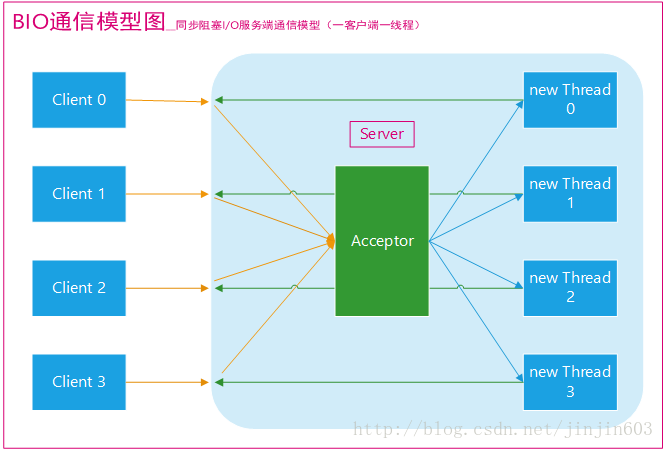
网络编程(3)-BIO,NIO,AIO

本文会从传统的BIO到NIO再到AIO自浅至深介绍，并附上完整的代码讲解。  
下面代码中会使用这样一个例子：客户端发送一段算式的字符串到服务器，服务器计算后返回结果到客户端。  
代码的所有说明，都直接作为注释，嵌入到代码中，看代码时就能更容易理解，代码中会用到一个计算结果的工具类，见文章代码部分。  
相关的基础知识文章推荐：  
Linux 网络 I/O 模型简介（图文）  
Java 并发（多线程）  
1、BIO编程  
1.1、传统的BIO编程  
网络编程的基本模型是C/S模型，即两个进程间的通信。  
服务端提供IP和监听端口，客户端通过连接操作想服务端监听的地址发起连接请求，通过三次握手连接，如果连接成功建立，双方就可以通过套接字进行通信。  
传统的同步阻塞模型开发中，ServerSocket负责绑定IP地址，启动监听端口；Socket负责发起连接操作。连接成功后，双方通过输入和输出流进行同步阻塞式通信。  
简单的描述一下BIO的服务端通信模型：采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理没处理完成后，通过输出流返回应答给客户端，线程销毁。即典型的一请求一应答通宵模型。  
传统BIO通信模型图：  
  
该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈1:1的正比关系，Java中的线程也是比较宝贵的系统资源，线程数量快速膨胀后，系统的性能将急剧下降，随着访问量的继续增大，系统最终就死-掉-了。  
同步阻塞式I/O创建的Server源码：

package com.anxpp.io.calculator.bio;

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

/\*\*

\* BIO服务端源码

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public final class ServerNormal {

*//默认的端口号*

private static int DEFAULT\_PORT = 12345;

*//单例的ServerSocket*

private static ServerSocket server;

*//根据传入参数设置监听端口，如果没有参数调用以下方法并使用默认值*

public static void start() throws IOException{

*//使用默认值*

start(DEFAULT\_PORT);

}

*//这个方法不会被大量并发访问，不太需要考虑效率，直接进行方法同步就行了*

public synchronized static void start(int port) throws IOException{

if(server != null) return;

try{

*//通过构造函数创建ServerSocket*

*//如果端口合法且空闲，服务端就监听成功*

server = new ServerSocket(port);

System.out.println("服务器已启动，端口号：" + port);

*//通过无线循环监听客户端连接*

*//如果没有客户端接入，将阻塞在accept操作上。*

while(true){

Socket socket = server.accept();

*//当有新的客户端接入时，会执行下面的代码*

*//然后创建一个新的线程处理这条Socket链路*

new Thread(new ServerHandler(socket)).start();

}

}finally{

*//一些必要的清理工作*

if(server != null){

System.out.println("服务器已关闭。");

server.close();

server = null;

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46

客户端消息处理线程ServerHandler源码：

package com.anxpp.io.calculator.bio;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import com.anxpp.io.utils.Calculator;

/\*\*

\* 客户端线程

\* @author yangtao\_\_anxpp.com

\* 用于处理一个客户端的Socket链路

\*/

public class ServerHandler implements Runnable{

private Socket socket;

public ServerHandler(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

BufferedReader in = null;

PrintWriter out = null;

try{

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

out = new PrintWriter(socket.getOutputStream(),true);

String expression;

String result;

while(true){

*//通过BufferedReader读取一行*

*//如果已经读到输入流尾部，返回null,退出循环*

*//如果得到非空值，就尝试计算结果并返回*

if((expression = in.readLine())==null) break;

System.out.println("服务器收到消息：" + expression);

try{

result = Calculator.cal(expression).toString();

}catch(Exception e){

result = "计算错误：" + e.getMessage();

}

out.println(result);

}

}catch(Exception e){

e.printStackTrace();

}finally{

*//一些必要的清理工作*

if(in != null){

try {

in.close();

} catch (IOException e) {

e.printStackTrace();

}

in = null;

