java内存模型与并发技术

汇报人：秦冲

目录

- Java Memory Model(java内存模型)
- 理解synchronized同步是如何工作的
- 分析程序什么时候需要同步
- 正确使用volatile关键字
- 并发程序设计的几个策略

Java内存模型（Java Memory Model,JMM）

1. 什么是Java内存模型
2. Java内存模型相关概念
3. Java线程和内存交互行为定义
4. Ordering（有序性）&visibility（可见性）
5. JMM相关java语言规范

- 物理计算机中的并发问题

“让计算机并发执行若干个任务”与“更充分地利用计算机处理器的效能”之间的关系并没有想象中的简单，其中一个重要的复杂性来源是绝大多数的运算任务都不可能只依靠处理器计算就能完成，处理器至少要与内存交互，如读取数据、存储运算结果等，这个I/O操作是很难消除的（无法仅仅依靠寄存器来完成所有任务）。

- 缓存一致性问题

在多处理器系统中，每个处理器都有自己的高速缓存，而他们又共享同一主内存（Main Memory）当多个处理器的运算任务都涉及到同一块主内存区域时，将可能导致各自的缓存数据不一致，真的出现这种情况的话那么会导致同步回到主内存的数据不知道以谁的缓存数据为准。为了解决缓存不一致问题，需要各个处理器在访问缓存时都遵循一些协议。

JMM(java内存模型)

特点：

- 内存管理的跨平台统一的模型

  write-once, run-anywhere concurrent applications in Java
- 定义了Java线程和内存交互的规则
- 通过一组语义规则来描述尤其是多线程之间共享内存的模式，保证多线程程序结果的可预测，语义一致性
- 不同于C或C++等其他语言，同平台无关
- 所有的实例变量，静态变量和数组元素都存放在堆内存里
- 线程本地变量在堆栈中，不受JMM影响

Java内存模型相关概念

- **工作内存（Thread working copy memory）**

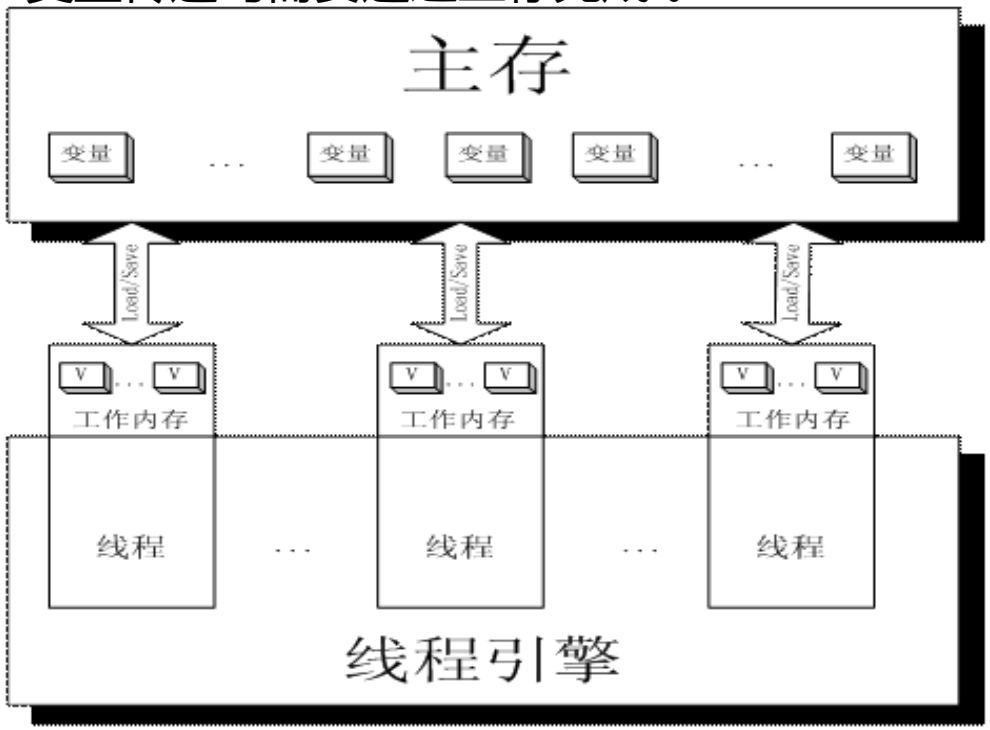
  - 每个线程都有自己的工作内存，这里工作内存可与之前讲的处理器高速缓存类比。
  - 线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量。
  - 不同线程之间无法相互直接访问，变量传递均需要通过主存完成 。

