

# 0. Overview

## 0.0 团队成员介绍

Role	Member
项目总设计	赵传博
数据库设计	赵传博、曾凡浩、程浩然
数据库搭建	曾凡浩
CS架构设计	赵传博
服务端开发	曾凡浩
客户端开发	赵传博
报告撰写	程浩然
报告审查	赵传博

## 0.1 项目介绍-Spread Shop

By Spread Zhao:

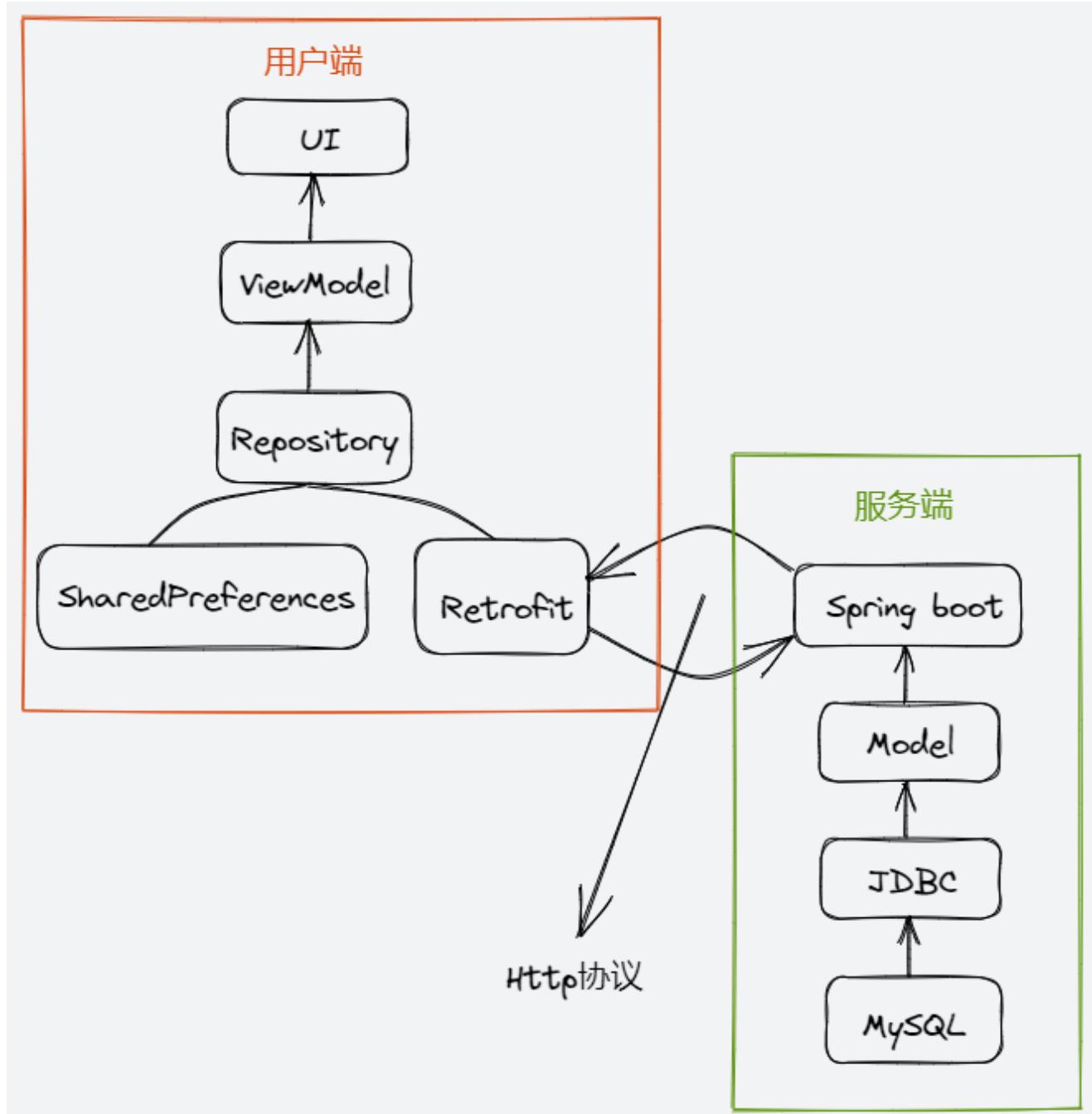
本项目是一个CS架构。服务端由Springboot + MySQL编写，给用户提供访问数据库的功能。正如项目的名字——Spread Shop，**它是一个模拟的电商平台**，数据库就是我们模拟的商店结构；而服务端我选择了比较擅长的安卓开发，使用MVVM架构来实现。这个项目中，我们模拟了**注册、登陆、记住密码**、浏览商品、浏览种类、搜索商品、查看商品详情、购买商品、查看订单功能。后序还可以不断进行扩充。

客户端的源代码我已经上传到了我的gitee仓库中：

[SpreadShop](#)

而服务端开发我本人后序也打算写一个使用socket的，更原始的http服务器。这是为了锻炼我的网络编程能力。这部分正在施工中，开发进度可以到这个笔记中追寻：

## 0.2 整体架构



## 0.3 数据库 Schema

### 账户表

列名	#	数据类型	非空	自增	键	默认	额外的	表达式	注释
account_name	1	varchar(15)	[v]	[ ]	PRI				
account_pwd	2	varchar(15)	[v]	[ ]					
ballance	3	decimal(10,2)	[ ]	[ ]		20000.00			

## 种类表

列名	#	数据类型	非空	自增	键	默认	额外的	表达式	注释
category	1	varchar(15)	[v]	[ ]	PRI				

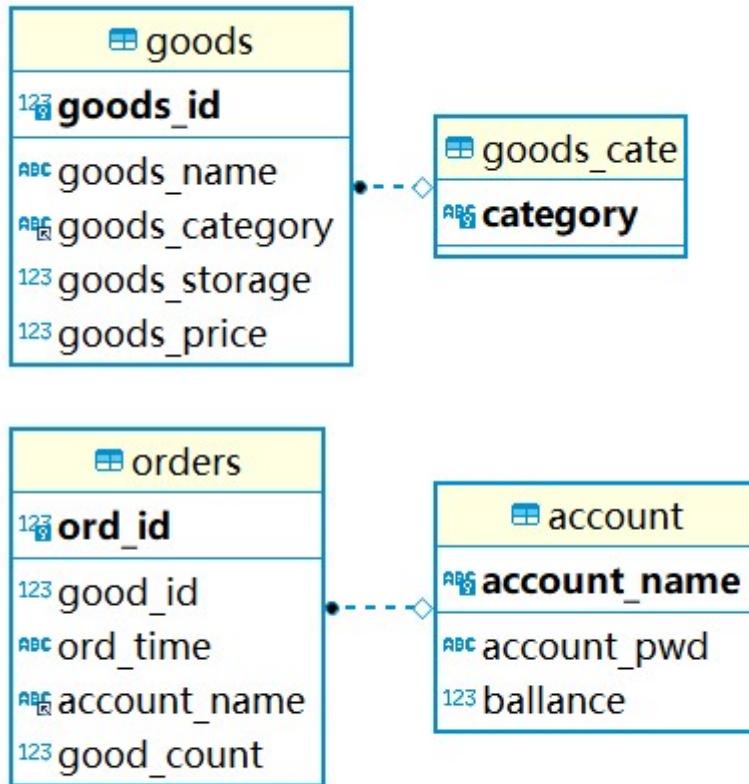
## 商品表

列名	#	数据类型	非空	自增	键	默认	额外的	表达式	注释
goods_id	1	int	[v]	[ ]	PRI				
goods_name	2	varchar(15)	[ ]	[ ]					
goods_categ...	3	varchar(15)	[ ]	[ ]	MUL				
goods_stora...	4	int	[ ]	[ ]					
goods_price	5	int	[ ]	[ ]					

## 订单表

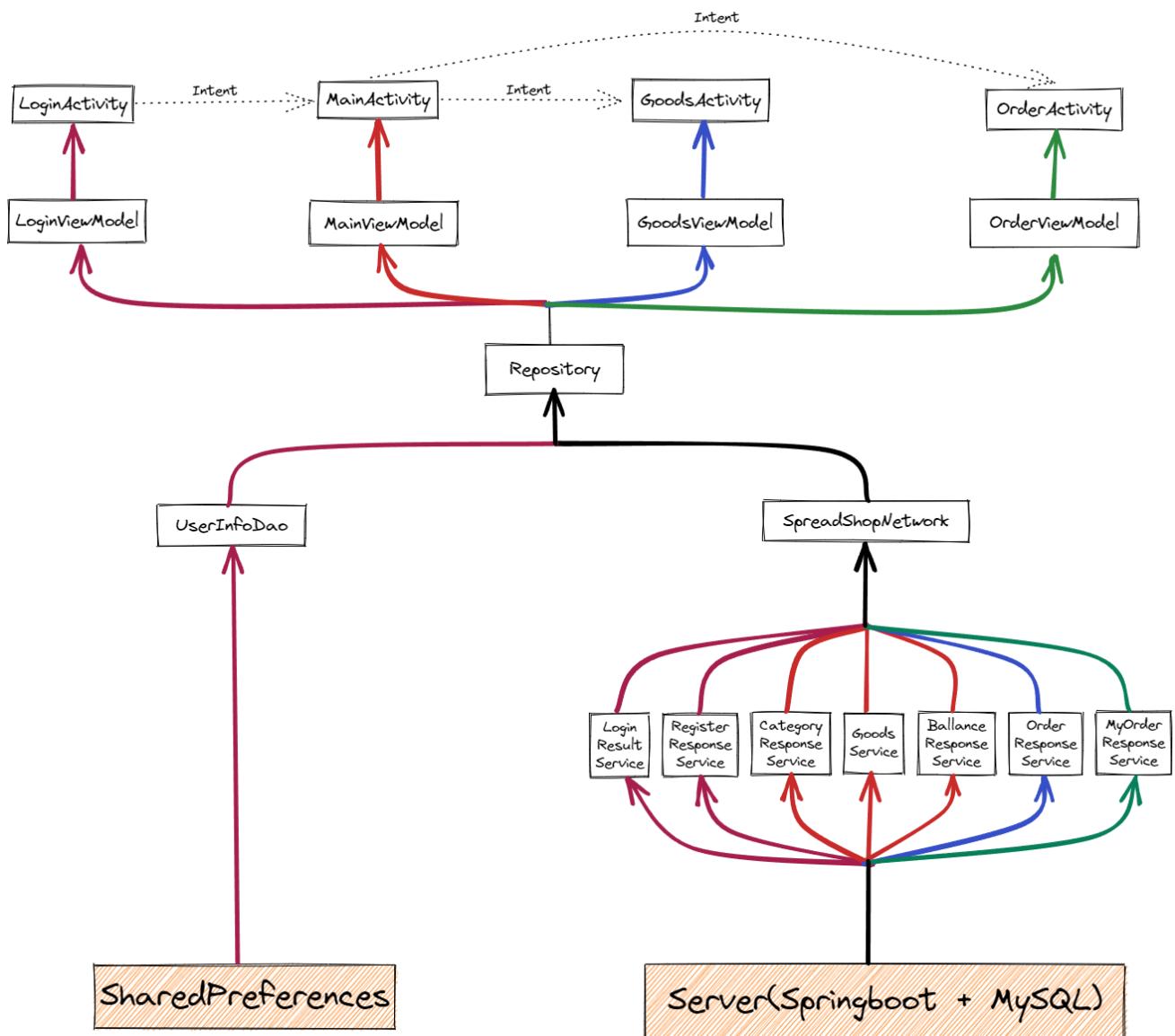
列名	#	数据类型	非空	自增	键	默认	额外的	表达式	注释
ord_id	1	int	[v]	[ ]	PRI				
good_id	2	int	[v]	[ ]					
ord_time	3	varchar(20)	[v]	[ ]					
account_name	4	varchar(15)	[ ]	[ ]	MUL				
good_count	5	int	[ ]	[ ]					

