以下是本文的目录大纲：

**一.内存模型的相关概念**

**二.并发编程中的三个概念**

**三.Java内存模型**

## 一.内存模型的相关概念

大家都知道，计算机在执行程序时，每条指令都是在CPU中执行的，而执行指令过程中，势必涉及到数据的读取和写入。由于程序运行过程中的**临时数据是存放在主存**（物理内存）当中的，这时就存在一个问题，由于**CPU执行速度很快，而从内存读取数据和向内存写入数据的过程跟CPU执行指令的速度比起来要慢的多**，因此**如果任何时候对数据的操作都要通过和内存的交互来进行，会大大降低指令执行的速度。因此在CPU里面就有了高速缓存。**

　　也就是，当程序在运行过程中，会将**运算需要的数据从主存复制一份到CPU的高速缓存当中**，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算**结束之后，再将高速缓存中的数据刷新到主存当**中。举个简单的例子，比如下面的这段代码：

|  |  |
| --- | --- |
| 1 | i = i + 1; |

 　　当线程执行这个语句时，会先从主存当中读取i的值，然后复制一份到高速缓存当中，然后CPU执行指令对i进行加1操作，然后将数据写入高速缓存，最后将**高速缓存中i最新的值刷新到主存**当中。

　　这个代码在单线程中运行是没有任何问题的，但是在**多线程中运行就会有问题**了。在多核CPU中，**每条线程可能运行于不同的CPU**中，因此**每个线程运行时有自己的高速缓存**（对单核CPU来说，其实也会出现这种问题，只不过是以线程调度的形式来分别执行的）。本文我们以多核CPU为例。

　　比如同时有2个线程执行这段代码，假如初始时i的值为0，那么我们希望两个线程执行完之后i的值变为2。但是事实会是这样吗？

　　可能存在下面一种情况：初始时，两个线程分别读取i的值存入各自所在的CPU的高速缓存当中，然后线程1进行加1操作，然后把i的最新值1写入到内存。此时线程2的高速缓存当中i的值还是0，进行加1操作之后，i的值为1，然后线程2把i的值写入内存。

　　最终结果i的值是1，而不是2。这就是著名的**缓存一致性**问题。通常称这种**被多个线程访问的变量为共享变量**。

　　也就是说，如果**一个变量在多个CPU中都存在缓存**（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题。

**为了解决缓存不一致性问题，通常来说有以下2种解决方法：**

　　1）通过在**总线加LOCK#锁的方式**

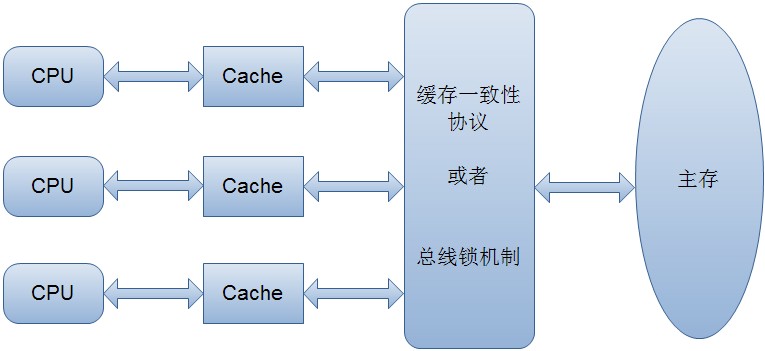
　　2）通过**缓存一致性协议**

　　这2种方式都是**硬件层面上提供**的方式。

　　在早期的CPU当中，是通过在总线上加LOCK#锁的形式来解决缓存不一致的问题。因为CPU和其他部件进行通信都是通过总线来进行的，如果对总线加LOCK#锁的话，也就是说阻塞了其他CPU对其他部件访问（如内存），从而使得只能有一个CPU能使用这个变量的内存。比如上面例子中 如果一个线程在执行 i = i +1，如果在执行这段代码的过程中，在总线上发出了LCOK#锁的信号，那么只有等待这段代码完全执行完毕之后，其他CPU才能从变量i所在的内存读取变量，然后进行相应的操作。这样就解决了缓存不一致的问题。

