

JAVA并发编程常识

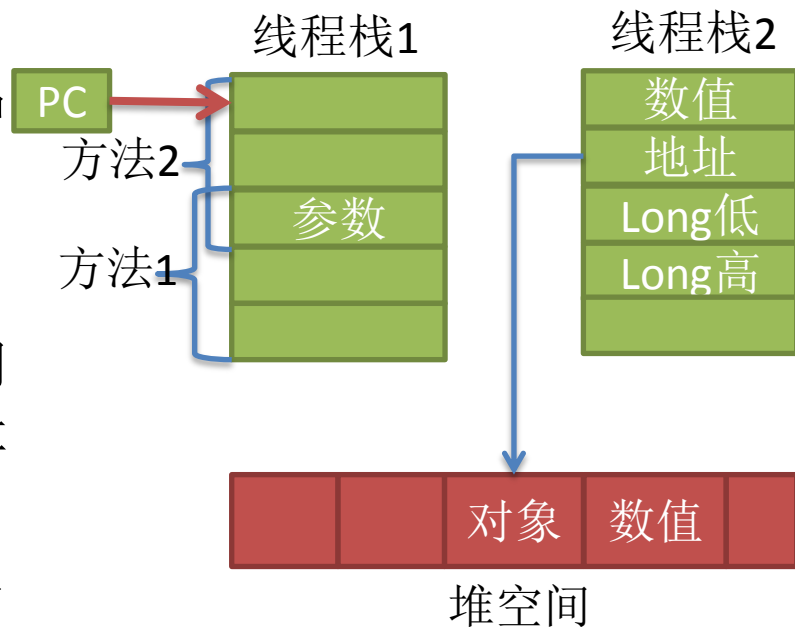
虚极(梁飞)

天猫 TMALL.COM



JVM内存模型

- 堆：
 - 所有对象全部放在共享堆空间中
 - 对象的属性在共享堆空间内
 - 堆内存单字节对齐，short不变
- 栈：
 - 每个线程都有独立的线程栈空间
 - 线程栈只存基本类型和对象地址
 - 栈内存4字节对齐，short变int
 - 对象地址4字节，引用堆空间
 - 方法中局部变量在线程栈空间内
 - 局部变量不会竞争，线程安全
 - 方法参数在栈顶交叉，不拷贝
 - 栈顶寄存，减少中间状态读取
 - PC指针记录当前执行位置



-Xss1M栈大小 -Xmx1G堆大小

原子性

- 对象类型：
 - 对象地址原子读写，线程安全
 - 并发读不可变状态，线程安全
 - 并发读写可变状态，非线程安全
- 基本类型：
 - int,char数值读写，线程安全
 - long,double高低位，非线程安全
 - i++等组合操作，非线程安全

可见性

- **final**
 - 初始化final字段确保可见性
- **volatile**
 - 读写volatile字段确保可见性
- **synchronized**
 - 同步块内读写字段确保可见性
- **happen before**
 - 遵守happen before次序可见性



可排序性

- **Happen Before 法则**
 - 程序次序法则
 - 如果A一定在B之前发生，则happen before,
 - 监视器法则
 - 对一个监视器的解锁一定发生在后续对同一监视器加锁之前
 - **Volatile**变量法则
 - 写volatile变量一定发生在后续对它的读之前
 - 线程启动法则
 - **Thread.start**一定发生在线程中的动作之前
 - 线程终结法则
 - 线程中的任何动作一定发生在括号中的动作之前（其他线程检测到这个线程已经终止，从**Thread.join**调用成功返回，**Thread.isAlive()**返回false）
 - 中断法则
 - 一个线程调用另一个线程的**interrupt**一定发生在另一线程发现中断之前。
 - 终结法则
 - 一个对象的构造函数结束一定发生在对象的**finalizer**之前
 - 传递性
 - A发生在B之前，B发生在C之前，A一定发生在C之前。

系统内存

- MESI协议:

- Modified

- 本CPU写，则直接写到Cache，不产生总线事务；其它CPU写，则不涉及本CPU的Cache，其它CPU读，则本CPU需要把Cache line中的数据提供给它，而不是让它去读内存。