## 0.4 ER图

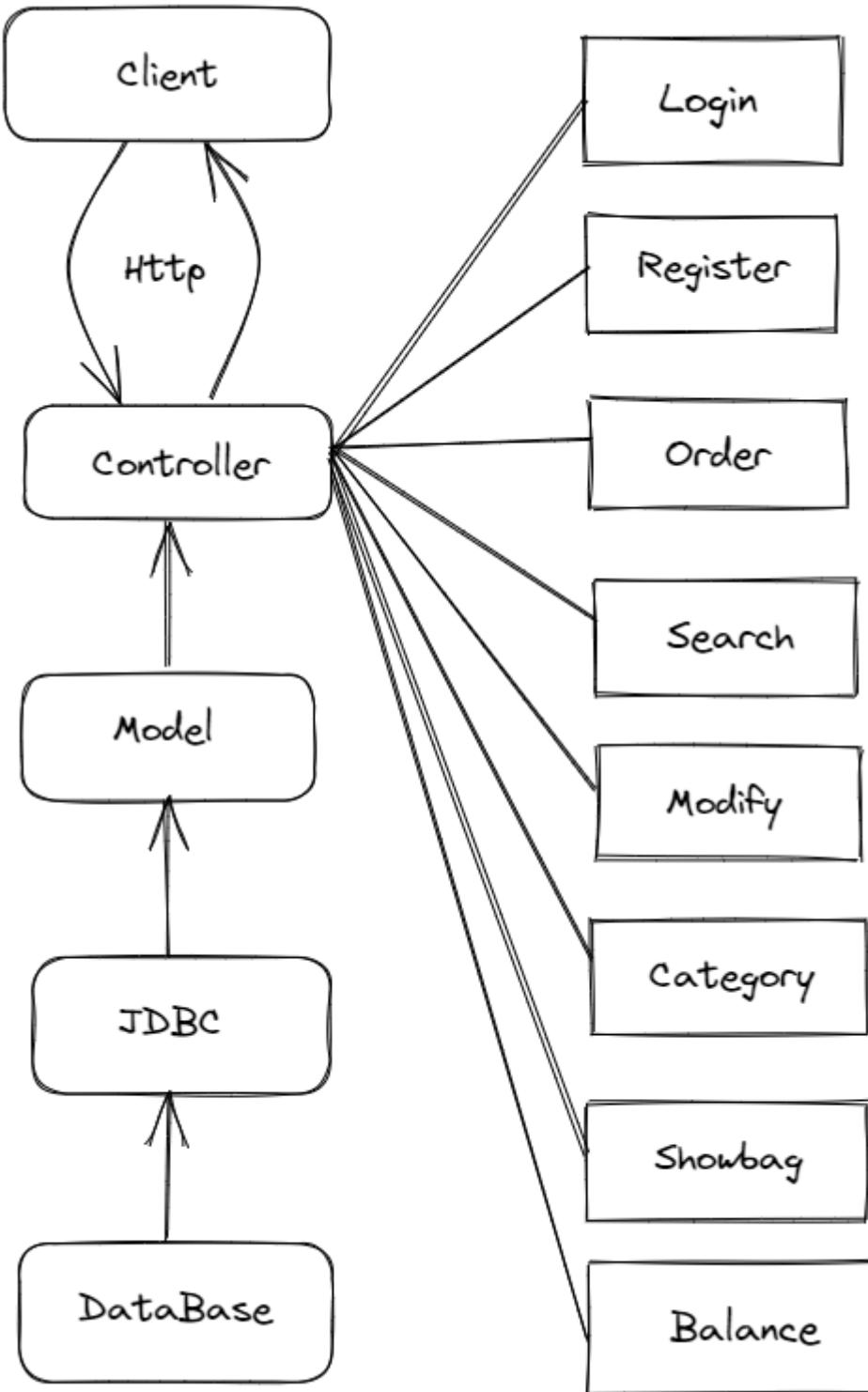


## 0.5 模块图

客户端



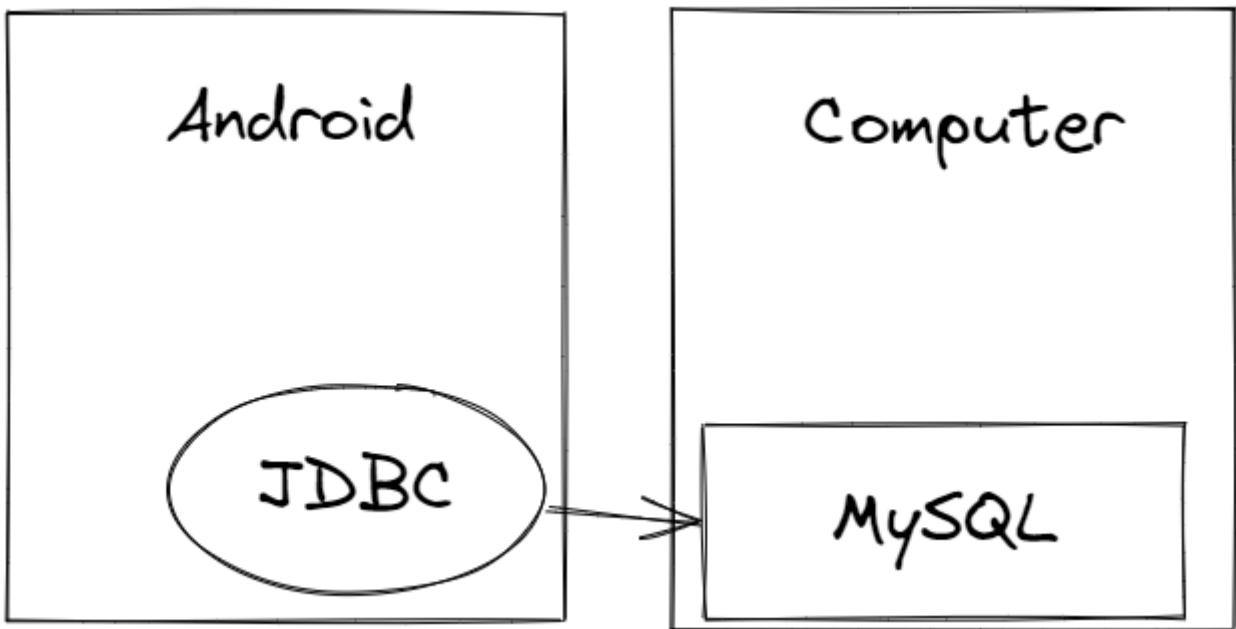
服务端



注：各模块的说明在下面已经展示的非常详细了，所以这里只给出模块图。

## 1. 用户端与服务端的连接

最一开始，我们的想法和上面的架构图还不是太一样：



可以看到，我们一开始打算直接将jdbc嵌入到安卓设备中，并让它访问电脑上的MySQL。这种做法能极大简化原来的架构，但是最终还是失败了。1.1介绍的就是我们失败的经历；而1.2开始就是我们最终完成整个项目的架构。

## 1.1 尝试用官方连接件连接

### 1.1.1 源码

```
package com.example.viewmodel
import java.sql.*
import java.util.Properties

/**
 * Program to list databases in MySQL using Kotlin
 */
object MySQLDatabaseExampleKotlin {

    internal var conn: Connection? = null
    internal var username = "root" // provide the username
    internal var password = "spreadzhao" // provide the
corresponding password

    @JvmStatic fun main(args: Array<String>) {
        // make a connection to MySQL Server
        getConnection()
        // execute the query via connection object
    }
}
```

```
executeMySQLQuery()
}

fun executeMySQLQuery() {
    var stmt: Statement? = null
    var resultSet: ResultSet? = null

    try {
        stmt = conn!!.createStatement()
        resultSet = stmt!!.executeQuery("SHOW DATABASES;")

        if (stmt.execute("SHOW DATABASES;")) {
            resultSet = stmt.resultSet
        }

        while (resultSet!!.next()) {
            println(resultSet.getString("Database"))
        }
    } catch (ex: SQLException) {
        // handle any errors
        ex.printStackTrace()
    } finally {
        // release resources
        if (resultSet != null) {
            try {
                resultSet.close()
            } catch (sqlEx: SQLException) {
            }

            resultSet = null
        }

        if (stmt != null) {
            try {
                stmt.close()
            } catch (sqlEx: SQLException) {
            }

            stmt = null
        }
    }

    if (conn != null) {
        .
    }
}
```

```

        try {
            conn!!.close()
        } catch (sqlEx: SQLException) {
        }

        conn = null
    }
}

/**
 * This method makes a connection to MySQL Server
 * In this example, MySQL Server is running in the local host
(so 127.0.0.1)
 * at the standard port 3306
*/
fun getConnection() {
    val connectionProps = Properties()
    connectionProps["user"] = username
    connectionProps["password"] = password
    try {
        Class.forName("com.mysql.cj.jdbc.Driver").newInstance()
        conn = DriverManager.getConnection(
            "jdbc:mysql://127.0.0.1:3306/?serverTimezone=UTC&useSSL=false",
            connectionProps)
    } catch (ex: SQLException) {
        // handle any errors
        ex.printStackTrace()
    } catch (ex: Exception) {
        // handle any errors
        ex.printStackTrace()
    }
}
}

```

参考文章：

[How to Connect to MySQL Database from Kotlin using JDBC?  
\(tutorialkart.com\)](https://www.tutorialkart.com/kotlin/kotlin-connect-mysql-database-jdbc/)

## 1.1.2 建立连接

```
fun getConnection() {  
    val connectionProps = Properties()  
    connectionProps["user"] = username  
    connectionProps["password"] = password  
    try {  
        Class.forName("com.mysql.cj.jdbc.Driver").newInstance()  
        conn = DriverManager.getConnection(  
            "jdbc:mysql://127.0.0.1:  
            3306/?serverTimezone=UTC&useSSL=false",  
            connectionProps)  
    } catch (ex: SQLException) {  
        // handle any errors  
        ex.printStackTrace()  
    } catch (ex: Exception) {  
        // handle any errors  
        ex.printStackTrace()  
    }  
}
```

`Class.forName("com.mysql.cj.jdbc.Driver").newInstance()`是Java里的反射，通过`"com.mysql.cj.jdbc.Driver"`这个文件名加载类`forName`的实例，这个实例就是JDBC的驱动，我们就是要使用它来通过JDBC连接到MySQL

有了JDBC的驱动之后，就可以通过`DriverManager.getConnection()`来得到一个连接，连接的参数如下：

url - a database url of the form `jdbc : subprotocol : subname`  
info - a list of arbitrary string tag/value pairs as connection arguments  
; normally at least a "user" and "password" property should be included

第一个参数里`mysql`为我们**要连接的数据库**，`127.0.0.1`为我们的**本地主机**，`3306`为**端口**

第二个参数connectionProps最重要的是至少要有一个**user**和**password**，这里就是我们MySQL的用户名和密码，该属性的设置如下：

```
val connectionProps = Properties()
connectionProps["user"] = username
connectionProps["password"] = password
```

把它的user字段制成用户名，password字段制成密码，而这两个变量是我们一开始定义的

```
internal var username = "root" // provide the username
internal var password = "spreadzhao" // provide the corresponding
password
```

### 1.1.3 执行查询

连接建立完成后，我们就可以调用**executeMySQLQuery()**来执行数据库的SQL语句，这里我们以**SHOW DATABASES**为例：

```
D:\programfiles\androidstudio\jre\bin\java.exe ...
information_schema
mysql
performance_schema
sys
xidian
```

### 1.1.4 遇到的问题

#### 时区不匹配

起初我们在执行查询时，出现了**The Server time zone value 'XXXXX'(乱码) is unrecognized or represents more than one time zone**的报错

这是由于MySQL使用的是美国的时区，所以我们需要连接到中国的时区，在数据库路径的最后加上**?serverTimezone=UTC**即可

#### SSL异常

在执行完**SHOW DATABASES**后，出现了SSL异常的报错

要解决这个问题，我们首先要了解SSL与SSH的区别，供学习的文章如下：

简而言之，SSL和SSH都属于应用层，但SSH可以让用户以某个主机用户的身份登录主机，并对主机执行操作（即执行一些命令），目前用的最多的就是远程登录和SFTP（还有简易版的SCP），而SSL和主机用户名登录没有任何关系，它本身并不实现主机登录的功能，它只是一个单纯的加密功能

解决办法是在配置文件中的url前添加一句话useSSL=false

至此，我们已经可以通过JDBC来连接主机中的MySQL。但现在仍存在一个问题，由于我们现在是在电脑终端上测试的，所以肯定不会出错。但之后我们进到手机上测试时，手机如何能拿到电脑的IP？这个涉及到内网相关的知识，也是我们后续必须解决的一个棘手问题

## 1.2 尝试用Http协议连接

当我们在手机上用Connector/J去连接MySQL时，抛出了Communication Link failure的错误：

```
.jdbc.exceptions.CommunicationsException: Communications link failure

    packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.
    com.mysql.cj.jdbc.exceptions.SQLException.createCommunicationsException(SQLException.java:174)
    com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:64)
    com.mysql.cj.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.java:835)
    com.mysql.cj.jdbc.ConnectionImpl.<init>(ConnectionImpl.java:455)
    com.mysql.cj.jdbc.ConnectionImpl.getInstance(ConnectionImpl.java:240)
    com.mysql.cj.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:207)
    java.sql.DriverManager.getConnection(DriverManager.java:580)
```

