第 2 章 相关技术和开发环境

2.1 网络 I/O 模型发展演进

在互联网飞速发展的当下，CPU 的性能越来越高，使用传统的 Socket 网络编程（套接字）技术进行网络开发的时候，网络 IO 成为了一个限制 CPU 的因素，对网络通信的性能会产生重要的影响[2]。下面介绍的是网络 IO 模型发展演进的过程。

2.1.1 传统的 BIO 模型

传统的 Socket 编程使用的就是经典的 BIO 模型，这种 BIO 模型是同步的阻塞 IO。BIO 的全称是 Basic Input OutPut，基础输入输出，这是经典的 Client/Server 模型[3]。服务器先阻塞监听某个端口，然后客户端对服务器的 IP 地址和服务器监听的端口号发起一个请求连接，服务器如果监听到这个请求连接之后，就会开始进入握手建立连接的阶段。经过一系列的连接措施成功后，客户端和服务器之间就可以通过这个已经建立好的连接进行数据传输，这个过程是同步阻塞式的[4] 。下图 2-1 是服务器使用传统 BIO 模型的情况。

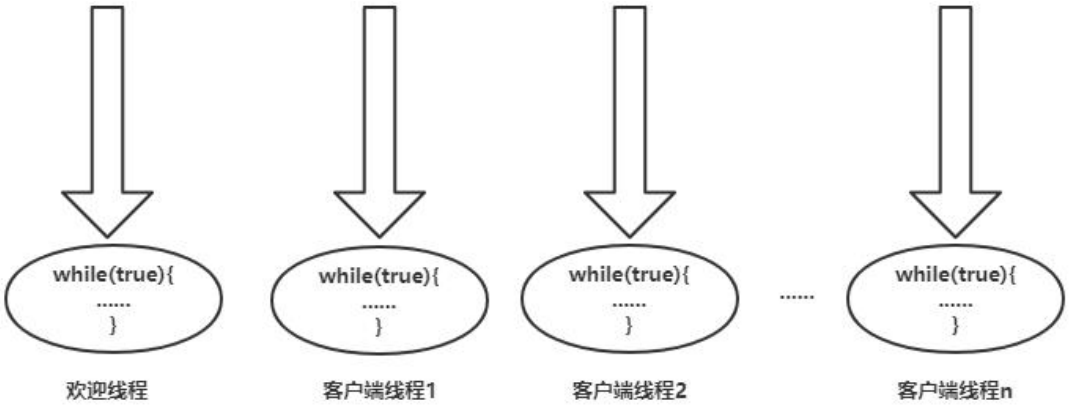

图 2-1 服务器 BIO 模型

每个客户端加入服务器的连接，服务器都需要一个 IO 线程阻塞来处理相关的请求[5]。该模型在高并发场景的大量请求下，会出现线程过多，都在等待数据，这个过程线程就会被挂起，然后只能干干地等待数据的处理，大量的线程被挂起，整个 CPU 的利用率非常低，系统的吞吐量变得很差。

2.1.2 伪异步 IO 模型

伪异步 IO 模型通过在 BIO 模型的基础上加入线程池这种方式，对 BIO 模型进行优化，来处理解决高并发情况下线程产生太多而导致资源占用过多的问题

[6]。 这个伪异步的 IO 模型通过配置好一个合理的线程池最大线程数，一个合理大小的线程阻塞队列，以及使用灵活可用的线程调用策略，确保服务器可以有大于客户端请求数量的线程数来处理请求。这样就减少了线程的浪费，节约了系统资源，这样服务端的性能就会有所提升。但是伪异步 IO 并没有改变 IO 在请求的时候线程会阻塞这一现况，该模型仅仅在线程的生成、关闭以及调度的方面做了优化，所以依然没有改变在网络 IO 同步的时候线程阻塞的问题。而且该模型在客户端和服务器之间连接请求的时候，如果响应时间太长或者稍微长了一点，例如网络 IO 堵塞、网络发生抖动现象、或者网络故障的时候，线程阻塞的时间会变长，就会使级联的线程阻塞的情况出现[7]。