}

if(out != null){

out.close();

out = null;

}

if(socket != null){

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

socket = null;

}

}

}

}

同步阻塞式I/O创建的Client源码：

package com.anxpp.io.calculator.bio;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

/\*\*

\* 阻塞式I/O创建的客户端

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public class Client {

*//默认的端口号*

private static int DEFAULT\_SERVER\_PORT = 12345;

private static String DEFAULT\_SERVER\_IP = "127.0.0.1";

public static void send(String expression){

send(DEFAULT\_SERVER\_PORT,expression);

}

public static void send(int port,String expression){

System.out.println("算术表达式为：" + expression);

Socket socket = null;

BufferedReader in = null;

PrintWriter out = null;

try{

socket = new Socket(DEFAULT\_SERVER\_IP,port);

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

out = new PrintWriter(socket.getOutputStream(),true);

out.println(expression);

System.out.println("\_\_\_结果为：" + in.readLine());

}catch(Exception e){

e.printStackTrace();

}finally{

*//一下必要的清理工作*

if(in != null){

try {

in.close();

} catch (IOException e) {

e.printStackTrace();

}

in = null;

}

if(out != null){

out.close();

out = null;

}

if(socket != null){

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

socket = null;

}

}

}

}

测试代码，为了方便在控制台看输出结果，放到同一个程序（jvm）中运行：

package com.anxpp.io.calculator.bio;

import java.io.IOException;

import java.util.Random;

/\*\*

\* 测试方法

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public class Test {

*//测试主方法*

public static void main(String[] args) throws InterruptedException {

*//运行服务器*

new Thread(new Runnable() {

@Override

public void run() {

try {

ServerBetter.start();

} catch (IOException e) {

e.printStackTrace();

}

}

}).start();

*//避免客户端先于服务器启动前执行代码*

Thread.sleep(100);

*//运行客户端*

char operators[] = {'+','-','\*','/'};

Random random = new Random(System.currentTimeMillis());

new Thread(new Runnable() {

@SuppressWarnings("static-access")

@Override

public void run() {

while(true){

*//随机产生算术表达式*

String expression = random.nextInt(10)+""+operators[random.nextInt(4)]+(random.nextInt(10)+1);

Client.send(expression);

try {

Thread.currentThread().sleep(random.nextInt(1000));

} catch (InterruptedException e) {

e.printStackTrace();

}

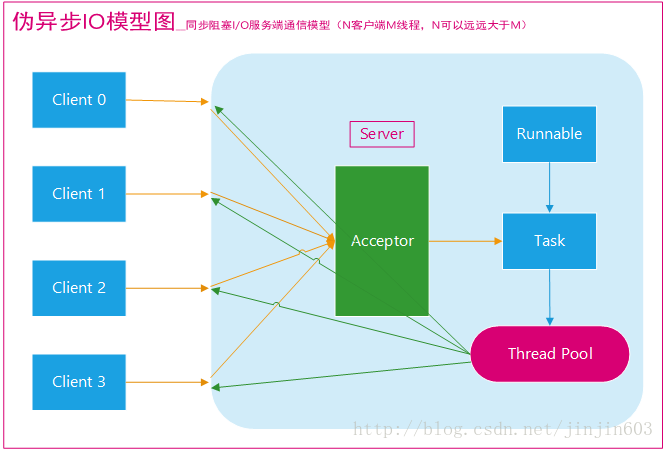
}

}

}).start();

}

}

其中一次的运行结果：  
服务器已启动，端口号：12345  
算术表达式为：4-2  
服务器收到消息：4-2  
*\_*结果为：2  
算术表达式为：5-10  
服务器收到消息：5-10  
*\_*结果为：-5  
算术表达式为：0-9  
服务器收到消息：0-9  
*\_*结果为：-9  
算术表达式为：0+6  
服务器收到消息：0+6  
*\_*结果为：6  
算术表达式为：1/6  
服务器收到消息：1/6  
*\_*结果为：0.16666666666666666  
…  
从以上代码，很容易看出，BIO主要的问题在于每当有一个新的客户端请求接入时，服务端必须创建一个新的线程来处理这条链路，在需要满足高性能、高并发的场景是没法应用的（大量创建新的线程会严重影响服务器性能，甚至罢工）。  
1.2、伪异步I/O编程  
为了改进这种一连接一线程的模型，我们可以使用线程池来管理这些线程（需要了解更多请参考前面提供的文章），实现1个或多个线程处理N个客户端的模型（但是底层还是使用的同步阻塞I/O），通常被称为“伪异步I/O模型“。  
伪异步I/O模型图：  
  
实现很简单，我们只需要将新建线程的地方，交给线程池管理即可，只需要改动刚刚的Server代码即可：

package com.anxpp.io.calculator.bio;

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

/\*\*

\* BIO服务端源码\_\_伪异步I/O

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public final class ServerBetter {

*//默认的端口号*

private static int DEFAULT\_PORT = 12345;

