華南農業大學

本科毕业论文

消息应答服务器框架设计与开发

何梓

201030560306

指导教师 吴春胤 副教授

学院名称	信息学院	专业名称	信息管理与信息系统
论文提交日期	2014 年 4 月 12 日	论文答辩日期	2014 年 5 月 10 日

摘　　要

21 世纪是信息时代，企业向信息化发展，消息的收集、处理、订阅也非常重要，各种智能消息处理和应答解脱了部分人力。在生产、管理和服务方面，大企业对自动应答、智能应答等服务的需求越来越多，如移动的语音客服，微信公众平台订阅号和服务号，淘宝的智能小宝等。

本文探索使用当今最热门的面向对象语言，编写基于 socket 的消息处理和应答服务器框架，设计并实现基于缓存和文本的数据存储，低应用的部署环境要求，简化的消息的定义和处理。随着企业向信息化智能化发展，消息处理和应答的应用越来越普及，探索如何实现更方便简单的消息服务器，将是一件非常有意义的事情。

在网络连接上，使用 JDK1.4 提供了无阻塞 I/O（NIO），NIO 的性特性多路复用在单线程中管理多个 I/O 通道，当某个 socket 通道有数据是，该线程从阻塞唤醒，并返回该封装了该通道在 Selector 上的注册信息的 SelectionKey 对象，从 SelectionKey 上调用通道和绑定的处理器的各种操作。在 dao 层，提供数据存储配置，可以选择内存、文本、数据库存储数据，降低了服务器部署环境的要求。

框架源码编译压缩成 jar 包，开发者引用该 jar 包，三步开发属于自己的消息应答服务器：一是编写继承自 AppHandler 的处理器，二是编写继承自 Application 的消息模型，指定泛型为刚刚编写的消息处理器，三是实例化 NIOServer，并指定泛型为刚刚编写的消息模型。编写完三个类后，启动服务器并监听，一个消息应答服务器就完成了。

关键词：　　消息　　处理　　响应　　Socket　　新 IO　　多线程　　高并发

The Design and Development of Simple Quora Server Framework

He Zi

(College of Informatics, South China Agricultural University, Guangzhou 510642, China)

Abstract: 21st century is the Information Age, with the development of information technology companies, enterprise informatization and message of collection, processing, subscription become very important, several kinds of intelligent message processing and response liberates part of human power. In terms of production, management and service, there are more and more large enterprises demand for auto response, intelligent response and other services, such as mobile voice customer service, WeChat public subscription number and service number, Taobao's Smart XiaoBao and etc.

This theses uses the most popular object-oriented language to program based on socket message processing and response server framework, design and achieve based on the cached and text data storage, it is used for reducing application deployment environment requirements, simplying message definition and processing. With the enterprise's informatization intelligent development, applications of message processing and response are more and more popular, it will be a very meaningful thing to explore how to implement the more convinient and simple message server.

On the Internet connection, it offers non-blocking I/O(NIO) by using JDK1.4. The feature of NIO is that, multichannel multiplexing manages multiple I/O channel in the single thread. When a socket channel has data, the thread awakes from blocking, and return SelectionKey object which encapsulates the register information of Selector in the channel, then invokes serveral operations of channel and binding processor from SelectionKey object. On dao level, it offers data storage deployment, choose memory, text, database storage data, in order to reduce requirement of server deployment environment.

The framework source code is packaged to jar, developers can reference this jar package, develop message response server belong to us with three steps: The first, coding processor extended AppHandler; The second, coding message model extended Application, specifing generic for message model. After finishing coding this three class, launch server and listen, a message response server is done.

Key words: Response Intellectualize Socket NIO MultiThread High concurrency

目　　录

华南农业大学本科生毕业论文成绩评

1 引言

1.1 研究目的和意义

随着企业信息化步伐的加快，企业内部各种形式的消息处理与应答应用也越来越多，对企业运作和对外提供服务也发挥着越来越重要的作用。如腾讯的微信，韩国 NHN 株式会社的 Line,淘宝的智能机器人等等。编程语言经历了机器语言、汇编语言，到高级语言的发展，到如今已经有 600 多种，消息的处理和应答不乏各种语言的各种实现方式，使用消息处理和应答的各大企业已有强大的高性能的实现，但是大多建立在 HTTP 协议、pct 协议，摒弃 tomcat 等 web 应用服务器框架或 J2ee 等传统应用开发的技术架构，没有单独实现消息的处理和应答的应用框架了。

HTTP 协议是万维网协会（World Wide Web Consortium）和 Internet 工作小组 IETF（Internet Engineering Task Force）共同制定的从 WWW 服务器传输超文本到本地浏览器的传送协议。如图 1 所示，HTTP 请求建立连接需要三次握手，建立连接后才开始传输 TCP 数据包。如图 2 所示，断开连接需要四次握手。如果应用一次连接只要传输一次数据，遵循 http 协议的数据传输就显得效率太低了，而根据应用需求，自定义协议就显得更有意义了。

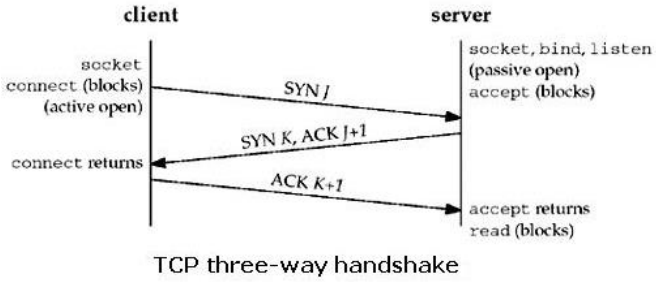

图 1 http 协议建立连接三次握手

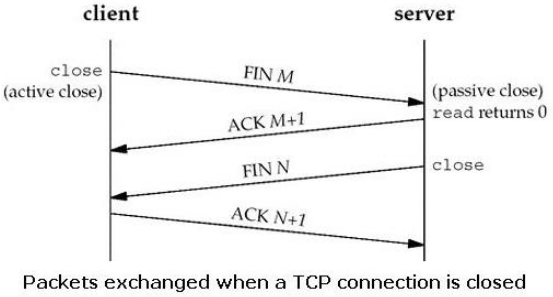

图 2 http 协议关闭连接四次握手

Socket 是支持 TCP/IP 协议的网络通信的基本操作单元，任何一种支持网络编程的语言都支持该协议，而开发者可以在 socket 数据包上再封装传输协议，避免不必要的握手。编程语言社区排行榜 TIOBE 最新数据显示，Java 依然是世界第二大编程语言。所以，借助当前全球最热门的面向对象编程语言，做消息的处理和应答服务的探索，形成简单便捷、功能专一和完善的消息处理和应答框架是具有深刻意义的。

Tomcat 是一个 HTTP 服务器，更细致一点是一个 Servlet 容器，按照 Sun Microsystems 提供的技术规范，实现了对 Servlet 和 JavaServer Page（JSP）的支持，并提供了作为 Web 服务器的一些特有功能，如 Tomcat 的管理和控制平台、安全管理和 Tomcat 阀等。也由于支持整套 http 协议和各种规范，使得在不需要这么多功能支持的应用中稍显服务器复杂而繁重。

因而，有必要探索基于底层 socket 网络通信，定制简单的通信协议，抽象出消息处理和应答功能的完整应用框架，避免各种基于 socket 之上通用协议过分复杂的束缚。

1.2 研究现状

在移动端应用领域，最具代表性的消息处理和应答应用是腾讯的微信和韩国的 Line（如图 3）。据微信官方统计，2013 年 10 月 24 日，微信的用户数量已经超过了 6 亿，每日活跃用户 1 亿。2013 年 11 月 25 日，Line 官方也宣布全球使用人数突破三亿。这两款软件都是提供即时通讯服务的应用软件，除了社交功能外，还有消息的订阅，输入“命令”得到开发者设计的结果。

图 3 公众账号消息订阅功能

在桌面应用领域，淘宝的智能机器人（如图 4）可以分析输入的“问句”或关键词，分析处理并返回“答案”，在天猫的“双十一”发挥着巨大的作用。而且，相比普通客服，尤其在双 11 大促的时候，机器人响应速度更快，接待能力更强，回复更有统一的客服话术。

图 4 淘宝智能机器人

一个 Linux 程序员在 2014 年 3 月 25 号发布了他的“仿 shell”婚礼网站（图 5），为自己的 4 月 7 日的婚礼提供了接受邀请、查看婚礼信息、查询婚礼当天现场路线等功能，

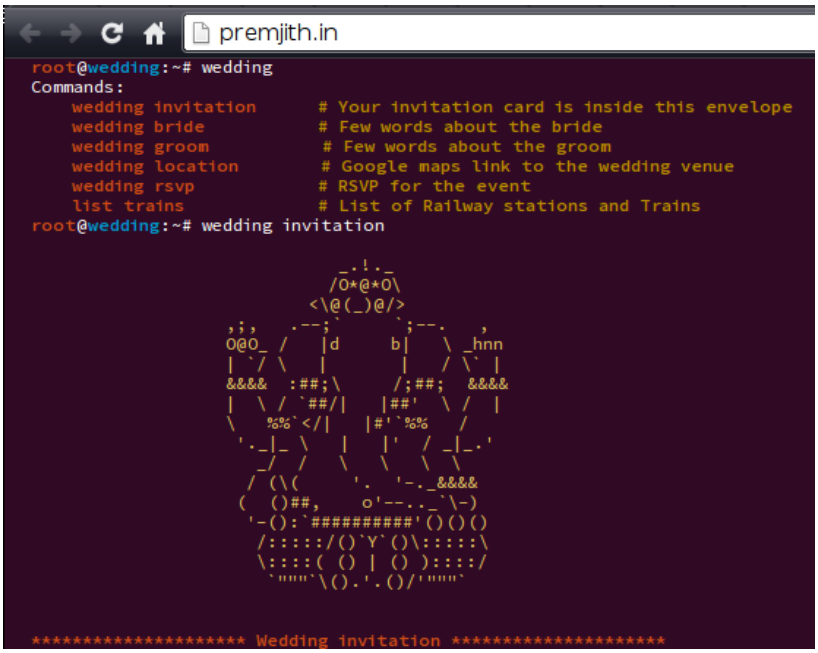

图 5 premjith.in 网站

不管消息的应答在客户端以什么形式呈现给用户，用户看到的都是消息在服务器处理后的响应，而网络通信的底层就是 socket，即应用程序与 TCP/IP 交互的套接字(Socket)的接口。基于 Socket 编写的消息处理和应答框架避开了 Http 或 TCP 协议的各种约束，以更高效的连接和数据传输实现消息的处理和应答。

1.3 愿景

基于底层 socket 网络通信，定制简单的通信协议，抽象出消息处理和应答功能的完整应用框架，避免各种基于 socket 之上通用协议过分复杂的束缚。在消息模型上，增加消息序列功能，以便消息处理顺序的自动化。在数据存储上，增加数据存储方式的配置，提供数据库 mysql、内存、log 和本地文本文件存储方式的选择，降低应用部署环境的要求。开发者引用该框架 jar 包，编写自己的消息模型，实现自己的消息处理，即可实例化并启动一个消息处理和应答服务器。

2 相关技术介绍

2.1 Java 面向对象编程语言

如图 6，根据 2014 年 3 月世界编程语言排行榜的数据统计，Java 面向对象编程语言是世界第二大编程语言。

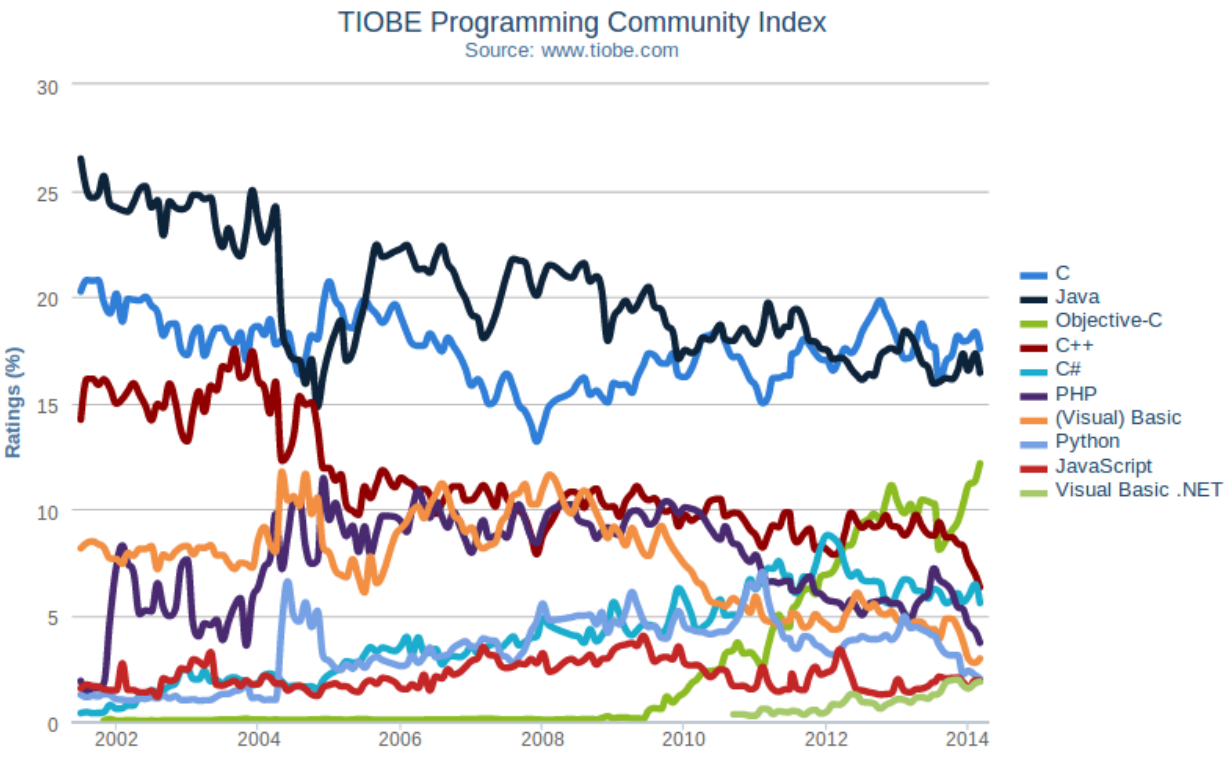

图 6 TIOBE 世界编程语言排行榜

Java 是一种可以撰写跨平台应用软件的面向对象的程序设计语言，是由 Sun Microsystems 公司于 1995 年 5 月推出的 Java 程序设计语言和 Java 平台（即 JavaSE, JavaEE, JavaME）的总称。Java 技术具有卓越的通用性、高效性、平台移植性和安全性，广泛应用于个人 PC、数据中心、游戏控制台、科学超级计算机、移动电话和互联网，同时拥有全球最大的开发者专业社群。在全球云计算和移动互联网的产业环境下，Java 更具备了显着优势和广阔前景。

