Android 应用软件设计

E 6　XML or JSON

学号：SA17225263

姓名：潘梦泽

报告撰写时间：2017/11/05

1.主题概述

1. 下载 tomcat 并安装

安装好 tomcat 并配置环境，在下列工程里引入相关依赖。

2. 新建工程 SCOSServer，定义源码包

使用 Eclipse(本实验用的是 idea)创建工程 SCOSServer，并定义源码“esd.scos.servlet”

3. 在 SCOSServer 包 esd.scos.servlet 下新建类 LoginValidator 继承 HttpServlet。

1）实现 doPost()方法完成 SCOS 客户端登录请求传入的用户名与密码验证，当验证成功，返回 JSON 串“{RESULTCODE:1}”；否则返回“{RESULTCODE:0}”

2）实现 doGet()方法，并在该方法中调用 doPost()

4. 在 SCOSServer 包 esd.scos.servlet 下新建类 FoodUpdateService 继承 HttpServlet。

1）实现 doGet()方法，当 SCOS 客户端请求菜品信息更新时，实现菜品更新信息发送

2）使用 JSON 封装菜品更新信息内容：更新菜品数量，每个菜品名称，每个菜品价格，每个菜品类型

3）将菜品更新信息以流的形式发送至 SCOS 客户端

4）实现 doPost()方法，并在该方法中调用 doGet()

5. 修改 LoginOrRegister 代码实现登录注册功能。

点击登录或注册按钮时，用 HttpURLConnection 访问 SCOSServer 的 Servlet 类 LoginValidator，并传入用户输入的用户名和密码，接收信息，code 为 1 时登录成功，否则失败。原有提示保留。

6. 修改 UpdateService 代码实现 get 请求并解析菜品数据，播放提示音，显示提示信息。

使用 MediaPlayer 播放更新提示音，并使用 NotificationManager 在状态栏提示用户“新品上架：菜品数量”，通知中含有“清除”按钮，当点击清除按钮时，通知消除；当点击通知其他区域时，页面跳转至 SCOS 的 MainScreen 屏幕。

7. 改用 Xml 封装数据，再次实现上述任务，并比较两种方式的区别

请统计： JSON 流的长度 VS XML 流的长度，生成 JSON 时间 VS XML 生成时间，JSON 解析时间 VS XML 解析时间：

1）当更新菜品数量为 100 时

2）当更新菜品数量为 10000 时

3）当更新菜品数量为 100000 时

8. 调试成功后，编译打包。

打包为 4.0 的版本。

2.假设

本应用总体目标是实现一款订餐软件，用来代替传统纸质订餐和电话订餐。

本次作业主要的练习目标是构建一个 servlet 服务端，客户端通过 post 和 get 请求实现与 servlet 的通信。并尝试了 xml 和 json 两种数据封装和解析的方式，通过实验来比较两种方式的优劣性。同时简单地使用了 Mediaplayer，了解了 android 是如何进行媒体加载和播放的。

3.实现或证明

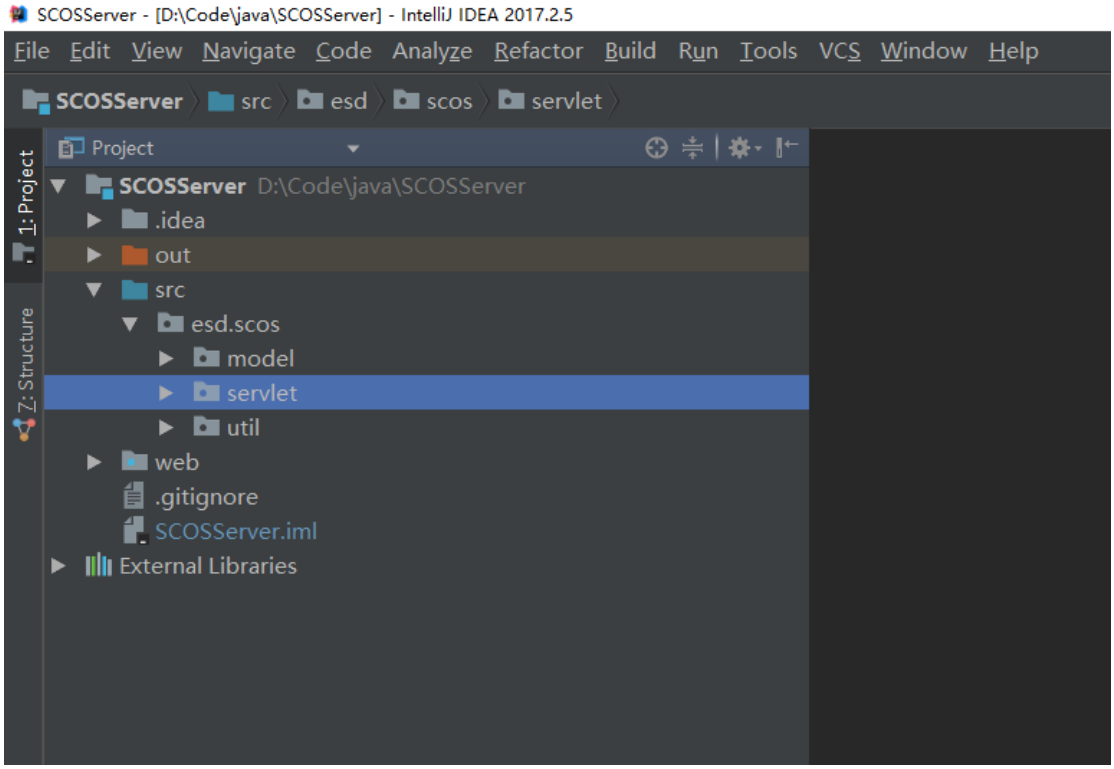
1. Github:

https://github.com/panmengze1991/SCOS

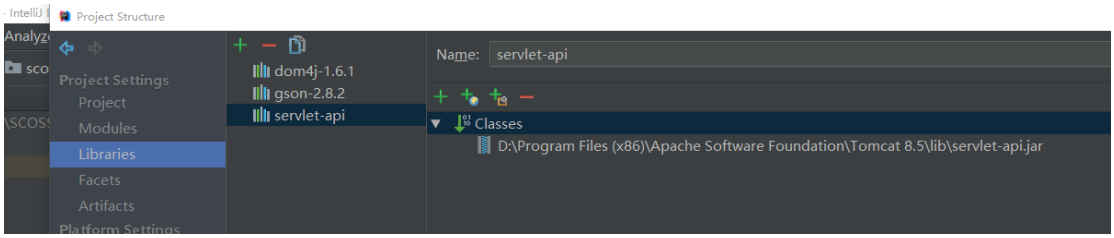
2. 新建 java 工程，并在其中建立 servlet，安装 tomcat，在 java 工程中引入相关依赖和配置

1. Tomcat 的下载比较简单，官网链接 https://tomcat.apache.org/download-80.cgi 即可下载相应版本，安装即可。安装好之后将其目录下的 lib 和 bin 目录配置到环境变量中的 path 中。

2. 新建 java 工程，创建 esd.scos.servlet 子包



3. 之后，在 File->Project Structure 的 Libraries 中添加 tomcat lib 文件夹下的 servlet-api.jar 包，如此，我们得以开始建立 servlet。