*//单例的ServerSocket*

private static ServerSocket server;

*//线程池 懒汉式的单例*

private static ExecutorService executorService = Executors.newFixedThreadPool(60);

*//根据传入参数设置监听端口，如果没有参数调用以下方法并使用默认值*

public static void start() throws IOException{

*//使用默认值*

start(DEFAULT\_PORT);

}

*//这个方法不会被大量并发访问，不太需要考虑效率，直接进行方法同步就行了*

public synchronized static void start(int port) throws IOException{

if(server != null) return;

try{

*//通过构造函数创建ServerSocket*

*//如果端口合法且空闲，服务端就监听成功*

server = new ServerSocket(port);

System.out.println("服务器已启动，端口号：" + port);

*//通过无线循环监听客户端连接*

*//如果没有客户端接入，将阻塞在accept操作上。*

while(true){

Socket socket = server.accept();

*//当有新的客户端接入时，会执行下面的代码*

*//然后创建一个新的线程处理这条Socket链路*

executorService.execute(new ServerHandler(socket));

}

}finally{

*//一些必要的清理工作*

if(server != null){

System.out.println("服务器已关闭。");

server.close();

server = null;

}

}

}

}

测试运行结果是一样的。  
我们知道，如果使用CachedThreadPool线程池（不限制线程数量，如果不清楚请参考文首提供的文章），其实除了能自动帮我们管理线程（复用），看起来也就像是1:1的客户端：线程数模型，而使用FixedThreadPool我们就有效的控制了线程的最大数量，保证了系统有限的资源的控制，实现了N:M的伪异步I/O模型。  
但是，正因为限制了线程数量，如果发生大量并发请求，超过最大数量的线程就只能等待，直到线程池中的有空闲的线程可以被复用。而对Socket的输入流就行读取时，会一直阻塞，直到发生：  
有数据可读  
可用数据以及读取完毕  
发生空指针或I/O异常  
所以在读取数据较慢时（比如数据量大、网络传输慢等），大量并发的情况下，其他接入的消息，只能一直等待，这就是最大的弊端。  
而后面即将介绍的NIO，就能解决这个难题。  
2、NIO 编程  
JDK 1.4中的java.nio.\*包中引入新的Java I/O库，其目的是提高速度。实际上，“旧”的I/O包已经使用NIO重新实现过，即使我们不显式的使用NIO编程，也能从中受益。速度的提高在文件I/O和网络I/O中都可能会发生，但本文只讨论后者。  
2.1、简介  
NIO我们一般认为是New I/O（也是官方的叫法），因为它是相对于老的I/O类库新增的（其实在JDK 1.4中就已经被引入了，但这个名词还会继续用很久，即使它们在现在看来已经是“旧”的了，所以也提示我们在命名时，需要好好考虑），做了很大的改变。但民间跟多人称之为Non-block I/O，即非阻塞I/O，因为这样叫，更能体现它的特点。而下文中的NIO，不是指整个新的I/O库，而是非阻塞I/O。  
NIO提供了与传统BIO模型中的Socket和ServerSocket相对应的SocketChannel和ServerSocketChannel两种不同的套接字通道实现。  
新增的着两种通道都支持阻塞和非阻塞两种模式。  
阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。  
对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用NIO的非阻塞模式来开发。  
下面会先对基础知识进行介绍。  
2.2、缓冲区 Buffer  
Buffer是一个对象，包含一些要写入或者读出的数据。  
在NIO库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，也是写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。  
缓冲区实际上是一个数组，并提供了对数据结构化访问以及维护读写位置等信息。  
具体的缓存区有这些：ByteBuffe、CharBuffer、 ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer。他们实现了相同的接口：Buffer。  
2.3、通道 Channel  
我们对数据的读取和写入要通过Channel，它就像水管一样，是一个通道。通道不同于流的地方就是通道是双向的，可以用于读、写和同时读写操作。  
底层的操作系统的通道一般都是全双工的，所以全双工的Channel比流能更好的映射底层操作系统的API。  
Channel主要分两大类：  
SelectableChannel：用户网络读写  
FileChannel：用于文件操作  
后面代码会涉及的ServerSocketChannel和SocketChannel都是SelectableChannel的子类。  
2.4、多路复用器 Selector  
Selector是Java NIO 编程的基础。  
Selector提供选择已经就绪的任务的能力：Selector会不断轮询注册在其上的Channel，如果某个Channel上面发生读或者写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的I/O操作。  
一个Selector可以同时轮询多个Channel，因为JDK使用了epoll()代替传统的select实现，所以没有最大连接句柄1024/2048的限制。所以，只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端。  
2.5、NIO服务端  
代码比传统的Socket编程看起来要复杂不少。  
直接贴代码吧，以注释的形式给出代码说明。  
NIO创建的Server源码：

package com.anxpp.io.calculator.nio;

public class Server {

private static int DEFAULT\_PORT = 12345;

private static ServerHandle serverHandle;

public static void start(){

start(DEFAULT\_PORT);

