基于网络编程的多服务器监管平台

编写环境： java + springboot + 标准socket + NIO

部署环境：CentOS + java8

系统架构：

整个系统分为前端+后端+被监控服务器

前端：

采用前后端分离的框架，直接发送Ajax请求到后端接口，回调数据并渲染界面，达到动态页面的效果

后端：

担任C/S结构中的Server，采用springboot框架支持前端，分别实现了check和command两部分的路由，并包装了统一Restful接口，给前端提供数据支持，另外，在版本2中使用了自启动线程，在springboot启动后自启动监听线程，及开始承担Server的职责，使用长连接+NIO实现了被监控服务器 Client 的状态监测，还未实现 Server 主动发送数据给 Client。

被监控服务器：

担任C/S结构中的Client，采用单程序的标准 socket ，在程序中，待 Client 同 Server 连接成功后，开启三个线程用于处理服务检测、状态发送、命令接收。

状态检测 / 心跳检测：

思路：Server 接受 Client 发送过来的一个心跳包，通过解析心跳包内容，获取特定时间间隔内的 Client 状况，在 Server 端采用统一静态 Hashmap<String, Integer> ServerStatus 存储当前的 Client 的状态，并开放几个需要用的方法，防止随意操作。

Server端

ServerStatus1.java

```
/**
 * 创建一个全局的静态HashMap<ip, 1/0>, 用于存储服务器状态
 * @author 97085
 */
public class ServerStatus1 {
    public static Map<String, Integer> getServerStatus() {
        return serverStatus;
    }
```

```
9
10      public static void setServerStatus(HashMap<String, Integer> serverStatus) {
11          ServerStatus1.serverStatus = serverStatus;
12      }
13
14      public static void putElement(String key, Integer value){
15          serverStatus.put(key, value);
16      }
17
18      public static Integer getElement(String key){
19          return serverStatus.get(key);
20      }
21
22      public static boolean isContainKeys(String key){
23          return serverStatus.containsKey(key);
24      }
25
26      private static Map<String, Integer> serverStatus = new HashMap<String, Integer>
    (16);
27  }
```

springboot自启动线程中使用了NIO的selector做多路复用，利用好每一个建立好的通道的资源，对特定资源进行分配特定的职责，对于状态检测中，selectionKey 感兴趣的操作是 read 。

状态检测的职责是读取 Client 发送过来的数据包，不断获取缓冲区的数据，再解析字符串后设置对应的状态。

倘若 Client 突然断裂，则 Server 获取解析断裂 channel 的 ip， 并设置ServerStatus的对应 ip 状态为0.

```
1   // 获得可读事件
2                       if (key.isReadable()) {
3                           //System.out.println("read be trigger");
4                           // 服务器可读取消息:得到事件发生的Socket通道
5                           SocketChannel clientChannel = (SocketChannel) key.channel();
6
7                           // 创建读取的缓冲区
8                           ByteBuffer readBuffer = ByteBuffer.allocate(30);
9                           int readBytes = clientChannel.read(readBuffer);
10                          // 客户端中断后执行
11                          if (readBytes == -1) {
12                              // 过滤ip
13                              Matcher matcher =
    foundIP(clientChannel.getRemoteAddress().toString());
14                              String clientIP = "";
15                              if(matcher.find()) {
16                                  clientIP = (String)matcher.group(0);
17                                  // 修改被监听服务器状态，两个if是因为两个hashmap有交叉
18                                  if(ServerStatus1.isContainKeys(clientIP)){
19                                      ServerStatus1.putElement(clientIP,0);
20                                  }
21                                  if(ServerStatus2.isContainKeys(clientIP)){
22                                      ServerStatus2.putElement(clientIP,0);
23                                  }
```

```
                            }
                            System.out.println(clientIP+" disconnect……");
                            clientChannel.close();
                        }
                        // 客户端存活执行
                        else if (readBytes > 0) {
                            // 对应格式字符串转hashmap----也可以转json
                            String s = new String(readBuffer.array());
                            Map<String, String> m = stringToHashmap(s);
                            String ip = m.get("ip");
                            String statusStr = m.get("status");
                            Integer status = Integer.valueOf(statusStr);

                            // 根据组别写入对应hashmap
                            if("1".equals(m.get("group"))){
                                ServerStatus1.putElement(ip,status);
                            }else if("2".equals(m.get("group"))){
                                ServerStatus2.putElement(ip,status);
                            }else{
                                ServerStatus1.putElement(ip,status);
                                ServerStatus2.putElement(ip,status);
                            }
                            // 切换到写
                            key.interestOps(SelectionKey.OP_WRITE);
                        }

                    }
```

前端获取各个服务器状态

只需要获取当前对应的 ServerStatus 的内容，即可得知更新时间内 Client 的状态

```
/**
     * 版本2
     * 检测各个服务器状态
     */
    @RequestMapping("/check2")
    @ResponseBody
    public Result<Map<String, Integer>> check2(@RequestParam("num") Integer num){
        Result<Map<String, Integer>> res;
        if(num==1){
            res = Result.success(ServerStatus1.getServerStatus());
        }else{
            res = Result.success(ServerStatus2.getServerStatus());
        }
        return res;
    }
```