3. 在 SCOSServer 包 esd.scos.servlet 下新建类 LoginValidator 继承 HttpServlet。

1. 实现 doPost()，验证传过来的用户名和密码。首先使用 BufferedReader 将内容读取到 StringBuilder 对象中，再用 Gson 包解析生成的字符串对象（后来发现 Gson 包也可以直接解析 BufferedReader 对象），将结果保存到我们定义的登录参数中。

```
response.setContentType("application/json;charset=utf-8");
request.setCharacterEncoding("utf-8");

StringBuilder stringBuilder = new StringBuilder();
String line;
LoginParam loginParam;
try {
    //读取输入流到StringBuilder中
    BufferedReader reader = request.getReader();
    while ((line = reader.readLine()) != null)
        stringBuilder.append(line);

} catch (Exception e) { /*report an error*/ }

try {
    Gson gson = new Gson();
    loginParam = gson.fromJson(stringBuilder.toString(), LoginParam.class);
} catch (Exception e) {
    throw new IOException("Error parsing JSON request string");
}
```

登录参数代码如下(getter、setter、构造方法略)：

```
public class LoginParam {
    // 用户名
    private  String userName;
    //  密码
    private  String password;
```

2. 判断是否正确（这里自定的规则是不为空即可）并返回 json 数据。

```
ResultBody resultBody = new ResultBody( RESULTCODE: 0,  msg: "登陆失败",  dataString: null);
if (loginParam != null && CommonUtils.isNotEmpty(loginParam.getUserName()) && CommonUtils.isNotEmpty
        (loginParam.getPassword())) {
    resultBody.setRESULTCODE(1);
    resultBody.setMsg("登录成功");
}

String responseString = new Gson().toJson(resultBody);
CommonUtils.putDataToResponse(response, responseString);
```

3. 返回的参数也定义了一个通用的对象格式，结构如下:

```
public class ResultBody {
    // 状态值
    private  int RESULTCODE;
    // 消息
    private  String msg;
    // 携带对象字符串
    private  String dataString;
```

因此返回的时候我们只需要将其封装好传回去即可，方便客户端统一解析。

返回对象构造好之后转为 Json 字符串，再通过如下工具类中的方法写入输出流中即可。

```
// 添加字符串到输出流中
public static void putDataToResponse(HttpServletResponse response, String jsonValue) {
    PrintWriter out = null;
    try {
        out = response.getWriter();
        out.write(jsonValue);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (out != null) {
            out.close();
        }
    }
}
```

4. 实现 doGet()调用 doPost()

```
protected void doGet(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse
        response) throws javax.servlet.ServletException, IOException {
    doPost(request, response);
}
```

至此登录业务的客户端代码已经完成。

4. 在 SCOSServer 包 esd.scos.servlet 下新建类 FoodUpdateService 继承 HttpServlet。使用 Json 或 Xml 封装菜品数据进行发送。

1. 实现 doGet()。

这里获取了请求内容格式来决定通过 Json 还是 Xml 进行数据封装，doGet 代码如下：

```
protected void doGet(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse
        response) throws javax.servlet.ServletException, IOException {

    // 判断发送数据内容
    Type = request.getContentType().equals("applicaiton/json")?TYPE_JSON:TYPE_XML;

    request.setCharacterEncoding("utf-8");

    List<Food> foodList = CommonUtils.getFoodList( amount: 1000);

    if (Type == TYPE_JSON) {
        // send json
        String foodListJson = new Gson().toJson(foodList);
        response.setContentType("application/json;charset=utf-8");
        ResultBody resultBody = new ResultBody( RESULTCODE: 1,  msg: "请求成功", foodListJson);
        String resultJson = new Gson().toJson(resultBody);
        CommonUtils.putDataToResponse(response, resultJson);
    } else if (Type == TYPE_XML) {
        // send xml
        Element dataString = CommonUtils.parseFoodListToXml(foodList);
        response.setContentType("text/xml; charset=utf-8");
        Document foodDoc = DocumentHelper.createDocument();
        Element result = foodDoc.addElement( s: "Result");
        Element resultCode = result.addElement( s: "RESULTCODE");
        resultCode.addText("1");
        Element message = result.addElement( s: "message");
        message.addText("请求成功");
        result.add(dataString);
        CommonUtils.putDataToResponse(response, foodDoc.asXML());
    }
}
```

首先，构造菜品列表，通过参数控制个数，方便下面封装数据调用：

```
// 获取食物列表
public static List<Food> getFoodList(int amount) {
    List<Food> foodList = new ArrayList<>();
    for (int i = 0; i < amount; i++) {
        foodList.add(new Food( foodName: "酸辣土豆丝",  price: 30,  store: 1,  order: false,  imgId: 0));
    }
    return foodList;
}
```

2. Json 格式代码封装：

这里的思路很简单，将 foodList 通过 Gson 转为 String，放到 ResultBody 的 dataString 域中，再将 ResultBody 转为 String 写入到输出流中，写入方法和登录返回的一样。

```
if (Type == TYPE_JSON) {
    // send json
    String foodListJson = new Gson().toJson(foodList);
    response.setContentType("application/json;charset=utf-8");
    ResultBody resultBody = new ResultBody( RESULTCODE: 1,  msg: "请求成功", foodListJson);
    String resultJson = new Gson().toJson(resultBody);
    CommonUtils.putDataToResponse(response, resultJson);
```

3. Xml 格式代码封装：

Xml 的封装就麻烦的多，这里我首先要通过菜品列表构造一个菜品结点组，这里使用了 dom4j 包，具体的思想是定义一个 dataString 根节点（为了保持命名一致，实际上不应该叫 String），然后遍历列表，建立 food 结点，再给 food 结点添加子结点，包括各项属性：

```
// 获取食物列表
public static Element parseFoodListToXml(List<Food> foodList) {
    Document foodDoc = DocumentHelper.createDocument();
    Element dataString = foodDoc.addElement( s: "dataString");
    for (Food food : foodList) {
        Element foodElement = dataString.addElement( s: "food");
        // 子节点
        Element foodName = foodElement.addElement( s: "foodName");
        foodName.addText(food.getFoodName());
        Element price = foodElement.addElement( s: "price");
        price.addText(String.valueOf(food.getPrice()));
        Element store = foodElement.addElement( s: "store");
        store.addText(String.valueOf(food.getStore()));
        Element order = foodElement.addElement( s: "order");
        order.addText(String.valueOf(food.isOrder()));
        Element imgId = foodElement.addElement( s: "imgId");
        imgId.addText(String.valueOf(food.getImgId()));
    }
    return dataString;
}
```

然后添加和 dataString 同级的结点，如 resultcode 和 message，将三个结点一起添加到 result 结点中，通过 asXML()获取字符串，写入到输出流中。

```
} else if (Type == TYPE_XML) {
    // send xml
    Element dataString = CommonUtils.parseFoodListToXml(foodList);
    response.setContentType("text/xml; charset=utf-8");
    Document foodDoc = DocumentHelper.createDocument();
    Element result = foodDoc.addElement( s: "Result");
    Element resultCode = result.addElement( s: "RESULTCODE");
    resultCode.addText("1");
    Element message = result.addElement( s: "message");
    message.addText("请求成功");
    result.add(dataString);
    CommonUtils.putDataToResponse(response, foodDoc.asXML());
}
```

