

# 聊聊JVM内存布局

苏三说技术 2022-03-23 19:11

以下文章来源于bin的技术小屋，作者bin的技术小屋



## bin的技术小屋

专注源码解析系列原创技术文章，分享自己的技术感悟。谈笑有鸿儒，往来无白丁。无丝竹之乱耳，无...

大家好，我是苏三，又到了每周我们见面的时刻了，我的公众号在1月10号那天发布了第一篇文章？[《从内核角度看IO模型的演变》](#)，在这篇文章中我们通过图解的方式以一个C10k的问题为主线，从内核角度详细阐述了5种IO模型的演变过程，以及两种IO线程模型的介绍，最后引出了Netty的网络IO线程模型。读者朋友们后台留言都觉得非常的硬核，在大家的支持下这篇文章的目前阅读量为2038，点赞量为80，在看为32。这对于刚刚诞生一个多月的小号来说，是一种莫大的鼓励。在这里bin再次感谢大家的认可，鼓励和支持~~

今天bin将再来为大家带来一篇硬核的技术文章，本文我们将从计算机组成原理的角度详细阐述对象在JVM内存中是如何布局的，以及什么是内存对齐，如果我们头比较铁，就是不进行内存对齐会造成什么样的后果，最后引出压缩指针的原理和应用。同时我们还介绍了在高并发场景下，False Sharing产生的原因以及带来的性能影响。

相信大家看完本文后，一定会收获很多，话不多说，下面我们正式开始本文的内容~~

mindmaster

### 一. Java对象的内存布局

对象头 (Header)

实例数据 (Instance Data)

### 二. 字段重排列

开启压缩指针，开启字段压缩

开启压缩指针，关闭字段压缩

关闭压缩指针，开启字段压缩

关闭压缩指针，关闭字段压缩

### 三. 对齐填充 (Padding)

解决伪共享问题带来的对齐填充

CPU缓存

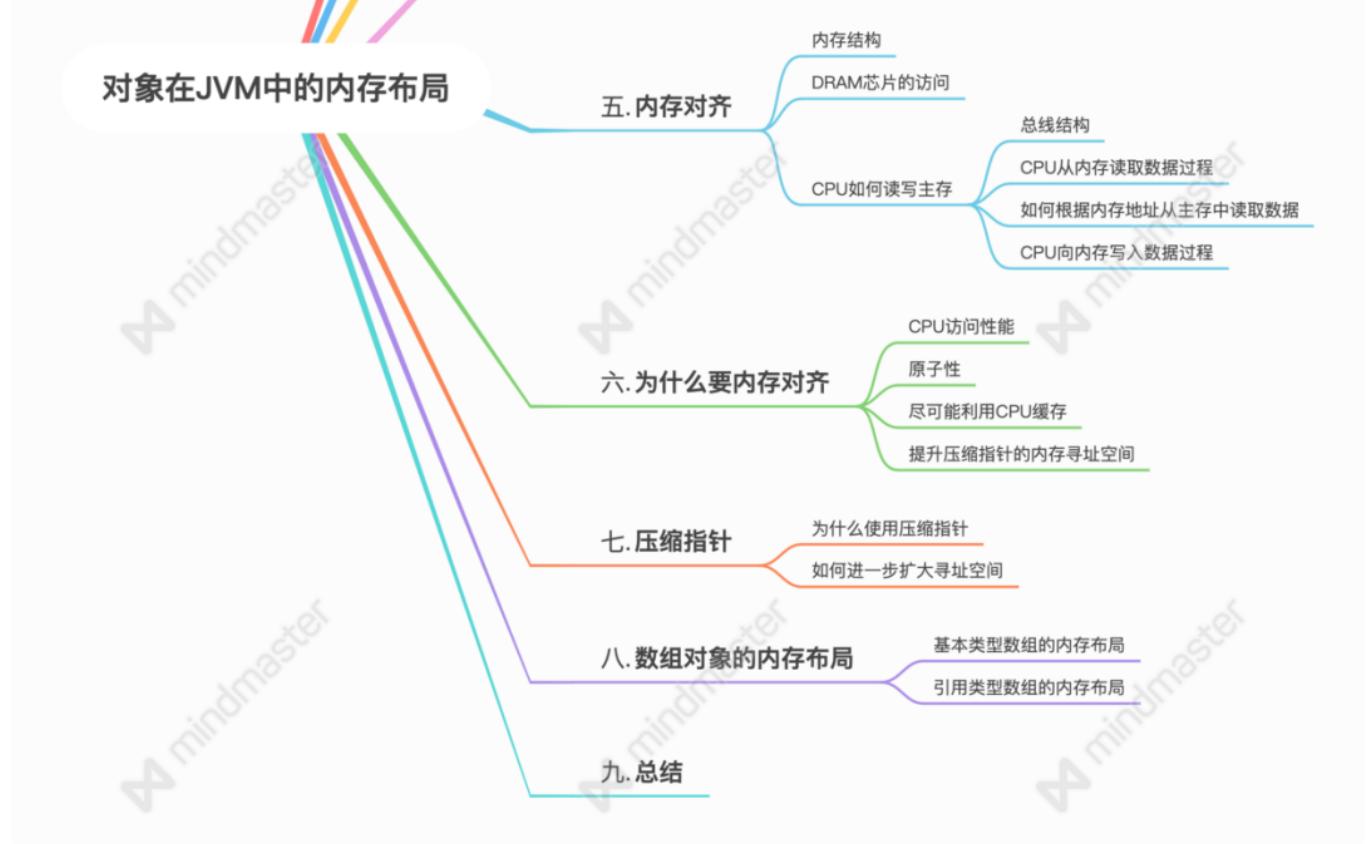
CPU缓存行

False Sharing (伪共享)

False Sharing的解决方案

@Contended注解

### 四. 对齐填充的应用



本文概要.png

在我们的日常工作中，有时候我们为了防止线上应用发生 OOM，所以我们需要在开发的过程中计算一些核心对象在内存中的占用大小，目的是为了更好的了解我们的应用程序内存占用的一个大概情况。

进而根据我们服务器的内存资源限制以及预估的对象创建数量级计算出应用程序占用内存的高低水位线，如果内存占用量超过 高水位线，那么就有可能有发生 OOM 的风险。

我们可以在程序中根据估算出的 高低水位线，做一些防止 OOM 的处理逻辑或者发岀告警。

那么核心问题是如何计算一个Java对象在内存中的占用大小呢？

在为大家解答这个问题之前，笔者先来介绍下Java对象在内存中的布局，也就是本文的主题。

## 1. Java对象的内存布局





Java对象的内存布局.png

如图所示，Java对象在JVM中是用 instanceOopDesc 结构表示而Java对象在JVM堆中的内存布局可以分为三部分：

## 1.1 对象头 (Header)

每个Java对象都包含一个对象头，对象头中包含了两类信息：

- MarkWord：在JVM中用 markOopDesc 结构表示用于存储对象自身运行时的数据。比如：hashcode，GC分代年龄，锁状态标志，线程持有的锁，偏向线程Id，偏向时间戳等。在32位操作系统和64位操作系统中 MarkWord 分别占用4B和8B大小的内存。
- 类型指针：JVM中的类型指针封装在 klassOopDesc 结构中，类型指针指向了 InstanceKlass 对象，Java类在JVM中是用 InstanceKlass 对象封装的，里边包含了Java类的元信息，比如：继承结构，方法，静态变量，构造函数等。
  - 在不开启指针压缩的情况下(-XX:-UseCompressedOops)。在32位操作系统和64位操作系统中类型指针分别占用4B和8B大小的内存。
  - 在开启指针压缩的情况下(-XX:+UseCompressedOops)。在32位操作系统和64位操作系统中类型指针分别占用4B和4B大小的内存。
- 如果Java对象是一个数组类型的话，那么在数组对象的对象头中还会包含一个4B大小的用于记录数组长度的属性。

由于在对象头中用于记录数组长度大小的属性只占4B的内存，所以Java数组可以申请的最大长度为： $2^{32}$ 。

## 1.2 实例数据 (Instance Data)

Java对象在内存中的实例数据区用来存储Java类中定义的实例字段，包括所有父类中的实例字段。也就是说，虽然子类无法访问父类的私有实例字段，或者子类的实例字段隐藏了父类的同名实例字段，但是子类的实例还是会为这些父类实例字段分配内存。

Java对象中的字段类型分为两大类：

- 基础类型：Java类中实例字段定义的基础类型在实例数据区的内存占用如下：

- long | double 占用8个字节。
- int | float 占用4个字节。
- short | char 占用2个字节。
- byte | boolean 占用1个字节。

- 引用类型：Java类中实例字段的引用类型在实例数据区内内存占用分为两种情况：

- 不开启指针压缩(-XX:-UseCompressedOops)：在32位操作系统中引用类型的内存占用为4个字节。在64位操作系统中引用类型的内存占用为8个字节。
- 开启指针压缩(-XX:+UseCompressedOops)：在64位操作系统下，引用类型内存占用则变为4个字节，32位操作系统中引用类型的内存占用继续为8个字节。

## 为什么32位操作系统的引用类型占4个字节，而64位操作系统引用类型占8字节？

在Java中，引用类型所保存的是被引用对象的内存地址。在32位操作系统中内存地址是由32个bit表示，因此需要4个字节来记录内存地址，能够记录的虚拟地址空间是 $2^{32}$ 大小，也就是只能够表示4G大小的内存。

而在64位操作系统中内存地址是由64个bit表示，因此需要8个字节来记录内存地址，但在 64 位系统里只使用了低 48 位，所以它的虚拟地址空间是  $2^{48}$  大小，能够表示 256T 大小的内存，其中低 128T 的空间划分为用户空间，高 128T 划分为内核空间，可以说是非常大了。

在我们从整体上介绍完Java对象在JVM中的内存布局之后，下面我们来看下Java对象中定义的这些实例字段在实例数据区是如何排列布局的：

## 2. 字段重排列

其实我们在编写Java源代码文件的时候定义的那些实例字段的顺序会被JVM重新分配排列，这样做的目的其实是为了内存对齐，那么什么是内存对齐，为什么要进行内存对齐，笔者会随着文章深入的解读为大家逐层揭晓答案~~

本小节中，笔者先来为大家介绍一下JVM字段重排列的规则：

JVM重新分配字段的排列顺序受 -XX:FieldsAllocationStyle 参数的影响，默认值为 1，实例字段的重新分配策略遵循以下规则：

1. 如果一个字段占用 X 个字节，那么这个字段的偏移量 OFFSET 需要对齐至 NX

偏移量是指字段的内存地址与Java对象的起始内存地址之间的差值。比如long类型的字段，它内存占用8个字节，那么它的OFFSET应该是8的倍数8N。不足8N的需要填充字节。

- 在开启了压缩指针的64位JVM中，Java类中的第一个字段的OFFSET需要对齐至4N，在关闭压缩指针的情况下类中第一个字段的OFFSET需要对齐至8N。
- JVM默认分配字段的顺序为：long / double, int / float, short / char, byte / boolean, oops(Ordinary Object Point 引用类型指针)，**并且父类中定义的实例变量会出现在子类实例变量之前**。当设置JVM参数 -XX +CompactFields 时（默认），占用内存小于long / double的字段会允许被插入到对象中第一个 long / double字段之前的间隙中，以避免不必要的内存填充。

CompactFields选项参数在JDK14中以被标记为过期了，并在将来的版本中很可能被删除。详细细节可查看 issue：<https://bugs.openjdk.java.net/browse/JDK-8228750>

上边的三条字段重排列规则非常非常重要，但是读起来比较绕脑，很抽象不容易理解，笔者把它们先列出来的目的是为了让大家先有一个朦朦胧胧的感性认识，下面笔者举一个具体的例子来为大家详细说明下，在阅读这个例子的过程中也方便大家深刻的理解这三条重要的字段重排列规则。

假设现在我们有这样一个类定义



```
public class Parent {  
    long l;  
    int i;  
}  
  
public class Child extends Parent {  
    long l;  
    int i;  
}
```

- 根据上面介绍的规则3我们知道父类中的变量是出现在子类变量之前的，并且字段分配顺序应该是long型字段l，应该在int型字段i之前。

如果JVM开启了 -XX +CompactFields 时，int型字段是可以插入对象中的第一个long型字段（也就是Parent.l字段）之前的空隙中的。如果JVM设置了 -XX -CompactFields 则int型字段的这种插入行为是不被允许的。

- 根据规则1我们知道long型字段在实例数据区的OFFSET需要对齐至8N，而int型字段的OFFSET需要对齐至4N。

- 根据规则2我们知道如果开启压缩指针 -XX:+UseCompressedOops，Child对象的第一个字段的OFFSET需要对齐至4N，关闭压缩指针时 -XX:-UseCompressedOops，Child对象的第一个字段的OFFSET需要对齐至8N。

由于JVM参数 UseCompressedOops 和 CompactFields 的存在，导致Child对象在实例数据区字段的排列顺序分为四种情况，下面我们结合前边提炼出的这三点规则来看下字段排列顺序在这四种情况下的表现。

## 2.1 -XX:+UseCompressedOops -XX -CompactFields 开启压缩指针，关闭字段压缩

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)	43 c0 00 f8 (01000011 11000000 00000000 11111000) (-134168509)
12	4	(alignment/padding gap)	
16	8	long Parent.l	0
24	4	int Parent.i	0
28	4	(alignment/padding gap)	
32	8	long Child.l	0
40	4	int Child.i	0
44	4	(loss due to the next object alignment)	
Instance size <b>48 bytes</b>			
Space losses: 8 bytes internal + 4 bytes external = 12 bytes total			

image.png

- 偏移量OFFSET = 8的位置存放的是类型指针，由于开启了压缩指针所以占用4个字节。对象头总共占用12个字节：MarkWord(8字节) + 类型指针(4字节)。
- 根据规则3：父类Parent中的字段是要出现在子类Child的字段之前的并且long型字段在int型字段之前。
- 根据规则2：在开启压缩指针的情况下，Child对象中的第一个字段需要对齐至4N。这里Parent.l字段的OFFSET可以是12也可以是16。
- 根据规则1：long型字段在实例数据区的OFFSET需要对齐至8N，所以这里Parent.l字段的OFFSET只能是16，因此OFFSET = 12的位置就需要被填充。Child.l字段只能在OFFSET = 32处存储，不能够使用OFFSET = 28位置，因为28的位置不是8的倍数无法对齐8N，因此OFFSET = 28的位置被填充了4个字节。

规则1也规定了int型字段的OFFSET需要对齐至4N，所以Parent.i与Child.i分别存储以OFFSET = 24和OFFSET = 40的位置。

因为JVM中的内存对齐除了存在于字段与字段之间还存在于对象与对象之间，Java对象之间的内存地址需要对齐至8N。

所以Child对象的末尾处被填充了4个字节，对象大小由开始的44字节被填充到48字节。

## 2.2 -XX:+UseCompressedOops -XX +CompactFields 开启压缩指针，开启字段压缩

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)	43 c0 00 f8 (01000011 11000000 00000000 11111000) (-134168509)
12	4	int Parent.i	0
16	8	long Parent.l	0
24	8	long Child.l	0
32	4	int Child.i	0
36	4	(loss due to the next object alignment)	