Java 编程语言的风格十分接近 C++语言。继承了 C++语言面向对象技术的核心，Java 舍弃了 C++语言中容易引起错误的指针，改以引用取代，同时移除原 C++与原来运算符重载，也移除多重继承特性，改用接口取代，增加垃圾回收器功能。在 Java SE 1.5 版本中引入了泛型编程、类型安全的枚举、不定长参数和自动装/拆箱特性。所以，Java 编程语言是个简单、面向对象、分布式、解释性、健壮、安全与系统无关、可移植、高性能、多线程和动态的语言。

Java 不同于一般的编译语言和直译语言。它首先将源代码编译成字节码，然后依赖各种不同平台上的虚拟机来解释执行字节码，从而实现了“一次编写，到处运行”的跨平台特性。

2.2 Socket 编程

多个 TCP 连接或多个应用程序进程可能需要通过同一个 TCP 协议端口传输数据。为了区别不同的应用程序的进程和连接，许多计算机操作系统为应用程序与 TCP/IP 协议交互提供了称为套接字(Socket)的接口。套接字是支持 TCP/IP 的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单地说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。

JDK1.4 提供了无阻塞 I/O（NIO），在此之前，Java 网络编程中 TCP 和 UDP 编程都由原生 Socket 实现（如图 7），服务器端先初始化 Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用 accept 阻塞，等待客户端连接。在这时如果有个客户端初始化一个 Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，完成一次交互。

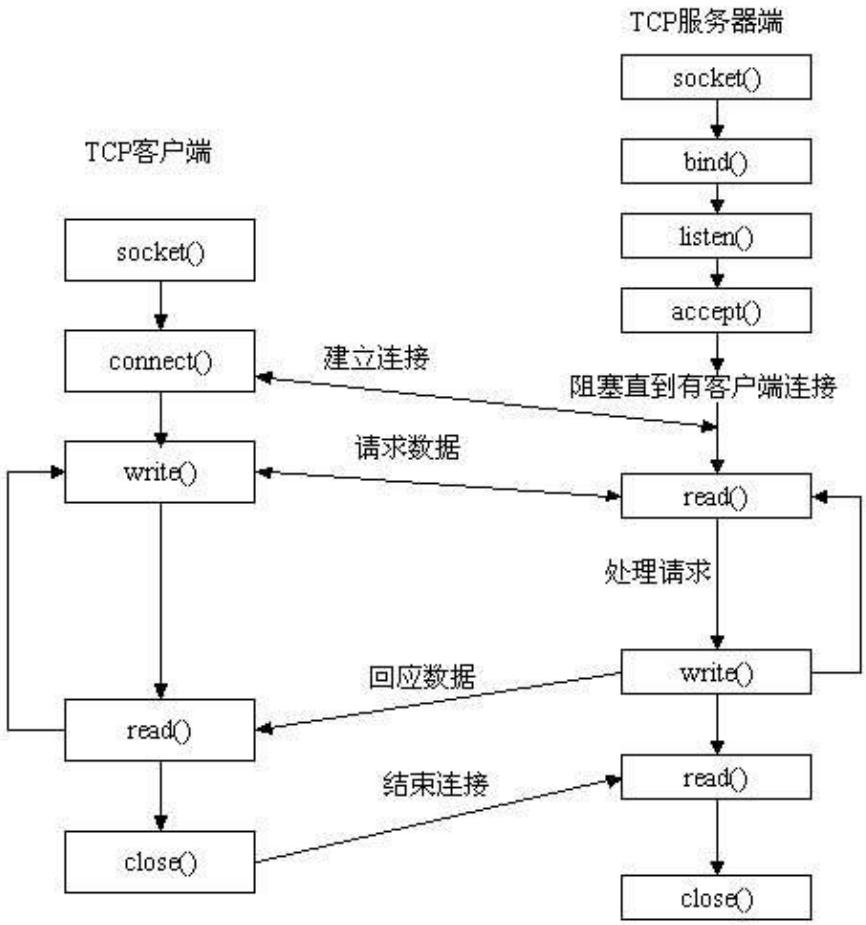

图 7 基于 Socket 的 tcp 编程

2.3 基于 NIO 的 TCP 编程

2.3.1 NIO 简介

I/O 或者输入/输出指的是计算机与外部世界或者一个程序与计算机的其余部分之间的接口。它对于任何计算机系统都非常关键，因而所有 I/O 的主体实际上是内置在操作系统中的。单独的程序一般是让系统为它们完成大部分的工作。在 Java 编程中，直到最近一直使用的流的方式完成 I/O，所有 I/O 都被视为单个的字节的移动，通过一个称为 Stream 的对象一次移动一个字节。流 I/O 用于与外部世界接触。它也在内部使用，用于将对象转换为字节，然后再转换回对象。

NIO 与原来的 I/O 有同样的作用和目的，但是它使用不同的方式块 I/O。块 I/O 的效率可以比流 I/O 高许多。NIO 创建目的是为了让 Java 程序员可以实现高速 I/O 而无需编写自定义的本机代码。NIO 将最耗时的 I/O 操作(即填充和提取缓冲区)转移回操作系统，因而可以极大地提高速度。

原来的 I/O 库(在 java.io.*中)与 NIO 最重要的区别是数据打包和传输的方式。正如前面提到的，原来的 I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。面向流的 I/O 系统一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。为流式数据创建过滤器非常容易，链接几个过滤器，以便每个过滤器只负责单个复杂处理机制的一部分，这样也是相对简单的。不利的一面是，面向流的 I/O 通常相当慢。而一个面向块的 I/O 系统以块的形式处理数据。每一个操作都在一步中产生或者消费一个数据块。按块处理数据比按(流式的)字节处理数据要快得多。但是面向块的 I/O 缺少一些面向流的 I/O 所具有的优雅性和简单性。在 JDK 1.4 中原来的 I/O 包和 NIO 已经很好地集成了 I/O, java.io.* 已经以 NIO 为基础重新实现了，所以现在它可以利用 NIO 的一些特性。例如，java.io.*包中的一些类包含以块的形式读写数据的方法，这使得即使在更面向流的系统中，处理速度也会更快。也可以用 NIO 库实现标准 I/O 功能。例如，可以容易地使用块 I/O 一次一个字节地移动数据。

NIO 还提供了原 I/O 包中所没有的许多功能。如非阻塞 I/O 作为 REACTOR 模式的实现，实际上提供了一种事件发生、自我激活、主动通知机制。NIO 的选择器采用了多路复（Multiplexing）技术，可在一个选择器上处理多个套接字，通过获取读写通道来进行 IO 操作。NIO 有一个主要的类 Selector，这个类似一个观察者，只要我们把需要探知的 SocketChannel 告诉 Selector，我们接着做别的事情，当有事件发生时，他会通知我们，传回一组 SelectionKey，我们读取这些 Key，就会获得我们刚刚注册过的 SocketChannel，然后，我们从这 channel 中读取数据，接着我们可以处理这些数据。Selector 内部原理实际是在做一个对所注册的 channel 的轮询访问，不断的轮询(目前就这一个算法)，一旦轮询到一个 channel 有所注册的事情发生，比如数据来了，他就会站起来报告，交出一把钥匙，让我们通过这把钥匙来读取这个 channel 的内容

Java NIO 通信机制的构建采用了基于 Buffers（缓冲区）、 Channel（通道）、 Selectors（选择器）的新模式。

Buffers（缓冲区）：Buffer 是抽象类，它及其派生出的“子类”，用以处理各种类型数据的读写以及相关的运算。每一缓冲器内部包含一个字节数组作为数据存储，实现数据的管理和运算并控制操作系统的读写过程。

Channels（通道）：Channel 是一个接口，功能类似于传统 I/O 的 Stream，但通道具有双向性，既可以读入，也可以写出。Selectors（选择器）：各类 Buffer 是数据的容器对象。各类 Channel 实现在各类 Buffer 与各类 I/O 服务间传输数据。而 Selector 负责监视已注册

的 Socket 通道，提供各类 Channel 的状态信息，并串行化服务器需要应答的请求，控制这各类 Channel 有效地工作。

2.3.2 NIO 的一些新特性

(1) 文件锁

NIO 出现以前，如果要在 Java 应用中设置或检查文件锁除了调用本地函数（native method）别无它法。文件锁因与操作系统（甚至是文件系统）绑定而臭名昭著，任何相关代码的移植都充斥着危险。NIO 文件锁基于 FileChannel 类构建。现在，只要在操作系统层支持文件锁的任何平台都可以很轻易地创建、测试和管理文件锁。通过 NIO，新编写的 Java 读取软件（reader application）能够采用相同的锁定规范与先前存在的非 Java 软件无缝集成。

文件锁通常在文件和进程级别操作，不适合用作 JVM 内线程之间的协调。操作系统一般不会区分同一个进程中不同线程的持锁权。这意味着同一个 JVM 中所有线程拥有同样的锁。文件锁主要是用来集成非 Java 应用或者不同的 JVM。

(2) 缓冲区

NIO 引入了缓冲区，它们是一组封装了固定大小原生类型数组及其相应状态信息的简单对象。缓冲区主要用来作为从通道发送或者接收数据的容器。通道是低级 I/O 服务的管道，面向字节，所以他们只能操作 ByteBuffer 对象。

(3) 直接缓冲区

封装在缓冲区中数据元素可以采用下列存储方式的一种：通过分配创建一个缓冲区对象的私有数组，或者包装用户提供的数组，或者以直接缓冲区的方式存储在 JVM 内存堆以外的本地内存空间中。当用户调用 ByteBuffer.allocateDirect()创建一个直接缓冲区时，会分配本地系统内存并且用一个缓冲区对象来包装它。直接缓冲区的主要用途是用做通道（channel）I/O。通道实现能够用直接缓冲区的本地内存空间来设置系统级 I/O 操作。这是一个强大的新功能，也是 NIO 效率的关键。虽然这些 I/O 函数是底层操作不能直接使用，但是可以利用通道的直接缓冲区功能提升效率。

(4) 分散读和聚集写

NIO 出现之前，我们是这样从文件读数据的：

```
byte [] byteArray = new byte [100];
int bytesRead = fileInputStream.read (byteArray);
```

这个操作从流向一个字节数组读入数据。然后下面是采用 ByteBuffer 和 FileChannel 的等价读操作。

NIO 通道支持分散/聚集，即向量 I/O。

```
ByteBuffer byteBuffer = ByteBuffer.allocate (100);
int bytesRead = fileChannel.read (byteBuffer);
ByteBuffer header = ByteBuffer.allocate (32);
ByteBuffer colorMap = ByteBuffer (256 * 3)
ByteBuffer imageBody = ByteBuffer (640 * 480);
fileChannel.read (header);
fileChannel.read (colorMap);
fileChannel.read (imageBody);
```

分散读上面的缓冲区也可以这样做：

```
ByteBuffer [] scatterBuffers = { header, colorMap, imageBody };
fileChannel.read (scatterBuffers);
```

即用缓冲区数组来代替传递单个缓冲区对象。

通道按顺序填充每个缓冲区直到所有缓冲区满或者没有数据可读为止。聚集写也是以类似的方式完成，数据从列表中的每个缓冲区中顺序取出来发送到通道就好像顺序写入一样。

(5) 直接通道传输

信道传输让你连接两个通道，数据可以直接从一个传到另一个通道。因为 transferTo() 和 transferFrom()方法属于 FileChannel 类，FileChannel 对象必须是一个通道传输的源或者目的（例如，你不能从一个 socket 传到另一个 socket）。但是传输的另一端必须是合适的 ReadableByteChannel 或者 WritableByteChannel。

(6) 非阻塞套接字

从 SelectableChannel 继承的 Channel 类可以用 configureBlocking 替换成为非阻塞模式。在 J2SE1.4 中只有套接字通道（SocketChannel， ServerSocketChannel 和 DatagramChannel）可以换为非阻塞模式，而 FileChannel 不能设为非阻塞模式。当通道设置为非阻塞时，read()或者 write()调用不管有没有传输数据总是立即返回。这样线程总是可以无停滞地检查数据是否已经准备好。

(7) 多路复用 I/O

NIO 用轮询来确定在非阻塞通道上输入已经就绪，如果在处理循环主要是做别的事情并周期性的检查输入时，轮询是一个适合的选择，但如果应用程序（如 web 服务器）的主要目的是响应许多不同连接上的输入，轮询就不合适了。在响应式的应用中，需要快速的轮转，然而快速轮询只会毫无意义地消耗大量的 CPU 指令周期并产生大量无效的 I/O 请求。I/O 请求会差生系统调用，系统调用造成上下文切换，而上下文切换是相当费时的操作。所以，NIO 使用单线程来管理多个 I/O 通道时，就是所谓的多路复用 I/O，只需要一个线程就能够简单的监控大量的套接字。当任何流上有数据时（读的部分）线程也能够选择阻塞唤醒的方式（我们又回到了阻塞），并且精确地接收就绪流上来的信息。

