存也会自动释放。当JVM方法栈空间不足时，会抛出StackOverflowError的错误，在SunJDK中可以通过-Xss来指定其大小。

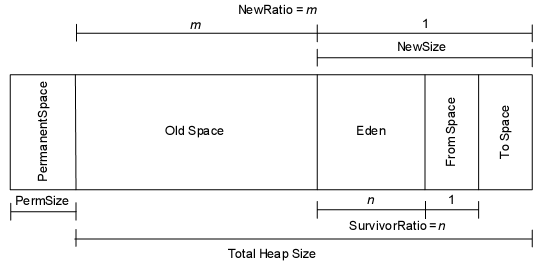
方法区：要加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的Field信息、类中的方法信息。方法区域也是全局共享的，在一定条件下它也会被GC，当方法区域要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。在SunJDK中这块区域对应PermanetGeneration，又称为持久代，默认最小值为16MB，最大值为64MB，可通过-XX:PermSize及-XX:MaxPermSize来指定最小值和最大值。

本地方法栈：用于支持native方法的执行，存储了每个native方法调用的状态。在SunJDK的实现中，和JVM方法栈是同一个。

PC寄存器：占用的可能为CPU寄存器或操作系统内存。



**JavaGC机制**



GC机制的基本算法是：分代收集。下面阐述每个分代的收集方法。

**年轻代：**

**年轻代（YoungGeneration）**：对象被创建时，内存的分配首先发生在年轻代（大对象可以直接被创建在年老代），大部分的对象在创建后很快就不再使用，因此很快变得**不可达**，于是被年轻代的GC机制清理掉（IBM的研究表明，98%的对象都是很快消亡的），这个GC机制被称为MinorGC或叫YoungGC。注意，MinorGC并不代表年轻代内存不足，它事实上只表示在Eden区上的GC。

　　年轻代上的内存分配是这样的，年轻代可以分为3个区域：**Eden区**（伊甸园，亚当和夏娃偷吃禁果生娃娃的地方，用来表示内存首次分配的区域，再贴切不过）和**两个存活区**（Survivor0、Survivor1）

1. 绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；
2. 最初一次，当**Eden区满**的时候，执行**MinorGC**，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；
3. 下次**Eden区满**了，再执行一次**MinorGC**，将存活的对象复制到Survivor1中，将消亡的对象清理掉，然后清空Eden区；
4. 将Survivor0中消亡的对象清理掉，将其中可以**晋级的对象晋级到Old区**，将存活的对象也复制到Survivor1区，然后清空Survivor0区；
5. **当两个存活区切换**了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代，但这只是个最大值，并不代表一定是这个值）之后，**仍然存活的对象**（其实只有一小部分，比如，我们自己定义的对象），**将被复制到老年代**。

从上面的过程可以看出，Eden区是连续的空间，且Survivor总有一个为空。经过一次GC和复制，一个Survivor中保存着当前还活着的对象，而Eden区和另一个Survivor区的内容都不再需要了，可以直接清空，到下一次GC时，两个Survivor的角色再互换。因此，这种方式分配内存和清理内存的效率都极高，这种垃圾回收的方式就是著名的“停止-复制（Stop-and-copy）”清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中），这不代表着停止复制清理法很高效，其实，它也只在这种情况下高效，如果在老年代采用停止复制，则挺悲剧的。

在Eden区，HotSpot虚拟机使用了两种技术来加快内存分配。分别是bump-the-pointer和TLAB（Thread-LocalAllocationBuffers），这两种技术的做法分别是：由于Eden区是连续的，因此bump-the-pointer技术的核心就是跟踪最后创建的一个对象，在对象创建时，只需要检查最后一个对象后面是否有足够的内存即可，从而大大加快内存分配速度；而对于TLAB技术是对于多线程而言的，将Eden区分为若干段，每个线程使用独立的一段，避免相互影响。TLAB结合bump-the-pointer技术，将保证每个线程都使用Eden区的一段，并快速的分配内存。

　　事实上，在新生代中，使用**“停止-复制”算法**进行清理，将新生代内存分为2部分，1部分Eden区较大，1部分Survivor比较小，并被划分为两个等量的部分。**每次进行清理时，将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中，而且将其中可以晋级的对象晋级到Old区，然后清理掉Eden和刚才的Survivor，**

　　这里也可以发现，停止复制算法中，用来复制的两部分并不总是相等的（传统的停止复制算法两部分内存相等，但新生代中使用1个大的Eden区和2个小的Survivor区来避免这个问题）

　　由于绝大部分的对象都是短命的，甚至存活不到Survivor中，所以，Eden区与Survivor的比例较大，HotSpot默认是8:1，即分别占新生代的80%，10%，10%。如果一次回收中，Survivor+Eden中存活下来的内存超过了10%，则需要将一部分对象分配到老年代。用-XX:SurvivorRatio参数来配置Eden区域Survivor区的容量比值，默认是8，代表Eden：Survivor1：Survivor2=8:1:1.

