# CPU Cache与Java高性能编程

修改记录表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 操作者 | 修改内容 | 修改时间 |
| 1 | 卫斯理 | 编写整理小组稿文  添加第三部分，Unsafe部分内容  修改数组部分说明图例  添加目录  忽略拼写错误检查 | 2016/10/30 23:41 |
| 2 | 卫斯理 | 语句通顺及逻辑思考调整  代码块风格统一为eclipse风格，并进行代码正确性检验 | 2016-10-31 15:37:43 |
|  |  |  |  |
|  |  |  |  |

[CPU Cache与Java高性能编程 1](#_Toc466235713)

[认识CPU Cache 2](#_Toc466235714)

[存储器层次结构中的Cache 2](#_Toc466235715)

[Cache中最小的寻址单位Cache Line 2](#_Toc466235716)

[如何查看本机的Cache信息 3](#_Toc466235717)

[Windows下查看Cache的信息 3](#_Toc466235718)

[Linux下查看Cache信息 3](#_Toc466235719)

[Java通过CacheSize API方式 3](#_Toc466235720)

[Cache Line内存加载 4](#_Toc466235721)

[Cache Line的使用 4](#_Toc466235722)

[Cache Line miss问题 4](#_Toc466235723)

[Cache Line伪共享 5](#_Toc466235724)

[跨Cache Line问题 5](#_Toc466235725)

[数据对齐问题 5](#_Toc466235726)

[Cache Line友好型Java程序 6](#_Toc466235727)

[友好的数据结构设计 6](#_Toc466235728)

[Java数组 6](#_Toc466235729)

[Unsafe构建连续地址的引用类型数组 7](#_Toc466235730)

[解决伪共享的Cache line友好类 8](#_Toc466235731)

[Padding方式 9](#_Toc466235732)

[Contended注解方式 9](#_Toc466235733)

[友好的逻辑控制 10](#_Toc466235734)

[循环的步长 10](#_Toc466235735)

[循环体的优化 10](#_Toc466235736)

[局部变量代替全局字段的访问 10](#_Toc466235737)

[Java中对Cache line经典设计 11](#_Toc466235738)

[Contended注解在JDK1.8的应用 11](#_Toc466235739)

[ConcurrentHashMap 11](#_Toc466235740)

[Thread 11](#_Toc466235741)

[Disruptor框架 11](#_Toc466235742)

[认识Disruptor 11](#_Toc466235743)

[Disruptor业务背景 11](#_Toc466235744)

[Disruptor运行流程分析 12](#_Toc466235745)

[Disruptor特点 12](#_Toc466235746)

[Disruptor对伪共享的处理 12](#_Toc466235747)

[RingBuffer类 12](#_Toc466235748)

[Sequence类 13](#_Toc466235749)

[Sequencer的实现 13](#_Toc466235750)

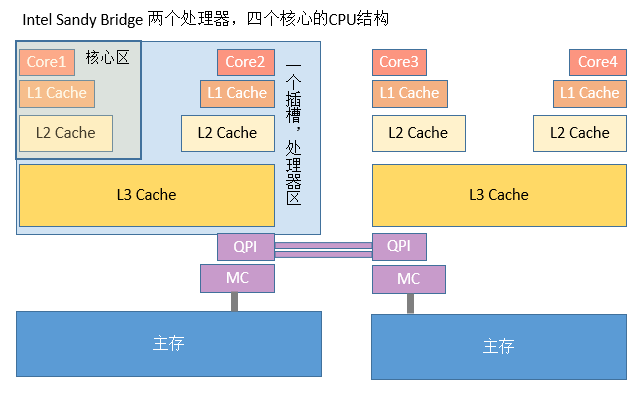
[总结与使用示例 14](#_Toc466235751)

[参考资料: 14](#_Toc466235752)

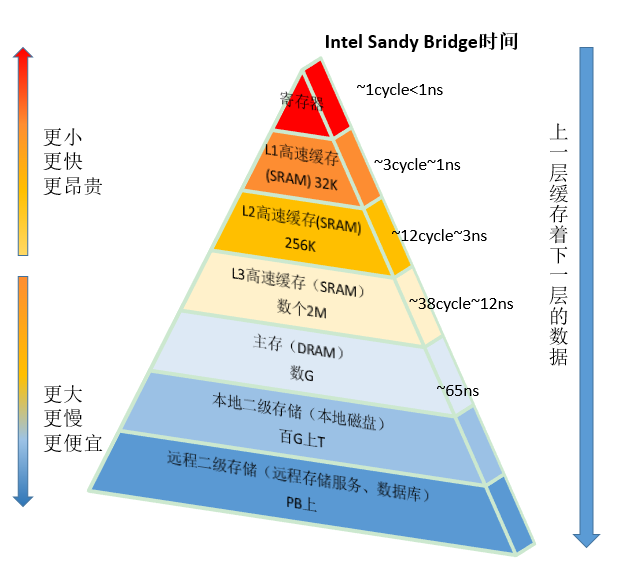
## 认识CPU Cache

### 存储器层次结构中的Cache

现代计算机进入多核时代甚至是多处理器时代。一个处理器对应一个物理插槽，多处理器间通过QPI总线相连，一个处理器包含多个核，一个处理器间的多核共享L3级缓存，多个处理器。一个核包含寄存器、L1级、L2级缓存共享L3级缓存，有些并行指令核心有两个线程（超核）。下图是Intel Sandy Bridge CPU架构，一个典型的NUMA多处理器结构：



