12 | 外部内存接口：零拷贝的障碍还有多少？

2021-12-10 范学雷

《深入剖析Java新特性》 课程介绍 >

讲述：范学雷
时长 08:08 大小 7.46M

你好，我是范学雷。今天，我们来讨论 Java 的外部内存接口。

Java 的外部内存接口这个新特性，现在还在孵化期，还没有发布预览版。我之所以选取了这样一个还处于孵化期的技术，主要是因为这个技术太重要了。我们需要提前认识它；然后在这项技术出来的时候，尽早地使用它。

我们从阅读案例开始，看一看 Java 在没有外部内存接口的时候，是怎么支持本地内存的；然后，我们再看看外部内存接口能够给我们的代码带来什么样的变化。

阅读案例

在我们讨论代码性能的时候，内存的使用效率是一个绕不开的话题。像 TensorFlow、Ignite、 Flink 以及 Netty 这样的类库，往往对性能有着偏执的追求。为了避免 Java 垃圾收集器不可预测的行为以及额外的性能开销，这些产品一般倾向于使用 JVM 之外的内存来存储和管理数据。这样的数据，就是我们常说的堆外数据（off-heap data）。

使用堆外存储最常用的办法，就是使用 ByteBuffer 这个类来分配直接存储空间（direct buffer）。JVM 虚拟机会尽最大努力直接在直接存储空间上执行 IO 操作，避免数据在本地和 JVM 之间的拷贝。

由于频繁的内存拷贝是性能的主要障碍之一。所以为了极致的性能，应用程序通常也会尽量避免内存的拷贝。理想的状况下，一份数据只需要一份内存空间，这就是我们常说的零拷贝。

下面的这段代码，就是用 ByteBuffer 这个类来分配直接存储空间的方法。

复制代码

```
1 public static ByteBuffer allocateDirect(int capacity);
```

ByteBuffer 所在的 Java 包是 java.nio。从这个 Java 包的命名我们就能感受到，ByteBuffer 设计的初衷是用于非阻塞编程的。的确，ByteBuffer 是异步编程和非阻塞编程的核心类，几乎所有的 Java 异步模式或者非阻塞模式的代码，都要直接或者间接地使用 ByteBuffer 来管理数据。

非阻塞和异步编程模式的出现，起始于对于阻塞式文件描述符（File descriptor）（包括网络套接字）读取性能的不满。而诞生于 2002 年的 ByteBuffer，最初的设想也主要是用来解决当时文件描述符的读写性能的。所以，它的设计也不能跳脱出当时的客观需求。

如果站在现在的角度重新审视这个类的设计，我们会发现它主要有两个缺陷。

第一个缺陷是没有资源释放的接口。一旦一个 ByteBuffer 实例化，它占用了内存的释放，就会完全依赖 JVM 的垃圾回收机制。使用直接存储空间的应用，往往需要把所有潜在的性能都挤压出来。依赖于垃圾回收机制的资源回收方式，并不能满足像 Netty 这样的类库的理想需求。

第二个缺陷是存储空间尺寸的限制。ByteBuffer 的存储空间的大小，是使用 Java 的整数来表示的。所以，它的存储空间，最多只有 2G。这是一个无意带来的缺陷。在网络编程的环境下，这并不是一个问题。可是，超过 2G 的文件，一定会越来越多；2G 以上的文件，映射到 ByteBuffer 上的时候，就会出现文件过大的问题。而像 Memcahed 这样的分布式内存，也会让应用程序需要控制的内存超越 2G 的界限。

这两个缺陷，也是横隔在“零拷贝”这个理想路上的两个主要设计障碍。

对于第一个缺陷，我们还可以在 ByteBuffer 的基础上修改，并且保持这个类的优雅。但是第二个缺陷，由于 ByteBuffer 类里到处都在使用的整数类型，我们就很难找到办法既保持这个类的优雅，又能够突破存储空间的尺寸限制了。

一个合理的改进，就是重新建造一个轮子。这个新的轮子，就是外部内存接口。

外部内存接口

外部内存接口沿袭了 ByteBuffer 的设计思路，但是使用了全新的接口布局。我们先来看看使用外部内存接口的代码看起来是什么样子的。下面的这段代码，要分配一段外部内存，并且存放 4 个字母 A。

