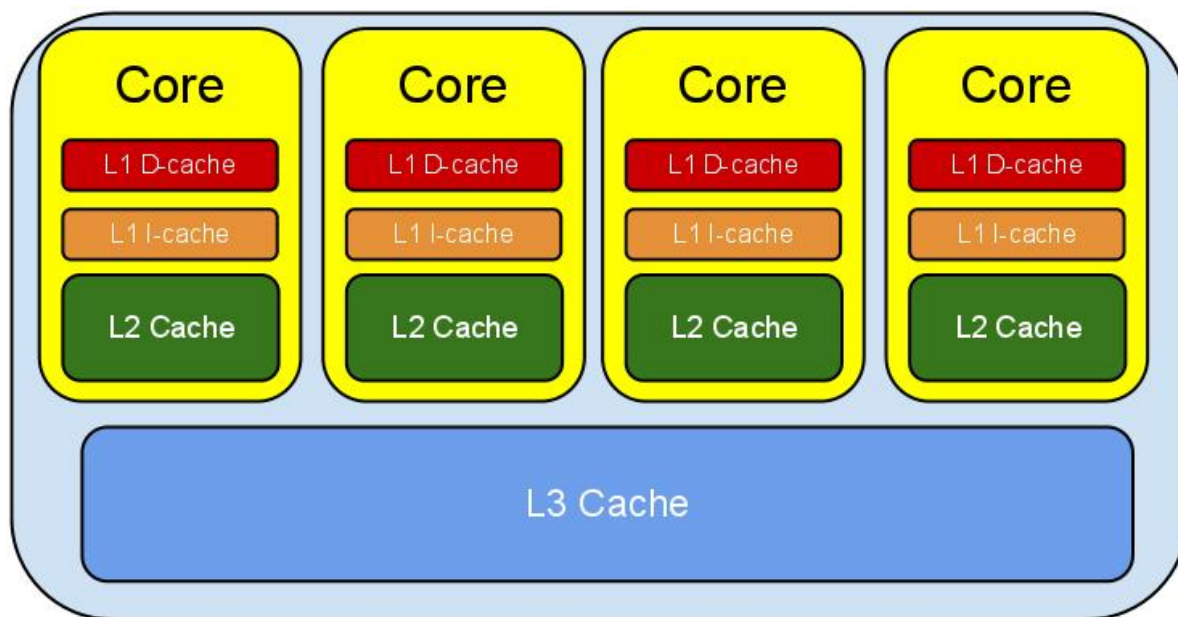


CPU 的结构

下图是计算的基本结构。

L1、L2、L3 分别表示一级缓存、二级缓存、三级缓存，越靠近 CPU 的缓存，速度越快，容量也越小。

- L1 缓存很小但很快，并且紧靠着在使用它的 CPU 内核；
- L2 大一些，也慢一些，并且仍然只能被一个单独的 CPU 核使用；
- L3 更大、更慢，并且被单个插槽上的所有 CPU 核共享；
- 最后是主存，由全部插槽上的所有 CPU 核共享。