举个例子说明这一情况：当某个客户端请求在服务端的某个节点发生了网络波动或者故障，需要花费十几秒，因为 IO 的过程依然是阻塞的，则该节点的线程也是会被阻塞而花费十几秒的闲置时间，而当其他线程访问已经发生故障的节点时，或者因为该节点的延迟而被延时往后移而等待排队，线程的任务就会被堆积起来，线程的阻塞数量就会逐渐增加，新接入的连接只能在阻塞队列种等待[8]，最终导致阻塞队列到达最大连接值，此时线程池会根据线程池配置的拒绝策略进行拒绝访问，如果有新的客户端请求建立连接，服务器只能拒绝响应，客户端视角则变成服务器不可以正常使用。下图 2-2 是伪异步 IO 模型的情况:

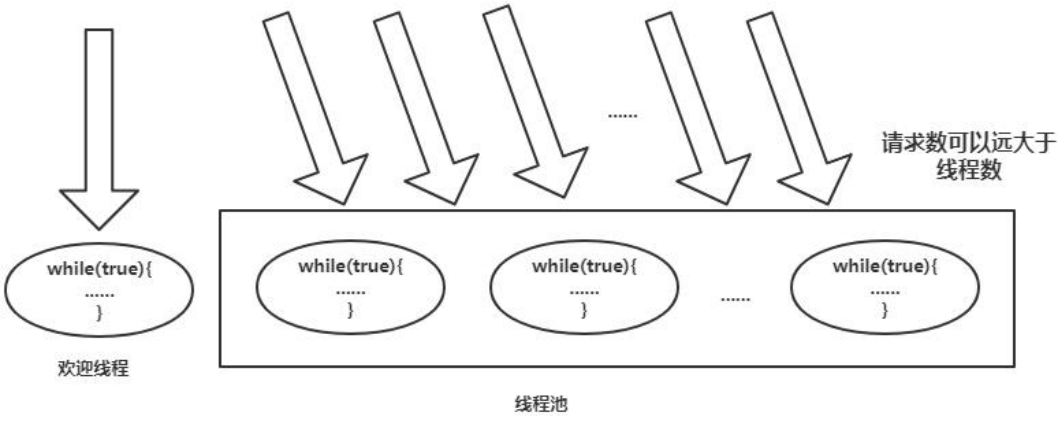

图 2-2 伪异步 IO 模型

2.1.3 NIO 模型

之前传统的 BIO 或者伪异步模型，都会在高并发的情况下出现问题，所以 JDK1.4 之后，提供了一系列的新 API，包名为 java.nio。NIO 的全称其实是 Non Blocking IO 即不阻塞的 IO，也有人解读名称为 New IO ，意为一种新型的 IO 方式。NIO 把旧的、原有的 IO 方式从面向流并且是阻塞的，更换成了面向块（缓冲区）的并且是非阻塞的 IO 方式。

NIO 这种 IO 方式使用了通道和缓冲区，在 Java 中通过 Native 本地方法库可

以调用相关的 API 直接在 JVM 的堆外分配内存，将数据直接存储在内存当中。如果操作系统允许且硬件有 DMA 支持的话，甚至可以根据该特性实现零拷贝，即该技术在系统允许的情况下能进一步的优化成零拷贝 IO。而 Java 的 NIO 带来了可以用于通道的缓冲区工具类 ByteBuffer，其中有三个主要的实现，最常用的是 DirectByteBuffer 类，这个类将使用 malloc()在堆外的空间申请内存，因为这一特点，导致其实际不是由 JVM 管理的，不受垃圾回收的影响，必须自己管理该内存。还有两个实现、其一是 HeapByteBuffer，在 JVM 的堆内空间生成，其二是 MappedByteBuffer 也是使用的堆外内存，其使用了操作系统的 mmap（）的 API 来做完主要实现途径。

NIO 的核心由三部分组成：Selector 器（多路复用）、Buffer 缓冲区（常用 ByteBuffer）以及 Channel 双工通道。

(1) ByteBuffer 缓冲区

ByteBuffer 是 NIO 中最常用的缓冲区，当然也还有其他种类的 Buffer，例如 CharBuffer、IntBuffer 等，但是 ByteBuffer 使用起来更加地方便和合乎逻辑，用来存储网络交互中的数据。在网络 IO 中，将接收到的数据通过 Channel 接收然后写入到 Buffer 当中，这个时候 NIO 线程可以从 Buffer 中读取数据或者处理数据。下图 2-3 是 Buffer 缓冲区跟 Channel 的数据传输：