复制代码

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    MemorySegment segment = MemorySegment.allocateNative(4, scope);
    for (int i = 0; i < 4; i++) {
        MemoryAccess.setByteAtOffset(segment, i, (byte)'A');
    }
}
现在，我们通过这个小例子，来看看外部内存接口的布局。

```

第一行的 ResourceScope 这个类，定义了内存资源的生命周期管理机制。这是一个实现了 AutoCloseable 的接口。我们就可以使用 try-with-resource 这样的语句，及时地释放掉它管理的内存了。这样的设计，就解决了 ByteBuffer 的第一个缺陷。

第二行的 MemorySegment 这个类，定义和模拟了一段连续的内存区域。第三行的 MemoryAccess 这个类，定义了可以对 MemorySegment 执行读写操作。在 ByteBuffer 的设计里，内存的表达和操作，是在 ByteBuffer 这一个类里完成的。在外部内存接口的设

计里，把对象表达和对象的操作，拆分成了两个类。这两类的寻址数据类型，使用的是长整形（long）。这样，长整形的寻址类型，就解决了 ByteBuffer 的第二个缺陷。

超预期的演进

无论是在我们生活的现实世界里，还是在软件的虚拟世界里，只要我们超前迈出了第一步，后续的发展往往会超出我们的预料。外部内存接口的出现，虽然还处在孵化期，也带来了远远超出预期的精彩局面。

在计算机的世界里，代码主要和两类计算资源打交道。一类是负责控制和运算的处理器；一类是临时存放运算数据的存储器。表现到编程语言的层面，就是函数和内存。函数之间的数据传递，也是用过内存的形式进行的。

现在，外部内存接口为我们提供了一个统一的内存操作接口。对应地，外部函数之间的数据传递问题也就有了思路。既然能够解决函数之间的数据传递问题，那么，不同语言间的函数调用能不能变得更简单、更有效率呢？

这个问题，就是我们下一次要讨论的内容。如果说，设计外部内存接口的最初动力是为了解决 ByteBuffer 的两个缺陷。那研发的持续推进，则给外部内存接口赋予了更大的责任和能量。

总结

好，到这里，我来做个小结。前面，我们讨论了 Java 的外部内存接口这个尚处于孵化阶段的新特性，对外部内存接口这个新特性有了一个初始的印象。

设计外部内存接口的最初动力，是为了解决 ByteBuffer 的两个缺陷。也就是 ByteBuffer 占用的资源不能及时释放，以及它的寻址空间太小这两个问题。但是外部内存接口的更大使命，是和外部函数接口联系在一起的。我们下一次再讨论这个更大的使命。

如果外部内存接口正式发布出来，现在使用 ByteBuffer 的类库（比如 Flink 和 Netty，甚至 JDK 本身），应该可以考虑切换到外部内存接口来获取性能的提升。

这一次学习的主要目的，就是让你对外部内存接口有一个基本的印象。由于外部内存接口尚处于孵化阶段，现在我们还不需要学习它的 API。只要知道 Java 有这个发展方向，能够

了解 ByteBuffer 的这两个缺陷能够给你的程序带来的影响就足够了。

如果面试中聊到了 ByteBuffer，你应该可以聊一聊零拷贝，以及 ByteBuffer 的这两个缺陷，还有未来的 Java 要做的改进。

思考题

其实，今天的这个新特性，也是练习使用 JShell 快速学习新技术的一个好机会。我们在前面的讨论里，分析了下面的这段代码。为了方便你阅读，我把这段代码重新拷贝到下面了。

复制代码

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    MemorySegment segment = MemorySegment.allocateNative(4, scope);
    for (int i = 0; i < 4; i++) {
        MemoryAccess.setByteAtOffset(segment, i, (byte)'A');
    }
}
```

虽然我们提到了使用 try-with-resource 这样的语句，可以及时地释放掉它管理的内存。但是，我们并没有验证这一说法。你能不能使用 JShell，快速地验证它的资源释放效果呢？

需要注意的是，要想使用孵化期的 JDK 技术，需要在 JShell 里导入孵化期的 JDK 模块。就像下面的例子这样。

复制代码

```
$ jshell --add-modules jdk.incubator.foreign -v
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro

jshell> import jdk.incubator.foreign.*;
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！

注：本文使用的完整的代码可以从GitHub下载，你可以通过修改GitHub上review template代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请

提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在外部内存接口专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 memory/review/xuelei 的目录下面。

上一篇 11 | 矢量运算：Java的机器学习要来了吗？

精选留言 (2)

写留言

松松
2021-12-10

```
jshell> try (ResourceScope scope = ResourceScope.newConfinedScope()) {
   ...> MemorySegment segment = MemorySegment.allocateNative(4, scope);
   ...> for (int i = 0; i < 4; i++) {
   ...> MemoryAccess.setByteAtOffset(segment, i, (byte)'A');
   ...> }...
```

展开

作者回复: 像是范例，谢谢！

1

aoe
2021-12-10

知道了大名鼎鼎的ByteBuffer有两个缺陷：
1. 没有资源释放的接口
2. 存储空间尺寸的限制。ByteBuffer 的存储空间的大小，是使用 Java 的整数来表示的。

所以，它的存储空间，最多只有 2G。

...

展开

作者回复: 虽然还在孵化期，还是值得期待的。