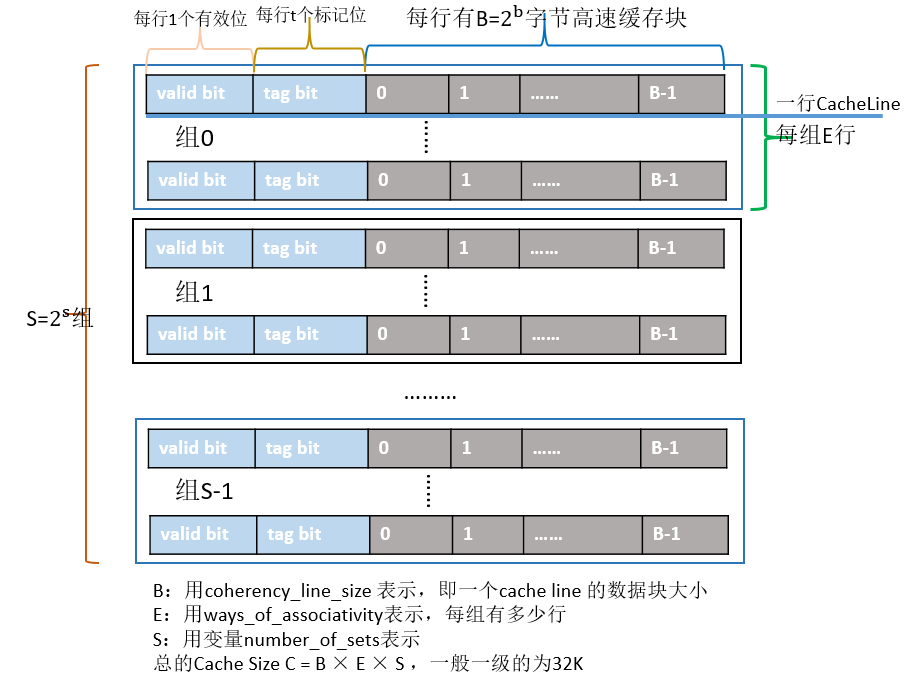
随着CPU的频率不断提升，而内存的访问速度却没有质的突破，为了弥补内存的访问速度，充分发挥CPU的计算资源，提高CPU整体吞吐量，在CPU与内存之间引入了缓存。随着热点数据体积越来越大，一级缓存已经不满足发展的要求，引入了二级缓存，三级缓存。（注：若无特别说明，本文的Cache指CPU Cache，高速缓存）缓存在存储器层次结构中的示意如下图：



作为程序员，需要理解存储器层次结构，它对应用程序的性能有巨大的影响。如果需要的程序是在CPU寄存器中的，指令执行时1个周期内就能访问到他们。如果在高速缓存中，需要1~30个周期；如果在主存中，需要50~200个周期；在磁盘上，大概需要几千万个周期。充分利用它的结构和机制，可以有效的提高程序的性能。

### Cache中最小的寻址单位Cache Line

缓存从内存中读取数据的方式有直接映射、全相联、组相联三种，因为现代芯片大部分使用组相联的方式，所以这里介绍的cache结构和从内存加载的知识基于组相联的方式。一个cache被分为S个组，每个组有E行Cache Line。一个Cache Line中，有B个字节存储单元，现代处理器中，这个存储单元一般是以字节(通常8个位)为单位的，也是最小的加载单元。Cache结构如图：

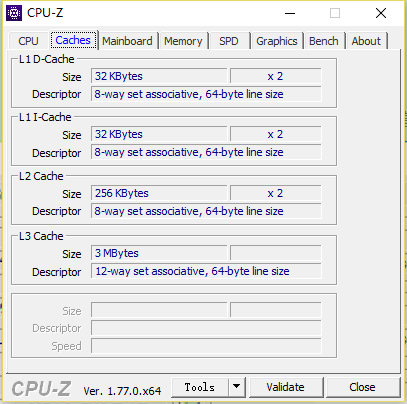
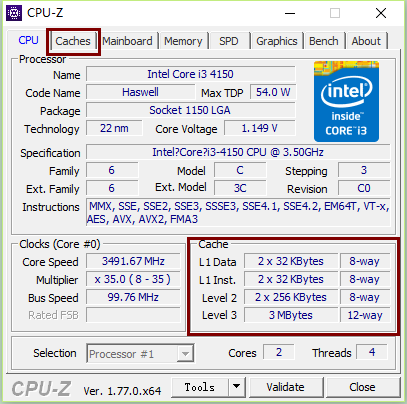


高速缓存的大小C是指所有块大小的和，标记位和有效位不包括在里内。因此，C=B×E×S。一个Cache Line包含一个有效位(valid bit)、t个标记位(tag bit)、大小数据块(cache block)构成。valid bit用来表示该缓存行是否有效，tag bit用来寻址，cache block则是实际对应在内存中的地址内容。

### 如何查看本机的Cache信息

#### Windows下查看Cache的信息

在windows下查看方式有多种方式，其中最直观的是，通过安装[CPU-Z](http://www.cpuid.com/downloads/cpu-z/cpu-z_1.77-cn.exe)软件，直接显示Cache信息，如下图：



还有两种方法：

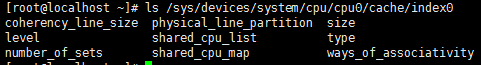
①Windows API调用[GetLogicalProcessorInfo](file:///E:\\study\\advance\\04\\msdn.microsoft.com\\en-us\\library\\ms683194(VS.85).aspx)。

②通过命令行系统内部工具[CoreInfo](https://technet.microsoft.com/en-us/sysinternals/cc835722.aspx)。

#### Linux下查看Cache信息

使用下面的命令：

|  |
| --- |
| ls /sys/devices/system/cpu/cpu0/cache/index0 |



还有lscpu等命令也可以查看相关信息

#### Java通过[CacheSize](https://code.google.com/archive/p/cachesize/) API方式

[CacheSize](https://code.google.com/archive/p/cachesize/)是一个谷歌的小项目，java语言通过它可以进行访问本机Cache的信息。示例代码如下：

|  |  |
| --- | --- |
|  | **public** **static** **void** main(String[] args) **throws** CacheNotFoundException {  CacheInfo info = CacheInfo.*getInstance*();  CacheLevelInfo l1Datainf = info.getCacheInformation(CacheLevel.***L1***, CacheType.***DATA\_CACHE***);  System.***out***.println("第一级数据缓存信息："+l1Datainf.toString());    CacheLevelInfo l1Instrinf = info.getCacheInformation(CacheLevel.***L1***, CacheType.***INSTRUCTION\_CACHE***);  System.***out***.println("第一级指令缓存信息："+l1Instrinf.toString());  } |

