二

# 08 网络通信优化之IO模型：如何解决高并发下IO瓶颈?

你好，我是刘超。

提到 Java I/O，相信你一定不陌生。你可能使用 I/O 操作读写文件，也可能使用它实现 Socket 的信息传输…这些都是我们在系统中最常遇到的和 I/O 有关的操作。

我们都知道，I/O 的速度要比内存速度慢，尤其是在现在这个大数据时代背景下，I/O 的性能问题更是尤为突出，I/O 读写已经成为很多应用场景下的系统性能瓶颈，不容我们忽视。

今天，我们就来深入了解下 Java I/O 在高并发、大数据业务场景下暴露出的性能问题，从源头入手，学习优化方法。

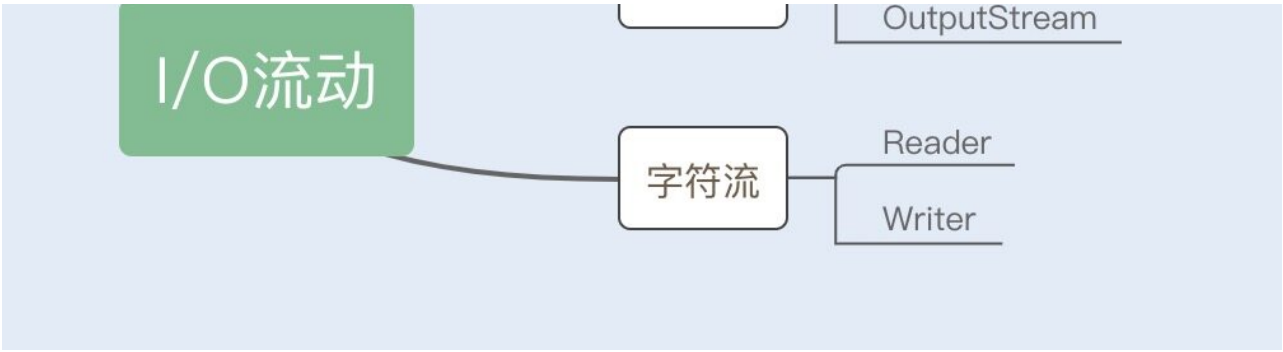
## 什么是 I/O

I/O 是机器获取和交换信息的主要渠道，而流是完成 I/O 操作的主要方式。

在计算机中，流是一种信息的转换。流是有序的，因此相对于某一机器或者应用程序而言，我们通常把机器或者应用程序接收外界的信息称为输入流（InputStream），从机器或者应用程序向外输出的信息称为输出流（OutputStream），合称为输入 / 输出流（I/O Streams）。

机器间或程序间在进行信息交换或者数据交换时，总是先将对象或数据转换为某种形式的流，再通过流的传输，到达指定机器或程序后，再将流转换为对象数据。因此，流就可以被看作是一种数据的载体，通过它可以实现数据交换和传输。

Java 的 I/O 操作类在包 java.io 下，其中 InputStream、OutputStream 以及 Reader、Writer 类是 I/O 包中的 4 个基本类，它们分别处理字节流和字符流。如下图所示：

