智能家居远程控制：实现 APP 与 ESP8266 远程通信

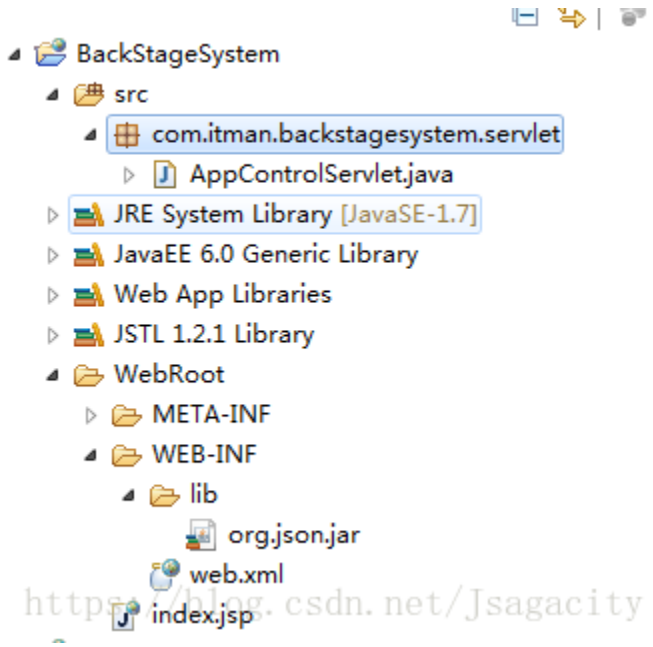
智能家居实现前，我们需要做一些准备。移动端针对的是 Android，服务端用的是 Java 实现。这里 Android 项目使用的是 Android Studio，服务端实现使用的是 MyEclipse，JDK 和服务器 Tomcat 的安装网上很多教程，这里就不一一说明了。

学前说明，如果是初学者建议是按照步骤先新建部署项目，将项目运行起来之后，再详细了解实现过程。学习过程中切莫过度理解每一个步骤、每一行代码，这样很影响学习热情的。如果觉得烦，可以下载源码，直接在局域网中部署运行了，在慢慢了解实现过程。

首先先实现服务器的返回 JSON 数据

在 MyEclipse 中新建一个 Web Project，而后在项目的 WebRoot -> WEB-INF -> lib 文件里面添加 JSON 的 jar 包，jar 包文末有提供。

服务端的项目结如下：



接着再添加一个 servlet 文件，AppControlServlet.java 代码全部贴出来，如下：

```
public class AppControlServlet extends HttpServlet {

    /**
     *
```

```
     */
    private static final long serialVersionUID =
-5826345371893667 87L;

    public void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        //让doGet请求也归类为doPost请求
        doPost(request, response);
    }

    public void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html;charset=UTF-8");
        //获取doPost请求中的传过来的数据
        String username = request.getParameter("username");
        String sessionId = request.getParameter("sessionId");
        String data = request.getParameter("data");

        JSONObject jObject = new JSONObject();
        if (sessionId != null) {
            // TODO 将获取的数据打印出来
            System.out.println("username:" + username);
            System.out.println("sessionId:" + sessionId);
            System.out.println("data:" + data);

            try {
                //返回json数据
                JSONObject record = new JSONObject();
                record.put("username", username);

                jObject.put("reason", "SUCCESSED");
                jObject.put("resultCode", 200);
                jObject.put("totalNum", 1);
                jObject.put("data", record);
            } catch (JSONException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        } else {
            // 未接收到该设备的id
            try {
                jObject.put("resultCode", 204);
                jObject.put("reason", "NULL");
                jObject.put("data", "");
            } catch (JSONException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                try {
```

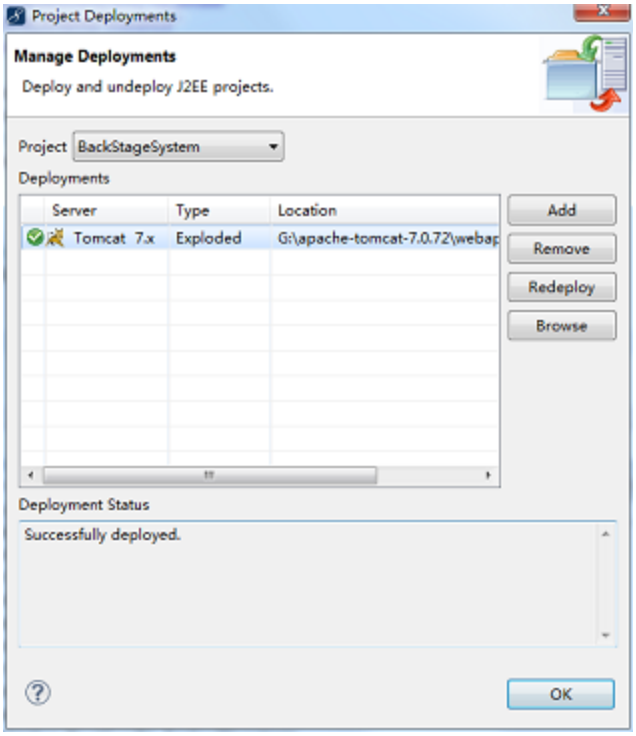
```
                    jObject.put("resultCode", 400);
                    jObject.put("reason", "ERROR");
                    jObject.put("data", "");

                } catch (JSONException ex) {
                    // TODO: handle exception
                    ex.printStackTrace();
                }
            }
        }
        PrintWriter out = response.getWriter();
        out.print(jObject);
        out.flush();
        out.close();

    }
}
```

接着在 web.xml 中添加映射：

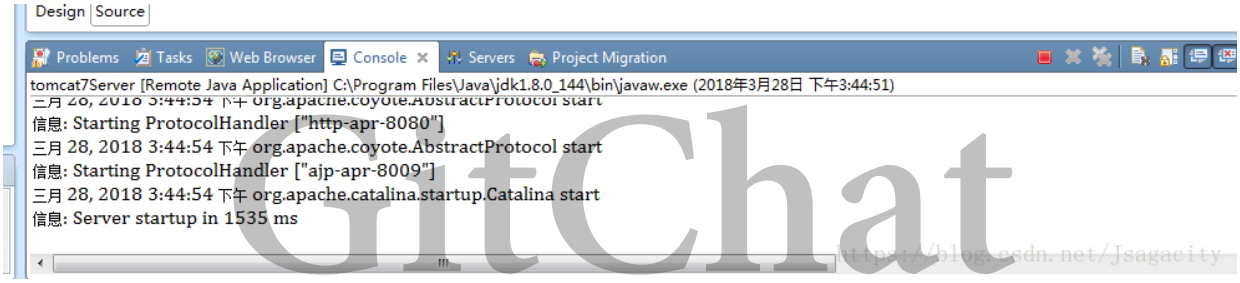
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <description>This is the description of my J2EE
component</description>
        <display-name>This is the display name of my J2EE
component</display-name>
        <servlet-name>AppControlServlet</servlet-name>
        <servlet-
class>com.itman.backstagesystem.servlet.AppControlServlet</servle
t-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>AppControlServlet</servlet-name>
        <url-pattern>/servlet/AppControlServlet</url-pattern>
    </servlet-mapping>