Instance size: 40 bytes  
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

image.png

- 在第一种情况的分析基础上，我们开启了 -XX +CompactFields 压缩字段，所以导致int型的Parent.i 字段可以插入到OFFSET = 12的位置处，以避免不必要的字节填充。
- 根据规则2： Child对象的第一个字段需要对齐至4N，这里我们看到 int型 的Parent.i字段是符合这个规则的。
- 根据规则1： Child对象的所有long型字段都对齐至8N，所有的int型字段都对齐至4N。

最终得到Child对象大小为36字节，由于Java对象与对象之间的内存地址需要对齐至8N，所以最后Child 对象的末尾又被填充了4个字节最终变为40字节。

这里我们可以看到在开启字段压缩 -XX +CompactFields 的情况下，Child对象的大小由48字节变成了40字节。

## 2.3 -XX:-UseCompressedOops -XX -CompactFields 关闭压缩指针，关闭字段压缩

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)	a0 67 dc 0d (10100000 01100111 11011100 00001101) (232548256)
12	4	(object header)	01 00 00 00 (00000001 00000000 00000000 00000000) (1)
16	8	long Parent.l	0
24	4	int Parent.i	0
28	4	(alignment/padding gap)	
32	8	long Child.l	0
40	4	int Child.i	0
44	4	(loss due to the next object alignment)	

Instance size: 48 bytes  
Space losses: 4 bytes internal + 4 bytes external = 8 bytes total

image.png

首先在关闭压缩指针 -UseCompressedOops 的情况下，对象头中的类型指针占用字节变成了8字节。导致对象头的大小在这种情况下变为了16字节。

- 根据规则1： long型的变量OFFSET需要对齐至8N。根据规则2： 在关闭压缩指针的情况下，Child对象的第一个字段Parent.l需要对齐至8N。所以这里的Parent.l字段的OFFSET = 16。

- 由于long型的变量OFFSET需要对齐至8N，所以Child.i字段的OFFSET 需要是32，因此OFFSET = 28的位置被填充了4个字节。

这样计算出来的Child对象大小为44字节，但是考虑到Java对象与对象的内存地址需要对齐至8N，于是又在对象末尾处填充了4个字节，最终Child对象的内存占用为48字节。

## 2.4 -XX:-UseCompressedOops -XX +CompactFields 关闭压缩指针，开启字段压缩

在第三种情况的分析基础上，我们来看下第四种情况的字段排列情况：

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)	a0 07 0c 14 (10100000 00000111 00001100 00010100) (336332704)
12	4	(object header)	01 00 00 00 (00000001 00000000 00000000 00000000) (1)
16	8	long Parent.l	0
24	4	int Parent.i	0
28	4	(alignment/padding gap)	
32	8	long Child.l	0
40	4	int Child.i	0
44	4	(loss due to the next object alignment)	
Instance size: 48 bytes			
Space losses: 4 bytes internal + 4 bytes external = 8 bytes total			

image.png

由于在关闭指针压缩的情况下类型指针的大小变为了8个字节，所以导致Child对象中第一个字段Parent.l前边并没有空隙，刚好对齐8N，并不需要int型变量的插入。所以即使开启了字段压缩 -XX +CompactFields，字段的总体排列顺序还是不变的。

默认情况下指针压缩 -XX:+UseCompressedOops 以及字段压缩 -XX +CompactFields 都是开启的

## 3. 对齐填充（Padding）

在前一小节关于实例数据区字段重排列的介绍中为了内存对齐而导致的字节填充不仅会出现在字段与字段之间，还会出现在对象与对象之间。

前边我们介绍了字段重排列需要遵循的三个重要规则，其中规则1，规则2定义了字段与字段之间的内存对齐规则。规则3定义的是对象字段之间的排列规则。

为了内存对齐的需要，对象头与字段之间，以及字段与字段之间需要填充一些不必要的字节。

比如前边提到的字段重排列的第一种情况 -XX:+UseCompressedOops -XX -CompactFields。

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)	43 c0 00 f8 (01000011 11000000 00000000 11111000) (-134168509)
12	4	(alignment/padding gap)	
16	8	long Parent.l	0
24	4	int Parent.i	0
28	4	(alignment/padding gap)	
32	8	long Child.l	0

```
40    4    int Child.i          0
44    4    {loss due to the next object alignment}
Instance size: 48 bytes
Space losses: 8 bytes internal + 4 bytes external = 12 bytes total
```

image.png

而以上提到的四种情况都会在对象实例数据区的后边在填充4字节大小的空间，原因是除了需要满足字段与字段之间的内存对齐之外，还需要满足对象与对象之间的内存对齐。

Java 虚拟机堆中对象之间的内存地址需要对齐至 $8N$ （8的倍数），如果一个对象占用内存不到 $8N$ 个字节，那么就必须在对象后填充一些不必要的字节对齐至 $8N$ 个字节。

虚拟机中内存对齐的选项为 -XX:ObjectAlignmentInBytes，默认为8。也就是说对象与对象之间的内存地址需要对齐至多少倍，是由这个JVM参数控制的。

我们还是以上边第一种情况为例说明：图中对象实际占用是44个字节，但是不是8的倍数，那么就需要再填充4个字节，内存对齐至48个字节。

以上这些为了内存对齐的目的而在字段与字段之间，对象与对象之间填充的不必要字节，我们就称之为对齐填充（Padding）。

## 4. 对齐填充的应用

在我们知道对齐填充的概念之后，大家可能好奇了，为啥我们要进行对齐填充，是要解决什么问题吗？

那就让我们带着这个问题，来接着听笔者往下聊~~

### 4.1 解决伪共享问题带来的对齐填充

除了以上介绍的两种对齐填充的场景（字段与字段之间，对象与对象之间），在JAVA中还有一种对齐填充的场景，那就是通过对齐填充的方式来解决 False Sharing（伪共享） 的问题。

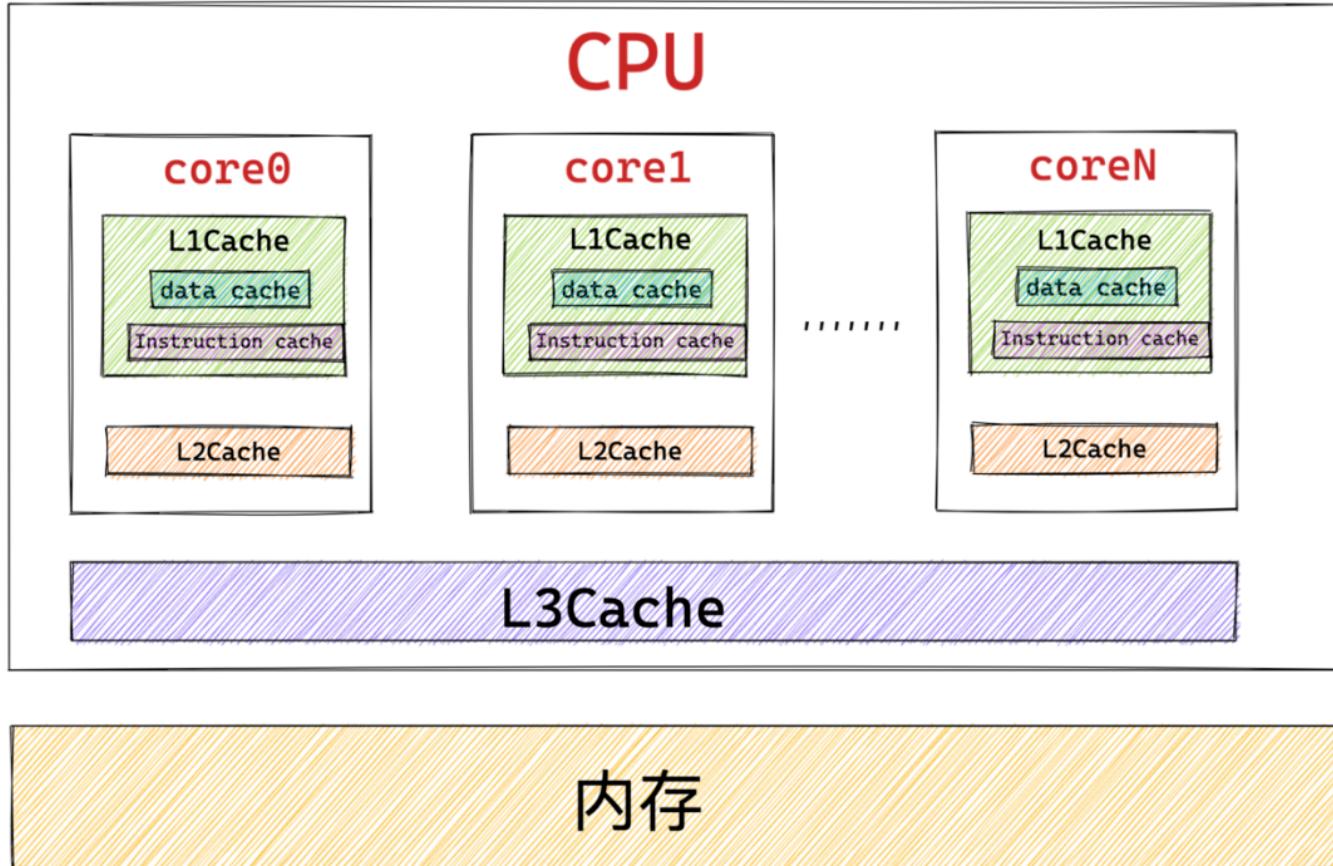
在介绍False Sharing（伪共享）之前，笔者先来介绍下CPU读取内存中数据的方式。

#### 4.1.1 CPU缓存

根据摩尔定律：芯片中的晶体管数量每隔 18 个月就会翻一番。导致CPU的性能和处理速度变得越来越快，而提升CPU的运行速度比提升内存的运行速度要容易和便宜的多，所以就导致了CPU与内存之间的速度差距越来越大。

为了弥补CPU与内存之间巨大的速度差异，提高CPU的处理效率和吞吐，于是人们引入了 L1, L2, L3 高

速缓存集成到CPU中。当然还有L0也就是寄存器，寄存器离CPU最近，访问速度也最快，基本没有时延。



CPU缓存结构.png

一个CPU里面包含多个核心，我们在购买电脑的时候经常会看到这样的处理器配置，比如4核8线程。意思是这个CPU包含4个物理核心8个逻辑核心。4个物理核心表示在同一时间可以允许4个线程并行执行，8个逻辑核心表示处理器利用超线程的技术将一个物理核心模拟出了两个逻辑核心，一个物理核心在同一时间只会执行一个线程，而超线程芯片可以做到线程之间快速切换，当一个线程在访问内存的空隙，超线程芯片可以马上切换去执行另外一个线程。因为切换速度非常快，所以在效果上看到是8个线程在同时执行。

图中的CPU核心指的是物理核心。

从图中我们可以看到L1Cache是离CPU核心最近的高速缓存，紧接着就是L2Cache，L3Cache，内存。

离CPU核心越近的缓存访问速度也越快，造价也就越高，当然容量也就越小。

其中L1Cache和L2Cache是CPU物理核心私有的（注意：这里是物理核心不是逻辑核心）

而L3Cache是整个CPU所有物理核心共享的。

CPU逻辑核心共享其所属物理核心的L1Cache和L2Cache

## L1Cache

L1Cache离CPU是最近的，它的访问速度最快，容量也最小。

从图中我们看到L1Cache分为两个部分，分别是：Data Cache和Instruction Cache。它们一个是存储数据的，一个是存储代码指令的。

我们可以通过 [cd /sys/devices/system/cpu/](#) 来查看linux机器上的CPU信息。

```
[root@13ed1f378d52 index1] cd /sys/devices/system/cpu/  
[root@13ed1f378d52 cpu]# ls  
cpu0  cpu2  cpufreq  hotplug  kernel_max  offline  possible  present  uevent  
cpu1  cpu3  cpuidle  isolated  modalias  online  power    smt      vulnerabilities  
[root@13ed1f378d52 cpu]#
```

image.png

在 [/sys/devices/system/cpu/](#) 目录里，我们可以看到CPU的核心数，当然这里指的是逻辑核心。

笔者机器上的处理器并没有使用超线程技术所以这里其实是4个物理核心。

下面我们进入其中一颗CPU核心（cpu0）中去看下L1Cache的情况：

CPU缓存的情况在 [/sys/devices/system/cpu/cpu0/cache](#) 目录下查看：

```
[root@13ed1f378d52 cache]# ls  
index0  index1  index2  index3  index4  uevent
```

image.png

[index0](#) 描述的是L1Cache中DataCache的情况：

```
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index0/level  
1  
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index0/type  
Data  
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index0/size  
32K  
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list  
0
```

image.png

- [level](#)：表示该cache信息属于哪一级，1表示L1Cache。
- [type](#)：表示属于L1Cache的DataCache。
- [size](#)：表示DataCache的大小为32K。
- [shared\\_cpu\\_list](#)：之前我们提到L1Cache和L2Cache是CPU物理核所私有的，而由物理核模拟出来的逻辑核是

共享L1Cache和L2Cache的，</sys/devices/system/cpu/> 目录下描述的信息是逻辑核。shared\_cpu\_list描述的是哪些逻辑核共享这个物理核。

index1 描述的是L1Cache中Instruction Cache的情况：

```
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index1/level
1
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index1/type
Instruction
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index1/size
32K
```

image.png

我们看到L1Cache中的Instruction Cache大小也是32K。

## L2Cache

L2Cache的信息存储在 index2 目录下：

```
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index2/level
2
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index2/type
Unified
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index2/size
256K
```

image.png

L2Cache的大小为256K，比L1Cache要大些。

## L3Cache

L3Cache的信息存储在 index3 目录下：

```
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index3/level
3
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index3/type
Unified
[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index3/size
6144K
```

image.png

到这里我们可以看到L1Cache中的DataCache和InstructionCache大小一样都是32K而L2Cache的大小为256K， L3Cache的大小为6M。

当然这些数值在不同的CPU配置上会是不同的，但是总体上来说L1Cache的量级是几十KB，L2Cache的量级是几百KB，L3Cache的量级是几MB。

## 4.1.2 CPU缓存行

前边我们介绍了CPU的高速缓存结构，引入高速缓存的目的在于消除CPU与内存之间的速度差距，根据程序的局部性原理我们知道，CPU的高速缓存肯定是用来存放热点数据的。

程序局部性原理表现为：时间局部性和空间局部性。时间局部性是指如果程序中的某条指令一旦执行，则不久之后该指令可能再次被执行；如果某块数据被访问，则不久之后该数据可能再次被访问。空间局部性是指一旦程序访问了某个存储单元，则不久之后，其附近的存储单元也将被访问。

