#### String和StringBuffer、StringBuilder的区别

可变性

String类中使用字符数组保存字符串，private

final char value[]，所以string对象是不可变的。StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，char[]

value，这两种对象都是可变的。

线程安全性

String中的对象是不可变的，也就可以理解为常量，线程安全。StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder并没有对方法进行加同步锁，所以是非线程安全的。

性能

StringBuffer相比StringBuilder加了锁，所以性能比StringBuilder差

#### 运行时数据区

根据 JVM 规范，JVM 内存共分为虚拟机栈、堆、方法区、程序计数器、本地方法栈五个部分。

运行时常量池在方法区中

虚拟机栈（JVM Stack）：一个线程的每个方法在执行的同时，都会创建一个栈帧（Statck Frame），栈帧中存储的有局部变量表、操作站、动态链接、方法出口等，当方法被调用时，栈帧在JVM栈中入栈，当方法执行完成时，栈帧出栈。局部变量表中存储着方法的相关局部变量，包括各种基本数据类型，对象的引用，返回地址等。在局部变量表中，只有long和double类型会占 用2个局部变量空间（Slot，对于32位机器，一个Slot就是32个bit），其它都是1个Slot。需要注意的是，局部变量表是在编译时就已经确定 好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变。虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出StatckOverFlowError（栈溢出）；不过多 数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，知道内存不足，此时，会抛出 OutOfMemoryError（内存溢出）。每个线程对应着一个虚拟机栈，因此虚拟机栈也是线程私有的。

堆区（Heap）：堆区是理解Java GC机制最重要的区域，没有之一。在JVM所管理的内存中，堆区是最大的一块，堆区也是Java GC机制所管理的主要内存区域，堆区由所有线程共享，在虚拟机启动时创建。堆区的存在是为了存储对象实例，原则上讲，所有的对象都在堆区上分配内存（不过现代技术里，也不是这么绝对的，也有栈上直接分配的）。堆内存需要在逻辑上是连续的（在物理上不需要），堆内存不够时将会抛出OutOfMemoryError:Java heap space异常。