}

public static synchronized void start(int port){

if(serverHandle!=null)

serverHandle.stop();

serverHandle = new ServerHandle(port);

new Thread(serverHandle,"Server").start();

}

public static void main(String[] args){

start();

}

}

ServerHandle：

package com.anxpp.io.calculator.nio;

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

import java.util.Iterator;

import java.util.Set;

import com.anxpp.io.utils.Calculator;

/\*\*

\* NIO服务端

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public class ServerHandle implements Runnable{

private Selector selector;

private ServerSocketChannel serverChannel;

private volatile boolean started;

/\*\*

\* 构造方法

\* @param port 指定要监听的端口号

\*/

public ServerHandle(int port) {

try{

*//创建选择器*

selector = Selector.open();

*//打开监听通道*

serverChannel = ServerSocketChannel.open();

*//如果为 true，则此通道将被置于阻塞模式；如果为 false，则此通道将被置于非阻塞模式*

serverChannel.configureBlocking(false);*//开启非阻塞模式*

*//绑定端口 backlog设为1024*

serverChannel.socket().bind(new InetSocketAddress(port),1024);

*//监听客户端连接请求*

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

*//标记服务器已开启*

started = true;

System.out.println("服务器已启动，端口号：" + port);

}catch(IOException e){

e.printStackTrace();

System.exit(1);

}

}

public void stop(){

started = false;

}

@Override

public void run() {

*//循环遍历selector*

while(started){

try{

*//无论是否有读写事件发生，selector每隔1s被唤醒一次*

selector.select(1000);

*//阻塞,只有当至少一个注册的事件发生的时候才会继续.*

*// selector.select();*

Set<SelectionKey> keys = selector.selectedKeys();

Iterator<SelectionKey> it = keys.iterator();

SelectionKey key = null;

while(it.hasNext()){

key = it.next();

it.remove();

try{

handleInput(key);

}catch(Exception e){

if(key != null){

key.cancel();

if(key.channel() != null){

key.channel().close();

}

}

}

}

}catch(Throwable t){

t.printStackTrace();

}

}

*//selector关闭后会自动释放里面管理的资源*

if(selector != null)

try{

selector.close();

}catch (Exception e) {

e.printStackTrace();

}

}

private void handleInput(SelectionKey key) throws IOException{

if(key.isValid()){

*//处理新接入的请求消息*

if(key.isAcceptable()){

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

*//通过ServerSocketChannel的accept创建SocketChannel实例*

*//完成该操作意味着完成TCP三次握手，TCP物理链路正式建立*

SocketChannel sc = ssc.accept();

*//设置为非阻塞的*

sc.configureBlocking(false);

*//注册为读*

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

}

*//读消息*

if(key.isReadable()){

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

*//创建ByteBuffer，并开辟一个1M的缓冲区*

ByteBuffer buffer = ByteBuffer.allocate(1024);

*//读取请求码流，返回读取到的字节数*

int readBytes = sc.read(buffer);

*//读取到字节，对字节进行编解码*

if(readBytes>0){

*//将缓冲区当前的limit设置为position=0，用于后续对缓冲区的读取操作*

buffer.flip();

*//根据缓冲区可读字节数创建字节数组*

byte[] bytes = new byte[buffer.remaining()];

*//将缓冲区可读字节数组复制到新建的数组中*

buffer.get(bytes);

String expression = new String(bytes,"UTF-8");

System.out.println("服务器收到消息：" + expression);

*//处理数据*

String result = null;

try{

result = Calculator.cal(expression).toString();

}catch(Exception e){

result = "计算错误：" + e.getMessage();

}

*//发送应答消息*

doWrite(sc,result);

}

*//没有读取到字节 忽略*

*// else if(readBytes==0);*

*//链路已经关闭，释放资源*

else if(readBytes<0){

key.cancel();

sc.close();

}

}

}

}

*//异步发送应答消息*

private void doWrite(SocketChannel channel,String response) throws IOException{

*//将消息编码为字节数组*

byte[] bytes = response.getBytes();

*//根据数组容量创建ByteBuffer*

ByteBuffer writeBuffer = ByteBuffer.allocate(bytes.length);

*//将字节数组复制到缓冲区*

writeBuffer.put(bytes);

*//flip操作*

writeBuffer.flip();

*//发送缓冲区的字节数组*

channel.write(writeBuffer);