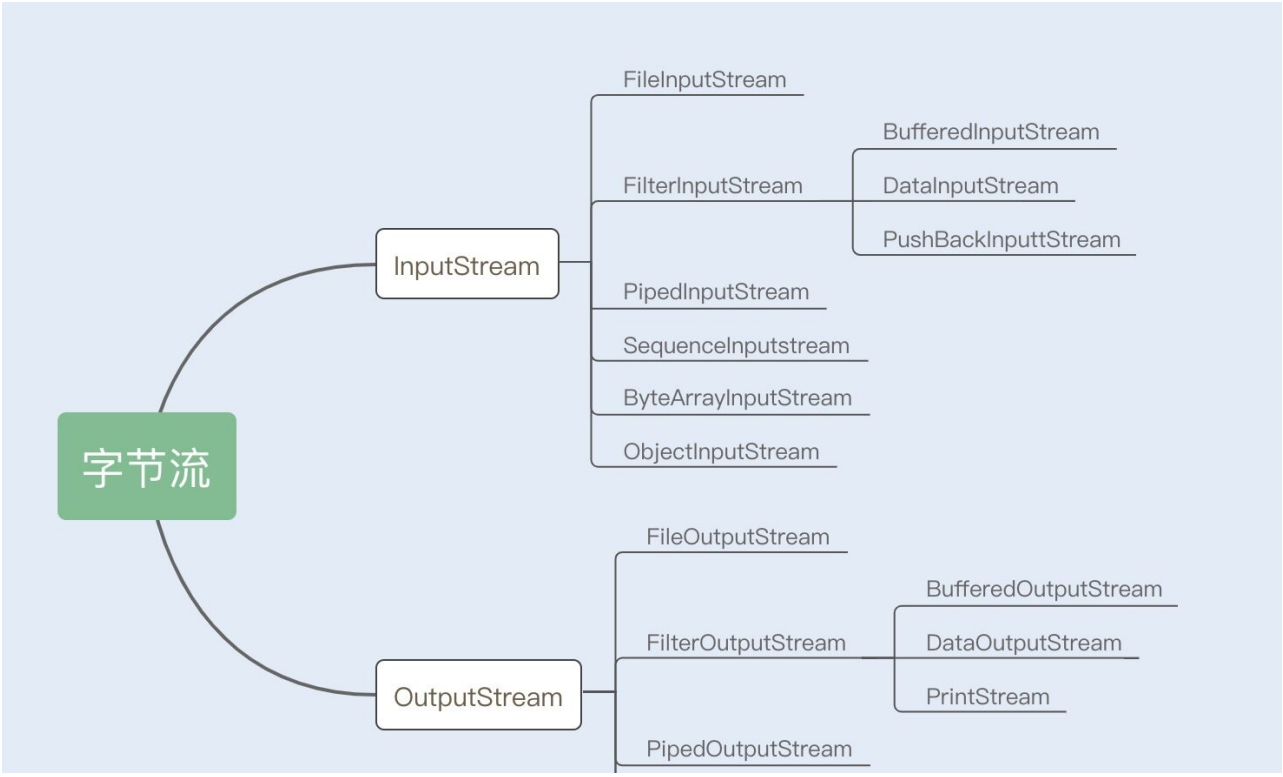


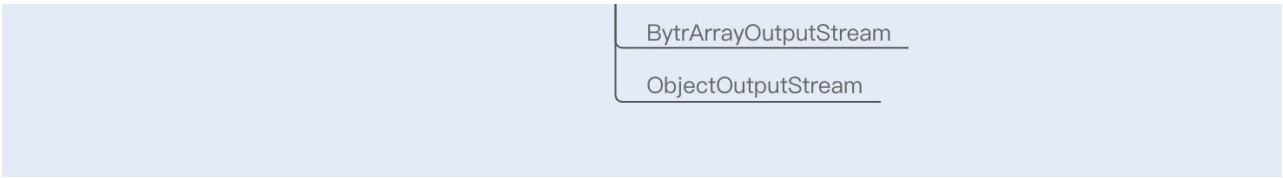

回顾我的经历，我记得在初次阅读 Java I/O 流文档的时候，我有过这样一个疑问，在这里也分享给你，那就是：“**不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？**”

我们知道字符到字节必须经过转码，这个过程非常耗时，如果我们不知道编码类型就很容易出现乱码问题。所以 I/O 流提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。下面我们就分别了解下“字节流”和“字符流”。

