Flink 原理与实现：内存管理

摘要： 如今，大数据领域的开源框架（Hadoop，Spark，Storm）都使用的 JVM，当然也包括 Flink。基于 JVM 的数据分析引擎都需要面对将大量数据存到内存中，这就不得不面对 JVM 存在的几个问题： 1. Java 对象存储密度低。一个只包含 boolean 属性的对象占用了16个字节内存：对象头占了8个，boolean 属性占了1个，对齐填充占了7个。而实际上只需要一个bit（1

如今，大数据领域的开源框架（Hadoop，Spark，Storm）都使用的 JVM，当然也包括 Flink。基于 JVM 的数据分析引擎都需要面对将大量数据存到内存中，这就不得不面对 JVM 存在的几个问题：

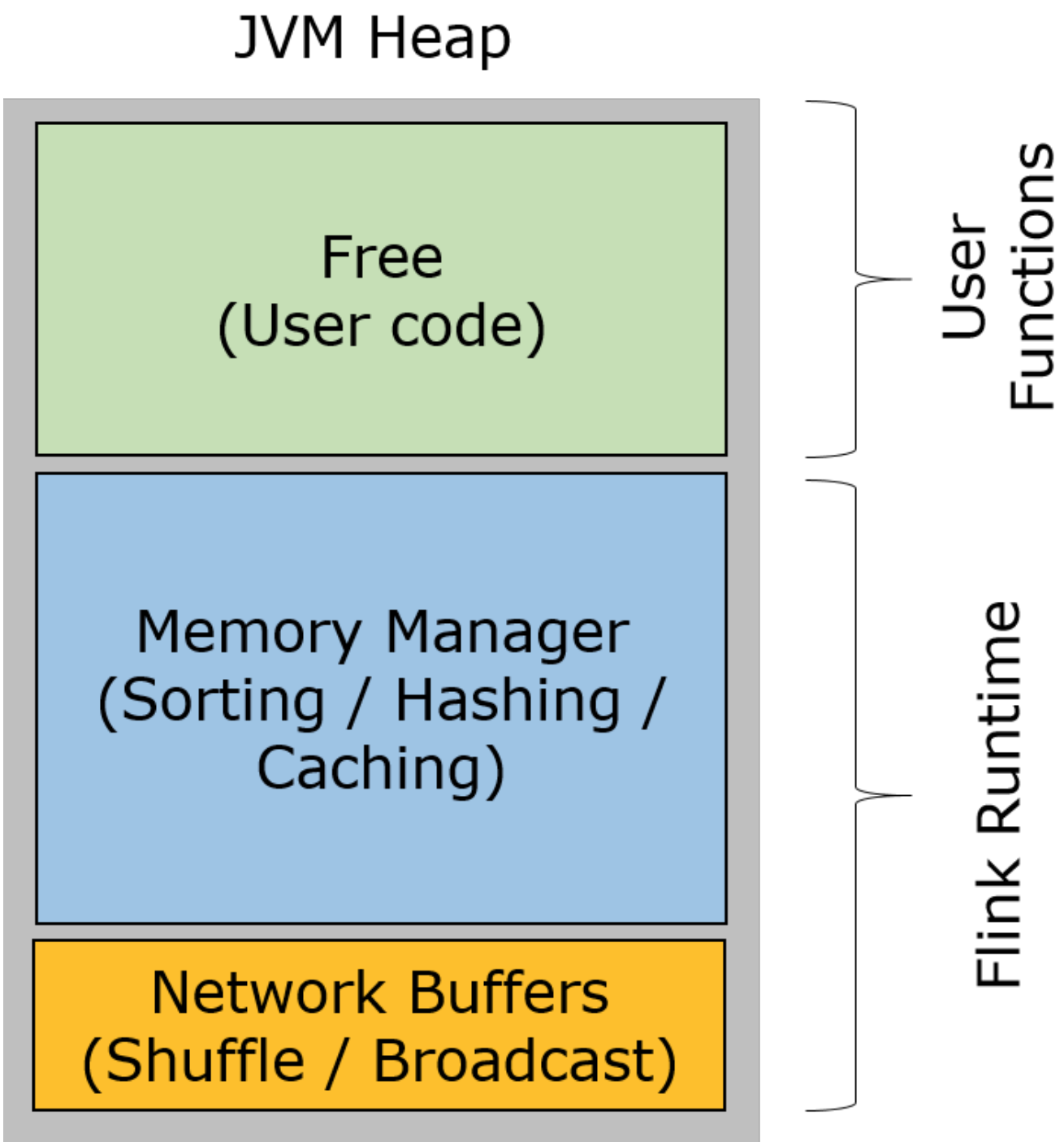
1. Java 对象存储密度低。一个只包含 boolean 属性的对象占用了16个字节内存：对象头占了8个，boolean 属性占了1个，对齐填充占了7个。而实际上只需要一个bit（1/8字节）就够了。
2. Full GC 会极大地影响性能，尤其是为了处理更大数据而开了很大内存空间的JVM来说，GC 会达到秒级甚至分钟级。
3. OOM 问题影响稳定性。OutOfMemoryError是分布式计算框架经常会遇到的问题，当JVM中所有对象大小超过分配给JVM的内存大小时，就会发生OutOfMemoryError错误，导致JVM崩溃，分布式框架的健壮性和性能都会受到影响。

