<http://tutorials.jenkov.com/java-concurrency>

多线程编程带来的问题：

多个线程同时执行同一段程序，会发生同时读写某一个内存区域，引发内存状态不一致的问题。

例如，线程1读某个内存区域时线程2在写入，线程1读到的数据是多少呢？旧数据？线程2写入的数据？两者的混合？

再例如，线程1和线程2同时往某个内存区域写入数据，结束后内存中的数据是什么呢？线程1写入的数据？线程2写入的数据？两者的混合？

上面两个例子的任一结果都可能发生，并且每次运行的结果都可能不一样。为了避免这种不确定性，程序应该采取措施控制多个线程对共享资源（内存，文件，数据库。。。）的同时访问。

多线程的优点：

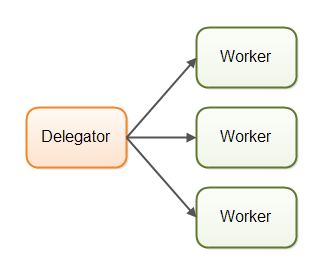
1. 更好的利用资源（例如一个线程在执行IO操作，IO操作一般很慢，让CPU一直等待很浪费，CPU可以执行其他线程，提高CPU的利用率）
2. 在某些情况下能简化程序设计
3. 程序的响应性更好

多线程的开销：

1. 更复杂的程序设计（控制访问共享资源）
2. 上下文切换开销大（从执行线程1切换到执行线程2，需要保存线程1的本地数据和程序指针等，然后加载程序2的本地数据和程序指针等）
3. 增加资源消耗（线程需要申请很多资源才能执行：CPU时间片，内存，操作系统资源等）

并发模型：指定多线程如何进行合作完成jobs。

1. 并行worker

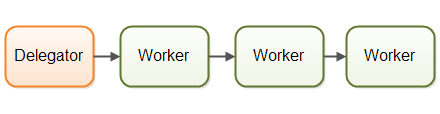


委托者把jobs分派给不同的worker（thread），每个worker完成一个完整的job，每个worker在各自的thread中执行，所有worker并行，可能在不同的CPU核心。

优点：容易理解，增加并行数量只需增加worker。

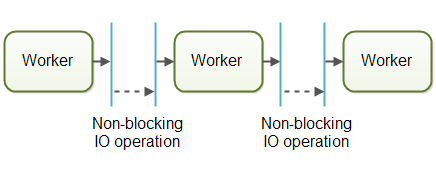
缺点：保持共享数据的状态一致性很复杂。worker不会保存共享数据的状态，每次需要时都必须重新读取，很耗时（如果从外部存储读取）。无法保证job执行的顺序。

1. 流水线worker

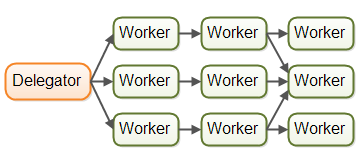


每个worker只执行整个job的一部分，完成后转交给下一个worker。每个worker在各自的thread中运行，worker之间无共享状态。

系统通常使用流水线并发模型设计非阻塞IO，即当一个worker开始执行IO操作时，不会阻塞等待IO结束。因为IO操作太慢了，等待太浪费CPU时间。CPU能够先做其他事，当IO操作结束后，把结果转交给下一个worker。



在非阻塞IO中，IO操作决定了worker之间的界限。worker尽可能多干活，如果遇到IO操作，就放弃手中的工作（放弃CPU），当IO操作结束，下一个worker拿到结果继续干活。