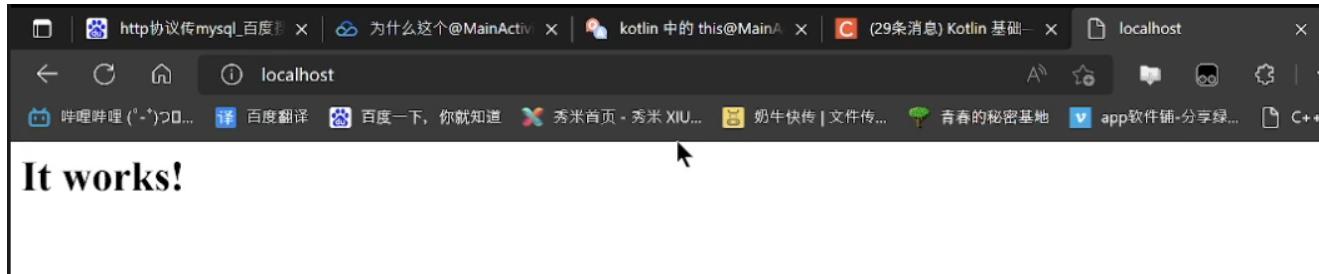
我们觉察到是网络权限的问题，在一开始忽略在AndroidManifest.xml文件里声明网络权限，在声明网络权限后，我们满怀期待地继续测试，结果依然失败了！在不断地尝试下，我们还是走不通这条道路，最后发现官方的连接件不支持在Android上使用！

[java - How to make MySQL Connector/J working on android? - Stack Overflow](#)

不得已，我们只能另辟蹊径，选择去探索借助Http协议来实现连接

### 1.2.1 Apache

首先我们需要配置好**Apache服务器**，打开Apache24服务。Apache24服务默认在本机下监听80端口，所以我们只要在浏览器输入localhost，就会打开.../Apache24/htdocs下的index.html文件



而我们所有服务器上的数据，都要存放在同目录下的get\_data.json下：

```
[  
  {"id":"5","version":"5.5","name":"Clash of Clans"},  
  {"id":"6","version":"7.0","name":"Boom Beach"},  
  {"id":"7","version":"3.5","name":"Clash of Royale"}  
]
```

## 1.2.2 测试手机电脑能否Ping通

服务器上有了数据，那么我们如何把这些数据转移到手机上呢？因为是跨平台转移数据，所以我们不能用本机来测试，这里选择去用局域网来测试，即用电脑连接手机的热点。

我们在电脑终端中输入ipconfig查到IP地址为192.168.183.39，接下来我们试试能不能通过这个IP来访问服务器数据，在浏览器输入192.168.183.39/get\_data.json，成功！

192.168.183.39/get\_data.json这一串地址是由两部分组成的，'/'前的部分叫做**base url**，也就是**基地址**，而后面的部分就是我们具体要访问的文件



### 1.2.3 测试数据能否转移到手机上

既然已经Ping通了，那么我们现在着手于编写一个测试程序将数据转移到手机上

#### 前端

*activity\_main* :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    <Button
        android:id="@+id/get_app_data_btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Get App Data"
        />

    <ListView
        android:id="@+id/app_list_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

该程序提供一个按钮，点击按钮以后，它就会把从服务器拿到的数据展示在下方的列表中

#### 后端

*AppService* :

```
package com.example.testhttp

import retrofit2.Call
```

```
import retrofit2.http.GET

interface AppService {
    @GET("get_data.json")
    fun getAppData(): Call<List<App>>
}
```

这里我们用到**Retrofit**，它需要调用自己的Retrofit接口协议

比如对于我们的APP对象，我们要写一个APP服务的接口，`@GET()`中的内容就是我们要具体访问的文件。接下来我们要定义一个函数`getAppData()`来得到数据，返回值是Retrofit自定义的一个Call类型的范型，这个范型里包裹的就是我们具体要取回来的数据，即App的List

为什么是一个List呢？我们看回到`get_data.json`中的[内容](#)，它的最外层是“[ ]”，表示它是一个**数组**。如果它的最外层是“{ }”，则表示它是一个对象。具体应该是什么是由开发用户端人和开发服务端的人协商规定的，所以这并不是一个太大的问题。

*App* :

```
package com.example.testhttp

data class App(val id: String, val name: String, val version: String)
```

这部分其实就是一个**数据模型**，就是我们从网络端拿回来的数据的其中一项，例如`{"id": "5", "version": "5.5", "name": "Clash of Clans"}`就是一个APP，与我们定义的数据模型的格式都是对应的

*MainActivity* :

```
package com.example.testhttp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import android.widget.ArrayAdapter
import android.widget.Button
import android.widget.ListView
```

```
import retrofit2.*  
import retrofit2.converter.gson.GsonConverterFactory  
  
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val btn = findViewById<Button>(R.id.get_app_data_btn)  
        val listView = findViewById<ListView>(R.id.app_list_view)  
  
        btn.setOnClickListener {  
            val retrofit = Retrofit.Builder()  
                .baseUrl("http://192.168.183.39")  
                .addConverterFactory(GsonConverterFactory.create())  
                .build()  
  
            val appService = retrofit.create(AppService::class.java)  
            appService.get pData().enqueue(object :  
                Callback<List<App>>{  
                    override fun onResponse(call: Call<List<App>>,  
response: Response<List<App>>) {  
                        val list = response.body()  
                        val listAppName = mutableListOf<String>()  
                        if(list != null){  
                            for(app in list){  
                                // Log.d("MainActivity", "id is  
                                // ${app.id}")  
                                // Log.d("MainActivity", "name is  
                                // ${app.name}")  
                                // Log.d("MainActivity", "version is  
                                // ${app.version}")  
                                listAppName.add(app.name)  
                            }  
                            val adapter =  
                                ArrayAdapter(this@MainActivity, android.R.layout.simple_list_item_1,  
                                listAppName)  
                                //val adapter = ArrayAdapter<String>(this,  
                                // android.R.layout.simple_list_item_1)  
                                listView.adapter = adapter  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        override fun onFailure(call: Call<List<App>>, t: Throwable) {
            t.printStackTrace()
        }
    }
}
}
```

`btn.setOnClickListener`是我们在手机上点击按钮后触发的**接口**。

首先它会定义一个Retrofit的Builder来建立Http的链接：添加的第一个属性就是我们先前提到的**基地址**——192.168.183.39，接下来添加的是一个转换的工厂，这里的**GsonConverterFactory**是一个公司研发的专门用于解析gson的工具，可以让我们得代码变得更简单

`appService`则是用反射创建的一个对象，即由我们在`AppService.kt`中写的接口文件生成的一个服务类，通过这个服务类我们就可以调用`getAppData()`函数来获得数据

`.enqueue()`是规定的写法，其中涉及到了一个**匿名函数的实现**，也就是`object`，它是一个`Callback`类型的变量。在`Callback`的接口中声明了`onResponse()`和`onFailure()`这两个函数，所以我们紧接着就要实现这两个函数。

```
override fun onResponse(call: Call<List<App>>, response: Response<List<App>>) {
    val list = response.body()
    val listAppName = mutableListOf<String>()
    if(list != null){
        for(app in list){
            // Log.d("MainActivity", "id is ${app.id}")
            // Log.d("MainActivity", "name is ${app.name}")
            // Log.d("MainActivity", "version is ${app.version}")
            listAppName.add(app.name)
        }
        val adapter = ArrayAdapter(this@MainActivity,
            android.R.layout.simple_list_item_1, listAppName)
        //val adapter = ArrayAdapter<String>(this,
```

```
        android.R.layout.simple_list_item_1)
        listView.adapter = adapter
    }
}

override fun onFailure(call: Call<List<App>>, t: Throwable) {
    t.printStackTrace()
}
```

如果我们成功从服务端拿回了数据，就执行`onResponse()`中的方法，它有`call`和`response`两个参数。`call`的内容就是`getAppData()`的返回值，所以我们一旦成功，就能拿到这个参数，也就代表我们拿到了从服务器取回来的数据。而第二个参数`response`则是和Retrofit协议一起做的，我们真正的数据就是存在`response`中的，它里面的范型和`call`里面的范型都是同一的。

接着我们就要从`response`的`body`字段中拿到我们需要的数据，拿到之后我们就可以对这个字段“为所欲为”了。我们这里再定义一个`String`类型的`List`，去把每一个`App`中的名字拿出来单独形成一个`List`

之后我们就要把`List`中的名字逐个展示在列表上了，这里用到`ArrayAdapter()`，它的作用就是帮助我们把数据显示在视图上。其中有三个参数，第一个参数是上下文，这里的上下文必须是`Activity`，因为我们是在`MainActivity`中做的，所以我们需要传的参数就是`MainActivity`；第二个参数是布局类型，是固定的写法；第三个参数就是我们需要展示的`List`，也就是真正的数据来源。最后我们把`listView.adapter`制成我们创建的`adapter`时，`listView`展示的就是这些名字

## 测试

在手机上点击`GET APP DATA`按钮，得到如下列表，成功！

下午4:12 | 0.0K/s

蓝牙 VPN 4G

TestHttp

GET APP DATA

Clash of Clans

Boom Beach

Clash of Royale

10月19日更新

在服务端搭建完成后，我们第一时间测试了完整的连接，成功在手机上显示了服务端传来的所有名字。

虽然对象是拿到了，但是返回的都是空。后来才突然想起来，**data class**里对应的成员名要和**json**中完全一致才可以！

假如这是要传的**json**：

```
{  
    "status": "ok", "query": "北京",  
    "places":  
    [  
        {"name": "北京市", "location":  
        {"lat": 39.9041999, "lng": 116.4073963},  
        "formatted_address": "中国?京市"},  
  
        {"name": "北京西站", "location":  
        {"lat": 39.89491, "lng": 116.322056},  
        "formatted_address": "中国 ?京市 丰台区 莲花池东路118  
号"},  
  
        {"name": "北京南站", "location":  
        {"lat": 39.865195, "lng": 116.378545},  
        "formatted_address": "中国 ?京市 丰台区 永外大街车站路  
12号"},  
  
        {"name": "北京站(地铁站)", "location":  
        {"lat": 39.904983, "lng": 116.427287},  
        "formatted_address": "中国 ?京市 东城区 2号线"}  
    ]  
}
```

那么下面就是对应的**data class**：

```
data class PlaceResponse(val status: String, val places:  
List<Place>)  
  
data class Place(val name: String, val location: Location,  
@SerializedName("formatted_address") val address: String)
```

```
data class Location(val lng: String, val lat: String)
```

由于JSON中一些字段的命名可能与Kotlin的命名规范不太一致，因此这里使用了`@SerializedName`注解的方式，来让JSON字段和Kotlin字段之间建立映射关系。

## 2. 服务端

### 2.1 对数据库的增删改查

作为**服务端**，需要正确处理客户端的请求，把客户端需要的数据库中的数据发送给客户端，因此首先就要求服务端本身能对数据库实现增删改查的操作

#### 2.1.1 源码

Dao :

```
import java.util.List;

public interface Dao {
    //查询
    List<Goods> find(String sql);

    public void update(String sql);
}
```

在其中我们仅定义了一个Dao接口，在接口中我们声明了`find`方法和`update`成员变量

`JDBC_connectorImpl` :

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class JDBC_connectorImpl extends BaseDao implements Dao {

    public List<Goods> find(String sql){
```

```

//System.out.println("start find");
//String sql="select * from goods";
/***
 * 开始的时候绑定的是params
 */
ResultSet rs=query(sql,null);
List<Goods> list = new ArrayList<>();
try {
    while (rs.next()) {
        Goods goods=new Goods();
        goods.setGoods_id(rs.getInt(1));
        goods.setGoods_name(rs.getString(2));
        goods.setGoods_category(rs.getString(3));
        goods.setGoods_storage(rs.getInt(4));
        goods.setGoods_price(rs.getInt(5));
        list.add(goods);
    }
    // boolean wrong = !rs.next();
    // System.out.println("!rs.next(): " + wrong);
} catch (SQLException e) {
    throw new RuntimeException(e);
}finally {
    closeAll();
}

return list;
}

public void update(String sql){
    doPs(sql,null);
}
}

```

JDBC\_connectorImpl是一个继承于BaseDao的类，是对Dao接口的具体实现

*BaseDao:*

```

import java.sql.*;
import java.util.List;
import java.util.Objects;
import java.util.ResourceBundle;

```

```
public class BaseDao {  
    private static String driver;  
    private static String url;  
    private static String username;  
    private static String passwd;  
    private Connection conn;  
    private PreparedStatement ps;  
    private ResultSet rs;  
  
    static {  
        ResourceBundle rb=ResourceBundle.getBundle("database");  
        driver=rb.getString("driver");  
        url=rb.getString("url");  
        username=rb.getString("user");  
        passwd=rb.getString("passwd");  
        try {  
            Class.forName(driver);  
        } catch (ClassNotFoundException e) {  
            throw new RuntimeException(e);  
        }  
    }  
  
