## CPU多核缓存架构

|  |
| --- |
|  |
|  |

## 缓存一致性协议：

|  |
| --- |
| 其实缓存一致性协议不止一种，但是目前大多数采用的是MESI协议。 |
| **如果两个cpu同时修改？**  一个指令周期内，会进行裁决(也就是裁决下到底由哪个cpu改)。 |
| **缓存行：**  cpu存储的最小单元。有的是32字节，有的是64字节，也有的是128字节。 |
| **缓存一致性协议失效问题：**   1. 如果变量的存储长度大于一个缓存行，缓存一致性协议会失效(不能一个数据横缓多个存行)，这时候是加总线索； 2. cpu本身并不支持缓存一致性协议(比如早期奔腾系列的cpu)。 |

## 线程

|  |
| --- |
|  |
| **为啥cpu级别分为ring0和ring3？**  安全性问题，最高级别的操作只允许内核空间的线程进行。否则会出问题。 |

### ULT(用户级线程) && KLT(内核级线程)

|  |
| --- |
|  |
| **UTL用户及线程**  优点：可以避免过度创建线程，还能避免大量上下文切换  缺点：一个线程阻塞，全部线程阻塞 |
| **KLT内核级线程**  每个进程中可以创建多个线程，每个线程都可以看作是一个微进程 |
| **Java中用的是ULT还是KLT?**  java1.2之前是ULT,  java1.2之后用的是KLT . |

### JAVA线程与内核线程的关系

|  |
| --- |
|  |

### Java线程的生命状态

|  |
| --- |
|  |
| 为啥要用到并发 |
|  |

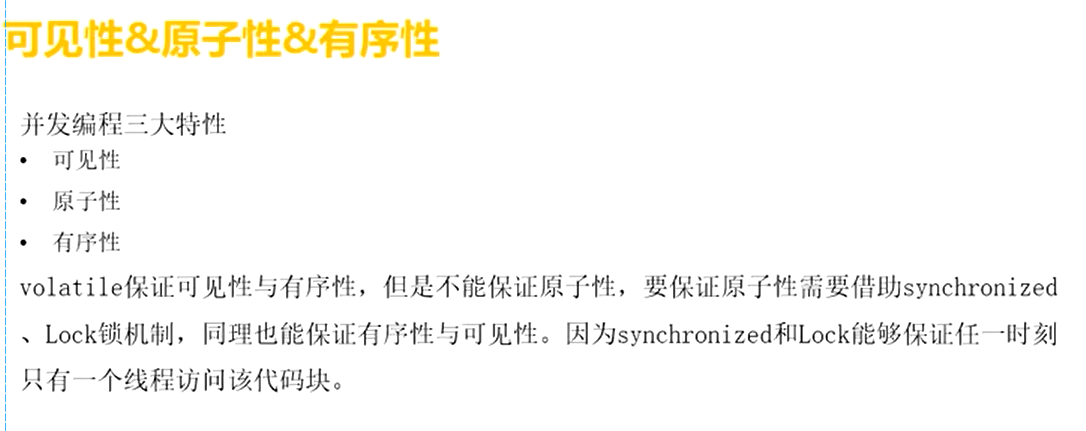
## JMM模型

|  |
| --- |
| **JMM是一组规范，目的是屏蔽不同的操作系统在底层实现的具体规则的不同，同时为了使Java并发编程可以从逻辑上区分开来。**  **jvm进程取申请空间的大部分时候，操作的使逻辑空间，而不是系统空间，**  **逻辑空间使有我们的系统已经划分好的**。 |
|  |
|  |
|  |

|  |
| --- |
|  |
| **示例：**      **原理：** |
| **修改：**    执行结果：    **原因：**没加synchronized的时候，程序一致在空跑，不会让出cpu执行权，也就不会导致线程上下文的切换，那样的话就会一致读取副本中的信息；  但是如果加上同步代码块，该线程有可能会发生短时间线程阻塞，这样的话会导致线程的上下文切换，这时load()方法（线程A会重新读取主内存的initFlag），如果initFlag被改写，那么读取到的将是true，所以输出了这句话。 |
| **再改写（*将刚才的同步代码块去掉*）**:    原因： |
| **八大操作顺序：**  *八大操作虽然要全部执行到，但是并不一定要连续执行*。  *read和load必须是成对出现的，store和write也是要成对出现的*。 |
|  |