</web-app>
```


完成之后项目基本可以完成简单的 JSON 数据返回，接着就是演示，给项目添加 Tomcat 服务器，然后运行起来。

添加 Tomcat：



运行 Tomcat：

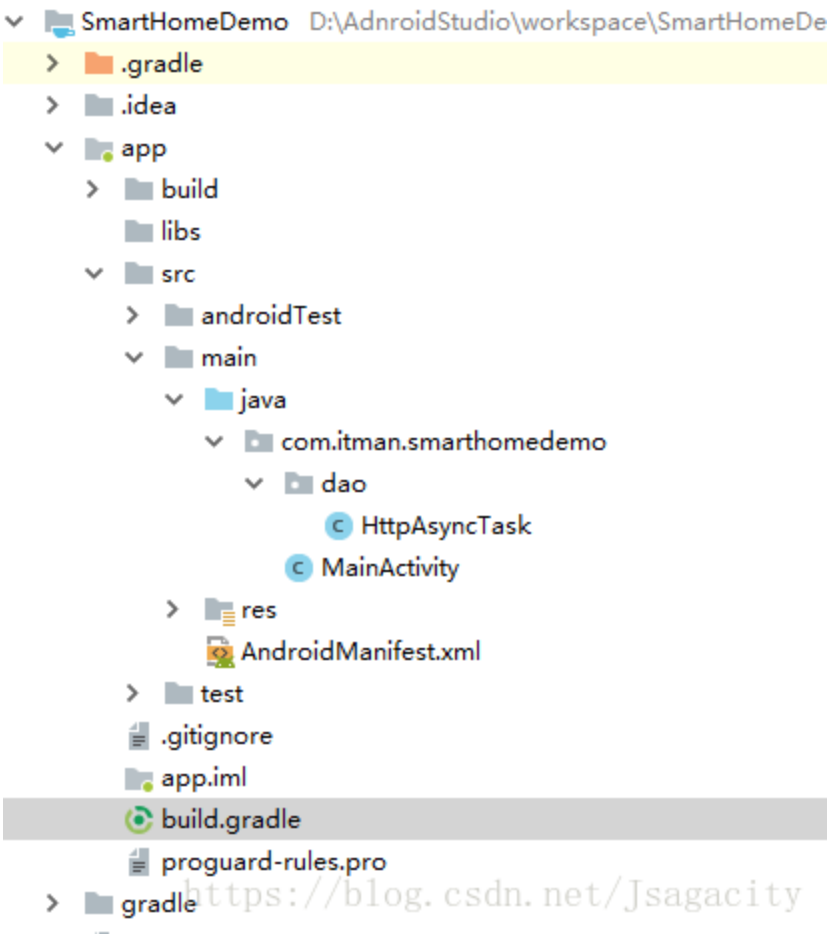


接下来测试一下服务器是否能成功返回数据：

到此为止，服务端的回传准备完毕。

实现 Android 请求服务端

Android 的项目结构如下：



先新建一个 Android 项目，在主布局页面 activity_main.xml 添加一个按钮：


```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/btnSend"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="发送" />

</RelativeLayout>
```

MainActivity.java 中的代码实现：

```
public class MainActivity extends AppCompatActivity implements
View.OnClickListener {
    private Button btnSend;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnSend = findViewById(R.id.btnSend);
        btnSend.setOnClickListener(this);
    }

    @Override
    public void onClick(View view) {
        switch (view.getId()){
            case R.id.btnSend:
                byte[] msg = new byte[]{(byte) 0x01, (byte) 0x02,
(byte) 0x03};
                String name = "layne";
                String sessionId = "CONN_9527";
                //发出请求
                serviceCONN(msg,name,sessionId);
                break;
        }
    }

    private void serviceCONN(byte[] bytes, String username,
String sessionId) {
        //将byte数组转化成字符串
        String data_msg = Arrays.toString(bytesToInt(bytes));
        //服务器的url
        String url =
"http://192.168.2.176:8080/BackStageSystem/servlet/AppControlServ
let";
        //将数据拼接起来
        String data = "username=" + username + "&sessionId=" +
sessionId + "&data=" + data_msg;
        String[] str = new String[]{url, data};

        //发出一个请求
        new HttpAsyncTask(MainActivity.this, new
HttpAsyncTask.PriorityListener() {

            @Override
            public void setActivity(int code) {
                switch (code) {
                    case 200:
                        //如果返回的resultCode是200,那么说明APP的数据
传送成功，并成功解析返回的json数据
                        Toast.makeText(MainActivity.this, "发送数
据：[0x01,0x02,0x03]", Toast.LENGTH_SHORT).show();
                        break;
                    case 202:
                        Toast.makeText(MainActivity.this, "设备离
线状态", Toast.LENGTH_SHORT).show();
```

```
                        break;
                    default:
                        Toast.makeText(MainActivity.this, "网络传
输异常", Toast.LENGTH_SHORT).show();
                        break;
                }

            }
        }).execute(str);
    }

    /**
     * byte转化为int
     */
    public static int[] bytesToInt(byte[] src) {
        int value[] = new int[src.length];
        for (int i = 0; i < src.length; i++) {
            value[i] = src[i] & 0xFF;
        }
        return value;
    }
}
```

以上会用到一个异步请求类 HttpAsyncTask，这是我封装好的异步处理类，直接拿来用就可以了，这里贴出来：


```
/**
 * Created by Layne_Yao on 2017/5/12.
CSDN:http://blog.csdn.net/Jsagacity
 */

public class HttpAsyncTask extends AsyncTask<String, Void,
String> {
    private Context context;

    /**
     * 自定义Dialog监听器
     */
    public interface PriorityListener {
        /**
         * 回调函数，用于在Dialog的监听事件触发后刷新Activity的UI显示
         */
        void setActivity(int code);
    }

    private PriorityListener listener;

    public HttpAsyncTask(Context context, PriorityListener
listener) {
        this.context = context;
```

```
        this.listener = listener;
    }

    @Override
    protected String doInBackground(String... params) {
        String path = params[0];
        String data = params[1];
        String content = null;
        try {
            // 创建一个URL对象，参数就是网址
            URL url = new URL(path);
            // 获取HttpURLConnection连接对象
            HttpURLConnection conn = (HttpURLConnection)
url.openConnection();
            // 默认请求是get，要大写
            conn.setRequestMethod("POST");
            // 设置网络连接的超时时间
            conn.setConnectTimeout(5000);
            // 设置两个请求头信息
            conn.setRequestProperty("Content-Type",
                    "application/x-www-form-urlencoded");
            conn.setRequestProperty("Content-Length",
data.length() + "");

            // 把组拼好的数据提交给服务器，以流的形式提交
            conn.setDoOutput(true);
            conn.getOutputStream().write(data.getBytes());

            // 获取服务器返回的状态码,200代表获取服务器资源全部成功，206
请求部分资源
            int code = conn.getResponseCode();
            if (code == 200) {
                InputStream in = conn.getInputStream();
                content = readStream(in);
                Log.e("SendAsyncTask", content);
            }

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return content;
    }

    @Override
    protected void onPostExecute(String result) {
        if (result != null) {
            try {
                JSONObject jsonObject = new JSONObject(result);
                int code = jsonObject.getInt("resultCode");

                if (code != 0) {
```

```
                    listener.setActivity(code);
                } else {
                    Toast.makeText(context, "数据类型不正确",
Toast.LENGTH_SHORT).show();
                }
            } catch (JSONException e) {
                e.printStackTrace();
            }
        } else {
            Toast.makeText(context, "网络连接错误",
Toast.LENGTH_SHORT).show();
        }

        super.onPostExecute(result);
    }

    public String readStream(InputStream in) throws Exception {
        //定义一个内存输出流
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        int len = -1;
        byte[] buffer = new byte[1024];
        while ((len = in.read(buffer)) != -1) {
            baos.write(buffer, 0, len);
        }
        in.close();
        String content = new String(baos.toByteArray());
        return content;
    }

}
```


我刚刚更新了 Android Studio3.1，这次新建这个项目并不用添加 Gson 的依赖，项目都可以运行，如果是之前的版本是需要添加 Gson 的依赖，在 app 的 build.gradle 下添加：

```
    implementation 'com.google.code.gson:gson:2.8.2'
```

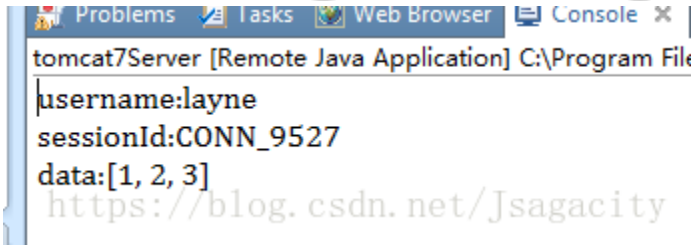
最后还需要添加一个网络请求权限：

```
    <uses-permission android:name="android.permission.INTERNET"/>
```

到此 APP 项目也实现完毕，将项目运行到 Android 手机（最好是运行到真机），然后进行测试一下，注意：这里还未将服务端部署到云服务器上，所以手机也必须在局域网内才能请求成功。

将 Android 项目运行到真机之后，确保服务器开启状态，请求成功：

服务器接收的数据也是正常的：



以上就是简单的 APP 与服务端通过 JSON 数据进行通讯的过程。

ESP8266 与服务器的长连接实现

其实我博文《ESP8266作为客户端通过路由器连接服务器的简单实现》中已经介绍了类似的实现。

只是将服务器部署到 JavaWeb 中有点不一样，接下来就先讲解服务器端长连接代码实现。就添加两个类，添加点映射就可以了。完成这里也就可以完成 APP 到 8266 的单向通信了，后面会演示，请详看代码。

服务器代码实现：