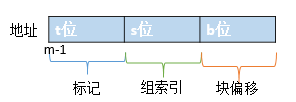
打印输出结果如下：

|  |
| --- |
| 第一级数据缓存信息：CacheLevelInfo [cacheLevel=L1, cacheType=DATA\_CACHE, cacheSets=64, cacheCoherencyLineSize=64, cachePhysicalLinePartitions=1, cacheWaysOfAssociativity=8, isFullyAssociative=false, isSelfInitializing=true, totalSizeInBytes=32768]  第一级指令缓存信息：CacheLevelInfo [cacheLevel=L1, cacheType=INSTRUCTION\_CACHE, cacheSets=64, cacheCoherencyLineSize=64, cachePhysicalLinePartitions=1, cacheWaysOfAssociativity=8, isFullyAssociative=false, isSelfInitializing=true, totalSizeInBytes=32768] |

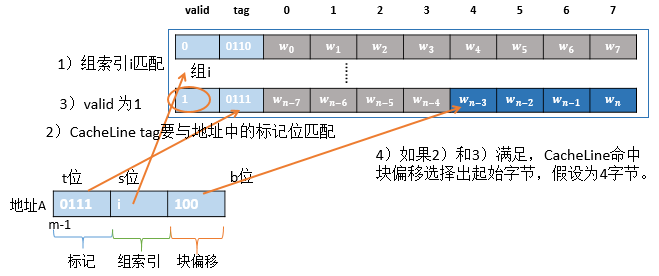
还可以查询L2、L3级缓存的信息，这里不做示例。从打印的信息和CPU-Z显示的信息可以看出，本机的Cache信息是一致的，L1数据/指令缓存大小都为：C=B×E×S=64×8×64=32768字节=32KB。

### Cache Line内存加载

高速缓存的结构将m个地址位分成了t个标记位、s个组索引位和b个块偏移位。参数s和b将m个地址位分为三个字段，如下图：



对于地址A，寻找过程：组索引用来确定地址在Cache中S个组中的哪一个组，标记位告诉我们在组中E行的哪一行，再配合Cache Line中的有效位（valid bit）确定是否有效，根据b位的块偏移找到在Cache Line中的偏移位置。从而在Cache Line中找到A地址的数据。如果Cache Line中，有效位不等于1（该数据失效），或者组中找不到跟标记一致的行，则表示Cache Line Miss，需要到下级缓存中查询加载，如果下层没有则需要到主存加载，主存没有则需要到磁盘，甚至更远的地方加载。越远离CPU的地方代价越大。可见Cache Line寻址是否命中对程序的性能有着极大的影响。



更多关于Cache和Cache相关知识可以参考这两篇博客：

[关于CPU Cache -- 程序猿需要知道的那些事](http://cenalulu.github.io/linux/all-about-cpu-cache/) 、 [Linux Kernel Cache 架构分析](http://blog.sina.com.cn/s/blog_b03dd80301017q1t.html)

## Cache Line的使用

### Cache Line miss问题

如何来提高程序的性能呢？降低Cache Miss率是一个很好的选择。通过前面对Cache存储层次的理解，所谓远水救不了近火，想要提高程序性能，则需要充分利用身边的Cache存储。也就是充分发挥局部性原理，局部性包括时间局部性、空间局部性。时间局部性：对于同一数据可能被多次使用，自第一次加载到Cache Line后，后面的访问就可以多次从Cache Line中命中，从而提高读取速度（而不是从下层缓存读取）。

空间局部性：一个Cache Line有64字节块，我们可以充分利用一次加载64字节的空间，把程序后继会访问的数据，一次性全部加载进来，从而提高Cache Line命中率（而不是重新去寻址读取）。

看个例子：内存地址是连续的数组（利用空间局部性），能一次被L1缓存加载完成。

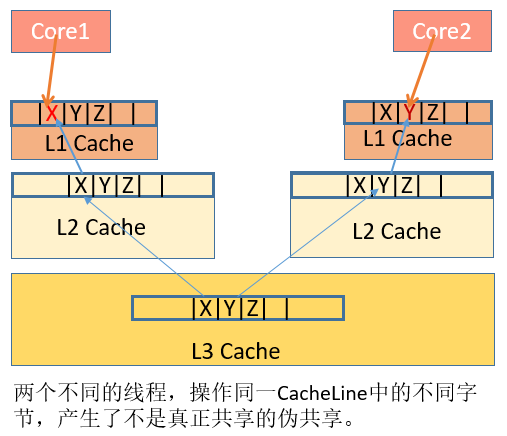
如下代码，长度为16的row和column数组，在Cache Line 64字节数据块上内存地址是连续的，能被一次加载到Cache Line中，所以在访问数组时，Cache Line命中率高，性能发挥到极致。

|  |  |
| --- | --- |
|  | **public** **int** run(**int**[] row, **int**[] column) {  **int** sum = 0;  **for**(**int** i = 0; i < 16; i++ ) {  sum += row[i] \* column[i];  }  **return** sum;  } |

而上面例子中变量i则体现了时间局部性，i作为计数器被频繁操作，一直存放在寄存器中，每次从寄存器访问，而不是从主存甚至磁盘访问。

### Cache Line伪共享

虽然连续紧凑的内存分配带来高性能，但并不代表它一直都能带来高性能。如果把它放在多线程中将会发生什么呢？如图：



数据X、Y、Z被加载到同一Cache Line中，线程A在Core1修改X，线程B在Core2上修改Y。根据MESI大法，假设是Core1是第一个发起操作的CPU核，Core1上的L1 Cache Line由S（共享）状态变成M（修改，脏数据）状态，然后告知其他的CPU核，图例则是Core2，引用同一地址的Cache Line已经无效了；当Core2发起写操作时，先导致Core1将X写回主存，Cache Line状态由M变为I（无效），而后才是Core2从主存重新读取该地址内容，Cache Line状态由I变成E（独占），修改Y Cache Line从E变成M。可见多个线程操作在同一Cache Line上的不同数据，相互竞争同一Cache Line，导致线程彼此牵制影响，变成了串行程序，极大的造成了性能抖动。此时我们则需要将共享在多线程间的数据进行隔离分开，使他们不在同一个Cache Line上，从而提升多线程的性能。