所以目前，越来越多的大数据项目开始自己管理JVM内存了，像 Spark、Flink、HBase，为的就是获得像 C 一样的性能以及避免 OOM 的发生。本文将会讨论 Flink 是如何解决上面的问题的，主要内容包括内存管理、定制的序列化工具、缓存友好的数据结构和算法、堆外内存、JIT编译优化等。

积极的内存管理

Flink 并不是将大量对象存在堆上，而是将对象都序列化到一个预分配的内存块上，这个内存块叫做 `MemorySegment`，它代表了一段固定长度的内存（默认大小为 32KB），也是 Flink 中最小的内存分配单元，并且提供了非常高效的读写方法。你可以把 MemorySegment 想象成是为 Flink 定制的 `java.nio.ByteBuffer`。它的底层可以是一个普通的 Java 字节数组（`byte[]`），也可以是一个申请在堆外的 `ByteBuffer`。每条记录都会以序列化的形式存储在一个或多个`MemorySegment`中。

Flink 中的 Worker 名叫 TaskManager，是用来运行用户代码的 JVM 进程。TaskManager 的堆内存主要被分成了三个部分：



- **Network Buffers:** 一定数量的32KB大小的 buffer，主要用于数据的网络传输。在 TaskManager 启动的时候就会分配。默认数量是 2048 个，可以通过 `taskmanager.network.numberOfBuffers` 来配置。（阅读这篇文章了解更多Network Buffer的管理）
- **Memory Manager Pool:** 这是一个由 `MemoryManager` 管理的，由众多`MemorySegment`组成的超大集合。Flink 中的算法（如 sort/shuffle/join）会向这个内存池申请 MemorySegment，将序列化后的数据存于其中，使用完后释放回内存池。默认情况下，池子占了堆内存的 70% 的大小。

- **Remaining (Free) Heap:** 这部分的内存是留给用户代码以及 TaskManager 的数据结构使用的。因为这些数据结构一般都很小，所以基本上这些内存都是给用户代码使用的。从GC的角度来看，可以把这里看成的新生代，也就是说这里主要都是由用户代码生成的短期对象。

注意：Memory Manager Pool 主要在Batch模式下使用。在Steaming模式下，该池子不会预分配内存，也不会向该池子请求内存块。也就是说该部分的内存都是可以给用户代码使用的。不过社区是打算在 Streaming 模式下也能将该池子利用起来。

Flink 采用类似 DBMS 的 sort 和 join 算法，直接操作二进制数据，从而使序列化/反序列化带来的开销达到最小。所以 Flink 的内部实现更像 C/C++ 而非 Java。如果需要处理的数据超出了内存限制，则会将部分数据存储到硬盘上。如果要操作多块MemorySegment就像操作一块大的连续内存一样，Flink会使用逻辑视图（`AbstractPagedInputView`）来方便操作。下图描述了 Flink 如何存储序列化后的数据到内存块中，以及在需要的时候如何将数据存储到磁盘上。