级别越小的缓存，越接近 CPU，意味着速度越快且容量越少。

L1 是最接近 CPU 的，它容量最小（比如 256 个字节），速度最快，

每个核上都有一个 L1 Cache(准确地说每个核上有两个 L1 Cache，一个存数据 L1d Cache，一个存指令 L1i Cache)；

L2 Cache 更大一些（比如 256K 个字节），速度要慢一些，一般情况下每个核上都有一个独立的 L2 Cache；

二级缓存就是一级缓存的存储器：

一级缓存制造成本很高因此它的容量有限，二级缓存的作用就是存储那些 CPU 处理时需要用到、一级缓存又无法存储的数据。

L3 Cache 是三级缓存中最大的一级，例如（比如 12MB 个字节），同时也是最慢的一级，在同一个 CPU 插槽之间的核共享一个 L3 Cache。

三级缓存和内存可以看作是二级缓存的存储器，它们的容量递增，但单位制造成本却递减。

L3 Cache 和 L1, L2 Cache 有着本质的区别。

L1 和 L2 Cache 都是每个 CPU core 独立拥有一个，而 L3 Cache 是几个 Cores 共享的，可以认为是一个更小但是更快的内存。

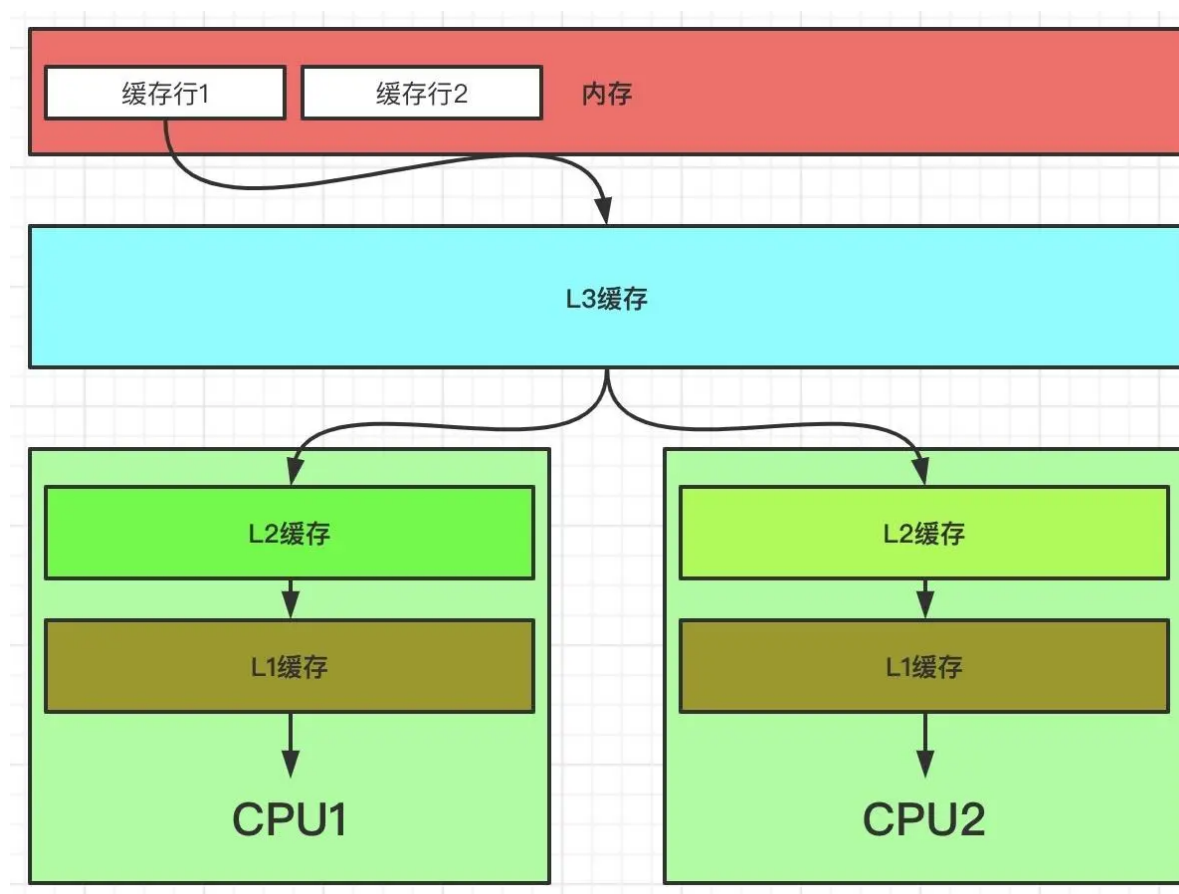
缓存行 cache line

为了提高 IO 效率，CPU 每次从内存读取数据，并不是只读取我们需要计算的数据，而是一批一批去读取的，这一批数据，也叫 Cache Line（缓存行）。

也可以理解为**批量读取，提升性能**。为啥要一批、一批的读取呢？这也满足空间的局部性原理（具体请参见葵花宝典）。

从读取的角度来说，缓存，是由缓存行 Cache Line 组成的。

所以使用缓存时，并不是一个一个字节使用，而是一行缓存行、一行缓存行这样使用；



换句话说，CPU 存取缓存都是按照一行，为最小单位操作的。并不是按照字节为单位，进行操作的。

一般而言，读取一行数据时，是将我们需要的数据周围的连续数据一次性全部读取到缓存中。这段连续的数据就称为一个**缓存行**。

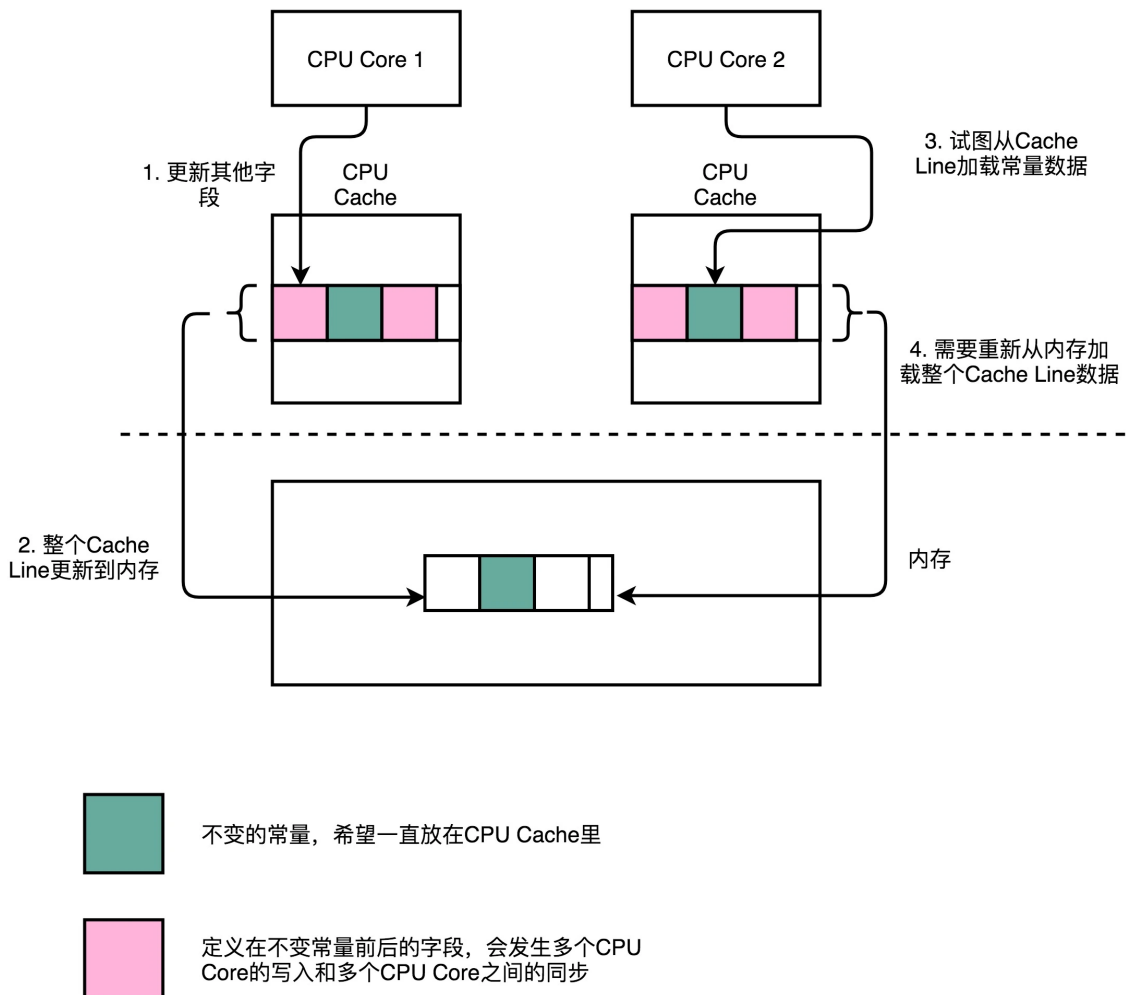
一般一行缓存行有 64 字节。intel 处理器的缓存行是 64 字节。目前主流的 CPU Cache 的 Cache Line 大小都是 64Bytes。

假设我们有一个 512 Bytes 的一级缓存，那么按照 64 Bytes 的缓存单位大小来算，这个一级缓存所能存放的缓存个数就是 $512/64 = 8$ 个。

所以，Cache Line 可以简单的理解为 CPU Cache 中的最小缓存单位。

这些 CPU Cache 的写回和加载，都不是以一个变量作为单位。这些都是以整个 Cache Line 作为单位。

如果一个常量和变量放在一行，那么变量的更新，也会影响常量的使用：



CPU 在加载数据时，整个缓存行过期了，加载常量的时候，自然也会把这个数据从内存加载到高速缓存。

什么是 伪共享（False Sharing）问题？

提前说明：翻译有瑕疵，伪共享（False Sharing），应该翻译为“错共享”，才更准确

CPU 的缓存系统是以缓存行 (cache line) 为单位存储的，一般的大小为 64bytes。

在多线程程序的执行过程中，存在着一种情况，多个需要频繁修改的变量存在同一个缓存行当中。

假设：有两个线程分别访问并修改 X 和 Y 这两个变量，X 和 Y 恰好在同一个缓存行上，这两个线程分别在不同的 CPU 上执行。

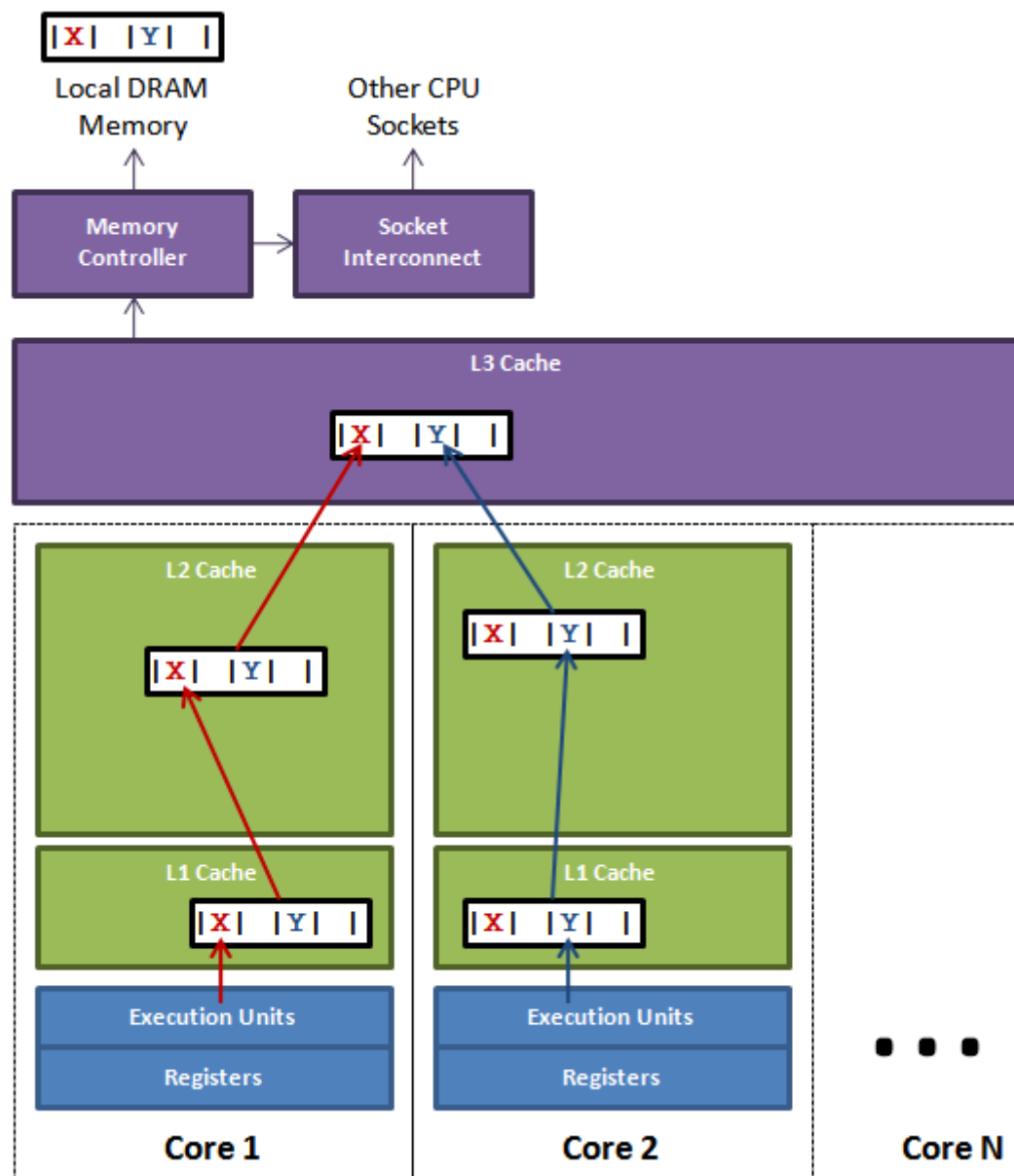
那么每个 CPU 分别更新好 X 和 Y 时将缓存行刷入内存时，发现有别的修改了各自缓存行内的数据，这时缓存行会失效，从 L3 中重新获取。