**老年代：**

**对象如果在年轻代存活了足够长的时间而没有被清理掉（即在几次YoungGC后存活了下来），则会被复制到年老代，**年老代的空间一般比年轻代大，能存放更多的对象，在**年老代上发生的GC次数也比年轻代少**。当年老代内存不足时，将执行**MajorGC，也叫FullGC。**

可以使用-XX:+UseAdaptiveSizePolicy开关来控制是否采用动态控制策略，如果动态控制，则动态调整Java堆中各个区域的大小以及进入老年代的年龄。

如果对象比较大（比如长字符串或大数组），Young空间不足，则**大对象会直接分配到老年代上**（大对象可能触发提前GC，应少用，更应避免使用短命的大对象）。用-XX:PretenureSizeThreshold来控制直接升入老年代的对象大小，大于这个值的对象会直接分配在老年代上。

可能存在年老代对象引用新生代对象的情况，如果需要执行YoungGC，则可能需要查询整个老年代以确定是否可以清理回收，这显然是低效的。解决的方法是，**年老代中维护一个512byte的块——”cardtable“**，**所有老年代对象引用新生代对象的记录都记录在这里**。YoungGC时，只要查这里即可，不用再去查全部老年代，因此性能大大提高。

　　老年代存储的对象比年轻代多得多，而且不乏大对象，对老年代进行内存清理时，如果使用停止-复制算法，则相当低效。一般，老年代用的算法是**标记-整理算法，即：标记出仍然存活的对象（存在引用的），将所有存活的对象向一端移动，以保证内存的连续。**

**在发生MinorGC时，虚拟机会检查每次晋升进入老年代的大小是否大于老年代的剩余空间大小，如果大于，则直接触发一次FullGC，否则**，就查看是否设置了-XX:+HandlePromotionFailure（允许担保失败），如果允许，则只会进行MinorGC，此时可以容忍内存分配失败；如果不允许，则仍然进行FullGC（这代表着如果设置-XX:+HandlePromotionFailure，则触发MinorGC就会同时触发FullGC，哪怕老年代还有很多内存，所以，最好不要这样做）。

**方法区（永久代）：**

　　永久代的回收有两种：**常量池中的常量**，**无用的类信息**，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的所有实例都已经被回收
2. 加载类的ClassLoader已经被回收
3. 类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

永久代的回收并不是必须的，可以通过参数来设置是否对类进行回收。HotSpot提供-Xnoclassgc进行控制

使用-verbose，-XX:+TraceClassLoading、-XX:+TraceClassUnLoading可以查看类加载和卸载信息

-verbose、-XX:+TraceClassLoading可以在Product版HotSpot中使用；

-XX:+TraceClassUnLoading需要fastdebug版HotSpot支持

**SSM框架原理,作用及使用方法**

**作用**：

SSM框架是springMVC，spring和mybatis框架的整合，是标准的MVC模式，将整个系统划分为表现层，controller层，service层，DAO层四层