```
/**
 * 硬件端与服务器端的长连接
 *
 * @author Administrator
 *
 */
public class WifiServerSocket extends Thread {
    private ServletContext servletContext;
    private ServerSocket serverSocket;

    private static Map<String, ProcessSocketData> socketMap = new
HashMap<>();

    public WifiServerSocket(ServletContext servletContext) {
        this.servletContext = servletContext;

        // 从web.xml中context-param节点获取socket端口
        String port =
this.servletContext.getInitParameter("socketPort");
        if (serverSocket == null) {
            try {

                this.serverSocket = new
ServerSocket(Integer.parseInt(port));
            } catch (IOException e) {
                e.printStackTrace();

            }

        }

    }

    public void run() {

        while (!this.isInterrupted()) { // 线程未中断执行循环

            try {
                // 开启服务器，线程阻塞，等待8266的连接
                Socket socket = serverSocket.accept();
                ProcessSocketData psd = new
ProcessSocketData(socket);
                new Thread(psd).start();

            } catch (IOException e) {
                e.printStackTrace();
            }

        }

    }
```

```
    public void closeServerSocket() {

        try {

            if (serverSocket != null && !serverSocket.isClosed())
                serverSocket.close();

        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    //将socket连接一静态集合变量的形式暴露出去
    public static Map<String, ProcessSocketData> getSocketMap() {
        return socketMap;
    }

    public class ProcessSocketData extends Thread {
        private Socket socket;
        private InputStream in = null;
        private DataOutputStream out = null;

        private String mStrName = null;
        private boolean play = false;

        // 构造方法，传入连接进来的socket
        public ProcessSocketData(Socket socket) {
            this.socket = socket;
            try {
                in = new
DataInputStream(socket.getInputStream());
                out = new
DataOutputStream(socket.getOutputStream());
            } catch (IOException e) {

                e.printStackTrace();
            }
            play = true;
        }

        public void run() {
            try {
                // 死循环，无线读取8266发送过来的数据
                while (play) {
                    byte[] msg = new byte[10];
                    in.read(msg);//读取流数据
                    System.out.println("WiFi发过来的数据：" +
Arrays.toString(msg));
                    String str = new String(msg).trim();
                    System.out.println(str);
```

```
                    if (str.contains("CONN")) {
                        mStrName = str.trim();
                        /*
                         * 判断发过的是CONN_9527,那么就将此socket对象添加到这个类的静态集合里面，以CONN_9527为索引。
                         * 很多人这里可能不太懂，APP与服务端的通信在AppControlServlet类中触发，想要实现APP与8266通信，只能将这个socket对象通过类的静态变量暴露出去。
                         * 等到AppControlServlet收到APP的信息，就立马通过CONN_9527作为索引取出socket，和8266进行通讯
                         */
                        WifiServerSocket.socketMap.put(mStrName, this);

                    }

                }
            } catch (IOException e) {
                e.printStackTrace();

            } finally {
                try {
                    in.close();
                    if (socket != null && !socket.isClosed()) {
                        socket.close();
                    }
                } catch (IOException e) {

                    e.printStackTrace();
                }
            }
        }

        //这是服务器发送数据到8266的函数
        public void send(byte[] bytes) {
            try {
                out.write(bytes);
            } catch (IOException e) {
                try {
                    // 移除集合里面的Socket
                    WifiServerSocket.socketMap.remove(mStrName);
                    out.close();
                    play = false;
                    in.close();
                    if (socket != null && !socket.isClosed()) {
                        socket.close();
                    }
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
```

```
                System.out.println("该客户端已退出！");
            }
        }

    }

}
```

接着实现侦听服务器：

```
public class WifiServerSocketListener implements
ServletContextListener {

    private WifiServerSocket wifiServerSocket;

    public void contextDestroyed(ServletContextEvent e) {

        if (wifiServerSocket != null &&
wifiServerSocket.isInterrupted()) {
            wifiServerSocket.closeServerSocket();
            wifiServerSocket.interrupt();
        }
    }

    public void contextInitialized(ServletContextEvent e) {
        ServletContext servletContext = e.getServletContext();
        if (wifiServerSocket == null) {
            wifiServerSocket = new
WifiServerSocket(servletContext);
            wifiServerSocket.start(); // servlet上下文初始化时启动
socket服务端线程
        }

    }

}
```

最后再映射文件 web.xml 里添加以下代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <description>This is the description of my J2EE
component</description>
        <display-name>This is the display name of my J2EE
component</display-name>
```

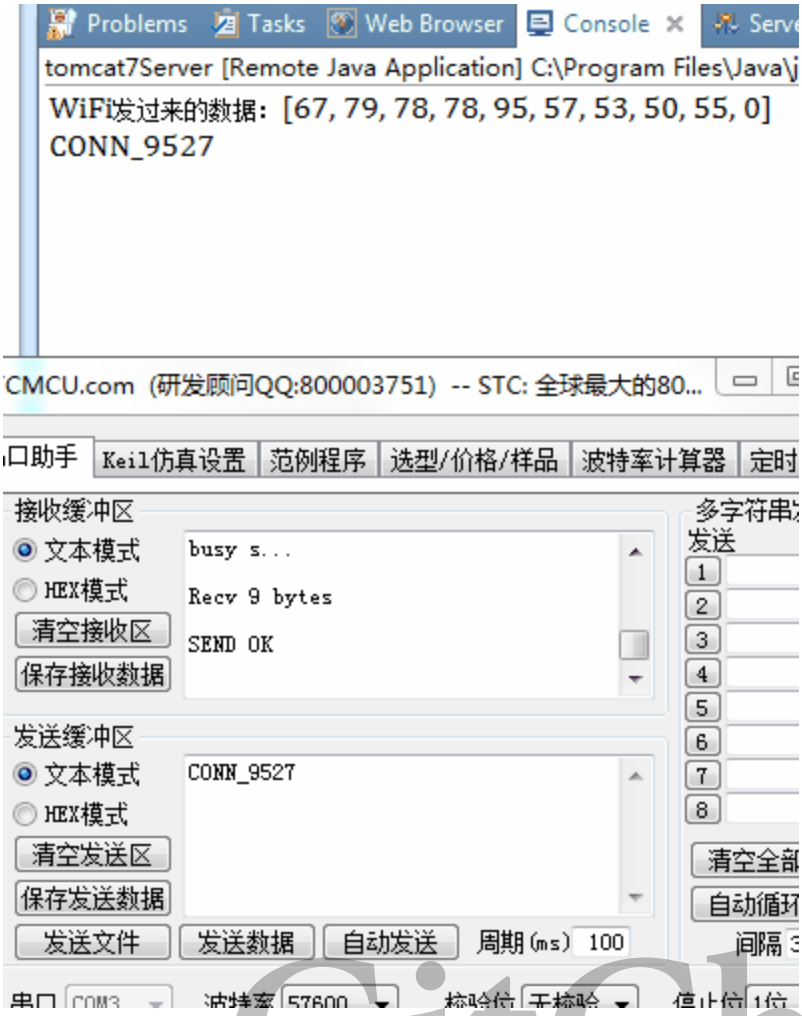
```
        <servlet-name>AppControlServlet</servlet-name>
        <servlet-
class>com.itman.backstagesystem.servlet.AppControlServlet</servle
t-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>AppControlServlet</servlet-name>
        <url-pattern>/servlet/AppControlServlet</url-pattern>
    </servlet-mapping>



    <context-param>
        <param-name>socketPort</param-name>
        <param-value>10086</param-value>
    </context-param>

    <listener>
        <description>WifiServerSocket 服务随 web 启动而启动
</description>
        <listener-
class>com.itman.backstagesystem.service.WifiServerSocketListener<
/listener-class>
    </listener>

</web-app>
```