- **主内存（The main memory）**

  所有的变量都存储在主内存中，类比一下就是我们所说的java堆内存。

- **线程引擎**

  保证线程的正确执行顺序。



java线程和内存交互行为定义

- 内存间交互操作

lock(锁定)：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。

unlock(解锁)：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其它线程锁定。

read(读取)：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便以后的load动作使用。

load(载入)：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。

use(使用)：作用于工作内存的变量，它把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。

assign(赋值)：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store(存储)：作用于工作内存的变量，它把工作内存中的一个变量的值传送到主内存中，以便随后的write操作使用。

write(写入)：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

Ordering（有序性）&visibility（可见性）

- 可见性： 可见性是指当一个线程修改了共享变量的值，其它线程能够立即得知这个修改。

- 有序性：Java中天然的有序性可以总结为一句话：如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的。前半句是指“线程内表现为串行的语义”（Within-Thread As-If-Serial Semantics），后半句是指“指令重排序”现象和“工作内存与主内存同步延迟现象”。

Ordering（有序性）&visibility（可见性）

- 场景分析

```
class Simple {
    int a = 1, b = 2;
      //Thread 1 executes
    void to() {
      a = 3; //This can appear to happen second
      b = 4; // This can appear to happen first
    }
    //Thread 2 executes
    void fro()
      System.out.println("a= " + a + ", b=" + b);
    }
   }
```

- 下面哪种结果是正确的：

a=1, b=2

a=1, b=4

a=3, b=2

a=3 ,b=4

先行发生原则（Happens-Before）

“先行发生”是Java内存模型中定义的两项操作之间的偏序关系，如果说操作A先行发生于操作B，其实就是说在发生操作B之前，操作A产生的影响可能被操作B观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等。

举个例子：

```
//以下操作在线程A中执行
i = 1;

//以下操作在线程B中执行
j = i;

//以下操作在线程C中执行
i = 2;
```

JMM相关java语言规范

- JMM定义了保证内存操作跨线程的正确的有序性（Ordering）和可见性（visibility） 方法

- Java提供的技术和工具
  - synchronized 块
  - volatile 变量(在JDK5+的JVM中得到修补)
  - 工具类： java.util.concurrent.locks
  - 原子变量java.util.concurrent.atomic
  - Final变量(在JDK5+的JVM中得到修补)

正确的理解synchronized

1. synchronized 作用说明
2. Synchronized内存模型语义分析
3. 如何判定多线程的程序何时需要Synchronized
4. 同步需求的JMM内存交互分析练习
5. 锁对象的引用在同步块中发生修改会出现什么？
6. Hostspot JVM的synchronized优化技术
7. 直接使用synchronized 不足之处

synchronized 作用说明

- 作用：给对象和方法或者代码块加锁，当它锁定一个方法或者一个代码块的时候，同一时刻最多只有一个线程执行这段代码

- 什么时候需要synchronized ？

  在多线程环境中，存在着共同操作访问一个对象类数据的情况，难免有数据不同步的情况出现。synchronized就是java内置修饰符专门来处理同步的，

- synchronized是一个内置锁的加锁机制，当某个方法加上synchronized关键字后，就表明要获得该内置锁才能执行，并不能阻止其他线程访问不需要获得该内置锁的方法。

