java自1.4以后,加入了新IO特性,NIO. 号称new IO. NIO带来了non-blocking特性. 这篇文章主要讲的是如何使用NIO的网络新特性,来构建高性能非阻塞并发服务器.

文章基于个人理解,我也来搞搞NIO.,求指正.

**在NIO之前**

服务器还是在使用阻塞式的java socket. 以Tomcat最新版本没有开启NIO模式的源码为例, tomcat会accept出来一个socket连接,然后调用processSocket方法来处理socket.

while(true) {

....

    Socket socket = null;

    try {

        // Accept the next incoming connection from the server

        // socket

        socket = serverSocketFactory.acceptSocket(serverSocket);

    }

...

...

    // Configure the socket

    if (running && !paused && setSocketOptions(socket)) {

        // Hand this socket off to an appropriate processor

        if (!processSocket(socket)) {

            countDownConnection();

            // Close socket right away(socket);

            closeSocket(socket);

        }

    }

....

}

使用ServerSocket.accept()方法来创建一个连接. accept方法是阻塞方法,在下一个connection进来之前,accept会阻塞.

在一个socket进来之后,Tomcat会在thread pool里面拿出一个thread来处理连接的socket. 然后自己快速的脱身去接受下一个socket连接. 代码如下:

protected boolean processSocket(Socket socket) {

        // Process the request from this socket

        try {

            SocketWrapper<Socket> wrapper = new SocketWrapper<Socket>(socket);

            wrapper.setKeepAliveLeft(getMaxKeepAliveRequests());

            // During shutdown, executor may be null - avoid NPE

            if (!running) {

                return false;

            }

            getExecutor().execute(new SocketProcessor(wrapper));

        } catch (RejectedExecutionException x) {

            log.warn("Socket processing request was rejected for:"+socket,x);

            return false;

        } catch (Throwable t) {

            ExceptionUtils.handleThrowable(t);

            // This means we got an OOM or similar creating a thread, or that

            // the pool and its queue are full

            log.error(sm.getString("endpoint.process.fail"), t);

            return false;

        }

        return true;

    }

而每个处理socket的线程,也总是会阻塞在while(true) sockek.getInputStream().read() 方法上.

总结就是, 一个socket必须使用一个线程来处理. 致使服务器需要维护比较多的线程. 线程本身就是一个消耗资源的东西,并且每个处理socket的线程都会阻塞在read方法上,使得系统大量资源被浪费.

以上这种socket的服务方式适用于HTTP服务器,每个http请求都是短期的,无状态的,并且http后台的业务逻辑也一般比较复杂. 使用多线程和阻塞方式是合适的.

倘若是做游戏服务器,尤其是CS架构的游戏.这种传统模式服务器毫无胜算.游戏有以下几个特点是传统服务器不能胜任的:  
1, 持久TCP连接. 每一个client和server之间都存在一个持久的连接.当CCU(并发用户数量)上升,阻塞式服务器无法为每一个连接运行一个线程.  
2, 自己开发的二进制流传输协议. 游戏服务器讲究响应快.那网络传输也要节省时间. HTTP协议的冗余内容太多,一个好的游戏服务器传输协议,可以使得message压缩到3-6倍甚至以上.这就使得游戏服务器要开发自己的协议解析器.  
3, 传输双向,且消息传输频率高.假设一个游戏服务器instance连接了2000个client,每个client平均每秒钟传输1-10个 message,一个message大约几百字节或者几千字节.而server也需要向client广播其他玩家的当前信息.这使得服务器需要有高速处理 消息的能力.  
4, CS架构的游戏服务器端的逻辑并不像APP服务器端的逻辑那么复杂. 网络游戏在client端处理了大部分逻辑,server端负责简单逻辑,甚至只是传递消息.

**在Java NIO出现以后**

出现了使用NIO写的非阻塞网络引擎,比如Apache Mina, JBoss Netty, Smartfoxserver BitSwarm. 比较起来, Mina的性能不如后两者.Tomcat也存在NIO模式,不过需要人工开启.

首先要说明一下, 与App Server的servlet开发模式不一样, 在Mina, Netty和BitSwarm上开发应用程序都是Event Driven的设计模式.Server端会收到Client端的event,Client也会收到Server端的event,Server端与 Client端的都要注册各种event的EventHandler来handle event.