那么在高速缓存中存取数据的基本单位又是什么呢？？

事实上热点数据在CPU高速缓存中的存取并不是我们想象中的以单独的变量或者单独的指针为单位存取的。

CPU高速缓存中存取数据的基本单位叫做缓存行 cache line。缓存行存取字节的大小为2的倍数，在不同的机器上，缓存行的大小范围在32字节到128字节之间。目前所有主流的处理器中缓存行的大小均为 64字节（注意：这里的单位是字节）。

```
[[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size  
64  
[[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index1/coherency_line_size  
64  
[[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index2/coherency_line_size  
64  
[[root@13ed1f378d52 index0]# cat /sys/devices/system/cpu/cpu0/cache/index3/coherency_line_size  
64
```

image.png

从图中我们可以看到L1Cache,L2Cache,L3Cache中缓存行的大小都是 64字节。

这也就意味着每次CPU从内存中获取数据或者写入数据的大小为64个字节，即使你只读一个bit，CPU也会从内存中加载64字节数据进来。同样的道理，CPU从高速缓存中同步数据到内存也是按照64字节的单位来进行。

比如你访问一个long型数组，当CPU去加载数组中第一个元素时也会同时将后边的7个元素一起加载进缓存中。这样以来就加快了遍历数组的效率。

long类型在Java中占用8个字节，一个缓存行可以存放8个long型变量。

事实上，你可以非常快速的遍历在连续的内存块中分配的任意数据结构，如果你的数据结构中的项在内存中不是彼此相邻的（比如：链表），这样就无法利用CPU缓存的优势。由于数据在内存中不是连续存

放的，所以在这些数据结构中的每一个项都可能会出现缓存行未命中（程序局部性原理）的情况。

还记得我们在？《Reactor在Netty中的实现(创建篇)》中介绍Selector的创建时提到，Netty利用数组实现的自定义SelectedSelectionKeySet类型替换掉了JDK利用HashSet类型实现的sun.nio.ch.SelectorImpl#selectedKeys。目的就是利用CPU缓存的优势来提高IO活跃的SelectionKeys集合的遍历性能。

## 4.2 False Sharing (伪共享)

我们先来看一个这样的例子，笔者定义了一个示例类FalseSharding，类中有两个long型的volatile字段a, b。



```
public class FalseSharding {  
  
    volatile long a;  
  
    volatile long b;  
  
}
```

字段a, b之间逻辑上是独立的，它们之间一点关系也没有，分别用来存储不同的数据，数据之间也没有关联。

FalseSharding类中字段之间的内存布局如下：

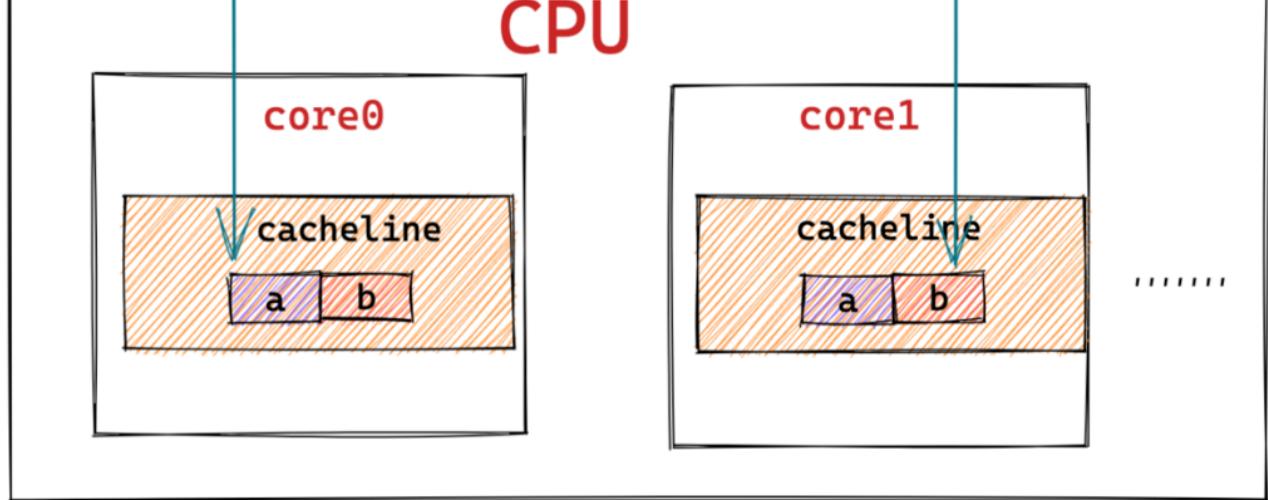
OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4		(object header)	01 00 00 00 (00000001 00000000 00000000 00000000) (1)
4	4		(object header)	00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8	4		(object header)	05 c0 00 f8 (00000101 11000000 00000000 11111000) (-134168571)
12	4		(alignment/padding gap)	0
16	8	long	FalseSharding.a	0
24	8	long	FalseSharding.b	0

image.png

FalseSharding类中的字段a,b在内存中是相邻存储，分别占用8个字节。

如果恰好字段a, b被CPU读进了同一个缓存行，而此时有两个线程，线程a用来修改字段a，同时线程b用来读取字段b。





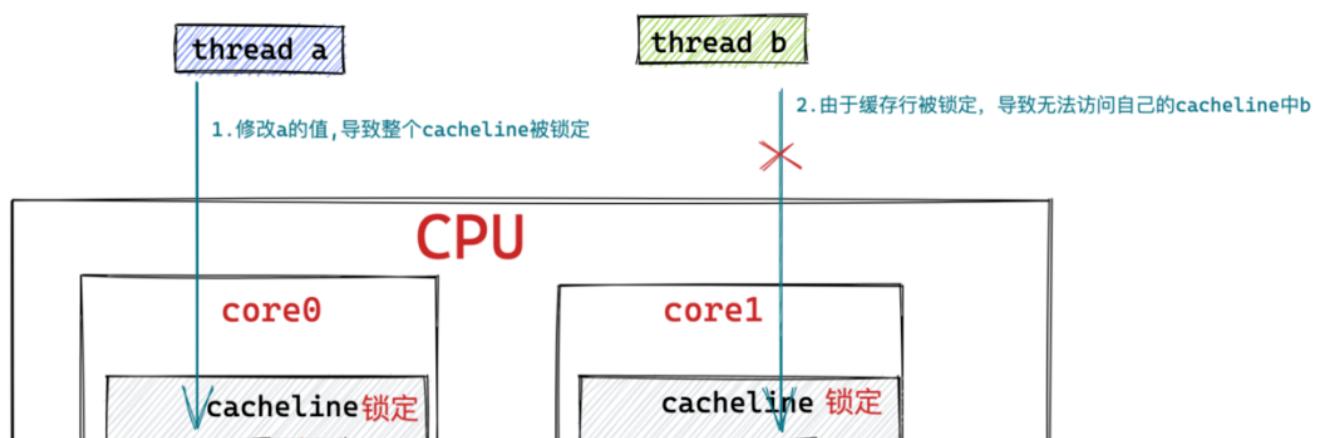
falsesharding1.png

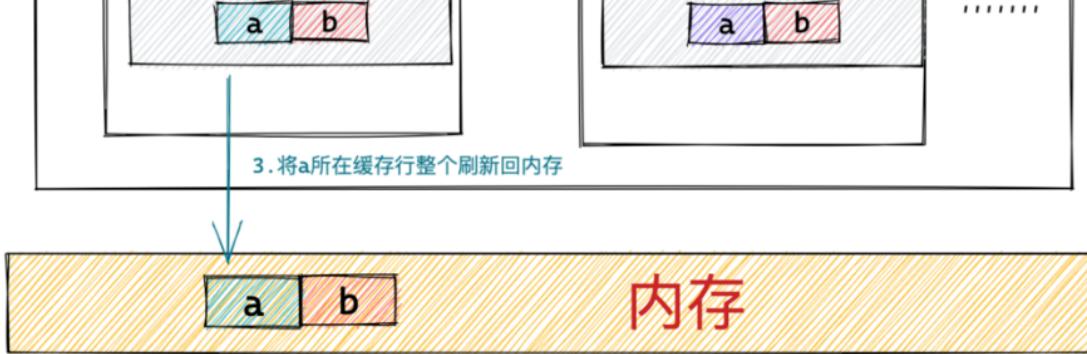
在这种场景下，会对线程b的读取操作造成什么影响呢？

我们知道声明了 volatile关键字 的变量可以在多线程处理环境下，确保内存的可见性。计算机硬件层会保证对被 volatile 关键字修饰的共享变量进行写操作后的内存可见性，而这种内存可见性是由 Lock前缀指令 以及 缓存一致性协议（MESI控制协议） 共同保证的。

- Lock前缀指令可以使修改线程所在的处理器中的相应缓存行数据被修改后立马刷新回内存中，并同时 锁定 所有处理器核心中缓存了该修改变量的缓存行，防止多个处理器核心并发修改同一缓存行。
- 缓存一致性协议主要是用来维护多个处理器核心之间的CPU缓存一致性以及与内存数据的一致性。每个处理器会在总线上嗅探其他处理器准备写入的内存地址，如果这个内存地址在自己的处理器中被缓存的话，就会将自己处理器中对应的缓存行置为 无效，下次需要读取的该缓存行中的数据的时候，就需要访问内存获取。

基于以上 volatile 关键字原则，我们首先来看第一种影响：





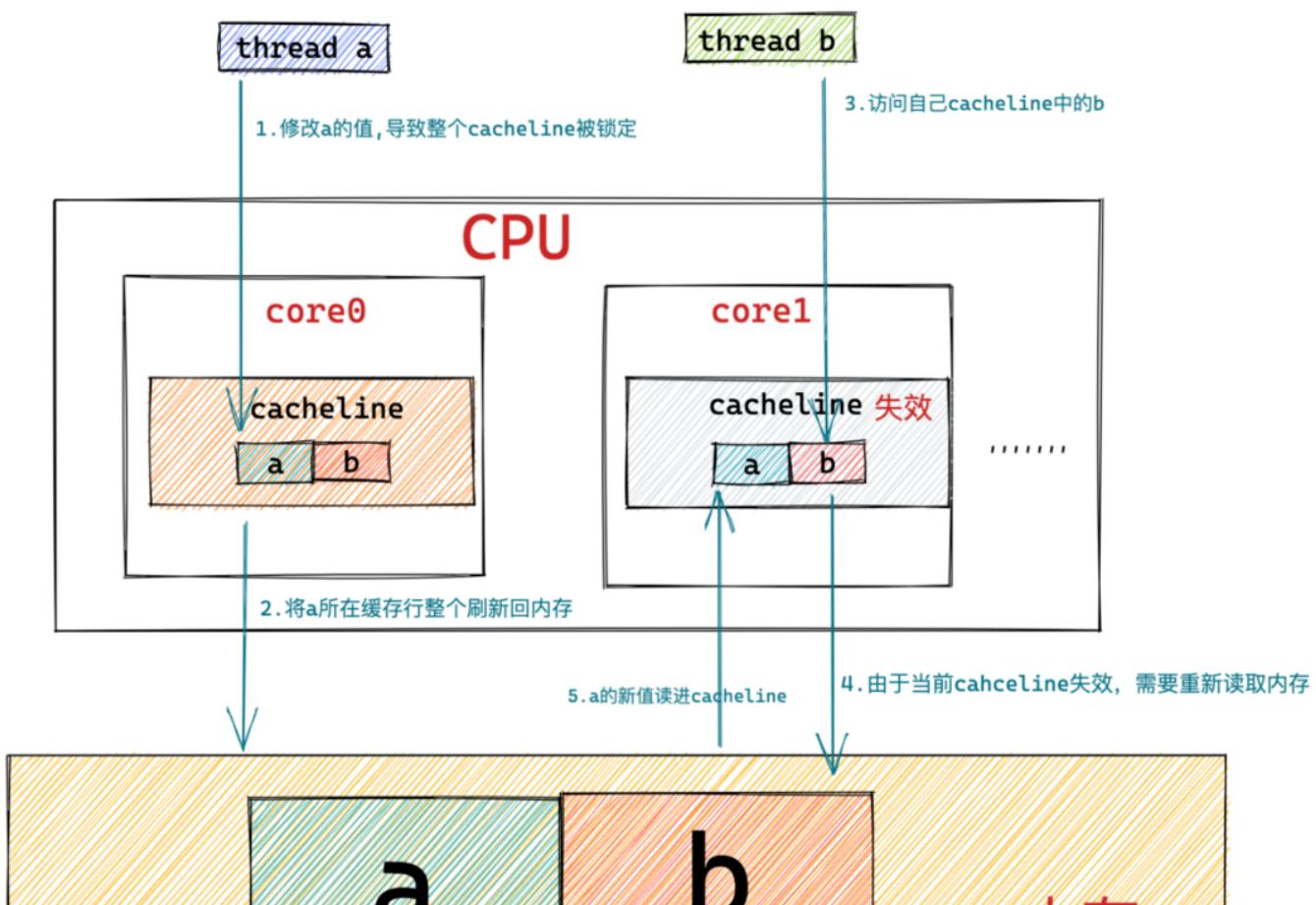
falsesharding2.png

- 当线程a在处理器core0中对字段a进行修改时，Lock前缀指令会将所有处理器中缓存了字段a的对应缓存行进行锁定，这样就会导致线程b在处理器core1中无法读取和修改自己缓存行的字段b。
- 处理器core0将修改后的字段a所在的缓存行刷新回内存中。

从图中我们可以看到此时字段a的值在处理器core0的缓存行中以及在内存中已经发生变化了。但是处理器core1中字段a的值还没有变化，并且core1中字段a所在的缓存行处于锁定状态，无法读取也无法写入字段b。

从上述过程中我们可以看出即使字段a, b之间逻辑上是独立的，它们之间一点关系也没有，但是线程a对字段a的修改，导致了线程b无法读取字段b。

## 第二种影响：





faslesharding3.png

当处理器core0将字段a所在的缓存行刷新回内存的时候，处理器core1会在总线上嗅探到字段a的内存地址正在被其他处理器修改，所以将自己的缓存行置为失效。当线程b在处理器core1中读取字段b的值时，发现缓存行已被置为失效，core1需要重新从内存中读取字段b的值即使字段b没有发生任何变化。

从以上两种影响我们看到字段a与字段b实际上并不存在共享，它们之间也没有相互关联关系，理论上线程a对字段a的任何操作，都不应该影响线程b对字段b的读取或者写入。

但事实上线程a对字段a的修改导致了字段b在core1中的缓存行被锁定（Lock前缀指令），进而使得线程b无法读取字段b。

线程a所在处理器core0将字段a所在缓存行同步刷新回内存后，导致字段b在core1中的缓存行被置为失效（缓存一致性协议），进而导致线程b需要重新回到内存读取字段b的值无法利用CPU缓存的优势。

由于字段a和字段b在同一个缓存行中，导致了字段a和字段b事实上的共享（原本是不应该被共享的）。这种现象就叫做False Sharing（伪共享）。

在高并发的场景下，这种伪共享的问题，会对程序性能造成非常大的影响。

如果线程a对字段a进行修改，与此同时线程b对字段b也进行修改，这种情况对性能的影响更大，因为这会导致core0和core1中相应的缓存行相互失效。

### 4.3 False Sharing的解决方案

既然导致False Sharing出现的原因是字段a和字段b在同一个缓存行导致的，那么我们就要想办法让字段a和字段b不在一个缓存行中。

那么我们怎么做才能够使得字段a和字段b一定不会被分配到同一个缓存行中呢？

这时候，本小节的主题字节填充就派上用场了~~

在Java8之前我们通常会在字段a和字段b前后分别填充7个long型变量（缓存行大小64字节），目的是让字段a和字段b各自独占一个缓存行避免False Sharing。