多条流水线，同时把多个job part交给同一个worker。

使用流水线并发模型的系统称为**响应式系统**或**事件驱动系统**。worker响应系统中发生的事件（HTTP请求等）。

优点：无共享状态。worker能够保持自己的状态。能够保证job执行的顺序。

缺点：一个job分散在多个worker身上。难以理解哪些代码干了什么工作。代码难写。

1. 函数式并行（Functional Parallelism）

通过函数调用实现程序，函数调用的过程就是消息的发送。通过函数的参数传递数据，避免共享数据的竞态条件，使函数的执行类似于原子操作，每个函数调用独立于其它函数调用。

1. 哪一种并发模型最好，取决于程序需要干什么活。

并发（Concurrency ）VS.并行（Parallelism）

并发：多个任务轮流使用CPU时间片，一段时间内多个任务都有进度。

并行：在某个时刻，有多个任务在同时执行（前提是有多个CPU核心，单核CPU无法并行）。

任务1执行完后任务2才能执行，这个系统是串行的。

任务1和任务2能够同时执行，这个系统是并行的。

任务1和任务2不能同时执行，但能够轮流使用CPU时间片，一段时间内都有进度，这个系统是并发的。

3个任务，2个CPU核心：

如果3个任务只能一个一个依次执行，那么是串行的。

如果任一时刻有2个任务在同时执行，那么是并行的。

如果任两个任务能够轮流使用某个CPU核心的时间片交替执行，那么是并发的。

在多核CPU的系统中，大多是既并行又并发。

**竞态条件**（Race Conditions）和**临界区**（Critical Sections）：

**竞态条件**：多个线程对临界资源进行读和**写**操作时（至少一个线程写），最终的结果取决于这些线程的执行顺序。称为竞态条件发生。例如多线程对共享变量进行write，共享变量的最终值是不确定的，与线程的执行顺序有关。

**临界区**：多个线程访问**临界资源**（一次只允许一个线程访问的资源，例如打印机）的那段代码。

为避免竞态条件的发生，必须保证临界区以原子操作执行，即一个线程在执行的时候，别的线程不能执行直到上一个线程离开临界区。通过对临界区进行线程同步即可实现。

通过把一个大的临界区分解为一些小的临界区，可以减小多线程对共享资源的竞争，增加整个临界区的吞吐量。

**共享资源**：可以被一个以上任务使用的资源叫做共享资源。为了防止数据被破坏，每个任务在与共享资源打交道时，必须独占该资源。此处与临界资源是相同意思。

**线程安全**：一个类或者一段代码或者一个变量，满足以下三个条件：

• 多个线程同时访问时，其表现出正确的行为。

• 无论操作系统如何调度这些线程， 无论这些线程的执行顺序如何交织（interleaving）。

• 调用端代码无须额外的同步或其他协调动作。

说明这个类或这段代码或这个变量是线程安全的。

**不可变的**类（对象）是线程安全的，因为它们的状态永远不会改变。例如，String,Integer

即使对某个对象的使用是线程安全的，但如果这个对象**引用了共享资源**（例如文件或数据库），整个应用可能是非线程安全的。例如，线程1和线程2分别创建了自己的数据库连接对象，连接对象1和连接对象2，对连接对象本身的使用是线程安全的，但是连接对象指向的数据库可能是非线程安全，这样整个应用对数据库的操作就是非线程安全的。

局部变量存储在执行线程自己的栈里面，因此局部变量不可能在线程间共享，所有**基本类型局部变量都是线程安全的**。对于引用类型的局部变量，有一种操作可能使其非线程安全，根本原因是虽然局部引用变量是线程安全的，但是其引用的堆内存是共享资源。

public void someMethod(){

LocalObject localObject = new LocalObject();

localObject.callMethod();

method2(localObject);

}

public void method2(LocalObject localObject){

localObject.setValue("value"); // localObject是线程安全的

mObj = localObject; // 如果把localObject赋值给某个成员变量mObj，由于mObj可能被其它线程访问，因此使得localObject非线程安全。