从上面我们能够得出 Flink 积极的内存管理以及直接操作二进制数据有以下几点好处：

1. **减少GC压力。**显而易见，因为所有常驻型数据都以二进制的形式存在 Flink 的`MemoryManager`中，这些`MemorySegment`一直呆在老年代而不会被GC回收。其他的数据对象基本上是由用户代码生成的短生命周期对象，这部分对象可以被 Minor GC 快速回收。只要用户不去创建大量类似缓存的常驻型对象，那么老年代的大小是不会变的，Major GC也就永远不会发生。从而有效地降低了垃圾回收的压力。另外，这里的内存块还可以是堆外内存，这可以使得 JVM 内存更小，从而加速垃圾回收。
2. **避免了OOM。**所有的运行时数据结构和算法只能通过内存池申请内存，保证了其使用的内存大小是固定的，不会因为运行时数据结构和算法而发生OOM。在内存吃紧的情况下，算法（sort/join等）会高效地将一大批内存块写到磁盘，之后再读回来。因此，`OutOfMemoryErrors`可以有效地被避免。
3. **节省内存空间。**Java 对象在存储上有很多额外的消耗（如上一节所谈）。如果只存储实际数据的二进制内容，就可以避免这部分消耗。
4. **高效的二进制操作 & 缓存友好的计算。**二进制数据以定义好的格式存储，可以高效地比较与操作。另外，该二进制形式可以把相关的值，以及hash值，键值和指针等相邻地放进内存中。这使得数据结构可以对高速缓存更友好，可以从 L1/L2/L3 缓存获得性能的提升（下文会详细解释）。

为 Flink 量身定制的序列化框架

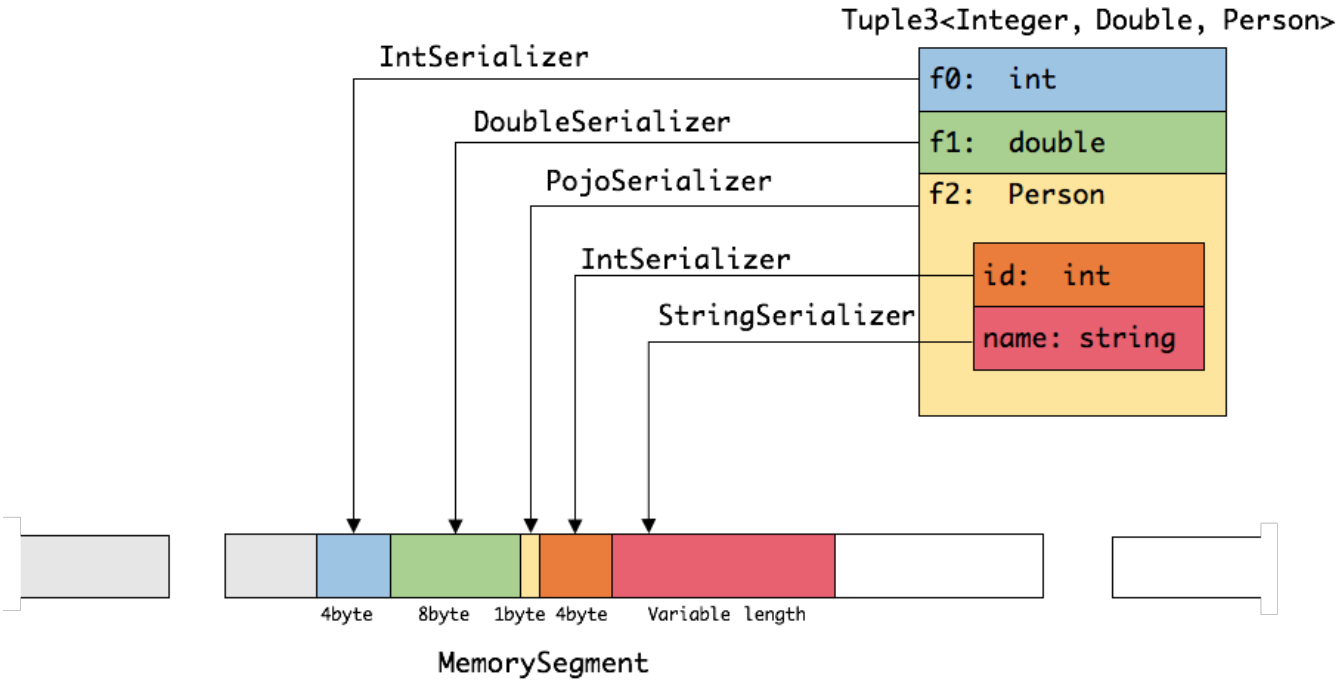
目前 Java 生态圈提供了众多的序列化框架：Java serialization, Kryo, Apache Avro 等等。但是 Flink 实现了自己的序列化框架。因为在 Flink 中处理的数据流通常是同一类型，由于数据集对象的类型固定，对于数据集可以只保存一份对象Schema信息，节省大量的存储空间。同时，对于固定大小的类型，也可通过固定的偏移位置存取。当我们需要访问某个对象成员变量的时候，通过定制的序列化工具，并不需要反序列化整个Java对象，而是可以直接通过偏移量，只是反序列化特

