**第五十三课：SparkTungsten彻底解密之Spark性能调优第九季**

**本期内容：**

    1、到底什么是page

    2、page具体的两种方式

    3、page的使用源码

**一、Tungsten中到底什么是page？**

1、在Spark其实是不存在Page这个类的，那到底什么是Spark中的page呢？实质Page是一种数据结构（类似于Stack、list等），从OS的层面来讲，Page代表了一个内存块，在Page里面可以存放数据，当需要获得数据的时候可以定位到具体为哪个Page中的数据，找到该Page中根据特定的规则（例如说数据的offSet和length等）取出数据；而在OS中，Page代表了一种数据结构，并在内存中占据了一块内存区域，OS中有很多Page，在page中有数据，因此可以在Page中检索数据。

2、在阅读源码的时候细致研究MemoryBlock.Java-》*TaskMemoryBlock.java*

public class TaskMemoryManager {

private final Logger logger = LoggerFactory.getLogger(TaskMemoryManager.class);

/\*\* The number of bits used to address the page table. \*/

private static final int PAGE\_NUMBER\_BITS = 13;

/\*\* The number of bits used to encode offsets in data pages. \*/

@VisibleForTesting

static final int OFFSET\_BITS = 64 - PAGE\_NUMBER\_BITS; // 51

/\*\* The number of entries in the page table. \*/

private static final int PAGE\_TABLE\_SIZE = 1 << PAGE\_NUMBER\_BITS;

这里说明MemoryBLock是一个Page，MemoryBlock是一个PAGE\_NUMBER\_BITS数组，下面代码定义了一个page变量，page就是一个内存块；

//MemoryBlock为Page

final MemoryBlock page = memoryManager.tungstenMemoryAllocator().allocate(acquired);

    tungstenMemoryAllocator的两种分配内存的方式：ON\_HEAP和OFF\_HEAP；

/\*\*

\* Allocates memory for use by Unsafe/Tungsten code.

\*/

private[memory] final val tungstenMemoryAllocator: MemoryAllocator = {

tungstenMemoryMode match {

case MemoryMode.ON\_HEAP => MemoryAllocator.HEAP

case MemoryMode.OFF\_HEAP => MemoryAllocator.UNSAFE

}

}

}

   分配的数据中可能有历史残留数据，分配前需要先进行数据清理，在此确认了Page就是MemoryBLock后，如何确定Page的唯一性？

/\*\*

\* Allocates a contiguous block of memory. Note that the allocated memory is not guaranteed

\* to be zeroed out (call `zero()` on the result if this is necessary).

\* 在此确定了MemoryBlock就是Page，其中有pageid

\*/

MemoryBlock allocate(long size) throws OutOfMemoryError;

 要确认不同Page的唯一性，Page中有PageId，此时引入Page的第二个成员为PageNumber：

/\*\*

\* Optional page number; used when this MemoryBlock represents a page allocated by a

\* TaskMemoryManager. This field is public so that it can be modified by the TaskMemoryManager,

\* which lives in a different package.

\* 在此确定了MemoryBlock为page

\*/

public int pageNumber = -1;

//定义MemoryBlock方法，三个参数中obj参数注释可以为null，

public MemoryBlock(@Nullable Object obj, long offset, long length) {

super(obj, offset);

this.length = length;

}

page代表了内存区域以及存放的数据，存放的数据有两种类型：offSet和onSet，如果为Onheap有对象，offHeap没有对象，因为上面obj是offset的类型，所以没有对象，所以可以Nullable。使用MemoryBlock将onheap和offHeap进行封装，因为page相当于一个适配器，上面的Task使用的时候只关心使用的数据。