处理伪共享的两种方式：

1. 增大数组元素的间隔使得由不同线程存取的元素位于不同的cache line上。典型的空间换时间。（Linux cache机制与之相关）

2. 在每个线程中创建全局数组各个元素的本地拷贝，然后结束后再写回全局数组

在Java中，JVM对象布局已经对这方面进行了对齐处理，JDK1.8中提供了@ Contended注解采用填充的方式，来进行处理这类问题。

### 跨Cache Line问题

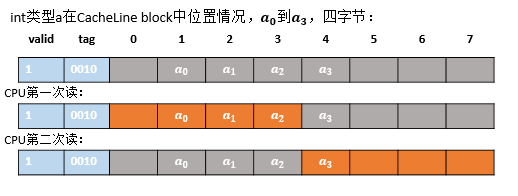
然而并不是我们所有的数据类型和结构都是（int）4个字节大小，更多时候是更大。例如一个普通的java对象，里面包含了如下结构：

|  |  |
| --- | --- |
|  | **public** **class** Student {  **private** Long id;  **private** Long lessonNo;  **private** String name;  **private** Integer studentNo;  **private** String address;  **private** String mobile;  **private** Integer age;  **private** Short score;  **private** Boolean sex;  **private** Date joinTime;  **private** Double height;  **private** Double weight;  **private** Double money;  **private** Long brithday;  **private** Long onlineTime;  } |

不用说，这个Java对象的大小已经超过64字节，当我们去访问这个对象时，一个Cache Line装不下这个对象，就会有两个甚至更多Cache Line来装载，这就造成了，跨Cache Line的问题，性能将大打折扣。在大数据计算、以及java高性能框架，都是自定义数据结构来解决这样的问题。我们也能认识到，java对象里的属性泛滥将会是个恐怖的事情。

### 数据对齐问题

是不是数据结构精简就可以避免跨Cache Line的问题呢？来看看这个例子，int类的a，4字节大小，在Cache Line中起始地址偏移量为1，CPU将如何进行读取呢？如图：



在32位系统下，编译器与cpu等相互协议，CPU寻址的起始地址，只能是0、4、8、12……这类4的倍数。CPU一周期，一次读4字节，而这里的a就需要读取两次了。本来一次性可以读取完成的a由于内存排序问题，起始地址不对齐造成了不必要的资源浪费，大大影响性能、安全问题。有些CPU运行时允许数据不对齐，在Intel中CPU则报错。

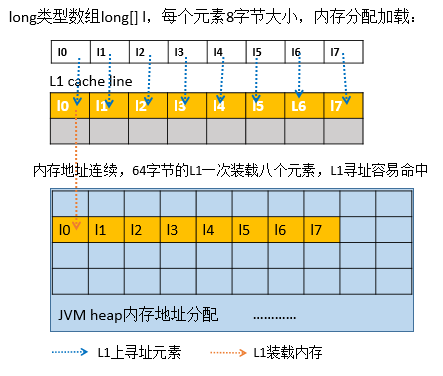
## Cache Line友好型Java程序

### 友好的数据结构设计

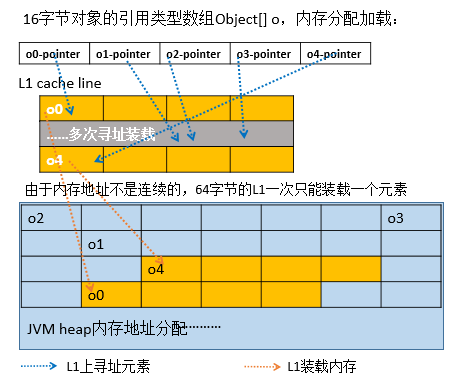
#### Java数组

当以可预测的方式访问存储器时，CPU能够通过预测哪个存储器可能被下一次访问并且在后台将其预取到高速缓存中来，隐藏访问主存储引起的等待时间成本。当然，这只有在处理器可以检测到访问模式（以可预测的“步幅”的访问内存）时才有效。链表，树的数据结构在存储器中更广泛的分布的特点，没有可预测的访问步幅。对数组进行迭代时，步幅是可预测的，可预存在高速缓存行中，达到最大化访问的效率。

在Java中，定义一个long类型数组，每个单位是大小为8byte。一个64byte 的cache line，可以将下标[0…7]的数据放入一个cache line中。避免后继访问频繁的从内存中取值，所以这种情况下连续的存取数据速度会极快。示意如图：



需要注意的是，java分配的引用类型数组地址可能不是连续的。引用类型数组Object[]可能保存在堆中（类成员），也可能保存在线程栈里（作为方法临时变量，指向数组的指针在栈中）。这里分析在堆里的情况，每个元素是对象的指针引用，这些指针是连续的虚拟地址，指针指向堆里的对象，并非真正的内存地址。往缓存加载的时候，相邻的两个对象真实地址可能相隔很远，不会出现在同一缓存行或者相邻的缓存行，迭代时导致多次Cache Line寻址加载。当CPU迭代读取数组元素时，并不能充分利用局部性原理。示意如下图：



所以一般来讲，数组对象是没有缓存友好性的。如果想让对象数组缓存友好，那么需要令所有对象的实际位置在内存中为连续的状态。要达成这个目的有三种办法：

1. 在程序启动时，预先生成数组中引用的对象。增加缓存友好性概率。
2. 使用Unsafe等方式，控制对象在指定的内存位置生成。
3. 不使用对象，全部使用基本类型。