以上代码添加完毕，代码有带注释，想要了解更清楚就自己详读源码。完了之后运行服务器，用 ESP8266 连接测试一下。

这里贴出 ESP8266 所要用到的指令：

1. AT+CWMODE=1
2. AT+RST
3. AT+CWJAP=”WiFi名”,”WiFi密码” //路由器的WiFi密码和账号
4. AT+CIPMUX=1
5. AT+CIPSERVER=1,8800
6. AT+CIPSTART=0,”TCP”,”192.168.2.176”,10086
7. AT+CIPSEND=0,9
8. CONN_9527

运行之后，亲自测试是可行的，下面是截图：



接下来连通 APP -> 服务器 -> 8266：

连通这两个连接通信，AppControlServlet 就需要修改一下，改得不多，无法详细说明，所以就把改完后的代码全部贴出来，哪里改动了，自己对比查看。

```
public class AppControlServlet extends HttpServlet {

    /**
     *
     */
    private static final long serialVersionUID =
-5826345371893667 87L;

    public void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        // 让doGet请求也归类为doPost请求
        doPost(request, response);
    }

    public void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html;charset=UTF-8");
```

```
        // 获取doPost请求中的传过来的数据
        String username = request.getParameter("username");
        String sessionId = request.getParameter("sessionId");
        String data = request.getParameter("data");

        // 将字符串的数据转化成byte数组
        byte[] msg = ToolUtils.stringToByte(data);

        JSONObject jObject = new JSONObject();
        if (sessionId != null) {
            // TODO 将获取的数据打印出来
            System.out.println("username:" + username);
            System.out.println("sessionId:" + sessionId);
            System.out.println("data:" + data);

            // 这里的sessionId是CONN_9527，通过这个索引取出相对应的
socket对象，然后将APP发送过来的数据，再发送到8266
            ProcessSocketData psd =
WifiServerSocket.getSocketMap().get(
                    new String(sessionId));
            if (psd != null) {
                // TODO 8266在线状态
                psd.send(msg);
                System.out.println("数据已发送到8266");
                try {
                    JSONObject record = new JSONObject();
                    record.put("username", username);

                    jObject.put("reason", "SUCCESSED");
                    jObject.put("resultCode", 200);
                    jObject.put("totalNum", 1);
                    jObject.put("data", record);
                } catch (JSONException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

            } else {
                // TODO 继电器离线状态
                System.out.println("socket连接为空，8266未连接服务
器");
                try {
                    JSONObject record = new JSONObject();
                    record.put("username", username);

                    jObject.put("reason", "SUCCESSED");
                    jObject.put("resultCode", 202);
                    jObject.put("totalNum", 0);
                    jObject.put("data", record);
                } catch (JSONException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
```

```
                }
            }

        } else {
            // 未接收到该设备的id
            try {
                jObject.put("resultCode", 204);
                jObject.put("reason", "NULL");
                jObject.put("data", "");
            } catch (JSONException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                try {
                    jObject.put("resultCode", 400);
                    jObject.put("reason", "ERROR");
                    jObject.put("data", "");

                } catch (JSONException ex) {
                    // TODO: handle exception
                    ex.printStackTrace();
                }
            }
        }
        PrintWriter out = response.getWriter();
        out.print(jObject);
        out.flush();
        out.close();

    }
}
```


上面用了一个工具类，是我自己写的，是将字符串的数据转换成 byte 数组的函数：

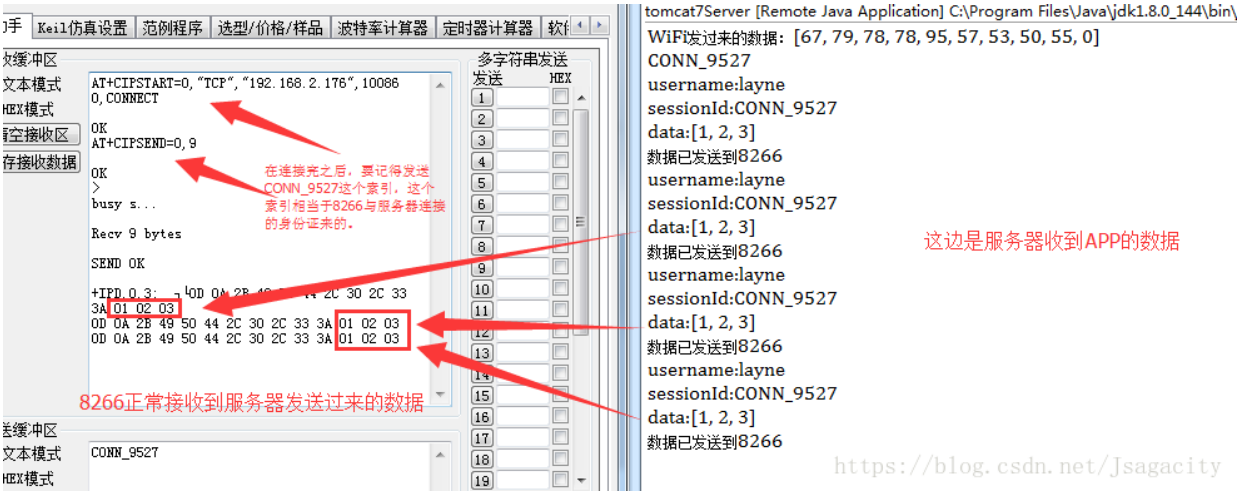
```
public class ToolUtils {
    public static byte[] stringToByte(String str) {
        byte[] bytes = null;
        if (str != null) {
            // 去掉头尾的中括号
            String str1 = str.replace("[", "").replace("]", "");
            // 以逗号分割每个字符，生成新的字符数组
            String[] str_msg = str1.split(",");
            bytes = new byte[str_msg.length];
            // 强制转换并生成byte数组
            for (int i = 0; i < str_msg.length; i++) {
                int msg = Integer.valueOf(str_msg[i].trim());
                bytes[i] = (byte) msg;
            }

        }
        return bytes;
```

```
        }

        public int[] bytesToInt(byte[] src) {
            int value[] = new int[src.length];
            for (int i = 0; i < src.length; i++) {
                value[i] = src[i] & 0xFF;
            }
            return value;
        }
    }
```

OK，添加完之后，就可以重启服务器，先把 8266 按照上面的操作连上服务器，而后在真机的 APP 上发送一下数据。

总共点击了四下，发送了四次数据：

下面截图说明一下：



以上是 8266 与服务器的长连接实现，演示起来不是很利索。

最后实现 8266 与 APP 的通信

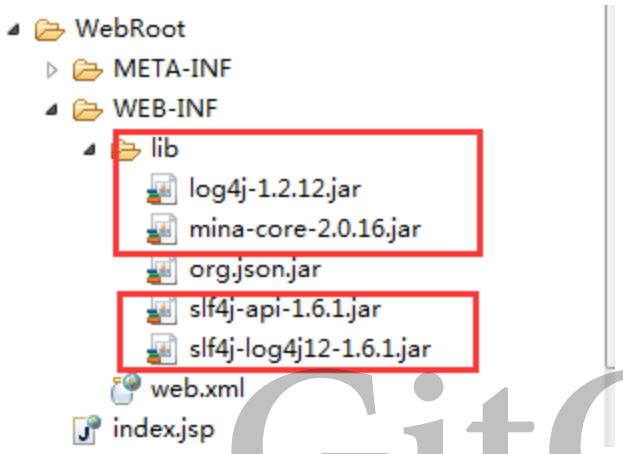
在上一节中，APP 到 8266 的单向通信已经实现成功，但是 8266 到 APP 的方向通信还未完全实现。其实 8266 到服务器的通信在上一节中也已经实现。

所以在这一节中，要实现的是服务器到 APP 的通信，并将 8266 到服务器和服务器到 APP 的通信连接起来。这一节需要点耐心的，还是有点难度，有点复杂的。

一般来说都是 APP 主动与服务器通信，然后服务器再做响应的。但是总会有其他的需要实现服务器主动与 App 通信的。服务器要与 App 主动通信，不太可能在 App 里开个服务器的，所以就只能使用长连接实现了，这里的长连接实现使用的是 MINA 框架。

