---

title: Apache Flink 是如何管理好内存的？ date: 2019-05-25 tags:

- Flink
- 大数据
- 实时计算

前言

如今，许多用于分析大型数据集的开源系统都是用 Java 或者是基于 JVM 的编程语言实现的。最着名的例子是 Apache Hadoop，还有较新的框架，如 Apache Spark、Apache Drill、Apache Flink。基于 JVM 的数据分析引擎面临的一个常见挑战就是如何在内存中存储大量的数据（包括缓存和高效处理）。合理的管理好 JVM 内存可以将 难以配置且不可预测的系统 与 少量配置且稳定运行的系统区分开来。

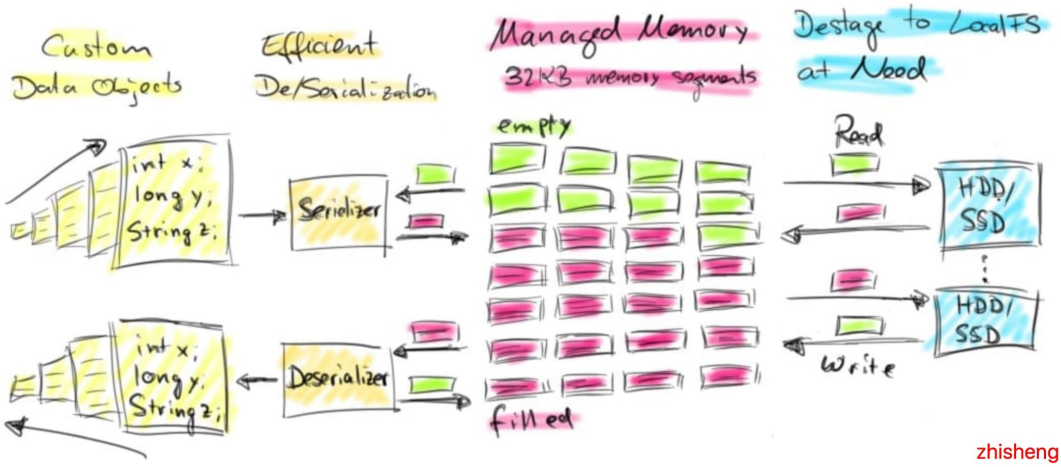
在这篇文章中，我们将讨论 Apache Flink 如何管理内存，讨论其自定义序列化与反序列化机制，以及它是如何操作二进制数据的。

数据对象直接放在堆内存中

在 JVM 中处理大量数据最直接的方式就是将这些数据做为对象存储在堆内存中，然后直接在内存中操作这些数据，如果想进行排序则就是对对象列表进行排序。然而这种方法有一些明显的缺点，首先，在频繁的创建和销毁大量对象的时候，监视和控制堆内存的使用并不是一件很简单的事情。如果对象分配过多的话，那么会导致内存过度使用，从而触发 OutOfMemoryError，导致 JVM 进程直接被杀死。另一个方面就是因为这些对象大都是生存在新生代，当 JVM 进行垃圾回收时，垃圾收集的开销很容易达到 50% 甚至更多。最后就是 Java 对象具有一定的空间开销（具体取决于 JVM 和平台）。对于具有许多小对象的数据集，这可以显著减少有效可用的内存量。如果你精通系统设计和系统调优，你可以根据系统进行特定的参数调整，可以或多或少的控制出现 OutOfMemoryError 的次数和避免堆内存的过多使用，但是这种设置和调优的作用有限，尤其是在数据量较大和执行环境发生变化的情况下。

Flink 是怎么做的?

Apache Flink 起源于一个研究项目，该项目旨在结合基于 MapReduce 的系统和并行数据库系统的最佳技术。在此背景下，Flink 一直有自己的内存数据处理方法。Flink 将对象序列化为固定数量的预先分配的内存段，而不是直接把对象放在堆内存上。它的 DBMS 风格的排序和连接算法尽可能多地对这个二进制数据进行操作，以此将序列化和反序列化开销降到最低。如果需要处理的数据多于可以保存在内存中的数据，Flink 的运算符会将部分数据溢出到磁盘。事实上，很多Flink 的内部实现看起来更像是 C / C ++，而不是普通的 Java。下图概述了 Flink 如何在内存段中存储序列化数据并在必要时溢出到磁盘：



Flink 的主动内存管理和操作二进制数据有几个好处：

1、**内存安全执行和高效的核外算法** 由于分配的内存段的数量是固定的，因此监控剩余的内存资源是非常简单的。在内存不足的情况下，处理操作符可以有效地将更大批的内存段写入磁盘，后面再将它们读回到内存。因此，OutOfMemoryError 就有效的防止了。

2、**减少垃圾收集压力** 因为所有长生命周期的数据都是在 Flink 的管理内存中以二进制表示的，所以所有数据对象都是短暂的，甚至是可变的，并且可以重用。短生命周期的对象可以更有效地进行垃圾收集，这大大降低了垃圾收集的压力。现在，预先分配的内存段是 JVM 堆上的长期存在的对象，为了降低垃圾收集的压力，Flink 社区正在积极地将其分配到堆外内存。这种努力将使得 JVM 堆变得更小，垃圾收集所消耗的时间将更少。

