**Atomic包**

# 什么是原子类？有什么作用？

## 不可分割

我们来说在化学反应中一般来讲，原子是不可分割的，我们分子啊，比如说水，经过电解呢，会生成氧气和氢气，说明分子是比较容易分解的，这就会牵扯到化学方程式，但是我们的化学方程式一般都是牵扯到分子级别的，所以原子的一个特点就是不太容易被分割。

在编程领域的原子操作它是不可中断的，即便是在多线程的情况下，也可以保证。

## 一个操作是不可中断的，即便是多线程的情况下也可以保证。

那么我们说你原子下面好像不是不可中断的，好像也可以分割嘛。对不对，由中子，质子，下面还有夸克，还有量子，到底什么是不可分割的？我这边可以确定的就是我和我的做一刻钟也不能分割。

## 3、java.util.concurrent.atomic

而在我们的java中啊，和原子相关的都放在了我们的java.util.concurrent.atomic这个包里，这个包里面都是具有原子特性的类。

## 原子类的作用和锁类似，是为了保证在并发情况下线程安全，不过原子类相比于锁，有一定的优势：

### 粒度更细：原子变量可以把竞争范围缩小到变量级别，这是我们可以获得的最细粒度的情况了，通常锁的粒度都要大于原子变量的粒度。

因为我们的原子变量可以把竞争范围缩小到变量级别，比如说我们使用一个AtomicInteger，那么它本身是一个变量，合适我们可以获取到最细的粒度了，而我们使用锁通常会锁住好几行代码，甚至锁住某个方法，那这种情况下他的粒度都要大于我们原子变量的粒度，那么对锁而言，其实对于粒度呢，一般是越小越好，这样可以提高我们的并发效率，避免锁住无用代码，

### 效率更高：通常，使用原子类的效率会比使用锁的效率更高，除了高度竞争的情况。

通常情况下原子类的效率是比锁高得，但是呢，他也有例外，原子类在高度竞争的情况下，那他的效率是不如我们的锁的，所以呢，要根据具体的场景去进行选择，到后面我们会讲原子类在高度竞争的情况下他的效率会降低。在这里我们主要需要知道的就是原子类就是有这两大特点。

# 6类原子类纵览

## Atomic\*基本类型原子类

1. AtomicInteger
2. AtomicLong
3. AtomicBool

## Atomic\*Array数组类型原子类

1. AtomicIntegerArray
2. AtomicLongArray
3. AtomicReferenceArray

## 3、Atomic\*Reference引用类型原子类

1、AtomicReference

2、AtomicStampedReference

3、AtomicMarkableReference

## 4、Atomic\*FieldUpdater升级类型原子类

1、AtomicIntegerfiledupdater

2、AtomicLongFiledUpdater

3、AtomicReferenceFiledUpdater

## 5、Adder累加器

1、LongAdder

2、DoubleAdder

## 6、Accemulator累加器

1、LongAccumulator

2、DoubleAccumulator

# Atomic\*基本类型原子类，以AtomicInteger为例

## AtomicInteger：整形原子类

### 1、概念

它是对int或者是说对Integer的一种封装，封装好了以后就提供了原子的更新和访问操作，他背后呢是基于CAS技术。在这里我们可以把它理解为有一系列指令，这个指令可以在不被打断的情况下，去更新某个值，并且在更新的时候呢会去检查，和我预期的是否一致。

### 常用方法。

1. public final int get() //获取当前的值
2. public final int getAndSet(int newValue) //获取当前的值，并设置新的值
3. public final int getAndIncrement() //获取当前的值，并自增

increment的意思是增加，他这里就体现了它原子性的特点，这个increment我们如果不用原子类来写的话，比如说我们写成a++的形式，那这a++本身不是一个原子操作，它内部会分为三个步骤：先读取，然后+，最后呢再写回去，那这种情况下是可能被打断的。等会我们会去演示，但是我们在使用原子类的getAndIncrement()之后，他就会把整个操作变成原子的，在此期间谁都打断不了，就可以保证线程安全，

1. public final int getAndDecrement() //获取当前值，并自减。
2. Public final int getAndAdd(int delta) //获取当前的值，并加上预期的值。
3. Boolean compareAndSet(int expect,int update) //如果输入的数值等于预期值，则以原子方式将该值设置为输入值(update)