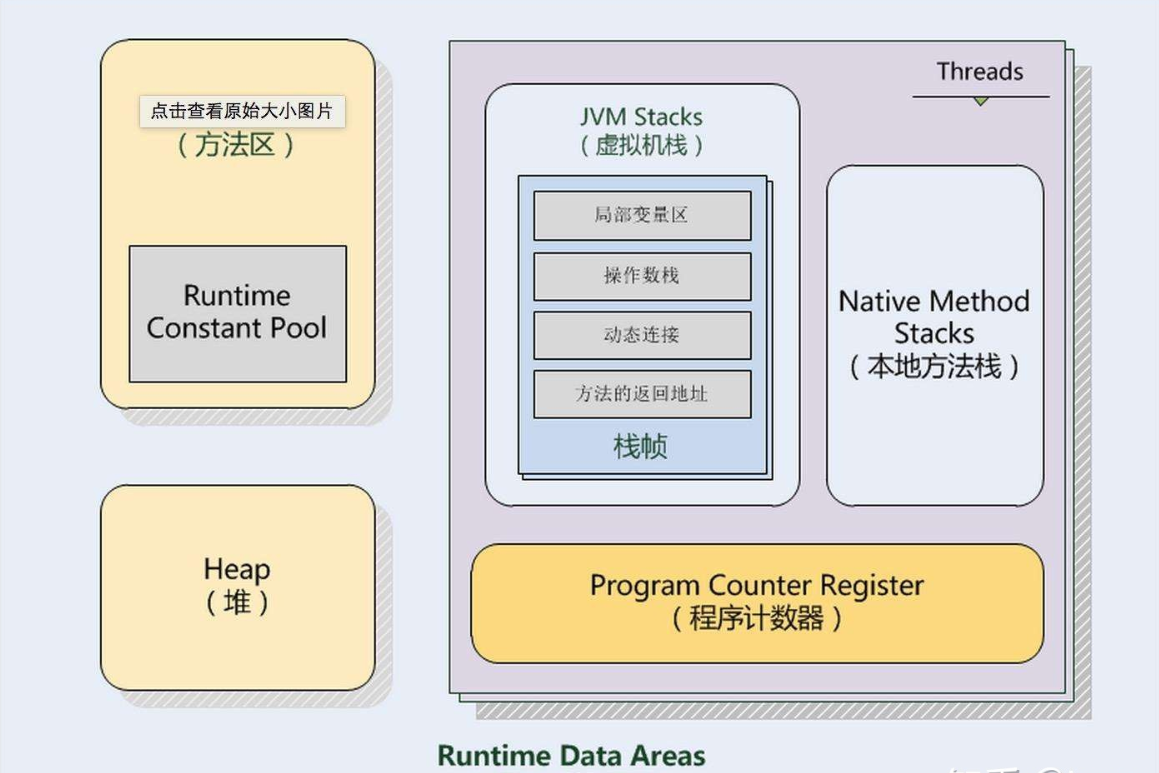
方法区（Method Area）：在Java虚拟机规范中，将方法区作为堆的一个逻辑部分来对待，但事实 上，方法区并不是堆（Non-Heap）；方法区是各个线程共享的区域，用于存储已经被虚拟机加载的类信息（即加载类时需要加载的信息，包括版本、field、方法、接口等信息）、final常量、静态变量、编译器即时编译的代码等。方法区在物理上也不需要是连续的，可以选择固定大小或可扩展大小，并且方法区比堆还多了一个限制：可以选择是否执行垃圾收集。一般的，方法区上 执行的垃圾收集是很少的，这也是方法区被称为永久代的原因之一（HotSpot），但这也不代表着在方法区上完全没有垃圾收集，其上的垃圾收集主要是针对 常量池的内存回收和对已加载类的卸载。在方法区上进行垃圾收集，条件苛刻而且相当困难，效果也不令人满意，所以一般不做太多考虑，可以留作以后进一步深入研究时使用。在方法区上定义了OutOfMemoryError:PermGen space异常，在内存不足时抛出。

运行时常量池（Runtime Constant Pool）是方法区的一部分，用于存储编译期就生成的字面常量、符号引用、翻译出来的直接引用（符号引用就是编码是用字符串表示某个变量、接口的位置，直接引用就是根据符号引用翻译出来的地址，将在类链接阶段完成翻译）；运行时常量池除了存储编译期常量外，也可以存储在运行时间产生的常量（比如String类的intern()方法，作用是String维护了一个常量池，如果调用的字符“abc”已经在常量池中，则返回池中的字符串地址，否则，新建一个常量加入池中，并返回地址）。

程序计数器（Program Counter Register）：程序计数器是一个比较小的内存区域，用于指示当前线程所执行的字节码执行到了第几行，可以理解为是当前线程的行号指示器。字节码解释器在工作时，会通过改变这个计数器的值来取下一条语句指令。每个程序计数器只用来记录一个线程的行号，所以它是线程私有（一个线程就有一个程序计数器）的。如果程序执行的是一个Java方法，则计数器记录的是正在执行的虚拟机字节码指令地址；如果正在执行的是一个本地（native，由C语言编写 完成）方法，则计数器的值为Undefined，由于程序计数器只是记录当前指令地址，所以不存在内存溢出的情况，因此，程序计数器也是所有JVM内存区 域中唯一一个没有定义OutOfMemoryError的区域。

本地方法栈（Native Method Statck）：本地方法栈在作用，运行机制，异常类型等方面都与虚拟机栈相同，唯一的区别是：虚拟机栈是执行Java方法的，而本地方法栈是用来执行native方法的，在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将本地方法栈与虚拟机栈放在一起使用。本地方法栈也是线程私有的。