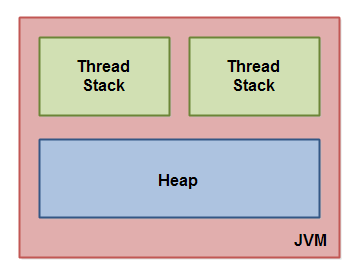
}

Java内存模型：

**内存可见性**：一个线程对共享变量值的修改，能够及时地被其他线程看到。

**共享变量：**如果一个变量在多个线程的工作内存中都存在副本，那么这个变量就是几个线程的共享变量。

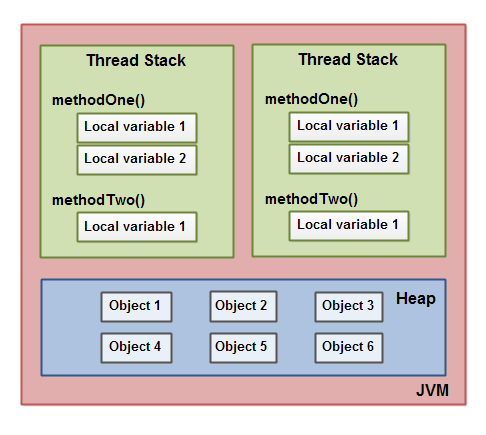
Java内存模型指定了JVM怎样工作在物理内存RAM之上。Java内存模型指定不同线程如何以及何时可以看到由其他线程写入共享变量的值，以及如何在必要时同步对共享变量的访问。



JVM将内存分为堆和栈，每个运行在JVM上的线程有自己的线程栈，线程栈包含一个方法调用栈，线程执行哪个方法，该方法就会入栈，执行完哪个方法，该方法就出栈。线程栈还包括所有调用栈里的方法的局部变量，一个线程只能访问自己的thread stack。线程执行创建的局部变量对其它线程不可见。即使两个线程执行一段完全相同的代码，这两个线程会分别在自己的thread stack里创建局部变量。

8大基本数据类型的局部变量是完全存储（声明和值）在thread stack里面的，对其他线程不可见。一个线程能把thread stack中基本数据类型的局部变量拷贝一份传递给另外一个线程，但不可能共享。

所有对象Object，不论是哪个线程创建的，都存放在堆heap里面。



一个局部变量，如果是8大基本数据类型，完全存储在thread stack里面。

一个局部变量，如果是引用类型，引用变量存储在thread stack里面，引用的对象存储在heap。

一个对象的成员变量，无论什么类型，都是和它所在的对象一起存在heap。

一个对象的成员方法里面的局部变量，待方法被调用时，存储在thread stack里面。

静态成员变量和类的定义一起，存储在heap。

Heap里面的对象可以被任何线程通过引用变量访问。



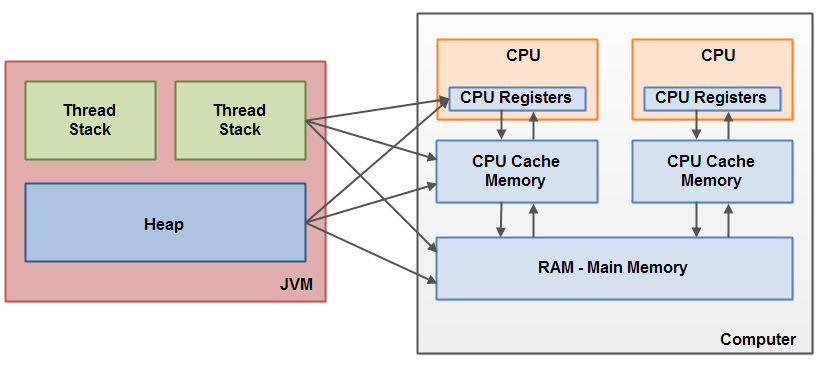
计算机内的物理内存架构：



通常，CPU访问主内存时，会read部分主内存到CPU缓存，可能read部分缓存到内部的寄存器，然后执行操作。当CPU需要将结果写回主内存时，它将从内部寄存器中将值刷新到高速缓冲存储器，并在某个时候将值刷新到主内存。