synchronized 作用说明

- 使用场景：大概分为四个：静态代码块、静态方法、非静态代码块、非静态方法。
- 非静态方法、非静态代码块

  其中，非静态方法锁定的是实例对象，在synchronized修饰的方法间是相斥的，调用时都要先获取到对象锁。非静态代码块跟非静态方法一致。多了一点是，可以指定锁定的对象为非当前类实例对象。非静态方法只有同一个实例的同一个synchronized或者不同的synchronized方法之间存在着同步关系。
- 关键字synchronized修饰非静态方法或者代码块时取得的是对象锁，而不是把一段代码或方法（函数）当作锁，这里如果是把一段代码或方法（函数）当作锁，其实获取的也是对象锁，只是监视器（对象）不同而已，哪个线程先执行带synchronized关键字的方法，哪个线程就持有该方法所属对象的锁，其他线程都只能呈等待状态。但是这有个前提：既然锁叫做对象锁，那么势必和对象相关，所以多个线程访问的必须是同一个对象。
- 静态方法、静态代码块

  静态方法锁定的是类，在所有synchronized修饰的静态方法间是相斥的。

  静态代码块跟静态方法一致。多了的就是，锁定的类可以为任意指定类（静态方法只能锁定当前类）。

- **Example:**

```
// block until obtain lock
synchronized(obj) {
        // get main memory value of field1 and field2
        int x = obj.field1;
        int y = obj.field2;
        obj.field3 = x+y;
         // commit value of field3 to main memory
}
// release lock
```

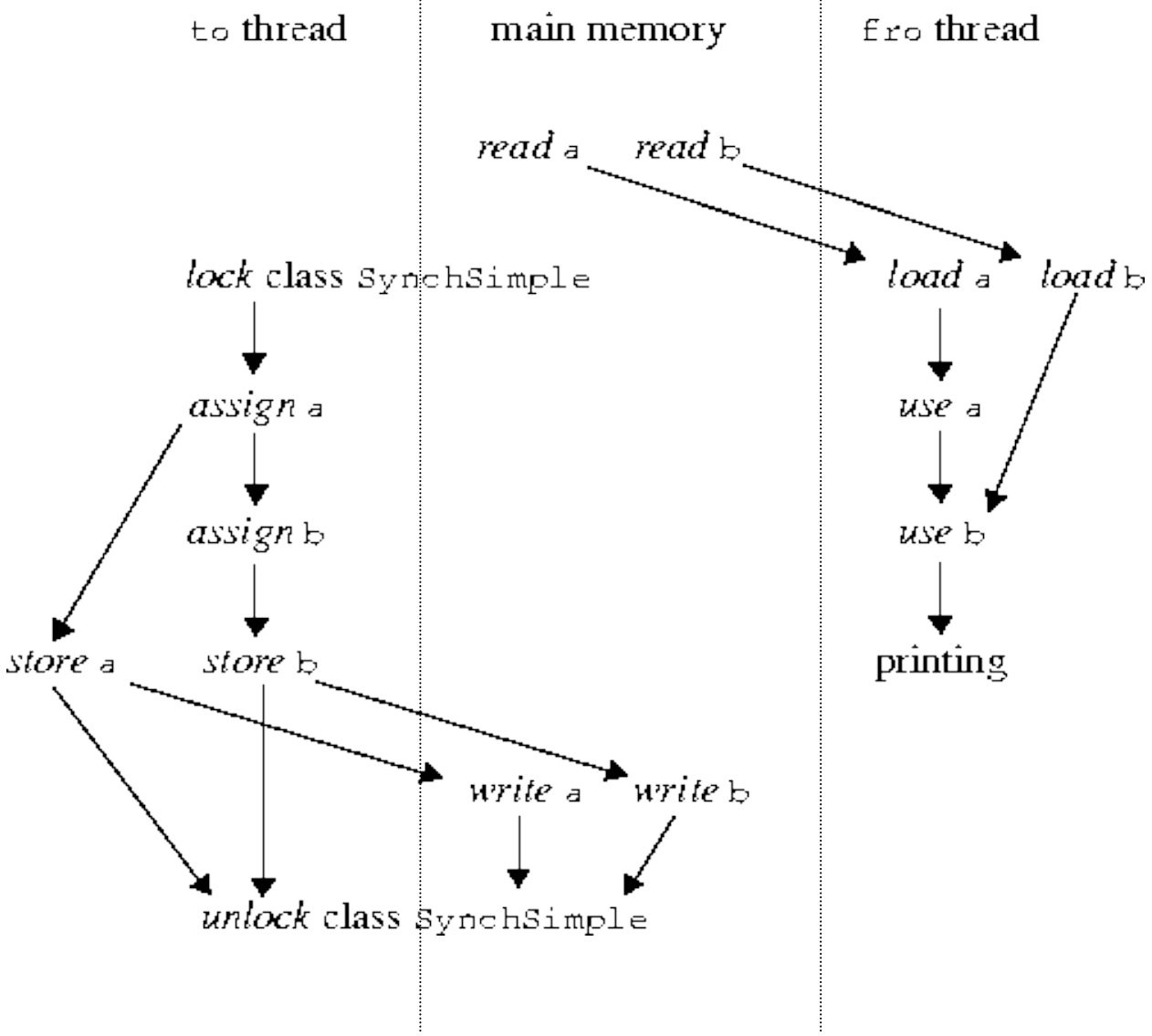
如何分析多线程并发需求

```
class SynchSimple {
   int a = 1, b = 2;
      //Thread 1 executes
   synchronized void to() {
      a = 3;
      b = 4;
   }
    //Thread 2 executes
   void fro()
      System.out.println("a= " + a + ", b=" + b);
   }
   }
```