先在服务器部署 MINA 框架的实现：

先导入四个关于 MINA 框架的 jar 包，jar 包文末或者项目源码中有提供。



添加与 APP 长连接的 Socket，继承多线程 Thread：

```
/**
 * APP移动端与服务器端的长连接
 *
 * @author Administrator
 *
 */
public class AppServiceSocket extends Thread {
    private static IoAcceptor acceptor = null;
    private static Map<String, IoSession> IoSessionMap = new
HashMap<>();

    @Override
    public void run() {
        acceptor = new NioSocketAcceptor();
        // 添加日志过滤器
        acceptor.getFilterChain().addLast("logger", new
LoggingFilter());
        acceptor.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new
ObjectSerializationCodecFactory()));
        acceptor.setHandler(new DemoServerHandler());
        acceptor.getSessionConfig().setReadBufferSize(2048);
```

```
acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE,
10);

        try {
            acceptor.bind(new InetSocketAddress(10011));
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("启动服务");

    }

    public static Map<String, IoSession> getAcceptorSessions() {
        return IoSessionMap;
    }

    private static class DemoServerHandler extends
IoHandlerAdapter {

        @Override
        public void sessionCreated(IoSession session) throws
Exception {
            // TODO 服务器与客户端创建连接
            // System.out.println("服务器与客户端创建连接...");
            super.sessionCreated(session);
        }

        @Override
        public void sessionOpened(IoSession session) throws
Exception {
            // TODO 服务器与客户端连接打开
            // System.out.println("服务器与客户端连接打开...");
            super.sessionOpened(session);
        }

        // TODO 消息的接收处理
        @Override
        public void messageReceived(IoSession session, Object
message)
                throws Exception {
            super.messageReceived(session, message);

            String str = message.toString().trim();

            IoSessionMap.put(str, session);

            System.out.println("客户端发送的数据:" + str);

acceptor.getManagedSessions().get(session.getId()).write("连接服务
器成功");
        }
```

```
        @Override
        public void messageSent(IoSession session, Object
message)
                throws Exception {
            super.messageSent(session, message);

        }

        @Override
        public void sessionClosed(IoSession session) throws
Exception {
            super.sessionClosed(session);

        }
    }
}
```

添加侦听：

```
public class AppServerSocketListener implements
ServletContextListener {

    private AppServiceSocket appServiceSocket;

    public void contextDestroyed(ServletContextEvent e) {

    }

    public void contextInitialized(ServletContextEvent e) {

        if (appServiceSocket == null) {
            appServiceSocket = new AppServiceSocket();
            appServiceSocket.start(); // servlet上下文初始化时启动
socket服务端线程
        }

    }

}
```

接着需要在 WifiServerSocket 类中添加一个发送数据到 APP 的函数：

```
/**
 * 硬件端与服务器端的长连接
 *
 * @author Administrator
 *
 */
```

```
public class WifiServerSocket extends Thread {
    private ServletContext servletContext;
    private ServerSocket serverSocket;

    private static Map<String, ProcessSocketData> socketMap = new
HashMap<>();

    public WifiServerSocket(ServletContext servletContext) {
        this.servletContext = servletContext;

        // 从web.xml中context-param节点获取socket端口
        String port =
this.servletContext.getInitParameter("socketPort");
        if (serverSocket == null) {
            try {

                this.serverSocket = new
ServerSocket(Integer.parseInt(port));
            } catch (IOException e) {
                e.printStackTrace();

            }

        }
    }

    public void run() {

        while (!this.isInterrupted()) { // 线程未中断执行循环

            try {
                // 开启服务器，线程阻塞，等待8266的连接
                Socket socket = serverSocket.accept();
                ProcessSocketData psd = new
ProcessSocketData(socket);
                new Thread(psd).start();

            } catch (IOException e) {
                e.printStackTrace();
            }

        }

    }

    public void closeServerSocket() {

        try {

            if (serverSocket != null && !serverSocket.isClosed())
                serverSocket.close();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    // 将socket连接一静态集合变量的形式暴露出去
    public static Map<String, ProcessSocketData> getSocketMap() {
        return socketMap;
    }

    public class ProcessSocketData extends Thread {
        private Socket socket;
        private InputStream in = null;
        private DataOutputStream out = null;

        private String mStrName = null;
        private boolean play = false;

        // 构造方法，传入连接进来的socket
        public ProcessSocketData(Socket socket) {
            this.socket = socket;
            try {
                in = new
DataInputStream(socket.getInputStream());
                out = new
DataOutputStream(socket.getOutputStream());
            } catch (IOException e) {

                e.printStackTrace();
            }
            play = true;
        }

        public void run() {
            try {
                // 死循环，无线读取8266发送过来的数据
                while (play) {
                    byte[] msg = new byte[10];
                    in.read(msg);// 读取流数据
                    System.out.println("WiFi发过来的数据：" +
Arrays.toString(msg));
                    String str = new String(msg).trim();
                    System.out.println(str);

                    if (str.contains("CONN")) {
                        mStrName = str.trim();
                        /*
                         * 判断发过的是CONN_9527,那么就将此socket对象
添加到这个类的静态集合里面，以CONN_9527为索引。
                         *  很多人这里可能不太懂，APP与服务端的通信在
```

AppControlServlet类中触发，想要实现APP与8266通信，只能将这个socket对象通过类的静态变量暴露出去。

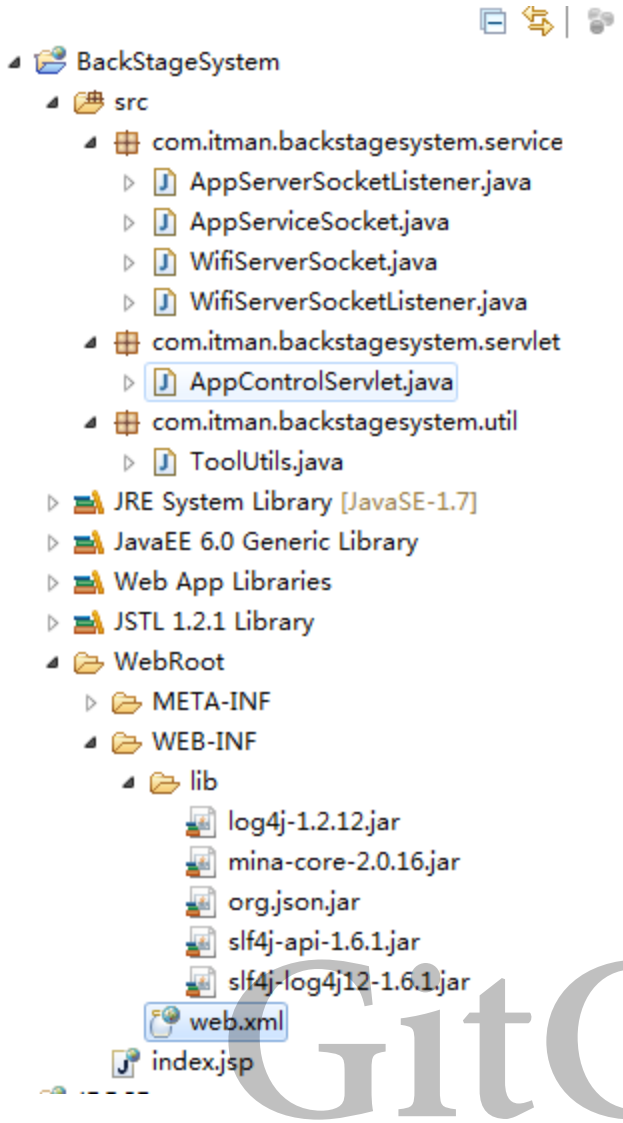
```
                         * 等到AppControlServlet收到APP的信息，就立马通过CONN_9527作为索引取出socket，和8266进行通讯
                         */
                        WifiServerSocket.socketMap.put(mStrName, this);

                    } else {
                        sendToAPP(mStrName, msg);
                    }

                }
            } catch (IOException e) {
                e.printStackTrace();

            } finally {
                try {
                    in.close();
                    if (socket != null && !socket.isClosed()) {
                        socket.close();
                    }

                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        /**
         * 发送数据到APP的方法
         * @param strName
         * @param msg
         */
        private void sendToAPP(String strName, byte[] msg) {
            System.out.println("sessionId:" + strName);

            if (AppServiceSocket.getAcceptorSessions().get(strName) != null) {

                AppServiceSocket.getAcceptorSessions().get(strName)
                        .write(new String(msg));
                System.out.println("已发送给客户端");

            } else {
                System.out.println("客户端没上线");
            }
        }

        // 这是服务器发送数据到8266的函数
        public void send(byte[] bytes) {
```

```
                try {
                    out.write(bytes);
                } catch (IOException e) {
                    try {
                        // 移除集合里面的Socket
                        WifiServerSocket.socketMap.remove(mStrName);
                        out.close();
                        play = false;
                        in.close();
                        if (socket != null && !socket.isClosed()) {
                            socket.close();
                        }
                    } catch (IOException e1) {
                        e1.printStackTrace();
                    }
                    System.out.println("该客户端已退出！");
                }
            }

        }

    }
```