*//\*\*\*\*此处不含处理“写半包”的代码*

}

}

可以看到，创建NIO服务端的主要步骤如下：  
打开ServerSocketChannel，监听客户端连接  
绑定监听端口，设置连接为非阻塞模式  
创建Reactor线程，创建多路复用器并启动线程  
将ServerSocketChannel注册到Reactor线程中的Selector上，监听ACCEPT事件  
Selector轮询准备就绪的key  
Selector监听到新的客户端接入，处理新的接入请求，完成TCP三次握手，简历物理链路  
设置客户端链路为非阻塞模式  
将新接入的客户端连接注册到Reactor线程的Selector上，监听读操作，读取客户端发送的网络消息  
异步读取客户端消息到缓冲区  
对Buffer编解码，处理半包消息，将解码成功的消息封装成Task  
将应答消息编码为Buffer，调用SocketChannel的write将消息异步发送给客户端  
因为应答消息的发送，SocketChannel也是异步非阻塞的，所以不能保证一次能吧需要发送的数据发送完，此时就会出现写半包的问题。我们需要注册写操作，不断轮询Selector将没有发送完的消息发送完毕，然后通过Buffer的hasRemain()方法判断消息是否发送完成。  
2.6、NIO客户端  
还是直接上代码吧，过程也不需要太多解释了，跟服务端代码有点类似。  
Client：

package com.anxpp.io.calculator.nio;

public class Client {

private static String DEFAULT\_HOST = "127.0.0.1";

private static int DEFAULT\_PORT = 12345;

private static ClientHandle clientHandle;

public static void start(){

start(DEFAULT\_HOST,DEFAULT\_PORT);

}

public static synchronized void start(String ip,int port){

if(clientHandle!=null)

clientHandle.stop();

clientHandle = new ClientHandle(ip,port);

new Thread(clientHandle,"Server").start();

}

*//向服务器发送消息*

public static boolean sendMsg(String msg) throws Exception{

if(msg.equals("q")) return false;

clientHandle.sendMsg(msg);

return true;

}

public static void main(String[] args){

start();

}

}

ClientHandle：

package com.anxpp.io.calculator.nio;

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.SocketChannel;

import java.util.Iterator;

import java.util.Set;

/\*\*

\* NIO客户端

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public class ClientHandle implements Runnable{

private String host;

private int port;

private Selector selector;

private SocketChannel socketChannel;

private volatile boolean started;

public ClientHandle(String ip,int port) {

this.host = ip;

this.port = port;

try{

*//创建选择器*

selector = Selector.open();

*//打开监听通道*

socketChannel = SocketChannel.open();

*//如果为 true，则此通道将被置于阻塞模式；如果为 false，则此通道将被置于非阻塞模式*

socketChannel.configureBlocking(false);*//开启非阻塞模式*

started = true;

}catch(IOException e){

e.printStackTrace();

System.exit(1);

}

}

public void stop(){

started = false;

}

@Override

public void run() {

try{

doConnect();

}catch(IOException e){

e.printStackTrace();

System.exit(1);

}

*//循环遍历selector*

while(started){

try{

*//无论是否有读写事件发生，selector每隔1s被唤醒一次*

selector.select(1000);

*//阻塞,只有当至少一个注册的事件发生的时候才会继续.*

*// selector.select();*

Set<SelectionKey> keys = selector.selectedKeys();

Iterator<SelectionKey> it = keys.iterator();

SelectionKey key = null;

while(it.hasNext()){

key = it.next();

it.remove();

try{

handleInput(key);

}catch(Exception e){

if(key != null){

key.cancel();

if(key.channel() != null){

key.channel().close();

}

}

}

}

}catch(Exception e){

e.printStackTrace();

System.exit(1);

}

}

*//selector关闭后会自动释放里面管理的资源*

if(selector != null)

try{

selector.close();

}catch (Exception e) {

e.printStackTrace();

}

}

private void handleInput(SelectionKey key) throws IOException{

if(key.isValid()){

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

if(key.isConnectable()){

if(sc.finishConnect());

else System.exit(1);

}

*//读消息*

if(key.isReadable()){

*//创建ByteBuffer，并开辟一个1M的缓冲区*

ByteBuffer buffer = ByteBuffer.allocate(1024);

*//读取请求码流，返回读取到的字节数*

int readBytes = sc.read(buffer);

*//读取到字节，对字节进行编解码*

if(readBytes>0){

*//将缓冲区当前的limit设置为position=0，用于后续对缓冲区的读取操作*

buffer.flip();

*//根据缓冲区可读字节数创建字节数组*

byte[] bytes = new byte[buffer.remaining()];

*//将缓冲区可读字节数组复制到新建的数组中*

buffer.get(bytes);

String result = new String(bytes,"UTF-8");

System.out.println("客户端收到消息：" + result);

}

*//没有读取到字节 忽略*

*// else if(readBytes==0);*

*//链路已经关闭，释放资源*

else if(readBytes<0){

key.cancel();

sc.close();

}

}

}

}

*//异步发送消息*

private void doWrite(SocketChannel channel,String request) throws IOException{

*//将消息编码为字节数组*

byte[] bytes = request.getBytes();

*//根据数组容量创建ByteBuffer*

ByteBuffer writeBuffer = ByteBuffer.allocate(bytes.length);

*//将字节数组复制到缓冲区*

writeBuffer.put(bytes);

*//flip操作*

writeBuffer.flip();

*//发送缓冲区的字节数组*

channel.write(writeBuffer);