3、**节省空间的数据存储** Java 对象具有存储开销，如果数据以二进制的形式存储，则可以避免这种开销。

4、**高效的二进制操作和缓存敏感性** 在给定合适的二进制表示的情况下，可以有效地比较和操作二进制数据。此外，二进制表示可以将相关值、哈希码、键和指针等相邻地存储在内存中。这使得数据结构通常具有更高效的缓存访问模式。

主动内存管理的这些特性在用于大规模数据分析的数据处理系统中是非常可取的，但是要实现这些功能的代价也是高昂的。要实现对二进制数据的自动内存管理和操作并非易事，使用 `java.util.HashMap` 比实现一个可溢出的 `hash-table`（由字节数组和自定义序列化支持）。当然，Apache Flink 并不是唯一一个基于 JVM 且对二进制数据进行操作的数据处理系统。例如 Apache Drill、Apache Ignite、Apache Geode 也有应用类似技术，最近 Apache Spark 也宣布将向这个方向演进。

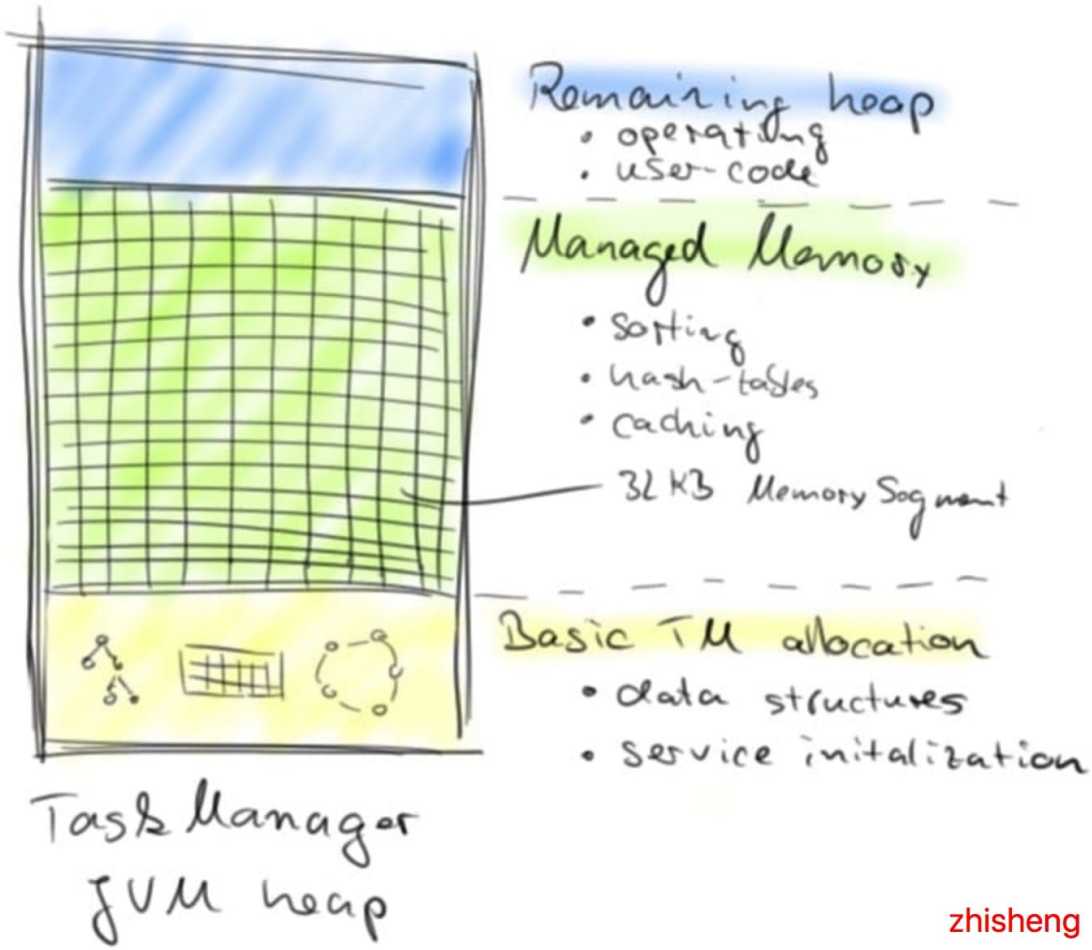
下面我们将详细讨论 Flink 如何分配内存、如果对对象进行序列化和反序列化以及如果对二进制数据进行操作。我们还将通过一些性能表现数据来比较处理堆内存上的对象和对二进制数据的操作。

Flink 如何分配内存?