最后给这个与 App 实现长连接的服务器在 web.xml 中增加映射：


```
        <listener>
            <description>AppServerSocket 服务随 web 启动而启动
    </description>
            <listener-
    class>com.itman.backstagesystem.service.AppServerSocketListener</
    listener-class>
        </listener>
```

如果正确完成以上的操作，那么服务器算是基本完成实现了。就剩下 App 端了。来看看服务器端的项目结构：



最后，我们进行 APP 端长连接的实现。

先添加 MINA 的依赖：

```
    implementation('org.apache.mina:mina-core:2.0.7') {
        exclude module: 'slf4j-api'
    }
    implementation 'org.slf4j:slf4j-android:1.6.1-RC1'
```

直接演示源码，长连接的配置：

```
/**
 * Created by Layne_Yao on 2017/10/8.
 * CSDN:http://blog.csdn.net/Jsagacity
 */
public class ConnectionConfig {

    private Context context;
    private String ip;
    private int port;
    private int readBufferSize;
```

```
    private long connectionTimeout;

    public Context getContext() {
        return context;
    }

    public String getIp() {
        return ip;
    }

    public int getPort() {
        return port;
    }

    public int getReadBufferSize() {
        return readBufferSize;
    }

    public long getConnectionTimeout() {
        return connectionTimeout;
    }

    public static class Builder{
        private Context context;
        private String ip = "192.168.2.176";
        private int port = 10011;
        private int readBufferSize = 10240;
        private long connectionTimeout = 10000;

        public Builder(Context context){
            this.context = context;
        }

        public Builder setIp(String ip){
            this.ip = ip;
            return this;
        }

        public Builder setPort(int port){
            this.port = port;
            return this;
        }

        public Builder setReadBufferSize(int readBufferSize){
            this.readBufferSize = readBufferSize;
            return this;
        }

        public Builder setConnectionTimeout(long
connectionTimeout){
            this.connectionTimeout = connectionTimeout;
            return this;
```

```
        }

        private void applyConfig(ConnectionConfig config){

            config.context = this.context;
            config.ip = this.ip;
            config.port = this.port;
            config.readBufferSize = this.readBufferSize;
            config.connectionTimeout = this.connectionTimeout;
        }

        public ConnectionConfig builder(){
            ConnectionConfig config = new ConnectionConfig();
            applyConfig(config);
            return config;
        }
    }
}
```

长连接的管理：

```
/**
 * Created by Layne_Yao on 2017/10/8.
 * CSDN:http://blog.csdn.net/Jsagacity
 */

public class ConnectionManager {

    private static final String BROADCAST_ACTION =
"com.ssy.mina.broadcast";
    private static final String MESSAGE = "message";
    private ConnectionConfig mConfig;
    private WeakReference<Context> mContext;

    private NioSocketConnector mConnection;
    private IoSession mSession;
    private InetSocketAddress mAddress;

    public ConnectionManager(ConnectionConfig config){

        this.mConfig = config;
        this.mContext = new WeakReference<Context>
(config.getContext());
        init();
    }

    private void init() {
        mAddress = new InetSocketAddress(mConfig.getIp(),
mConfig.getPort());
        mConnection = new NioSocketConnector();
```

```
mConnection.getSessionConfig().setReadBufferSize(mConfig.getReadB
ufferSize());
        mConnection.getFilterChain().addLast("logging", new
LoggingFilter());
        mConnection.getFilterChain().addLast("codec", new
ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
        mConnection.setHandler(new
DefaultHandler(mContext.get()));
        mConnection.setDefaultRemoteAddress(mAddress);
    }

    /**
     * 与服务器连接
     * @return
     */
    public boolean connect(){
        Log.e("tag", "准备连接");
        try{
            ConnectFuture future = mConnection.connect();
            future.awaitUninterruptibly();
            mSession = future.getSession();

            SessionManager.getInstance().setSession(mSession);

        }catch (Exception e){
            e.printStackTrace();
            Log.e("tag", "连接失败");
            return false;
        }

        return mSession == null ? false : true;
    }

    /**
     * 断开连接
     */
    public void disConnect(){
        mConnection.dispose();
        mConnection=null;
        mSession=null;
        mAddress=null;
        mContext = null;
        Log.e("tag", "断开连接");
    }

    private static class DefaultHandler extends IoHandlerAdapter
{

        private Context mContext;
        private DefaultHandler(Context context){
            this.mContext = context;
```

```
        }

        @Override
        public void sessionOpened(IoSession session) throws
Exception {
            super.sessionOpened(session);
        }

        @Override
        public void messageReceived(IoSession session, Object
message) throws Exception {
            Log.e("tag", "接收到服务器端消息："+message.toString());
            if(mContext!=null){
                Intent intent = new Intent(BROADCAST_ACTION);
                intent.putExtra(MESSAGE, message.toString());

LocalBroadcastManager.getInstance(mContext).sendBroadcast(intent)
;
            }
        }
    }
}
```

session 的管理：


```
/**
 * Created by Layne_Yao on 2017/10/8.
 * CSDN:http://blog.csdn.net/Jsagacity
 */
public class SessionManager {

    private static SessionManager mInstance=null;

    private IoSession mSession;
    public static SessionManager getInstance(){
        if(mInstance==null){
            synchronized (SessionManager.class){
                if(mInstance==null){
                    mInstance = new SessionManager();
                }
            }
        }
        return mInstance;
    }

    private SessionManager(){}

    public void setSession(IoSession session){
        this.mSession = session;
```

```
    }

    public void writeToServer(Object msg){
        if(mSession!=null){
            Log.e("tag", "客户端准备发送消息");
            mSession.write(msg);
        }
    }

    public void closeSession(){
        if(mSession!=null) {
            mSession.getCloseFuture().setClosed();
        }
    }

    public void removeSession(){
        this.mSession=null;
    }
}
```

最后是在服务中实现长连接：

```
/**
 * Created by Layne_Yao on 2017/10/8.
 * CSDN:http://blog.csdn.net/Jsagacity
 */
public class MinaService extends Service {

    private ConnectionThread thread;


    @Override
    public void onCreate() {
        super.onCreate();

        thread = new ConnectionThread("mina",
getApplicationContext());
        thread.start();
        Log.e("tag", "启动线程尝试连接");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int
startId) {
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
```

```
            thread.disConnect();
            thread = null;

        }

        @Nullable
        @Override
        public IBinder onBind(Intent intent) {
            return null;
        }

        class ConnectionThread extends HandlerThread {

            private Context context;
            boolean isConnection;
            ConnectionManager mManager;

            public ConnectionThread(String name, Context context) {
                super(name);
                this.context = context;
                ConnectionConfig config = new
ConnectionConfig.Builder(context)
                        .setIp("192.168.2.176")
                        .setPort(10011)
                        .setReadBufferSize(10240)
                        .setConnectionTimeout(10000).builder();

                mManager = new ConnectionManager(config);
            }

            @Override
            protected void onLooperPrepared() {
                while (true) {
                    isConnection = mManager.connect();
                    if (isConnection) {
                        String macId = "CONN_9527";
                        Log.e("tag", "连接成功");
                        Log.e("SendAsyncTask", "设备id:" + macId);

SessionManager.getInstance().writeToServer(macId);
                        break;
                    }
                    try {
                        Log.e("tag", "尝试重新连接");
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }

            public void disConnect() {
```

```
                mManager.disConnect();
            }
        }
    }
```

别忘了，Android 四大组件都是要注册的：

```
    <service android:name=".service.MinaService"/>
```