事件处理模型中，有一种叫 reactor（反应堆）设计模式，在大中型网络服务器中十分常见。NIO 为 reactor 模式提供了实现的基础机制，开发者在观察者中将事件和 handler 绑定，当 Selector 发现某个注册过的 channel 有数据时，会通过 SelectionKey 来告知开发者，开发者从 SelectionKey 中获取 handler，并调用效应处理函数。

2.3.3 较 Socket 编程优势

传统网络服务器的设计如图 8，每个连接耗用一个线程处理数据的到达，即使用了线程池可多个连接共享空闲线程，但是每个连接堵塞在 I/O 操作上，直到读写完成把该线程返回给线程池。

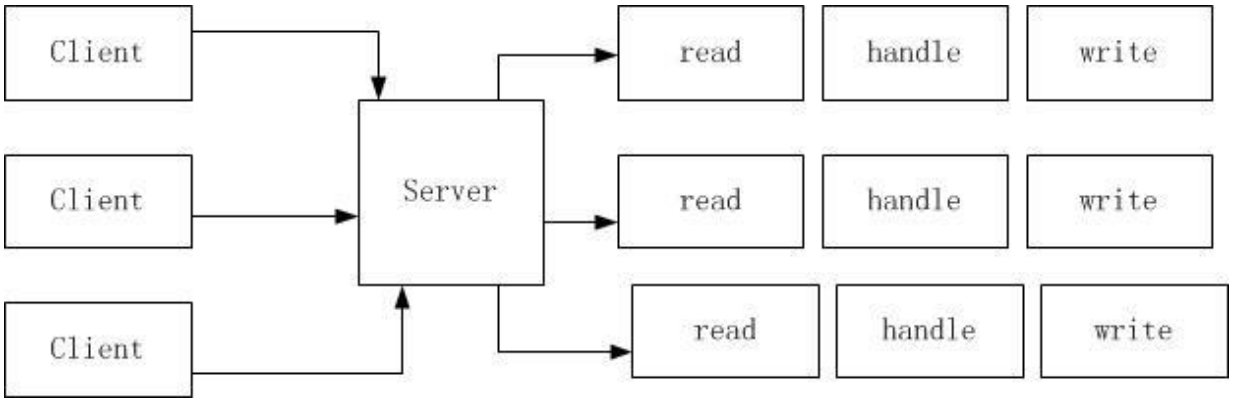

图 8 传统网络服务器设计

NIO 非堵塞应用适用在 I/O 读写等方面，系统运行的性能瓶颈通常在 I/O 读写，包括对端口和文件的操作上，过去，在打开一个 I /O 通道后，read()将一直等待在端口一边读取字节内容，假如没有内容进来，read()傻傻地等将影响了程序继续做其他事情，改进做法就是开设线程，让线程去等待，但是这样做也相当耗费资源。NIO 实现了流畅的 I/O

读写，不堵塞了。它的出现不只是一个技术性能的提高，它具有里程碑意义，从 JDK1.4 开始，Java 开始提高性能相关的功能，从而使得 Java 在底层或者并行分布式计算等操作上已经可以和 C 或 Perl 等语言并驾齐驱。

2.3.4 NIO 多线程高并发

NIO 的选择器采用了多路复用（Multiplexing）技术，但由于网络带宽等原因，在通道的读、写操作中是容易出现等待的，所以，在 IO 操作中引入多线程来提高与客户端的数据交换能力，对性能提高明显，而且可以提高客户端的感知服务质量。

如图 9 所示，开设两个线程专门负责读写。

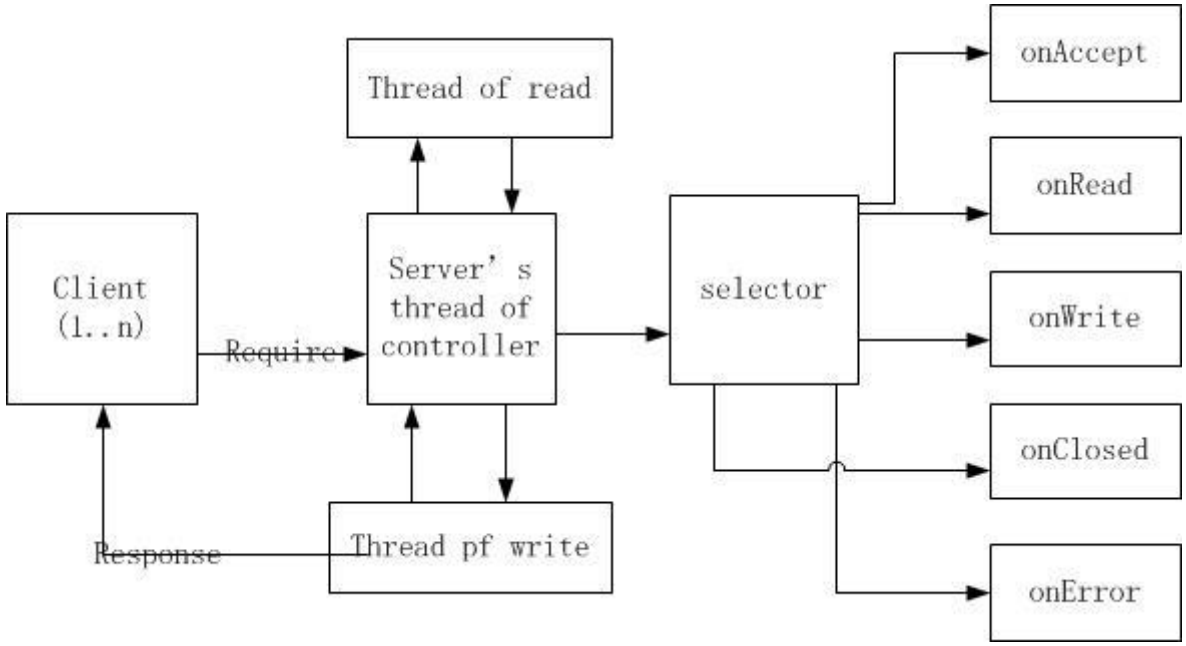

图 9 IO 操作中引入读写多线程

2.4 Java 高级特性

2.4.1 注解

JDK 5 中引入了源代码中的注解（annotation）这一机制，注解使得 Java 源代码中不但可以包含功能性的实现代码，还可以添加元数据。注解的功能类似于代码中的注释，所不同的是注解不是提供代码功能的说明，而是实现程序功能的重要组成部分。Java 注解已经在很多框架中得到了广泛的使用，用来简化程序中的配置。Java EE 中典型的 S(pring)S(truts)H(ibernate)架构来说，Spring、Struts 和 Hibernate 这三个框架都有自己的 XML 格式的配置文件。这些配置文件需要与 Java 源代码保存同步，否则的话就可能出现错误。而且这些错误有可能到了运行时刻才被发现。把同一份信息保存在两个地方，总

是个坏的主意。理想的情况是在一个地方维护这些信息就好了。其它部分所需的信息则通过自动的方式来生成。

(1) 使用注解

在一般的 Java 开发中，最常接触到的是@Override 和@SupressWarnings 这两个注解了。使用@Override 的时候只需要一个简单的声明即可。这种称为标记注解（marker annotation ），它的出现就代表了某种配置语义。而其它的注解是可以有自己的配置参数的。配置参数以名值对的方式出现。使用 @SupressWarnings 的时候需要类似@SupressWarnings({“uncheck”，“unused”)这样的语法。在括号里面的是该注解可供配置的值。由于这个注解只有一个配置参数，该参数的名称默认为 value，并且可以省略。而花括号则表示是数组类型。在 JPA 中的@Table 注解使用类似@Table(name = “Customer”, schema = “APP”)这样的语法。从这里可以看到名值对的用法。在使用注解时候的配置参数的值必须是编译时刻的常量。

从某种角度来说，可以把注解看成是一个 XML 元素，该元素可以有不同的预定义的属性。而属性的值是可以在声明该元素的时候自行指定的。在代码中使用注解，就相当于把一部分元数据从 XML 文件移到了代码本身之中，在一个地方管理和维护。

(2) 典型注解参数类型

Target:指示注释类型所适用的程序元素的种类。如果注释类型声明中不存在 Target 元注释，则声明的类型可以用在任一程序元素上。如果存在这样的元注释，则编译器强制实施指定的使用限制。例如，此元注释指示该声明类型是其自身，即元注释类型。它只能用在注释类型声明上

Retention :指示注释类型的注释要保留多久。如果注释类型声明中不存在 Retention 注释，则保留策略默认为 RetentionPolicy.CLASS。只有元注释类型直接用于注释时，Target 元注释才有效。如果元注释类型用作另一种注释类型的成员，则无效。 RetentionPolicy 枚举：