　　但是上面的方式会有一个问题，由于在**锁住总线期间，其他CPU无法访问内存，导致效率低下。**

　　所以就**出现了缓存一致性协议**。最出名的就是Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，**如果发现操作的变量是共享变量，**即在其他CPU中也存在该变量的副本，会发出信号**通知其他CPU将该变量的缓存行置为无效状态**，因此**当其他CPU需要读取这个变量**时，发现自己缓存中缓存该变量的缓存行是无效的，**那么它就会从内存重新读取。**



## 二.并发编程中的三个概念

在并发编程中，我们通常会遇到以下三个问题：**原子性问题，可见性问题，有序性问题**。我们先看具体看一下这三个概念：

**1.原子性**

原子性：即一个操作或者多个操作 **要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行**。

一个很经典的例子就是银行账户转账问题：

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。

试想一下，如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。然后又从B取出了500元，取出500元之后，再执行 往账户B加上1000元 的操作。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

所以这2个操作必须要具备原子性才能保证不出现一些意外的问题。

同样地反映到并发编程中会出现什么结果呢？

举个最简单的例子，大家想一下假如为一个32位的变量赋值过程不具备原子性的话，会发生什么后果？

|  |  |
| --- | --- |
| 1 | i = 9; |

假若一个线程执行到这个语句时，我暂且假设为一个32位的变量赋值包括两个过程：**为低16位赋值，为高16位赋值。**

那么就可能发生一种情况：**当将低16位数值写入之后，突然被中断，而此时又有一个线程去读取i的值，那么读取到的就是错误的数据**。

**2.可见性**

可见性是指当**多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。**

举个简单的例子，看下面这段代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //线程1执行的代码  int i = 0;  i = 10;    //线程2执行的代码  j = i; |

假若执行线程1的是CPU1，执行线程2的是CPU2。由上面的分析可知，当线程1执行 i =10这句时，会先**把i的初始值加载到CPU1的高速缓存中，然后赋值为10**，那么在CPU1的高速缓存当中i的值变为10了，却**没有立即写入到主存**当中。

此时线程2执行 j = i，它会先去**主存读取i的值并加载到CPU2的缓存当**中，注意此时内存当中i的值还是0，**那么就会使得j的值为0，而不是10.**

这就是**可见性的问题**，线程1对变量i修改了之后，**线程2没有立即看到线程1修改的值**。

**3.有序性**

有序性：即程序执行的顺序按照代码的先后顺序执行。举个简单的例子，看下面这段代码：

|  |  |
| --- | --- |
| 1  2  3  4 | int i = 0;  boolean flag = false;  i = 1;                //语句1  flag = true;          //语句2 |

上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操作。从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生**指令重排序**（Instruction Reorder）。

下面解释一下什么是指令重排序，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它**不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。**

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

但是要注意，虽然处理器会对指令进行重排序，但是它会保证程序最终结果会和代码顺序执行结果相同，那么它靠什么保证的呢？再看下面一个例子：

|  |  |
| --- | --- |
| 1  2  3  4 | int a = 10;    //语句1  int r = 2;    //语句2  a = a + 3;    //语句3  r = a\*a;     //语句4 |

这段代码有4个语句，那么可能的一个执行顺序是：

https://images0.cnblogs.com/blog/288799/201408/212305263939989.jpg那么可不可能是这个执行顺序呢： 语句2   语句1    语句4   语句3

**不可能，因为处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行。**

虽然重排序不会影响单个线程内程序执行的结果，但是多线程呢？下面看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //线程1:  inited = false;  context = loadContext();   //语句1  inited = true;             //语句2    //线程2:  while(!inited ){    sleep()  }  doSomethingwithconfig(context); |