这样的话，程序执行效率明显下降。

为了减少这种情况的发生，其实就是避免 X 和 Y 在同一个缓存行中，

如何操作呢？可以主动添加一些无关变量将缓存行填满，

比如在 X 对象中添加一些变量，让它有 64 Byte 那么大，正好占满一个缓存行。



两个线程（Thread1 和 Thread2）同时修改一个同一个缓存行上的数据 X Y:

如果线程 1 打算更改 a 的值，而线程 2 准备更改 b 的值：

```
Thread1: x=3;
```

```
Thread2: y=2;
```

由 x 值被更新了，所以 x 值需要在线程 1 和线程 2 之间传递（从线程 1 到线程 2），

x、y 的变更，都会引起 cache line 整块 64 bytes 被刷新，因为 cpu 核之间以 cache line 的形式交换数据 (cache lines 的大小一般为 64bytes)。

在并发执行的场景下，每个线程在不同的核中被处理。

假设 x,y 是两个频繁修改的变量，x,y，还位于同一个缓存行。

如果，CPU1 修改了变量 x 时，L3 中的缓存行数据就失效了，也就是 CPU2 中的缓存行数据也失效了，CPU2 需要的 y 需要重新从内存加载。

如果，CPU2 修改了变量 y 时，L3 中的缓存行数据就失效了，也就是 CPU1 中的缓存行数据也失效了，CPU1 需要的 x 需要重新从内存加载。

x,y 在两个 cpu 上进行修改，本来应该是互不影响的，但是由于缓存行在一起，导致了相互受到了影响。

伪共享问题（False Sharing）的本质

出现伪共享问题（False Sharing）的原因：

- 一个缓存行可以存储多个变量（存满当前缓存行的字节数）；64 个字节可以放 8 个 long，16 个 int
- 而 CPU 对缓存的修改又是以缓存行为最小单位的；不是以 long、byte 这样的数据类型为单位的
- 在多线程情况下，如果需要修改“共享同一个缓存行的其中一个变量”，该行中其他变量的状态就会失效，甚至进行一致性保护

所以，伪共享问题（False Sharing）的本质是：

对缓存行中的单个变量进行修改了，导致整个缓存行其他不相关的数据也就失效了，需要从主存重新加载

如果其中有 **volatile** 修饰的变量，需要保证线程可见性的变量，还需要进入缓存与数据一致性的保障流程，如 **mes**i 协议的数据一致性保障用了其他变量的 **Core** 的缓存一致性。

缓存一致性是根据缓存行为单元来进行同步的，假如 y 是 volatile 类型的，假如 a 修改了 x，而其他的线程用到 y，虽然用到的不是同一个数据，但是他们（数据 X 和数据 Y）在同一个缓存行中，其他的线程的缓存需要保障数据一致性而进行数据同步，当然，同步也需要时间。

一个 CPU 核心在加载一个缓存行时要执行上百条指令。如果一个核心要等待另外一个核心来重新加载缓存行，那么他就必须等在那里，称之为 **stall** (停止运转)。

伪共享问题的解决方案

减少伪共享也就意味着减少了 **stall** 的发生，其中一个手段就是通过填充 (Padding) 数据的形式，来保证本应有可能位于同一个缓存行的两个变量，在被多线程访问时必定位于不同的缓存行。

简单的说，就是 **以空间换时间**：使用占位字节，将变量的所在的缓冲行塞满。

disruptor 无锁框架就是这么干的。

一个缓冲行填充的例子

下面是一个填充了的缓存行的，尝试 p1, p2, p3, p4, p5, p6 为 AtomicLong 的 value 的缓存行占位，将 AtomicLong 的 value 变量的所在的缓冲行塞满，

代码如下：

```
package com.crazymakercircle.demo.cas;

import java.util.concurrent.atomic.AtomicLong;

public class PaddedAtomicLong extends AtomicLong {
    private static final long serialVersionUID = -3415778863941386253L;

    /**
     * Padded 6 long (48 bytes)
     */
    public volatile long p1, p2, p3, p4, p5, p6 = 7L;

    /**
     * Constructors from {@link AtomicLong}
     */
}
```

```

    */
    public PaddedAtomicLong() {
        super();
    }

    public PaddedAtomicLong(long initialValue) {
        super(initialValue);
    }

    /**
     * To prevent GC optimizations for cleaning unused padded references
     */
    public long sumPaddingToPreventOptimization() {
        return p1 + p2 + p3 + p4 + p5 + p6;
    }
}

```

例子的部分结果如下:

```

printable = com.crazymakercircle.basic.demo.cas.busi.PaddedAtomicLong object
internals:

```

OFFSET	SIZE	TYPE DESCRIPTION	VALUE
0	4	(object header)	01 00 00 00
(00000001 00000000 00000000 00000000) (1)			
4	4	(object header)	00 00 00 00
(00000000 00000000 00000000 00000000) (0)			
8	4	(object header)	50 08 01 f8
(01010000 00001000 00000001 11111000) (-134150064)			
12	4	(alignment/padding gap)	
16	8	long AtomicLong.value	0
24	8	long PaddedAtomicLong.p1	0
32	8	long PaddedAtomicLong.p2	0
40	8	long PaddedAtomicLong.p3	0
48	8	long PaddedAtomicLong.p4	0
56	8	long PaddedAtomicLong.p5	0
64	8	long PaddedAtomicLong.p6	7

Instance size: 72 bytes

Space losses: 4 bytes internal + 0 bytes external = 4 bytes total

伪共享 False Sharing 在 java 8 中解决方案

JAVA 8 中添加了一个 @Contended 的注解, 添加这个的注解, 将会在自动进行缓存行填充。

```
116 *  
117 * JVM intrinsics note: It would be possible to use a release-only  
118 * form of CAS here, if it were provided.  
119 */  
120 @sun.misc.Contended static final class Cell {  
121     volatile long value;  
122     Cell(long x) { value = x; }  
123     final boolean cas(long cmp, long val) { return UNSAFE.compareAndSwapLong(this, valueOffset, cmp, val); }  
124  
125     // Unsafe mechanics  
126     private static final sun.misc.Unsafe UNSAFE;  
127     private static final long valueOffset;  
128     static {  
129         try {  
130             UNSAFE = sun.misc.Unsafe.getUnsafe();  
131             Class<?> ak = Cell.class;  
132             valueOffset = UNSAFE.objectFieldOffset  
133                 (ak.getDeclaredField("value"));  
134         } catch (Exception e) {  
135             throw new Error(e);  
136         }  
137     }  
138 }  
139  
140 }  
141  
142 /** Number of CPUs, to place bound on table size */
```

下面有一个 @Contended 的例子：

```
package com.crazymakercircle.basic.demo.cas.busi;  
import sun.misc.Contended;  
public class ContendedDemo  
{  
    //有填充的演示成员  
    @Contended  
    public volatile long padVar;  
  
    //没有填充的演示成员  
    public volatile long notPadVar;  
  
}
```

以上代码使得 padVar 和 notPadVar 都在不同的 cache line 中。@Contended 使得 notPadVar 字段远离了对象头部分。

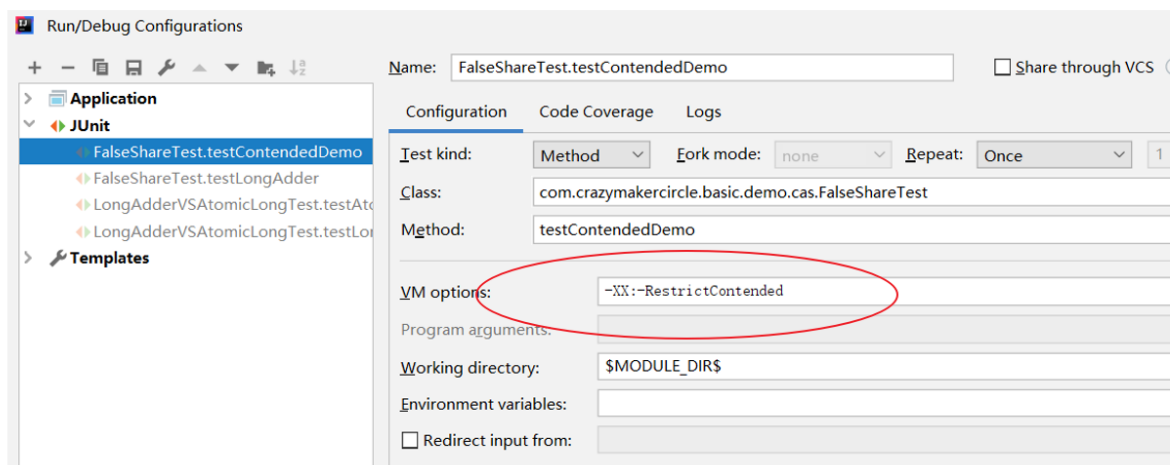
```

printable = com.crazymakercircle.basic.demo.cas.busi.ContendedDemo object
internals:
  OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4      (object header)                   01 00 00 00 (00000001 00000000
00000000 00000000) (1)
      4     4      (object header)                   00 00 00 00 (00000000 00000000
00000000 00000000) (0)
      8     4      (object header)                   50 08 01 f8 (01010000 00001000
00000001 11111000) (-134150064)
     12     4      (alignment/padding gap)
     16     8    long ContendedDemo.padVar           0
     24     8    long ContendedDemo.notPadVar         0
Instance size: 32 bytes
Space losses: 4 bytes internal + 0 bytes external = 4 bytes total

