无界队列会导致线程积压，消耗内存。

newFixedThreadPool:

无界队列

newCacheThreadPool、

newSingelThreadPool、

只有一个线程

newScheduledThreadPool

new ThreadPoolExcutor

线程池的提交优先级、执行优先级

提交优先级:核心线程>队列>非核心线程

执行优先级:核心线程>非核心线程>队列

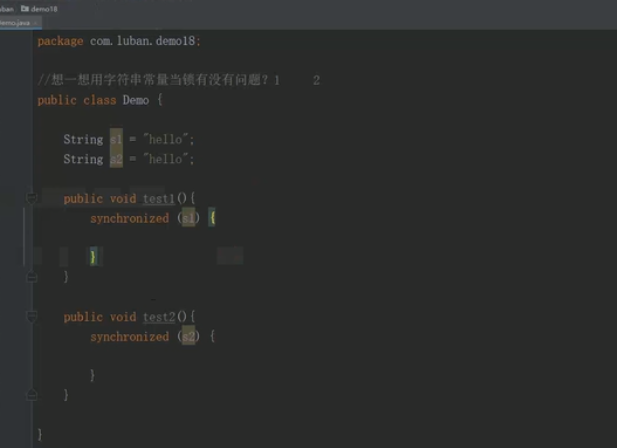
JMM模型(java线程内存模型)

可见性：线程之间的可见性，一个线程修改的状态对另一个线程是可见的，即一个线程修改的结果，另外一个线程马上就能看到。Java中volatile、synchronized 和 final 具有可见性。

原子性：

有序性：

不要用常量做同步,因为可能第三方包用了相同的常量做同步，这样会导致一直获取不到锁：



图中，s1和s2指向的是同一个地址，所以test1和test2方法其实是获取的同一个对象的锁。

wait和notify、notifyAll

都是Object基础类的方法，而非Thread类的方法。当需要调用以上方法时，需要对竞争资源加锁(即需要将这些方法放到synchronized(obj)中)。

wait会释放锁,但是notify后不会立即释放锁，直到执行完synchronized。

线程优先级：

线程优先级有10个等级，分别用整数1-10表示。其中1位最低优先级，10为最高优先级，5为默认值。但是，线程的优先级与代码执行顺序之间并无关联，只不过,优先级高的线程获取CPU资源的概率较大，优先级低的并非没机会执行。线程的优先级具有继承性，比如A线程启动B线程，则A和B的线程优先级是一样的。

sleep()与wait():

sleep()是Thread的静态方法。

被调用后，都会暂停当前线程并让出cpu的执行时间，但是sleep不会释放当前持有的对象锁资源，到时间后会继续执行(不是立即执行，只是达到可执行的状态，需要获取到cpu资源后方可执行)；而wait被调用后立即释放锁资源，需要调用对象的notify()/notifyAll()方法唤醒.

volatile()：

Java之八大基础数据类型

byte 8位 -128~127

short 16位 -216-1~216-1-1

int 32位 -232-1~232-1-1

long 64位 -264-1~264-1-1

float 32位

double 64位

char 16位

boolean true或false

类型转换：

char--> 自动转换：byte-->short-->int-->long-->float-->double

**equals和==区别：**

1. equals是方法，==是运算符;
2. 如果使用==比较基础数据类型，则是比较两者数值是否相等，如果比较的是引用数据类型，则比较两者地址是否相等;equals是用来比较两个对象的内容是否一致;
3. 如果equals和==都用来比较对象时，且两个对象的引用地址一致时，equals可能返回true或者false，这要取决于是否重写了对象的equals方法已经重写的内容，而==一定返回true。

**重写equals方法时，为什么要必须重写hashCode()?**

hashCode()方法是用来获取对象的哈希码值，这个值是用来确定该对象再哈希表中索引所在的位置。

equals()是用来判断两个对象是否相等，如果对象没有重写equals方法，即比较两个对象的地址是否相同，等价于”==”。

先看看代码的例子,有两个Person类，Person2只重写了equals()，Person1重写了equals()和hashCode():





Main():



运行结果：



根据上图的结果想要说明的是:

①如果只重写equals方法，虽然equals的结果是true，但是如果把对象放到HashSet或者其他自动去重的集合中，因为没有重写hashCode，所以此时两个对象的hashcode值其实是不一样的，HashSet会把他们当作两个不同的对象，不会去重。所以，重写equals时必须重写hashCode,是指在需要用到集合(HashSet)去重对象时，必须重写hashCode().