比如我们将一开始的实例代码修改成这个样子，就可以保证字段a和字段b各自独占一个缓存行了。

```

public class FalseSharding {
    long p1,p2,p3,p4,p5,p6,p7;
    volatile long a;
    long p8,p9,p10,p11,p12,p13,p14;
    volatile long b;
    long p15,p16,p17,p18,p19,p20,p21;
}

```

修改后的对象在内存中布局如下：

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)	05 c0 00 f8 (00000101 11000000 00000000 11111000) (-134168571)
12	4	(alignment/padding gap)	
16	8	long FalseSharding.p1	0
24	8	long FalseSharding.p2	0
32	8	long FalseSharding.p3	0
40	8	long FalseSharding.p4	0
48	8	long FalseSharding.p5	0
56	8	long FalseSharding.p6	0
64	8	long FalseSharding.p7	0
72	8	long FalseSharding.a	0
80	8	long FalseSharding.p8	0
88	8	long FalseSharding.p9	0
96	8	long FalseSharding.p10	0
104	8	long FalseSharding.p11	0
112	8	long FalseSharding.p12	0
120	8	long FalseSharding.p13	0
128	8	long FalseSharding.p14	0
136	8	long FalseSharding.b	0
144	8	long FalseSharding.p15	0
152	8	long FalseSharding.p16	0
160	8	long FalseSharding.p17	0
168	8	long FalseSharding.p18	0
176	8	long FalseSharding.p19	0
184	8	long FalseSharding.p20	0
192	8	long FalseSharding.p21	0
Instance size: 200 bytes			

image.png

我们看到为了解决False Sharing问题，我们将原本占用32字节的FalseSharding示例对象硬生生的填充到了200字节。这对内存的消耗是非常可观的。通常为了极致的性能，我们会在一些高并发框架或者JDK的源码中看到False Sharing的解决场景。因为在高并发场景中，任何微小的性能损失比如False Sharing，都会被无限放大。

但解决False Sharing的同时又会带来巨大的内存消耗，所以即使在高并发框架比如disrupter或者JDK中也只是针对那些在多线程场景下被频繁写入的共享变量。

这里笔者想强调的是在我们日常工作中，我们不能因为自己手里拿着锤子，就满眼都是钉子，看到任何钉子都想去锤两下。





image.png

我们要清晰的分辨出一个问题会带来哪些影响和损失，这些影响和损失在我们当前业务阶段是否可以接受？是否是瓶颈？同时我们也要清晰的了解要解决这些问题我们所要付出的代价。一定要综合评估，讲究一个投入产出比。某些问题虽然是问题，但是在某些阶段和场景下并不需要我们投入解决。而有些问题则对于我们当前业务发展阶段是瓶颈，我们不得不去解决。我们在架构设计或者程序设计中，方案一定要简单，合适。并预估一些提前量留有一定的演化空间。

#### 4.3.1 @Contended注解

在Java8中引入了一个新注解@Contended，用于解决False Sharing的问题，同时这个注解也会影响到Java对象中的字段排列。

在上一小节的内容介绍中，我们通过手段填充字段的方式解决了False Sharing的问题，但是这里也有一个问题，因为我们在手动填充字段的时候还需要考虑CPU缓存行的大小，因为虽然现在所有主流的处理器缓存行大小均为64字节，但是也还是有处理器的缓存行大小为32字节，有的甚至是128字节。我们需要考虑很多硬件的限制因素。

Java8中通过引入@Contended注解帮我们解决了这个问题，我们不在需要去手动填充字段了。下面我们就来看下@Contended注解是如何帮助我们来解决这个问题的~~

上小节介绍的手动填充字节是在共享变量前后填充64字节大小的空间，这样只能确保程序在缓存行大小为32字节或者64字节的CPU下独占缓存行。但是如果CPU的缓存行大小为128字节，这样依然存在False Sharing的问题。

引入@Contended注解可以使我们忽略底层硬件设备的差异性，做到Java语言的初衷：平台无关性。

@Contended注解默认只是在JDK内部起作用，如果我们的程序代码中需要使用到@Contended注解，那么需要开启JVM参数 -XX:-RestrictContended 才会生效。

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE})
public @interface Contended {
    //contention group tag
    String value() default "";
}

```

@Contended注解可以标注在类上也可以标注在类中的字段上，被@Contended标注的对象会独占缓存行，不会和任何变量或者对象共享缓存行。

- @Contended标注在类上表示该类对象中的实例数据整体需要独占缓存行。不能与其他实例数据共享缓存行。
- @Contended标注在类中的字段上表示该字段需要独占缓存行。
- 除此之外@Contended还提供了分组的概念，注解中的value属性表示contention group。属于统一分组下的变量，它们在内存中是连续存放的，可以允许共享缓存行。不同分组之间不允许共享缓存行。

下面我们来分别看下@Contended注解在这三种使用场景下是怎样影响字段之间的排列的。

### @Contended标注在类上

```
@Contended
```

```

public class FalseSharding {
    volatile long a;
    volatile long b;

    volatile int c;
    volatile int d;
}

```

当@Contended标注在FalseSharding示例类上时，表示FalseSharding示例对象中的整个实例数据区需要独占缓存行，不能与其他对象或者变量共享缓存行。

这种情况下的内存布局：

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)	05 c0 00 f8 (00000101 11000000 00000000 11111000) (-134168571)
12	128		(alignment/padding gap)	
140	4	int	FalseSharding.c	0
144	8	long	FalseSharding.a	0

```
152    8    long FalseSharding.b          0  
160    8    long FalseSharding.d          0  
168  128    (loss due to the next object alignment)  
Instance size: 296 bytes
```

image.png

如图中所示，FalseSharding示例类被标注了@Contended之后，JVM会在FalseSharding示例对象的实例数据区前后填充128个字节，保证实例数据区内的字段之间内存是连续的，并且保证整个实例数据区独占缓存行，不会与实例数据区之外的数据共享缓存行。

细心的朋友可能已经发现了问题，我们之前不是提到缓存行的大小为64字节吗？为什么这里会填充128字节呢？

而且之前介绍的手动填充也是填充的64字节，为什么@Contended注解会采用两倍的缓存行大小来填充呢？

其实这里的原因有两个：

- 首先第一个原因，我们之前也已经提到过了，目前大部分主流的CPU缓存行是64字节，但是也有部分CPU缓存行是32字节或者128字节，如果只填充64字节的话，在缓存行大小为32字节和64字节的CPU中是可以做到独占缓存行从而避免FalseSharding的，但在缓存行大小为128字节的CPU中还是会出现FalseSharding问题，这里Java采用了悲观的一种做法，默认都是填充128字节，虽然对于大部分情况下比较浪费，但是屏蔽了底层硬件的差异。

不过@Contended注解填充字节的大小我们可以通过JVM参数 -XX:ContendedPaddingWidth 指定，有效值范围0 - 8192，默认为128。

- 第二个原因其实是最为核心的一个原因，主要是为了防止CPU Adjacent Sector Prefetch (CPU相邻扇区预取) 特性所带来的FalseSharding问题。

CPU Adjacent Sector Prefetch: <https://www.techarp.com/bios-guide/cpu-adjacent-sector-prefetch/>

**CPU Adjacent Sector Prefetch**是Intel处理器特有的BIOS功能特性，默认是enabled。主要作用就是利用程序局部性原理，当CPU从内存中请求数据，并读取当前请求数据所在缓存行时，会进一步预取与当前缓存行相邻的下一个缓存行，这样当我们的程序在顺序处理数据时，会提高CPU处理效率。这一点也体现了程序局部性原理中的空间局部性特征。

当CPU Adjacent Sector Prefetch特性被disabled禁用时，CPU就只会获取当前请求数据所在的缓存行，不会预取下一个缓存行。

所以在当**CPU Adjacent Sector Prefetch**启用 (enabled) 的时候，CPU其实同时处理的是两个缓存行，在这种情况下，就需要填充两倍缓存行大小 (128字节) 来避免CPU Adjacent Sector Prefetch所带来的的FalseSharding问题。

@Contended标注在字段上



```

public class FalseSharding {

    @Contended
    volatile long a;

    @Contended
    volatile long b;

    volatile int c;
    volatile long d;
}

```

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)	05 c0 00 f8 (00000101 11000000 00000000 11111000) (-134168571)
12	4	int FalseSharding.c	0
16	8	long FalseSharding.d	0
24	128	(alignment/padding gap)	
152	8	long FalseSharding.a	0
160	128	(alignment/padding gap)	
288	8	long FalseSharding.b	0
296	128	(loss due to the next object alignment)	

Instance size: 424 bytes  
Space lost due to internal + external = 284 bytes total

image.png

这次我们将 `@Contended`注解标注在了`FalseSharding`示例类中的字段`a`和字段`b`上，这样带来的效果是字段`a`和字段`b`各自独占缓存行。从内存布局上看，字段`a`和字段`b`前后分别被填充了128个字节，来确保字段`a`和字段`b`不与任何数据共享缓存行。

而没有被`@Contended`注解标注字段`c`和字段`d`则在内存中连续存储，可以共享缓存行。

## @Contended分组

```

public class FalseSharding {

    @Contended("group1")
    volatile int a;

    @Contended("group1")
    volatile long b;