```

执行时，必须加上虚拟机参数 -XX:-RestrictContended，@Contended 注释才会生效。

很多文章把这个漏掉了，那样的话实际上就没有起作用。



新的结果；

```

printable = com.crazymakercircle.basic.demo.cas.busi.ContendedDemo object
internals:
  OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4      (object header)                   01 00 00 00 (00000001 00000000
00000000 00000000) (1)
      4     4      (object header)                   00 00 00 00 (00000000 00000000
00000000 00000000) (0)
      8     4      (object header)                   50 08 01 f8 (01010000 00001000
00000001 11111000) (-134150064)
     12     4      (alignment/padding gap)
     16     8    long ContendedDemo.notPadVar           0
     24    128      (alignment/padding gap)
    152     8    long ContendedDemo.padVar           0
    160    128      (loss due to the next object alignment)
Instance size: 288 bytes
Space losses: 132 bytes internal + 128 bytes external = 260 bytes total

```


在 Java 8 中，使用 `@Contended` 注解的对象或字段的前后各增加 128 字节大小的 padding，使用 2 倍于大多数硬件缓存行的大小来避免相邻扇区预取导致的伪共享冲突。我们目前的缓存行大小一般为 64Byte，这里 `Contended` 注解为我们前后加上了 128 字节绰绰有余。

注意：如果想要 `@Contended` 注解起作用，需要在启动时添加 JVM 参数 -XX:-RestrictContended 参数后 `@sun.misc.Contended` 注解才有。

