1. **Java内存模型的理解（JMM）**

答案：<https://www.cnblogs.com/kaleidoscope/p/9505829.html>

Java内存模型即Java Memory Model，简称JMM。JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。

Java内存模型定义了多线程之间共享变量的可见性以及如何在需要的时候对共享变量进行同步。原始的Java内存模型效率并不是很理想，因此Java1.5版本对其进行了重构，现在的Java8仍沿用了Java1.5的版本。

在并发编程领域，有两个关键问题：线程之间的通信和同步。

线程的通信

是指线程之间以何种机制来交换信息。在命令式编程中，线程之间的通信机制有两种**共享内存**和**消息传递**。

在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。

在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是wait()和notify()。

线程之间的同步

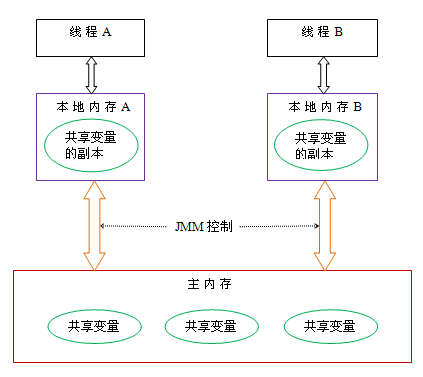
同步是指程序用于控制不同线程之间操作发生相对顺序的机制。

在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。

在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

**Java的并发采用的是共享内存模型**

JMM决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：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。



从图来看，线程A与线程B之间如要通信的话，

必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享

变量刷新到主内存中去。

1. 然后，线程B到主内存中去读取线程A之前

已更新过的共享变量。



如上图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更 新后的x值，此时线程B的本地内存的x值也变为了1。

从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。

### JVM对Java内存模型的实现

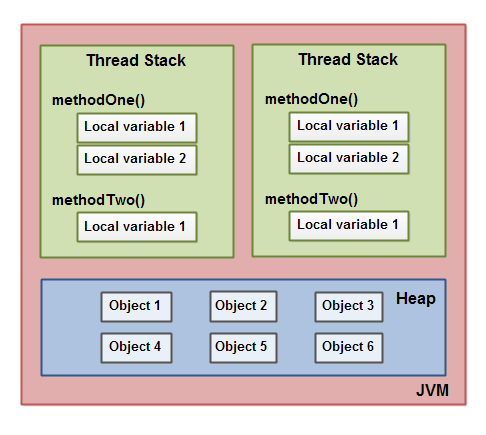
JVM中运行的每个线程都拥有自己的线程栈，线程栈包含了当前线程执行的方法调用相关信息，我们也把它称作调用栈。随着代码的不断执行，调用栈会不断变化。

线程栈还包含了当前方法的所有本地变量信息。一个线程只能读取自己的线程栈，也就是说，线程中的本地变量对其它线程是不可见的。即使两个线程执行的是同一段代码，它们也会各自在自己的线程栈中创建本地变量，因此，每个线程中的本地变量都会有自己的版本。

所有原始类型(boolean,byte,short,char,int,long,float,double)的**本地变量**都直接保存在线程栈当中，对于它们的值各个线程之间都是独立的。对于原始类型的本地变量，一个线程可以传递一个副本给另一个线程，当它们之间是无法共享的。

堆区包含了Java应用创建的所有对象信息，不管对象是哪个线程创建的，其中的对象包括原始类型的封装类（如Byte、Integer、Long等等）。不管对象是属于一个成员变量还是方法中的本地变量，它都会被存储在堆区。

下图展示了调用栈和本地变量都存储在栈区，对象都存储在堆区：



一个本地变量如果是原始类型，那么它会被完全存储到栈区。   
一个本地变量也有可能是一个对象的引用，这种情况下，这个本地引用会被存储到栈中，

但是对象本身仍然存储在堆区。

对于一个对象的成员方法，这些方法中包含本地变量，仍需要存储在栈区，即使它们所属的对象在堆区。   
 对于一个对象的成员变量，不管它是原始类型还是包装类型，都会被存储到堆区。