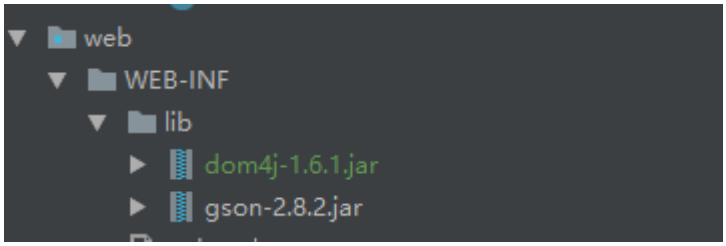
这样就可以达到 Xml 封装的效果，包含了 Result 对象的三个子结点，方便客户端进行解析。

5. 配置 Servlet

配置 web.xml：

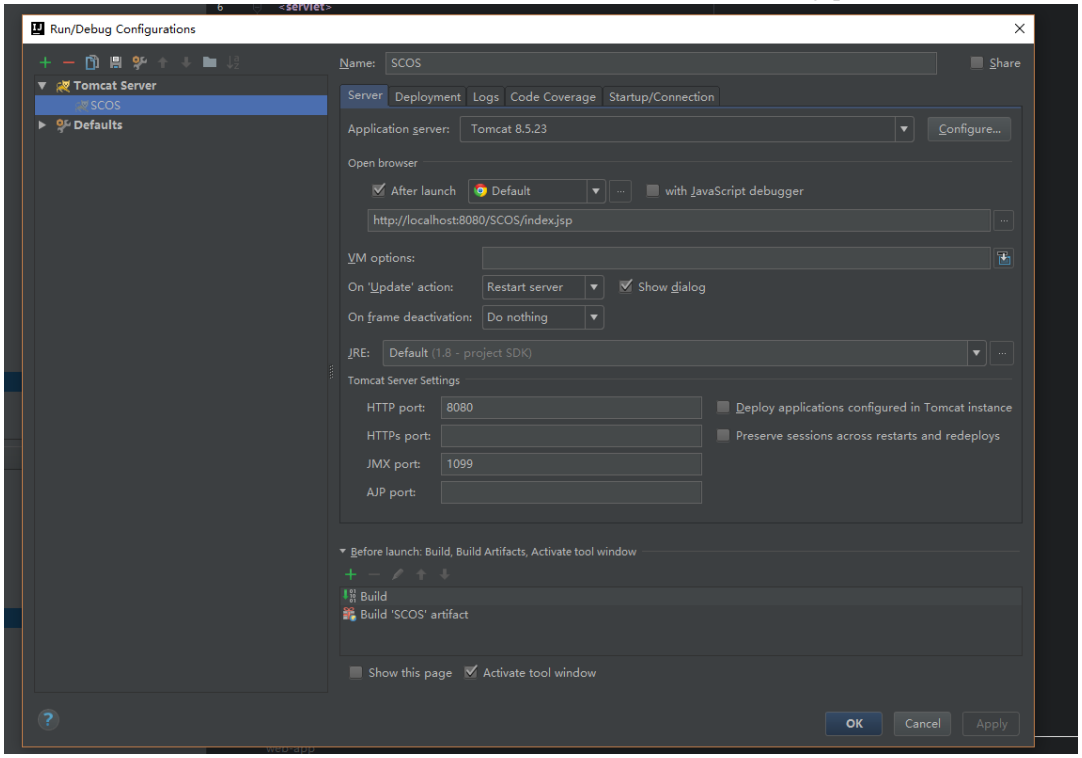
```
    version="3.1">
  <servlet>
    <servlet-name>LoginValidator</servlet-name>
    <servlet-class>esd.scos.servlet.LoginValidator</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>FoodUpdateService</servlet-name>
    <servlet-class>esd.scos.servlet.FoodUpdateService</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>LoginValidator</servlet-name>
    <url-pattern>/Login</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>FoodUpdateService</servlet-name>
    <url-pattern>/Food</url-pattern>
  </servlet-mapping>
</web-app>
```