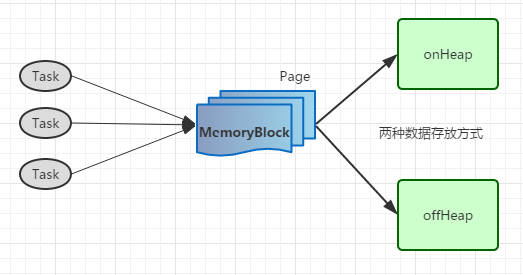


图 1 MemoryBlock的两种方式

 3、分配内存的两种方式ON\_HEAP和OFF\_HEAP如何寻址？

 onHeap怎么寻址？onHeap是先找到对象，再找到索引（找到地址，即找到长度？使用固定长度，page定位到数据，定位到偏移量offSet，需要长度length，找到位置后使用固定长度访问**貌似是4bit的长度**）；

*In off-heap mode, memory can be directly addressed with 64-bit longs.*

  I*n on-heap mode, memory is addressed by the combination of a base Object reference and a 64-bit offset within that object.*

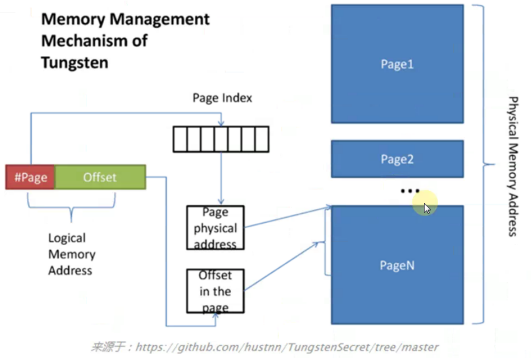
**二、如何使用Page呢？**

 1、在TaskMemoryManager中通过封装Page来定位数据，定位的时候如果是OnHeap的话，则先找到对象，然后在对象中通过OffSet来具体定位地址，而如果是OffHeap的话则直接定位；

/\*\* The number of bits used to address the page table. \*/

//13个bit的PageNumber就有8192个Page，即2的13次方

private static final int PAGE\_NUMBER\_BITS = 13;



 Page使用逻辑地址（Logical Address）做指针，记录Page中位置，逻辑地址是使用Long类型（64bit）的数字表示。前面13个位是Page的位，数据是一个个Record，因为Partition是一个个的Record，如果是OnHeap的方式，内存的分配是由HeapMemoryAllocator完成的。

上图中的#Page代表了13个bit的Page numble，即相当于有8192个Page，前边我们透彻的分析了一个Page就是一个MemoryBlock，所以就有8192个MemoryBlock，后面是offSet，数据进入的时候是一个编码的过程，读取时解码。

/\*\* The number of bits used to encode offsets in data pages. \*/

@VisibleForTesting

//51个位的OFFSET bit，

static final int OFFSET\_BITS = 64 - PAGE\_NUMBER\_BITS; // 51

2、一个关键的问题是如果

下面是ON\_HEAP的情况：

*/\*\**

*\* 变成Int后，以8个单位进行内存上的对齐（+7）  
 \* LONG\_ARRAY\_OFFSET：是一个地址  
 \* 根据该三个参数定位地址和具体的数据  
 \* 注：使用Array指的是，array中存放数据的地址，  
 \*/*

//为什么使用array，GC的时候要保持内存中的对象不变，因为GC的时候对象的地址可能发生变化（GC会整理内存），此时需要尽量保持对象的地址不变，就需要array存放地址

//onHeap中地址是放在array中的，因为有很多数据，所以需要使用array存放地址（指针，array中引用的名称不变，访问的时候通过引用的名称去访问），限制了数组的大小

long[] array = new long[(int) ((size + 7) / 8)]; //long类型的数组，范围大，/8变成Int，以8个单位进行内存上的对齐（+7），默认64bit

return new MemoryBlock(array, Platform.LONG\_ARRAY\_OFFSET, size); //传进Array，LONG\_ARRAY\_OFFSET是一个地址、size和offset完全能够定位数据

1）**ON\_HEAP**的方式：是jvm管理的，其中是对象和对象的成员，当分配一个page后，page中有很多的数据，所以有一些列的对象的地址，使用数组array保存对象的地址，因为GC的时候对象的地址可能会改变,On-heap的方式，内存的分配是有HeapMemoryAllocator完成的；

ON\_HEAP中的数据是对象和对象的成员，所以就会有一系列的地址和对象，保证对象和地址的引用，原因是在GC的时候对象的地址可能改变，所以使用数组访问对象和地址，不然对象的地址改变后，就找不到数据。

注：可以看出Spark还不能使用完全的堆外内存，因为Spark中有Java对象。

2）**OFF\_HEAP**的方式：是os的，offheap中存放的是二进制数据，源码如下：

/\*\*

\* A simple {@link MemoryAllocator} that uses {@code Unsafe} to allocate off-heap memory.