    /**  
     * 获取连接  
     * @return  
     */  
    public Connection get_connection(){  
        try {  
            conn=DriverManager.getConnection(url,username,passwd);  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
        return conn;  
    }  
  
    /**  
     * 关闭连接  
     */  
    public void closeAll(){  
        if(rs!=null){  
            try {  
                rs.close();  
            } catch (SQLException e) {  
                e.printStackTrace();  
            }  
        }  
        ps=null;  
        conn=null;  
    }  
}
```

```
        rs.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
if(ps!=null){
    try {
        ps.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
if(conn!=null){
    try {
        conn.close();
        //System.out.println("关闭成功");
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
}

public boolean doPs(String sql, List<Objects> params){
    this.get_connection();
    boolean flag=true;
    try {
        ps=conn.prepareStatement(sql);
        if(params!=null){
            for(int i=0;i<params.size();i++){
                ps.setObject(i+1,params.get(i));
            }
        }
        flag=ps.execute();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return flag;
}

public ResultSet query(String sql,List<Objects> params){

    this.doPs(sql,params);
}
```

```
    try {
        rs=ps.getResultSet();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return rs;
}
```

在这里我们定义了BaseDao类用于实现对数据库的连接以及增删改查的操作

## 2.1.2 实现对数据库的连接

首先我们在类中定义了一些私有的静态变量，让它们只能在当前类被访问。而它们的值单独存放在`database.properties`配置文件中，这样做好处是便于我们后期做代码维护时快速检索并修改这些变量的值

```
driver=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/store
user=root
passwd=zfhzfh123
```

1. `driver`是我们连接数据库所需要的驱动，它由数据库厂商提供
2. `url`就是我们所要连接的数据库的**地址**，也就是数据文件的目录地址
3. `user`和`passwd`就是我们登录数据库的用户名和密码

那么我们如何让计算机知道这些变量的值存放在配置文件中并获取这些值呢？据我们所知有两种方法：第一种方法是用输入/输出流实现对接，但这种方法代码量会稍长一些；第二种方法也就是我们采用的方法，我们用到了**ResourceBundle类**，通过它的`getString`方法来获取配置文件中的内容并把对应值赋给这些变量

接下来是一个存在一些疑问的部分，众所周知，使用`Class.forName()`静态方法的目的是为了动态加载类，在加载完成后，一般还要调用`Class.newInstance()`静态方法来实例化对象以便操作。为什么我们调用`Class.forName()`加载数据库驱动包但却没有调用`newInstance()`方法呢？

这是因为`Class.forName()`的作用是要求JVM查找并加载指定的类，首先要知道，静态代码是和`Class`绑定的，`Class`装载成功就表示执行了你的静态代码，而且以后都不会再走这段静态代码了。所以，如果在类中有静态初始化

器的话，JVM必然会执行该类的静态代码段，而在JDBC规范中明确要求驱动必须在静态初始化器中注册，所以我们在使用JDBC时只需要调用 `Class.forName()` 就可以了

另外，不得不提的是，我们通过JDBC去连接MySQL和进行增删改查操作，都必须使用 **try-catch** 语句，如果我们不进行异常捕获或处理，当查询结果为空时，是很容易导致线上事故的

在 `get_connection()` 中，我们通过 `DriverManager.getConnection()` 方法来建立与数据库的连接，它有三个参数——就是我们 [前面](#) 提到的静态变量中的 `url`、`user` 和 `passwd`

能够获取连接，我们也要能够关闭连接。在 `closeAll()` 中，我们只需要通过简单的条件判断：如果判断连接已建立，即 `rs`、`ps`、`conn` 非空，就调用 `close()` 方法关闭它们

### 2.1.3 实现增删改查

要实现数据库的增删改查操作，按理说我们需要封装四个方法分别实现，但这里其实我们只用封装两个方法即可。让我们将目光瞄准增删改查操作的本质——增删改要对数据库进行修改，而查只要 **返回数据库中的数据** 并不对数据库进行修改。

于是我们选择将增删改这三个操作封装到 `doPs()` 方法中。获取连接后，我们先生成一个空的数据库对象，然后将其与我们从数据库中拿到的数据对象通过 `setObject()` 方法一一绑定。

在 JDBC 中常用的增删改查方法有 `execute()`、`executeQuery()`、`executeUpdate()`，下面我们来具体看看这三个方法

1. `execute()` 可执行任何sql语句，方法的返回值是 `Boolean` 类型，如果执行查询操作并有返回结果，则此方法返回 `true`；如果执行增删改操作，没有返回结果，则此方法返回 `false`
2. `executeQuery()` 只能用来执行查询语句，执行后返回结果集 `ResultSet` 对象
3. `executeUpdate()` 能用来执行增删改语句，它的返回值是 `int` 类型，表示受影响的行数

这里选择第一种方法主要是因为我们在封装查询功能的时候发现，也需要进行数据库的连接和动态绑定。而这些操作我们在前面已经完成过了，所以在查询操作中继续使用这种方法就可以直接调用了

在实现查询操作的`query()`方法中，我们直接调用`doPs()`方法。但由于得到的返回值是`false`或者`true`，它们是对我们的查询操作没有任何帮助的，所以我们需要再通过`getResultSet()`方法获取查询结果集，才能达成目的

至此，我们通过Java代码实现了对数据库的增删改查操作，但只能在本地终端中进行测试。为了将数据库的数据发送给客户端，接下来的任务就是将输出内容上传到网站，让客户端能通过Http协议获取数据。

## 2.2 数据上传

关于数据上传，起初我们计划用Socket纯手写一个服务器，利用一个无限`while`循环去监听目的端口。因为这个方法的代码逻辑过于底层，尝试之后遇到了许多问题，而国内用Socket写服务器的相关文章甚少，学习成本过高。最终我们弃用了这个方法，决定用Spring Boot来实现。

### 2.2.1 搭建Spring Boot项目

IDEA的社区版中没有一键生成Spring Boot项目功能，在完整版中，这个功能也是通过下面这个网站生成后直接导入IDEA的：

[Spring Initializr](https://start.spring.io/)

The screenshot shows the configuration page for a Spring Boot project. The 'Project' section is set to 'Maven Project'. The 'Language' section is set to 'Java'. The 'Spring Boot' section is set to '2.7.5'. The 'Project Metadata' section includes fields for Group (com.zfh.jdbc1111), Artifact (TestSpringBoot1111), Name (TestSpringBoot1111), Description (Demo project for Spring Boot), and Package name (com.zfh.jdbc1111.TestSpringBoot1111). The 'Dependencies' section includes 'Spring Web' (selected), 'Lombok', 'Spring Boot DevTools', and 'Spring for GraphQL'. The 'Java' section shows Java 8 selected. The bottom of the page shows a progress bar with 8 steps completed.

Project	Language	Dependencies
<input type="radio"/> Gradle Project	<input checked="" type="radio"/> Java	<b>Spring Web</b> <small>WEB</small>
<input checked="" type="radio"/> Maven Project	<input type="radio"/> Kotlin	Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
<b>Spring Boot</b>	<input type="radio"/> 3.0.0 (SNAPSHOT)	<b>Lombok</b> <small>DEVELOPER TOOLS</small>
	<input type="radio"/> 3.0.0 (RC1)	Java annotation library which helps to reduce boilerplate code.
	<input type="radio"/> 2.7.6 (SNAPSHOT)	<b>Spring Boot DevTools</b> <small>DEVELOPER TOOLS</small>
	<input type="radio"/> 2.6.14 (SNAPSHOT)	Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
	<input type="radio"/> 2.6.13	<b>Spring for GraphQL</b> <small>WEB</small>
<b>Project Metadata</b>		Build GraphQL applications with Spring for GraphQL and GraphQL Java.
Group	com.zfh.jdbc1111	
Artifact	TestSpringBoot1111	
Name	TestSpringBoot1111	
Description	Demo project for Spring Boot	
Package name	com.zfh.jdbc1111.TestSpringBoot1111	
Packaging	<input checked="" type="radio"/> Jar	
	<input type="radio"/> War	
Java	<input type="radio"/> 19	
	<input type="radio"/> 17	
	<input type="radio"/> 11	
	<input checked="" type="radio"/> 8	

这里我们在网站上设置好参数和依赖后生成Spring Boot项目，手动导入IDEA中。可气的是测试文件中的import语句全部标红，这说明依赖包没有成功导入。解决办法是进入到Setting-Build,Execution,Deployment-Build Tools-Maven-Runner中勾选*Delegate IDE build/run actions to Maven*选项，应用之后便不再报错

## 2.2.2 源码

```
import
org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
//http://localhost:8080/demo?name=hahaha
@RestController
@EnableAutoConfiguration
public class Controller {

    @GetMapping={"/demo"}
    public List<Goods>    test() {
        Dao dao = new JDBC_connectorImpl();
        //System.out.println("mysql connected !");
        List<Goods> list;
        //if (query.equals("showAll")) {
        //    System.out.println("Equals! opening sql");
        //    String sql = "select * from goods";
        //    list = dao.find(sql);

        //}

        return list;
        //return "123";
    }
}
```

## 2.2.3 遇到的问题

## 404 Not Found

我们运行测试程序，并在浏览器输入`localhost:8080/demo`，出现了如下报错：

## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Oct 19 21:20:09 CST 2022

There was an unexpected error (type=Not Found, status=404).

这个错误说明该网址是能连接上我们的程序的，但它没有得到任何拥有值。一番检查后发现，终端中存在报错，报错信息显示8080端口被占用。为了检查8080端口的占用情况，我们在Windows PowerShell中检查占用该端口进程的PID，再到任务管理器中检索这个PID，发现是`java.exe`占用了8080端口。我们手动杀掉这个“罪魁祸首”，再去测试，发现这个问题依然存在

参考连接：

[# Java项目启动报错： Process exited with an error: 1 \(Exit value: 1\)](#)

起初我们认为是代码出了问题，就把测试程序的返回值改为了最简单的字符串“123”进行测试，却依旧出错。于是我们把问题聚焦到手动搭建的Spring Boot项目上，我们去询问拥有完整版IDEA的同学，看看它的Spring Boot一键生成功能有哪些选项，发现只需要`Spring Web`和`Lombok`两个依赖包即可。

我们重新搭建项目，测试了最简单的字符串“123”，成功在网页上显示出来了！

## 500 Not Found

“123”成功后，我们就觉得Spring Boot项目已经构建成功了，能够满足我们的需求。于是我们开始测试自己的代码，再次运行测试程序，并在浏览器输入`localhost:8080/demo`，然而，却出现了如下报错：

# Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Oct 19 21:17:56 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

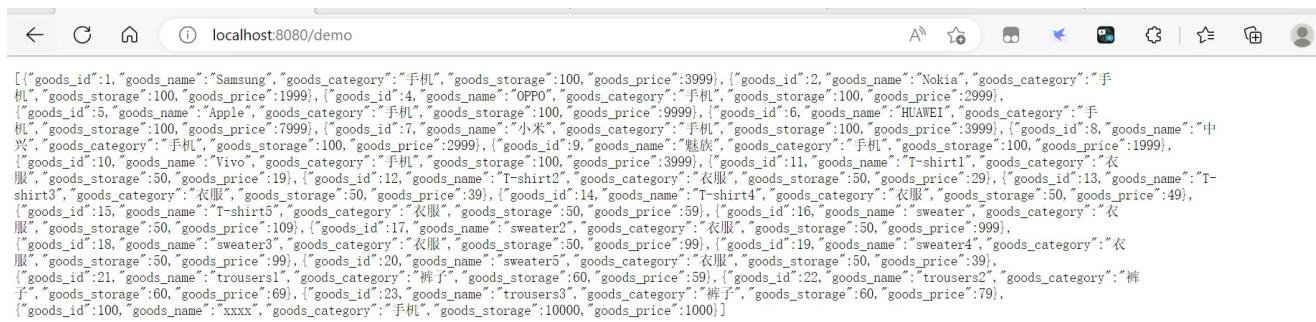
查询资料后得知，Spring Boot项目中**自带配置文件**，会自动配置来连接到数据库，而不是调用我们**自己写的配置文件**。我们按照它的格式重新编写，再次测试，出乎意料地又失败了！

原来，Spring Boot项目不仅自带配置文件，甚至自带了**封装好的JDBC**。而由于我们是自己写的JDBC，肯定和它自带的操作格式有所差别，无法进行连接。所以我们需要添加这样一段代码：

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
```