导入 jar 包：

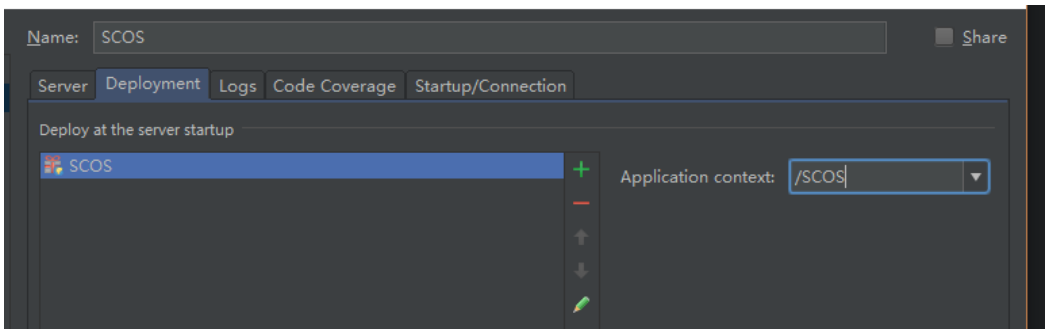


这里需要和 servlet-api 一样去 Project Structure 中进行配置，加上这两个 jar 包，即可正常使用。

配置启动（路径里多加了一个 SCOS），启动时会弹出一个默认的 jsp，访问相应地址即可：

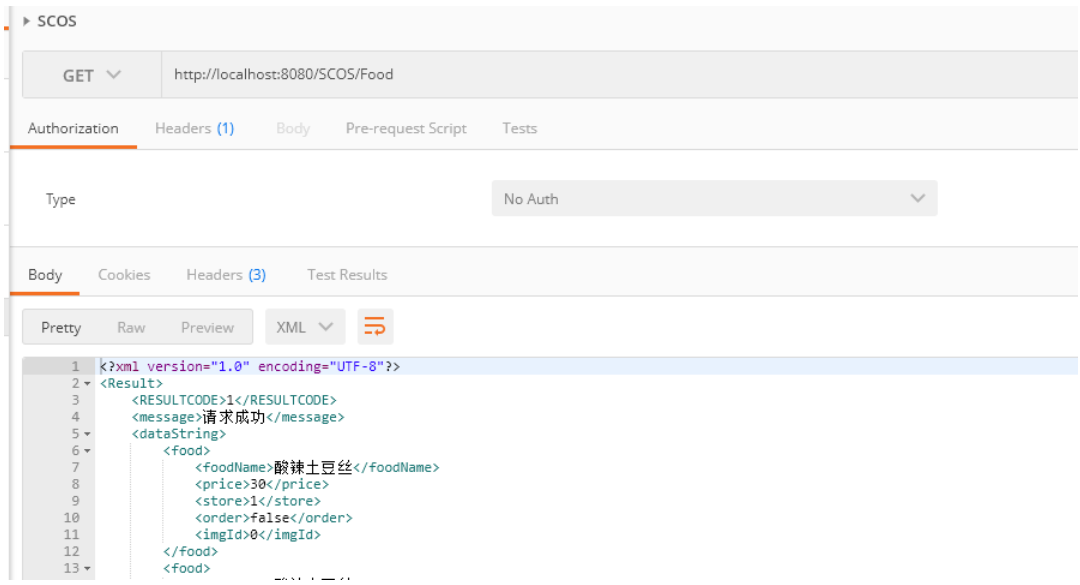


注意此时要在应用上下文里加上自己的路径才会正确访问到。

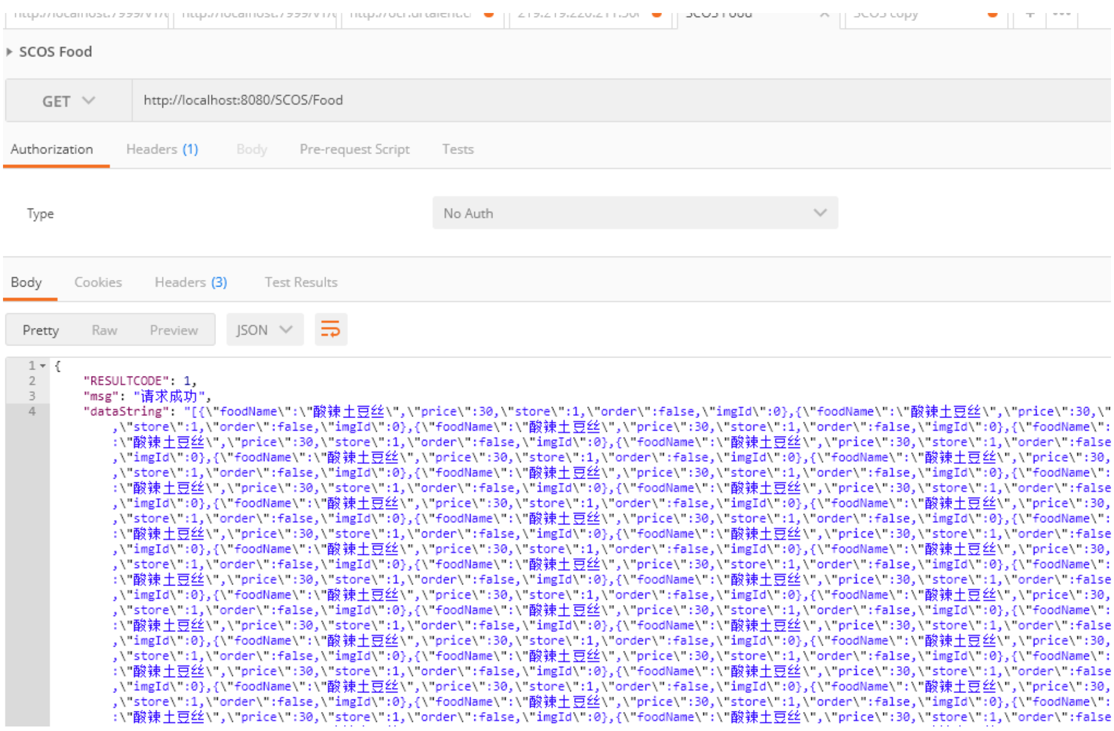


通过 postman 进行测试，没有问题则说明 servlet 已完成开发：

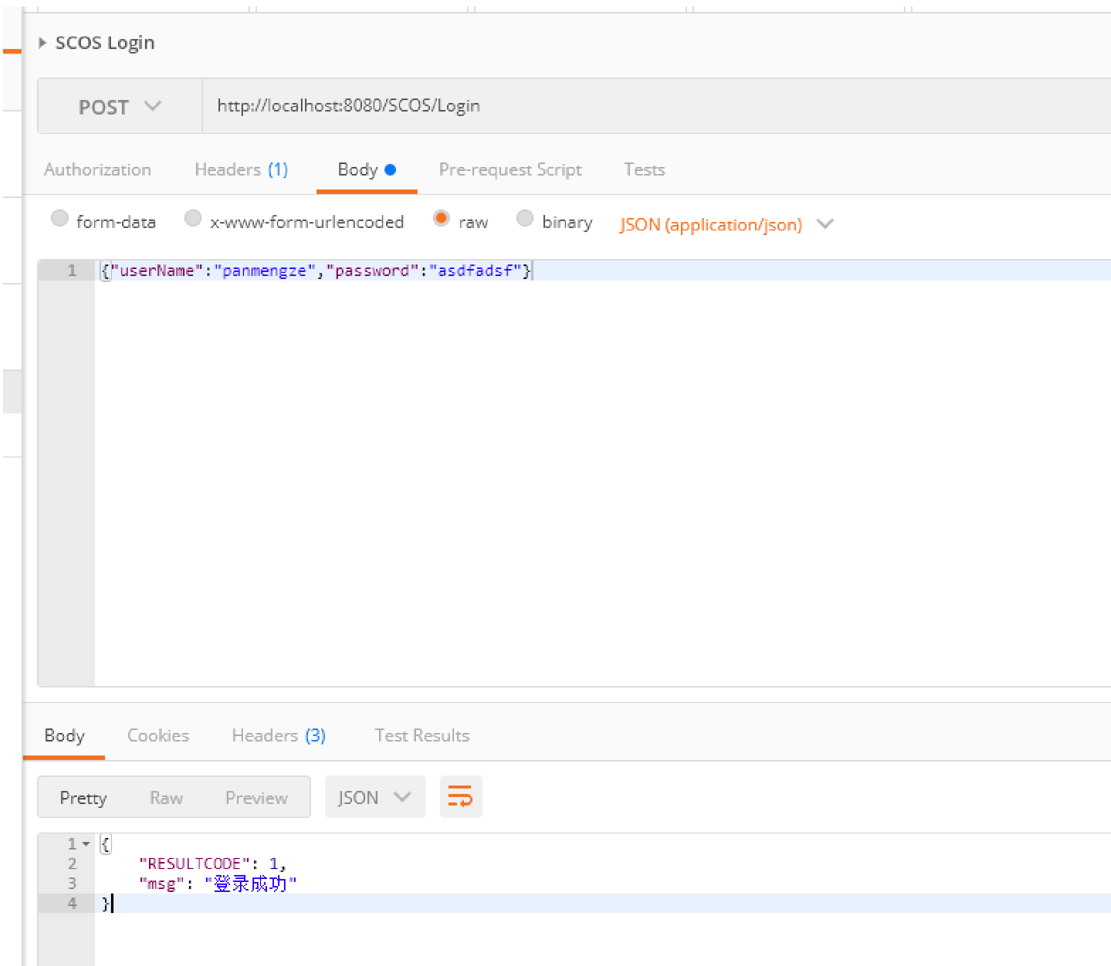
Get 请求 Xml 没有问题：



Get 请求 Json 没有问题：



Post 请求也没有问题：



6. 修改 LoginOrRegister 代码实现登录注册功能。

业务代码如下：

```
/**
 * author:Daniel
 * description: 注册登录
 */
private void doLoginOrRegister(final boolean oldUser) {
    final String name = etName.getText().toString();
    final String password = etPassword.getText().toString();
    showProgress("登录中...");
    if (startValid()) {
        Observable
                .create((e) → {
                    String resultString = CommonUtil.requestPost(new LoginParam(name, password), Const.URL
                            .LOGIN);
                    ResultJson resultJson = new Gson().fromJson(resultString, ResultJson.class);
                    e.onNext(resultJson);
                })
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe((Consumer) (result) → {
                    dismissProgress();
                    if (result == null) {
                        showToast("请求失败");
                    } else {
                        showToast(result.getMsg());
                        if (result.getRESULTCODE() == 1) {
                            // 登陆成功
                            User user = new User(etName.getText().toString().trim(), etPassword.getText()
                                    .toString().trim(), oldUser);
                            App.getInstance().setUser(user).setLoginStatus(Const.SharedPreferenceValue
                                    .LOGIN_SUCCESS);
                            Intent intent = new Intent(mContext, MainScreen.class);
                            intent.putExtra(Const.IntentKey.LOGIN_STATUS, oldUser ? Const.IntentValue
                                    .LOGIN_SUCCESS : Const.IntentValue.REGISTER_SUCCESS);
                            setResult(Const.ActivityCode.LOGIN_OR_REGISTER, intent);
                            finish();
                        }
                    }
                });
    } else {
        dismissProgress();
        showToast("您的输入内容不规范，请输入英文字母和数字组成的账户名和密码");
    }
}
```