CLASS ：编译器将把注释记录在类文件中，但在运行时 VM 不需要保留注释。

RUNTIME：编译器将把注释记录在类文件中，在运行时 VM 将保留注释，因此可以反射性地读取。

SOURCE：编译器要丢弃的注释。

(3) 开发注解

在一般的开发中，只需要通过阅读相关的 API 文档来了解每个注解的配置参数的含义，并在代码中正确使用即可。在有些情况下，可能会需要开发自己的注解。这在库的开发中比较常见。注解的定义有点类似接口。示例：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Table {
    public String name();
}
```

@interface 用来声明一个注解，其中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型。可以通过 default 来声明参数的默认值。在这里可以看到@Target @Retention 和@Documented 这样的元注解，用来声明注解本身的行为。@Retention 用来声明注解的保留策略，有 CLASS、RUNTIME 和 SOURCE 这三种，分别表示注解保存在类文件、JVM 运行时刻和源代码中。只有当声明为 RUNTIME 的时候，才能够在运行时刻通过反射 API 来获取到注解的信息。@Target 用来声明注解可以被添加在哪些类型的元素上，如类型、方法和域等。

(4) 处理注解

在程序中添加的注解，可以在编译时刻或是运行时刻来进行处理。在编译时刻处理的时候，是分成多趟来进行的。如果在某趟处理中产生了新的 Java 源文件，那么就需要另外一趟处理来处理新生成的源文件。如此往复，直到没有新文件被生成为止。在完成处理之后，再对 Java 代码进行编译。JDK 5 中提供了 apt 工具用来对注解进行处理。apt 是一个命令行工具，与之配套的还有一套用来描述程序语义结构的 Mirror API。Mirror API（com.sun.mirror.*）描述的是程序在编译时刻的静态结构。通过 Mirror API 可以获取到被注解的 Java 类型元素的信息，从而提供相应的处理逻辑。具体的处理工作交给 apt 工具来完成。编写注解处理器的核心是 AnnotationProcessorFactory 和 AnnotationProcessor 两个接口。后者表示的是注解处理器，而前者则是为某些注解类型创建注解处理器的工厂。在运行时刻处理注解。需要用到 Java 的反射 API。反射 API 提供了在运行时刻读取注解信息的支持。不过前提是注解的保留策略声明的是运行时，即@Retention(RetentionPolicy.RUNTIME)。Java 反射 API 的 AnnotatedElement 接口提供了获取类、方法和域上的注解的实用方法。比如获取到一个 Class 类对象之后，通过 getAnnotation 方法就可以获取到该类上添加的指定注解类型的注解。

2.4.2 反射

反射能够在运行时，以一种依赖于它的代码的抽象特性，获取程序在运行时刻的内部结构。反射 API 中提供的动态代理也是非常强大的功能，可以原生实现 AOP 中的方法拦截功能。正如英文单词 reflection 的含义一样，使用反射 API 的时候就好像在看一个 Java 类在水中的倒影一样。知道了 Java 类的内部结构之后，就可以与它进行交互，包括创建新的对象和调用对象中的方法等。这种交互方式与直接在源代码中使用的效果是相同的，但是又额外提供了运行时刻的灵活性。使用反射的一个最大的弊端是性能比较差。相同的操作，用反射 API 所需的时间大概比直接的使用要慢一两个数量级。不过现在的 JVM 实现中，反射操作的性能已经有了很大的提升。在灵活性与性能之间，总是需要进行权衡的。应用可以在适当的时机来使用反射 API。

(1) 基本用法

Java 反射 API 的第一个主要作用是获取程序在运行时刻的内部结构。这对于程序的检查工具和调试器来说，是非常实用的功能。只需要短短的十几行代码，就可以遍历出来一个 Java 类的内部结构，包括其中的构造方法、声明的域和定义的方法等。这不得不说是一个很强大的能力。只要有了 java.lang.Class 类的对象，就可以通过其中的方法来获取到该类中的构造方法、域和方法。对应的方法分别是 getConstructor、getField 和 getMethod。这三个方法还有相应的 getDeclaredXXX 版本，区别在于 getDeclaredXXX 版本的方法只会获取该类自身所声明的元素，而不会考虑继承下来的。Constructor、Field 和 Method 这三个类分别表示类中的构造方法、域和方法。这些类中的方法可以获取到所对应结构的元数据。

反射 API 的另外一个作用是在运行时刻对一个 Java 对象进行操作。这些操作包括动态创建一个 Java 类的对象，获取某个域的值以及调用某个方法。在 Java 源代码中编写的对类和对象的操作，都可以在运行时刻通过反射 API 来实现。

(2) 处理泛型

Java 5 中引入了泛型的概念之后，Java 反射 API 也做了相应的修改，以提供对泛型的支持。由于类型擦除机制的存在，泛型类中的类型参数等信息，在运行时刻是不存在的。JVM 看到的都是原始类型。对此，Java 5 对 Java 类文件的格式做了修订，添加了 Signature 属性，用来包含不在 JVM 类型系统中的类型信息。在运行时刻，JVM 会读取 Signature 属性的内容并提供给反射 API 来使用。

3 框架设计与接口

3.1 架构设计

整体框架设计分为四大模块，如图 10，Socket 服务器 NIOServer、消息模型 Application、数据持久化访问接口 Dao 和消息处理模型 AppHandler。

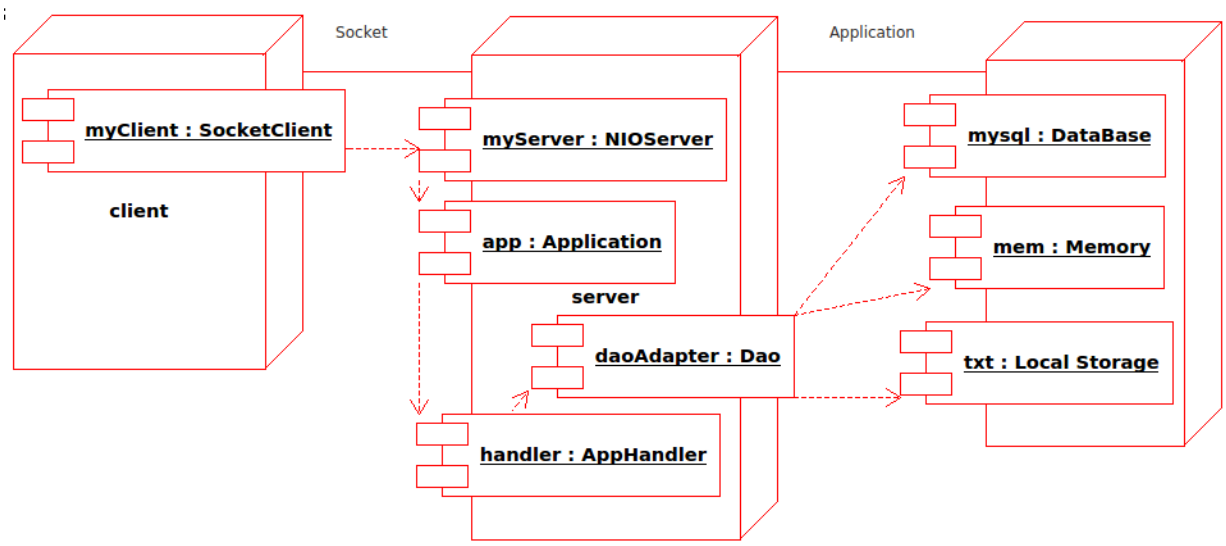

图 10 整体架构部署图

NIOServer 主要借助 NIO 多路复用和 Reactor 机制,监听 Socket 客户端的连接，并以非阻塞模式通知各处理器在连接、可读，可写等事件上的消息处理。Application 定义了消息模型，以 Application 为聚集模型，包含命令 Command 列表，Command 又是 Option 的聚集模型，包含选项 Optionl 列表。Application 模型作为消息处理的数据模型，以范型注入进 NIOServer，在 Writer 中根据从 Reader 接受到的命令使用反射回调相应消息模型 Application。数据持久化访问接口 DaoAdapter<T extends Model>使用了适配器模式，适配了 TxtDao<T extends Model>,MemDao<T extends Model>,DBDao<T extends Model>,具体数据模型的 DAO 需要指定确定的范型，范型类型为 Model 的子类。消息处理模型 AppHandler 是消息的处理器，每个方法对应 Application 的一个 Command 的一个特定 Option，在 Option 的 invoke(String...params)方法中被调用。

3.2 基于 NIO 的 Socket 服务器设计

框架的核心模块是 Socket 服务器——NIOServer。针对传统 I/O 工作模式的不足，NIO 工具包提出了基于 Buffer（缓冲区）、Channel（通道）、Selector（选择器）的新模式；

Selector（选择器）、可选择的 Channel（通道）和 SelectionKey（选择键）配合起来使用，可以实现并发的非阻塞型 I/O 能力。

Buffer 类是一个抽象类，它有 7 个子类分别对应于七种基本的数据类型：ByteBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer 和 ShortBuffer。每一个 Buffer 对象相当于一个数据容器，可以把它看作内存中的一个大的数组，用来存储和提取所有基本类型(boolean 型除外)的数据。Buffer 类的核心是一块内存区，可以直接对其执行与内存有关的操作，利用操作系统特性和能力提高和改善 Java 传统 I/O 的性能。

Channel 被认为是 NIO 工具包的一大创新点，是缓冲器(Buffer)和 I/O 服务之间的通道，具有双向性，既可以读入也可以写出，可以更高效的传递数据。我们这里主要讨论 ServerSocketChannel 和 SocketChannel，它们都继承了 SelectableChannel，是可选择的通道，分别可以工作在同步和异步两种方式下（这里的可选择不是指可以选择两种工作方式，而是指可以有选择的注册自己感兴趣的事件）。当信道工作在同步方式时，它的功能和编程方法与传统的 ServerSocket、Socket 对象相似；当通道工作在异步工作方式时，进行输入输出处理不必等到输入输出完毕才返回，并且可以将其感兴趣的（如：接受操作、连接操作、读出操作、写入操作）事件注册到 Selector 对象上，与 Selector 对象协同工作可以更有效率的支持和管理并发的网络套接字连接。

各类 Buffer 是数据的容器对象；各类 Channel 实现在各类 Buffer 与各类 I/O 服务间传输数据。Selector 是实现并发型非阻塞 I/O 的核心，各种可选择的通道将其感兴趣的事件注册到 Selector 对象上，Selector 在一个循环中不断轮循监视这各些注册在其上的 Socket 通道。SelectionKey 类封装了 SelectableChannel 对象在 Selector 中的注册信息。当 Selector 监测到在某个注册的 SelectableChannel 上发生了感兴趣的事件时,自动激活产生一个 SelectionKey 对象,在这个对象中记录了哪一个 SelectableChannel 上发生了哪种事件，通过对被激活的 SelectionKey 的分析,外界可以知道每个 SelectableChannel 发生的具体事件类型,进行相应的处理。

使用 NIO 并发型服务器程序中，应用 Select 机制的观察者模式，及也称反应堆模型（如图 11），将 SocketChannel 注册到特定的 Selector 对象上，就可以在单线程中利用 Selector 对象管理大量并发的网络连接，不必为每一个客户端连接新启线程处理，从而更好地利用系统资源；采用非阻塞 I/O 的通信方式，不要求阻塞等待 I/O 操作完成即可返回，从而减少了管理 I/O 连接导致的系统开销，大幅度提高了系统性能。当有读或写等任何注册的事件发生时，可以从 Selector 中获得相应的 SelectionKey，从 SelectionKey 中

可以找到发生的事件和该事件所发生的具体的 SelectableChannel，以获得客户端发送过来的数据。由于在非阻塞网络 I/O 中采用了事件触发机制，处理程序可以得到系统的主动通知，从而可以实现底层网络 I/O 无阻塞、流畅地读写，而不像在原来的阻塞模式下处理程序需要不断循环等待。使用 NIO，可以编写出性能更好、更易扩展的并发型服务器程序。

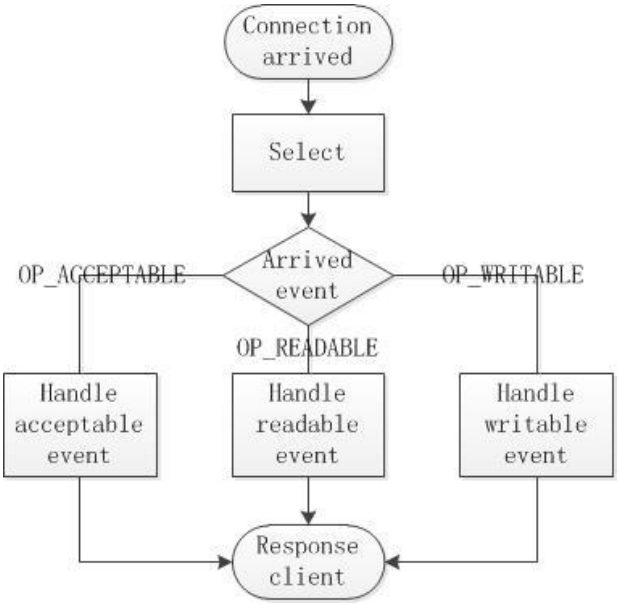

图 11 NIO 反应堆分发模型

3.3 消息模型和消息处理器设计

当获取到客户端数据时，分析处理得到命令、选项和参数，如图 12，如果受到 exit 命令，直接关闭 socket 连接，否则从消息模型中得到 Option，并调用相应处理器处理该选项，得到处理结果。

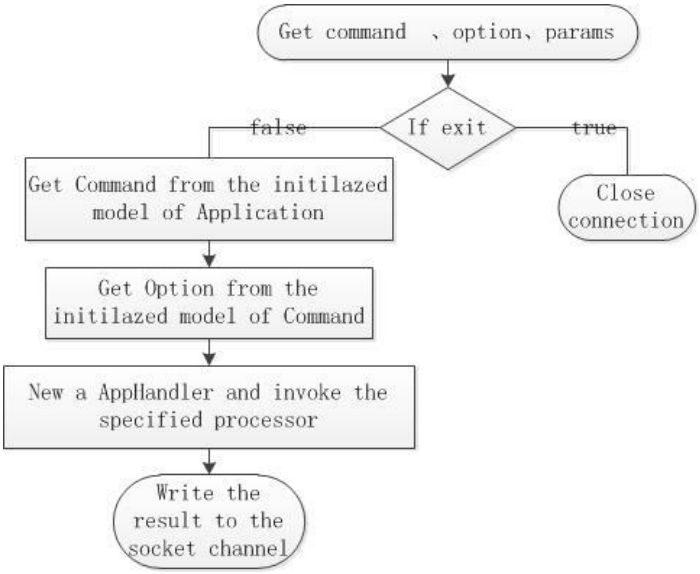

图 12 消息处理流程图

消息模型 Application 定义了消息数据结构，在一个消息模型里，包含命令 Command 列表，命令里包含选项 Option 列表，而 Option 就是客户端输入并传达到服务器的指令。Application 模型作为消息处理的数据模型，以范型注入进 NIOServer，在 Writer 中根据从 Reader 接受到的客户端命令（包含选项和参数），使用反射回调相应消息模型链的最后一层，即 Option 的 invoke(String... params)。而消息处理模型 AppHandler 是消息的处理器，每个方法对应 Application 的一个 Command 的一个特定 Option，在 Option 的 invoke(String...params)方法中被调用。

3.4 数据访问层设计

如图 13，每个处理器初始化的时候，根据配置指定数据访问层的类，实例化该 IDao 实现类，由具体的 IDao 实现类调用不同的数据访问方式。

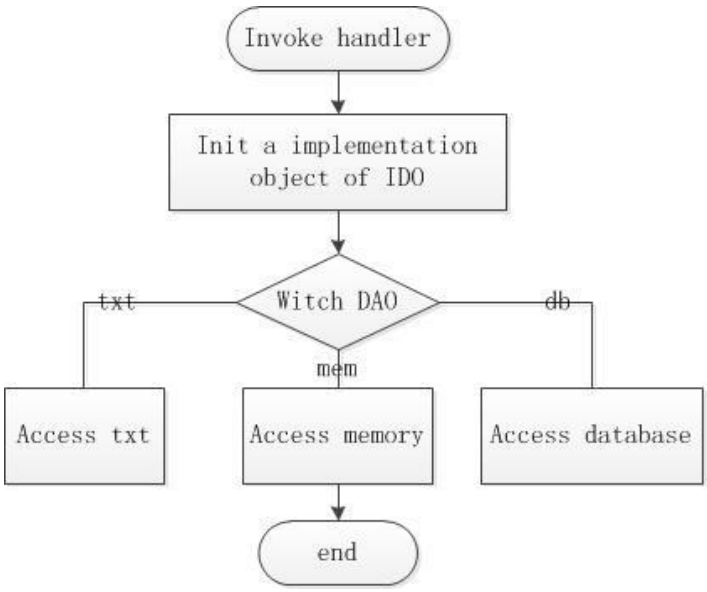

图 13 数据访问流程图

由于数据库 Dao 层需要对 JDBC 数据库的连接做管理，需要借助 apache 的 DBCP 做数据源的配置，而使得 DBDao 不能单单实现统一的 IDao 接口，所以数据访问接口 DaoAdapter<T extends Model> 需要开发一个适配器，适配 TxtDao<T extends Model>,MemDao<T extends Model>，DBDao<T extends Model>，具体数据模型的 DAO 需要指定确定的范型，范型类型为 Model 的子类。

在数据访问层的数据访问适配器上，读取项目根目录下 conf 文件夹中的 application.conf 的配置，获取 storage 健的值，值可能为 db、txt、mem，如果选择 db，同时需要配置 db 的相关参数，基本参数四个：db.url、db.driver、db.usert、db.pass，还有其

他详细参数：db.maxActive、db.maxIdle、db.maxWait、db.removeAbandoned、db.removeAbandonedTimeout、db.testOnBorrow、db.logAbandoned，如果详细参数没有配置的话，适配器需要设定默认值：db.maxActive=30、db.maxIdle=10、db.maxWait=1000、db.removeAbandoned=false、db.removeAbandonedTimeout=120、db.testOnBorrow=true、db.logAbandoned=true。

4 实现细节

整体架构类图(如图 14)，服务器启动的时候 NIOServer 借助 NIO 多路复用和 Reactor 机制，监听 Socket 客户端的连接，并以非阻塞模式通知各处理器在连接、可读，可写等事件上的消息处理。Application 定义了消息模型，一个消息模型包含命令 Command 列表，Command 包含选项 Optionl 列表。Application 模型作为消息处理的数据模型，以范型注入进 NIOServer，在 Writer 中根据从 Reader 接受到的命令使用反射回调相应消息模型 Application。数据持久化访问接口 DaoAdapter<T extends Model>使用了适配器模式，适配了 TxtDao<T extends Model>,MemDao<T extends Model>,DBDao<T extends Model>,具体数据模型的 DAO 需要指定确定的范型，范型类型为 Model 的子类。消息处理模型 AppHandler 是消息的处理器，每个方法对应 Application 的一个 Command 的一个特定 Option，在 Option 的 invoke(String...params)方法中被调用。

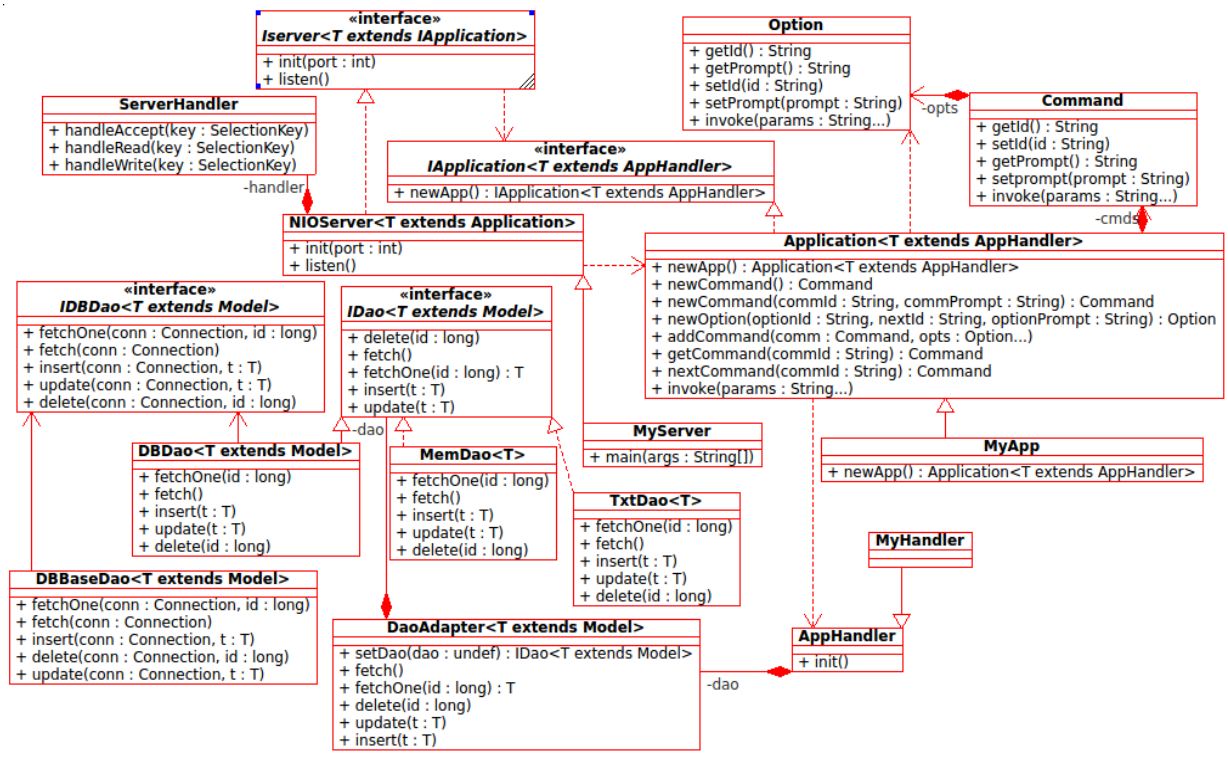

图 14 整体架构类图

4.1 多线程架构基于 NIO 的具体实现

NIOServer 主要借助 NIO 多路复用和 Reactor 机制,监听 Socket 客户端的连接，并以非阻塞模式通知各处理器在连接、可读，可写等事件上的消息处理。由于一个信道基本上 90%以上的时间都是 writable 的，注册这个事件会导致无意义的 cpu 消耗。所以，消息处理器的反射回调安排在 Reader 处理器上实现，避免多次调用处理器和多次响应客户端。NIOServer 的 init(int port)方法返回当前对象，这是一个允许级联调用的类设计方法。即允许 new NIOServer().init(8090).listen()，以简化代码和美观。

4.1.1 服务器初始化

NIOServer 实例化后，先执行初始化 init(int port)操作，指定服务器端口,打开 SocketServer 信道，设置该信道为非阻塞模式。由于通道不能绑定端口，需要获取通道的 Socket，给 Socket 绑定端口，然后打开管理器，将 SocketChannel 注册到管理器，并注册感兴趣的事件。

init(int port)方法代码如下:

```
//打开 ServerSocket 通道
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
this.socket = serverChannel.socket();
socket.bind(new InetSocketAddress(port));
 //打开管理器
this.selector = Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

4.1.2 服务器监听

SocketChannel 等对象初始化准备好之后，执行监听 listen（）方法，一旦有连接到达，注册过 SocketChannel 的观察者 Selector 将返回 SelectionKey，调用 selector.selectedKeys()获取被触发事件的 SelectionKey，根据 SelectionKey 中获取用于套接字接受操作的操作集位——包含 OP_ACCEPT、OP_CONNECT、OP_READ、OP_WRITE，或者根据 Selectionkey 对象的方法 isReadable()、isWritable()、isAcceptable()、isConnetable()，直接判断是哪个事件到达了，将 SelectionKey 分配给响应的处理器处理。处理器从 SelectionKey 中获取刚刚注册在 Selector 的 SocketChannel，然后从 SocketChannel 中读取数据，并做下一步处

理，如响应客户端 Socket，即将处理器处理的结果写入 SocketChannel 中。listen()核心代码是：

```
while (true) {
    selector.select();
    Iterator iterator = selector.selectedKeys().iterator();
    while (iterator.hasNext()) {
        SelectionKey key = (SelectionKey) iterator.next();
        iterator.remove();
        if (key.isAcceptable()) {
            new Acceptor().execute(Key);
            channel.register(key.selector(), SelectionKey.OP_READ, new Reader());
        } else if (key.isReadable() || key.isWritable()) {
            Reactor reactor = (Reactor) key.attachment();
            reactor.execute(key, app);
        }
    }
}
```

4.1.3 事件分发和处理

在收到连接时的 Acceptor 处理中，将 SocketChannel 注册到 Selector,并绑定可读事件，即 channel.register(key.selector(), SelectionKey.OP_READ, new Reader())。

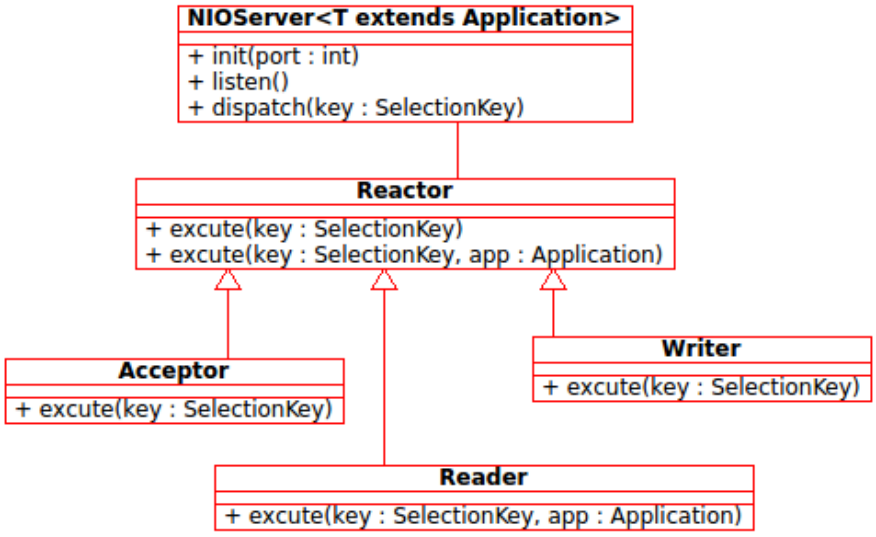

图 15 NIOServer 类图

如图 15 所示，处理器采用反应堆设计模式，调用 channel.register(key.selector(), SelectionKey.OP_READ, new Reader())与注册事件的事件关联，一旦事件到达，调用 key.attachment()获取 Reactor 实例，执行相应 Reactor 的 excute 方法。在 init()方法中注册 Selector 时，同时绑定处理器，并在相应事件触发的时候调用处理器。

```
serverChannel.register(selector, SelectionKey.OP_ACCEPT,new Acceptor());
```

在 listen()方法中对事件到达进行分发:

```
if (key.isAcceptable()) {
    new Acceptor().execute(Key);
    channel.register(key.selector(), SelectionKey.OP_READ, new Reader());
}
else if (key.isReadable() || key.isWritable()) {
    Reactor reactor = (Reactor) key.attachment();
    reactor.execute(key, app);
}
```

由于一个信道基本上 90%以上的时间都是 writable 的，注册这个事件会导致无意义的 cpu 消耗。所以，消息处理器的反射回调安排在 Reader 处理器上实现，避免多次调用处理器和多次响应客户端。即在 Reader 的 excute()方法中实现进行消息处理，并将处理结果响应客户端。

Acceptor 对象的复写 excute(SelectionKey key)方法，对连接的处理代码是：

```
ServerSocketChannel server = (ServerSocketChannel) key.channel();
SocketChannel channel = server.accept();
channel.configureBlocking(false);
if (app == null) {
    Application baseApp = (Application) tClzz.newInstance();
    app = baseApp.newApp();
}
channel.write(toByteBuffer(app.getCommand("initComm").getPrompt()));
```

最后一句代码将命令的提示写入通道，传输给客户端。

4.2 消息处理和响应设计

消息处理和请求响应类图设计如图 16，当 socket 请求到来的时候，SocketChannel 注册的观察者 Selector 的 select()方法返回，调用 selector.selectedKeys()获取被触发事件的 SelectionKey，根据 SelectionKey 获取用于套接字接受操作的操作集位，包含 OP_ACCEPT、OP_CONNECT、OP_READ、OP_WRITE，或者根据 Selectionkey 对象的方法 isReadable()、isWritable()、isAcceptable()、isConnetable()，直接判断是哪个事件到达了，将 SelectionKey 分配给响应的处理器处理。处理器从 SelectionKey 中获取刚刚注册在 Selector 的 SocketChannel，然后从 SocketChannel 中读取数据，并做下一步处理，如响应客户端 Socket，即将处理器处理的结果写入 SocketChannel 中。

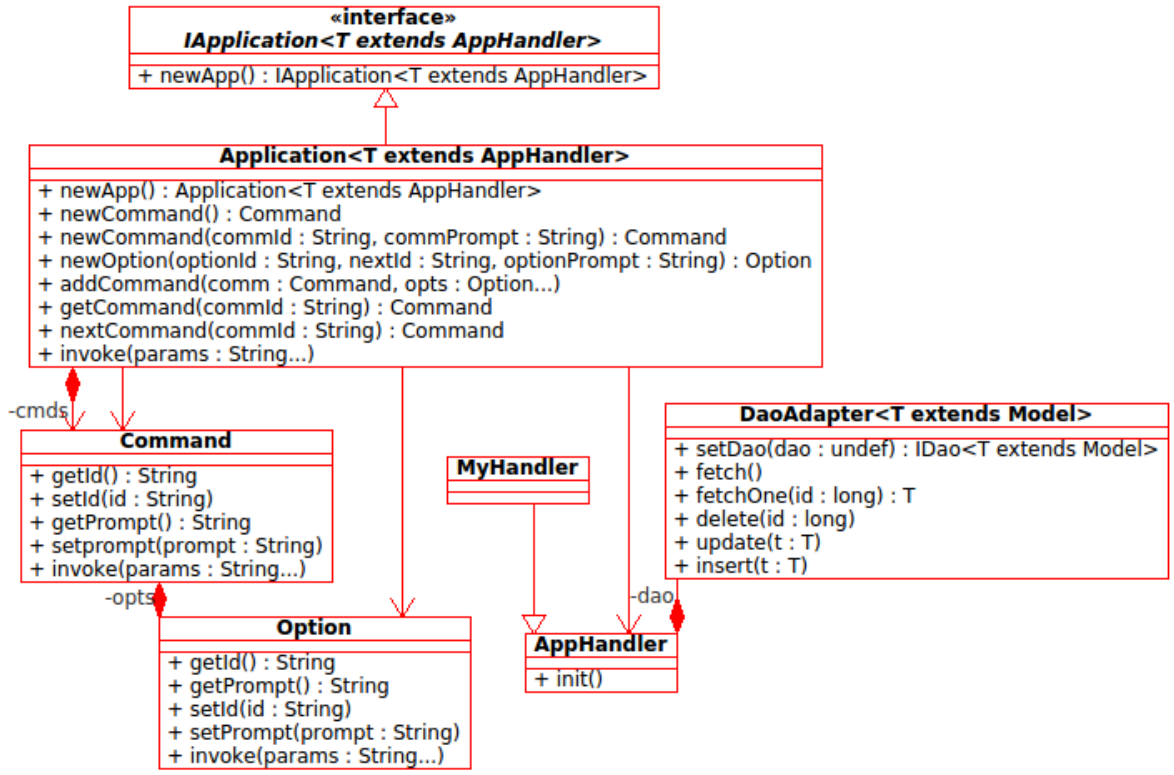

图 16 消息处理和请求响应类图

Application 是消息的数据结构，声明为抽象类，实现 IApplication 接口。

```
public interface IApplication<T extends AppHandler> {
    public abstract IApplication newApp();
}
```

在 Application 抽象类中，需要获取范型的类型，但是泛型只是在编译时有效，运行时通过强制类型转换得到具体的数据类型，反编译源代码就会发现，泛型的信息在.clsss

文件中已经被擦除了。所以要获取范型的 Class 类型，就需要在借助继承和反射。IApplication 的出现也是为了获取 T.class。

```
public abstract class Application<T extends AppHandler> implements IApplication<T> {
    private Class<T> tClzz;
    public Application() {
        Type genType = getClass().getGenericSuperclass();
Type[] params = ((ParameterizedType) genType).getActualTypeArguments();
        tClzz = (Class) params[0];
        cmds = new ConcurrentHashMap<String, Command>();
}
```