## AtomicLong：长整型原子类

## AtomicBoolean：布尔型原子类

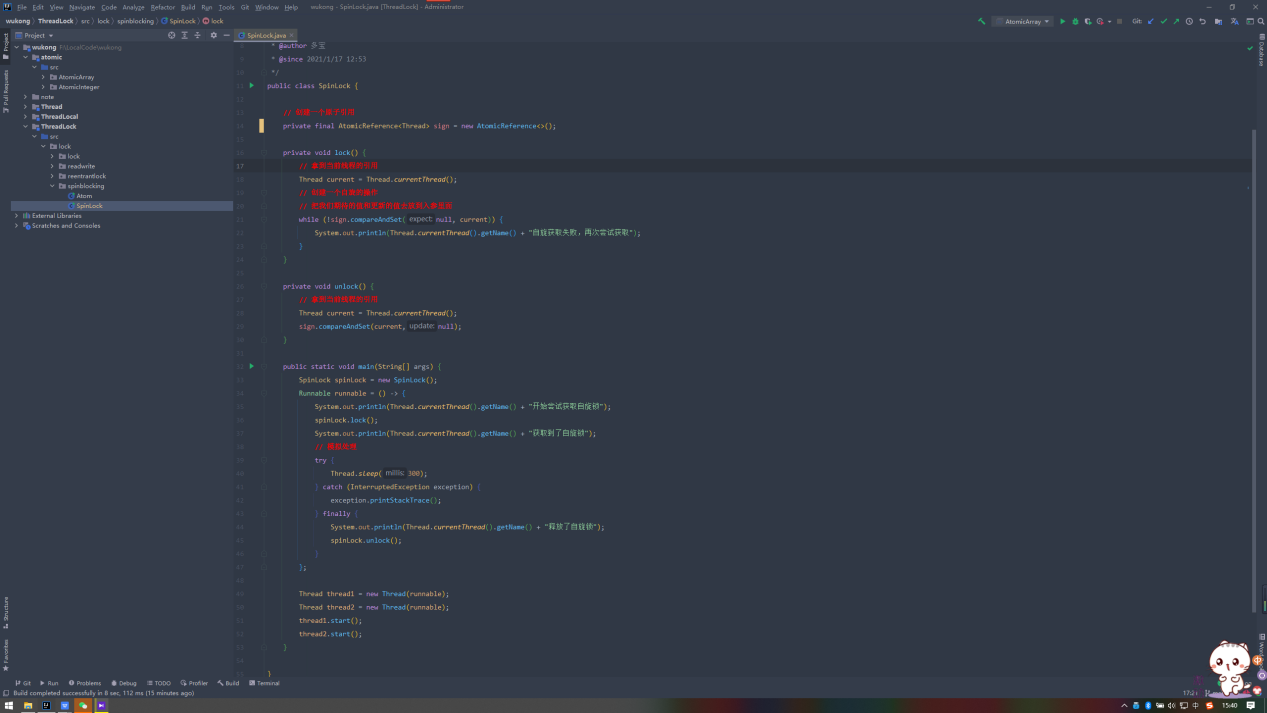
# Atomic\*Array数组类型原子类

# Atomic\*Reference引用类型原子类

## 1、作用

AtomicReference:AtomicReference类的作：，和AtomicInteger并没有本质区别，AtomicInteger可以让一个整数保证了原子性，而AtomicReference可以让一个对象保证原子性，当然，AtomicReference的功能明显比AtomicInteger强，因为一个对象里可以包含很多属性，用法和AtomincInteger类似。

## 2、使用(以自旋锁为例)



在自旋锁中，加锁的时候，我们将线程作为原子引用类型的泛型，我们在使用原子引用类型的compareAndSet()的时候，我们要判断当前原子引用的线程是不是空，如果是空我们将线程设置为当前线程。

其实原子引用类型就是针对对象做原子操作，而并非是单独的针对基本类型的封装。

# 把普通变量升级为原子类：用AtomicIntegerFieldUpdater升级原有变量。

## AtomicIntegerFieldUpdater对普通变量进行升级

### 思想

AtomicIntegerFieldUpdater还是比较厉害的，我们用了它之后呢，可以对普通的变量进行升级。比如说我们之前已经是声明过一个对象，这个对象里面啊有一个int这个类型的一个数值，那这个int明显他是基本类型，所以它是不具备原子性的，那么在这种情况下，木已成舟，因为类已经被定义好了，那么我们也不方便去改变这个类，此时又想对他这里面的一个变量进行原子操作，怎么办呢？在这种场景下啊，我们就可以利用我们的AtomicIntegerFieldUpdater，相当于使用它来辅助我们自己的一个普通的对象，让它拥有一个原子的能力。

那你会说啊，哎，你这样是不是多此一举？你既然知道这里面有一个变量会被多线程访问，而且呢，还有可能是并发+的，那你一开始就声明成原子的不好吗？你一开始就把他生命成一个原子的AtomicInteger，那这样一来，你就不需要费时费力去对它进行升级了，这种情况呢也是对的。如果我们在某一个类中某一个变量经常被并发的读写，而我们希望他是个原子的话，可以在最开始就声明为是一个原子类型。

