java 线程安全

要认识java线程安全，必须了解两个主要的点：java的内存模型，java的线程同步机制。特别是内存模型，java的线程同步机制很大程度上都是基于内存模型而设定的。

浅谈java内存模型：

不同的平台，内存模型是不一样的，但是jvm的内存模型规范是统一的。**其实java的多线程并发问题最终都会反映在java的内存模型上，所谓线程安全无非是要控制多个线程对某个资源的有序访问或修改**。总结java的内存模型，要解决两个主要的问题：可见性和有序性。我们都知道计算机有高速缓存的存在，处理器并不是每次处理数据都是取内存的。JVM定义了自己的内存模型，屏蔽了底层平台内存管理细节，对于java开发人员，要清楚在jvm内存模型的基础上，如果解决多线程的可见性和有序性。

那么，何谓可见性?

多个线程之间是不能互相传递数据通信的，它们之间的沟通只能通过共享变量来进行。Java内存模型（JMM）规定了jvm有主内存，主内存是多个线程共享 的。当new一个对象的时候，也是被分配在主内存中，每个线程都有自己的工作内存，工作内存存储了主存的某些对象的副本，当然线程的工作内存大小是有限制 的。当线程操作某个对象时，执行顺序如下：

(1) 从主存复制变量到当前工作内存 (read and load)
(2) 执行代码，改变共享变量值 (use and assign)
(3) 用工作内存数据刷新主存相关内容 (store and write)

JVM规范定义了线程对主存的操作指 令：read，load，use，assign，store，write。当一个共享变量在多个线程的工作内存中都有副本时，如果一个线程修改了这个共享 变量，那么其他线程应该能够看到这个被修改后的值，这就是多线程的可见性问题。

那么，什么是有序性呢?

线程在引用变量时不能直接从主内存中引用,如果线程工作内存中没有该变量,则会从主内存中拷贝一个副本到工作内存中,这个过程为read-load,完成后线程会引用该副本。当同一线程再度引用该字段时,有可能重新从主存中获取变量副本(read-load-use),也有可能直接引用原来的副本(use),也就是说 read,load,use顺序可以由JVM实现系统决定。

线程不能直接为主存中中字段赋值，它会将值指定给工作内存中的变量副本(assign),完成后这个变量副本会同步到主存储区(store-write)，至于何时同步过去，根据JVM实现系统决定.有该字段,则会从主内存中将该字段赋值到工作内存中,这个过程为read-load,完成后线程会引用该变量副本，当同一线程多次重复对字段赋值时,比如：

```
for(int i=0;i<10;i++)
 a++;
```

线程有可能只对工作内存中的副本进行赋值,只到最后一次赋值后才同步到主存储区，所以assign,store,weite顺序可以由JVM实现系统决定。假设有一个共享变量x，线程a执行x=x+1。从上面的描述中可以知道x=x+1并不是一个原子操作，它的执行过程如下：

1 从主存中读取变量x副本到工作内存

2 给x加1

3 将x加1后的值写回主存

如果另外一个线程b执行x=x-1，执行过程如下：

1 从主存中读取变量x副本到工作内存

2 给x减1

3 将x减1后的值写回主存

那么显然，最终的x的值是不可靠的。假设x现在为10，线程a加1，线程b减1，从表面上看，似乎最终x还是为10，但是多线程情况下会有这种情况发生：

1：线程a从主存读取x副本到工作内存，工作内存中x值为10
2：线程b从主存读取x副本到工作内存，工作内存中x值为10
3：线程a将工作内存中x加1，工作内存中x值为11
4：线程a将x提交主存中，主存中x为11
5：线程b将工作内存中x值减1，工作内存中x值为9
6：线程b将x提交到中主存中，主存中x为9

同样，x有可能为11，如果x是一个银行账户，线程a存款，线程b扣款，显然这样是有严重问题的，要解决这个问题，必须保证线程a和线程b是有序执行的，并且每个线程执行的加1或减1是一个原子操作。看看下面代码：

```
public class Account {

    private int balance;

    public Account(int balance) {
        this.balance = balance;
    }
```

```
    public int getBalance() {
        return balance;
    }

    public void add(int num) {
        balance = balance + num;
    }

    public void withdraw(int num) {
        balance = balance - num;
    }

    public static void main(String[] args) throws InterruptedException {
        Account account = new Account(1000);
        Thread a = new Thread(new AddThread(account, 20), "add");
        Thread b = new Thread(new WithdrawThread(account, 20), "withdraw");
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.println(account.getBalance());
    }

    static class AddThread implements Runnable {
        Account account;
        int     amount;

        public AddThread(Account account, int amount) {
            this.account = account;
            this.amount = amount;
        }

        public void run() {
            for (int i = 0; i < 200000; i++) {
                account.add(amount);
            }
        }
    }

    static class WithdrawThread implements Runnable {
        Account account;
        int     amount;

        public WithdrawThread(Account account, int amount) {
            this.account = account;
            this.amount = amount;
        }

        public void run() {
            for (int i = 0; i < 100000; i++) {
                account.withdraw(amount);
            }
        }
    }
}
```