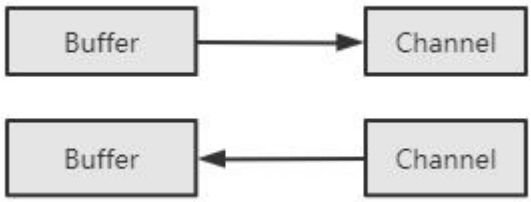

图 2-3 Buffer 缓冲区跟 Channel 的数据传输

旧 IO 方式由于没有 Buffer 机制，是面向流的 IO，如果需要对数据进行修改的话，例如随机读取，则需要拷贝一份到用户态的内存里再进行处理，造成了一定的浪费。而且 Buffer 缓冲区，可以让线程快速判断是否有新的数据到达，如果没有数据到达的话，将不会为该 Channel 分配线程资源。所以缓冲区 Buffer 是 NIO 的一个重要特征，可以用来区分开 NIO 和旧 IO。

(2) Channel 通道

通道 Channel 是全双工的，所以亦是此 NIO 模型的重要特征。需要注意区分开 Channel 跟流，流是单工的，只能一个方向输入或者输出，读取的时候也不能

随机处理数据，如果需要处理的话还需把数据拷贝到 Java 的内存空间中处理，但 Channel 不一样，Channel 的读写操作可以同时进行，并且可以通过移动读写指针来完成一些有限的随机读取操作而无需另外拷贝数据，这个特性使得 Channel 可以在一个线程写入数据的时候，另一个线程从中读取数据，全双工的特性毫无疑问地可以使得系统的吞吐量或者说处理速度增加不少。下图 2-4 是客户端向服务器发送请求的时候数据的流动过程。