它的作用是告诉计算机不再使用Spring Boot项目自带的配置文件和封装好的JDBC，而使用我们自己写的，之前的工作不能白干！

再再次运行测试程序，并在浏览器输入localhost:8080/demo，终于成功了！



## 2.3. 功能实现

### 2.3.1 用户登录

数据模型LoginResult：

```
public class LoginResult {  
    private boolean isSuccess;  
    private String username;  
    private String passwd;  
    private String message;  
    public String getMessage() {  
        return message;  
    }  
    public void setMessage(String message) {  
        this.message = message;  
    }  
    public boolean isSuccess() {  
        return isSuccess;  
    }  
    public void setSuccess(boolean success) {  
        isSuccess = success;  
    }  
    public String getUsername() {  
        return username;  
    }  
    public void setUsername(String username) {  
        this.username = username;  
    }  
    public String getPasswd() {  
        return passwd;  
    }  
    public void setPasswd(String passwd) {  
        this.passwd = passwd;  
    }  
}
```

*JDBC\_connectorImpl :*

```
public LoginResult find_account(String sql){  
    ResultSet rs=query(sql,null);  
    LoginResult LR=new LoginResult(); //默认所有参数都是null;  
    LR.setUsername("default");  
    try {  
        while (rs.next()) {  
            LR.setUsername(rs.getString(1));  
            LR.setPasswd(rs.getString(2));  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

```

        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }finally {
        closeAll();
    }
    return LR;
}

```

*GetController :*

```

//http://localhost:8080/login?username=spreadzhao&password=1234;
@RequestMapping({"/login"})
public LoginResult test2(@RequestParam Map<String, LoginResult>
params){
    LoginResult res;
    String sql="select * from account where account_name='" +
params.get("username")+"'";
    //select * from account where account_name='username';
    Dao dao = new JDBC_connectorImpl();
    res = dao.find_account(sql);
    res.setSuccess(false);
    if(!res.getUsername().equals(params.get("username"))){
        res.setMessage("Unknown username");
    }else if(!res.getPassword().equals(params.get("password"))){
        res.setMessage("Wrong password for user [" +
params.get("username") + "]");
    }else{
        res.setSuccess(true);
        res.setMessage("Login Success!!!");
    }
    return res;
}

```

由于实现登录功能时，get请求会传入username和password两个参数（参数化之间用&分隔），所以我们这里最好选择Map来承接参数，接着定义sql语句

```

String sql="select * from account where account_name='" +
params.get("username")+"'";

```

这里存在一个问题，起初我们并没有在语句中添加 '，测试时报错，显示sql语句中不知道account\_name所指的是哪一列。我们来看看完整的sql语句应该怎么写：

```
select * from account where account_name='username'
```

因为我们这里的username传进来的的是一个字符串，如果我们在定义的sql语句中不添加 '，他就无法识别这是一个字符串，sql语句就会抛出异常

得到从数据库中传回的username和password后，我们就可以进行条件判断了：

1. username不等——用户名不存在，返回
2. username相等，password不等——密码错误，返回
3. username相等，password相等——登录成功

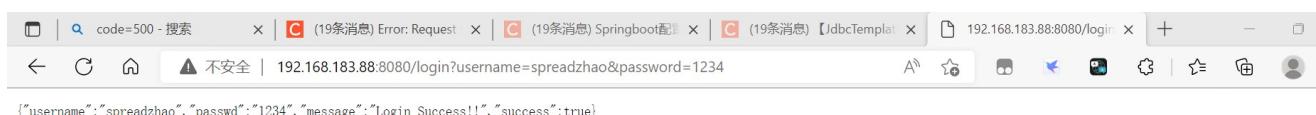
但在我们具体测试时，当我们输入错误的用户名时，终端报出了code=500的错误，这说明我们的服务端内部出现了问题。我们发现，如果输入的username和password在数据库中没有查询到，find\_account()方法返回的结果默认参数值都为null，那我们在比较时就会报错，跳过所有的判断语句，直接返回结果

所以需要在find\_account()方法中添加这样一行代码

```
LR.setUsername("default");
```

当输入的username和password在数据库中没有查询到，将它的默认参数值改为default，这样就能进行正常的条件判断流程了

最后我们看看结果如何，成功了！



## 2.3.2 用户注册

首先我们来看一下注册功能的数据模型：

```
// http://hocalhost:8080/register?username=zfh&password=666
public class Register {
    private String username;
    private String password;
    private String message;
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

这其中的三个变量非常好理解，*username*与*password*对应用户名与密码，*message*则用来储存展示出来的结果信息——注册成功/失败原因

接着我们写一个*RegistResponse*，当用户端发出注册请求时，将其作为信息发回客户端：

```
public class RegistResponse {
    private boolean isSuccess;
    private Register register;
    public boolean isSuccess() {
        return isSuccess;
    }
}
```

```

public void setSuccess(boolean success) {
    isSuccess = success;
}
public Register getRegister() {
    return register;
}
public void setRegister(Register register) {
    this.register = register;
}
}

```

RegistResponse中除了包含我们定义的数据模型Register外，还有注册请求的结果

GetController :

```

// http://hocalhost:8080/register?username=zfh&password=666
@RequestMapping({"/register"})
public RegistResponse register(@RequestParam Map<String, String> params) {
    Dao dao = new JDBC_connectorImpl();
    Register Rs=new Register();
    RegistResponse Rr=new RegistResponse();
    Rr.setSuccess(false);
    Rs.setUsername(params.get("username"));
    Rs.setPassword(params.get("password"));
    String sql = "select * from account where account_name=' " +
    params.get("username") + "'";
    LoginResult rs;
    String rst = null;
    rs = dao.find_account(sql);
    if (!(rs.getUsername().equals("default"))){
        Rs.setMessage("Unseccess!!username:" +
    params.get("username") + " is existed!!!");
    }
    else {
        if (params.get("username").length() > 15)
    Rs.setMessage("Unseccess!!username too long more than 15");
        else if (params.get("password").length() > 15)
    Rs.setMessage("Unseccess!!password too long more than 15");
        else {
    }
}

```

```
//insert into account values ('zfh','134',0);
    sql = "insert into account values ('" +
params.get("username") + "','" + params.get("password") +
"',20000)";
    dao.update(sql);
    Rs.setMessage("Regist success!!!");
    Rr.setSuccess(true);
}
Rr.setRegister(Rs);
return Rr;
}
```

我们需要到数据库中搜索并判断：

1. 如果这个用户名在数据库中已经存在了，我们当然不能允许注册一个相同的用户名。注册失败，返回结果信息用户名已存在
2. 如果这个用户名在数据库中不存在，需要我们再对username与password进行一次判断——因为MySQL对于用户名和密码有着不能大于15个字符的要求。若大于，则注册失败，返回结果信息用户名/密码过长；若不大于，则注册成功，返回结果信息

确认注册成功后，我们还要将username与password插入数据库中，同时偷偷塞给这个用户20000元余额：

```
sql = "insert into account values ('" + params.get("username") +
"', '" + params.get("password") + "',20000);
```

最后把RegistResponse发给用户端，任务就完成了！

### 2.3.3 购买商品

老样子，我们来看一下购买功能的数据模型：

```
// http://localhost:8080/order?
username=spreadzhao&goods_id=5&number=2;
public class Order {
    private String username;
    private int goods_id;
    private int num;
```

```

public String getMessage() {
    return message;
}
public void setMessage(String message) {
    this.message = message;
}
private String message;
public String getUsername() {
    return username;
}
public void setUsername(String username) {
    this.username = username;
}
public int getGoods_id() {
    return goods_id;
}
public void setGoods_id(int goods_id) {
    this.goods_id = goods_id;
}
public int getNum() {
    return num;
}
public void setNum(int num) {
    this.num = num;
}
}

```

这里`username`对应用户名, `goods_id`对应商品id号, `num`则是购买数量

接着我们写一个`OrderResponse`, 这里与注册功能中的`RegistResponse`十分类似, 不多赘述

```

public class OrderResponse {
    private boolean success;
    private Order order;
    public boolean isSuccess() {
        return success;
    }
    public void setSuccess(boolean success) {
        this.success = success;
    }
}

```

```

public Order getOrder() {
    return order;
}
public void setOrder(Order order) {
    this.order = order;
}
}

```

GetController :

```

//http://localhost:8080/order?
username=spreadzhao&goods_id=5&number=2;
@RequestMapping({"order"})
public OrderResponse order(@RequestParam Map<String, String> params){
    Dao dao=new JDBC_connectorImpl();
    Order order=new Order();
    order.setUsername(params.get("username"));
    order.setGoods_id(Integer.parseInt(params.get("goods_id")));
    order.setNum(Integer.parseInt(params.get("number")));
    OrderResponse OR=new OrderResponse();
    OR.setSuccess(false);
    int storage;
    int price;
    double ballance;
    String sql="select goods_storage from goods where
goods_id="+params.get("goods_id");
    storage=dao.find_Intdetail(sql);
    if(storage<Integer.parseInt(params.get("number"))){
        order.setMessage("The item is out of stock");
    }
    else{
        sql="select goods_price from goods where
goods_id="+params.get("goods_id");
        price=dao.find_Intdetail(sql);
        sql="select ballance from account where
account_name='"+params.get("username")+"'";
        ballance=dao.find_Doudetail(sql);
        if(price*Integer.parseInt(params.get("number"))>ballance){
            order.setMessage("Insufficient balance");
        }
        else{

```

```

        //update account set ballance =400 where account_name
        ='zfh';
        sql="update account set ballance="+(ballance-
        price*Integer.parseInt(params.get("number")))+" where
        account_name='"+params.get("username")+"'";
        dao.update(sql);
        //update goods set goods_storage=99 where goods_id=5;
        sql="update goods set goods_storage="+(storage-
        Integer.parseInt(params.get("number")))+" where
        goods_id="+Integer.parseInt(params.get("goods_id"));
        dao.update(sql);
        sql="select count(*) from orders";
        //sql 在使用count查询所有元组时*得用一个括号括起来
        int orderNum=dao.find_Intdetail(sql)+1;
        LocalDateTime now = LocalDateTime.now();
        sql="insert into orders
        values("+orderNum+","+params.get("goods_id")+","
        +now.toString().substring(0,19)+",'"+params.get("username")+"','"+
        Integer.parseInt(params.get("number"))+"')";
        dao.update(sql);
        order.setMessage("The order was successful");
        OR.setSuccess(true);
        //res(sql);
    }
}
OR.setOrder(order);
return OR;
}

```

这里我们需要到数据库中搜索并判断：

1. 商品库存是否足够
2. 用户余额是否足够

确认购买成功后，我们当然要为这笔订单“处理后事”，更新数据库中的商品库存与用户余额：

```

sql="update account set ballance="+(ballance-
        price*Integer.parseInt(params.get("number")))+" where
        account_name='"+params.get("username")+"'";
        dao.update(sql);

```

```
sql="update goods set goods_storage="+(storage-  
Integer.parseInt(params.get("number")))+" where  
goods_id="+Integer.parseInt(params.get("goods_id"));  
dao.update(sql);
```

这里要注意`Integer.parseInt()`这个方法。最开始我们在这一步时一直报错，原因是**定义的Map params中的values是String，而database中的values是int**所以会导致报错！而`Integer.parseInt()`在这里就能将String转化为int

我们在做这个项目的时候希望能展示每一笔订单，因此我们在建立数据库时创建了一个订单表，购买操作完成后我们也要将订单信息存入数据库

其中这段代码值得注意：

```
LocalDateTime now = LocalDateTime.now();  
sql="insert into orders  
values("+orderNum+"," +params.get("goods_id")+",'"  
+now.toString().substring(0,19)+"', '"+params.get("username")+"', "+In  
teger.parseInt(params.get("number"))+");
```

我们计划将订单完成的时间也作为订单的一个属性存入数据库，因此需要用到`LocalDateTime.now()`方法。但在测试的时候，却出现了问题。原来MySQL中的时间值给了十五个字符长度，而`LocalDateTime.now()`返回的时间精度是纳秒，如果直接把它返回的时间存入数据库就会报错，所以需要将时间精度用`subString()`截取一下