使用springMVC负责请求的转发和视图管理

spring实现业务对象管理，mybatis作为数据对象的持久化引擎

**原理**：

SpringMVC：

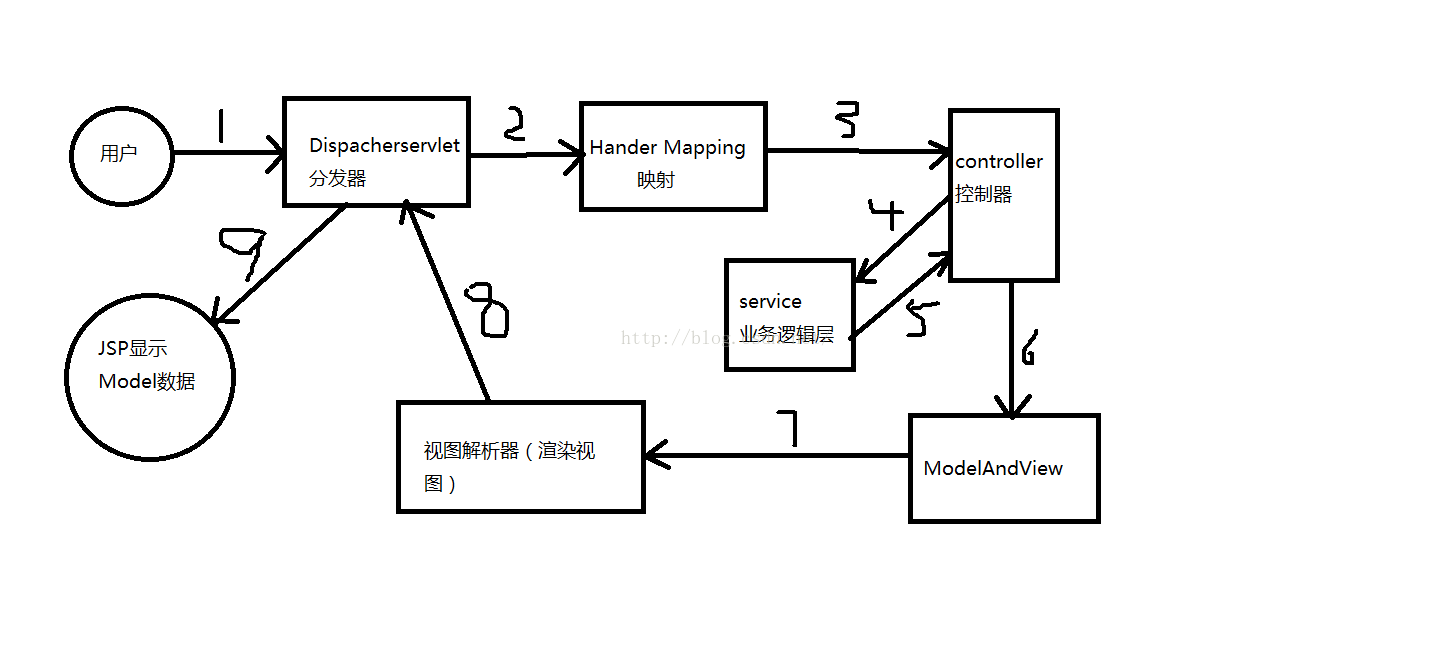
1.客户端发送请求到DispacherServlet（分发器）

2.由DispacherServlet控制器查询HanderMapping，找到处理请求的Controller

3.Controller调用业务逻辑处理后，返回ModelAndView

4.DispacherSerclet查询视图解析器，找到ModelAndView指定的视图

5.视图负责将结果显示到客户端



Spring：我们平时开发接触最多的估计就是IOC容器，它可以装载bean（也就是我们[Java](http://lib.csdn.net/base/java" \t "_blank" \o "Java 知识库)中的类，当然也包括servicedao里面的），有了这个机制，我们就不用在每次使用这个类的时候为它初始化，很少看到关键字new。另外spring的aop，事务管理等等都是我们经常用到的。

Mybatis：mybatis是对jdbc的封装，它让数据库底层操作变的透明。mybatis的操作都是围绕一个sqlSessionFactory实例展开的。mybatis通过配置文件关联到各实体类的Mapper文件，Mapper文件中配置了每个类对数据库所需进行的sql语句映射。在每次与数据库交互时，通过sqlSessionFactory拿到一个sqlSession，再执行sql命令。

**使用方法：**

要完成一个功能：

先写实体类entity，定义对象的属性，（可以参照数据库中表的字段来设置，数据库的设计应该在所有编码开始之前）。

写Mapper.xml（Mybatis），其中定义你的功能，对应要对数据库进行的那些操作，比如insert、selectAll、selectByKey、delete、update等。

写Mapper.java，将Mapper.xml中的操作按照id映射成Java函数。

写Service.java，为控制层提供服务，接受控制层的参数，完成相应的功能，并返回给控制层。

写Controller.java，连接页面请求和服务层，获取页面请求的参数，通过自动装配，映射不同的URL到相应的处理函数，并获取参数，对参数进行处理，之后传给服务层。

写JSP页面调用，请求哪些参数，需要获取什么数据。

DataBase===>Entity===>Mapper.xml===>Mapper.**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "_blank)**===>Service.java===>Controller.java===>Jsp.