第一次执行结果为10200，第二次执行结果为1060，每次执行的结果都是不确定的，因为线程的执行顺序是不可预见的。这是java同步产生的根源，synchronized关键字保证了多个线程对于同步块是互斥的，synchronized作为一种同步手段，解决java多线程的执行有序性和内存可见性，而volatile关键字之解决多线程的内存可见性问题。后面将会详细介绍。

synchronized关键字

上面说了，java用synchronized关键字做为多线程并发环境的执行有序性的保证手段之一。当一段代码会修改共享变量，这一段代码成为互斥区或临界区，为了保证共享变量的正确性，synchronized标示了临界区。典型的用法如下：

Java代码 ：

```
synchronized(锁){
  临界区代码
}
```

为了保证银行账户的安全，可以操作账户的方法如下：

```
public synchronized void add(int num) {
     balance = balance + num;
}
public synchronized void withdraw(int num) {
     balance = balance - num;
}
```

刚才不是说了synchronized的用法是这样的吗：

```
synchronized(锁){
临界区代码
```

```
}
```

那么对于public synchronized void add(int num)这种情况，意味着什么呢？其实这种情况，锁就是这个方法所在的对象。同理，如果方法是public static synchronized void add(int num)，那么锁就是这个方法所在的class。

理论上，每个对象都可以做为锁，但一个对象做为锁时，应该被多个线程共享，这样才显得有意义，在并发环境下，一个没有共享的对象作为锁是没有意义的。假如有这样的代码：

```
public class ThreadTest{
  public void test(){
     Object lock=new Object();
     synchronized (lock){
        //do something
     }
  }
}
```

lock变量作为一个锁存在根本没有意义，因为它根本不是共享对象，每个线程进来都会执行Object lock=new Object();每个线程都有自己的lock，根本不存在锁竞争。

每个锁对象都有两个队列，一个是就绪队列，一个是阻塞队列，就绪队列存储了将要获得锁的线程，阻塞队列存储了被阻塞的线程，当一个被线程被唤醒(notify)后，才会进入到就绪队列，等待cpu的调度。当一开始线程a第一次执行account.add方法时，jvm会检查锁对象account的就绪队列是否已经有线程在等待，如果有则表明account的锁已经被占用了，由于是第一次运行，account的就绪队列为空，所以线程a获得了锁，执行account.add方法。如果恰好在这个时候，线程b要执行account.withdraw方法，因为线程a已经获得了锁还没有释放，所以线程b要进入account的就绪队列，等到得到锁后才可以执行。

一个线程执行临界区代码过程如下：

1 获得同步锁

2 清空工作内存

3 从主存拷贝变量副本到工作内存

4 对这些变量计算

5 将变量从工作内存写回到主存

6 释放锁

可见，synchronized既保证了多线程的并发有序性，又保证了多线程的内存可见性。

volatile关键字:

volatile是java提供的一种同步手段，只不过它是轻量级的同步，为什么这么说，因为volatile只能保证多线程的内存可见性，不能保证多线程的执行有序性。而最彻底的同步要保证有序性和可见性，例如synchronized。任何被volatile修饰的变量，都不拷贝副本到工作内存，任何修改都及时写在主存。因此对于Valatile修饰的变量的修改，所有线程马上就能看到，但是volatile不能保证对变量的修改是有序的

java 工作内存

所谓线程的“工作内存”到底是个什么东西？有的人认为是线程的栈，其实这种理解是不正确的。看看JLS（java语言规范）对线程工作 内存的描述，线程的working memory只是cpu的寄存器和高速缓存的抽象描述。

可能 很多人都觉得莫名其妙，说JVM的内存模型，怎么会扯到cpu上去呢？在此，我认为很有必要阐述下，免 得很多人看得不明不白的。先抛开java虚拟机不谈，我们都知道，现在的计算机，cpu在计算的时候，并不总是从内存读取数据，它的数据读取顺序优先级 是：寄存器－高速缓存－内存。线程耗费的是CPU，线程计算的时候，原始的数据来自内存，在计算过程中，有些数据可能被频繁读取，这些数据被存储在寄存器 和高速缓存中，当线程计算完后，这些缓存的数据在适当的时候应该写回内存。当个多个线程同时读写某个内存数据时，就会产生多线程并发问题，涉及到三个特 性：原子性，有序性，可见性。在《线程安全总结》这篇文章中，为了理解方便，我把原子性和有序性统一叫做“多线程执行有序性”。支持多线程的平台都会面临 这种问题，运行在多线程平台上支持多线程的语言应该提供解决该问题的方案。