后事都处理完了，依旧是将OrderResponse发送给用户端

### 2.3.4 展示商品（分类/搜索/推荐）

我们这里直接给出搜索功能的数据模型：

```
//实体类  
public class Goods {  
    private int goods_id;//商品id号  
    private String goods_name;//商品名称  
    private String goods_category;//商品种类  
    private int goods_storage;//商品库存  
    private int goods_price;//商品价格
```

```
public int getGoods_id(){
    return goods_id;
}
public void setGoods_id(int id){
    this.goods_id=id;
}
public String getGoods_name(){
    return goods_name;
}
public void setGoods_name(String name){
    this.goods_name=name;
}
public String getGoods_category(){
    return goods_category;
}
public void setGoods_category(String category){
    this.goods_category=category;
}
public int getGoods_storage(){
    return goods_storage;
}
public void setGoods_storage(int storage){
    goods_storage=storage;
}
public int getGoods_price(){
    return goods_price;
}
public void setGoods_price(int price){
    goods_price=price;
}
}
```

```
import java.util.List;
public class GoodsResponse {
    private boolean success;
    private List<Goods> goods;
    public boolean isSuccess() {
        return success;
    }
    public void setSuccess(boolean success) {
        this.success = success;
    }
}
```

```

    }
    public List<Goods> getGoods() {
        return goods;
    }
    public void setGoods(List<Goods> goods) {
        this.goods = goods;
    }
}

```

因为和之前的数据模型异曲同工，这里直接在注释中给出，不做赘述

*GetController* :

```

//http://localhost:8080/searchgoods?cmd=showall(cmd=categoryphone)
(cmd=searchshirt)
@RequestMapping={"/searchgoods"})
public GoodsResponse test4(@RequestParam("cmd") String query) {
    Dao dao = new JDBC_connectorImpl();
    String sql;
    GoodsResponse response = new GoodsResponse();
    if(query.contains("category")){
        String queryNew=query.substring(8);
        sql="select * from goods where
goods_category='"+queryNew+"'";
        response.setGoods(dao.find_goods(sql));
        if(response.getGoods().isEmpty()){
            response.setSuccess(false);
        }else {
            response.setSuccess(true);
        }
    } else if (query.contains("search")) {
        String queryNew=query.substring(6);
        sql="select * from goods where goods_name
like'%" +queryNew+"%'";
        response.setGoods(dao.find_goods(sql));
        if(response.getGoods().isEmpty()){
            response.setSuccess(false);
        }else {
            response.setSuccess(true);
        }
    }else {

```

```
    if (query.equals("showall")) {
        sql = "select * from goods";
        response.setGoods(dao.find_goods(sql));
        if (response.getGoods().isEmpty()) {
            response.setSuccess(false);
        } else {
            response.setSuccess(true);
        }
    } else if(query.equals("getrecommand")){
        sql="select * from goods order by rand() limit 20";
        response.setGoods(dao.find_goods(sql));
        if (response.getGoods().isEmpty()) {
            response.setSuccess(false);
        } else {
            response.setSuccess(true);
        }
    }
    return response;
}
```

因为我们展示商品时即可以按种类展示，又可以按搜索结果展示，还可以展示推荐商品，甚至可以展示全部商品。具体体现在用户端传来请求中的`cmd`字段的值不同

首先我们要处理的是分类和搜索两种请求，例如`cmd=categoryphone`和`cmd=searchshirt`。所以我们就必须要在这里进行判断。这里我们采用的方法是用`contains()`方法判断`cmd`字段是否包含`category`和`search`，接着用`substring()`将搜索内容的关键字截取再进到数据库中搜索

我们关注一下处理搜索的sql语句：

```
select * from goods where goods_name like'%" +queryNew+"%'
```

这里我们利用了MySQL中用“%”匹配任意个字符的特性，为的是让我们的搜索功能更加人性化。例如用户搜索`phone`，我们商品列表中的`iphone13`也符合搜索结果

展示全部商品没有什么好说的，这里我们重点看看展示推荐商品的sql语句：

```
select * from goods order by rand() limit 20
```

我们的电商平台没有广告推荐一说，为了保证琳琅满目的商品都有被推荐的机会，我们选择用公平的随机来决定推荐商品。而如果想要在MySQL中实现随机查询，我们要在sql语句末尾添加`order by rand() limit x`，其中x就是随机输出的结果数量，这里我们设置为20

最后把GoodsResponse发给用户端

### 2.3.5 展示商品种类（分类）

依旧先是看看数据模型，这部分的数据模型非常简单，不再赘述

```
public class Category {  
    private String category;  
    public String getCategory() {  
        return category;  
    }  
    public void setCategory(String category) {  
        this.category = category;  
    }  
}
```

```
import java.util.List;  
public class CategoryResponse {  
    private Boolean success;  
    private List<Category> categories;  
    public Boolean getSuccess() {  
        return success;  
    }  
    public void setSuccess(Boolean success) {  
        this.success = success;  
    }  
    public List<Category> getCategories() {  
        return categories;  
    }  
    public void setCategories(List<Category> categories) {  
        this.categories = categories;  
    }  
}
```

GetController :

```
//http:localhost:8080/searchcategory?cmd=getallcategory
@RequestMapping({"/searchcategory"})
public CategoryResponse searchcategory(@RequestParam("cmd") String
query){
    Dao dao=new JDBC_connectorImpl();
    CategoryResponse CR=new CategoryResponse();
    String sql;
    List<Category> list=null;
    if (query.equals("getallcategory")){
        sql="select distinct goods_category from goods";
        list=dao.find_category(sql);
        CR.setSuccess(true);
    }else{
        CR.setSuccess(false);
    }
    CR.setCategories(list);
    return CR;
}
```

这部分做的操作和展示商品部分几乎一样，甚至不需要做复杂的判断。但考虑到它只需要从数据库中获取商品的种类信息，数据模型比较简单，就不和展示商品部分一起编写了

### 2.3.6 展示订单

先来看数据模型：

```
public class Bag {
    private int order_id;//订单id号
    private int goods_id;//商品id号
    private String order_time;//下单时间
    private String account_name;//下单用户
    private int goods_number;//商品数量
    private String goods_name;//商品名称
    public String getGoods_name() {
        return goods_name;
    }
    public void setGoods_name(String goods_name) {
        this.goods_name = goods_name;
    }
}
```

```
    }
    public int getOrder_id() {
        return order_id;
    }
    public void setOrder_id(int order_id) {
        this.order_id = order_id;
    }
    public int getGoods_id() {
        return goods_id;
    }
    public void setGoods_id(int goods_id) {
        this.goods_id = goods_id;
    }
    public String getOrder_time() {
        return order_time;
    }
    public void setOrder_time(String order_time) {
        this.order_time = order_time;
    }
    public String getAccount_name() {
        return account_name;
    }
    public void setAccount_name(String account_name) {
        this.account_name = account_name;
    }
    public int getGoods_number() {
        return goods_number;
    }
    public void setGoods_number(int goods_number) {
        this.goods_number = goods_number;
    }
}
```

```
import java.util.List;
public class BagResponse {
    private boolean success;
    private List<Bag> OrderList;
    public boolean isSuccess() {
        return success;
    }
    public void setSuccess(boolean success) {
```

```
        this.success = success;
    }
    public List<Bag> getOrderList() {
        return OrderList;
    }
    public void setOrderList(List<Bag> orderList) {
        OrderList = orderList;
    }
}
```

比较简单，注释中解释

GetController :

```
//http://localhost:8080/showbag?username=zfh;
@RequestMapping={"/showbag"})
public BagResponse showbag(@RequestParam("username") String
username){
    Dao dao=new JDBC_connectorImpl();
    BagResponse bagResponse=new BagResponse();
    String sql="select * from orders where
account_name='"+username+"'";
    List<Bag> bag=dao.find_bag(sql);
    if(bag.isEmpty()){
        bagResponse.setSuccess(false);
    }else {
        bagResponse.setSuccess(true);
    }
    bagResponse.setOrderList(bag);
    return bagResponse;
}
```

之前我们在购买商品的部分提到了订单表，相应的操作也在当时实现了。这里只是为了展示某个用户的所有订单，所以只需要从数据库中取就可以了

## 2.3.7 展示余额

这部分用来展示用户的当前余额，非常简单，仅展示代码

数据模型：

```
public class Balance {  
    private boolean success;  
    public boolean isSuccess() {  
        return success;  
    }  
    public void setSuccess(boolean success) {  
        this.success = success;  
    }  
    public double getBalance() {  
        return balance;  
    }  
    public void setBalance(double balance) {  
        this.balance = balance;  
    }  
    private double balance;  
}
```

```
public class BalanceResponse {  
    private boolean success;  
    private Balance balance;  
    public boolean isSuccess() {  
        return success;  
    }  
    public void setSuccess(boolean success) {  
        this.success = success;  
    }  
    public Balance getBalance() {  
        return balance;  
    }  
    public void setBalance(Balance balance) {  
        this.balance = balance;  
    }  
}
```

GetController :

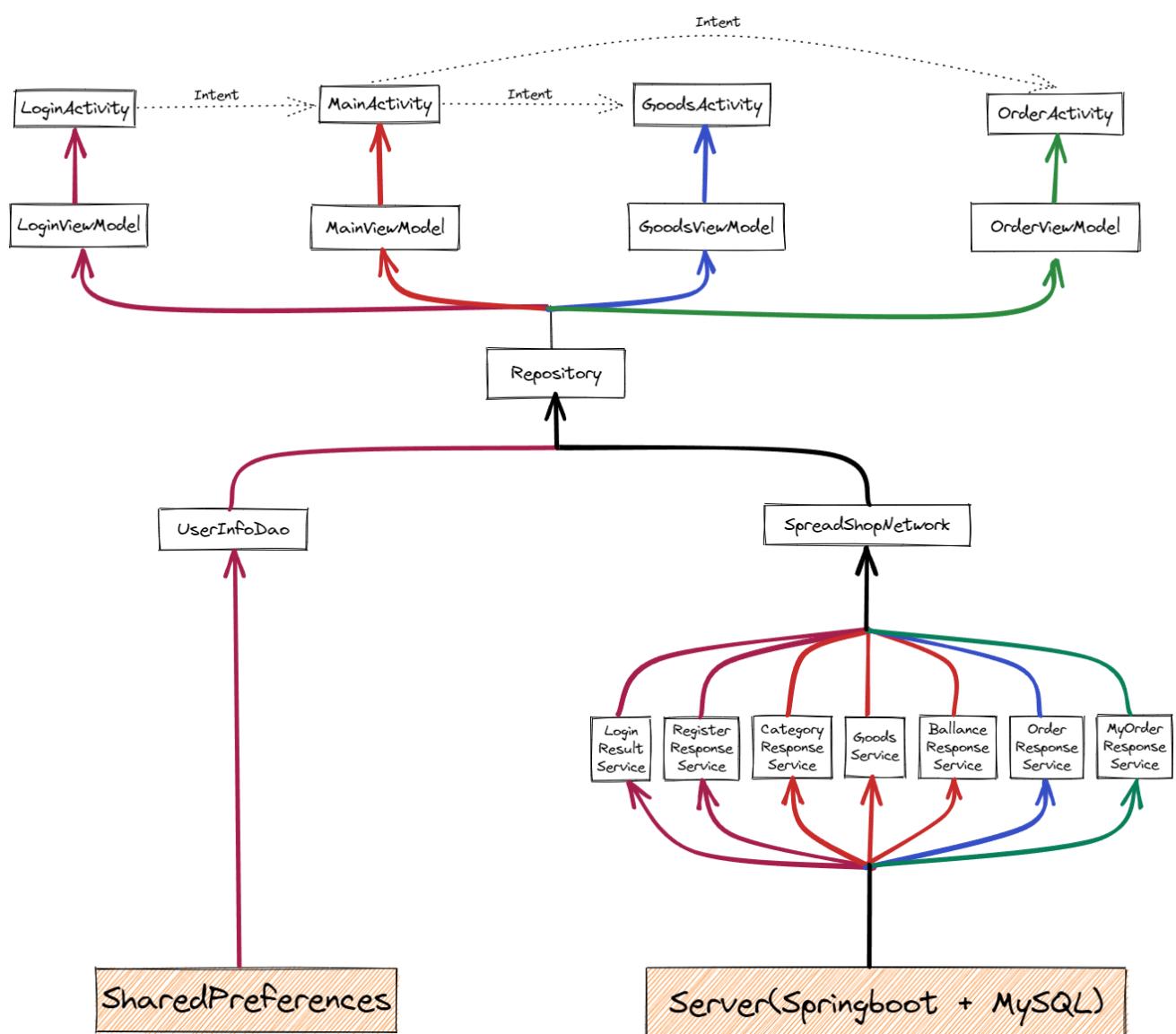
```
//http:loclahost:8080/balance?username=zfh  
@RequestMapping={"/balance"})  
public Balance balance(@RequestParam("username") String username){  
    Dao dao =new JDBC_connectorImpl();  
    String sql="select ballance from account where
```

```

account_name=' "+username+" ';
Balance balance=new Balance();
balance.setBalance(dao.find_balance(sql));
if(balance.getBalance()!=-1){
    balance.setSuccess(true);
}
else{
    balance.setSuccess(false);
}
return balance;
}