## 1. 字节流

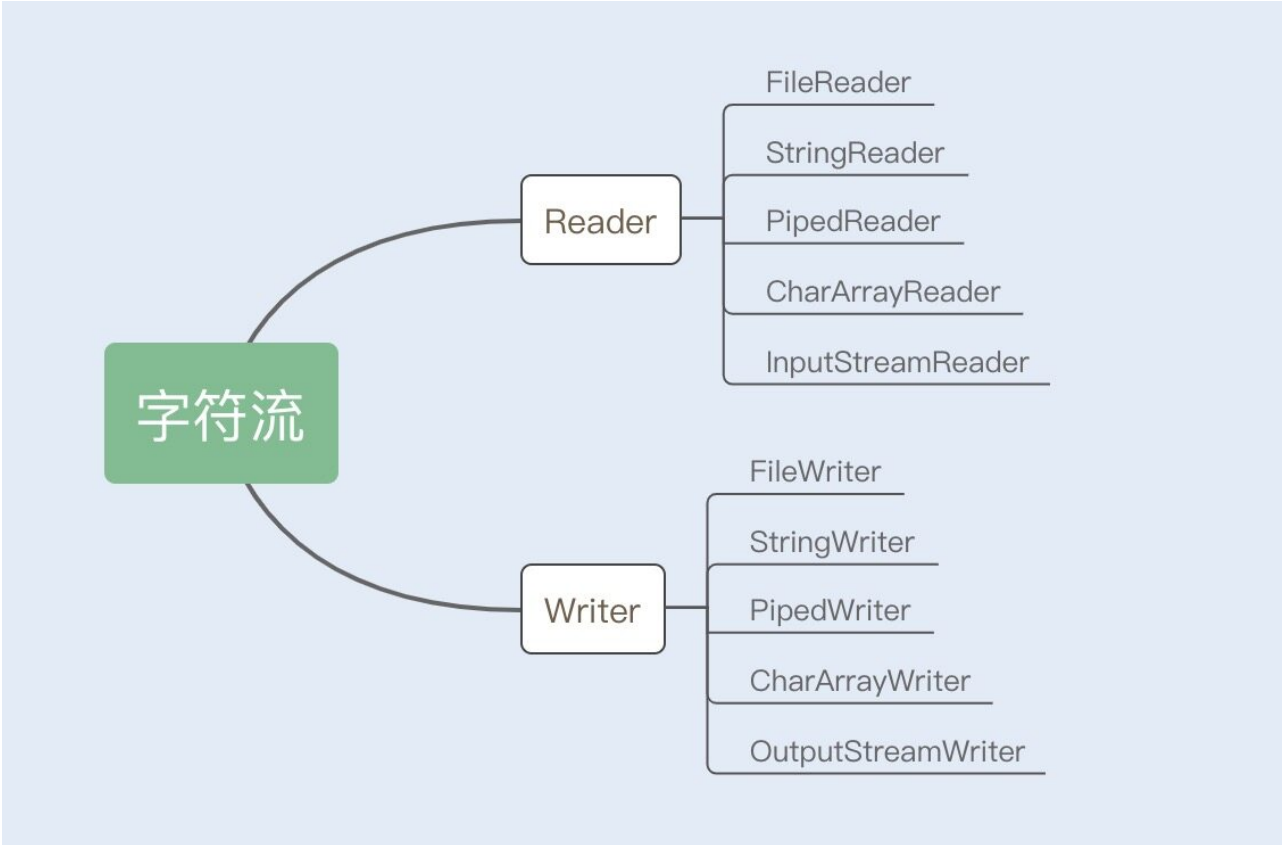
InputStream/OutputStream 是字节流的抽象类，这两个抽象类又派生出了若干子类，不同的子类分别处理不同的操作类型。如果是文件的读写操作，就使用 FileInputStream/FileOutputStream；如果是数组的读写操作，就使用 ByteArrayInputStream/ByteArrayOutputStream；如果是普通字符串的读写操作，就使用 BufferedInputStream/BufferedOutputStream。具体内容如下图所示：