*//\*\*\*\*此处不含处理“写半包”的代码*

}

private void doConnect() throws IOException{

if(socketChannel.connect(new InetSocketAddress(host,port)));

else socketChannel.register(selector, SelectionKey.OP\_CONNECT);

}

public void sendMsg(String msg) throws Exception{

socketChannel.register(selector, SelectionKey.OP\_READ);

doWrite(socketChannel, msg);

}

}

2.7、演示结果  
首先运行服务器，顺便也运行一个客户端：

package com.anxpp.io.calculator.nio;

import java.util.Scanner;

/\*\*

\* 测试方法

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public class Test {

*//测试主方法*

@SuppressWarnings("resource")

public static void main(String[] args) throws Exception{

*//运行服务器*

Server.start();

*//避免客户端先于服务器启动前执行代码*

Thread.sleep(100);

*//运行客户端*

Client.start();

while(Client.sendMsg(new Scanner(System.in).nextLine()));

}

}

我们也可以单独运行客户端，效果都是一样的。  
一次测试的结果：  
服务器已启动，端口号：12345  
1+2+3+4+5+6  
服务器收到消息：1+2+3+4+5+6  
客户端收到消息：21  
1\*2/3-4+5\*6/7-8  
服务器收到消息：1\*2/3-4+5\*6/7-8  
客户端收到消息：-7.0476190476190474  
运行多个客户端，都是没有问题的。  
3、AIO编程  
NIO 2.0引入了新的异步通道的概念，并提供了异步文件通道和异步套接字通道的实现。  
异步的套接字通道时真正的异步非阻塞I/O，对应于UNIX网络编程中的事件驱动I/O（AIO）。他不需要过多的Selector对注册的通道进行轮询即可实现异步读写，从而简化了NIO的编程模型。  
直接上代码吧。  
3.1、Server端代码  
Server：

package com.anxpp.io.calculator.aio.server;

/\*\*

\* AIO服务端

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public class Server {

private static int DEFAULT\_PORT = 12345;

private static AsyncServerHandler serverHandle;

public volatile static long clientCount = 0;

public static void start(){

start(DEFAULT\_PORT);

}

public static synchronized void start(int port){

if(serverHandle!=null)

return;

serverHandle = new AsyncServerHandler(port);

new Thread(serverHandle,"Server").start();

}

public static void main(String[] args){

Server.start();

}

}

AsyncServerHandler：

package com.anxpp.io.calculator.aio.server;

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.channels.AsynchronousServerSocketChannel;

import java.util.concurrent.CountDownLatch;

public class AsyncServerHandler implements Runnable {

public CountDownLatch latch;

public AsynchronousServerSocketChannel channel;

public AsyncServerHandler(int port) {

try {

*//创建服务端通道*

channel = AsynchronousServerSocketChannel.open();

*//绑定端口*

channel.bind(new InetSocketAddress(port));

System.out.println("服务器已启动，端口号：" + port);

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void run() {

*//CountDownLatch初始化*

*//它的作用：在完成一组正在执行的操作之前，允许当前的现场一直阻塞*

*//此处，让现场在此阻塞，防止服务端执行完成后退出*

*//也可以使用while(true)+sleep*

*//生成环境就不需要担心这个问题，以为服务端是不会退出的*

latch = new CountDownLatch(1);

*//用于接收客户端的连接*

channel.accept(this,new AcceptHandler());

try {

latch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

AcceptHandler：

package com.anxpp.io.calculator.aio.server;

import java.nio.ByteBuffer;

import java.nio.channels.AsynchronousSocketChannel;

import java.nio.channels.CompletionHandler;

*//作为handler接收客户端连接*

public class AcceptHandler implements CompletionHandler<AsynchronousSocketChannel, AsyncServerHandler> {

@Override

public void completed(AsynchronousSocketChannel channel,AsyncServerHandler serverHandler) {

*//继续接受其他客户端的请求*

Server.clientCount++;

System.out.println("连接的客户端数：" + Server.clientCount);

serverHandler.channel.accept(serverHandler, this);

*//创建新的Buffer*

ByteBuffer buffer = ByteBuffer.allocate(1024);

*//异步读 第三个参数为接收消息回调的业务Handler*

channel.read(buffer, buffer, new ReadHandler(channel));

}

@Override

public void failed(Throwable exc, AsyncServerHandler serverHandler) {

exc.printStackTrace();

serverHandler.latch.countDown();

}

}

ReadHandler：

package com.anxpp.io.calculator.aio.server;

import java.io.IOException;

import java.io.UnsupportedEncodingException;

import java.nio.ByteBuffer;

import java.nio.channels.AsynchronousSocketChannel;

import java.nio.channels.CompletionHandler;

import com.anxpp.io.utils.Calculator;

public class ReadHandler implements CompletionHandler<Integer, ByteBuffer> {

*//用于读取半包消息和发送应答*

private AsynchronousSocketChannel channel;

public ReadHandler(AsynchronousSocketChannel channel) {

this.channel = channel;

