

Java并发和多线程基础总结

基础volatile和synchronized

volatile

功能

1. 保证多线程的可见性
2. 禁止一部分的重排序。
3. volatile 是轻量级的synchronized
4. 对任意单个的volatile的读/写是原子性的（volatile=1/return volatile），但是复合型操作不支持。（volatile++操作）

特性

对于volatile来说具有以下特性 1.volatile标记的变量在读操作的时候，一定是最新的值。为什么? 1、lock前缀的支持,volatile规定每次修改操作必须刷新到内存，读操作也要到内存中去取新值。 2、总线的支持，多条总线事务同一时刻同一时刻只有一条能获得访问内存的权限，因此，只要有线程修改了，那么读操作一定取到的是新值。####内存语意 读：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变 写：当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内

原理

```
instance = new Singleton(); // instance是volatile变量
0x01a3de1d: movb $0x0,0x1104800(%esi);0x01a3de24: **lock** addl $0x0,(%esp);
```

lock前缀的指令在多核处理器下会触发两种操作

1. 将当前处理器缓存行的数据写回到系统内存。
2. 这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

依赖CPU的缓存一致性的支持

volatile规则表

当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序。这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。

当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前。

当第一个操作是volatile写，第二个操作是volatile读时，不能重排

在每个volatile写操作的前面插入一个StoreStore屏障。

在每个volatile写操作的后面插入一个StoreLoad屏障。

在每个volatile读操作的后面插入一个LoadLoad屏障。

在每个volatile读操作的后面插入一个LoadStore屏障。

synchronized

1. 原子性。
2. 由于线程同步，某一时刻，只能一个线程操作共享变量，也就保证了多线程间的可见性

使用

- 普通同步方法:锁是this锁
- 代码块:锁是该对象锁
- 静态同步方法:锁是当前Class对象锁

原理

- 都是使用Monitor(监视器)对象来操作的 monitorenter在编译时插入到synchronized的开始位置,monitorexit在编译时插入在synchronized的结束位置和者异常结束的位置。
- 任何对象都是锁对象，那么任何对象都与一个monitor相关联，当获取到monitor，该monitor就会处于锁定状态。当指令执行到monitorenter时，就会去尝试获取该monitor的所有权。

原子性

- 通过锁来实现一些列操作的原子性。但是性能差，会造成阻塞，增加性能消耗。
- 通过CAS(compare and swap)来实现，此实现针对于赋值操作。i++，
 - 通过E(期望值,也就是缓存/本地内存/工作内存的副本),V(当前值)来操作。通过E和主内存的值进行比较，如果相等的话，就会将新值赋值给当前值。
 - 问题，会造成ABA的问题，当其他线程将共享变量的值变换为B，然后再该为A，那么该线程去将本地内存的值刷到主内存的时候，将会执行成功。但是结果可能是错的。
- 通过CAS+version的方式来解决ABA的问题。
 - 也就是，共享变量V的值为 1A->2B->3A 那么通过此版本号去控制，就不会巧妙的去除ABA问题。
- 注意，CAS只能处处理一个变量，如果多个可以使用ij = 1a的方式，AtomicReference保证引用对象的原子性。
- CAS 不需要添加锁，但是循环验证会增加CPU的消耗。

核心代码

- CAS实现操作,CASXXXX方法就是CAS操作。Unsafe.class 操作c代码。

```
do{ N = E+1; }while(CASXXXX((E,N));
```

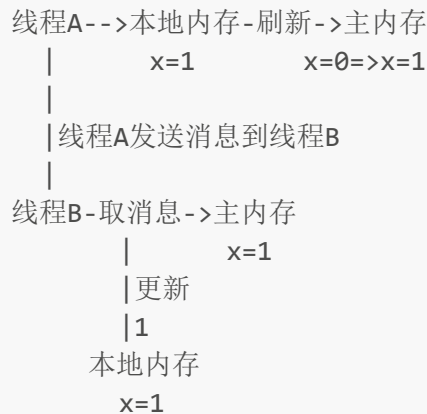
java concurrent包下的atomic类

AtomicInteger AtomicBoolean... (CAS,会出现BAB的问题)
AtomicStampedReference 解决了CAS的ABA的问题