    @Contended("group2")
    volatile long c;
    @Contended("group2")
    volatile long d;
}

```

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)	05 c0 00 f8 (00000101 11000000 00000000 11111000) (-134168571)
12	128	(alignment/padding gap)	
140	4	int FalseSharding.a	0
144	8	long FalseSharding.b	0
152	128	(alignment/padding gap)	
280	8	long FalseSharding.c	0
288	8	long FalseSharding.d	0
296	128	(loss due to the next object alignment)	
Instance size: 424 bytes			
Space losses: 256 bytes internal + 128 bytes external = 384 bytes total			

image.png

这次我们将字段a与字段b放在同一content group下，字段c与字段d放在另一个content group下。

这样处在同一分组 group1 下的字段a与字段b在内存中是连续存储的，可以共享缓存行。

同理处在同一分组group2下的字段c与字段d在内存中也是连续存储的，也允许共享缓存行。

但是分组之间是不能共享缓存行的，所以在字段分组的前后各填充 128字节，来保证分组之间的变量不能共享缓存行。

## 5. 内存对齐

通过以上内容我们了解到Java对象中的实例数据区字段需要进行内存对齐而导致在JVM中会被重排列以及通过填充缓存行避免false sharding的目的所带来的字节对齐填充。

我们也了解到内存对齐不仅发生在对象与对象之间，也发生在对象中的字段之间。

那么在本小节中笔者将为大家介绍什么是内存对齐，在本节的内容开始之前笔者先来抛出两个问题：

- 为什么要进行内存对齐？如果就是头比较铁，就是不内存对齐，会产生什么样的后果？
- Java 虚拟机堆中对象的起始地址为什么需要对齐至 8 的倍数？为什么不对齐至4的倍数或16的倍数或32的倍数呢？

带着这两个问题，下面我们正式开始本节的内容~~~

### 5.1 内存结构

我们平时所称的内存也叫随机访问存储器（random-access memory）也叫RAM。而RAM分为两类：

- 一类是静态RAM (SRAM)，这类SRAM用于前边介绍的CPU高速缓存L1Cache, L2Cache, L3Cache。其特点是访问速度快，访问速度为1 - 30个时钟周期，但是容量小，造价高。
- 另一类则是动态RAM(DRAM)，这类DRAM用于我们常说的主存上，其特点的是访问速度慢（相对高

速缓存），访问速度为 50 - 200 个时钟周期，但是容量大，造价便宜些（相对高速缓存）。

内存由一个一个的存储器模块（memory module）组成，它们插在主板的扩展槽上。常见的存储器模块通常以64位为单位（8个字节）传输数据到存储控制器上或者从存储控制器传出数据。

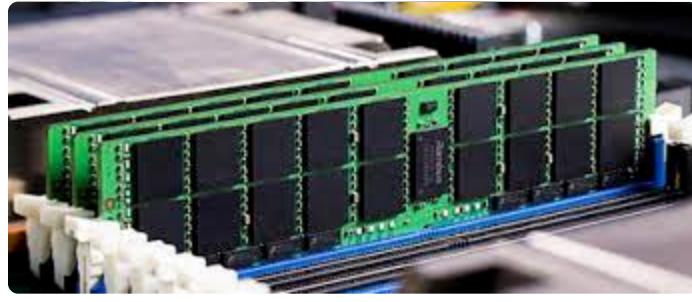
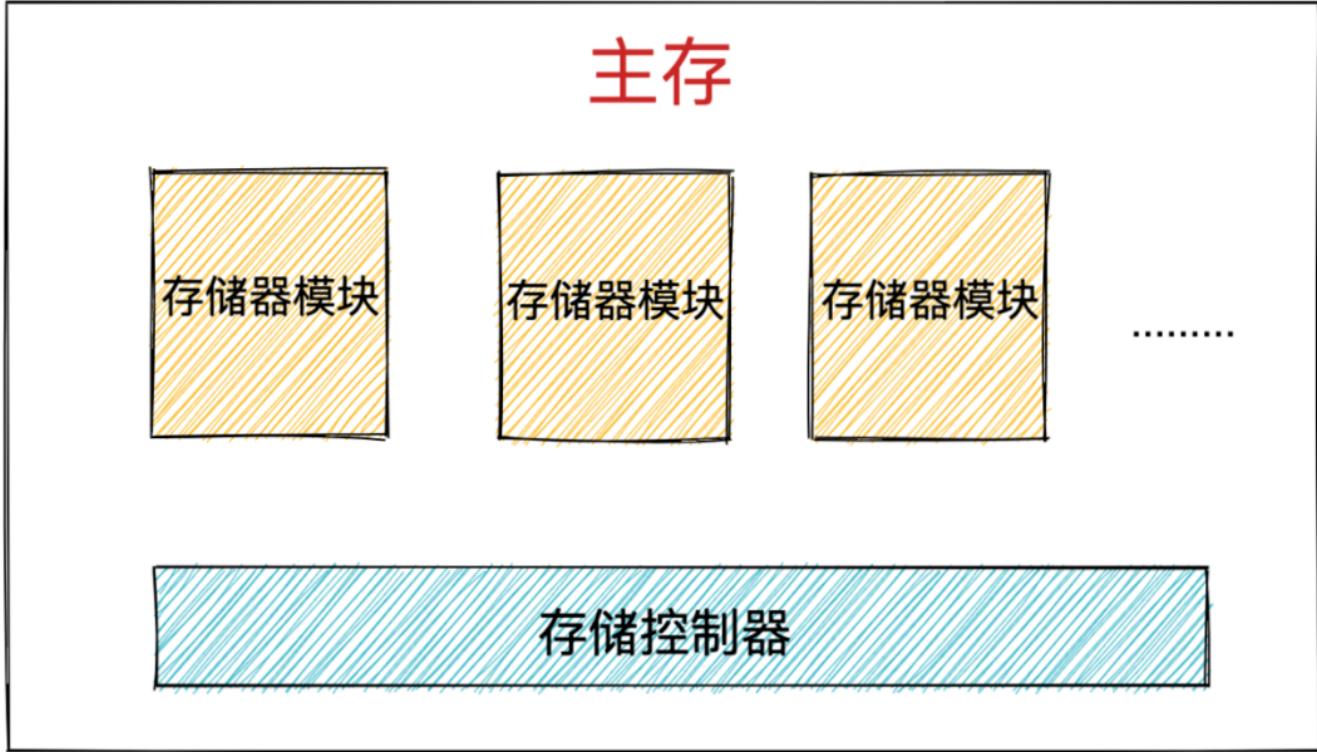


image.png

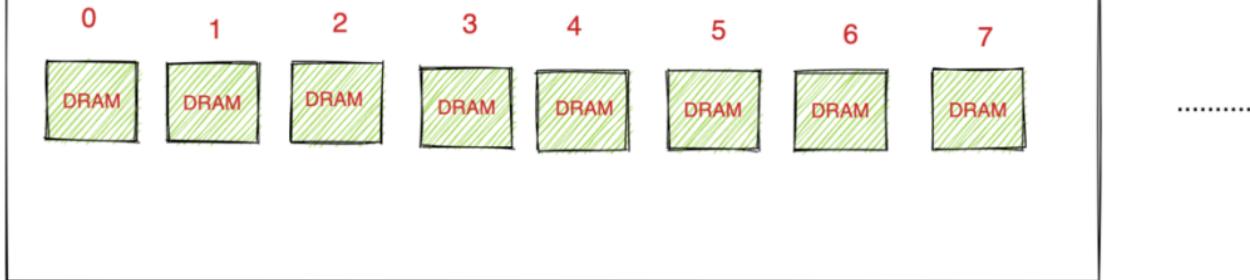
如图所示内存条上黑色的元器件就是存储器模块（memory module）。多个存储器模块连接到存储控制器上，就聚合成了主存。



内存结构.png

而前边介绍到的 DRAM芯片 就包装在存储器模块中，每个存储器模块中包含 8个DRAM芯片，依次编号为 0 - 7。

存储器模块



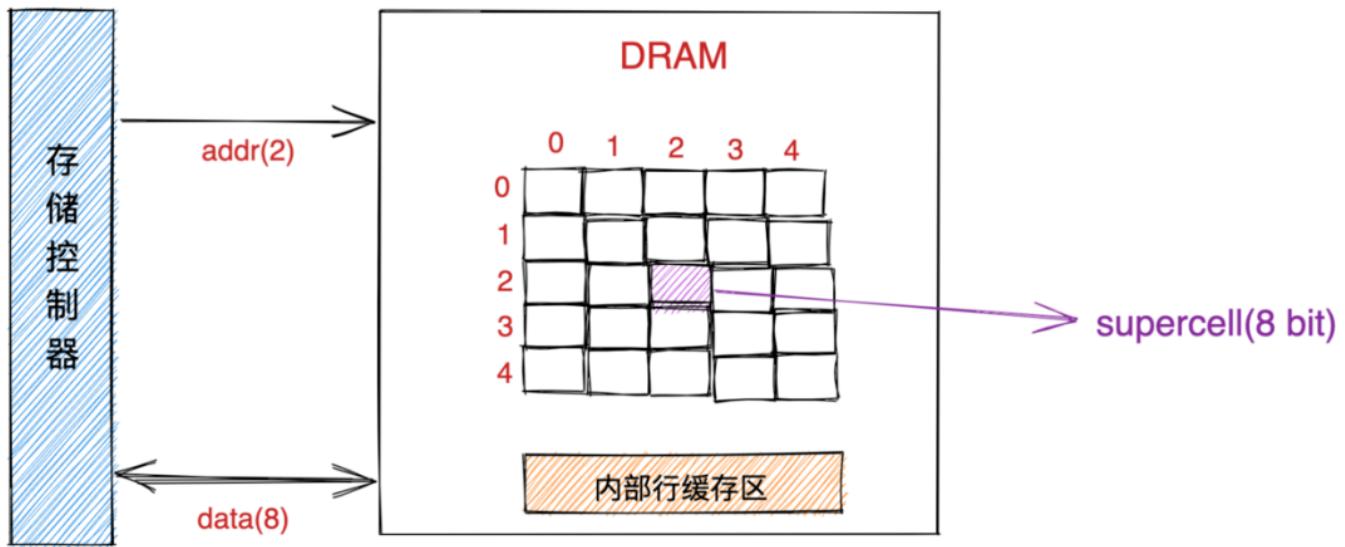
## 存储控制器

存储器模块.png

而每一个 DRAM芯片 的存储结构是一个二维矩阵，二维矩阵中存储的元素我们称为超单元（supercell），每个supercell大小为一个字节（8 bit）。每个supercell都由一个坐标地址（i, j）。

i表示二维矩阵中的行地址，在计算机中行地址称为RAS(row access strobe, 行访问选通脉冲)。j表示二维矩阵中的列地址，在计算机中列地址称为CAS(column access strobe, 列访问选通脉冲)。

下图中的supercell的RAS = 2, CAS = 2。



DRAM结构.png

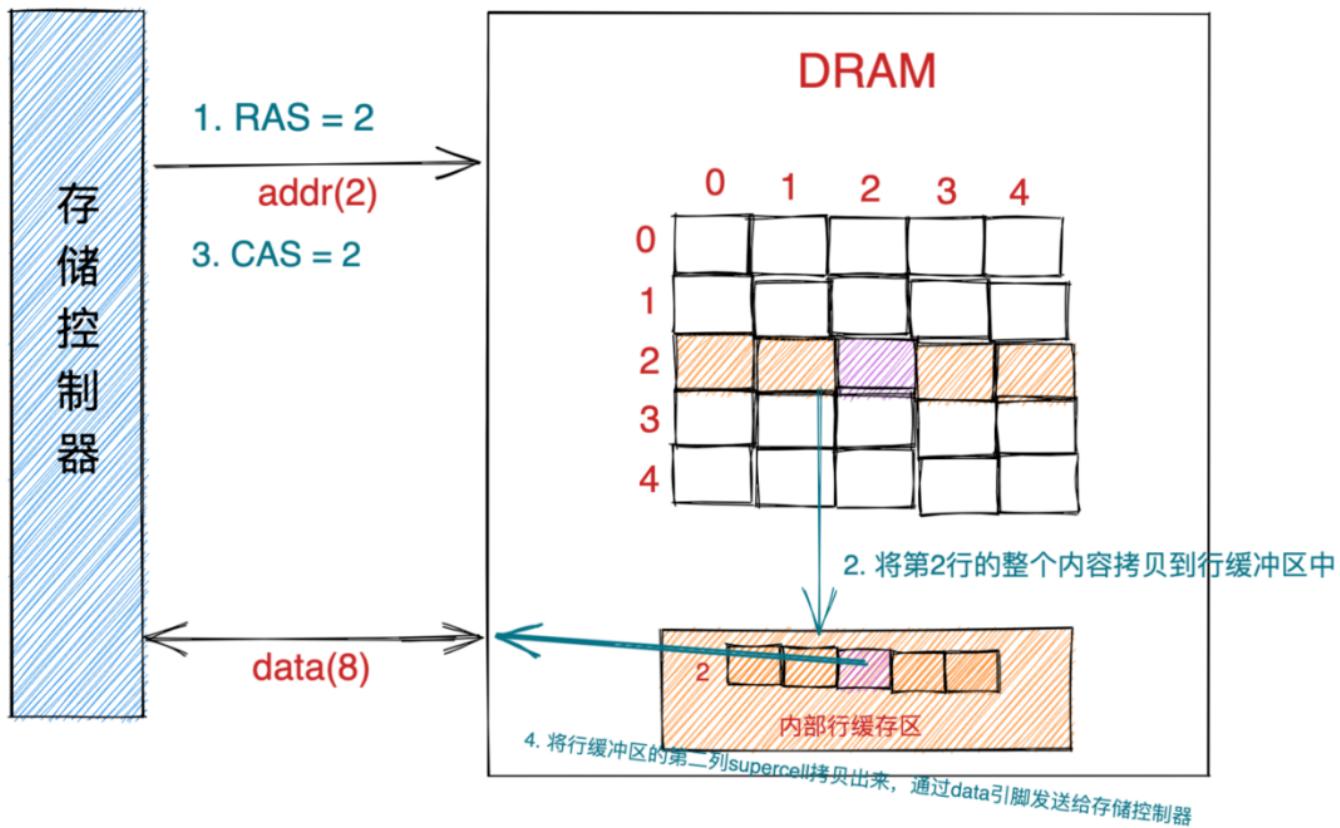
DRAM芯片 中的信息通过引脚流入流出DRAM芯片。每个引脚携带 1 bit 的信号。

图中DRAM芯片包含了两个地址引脚(addr)，因为我们要通过RAS, CAS来定位要获取的 supercell。还有8个数据引脚 (data) ,因为DRAM芯片的IO单位为一个字节 (8 bit) ,所以需要8个data引脚从DRAM芯片传入传出数据。

注意这里只是为了解释地址引脚和数据引脚的概念，实际硬件中的引脚数量是不一定的。

## 5.2 DRAM芯片的访问

我们现在就以读取上图中坐标地址为(2, 2)的supercell为例，来说明访问DRAM芯片的过程。



DRAM芯片访问.png

- 首先存储控制器将行地址 RAS = 2 通过地址引脚发送给 DRAM芯片。
- DRAM芯片根据 RAS = 2 将二维矩阵中的第二行的全部内容拷贝到 内部行缓冲区 中。
- 接下来存储控制器会通过地址引脚发送 CAS = 2 到DRAM芯片中。
- DRAM芯片从内部行缓冲区中根据 CAS = 2 拷贝出第二列的supercell并通过数据引脚发送给存储控制器。

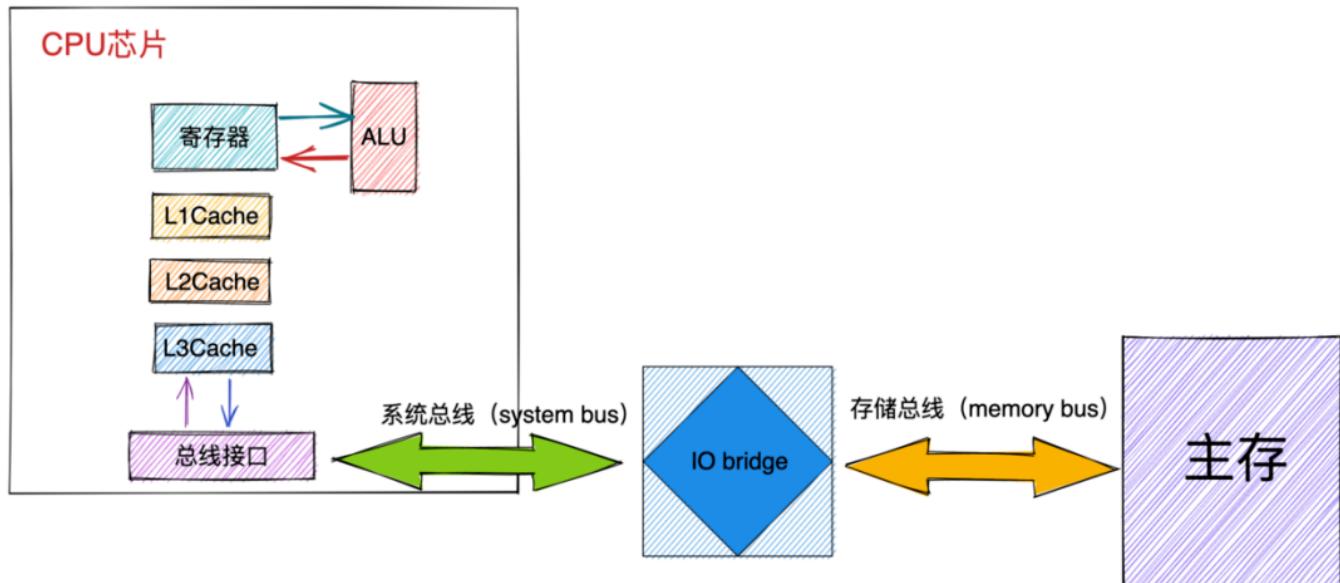
DRAM芯片的IO单位为一个supercell，也就是一个字节(8 bit)。

## 5.3 CPU如何读写主存

前边我们介绍了内存的物理结构，以及如何访问内存中的DRAM芯片获取supercell中存储的数据（一个

字节)。

本小节我们来介绍下CPU是如何访问内存的。



CPU与内存之间的总线结构.png

其中关于CPU芯片的内部结构我们在介绍false sharding的时候已经详细的介绍过了，这里我们主要聚焦在CPU与内存之间的总线架构上。

### 5.3.1 总线结构

CPU与内存之间的数据交互是通过总线（bus）完成的，而数据在总线上的传送是通过一系列的步骤完成的，这些步骤称为总线事务（bus transaction）。

其中数据从内存传送到CPU称之为读事务 (read transaction)，数据从CPU传送到内存称之为写事务 (write transaction)。

总线上传输的信号包括：地址信号，数据信号，控制信号。其中控制总线上传输的控制信号可以同步事务，并能够标识出当前正在被执行的事务信息：

- 当前这个事务是到内存的？还是到磁盘的？或者是到其他IO设备的？
- 这个事务是读还是写？
- 总线上传输的地址信号（内存地址），还是数据信号（数据）？。

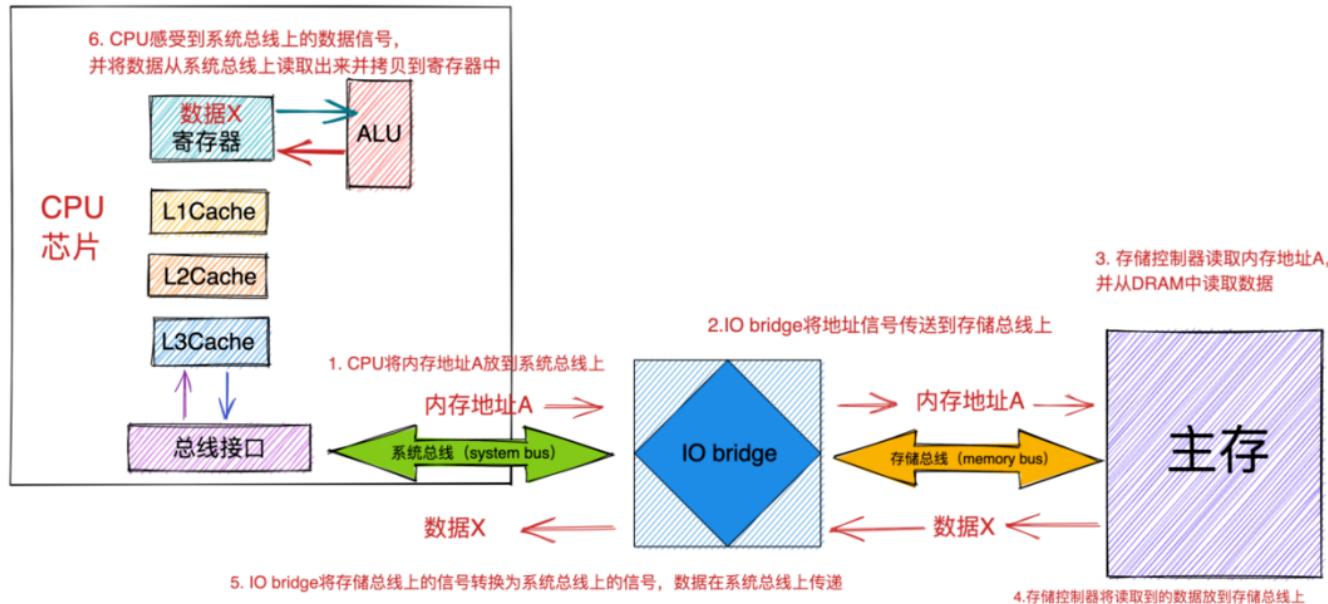
还记得我们前边讲到的MESI缓存一致性协议吗？当core0修改字段a的值时，其他CPU核心会在总线上嗅探字段a的内存地址，如果嗅探到总线上出现字段a的内存地址，说明有人在修改字段a，这样其他CPU核心就会失效自己缓存字段a所在的cache line。

如上图所示，其中系统总线是连接CPU与IO bridge的，存储总线是用来连接IO bridge和主存的。

IO bridge 负责将系统总线上的电子信号转换成存储总线上的电子信号。IO bridge也会将系统总线和存储总线连接到IO总线（磁盘等IO设备）上。这里我们看到IO bridge其实起的作用就是转换不同总线上的电子信号。

### 5.3.2 CPU从内存读取数据过程

假设CPU现在要将内存地址为A的内容加载到寄存器中进行运算。



CPU读取内存.png

首先CPU芯片中的总线接口会在总线上发起读事务（read transaction）。该读事务分为以下步骤进行：

1. CPU将内存地址A放到系统总线上。随后IO bridge将信号传递到存储总线上。
2. 主存感受到存储总线上的地址信号并通过存储控制器将存储总线上的内存地址A读取出来。
3. 存储控制器通过内存地址A定位到具体的存储器模块，从DRAM芯片中取出内存地址A对应的数据X。
4. 存储控制器将读取到的数据X放到存储总线上，随后IO bridge将存储总线上的数据信号转换为系统总线上的数据信号，然后继续沿着系统总线传递。
5. CPU芯片感受到系统总线上的数据信号，将数据从系统总线上读取出来并拷贝到寄存器中。

以上就是CPU读取内存数据到寄存器中的完整过程。

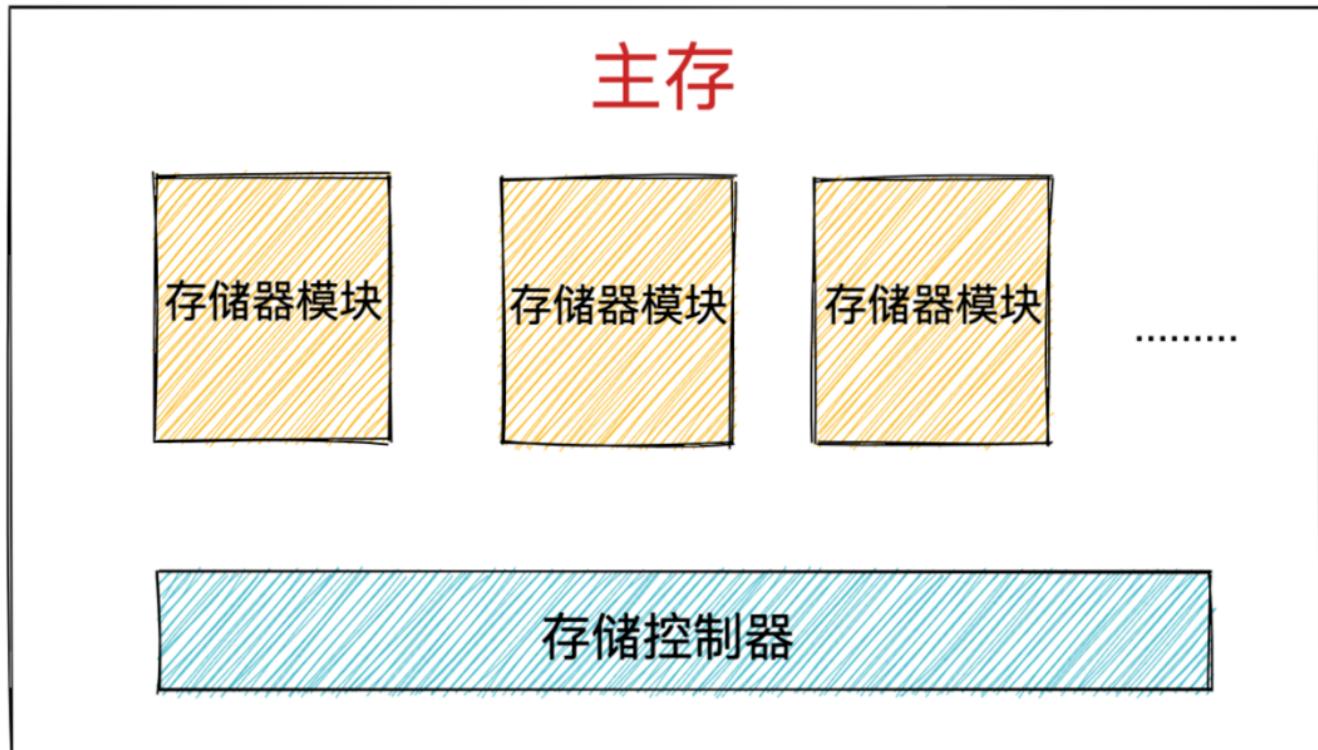
但是其中还涉及到一个重要的过程，这里我们还是需要摊开来介绍一下，那就是存储控制器如何通过内存地址A从主存中读取出对应的数据X的？

接下来我们结合前边介绍的内存结构以及从DRAM芯片读取数据的过程，来总体介绍下如何从主存中读取数据。

### 5.3.3 如何根据内存地址从主存中读取数据

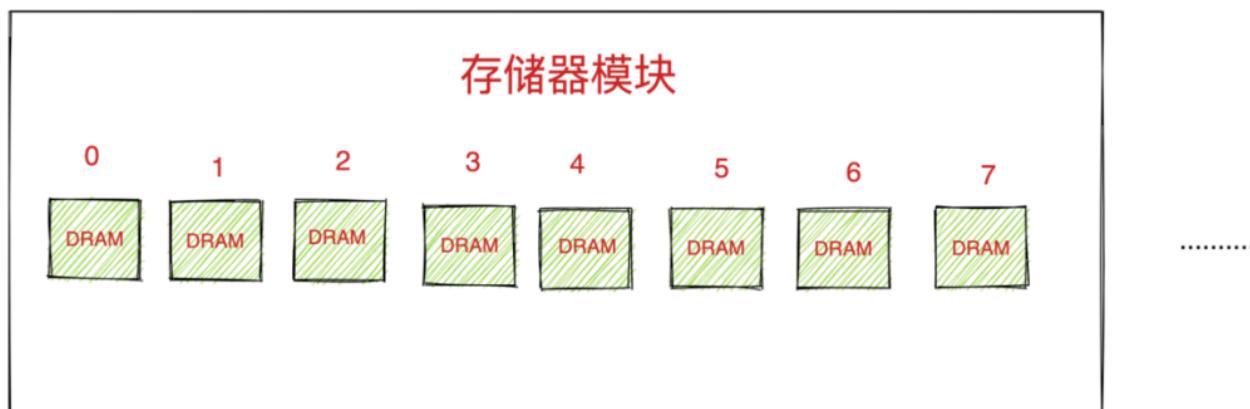
前边介绍到，当主存中的存储控制器感受到了存储总线上的地址信号时，会将内存地址从存储总线上读取出来。

随后会通过内存地址定位到具体的存储器模块。还记得内存结构中的存储器模块吗？？



内存结构.png

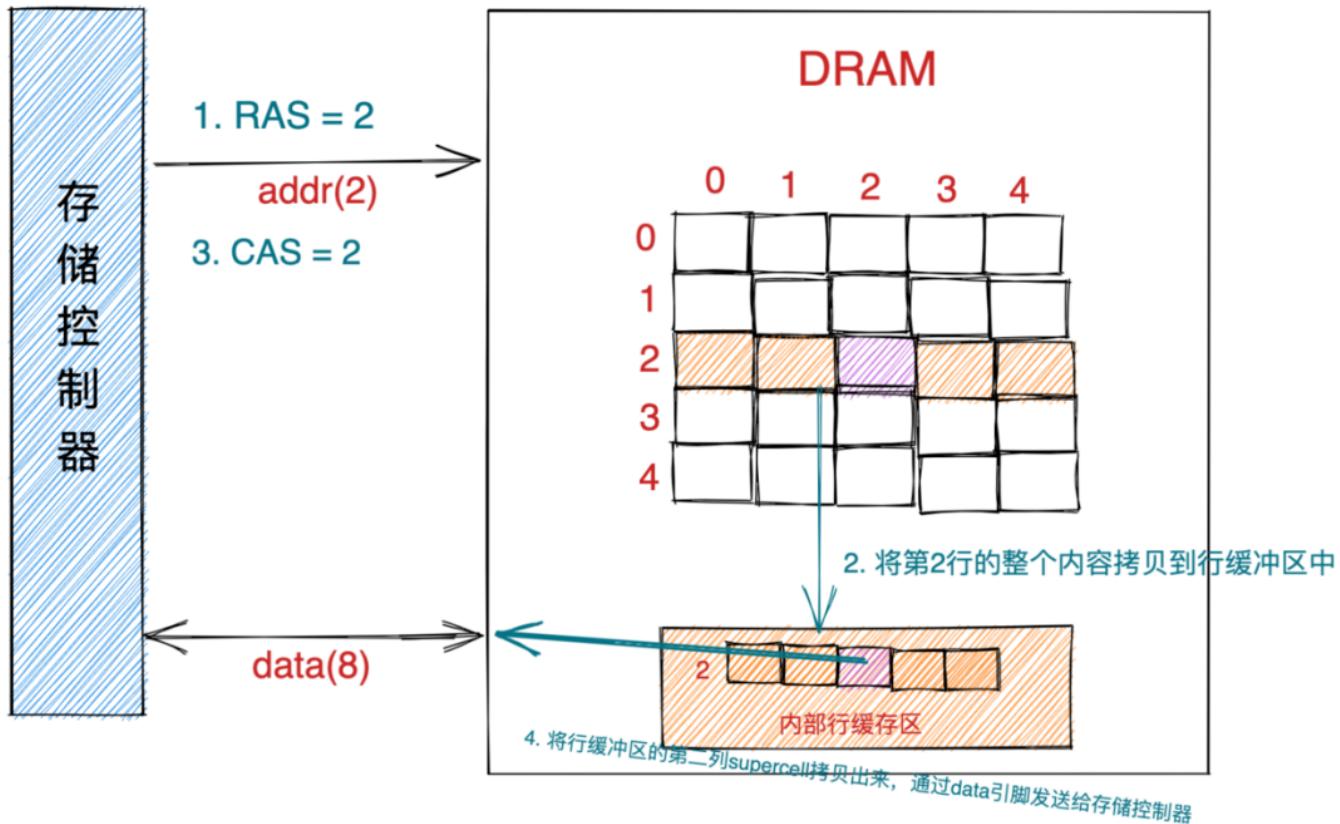
而每个存储器模块中包含了8个DRAM芯片，编号从0 - 7。



# 存储控制器

存储器模块.png

存储控制器会将内存地址转换为DRAM芯片中supercell在二维矩阵中的坐标地址(RAS, CAS)。并将这个坐标地址发送给对应的存储器模块。随后存储器模块会将RAS和CAS广播到存储器模块中的所有DRAM芯片。依次通过(RAS, CAS)从DRAM0到DRAM7读取到相应的supercell。



DRAM芯片访问.png

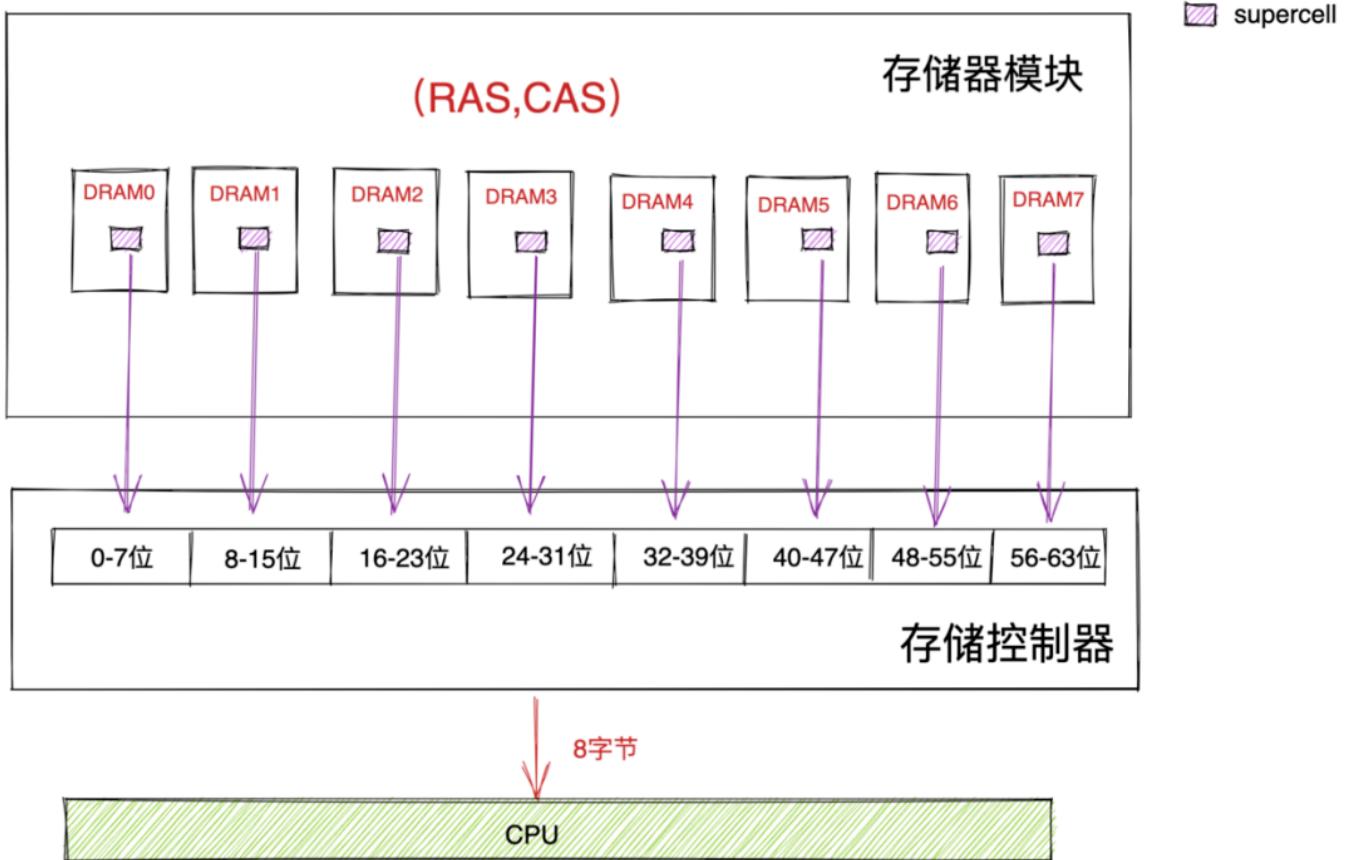
我们知道一个supercell存储了8 bit数据，这里我们从DRAM0到DRAM7 依次读取到了8个supercell也就是8个字节，然后将这8个字节返回给存储控制器，由存储控制器将数据放到存储总线上。

CPU总是以word size为单位从内存中读取数据，在64位处理器中的word size为8个字节。64位的内存也只能每次吞吐8个字节。

CPU每次会向内存读写一个cache line大小的数据（64个字节），但是内存一次只能吞吐8个字节。