可见至少在 JDK1.8 以上环境下，只有 `@Contended` 注解才能解决伪共享问题，但是消耗也很大，占用了宝贵的缓存，用的时候要谨慎。

另外：

`@Contended` 注释还可以添加在类上，每一个成员，都会加上。

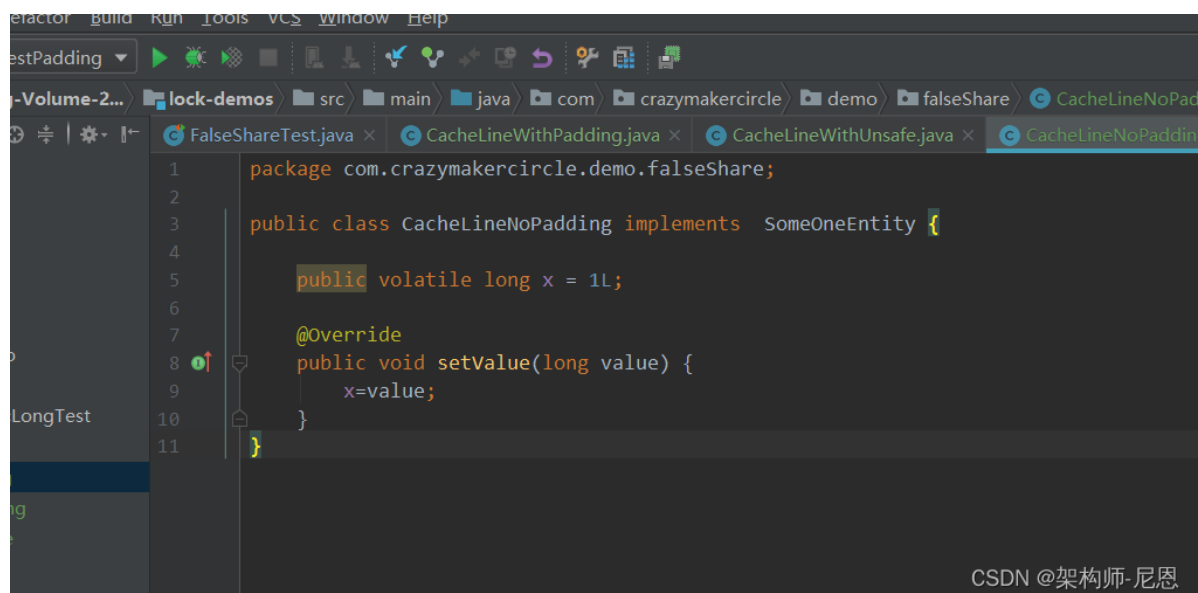
伪共享性能比对实操：结论，差 6 倍

三个实操：

- 首先存在伪共享场景下的 耗时计算
- 其次是消除伪共享场景下的 耗时计算
- 再次是使用 `unsafe` 访问变量时的耗时计算

存在伪共享场景下的 耗时计算

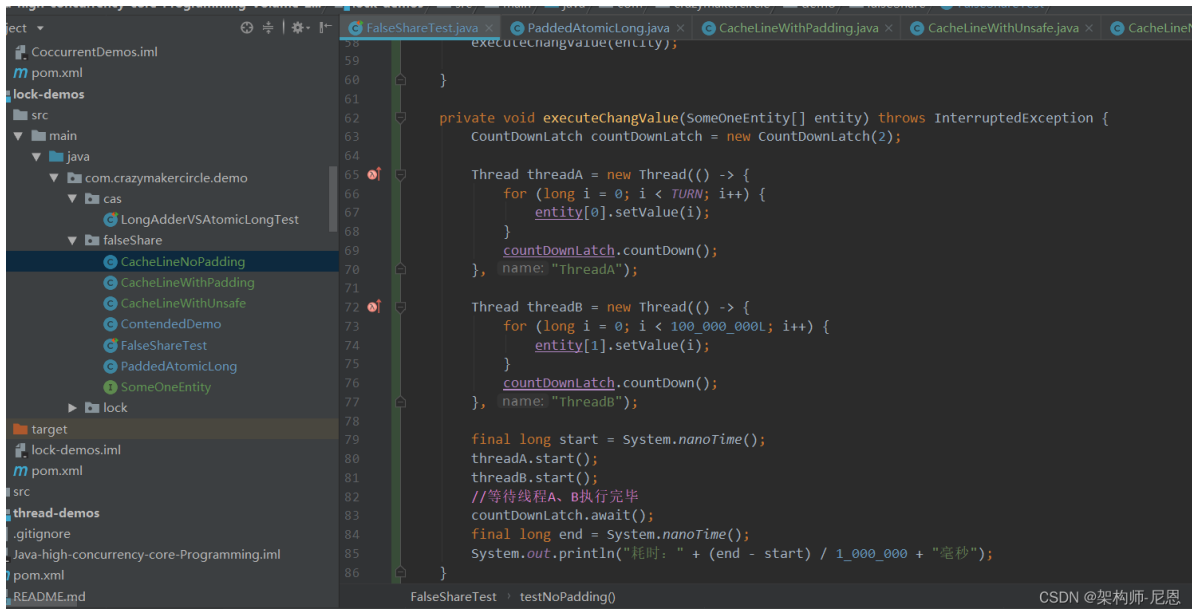
entity 类



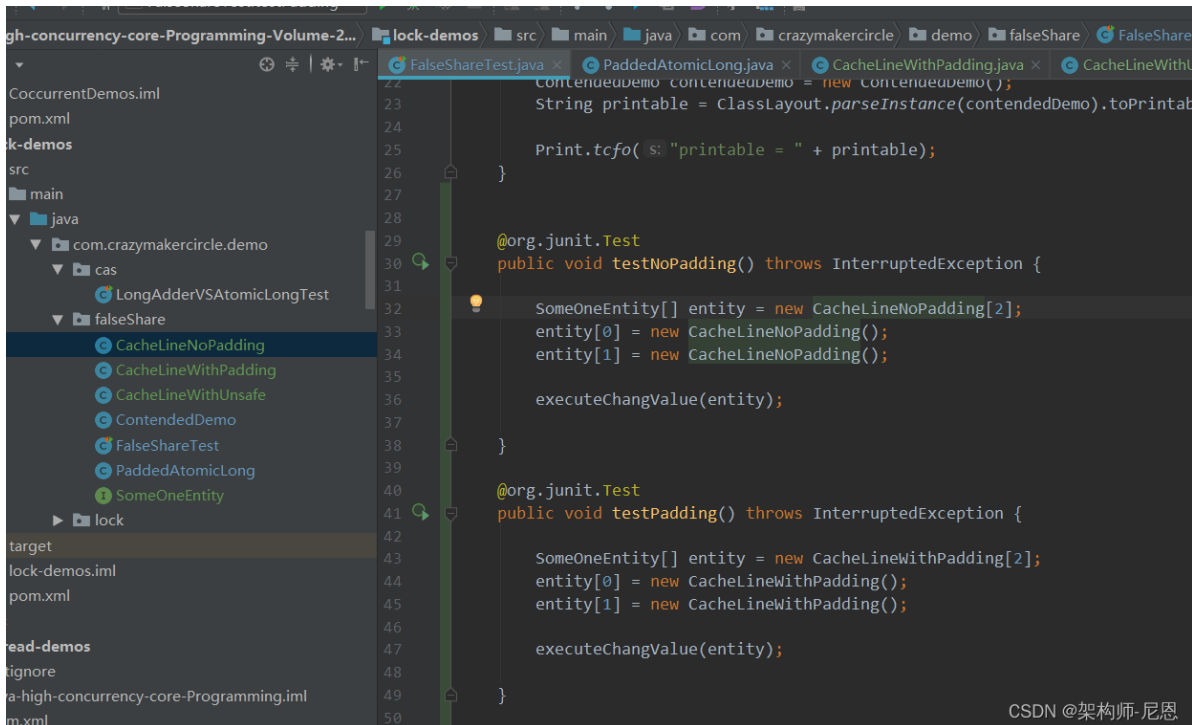
```
1 package com.crazymakercircle.demo.falseShare;
2
3 public class CacheLineNoPadding implements SomeoneEntity {
4
5     public volatile long x = 1L;
6
7     @Override
8     public void setValue(long value) {
9         x=value;
10    }
11 }
```

CSDN @架构师-尼恩

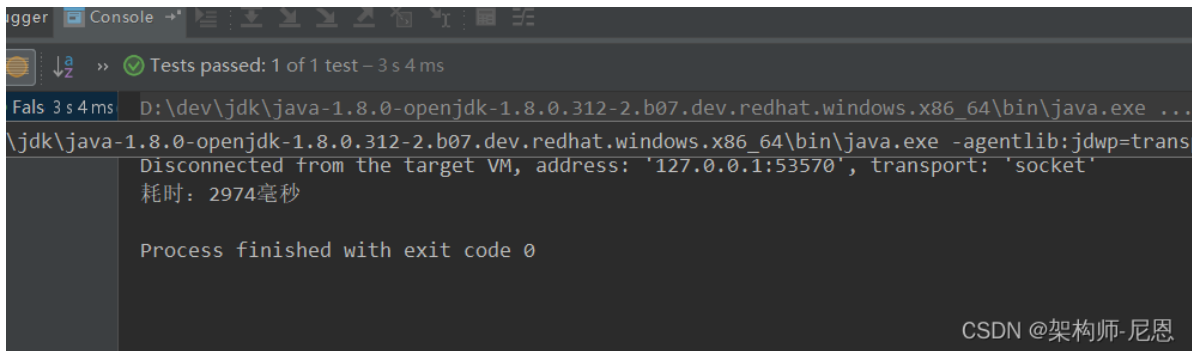
并行的执行数据修改，这里抽取成为了一个通用的方法



测试用例

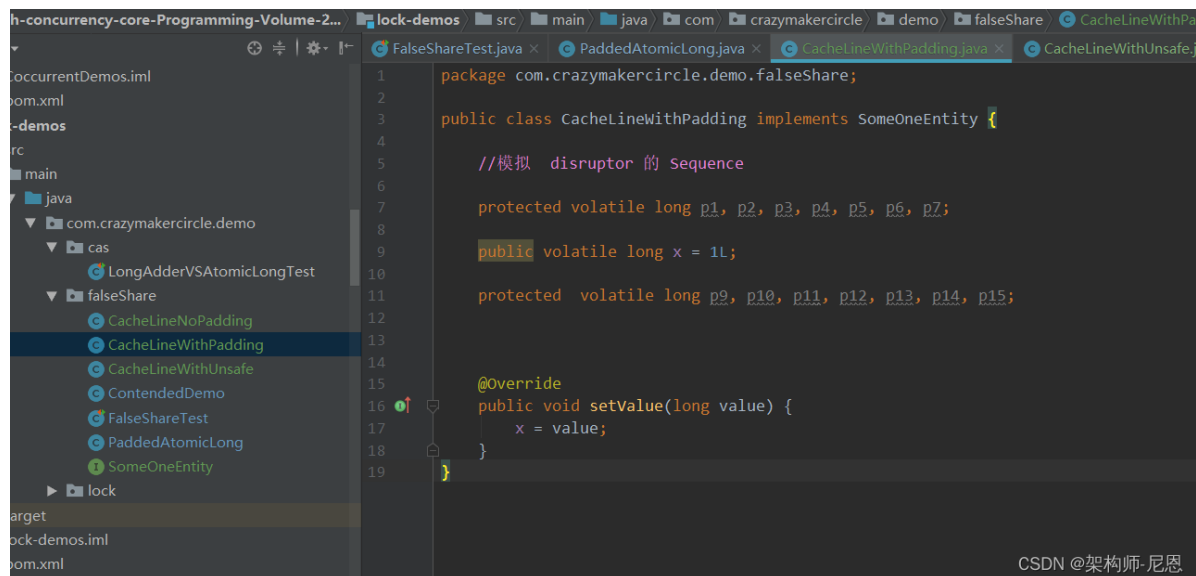


执行的时间



消除伪共享场景下的 耗时计算

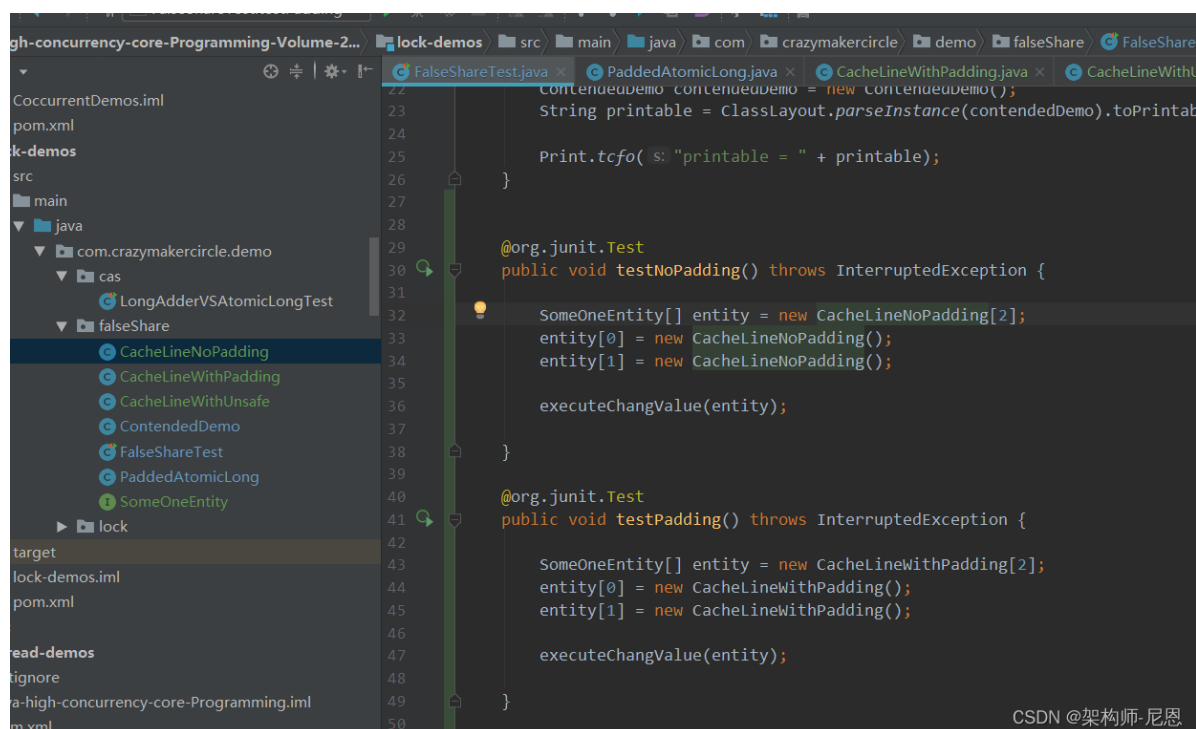
entity 类



```
1 package com.crazymakercircle.demo.falseShare;
2
3 public class CacheLineWithPadding implements SomeOneEntity {
4
5     //模拟 disruptor 的 Sequence
6
7     protected volatile long p1, p2, p3, p4, p5, p6, p7;
8
9     public volatile long x = 11;
10
11     protected volatile long p9, p10, p11, p12, p13, p14, p15;
12
13
14
15     @Override
16     public void setValue(long value) {
17         x = value;
18     }
19 }
```