直接内存（Direct Memory）：直接内存并不是JVM管理的内存，可以这样理解，直接内存，就是 JVM以外的机器内存，比如，你有4G的内存，JVM占用了1G，则其余的3G就是直接内存，JDK中有一种基于通道（Channel）和缓冲区 （Buffer）的内存分配方式，将由C语言实现的native函数库分配在直接内存中，用存储在JVM堆中的DirectByteBuffer来引用。 由于直接内存收到本机器内存的限制，所以也可能出现OutOfMemoryError的异常。



#### HashMap和HashTable区别

1）.HashTable的方法前面都有synchronized来同步，是线程安全的；HashMap未经同步，是非线程安全的。

2）.HashTable不允许null值(key和value都不可以) ；HashMap允许null值(key和value都可以)。

3）.HashTable有一个contains(Object

value)功能和containsValue(Object

value)功能一样。

4）.HashTable使用Enumeration进行遍历；HashMap使用Iterator进行遍历。

5）.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。

6）.哈希值的使用不同，HashTable直接使用对象的hashCode； HashMap重新计算hash值，而且用与代替求模。

#### ArrayList和vector区别

ArrayList和Vector都实现了List接口，都是通过数组实现的。

Vector是线程安全的，而ArrayList是非线程安全的。

List第一次创建的时候，会有一个初始大小，随着不断向List中增加元素，当List 认为容量不够的时候就会进行扩容。Vector缺省情况下自动增长原来一倍的数组长度，ArrayList增长原来的50%。

#### ArrayList和LinkedList区别及使用场景

区别

ArrayList底层是用数组实现的，可以认为ArrayList是一个可改变大小的数组。随着越来越多的元素被添加到ArrayList中，其规模是动态增加的。

LinkedList底层是通过双向链表实现的， LinkedList和ArrayList相比，增删的速度较快。但是查询和修改值的速度较慢。同时，LinkedList还实现了Queue接口，所以他还提供了offer(),