Static类型的变量以及类本身相关信息都会随着类本身存储在堆区。

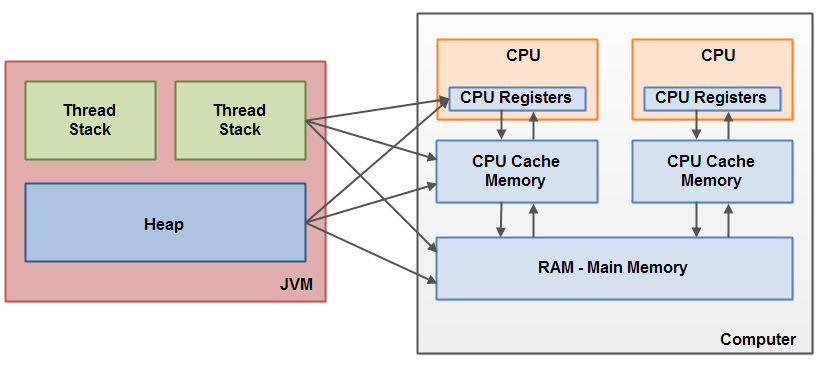
堆中的对象可以被多线程共享。如果一个线程获得一个对象的应用，它便可访问这个对象的成员变量。如果两个线程同时调用了同一个对象的同一个方法，那么这两个线程便可同时访问这个对象的成员变量，但是对于本地变量，每个线程都会拷贝一份到自己的线程栈中。



现代计算机一般都有2个以上CPU，而且每个CPU还有可能包含多个核心。因此，如果我们的应用是多线程的话，这些线程可能会在各个CPU核心中并行运行。

在CPU内部有一组CPU寄存器，也就是CPU的储存器。CPU操作寄存器的速度要比操作计算机主存快的多。在主存和CPU寄存器之间还存在一个CPU缓存，CPU操作CPU缓存的速度快于主存但慢于CPU寄存器。某些CPU可能有多个缓存层（一级缓存和二级缓存）。计算机的主存也称作RAM，所有的CPU都能够访问主存，而且主存比上面提到的缓存和寄存器大很多。

当一个CPU需要访问主存时，会先读取一部分主存数据到CPU缓存，进而在读取CPU缓存到寄存器。当CPU需要写数据到主存时，同样会先flush寄存器到CPU缓存，然后再在某些节点把缓存数据flush到主存。

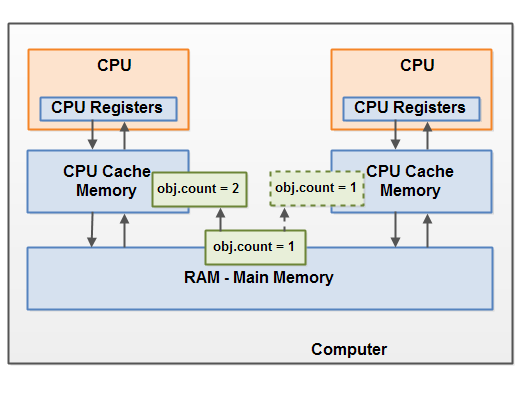
正如上面讲到的，Java内存模型和硬件内存架构并不一致。硬件内存架构中并没有区分栈和堆，从硬件上看，不管是栈还是堆，大部分数据都会存到主存中，当然一部分栈和堆的数据也有可能会存到CPU寄存器中，如下图所示，Java内存模型和计算机硬件内存架构是一个交叉关系：   