定的对象成员变量。如果对象的成员变量较多时，能够大大减少Java对象的创建开销，以及内存数据的拷贝大小。

Flink支持任意的Java或是Scala类型。Flink 在数据类型上有很大的进步，不需要实现一个特定的接口（像Hadoop中的`org.apache.hadoop.io.Writable`），Flink 能够自动识别数据类型。Flink 通过 Java Reflection 框架分析基于 Java 的 Flink 程序 UDF (User Define Function)的返回类型的类型信息，通过 Scala Compiler 分析基于 Scala 的 Flink 程序 UDF 的返回类型的类型信息。类型信息由 `TypeInformation` 类表示，TypeInformation 支持以下几种类型：

- `BasicTypeInfo`: 任意Java 基本类型（装箱的）或 String 类型。
- `BasicArrayTypeInfo`: 任意Java基本类型数组（装箱的）或 String 数组。
- `WritableTypeInfo`: 任意 Hadoop Writable 接口的实现类。
- `TupleTypeInfo`: 任意的 Flink Tuple 类型(支持Tuple1 to Tuple25)。Flink tuples 是固定长度固定类型的Java Tuple实现。
- `CaseClassTypeInfo`: 任意的 Scala CaseClass(包括 Scala tuples)。
- `PojoTypeInfo`: 任意的 POJO (Java or Scala)，例如，Java对象的所有成员变量，要么是 public 修饰符定义，要么有 getter/setter 方法。
- `GenericTypeInfo`: 任意无法匹配之前几种类型的类。

前六种数据类型基本上可以满足绝大部分的Flink程序，针对前六种类型数据集，Flink皆可以自动生成对应的TypeSerializer，能非常高效地对数据集进行序列化和反序列化。对于最后一种数据类型，Flink会使用Kryo进行序列化和反序列化。每个TypeInformation中，都包含了serializer，类型会自动通过serializer进行序列化，然后用Java Unsafe接口写入MemorySegments。对于可以用作key的数据类型，Flink还同时自动生成TypeComparator，用来辅助直接对序列化后的二进制数据进行compare、hash等操作。对于 Tuple、CaseClass、POJO 等组合类型，其TypeSerializer和TypeComparator也是组合的，序列化和比较时会委托给对应的serializers和comparators。如下图展示 一个内嵌型的Tuple3 对象的序列化过程。