====================================================================================================================================

SpringMVC拥有控制器，作用跟Struts类似，接收外部请求，解析参数传给服务层  
Spring容器属于协调上下文，管理对象间的依赖，提供事务机制  
mybatis属于orm持久层框架，将业务实体与数据表联合起来

SpringMVC控制层，想当与Struts的作用  
Spring控制反转和依赖注入创建对象交由容器管理，达到了解耦的作用  
mybatis主要用来操作数据库（数据库的增删改查）

IOC:控制反转，是一种降低对象之间耦合关系的设计思想，面试的时候最好能说出来个例子，加深理解。例子：租房子，以前租房子需要一个房子一个房子找，费时费力，然后现在加入一个房屋中介，把你需要的房型告诉中介，就可以直接选到需要的房子，中介就相当于spring容器。

AOP:面向切面编程，是面向对象开发的一种补充，它允许开发人员在不改变原来模型的基础上动态的修改模型以满足新的需求，如：动态的增加日志、安全或异常处理等。AOP使业务逻辑各部分间的耦合度降低，提高程序可重用性，提高开发效率。

持久层：DAO层（mapper）

* DAO层：DAO层主要是做数据持久层的工作，负责与数据库进行联络的一些任务都封装在此，  
  + DAO层的设计首先是设计DAO的接口，
  + 然后在Spring的配置文件中定义此接口的实现类，
  + 然后就可在模块中调用此接口来进行数据业务的处理，而不用关心此接口的具体实现类是哪个类，显得结构非常清晰，
  + DAO层的数据源配置，以及有关数据库连接的参数都在Spring的配置文件中进行配置。

业务层：Service层

* Service层：Service层主要负责业务模块的逻辑应用设计。
  + 首先设计接口，再设计其实现的类
  + 接着再在Spring的配置文件中配置其实现的关联。这样我们就可以在应用中调用Service接口来进行业务处理。
  + Service层的业务实现，具体要调用到已定义的DAO层的接口，
  + 封装Service层的业务逻辑有利于通用的业务逻辑的独立性和重复利用性，程序显得非常简洁。

表现层：Controller层（Handler层）

* Controller层:Controller层负责具体的**业务模块流程的控制**，
  + 在此层里面要调用Service层的接口来控制业务流程，
  + 控制的配置也同样是在Spring的配置文件里面进行，针对具体的业务流程，会有不同的控制器，我们具体的设计过程中可以将流程进行抽象归纳，设计出可以重复利用的子单元流程模块，这样不仅使程序结构变得清晰，也大大减少了代码量。

View层

* View层此层与控制层结合比较紧密，需要二者结合起来协同工发。View层主要负责前台jsp页面的表示.

各层联系

* DAO层，Service层这两个层次都可以单独开发，互相的耦合度很低，完全可以独立进行，这样的一种模式在开发大项目的过程中尤其有优势
* Controller，View层因为耦合度比较高，因而要结合在一起开发，但是也可以看作一个整体独立于前两个层进行开发。这样，在层与层之前我们只需要知道接口的定义，调用接口即可完成所需要的逻辑单元应用，一切显得非常清晰简单。
* Service逻辑层设计
  + Service层是建立在DAO层之上的，建立了DAO层后才可以建立Service层，而Service层又是在Controller层之下的，因而Service层应该既调用DAO层的接口，又要提供接口给Controller层的类来进行调用，它刚好处于一个中间层的位置。每个模型都有一个Service接口，每个接口分别封装各自的业务处理方法。

**双层for循环的跳出**

利用标记跳出循环

1. **loop:**for(Mapmap:listMap){
2. for(Iteratorit=map.keySet().iterator();it.hasNext();){
3. Stringkey=(String)it.next();
4. Objectvalue=map.get(key);
6. System.out.println("第"+count+"个Map的键值对是：Key="+key+"，Value="+value.toString());
8. if(key.equals("4")){
9. System.out.println("跳出循环");
10. **break loop;**
11. }
12. }
13. }

将循环数做下变动

1. for(inti=0;i<listMap.size();i++;){
2. Map<String,Object>map=listMap.get(i);
3. for(Iteratorit=map.keySet().iterator();it.hasNext();){
4. Stringkey=(String)it.next();
5. Objectvalue=map.get(key);
7. System.out.println("第"+count+"个Map的键值对是：Key="+key+"，Value="+value.toString());
9. if(key.equals("4")){
10. System.out.println("跳出循环");
11. **i=listMap.size();**
12. }
13. }
14. }