### 2. 字符流

Reader/Writer 是字符流的抽象类，这两个抽象类也派生出了若干子类，不同的子类分别处理不同的操作类型，具体内容如下图所示：



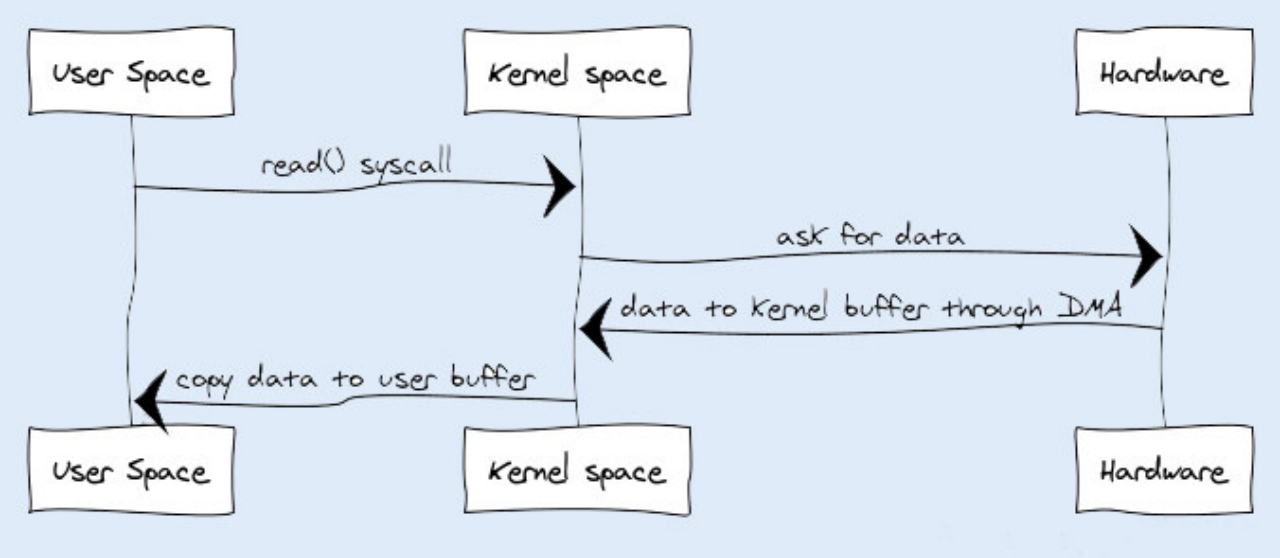

## 传统 I/O 的性能问题

我们知道，I/O 操作分为磁盘 I/O 操作和网络 I/O 操作。前者是从磁盘中读取数据源输入到内存中，之后将读取的信息持久化输出在物理磁盘上；后者是从网络中读取信息输入到内存，最终将信息输出到网络中。但不管是磁盘 I/O 还是网络 I/O，在传统 I/O 中都存在严重的性能问题。

### 1. 多次内存复制

在传统 I/O 中，我们可以通过 InputStream 从源数据中读取数据流输入到缓冲区里，通过

OutputStream 将数据输出到外部设备（包括磁盘、网络）。你可以先看下输入操作在操作系统中的具体流程，如下图所示：




- JVM 会发出 read() 系统调用，并通过 read 系统调用向内核发起读请求；
- 内核向硬件发送读指令，并等待读就绪；
- 内核把将要读取的数据复制到指向的内核缓存中；
- 操作系统内核将数据复制到用户空间缓冲区，然后 read 系统调用返回。

在这个过程中，数据先从外部设备复制到内核空间，再从内核空间复制到用户空间，这就发生了两次内存复制操作。这种操作会导致不必要的数据拷贝和上下文切换，从而降低 I/O 的性能。

### 2. 阻塞

在传统 I/O 中，InputStream 的 read() 是一个 while 循环操作，它会一直等待数据读取，直到数据就绪才会返回。**这就意味着如果没有数据就绪，这个读取操作将会一直被挂起，用户线程将会处于阻塞状态。**