- 以下哪种结果是正确的?
  a=1, b=2
  a=1, b=4
  a=3, b=2
  a=3 ,b=4
- 这是一个线程安全的类吗?

接下来我们通过JMM语义来分析这个例子

JMM内存交互分析


to thread
main memory
fro thread
read a
read b
lock class SynchSimple
load a
load b
assign a
use a
assign b
use b
store a
store b
printing
write a
write b
unlock class SynchSimple

JMM内存交互分析

- SynchSimple 分析：
  - *write* a 和 *write* b 在synchronized 规则中并没有规定发生的顺序约束
  - *read* a 和 *read* b. 同样也没有规定发生的顺序
  - **结果说明的什么问题？**
  - 对一个方法声明synchronized 并不能够保证这个方法行为产生的结果是一个原子操作，也就是说*write* a 和 *write* b 两个操作在main memory不是原子行为，虽然单个都是原子操作。

JMM内存交互分析

- **Example:**

```
class Foo{
    int a = 1, b = 2;
    synchronized void to() {
            a = 3;
            b = 4;
    }
    synchronized void fro()
            System.out.println("a= " + a + ", b=" + b);
    }
   }
```

分析结果是什么?

问题？

- **锁对象的引用在同步块中发生修改会出现什么?**

```
public void foo(int isze){
    synchronized(intArr){
      if(intArr.length < size){
                int []newIntArr = new int[size];
                System.arraycopy(intArr,0,newIntArr,0,intArr.length);
                intArr = newintArr;
      }
      ...
    }
    ...
}
```

- **这个方法的同步块有可能被多个线程并发执行！！**
- 有可能在**intArr.length < size**的条件下获得两把不同的锁

直接使用synchronized 不足之处和发展

- 不能够扩展多个对象
- 当在等待锁对象的时候不能中途放弃，直到成功
- 等待没有超时限制
- Thread.interrupt()不能中断阻塞
- JDK5中提供更加灵活的机制：Lock和Condition
- synchronized在JDK6中性能将会有很大提升