但是我们的AtomicIntegerFieldUpdater他有自己的使用场景。

第一个适用场景就是说：我们如果一个类不是我们自己编写，我们没有权利修改它，那么我们可以去适用AtomicIntegerFieldUpdater。

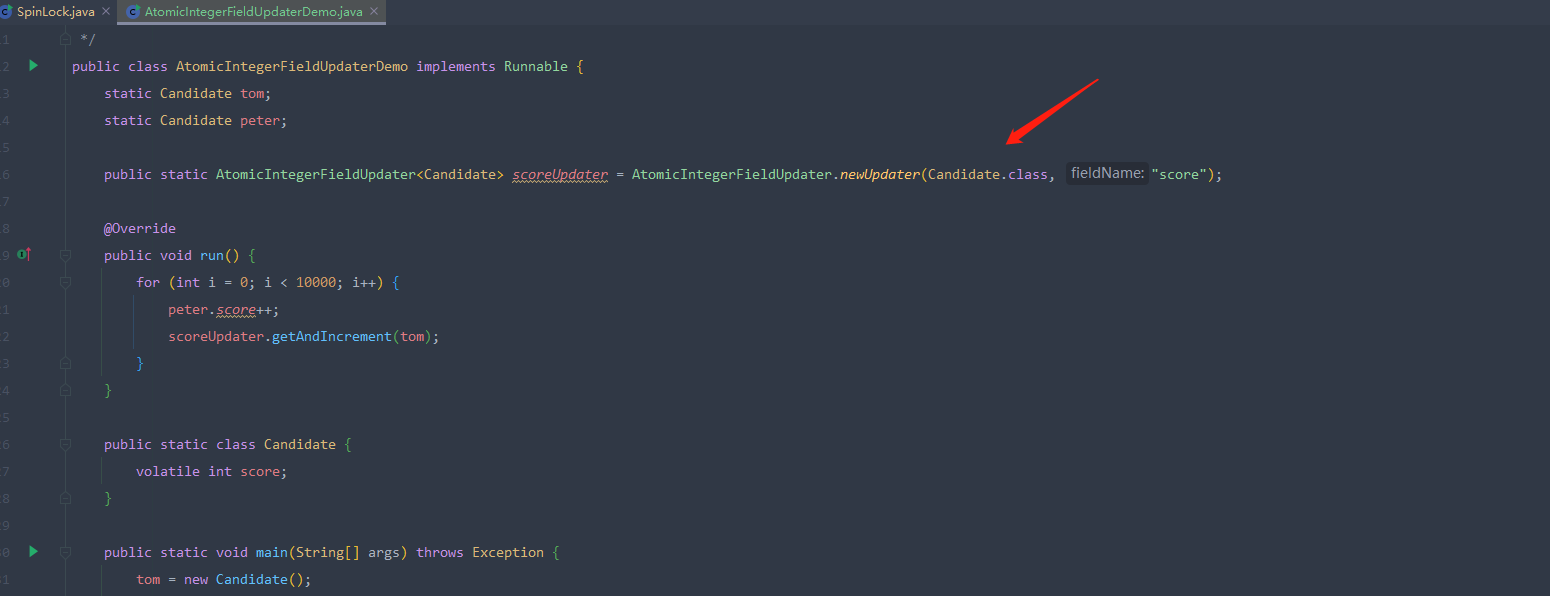
第二个适用场景就是说可能我们偶尔需要原子的get-set操作。这种情况不是特别多，那么如果我们的业务场景是，比如说每天只有是在0点的时候，才会对它进行一个原子的更新。并且这个时候还存在并发问题。我们必须要保持原子性的话，我们就必须对它进行升级。但是除了晚上0点的那一个时刻之外，其他时刻并不存在线程并发访问的一个情况，所以在其他时刻我们只需要把它当做是一个普通变量就可以了。

那么如果是说在这种情况下，也就是说我们大部分情况下希望它是一个普通的类型，只有很少数的情况下才需要这种普通类型具备原子性，那么我们就没有必要在这个类中去嵌入一个额外的原子对象，因为实际上我们生成原子对象，因为我们实际上生成原子对象并且对它进行操作的话，也是有资源消耗的，一个AtomicInteger，它所带来的消耗远比我们的基本类型要大，包括内存、包括操作时候的消耗，所以如果我们这样的类呢有很多个，比如说成千上万个，那么这个时候内存的占用就是一个值得考虑的问题，所以在这种情况下我们就可以把我们类中的数值，声明为一个普通类型，比如说int，那么这样的资源占用是最小的，那么我们需要原子操作的时候呢，我们再去利用AtomicIntegerFieldUpdater，去对它进行升级，尤其是当数量庞大之后，这个节约的资源就是可观的。这样一来就优化了性能。