\*/

public class UnsafeMemoryAllocator implements MemoryAllocator {

@Override

public MemoryBlock allocate(long size) throws OutOfMemoryError {

long address = Platform.allocateMemory(size); //platform是OS的类

return new MemoryBlock(null, address, size); //没有对象的引用，此处的address直接决定了地址

}

下面是TaskMemoryManager注释：

/\* \* Instead, we use the following approach to encode record pointers in 64-bit longs: for off-heap

\* mode, just store the raw address, and for on-heap mode use the upper 13 bits of the address to

\* store a "page number" and the lower 51 bits to store an offset within this page. These page

\* numbers are used to index into a "page table" array inside of the MemoryManager in order to

\* retrieve the base object.

ON\_HEAP的方式最大可以根据long[ ]的限制使用35TB的内存(实际使用不到这么大的内存，因为OS和Shuffle的限制)，由此也说明Spark在为大规模，大机器的内存使用，数据计算处理打下坚实基础：

/\*\*

\*This allows us to address 8192 pages（13个bit的PageNumber就有8192个Page，即2的13次方）.

\*In on-heap mode, the maximum page size is limited by the

\* maximum size of a long[] array, allowing us to address 8192 \* 2^32 \* 8 bytes, which is

\* approximately 35 terabytes of memory.

\* on-heap最大大约使用35TB内存

\*/

 page搞了一套编码和解码的方式：

public long encodePageNumberAndOffset(MemoryBlock page, long offsetInPage) {

if (tungstenMemoryMode == MemoryMode.OFF\_HEAP) {

// In off-heap mode, an offset is an absolute address that may require a full 64 bits to

// encode. Due to our page size limitation, though, we can convert this into an offset that's

// relative to the page's base offset; this relative offset will fit in 51 bits.

offsetInPage -= page.getBaseOffset();

}

return encodePageNumberAndOffset(page.pageNumber, offsetInPage);

}

//page 搞了一套编码和解码方式

@VisibleForTesting

public static long encodePageNumberAndOffset(int pageNumber, long offsetInPage) {

assert (pageNumber != -1) : "encodePageNumberAndOffset called with invalid page";

return (((long) pageNumber) << OFFSET\_BITS) | (offsetInPage & MASK\_LONG\_LOWER\_51\_BITS);

}

**三、课外扩展**

**（注：**下面部分摘自：文／祝威廉（简书作者）原文链接：<http://www.jianshu.com/p/34729f9f833c>）

**内存管理器(MemoryManager)**

实际的内存管理放在了两个层次：

**org**.apache.spark.unsafe.memory.ExecutorMemoryManager**org**.apache.spark.unsafe.memory.TaskMemoryManager

  TaskMemeoryManager是针对每个Task而言的，但是这些Task都是运行在一个JVM实例上，对应的是Executor,所以内存应该由ExecutorMemoryManager统一进行管理。但是每个task需要交互，所以就让TaskMemeoryManager来进行这种交互。这是他们的分工，设计的很漂亮。

## **1、ExecutorMemoryManager**

我们先分析下ExecutorMemoryManager，该类根据你的配置，决定是使用什么样的MemoryAllocator，默认是in-heap。你当然也可以设置啦，通过：

spark.**unsafe**.offHeap=**true**

来进行开启off-heap 模式。

另外，如果发现你是在使用in-heap模式，则ExecutorMemoryManage 会维护一个MemoryBlock的池子，对象池，大家应该很熟悉了。那为啥只有in-heap模式有池子呢？那是因为in-heap 需要申请long[] 数组，维护一个池子，就不用到heap里反复去做申请动作了。