用对象数组和基本类型数组做个比较，示例代码如下：

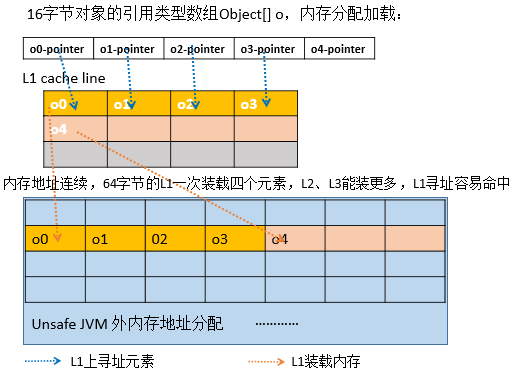
|  |  |
| --- | --- |
|  | **private** **final** **static** **int** ***SIZE*** = 10000000;  **public** **static** **void** main(String[] args){  **int**[] array = **new** **int**[***SIZE***];  **for** (**int** i=0;i<***SIZE***;i++){  array [i] = 65535;  }  **long** startTime = System.*nanoTime*();  **for**(**int** i = 0;i<***SIZE***;i++){  array[i] ++;  }  **long** time = System.*nanoTime*()-startTime;  System.***out***.println("基本类型数组的时间: "+time +" nm");  Integer[] array2 = **new** Integer[***SIZE***];  **for** (**int** i=0;i<***SIZE***;i++){  array2 [i] = **new** Integer(65535);  }  startTime = System.*nanoTime*();  **for**(**int** i = 0;i<***SIZE***;i++){  array[i] ++;  }  time = System.*nanoTime*()-startTime;  System.***out***.println("引用类型数组的时间: "+time +" nm");  } |

笔者机器运行结果为：

|  |  |
| --- | --- |
|  | 基本类型数组的时间: 9367245 nm  引用类型数组的时间: 10768974 nm |

#### Unsafe构建连续地址的引用类型数组

Java sun.misc.Unsafe提供了分配释放内存的API。用unsafe来控制内存，开辟一块连续地址的内存空间，定义固定的元素大小，让对象连续存储，访问时根据起始地址+元素大小\*元素索引来寻找元素。这样可以设计缓存友好的java的对象数组，根据就近原则（空间局部原理）可以节省缓存加载次数，提高缓存命中率。示意如下图：



使用Unsafe生成连续内存地址的示意代码如下，（笔者水平有限，具体代码，读者可以尝试下）：

|  |  |
| --- | --- |
|  | **public** ContinuousArray(Class<T> type, **int** capacity) {  // 计算一个引用对象占空间的大小  **this**.firstFieldOffset = UnsafeHelper.*firstFieldOffset*(type);  **this**.elementsize = UnsafeHelper.*sizeOf*(type) - firstFieldOffset;  **this**.unsafe = UnsafeHelper.*getUnsafe*();  // 分配指定数组大小的内存空间  **this**.type = type;  base = unsafe.allocateMemory(elementsize - capacity);  }  **public** **long** offset(**int** index) {  **return** base + (index \* elementsize);  }  // List接口实现  **public** T set(**int** index, T element) {  unsafe.copyMemory(element, firstFieldOffset, **null**, offset(index), elementsize);  **return** element;  }  // List接口实现  **public** T get(T dest, **int** index) {  unsafe.copyMemory(**null**, offset(index), dest, firstFieldOffset, elementsize);  **return** dest;  }  // List接口实现  **public** T get(**int** index) {  **try** {  **return** get((T) unsafe.allocateInstance(type), index);  } **catch** (InstantiationException e) {  e.printStackTrace();  }  **return** **null**;  } |

这种方式存在的优点很明显：①List接口可以把对象存储在连续的内存地址中；②更好的缓存局部性和cpu性能；③极小的内存开销；

但是它也存在缺点：①使用sun.misc.Unsafe类，它不能被直接引用。②读取设置对象，需要额外的CPU周期来复制对象操作；③复制的对象不受GC垃圾回收管理，因此，如果一个对象仍然被堆外对象引用，垃圾回收器并不知道；

#### 解决伪共享的Cache line友好类

在Java类中，最优化的设计是考虑清楚哪些变量是不变的，哪些是经常变化的，哪些变化是完全相互独立的，哪些属性一起变化。

举个例子：

|  |  |
| --- | --- |
|  | **public** **class** Data{  **long** modifyTime;  **boolean** flag;  **long** createTime;  **char** key;  **int** value;  } |

上述的类满足以下几个特点：

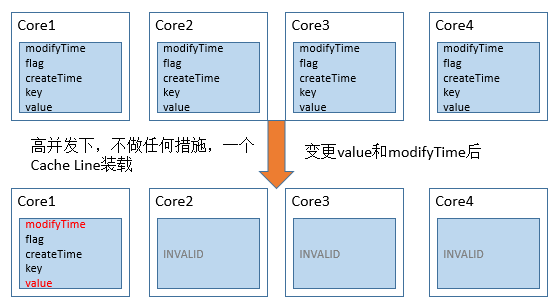
1. 当value变量改变时，modifyTime肯定会改变

2. createTime变量和key变量在创建后，就不会再变化。

3. flag也经常会变化，不过与modifyTime和value变量毫无关联。

当上面的对象需要由多个线程同时的访问时，从Cache角度来说，就会有一些有趣的问题。

当我们没有加任何措施时，Data对象所有的变量极有可能被加载在L1缓存的一行Cache Line中。在高并发访问下，会出现这种问题：



如上图所示，每次value变更时，而且根据MESI协议，对象其他CPU上相关的Cache Line全部被设置为失效。其他的处理器想要访问未变化的数据(key 和 createTime)时，必须从内存中重新拉取数据，增大了数据访问的开销。

正确的方式应该将该对象分组，将一起变化的放在一组，与其他变量无关的变量放到一组，将不变的变量放到一组。这样当每次对象变化时，不会带动所有的变量重新加载缓存，提升了读取效率。

##### Padding方式

在JDK8以前，我们一般是在属性间增加长整型变量来分隔每一组变量，长整型的数量没有特别的限制，只需要长过一个cache line的长度就好：

|  |  |
| --- | --- |
|  | **public** **class** DataPadding{  **long** a1,a2,a3,a4,a5,a6,a7,a8;//防止与前一个对象产生伪共享  **int** value;  **long** modifyTime;  **long** b1,b2,b3,b4,b5,b6,b7,b8;//防止不相关变量伪共享;  **boolean** flag;  **long** c1,c2,c3,c4,c5,c6,c7,c8;//  **long** createTime;  **char** key;  **long** d1,d2,d3,d4,d5,d6,d7,d8;//防止与下一个对象产生伪共享  } |

通过填充变量，使不相关的变量分开

##### Contended注解方式

在JDK8中，新增了一种注解@sun.misc.Contended，来使各个变量在Cache line中分隔开。注意的是在用的时候需要加上jvm参数-XX:-RestrictContended才能开启此功能