Hostspot JVM的synchronized优化技术

- 锁省略：锁对象的引用是线程本地对象（线程的堆栈内的对象）

```
public String getStoogeNames() {
    Vector v = new Vector();
    v.add("Moe");
    v.add("Larry");
    v.add("Curly");
    return v.toString();
}
```

Hostspot JVM的synchronized优化技术

- 锁粗化：锁粗化就是把使用同一锁对象的相邻同步块合并的过程

```
public void addStooges(Vector v) {
    v.add("Moe");
    v.add("Larry");
    v.add("Curly");
}
```

Hostspot JVM的synchronized优化技术

- 自适应锁优化技术
  - 实现阻塞有两种的技术，即让操作系统暂挂线程，直到线程被唤醒，或者使用旋转（spin） 锁。旋转锁基本上相当于以下代码：

```
while (lockStillInUse)
;
```

  - Hotspot JVM可以对持有时间短的锁使用旋转，对持有时间长的锁使用暂挂。

理解和使用Volatile变量

- 理解Volatile变量
- 使用valatile

理解Volatile变量

- Volatile关键字是Java虚拟机提供的最轻量级的同步机制

- 当一个变量定义为volatile之后，它具备两种特性：

第一是保证此变量对所有线程的可见性，这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其它线程来说是立即得知的，而普通变量不能做到这一点，普通变量的值在线程间传递均需要通过主内存来完成。例如，线程A修改一个普通变量的值，然后向主内存进行回写，另外一条线程B在线程A回写完成了之后再从主内存进行读取操作，新变量值才会对线程B可见。

第二个语义是禁止指令重排序优化，普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果而不能保证变量赋值操作的顺序与程序代码中的执行顺序一致。因为在一个线程的方法执行过程中无法感知到这点，这也就是Java内存模型中描述的所谓的“线程内表现为串行的语义”

理解Volatile变量

- 关于Volatile变量在并发下是不是安全的讨论？

```
//volatile变量自增运算测试
public class volatileTest{
        public static volatile int race = 0;
        public static void increase(){
                race ++;
        }

        public static final int THREAD_COUNT = 20;
```

理解Volatile变量

```
    public static void main(String[] args){
        Thread[] threads = new Thread(THREAD_COUNT);
        for(int i = 0 ; i< THREAD_COUNT ;i++){
            threads[i] = new Thread(new Runnable(){
                public void run(){
                    for(int i = 0;i<10000;i++){
                        increase();
                    }
                }
            });
            threads[i].start();
        }
        //等待所有累加线程都结束
        while(Thread.activeCount() > 1)
            Thread.yield();

        System.out.println(race);
    }
}
```

理解Volatile变量

- Volatile变量第二个语义禁止指令重排序

```
Map configOptions;
char[] configText;
//此变量必须定义为volatile
volatile boolean initialized = false;
//假设以下代码在线程A中执行，模拟读取配置信息，当读取完成后将initialized设置为true以通知其它线程配置可用
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText,configOptions);
initialized = true;

//假设以下代码在线程B中执行，等待initialized为true,代表线程A已经把配置信息初始化完成
while(!initialized){
        sleep();
}
//使用线程A初始化的配置信息
doSomethingWithConfig();
```

使用valatile

- volatile 变量++操作不是原子行为

```
volatile int x= 1;
…
x++;//不是一个原子操作,需要多条指令
```

- Volatile关键字与synchronized块相比，所需的编码较少，并且运行时开销也比较少，但是但是它所能实现的功能也仅仅是synchronized的一部分。
- 由于volatile变量只能保证可见性，在不符合以下两条规则的运算场景中仍然需要通过加锁（使用synchronized或java.util.concurrent中的原子类）来保证原子性。

1. 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。
2. 变量不需要与其它的状态变量共同参与不变约束。