### 用法，代码演示

### AtomicIntegerFieldUpdater注意点

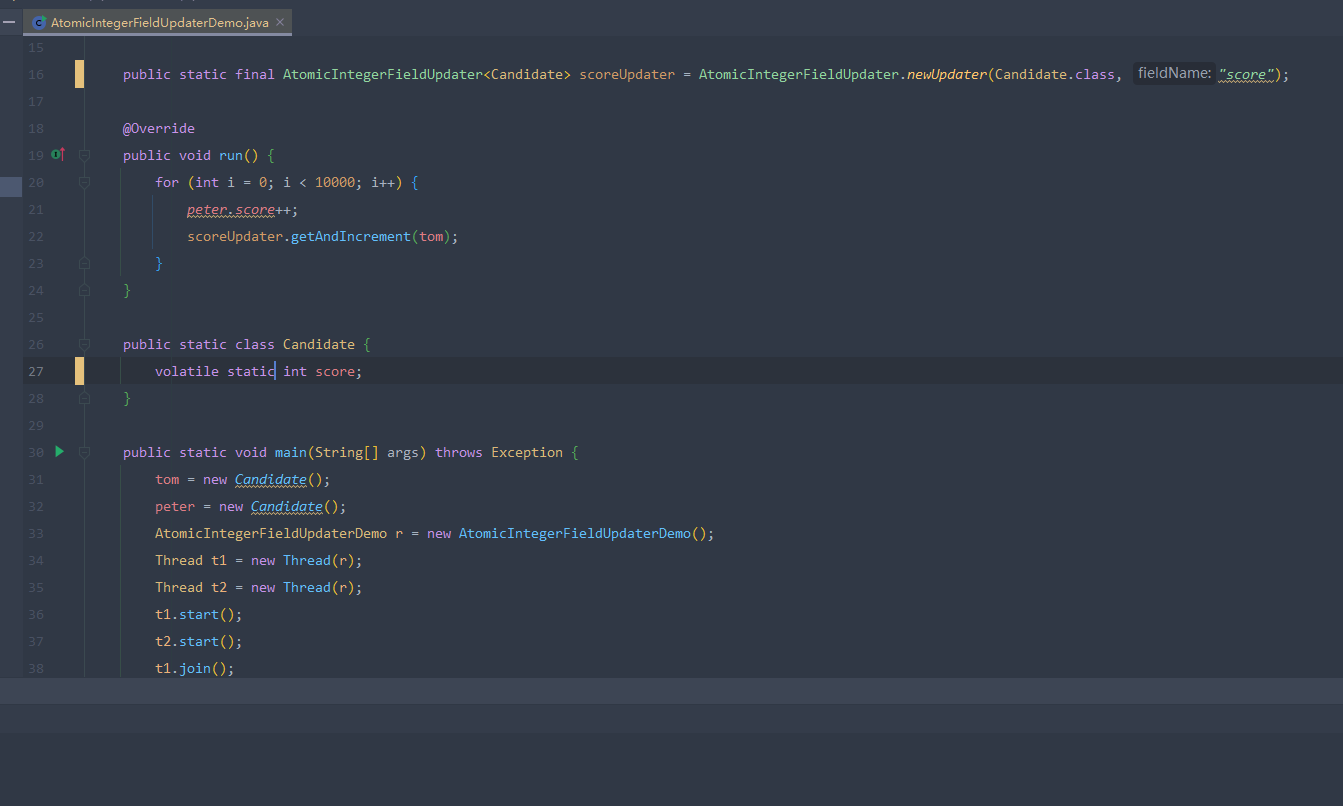
1. 可见范围



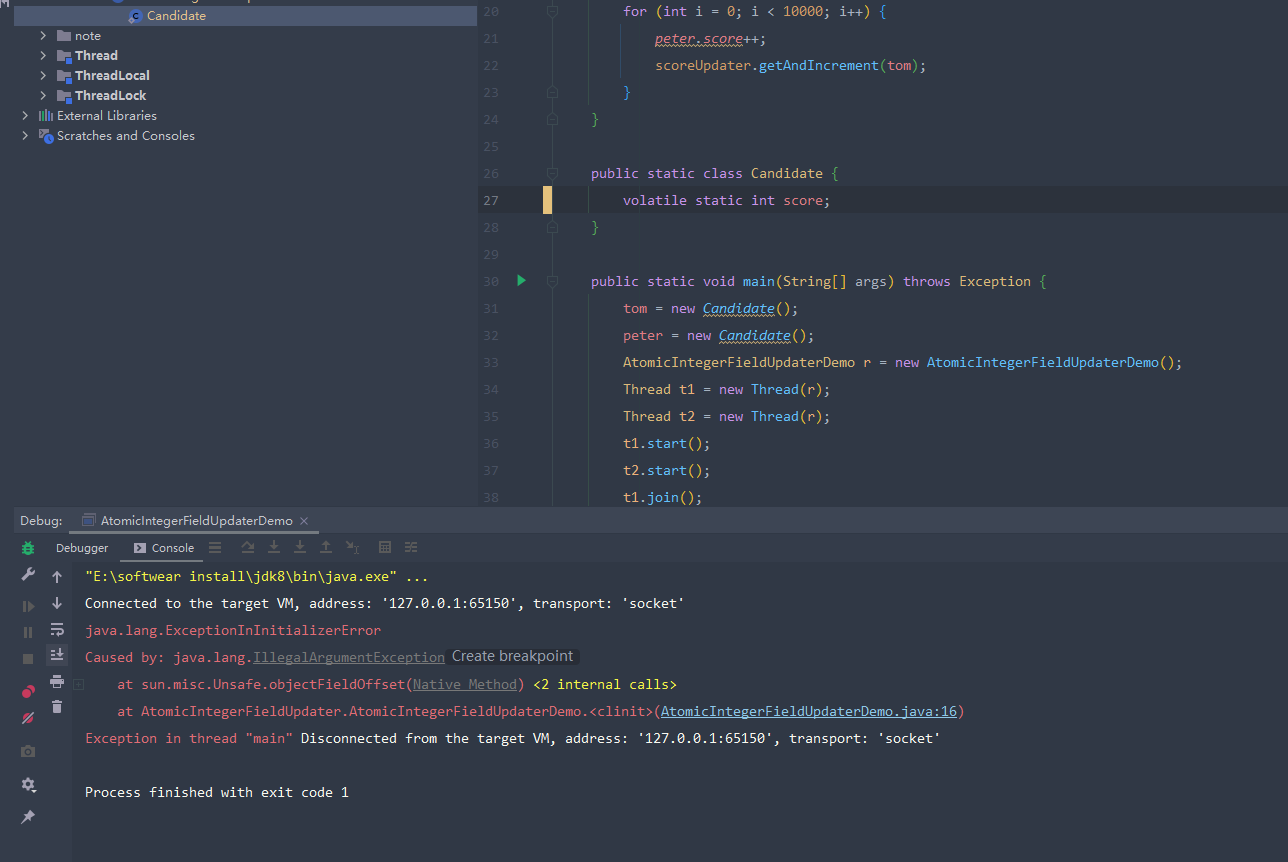
他所能修改的数值是有一个可见范围的，我们刚才所看到的这块代码AtomicIntegerFieldUpdater.newUpdater(Candidate.class, "score");在创建原子变量升级的时间我们传入了一个类名，还传入了一个变量名，其实呢已经是非常明显了，我们之前什么时候会遇见这种情况？又要传入类名，又要传入变量名？就是在反射的时候，如果是熟悉反射的话，折合反射的用法非常相似。