CSDN @架构师-尼恩

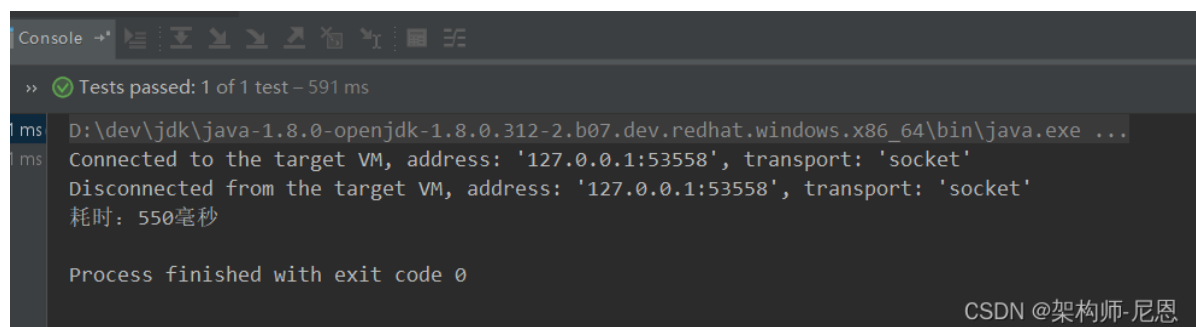
测试用例



```
22
23
24 ContentedDemo contendedemo = new ContentedDemo();
25 String printable = ClassLayout.parseInstance(contendedemo).toPrintable();
26
27 Print.tcfo( s: "printable = " + printable);
28
29
30 @org.junit.Test
31 public void testNoPadding() throws InterruptedException {
32
33     SomeOneEntity[] entity = new CacheLineNoPadding[2];
34     entity[0] = new CacheLineNoPadding();
35     entity[1] = new CacheLineNoPadding();
36
37     executeChangeValue(entity);
38 }
39
40 @org.junit.Test
41 public void testPadding() throws InterruptedException {
42
43     SomeOneEntity[] entity = new CacheLineWithPadding[2];
44     entity[0] = new CacheLineWithPadding();
45     entity[1] = new CacheLineWithPadding();
46
47     executeChangeValue(entity);
48 }
49
50 }
```

CSDN @架构师-尼恩

消除伪共享场景下的 耗时计算 (550ms)

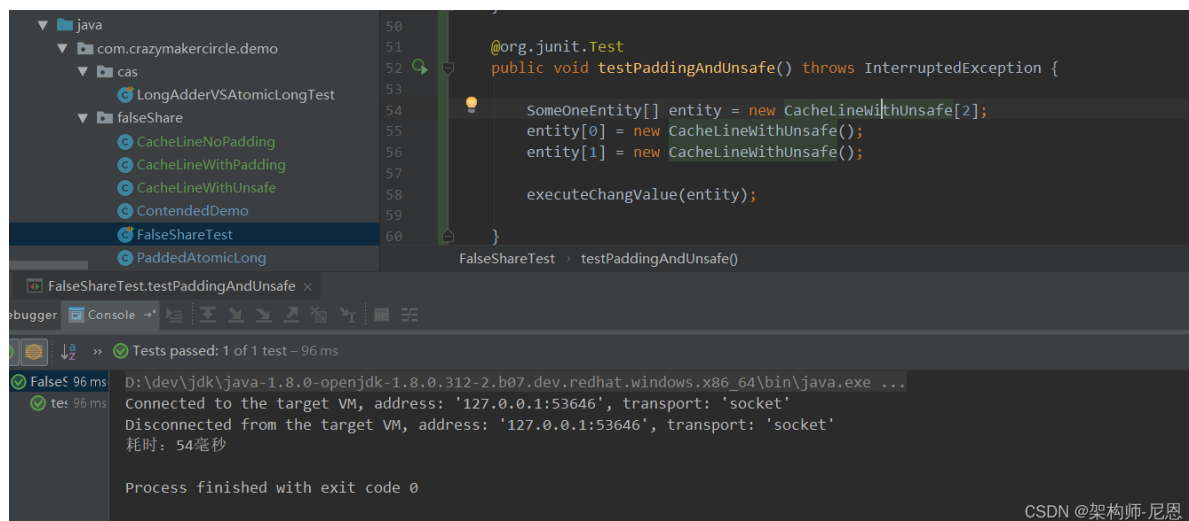
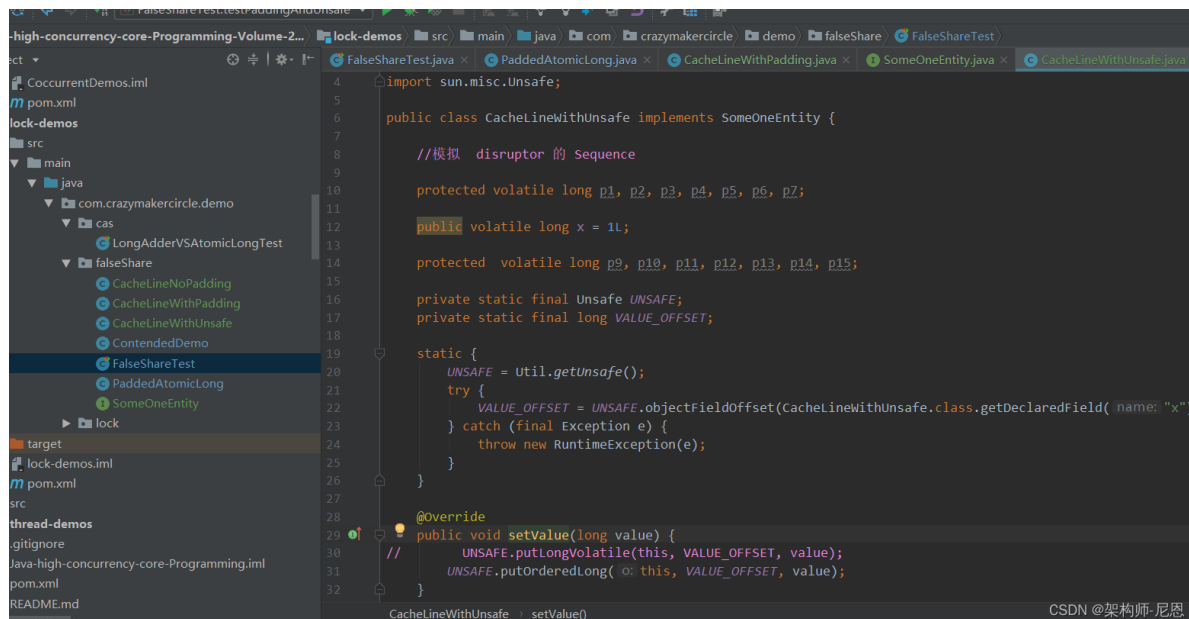


```
>> Tests passed: 1 of 1 test – 591 ms
D:\dev\jdk\java-1.8.0-openjdk-1.8.0.312-2.b07.dev.redhat.windows.x86_64\bin\java.exe ...
Connected to the target VM, address: '127.0.0.1:53558', transport: 'socket'
Disconnected from the target VM, address: '127.0.0.1:53558', transport: 'socket'
耗时: 550毫秒
Process finished with exit code 0
```

CSDN @架构师-尼恩

使用 unsafe 访问变量的耗时计算

entity



使用 unsafe 访问变量的耗时计算:

54ms

性能总结

消除伪共享场景，比存在伪共享场景 的性能，性能提升 6 倍左右

实验数据，从 3000ms 提升 到 500ms

使用 unsafe 取消内存可见性，比消除伪共享场景，性能提升 10 倍左右

实验数据，从 500ms 提升 到 50ms

通过实验的对比，可见 Java 的性能，是可以大大优化的，尤其在高性能组件

以上实操的 详细介绍，请参见 《100wqps 日志平台实操》

JDK 源码中如何解决 伪共享问题

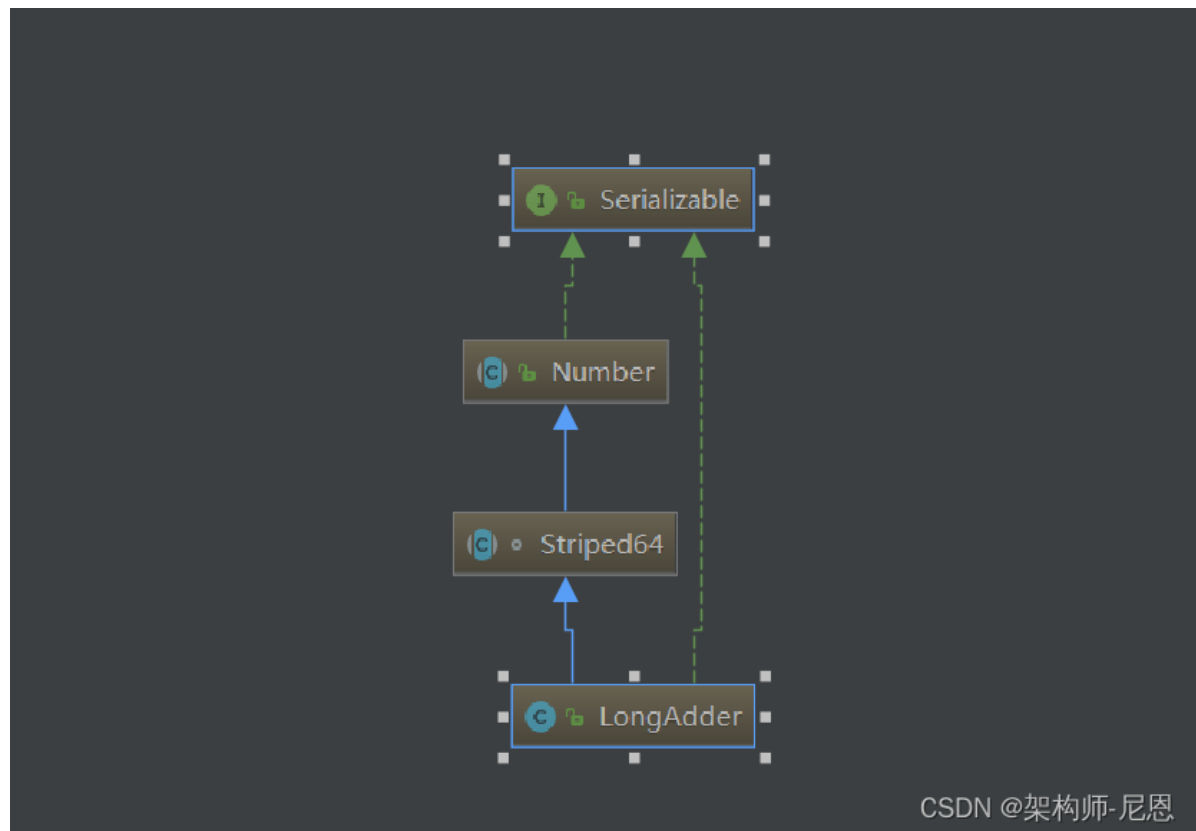
在 LongAdder 在 java8 中的实现已经采用了 @Contended。

LongAdder 以及 Striped64 如何解决伪共享问题

LongAdder 是大家常用的 高并发累加器

通过分而治之的思想，实现 超高并发累加。

LongAdder 的结构如下：



Striped64 是在 java8 中添加用来支持累加器的并发组件，它可以在并发环境下使用来做某种计数，

Striped64 的设计思路是在竞争激烈的时候尽量分散竞争，

Striped64 维护了一个 base Count 和一个 Cell 数组，计数线程会首先试图更新 base 变量，如果成功则退出计数，否则会认为当前竞争是很激烈的，那么就会通过 Cell 数组来分散计数，

Striped64 根据线程来计算哈希，然后将不同的线程分散到不同的 Cell 数组的 index 上，然后这个线程的计数内容就会保存在该 Cell 的位置上面，

基于这种设计，最后的总计数需要结合 base 以及散落在 Cell 数组中的计数内容。

这种设计思路类似于 java7 的 ConcurrentHashMap 实现，也就是所谓的分段锁算法，ConcurrentHashMap 会将记录根据 key 的 hashCode 来分散到不同的 segment 上，

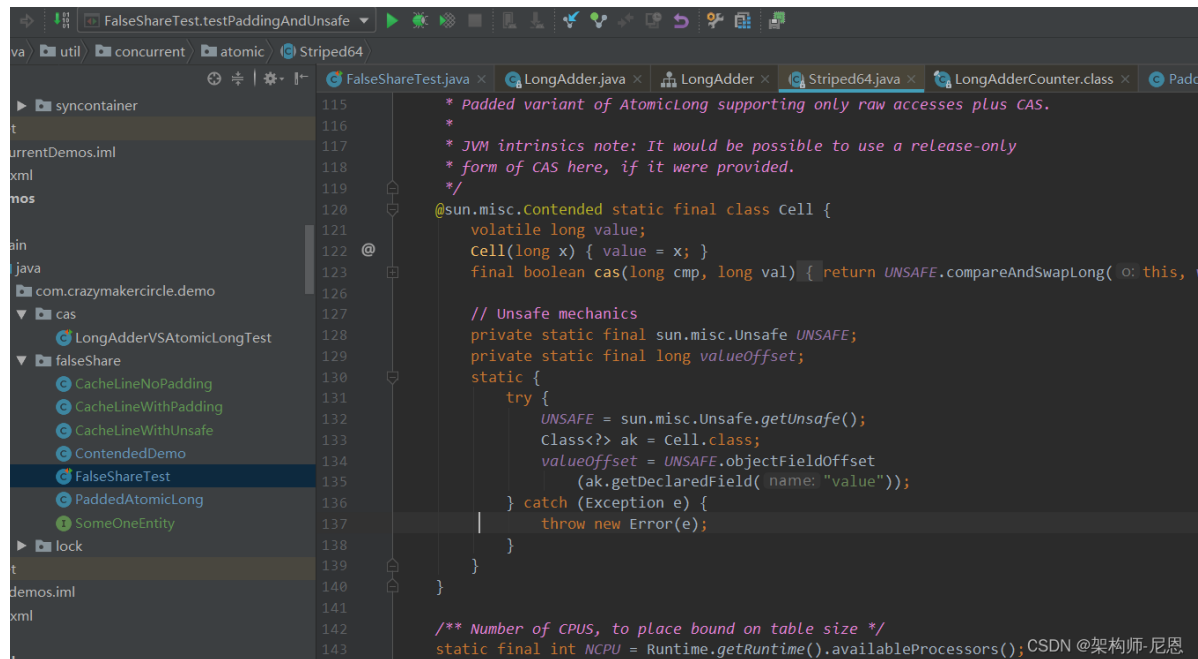
线程想要操作某个记录，只需要锁住这个记录对应着的 segment 就可以了，而其他 segment 并不会被锁住，其他线程任然可以去操作其他的 segment，