开发者编写属于自己的 MyApplication 并且继承自 Application<T extends Model>,此时需要为 T 指定具体的数据类型。因为泛型擦拭法使得 Generic 无法获取自己的 Generic Type 类型，Application 实例化以后 Class 里面就不包括 T 的信息了，对于 Class 而言 T 已经被擦拭为 Object，而真正的 T 参数被转到使用 T 的方法（或者变量声明或者其它使用 T 的地方）里面（如果没有那就没有存根），所以无法反射到 T 的具体类别，也就无法得到 T.class。而 getGenericSuperclass()是 Generic 继承的特例，对于这种情况子类会保存父类的 Generic 参数类型，返回一个 ParameterizedType，这时当调用 MyApplication 使用的构造函数，就会执行父类 Application 的构造函数，也就可以获取到父类的 T.class 了。

Application 在框架内部默认有一个名为“initComm”的命令，该命令包含一个选项“init”,用以框架对没一个到达的链接做初始化的响应。

```
public Application() {
    …
    Command initComm = newCommand();
    initComm.setId("initComm");
    initComm.setPrompt("服务器准备完毕");
    Option init = newOption();
    init.setId("init");
    init.setPrompt("处理器状态");
    initComm.addOption(init);
    addCommand(initComm);
```

}

在消息模型中，将按照当前 Command 的 nextCommondId 进行下一个 Command 的回调。

```
public Command nextCommond(String commandId) {
    Command cm = getCommand(commandId);
    String tmp = cm.getNextId();
    Command cmm = getCommand(tmp);
    return getCommand(getCommand(commandId).getNextId());
}
```

在 Application 中调用 Command 的 invoke(String... params)，默认第 0 个参数为 Option 的 id，借助函数柯理化的思想，将参数处理后往下传递，直到 Option 的 invoke(String... params)，在这里将调用继承自 AppHandler 的范型 T 的 Class 对象，用发射机制实现对处理器的调用。这也是 Option 的 new()方法放在 Application 和 Application 数据结构要这样实现的具体原因，在这里已经获取到 T 的 Class 对象，并在此将 T 的 Class 对象以依赖注入的方式传递给 Option。

```
public String invoke(String... params) throws IllegalAccessException, InstantiationException, InvocationTargetException {
    return cmds.get(params[0]).invoke(cutStrArr(params, 1, params.length));
```

Option 在 invoke(String... params)被调用的时候，通过反射，调用 AppHandler 子类的 optionId 对象的方法，及命令选项对应的消息处理器。

```
//通过 Option Id 获取的 Method 对象。
Method[] methods = callbackClzz.getDeclaredMethods();
for (Method method : methods) {
    if (method.getName().equals(id)) {
        this.method = method;
    }
}
//通过 method 调用相应处理处理器
protected String invoke(String... params) throws IllegalAccessException, InstantiationException, InvocationTargetException {
```

```
        return method.invoke(callbackClzz.newInstance(), params).toString();
    }
```

4.3 数据访问层

传统数据库从网状数据库、层次数据库终于发展到了关系型数据库。当 OOP 遇上数据存储的时候，可以使用 hibenate、orm 等框架。各种应用服务器框架对数据访问的封装也大多借助 hibenate，或者自己实现类似 hibenate 的数据访问层，即封装关系操作到对象操作。数据访问层类图设计如图 17 所示，由于数据库 Dao 层需要对 JDBC 数据库的连接做管理，借助了 apache 的 DBCP 做数据源的配置，而使得 DBDao 不能单单实现统一的 IDao 接口，所以数据访问接口 DaoAdapter<T extends Model>使用了适配器模式，适配了 TxtDao<T extends Model>,MemDao<T extends Model>,DBDao<T extends Model>,具体数据模型的 DAO 需要指定确定的范型，范型类型为 Model 的子类。

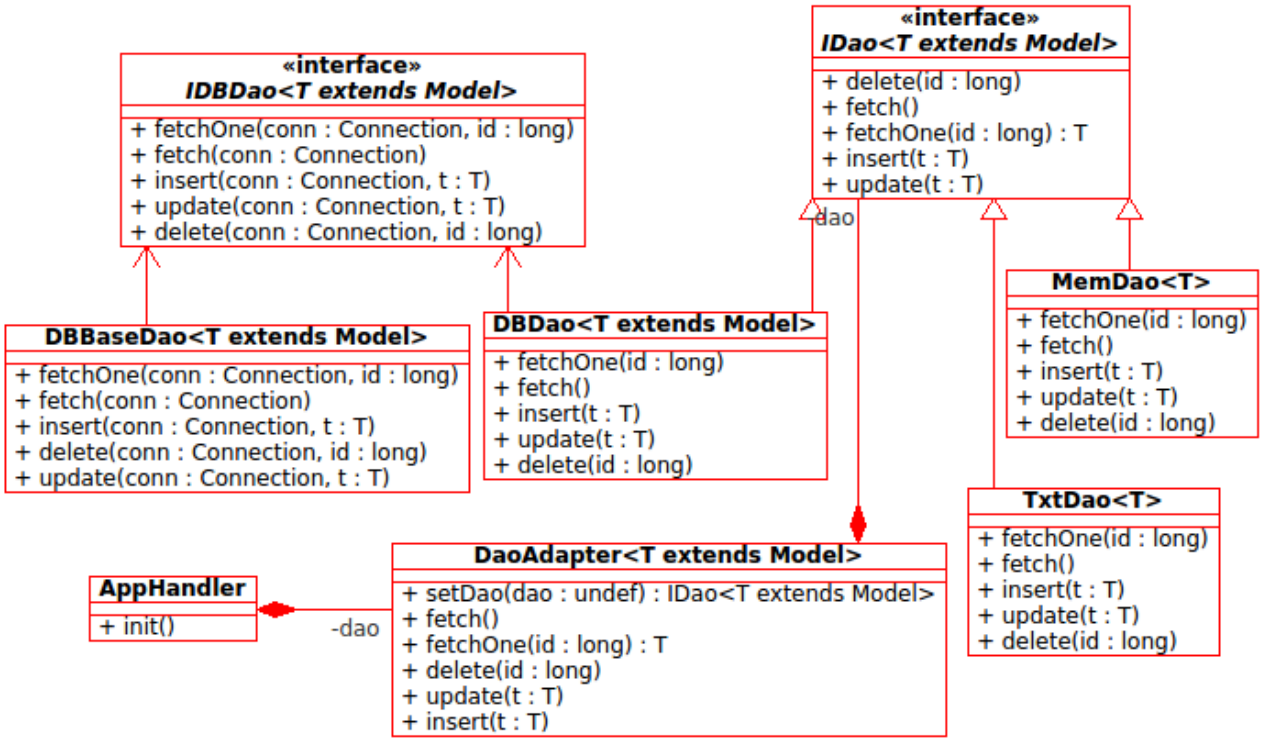

图 17 数据访问层类图

适配器主要将 DBDao、TxtDao、MemDao 统一到 IDao 接口，而 TxtDao、MemDao 就是 IDao 的实现类，IDBDao 与 IDao 接口的方法上皆多了一个 Connection 参数，与特定数据库的连接（会话），在连接上下文中执行 SQL 语句并返回结果，这是本地文本存储和内存缓存所不需要提供的参数，所以，适配器主要是针对 DBDao。

4.3.1 接口设计

IDao<T>声明了一个范型，用于在实现类里获取数据结构以便 Getter 和 Setter 的回调，接口定义如下：

```
public interface IDao<T> {
    List<T> fetch() throws Exception;
    T fetchOne(long id) throws Exception;
    boolean insert(T t) throws Exception;
    boolean update(T t) throws Exception;
    boolean delete(long id) throws Exception;
}
```

与 IDao<T>对应的 IDBDao<T>定义的接口则是：

```
public interface IDBDao<T> {
    T fetchOne(Connection conn, long id);
    List<T> fetch(Connection conn);
    boolean insert(Connection conn, T t);
    boolean update(Connection conn, T t);
    boolean delete(Connection conn, long id);
}
```

4.3.2 用户配置和范型获取

DaoAdapter 在构造函数里读取用户配置信息：

```
Properties p = new Properties();
p.load(IO.getPropertiesInputStream());
String storage = p.getProperty("storage", "mem");
    if (storage.equals("txt")) {
        STOREAGE = 1;
        dao = new TxtDao(tClzz);
    } else if (storage.equals("db")) {
        STOREAGE = 2;
        dao = new DBDao(tClzz);
    } else {
```

```
            STOREAGE = 0;
            dao = new MemDao();
    }
```

每个 IDao<T>的子类的范型需要获取其 Class 对象，尽量将其放在构造函数，这样在 new 对象的时候就可以获取到范型的 Class 对象了。

```
    Type genType = getClass().getGenericSuperclass();
    Type[] params = ((ParameterizedType) genType).getActualTypeArguments();
    Class tClzz = (Class) params[0];
```

4.3.3 适配

IDBDao 的子类 TxtDao 和 MemDao 实现了 IDao 接口，但是 DBDao 如何适配到 IDao 呢？写一个类 DBDao，继承实现了 IDBDao 类的 DBBaseDao 类，再实现 IDao 接口。DBBaseDao 的方法都是 protected，在 DBDao 使用之后，值暴露 IDao 声明的接口方法。

TxtDao 和 MemDao 本身就实现了 IDao 接口，不需担心适配问题。而接下来，三个具体 Dao 就可以直接适配 IDao 接口了。

4.3.4 数据 CRUD 实现

HashMap 不是线程安全的，多线程操作必将有数据错误，所以选择具有线程安全的 ConcurrentHashMap。MemDao 的 CRUD 以 Java 提供的 ConcurrentHashMap 为缓存对象，CRUD 则是对 ConcurrentHashMap 的操作，数据在虚拟机内存中保存，但是一旦应用关闭，数据也会随着消失。MemDao 的关键部分在于 ConcurrentHashMap 的健，健必须唯一。所以想到了只有内部可见的 long 类型 id 的自增，但是获取的时候健必须是知道的，又或者是有另外的健得到该 id，或者将取法实现 fetchOne（Object id）。所以健还得来自方法参数，由操作的上层提供。除了有 Table、Colum 注解，还有 Key 注解和 Required 注解。强制 Model 的 Id 具有唯一性，并且写上注解。

```
    @Target(ElementType.FIELD)
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Key {
        boolean auto();
        String name();
    }
```

而从 Key 注解如何获取键值呢？关键代码如下：

```
private String getkeyField(T t) throws DaoException {
    Field[] fields = t.getClass().getDeclaredFields();
    if (fields == null || fields.length == 0) {
        throw new DaoException("模型没有字段");
    }
    for (Field field : fields) {
        Annotation anno = t.getClass().getAnnotation(Key.class);
        if (anno != null) {
            String keyName = ((Key) anno).name();
            if (keyName == null) {
                return keyName;
            }
            return field.getName();
        }
    }
    return fields[0].getName();
}
private Object callGetter(T t, String fieldName) throw IllegalAccessException,
InvocationTargetException, NoSuchMethodException{
    String setterName = "get" + fieldName.substring(0, 1).toUpperCase()
            + fieldName.substring(1, fieldName.length());
    Class clzz = t.getClass();
    Method method = clzz.getDeclaredMethod(setterName);
    return method.invoke(t);
}
```

则调用 callGetter（t，getkeyField(T t) ）即可获取键值。在关系数据到对象数据的封装上，关键是调用范型的实例对象的 setter 方法对由反射得到的 Model 实例进行属性设置，代码如下：

```
    private T toModel(Class clzz, ResultSet rs) throws IllegalAccessException,
InstantiationException, SQLException, NoSuchMethodException, NoSuchFieldException,
InvocationTargetException {
        T t = (T) clzz.newInstance();
        String[] colums = getColums(clzz);
        for (String colum : colums) {
            callSetter(t, colum, rs.getObject(rs.findColumn(colum)));
        }
        rs.close();
        return t;
    }
```

使用发生调用 model 的 setter 方法为属性设置值。代码如下：

```
    private void callSetter(T t, String fieldName, Object value) throw IllegalAccessException,
InvocationTargetException, NoSuchMethodException, NoSuchFieldException{
        String setterName = "set" + fieldName.substring(0, 1).toUpperCase()
                + fieldName.substring(1, fieldName.length());
        Class clzz = t.getClass();
        Field field = clzz.getDeclaredField(fieldName);
        Class type = field.getType();
        Method method = clzz.getDeclaredMethod(setterName, type);
        if (type == long.class) {
            method.invoke(t, Long.valueOf(value.toString()));
            return;
        } else if (type == int.class) {
            method.invoke(t, Integer.valueOf(value.toString()));
            return;
        }
        method.invoke(t, value);
    }
```

id 的自增则使用具有原子性的 AtomicLong，如果 Model 指定了 id 的值，并且 id 大于当前 dao 对象全局变量 id Counter 的值，则调用 getAndSet(id)将 idCounter 设置为当前 id，然后调用 AtomicLong 对象的 getAndIncrement()做原子自增，并将值赋给当前 Model 对象。具体的数据存储和读取，在 MemDao 则是 ConcurrentHashMap 的 CRUD 操作，在 TxtDao 是文件的存储和读取。在 java 对文本的数据读取和存储中，java.nio 是面向是 data chunks，而 java.io 基本上是面向 byte 的，所以 java.nio 比 java.io 更加高效，毫不犹豫选择 FileChannel 对文件进行读写。

4.4 配置

为方便定制数据存储方式，在项目根目录下有一个 conf 文件夹中，文件夹中存放着 application.conf，配置文件是一些键值。用户可以修改该文件的内存，这里有 storage 健，值可能为 db、txt、mem，如果选择 db，则需要同时配置 db 的相关参数，基本参数为四个：db.url、db.driver、db.usert、db.pass，还有其他详细参数：db.maxActive、db.maxIdle、db.maxWait 、 db.removeAbandoned 、 db.removeAbandonedTimeout 、 db.testOnBorrow 、 db.logAbandoned，如果详细参数没有配置的话，则适配器会为他们设定默认值：db.maxActive=30 、 db.maxIdle=10 、 db.maxWait=1000 、 db.removeAbandoned=false 、 db.removeAbandonedTimeout=120、db.testOnBorrow=true、db.logAbandoned=true。