所以呢我们这边的FieldUpdater背后的原理啊也是利用了反射，那么如果这个变量是不可见的，比如说是private，那样就是不行的。  
2.不支持被static修饰的变量

我们将泛型的类里面变量用static修饰运行一下我们的程序



运行程序



他报错了，他报的错是Caused by: java.lang.IllegalArgumentException。

# Adder累加器

## 是java8引入的，相对是比较新的一个类

## 高并发下LongAdder比AtomicLong效率高，不过本质是空间换时间。

AtomicLong的缺点是效率低，那么我们学完这个LongAdder之后，就可以解决一些性能上的问题，比如说并发情况特别高的时候，我们可以用我们的LongAdder，由于我们目前啊，之前做过一个调研，就是说在生产环境用的java版本各个比例是多少？那么在19年的时候呢，java8的版本依然有超过70%上的公司都在用，所以说其实我们大部分的版本呢是java8，那这个类啊也是java8索引入的，很多比如说有三年五年经验的程序员他们可能对于java8中很多新的工具类啊，新的特性啊，都不一定了解的那么清楚，比如说你问他LongAdder，比如说你问他Lambda表达式怎么写的特别漂亮之类的问题呢，如果他不是一直保持在持续学习的情况下，那他可能就不知道，所有这正好是我们可以发挥的空间，所以呢我们就来讲一下LongAdder，他有一个什么好处。