并行程序设计的几个策略

1. 如何安全可靠的取消正在执行的任务
2. 如何安全可靠的结束正在执行的线程
3. 如何处理InterruptedException

取消任务和中断进程

- 为什么需要cancellable （中断正在进行的活动）
  - User-requested cancellation
  - Time-limited activities
  - Application events
  - Errors
  - Shutdown
- 安全快速可靠结束一个线程是一个比较复杂的话题
- Java没有提供安全的强迫一个Thread结束的方法
- Java提供了一种协商的机制：Interruption

java中断机制

关于java的中断机制主要是以下三个问题：

- How：
  - ⑩其他线程发起中断请求怎么办
- When
  - ⑩何时检测正在执行的任务被请求中断
- What
  - ⑩如何响应中断请求

Example: java中断机制

下面来分析一个例子看看这样做有何缺点?

```
class Task implements Runnable {
    private volatile boolean stop = false;
    public void stop() { stop = true; }
    public void run() {
      while (!stop)
                runTask();
      try { Thread.sleep(100); }
        ...;
    }
    private void runTask() { /*...*/ }
}
```

一个更好的方法：Interruption

- 礼貌地劝告另一个线程在它愿意并且方便的时候停止它正在做的事情。(Interruption is a cooperative mechanism)
- Thread中断状态（interrupted status）：线程的一个内部属性，类型：boolean,初始值：false.
- Thread.interrupt()：设置interrupted status 为true

- 分析:
  - How:Thread.interrupt()方法. interrupt() 只是设置线程的中断状态。表示当前线程应该停止运行。
  - When:在 Thread.sleep()、 Thread.join() 或 Object.wait()等方法中取消阻塞并抛出 InterruptedException,也可以程序检测:
    - 轮询中断状态：Thread.isInterrupted()
    - 读取并清除：Thread.interrupted()
  - InterruptibleChannel (java.nio):Most standard Channels implement InterruptibleChannel
  - 等待获取文件锁定时可以被另一个线程中断

- **其他问题**
  - 不能打断一些IO操作，比如文件操作
  - 无法终止在synchronized块上锁住的Thread
  - Synchronous socket I/O in java.io

- **检查型异常：java.lang.InterruptedException**

  当线程在很长一段时间内一直处于正在等待、休眠或暂停状态（Object.wait(), Object.wait(long), Object.wait(long, int), Thread.sleep(long)），而另一个线程用 Thread 类中的 interrupt 方法中断它时，抛出该异常。

- ***阻塞（blocking）方法***

  方法签名包括抛出 InterruptedException，调用一个阻塞方法则意味着这个方法也是一个阻塞方法

- **当中断阻塞方法时，抛出InterruptedException**

  Thread 在 Thread.sleep() 和 Object.wait() 等方法中支持的取消机制,表明可以提前返回。

- 当一个阻塞方法检测到中断并抛出 InterruptedException 时，它清除中断状态

如何处理InterruptedException

- 错误的做法：swallow interruption requests

```
try {
        Task task = queue.take();
         task.execute();
 } catch (InterruptedException e) {
        log.error(...);
}
```

如何处理InterruptedException

- Restore the interrupt

  如果真的要湮灭InterruptedException，应该保留被别人中断的证据，交给高层去决定。

```
//恰当的做法:
try {
        Task task = queue.take();
         task.execute();
 } catch (InterruptedException e) {
        log.error(...);

        Thread.currentThread().interrupt();
}
```

- Propagate the InterruptedException
  - ⑩ 重新抛出它，如果抛到顶层，可以结束当前线程

几点建议

- 开发多线程系统一定要理解java的内存模型，那样你才能够正确地分析线程需求
- 尽量使用现成的解决方案，util.concurrent
- 优化的前提是保证程序的正确性，其次才是提高程序的性能
- 缩小锁的作用范围
- 缩短锁的存在时间

谢谢观看！