Flink TaskManager 是由几个内部组件组成的：actor 系统（负责与 Flink master 协调）、IOManager（负责将数据溢出到磁盘并将其读取回来）、MemoryManager（负责协调内存使用）。在本篇文章中，我们主要讲解 MemoryManager。

MemoryManager 负责将 MemorySegments 分配、计算和分发给数据处理操作符，例如 sort 和 join 等操作符。MemorySegment 是 Flink 的内存分配单元，由常规 Java 字节数组支持(默认大小为 32 KB)。MemorySegment 通过使用 Java 的 unsafe 方法对其支持的字节数组提供非常有效的读写访问。你可以将 MemorySegment 看作是 Java 的 NIO ByteBuffer 的定制版本。为了在更大的连续内存块上操作多个 MemorySegment，Flink 使用了实现 Java 的 java.io.DataOutput 和 java.io.DataInput 接口的逻辑视图。

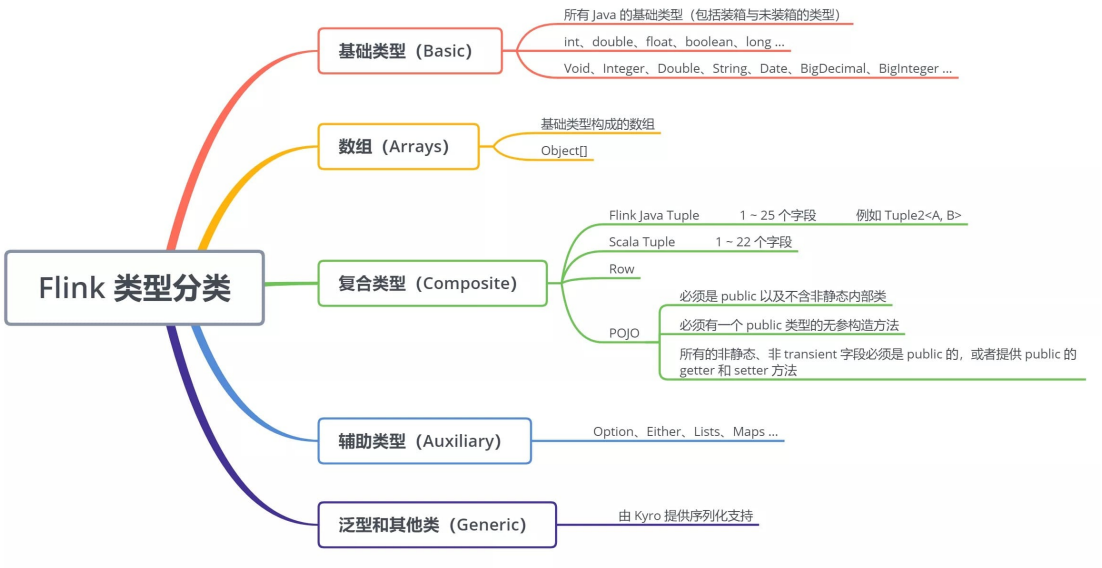
MemorySegments 在 TaskManager 启动时分配一次，并在 TaskManager 关闭时销毁。因此，在 TaskManager 的整个生命周期中，MemorySegment 是重用的，而不会被垃圾收集的。在初始化 TaskManager 的所有内部数据结构并且已启动所有核心服务之后，MemoryManager 开始创建 MemorySegments。默认情况下，服务初始化后，70% 可用的 JVM 堆内存由 MemoryManager 分配（也可以配置全部）。剩余的 JVM 堆内存用于在任务处理期间实例化的对象，包括由用户定义的函数创建的对象。下图显示了启动后 TaskManager JVM 中的内存分布：



Flink 如何序列化对象?

Java 生态系统提供了几个库，可以将对象转换为二进制表示形式并返回。常见的替代方案是标准 Java 序列化，Kryo，Apache Avro，Apache Thrift 或 Google 的 Protobuf。Flink 包含自己的自定义序列化框架，以便控制数据的二进制表示。这一点很重要，因为对二进制数据进行操作需要对序列化布局有准确的了解。此外，根据在二进制数据上执行的操作配置序列化布局可以显著提升性能。Flink 的序列化机制利用了这一特性，即在执行程序之前，要序列化和反序列化的对象的类型是完全已知的。