## volatile

|  |
| --- |
| 备注：volatile关键字无法保证原子性。 |



原子性：

|  |
| --- |
| *输出结果可能是10000，也可能小于10000*.  **产生这种情况的原因**：  上面的红框和下图： |
|  |

有序性与指令重排：

|  |
| --- |
| **指令重排：**  cpu或者编译器为了保证程序最大的并发性能，会对指令进行优化 |
| **指令重排发生在那个阶段？**   1. 编译期：字节码编译成机器码取执行的时候， 2. 或者在cpu执行汇编指令的时候也会执行指令重排。 |
| cpu或者jit对代码进行了指令重排后，才会数据0，0 |
| **内存屏障：** |

可见性：

|  |
| --- |
|  |

### cas和volatile关键字在大规模并发时的使用会出现什么问题？

|  |
| --- |
| **总线风暴：**  工作内存与主内存大量交互并且时*无效交互*，并且用到了大量的volatile或者cas关键字，会造成其他线程的工作内存中产生大量的无效工作内存变量，，而且会不停的嗅探主存中的状态，多余的交互会占用总线的带宽，导致总线被过分占用，使其他的线程大规模延迟。  也就是说：避免堵车。  解决方案：使用同步。 |
|  |

## 同步机制

### synchronized

|  |
| --- |
| **并发场景下，为啥需要保证资源的原子性**： |
| **隐式锁**：synchronized，使jvm内置锁，不需要手动加索和解锁，jvm会自动加锁和解锁；  **显式锁**：ReentrantLock,实现juc里的Lock，实现使基于AQS实现，需要手动加索和解锁。  由于synchronized不能跨方法加锁和解锁，所以说隐式锁灵活性不如显式锁。 |

#### synchronized加锁机制

|  |
| --- |
| *每个对象在创建后，都会在jvm内部维护一个Monitor对象*。 |
|  |
| **Monitor监视器锁** |

#### 对象内存结构

|  |
| --- |
| ***对象头***： |

#### 面试题

|  |
| --- |
| ***实例对象一定存在堆区吗？***  不一定，不过实例对象没有线程逃逸行为的话，那么他就不会在堆中开辟空间，会有大部分对象直接在当前线程栈上开辟空间。  ***实例对象在内存中储存在哪里？***   1. 存储在堆区时：实例对象内存存在堆区，实例的引用存在栈上，实例的元数据class存在方法区或者元空间； 2. 如果实例对象不存储在堆区时：会有大部分实例对象存放在当前线程栈。 |

#### 锁的粗化和消除

|  |
| --- |
| package tencent.thread;  import org.junit.Test;  */\*\*  \* @description: 锁的粗化和消除  \* @date: 2020/5/30 15:09  \* @author: lizhenhong  \*/* public class LockCuhuaAndXiaochu {   private static Object *object*;    */\*\*  \* 粗化  \*/* @Test  public void testCuHua() {  StringBuffer sb = new StringBuffer(); *// synchronized (object) { // sb.append("1"); // } // synchronized (object) { // sb.append("2"); // } // synchronized (object) { // sb.append("3"); // } // synchronized (object) { // sb.append("4"); // }  //上面的每一步操作实际上都会加一个锁，但是这些锁经过jvm锁的粗化，会等效于下列的形式* synchronized (*object*) {  sb.append("1");  sb.append("2");  sb.append("3");  sb.append("4");  }  }   */\*\*  \* 消除  \*/* public void testXiaoChu() {  *//jvm的优化，jvm不会对同步块进行加锁，因为访问该方法的其他线程不能看到该线程的信息* synchronized (new Object()) {  *//伪代码：很多逻辑  //jvm是否加锁  //jvm会进行逃逸分析* }  }  } |