直接贴出 MINA 的使用代码：

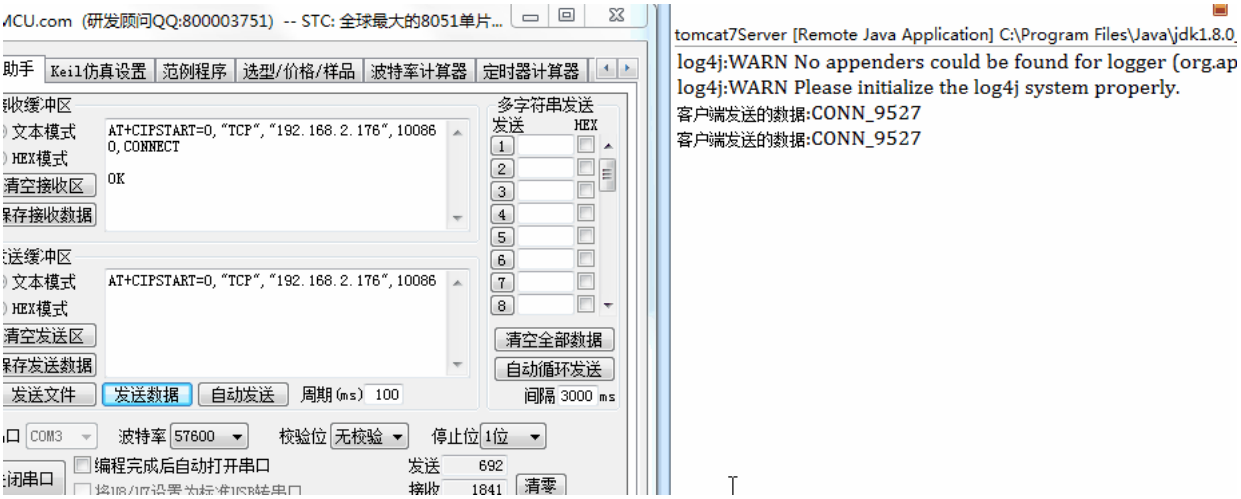
```
public class MainActivity extends AppCompatActivity implements
View.OnClickListener {
    private Button btnSend;

    private Intent serviceIntent;
    private MessageBroadcastReceiver receiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnSend = findViewById(R.id.btnSend);
        btnSend.setOnClickListener(this);

        //开启长连接服务
        serviceIntent = new Intent(this, MinaService.class);
        startService(serviceIntent);

        registerBroadcast();
    }

    @Override
    public void onClick(View view) {
        switch (view.getId()){
            case R.id.btnSend:
                byte[] msg = new byte[]{(byte) 0x01, (byte) 0x02,
(byte) 0x03};
                String name = "layne";
                String sessionId = "CONN_9527";
                //发出请求
                serviceCONN(msg,name,sessionId);
                break;
        }
    }

    private void serviceCONN(byte[] bytes, String username,
String sessionId) {
        //将byte数组转化成字符串
```

```
        String data_msg = Arrays.toString(bytesToInt(bytes));
        //服务器的url
        String url =
"http://192.168.2.176:8080/BackStageSystem/servlet/AppControlServ
let";
        //将数据拼接起来
        String data = "username=" + username + "&sessionId=" +
sessionId + "&data=" + data_msg;
        String[] str = new String[]{url, data};

        //发出一个请求
        new HttpAsyncTask(MainActivity.this, new
HttpAsyncTask.PriorityListener() {

            @Override
            public void setActivity(int code) {
                switch (code) {
                    case 200:
                        //如果返回的resultCode是200,那么说明APP的数据
传送成功，并成功解析返回的json数据
                        Toast.makeText(MainActivity.this, "发送数
据：[0x01,0x02,0x03]", Toast.LENGTH_SHORT).show();
                        break;
                    case 202:
                        Toast.makeText(MainActivity.this, "设备离
线状态", Toast.LENGTH_SHORT).show();
                        break;
                    default:
                        Toast.makeText(MainActivity.this, "网络传
输异常", Toast.LENGTH_SHORT).show();
                        break;
                }

            }
        }).execute(str);
    }

    /**
     * byte转化为int
     */
    public static int[] bytesToInt(byte[] src) {
        int value[] = new int[src.length];
        for (int i = 0; i < src.length; i++) {
            value[i] = src[i] & 0xFF;
        }
        return value;
    }

    //动态注册广播
    private void registerBroadcast() {
        receiver = new MessageBroadcastReceiver();
        IntentFilter filter = new
```

```
IntentFilter("com.ssy.mina.broadcast");

LocalBroadcastManager.getInstance(this).registerReceiver(receiver
, filter);
    }

    //接收发送的广播
    private class MessageBroadcastReceiver extends
BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {

            String msg = intent.getStringExtra("message");
            Toast.makeText(MainActivity.this, "esp8266发送过来的数
据："+msg, Toast.LENGTH_SHORT).show();
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        //退出时关掉长连接服务
        stopService(serviceIntent);
    }
}
```


最后说明一下，广播虽然也是四大组件之一，但是以上是动态注册的。还有长连接是在服务中接收的，不在主线程中的，所以服务中接收到消息，想通知主线程，可以通过 handler 消息机制，或者像以上的操作，使用广播机制来通知主线程。

好了到此为止，APP 与 8266 的远程通讯基本实现，演示一下：



真机中收到 8266 发送的消息：

经测试，已成功实现，不过现在还不是真正意义上的远程，全部的实现需要在局域网中才可以实现的。这是因为我们的服务端 JavaWeb 项目还没有部署在云服务器，所以就只能在局域网中测试。

接下来我们将学习如何使用阿里云服务器 ECS 进行配置环境，然后将 JavaWeb 项目部署上去。

购买阿里云服务器 ECS，并配置部署环境

这次买的阿里云是入门级中最便宜的配置。

系统的话，如果是新手就选 Windows 2012 Server，因为 Linux 不是初学者能玩的，另外如果买了最小配置的，跑 2012 系统是非常吃力的，是需要开启虚拟内存的，这是阿里云里面的详细解决办法：配置 Windows 系统虚拟内存。

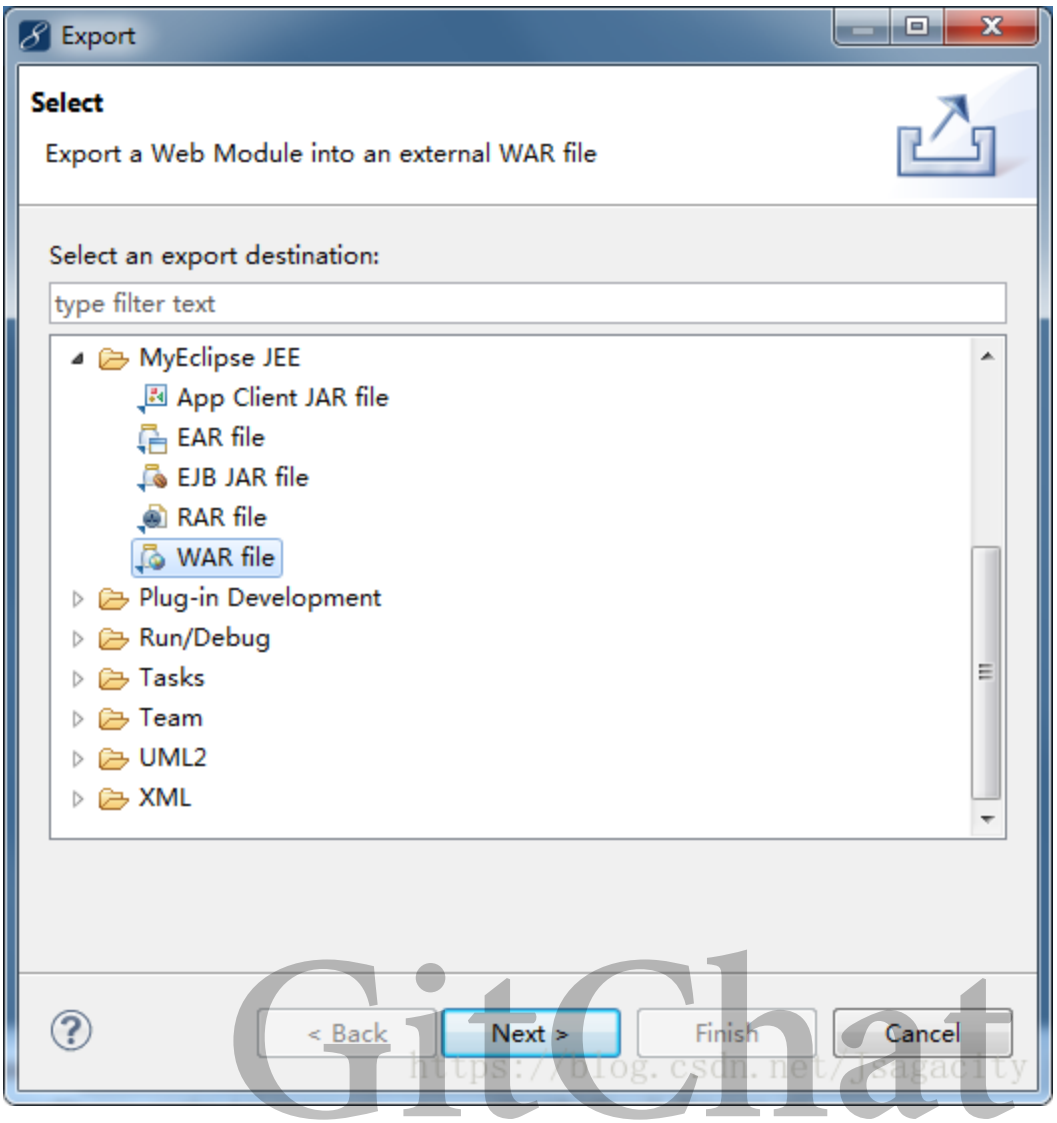
配置完虚拟内存之后，就可以安装 JDK 和 Tomcat 了，**注意：得先安装 JDK，并且是安装 JDK1.7 以下，最好就 JDK1.7。**然后再装 Tomcat，安装包文末有资源链接。JDK 和 Tomcat 的安装步骤这里就不演示了，网上大把教程，有了安装包，照着教程一步步来，基本没什么问题的。