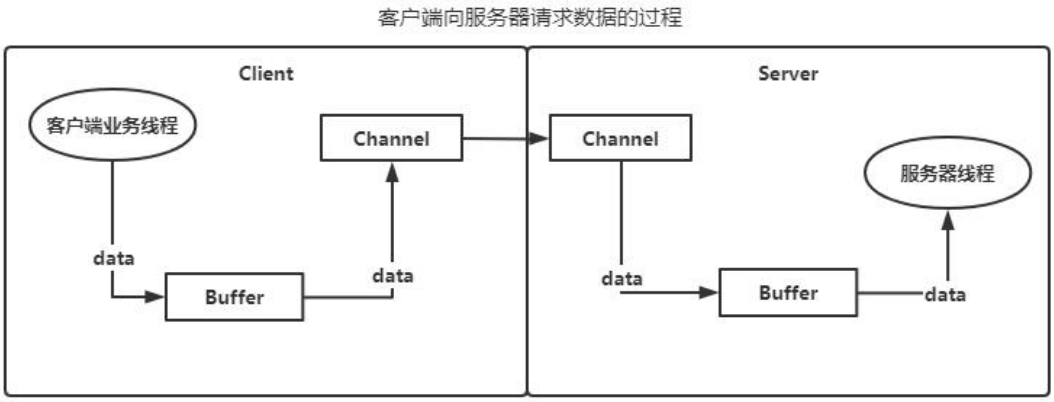

图 2-4 客户端向服务器发送 data

（3）Selector 多路复用选择器

Selector 实现多路复用的方式就是轮询，不停地对 Channel 中是否有新的数据到达进行轮询监听。下图 2-5 为客户端连接时多路复用选择器的图示情况。

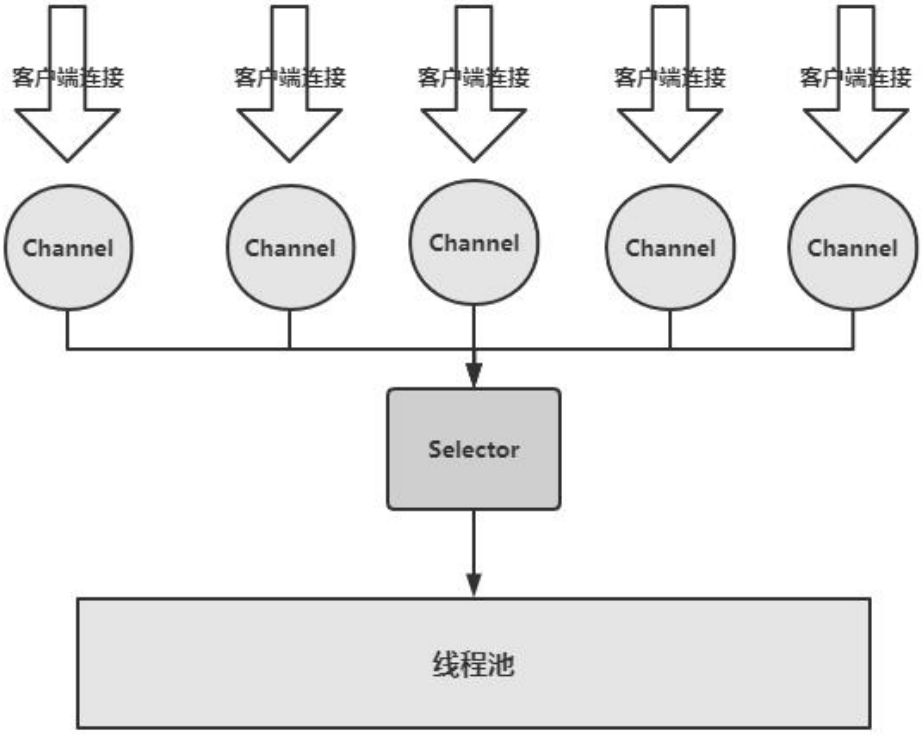

图 2-5 Selector 多路复用选择器

一个 Selector 在同一时间内监听很多个通道 Channel，检测通道上面是否有数据到达，即检测是否有就绪的 IO 时间，然后使用其内置的 SelectionKey 来获取已经就绪的 Channel 然后操作。多路复用在 JDK 中是由 epoll()来实现轮询的，这样做也可以避免了传统 socket 中存在的最大连接数的限制。

2.2 Netty 技术

Netty 技术是在 JDK 的 NIO 基础上进行封装并改进的高性能框架，其使用方式比 JDK 中原生的 NIO 方便很多。其主要的技术框架有 3 部分值得注意：Netty 缓冲区部分、Netty 的线程模型、Netty 中的责任链模式 ChannelPipeline。

2.2.1 Netty 的缓冲区与通道

Netty 中对 JDK 中内置的 ByteBuffer 进行封装改进，提供了一个缓冲区类 ByteBuf，它的功能基本上与 ByteBuffer 保持一致，是由一个 Byte 数组组成的缓冲区。ByteBuf 的实现是利用双指针算法对缓冲区操作的，分别是读指针（ReadIndex）和写指针（WriterIndex）[9]。下图 2-6 是 ByteBuf 双指针操作的示意图。

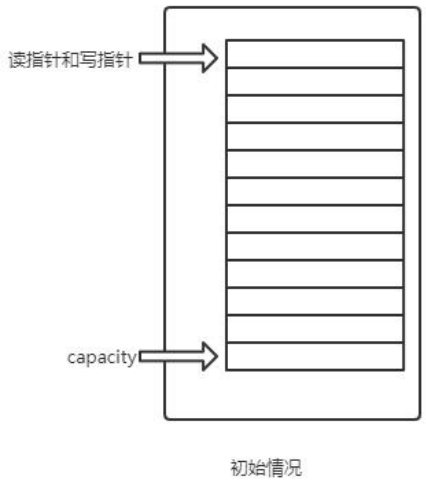

图 2-6 ByteBuf 双指针初始化情况

当初始化后，读指针跟写指针都是在下标 0 的位置，此时只能写入，因为读指针最多只能读到写指针位置。此时写入数据和读取数据的话，情况如下图 2-7 所示。

写入数据的时候，写指针往后移动，但是最大不能超过 capacity 的指针，然后此时读指针跟写指针不是在同一个位置，所以读指针可以开始读取了，读取的时候，读指针也是往后移动，但是最大不能超过写指针，这样就维护了一个可以全双工工作的一个缓冲区。