当对象和变量存储到计算机的各个内存区域时(计算机主内存和cpu缓存、cpu寄存器)，必然会面临一些问题，其中最主要的两个问题是：

**1. 共享对象对各个线程的可见性  
2. 共享对象的竞争现象**

当多个线程同时操作同一个共享对象时，如果没有合理的使用volatile和synchronization关键字，一个线程对共享对象的更新有可能导致其它线程不可见。

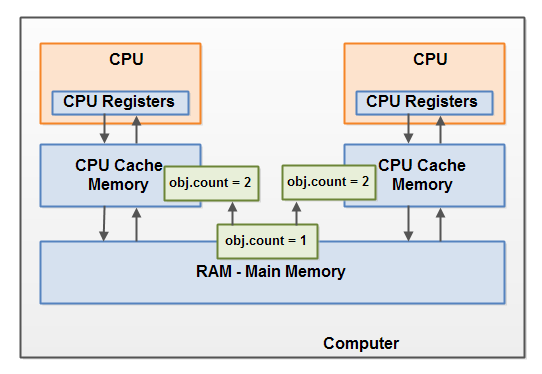
想象一下我们的共享对象存储在主存，一个CPU中的线程读取主存数据到CPU缓存，然后对共享对象做了更改，但CPU缓存中的更改后的对象还没有flush到主存，此时线程对共享对象的更改对其它CPU中的线程是不可见的。最终就是每个线程最终都会拷贝共享对象，而且拷贝的对象位于不同的CPU缓存中。

下图展示了上面描述的过程。左边CPU中运行的线程从主存中拷贝共享对象obj到它的CPU缓存，把对象obj的count变量改为2。但这个变更对运行在右边CPU中的线程不可见，因为这个更改还没有flush到主存中：   
   
 要解决共享对象可见性这个问题，我们可以使用java volatile关键字。 Java’s volatile keyword. volatile 关键字可以保证变量会直接从主存读取，而对变量的更新也会直接写到主存。volatile原理是基于CPU内存屏障指令实现的，后面会讲到。

#### 竞争现象

如果多个线程共享一个对象，如果它们同时修改这个共享对象，这就产生了竞争现象。

如下图所示，线程A和线程B共享一个对象obj。假设线程A从主存读取Obj.count变量到自己的CPU缓存，同时，线程B也读取了Obj.count变量到它的CPU缓存，并且这两个线程都对Obj.count做了加1操作。此时，Obj.count加1操作被执行了两次，不过都在不同的CPU缓存中。

如果这两个加1操作是串行执行的，那么Obj.count变量便会在原始值上加2，最终主存中的Obj.count的值会是3。然而下图中两个加1操作是并行的，不管是线程A还是线程B先flush计算结果到主存，最终主存中的Obj.count只会增加1次变成2，尽管一共有两次加1操作。   


要解决上面的问题我们可以使用java synchronized代码块。synchronized代码块可以保证同一个时刻只能有一个线程进入代码竞争区，synchronized代码块也能保证代码块中所有变量都将会从主存中读，当线程退出代码块时，对所有变量的更新将会flush到主存，不管这些变量是不是volatile类型的。

## 支撑Java内存模型的基础原理

### 指令重排序

在执行程序时，为了提高性能，编译器和处理器会对指令做重排序。但是，JMM确保在不同的编译器和不同的处理器平台之上，通过插入特定类型的Memory Barrier来禁止特定类型的编译器重排序和处理器重排序，为上层提供一致的内存可见性保证。

1、编译器优化重排序：编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

2、指令级并行的重排序：如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3、内存系统的重排序：处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

### 数据依赖性

如果两个操作访问同一个变量，其中一个为写操作，此时这两个操作之间存在数据依赖性。 编译器和处理器不会改变存在数据依赖性关系的两个操作的执行顺序，即不会重排序。

### ****as-if-serial****

不管怎么重排序，单线程下的执行结果不能被改变，编译器、runtime和处理器都必须遵守as-if-serial语义。

**内存屏障（Memory Barrier ）**

上面讲到了，通过内存屏障可以禁止特定类型处理器的重排序，从而让程序按我们预想的流程去执行。内存屏障，又称内存栅栏，是一个CPU指令，基本上它是一条这样的指令：

1、保证特定操作的执行顺序。

2、影响某些数据（或则是某条指令的执行结果）的内存可见性。

编译器和CPU能够重排序指令，保证最终相同的结果，尝试优化性能。插入一条Memory Barrier会告诉编译器和CPU：不管什么指令都不能和这条Memory Barrier指令重排序。

Memory Barrier所做的另外一件事是强制刷出各种CPU cache，如一个Write-Barrier（写入屏障）将刷出所有在Barrier之前写入 cache 的数据，因此，任何CPU上的线程都能读取到这些数据的最新版本。