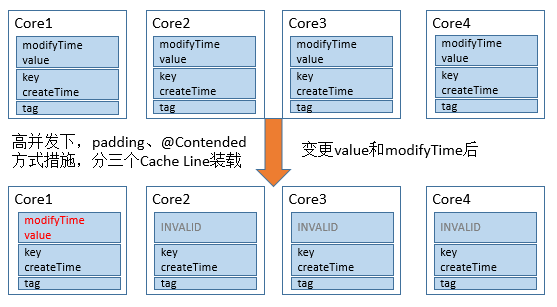
用时，可以在类前或属性前加上此注释：

|  |  |
| --- | --- |
|  | // 类前加上代表整个类的每个变量都会在单独的cache line中  @sun.misc.Contended  @SuppressWarnings("restriction")  **public** **class** ContendedData {  **int** value;  **long** modifyTime;  **boolean** flag;  **long** createTime;  **char** key;  } |

或者这种：

|  |  |
| --- | --- |
|  | // 属性前加上时需要加上组标签  @SuppressWarnings("restriction")  **public** **class** ContendedGroupData {  @sun.misc.Contended("group1")  **int** value;  @sun.misc.Contended("group1")  **long** modifyTime;  @sun.misc.Contended("group2")  **boolean** flag;  @sun.misc.Contended("group3")  **long** createTime;  @sun.misc.Contended("group3")  **char** key;  } |

采取上述措施图示：



### 友好的逻辑控制

既然知道了数据体的设计，那么在逻辑控制方面也有对Cache Line局部性的利用和破坏。

#### 循环的步长

数组是一个天然Cache Line友好型的数据结构，而且通常大部分程序中大部分时间花在少量的核心函数，核心函数大部分时间花在少量循环上。这里就用数组和循环来进行逻辑上的分析。

如下代码，a是一个int二维数组，下面是两种方式来进行访问数组中的元素：

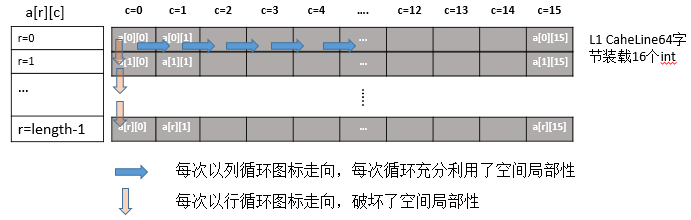
第一种，先遍历行，再遍历列

|  |  |
| --- | --- |
|  | **int**[][] a = **new** **int**[1024][64\*1024\*1024];  **for**(**int** r = 0; r < a.length; r++) { // first loop row  **for**(**int** c = 0; c < a [r].length; c++ ) { // loop column  a [r][c] ++;  }  } |

第二种，先遍历列，再遍历行

|  |  |
| --- | --- |
|  | **int**[][] a = **new** **int**[1024][64\*1024\*1024];  **for**(**int** c = 0; c < a [0].length; c++) { // first loop column  **for**(**int** r = 0; r < a.length; r++) { // loop row  a [r][c] ++;  }  } |

通过对比，第一种方式比第二种要快。为什么呢？聪明的你一定会想到前面提到过的局部性原理。没错，就是空间局部性原理。虽然两个循环的下标都是步长为1，绝大部分数底层中，默认按照行的方式来加载数据，这就造成了这两种方式的差异。



第一种方式的内循环，下标c递增访问的是已经加载到Cache Line上的数据，一个64字节的Cache Line已经将后续需要访问的数据加载完成了（Cache Hit局部性原理），只需要访问L1 Cache就能命中数据。

第二种方式看似是步长为1，其实是步长为array[0].length，也就是列的长度，内循环是一个跨array[0].length步长的访问，如果length大于16，那么一行数据就会占据一行Cache Line。每次循环每行Cache Line只问一次，下次就跳到下一行Cache Line中去访问相同的列。前面说过Cache Line是按照行来加载数据的，所以第后面的循环需要重新去下级缓存中加载数据（Cache Miss打破了局部性原理）。

同样，对于int类型一维数组，步长为16，同样是打破了局部性原理。可见循环体中对步长的正确理解和使用非常重要。

#### 循环体的优化

提高循环体操作并发，即循环体内的操作不是相互依赖的，可以进行CPU重排序。

精简循环体代码，代码放入在缓存指令中的，如果代码大小超过缓存大小，则失去缓存的优势。（时间局部性）

#### 局部变量代替全局字段的访问

另外对于经常使用全局静态变量的地方，使用局部变量代替。方法内的局部变量在栈上，全局静态变量存在全局数据段，频繁访问全局静态变量比访问栈上（cache line上）局部变量慢很多。以前不明白，开源框架中用一个局部变量来接受全局变量，总觉得直接使用全局变量来的爽快，现在明白了，原来是考虑时间局部性。

## Java中对Cache line经典设计

### Contended注解在JDK1.8的应用

#### ConcurrentHashMap

java.util.concurrent.ConcurrentHashMap在这个如雷贯耳的数据结构中，有一个很基本的操作问题，在并发条件下进行++操作。因为++这个操作并不是原子的，而且在连续的Atomic中，很容易产生伪共享（false shareing）。所以在其内部有专门的数据结构来保存long型的数据:（openjdk\jdk\src\share\classes\java\util\concurrent\ConcurrentHashMap.java line:2506）：

|  |  |
| --- | --- |
|  | /\* ---------------- Counter support -------------- \*/  /\*\*  \* A padded cell for distributing counts. Adapted from LongAdder  \* and Striped64. See their internal docs for explanation.  \*/  @sun.misc.Contended **static** **final** **class** CounterCell {  **volatile** **long** value;  CounterCell(**long** x) { value = x; }  } |

在该类中，则是通过@sun.misc.Contended达到防止false shareing的目的

#### Thread

java.lang.Thread在java中，生成随机数是和线程有着关联。而且在很多情况下，多线程下产生随机数的操作是很常见的，JDK为了确保产生随机数的操作不会产生false sharing ,把产生随机数的三个相关值设为独占cache line。