## 在竞争激烈的时候，LongAdder把不同线程对应到不同的cell上进行修改，降低了冲突的概率，是多段锁的理念，提高了并发性。

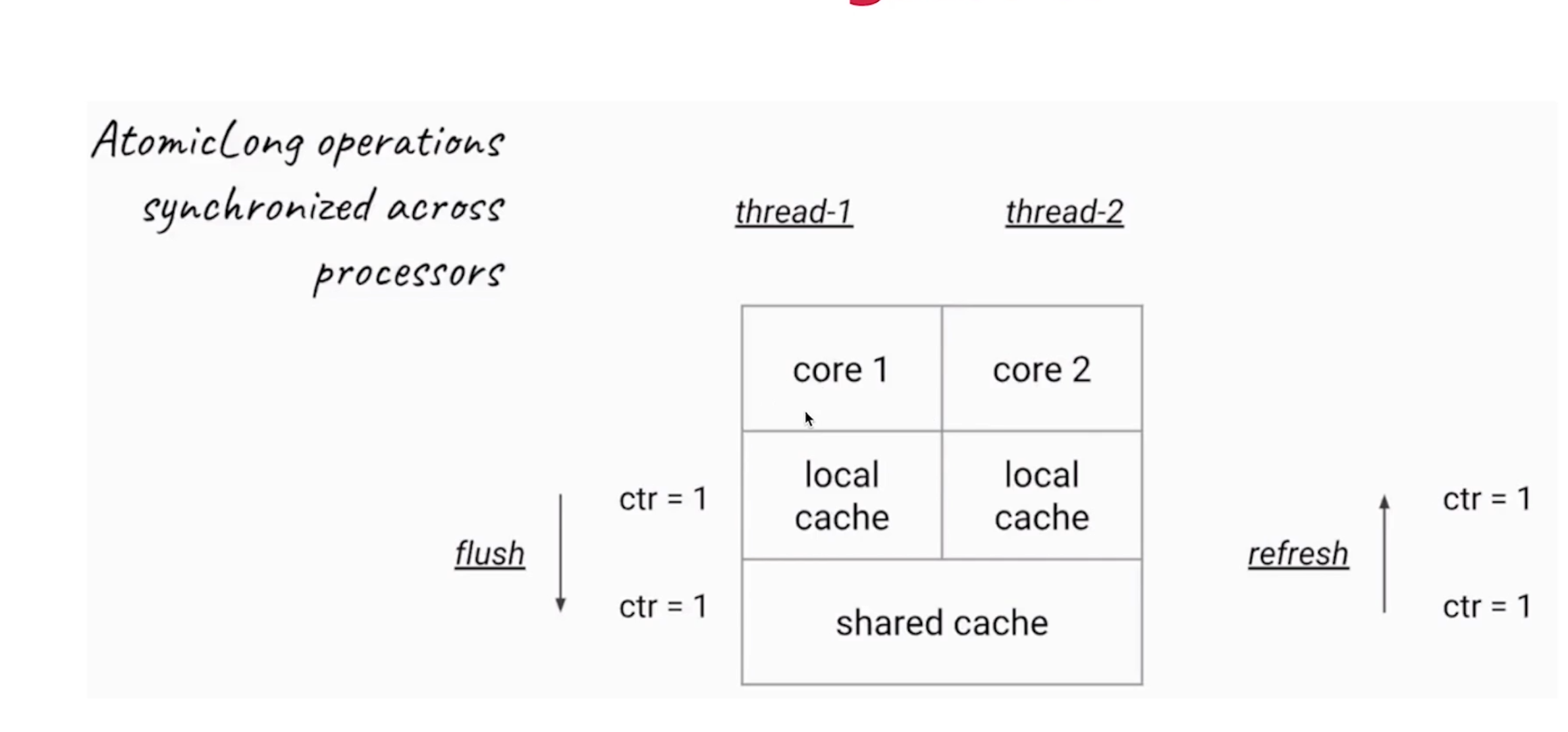
## 代码演示

### 演示多线程情况下AtomicLong的性能，有16个线程对同一个AtomicLong累加

在多线程情况下，我们的AtomicLong的性能不如我们的LongAdder，我刚才说，你介绍它为什么要介绍它？他最大的好处就是说性能要比我们的之前有提高，所以我们会用代码来让小伙伴们亲眼看到在执行同样的一个任务，他究竟性能能相差多少？

### 由于竞争激烈，每一次加法，都要flesh和reflesh，导致很消耗资源。

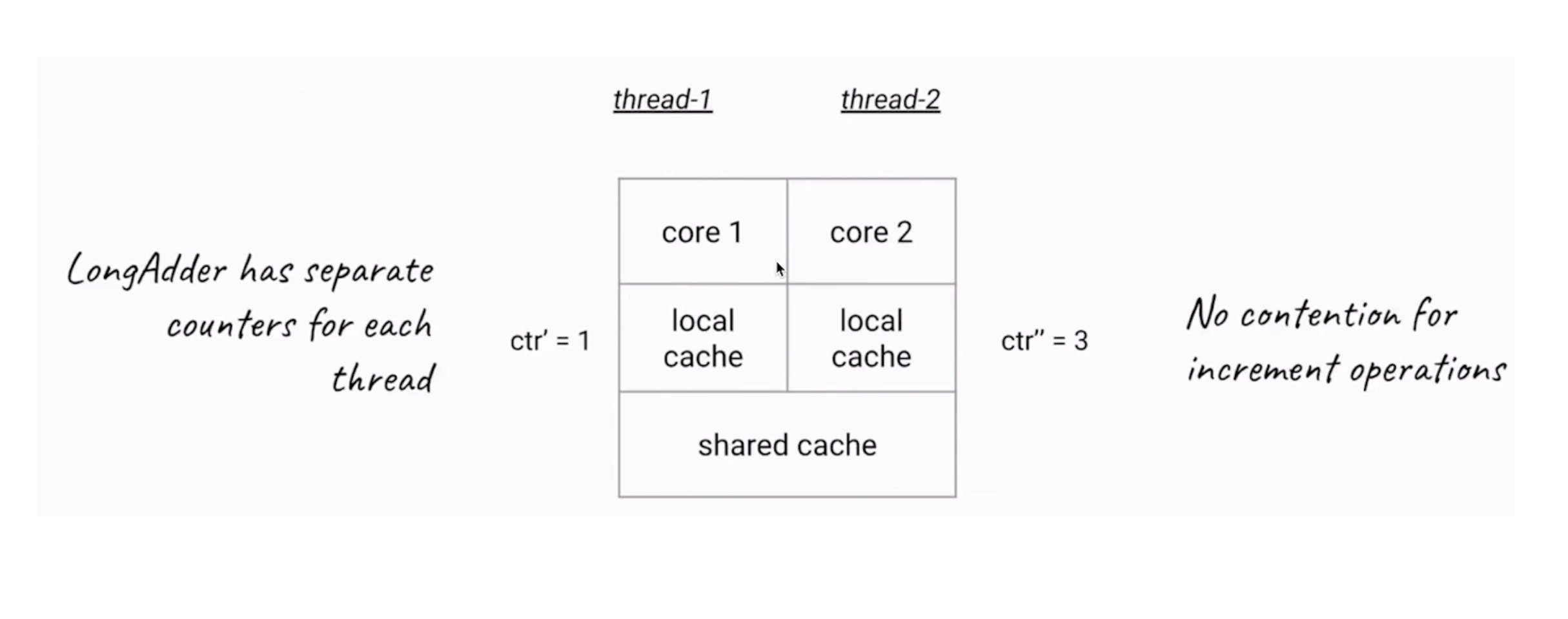
1. AtomicLong为什么耗时那么多？



如果这是一个AtomicLong，他的弊端就在于你在第一个核心中，也就是在core1，也就是cpu的多核嘛，你在第一个核中，你给它修改了之后，他为了保证每一个核之间他们的数据一致性，就会把它刷到主内存中，或者叫做共享内存，也就是图中的shared cache，然后呢再去refresh到每一个线程所独有的工作内存中，这个实际上是涉及到java内存模型，关于这一块啊，包括关于工作内存，主内存，以及happens-before啊，欧这是valitail啊，包括可见性，原子性， 桶排序这一系列的内容啊都和我们的jmm有关，jmm实际上是我们学习并发的一个必不可少的重要基础，它是对我们整个并发打下了基础，有了jmm之后，我们理解起来会非常容易，在这里呢我们对jmm涉及到这一部分的回顾，这里就是说，这边thread1和thread2是两个线程，他们抛在不同的CPU上，比如说是core1和core2，他们之间是无法直接通信的，比如我在这里修改了这个ctr，相对于第二个线程而言他看不到，他们必须通过下面的共享内存(主内存(shared cache))，通过一个中转，也就是说线程1把它写回到主内存中，而线程2在把它读回去，这样他们才能达到一个通讯，相当于是间接通讯，那么通过这种flush和refresh操作之后啊，线程2才能拿到，而我们的AtomicLong呢就会一直做这样的事情，相当于啊他一直保持不同的核心以及线程之间的一个数据同步。