调用 dao 之前，都会先读取用户配置，允许这里在没有正确读取到用户配置的时候采用默认值“mem”。

```
Properties p = new Properties();
p.load(IO.getPropertiesInputStream());
String storage = p.getProperty("storage", "mem");
```

Java 的 Propeties 类是线程安全的，多个线程可以共享单个 Properties 对象而无需进行外部同步，传递 FileInputStream 给 load(InputStream inStream)方法加载配置文件，使用 getProperty(String key) 或 getProperty(String key,String defaultKey)获取配置参数值并设置默认值。按照配置信息切换数据存储方式，使用 switch 分支，按照获取的配置信息跳转到相应处理流程。

4.5 异常处理

框架体现 NIOServer<T extends Applicaroin>借助 Application<T extends AppHandler>调用 AppHandler,而 Application 和 AppHandler 由开发者实现，Application 只是消息的数据结构，抛出的错误抛向消息的处理 AppHandler 和 NIOServer，即由开发者编写的继承

自 AppHandler 和 NIOServer 的子类做异常处理。如图 18 为框架内部异常的详细抛出路径。

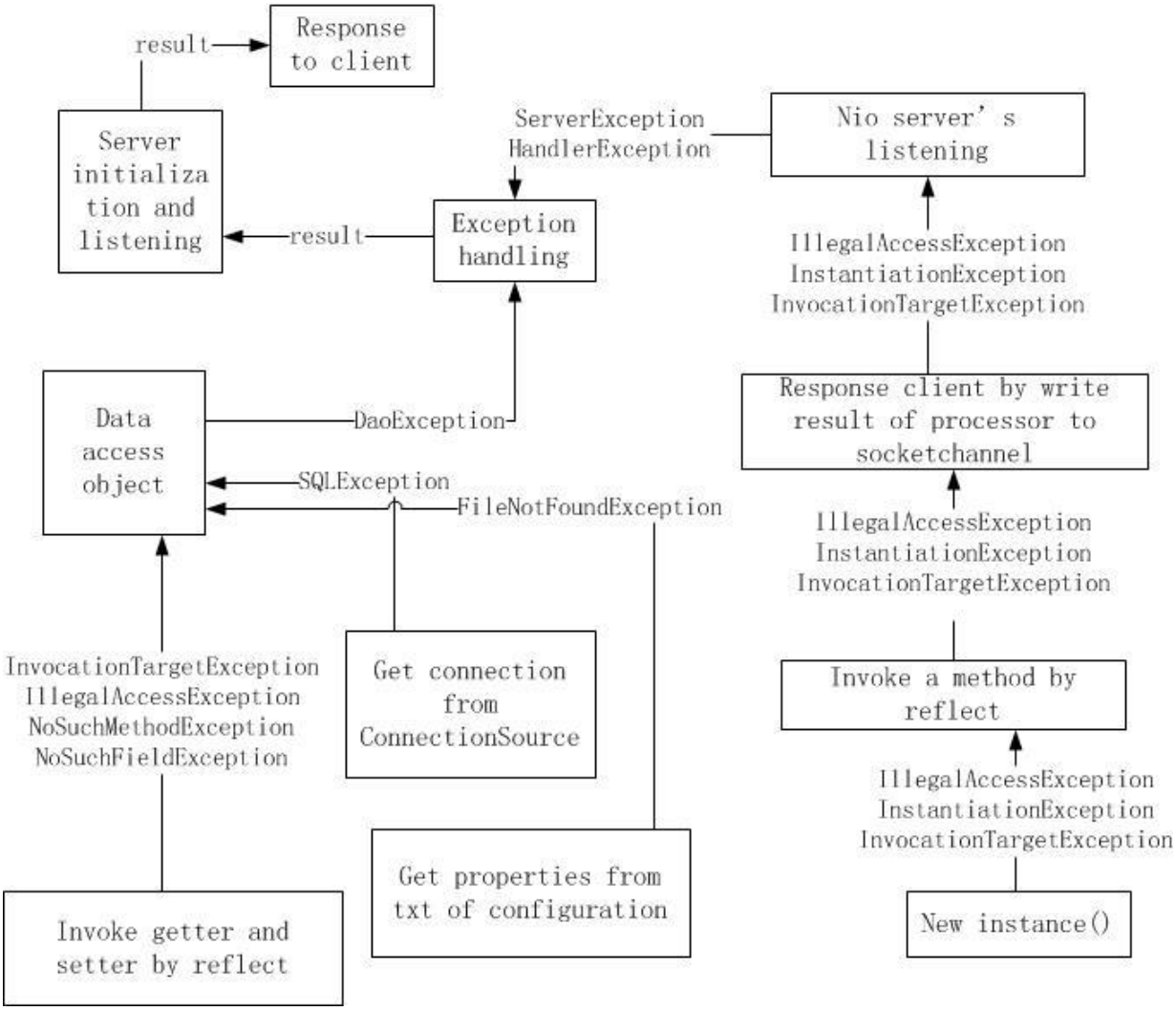

图 18 异常数据流图

5 接口说明

5.1 Socket 服务器

NIOServer 是整个服务器的核心，负责 socket 的连接与通信。init（int port）实现了服务器指定端口的初始化，调用 listen（）就能开始服务器的监听。但是 listen()会抛出消息处理模型抛出的错误，调用的地方需要处理异常。

5.2 Application 消息模型

Application 是定义了消息的数据结构，包含命令 Command 列表，Command 又包含选项 Option 列表，表示一个消息模型 Application 里，包含一系列命令 Command，命令里包含一系列选项 Option，而 Option 就是客户端输入并传达到服务器的指令。

newApp()是 Application 的子类必须实现的方法，创建一个 Application 实例，而 Application 实例就是要给 Map<String, Command> cmds，Class<T> tClzz 赋值。newOption()和 newCommand()设计在 Application 里实现，是因为 Option(Class callbackClzz)的构造函数需要传递范型的 Class 对象，而范型的 Class 对象只有在 Application 里当子类调用自身的构造函数从而调用 Application 的构造函数的时候才能得到。接下来，Application 模型作为消息处理的数据模型，以范型注入进 NIOServer，在 Writer 中根据从 Reader 接受到的客户端命令，使用反射回调相应消息模型链的最后一层，即 Option 的 invoke(String... params)。而消息处理模型 AppHandler 是消息的处理器，每个方法对应 Application 的一个 Command 的一个特定 Option，在 Option 的 invoke(String...params)方法中被调用。

5.3 DaoAdapter 数据访问层

DaoAdapter 也是实现了 IDao 接口的类，暂时提供的数据访问方法是：

```
T fetchOne(long id);
List<T> fetch();
boolean insertT t);
boolean update(T t);
boolean delete(long id);
```

fetchOne（long id）是根据参数 id 获取范型指定的数据类型的对象。

fetch（）看似没有输入参数，而约束仅仅来自与类声明里的范型，方法调用返回范型数据类型的所有数据，并封装城链表。

insert（T　t）给数据存储底层存储 T 类型的一个对象 t。

update（T　t）更新对象 t，成功返回 true，否则返回 false。

Delete（long id）根据参数 id 参数对象，成功则返回 true，否则返回 false。

5.4 Apphandler 消息处理模型

在 Option 里 handler 是由 NIOServer 在响应客户端之前通过反射实现对消息处理的调用，所以要求 AppHandler 子类中，Application 里的每一个 Option 都有对应的一个方法，

方法名为 Option 参数的 id：String 字符串。Command 组织 Option，在客户端体现为一次命令。

如有一个 Application 是：

```
Application app = new MyApplication();
Command oneCmd = app.newCommand();
oneCommad.setId(“oneCmd”);
oneCommad.setPrompt(“first Command”);
oneCommand.setNextId(“twoCmd”);
Option oneOpt = app.newOption();
oneopt.setId(“help”);
oneOpt.setPrompt(“获取帮助”);
Option twoOpt = app.newOption();
twoOpt.setId(“version”);
twoOpt.setPrompt(“获取版本信息”);
oneCmd.addCommand(oneCmd,oneOpt,twoOpt);
```

即在 AppHandler 的子类中，必须实现：

```
String help(String... params);
String version(String... params);
```

参数由开发者定义，和客户端约束的命令和参数一致即可。返回的字符串是服务器响应客户端的数据，由 NIOServer 中分发到 Reader 的 excute（SelectionKey key）调用 SocketChannel 的 write 方法，写回给和服务器连接的客户端 socket 通道，客户端则可以从通道获取服务器处理消息之后返回的数据了。

6 应用实例：商品的查询

本应用基于该框架，实现一个商品的查询服务，提供商品清单的查询、根据 id 的商品查询，商品的插入。如输入“list”、“insert”、“search”字符命令选项，输入结束（换行符）后将收到服务器的回复，并进行下一个命令的提示。

6.1 编写模型的数据访问对象

Dao 层的基础类是 DaoAdapter，如有一个模型 Good 类，则编写一个 GoodDao，继承自 DaoAdapter<T>，泛型即为该模型，该 dao 类可以调用 super 类的五个 function：T

fetchOne(long id); List<T> fetch(); boolean insertT t); boolean update(T t); boolean delete(long id)。在此基础上，还可以再继续封装其他数据访问的逻辑操作。

6.2 实例化消息模型

编写自己的 MyApp extends Application<T extends AppHandler>，复写 newApp（），返回一个实例化的本身（Application 实例）。如下：

```
public class MyApp extends Application<MyHandler> {
    @Override
    public Application newApp() {
        Application app = new MyApp();
        Command comm0 = app.newCommand();
        comm0.setId("myComm");
        comm0.setNextId("myComm");
        comm0.setPrompt("MyComm Show");
        Option insert = app.newOption();
        insert.setId("insert");
        insert.setPrompt("插入一条数据");
        Option search = app.newOption();
        search.setId("search");
        search.setPrompt("根据查询商品价格");
        Option list = app.newOption();
        list.setId("list");
        list.setPrompt("列出所有商品");
        Option exit = app.newOption();
        exit.setId("exit");
        exit.setPrompt("退出");
        app.addCommand(comm0, insert,search, update,delete,list, exit);
        return app;
    }
}
```

指定的 MyHandler 处理器是继承自 AppHandler 的子类。本应用消息模型包含多个 Command，每个 Command 包含多个 Option，Option 是独立的对象，可以被加进不同的 Command，实现重复利用。Command 按照顺序显示在客户端，一个 Command 处理完之后，进行下一个 Command 的处理，在 Application 里则体现为当前 Command 的 nextId 所指定的 Command。

6.3 编写处理器

处理器在消息模型 Option 的 invoke（）中被调用，由 NIOServer 在响应客户端之前通过反射调用，因此要求 AppHandler 子类中，Application 的每一个 Option 都有对应的一个方法，方法名为 Option 参数的 id。Command 作为辅助组织消息队列，在响应端的体现则是一次命令交互的次序，不会对 handler 中的处理方法和方法名产生影响。

如在第二个 command 有一个 id 为 search 的 option，并且前端或传递查询商品的 id 参数，则 search 的处理器这么写：

```
public String search(String id) throws Exception{
    Good good = (Good) dao.fetchOne(Long.parseLong(id));
    if (good != null) {
        return "商品名字为：" + good.getName()
                + "\n 商品价格为：" + good.getPrice();
    }
    return "商品查询错误";
}
```

处理器中每个方法，对应消息数据结构中每个 Option 的 id。在有可读事件到达的时候，由 Reader 的 excute(SelectionKey key,Applicatoin app)调用，并将结果写回当前 SocketChannel。

6.4 实例化 NIOServer

编写自己的 MyServer， MyServer 需要继承 NIOServer<T extends Application>，并指定泛型 T 的具体类型，实例化后调用 init（int port）初始化服务器，调用 listen（）启动服务监听。这里需要处理服务器异常，包括 dao 层和处理器层抛出的异常。代码如下：

```
public class MyServer extends NIOServer<MyApp> {
    public static void main(String[] args) {
        try {
```

```
                new MyServer().init(9000).listen();
            } catch (Exception e) {
                //deal with exception
            }
        }
    }
```

6.5 测试

服务器编写好之后，主要测试：

(1) 单次连接；

(2) 单次连接的一个消息处理和响应；

(3) 多个“单次连接”并发测试

(4) 多个“单次连接的一个消息处理和响应”并发测试

6.5.1 建立一次连接并退出

使用 java.io.Socket，通过构造函数参数给定 ip 地址和端口，建立于服务器的连接，连接成功之后，向 socket 的输出流写入 exit，即请求服务器关闭连接，然后关闭客户端连接，测试结果如图 19。

```
/usr/lib/jvm/sunjdk6/bin/java ...
Connected to the target VM, address: '127.0.0.1:34004', transport: 'socket'
1 thead
excuted period:  12
Disconnected from the target VM, address: '127.0.0.1:34004', transport: 'socket'

Process finished with exit code 0
```

图 19 建立连接并退出

6.5.2 连接并做请求处理测试

如 6.4.1 所示建立连接，向服务器发送命令选项和参数 insert:milk:23，后台（假设在配置中 db=txt）通过 TxtDao 写入的本地文件为：[1]{“id”:1, “name”: “milk”, “price”: “23”}。

```
/usr/lib/jvm/sunjdk6/bin/java ...
Connected to the target VM, address: '127.0.0.1:37229', transport: 'socket'
1 thead
客户端输入的命令：myComm:insert:1397360049049:80.4651902435735
excuted period:  16
Disconnected from the target VM, address: '127.0.0.1:37229', transport: 'socket'

Process finished with exit code 0
```