- Exclusive

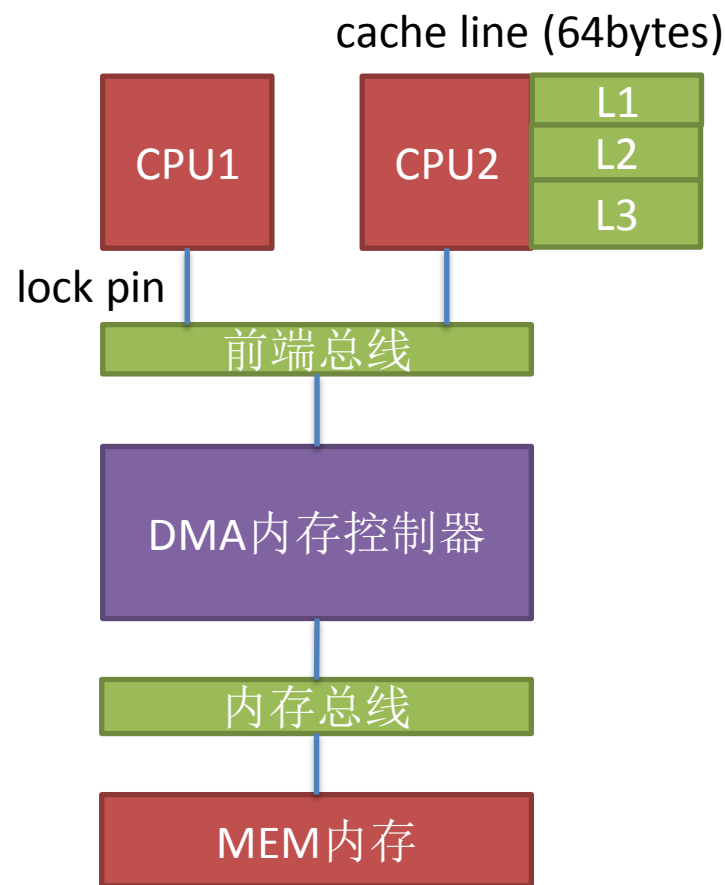
- 只有本CPU有该内存的Cache，而且和内存一致。本CPU的写操作会导致转到Modified状态。

- Shared

- 多个CPU都对该内存有Cache，而且内容一致。任何一个CPU写自己的这个Cache都必须通知其它的CPU。

- Invalid

- 一旦Cache line进入这个状态，CPU读数据就必须发出总线事务，从内存读。



内存栅栏

- 读：
 - volatile int a, b; if(a == 1 && b == 2)
 - JIT通过load acquire依赖保证读顺序：
 - 0x2000000001de819c: adds r37=597,r36;; ;...84112554
 - 0x2000000001de81a0: **ld1.acq** r38=[r37];; ;...0b30014a a010
- 写：
 - volatile A a; a = new A();
 - JIT通过lock addl使CPU的cache line失效：
 - 0x01a3de1d: movb \$0x0,0x1104800(%esi);
 - 0x01a3de24: **lock addl** \$0x0,(%esp);

查看JIT编译结果

- `java -XX:+UnlockDiagnosticVMOptions -XX:PrintAssemblyOptions=hsdis-print-bytes -XX:CompileCommand=print,*AtomicInteger.incrementAndGet`



对齐

LinkedTransferQueue

```
static final class PaddedAtomicReference <T> extends AtomicReference <T> {  
    Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;  
    PaddedAtomicReference(T r) {  
        super(r);  
    }  
}
```

16个地址的长度，刚好占满一个cache line的长度。

确保两个引用，不在同一cache line上，防止多锁竞争。

引用

```
private Channel channel;

public void setChannel (Channel channel ) {
    this.channel = channel;
}

public void run() {
    Channel channel = this.channel; // located reference
    if (channel != null && channel.isConnected()) {
        // do something ...
    }
}

public void check() {
    if (channel != channel)
        throw new Error("check error!");
}
```



单例

```
public class Singleton {  
    private Singleton() {}  
    private static final Singleton  
        instance = new Singleton();  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

必然
使用

```
public class Singleton {  
    private static class Holder { // lazy class  
        static final Singleton  
            instance = new Singleton();  
    }  
    public static Singleton getInstance() {  
        return Holder.instance;  
    }  
}
```

可能
使用

```
public class Singleton {  
    private static Singleton instance = null;  
    public static synchronized Singleton getInstance() {  
        if (instance == null)  
            instance = new Singleton(); // lazy load  
        return instance;  
    }  
}
```

```
public class Singleton {  
    private static volatile Singleton instance = null;  
    public static Singleton getInstance() {  
        if (instance == null) // double check (jdk1.5+)  
            synchronized (Singleton.class)  
                if (instance == null)  
                    instance = new Singleton();  
        return instance;  
    }  
}
```

多锁

```
Object getBean(String id) {  
    synchronized(singletonObjects) {  
        if (! singletonObjects.containsKey(id)) {  
            initBean(id);  
        }  
        return singletonObjects.get(id);  
    }  
}
```