#### java锁体系

|  |
| --- |
|  |

#### 使用和原理：

#### 如何跨方法加索和解锁

|  |
| --- |
| 这种方法不推荐使用，因为这个方法会越过jvm直接去操作底层。 |

#### JVM内置锁优化升级过程

|  |
| --- |
| 由无锁向重量级升级，不能反方向降级  而且，锁的升级过程不可逆    开启偏向锁，性能大约能提升10%左右(jdk1.7 && 1.8时默认开启的)。 |
| 偏向锁：    轻量级锁：    自旋锁：  重量级锁： |
| ***锁升级的场景：***  *无锁->偏向锁*：单独的线程访问的场景；  *偏向锁->轻量级锁*：竞争不激烈，线程间交替执行(线程中的逻辑相对比较简单，执行时间段)；    *轻量级锁->重量级锁*：存在激烈的竞争，并且带啊中逻辑较复杂。 |

#### 锁的升级优化过程

|  |
| --- |
| 这个图片太复杂，可能看不清，路径：D:\work\work\笔记\腾讯课堂录屏\java-synchronized锁的升级过程.jpg  java-synchronized锁的升级过程 |
| 拆分一下： |
| 1. 无锁->轻量级锁： |
| 1. 无锁->重量级锁： |

#### 公平锁和非公平锁

|  |
| --- |
|  |

### ReentrantLock(AQS实现的锁机制)

|  |
| --- |
|  |

### AbstractQueuedSynchronizer(AQS)

|  |
| --- |
|  |
|  |
|  |
|  |

## Tools

|  |
| --- |
|  |
|  |

### Semaphhore

|  |
| --- |
| 可对比限流  下图中红框的意义：同时允许访问后端服务的线程个数。  new对象的时候，默认创建非公平锁      输出结果：同一时间只有两个线程进来。 |
|  |

### CountDownLatch

|  |
| --- |
|  |
|  |

### CyclicBarrier

|  |
| --- |
| 和CountDownLatch差不多，只是没有主次之分，相当于团战。  必须等所有线程都到达某个位置，才能一起出发，否则全部一直等待。 |
| 和CountDownLatch,还有一个区别，就是线程是可以重复的。 |

### Exchanger线程间数据的交换(知道就行了)

|  |
| --- |
|  |
|  |

### Executors

|  |
| --- |
| 是一个工具类，主要是做线程池的初始化 |

## 原子操作：

|  |
| --- |
|  |
|  |
|  |

Atomic

|  |
| --- |
|  |
|  |

Unsafe

|  |
| --- |
|  |
|  |

### Atomic

|  |
| --- |
| package tencent.thread;  import java.util.concurrent.atomic.AtomicInteger;  */\*\*  \* @description:  \* @date: 2020/6/5 18:41  \* @author: lizhenhong  \*/* public class AtomicIntergerTest {   static AtomicInteger *atomicInteger* = new AtomicInteger(1); *//括号里面表示从几开始操作* public static void main(String[] args) throws InterruptedException {  for (int i = 0; i < 10; i++) {  new Thread(new Runnable() {  @Override  public void run() {  *atomicInteger*.incrementAndGet();  }  }).start();  }  *//防止上面的线程没有跑完就输出，让他睡眠一秒* Thread.*sleep*(1000);  System.*out*.println("=====>从1开始自加10次的数值：" + *atomicInteger*.get()); *//=====>自加10次的数值：11  /\*\*  \*   \*/* }   } |
| ***Atomic的A.B.A问题*** |
| 危害： |
| ***解决：***加个版本号 |
|  |

### Unsafe

|  |
| --- |
|  |
| ***Unsafe功能***：最好只弄CAS && 内存屏障&& 线程调度，其他的不要去玩，容易搞出事情。 |