在少量连接请求的情况下，使用这种方式没有问题，响应速度也很高。但在发生大量连接请求时，就需要创建大量监听线程，这时如果线程没有数据就绪就会被挂起，然后进入阻塞状态。一旦发生线程阻塞，这些线程将会不断地抢夺 CPU 资源，从而导致大量的 CPU 上下文切换，增加系统的性能开销。

## 如何优化 I/O 操作

面对以上两个性能问题，不仅编程语言对此做了优化，各个操作系统也进一步优化了 I/O。JDK1.4 发布了 java.nio 包（new I/O 的缩写），NIO 的发布优化了内存复制以及阻塞导致的严重性能问题。JDK1.7 又发布了 NIO2，提出了从操作系统层面实现的异步 I/O。下面我们就来了解下具体的优化实现。

## 1. 使用缓冲区优化读写流操作

在传统 I/O 中，提供了基于流的 I/O 实现，即 InputStream 和 OutputStream，这种基于流的实现以字节为单位处理数据。

NIO 与传统 I/O 不同，它是基于块（Block）的，它以块为基本单位处理数据。在 NIO 中，最为重要的两个组件是缓冲区（Buffer）和通道（Channel）。Buffer 是一块连续的内存块，是 NIO 读写数据的中转地。Channel 表示缓冲数据的源头或者目的地，它用于读取缓冲或者写入数据，是访问缓冲的接口。

传统 I/O 和 NIO 的最大区别就是传统 I/O 是面向流，NIO 是面向 Buffer。Buffer 可以将文件一次性读入内存再做后续处理，而传统的方式是边读文件边处理数据。虽然传统 I/O 后面也使用了缓冲块，例如 BufferedInputStream，但仍然不能和 NIO 相媲美。使用 NIO 替代传统 I/O 操作，可以提升系统的整体性能，效果立竿见影。

## 2. 使用 DirectBuffer 减少内存复制

NIO 的 Buffer 除了做了缓冲块优化之外，还提供了一个可以直接访问物理内存的类 DirectBuffer。普通的 Buffer 分配的是 JVM 堆内存，而 DirectBuffer 是直接分配物理内存。

我们知道数据要输出到外部设备，必须先从用户空间复制到内核空间，再复制到输出设备，而 DirectBuffer 则是直接将步骤简化为从内核空间复制到外部设备，减少了数据拷贝。

这里拓展一点，由于 DirectBuffer 申请的是非 JVM 的物理内存，所以创建和销毁的代价很高。DirectBuffer 申请的内存并不是直接由 JVM 负责垃圾回收，但在 DirectBuffer 包装类被回收时，会通过 Java Reference 机制来释放该内存块。

## 3. 避免阻塞，优化 I/O 操作

NIO 很多人也称之为 Non-block I/O，即非阻塞 I/O，因为这样叫，更能体现它的特点。为什么这么说呢？

传统的 I/O 即使使用了缓冲块，依然存在阻塞问题。由于线程池线程数量有限，一旦发生大量并发请求，超过最大数量的线程就只能等待，直到线程池中有空闲的线程可以被复用。而对 Socket 的输入流进行读取时，读取流会一直阻塞，直到发生以下三种情况的任意一种才

会解除阻塞：

- 有数据可读；
- 连接释放；
- 空指针或 I/O 异常。

阻塞问题，就是传统 I/O 最大的弊端。NIO 发布后，通道和多路复用器这两个基本组件实现了 NIO 的非阻塞，下面我们就一起来了解下这两个组件的优化原理。

**通道（Channel）**

前面我们讨论过，传统 I/O 的数据读取和写入是从用户空间到内核空间来回复制，而内核空间的数据是通过操作系统层面的 I/O 接口从磁盘读取或写入。

最开始，在应用程序调用操作系统 I/O 接口时，是由 CPU 完成分配，这种方式最大的问题是“发生大量 I/O 请求时，非常消耗 CPU“；之后，操作系统引入了 DMA（直接存储器存储），内核空间与磁盘之间的存取完全由 DMA 负责，但这种方式依然需要向 CPU 申请权限，且需要借助 DMA 总线来完成数据的复制操作，如果 DMA 总线过多，就会造成总线冲突。

