**东 莞 理 工 学 院**

**本 科 毕 业 设 计**

**毕业设计题目：智障人员跟踪救助系统**

**—服务器端设计与实现**

**学生姓名：张作强**

**学 号：200741402138**

**系 别：计算机学院**

**专业班级：计算机科学与技术1班**

**指导教师姓名及职称： 高晓雷 副教授**

**起止时间：2011年3月—— 2011年6月**

**摘 要**

本文对智障人员跟踪救助系统(Mentally retarded persons Tracking rescue system，MRTRS)的工作原理进行了研究，论述了PC服务端、客户端的系统架构，论证了采用基于NIO的高并发网络服务器模型的必要性，重点介绍NIO服务器的分析、设计与实现，主要技术包括：Java NIO编程、数据库（DB2）技术等；经过对系统的测试，该服务器能为客户端提供很好的请求服务，并能安全地协调与客户端的通信工作。另外，本项目使用SVN作版本控制，项目代码寄存于谷歌代码服务器。

**关键字** 智障人员跟踪救助系统 NIO 选择器 通道 非阻塞 SelectionKey

数据库 DB2 用户自定义函数 Google Map API ActionScript 3.0

事件 侦听 接口 设计模式 SVN

**Abstract**

In this article, studying the working principle of mentally retarded persons tracking rescue system, and discussing the system architecture including the server, client, demonstrating the necessity of the server program java nio-based high concurrent network server model, focusing on introducing the analysis, design and implementation of the nio-based server, the main technologies include: java nio programming, database technology(DB2); After testing the system, the user nio-based server can provide good service to the client’s request and can coordinate securely with the communication work of client. In addition, the project use SVN for version control, project code on Google code hosting server.

**Key Words** mentally retarded persons tracking rescue system NIO

selector channel non-blocking SelectionKey database

DB2 udf(user-defined function) Google Map API ActionScript 3.0

Event listener Interface Design Patterns SVN

目录

[1 引言 6](#_Toc293478734)

[**1.1 研究背景 6**](#_Toc293478735)

[**1.2 目的和意义 6**](#_Toc293478736)

[**1.3 课题来源与主要研究内容 6**](#_Toc293478737)

[2 方案论证 7](#_Toc293478738)

[**2.1 基于Java NIO模型的选择 7**](#_Toc293478739)

[**2.2 自定义NIO框架的思考 9**](#_Toc293478740)

[**2.3 数据库的选择 24**](#_Toc293478741)

[3 软件工程方法 25](#_Toc293478742)

[**3.1 软件开发模型--瀑布模型 25**](#_Toc293478743)

[4 服务器需求分析 26](#_Toc293478744)

[**4.1 任务概述 26**](#_Toc293478745)

[**4.2 需求规定 26**](#_Toc293478746)

[**4.3 运行环境 26**](#_Toc293478747)

[5 服务器概要设计 26](#_Toc293478748)

[**5.1 总体设计 26**](#_Toc293478749)

[**5.2 接口设计 27**](#_Toc293478750)

[**5.3 数据传输安全性 27**](#_Toc293478751)

[**5.4 数据传输协议 27**](#_Toc293478752)

[**5.5 系统出错处理设计 27**](#_Toc293478753)

[**5.6 国际化和版本控制 27**](#_Toc293478754)

[6 服务器详细设计 27](#_Toc293478755)

[**6.1 程序系统的结构 27**](#_Toc293478756)

[**6.2 程序设计说明 27**](#_Toc293478757)

[7 数据库设计 27](#_Toc293478758)

[**7.1 外部设计 27**](#_Toc293478759)

[**7.2 结构设计 28**](#_Toc293478760)

[**7.3 运用设计 28**](#_Toc293478761)

[8 测试结果分析 28](#_Toc293478762)

[9 总结 28](#_Toc293478763)

[参考文献 28](#_Toc293478764)

[致谢 28](#_Toc293478765)