```

### 3. 用户端

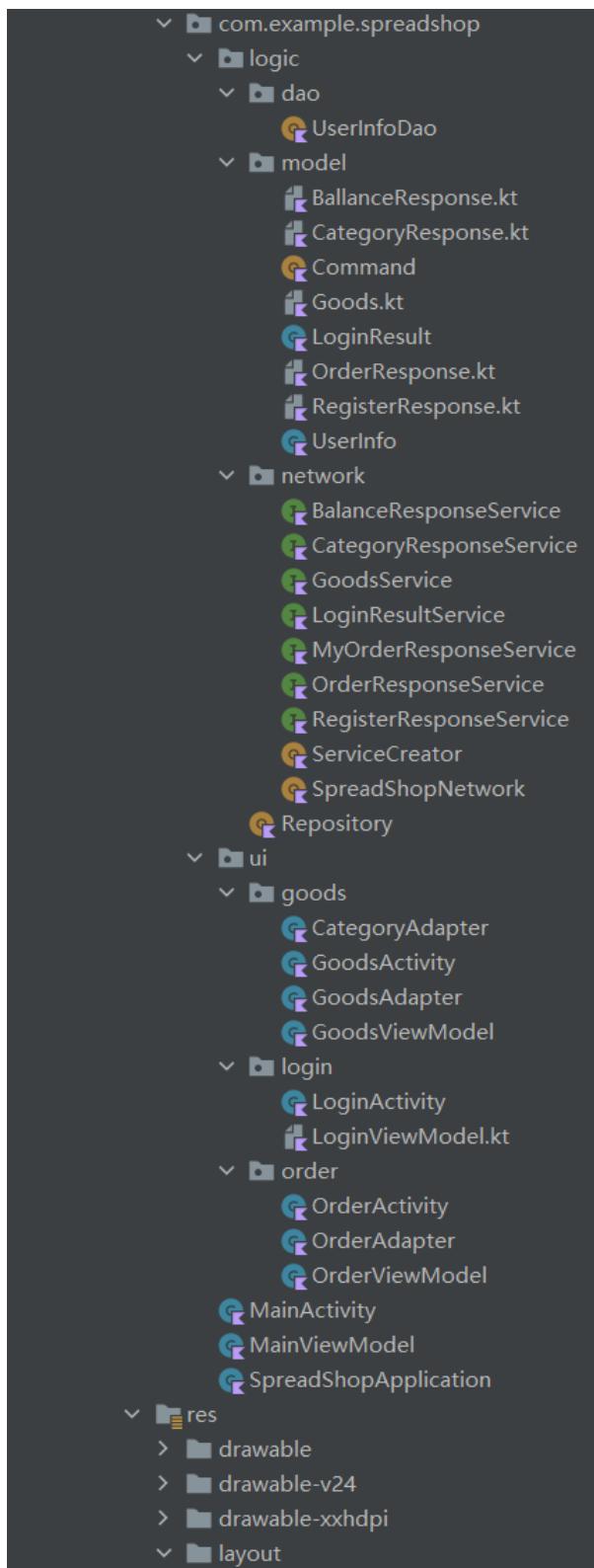


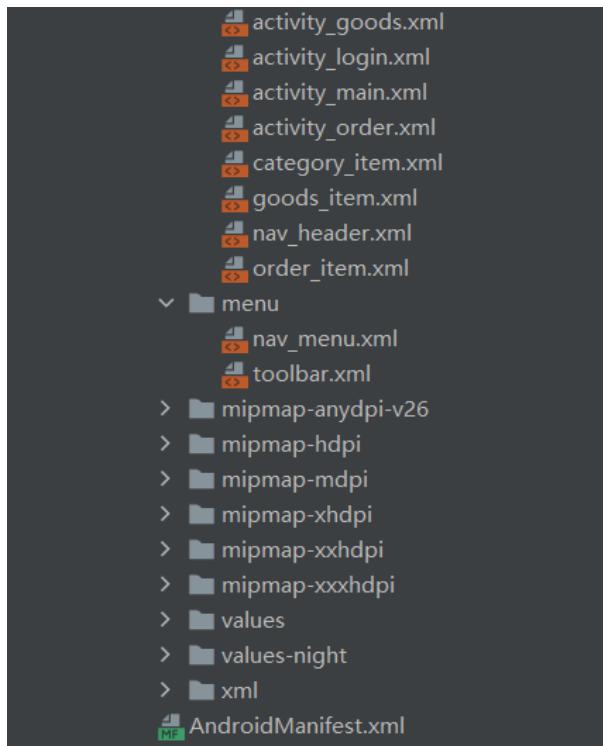
这是整个用户端的MVVM架构。实线箭头表示数据的流动方向；虚线箭头表示打开关系；每个箭头的颜色表示了数据是由哪个类掌管的，从Server或者

SharedPreferences开始按着一个颜色走才能走通，黑色表示公共路径。

最顶层的Activity是用户能看到的东西——打开程序会进入`LoginActivity`；登录成功后就会通过`Intent`打开`MainActivity`，`MainActivity`就是用来搜索各种商品和种类的；每点击一个商品都会打开`GoodsActivity`，也就是商品的详情页；另外在侧滑菜单点击`My Order`按钮还能打开`OrderActivity`，这里能看到用户所有的订单信息。

下面是整个客户端的代码结构：





## 3.1 登录界面

### 3.1.1 前端

这里给出登录界面的前端代码：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    <TextView
        android:id="@+id/login_info_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_above="@id/account_layout"
        android:text="Spread Shop"
        android:gravity="center"
        android:textSize="25sp"
        />

    <LinearLayout
```

```
        android:id="@+id/account_layout"
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="60dp"
        android:layout_above="@+id/password_layout"
        android:layout_margin="20dp"
    >

    <TextView
        android:layout_width="90dp"
        android:layout_height="match_parent"
        android:gravity="center_vertical"
        android:textSize="12sp"
        android:text="Account: "
    />

    <EditText
        android:id="@+id/account_edit"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_gravity="center_vertical"
        android:hint="Enter your username"
    />
</LinearLayout>

<LinearLayout
    android:id="@+id/password_layout"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="60dp"
    android:layout_above="@+id/remember_layout"
    android:layout_margin="20dp"
>

    <TextView
        android:layout_width="90dp"
        android:layout_height="match_parent"
        android:gravity="center_vertical"
        android:textSize="12sp"
        android:text="Password: "
    />
```

```
<EditText
    android:id="@+id/password_edit"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:layout_gravity="center_vertical"
    android:inputType="textPassword"
    android:hint="Enter your password"
    />

</LinearLayout>

<LinearLayout
    android:id="@+id/remember_layout"
    android:orientation="horizontal"
    android:layoutDirection="rtl"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_above="@+id/login_btn"
    >

    <CheckBox
        android:id="@+id/remember_pwd"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="18sp"
        android:text="Remember Password"
        />

</LinearLayout>

<Button
    android:id="@+id/login_btn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="Login"
    />
```

```
        android:layout_margin="20dp"
    />

    <Button
        android:id="@+id/register_btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Register"
        android:layout_below="@+id/login_btn"
        android:layout_margin="20dp"
    />

    <Button
        android:id="@+id/test_btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Force Login"
        android:layout_below="@+id/register_btn"
        android:layout_margin="20dp"
    />

</RelativeLayout>
```

这整个代码用的是RelativeLayout，每一个组件之间的布局都是相关的，它的布局方式是先把一个组件和父布局捆到一起。*LOGIN*按钮中有一句

`android:layout_centerInParent="true"`，表示我们把这个按钮放到整个页面的正中间，有个这个“源”之后，剩下的所有布局都可以以它为基准了。比方说*REGISTER*按钮中有一句`android:layout_below="@+id/login_btn"`，表示该按钮就在*LOGIN*的下方

然后是*Remember Password*这个选框，它是一个横向线性布局，由CheckBox和TextView两部分组成

再往上就是输入用户名密码的两个框，他俩也同样是横向线性布局，这里我们关注一下EditText下的这句代码：`android:layout_weight="1"`，这里我们把它的权重设为1，权重也就是这一部分在整个线性布局中所占的比例，实际作用是让EditText占用TextView以外所有的空间

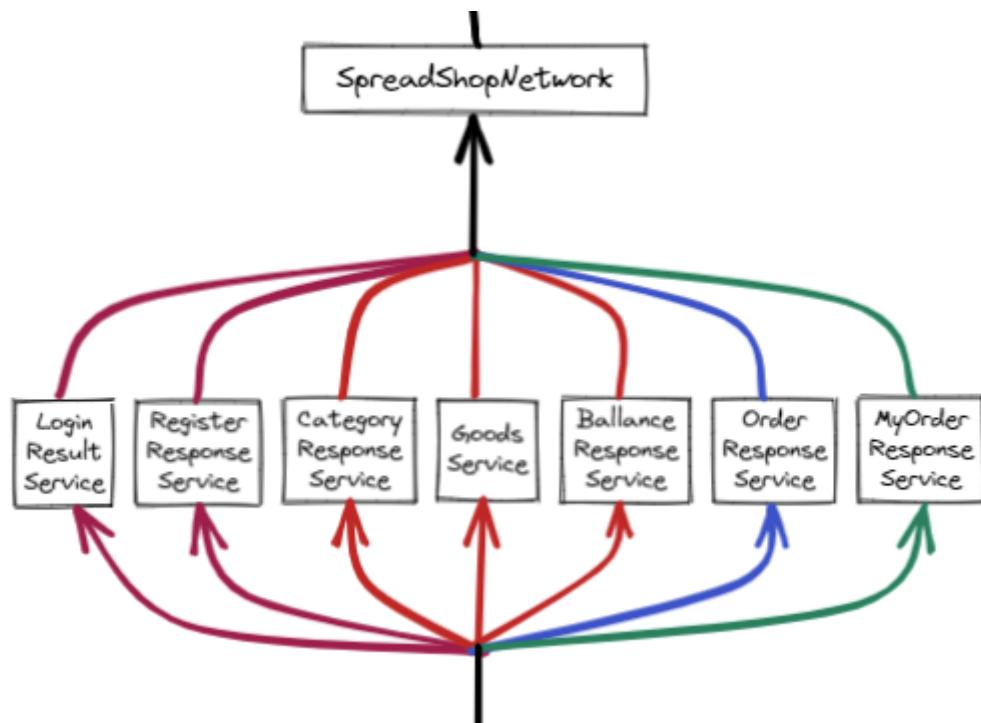
最顶部的TextView用来展示信息，最开始展示的是我们程序的名字“Spread Shop”。在用户执行登录/注册操作后，也可以用来展示相关信息，例如：“Regist success!!!”，“Unknown username”等

### 3.1.2 后端

来到`LoginActivity`的数据主要有以下三种，对应MVVM架构图中的粉线

1. 登录返回信息
2. 注册返回信息
3. 记住密码后的用户名与密码（注意，记住密码后，用户名和密码是保存到本地的，所以数据应该从本地的`SharedPreferences`传来）

之前在服务端中提到过，实现每个功能后返回给用户端的都是一个定义好的数据模型，将它打包成json文件通过Http协议发送过来，我们就可以在网络层接口利用Gson解析得到数据。而我们这里将所有数据模型的网络接口都封装成了一个`SpreadShopNetwork`，统一由它去发起网络请求，就是架构图中的这部分：



封装一次还是不够的，因为前面说过，我们的数据不仅是由网络传来的，还有本地传来的数据。我们需要统一封装到`Repository`中，让它作为数据的唯一来源，方便处理所有的数据。

数据经仓库往上传，就来到了`ViewModel`层，这里用来存储所有用来显示在屏幕的数据。我们这里着重看一下`LoginViewModel`中`LoginResult`的写法：

```
val loginResult: LiveData<Call<LoginResult>>
    get() = _loginResult
private val _loginResult = MutableLiveData<Call<LoginResult>>()
```

首先是一个暴露给外部的不可变接口`loginResult`，底下是一个可变的接口`_loginResult`，然后将`loginResult`的`get`属性值设置为`_loginResult`，这种写法是安卓官方推荐的。

```
fun getLoginResult(username: String, password: String){
    _loginResult.value = Repository.getLoginResult(username,
password)
}
```

`LoginViewModel`中的这个方法调用仓库层的`getLoginResult`将数据传到`_loginResult`的`value`字段中，它的值发生改变了，由于`loginResult`的`get`属性为`_loginResult`，`LoginResult`也会随之改变，我们就可以在`LoginActivity`中观察它的变化了：