用大白话来解释NIO:  
1, Buffers, 网络传输字节存放的地方.无论是从channel中取,还是向channel中写,都必须以Buffers作为中间存贮格式.  
2, Socket Channels. Channel是网络连接和buffer之间的数据通道.每个连接一个channel.就像之前的socket的stream一样.  
3, Selector. 像一个巡警,在一个片区里面不停的巡逻. 一旦发现事件发生,立刻将事件select出来.不过这些事件必须是提前注册在selector上的. select出来的事件打包成SelectionKey.里面包含了事件的发生事件,地点,人物. 如果警察不巡逻,每个街道(socket)分配一个警察(thread),那么一个片区有几条街道,就需要几个警察.但现在警察巡逻了,一个巡警 (selector)可以管理所有的片区里面的街道(socketchannel).

以上把警察比作线程,街道比作socket或socketchannel,街道上发生的一切比作stream.把巡警比作selector,引起巡警注意的事件比作selectionKey.

从上可以看出,使用NIO可以使用一个线程,就能维护多个持久TCP连接.

**NIO实例**

下面给出NIO编写的EchoServer和Client. Client连接server以后,将发送一条消息给server. Server会原封不懂的把消息发送回来.Client再把消息发送回去.Server再发回来.用不休止. 在性能的允许下,Client可以启动任意多.

以下Code涵盖了NIO里面最常用的方法和连接断开诊断.注释也全.

首先是Server的实现. Server端启动了2个线程,connectionBell线程用于巡逻新的连接事件. readBell线程用于读取所有channel的数据. **注解: Mina采取了同样的做法,只是readBell线程启动的个数等于处理器个数+1.** 由此可见,NIO只需要少量的几个线程就可以维持非常多的并发持久连接.

每当事件发生,会调用dispatch方法去处理event. 一般情况,会使用一个ThreadPool来处理event. ThreadPool的大小可以自定义.但不是越大越好.如果处理event的逻辑比较复杂,比如需要额外网络连接或者复杂数据库查询,那 ThreadPool就需要稍微大些.**(猜测)**Smartfoxserver处理上万的并发,也只用到了3-4个线程来dispatch event.

EchoServer

public class EchoServer {

    public static SelectorLoop connectionBell;

    public static SelectorLoop readBell;

    public boolean isReadBellRunning=false;

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

        new EchoServer().startServer();

    }

    // 启动服务器

    public void startServer() throws IOException {

        // 准备好一个闹钟.当有链接进来的时候响.

        connectionBell = new SelectorLoop();

        // 准备好一个闹装,当有read事件进来的时候响.

        readBell = new SelectorLoop();

        // 开启一个server channel来监听

        ServerSocketChannel ssc = ServerSocketChannel.open();

        // 开启非阻塞模式

        ssc.configureBlocking(false);

        ServerSocket socket = ssc.socket();

        socket.bind(new InetSocketAddress("localhost",7878));

        // 给闹钟规定好要监听报告的事件,这个闹钟只监听新连接事件.

        ssc.register(connectionBell.getSelector(), SelectionKey.OP\_ACCEPT);

        new Thread(connectionBell).start();

    }

    // Selector轮询线程类

    public class SelectorLoop implements Runnable {

        private Selector selector;

        private ByteBuffer temp = ByteBuffer.allocate(1024);

        public SelectorLoop() throws IOException {

            this.selector = Selector.open();

        }

        public Selector getSelector() {

            return this.selector;

        }

        @Override

        public void run() {

            while(true) {

                try {

                    // 阻塞,只有当至少一个注册的事件发生的时候才会继续.

                    this.selector.select();

                    Set<SelectionKey> selectKeys = this.selector.selectedKeys();

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

                    while (it.hasNext()) {

                        SelectionKey key = it.next();

                        it.remove();

                        // 处理事件. 可以用多线程来处理.

                        this.dispatch(key);

                    }

                } catch (IOException e) {

                    e.printStackTrace();

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

        public void dispatch(SelectionKey key) throws IOException, InterruptedException {

            if (key.isAcceptable()) {

                // 这是一个connection accept事件, 并且这个事件是注册在serversocketchannel上的.

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

                // 接受一个连接.