1. 引言
   1. 研究背景

据统计，中国从2001年到2020年是快速老龄化阶段。这一阶段，中国将平均每年新增596万老年人口，年均增长速度达到3.28%，到2020年，老年人口将达到2.48亿，老龄化水平将达到17.17%，其中，80岁及以上老年人口将达到3067万人，占老年人口的12.37%【引用2008年中国人口数据】，根据最新数据（2005年）统计，在我国现有人口中，因各种先、后天性疾病引起的智残、智障人数已达1300万左右。国家“九五”攻关项目调查了北京、上海、西安、成都、广州、沈阳等39个区县55岁以上老人42890名，共筛查出各类痴呆症患者1258例，痴呆症总患病率为2.9%，在65岁以上人口中痴呆症总患病率为8.22%。估计全国共有各类痴呆症患者500万，也就是说，中国痴呆症患者占世界总病例的1/4以上，已经成为威胁老年人健康的重要疾病。而且在我国老年人中，痴呆症的发病率仍在逐年上升，老年因病走失事件时有发生。老人走失现象，已成为我国——这个进入老龄化国家的一个不可忽视的问题。

据香港中文大学2007年痴呆症患者在社区走失之状况调查统计分析，在老人发生走失事件之后，约四分之一(26.9%)的家庭表示会禁止老人单独外出以预防再次走失，同时亦有四分之一(26.0%)人表示会为老人准备一个备有个人资料的名牌；约一成人(11.5%)会因此而聘请家庭佣工照顾老人。走失事件发生以后，超过四成(40.5%)家人表示高度担心老人再次走失，而情绪亦因为担心而受到显着的困扰。【[引用香港中文大学2007年调查报告](http://www.cpr.cuhk.edu.hk/sc/press_detail.php?id=547)】

智障人员跟踪救助系统(Mentally retarded persons Tracking rescue system，MRTRS)正是运用当前发展迅速的地理位置服务和网络技术，去协助及监察智障人员的活动，从而保障智障人员的人身财产安全，在走失的情况下发挥及时救助的作用。通过具体应用，体现出了很强的实用性和社会价值。

* 1. 目的和意义

智障人员跟踪救助系统主要是为了解决预防智障人员走失、智障人员走失后及时救助的问题。当智障人员超出用户预设的位置范围时，服务器及时将智障人员的地理位置通报给用户，从而保证用户能够及时监护智障人员，保护智障人员的人身安全。

* 1. 课题来源与主要研究内容

本课题是由指导老师提出，该课题所实现的系统有较强的实用性，并能解决现实的问题，有十分重要的意义。

本软件系统包括：手机客户端程序、子网号与地理坐标程序、服务器端程序。笔者主要负责服务器端的设计与实现工作。

1. 方案论证
   1. 基于Java NIO模型的选择

服务器在合理的时间之内处理大量客户机请求的能力取决于服务器使用 I/O 流的效率。同时为成百上千个客户机提供服务的服务器必须能够并发地使用 I/O 服务。Java 平台直到 JDK 1.4才支持非阻塞 I/O 调用。用 Java 语言写的服务器，由于其线程与客户机之比几乎是一比一，因而易于受到大量线程开销的影响，其结果是既导致了性能问题又缺乏可伸缩性。

为了解决这个问题，Java 平台的最新发行版引入了一组新的类。java.nio 包充满了解决线程开销问题的技巧，包中最重要的是新的 SelectableChannel 类和 Selector 类。 通道（channel）是客户机和服务器之间的一种通信方式。选择器（selector）与 Windows 消息循环类似，它从不同客户机捕获各种事件并将它们分派到相应的事件处理程序。这两个类通过协同工作，从而为 Java 平台创建非阻塞 I/O 机制。

首先，对比Java NIO之前的服务器-套接字（server-socket）程序：在 ServerSocket 类的生存期中，其重要功能如下：

\* 接受传入连接

\* 从客户机读取请求

\* 为请求提供服务

分析以上的每一个步骤，通过代码片段来说明。首先，我们创建一个新的 ServerSocket：

ServerSocket s = new ServerSocket();

接着，我们要接受传入调用。这里，调用 accept() 应该可以完成任务，但其中有个小陷阱得当心：

Socket conn = s.accept();

对 accept() 的调用将一直阻塞，直到服务器套接字接受了一个请求连接的客户机请求。一旦建立了连接，服务器就使用 LineNumberReader 读取客户机请求。因为 LineNumberReader 要到缓冲区满时才成批地读取数据，所以这个调用在读时阻塞。下面的片段显示了工作中的 LineNumberReader （阻塞等待）。

InputStream in = conn.getInputStream();

InputStreamReader rdr = new InputStreamReader(in);