以上的我是踩过无数的坑才得出来的，我从事 Android 开发，服务器我也不是很精通，所以就只能用 Windows 2012 Server，我试过转换成 Linux，一进去就是全程指令操作，我当时是懵逼的。所以没办法又换回 Windows。

刚开始用起来一直很卡，时而行，时而不行，用了好几天没办法，才问客服。后来才得知需要开虚拟内存。至于 JDK 和 Tomcat 我都不知道装了多少次了，最后才得出只能装 JDK1.7 以下。以上这些点，希望你在实操的时候多多注意。

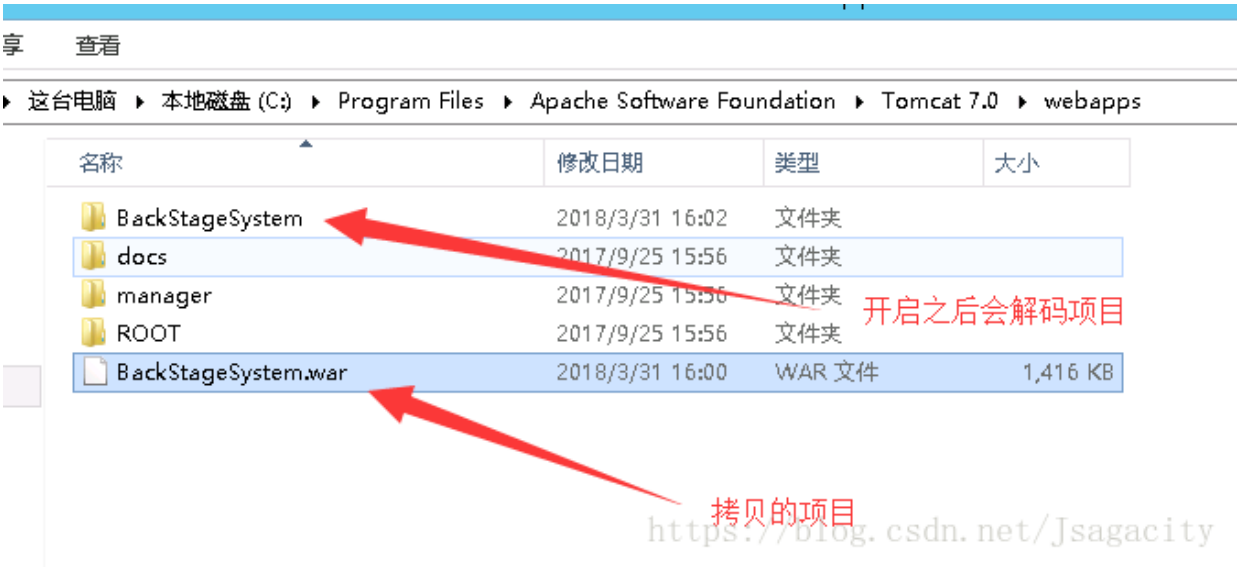
部署 JavaWeb 项目到云服务器

首先将 JavaWeb 项目以 war 包的方式导出：



导出之后：

导出之后将项目拷贝至 Tomcat 安装目录下的 webapps 文件夹下，运行 Tomcat，这样云服务器就算开启了：



服务器开启之后还不算完成，还需要在阿里云控制台中，进入你购买的这个服务器实例，找到安全组配置，在配置中加入三个端口，这三个端口分别是 APP 长连接服务器的端口、8266 长连接服务器的端口和云服务器 Tomcat 的端口：

配置完以上的云服务器才算是真正的启动了。接下来 8266 连接服务器的指令就要修改 IP 了：

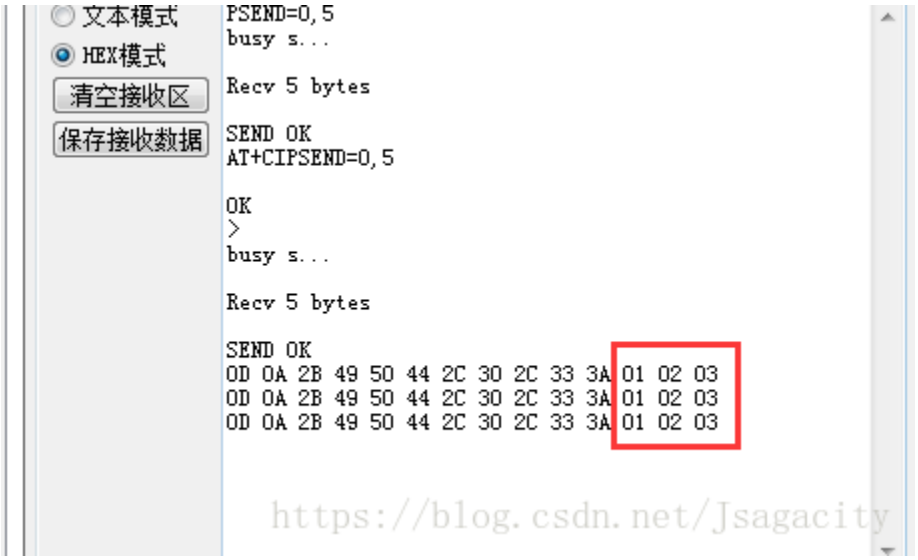
1. AT+CWMODE=1
2. AT+RST
3. AT+CWJAP=”SSY”,”gzsssydzkjyxgs2016.”
4. AT+CIPMUX=1
5. AT+CIPSERVER=1,8800
6. AT+CIPSTART=0,”TCP”,”47.95.15.238”,10086 //此IP修改成云服务器的公网IP
7. AT+CIPSEND=0,9
8. CONN_9527

完了修改 APP 里面的 IP 地址，在 ConnectionConfig.java、MinaService.java 和 MainActivity.java 中的请求链接那里。修改完之后运行到真机，测试一下是否能成功通信。

将 Android 项目运行到真机之后会看到这样一个 toast，说明 APP 长连接服务器成功：

完了之后操作 8266 连接服务器并发送 layne 给 App，也是能正常收到的：

App 端点击发送，向 8266 发送 [1,2,3] 数组，同样的也是能正常收到的：



好了到此真正意义上的 APP 和 8266 远程通信就这样实现了，这是最简化的实现了。接下来你们就可以将它扩展成自己的项目实现了。

大家可以在这里获得：相关源码和 jar 包。

同时，也欢迎大家关注微信公众号“Layne的编译”（微信号：LayneYao）与我交流。