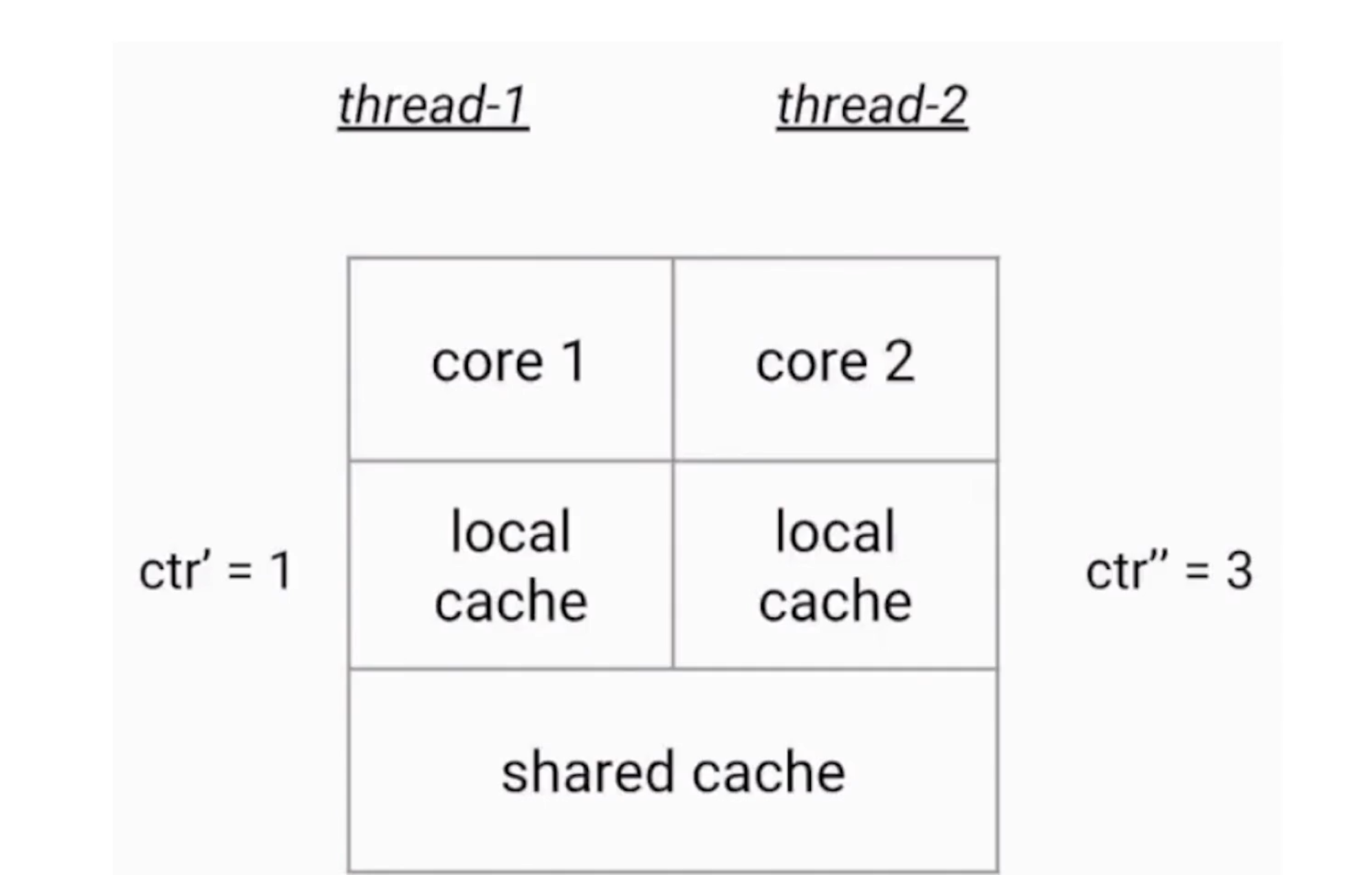
2、为什么LongAdder效率那么高？

但是我们的LongAdder不一样，我们来看下面这张图。



它是怎么样啊，他不需要保证线程之间她们数据始终一致，你看，比如说对于线程1而言，他在这边对于ctr最开始是0，但是它加了1呢，就变成了1，所以这个叫ctr’，而这个’也没有同步到线程2去，因为线程2拿的是’’，这’’和’是两个独立的变量，他们不仅名字不一样，连值都不一样，你看在线程1里面ctr’是1，他只是加了一次，线程2呢可能跑的速度快一点，加了三次变成了3，所以在这个时候，他们不会去通过下面的共享内存(shared cache)进行一个中转，他们就独立的在自己的工作内存中，不停地去进行+的操作。