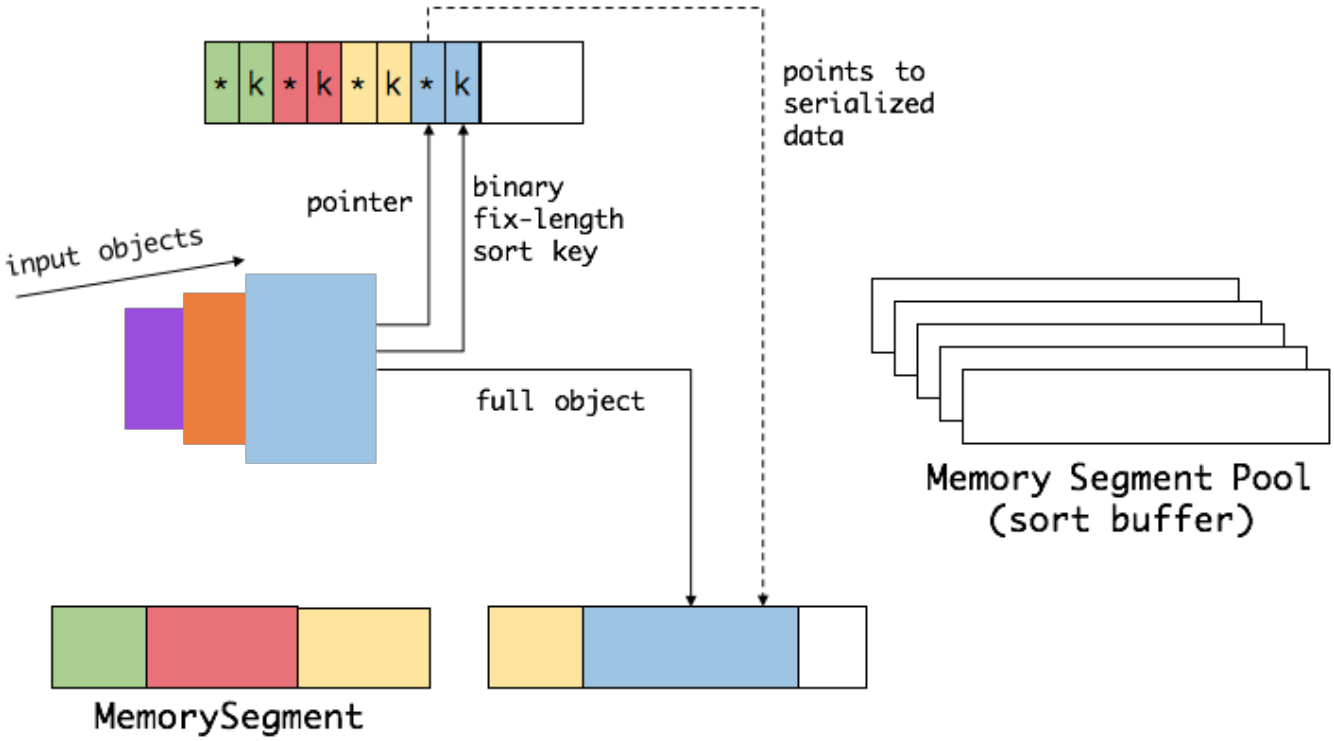
可以看出这种序列化方式存储密度是相当紧凑的。其中 int 占4字节，double 占8字节，POJO多个一个字节的header，PojoSerializer只负责将header序列化进去，并委托每个字段对应的serializer对字段进行序列化。

Flink 的类型系统可以很轻松地扩展出自定义的TypeInformation、Serializer以及Comparator，来提升数据类型在序列化和比较时的性能。

Flink 如何直接操作二进制数据

Flink 提供了如 group、sort、join 等操作，这些操作都需要访问海量数据。这里，我们以sort为例，这是一个在 Flink 中使用非常频繁的操作。