```
Object getBean(String id) {  
    synchronized(beanDefinitionMap) {  
        synchronized(singletonObjects) {  
            if (! singletonObjects.containsKey(id)) {  
                initBean(id);  
            }  
            return singletonObjects.get(id);  
        }  
    }  
}
```

```
void initBean(String id) {  
    synchronized(beanDefinitionMap) {  
        if (! beanDefinitionMap.containsKey(id)) {  
            BeanDefinition def = parseConfig(id);  
            beanDefinitionMap.put(id, def);  
            Object bean = createBean(def);  
            synchronized(singletonObjects) {  
                if (! singletonObjects.containsKey(id)) {  
                    singletonObjects.put(id, bean);  
                }  
            }  
        }  
    }  
}
```



计数

加锁?

计数:

```
int inc = 0;
public int increment() {
    return inc ++;
}
```

AtomicInteger.incrementAndGet();

```
for (;;) {
    int current = get();
    int next = current + 1;
    if (compareAndSet(current, next))
        return next;
}
```

CAS (Lock-Free)

ABA问题



加锁?

最大值:

```
int max = 0;
public int max(int value) {
    if (max < value) max = value;
}
```

场景: 当前值为1, 并发写入2和3

```
AtomicInteger max = new AtomicInteger();
for (;;) {
    int current = max.get();
    if (current > value)
        return current;
    if (max.compareAndSet(current, value))
        return value;
}
```

计数

多线程完成计算:

```
int n = 10;
latch = new CountDownLatch(n);
for (int i = 0; i < n; i++) {
    new Thread() {
        public void run() {
            try {
                // do something ...
            } finally {
                latch.countDown();
            }
        }
    }.start();
}
latch.await();
```

多线程同时并发计数:

```
int n = 10;
barrier = new CyclicBarrier(n);
for (int i = 0; i < n; i++) {
    new Thread() {
        public void run() {
            barrier.await();
            // do something ...
        }
    }.start();
}
```



缓存

```
Map cache = new HashMap();
synchronized(cache) { // wait
    value = cache.get(key);
    if (value == null) {
        value = load(key); // slow
        cache.put(key, value);
    }
}
```



```
ConcurrentMap cache = new ConcurrentHashMap();
value = cache.get(key);
if (value == null) {
    value = load(key); // repeat // lock?
    cache.putIfAbsent(key, value);
}
```



```
Map cache = new HashMap();
value = cache.get(key); // cpu100%
if (value == null) {
    synchronized(cache) {
        value = cache.get(key);
        if (value == null) {
            value = load(key);
            cache.put(key, value);
        }
    }
}
```




```
Map cache = new ConcurrentHashMap();
value = cache.get(key);
if (value == null) {
    synchronized(cache) {
        value = cache.get(key);
        if (value == null) {
            value = load(key);
            cache.put(key, value);
        }
    }
}
```

缓存

```
class Item { volatile Object value; Object get() {...} set(Object value) {...} }
```

```
Map cache = new HashMap();           ConcurrentMap cache = new ConcurrentHashMap();
synchronized(cache) { // map lock item = cache.get(key);
    item = cache.get(key);
    if (item == null) {
        item = new Item(); // quick
        cache.put(key, item);
    }
}                                     if (item == null) {
                                     item = new Item(); // low cost
                                     oldItem = cache.putIfAbsent(key, item);
                                     if (oldItem != null) {
                                     item = oldItem;
                                     }
synchronized(item) { // entry lock }
    value = item.get();
    if (value == null) {
        value = load(key); // slow
        item.set(value);
    }
}                                     synchronized(item) {
                                     value = item.get();
                                     if (value == null) {
                                     synchronized(item) {
                                     value = item.get();
                                     if (value == null) {
                                     value = load(key);
                                     item.set(value);
                                     }
                                     }
                                     }
                                     }
                                     }
```



线程安全策略

- 不可变类
 - 如果一个类初始化后，所有属性和类都是final不可变的，则它是线程安全，不需要任何同步，活性高。
- 线程栈内使用
 - 方法内局部变量使用
 - 线程内参数传递
 - ThreadLocal持有
- 同步锁
 - synchronized的代码串行执行，线程安全，但活性低。
 - volatile变量锁外双重检测(JDK1.5+)，降低锁竞争。
 - 读写条件分离，锁粒度分级，排序锁。
- CAS (CompareAndSet)
 - 循环设新值，如果旧值变化，则重设，乐观并发。



习惯

- 敲每个点号时，考虑：
 - 会不会出现空指针？
 - 有没有异常抛出？
 - 是不是在热点区域？
 - 在哪个线程执行？
 - 有没有并发锁间隙？
 - 会不会并发修改不可见？





天猫 Tmall.com