这里的思路为：

1. 显示登录加载对话框。

2. 判断用户名密码是否符合规范，判断方法和之前的作业相同，如果不符合，弹出提示，取消对话框。

```
/**
 * author:Daniel
 * description: 校验字符串是否符合正则表达式
 */
private boolean startValid() {
    return pattern.matcher(etName.getText().toString().trim()).matches() && pattern.matcher(etPassword.getText()
            .toString().trim()).matches();
}
```

3. 使用 RxJava 开启 io 线程执行 http 请求。这里是在 CommonUtil 中定义了 requestPost 方法：

```
/**
 * author:      Daniel
 * description: 返回result的JsonString
 */
public static <T extends Param> String requestPost(T param, String urlString) {
    try {
        String postUrl = Const.URL.BASE + urlString;

        // 构造参数json字符串
        String paramString = new Gson().toJson(param);
        // 请求的参数转换为byte数组
        byte[] postData = paramString.getBytes("utf8");

        // 新建一个URL对象
        URL url = new URL(postUrl);
        // 打开一个HttpURLConnection连接
        HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
        // 设置连接超时时间
        urlConnection.setConnectTimeout(5 * 1000);
        //设置从主机读取数据超时
        urlConnection.setReadTimeout(5 * 1000);
        // Post请求必须设置允许输出 默认false
        urlConnection.setDoOutput(true);
        //设置请求允许输入 默认是true
        urlConnection.setDoInput(true);
        // Post请求不能使用缓存
        urlConnection.setUseCaches(false);
        // 设置为Post请求
        urlConnection.setRequestMethod("POST");
        //设置本次连接是否自动处理重定向
        urlConnection.setInstanceFollowRedirects(true);
        // 配置请求Content-Type
        urlConnection.setRequestProperty("Content-Type", "application/json");
        // 开始连接
        urlConnection.connect();
        // 发送请求参数
        DataOutputStream dos = new DataOutputStream(urlConnection.getOutputStream());
        dos.write(postData);
        dos.flush();
        dos.close();
```

方法的前半段为发送请求，主要是将对象参数转为 json 字符串，再转为参数比特流，构造 HttpURLConnection 对象进行连接，然后构造数据发送流对象，写入参数比特流。

后半段则是接收处理，判断请求成功与否，如果成功，返回从流转换为字符串的数据，如果失败，打印 Log，最后关闭连接：

```
// 判断请求是否成功
if (urlConnection.getResponseCode() == 200) {
    // 获取返回的数据
    return CommonUtil.streamToString(urlConnection.getInputStream());
} else {
    Log.d(TAG, "请求失败");
}
// 关闭连接
urlConnection.disconnect();
```

流转为字符串的方法（这里顺便统计了 Json 字节流的长度，后面能用上）：

```
/**
 * author:      Daniel
 * description: 从网络获取的输入流转为String
 */
public static String streamToString(InputStream is) {
    try {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        byte[] buffer = new byte[1024];
        int len;
        while ((len = is.read(buffer)) != -1) {
            outputStream.write(buffer, 0, len);
        }
        outputStream.close();
        is.close();
        Log.d(TAG, "Json test outputStream.size() = " + outputStream.size());
        byte[] byteArray = outputStream.toByteArray();
        return new String(byteArray);
    } catch (Exception e) {
        Log.e(TAG, e.toString());
        return null;
    }
}
```

4. 拿到数据后转为封装好的对象，发送事件给主线程订阅者处理。

```
ResultJson resultJson = new Gson().fromJson(resultString, ResultJson.class);
e.onNext(resultJson);
```

5. 订阅者接到数据，取消对话框，判断非空，如空弹出提示，否则解析数据，toast 显示返回的 msg 内容，执行之前的登录逻辑。

```
dismissProgress();
if (result == null) {
    showToast("请求失败");
} else {
    showToast(result.getMsg());
    if (result.getRESULTCODE() == 1) {
        // 登陆成功
        User user = new User(etName.getText().toString().trim(), etPassword.getText()
                .toString().trim(), oldUser);
        App.getInstance().setUser(user).setLoginStatus(Const.SharedPreferenceValue
                .LOGIN_SUCCESS);
        Intent intent = new Intent(mContext, MainScreen.class);
        intent.putExtra(Const.IntentKey.LOGIN_STATUS, oldUser ? Const.IntentValue
                .LOGIN_SUCCESS : Const.IntentValue.REGISTER_SUCCESS);
        setResult(Const.ActivityCode.LOGIN_OR_REGISTER, intent);
        finish();
    }
}
```

如此，客户端 login 业务的流程基本就写完了，经测试可以完整实现 login 流程。

7. 修改 UpdateService 代码实现 get 请求并解析菜品数据，播放提示音，显示提示信息。

Get 请求业务代码如下，依然通过 RxJava 开启 io 线程完成：

```
 * author:     Daniel
 * description: 获取服务端数据
 */
private void getServerUpdate() {
    Observable.create((e) -> {
        if (Type == TYPE_JSON) {
            // json
            InputStream resultStream = CommonUtil.requestGet(null, Const.URL.FOOD, "application/json");
            if (resultStream == null) {
                // 取消业务
                e.onComplete();
            }
            String resultString = CommonUtil.streamToString(resultStream);
            Log.d(TAG, "request = " + resultString);
            ResultJson resultJson = new Gson().fromJson(resultString, ResultJson.class);
            String foodString = resultJson.getDataString();
            Type type = new TypeToken<ArrayList<Food>>() {
            }.getType();

            // 解析列表统计时间
            Date startDate = new Date(System.currentTimeMillis());
            List<Food> foodList = new Gson().fromJson(foodString, type);
            Date endDate = new Date(System.currentTimeMillis());
            long duration = endDate.getTime() - startDate.getTime();
            Log.d(TAG, "Json test parse time = " + String.valueOf(duration) + "ms , size = " + String
                    .valueOf(foodList.size()));
            e.onNext(foodList);
        } else if (Type == TYPE_XML) {
            // xml
            InputStream resultStream = CommonUtil.requestGet(null, Const.URL.FOOD, "text/xml");
            if (resultStream == null) {
                // 取消业务
                e.onComplete();
            }
            ResultXml resultXml = CommonUtil.getResultFromXml(CommonUtil.streamToXml(resultStream));
            e.onNext(resultXml.getDataList());
        } else {
            e.onComplete();
        }
    })
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
```

这里的思路是：

1. 通过初始化服务时定义的请求类型，判断接收 Json 或是 Xml 的数据。

```
private static final int TYPE_JSON = 1;
private static final int TYPE_XML = 2;
private static int Type;
```

```
@Override
protected void onHandleIntent(@Nullable Intent intent) {
    Log.d(TAG, "onHandleIntent");
//      getFoodCollection();
    mContext = this;
    // 这里修改类型
//      Type = TYPE_JSON;
    Type = TYPE_XML;
    getServerUpdate();
}
```

2. 构造 Get 请求方法 CommonUtil.requestGet:

由于这里不需要传参数了，省略了构造参数的部分，如果有就在 url 后面跟上?和&构造的参数字符串即可。通过请求 Content-Type 来标记后台返回的类型，以便于服务端选择构造 Xml 还是 Json 进行返回。

```
/**
 * author:      Daniel
 * description: 返回result的JsonString
 */
public static InputStream requestGet(String param, String urlString, String contentType) {
    try {
        String postUrl = Const.URL.BASE + urlString;

        // 新建一个URL对象
        URL url = new URL(postUrl);
        // 打开一个HttpURLConnection连接
        HttpURLConnection urlConn = (HttpURLConnection) url.openConnection();
        // 设置连接主机超时时间
        urlConn.setConnectTimeout(5 * 1000);
        //设置从主机读取数据超时
        urlConn.setReadTimeout(5 * 1000);
        // 设置是否使用缓存  默认是true
        urlConn.setUseCaches(true);
        // 设置为Post请求
        urlConn.setRequestMethod("GET");
        //urlConn设置请求头信息
        //设置请求中的媒体类型信息。
          urlConn.setRequestProperty("Content-Type", "application/json");
        urlConn.setRequestProperty("Content-Type", contentType);
          urlConn.setRequestProperty("charset", "utf-8");
        //设置客户端与服务连接类型
        urlConn.addRequestProperty("Connection", "Keep-Alive");
        // 开始连接
        urlConn.connect();
        // 判断请求是否成功
        if (urlConn.getResponseCode() == 200) {
            // 统计Xml输入流长度
            if( contentType.startsWith("t")) {
                Log.d(TAG, "Xml test , urlConn.getContentLength() = " + urlConn.getContentLength());
            }
            // 返回输入流以便于分情况处理
            return urlConn.getInputStream();
        } else {
            Log.d(TAG, "Get方式请求失败");
        }
        // 关闭连接
        urlConn.disconnect();
```

这里返回了输入流，当类型为 Xml 的时候我们统计了输入流的长度（Json 在这里统计无效），由于我们需要更灵活的处理，因此不能转为 String 再返回。同时要注意 http 请求都需要 try/catch 进行包裹。

```
} catch (Exception e) {
    Log.e(TAG, e.toString());
}
```

3. Json 数据请求与解析：

请求到输入流后，我们仍通过之前 post 一样的处理方法，将流转为 string，再转为结果对象。

如果为空，就取消请求：

```
if (resultStream == null) {
    // 取消业务
    e.onComplete();
}
```

否则解析数据，用 Date 对象获取时间来统计 Json 解析时间并打印，完成后发送给订阅者

```
String resultString = CommonUtil.streamToString(resultStream);
Log.d(TAG, "request = " + resultString);
ResultJson resultJson = new Gson().fromJson(resultString, ResultJson.class);
String foodString = resultJson.getDataString();
Type type = new TypeToken<ArrayList<Food>>() {
}.getType();

// 解析列表统计时间
Date startDate = new Date(System.currentTimeMillis());
List<Food> foodList = new Gson().fromJson(foodString, type);
Date endDate = new Date(System.currentTimeMillis());
long duration = endDate.getTime() - startDate.getTime();
Log.d(TAG, "Json test parse time = " + String.valueOf(duration) + "ms , size = " + String
        .valueOf(foodList.size()));
e.onNext(foodList);
```

其中 ResultJson 如下：

```
public class ResultJson extends ResultBase {

    // 对象jsonString
    private String dataString;
```

```
 * Description: 结果基类
 */
public class ResultBase {
    // 结果码
    private int RESULTCODE;

    // 携带信息
    private String msg;
```

4. Xml 数据请求与解析：

同样需要判断输入流非空，实际上这里代码可以提到外层。

那到输入流，先转为 Xml，再进行解析：

```
ResultXml resultXml = CommonUtil.getResultFromXml(CommonUtil.streamToXml(resultStream));
```

其中 ResulXml 如下：

```
 * Description: XML请求结果
 */
public class ResultXml extends ResultBase{

    // Food列表
    private List<Food> dataList;
```

先从输入流转为 Xml，通过 w3c.dom 包来进行转换：

```
/**
 * author:     Daniel
 * description: 从网络获取的输入流转化为Xml的Document
 */
public static Document streamToXml(InputStream is) throws ParserConfigurationException, IOException, SAXException {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    return builder.parse(is);
}
```

Builder.parse(is)会生成一个 Document 对象，即为一个 Xml 的 Document。

之后便是繁杂的解析过程：

```
/**
 * author:     Daniel
 * description: 解析Xml
 */
public static ResultXml getResultFromXml(Document document) {
    ResultXml resultXml = new ResultXml();

    Element result = document.getDocumentElement();

    // 获取该节点下面的所有子节点
    NodeList resultChildNodes = result.getChildNodes();
    Log.d(TAG, "resultChildNodes.getLength() = " + resultChildNodes.getLength());
    //把节点转换成元素节点
    Element resultCodeElement = (Element) resultChildNodes.item(0);
    Element messageElement = (Element) resultChildNodes.item(1);
    resultXml.setRESULTCODE(Integer.valueOf(resultCodeElement.getTextContent()));
    resultXml.setMsg(messageElement.getTextContent());
```

1. 获取到根结点为 result。
2. 通过 getChildNodes 方法获取到子结点的 list(这里走了一段弯路，开始我用的是 getAttributeNode 方法，然后发现怎么调试都是空，后来又用了很多其他办法，还是获取不到值)。
3. 通过位置分别获取简单子结点。并写入 resultXml 对象中。
4. 开始解析列表：

(1) 解析计时开始，获取列表总结点，通过 getElementsByTagName 获取到子结点列表，也就是包含每个菜品结点的列表，新建一个空菜品列表用于保存。

```
// 计时开始
Date startDate = new Date(System.currentTimeMillis());
// 开始解析列表
Element dataStringElement = (Element) resultChildNodes.item(2);
NodeList foodElemList = dataStringElement.getElementsByTagName(Const.ELEMENT_ID.FOOD);
List<Food> foodList = new ArrayList<>();
```

(2) 遍历列表，取到每一个 food 结点：

```
// 遍历food结点
for (int i = 0; i < foodElemList.getLength(); i++) {
    Element foodElement = (Element) foodElemList.item(i);
```

(3) 再通过之前的方法，获取到 food 结点的所有子节点并遍历，写入 food 中，再将 food 写入 foodList 中：

```
NodeList foodElemList = dataStringElement.getElementsByTagName(Const.ELEMENT_ID.FOOD);
List<Food> foodList = new ArrayList<>();
// 遍历food结点
for (int i = 0; i < foodElemList.getLength(); i++) {
    Element foodElement = (Element) foodElemList.item(i);
    // 遍历food内容
    NodeList foodParams = foodElement.getChildNodes();
    Food food = new Food();
    for (int j = 0; j < foodParams.getLength(); j++) {
        Element param = (Element) foodParams.item(j);
        switch (param.getTagName()) {
            case Const.ELEMENT_ID.FOOD_NAME:
                food.setFoodName(param.getTextContent());
                break;
            case Const.ELEMENT_ID.PRICE:
                food.setPrice(Integer.valueOf(param.getTextContent()));
                break;
            case Const.ELEMENT_ID.STORE:
                food.setStore(Integer.valueOf(param.getTextContent()));
                break;
            case Const.ELEMENT_ID.ORDER:
                food.setOrder(Boolean.valueOf(param.getTextContent()));
                break;
            case Const.ELEMENT_ID.IMGID:
                food.setImgId(Integer.valueOf(param.getTextContent()));
                break;
        }
    }
    foodList.add(food);
}
```