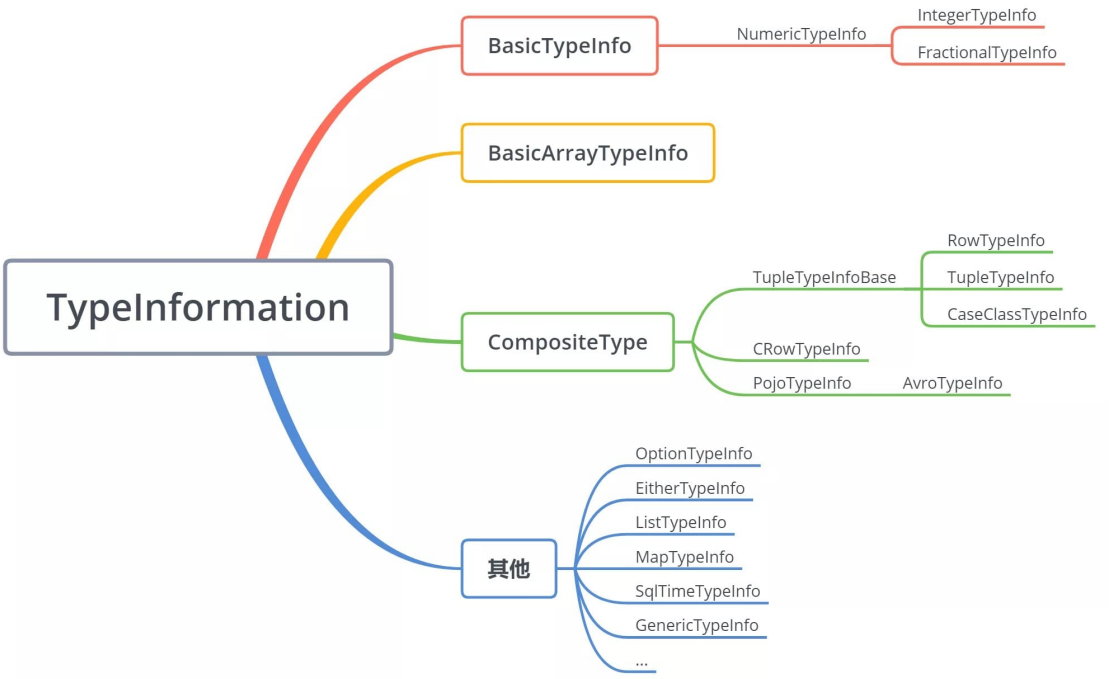
Flink 程序可以处理表示为任意 Java 或 Scala 对象的数据。在优化程序之前，需要识别程序数据流的每个处理步骤中的数据类型。对于 Java 程序，Flink 提供了一个基于反射的类型提取组件，用于分析用户定义函数的返回类型。Scala 程序可以在 Scala 编译器的帮助下进行分析。Flink 使用 TypeInformation 表示每种数据类型。



> 注：该图选自董伟柯的文章《Apache Flink 类型和序列化机制简介》，侵删

Flink 有如下几种数据类型的 TypeInformations：

- BasicTypeInfo：所有 Java 的基础类型或 java.lang.String
- BasicArrayTypeInfo：Java 基本类型构成的数组或 java.lang.String
- WritableTypeInfo：Hadoop 的 Writable 接口的任何实现
- TupleTypeInfo：任何 Flink tuple（Tuple1 到 Tuple25）。Flink tuples 是具有类型化字段的固定长度元组的 Java 表示
- CaseClassTypeInfo：任何 Scala CaseClass（包括 Scala tuples）
- PojoTypeInfo：任何 POJO（Java 或 Scala），即所有字段都是 public 的或通过 getter 和 setter 访问的对象，遵循通用命名约定
- GenericTypeInfo：不能标识为其他类型的任何数据类型

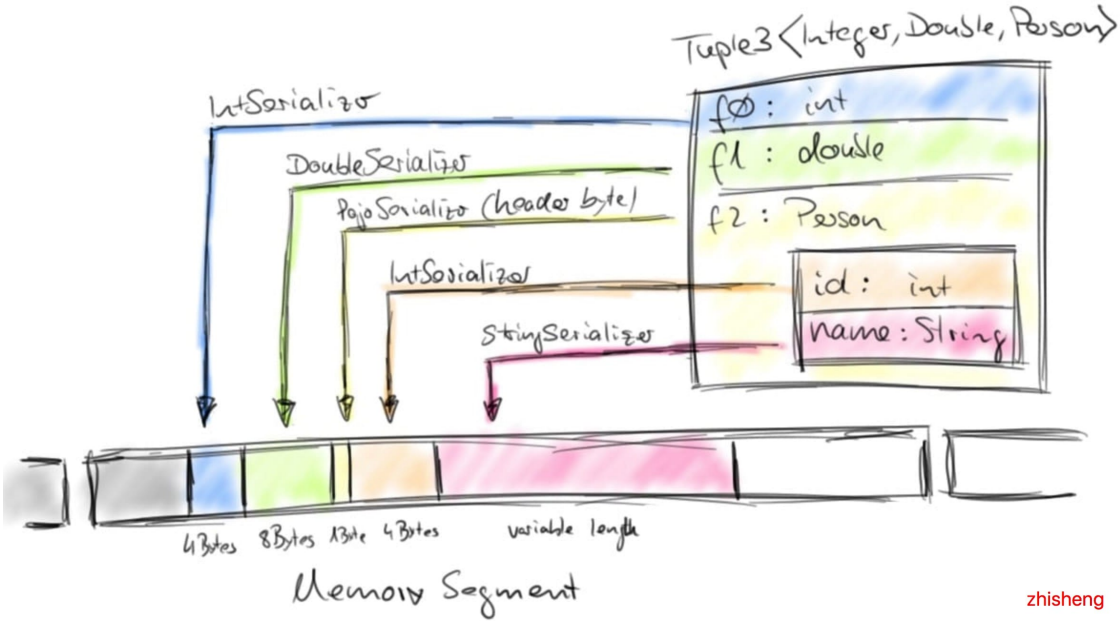


> 注：该图选自董伟柯的文章《Apache Flink 类型和序列化机制简介》，侵删

每个 TypeInformation 都为它所代表的数据类型提供了一个序列化器。例如，BasicTypeInfo 返回一个序列化器，该序列化器写入相应的基本类型；WritableTypeInfo 的序列化器将序列化和反序列化委托给实现 Hadoop 的 Writable 接口的对象的 write() 和 readFields() 方法；GenericTypeInfo 返回一个序列化器，该序列化器将序列化委托给 Kryo。对象将自动通过 Java 中高效的 Unsafe 方法来序列化到 Flink MemorySegments 支持的 DataOutput。对于可用作键的数据类型，例如哈希值，TypeInformation 提供了 TypeComparators，TypeComparators 比较和哈希对象，并且可以根据具体的数据类型有效的比较二进制并提取固定长度的二进制 key 前缀。