这样就显著提高了并发度，

虽然如此，java8 中的 ConcurrentHashMap 实现已经抛弃了 java7 中分段锁的设计，而采用更为轻量级的 CAS 来协调并发，效率更佳。

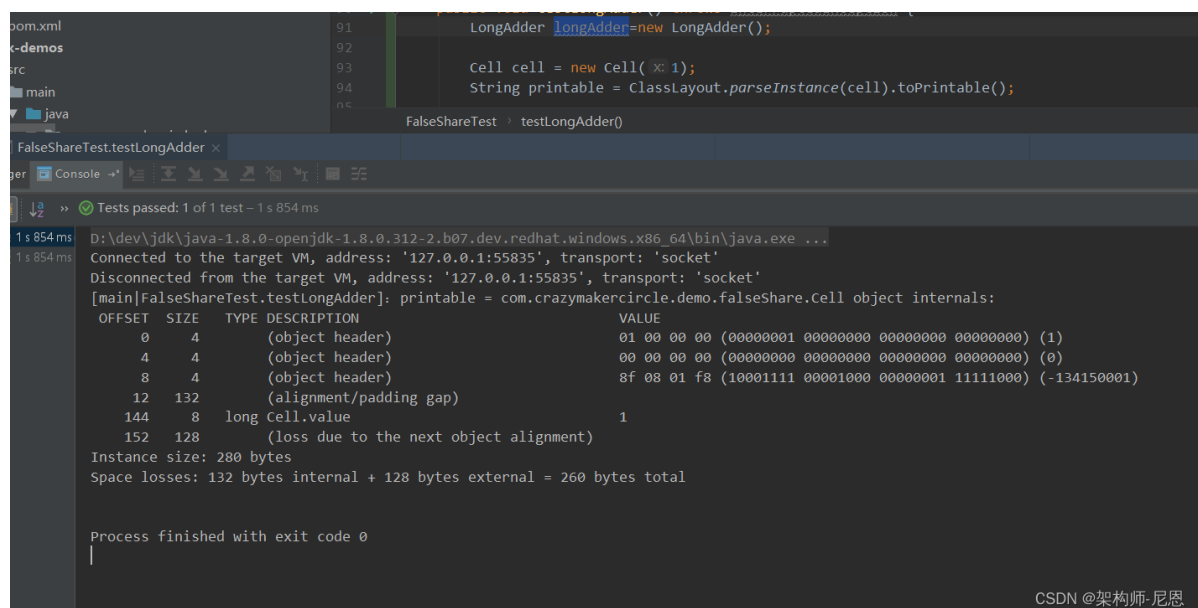
Cell 元素如何消除伪共享

Striped64 中的 Cell 元素，是如何消除伪共享的呢？



```
115 * Padded variant of AtomicLong supporting only raw accesses plus CAS.
116 *
117 * JVM intrinsics note: It would be possible to use a release-only
118 * form of CAS here, if it were provided.
119 */
120 @sun.misc.Contended static final class Cell {
121     volatile long value;
122     Cell(long x) { value = x; }
123     final boolean cas(long cmp, long val) { return UNSAFE.compareAndSwapLong( @this,
124
125 // Unsafe mechanics
126 private static final sun.misc.Unsafe UNSAFE;
127 private static final long valueOffset;
128 static {
129     try {
130         UNSAFE = sun.misc.Unsafe.getUnsafe();
131         Class<?> ak = Cell.class;
132         valueOffset = UNSAFE.objectFieldOffset
133             (ak.getDeclaredField( name: "value"));
134     } catch (Exception e) {
135         throw new Error(e);
136     }
137 }
138 }
139 }
140
141 /** Number of CPUs, to place bound on table size */
142 static final int NCPU = Runtime.getRuntime().availableProcessors();
143
```

可以打印一下 cell 的 内存结构



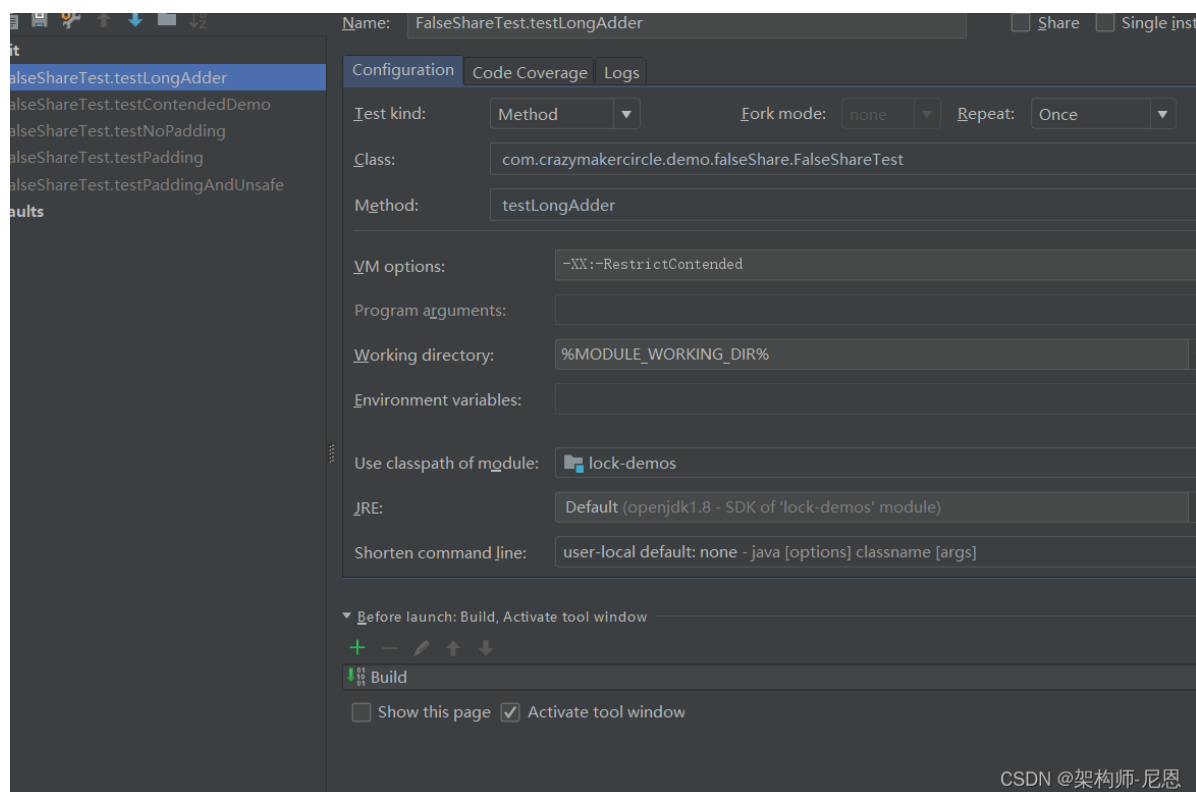
```
91 LongAdder longAdder=new LongAdder();
92
93 Cell cell = new Cell( 1);
94 String printable = ClassLayout.parseInstance(cell).toPrintable();
95
FalseShareTest - testLongAdder()

Tests passed: 1 of 1 test - 1 s 854 ms

D:\dev\jdk\java-1.8.0-openjdk-1.8.0.312-2.b07.dev.redhat.windows.x86_64\bin\java.exe ...
Connected to the target VM, address: '127.0.0.1:55835', transport: 'socket'
Disconnected from the target VM, address: '127.0.0.1:55835', transport: 'socket'
[main]FalseShareTest.testLongAdder(): printable = com.crazymakercircle.demo.falseShare.Cell object internals:
  OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
    0      4         (object header)                   01 00 00 00 (00000001 00000000 00000000 00000000) (1)
    4      4         (object header)                   00 00 00 00 (00000000 00000000 00000000 00000000) (0)
    8      4         (object header)                   8f 08 01 f8 (10001111 00001000 00000001 11111000) (-134150001)
   12     132        (alignment/padding gap)
  144      8       long Cell.value                               1
  152     128        (loss due to the next object alignment)
Instance size: 280 bytes
Space losses: 132 bytes internal + 128 bytes external = 260 bytes total

Process finished with exit code 0
```

当然，别忘记加上 vm 选项：-XX:-RestrictContended



对于伪共享，我们在实际开发中该怎么做？

通过上面大篇幅的介绍，我们已经知道伪共享的对程序的影响。

那么，在实际的生产开发过程中，我们一定要通过缓存行填充去解决掉潜在的伪共享问题吗？

其实并不一定。

首先就是多次强调的，伪共享是很隐蔽的，我们暂时无法从系统层面上通过工具来探测伪共享事件。

其次，不同类型的计算机具有不同的微架构（如 32 位系统和 64 位系统的 java 对象所占自己数就不一样），如果设计到跨平台的设计，那就更难以把握了，一个确切的填充方案只适用于一个特定的操作系统。

还有，缓存的资源是有限的，如果填充会浪费珍贵的 cache 资源，并不适合大范围应用。

最后，目前主流的 Intel 微架构 CPU 的 L1 缓存，已能够达到 80% 以上的命中率。

综上所述，并不是每个系统都适合花大量精力去解决潜在的伪共享问题

参考文献

<https://blog.csdn.net/hxg117/article/details/78064632>

<http://openjdk.java.net/projects/jdk8/features>

<http://beautynbits.blogspot.co.uk/2012/11/the-end-for-false-sharing-in-java.html>

<http://openjdk.java.net/jeps/142>

<http://mechanical-sympathy.blogspot.co.uk/2011/08/false-sharing-java-7.html>

<http://stackoverflow.com/questions/19892322/when-will-jvm-use-intrinsics>

https://blogs.oracle.com/dave/entry/java_contented_annotation_to_help

<https://www.jianshu.com/p/b38ffa33d64d>

<https://blog.csdn.net/MrYushiwen/article/details/123171635>

<https://blog.csdn.net/everyok/article/details/88889057>