(4) 计时完成，输出时间，将 foodList 写入到 resultXml 中，返回该对象。

```
Date endDate = new Date(System.currentTimeMillis());
long duration = endDate.getTime() - startDate.getTime();
Log.d(TAG, "Xml test parse time = " + String.valueOf(duration) + "ms , size = " + String
        .valueOf(foodList.size()));

resultXml.setDataList(foodList);
return resultXml;
```

(5) 发送事件，传递 foodList 给订阅者：

```
e.onNext(resultXml.getDataList());
```

到这里，请求和解析部分就完成了，后面是订阅者的处理（发送 Notification 和播放提示音）。

订阅者处理：

```
.observeOn(AndroidSchedulers.mainThread())
.subscribe((Consumer) (foodList) → {
        // 构造打开首页的PendingIntent
        Intent intent = new Intent(mContext, MainScreen.class);
        PendingIntent pendingIntent = PendingIntent.getActivity(mContext, 0, intent, PendingIntent
                .FLAG_CANCEL_CURRENT);
        // 发送状态栏通知
        sendServerNotification(foodList, pendingIntent);
        // 播放提示音
        playNotification();
});
```

订阅者在主线程街道数据，和 E5 中一样，构造一个 pendingIntent，这里注意打开的是 MainScreen，传递给要发送的 Notification。然后播放提示音。

发送通知的过程和 E5 类似，但这里多了一个需求是增加一个删除按钮，这里使用 PendingIntent.getBroadcast()定义了一个关闭广播，然后调用 builder.addAction()即可在通知中加入按钮，同时包括按钮的点击事件（发送关闭广播）。

```
/**
 * author:      Daniel
 * description: 发服务器通知
 */
private void sendServerNotification(List<Food> foodList, PendingIntent intent) {
    String content = "新品上架" + "菜品数量：{foodList.size()}";

    Intent serviceIntent = new Intent(Const.IntentAction.CLOSE_NOTIFICATION);
    serviceIntent.putExtra(Const.IntentKey.NOTIFICATION_ID, NOTIFICATION_ID);
    PendingIntent closeIntent = PendingIntent.getBroadcast(mContext, FLAG_CLEAN, serviceIntent, PendingIntent
            .FLAG_UPDATE_CURRENT);

    NotificationManager notifyManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
            // 设置图标和食物图片、标题、内容
            .setSmallIcon(R.mipmap.ic_launcher)
            .setContentTitle("阿黄点餐")
            .setContentText(content)
            .addAction(R.drawable.ic_close, getString(R.string.text_back), closeIntent)
            //设置intent、点击自动消除
            .setContentIntent(intent)
            .setAutoCancel(true);
    //通过builder.build()方法生成Notification对象，设置提示音，并发送通知,id=1
    Notification notification = builder.build();
    notification.sound = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
    notifyManager.notify(NOTIFICATION_ID, builder.build());
}
```

当 DeviceStartedListener 接收到该广播的处理，通过定义好的常量 ID，关闭通知：

```
} else if (Const.IntentAction.CLOSE_NOTIFICATION.equals(intent.getAction())) {
    NotificationManager notifyManager = (NotificationManager) context.getSystemService(Context
            .NOTIFICATION_SERVICE);
    notifyManager.cancel(intent.getIntExtra(Const.IntentKey.NOTIFICATION_ID, 0));
}
```

同时注意需要在 Manifest 注册该广播的监听：

```
<intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
    <action android:name="scos.intent.action.CLOSE_NOTIFICATION"/>
</intent-filter>
```

最后是提示音的播放：

```
/**
 * author:     Daniel
 * description: 调用MediaPlayer播放消息通知
 */
private void playNotification() {
    Uri ringtone = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
    MediaPlayer mediaPlayer = MediaPlayer.create(mContext, ringtone);
    mediaPlayer.start();
    mediaPlayer.setOnCompletionListener((mp) → { mp.release(); });
}
```

通过 RingtoneManager.getDefaultUri 我们可以借用类型获取到各种默认铃声的资源 Uri。这里获取的是默认通知的铃声 Uri，之后调用 MediaPlayer.create()生成一个播放器对象，然后 start()即可播放。最后，设置一个监听，播放完成后释放播放器对象，避免内存泄漏。这里注意，如果是用 new 的方式产生的播放器对象，需要先 prepare()才能正常使用。查看源码可以发现 create()方法包含了这个过程：

```
public static MediaPlayer create(Context context, Uri uri, SurfaceHolder holder,
        AudioAttributes audioAttributes, int audioSessionId) {

    try {
        MediaPlayer mp = new MediaPlayer();
        final AudioAttributes aa = audioAttributes != null ? audioAttributes :
            new AudioAttributes.Builder().build();
        mp.setAudioAttributes(aa);
        mp.setAudioSessionId(audioSessionId);
        mp.setDataSource(context, uri);
        if (holder != null) {
            mp.setDisplay(holder);
        }
        mp.prepare();
        return mp;
    } catch (IOException ex) {
        Log.d(TAG, "create failed:", ex);
        // fall through
    } catch (IllegalArgumentException ex) {
        Log.d(TAG, "create failed:", ex);
        // fall through
    } catch (SecurityException ex) {
        Log.d(TAG, "create failed:", ex);
        // fall through
    }

    return null;
}
```

至此，代码部分已经完结，两种方式的比较放在结果中进行。

8. 调试成功后，编译打包。

修改版本为 4.0

```
android {
    compileSdkVersion 25
    buildToolsVersion "26.0.2"
    defaultConfig {
        applicationId "es.source.code"
        minSdkVersion 15
        targetSdkVersion 25
        versionCode 4
        versionName "4.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        resValue "string", "preferences_provider_authority", "${applicationId}.preferencesprovider"
    }
```

4.结论

本次练习收获有：

1. 学会了 Servlet 的搭建方法，熟练了如何构造 response，如何接收 request，了解 post 和 get 的区别。并使用 tomcat 进行部署和调试。

2. 学会了 Xml 和 Json 的数据构造方法，并可以通过 Servlet 接收和返回。

3. 熟悉了 HttpUrlConnection 的使用，熟练了在客户端进行 get 和 post 请求并接收数据的操作。

4. 学会了 Xml 和 Json 的数据解析方法，并可以在客户端接收到数据之后进行相应的解析。

5. 掌握了 notification 的特殊用法：添加按钮，并使用广播关闭该通知。

6. 初步掌握了 MediaPlayer 的用法，播放一个简单的资源。

问题：

1. 说明 Xml 的不同解析方式，比较各种方式的优缺点：

答：在 Android 中 Xml 共有三种解析方式：

（1）DOM 解析：

使用 DOM 的接口来遍历树级结构，在本应用中使用的就是这个方法。但对于大文件来说，由于在内存中存放的是树形结构，解析和加载会很耗资源。而且本身的代码写起来就比较费神，不利于维护。

（2）SAX 解析：

SAX 是基于事件的解析器，这里做完了 DOM 解析才看到这个，有点后悔没有用上，不过自己写代码解析一遍也有助于自己理解 Xml 的解析过程。看了一下 SAX 解析过程，非常简单，有点像解析 JSON 时用 Gson.fromJson()，不过需要多在泛型中定义好需要的实体类型，在 handler 中重写开头和结尾的取值方法，就可以将 Xml 的输入流直接转换为对象。而且它解析速度快，占用内存少。缺点是稍微重了一点，很多东西定义好了不太灵活。