所以在内存地址对应的存储器模块中，DRAM0芯片存储第一个低位字节（supercell），DRAM1芯片存储第二个字节，……依次类推DRAM7芯片存储最后一个高位字节。

内存一次读取和写入的单位是8个字节。而且在程序员眼里连续的内存地址实际上在物理上是不连续的。因为这连续的8个字节其实是存储于不同的DRAM芯片上的。每个DRAM芯片存储一个字节（supercell）。



读取存储器模块数据.png

### 5.3.4 CPU向内存写入数据过程

我们现在假设CPU要将寄存器中的数据X写到内存地址A中。同样的道理，CPU芯片中的总线接口会向总线发起写事务（write transaction）。写事务步骤如下：

1. CPU将要写入的内存地址A放入系统总线上。
2. 通过IO bridge的信号转换，将内存地址A传递到存储总线上。
3. 存储控制器感受到存储总线上的地址信号，将内存地址A从存储总线上读取出来，并等待数据的到达。
4. CPU将寄存器中的数据拷贝到系统总线上，通过IO bridge的信号转换，将数据传递到存储总线上。
5. 存储控制器感受到存储总线上的数据信号，将数据从存储总线上读取出来。

6. 存储控制器通过内存地址A定位到具体的存储器模块，最后将数据写入存储器模块中的8个DRAM芯片中。

## 6. 为什么要内存对齐

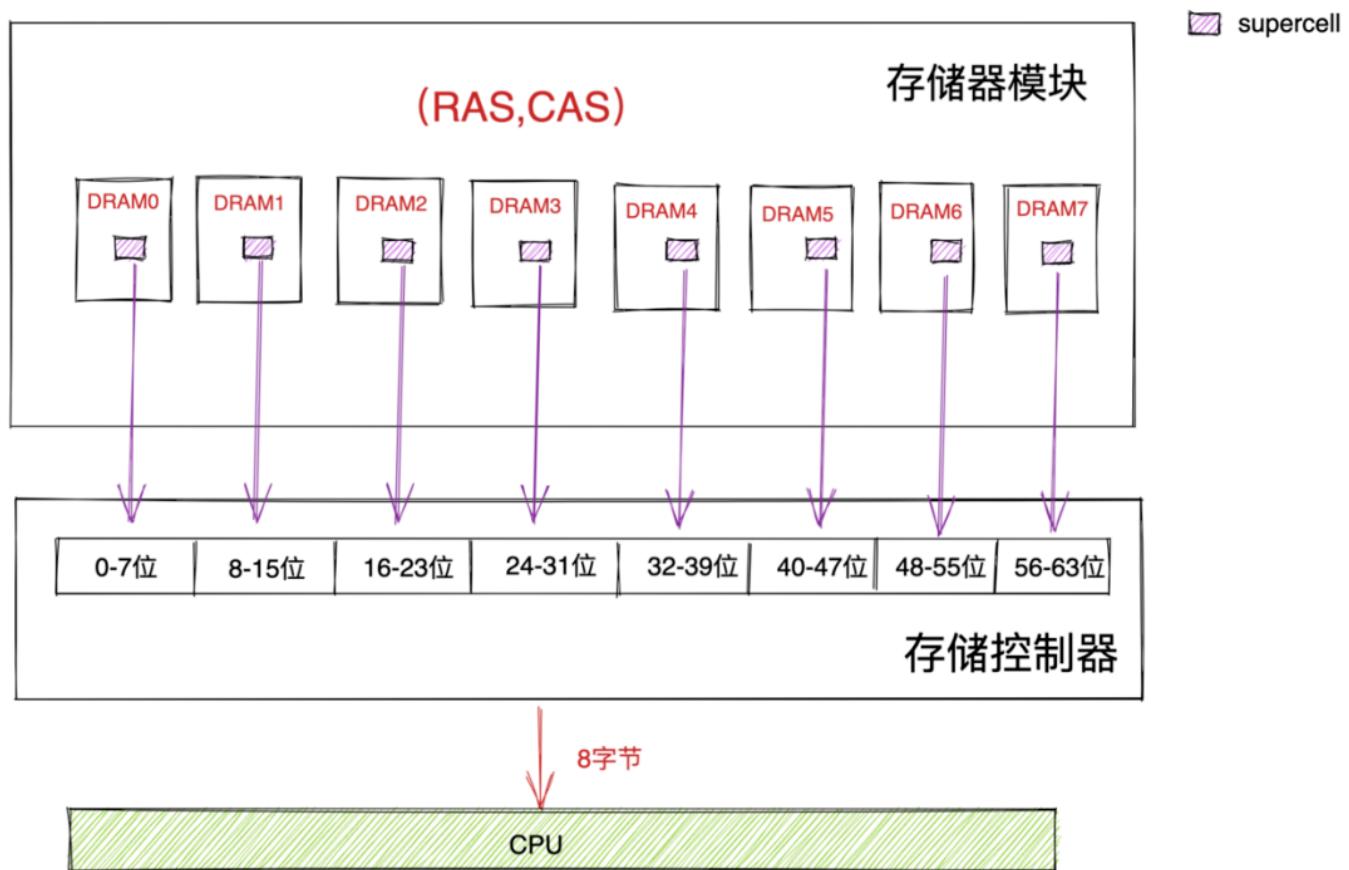
我们在了解了内存结构以及CPU读写内存的过程之后，现在我们回过头来讨论下本小节开头的问题：为什么要内存对齐？

下面笔者从三个方面来介绍下要进行内存对齐的原因：

### 速度

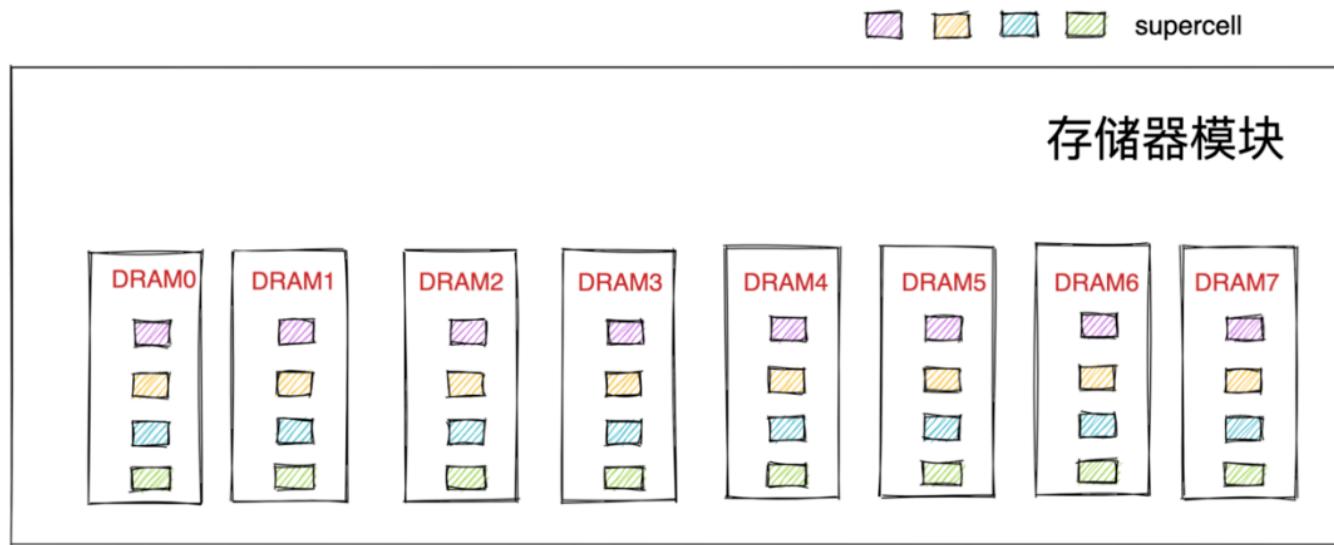
CPU读取数据的单位是根据word size来的，在64位处理器中word size = 8字节，所以CPU向内存读写数据的单位为8字节。

在64位内存中，内存IO单位为8个字节，我们前边也提到内存结构中的存储器模块通常以64位为单位（8个字节）传输数据到存储控制器上或者从存储控制器传出数据。因为每次内存IO读取数据都是从数据所在具体的存储器模块中包含的这8个DRAM芯片中以相同的(RAM, CAS)依次读取一个字节，然后在存储控制器中聚合成8个字节返回给CPU。



读取存储器模块数据.png

由于存储器模块中这种由8个DRAM芯片组成的物理存储结构的限制，内存读取数据只能是按照地址顺序8个字节的依次读取----8个字节8个字节地来读取数据。



依次按照坐标地址 (RAS,CAS) 读取 8个字节 (supercell)

内存IO单位.png

- 假设我们现在读取 0x0000 - 0x0007 这段连续内存地址上的8个字节。由于内存读取是按照 8个字节 为单位依次顺序读取的，而我们要读取的这段内存地址的起始地址是0（8的倍数），所以0x0000 - 0x0007 中每个地址的坐标都是相同的（RAS, CAS）。所以他可以在8个DRAM芯片中通过相同的（RAS, CAS）一次性读取出来。
- 如果我们现在读取 0x0008 - 0x0015 这段连续内存上的8个字节也是一样的，因为内存段起始地址为8（8的倍数），所以这段内存上的每个内存地址在DRAM芯片中的坐标地址（RAS, CAS）也是相同的，我们也可以一次性读取出来。

注意： 0x0000 - 0x0007 内存段中的坐标地址（RAS,CAS）与 0x0008 - 0x0015 内存段中的坐标地址（RAS,CAS）是不相同的。

- 但如果我们现在读取 0x0007 - 0x0014 这段连续内存上的8个字节情况就不一样了，由于起始地址 0x0007 在 DRAM芯片中的（RAS,CAS）与后边地址 0x0008 - 0x0014 的（RAS,CAS）不相同，所以CPU只能先从 0x0000 - 0x0007 读取8个字节出来先放入 结果寄存器 中并左移7个字节（目的是只获取 0x0007），然后CPU在从 0x0008 - 0x0015 读取8个字节出来放入临时寄存器中并右移1个字节（目的是获取 0x0008 - 0x0014）最后与结果寄存器或运算。最终得到 0x0007 - 0x0014 地址段上的8个字节。

从以上分析过程来看，当CPU访问内存对齐的地址时，比如 0x0000 和 0x0008 这两个起始地址都是对齐至 8的倍数。CPU可以通过一次read transaction读取出来。

但是当CPU访问内存没有对齐的地址时，比如 0x0007 这个起始地址就没有对齐至 8的倍数。CPU就需要两次read transaction才能将数据读取出来。

还记得笔者在小节开头提出的问题吗？"Java 虚拟机堆中对象的起始地址为什么需要对齐至 8的倍数？为什么不对齐至4的倍数或16的倍数或32的倍数呢？" 现在你能回答了吗？？？

## 原子性

CPU可以原子地操作一个对齐的word size memory。64位处理器中word size = 8字节。

## 尽量分配在一个缓存行中

前边在介绍false sharding的时候我们提到目前主流处理器中的cache line大小为64字节，堆中对象的起始地址通过内存对齐至8的倍数，可以让对象尽可能的分配到一个缓存行中。一个内存起始地址未对齐的对象可能会跨缓存行存储，这样会导致CPU的执行效率慢2倍。

其中对象中字段内存对齐的其中一个重要原因也是让字段只出现在同一 CPU 的缓存行中。如果字段不是对齐的，那么就有可能出现跨缓存行的字段。也就是说，该字段的读取可能需要替换两个缓存行，而该字段的存储也会同时污染两个缓存行。这两种情况对程序的执行效率而言都是不利的。

另外在《2. 字段重排列》这一小节介绍的三种字段对齐规则，是保证在字段内存对齐的基础上使得实例数据区占用内存尽可能的小。

## 7. 压缩指针

在介绍完关于内存对齐的相关内容之后，我们来介绍下前边经常提到的压缩指针。可以通过JVM参数X:+UseCompressedOops开启，当然默认是开启的。

在本小节内容开启之前，我们先来讨论一个问题，那就是为什么要使用压缩指针？？

假设我们现在正在准备将32位系统切换到64位系统，起初我们可能会期望系统性能会立马得到提升，但现实情况可能并不是这样的。

在JVM中导致性能下降的最主要原因是64位系统中的对象引用。在前边我们也提到过，64位系统中对象的引用以及类型指针占用64 bit也就是8个字节。

这就导致了在64位系统中的对象引用占用的内存空间是32位系统中的两倍大小，因此间接的导致了在64位系统中更多的内存消耗以及更频繁的GC发生，GC占用的CPU时间越多，那么我们的应用程序占用CPU的时间就越少。

另外一个就是对象的引用变大了，那么CPU可缓存的对象相对就少了，增加了对内存的访问。综合以上几点从而导致了系统性能的下降。

从另一方面来说，在64位系统中内存的寻址空间为 $2^{48} = 256T$ ，在现实情况中我们真的需要这么大

的寻址空间吗？？好像也没必要吧~~

于是我们就有了新的想法：那么我们是否应该切换回32位系统呢？

如果我们切换回32位系统，我们怎么解决在32位系统中拥有超过4G的内存寻址空间呢？因为现在4G的内存大小对于现在的应用来说明显是不够的。

我想以上的这些问题，也是当初JVM的开发者需要面对和解决的，当然他们也交出了非常完美的答卷，那就是使用压缩指针可以在64位系统中利用32位的对象引用获得超过4G的内存寻址空间。

## 7.1 压缩指针是如何做到的呢？

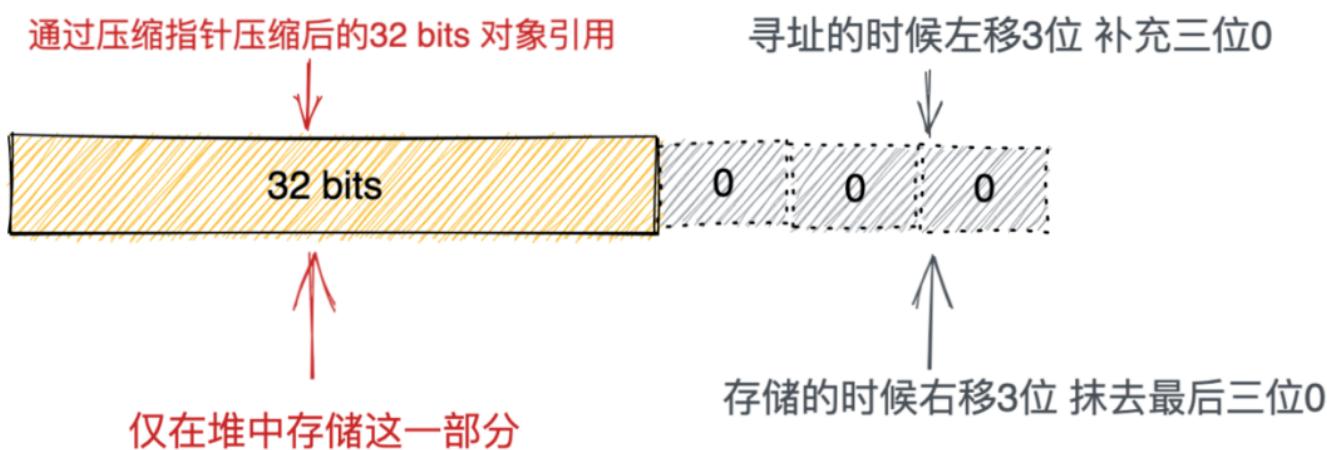
还记得之前我们在介绍对齐填充和内存对齐小节中提到的，在Java虚拟机堆中对象的起始地址必须对齐至8的倍数吗？

由于堆中对象的起始地址均是对齐至8的倍数，所以对象引用在开启压缩指针情况下的32位二进制的后三位始终是0（因为它们始终可以被8整除）。

既然JVM已经知道了这些对象的内存地址后三位始终是0，那么这些无意义的0就没必要在堆中继续存储。相反，我们可以利用存储0的这3位bit存储一些有意义的信息，这样我们就多出3位bit的寻址空间。

这样在存储的时候，JVM还是按照32位来存储，只不过后三位原本用来存储0的bit现在被我们用来存放有意义的地址空间信息。

当寻址的时候，JVM将这32位的对象引用左移3位（后三位补0）。这就导致了在开启压缩指针的情况下，我们原本32位的内存寻址空间一下变成了35位。可寻址的内存空间变为 $2^{32} * 2^3 = 32G$ 。



压缩指针.png

这样一来，JVM虽然额外的执行了一些位运算但是极大的提高了寻址空间，并且将对象引用占用内存大小降低了一半，节省了大量空间。况且这些位运算对于CPU来说是非常容易且轻量的操作

通过压缩指针的原理我挖掘到了内存对齐的另一个重要原因就是通过内存对齐至 8的倍数，我们可以在64位系统中使用压缩指针通过32位的对象引用将寻址空间提升至 32G。

从Java7开始，当maximum heap size小于32G的时候，压缩指针是默认开启的。但是当maximum heap size大于32G的时候，压缩指针就会关闭。

那么我们如何在压缩指针开启的情况下进一步扩大寻址空间呢？？？

## 7.2 如何进一步扩大寻址空间

前边提到我们在Java虚拟机堆中对象起始地址均需要对齐至 8的倍数，不过这个数值我们可以通过JVM参数 -XX:ObjectAlignmentInBytes 来改变（默认值为8）。当然这个数值的必须是2的次幂，数值范围需要在 8 - 256之间。

正是因为对象地址对齐至8的倍数，才会多出3位bit让我们存储额外的地址信息，进而将4G的寻址空间提升至32G。

同样的道理，如果我们将 ObjectAlignmentInBytes 的数值设置为16呢？

对象地址均对齐至16的倍数，那么就会多出4位bit让我们存储额外的地址信息。寻址空间变为  $2^{32} * 2^4 = 64G$ 。

通过以上规律，我们就能知道，在64位系统中开启压缩指针的情况，寻址范围的计算公式： $4G * ObjectAlignmentInBytes = 寻址范围$ 。

但是笔者并不建议大家贸然这样做，因为增大了 ObjectAlignmentInBytes 虽然能扩大寻址范围，但是这同时也可能增加了对象之间的字节填充，导致压缩指针没有达到原本节省空间的效果。

## 8. 数组对象的内存布局

前边大量的篇幅我们都是在讨论Java普通对象在内存中的布局情况，最后这一小节我们再来说下Java中的数组对象在内存中是如何布局的。

### 8.1 基本类型数组的内存布局





基本类型数组内存布局.png

上图表示的是基本类型数组在内存中的布局，基本类型数组在JVM中用 typeArrayOop 结构体表示，基本类型数组类型元信息用 TypeArrayKlass 结构体表示。

数组的内存布局大体上和普通对象的内存布局差不多，唯一不同的是在数组类型对象头中多出了 4个字节 用来表示数组长度的部分。

我们还是分别以开启指针压缩和关闭指针压缩两种情况，通过下面的例子来进行说明：