这和java有什么关系？上面java内存模型中讲到的**volatile是基于Memory Barrier实现的。**

如果一个变量是volatile修饰的，JMM会在写入这个字段之后插进一个Write-Barrier指令，并在读这个字段之前插入一个Read-Barrier指令。这意味着，如果写入一个volatile变量，就可以保证：

1、一个线程写入变量a后，任何线程访问该变量都会拿到最新值。

2、在写入变量a之前的写入操作，其更新的数据对于其他线程也是可见的。因为Memory Barrier会刷出cache中的所有先前的写入。

**2 volatile、synchronized关键字的区别**

synchronized关键字解决的是执行控制的问题，它会阻止其它线程获取当前对象的监控锁，这样就使得当前对象中被synchronized关键字保护的代码块无法被其它线程访问，也就无法并发执行。更重要的是，synchronized还会创建一个内存屏障，内存屏障指令保证了所有CPU操作结果都会直接刷到主存中，从而保证了操作的内存可见性，同时也使得先获得这个锁的线程的所有操作，都happens-before于随后获得这个锁的线程的操作。

volatile关键字解决的是内存可见性的问题，会使得所有对volatile变量的读写都会直接刷到主存，即保证了变量的可见性。这样就能满足一些对变量可见性有要求而对读取顺序没有要求的需求

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

**3、垃圾回收机制，垃圾回收算法优缺点**

jvm内存结构分为五大区域：虚拟机栈、本地方法栈、程序计数器、堆区、方法区。其中虚拟机栈、本地方法栈与程序计数器这3个区域随线程而生、随线程而灭，因此就不需要考虑过多内存垃圾回收问题，因为一个方法调用结束或者线程结束时，内存自然就跟随着回收了。

我们就把重点放在方法区与堆区，这部分内存的分配和回收是动态的，正是垃圾收集器所需关注的部分

垃圾收集器在对堆区和方法区进行回收工作前，首先肯定确定这些区域内对象哪些可以被回收，哪些暂时还不能回收，这时就要用到判断对象是否存活的算法！如何失去任何引用，垃圾收集器就把它收走。

**（1）引用计数算法**

早期策略。在这种算法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，就将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

缺点：循环引用时无效

如：如父对象有一个对子对象的引用，子对象反过来引用父对象。

**2）可达性分析算法**

可达性分析算法是从离散数学中引入的，也是如今正在使用的策略，程序把所有的引用关系看作一张图（DOM图类似），从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，就如递归思想一般，遍历所有，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点，无用的节点将会被判定为是可回收的对象。**（java采用，gcroot存在于**虚拟机栈，本地方法区，方法区，包括类的静态属性引用，常量对象引用，本地方法栈中对象引用等**）**

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象的引用链是否可达，判定对象是否存活都与“引用”有关。

当对象失去所有引用时，我们就可以说对象生命周期结束了，即为死亡，就该回收它啦

常见的垃圾回收器

**1. Serial收集器**

新生代单线程收集器，标记和清理都是单线程，优点是简单高效。是client级别默认的GC方式。

**2. Serial Old收集器**

老年代单线程收集器，Serial收集器的老年代版本。

**3. ParNew收集器**

新生代收集器，可以认为是Serial收集器的多线程版本,在多核CPU环境下有着比Serial更好的表现。

**4. Parallel Scavenge收集器**

并行收集器，追求高吞吐量，高效利用CPU。吞吐量一般为99%， 吞吐量= 用户线程时间/(用户线程时间+GC线程时间)。适合后台应用等对交互相应要求不高的场景。是server级别默认采用的GC方式。

**5. Parallel Old收集器**

Parallel Scavenge收集器的老年代版本，并行收集器，吞吐量优先。

**6. CMS(Concurrent Mark Sweep)收集器**

高并发、低停顿，追求最短GC回收停顿时间，cpu占用比较高，响应时间快，停顿时间短，多核cpu 追求高响应时间的选择。

**7.G1收集器**

G1收集器是一款成熟的商用的垃圾收集器，将内存分为一些一定大小的内存区域，基于“标记——整理”算法实现的，基于标记——整理算法和复制算法保证不会产生内存空间碎片

回收方法：

**【标记-清除算法**】首先标记需要回收的对象，然后进行回收；缺点：回收速度慢，回收之后产后生大量不连续的内存碎片，后期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造成内存空间浪费。