LineNumberReader lnr = new LineNumberReader(rdr);

Request req = new Request();

while (!req.isComplete()){

String s = lnr.readLine();

req.addLine(s);

}

InputStream.read()是另一种读取数据的方式。不幸的是，read 方法也要一直阻塞到数据可用为止，write方法也一样。

图 1 描绘了服务器的典型工作过程。黑体线表示处于阻塞的操作。

图 1. 典型的工作中的服务器

Server Ready

Accept Connection

Read Client Request

Send O/P to client

Close Connection

Blocking Calls

在 JDK 1.4 之前，自由地使用线程是处理阻塞问题最典型的办法。但这个解决办法会产生它自己的问题 ― 即线程开销，线程开销同时影响性能和可伸缩性。Java NIO包的引入改变了这一现状。

在了解Java NIO工作原理之前，先来了解什么是反应器模式（Reactor pattern）：当一个主体发生改变时，所有依属体都得到通知。其中反应器模式与多个事件源关联。

反应器模式的核心功能：

\* 将事件多路分用

\* 将事件分派到各自相应的事件处理程序

NIO 设计背后的基石是反应器设计模式。分布式系统中的服务器应用程序必须处理多个向它们发送服务请求的客户机。然而，在调用特定的服务之前，服务器应用程序必须将每个传入请求多路分用并分派到各自相应的服务提供者。反应器模式正好适用于这一功能。它允许事件驱动应用程序将服务请求多路分用并进行分派，然后，这些服务请求被并发地从一个或多个客户机传送到应用程序。

NIO 的非阻塞 I/O 机制是围绕***选择器***和***通道***构建的。Channel类表示服务器和客户机之间的一种通信机制。与反应器模式一致，Selector类是Channel的多路复用器。Selector 类将传入客户机请求多路分用并将它们分派到各自的请求处理程序。

NIO实现的主要优点有两方面：线程不再在读或写时阻塞，以及 Selector 能够处理多个连接，从而大幅降低了服务器应用程序开销。

在下面NIO框架的设计中将详细讲解NIO的实现原理和工作方式。

* 1. 自定义NIO框架的思考

JDK1.4提供的无阻塞 I/O（NIO）有效解决了多线程服务器存在的线程开销问题，但在使用上略显得复杂一些。许多基于NIO的多线程服务器程序往往直接基于选择器（Selector）的反应器模式（Reactor pattern）实现。这种简单的事件机制对于较复杂的服务器应用，显然缺乏扩展性和可维护性，而且缺乏直观清晰的结构层次。采用ActionScript3.0语言中基于事件侦听的编程思想，设计一个基于事件侦听的 NIO 多线程服务器框架，提高程序代码的简洁性、可靠性、扩展性和高性能。

NIO有效解决了原有流式 IO 存在的线程开销的问题，在NIO中使用多线程，主要目的不是为了应对每个客户端请求而分配独立的服务线程，而是通过多线程充分使用多个 CPU 的处理能力和处理中的等待时间，达到提高服务能力的目的。多线程的引入，容易为本来就略显复杂的 NIO 代码进一步降低可读性和可维护性。所以引入良好的设计模型，将不仅带来高性能、高可靠的代码，也将带来一个惬意的开发过程。

NIO 的选择器采用了多路复用（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。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，所以在读、写操作中引入多线程，对性能提高明显，而且可以提高客户端的感知服务质量。所以本服务器框架将主要通过使用读、写线程池来提高与客户端的数据交换能力。

如下图所示，服务端接受客户端请求后，控制线程将该请求的读通道交给读线程池，由读线程池分配线程完成对客户端数据的读取操作；当读线程完成读操作后，将数据返回控制线程，进行服务端的业务处理；完成业务处理后，将需回应给客户端的数据和写通道提交给写线程池，由写线程完成向客户端发送回应数据的操作。

NIO 多线程服务器框架模型



同时整个服务端的流程处理，建立于事件机制上。在 [接受连接－＞读－＞业务处理－＞写 －＞关闭连接 ]这个过程中，触发器将触发相应事件，由事件处理器对相应事件分别响应，完成服务器端的业务处理。