Java内存模型

概念

- Java Memory Model(JMM)决定了一个线程对共享变量做出的修改 *何时* 对另一个线程可见*。
- 每一个线程对都有一块儿 *本地内存*（本地内存是JVM的一个抽象概念，并不是真实存在的，它涵盖了缓存、写缓冲区、寄存器以及其它硬件和编译器优化）本地内存存放的是用以 *读/写的共享变量在主内存的一个副本*
- 多线程通信



重排序

概念

1. 编译器优化的重排序。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。现代处理器采用了指令级并行技术（Instruction-Level Parallelism, ILP）来将多条指令重叠执行。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。由于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



ALT

- JVM通过在编译器编译时，使用内存屏障指令，来禁止特定类型的处理器重排序
- JVM 内存屏障屏障指令

StoreStore Barriers: 写写屏障
 LoadStore Barriers: 读写屏障
 StoreLoad Barriers: 写读屏障

- 示例：
 1. Load1;LoadLoad;Load2;
 2. Store1;StoreStore;Store2;
 3. Load1;LoadStore;Store2;
 4. Store1;StoreLoad;Load2;
- 写读(StoreLoad)屏障能完成上面3种的效果
 1. Store1对其他处理器可见，即将本地内存变量刷新到主内存操作，必须要在Load2及Load2后面的指令先执行;

2. StoreLoad Barriers会使在该屏障之前的所有内存访问指令（Store和Load）完成之后，才执行屏障之后的内存访问指令

happens-before

- 解释 A操作----->B操作
 - 表示A操作的结果对B操作可见 && A操作的按顺序排在B操作之前 && 并不是所有的操作都在后续操作之前,取决于数据的依赖性
- 三种比较常见的happens-before规则
 1. 一个线程中的前面一个操作必定happens-before该操作的后续操作。（重排序取决于数据的依赖性）
 2. Monitor加锁happens-before解锁之前
 3. volatile修饰的变量，写操作必定happens-before后续对该变量的操作

as-if-serial

无论编译器和处理器怎么重排序优化，必须要保证单线程情况下结果不能改变。

其实java程序并不是顺序执行的。

在不改变程序执行结果的前提下，尽可能提高并行度

重排序对多线程的影响

```
public class Test{
    int a;
    boolean flag = false;

    public void read(){
        //控制依赖关系
        if (flag){//操作1 flag可能读不到true 所以就不会执行下面的操作
            // ==>分解
            // 操作21 int temp = a+1;
            // 操作22 int c = temp
            int c = a+1;
        }
    }

    public void write(){
        a = 1;//操作3
        flag = 0;//操作4
        // 3,4的刷新到主内存的时机未知，有可能是2和4批量刷新至主内存，也有可能是分开刷。
    }

    public static void main(String[] args){
        Thread t2;//执行write()方法 操作 3、4
        Thread t1;//执行read()方法 操作1、2
    }
}
```

t2的执行顺序: 3,4 4,3 t1的执行顺序 21 1 22 1 21 22


最终导致的结果：t2的flag可能会读到false,c的值可能是0,读取线程将会出现和期望不一致的情况

在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果（这也是as-if-serial 语义允许对存在控制依赖的操作做重排序的原因）；但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果。


处理器总线机制

总线工作机制 多个处理器需要对内存进行读取操作的时候,都会向总线发起总线事务。这时候会出现竞争，总线仲裁保证了当前只会有一条总线事务会获取访问内存的权限。这样就保证了内存读写的原子性

JMM不保证对64位的long型和double型变量的写操作具有原子性，而顺序一致性模型保证对所有的内存读/写操作都具有原子性。在一些32位的处理器上，如果要求对64位数据的写操作具有原子性，会有比较大的开销。为了照顾这种处理器，Java语言规范鼓励但不强求JVM对64位的long型变量和double型变量的写操作具有原子性。当JVM在这种处理器上运行时，可能会把一个64位long/double型变量的写操作拆分为两个32位的写操作来执行。这两个32位的写操作可能会被分配到不同的总线事务中执行，此时对这个64位变量的写操作将不具有原子性