②如果两个对象equals相等，那么他们的hashCode值一定相等。

③两个对象如果hashCode值一样，他们也不一定相等，需要调用equals来比较。

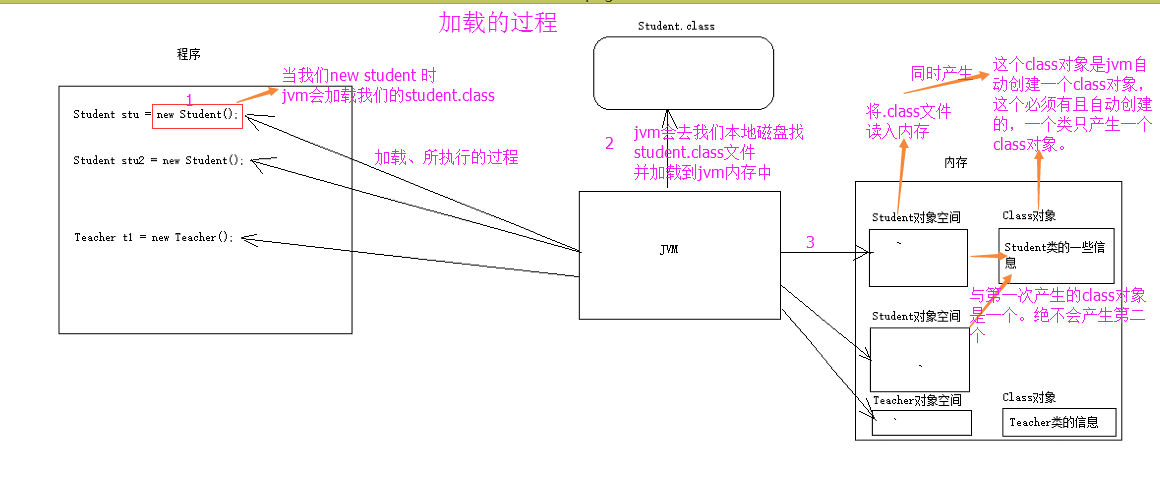
④hashCode()默认是在堆栈上的对象产生独特的哈希值。如果没有重写hashCode()，那么两个对象无论如何都不会相等。

**Java中类的加载过程**

.java文件在编译成.class文件之后生命周期会经历以下7个阶段，前面5个阶段是类的加载过程：

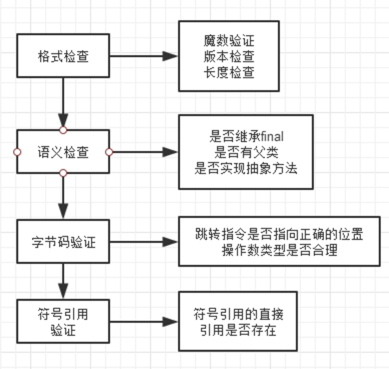
1. 加载

由类的加载器根据类的全限定名读取此类的二进制字节流到jvm中，并存储在运行时方法区内，然后将其转换为对应的class对象，该对象描述了这个类创建的对象的所有信息，比如成员变量、构造方法、成员方法等。



2、验证

该阶段只要是确保.class文件的字节流满足java虚拟机规范，不会造成安全错误。



3、准备

为类中的所有静态变量分配内存，并为其设置一个默认初始值，如果是被final修饰的静态变量，则直接赋予原值。

1. 解析

将二进制字节流中的符号引用转为直接引用。

说明：

符号引用，即一个字符串，但是这个字符串给出了一些能够唯一性标识的一个方法或者一个变量或者是一个类的相关信息。

直接引用，可以理解为一个内存地址，或者一个偏移量。比如类方法、类变量的直接引用就是指向方法区的指针；而实例方法，实例变量的直接引用则是从实例的头指针开始算起到这个实例变量位置的偏移量。