Tuple，Pojo 和 CaseClass 类型是复合类型，它们可能嵌套一个或者多个数据类型。因此，它们的序列化和比较也都比较复杂，一般将其成员数据类型的序列化和比较都交给各自的 Serializers（序列化器） 和 Comparators（比较器）。下图说明了 `Tuple3<Integer, Double, Person>` 对象的序列化，其中 `Person` 是 POJO 并定义如下：

```
public class Person {
    public int id;
    public String name;
}
```

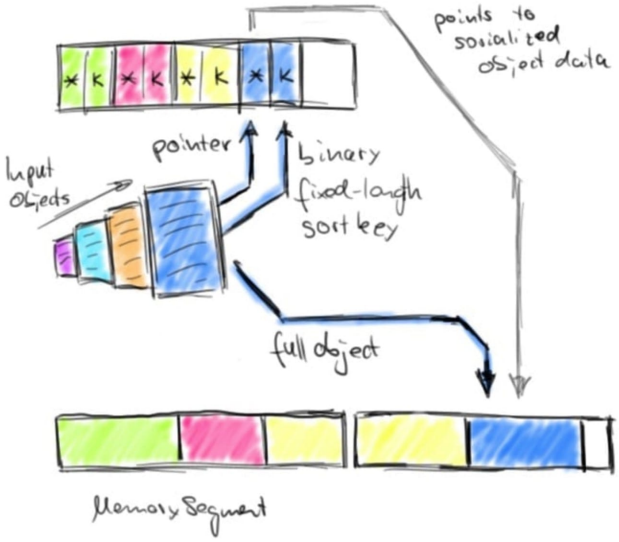


通过提供定制的 TypeInformations、Serializers（序列化器） 和 Comparators（比较器），可以方便地扩展 Flink 的类型系统，从而提高序列化和比较自定义数据类型的性能。

Flink 如何对二进制数据进行操作?

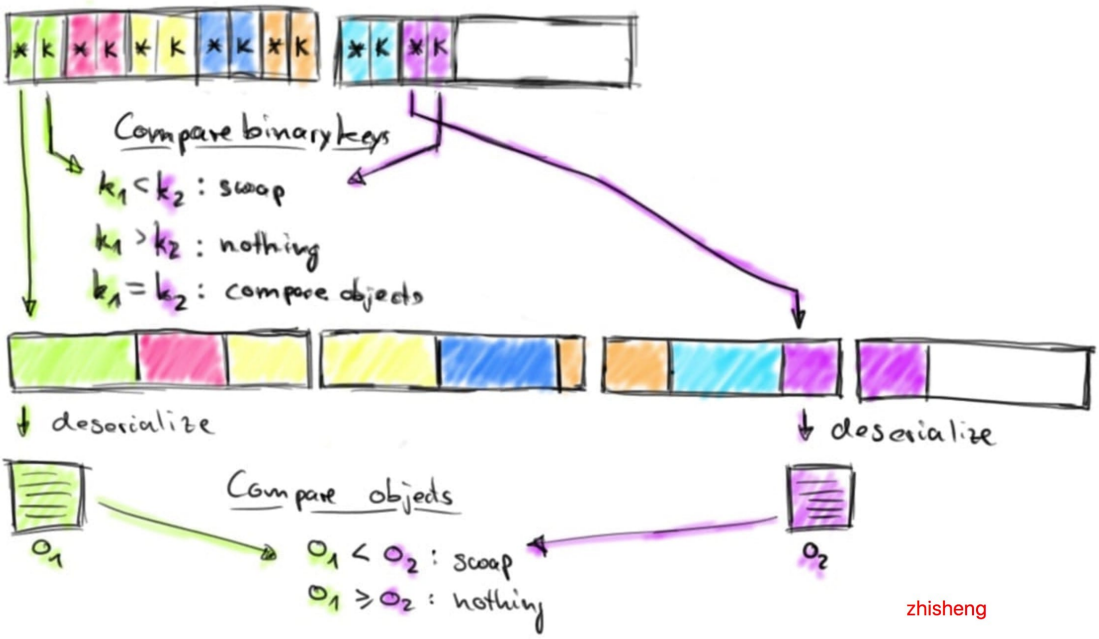
与其他的数据处理框架的 API（包括 SQL）类似，Flink 的 API 也提供了对数据集进行分组、排序和连接等转换操作。这些转换操作的数据集可能非常大。关系数据库系统具有非常高效的算法，比如 merge-sort、merge-join 和 hash-join。Flink 建立在这种技术的基础上，但是主要分为使用自定义序列化和自定义比较器来处理任意对象。在下面文章中我们将通过 Flink 的内存排序算法示例演示 Flink 如何使用二进制数据进行操作。