}

*//读取到消息后的处理*

@Override

public void completed(Integer result, ByteBuffer attachment) {

*//flip操作*

attachment.flip();

*//根据*

byte[] message = new byte[attachment.remaining()];

attachment.get(message);

try {

String expression = new String(message, "UTF-8");

System.out.println("服务器收到消息: " + expression);

String calrResult = null;

try{

calrResult = Calculator.cal(expression).toString();

}catch(Exception e){

calrResult = "计算错误：" + e.getMessage();

}

*//向客户端发送消息*

doWrite(calrResult);

} catch (UnsupportedEncodingException e) {

e.printStackTrace();

}

}

*//发送消息*

private void doWrite(String result) {

byte[] bytes = result.getBytes();

ByteBuffer writeBuffer = ByteBuffer.allocate(bytes.length);

writeBuffer.put(bytes);

writeBuffer.flip();

*//异步写数据 参数与前面的read一样*

channel.write(writeBuffer, writeBuffer,new CompletionHandler<Integer, ByteBuffer>() {

@Override

public void completed(Integer result, ByteBuffer buffer) {

*//如果没有发送完，就继续发送直到完成*

if (buffer.hasRemaining())

channel.write(buffer, buffer, this);

else{

*//创建新的Buffer*

ByteBuffer readBuffer = ByteBuffer.allocate(1024);

*//异步读 第三个参数为接收消息回调的业务Handler*

channel.read(readBuffer, readBuffer, new ReadHandler(channel));

}

}

@Override

public void failed(Throwable exc, ByteBuffer attachment) {

try {

channel.close();

} catch (IOException e) {

}

}

});

}

@Override

public void failed(Throwable exc, ByteBuffer attachment) {

try {

this.channel.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

OK，这样就已经完成了，其实说起来也简单，虽然代码感觉很多，但是API比NIO的使用起来真的简单多了，主要就是监听、读、写等各种CompletionHandler。此处本应有一个WriteHandler的，确实，我们在ReadHandler中，以一个匿名内部类实现了它。  
下面看客户端代码。  
3.2、Client端代码  
Client：

package com.anxpp.io.calculator.aio.client;

import java.util.Scanner;

public class Client {

private static String DEFAULT\_HOST = "127.0.0.1";

private static int DEFAULT\_PORT = 12345;

private static AsyncClientHandler clientHandle;

public static void start(){

start(DEFAULT\_HOST,DEFAULT\_PORT);

}

public static synchronized void start(String ip,int port){

if(clientHandle!=null)

return;

clientHandle = new AsyncClientHandler(ip,port);

new Thread(clientHandle,"Client").start();

}

*//向服务器发送消息*

public static boolean sendMsg(String msg) throws Exception{

if(msg.equals("q")) return false;

clientHandle.sendMsg(msg);

return true;

}

@SuppressWarnings("resource")

public static void main(String[] args) throws Exception{

Client.start();

System.out.println("请输入请求消息：");

Scanner scanner = new Scanner(System.in);

while(Client.sendMsg(scanner.nextLine()));

}

}

AsyncClientHandler：

package com.anxpp.io.calculator.aio.client;

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.AsynchronousSocketChannel;

import java.nio.channels.CompletionHandler;

import java.util.concurrent.CountDownLatch;

public class AsyncClientHandler implements CompletionHandler<Void, AsyncClientHandler>, Runnable {

private AsynchronousSocketChannel clientChannel;

private String host;

private int port;

private CountDownLatch latch;

public AsyncClientHandler(String host, int port) {

this.host = host;

this.port = port;

try {

*//创建异步的客户端通道*

clientChannel = AsynchronousSocketChannel.open();

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void run() {

*//创建CountDownLatch等待*

latch = new CountDownLatch(1);

*//发起异步连接操作，回调参数就是这个类本身，如果连接成功会回调completed方法*

clientChannel.connect(new InetSocketAddress(host, port), this, this);

try {

latch.await();

} catch (InterruptedException e1) {

e1.printStackTrace();

}

try {

clientChannel.close();

} catch (IOException e) {

e.printStackTrace();

}

}

*//连接服务器成功*

*//意味着TCP三次握手完成*

@Override

public void completed(Void result, AsyncClientHandler attachment) {

System.out.println("客户端成功连接到服务器...");

}

*//连接服务器失败*

@Override

public void failed(Throwable exc, AsyncClientHandler attachment) {

System.err.println("连接服务器失败...");

exc.printStackTrace();

try {

clientChannel.close();

latch.countDown();

} catch (IOException e) {

e.printStackTrace();

}

}

*//向服务器发送消息*

public void sendMsg(String msg){

byte[] req = msg.getBytes();

ByteBuffer writeBuffer = ByteBuffer.allocate(req.length);

writeBuffer.put(req);

writeBuffer.flip();

*//异步写*

clientChannel.write(writeBuffer, writeBuffer,new WriteHandler(clientChannel, latch));