**【复制算法】**堆中edian区域采用此种回收方法，将内存空间等分为两份，每次只使用其中一份，当满了之后将还有效的对象复制到另一份内存中，，然后把原来的空间进行清除，不会产生内存碎片，但是可用内存空间减半。

**【标记-整理算法】**不仅对需要回收的对象进行整理，还对有效对象进行整理，不会产生内存碎片

**【分代收集算法】**是一种比较智能的算法，也是现在JVM使用最多的一种算法，其实不是一个新的算法，而是在具体的场景自动选择以上三种算法进行垃圾对象回收。

新生代：目的是回收那些生命周期短的对象，主要存放新产生的对象。新生代按照8:1:1分为eden区、survivor0、survivor1，大部分对象在eden区中生成，当eden满时，将存活的对象复制到survivor0，然后清空eden，当eden、survivor0都满了时，将这两个区中存活的对象复制到survivor1，然后清空eden、survivor0，当着三个区都满了时则把存货对象复制到老年代，如果老年代也满了则触发FullGC。新生代的全回收叫MinorGC，MinorGC发生频率比较高，不一定等到新生代满了时才进行。

老年代：存放对象生命周期较长，且内存大概是新生代的两倍，老年代存活对象生命周期长，因此MajorGC发生频率较低。

永久代：主要存放静态文件，如Java类，方法等。永久带对垃圾回收基本没有影响，当应用动态生成或者调用一些类的时候，例如反射、动态代理CGLib等bytecode框架时需要永久带来保存新生成的类。

总结：【1】在新生代中，每次垃圾收集时都有大批对象死去，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。【2】老年代中因为对象存活率高、没有额外空间对他进行分配担保，就必须用标记-清除或者标记-整理。

由于永久代经常会内存不够用或者发生内存泄露，JDK1.8开始废弃了永久代，取而代之的是元空间（直接存在内存中可自定义大小）,主要存放类的元数据。

**6、Lock机制**

synchronized关键字虽然可以解决大部分多线程锁的问题，但是仍旧存在下述问题：

假如持有锁的某线程因等待长时IO或者其他原因阻塞，其他等待的线程无法响应中断，只能不断等待；

多线程下只有读操作是不会发生冲突的，但synchronized关键字对读和写操作均一视同仁，所以当一个线程进行读取操作时，其他线程只能不断等待；

使用synchronized关键字无法确认线程是否成功获取到锁。

与synchronsized关键字不同的是，Lock必须由用户手动执行加锁/释放锁操作，当持有锁的线程发生异常时，该线程不会自动释放锁，可能会导致死锁，故Lock必须在try{}catch{}块中进行，并且将释放锁的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。   
 公平锁即尽量以请求锁的顺序来获取锁。比如同时有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该锁。非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

总结一下ReenTrantLock的lock()方法：

ReenTrantLock的lock()方法是通过其内部抽象类Sync的lock()方法来实现的；

Sync没有直接实现lock()方法，由其方法实现由ReentrantLock的内部类FairSync、NonfairSync来完成;

Sync继承自AQS，而FairSync、NonfairSync又继承了Sync，故Sync、FairSync、NonfairSync均是AQS的子类；

FairSync、NonfairSync的lock()方法是调用AQS的acquire(1)来实现的，AQS的acquire(int)方法又涉及到tryAcquire(int)方法，但是AQS没有实现tryAcquire(int)方法，是由其子类来实现的。

7、锁机制

9、线程池原理  
10、类加载机制  
11、HTTP、HTTPS区别  
12、HTTP报文  
13、TCP三次握手、四次挥手  
14、wait()、sleep()区别  
15、常用集合类实现原理  
16、ConcurrentHashMap原理  
17、List、Map、Set 哪个能存null，哪个能重复？  
18、HashMap实现原理，如何计算key的index，存储方式  
19、反射原理  
20、AQS