该类有两个核心方法：

MemoryBlock allocate(**long** size)

void free(**MemoryBlock** memory)

看名字就知道含义了：申请内存和释放内存。内存的单元是MemoryBlock,逻辑上是Page的概念。

## **2、TaskMemeoryManager**

为了统一对in-heap,off-heap进行建模，避免上层应用要自己区分自己处理的是in-heap还是off-heap内存块,所以这个时候就提出了一个Page的概念，并且使用逻辑地址来做指针，通过这个逻辑地址可以定位到特定一条记录在MemoryBlock的位置。

那么逻辑地址怎么表示呢？答案是用一个Long类型(64-bit)来表示。任何一条记录的位置都可以用一个Long来记录。

我们先来分析复杂的，in-heap模式：

[13-bit page num][51-bit offset]

这样就能可以表示8192个page。一个Page对应一个MemoryBlock。然后54-bit 可以表示Pb级别的，也就是说这个MemoryBlock可以是超级大的。

不过如果你还记得前文提到的in-heap模式里使用了一个long[]数组作为数据存储的，那么long的长度最大被限制为 Int的最大值，2^32 \* 8，也就是32GB。然后所有的Page加起来，大约35个TB。足够大了 其实。

当然这里是这里的限制，在上层里，比如shuffle，可能又会有其他的限制，导致能表示的内存会更小些。这个后续的文章我会进一步阐述。

申请一个Page的流程为：

1. 申请到空闲的Page number号
2. 进行实际的内存分配，得到一个MemoryBlock
3. 将Page number 赋给MemoryBlock

另外这个类也提供了一个不使用Page管理的方法申请内存，然后通过 allocatedNonPageMemory 对象进行追踪。

得到MemoryBlock，就代表我们真的拿到了内存，现在我们还要做一件事情，就是把一个记录用一个long类型表示出来,TaskMemoryManager 提供了encodePageNumberAndOffset(MemoryBlock page, long offsetInPage) 方法进行编码，编码的方式就是其那面提到的：

[13-bit page num][54-bit offset]

内部具体的就是一些位操作了。对应的还有各种decode方法。

你会好奇，只有offset,怎么知道一条记录的长度的？这个长度应该也要存储，才能还原回一条信息吧？

目前基本的做法是从offset开始，前四个字节来表示这条记录的长度，然后后面放具体的字节数组。为了解释这个问题，我从UnsafeShuffleExternalSorter类里扣了一段代码出来：

获得这条记录的逻辑地址，也就是一个64-bit的编码

final **long** recordAddress = taskMemoryManager.encodePageNumberAndOffset(dataPage, dataPagePosition);//dataPageBaseObject 其实就是数组对象的地址，然后以他为基准， 在dataPagePosition 处写入一个int类型数据，这个就是内容的长度。实际的内容就会放到这个位置之后

Platform.putInt(dataPageBaseObject, dataPagePosition, lengthInBytes);//最后把数据要拷贝的实际的内存中，就需要多要4个字节了。所以这里要加回来

dataPagePosition += 4;

Platform.copyMemory( recordBaseObject, recordBaseOffset, dataPageBaseObject, dataPagePosition, lengthInBytes);

上面分析的都是in-heap。那off-heap呢？

整个流程也是一致的。区别在于 off-heap拿到的是绝对地址，不是某个页的偏移量，为了统一处理，在进行编码的时候，我们要通过下面的公式重新算off-heap 在page中的相对位置：

offsetInPage -= page.getBaseOffset();

这里，page.getBaseOffset()是page对应的内存块的起始位置，也就是MemoryBlock的offset变量。如果你还记得上面off-heap申请MemoryBlock的方式，这个就是一开始拿到的偏移量。

这样就拿到相对于MemoryBlock的相对地址了，处理起来就可以和in-heap一致了。

解析的时候，就是反过来就行了，重新得到实际的绝对地址，然后类似in-heap,往前四个字节写长度，后面写实际的内容。