}

}

WriteHandler：

package com.anxpp.io.calculator.aio.client;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.channels.AsynchronousSocketChannel;

import java.nio.channels.CompletionHandler;

import java.util.concurrent.CountDownLatch;

public class WriteHandler implements CompletionHandler<Integer, ByteBuffer> {

private AsynchronousSocketChannel clientChannel;

private CountDownLatch latch;

public WriteHandler(AsynchronousSocketChannel clientChannel,CountDownLatch latch) {

this.clientChannel = clientChannel;

this.latch = latch;

}

@Override

public void completed(Integer result, ByteBuffer buffer) {

*//完成全部数据的写入*

if (buffer.hasRemaining()) {

clientChannel.write(buffer, buffer, this);

}

else {

*//读取数据*

ByteBuffer readBuffer = ByteBuffer.allocate(1024);

clientChannel.read(readBuffer,readBuffer,new ReadHandler(clientChannel, latch));

}

}

@Override

public void failed(Throwable exc, ByteBuffer attachment) {

System.err.println("数据发送失败...");

try {

clientChannel.close();

latch.countDown();

} catch (IOException e) {

}

}

}

ReadHandler：

package com.anxpp.io.calculator.aio.client;

import java.io.IOException;

import java.io.UnsupportedEncodingException;

import java.nio.ByteBuffer;

import java.nio.channels.AsynchronousSocketChannel;

import java.nio.channels.CompletionHandler;

import java.util.concurrent.CountDownLatch;

public class ReadHandler implements CompletionHandler<Integer, ByteBuffer> {

private AsynchronousSocketChannel clientChannel;

private CountDownLatch latch;

public ReadHandler(AsynchronousSocketChannel clientChannel,CountDownLatch latch) {

this.clientChannel = clientChannel;

this.latch = latch;

}

@Override

public void completed(Integer result,ByteBuffer buffer) {

buffer.flip();

byte[] bytes = new byte[buffer.remaining()];

buffer.get(bytes);

String body;

try {

body = new String(bytes,"UTF-8");

System.out.println("客户端收到结果:"+ body);

} catch (UnsupportedEncodingException e) {

e.printStackTrace();

}

}

@Override

public void failed(Throwable exc,ByteBuffer attachment) {

System.err.println("数据读取失败...");

try {

clientChannel.close();

latch.countDown();

} catch (IOException e) {

}

}

}

这个API使用起来真的是很顺手。  
3.3、测试  
Test：

package com.anxpp.io.calculator.aio;

import java.util.Scanner;

import com.anxpp.io.calculator.aio.client.Client;

import com.anxpp.io.calculator.aio.server.Server;

/\*\*

\* 测试方法

\* @author yangtao\_\_anxpp.com

\* @version 1.0

\*/

public class Test {

*//测试主方法*

@SuppressWarnings("resource")

public static void main(String[] args) throws Exception{

*//运行服务器*

Server.start();

*//避免客户端先于服务器启动前执行代码*

Thread.sleep(100);

*//运行客户端*

Client.start();

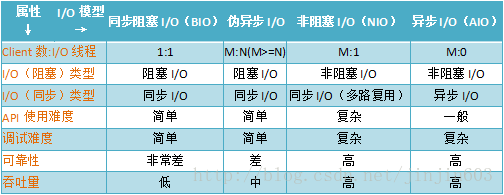
System.out.println("请输入请求消息：");

Scanner scanner = new Scanner(System.in);

while(Client.sendMsg(scanner.nextLine()));

}

}

我们可以在控制台输入我们需要计算的算数字符串，服务器就会返回结果，当然，我们也可以运行大量的客户端，都是没有问题的，以为此处设计为单例客户端，所以也就没有演示大量客户端并发。  
读者可以自己修改Client类，然后开辟大量线程，并使用构造方法创建很多的客户端测试。  
下面是其中一次参数的输出：  
服务器已启动，端口号：12345  
请输入请求消息：  
客户端成功连接到服务器…  
连接的客户端数：1  
123456+789+456  
服务器收到消息: 123456+789+456  
客户端收到结果:124701  
9526\*56  
服务器收到消息: 9526\*56  
客户端收到结果:533456  
…  
AIO是真正的异步非阻塞的，所以，在面对超级大量的客户端，更能得心应手。  
下面就比较一下，几种I/O编程的优缺点。  
4、各种I/O的对比  
先以一张表来直观的对比一下：  
  
具体选择什么样的模型或者NIO框架，完全基于业务的实际应用场景和性能需求，如果客户端很少，服务器负荷不重，就没有必要选择开发起来相对不那么简单的NIO做服务端；相反，就应考虑使用NIO或者相关的框架了。  
5、附录  
上文中服务端使用到的用于计算的工具类：

package com.anxpp.utils;

import javax.script.ScriptEngine;

import javax.script.ScriptEngineManager;

import javax.script.ScriptException;

public final class Calculator {

private final static ScriptEngine jse = new ScriptEngineManager().getEngineByName("JavaScript");

public static Object cal(String expression) throws ScriptException{

return jse.eval(expression);

}

}