举个例子来说，现在调用方法hello(),这个方法的地址是0xaabbccdd，那么hello()就是符号引用，0xaabbccdd就是直接引用。

1. 初始化

为标记为常量的属性赋值的过程。即对static修饰的变量或者代码块进行初始化，如果初始化的时候，其父类未初始化，则优先初始化其父类。

6、使用、卸载

初始化完成后，就可以对该类使用了，使用完之后，在方法区垃圾回收的过程中进行卸载(垃圾回收);

**Java中类的加载器**

类加载器的任务就是根据类的全限定名来读取此类的二进制字节流到JVM中，然后转换成一个与此类对应的java.lang.Class对象实例，这个对象实例用来表示一个Java类。

所有的类加载器都继承了抽象类java.lang.ClassLoader。

Java中的类加载器大致可以分成2类，一类是系统提供的，另一类是自建的。系统提供的主要有三个：

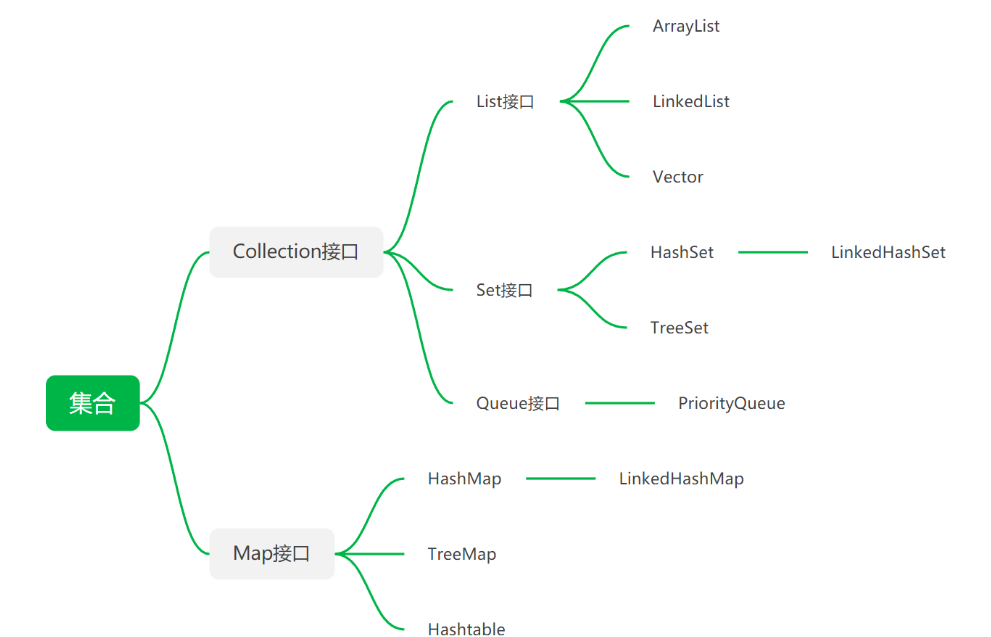
BootstrapClassLoader:用来加载java的核心库，他不继承java.lang.ClassLoader;