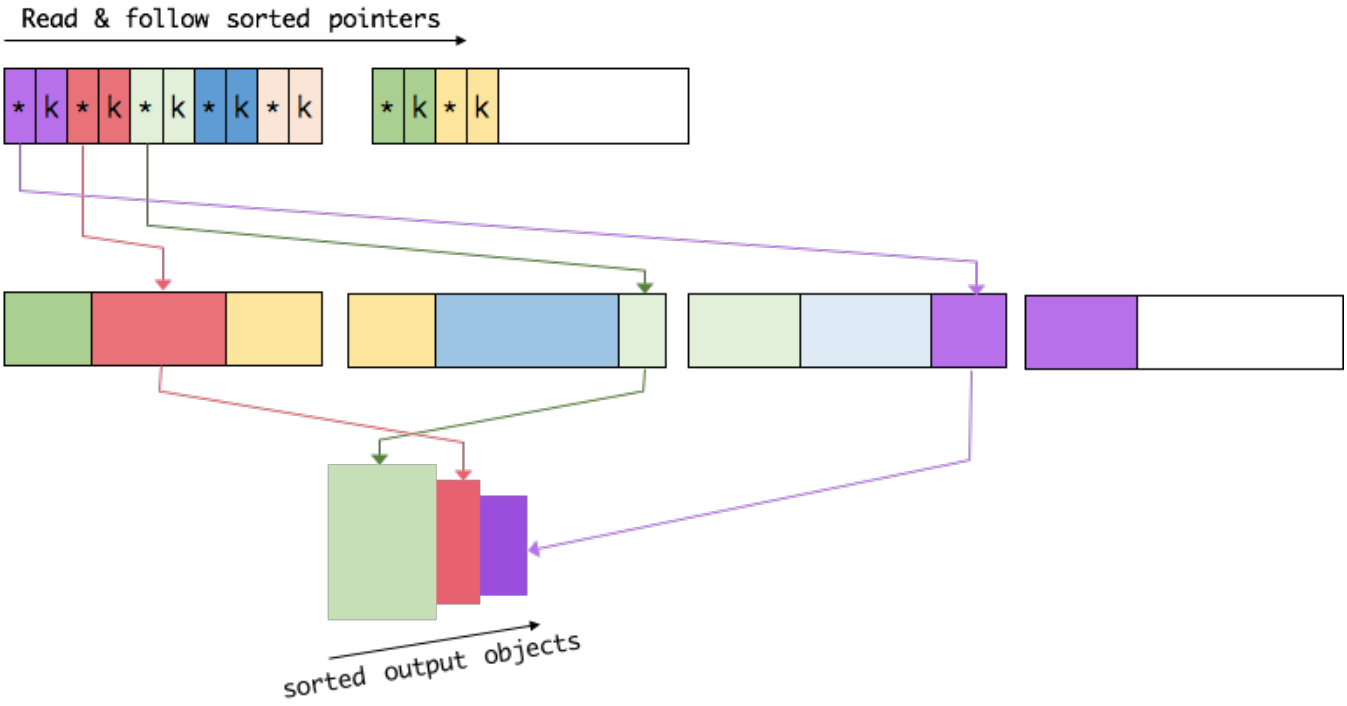
首先，Flink 会从 MemoryManager 中申请一批 MemorySegment，我们把这批 MemorySegment 称作 sort buffer，用来存放排序的数据。



我们会把 sort buffer 分成两块区域。一个区域是用来存放所有对象完整的二进制数据。另一个区域用来存放指向完整二进制数据的指针以及定长的序列化后的key（key+pointer）。如果需要序列化的key是个变长类型，如String，则会取其前缀序列化。如上图所示，当一个对象要加到 sort buffer 中时，它的二进制数据会被加到第一个区域，指针（可能还有key）会被加到第二个区域。

将实际的数据和指针加定长key分开存放有两个目的。第一，交换定长块（key+pointer）更高效，不用交换真实的数据也不用移动其他key和pointer。第二，这样做是缓存友好的，因为key都是连续存储在内存中的，可以大大减少 cache miss（后面会详细解释）。

排序的关键是比大小和交换。Flink 中，会先用 key 比大小，这样就可以直接用二进制的key比较而不需要反序列化出整个对象。因为key是定长的，所以如果key相同（或者没有提供二进制key），那就必须将真实的二进制数据反序列化出来，然后再做比较。之后，只需要交换key+pointer就可以达到排序的效果，真实的数据不用移动。



最后，访问排序后的数据，可以沿着排好序的key+pointer区域顺序访问，通过pointer找到对应的真实数据，并写到内存或外部（更多细节可以看这篇文章 Joins in Flink）。

缓存友好的数据结构和算法

随着磁盘IO和网络IO越来越快，CPU逐渐成为了大数据领域的瓶颈。从 L1/L2/L3 缓存读取数据的速度比从主内存读取数据的速度快好几个量级。通过性能分析可以发现，CPU时间中的很大一部分都是浪费在等待数据从主内存过来上。如果这些数据可以从 L1/L2/L3 缓存过来，那么这些等待时间可以极大地降低，并且所有的算法会因此而受益。

在上面讨论中我们谈到的，Flink 通过定制的序列化框架将算法中需要操作的数据（如sort中的key）连续存储，而完整数据存储在其他地方。因为对于完整的数据来说，key+pointer更容易装进缓存，这大大提高了缓存命中率，从而提高了基础算法的效率。这对于上层应用是完全透明的，可以充分享受缓存友好带来的性能提升。

走向堆外内存

Flink 基于堆内存的内存管理机制已经可以解决很多JVM现存问题了，为什么还要引入堆外内存?