                SocketChannel sc = ssc.accept();

                // 对新的连接的channel注册read事件. 使用readBell闹钟.

                sc.configureBlocking(false);

                sc.register(readBell.getSelector(), SelectionKey.OP\_READ);

                // 如果读取线程还没有启动,那就启动一个读取线程.

                synchronized(EchoServer.this) {

                    if (!EchoServer.this.isReadBellRunning) {

                        EchoServer.this.isReadBellRunning = true;

                        new Thread(readBell).start();

                    }

                }

            } else if (key.isReadable()) {

                // 这是一个read事件,并且这个事件是注册在socketchannel上的.

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

                // 写数据到buffer

                int count = sc.read(temp);

                if (count < 0) {

                    // 客户端已经断开连接.

                    key.cancel();

                    sc.close();

                    return;

                }

                // 切换buffer到读状态,内部指针归位.

                temp.flip();

                String msg = Charset.forName("UTF-8").decode(temp).toString();

                System.out.println("Server received ["+msg+"] from client address:" + sc.getRemoteAddress());

                Thread.sleep(1000);

                // echo back.

                sc.write(ByteBuffer.wrap(msg.getBytes(Charset.forName("UTF-8"))));

                // 清空buffer

                temp.clear();

            }

        }

    }

}

接下来就是Client的实现.Client可以用传统IO,也可以使用NIO.这个例子使用的NIO,单线程.

public class Client implements Runnable {

    // 空闲计数器,如果空闲超过10次,将检测server是否中断连接.

    private static int idleCounter = 0;

    private Selector selector;

    private SocketChannel socketChannel;

    private ByteBuffer temp = ByteBuffer.allocate(1024);

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

        Client client= new Client();

        new Thread(client).start();

        //client.sendFirstMsg();

    }

    public Client() throws IOException {

        // 同样的,注册闹钟.

        this.selector = Selector.open();

        // 连接远程server

        socketChannel = SocketChannel.open();

        // 如果快速的建立了连接,返回true.如果没有建立,则返回false,并在连接后出发Connect事件.

        Boolean isConnected = socketChannel.connect(new InetSocketAddress("localhost", 7878));

        socketChannel.configureBlocking(false);

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

        if (isConnected) {

            this.sendFirstMsg();

        } else {

            // 如果连接还在尝试中,则注册connect事件的监听. connect成功以后会出发connect事件.

            key.interestOps(SelectionKey.OP\_CONNECT);

        }

    }

    public void sendFirstMsg() throws IOException {

        String msg = "Hello NIO.";

        socketChannel.write(ByteBuffer.wrap(msg.getBytes(Charset.forName("UTF-8"))));

    }

    @Override

    public void run() {

        while (true) {

            try {

                // 阻塞,等待事件发生,或者1秒超时. num为发生事件的数量.

                int num = this.selector.select(1000);

                if (num ==0) {

                    idleCounter ++;

                    if(idleCounter >10) {

                        // 如果server断开了连接,发送消息将失败.

                        try {

                            this.sendFirstMsg();

                        } catch(ClosedChannelException e) {

                            e.printStackTrace();

                            this.socketChannel.close();

                            return;

                        }

                    }

                    continue;

                } else {

                    idleCounter = 0;

                }

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

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

                while (it.hasNext()) {

                    SelectionKey key = it.next();

                    it.remove();

                    if (key.isConnectable()) {

                        // socket connected

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

                        if (sc.isConnectionPending()) {

                            sc.finishConnect();

                        }

                        // send first message;

                        this.sendFirstMsg();

                    }

                    if (key.isReadable()) {

                        // msg received.

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

                        this.temp = ByteBuffer.allocate(1024);

                        int count = sc.read(temp);

                        if (count<0) {

                            sc.close();

                            continue;

                        }

                        // 切换buffer到读状态,内部指针归位.

                        temp.flip();

                        String msg = Charset.forName("UTF-8").decode(temp).toString();

                        System.out.println("Client received ["+msg+"] from server address:" + sc.getRemoteAddress());

                        Thread.sleep(1000);

                        // echo back.

                        sc.write(ByteBuffer.wrap(msg.getBytes(Charset.forName("UTF-8"))));

                        // 清空buffer

                        temp.clear();

                    }

                }

            } catch (IOException e) {

                e.printStackTrace();

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}