Flink 为其数据处理操作符预先分配内存，初始化时，排序算法从 MemoryManager 请求内存预算，并接收一组相应的 MemorySegments。这些 MemorySegments 变成了缓冲区的内存池，缓冲区中收集要排序的数据。下图说明了如何将数据对象序列化到排序缓冲区中：

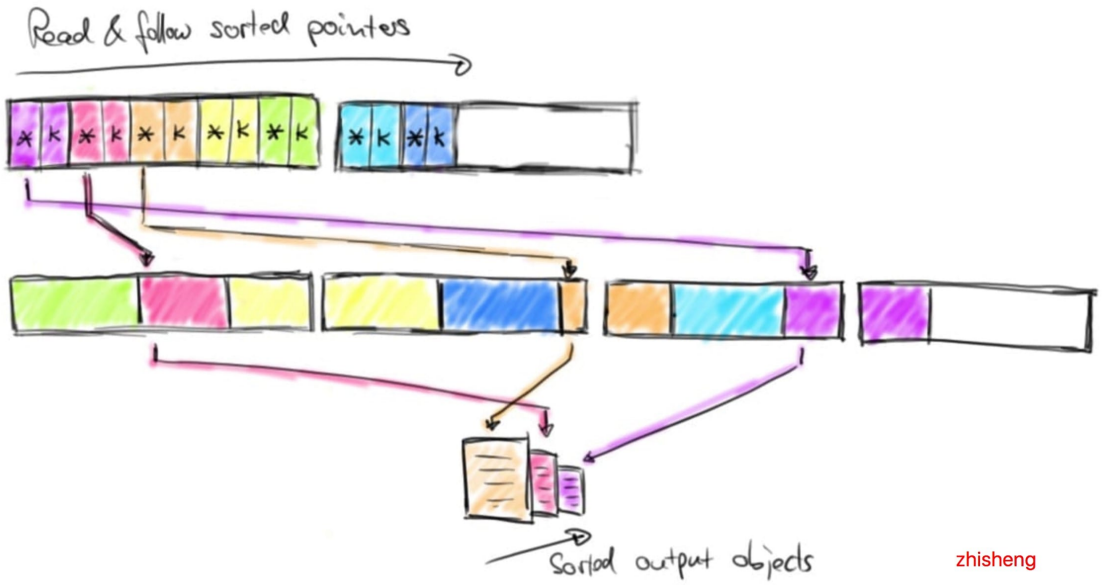


zhisheng

排序缓冲区在内部分为两个内存区域：第一个区域保存所有对象的完整二进制数据，第二个区域包含指向完整二进制对象数据的指针（取决于 key 的数据类型）。将对象添加到排序缓冲区时，它的二进制数据会追加到第一个区域，指针(可能还有一个 key)被追加到第二个区域。分离实际数据和指针以及固定长度的 key 有两个目的：它可以有效的交换固定长度的 entries（key 和指针），还可以减少排序时需要移动的数据。如果排序的 key 是可变长度的数据类型（比如 String），则固定长度的排序 key 必须是前缀 key，比如字符串的前 n 个字符。请注意：并非所有数据类型都提供固定长度的前缀排序 key。将对象序列化到排序缓冲区时，两个内存区域都使用内存池中的 MemorySegments 进行扩展。一旦内存池为空且不能再添加对象时，则排序缓冲区将会被完全填充并可以进行排序。Flink 的排序缓冲区提供了比较和交换元素的方法，这使得实际的排序算法是可插拔的。默认情况下， Flink 使用了 Quicksort（快速排序）实现，可以使用 HeapSort（堆排序）。下图显示了如何比较两个对象：



排序缓冲区通过比较它们的二进制固定长度排序 key 来比较两个元素。如果元素的完整 key（不是前缀 key） 或者二进制前缀 key 不相等，则代表比较成功。如果前缀 key 相等(或者排序 key 的数据类型不提供二进制前缀 key)，则排序缓冲区遵循指向实际对象数据的指针，对两个对象进行反序列化并比较对象。根据比较结果，排序算法决定是否交换比较的元素。排序缓冲区通过移动其固定长度 key 和指针来交换两个元素，实际数据不会移动，排序算法完成后，排序缓冲区中的指针被正确排序。下图演示了如何从排序缓冲区返回已排序的数据：



通过顺序读取排序缓冲区的指针区域，跳过排序 key 并按照实际数据的排序指针返回排序数据。此数据要么反序列化并作为对象返回，要么在外部合并排序的情况下复制二进制数据并将其写入磁盘。