通道的出现解决了以上问题，Channel 有自己的处理器，可以完成内核空间和磁盘之间的 I/O 操作。在 NIO 中，我们读取和写入数据都要通过 Channel，由于 Channel 是双向的，所以读、写可以同时进行。

**多路复用器（Selector）**

Selector 是 Java NIO 编程的基础。用于检查一个或多个 NIO Channel 的状态是否处于可读、可写。

Selector 是基于事件驱动实现的，我们可以在 Selector 中注册 accpet、read 监听事件，Selector 会不断轮询注册在其上的 Channel，如果某个 Channel 上面发生监听事件，这个 Channel 就处于就绪状态，然后进行 I/O 操作。

一个线程使用一个 Selector，通过轮询的方式，可以监听多个 Channel 上的事件。我们可以在注册 Channel 时设置该通道为非阻塞，当 Channel 上没有 I/O 操作时，该线程就不会一直等待了，而是会不断轮询所有 Channel，从而避免发生阻塞。

目前操作系统的 I/O 多路复用机制都使用了 epoll，相比传统的 select 机制，epoll 没有最大连接句柄 1024 的限制。所以 Selector 在理论上可以轮询成千上万的客户端。

**下面我用一个生活化的场景来举例，**看完你就更清楚 Channel 和 Selector 在非阻塞 I/O

中承担什么角色，发挥什么作用了。

我们可以把监听多个 I/O 连接请求比作一个火车站的进站口。以前检票只能让搭乘就近一趟发车的旅客提前进站，而且只有一个检票员，这时如果有其他车次的旅客要进站，就只能在站口排队。这就相当于最早没有实现线程池的 I/O 操作。

后来火车站升级了，多了几个检票入口，允许不同车次的旅客从各自对应的检票入口进站。这就相当于用多线程创建了多个监听线程，同时监听各个客户端的 I/O 请求。

最后火车站进行了升级改造，可以容纳更多旅客了，每个车次载客更多了，而且车次也安排合理，乘客不再扎堆排队，可以从一个大的统一的检票口进站了，这一个检票口可以同时检票多个车次。这个大的检票口就相当于 Selector，车次就相当于 Channel，旅客就相当于 I/O 流。

## 总结

Java 的传统 I/O 开始是基于 InputStream 和 OutputStream 两个操作流实现的，这种流操作是以字节为单位，如果在高并发、大数据场景中，很容易导致阻塞，因此这种操作的性能是非常差的。还有，输出数据从用户空间复制到内核空间，再复制到输出设备，这样的操作会增加系统的性能开销。

传统 I/O 后来使用了 Buffer 优化了“阻塞”这个性能问题，以缓冲块作为最小单位，但相比整体性能来说依然不尽人意。

于是 NIO 发布，它是基于缓冲块为单位的流操作，在 Buffer 的基础上，新增了两个组件“管道和多路复用器”，实现了非阻塞 I/O，NIO 适用于发生大量 I/O 连接请求的场景，这三个组件共同提升了 I/O 的整体性能。

你可以在Github上通过几个简单的例子来实践下传统 IO、NIO。

## 思考题

在 JDK1.7 版本中，Java 发布了 NIO 的升级包 NIO2，也就是 AIO。AIO 实现了真正意义上的异步 I/O，它是直接将 I/O 操作交给操作系统进行异步处理。这也是对 I/O 操作的一种优化，那为什么现在很多容器的通信框架都还是使用 NIO 呢？

上一页 下一页