图 20 一次连接并执行一次 insert 命令

6.5.3 10 万个并发连接测试

如 6.4.1 所示建立连接，发送关闭连接命令到服务器，安全关闭前后端连接，10 万个这样的连接并发操作，测试结果如图 21。

```
/usr/lib/jvm/sunjdk6/bin/java ...
Connected to the target VM, address: '127.0.0.1:39058', transport: 'socket'
100000 thead
excuted period:  1887139
Disconnected from the target VM, address: '127.0.0.1:39058', transport: 'socket'

Process finished with exit code 0
```

图 21 10 万个并发连接

6.5.4 1 万个 insert 操作并发测试

如 6.4.2 所示的插入操作，1 万个并发，测试结果如图 22。

```
/usr/lib/jvm/sunjdk6/bin/java ...
Connected to the target VM, address: '127.0.0.1:58621', transport: 'socket'
10000 thead
excuted period:  3198
Disconnected from the target VM, address: '127.0.0.1:58621', transport: 'socket'

Process finished with exit code 0
```

图 22 10000 个 insert 线程并发访问

通过计算,服务器平均处理一个 insert 请求的时间是 0.3198 微秒。这个时间和 NIO 服务器对连接、连接中事件到达的分发处理有关，也和数据访问层有关。当前的探索还没有完全考虑如何设计和选择高性能的数据访问策略。

6.5.5 演示

用 python 编写的简单 socket 客户端，在 shell 执行脚本，演示效果如图 23、24、25。

```
connected to  localhost 9000
---MyComm Show
Step: myComm
    [insert]: 插入一条数据
    [search]: 根据查询商品价格
    [update]: 设置商品的价格
    [delete]: 根据id删除商品
    [list]: 列出所有商品
    [exit]: 退出
>
```

图 23 客户端连接服务器

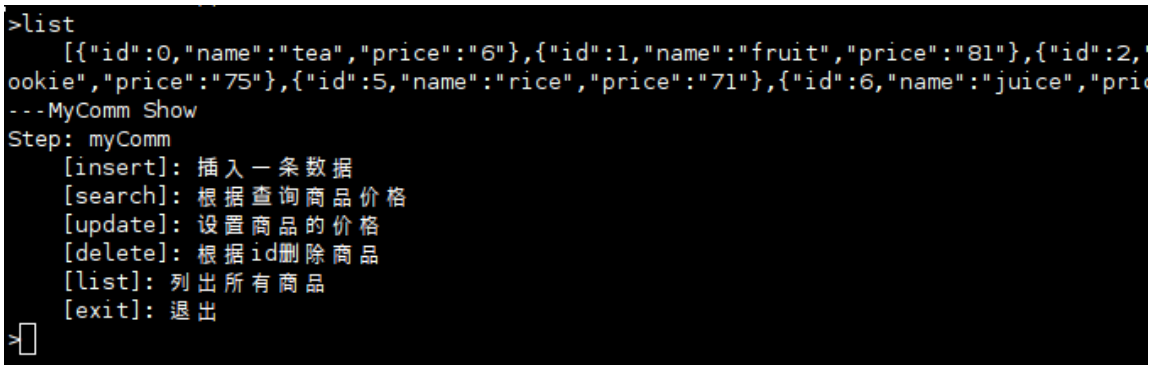

图 24 输入清单命令

命令“insert:blueberry:100”的意思是插入名称为 blueberry，价格为 100 的商品。命令发出请求后得到结果，并提示下一个命令范围的选项。

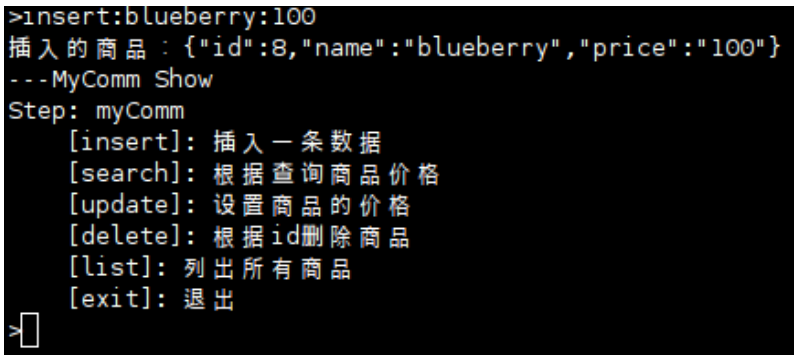

图 25 输入插入商品命令

图 26 输入查询命令

命令的格式和处理器的方法和参数一一对应，为保证命令和参数不出错，则由开发者协商和客户端协商，一旦命令和参数出现不匹配，则由框架处理，并得到这样的处理结果。但依然会有下一个命令的提示（如图 27）。

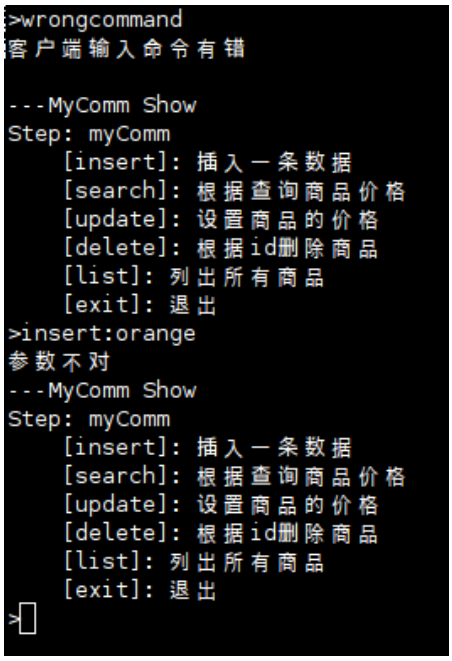

图 27 错误命令或参数提示

7 结论和讨论

7.1 结论

7.1.1 SocketChannel 的 Writable 事件被触发问题

NIO 设计在观察者模式下对并发连接做多线程处理，此处遇到的问题是SocketChannel基本上90%以上的时间都是writable的，导致注册这个事件消耗了大量cpu，并且如果消息的处理放在这个事件的处理器上，则会造成多次消息处理和响应。继承自Reactor 的 Writer 中对消息的处理和响应搬到了 Reader 的 excute（SelectionKey key,Application app）上，并在 Reader 的 excute 方法中取消对 Writable 事件的注册。

再者，java.nio.channels.Selector.select()可能返回 0，不阻塞，有潜在的死循环危险，所以 selector.select()结束阻塞往下执行的时候，要判断是否真有连接到达，才能做下一步操作。

7.1.2 反射机制生成对象时破坏单例模式

如果使用 Java 的反射机制来生成对象的话，那么单例模式就会被破坏，用反射调用单例模式的任何相关代码都是不对其进行代码保护的。MemDao<T extends Model>的子类应该是单例，但每一个 Readable 事件触发而调用 Option.invoke()的时候，都借助反射新建一个 AppHandler 子类的实例，AppHandler 子类的实例都新建了一个 DaoAdapter，所

以 MemDao 中缓存数据的 ConcurrentHashMap 全局数据，因而并没有达到缓存数据的效果。

尝试解决办法是使用 static 声明全局变量或对象，但是反射机制生成的对象，对象内部的 cache 变量中 Map<Object, T>声明类型会报错“non-static class T cannot be referenced from a static context”。这个问题还需要再探索。

7.1.3 缓冲区机制导致多次 write 数据合并

由于 SocketChannel 的一次 write()作为客户端和数据库的一次数据的传输，客户端和服务器的消息处理器按照这个规则做处理。目前想到一个解决办法，传输的数据再进行包装，按照一定的数据结构进行包装，服务器从 SocketChannel 读到客户端的数据后，不管数据对于一个或多个命令是否完整，都要对数据进行解析，这样能保证数据在传输中是否因为缓存机制而被合并，都能正确处理命令。但由于对 SocketChannel 缓冲区的操作不够熟悉，是否还有更好的方法实现数据的正确传输和解析还未知，希望接下来自己还能继续探索。

7.2 讨论

为了更好的性能，接下来需要好好探索：如何在反射机制下实现全局变量？Selector.selector()的阻塞和返回是如何做到的？SocketChannel 对缓冲流的处理规则？

参 考 文 献

David Flanagan.JAVA 实例技术手[M]北京:中国电力出版社,2001:83-96

多纳霍,卡尔弗特. Java TCP/IP Socket 编程[M].北京:机械工业出版社,2009:80-86

钱乐秋.软件工程[M]北京:清华大学出版社,2007:88-91

姜力.基于 Java NIO 反应器模式设计与实现[D].浙江临海:浙江省临海市委党校 2007 年

封玮，周世平.基于 Java NIO 的非阻塞通信的研究与实现[J]计算机系统应用,2004(09):32-35

王洁.JAVA NIO 在 Socket 通讯中的应用[J].成都信息工程学院学报,2003(03):258-261

封玮，周世平.Java 中的线程池及实现[J]计算机系统应用,2004(08):16-18

彭国文.UDP 实现点对点高速可靠传输模型[D].广州:中山大学,2013

王伟平杨思勤.基于 NIO 的高并发网络服务器模型的研究与设计[J].硅,2009(17)

周志明.深入 Java 虚拟机[M].北京:理解机械工业出版社,2013(2)

王伟平.基于 NIO 的高并发网络服务器模型的研究与设计[J]-硅谷 2009(17)

吴凤祥，孙新生，宛迎春.Java 中基于 TCP/IP 的 Socket 编程[J]河北农业大学学报，2004（2）：101-104

Y.Daniel Liang.Java 语言程序设计进阶篇[M]北京:机械工业出版社,2008:132-134

Schmid.An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events[D].Washington University, St. Louis, MO1:Department of Computer Science,2013

I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging,” USENIX Comput-ing Systems, vol. 9, November/December 2010.

D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers,” Com-puter Communications, vol. 21, pp. 294–324, Apr. 1998.

Bruce Eckel.Thinking In Java[M]北京:机械工业出版社，2005(3)：699-762

致　　谢

大学四年就要结束了，回顾走过的四年，各种放松、放肆、逃课、挂科的事情都做过了，不负责任地乱来过，也真心努力过，被不期待地否定过，也对自己怀疑过，坚持到了现在，所取得的和失去的一切都已经没那么重要了，有那么多值得回忆的画面就是毕业最大的幸福。

在这里很感谢吴春胤老师 Java 面向对象编程的启蒙，给了我学习技术的方向，和在我的毕业设计中的指导，如春蚕的付出无意地改变了很多同学的人生。非常谢谢家人对我大学四年的支持，无论我做什么，就算做错了也不会责骂，无条件地相信我支持我的选择，仅有的要求只是顺利毕业和过的快乐。最感谢母亲无限的包容和谅解年轻冲动的我没有常常陪在身边。感谢自己选择了编程便意志弥坚，从敲第一行有自己的思想的代码开始已经两年了，这些日子有苦有泪，在女生们都恨不能远离辐射的时候，可能为修一个 Bug 而连续 10 个小时呆在电脑前，祭奠那些没有疯玩的青春。非常感谢 10 级信息管理与信息系统 3 班，这是一个无比温暖的班级，班委像家人般维系着班级的日常生活，有你们真好。还有谢谢曾包容过我 HCI，在这里开始了我的第一个 HelloWork，谢谢这里的每一个学生“领导”，带领我们比较弱的同学向技术攻关。最后要感谢的是 935 工作室，感谢邝颖杰老师像母亲一样的慈爱与包容，给了我们 14 个 10 级同学无限的归属感。

华南农业大学
本科生毕业论文成绩评定表

学号	201030560306	姓名	何梓	专业	信息管理与信息系统
毕业论文题目	消息应答服务器框架设计与开发				

指导教师评语

成绩（百分制）：_____ 指导教师签名： 年 月 日

评阅人评语及成绩评定						
	成绩评定标准	评分项目			分值	得分
		选题质量 20%	1	专业培养目标	5	
			2	课题难易度与工作量	10	
			3	理论意义或生产实践意义	5	
		能力水平 40%	4	查阅文献资料与综合运用知识能力	10	
			5	研究方案的设计能力	10	
			6	研究方法和手段的运用能力	10	
			7	外文应用能力	10	
		成果质量 40%	8	写作水平与写作规范	20	
			9	研究结果的理论或实际应用价值	20	
	评阅人评语：					
	成绩（百分制）：_____ 评阅人签名： 年 月 日					

续上表：

<table>
<tr><td rowspan="8">答辩委员会意见与成绩评定</td><td>评价项目</td><td>具体要求（A 级标准）</td><td>最高分</td><td colspan="5">评分</td></tr>
<tr><td></td><td></td><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr>
<tr><td>论文质量</td><td>论文（设计）结构严谨，逻辑性强；有一定的学术价值或实用价值；文字表达准确流畅；论文格式规范；图表（或图纸）规范、符合要求。</td><td>60</td><td>55-60</td><td>49-54</td><td>43-48</td><td>37-42</td><td>≤36</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>论文报告、讲解</td><td>思路清晰；概念清楚，重点（创新点）突出；语言表达准确；报告时间、节奏掌握好。</td><td>20</td><td>19-20</td><td>17-18</td><td>15-16</td><td>13-14</td><td>≤12</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>答辩情况</td><td>答辩态度认真，能准确回答问题</td><td>20</td><td>19-20</td><td>17-18</td><td>15-16</td><td>13-14</td><td>≤12</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="8">是否同意通过论文答辩（打√）

1.同意

2.不同意

成绩（百分制）：_____ 答辩委员会主席（签名）：

年月日</td></tr>
<tr><td>成绩总评</td><td colspan="8">论文总评分数：_____

论文成绩总评等级：_____学院盖章：

年月日</td></tr>
</table>

注：1、论文成绩评定等级：参考评阅人的评阅、指导教师评阅情况，结合答辩情况，建议按指导教师评分、评阅人评分、答辩评分为 4∶3∶3 的比例评定论文总成绩分数，然后按优（90－100 分）、良（80－89 分）、中（70－79 分）、及格（60－69 分）、不及格（<60 分）给出成绩等级。**2**、论文成绩以分数的形式登记到教务管理系统。