基准测试数据

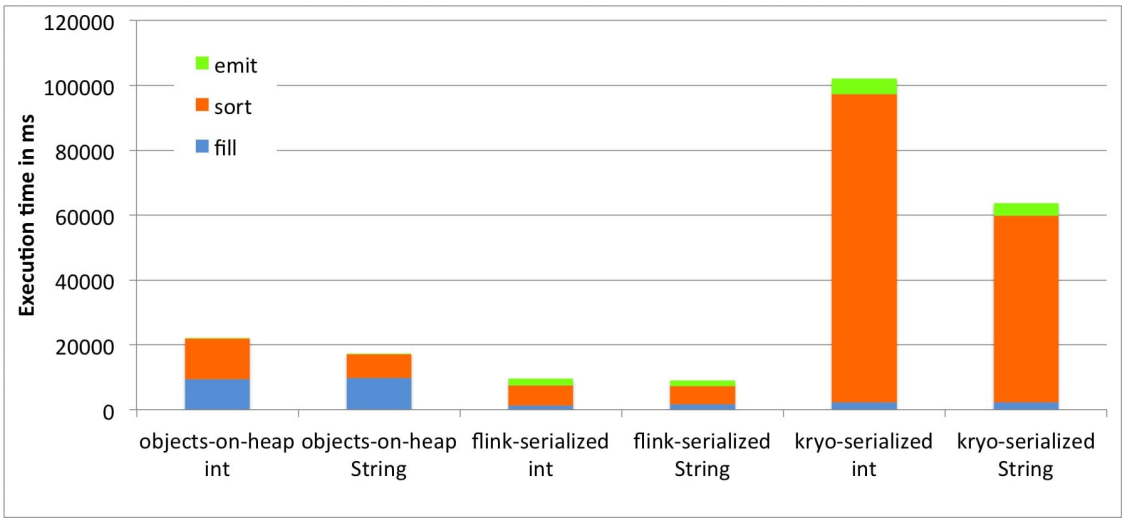
那么，对二进制数据进行操作对性能意味着什么？我们将运行一个基准测试，对 1000 万个 `Tuple2<Integer, String>` 对象进行排序以找出答案。整数字段的值从均匀分布中采样。String 字段值的长度为 12 个字符，并从长尾分布中进行采样。输入数据由返回可变对象的迭代器提供，即返回具有不同字段值的相同 Tuple 对象实例。Flink 在从内存，网络或磁盘读取数据时使用此技术，以避免不必要的对象实例化。基准测试在具有 900 MB 堆大小的 JVM 中运行，在堆上存储和排序 1000 万个 Tuple 对象并且不会导致触发 OutOfMemoryError 大约需要这么大的内存。我们使用三种排序方法在Integer 字段和 String 字段上对 Tuple 对象进行排序：

1、**对象存在堆中**：Tuple 对象存储在常用的 `java.util.ArrayList` 中，初始容量设置为 1000 万，并使用 Java 中常用的集合排序进行排序。

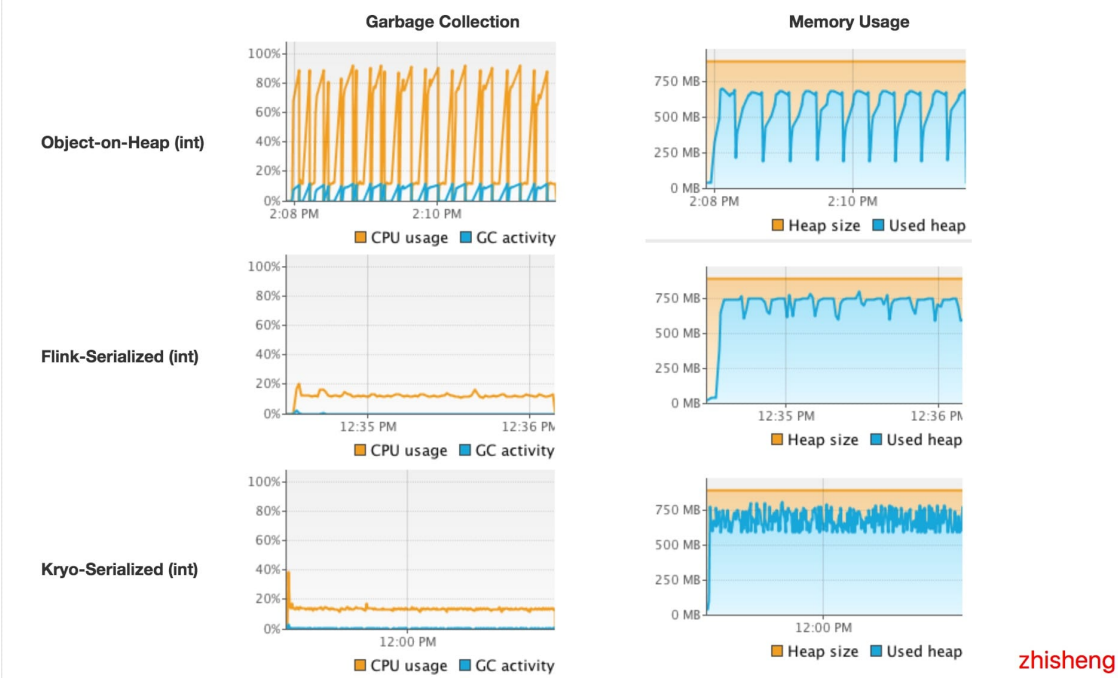
2. **Flink 序列化**：使用 Flink 的自定义序列化程序将 Tuple 字段序列化为 600 MB 大小的排序缓冲区，如上所述排序，最后再次反序列化。在 Integer 字段上进行排序时，完整的 Integer 用作排序 key，以便排序完全发生在二进制数据上（不需要对象的反序列化）。对于 String 字段的排序，使用 8 字节前缀 key，如果前缀 key 相等，则对 Tuple 对象进行反序列化。