peek(), poll()等方法。

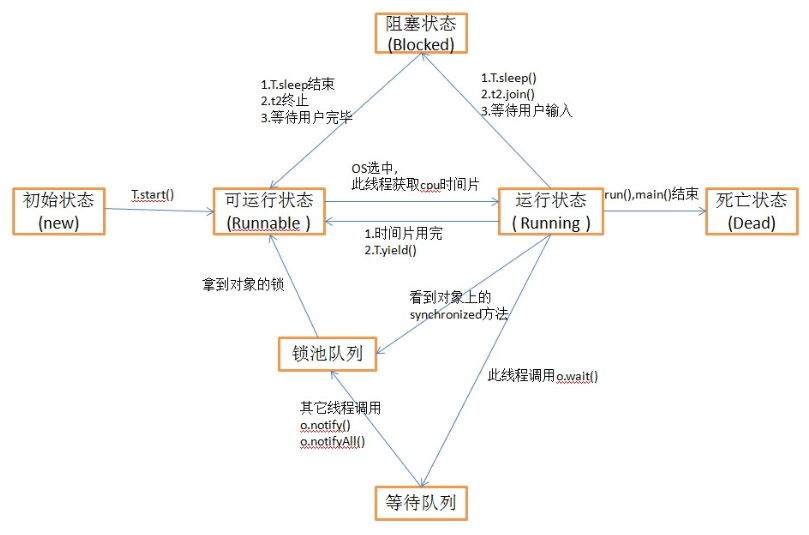
使用场景

LinkedList更适合从中间插入或者删除（链表的特性）。

ArrayList更适合检索和在末尾插入或删除（数组的特性）。

#### 线程的状态转换

线程一共有如图所示的5种状态



这里记录一下等待队列的状态切换

调用obj的wait(), notify()方法前，必须获得obj锁，也就是必须写在synchronized(obj) 代码段内。

与等待队列相关的步骤和图

线程1获取对象A的锁，正在使用对象A。

线程1调用对象A的wait()方法。

线程1释放对象A的锁，并马上进入等待队列。

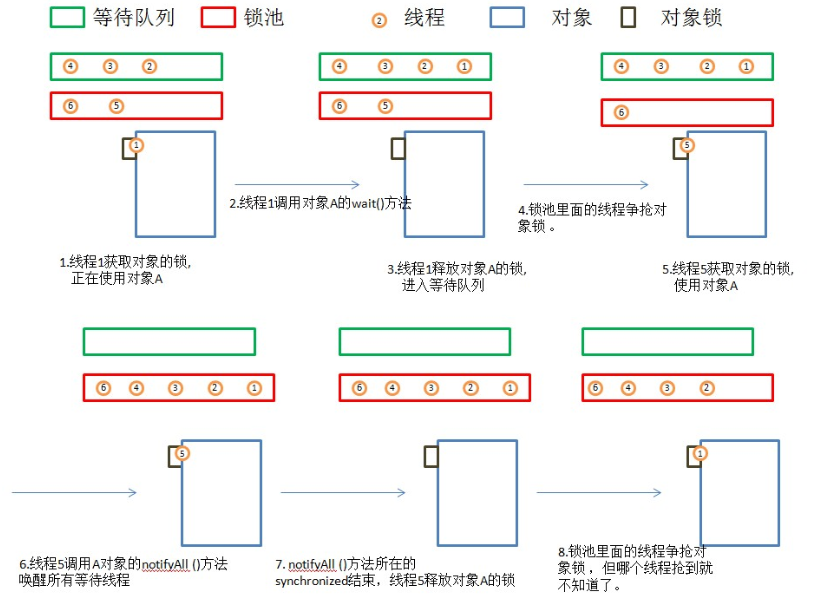
锁池里面的对象争抢对象A的锁。

线程5获得对象A的锁，这时候进入的是可运行状态，被系统分配了时间片，才进入运行状态，进入synchronized块，使用对象A。

线程5调用对象A的notifyAll()方法，唤醒所有线程，所有线程进入锁池。||||| 线程5调用对象A的notify()方法，唤醒一个线程，不知道会唤醒谁，被唤醒的那个线程进入锁池。

notifyAll()方法所在的synchronized结束，线程5释放对象A的锁。

锁池里面的线程争抢对象锁，但线程1什么时候能抢到就不知道了。||||| 原本锁池+第6步被唤醒的线程一起争抢对象锁。



#### sleep和wait的区别(考察的方向是是否会释放锁)

sleep()方法是Thread类中方法，而wait()方法是Object类中的方法。

sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态，在调用sleep()方法的过程中，线程不会释放对象锁。而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备。