下面详细分析这个框架的各个组成部分，以及NIO服务器的实现过程。

首先，介绍该框架的事件侦听模型：下面是自定义的一些基本事件：

（1）doAccept：当服务端收到客户端连接请求时，触发该事件。通过该事件可以知道有新的客户端进入。该事件可用来控制服务端的负载。例如，服务器可设定同时只为一定数量客户端提供服务，当同时请求数超出数量时，可在响应该事件时直接抛出异常，以拒绝新的连接。

（2）doAccepted：当客户端请求被服务器接受后触发该事件。该事件表明一个新的客户端与服务器正式建立连接。

（3）doRead：当客户端发来数据，并已被服务器控制线程正确读取时，触发该事件。该事件通知各事件处理器可以对客户端发来的数据进行实际处理了。需要注意的是，在本框架中，客户端的数据读取是由控制线程交由读线程完成的，事件处理器不需要在该事件中进行专门的读操作，而只需将控制线程传来的数据进行直接处理即可。

（4）doWrite：当客户端可以开始接受服务端发送数据时触发该事件，通过该事件，可以向客户端发送回应数据。在本模型中，事件处理器只需要在该事件中设置。

（5）doClosed：当客户端与服务器断开连接时触发该事件。

（6）doError：当客户端与服务器从连接开始到最后断开连接期间发生错误时触发该事件。通过该事件可以知道有什么错误发生，并根据需求处理此错误。

受ActionScript3.0中事件侦听的编程思想启发，在本框架核心中实现了基于事件侦听的处理中心，其他业务请求均当作触发事件的源，只要定义了该事件的处理器（Handler），服务器就可以处理此请求。并且事件采用广播方式，也就是所有注册的事件处理器都能获得事件通知。这样可以将不同性质的业务处理，分别用不同的处理器实现，使每个处理器的业务功能尽可能单一。

如下图：整个框架的事件侦听模型由监听器、事件适配器、事件触发器、事件处理器组成。

**⑴ 监听器（Ilistener）：**这是一个事件接口，定义需监听的服务器事件，如果您需要定义更多的事件，可在这里进行扩展。

public interface Ilistener {

public void doError(String error);

public void doAccept() throws Exception;

public void doAccepted(Request request) throws Exception;

public void doRead(Request request) throws Exception;

public void doWrite(Request request, Response response) throws Exception;

public void doClosed(Request request) throws Exception;

}

Notifier

listeners

Notifier()

getNotifier()

addListener()

fireDoAccept()

fireDoAccepted()

fireDoRead()

fireDoWrite()

fireDoClosed()

fireDoError()

//注册事件处理器

AddHandlerListener()

Handler

IListener

doError()

doAccept()

doAccepted()

doRead()

doWrite()

doClosed()

ListenAdapter

instance

**⑵ 事件适配器（ListenAdapter）：**对Ilistener接口实现一个适配器 (ListenAdapter)，这样的好处是最终的事件处理器可以只处理所关心的事件。

public abstract class ListenAdapter implements IListener {

public ListenAdapter() {}

public void doError(String error) {}

public void doAccept() throws Exception {}

public void doAccepted(Request request) throws Exception {}

public void doRead(Request request) throws Exception {}

public void doWrite(Request request, Response response) throws Exception {}

public void doClosed(Request request) throws Exception{}

}

**⑶ 事件触发器（Notifier）：**用于在适当的时候通过触发服务器事件，通知注册的事件处理器对事件做出响应。触发器以Singleton模式实现，统一控制整个服务器端的事件，避免造成混乱。

public class Notifier {

private static ArrayList<IListener> listeners = null;

private static Notifier instance = null;

private Notifier() {

listeners = new ArrayList<IListener>();

}

public static synchronized Notifier getNotifier() {

if (instance == null) {

instance = new Notifier();

return instance;

}else{

return instance;

}

}

public void addListener(IListener l) {

synchronized (listeners) {

if (!listeners.contains(l))

listeners.add(l);

}

}

public void fireDoAccept() throws Exception {

for (int i = listeners.size() - 1; i >= 0; i--)

( (IListener) listeners.get(i)).doAccept();

}

....// other fire method

}

**⑷ 事件处理器（Handler）：**继承事件适配器，对感兴趣的事件进行响应处理，实现业务处理。以下是一个简单的事件处理器实现，它响应 onRead 事件，在终端打印出从客户端读取的数据。

public class TimeHandler extends ListenAdapter {

public TimeHandler() {

}

public void doWrite(Request request, Response response) throws Exception {

int command = request.getCommand();