**Java中类的加载机制**

.java

**Java中双亲委派模型**

**集合**



|  |  |  |
| --- | --- | --- |
| ArrayList  (List) | ArrayList是基于动态数组实现的;  ArrayList的默认初始化容量是10，每次扩容时候增加原先容量的一半，也就是变为原来的1.5倍;  删除元素时不会减少容量，若希望减少容量则调用trimToSize();  随机查找速度快，插入删除速度慢（插入删除均需移动元素）;  非线程安全（需要同步时可使用Vector类）;  能存放null值;  如果想做成线程安全，可以用如下方式：  List<Object> arryList = Collections.*synchronizedList*(new ArrayList<>()); | +add();+add(index, obj);  +addAll();+addAll(index, obj);  +size();+toArray();转数组  +clear();+remove();+removeAll();  +get();+contains();  +set(index, obj);替换位置的元素  ... |
| LinkedList  (List) | 一种可以在任何位置进行高效地插入和删除操作的有序序列;  随机查找速度慢，插入删除速度快;  底层实现是链表（双向链表）; | +add();+add(index, obj);  +addAll();+addAll(index, obj);  +addFirst();+addLast();  +size();+toArray();转数组  +clear();+remove();  +removeAll();  +get();+contains();  +set(index, obj);替换位置的元素  ... |
| Vector  (List) | Vector底层也是数组，与ArrayList最大的区别就是：同步(线程安全)；  ArrayList在底层数组不够用时在原来的基础上扩展0.5倍，Vector是扩展1倍；  Vector的默认初始化容量是10；  能存放null值; | 方法跟ArrayList差不多 |
| HashSet  (Set) | 元素是无序不可重复的,可以存在null;  非线程安全;  像HashSet中插入对象时，HashSet调用该对象的hashCode()方法来得到该对象的hashCode值，然后根据 hashCode值来决定该对象在HashSet中存储位置。修改了元素，元素的hashCode值也会改变。它的实现是依赖HashMap。 |  |
| LinkedHashSet  (Set) | 有序(先进先出)，不可重复，可以存在null,非线程安全，是HashSet的子类，根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起来像是以插入顺序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。  LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，因为它是根据链表维护元素的次序的，但是插入时性能稍微逊色于HashSet(百万数据量时才能体现出)。它的实现依赖于LinkedHashMap |  |
| TreeSet  (Set) | 元素是有序不可重复的,不能存在null;  非线程安全;  依赖TreeMap实现;  支持两种排序方式，自然排序和自定义排序。  自然排序:实现Comparable接口，重写了compareTo()来进行排序。  自定义排序:  实现Comparable接口或者Comparator接口，重写compareTo()方法。  如下图 | 注意：1、TreeSet只能添加相同类型的元素。  Set set = new TreeSet();  set.add(1);  set.add("2");  System.out.println(set);  这种是错误的；  2、获取TreeSet的元素时，元素的类型必须时实现了Comparable接口或者Comparator接口 |
| PriorityQueue  (Queue，优先队列) | 参考：  <https://www.cnblogs.com/wei-jing/p/10806236.html>  https://blog.csdn.net/u010623927/article/details/87179364  PriorityQueue是基于优先堆的一个无界队列，这个优先队列中的元素可以默认自然排序或者通过提供的Comparator（比较器）在队列实例化的时排序。  不允许null;不支持不可比较的对象(未实现Comparable或Comparator接口的对象);非线程安全;默认长度是11，利用二叉小顶锥实现(任意一个非叶子节点的权值，都不大于其左右子节点的权值），也就意味着可以通过数组来作为PriorityQueue的底层实现)。 | +peek()//返回队首元素  +poll()//返回队首元素，队首元素出队列  +add()//添加元素  +size()//返回队列元素个数  +isEmpty()//判断队列是否为空，为空返回true,不空返回false |
| PriorityBlockingQueue | 线程安全; |  |

TreeSet-自然排序：





执行结果：



TreeSet-自定义排序：







执行结果：



|  |  |  |
| --- | --- | --- |
| HashMap | key不能重复，但是value可以重复;  非线程安全;无序;key和value都允许为空; | 1.8以前是数组+链表  18及以后是数组+链表+红黑树 |
| TreeMap | 对键有序的遍历 | key 必须实现 Comparable 接口或者在构造 TreeMap 传入自定义的Comparator |
| LinkedHashMap | 按照插入的顺序存储;  非线程安全 |  |
| Hashtable | Key、value都不能为空  线程安全,实现线程安全的方式是在修改数据时锁住整个哈希表，效率低，ConcurrentHashMap做了相关优化;  Hashtable的初始size为11，扩容：newsize = olesize\*2+1。 | 数组+链表 |
| ConcurrentHashMap | 线程安全;，比Hashtable效率高(采用的是分段锁) |  |

**守护线程：**

为用户线程提供公共服务，在没有用户线程可服务时会自动离开，守护线程的优先级比较低。用线程对象的setDaemon(true)将线程设置为守护线程。在守护线程中产生的新线程也是守护线程。

**锁：**

乐观锁：

Hadoop(分布式存储、分布式计算)

Common

Fdhs 分布式文件系统

存储并管理PB级数据



Yarn

MapReduce