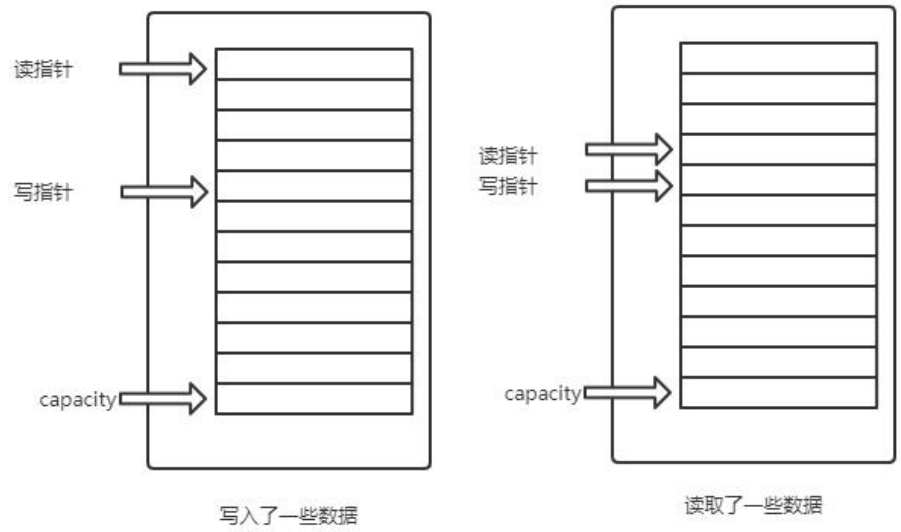

图 2-7 ByteBuf 的读写操作

读指针在读取了一些数据之后，下标 0 到读指针之间的数据其实已经没有用了，是可以丢弃的，所以内部会自动调用整理方法 discardReadBytes 对这部分的数据进行整理与清除弃用内存，ByteBuf 的数据情况变成下图 2-8 所示:

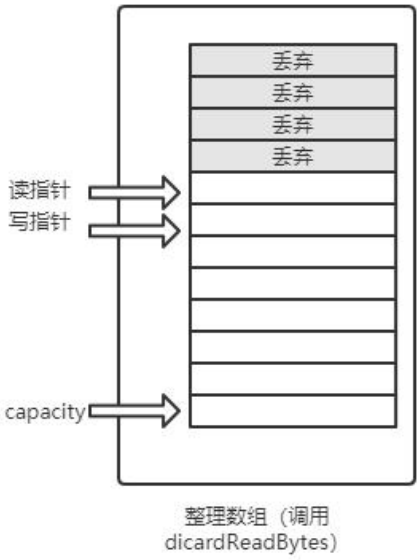

图 2-8 整理数组

如果不再需要读取数据的话，执行 clear 方法，会使读写指针都移动下标 0 的位置，状态如初始化时一样，见上图 2.6 初始化时的图。当 ByteBuf 判断到它的写指针即将写入到 capacity 的时候，会根据实际的内存使用情况决定重新分配空间，把他原来的所有数据拷贝到一个新的且更大的内存空间中，以此来实现动态扩容，这个过程受硬件和系统的限制，不能无限制的扩容，所以存在内存最大使用量限制。

Netty 框架中的通道在使用方法上跟 NIO 技术的保持了一致，并进行了一系列的优化封装，同时添加了一些 Netty 特有的一些特性，例如增加了 NioEventLoop 线程，还有一个责任链模式工具 ChannelPipeline 类。

2.2.2 Netty 的线程模型

(1) 线程模型

Netty 框架中使用的线程模型是灵活的，既可以单 Reactor 搭配单线程或者线程组，也可以主从 Reactor 搭配线程组，可以按照需要来配置合理的参数[10]。

Netty 服务器启动的时候，需要创建两个 NioEventLoopGroup 线程组，一个用来接收请求（如传统 BIO 的欢迎线程），另一个用来处理网络 IO 的事件和定时的任务等。使用线程池可以提高服务器的高并发能力[11]。