3、**Kryo 序列化**：使用 Kryo 序列化将 Tuple 字段序列化为 600 MB 大小的排序缓冲区，并在没有二进制排序 key 的情况下进行排序。这意味着每次比较需要对两个对象进行反序列化。

所有排序方法都使用单线程实现。结果的时间是十次运行结果的平均值。在每次运行之后，我们调用 `System.gc()` 请求垃圾收集运行，该运行不会进入测量的执行时间。下图显示了将输入数据存储在内存中，对其进行排序并将其作为对象读回的时间。



我们看到 Flink 使用自己的序列化器对二进制数据进行排序明显优于其他两种方法。与存储在堆内存上相比，我们看到将数据加载到内存中要快得多。因为我们实际上是在收集对象，没有机会重用对象实例，但必须重新创建每个 Tuple。这比 Flink 的序列化器（或Kryo序列化）效率低。另一方面，与反序列化相比，从堆中读取对象是无性能消耗的。在我们的基准测试中，对象克隆比序列化和反序列化组合更耗性能。查看排序时间，我们看到对二进制数据的排序也比 Java 的集合排序更快。使用没有二进制排序 key 的 Kryo 序列化的数据排序比其他方法慢得多。这是因为反序列化带来很大的开销。在String 字段上对 Tuple 进行排序比在 Integer 字段上排序更快，因为长尾值分布显着减少了成对比较的数量。为了更好地了解排序过程中发生的状况，我们使用 VisualVM 监控执行的 JVM。以下截图显示了执行 10 次 运行时的堆内存使用情况、垃圾收集情况和 CPU 使用情况。



测试是在 8 核机器上运行单线程，因此一个核心的完全利用仅对应 12.5% 的总体利用率。截图显示，对二进制数据进行操作可显著减少垃圾回收活动。对于对象存在堆中，垃圾收集器在排序缓冲区被填满时以非常短的时间间隔运行，并且即使对于单个处理线程也会导致大量 CPU 使用（排序本身不会触发垃圾收集器）。JVM 垃圾收集多个并行线程，解释了高CPU 总体利用率。另一方面，对序列化数据进行操作的方法很少触发垃圾收集器并且 CPU 利用率低得多。实际上，如果使用 Flink 序列化的方式在 Integer 字段上对 Tuple 进行排序，则垃圾收集器根本不运行，因为对于成对比较，不需要反序列化任何对象。Kryo 序列化需要比较多的垃圾收集，因为它不使用二进制排序 key 并且每次排序都要反序列化两个对象。

内存使用情况上图显示 Flink 序列化和 Kryo 序列化不断的占用大量内存

存使用情况图表显示flink-serialized和kryo-serialized不断占用大量内存。这是由于 MemorySegments 的预分配。实际内存使用率要低得多，因为排序缓冲区并未完全填充。下表显示了每种方法的内存消耗。1000 万条数据产生大约 280 MB 的二进制数据（对象数据、指针和排序 key），具体取决于使用的序列化程序以及二进制排序 key 的存在和大小。将其与数据存储在堆上的方法进行比较，我们发现对二进制数据进行操作可以显著提高内存效率。在我们的基准测试中，如果序列化为排序缓冲区而不是将其作为堆上的对象保存，则可以在内存中对两倍以上的数据进行排序。

占用内存	对象存在堆中	Flink 序列化	Kryo 序列化
对 Integer 排序	约 700 MB（堆内存）	277 MB（排序缓冲区）	266 MB（排序缓冲区）
对 String 排序	约 700 MB（堆内存）	315 MB（排序缓冲区）	266 MB（排序缓冲区）

总而言之，测试验证了文章前面说的对二进制数据进行操作的好处。

展望未来

Apache Flink 具有相当多的高级技术，可以通过有限的内存资源安全有效地处理大量数据。但是有几点可以使 Flink 更有效率。Flink 社区正在努力将管理内存移动到堆外内存。这将允许更小的 JVM，更低的垃圾收集开销，以及更容易的系统配置。使用 Flink 的 Table API，所有操作（如 aggregation 和 projection）的语义都是已知的（与黑盒用户定义的函数相反）。因此，我们可以为直接对二进制数据进行操作的 Table API 操作生成代码。进一步的改进包括序列化设计，这些设计针对应用于二进制数据的操作和针对序列化器和比较器的代码生成而定制。

总结

- Flink 的主动内存管理减少了因触发 OutOfMemoryErrors 而杀死 JVM 进程和垃圾收集开销的问题。
- Flink 具有高效的数据序列化和反序列化机制，有助于对二进制数据进行操作，并使更多数据适合内存。
- Flink 的 DBMS 风格的运算符本身在二进制数据上运行，在必要时可以在内存中高性能地传输到磁盘。

> 本文翻译自：https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html 翻译：zhisheng，二次转载请注明地址，否则保留追究法律责任。

更多文章

更多私密资料请加入知识星球！

Flink 精进学习

星主: zhisheng

知识星球

微信扫描预览星球详情