```
long[] longArrayLayout = new long[1];
```

### 开启指针压缩 -XX:+UseCompressedOops

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)	a9 01 00 f8 (10101001 00000001 00000000 11111000) (-134217303)
12	4		(object header)	01 00 00 00 (00000001 00000000 00000000 00000000) (1)
16	8	long	[J.<elements>]	N/A
Instance size: 24 bytes				

image.png

我们看到红框部分即为数组类型对象头中多出来一个 4字节 大小用来表示数组长度的部分。

因为我们示例中的long型数组只有一个元素，所以实例数据区的大小只有8字节。如果我们示例中的long型数组变为两个元素，那么实例数据区的大小就会变为16字节，以此类推.....。

### 关闭指针压缩 -XX:-UseCompressedOops

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)	48 1d 00 2a (01001000 00011101 00000000 00101010) (704650568)
12	4	(object header)	02 00 00 00 (00000010 00000000 00000000 00000000) (2)
16	4	(object header)	01 00 00 00 (00000001 00000000 00000000 00000000) (1)
20	4	(alignment/padding gap)	N/A
24	8	long [J.<elements>	
Instance size: 32 bytes			

image.png

当关闭了指针压缩时，对象头中的MarkWord还是占用8个字节，但是类型指针从4个字节变为了8个字节。数组长度属性还是不变保持4个字节。

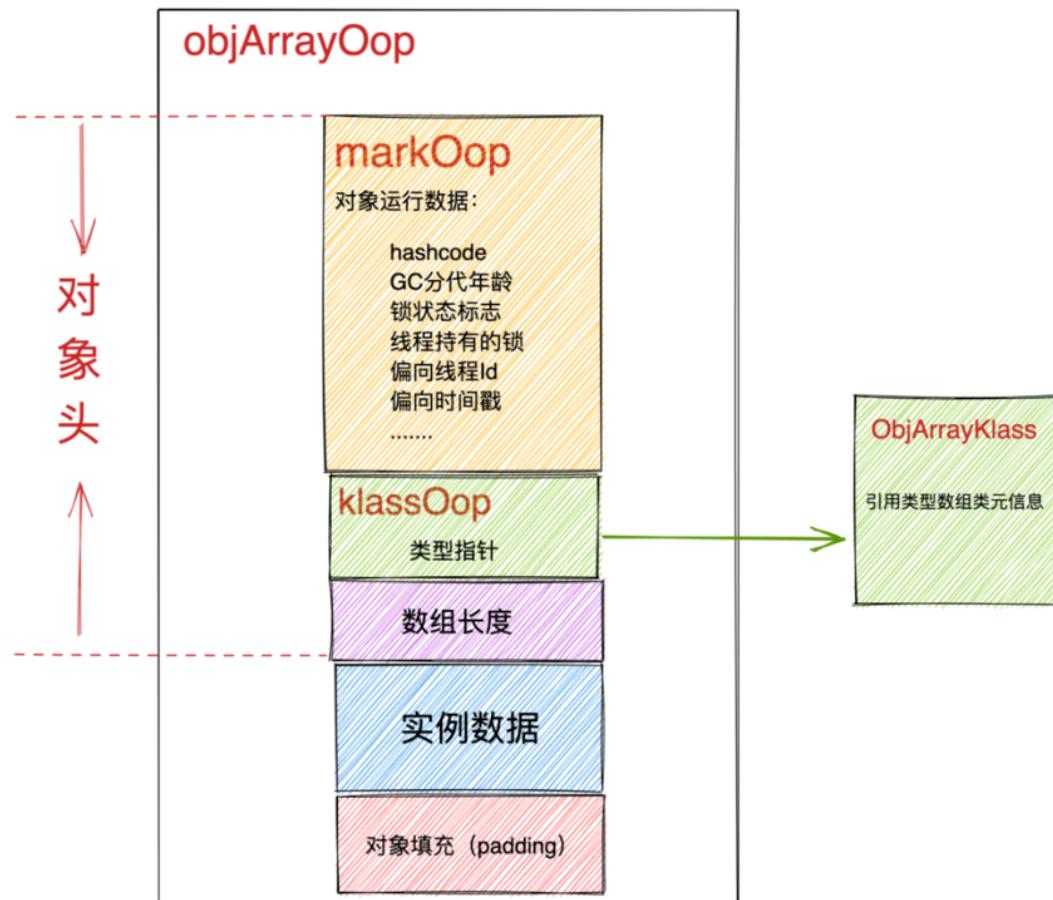
这里我们发现是实例数据区与对象头之间发生了对齐填充。大家还记得这是为什么吗？？

我们前边在字段重排列小节介绍了三种字段排列规则在这里继续适用：

- 规则1：如果一个字段占用 X 个字节，那么这个字段的偏移量OFFSET需要对齐至 NX。
- 规则2：在开启了压缩指针的64位JVM中，Java类中的第一个字段的OFFSET需要对齐至 4N，在关闭压缩指针的情况下类中第一个字段的OFFSET需要对齐至 8N。

这里基本数组类型的实例数据区中是long型，在关闭指针压缩的情况下，根据规则1和规则2需要对齐至8的倍数，所以要在其与对象头之间填充4个字节，达到内存对齐的目的，起始地址变为 24。

## 8.2 引用类型数组的内存布局



引用类型数组的内存布局.png

上图表示的是引用类型数组在内存中的布局，引用类型数组在JVM中用 objArrayOop 结构体表示，基本类型数组类型元信息用 ObjArrayKlass 结构体表示。

同样在引用类型数组的对象头中也会有一个 4字节 大小用来表示数组长度的部分。

我们还是分别以开启指针压缩和关闭指针压缩两种情况，通过下面的例子来进行说明：