- 处理器写long/double的操作流程  ALT

JDK 1.5之前,处理器读取long/double的操作流程

处理器写long/double的操作流程  ALT

JDK 1.5之后,JMM要求读操作必须在单个读事务中完成，保证读操作的原子性; 仅仅允许把64位的long/double拆分成两个32位的写操作来执行，那么这两个写操作就破坏了写操作的原子性

锁

- 锁释放和获取的内存定义

前提：事务总线，保证读写的原子性 释放：在释放锁的时候，JMM会将本地内存中的共享变量刷新到主内存 获取：在获取锁时，JMM会将本地内存的共享变量置为无效，从而被Monitor保护的临界区代码必须从主内存读取共享变量。

- 总结
 1. 线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。
 2. 线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。
 3. 线程A释放锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息。

final内存语义

1) 在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。2) 初次读一个包含final域的对象引用，与随后初次读这个final域，这两个操作之间不能重排序

- final域不能从构造方法中溢出。

```

class A{
    final int a;
    public A(int a){
        this.a = a;
    }
}
//Thread 1 A a = new A(1);
//Thread2 if(a!=null){a.a++}这里可能会出现问题
//this.a = a;在构造方法中可能逃逸，在实例化对象之后才去执行该赋值语句

```

多线程情况下，取到的`a`的值可能是初始值`0`，造成线程不安全 **final**的重排序规则，禁止溢出构造方法之外

双重检查锁(Double-Checked-Locking)

```

public class DoubleCheckLocking{
    private static DoubleCheckLocking singleInstance;
    public static DoubleCheckLocking getInstance(){
        if(null == singleInstance){//1 第一次check
            synchronized(singleInstance){//2 同步
                if(null == singleInstance){//3 第二次check
                    singleInstance = new
DoubleCheckLocking();//4 1.分配内存空间 2.初始化对象 3.将对象指向刚分配的内存地址//不
是原子操作
                }
            }
        }
        //5 可能singleInstance 拥有一块内存地址，但是该内存空间还没有
        初始化完成，看起来singleInstance是!=null的，但是初始化还没结束
        return singleInstance;
    }
}

```

- 由于重排序的影响(正确顺序是 1、2、3)

`memory = allocate();` // 1: 分配对象的内存空间 `instance = memory;` // 3: 设置instance指向刚分配的内存地址 // 注意，此时对象还没有被初始化！ `ctorInstance(memory);` // 2: 初始化对象

解决方案 1.不允许2、3重排序 2.允许2、3重排序，但是不允许其他线程看到这个重排序

- 不允许2、3重排序

jdk 1.5以上版本，将`singleInstance`加上`volatile`关键字,可以防止重排序自身重排序，之前、之后的操作不能在`volatile`之后或之前执行

```

public class DoubleCheckLocking{
    private volatile static DoubleCheckLocking singleInstance;
    public static DoubleCheckLocking getInstance(){
        if(null == singleInstance){

```

```

        synchronized(singleInstance){
            if(null == singleInstance){
                singleInstance = new
DoubleCheckLocking();//不会重排序,分配内存 初始化对象 将对象指向分配的内存地址
            }
        }
    }
    return singleInstance;
}
}
}


```

通过JVM初始化阶段(在Class被加载后,且在线程使用之前),会执行类的初始化,此时JVM会获取一把锁,会同步多个线程对类的初始化操作。

```

public class DoubleCheckLocking{
    private static class DoubleCheckLockingHolder{
        private static DoubleCheckLocking doubleCheckLocking = new
DoubleCheckLocking();
    }
    public static DoubleCheckLocking getInstance(){
        return DoubleCheckLockingHolder.doubleCheckLocking;
    }
}

```

 类的初始化