上面代码中，由于语句1和语句2**没有数据依赖性，因此可能会被重排序**。假如发生了重排序，在线程1执行过程中**先执行语句2，**而此是线程2会以为初始化工作已经完成，那么就会跳出while循环，去执行doSomethingwithconfig(context)方法，**而此时context并没有被初始化，就会导致程序出错。**

从上面可以看出，**指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。**

也就是说，要想**并发程序正确地执行**，必须要**保证原子性、可见性以及有序性**。只要有一个没有被保证，就有可能会导致程序运行不正确。

## 三.Java内存模型

在前面谈到了一些关于内存模型以及并发编程中可能会出现的一些问题。下面我们来看一下Java内存模型，**研究一下Java内存模型为我们提供了哪些保证**以及在java中提供了哪些方法和机制来让我们在进行多线程编程时能够保证程序执行的正确性。

　　在Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽各个硬件平台和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。那么Java内存模型规定了哪些东西呢，它定义了程序中变量的访问规则，往大一点说是定义了程序执行的次序。注意，为了获得较好的执行性能，Java内存模型并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序。也就是说，**在java内存模型中，也会存在缓存一致性问题和指令重排序的问题。**

　　Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存（**各自的缓存中**）中进行，而不能直接对主存进行操作。并且每个线程**不能访问其他线程的工作内存**。

　　举个简单的例子：在java中，执行下面这个语句：

|  |  |
| --- | --- |
| 1 | i  = 10; |

 　　执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值10写入主存当中。

　　那么Java语言 本身对 **原子性、可见性以及有序**性提供了哪些保证呢？

**1.原子性**

　　在Java中，对基本数据类型的变量的读取和赋值操作**是原子性操作**，即**这些操作是不可被中断的，要么执行，要么不执行**。

　　上面一句话虽然看起来简单，但是理解起来并不是那么容易。看下面一个例子i：

　　请分析以下哪些操作是原子性操作：

|  |  |
| --- | --- |
| 1  2  3  4 | x = 10;         //语句1  y = x;         //语句2  x++;           //语句3  x = x + 1;     //语句4 |

 　　咋一看，有些朋友可能会说上面的4个语句中的操作都是原子性操作。其实**只有语句1是原子性操作**，其他三个语句都不是原子性操作。

　　语句1是直接将数值10赋值给x，也就是说**线程执行这个语句的会直接将数值10写入到工作内存(高速缓存)中（直接复制，唯一一个操作，不可以被中断）。**

　　语句2实际上包含2个操作，它**先要去读取x的值，再将x的值写入工作内存**，虽然读取x的值以及 将x的值写入工作内存 这**2个操作都是原子性操作**，但是**合起来就不是原子性操作了（可能读取完，还没写入就被中断了）。**

　　同样的，x++和 x = x+1包括3个操作：**读取x的值，进行加1操作，写入新的值**。

 　　所以上面4个语句**只有语句1的操作具备原子性**。

　　也就是说，只有**简单的读取、赋值**（而且**必须是将数字赋值给某个变量**，变量之间的相互赋值不是原子操作）**才是原子操作。**

　　不过这里有一点需要注意：在32位平台下，对64位数据的读取和赋值是需要通过两个操作来完成的，不能保证其原子性。但是好像在最新的JDK中，JVM已经保证对64位数据的读取和赋值也是原子性操作了。

　　从上面可以看出，Java内存模型**只保证了基本读取和赋值是原子性操作**，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻**只有一个线程执行该代码块**，那么自然就不存在原子性问题了，从而保证了原子性。

**2.可见性**

　　对于可见性，Java提供了**volatile关键字来保证可见性。**

　　当一个共享变量**被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。（volatile会立刻更新）**

　　而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

　　另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。**（个人的理解是他们的效率太慢了）**

**3.有序性**

　　在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

　　在Java里面，可以通过**volatile关键字来保证一定的“有序性”**（具体原理在下一篇博文讲述）。另外可以通过**synchronized和Lock来保证有序性**，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

　　另外，**Java内存模型具备一些先天的“有序性”**，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。**如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序**。