存储在CPU Cache里的数据通常会在它需要存储其它值时被刷新到主内存，并且不一定一次性刷新所有Cache里的数据。

Java内存模型和物理内存的关系：

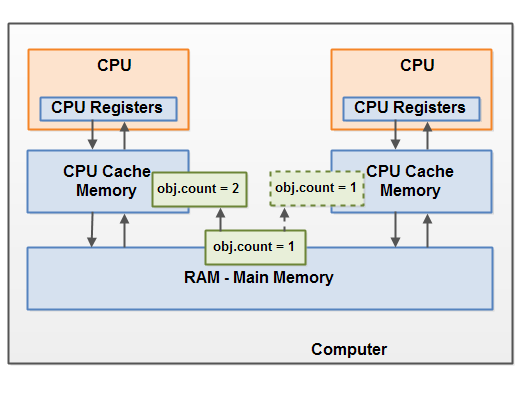


Java内存模型中的thread stack和heap都在RAM里面分配，它们在某些时候会出现在CPU Cache 和CPU寄存器中。因为对象和变量可能出现在不同的物理内存中，这会带来问题：

**1.共享变量的可见性**：

某个线程update了共享变量的值，没有及时刷新到主内存，导致其他线程read的仍是旧值。

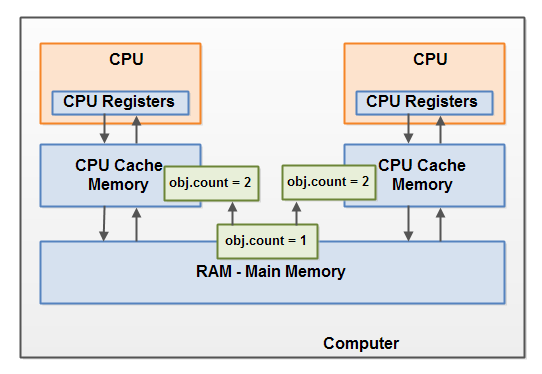
Java 1.5开始引入的volatile关键字能够解决，它能保证直接从主内存read变量值，更新变量值直接write回主内存。



**2.发生竞态条件：**

两个线程同时update共享变量的值，共享变量总共被update了两次，但是在每个CPU缓存里，共享变量只被update了一次，将来某个时候写回主内存时，主内存里的值可能是线程1更新的值，可能是线程2更新的值，也可能是两者的混合（竞态条件发生）。

同步机制（synchronized 和 lock）能够解决，同步能够保证在某一时刻最多只有一个线程能够执行临界区的代码，并且能够保证同步代码块里访问的所有变量都是从主内存直接read，当线程退出同步代码块，所有update的变量全部刷新到主内存，无论变量是否声明为volatile.



synchronized关键字实现线程同步：

synchronized锁的是一个对象，当不同的线程想要执行synchronized包围的一段代码或方法时，如果synchronized锁的是同一个对象，那么在某个时刻只能有一个线程执行这段代码，其它线程会一直阻塞知道上个线程离开同步代码块。

synchronized修饰实例成员方法，锁的是this（当前方法所属的实例对象）。

synchronized修饰静态成员方法，锁的是当前类对应的class对象（Class类的实例）。

synchronized修饰实例成员方法内部的代码块，需要显式指定锁（也叫同步监视器对象）。

synchronized修饰实例静态方法内部的代码块，需要显式指定锁。

synchronized不够先进，java.util.concurrent包里提供了能够更精细控制并发的工具类。

**volatile关键字保证内存可见性、禁止编译器指令重排序：**

从Java 5开始引入。

保证内存可见性：每次read volatile变量都是从主内存直接读取，每次write volatile变量都是直接写入主内存。

volatile还能保证Happens-Before：

1. 线程A先write一个volatile变量，然后线程B read这个volatile变量。线程A write**之前**所有变量对线程A可见，线程B read**之后**所有变量对线程B可见。进一步解释，当一个线程write一个volatile变量时，不仅仅是这个volatile变量本身被写入主内存，而且在write这个volatile变量之前，所有被线程change的其它变量都会被刷新到主内存。当一个线程read一个volatile变量时，该线程也会从主内存中read所有其它变量。
2. 对volatile变量的read和write指令不会被JVM重排序（JVM出于对性能的考虑可能会重排序指令，如果它监测到排序后程序行为不会改变）。

sharedObject.nonVolatile1 = 123;

sharedObject.nonVolatile2 = 456;

sharedObject.nonVolatile3 = 789;

sharedObject.volatile = true; //a volatile variable

int someValue1 = sharedObject.nonVolatile4;

int someValue2 = sharedObject.nonVolatile5;

int someValue3 = sharedObject.nonVolatile6;