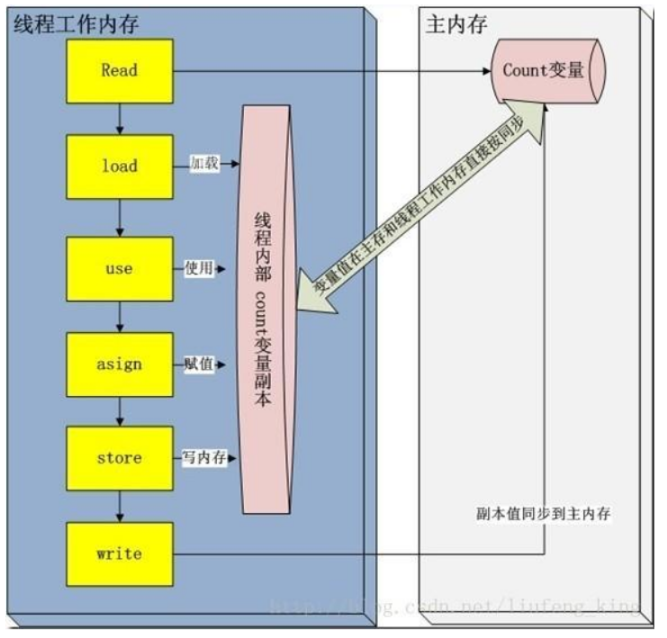
#### volatile关键字的理解

1. 阻止指令重排序

指令重排序可能导致异常，比如双检锁单例中如果没有保证指令顺序，有可能导致引用了空的对象，抛出空指针异常

1. volatile无法保证原子性

每一个线程运行时都有一个线程栈，线程栈保存了线程运行时候变量值信息。当线程访问某一个对象时候值的时候，首先通过对象的引用找到对应在堆内存的变量的值，然后把堆内存变量的具体值load到线程本地内存中，建立一个变量副本，之后线程就不再和对象在堆内存变量值有任何关系，而是直接修改副本变量的值，在修改完之后的某一个时刻（线程退出之前），自动把线程变量副本的值回写到对象在堆中变量。这样在堆中的对象的值就产生变化了。由于volatile不具有原子性，所以存在两个线程都将volatile对象存入线程副本变量，然后重复写到主内存，导致最终的结果并不正确。



read and load 从主存复制变量到当前工作内存

use and assign 执行代码，改变共享变量值

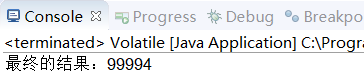
store and write 用工作内存数据刷新主存相关内容

其中use and

assign 可以多次出现，但是这一些操作并不是原子性，也就是在read load之后，如果主内存count变量发生修改之后，线程工作内存中的值由于已经加载，不会产生对应的变化，所以计算出来的结果会和预期不一样。

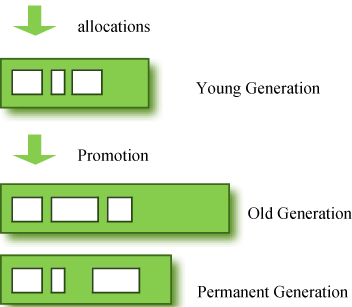
相对应的代码示例，由于无法保证原子性，最终结果和预期并不相同





#### Java内存分配机制

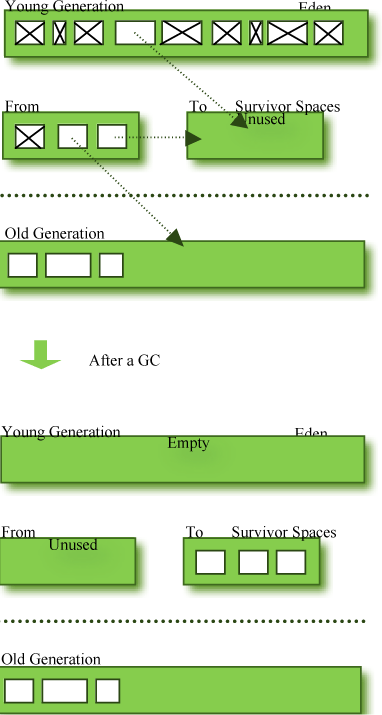
jvm内存分为：年轻代，年老代，永久代（Permanent Generation，也就是方法区）



年轻代（Young Generation）：一个Eden区，两个Survivor区。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.

年轻代的gc叫Minor GC或叫Young GC。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；当Eden区满的时候，执行Minor GC，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）；此后，每次Eden区满了，就执行一次Minor GC，并将剩余的对象都添加到Survivor0；当Survivor0也满的时候，将其中仍然活着的对象直接复制到Survivor1，以后Eden区执行Minor GC后，就将剩余的对象添加Survivor1（此时，Survivor0是空白的）。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）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。年轻代使用的是“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）

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- Local Allocation Buffers），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。



年老代（Old Generation）：年老代分配两类对象，1.多次young gc还存活的对象2.大对象不分配在年轻代，直接分配在年老代。年老代的gc称为Major GC，也叫 Full GC。

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Young GC，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512 byte的块——”card table“，所有老年代对象引用新生代对象的记录都记录在这里。Young GC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。

在发生Minor GC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Full GC，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Full GC（这代表着如果设置-XX:+Handle PromotionFailure，则触发MinorGC就会同时触发Full GC，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）的内存回收

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