**HashMap和Hashtable都实现了Map接口，主要的区别有：**

同步方面：Hashtable的方法是同步的（synchronized），HashMap未经同步，所以在多线程场合要手动同步HashMap。HashMap是非线程安全的。

赋值方面：Hashtable不允许null值(key和value都不可以)，HashMap允许null值(key和value都可以)。

遍历方式不同：Hashtable比HashMap多一个elements方法。

动态数组增加方式不同：Hashtable中hash数组默认大小是11，增加的方式是old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。

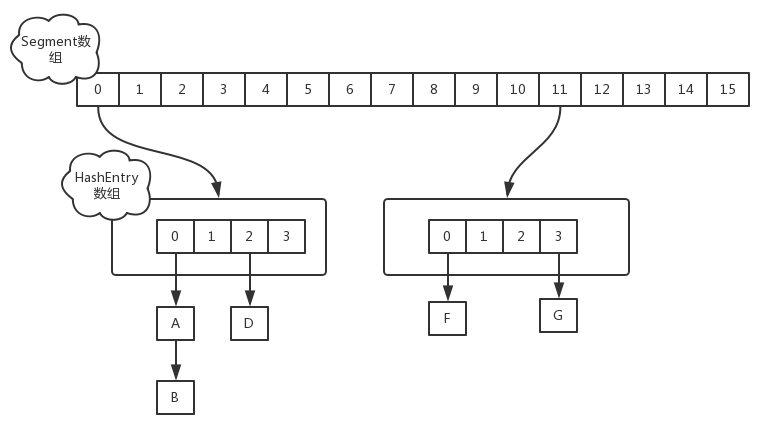
**HashMap的底层实现**

（1.7之前（包括1.7）用的是位桶+链表的方式，）HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。

（1.8之后（包括1.8）采用的是位桶+链表/红黑树）当同一个hash值的节点数不小于8时，将不再以单链表的形式存储了，会被调整成一颗红黑树。（红黑树，一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是Red或Black。通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出俩倍，因而是接近平衡的。）

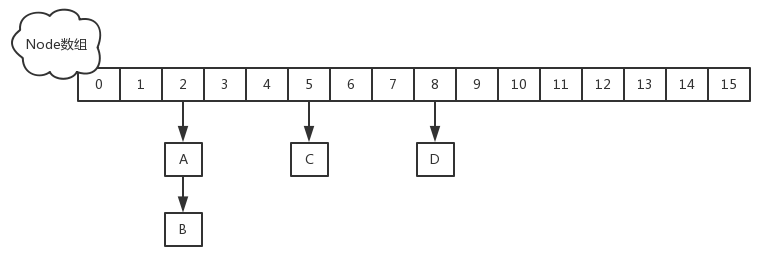
**ConcurrentHashMap**

在JDK1.7版本中，ConcurrentHashMap的数据结构是由一个Segment数组和多个HashEntry组成，如下图所示：



Segment数组的意义就是将一个大的table分割成多个小的table来进行加锁，也就是上面的提到的锁分离技术，而每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样

JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作，整个看起来就像是优化过且线程安全的HashMap。



**hashCode方法**

对于包含容器类型的程序设计语言来说，基本上都会涉及到hashCode。在Java中也一样，**hashCode方法的主要作用是为了配合基于散列的集合一起正常运行，这样的散列集合包括HashSet、HashMap以及HashTable。**

　　为什么这么说呢？考虑一种情况，当向集合中插入对象时，如何判别在集合中是否已经存在该对象了？（注意：集合中不允许重复的元素存在）

　　也许大多数人都会想到调用equals方法来逐个进行比较，这个方法确实可行。但是如果集合中已经存在一万条数据或者更多的数据，如果采用equals方法去逐一比较，效率必然是一个问题。此时hashCode方法的作用就体现出来了，**当集合要添加新的对象时，先调用这个对象的hashCode方法，得到对应的hashcode值，实际上在HashMap的具体实现中会用一个table保存已经存进去的对象的hashcode值，如果table中没有该hashcode值，它就可以直接存进去，不用再进行任何比较了；如果存在该hashcode值，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址，**所以这里存在一个冲突解决的问题，这样一来实际调用equals方法的次数就大大降低了，说通俗一点：Java中的hashCode方法就是根据一定的规则将与对象相关的信息（比如对象的存储地址，对象的字段等）映射成一个数值，这个数值称作为散列值。

在有些情况下，程序设计者在设计一个类的时候为需要重写equals方法，比如String类，但是千万要注意，**在重写equals方法的同时，必须重写hashCode方法。默认情况下，hashCode方法是将对象的存储地址进行映射**

