## 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 |
|  |

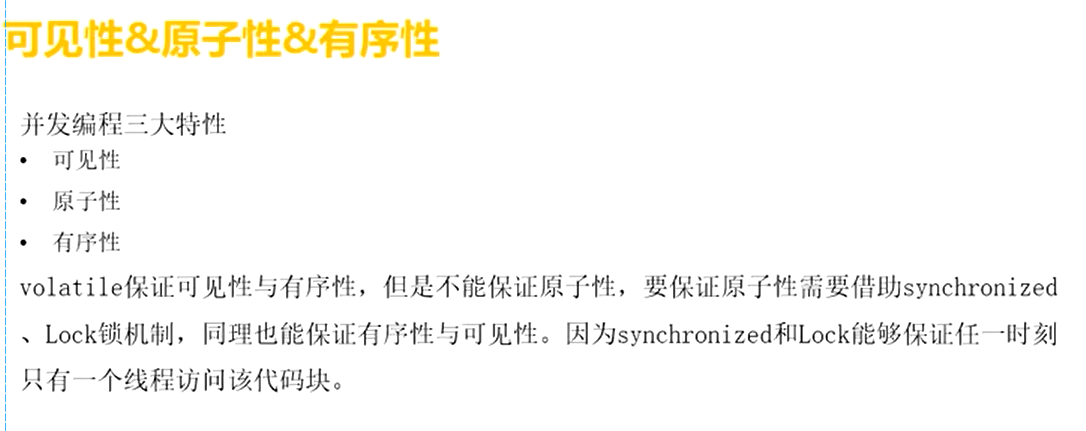
## JMM模型

|  |
| --- |
| **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

|  |
| --- |
|  |



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：

|  |
| --- |
|  |

可见性：

|  |
| --- |
|  |