Client端

服务检测线程

对于常规有开启对应端口的程序，可直接检测对应端口是否打开

对于其他程序，可直接通过检测当前系统是否有对应的进程号即可

```
class checkPortThread implements Runnable{
        @Override
        public void run(){
            //适用于nginx、haproxy，web，dns
            while(true){
                try {
                    Thread.sleep(500);
                    // 简单socket检测
                    Socket socket = new Socket();
                    SocketAddress add = new InetSocketAddress("127.0.0.1",CHECKPORT);
                    // 超时时间为0.5s
                    socket.connect(add,500);
                    socket.close();
                    status = "1";
                } catch (Exception e) {
                    status = "0";
                }
            }
            // 使用于lvs监听keepalived
//            while(true){
//                try {
//                    // 通过java执行linux命令
//                    String cmd = "ps -ef | grep \"keepalived\" | grep -v \"grep\" | awk '{print $2}'";
//                    String[] cmdA = {"/bin/sh", "-c", cmd};
//                    Process process = Runtime.getRuntime().exec(cmdA);
//                    LineNumberReader br = new LineNumberReader(new InputStreamReader(process.getInputStream()));
//                    StringBuffer sb = new StringBuffer();
//                    String line;
//                    while ((line = br.readLine()) != null) {
//                        sb.append(line).append("\n");
//                    }
//                    if(sb.length()>0){
//                        status = "1";
//                    }else{
//                        status = "0";
//                    }
//                } catch (Exception e) {
//                    status = "0";
//                    e.printStackTrace();
//                }
//            }
        }
    }
```

状态发送

鉴于Server使用了NIO，因此需要保证传输使用的是字节流，所以连接后发送信息需要先处理好信息----->字节流，每过一秒发送一次心跳

```
/**
     * 心跳检测线程
     */
    class heartThread implements Runnable {
        @Override
        public void run() {
            while(true) {
                try {
                    sendMsg("ip="+LOCALHOST+"&group="+GROUP+"&status="+status);
                    //System.out.println(LOCALHOST+"send message to server
Successfully");
                    Thread.sleep(1000);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    /**
     * 给服务端发送信息
     * @param content
     */
    public void sendMsg(String content) {
        try {
            byte[] data = content.getBytes();
            dos.write(data);
            dos.flush();
        } catch (Exception e) {
            e.printStackTrace();
            closeSocket();
        }
    }
```

命令执行：

思路：前端通过按钮发送对应的命令以及 Client ip，后端包装完毕后将命令放入事先声明的静态队列，并开放几个需要使用到的方法

Server端

CommandQueue.java

```
/**
```

```
 * @author 97085
 */
public class CommandQueue{
    private static Queue<String> commandQueue = new LinkedList<>();

    /**
     * 向尾部添加一个元素
     * @param element
     */
    public static void putLast(String element){
        commandQueue.offer(element);
    }

    /**
     * 获取第一个元素并删除
     * @return
     */
    public static String getFirst(){
        return commandQueue.poll();
    }

    /**
     * 返回一个长度
     * @return
     */
    public static int getLength(){
        return commandQueue.size();
    }
}
```

springboot自启动方法开启后，建立连接后，标注selectionKey的感兴趣的操作为 read | write。

```
clientChannel.register(key.selector(), SelectionKey.OP_READ|SelectionKey.OP_WRITE);
```

之后开始对连接轮询的时候便可能会触发 write 广播，触发条件是当前命令队列不为空，即之前前端有发送过指令

```
String temp = CommandQueue.getFirst();
// ……
// 中间省略了一段
// ……
while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                if(!temp.isEmpty()) {
                      if (key.isWritable()) {
                          SocketChannel channel = (SocketChannel) key.channel();
                          byte[] asd=
(Thread.currentThread().getName()+temp).getBytes();
                          ByteBuffer bf=ByteBuffer.allocate(asd.length);
                          bf.put(asd);
```

```
                            bf.flip();
                            while(bf.hasRemaining()) {
                                try {
                                    channel.write(bf);
                                } catch (IOException e) {
                                    key.cancel();
                                    channel.socket().close();
                                    channel.close();
                                    e.printStackTrace();
                                }
                            }
//                            if (CommandQueue.getLength() > 0) {
//                                SocketChannel clientChannel = (SocketChannel)
key.channel();
//
//                                // 内容写入缓冲区
//                                //String temp = CommandQueue.getFirst();
//                                System.out.println("write to buffer : " + temp);
//
clientChannel.write(ByteBuffer.wrap(temp.getBytes()));
//                            }
//                            // 切换回读
//                            key.interestOps(SelectionKey.OP_READ);
                        }
                    }
}
```

Client端

命令接收

Server 端发送命令，拟采用广播的形式，信息中带上对应 Client 的 ip ，Client 获取对应信息后解析，若是同一 ip 则可以执行，否则就跳过。

```
/**
     * 接受信息的线程
     */
    class MsgThread implements Runnable {
        @Override
        public void run() {
            int temp;
            while(true) {
                try {
                    if(socket.getInputStream().available() > 0) {
                        byte[] bytes = new byte[1024];
                        // if语句中去掉了第一个字符
                        if(dis.read()!='\0'){
                                dis.read(bytes);
                                Str = new String(bytes);
                                System.out.println(Str);
                                // 服务端发送的字符串（去掉首个字符）格式是"ip:command"
```

```
                                String[] str = Str.split(":");
                                if(str.length>1){
                                    String ip = str[0];
                                    String command = str[1];
                                    // 当ip为本机ip时，可执行对应命令
                                    if(LOCALHOST.equals(ip)){
                                        try{
                                            Process ps =
Runtime.getRuntime().exec(command);
                                        }catch (Exception e){
                                            e.printStackTrace();
                                        }
                                    }
                                }else {
                                    Thread.sleep(500);
                                }
                            }
                        }
                    } catch (Exception e) {
                        closeSocket();
                    }
                }
            }
        }
```