1. 启动超大内存（上百GB）的JVM需要很长时间，GC停留时间也会很长（分钟级）。使用堆外内存的话，可以极大地减小堆内存（只需要分配Remaining Heap那一块），使得 TaskManager 扩展到上百GB内存不是问题。
2. 高效的 IO 操作。堆外内存在写磁盘或网络传输时是 zero-copy，而堆内存的话，至少需要 copy 一次。
3. 堆外内存是进程间共享的。也就是说，即使JVM进程崩溃也不会丢失数据。这可以用来做故

障恢复（Flink暂时没有利用起这个，不过未来很可能会去做）。

但是强大的东西总是会有其负面的一面，不然为何大家不都用堆外内存呢。

1. 堆内存的使用、监控、调试都要简单很多。堆外内存意味着更复杂更麻烦。
2. Flink 有时需要分配短生命周期的 `MemorySegment`，这个申请在堆上会更廉价。
3. 有些操作在堆内存上会快一点点。

Flink用通过`ByteBuffer.allocateDirect(numBytes)`来申请堆外内存，用 `sun.misc.Unsafe` 来操作堆外内存。

基于 Flink 优秀的设计，实现堆外内存是很方便的。Flink 将原来的 `MemorySegment` 变成了抽象类，并生成了两个子类。`HeapMemorySegment` 和 `HybridMemorySegment`。从字面意思上也很容易理解，前者是用来分配堆内存的，后者是用来分配堆外内存**和堆内存**的。是的，你没有看错，后者既可以分配堆外内存又可以分配堆内存。为什么要这样设计呢?

首先假设`HybridMemorySegment`只提供分配堆外内存。在上述堆外内存的不足中的第二点谈到，Flink 有时需要分配短生命周期的 buffer，这些buffer用`HeapMemorySegment`会更高效。那么当使用堆外内存时，为了也满足堆内存的需求，我们需要同时加载两个子类。这就涉及到了 JIT 编译优化的问题。因为以前 `MemorySegment` 是一个单独的 final 类，没有子类。JIT 编译时，所有要调用的方法都是确定的，所有的方法调用都可以被去虚化（de-virtualized）和内联（inlined），这可以极大地提高性能（MemroySegment的使用相当频繁）。然而如果同时加载两个子类，那么 JIT 编译器就只能在真正运行到的时候才知道是哪个子类，这样就无法提前做优化。实际测试的性能差距在 2.7 被左右。

Flink 使用了两种方案：

方案1：只能有一种 MemorySegment 实现被加载

代码中所有的短生命周期和长生命周期的MemorySegment都实例化其中一个子类，另一个子类根本没有实例化过（使用工厂模式来控制）。那么运行一段时间后，JIT 会意识到所有调用的方法都是确定的，然后会做优化。

方案2：提供一种实现能同时处理堆内存和堆外内存

这就是 `HybridMemorySegment` 了，能同时处理堆与堆外内存，这样就不需要子类了。这里 Flink 优雅地实现了一份代码能同时操作堆和堆外内存。这主要归功于 `sun.misc.Unsafe`提供的一系列方法，如getLong方法：

```
sun.misc.Unsafe.getLong(Object reference, long offset)
```