1. 在内部呢，这个LongAdder的实现原理和刚才的AtomicLong是有不同的，刚才的AtomicLong的实现原理是，每一次加法都需要做同步，所以在高并发的时候，导致冲突比较多，也就是降低了效率。
2. 而此时的LongAdder，每个线程会有自己的一个计数器，仅用来在自己线程内计数，这样一来就不会和其它线程的计数器干扰。
3. 如图中所示，第一个线程的计数器数值，也就是ctr’，为1的时候，可能线程2的计数器ctr’’的数值已经是3了，他们之间并不存在竞争关系，所以在加和的过程中，根本不需要同步机制，也不需要刚才的fresh和refresh。这里也没有一个公共的counter来给所有线程统一计数。
4. 那么他需要计数怎么办呢?来看下面一张图的LongAdder计数实现



这是一个最后的汇总的这个阶段，能为你说你一开始啊，两个线程你加你的，我加我的，可是最终你总是要把所有的结果给汇总，所以这个汇总的阶段是非常重要的。

1. LongAdder引入了分段累加的概念，背部有一个base变量和一个Cell[]数组共同参与技数：

这个分段累加呢就是把累计分散到不同的线程中去，但是啊他也不是完完全全的分段，它内部呢有一个base变量，和一个Cell数组，

1. base变量：竞争不激烈，直接累加到该变量上。
2. Cell[]数组：竞争激烈，各个线程分散累加到自己的槽Cell[i]中

这个Cell[]就是我们所说的每一个核心就是这个数组的一个成员，那么每一个线程呢都会分散累加到自己所对应的那个槽中去。他在对应的时候啊是计算了hash值，通过这个值呢，给不同的线程去分配到不同的Cell，每一个Cell都是一个独立的计数器，这样一来呢就不会和其他线程发生干扰了，Cell之间也不存在任何的竞争关系，所以在自加的过程中就大大降低了会签所说的flush和refresh这样的操作，发生冲突的概率也同时见笑了，这就是LongAdder吞吐量比AtomicLong大的原因，其本质呢就是用空间换时间，我们的空间啊确实是用的多了一些，因为我们用了一个数组，每个线程内部都会独立的去存储一份，但是实际上啊，虽然多加了这一点点的空间，却节省了大量的时间，这是一种很好的取舍。

1. sum()源码分析。

public long sum(){

Cell[] as = cells; Cell a;

Long sum = base;

If(as != null){

for(int i = 0; i < as.length; i++){

If((a = as[i]) != null)

Sum += a.value;

}

}

return sum;

}

首先呢会有一个Cell数组，就叫做as，然后呢还会有一个base，这个base就是我们刚才所说的在竞争不激烈的情况下他会利用这个base，所以最终要加和的话实际上你要把两部分都加起来，先去加base，再去加每一个的cell。当cell不为空的时候，他会去进行遍历，我们先看下面为空的时候，当cell这个数组为空的时候，说明都没用到Cell，只用到了base，所以呢直接返回这个sun就可以了，sum的值呢就是base，当这个数组不为空，那么这个时候我们就会用一个for循环来比那里这个数组，然后呢我们用sum+=这样的一个方式把它逐个的给累加起来，最终呢就形成了base加上这个数组里面每一个值得求和。

不知道在这里小伙伴能不能想到一个注意点，因为可以看出在这个sum过程中是没有加锁的，那是我们的每一个Cell其实它在任何时候都有可能被更改，所以它在求和的过程中由于没有进行加锁，那么可能啊会发生加数组加到一般的时候这个数组前面的那些又发生新的变化了，可能又加了1，可是呢，由于这个位置的值已经被我们遍历过了，已经加过去了，所以这一次的变化呢就没办法体现在最终求和中，这也是我们的LongAdder的一个特点。

它的特点就是说，他最终这个求和不是说十分十分的精确，因为他在求和的过程中，也有可能内部的元素发生变动。

## AtomicLong和LongAdder的应用场景

### 在低争用下，AtomicLong和LongAdder这两个类具有相似的特征。但是在竞争激烈的情况下，LongAdder的预期吞吐量要高得多，但是消耗更多的空间。

### LongAdder是和的场景是统计求和技术的场景，而且LongAdder基本只提供了add方法，而AtomicLong还具有cas方法。

# Accumulator累加器

## Accumulator和Adder非常相似，Accumulator就是一个更通用版本的Adder。

## 代码演示，用法。

## 使用场景

### 适合于需要大量计算，并且需要并行计算场景。

如果不需要并行计算，我们使用for循环确实是可以解决问题，那是我们用了它之后，可以用多核同时进行计算大大提高我们计算效率。

### 计算的顺序不能成为瓶颈

也就是说我们对顺序不能有要求，线程1可能在线程5之后执行，也可能在它之前执行，而不影响最终结果，只有满足这个条件我们才能用。因为我们使用这个工具的时间不要求顺序执行，他只是作为计算嘛，并行执行时没问题的，因为我们数学上的交换律。

但是如果我们要求顺序进行执行的话，我们就不能使用这个工具类了。

# 总结

## 什么是原子类？有什么作用？

我们说到院子类就可以通过化学方程式去进行联想，因为在化学反应中，通常原子是不被拆分的，所以原子类他的作用也正是如此，他可以保证一些操作被原子的执行，中间不会被打断，保证了线程安全，

## 6类原子类纵览

以AtomicInteger为代表的基本原子类型

以及Atomic\*Array数组类型原子类

以及Atomic\*Reference引用类型原子类

以及是可以将普通基本类型进行升级的fieldUpdater类型

Adder累加器

Accumulator累加器

## 3、Atomic\*基本类型原子类，以AtomicInteger为例