（openjdk\jdk\src\share\classes\java\lang\Thread.java line:2023）

|  |  |
| --- | --- |
|  | // The following three initially uninitialized fields are exclusively  // managed by class java.util.concurrent.ThreadLocalRandom. These  // fields are used to build the high-performance PRNGs in the  // concurrent code, and we can not risk accidental false sharing.  // Hence, the fields are isolated with @Contended.  /\*\* The current seed for a ThreadLocalRandom \*/  @sun.misc.Contended("tlr")  **long** threadLocalRandomSeed;  /\*\* Probe hash value; nonzero if threadLocalRandomSeed initialized \*/  @sun.misc.Contended("tlr")  **int** threadLocalRandomProbe;  /\*\* Secondary seed isolated from public ThreadLocalRandom sequence \*/  @sun.misc.Contended("tlr")  **int** threadLocalRandomSecondarySeed; |

### Disruptor框架

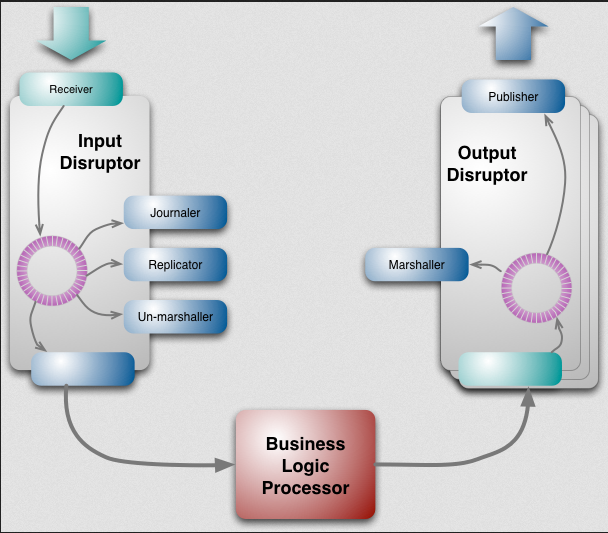
#### 认识Disruptor

##### Disruptor业务背景

LMAX是在英国注册并受到FCA监管的外汇黄金交易所。也是欧洲第一家也是唯一一家采用多边交易设施Multilateral Trading Facility（MTF）拥有交易所牌照和经纪商牌照的欧洲顶级金融公司。LMAX是一种新型零售金融交易平台，它能够以很低的延迟(latency)产生大量交易(吞吐量)。这个系统是建立在JVM平台上，核心是一个业务逻辑处理器，它能够在一个线程里每秒处理6百万订单。业务逻辑处理器完全是运行在内存中(in-memory)，使用事件源驱动方式(event sourcing)。业务逻辑处理器的核心是[Disruptor](http://lmax-exchange.github.io/disruptor/)（注，本文Disruptor基于当前最新3.3.6版本），这是一个并发组件，能够在无锁的情况下实现网络的Queue并发操作。

##### Disruptor运行流程分析

Disruptor是一个线程内通信框架，用于线程里共享数据。与LinkedBlockingQueue类似，提供了一个高速的生产者消费者模型，广泛用于批量IO读写，在硬盘读写相关的程序中应用的十分广泛，Apache旗下的[HBase、Hive、Storm](http://mvnrepository.com/artifact/com.lmax/disruptor/usages)等框架都有在使用Disruptor。LMAX 创建Disruptor作为可靠消息架构的一部分，并将它设计成一种在不同组件中共享数据非常快的方法。Disruptor框架中，业务逻辑处理器高性能的关键是按单线程顺序地做事。Disruptor运行大致流程入下图：



图中左侧（Input Disruptor部分）可以看作多生产者单消费者模式。外部多个线程作为多生产者并发请求业务逻辑处理器（Business Logic Processor），这些请求的信息经过Receiver存放在粉红色的圆环中，业务处理器则作为消费者从圆环中取得数据进行处理。右侧（Output Disruptor部分）则可看作单生产者多消费者模式。业务逻辑处理器作为单生产者，发布数据到粉红色圆环中，Publisher作为多个消费者接受业务逻辑处理器的结果。这里两处地方的数据共享都是通过那个粉红色的圆环，它就是Disruptor的核心设计RingBuffer。

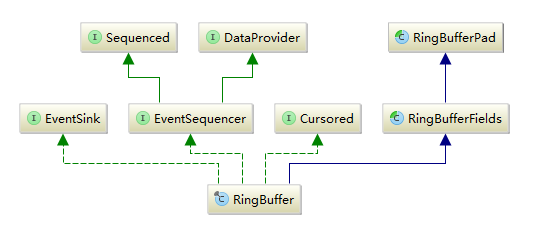
##### Disruptor特点

* 1. 无锁机制（确保任何数据只由一个线程拥有以进行写访问，从而消除写争用的设计， 这种设计被称作“破坏者”，也是这样命名这个框架的）。
  2. 没有CAS操作，避免了内存屏障指令的耗时。
  3. 便是避开了Cache line伪共享的问题，也是本文主要关注的主题