那么，我们看看JVM，JVM是一个虚拟 的计算机，它也会面临多线程并发问题，java程序运行在java虚拟机平台上，java程序员不可能直接去控制底层线程对寄存器高速缓存内存之间的同 步，那么java从语法层面，应该给开发人员提供一种解决方案，这个方案就是诸如 **synchronized, volatile,锁机制（如同步块，就绪队 列，阻塞队列）等等。这些方案只是语法层面的，但我们要从本质上去理解它，不能仅仅知道一个 synchronized 可以保证同步就完了。** 在这里我说的是jvm的内存模型，是动态的，面向多线程并发的，沿袭JSL的“working memory”的说法，只是不想牵扯到太多底层细节，因为 《线程安全总结》这篇文章意在说明怎样从语法层面去理解java的线程同步，知道各个关键字的使用场 景。

今天有人问我，那java的线程不是有栈吗？难道栈不是工作内存吗？工作内存这四个字得放到具体的场景 中描述，方能体现它具体的意义，在描述JVM的线程同步时，工作内存指的是寄存器和告诉缓存的抽象描述，具体请自行参阅JLS。上面讲的都是动态的内存模 型，甚至已经超越了JVM的范围，那么JVM的内存静态存储是怎么划分的？今天还有人问我，jvm的内存模型不是有eden区吗？也不见你提起。我跟他说，这是两个角度去看的，甚至是两个不同的范围，动态的线程同步的内存模型，涵盖了cpu，寄存器，高速缓存，内存；JVM的静态内存储模型只是一种对内 存的物理划分而已，它只局限在内存，而且只局限在JVM的内存。那些什么线程栈，eden区都仅仅在JVM内存。

说说JVM的线程栈和有个朋友反复跟我纠结的eden区吧。JVM的内存，被划分了很多的区域：

1.程序计数器
每一个Java线程都有一个程序计数器来用于保存程序执行到当前方法的哪一个指令。

2.线程栈
线程的每个方法被执行的时候，都会同时创建一个帧（Frame）用于存储本地变量表、操作栈、动态链接、方法出入口等信息。每一个方法的调用至完成，就意味着一个帧在VM栈中的入栈至出栈的过程。如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果VM栈可以动态扩展（VM Spec中允许固定长度的VM栈），当扩展时无法申请到足够内存则抛出OutOfMemoryError异常。

3.本地方法栈
4.堆
每个线程的栈都是该线程私有的，堆则是所有线程共享的。当我们new一个对象时，该对象就被分配到了堆中。但是堆，并不是一个简单的概念，堆区又划分了很多区域，为什么堆划分成这么多区域，这是为了JVM的内存垃圾收集，似乎越扯越远了，扯到垃圾收集了，现在的jvm的gc都是按代收集，堆区大致被分为三大块：新生代，旧生代，持久代（虚拟的）；新生代又分为eden区，s0区，s1区。新建一个对象时，基本小的对象，生命周期短的对象都会放在新生代的eden区中，eden区满时，有一个小范围的gc（minor gc），整个新生代满时，会有一个大范围的gc（major gc），将新生代里的部分对象转到旧生代里。

5.方法区
其实就是永久代（Permanent Generation），方法区中存放了每个Class的结构信息，包括常量池、字段描述、方法描述等等。VM Space描述中对这个区域的限制非常宽松，除了和Java堆一样不需要连续的内存，也可以选择固定大小或者可扩展外，甚至可以选择不实现垃圾收集。相对来说，垃圾收集行为在这个区域是相对比较少发生的，但并不是某些描述那样永久代不会发生GC（至 少对当前主流的商业JVM实现来说是如此），这里的GC主要是对常量池的回收和对类的卸载，虽然回收的“成绩”一般也比较差强人意，尤其是类卸载，条件相当苛刻。

6.常量池
Class
文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量表(constant_pool table)，用于存放编译期已可知的常量，这部分内容将在类加载后进入方法区（永久代）存放。但是Java语言并不要求常量一定只有编译期预置入Class的常量表的内容才能进入方法区常量池，运行期间也可将新内容放入常量池（最典型的String.intern()方法）。