String time = null;

Date date = new Date();

if (command == GlobalConst.C\_TEST) {

DateFormat enDate = DateFormat.getDateTimeInstance(DateFormat.FULL,

DateFormat.FULL, Locale.US);

time = enDate.format(date);

BaseCmd cmd = new BaseCmd(GlobalConst.S\_TEST);

cmd.writeString(time);

response.send(cmd);

}

}

}

**⑸ 事件处理器的注册：**为了能让事件处理器获得服务线程的事件通知，事件处理器需在触发器中注册。

public final class HandlerManager{

public static void AddHandlerListener(){

Notifier notifier = Notifier.getNotifier();

TimeHandler timer = new TimeHandler();

notifier.addListener(timer);

}

}

介绍完了事件侦听模型的实现，下面介绍的是该NIO多线程服务器框架的线程模型:

该NIO多线程服务器主要由主控服务线程、读线程和写线程组成。

线程模型见下图：

1: 连接（doAccept）

2: 接受连接（doAccepted）

: NIOServerManager

:Reader

doRead

4: 完成读操作返回

3: 加入读线程池

:Writer

doWrite doClosed

5: 加入写线程池

6: 完成并关闭连接

**⑴ 主控服务线程（NIOServerManager）**：主控线程将创建读、写线程池，实现监听、接受客户端请求，同时将读、写通道提交由相应的读线程（Reader）和写服务线程 (Writer），由读写线程分别完成对客户端数据的读取和对客户端的回应操作。

public class NIOServerManager implements Runnable {

/\*\*回应池\*\*/

private static List<SelectionKey> wpool = new LinkedList<SelectionKey>();

public static Selector selector;

protected static Notifier notifier;

private int port;

/\*\*创建主控服务线程\*\*/

public NIOServerManager(int port) throws Exception {

this.port = port;

// 获取事件触发器

notifier = Notifier.getNotifier();

// 创建读/写线程池

for (int i = 1; i <= ServerConfig.MAX\_THREADS; i++) {

Thread r = new Reader();

Thread w = new Writer();

r.start();

w.start();

}

selector = this.getSelector(port);

}

public void run() {

listen();

}

/\*\*获取Selector，创建无阻塞网络套接\*\*/

protected Selector getSelector(int port) throws IOException {

ServerSocketChannel server = ServerSocketChannel.open();

Selector selector = Selector.open();

server.socket().bind(new InetSocketAddress(port));

server.configureBlocking(false);

server.register(selector, SelectionKey.OP\_ACCEPT);

return selector;

}

/\*\*监听端口\*\*/

public void listen() {

try {

while (true) {

int num = selector.select();

if (num > 0) {

// 获得就绪信道的键迭代器

Iterator<SelectionKey> it = selector.selectedKeys().iterator();

while (it.hasNext()) {

SelectionKey key = (SelectionKey) it.next();

//处理I/O操作

processIO(key);

it.remove();

}

} else {

// 添加新的通道注册

addRegister();

}

}

} catch (Exception e) {

if (Debug.printException) {

e.printStackTrace();

}

if (Debug.printFireEvent) {

notifier.fireDoError(LangUtil.get("10010") + e.getMessage());

}

}

}

/\*\*处理I/O事件\*\*/

protected static void processIO(SelectionKey key) throws Exception {

if(key.isAcceptable()){

// 接收新的连接请求

ServerSocketChannel ssc = (ServerSocketChannel) key.channel();

notifier.fireDoAccept();

SocketChannel sc = ssc.accept();

// 设置非阻塞模式

sc.configureBlocking(false);

// 触发接受连接事件

Request request = new Request(sc);

notifier.fireDoAccepted(request);

// 注册读操作,以进行下一步的读操作

sc.register(selector, SelectionKey.OP\_READ, request);

}else if (key.isReadable()) {

// 提交读服务线程读取客户端数据

Reader.processRequest(key);

//将本socket的事件在选择器中删除

key.cancel();

}else if (key.isWritable()) {

// 提交写服务线程向客户端发送回应数据

Writer.processRequest(key);

key.cancel();