在程序执行期间，只要equals方法的比较操作用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法必须始终如一地返回同一个整数。

如果两个对象根据equals方法比较是相等的，那么调用两个对象的hashCode方法必须返回相同的整数结果。

如果两个对象根据equals方法比较是不等的，则hashCode方法不一定得返回不同的整数。

hashCode是用于查找使用的，而equals是用于比较两个对象的是否相等的。

**Arraylist与linkedlist的区别**

都是实现list接口的列表，arraylist是基于数组的数据结构，linkedlist是基于链表的数据结构。

当获取特定元素时，ArrayList效率比较快，它通过数组下标即可获取，而linkedlist则需要移动指针。

当存储元素与删除元素时linkedlist效率较快，只需要将指针移动指定位置增加或者删除即可，而arraylist需要移动数据。

**Struts2和springmvc的本质区别:**

1.springmvc入口是一个servlet前端控制器(DispatcherServlet),struts2入口是一filter过滤器(StrutsPrepareAndExecuteFilter)

2.struts2通过在action类中定义成员变量接收参数,(属性驱动和模型驱动),它只能使用多例模式管理action.

springmvc通过在coontroller方法中定义形参接收参数,springmvc可以使用单例模式管理controller.

3.springmvc是基于方法开发的,注解开发中使用requestMapping将url和方法进行映射,如果根据url找到controller类的方法生成一个handler处理器对象(只包括一个method).

struts2是基于类开发的,每个请求过来创建一个action实例,实例对象中有若干个方法.

开发中建议使用springmvc,springmvc方法更类似service业务方法.

4.struts2采用值栈存储请求和相应的数据,通过OGNL存取数据,springmvc通过参数绑定将request请求内容解析,并给方法形参赋值.

5.struts2和springmvc的速度是相当的,由于struts2的漏洞较多,很多企业使用springmvc

1.核心控制器（前端控制器、预处理控制器）：对于使用过mvc框架的人来说这个词应该不会陌生，核心控制器的主要用途是处理所有的请求，然后对那些特殊的请求（控制器）统一的进行处理(字符编码、文件上传、参数接受、异常处理等等),springmvc核心控制器是Servlet(前端控制器DispatcherServlet)，而Struts2是Filter(过滤器StrutsPrepareAndExecuteFilter)。

2.控制器实例：SpringMvc会比Struts快一些（理论上）。SpringMvc是基于方法设计，而Sturts是基于对象，每次发一次请求都会实例一个action，每个action都会被注入属性，而Spring更像Servlet一样，只有一个实例，每次请求执行对应的方法即可(注意：由于是单例实例，所以应当避免全局变量的修改，这样会产生线程安全问题)。

3.管理方式：大部分的公司的核心架构中，就会使用到spring,而springmvc又是spring中的一个模块，所以spring对于springmvc的控制器管理更加简单方便，而且提供了全注解方式进行管理，各种功能的注解都比较全面，使用简单，而struts2需要采用XML很多的配置参数来管理（虽然也可以采用注解，但是几乎没有公司那样使用）。

4.参数传递：Struts2中自身提供多种参数接受，其实都是通过（ValueStack）进行传递和赋值，而SpringMvc是通过方法的参数进行接收。

5.学习难度：Struts更加很多新的技术点，比如拦截器、值栈及OGNL表达式，学习成本较高，springmvc比较简单，很较少的时间都能上手。

6.intercepter的实现机制：struts有以自己的interceptor机制，springmvc用的是独立的AOP方式。这样导致struts的配置文件量还是比springmvc大，虽然struts的配置能继承，所以我觉得论使用上来讲，springmvc使用更加简洁，开发效率SpringMVC确实比struts2高。springmvc是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应，所以说从架构本身上spring3mvc就容易实现restfulurl。struts2是类级别的拦截，一个类对应一个request上下文；实现restfulurl要费劲，因为struts2action的一个方法可以对应一个url；而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。spring3mvc的方法之间基本上独立的，独享requestresponse数据，请求数据通过参数获取，处理结果通过ModelMap交回给框架方法之间不共享变量，而struts2搞的就比较乱，虽然方法之间也是独立的，但其所有Action变量是共享的，这不会影响程序运行，却给我们编码，读程序时带来麻烦。

7.springmvc处理ajax请求,直接通过返回数据，方法中使用注解@ResponseBody，springmvc自动帮我们对象转换为JSON数据。