- 如果reference不为空，则会取该对象的地址，加上后面的offset，从相对地址处取出8字节并

得到 long。这对应了堆内存的场景。

- 如果reference为空，则offset就是要操作的绝对地址，从该地址处取出数据。这对应了堆外内存的场景。

这里我们看下 `MemorySegment` 及其子类的实现。

```
public abstract class MemorySegment {
  // 堆内存引用
  protected final byte[] heapMemory;
  // 堆外内存地址
  protected long address;

  //堆内存的初始化
  MemorySegment(byte[] buffer, Object owner) {
    //一些先验检查
    ...
    this.heapMemory = buffer;
    this.address = BYTE_ARRAY_BASE_OFFSET;
    ...
  }

  //堆外内存的初始化
  MemorySegment(long offHeapAddress, int size, Object owner) {
    //一些先验检查
    ...
    this.heapMemory = null;
    this.address = offHeapAddress;
    ...
  }

  public final long getLong(int index) {
    final long pos = address + index;
    if (index >= 0 && pos <= addressLimit - 8) {
      // 这是我们关注的地方，使用 Unsafe 来操作 on-heap & off-heap
      return UNSAFE.getLong(heapMemory, pos);
    }
    else if (address > addressLimit) {
      throw new IllegalStateException("segment has been freed");
    }
    else {
      // index is in fact invalid
      throw new IndexOutOfBoundsException();
    }
  }
  ...
}

public final class HeapMemorySegment extends MemorySegment {
  // 指向heapMemory的额外引用，用来如数组越界的检查
  private byte[] memory;
  // 只能初始化堆内存
```

```
  HeapMemorySegment(byte[] memory, Object owner) {
    super(Objects.requireNonNull(memory), owner);
    this.memory = memory;
  }
  ...
}

public final class HybridMemorySegment extends MemorySegment {
  private final ByteBuffer offHeapBuffer;

  // 堆外内存初始化
  HybridMemorySegment(ByteBuffer buffer, Object owner) {
    super(checkBufferAndGetAddress(buffer), buffer.capacity(), owner);
    this.offHeapBuffer = buffer;
  }

  // 堆内存初始化
  HybridMemorySegment(byte[] buffer, Object owner) {
    super(buffer, owner);
    this.offHeapBuffer = null;
  }
  ...
}
```

可以发现，HybridMemorySegment 中的很多方法其实都下沉到了父类去实现。包括堆内堆外内存的初始化。`MemorySegment` 中的 `getXXX`/`putXXX` 方法都是调用了 unsafe 方法，可以说`MemorySegment`已经具有了些 Hybrid 的意思了。`HeapMemorySegment`只调用了父类的`MemorySegment(byte[] buffer, Object owner)`方法，也就只能申请堆内存。另外，阅读代码你会发现，许多方法（大量的 getXXX/putXXX）都被标记成了 final，两个子类也是 final 类型，为的也是优化 JIT 编译器，会提醒 JIT 这个方法是可以被去虚化和内联的。

对于堆外内存，使用 `HybridMemorySegment` 能同时用来代表堆和堆外内存。这样只需要一个类就能代表长生命周期的堆外内存和短生命周期的堆内存。既然`HybridMemorySegment`已经这么全能，为什么还要方案1呢？因为我们需要工厂模式来保证只有一个子类被加载（为了更高的性能），而且HeapMemorySegment比heap模式的HybridMemorySegment要快。

下方是一些性能测试数据，更详细的数据请参考这篇文章。

Segment	Time
HeapMemorySegment, exclusive	1,441 msecs
HeapMemorySegment, mixed	3,841 msecs
HybridMemorySegment, heap, exclusive	1,626 msecs
HybridMemorySegment, off-heap, exclusive	1,628 msecs

HybridMemorySegment, heap, mixed	3,848 msecs
HybridMemorySegment, off-heap, mixed	3,847 msecs

总结

本文主要总结了 Flink 面对 JVM 存在的问题，而在内存管理的道路上越走越深。从自己管理内存，到序列化框架，再到堆外内存。其实纵观大数据生态圈，其实会发现各个开源项目都有同样的趋势。比如最近炒的很火热的 Spark Tungsten 项目，与 Flink 在内存管理上的思想是及其相似的。

参考资料

- [Off-heap Memory in Apache Flink and the curious JIT compiler](https://flink.apache.org/news/2015/09/16/off-heap-memory.html)
- [Juggling with Bits and Bytes](https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html)
- [Peeking into Apache Flink's Engine Room](https://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html)
- Flink: Memory Management
- [Big Data Performance Engineering](http://www.bigsynapse.com/addressing-big-data-performance)
- sun.misc.misc.Unsafe usage for C style memory management
- sun.misc.misc.Unsafe usage for C style memory management - How to do it.
- Memory usage of Java objects: general guide
- 脱离JVM？ Hadoop生态圈的挣扎与演化