}

}

/\*\*添加新的通道注册\*\*/

private void addRegister() {

synchronized (wpool) {

while (!wpool.isEmpty()) {

SelectionKey key = (SelectionKey) wpool.remove(0);

SocketChannel schannel = (SocketChannel)key.channel();

try {

schannel.register(selector, SelectionKey.OP\_WRITE, key.attachment());

}

catch (Exception e) {

try {

schannel.finishConnect();

schannel.close();

schannel.socket().close();

notifier.fireDoClosed((Request)key.attachment());

}

catch (Exception e1) {}

if (Debug.printException) {

e.printStackTrace();

}

notifier.fireDoError(LangUtil.get("10013") + e.getMessage());

}

}

}

}

/\*\*提交新的客户端写请求于主服务线程的回应池中\*\*/

public static void processWriteRequest(SelectionKey key) {

synchronized (wpool) {

wpool.add(wpool.size(), key);

wpool.notifyAll();

}

// 解除selector的阻塞状态，以便注册新的通道

selector.wakeup();

}

}

**⑵ 读线程（Reader）：**使用线程池技术，通过多个线程读取客户端数据，以充分利用网络数据传输的时间，提高读取效率。

public class Reader extends Thread {

private static List<SelectionKey> pool = new LinkedList<SelectionKey>();

protected static ByteBuffer buffer = null;

private int command = 0;

private static Notifier notifier = Notifier.getNotifier();

public Reader() {

}

public void run() {

while (true) {

try {

SelectionKey key;

synchronized (pool) {

while (pool.isEmpty()) {

pool.wait();

}

key = (SelectionKey) pool.remove(0);

}

// 读取数据

read(key);

}catch (Exception e) {}

}

}

/\*\*处理连接数据读取\*\*/

public void read(SelectionKey key) {

try {

Packet clientData = readRequest(key);

if (clientData != null) {

// 输出客户端所有的请求内容

clientData.printInfo(clientData.array());

// 读取消息号，分发处理写逻辑

command = clientData.readInt();

BaseMsg msg = new BaseMsg(command, clientData.readInt(), clientData.readInt());

// 设置key.attachment

Request request = (Request)key.attachment();

request.setPacket(clientData);

request.setCommand(command);

request.setMsg(msg);

// 触发读处理

notifier.fireDoRead(request);

// 提交主控线程进行写处理

NIOServerManager.processWriteRequest(key);

}

}catch (Exception e) {}

}

/\*\*读取客户端发出请求数据\*\*/

public static Packet readRequest(SelectionKey key){

SocketChannel sc = (SocketChannel) key.channel();

// buffer的长度

int count = 0;

Packet packet = null;

buffer = ByteBuffer.allocate(ServerConfig.BUFFER\_SIZE);

buffer.clear();

// 读取客户端消息长度

try {

count = sc.read(buffer);

} catch (IOException e) {}

notifier.fireDoError(LangUtil.get("10016") + e.getMessage());

key.cancel();

}

if (count > 0) {

buffer.flip();

packet = new Packet(buffer);

}

buffer.clear();

return packet;

}

/\*\*处理客户请求,管理用户的连接池,并唤醒队列中的线程进行处理\*\*/

public static void processRequest(SelectionKey key) {

synchronized (pool) {

pool.add(pool.size(), key);

pool.notifyAll();

}

}

/\*\*返回读线程池大小\*\*/

public static int getReadPoolSize(){

return pool.size();

}

}

**⑶ 写线程（Writer）：**和读操作一样，使用线程池，负责将服务器端的数据发送回客户端。

public final class Writer extends Thread {

private static List<SelectionKey> pool = new LinkedList<SelectionKey>();

private static Notifier notifier = Notifier.getNotifier();

public Writer() {

}

public void run() {

while (true) {

try {

SelectionKey key = null;

synchronized (pool) {

while (pool.isEmpty()) {

pool.wait();

}

key = (SelectionKey) pool.remove(0);

}

// 处理写事件

write(key);

}catch (Exception e) {}

}

}

/\*\*处理向客户发送数据\*\*/

public void write(SelectionKey key) {

SocketChannel sc = (SocketChannel) key.channel();

try {

Response response = new Response(key);

// 触发doWrite事件

notifier.fireDoWrite((Request)key.attachment(), response);

notifier.fireDoClosed((Request)key.attachment());