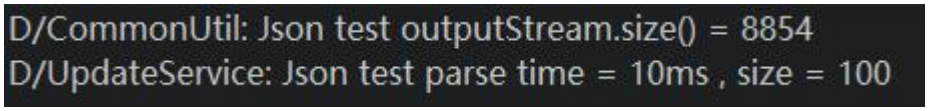
（3）PULL 解析：

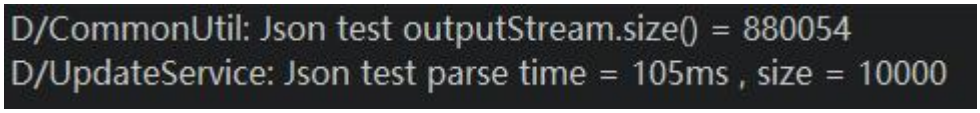
PULL 解析器也是基于事件的模式，但它封装的更简单，总体感觉更轻巧。事件由我们自己触发执行，不用再通过 handler 来接收。解析速度同样很快。看到这里又感觉 PULL 可能是最适合这个项目中使用的了，毕竟客户端处理的数据一般不会特别大，处理工具自然越轻越好。

综上所述，三种方式各有优劣，但用起来都没有什么问题，推荐用 PULL 解析，android 内部也是用它来解析的。

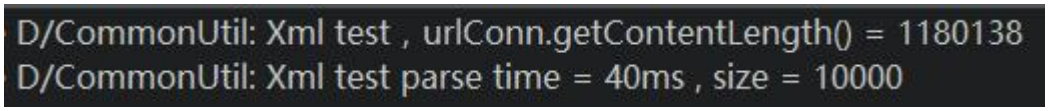
2. 对比 Json 和 Xml 作为数据封装方式的优缺点：

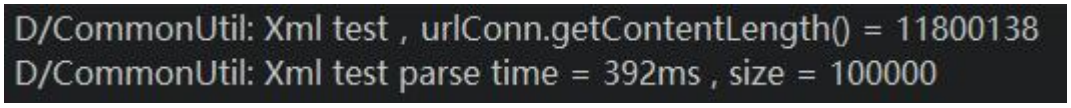
答：如下是 Xml 与 Json 区别的截图和比较：
Json:





Xml:



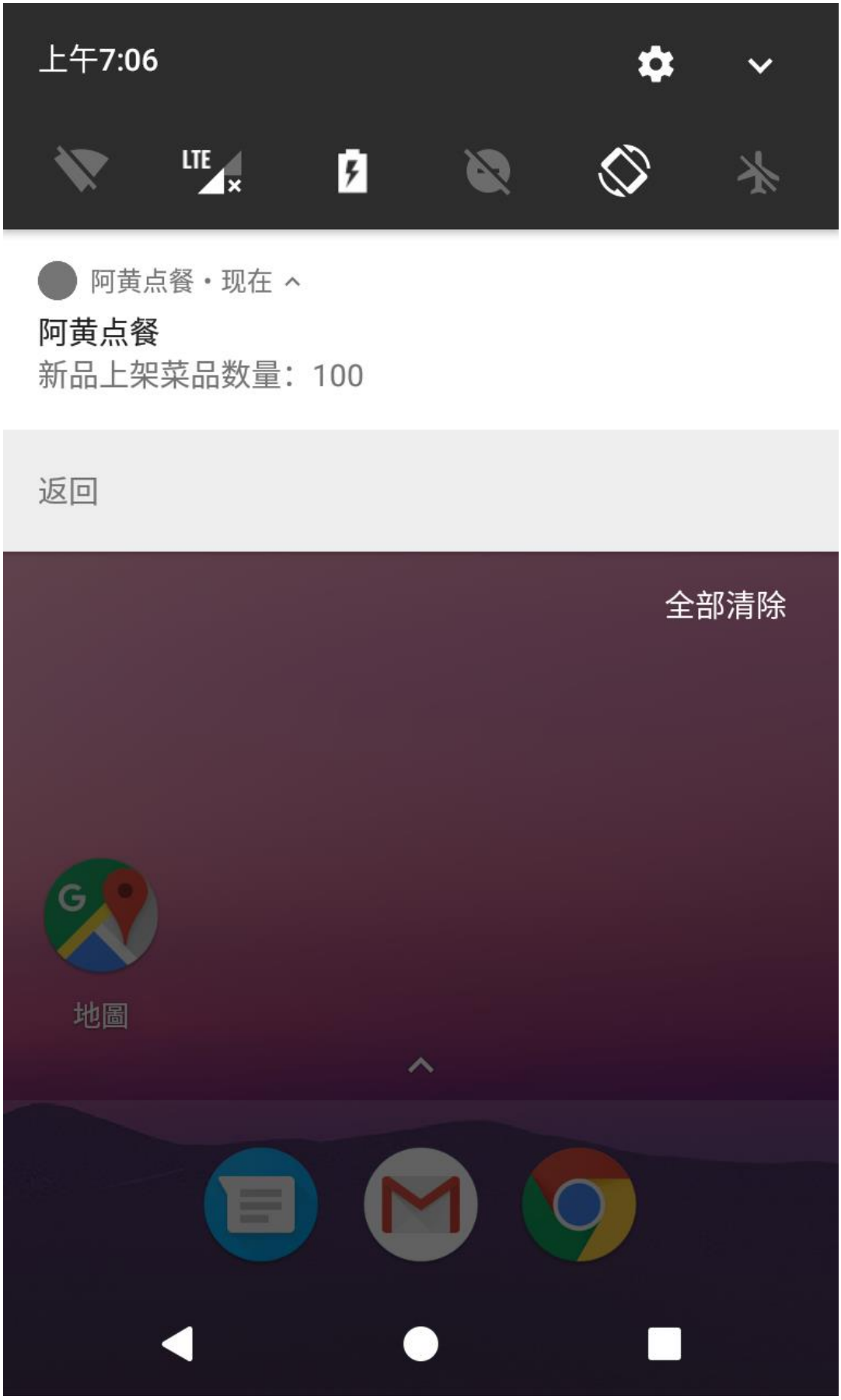


观察可以发现，Xml 的长度比 Json 要长 30%左右，这个显而易见，毕竟 Xml 的内容要多出很多结束结点名称等冗余信息，而 Json 只需要大括号和键值对即可表达出来对象关系。

在解析上，Xml 要稍快一些，这可能与我用的 Gson 框架有关，总体来说这个时间差可以忽略不计，但 Xml 的优点可能就在结构性更强吧，这也可能是解析更快的原因。

在封装和解析的代码上，程序员的压力是完全不同的，Json 仅用几行代码就可以解析和封装，但在 Xml 中则非常麻烦。当然，Xml 的应用目前更广泛，看起来 Json 更轻一点，两种都支持多语言。可以说两种方式各有各的好处，没有最好的语言，只有最适合的语言。

实验截图（登录状态和之前效果一样，省略截图，这里只展示通知）：



5.参考文献

1. Java Servlet 完全教程
2. Android 探索之 HttpURLConnection 网络请求
3. Stackoverflow：Can't create handler inside thread that has not called Looper.prepare()
4. Stackoverflow：Android Studio: Unable to start the daemon process
5. Android 中解析 XML
6. 几种 Java 的 JSON 解析库速度对比
7. XML 和 JSON 两种数据交换格式的比较