(2) NioEventLoop 线程

NioEventLoopGroup 线程组中存在 NioEventLoop 线程，这个线程不仅会被用于处理 IO 事件，还会被用于处理系统任务等[12]。

这个线程会把用户的操作封装成一个 Task，submit 到线程池中等待调度和处理，同时也开发者也可以使用 schedule 方法来实现定时任务。

在这个线程模型中，NioEventLoop 使用串行化的方式来对消息处理，互相不会干扰而且避免了线程的频繁切换。一个 NioEventLoop 线程只会有单个客户端注册使用，所以能够确保线程组中的线程可以并发运行，相对独立，提高了服务器的多处理器并行的能力。

2.2.3 Netty 的 ChannelPipeline

Netty 的 Channel 处理使用的是责任链设计模式，将 Channel 的数据抽象为一个责任链 ChannelPipeline，一个消息的处理会由责任链进行传递并且依次处理，可以避免程序的分支结构太多且重复的情况出现。链上的每个节点都是一个处理器，处理器可以处理读取输入方向（InBound）的数据和输出方向（OutBound）的数据。开发者可以通过 ChannelHandler 的功能配置和灵活的组合，从而实现数据和消息的清洗，拦截，处理等业务。

下图 2-8 便是这个责任链的示意图:

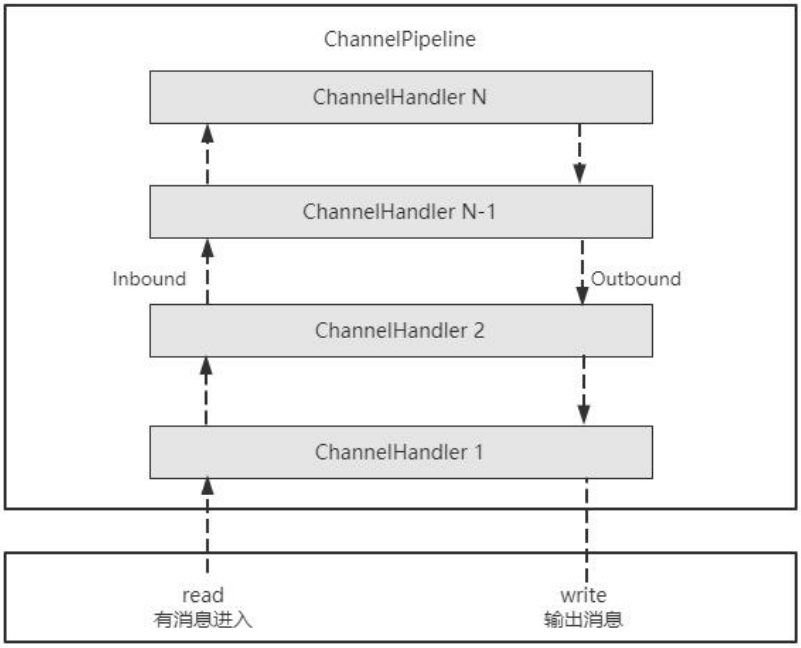

图 2-8 ChannelPipeline 的处理过程

这个 ChannelPipeline 是一个双向链表，当有网络 IO 数据就绪的时候，先从头部向尾部按顺序的执行读取处理器（ChannelInboundHandler），然后经过处理后的数据又从尾部向头部反方向进行执行写出处理器操作（ChannelOutboundHandler），其中消息的粘包半包解析器，消息解码编码器都需要在里面实现。

2.3 MySQL 数据库

MySQL 是一个开放源码的关系型数据库。在开发环境中被广泛使用，支持几乎所有的操作系统，并且对大多数语言都有支持，简单易用。其开源并且支持多线程多用户等特点，深受开发者们的喜欢。MySQL 相比其他商业级的数据库而言，速度和效率可能稍有逊色，但是 MySQL 体积小，也有很不错的性能表现，是作为一个低成本系统的不二之选。

2.4 Mybatis

Mybatis 是一个基于 Java 的持久层框架，其本是 Apache 公司的开源项目 Ibatis。MyBatis 这个持久层框架是非常优秀的框架，支持定制化 SQL 查询、高级映射等操作，其提供的 SqlSessionFactory 类亦是很简单易用。由于其半自动化的 SQL 语句，以及提供非常好用的动态 SQL 组合，使得在处理 SQL 数据库持久化的时候非常灵活，而且可以对 SQL 的性能进行优化，可以最大程度地提高系统的高并发能力。