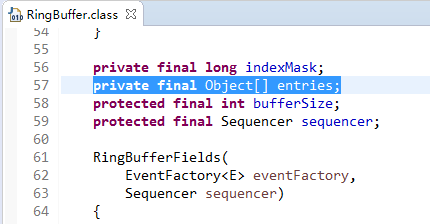
#### Disruptor对伪共享的处理

##### RingBuffer类

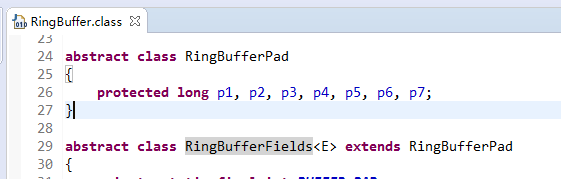
RingBuffer类（即上节中粉红色的圆环）的类关系图如下：

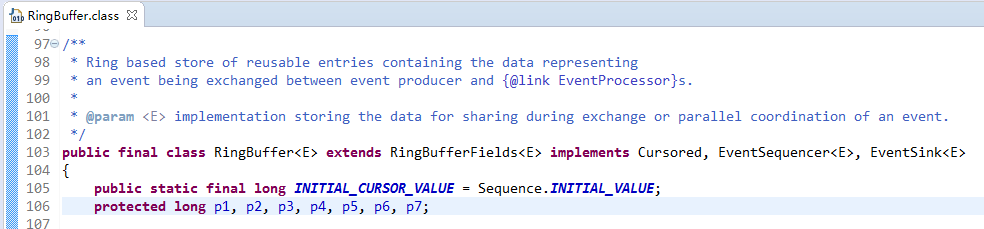


通过源码分析，RingBuffer的父类，RingBufferFields采用数组来实现存放线程间的共享数据。下图，第57行，entries数组。

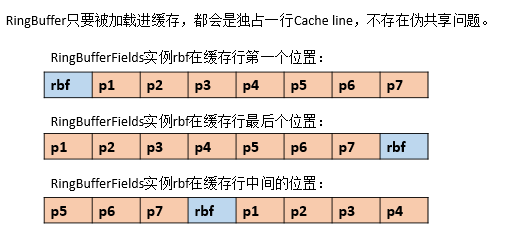


前面分析过数组比链表、树更具有缓存友好性，此处不做细表。不使用LinkedBlockingQueue队列，是基于无锁机制的考虑。详细分析可参考，[并发编程网的翻译](http://ifeve.com/locks-are-bad/)。这里我们主要分析RingBuffer的继承关系中的填充，解决缓存伪共享问题。如下图：





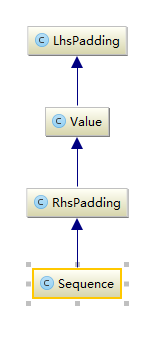
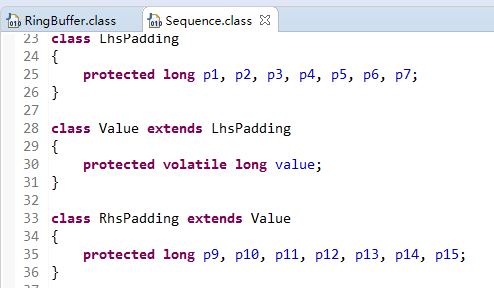
RingBufferPad的protected long P1,p2,p3,p4,p5,p6,p7;作为缓存前置填充，RingBuffer中的protected long P1,p2,p3,p4,p5,p6,p7;作为缓存后置填充。这样不管任意线程访问RingBuffer时，RingBuffer被操作的字段RingBufferFields都是独占一行Cache line不会产生伪共享问题。示意如图：



按照一行缓存64字节计算，前后填充56字节（7个long），中间大于等于8字节的内容都能独占一行Cache line，此处rbf大于8字节的。

##### Sequence类

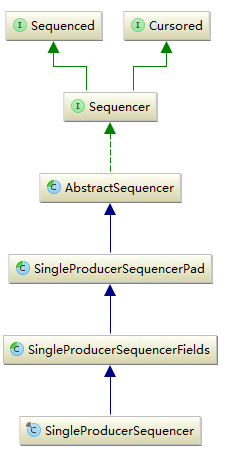
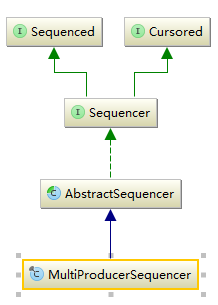
Sequence类用来跟踪RingBuffer和事件处理器的增长步数，支持多个并发操作包括CAS指令和写指令。同时使用了Padding方式来实现，如下为其类结构图及Padding的类。

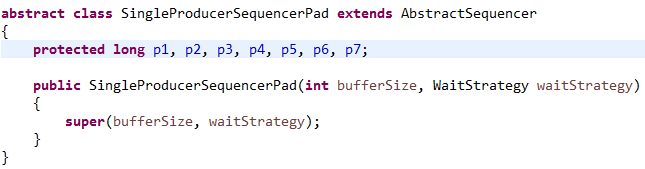
Sequence里在volatile long value前各放置了7个long padding，来解决伪共享的问题。

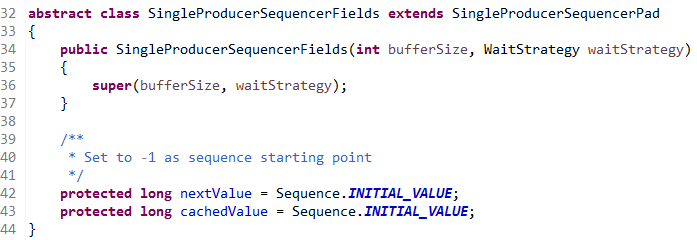
##### Sequencer的实现

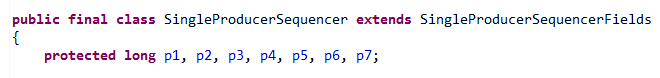
在RingBuffer构造函数里面存在一个Sequencer接口，用来遍历数据，在生产者和消费者之间传递数据。Sequencer有两个实现类，单生产者模式的实现SingleProducerSequencer与多生产者模式的实现MultiProducerSequencer。它们的类结构如图：

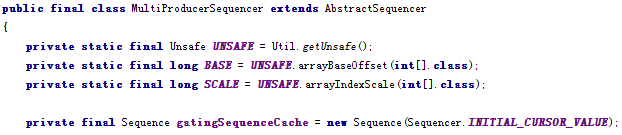
单生产者是在Cache line中使用pading方式实现，源码如下：







多生产者则是使用 sun.misc.Unsafe来实现的。如下图：



#### 总结与使用示例

可见padding方式在Disruptor中是处理伪共享常见的方式，JDK8的@Contended很好的解决了这个问题，不知道Disruptor后面的版本是否会考虑使用它。

Disruptor使用示例代码[参考地址](https://github.com/EasonFeng5870/disruptor_demo)。

## 参考资料:

7个示例科普CPU Cache：<http://coolshell.cn/articles/10249.html>

Linux Cache 机制：<http://www.cnblogs.com/liloke/archive/2011/11/20/2255737.html>

《深入理解计算机系统》：第六章部分

Disruptor官方文档：<https://github.com/LMAX-Exchange/disruptor/tree/master/docs>

并发编程网文档翻译：<http://ifeve.com/disruptor/>