```
public class ReferenceArrayList {
    char a;
    int b;
    short c;
}

ReferenceArrayList[] referenceArrayList = new ReferenceArrayList[1];
```

开启指针压缩 -XX:+UseCompressedOops

OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4		(object header)	01 00 00
4	4		(object header)	00 00 00
8	4		(object header)	43 c0 00
12	1		'\0'	00 00 00

12	4	(object header)	01 00 00
16	4	io.netty.bootstrap.ReferenceArrayLayout ReferenceArrayLayout;.<elements>	N/A
20	4	(loss due to the next object alignment)	
Instance size: 24 bytes			
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total			

image.png

引用数组类型内存布局与基础数组类型内存布局最大的不同在于它们的实例数据区。由于开启了压缩指针，所以对象引用占用内存大小为4个字节，而我们示例中引用数组只包含一个引用元素，所以这里实例数据区中只有4个字节。相同的道理，如果示例中的引用数组包含的元素变为两个引用元素，那么实例数据区就会变为8个字节，以此类推.....。

最后由于Java对象需要内存对齐至8的倍数，所以在该引用数组的实例数据区后填充了4个字节。

### 关闭指针压缩 -XX:-UseCompressedOops

OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4		(object header)	01 00 00 00 (00000001 000000)
4	4		(object header)	00 00 00 00 (00000000 000000)
8	4		(object header)	b0 76 95 2a (10110000 011101)
12	4		(object header)	02 00 00 00 (00000010 000000)
16	4		(object header)	01 00 00 00 (00000001 000000)
20	4		(alignment/padding gap)	
24	8	io.netty.bootstrap.ReferenceArrayLayout	ReferenceArrayLayout;.<elements>	N/A
Instance size: 32 bytes				

image.png

当关闭压缩指针时，对象引用占用内存大小变为了8个字节，所以引用数组类型的实例数据区占用了8个字节。

根据字段重排列规则2，在引用数组类型对象头与实例数据区中间需要填充4个字节以保证内存对齐的目的。

## 总结

本文笔者详细介绍了Java普通对象以及数组类型对象的内存布局，以及相关对象占用内存大小的计算方法。

以及在对象内存布局中的实例数据区字段重排列的三个重要规则。以及后边由字节的对齐填充引出来的false sharding问题，还有Java8为了解决false sharding而引入的@Contented注解的原理及使用方式。

为了讲清楚内存对齐的底层原理，笔者还花了大量的篇幅讲解了内存的物理结构以及CPU读写内存的完整过程。

最后又由内存对齐引出了压缩指针的工作原理。由此我们知道进行内存对齐的四个原因：

- CPU访问性能：当CPU访问内存对齐的地址时，可以通过一个read transaction读取一个字长（word

size) 大小的数据出来。否则就需要两个read transaction。

- 原子性：CPU可以原子地操作一个对齐的word size memory。
- 尽可能利用CPU缓存：内存对齐可以使对象或者字段尽可能的被分配到一个缓存行中，避免跨缓存行存储，导致CPU执行效率减半。
- 提升压缩指针的内存寻址空间：对象与对象之间的内存对齐，可以使我们在64位系统中利用32位对象引用将内存寻址空间提升至32G。既降低了对象引用的内存占用，又提升了内存寻址空间。

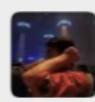
在本文中我们顺带还介绍了和内存布局相关的几个JVM参数：-XX:+UseCompressedOops, -XX+CompactFields, -XX:-RestrictContended, -XX:ContendedPaddingWidth, -XX:ObjectAlignmentInBytes。

最后感谢大家能看到这里，我们下篇文章再见~~~



java突击队  
技术经验分享

公众号



耳朵

不一定吧



苹果树下

这个群是我至今见过最卷的一个



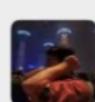
18

我还用过 awt类



苹果树下

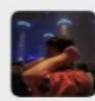
再晚，都有聊技术的



耳朵

这确实

苹果树下：这个群是我至今见过最卷的一个



耳朵

这群最活跃



18

是我见过码农群发言最多的。。



chill

确实



18

找对组织了

扫描下发二维码，备注：加群，可以拉你进群。



喜欢此内容的人还喜欢

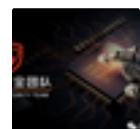
瞧瞧人家，那后端API接口写得，那叫一个优雅！

首席架构师专栏



【渗透技巧】APP、小程序、exe客户端抓包

闪焰安全服务团队



SDK 和 API 的区别是什么？

Go编程时光