MoreUtils.trace(key.attachment(), Debug.printTestInfo);

// 一次请求处理完毕，关闭连接（短连接）

sc.finishConnect();

sc.socket().close();

sc.close();

}catch (Exception e) {}

}

/\*\*处理客户请求,管理用户的联结池,并唤醒队列中的线程进行处理\*\*/

public static void processRequest(SelectionKey key) {

synchronized (pool) {

pool.add(pool.size(), key);

pool.notifyAll();

}

}

/\*\*返回回应池的大小\*\*/

public static int getWritePoolSize(){

return pool.size();

}

}

* 1. 数据库的选择

DB2数据库是IBM公司研制的一种关系型数据库系统。DB2主要应用于大型应用系统，具有较好的可伸缩性，可支持从大型机到单用户环境，应用于OS/2、Windows等平台下。 DB2提供了高层次的数据利用性、完整性、安全性、可恢复性，以及小规模到大规模应用程序的执行能力，具有与平台无关的基本功能和SQL命令。DB2采用了数据分级技术，能够使大型机数据很方便地下载到LAN数据库服务器，使得客户机/服务器用户和基于LAN的应用程序可以访问大型机数据，并使数据库本地化及远程连接透明化。它以拥有一个非常完备的查询优化器而著称，其外部连接改善了查询性能，并支持多任务并行查询。 DB2具有很好的网络支持能力，每个子系统可以连接十几万个分布式用户，可同时激活上千个活动线程，对大型分布式应用系统尤为适用。

本系统服务器的开发选用了Java语言，DB2可以很好的支持Java的自定义函数，于是在使用DB2的触发器时可以方便的调用Java函数实现预定的功能。

1. 软件工程方法
   1. 软件开发模型--瀑布模型

本系统的开发模型是瀑布模型。瀑布模型（Waterfall Model）又称为软件生命周期瀑布模型。瀑布模型核心思想是按工序将问题化简，将功能的实现与设计分开，便于分工协作，即采用结构化的分析与设计方法将逻辑实现与物理实现分开。将软件生命周期划分为制定计划、需求分析、软件设计、程序编写、软件测试和运行维护等六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。从本质来讲，它是一个软件开发架构，开发过程是通过一系列阶段顺序展开的，从系统需求分析开始直到产品发布和维护，每个阶段都会产生循环反馈，因此，如果有信息未被覆盖或者发现了问题，那么最好 “返回”上一个阶段并进行适当的修改，开发进程从一个阶段“流动”到下一个阶段，这也是瀑布开发名称的由来。

当采用瀑布模型进行开发组织时，应制定软件开发规范或开发标准。其中要明确规定各个开发阶段应交付的产品， 这就为严格控制软件开发项目的进度，最终按时交付产品以及保证软件产品质量创造了有利条件。

为了保障软件开发的正确性，每一阶段任务完成后，都必须对它的阶段性产品进行评审，确认之后再转入下一阶段的工作。因此，它是一种以文档作为驱动的模式。例如在评审过程中，如果发现错误和疏漏，应该反馈到前面的有关阶段修正错误、弥补疏漏，然后再重复前面的工作，直至某一阶段通过评审后再进入下一阶段。

瀑布模型在支持结构化软件开发、控制软件开发的复杂性、促进软件开发工程化等方面具有显著作用。它提供了软件开发的基本框架，有利于大型软件开发过程中人员的组织、管理，有利于软件开发方法和工具的研究与使用，提高了大型软件项目开发的质量与效率。瀑布模型软件过程的基本框架如图3.1所示。

问题定义

可行性研究

需求分析

总体设计

详细设计

编码

测试

维护

问题计划

开发时期

运行时期

图3.1 带有反馈的瀑布模型

Fig.3.1 Waterfall model with feedback

1. 服务器需求分析
   1. 任务概述
   2. 需求规定
   3. 运行环境
2. 服务器概要设计
   1. 总体设计
   2. 接口设计
   3. 数据传输安全性
   4. 数据传输协议
   5. 系统出错处理设计
   6. 国际化和版本控制
3. 服务器详细设计
   1. 程序系统的结构
   2. 程序设计说明
4. 数据库设计
   1. 外部设计
   2. 结构设计
   3. 运用设计
5. 测试结果分析
6. 总结

参考文献

致谢

NIOServer code rows: 6344