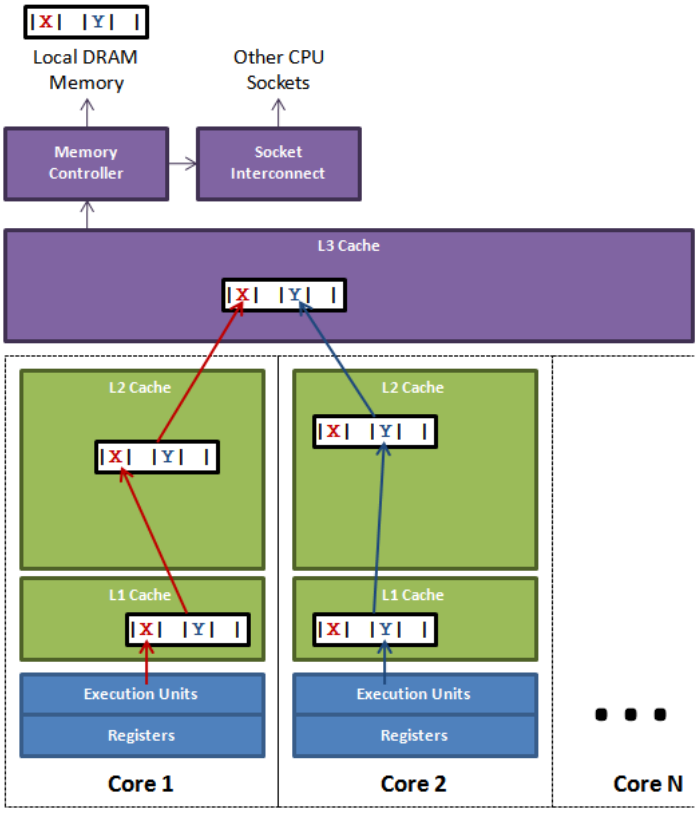
简述

缓存的存储方式，是以缓存行(Cache Line)为单位的。一般缓存行的大小是64字节。这意味着，小于64字节的变量，是有可能存在于同一条缓存行32字节，那么他们有可能会存在于一条缓存行上，而每个线程都要去竞争缓存行的所有权来更新变量

定义：伪共享，就是多个线程同时修改共享在同一个缓存行里的独立变量，无意中影响了性能

伪共享是怎么发生的



核心1上的线程要操作变量X，发现X变量长度不足64字节，X变量旁边有一个变量Y，两个变量各自32字节，加起来正好64字节。虽然现在用到了变量X，但是Y以后可能也会用的。就一次将两个变量都存入了同一个缓存行。

而核心2上的线程需要操作变量Y，发现旁边有X，同样最终把X、Y存入了同一个缓存行。

当核心1上的线程想更新X，而核心2上的线程想更新Y，而各自的缓存行中都有X变量和Y变量。

如果核心1获得所缓存行它所在缓存行的所有权，核心2所在的缓存行Y将处于失效状态，核心2就要去L3 Cache缓存拿数据。

同样如果核心2获得了所有权，核心1的缓存行X失效，重新去L3拿数据

这种情况，就像多个线程同事竞争锁的所有权一样。如果互相竞争的核心位于不同的插槽，就要额外横跨插槽连接，问题可能更加严重

如何避免伪共享

在JDK8之前一般都是通过字节填充的方式避免伪共享问题。

```
public final static class FilledLong{
    public volatile long value = 0L;
    public long p1,p2,p3,p4,p5,p6;
}
```

l例如缓存行为64字节，那么在FilledLong类里面填充了6个long类型的变量，每个long类型占用8个字节，加上 value也是long类型，加起来
一个类对象，而类对象的字节码的对象头占用8字节，加起来正好64字节，可以放入同一个缓存行

而在JDK8中提供了sun.misc.Contended注解，用来解决伪共享问题

```
@sun.misc.Contended
public final static class FilledLong{
    public volatile long value = 0L;
}
```

代码验证

```
public class FalseShareTest implements Runnable {
    public static int NUM_THREADS = 4;
    public final static long ITERATIONS = 500L * 1000L;
    private final int arrayIndex;
    private static VolatileLong[] longs;
    public static long SUM_TIME = 0L;
    public FalseShareTest(final int arrayIndex) {
        this.arrayIndex = arrayIndex;
    }
     public static void main(final String[] args) throws Exception {
         for(int j=0; j<10; j++){
             System.out.println(j);
             if (args.length == 1) {
                 NUM_THREADS = Integer.parseInt(args[0]);
             }
             longs = new VolatileLong[NUM_THREADS];
             for (int i = 0; i < longs.length; i++) {
                 longs[i] = new VolatileLong();
             }
             final long start = System.nanoTime();
             runTest();
             final long end = System.nanoTime();
             SUM_TIME += end - start;
         }
         System.out.println("平均耗时: "+SUM_TIME/10/1000);
```

```
    }
    private static void runTest() throws InterruptedException {
        Thread[] threads = new Thread[NUM_THREADS];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new FalseShareTest(i));
        }
        for (Thread t : threads) {
            t.start();
        }
        for (Thread t : threads) {
            t.join();
        }
    }
    @Override
    public void run() {
        long i = ITERATIONS + 1;
        while (0 != --i) {
            longs[arrayIndex].value = i;
        }
    }
    public final static class VolatileLong {
        public volatile long value = 0L;
        public long p1, p2, p3, p4, p5, p6;     //屏蔽此行
    }
}
```

屏蔽代码 public long p1, p2, p3, p4, p5, p6; 和 不屏蔽的区别很明晰，不屏蔽的速度更快一些