上面的代码，JVM可能会对前3条指令重排序，但保证都会在volatile variable write之前执行；JVM也可能会对后3条指令重排序，但保证都会在volatile variable write之后执行。JVM不会对volatile read/write指令重排序。

下面的代码，不用synchronized，只需一个volatile修饰就能保证，只有线程1调put()，只有线程2调take()程序就能表现正常。

public class Exchanger {

private Object object = null;

private volatile hasNewObject = false;

public void put(Object newObject) {

while(hasNewObject) {

//wait - do not overwrite existing new object

}

object = newObject;

hasNewObject = true; //volatile write，既保证之前被线程change的变量全部可见（刷新到主内存），又能禁止指令重排序。

}

public Object take(){

while(!hasNewObject){ //volatile read

//wait - don't take old object (or null)

}

Object obj = object;

hasNewObject = false; //volatile write

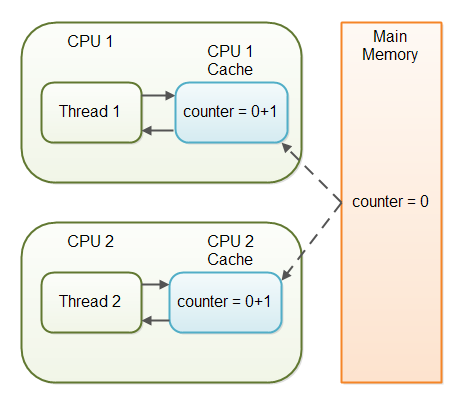
return obj;

}

}

volatile不能胜任的情景：

如果一个线程需要先read volatile变量的值，然后基于这个值生成一个新值，再把这个新值write给这个volatile变量，比如自增自减操作，这种情况下，volatile不能保证可见性。原因很简单，因为对volatile变量的读或写是直接从主内存读或直接写入主内存，能够保证可见性，但是如volatile变量自增操作，实际是读和写的组合操作，不是原子操作，在读和写的之间，可能会产生竞态条件。例如，



counter是volatile变量。

Thread1 read counter的值0从主内存到CPU cache，将counter的值加1，还没来得及将改变后的值写回主内存，Thread2从主内存read的counter的值仍为0，在自己的CPU cache里面也将counter的值加1，改变后的值也没写回主内存。这种情况下Thread1和Thread2实际上是不同步的，counter的值应该增加2，但是每个线程在各自的CPU cache里面counter的值为1，主内存里面counter的值为0。即使最后Thread1和Thread2将counter的值写回主内存，其值也是错误的。

**volatile总结：**

1. volatile适合多线程中每个线程只对volatile共享变量进行读**or**写操作的情况，这样即使不额外同步也能保证程序逻辑正确。
2. volatile不适合多线程中某些线程对volatile共享变量的写操作依赖读取先前值的情况，因为对volatile的读操作和写操作不是原子操作，读和写之间很可能别的线程修改了共享变量的值，导致程序逻辑错误。这种情况需要把读操作和写操作放到同步代码块中，变为原子操作。举例，

volatile int i = ... // 共享变量

// 临界区

If (i <100) { // read volatile

... // 别的线程修改了i，使其>=100

read i or write i; // 如果继续执行，程序的逻辑就出错了。因此必须对临界区进行同步。

}

1. 因为read volatile是直接从主内存read，write volatile是直接write入主内存，因此开销比直接访问CPU缓存大，效率低。并且volatile还阻止指令重排序，可能降低性能。所以应当仅在需要强制实现可见性时才使用volatile。