```
loginViewModel.loginResult.observe(this){
    it.enqueue(object: Callback<LoginResult>{
        override fun onResponse(
            call: Call<LoginResult>,
            response: Response<LoginResult>
        ) {
            Log.d("SpreadShopTest", "on Response")
            val loginResult = response.body()
            if(loginResult != null){
                Log.d("SpreadShopTest", "loginResult.message: ${loginResult.message}")
                if(loginResult.success){
                    Log.d("SpreadShopTest", "Login Success")
                    loginViewModel.isSetFullyLiveData.value = true
                    val intent = Intent(this@LoginActivity,
MainActivity::class.java)
                    intent.putExtra("username",
loginResult.username)
                    startActivity(intent)
                }
            }
        }
    })
}
```

```
        }else{
            Log.d("SpreadShopTest", "Login Fail")
            bindingLogin.loginInfoText.text =
        loginResult.message
    }
    }else{
        Log.d("SpreadShopTest", "LoginResult is Null")
    }
}
override fun onFailure(call: Call<LoginResult>, t:
Throwable) {
    Log.d("SpreadShopTest", "on Failure")
    bindingLogin.loginInfoText.text = "Over Time"
    t.printStackTrace()
}
})
}
}
```

一旦LoginResult中的数据发生改变，我们就要执行它的回调方法，这里的it就是我们最终拿到的对象，它的类型是Call<LoginResult>，这里我们直接用Retrofit那一套去enqueue就可以了。接着就是处理成功与失败的状态

如果失败了，就会将显示在登录界面最上方的那行字改为“Over Time”

如果成功了，这里先注意一下这段代码的操作：

```
loginViewModel.isSetFullyLiveData.value = true
```

我们把又一个LiveData的的value值赋为true，这时我们也可以在LoginActivity中观察它了：

```
loginViewModel.isSetFullyLiveData.observe(this){
    if(it == true && bindingLogin.rememberPwd.isChecked){
        loginViewModel.saveUserInfo(this)
    }else{
        loginViewModel.clear(this)
    }
}
```

顺带提一下LiveData的特点：哪怕它原来的值就是true，我们再对它赋true，也会执行这里的方法，调用loginViewModel的saveUserInfo，层层调用将用户名密码存入本地数据库

接下来通过intent从LoginActivity跳转到MainActivity，与此同时我们往intent中传了一个用户名，这样我们在MainActivity里就能拿到用户名了

然后我们来看看从本地传数据的那条线，因为这部分在类外面，我们需要同时存用户名和密码，这属于两个信息，所以对应的是一个用户。这里我们必须将用户名和密码再进行一次封装，以UserInfo的形式才能比较合理的存进去

那么对于这个情况我们需要在UserInfoDao里去创建一个SharedPreferences，它能以键值对的方式将我们的数据存入数据库，键值对的键为"userinfo"，值为Gson().toJson(userInfo)，这里给出整个UserInfoDao的代码：

```
import android.content.Context
import androidx.core.content.edit
import com.example.spreadshop.logic.model.UserInfo
import com.google.gson.Gson

object UserInfoDao {
    fun saveUserInfo(userInfo: UserInfo, context: Context){
        context.getSharedPreferences("uinfo",
        Context.MODE_PRIVATE).edit {
            putString("userinfo", Gson().toJson(userInfo))
        }
    }

    fun getSavedUserInfo(context: Context): UserInfo{
        val userInfoJson = context.getSharedPreferences("uinfo",
        Context.MODE_PRIVATE).getString("userinfo", "")
        return Gson().fromJson(userInfoJson, UserInfo::class.java)
    }

    fun isUserInfoSaved(context: Context): Boolean =
    context.getSharedPreferences("uinfo",
    Context.MODE_PRIVATE).contains("userinfo")

    fun clear(context: Context) =
```

```
context.getSharedPreferences("uinfo",  
Context.MODE_PRIVATE).edit().clear().apply()  
}
```

另外这里遇到了一个小插曲，就是实现记住密码功能的时候。一开始我们是按照《第一行代码》中天气预报程序里保存搜索过的城市那样做的。其中使用了一个获取全局Context的方式，这个技术在购买成功自动返回MainActivity并发起网络请求功能时也会用到。但是在这里居然行不通，只要一调用程序就会崩溃。之所以我们这么做会崩溃，而书中却不会，最根本的原因就是：书中的**context是在Fragment中获取的，而我们是在Activity中获取的**。在Fragment中获取时，Activity已经创建好了，所以这样的代码是没问题的：

```
override fun onCreate() {  
    super.onCreate()  
    context = applicationContext  
}
```

但是我们现在做的操作是：在Activity的**onCreate**方法中去调用**applicationContext**方法，显然是不可能完成的，也就导致了**context**没有被初始化。所以，在Activity中想要将自己这个**context**传递出去，还是老老实实**把this当参数传出去吧**：

```
if(loginViewMode.isUserInfoSaved(this)){  
    val uinfo = loginViewMode.getSavedUserInfo(this)  
    // 给输入框设置值要用setText不能用语法糖  
    bindingLogin.accountEdit.setText(uinfo.username)  
    bindingLogin.passwordEdit.setText(uinfo.password)  
    bindingLogin.rememberPwd.isChecked = true  
}
```

## 3.2 主界面

### 3.2.1 前端

接下来就是主界面的部分，老样子，先看前端：

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.drawerlayout.widget.DrawerLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    <androidx.coordinatorlayout.widget.CoordinatorLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <com.google.android.material.appbar.AppBarLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content">

            <androidx.appcompat.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr actionBarSize"

                android:background="@color/material_dynamic_neutral60"

                android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
                    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
                    app:layout_scrollFlags="scroll|enterAlways|snap"
                />

            </com.google.android.material.appbar.AppBarLayout>

            <androidx.recyclerview.widget.RecyclerView
                android:id="@+id/category_recycler"
                android:layout_width="match_parent"
                android:layout_height="50dp"

                app:layout_behavior="@string/appbar_scrolling_view_behavior"
            />
        <!--
            想让谁实现下拉刷新功能，就把谁放到SwipeRefreshLayout里，  

            记得添加依赖  

            recyclerView里面的layout_behavior搬到外面
        -->
        <androidx.swiperefreshlayout.widget.SwipeRefreshLayout
            android:id="@+id/swipe_refresh"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="110dp"
        app:layout_anchor="@+id/category_recycler"
        app:layout_anchorGravity="bottom"

    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    >

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/goods_recycler"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
    />

</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>

</androidx.coordinatorlayout.widget.CoordinatorLayout>

<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:menu="@menu/nav_menu"
    app:headerLayout="@layout/nav_header"
/>

</androidx.drawerlayout.widget.DrawerLayout>
```

整体上用了Material Design DrawerLayout这种抽屉布局

第一部分是一个CoordinatorLayout协调布局， CoordinatorLayout是加强版的FrameLayout， 它专为MaterialDesign设计， 能够监听其中控件的变化

先看AppBarLayout部分， 这部分只包含一个Toolbar， Toolbar也就是我们程序主界面最上方的那一条， 其中的属性没什么好说的， 最主要的是Toolbar的专用菜单：

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:app="http://schemas.android.com/apk/res-auto">
<!--
    always: 永远显示在Toolbar中，空间不够不显示
    ifRoom: 屏幕够就显示，不够显示再菜单中
    never: 永远显示在菜单
-->
<item
    android:id="@+id/search_edit"
    android:icon="@drawable/ic_search"
    app:actionViewClass="android.widget.SearchView"
    app:showAsAction="always|collapseActionView"
    android:title="search_edit"
/>
</menu>
```

这里我们只添加了一个项，`app:actionViewClass`就是制定当前的item到底是什么类型的，这里它的类是SearchView，也就是主界面右上角的搜索按钮

接着我们注意到下方的两个RecyclerView

一个独立出来——用于展示种类，我们在当中嵌入了category\_item：

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    app:rippleColor="@color/material_dynamic_neutral90"
    android:layout_marginEnd="5dp"
>

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        >

        <ImageView
            android:id="@+id/category_image"
            android:layout_width="wrap_content"
            >
```

```
        android:layout_height="match_parent"
        android:layout_gravity="center_vertical"
        android:scaleType="fitCenter"

    android:background="@color/material_dynamic_neutral90"
    />

    <TextView
        android:id="@+id/category_name"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:layout_gravity="center"
        android:text="test"
        android:gravity="center"
        android:textColor="@color/black"
        android:textSize="16sp"

    android:background="@color/material_dynamic_neutral70"
    />

    </LinearLayout>

</com.google.android.material.card.MaterialCardView>
```

它是一个MaterialCardView，采用了横向线性布局，每部分由ImageView和TextView组成。这里的精髓还是在最外层布局：width是wrap，所以横向只包住图片和文字的宽度；而height是match，而它的parent正好就是上面的RecyclerView，而它的高度是50dp，所以卡片的高度也是50dp。TextView的部分和登录界面类似，这里不做赘述。我们主要来看看ImageView的这段代码——`android:scaleType="fitCenter"`，它的意思是将我们的原图片缩放到合适的大小并居中

另一个则被SwipeRefreshLayout包裹住——用于展示商品，同时由于他被SwipeRefreshLayout包裹，能够实现上滑刷新的功能，我们在当中嵌入了goods\_item：

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
xmlns:app="http://schemas.android.com/apk/res-auto"
android:layout_width="match_parent"
android:layout_height="wrap_content"
app:rippleColor="@color/material_dynamic_neutral90"
android:layout_margin="5dp"
/>>
```

<!--

一张Material卡片，之后这一张张卡片都会添加到  
recyclerView当中

centerCrop：让图片保持原有比例填充满ImageView，  
并将超出屏幕的部分裁剪掉

-->

```
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />

<ImageView
    android:id="@+id/goods_image"
    android:layout_width="match_parent"
    android:layout_height="100dp"
    android:scaleType="centerCrop"
    />

<TextView
    android:id="@+id/goods_name"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_margin="5dp"
    android:text="test"
    android:textSize="16sp"
    android:gravity="center"
    android:background="@color/material_dynamic_neutral70"
    android:textColor="@color/black"
    />

</LinearLayout>
```

```
</com.google.android.material.card.MaterialCardView>
```

它和category\_item非常相似，唯一不同的是这部分采用的是纵向线性布局，即上图下字而非左图右字的排布方式，其他的这里就不多赘述了

第二部分是一个NavigationView，也就是我们程序中的侧滑菜单，它也由两部分组成：头部headerLayout和菜单menu，它们分别对应两个布局

我们先来看nav\_header，它用的是RelativeLayout布局：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:padding="10dp"
    android:background="@color/material_dynamic_primary40"
    android:layout_height="200dp">

    <de.hdodenhof.circleimageview.CircleImageView
        android:id="@+id/user_icon"
        android:layout_width="70dp"
        android:layout_height="70dp"
        android:src="@drawable/nav_user"
        android:layout_centerInParent="true"
        />

    <TextView
        android:id="@+id/app_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="Spread Shop"
        android:textColor="@color/black"
        android:textSize="14sp"
        />

    <TextView
        android:id="@+id/user_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
```

```
        android:layout_above="@+id/app_name"
        android:textColor="@color/material_dynamic_neutral60"
        android:text="user name: null"
    />

    <TextView
        android:id="@+id/ballance"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_alignParentBottom="true"
        android:textColor="@color/material_dynamic_neutral60"
        android:text="ballance: 0"
    />

</RelativeLayout>
```

我们先用了一个CircleImageView，它能调用第三方库把我们的头像图片截取成圆形，更加美观；紧接着是三个TextView，分别显示我们的用户名、余额和“Spread Shop”

然后是nav\_menu，它用的是一个menu：

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_mybag"
            android:title="My Bag"
        />

        <item
            android:id="@+id/nav_order"
            android:title="My Order"
        />

        <item
            android:id="@+id/nav_contact"
            android:title="Contact Customer Service"
        />
    </group>
</menu>
```

```
<item
    android:id="@+id/nav_logout"
    android:title="Logout"
/>
</group>
</menu>
```

当中共有四个项，group表示这些项被分成一个组；

checkableBehavior="single"表示同时只能选中一个，如果不指定的话理论上可以同时选择多个，因为它不是一个个单独的按钮而是一个可以多选的菜单。这里由于时间原因我们只实现了My Order功能

### 3.2.2 后端

主界面的后端是我们整个用户端最复杂的一部分，Repository以下部分的封装之前在登录界面已经解释过了，我们这里直接从MainViewModel看起

```
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import com.example.spreadshop.logic.Repository
import com.example.spreadshop.logic.model.-
import retrofit2.Call

class MainViewModel: ViewModel() {

    val goodsList = ArrayList<Goods>()
    val categoryList = ArrayList<Category>()

    var username = ""
    var balance = ""

    val isGotGoodsLiveData = MutableLiveData<Boolean>()
    val isGotCategoryLiveData = MutableLiveData<Boolean>()

    val goodsLiveData: LiveData<Call<GoodsResponse>>
        get() = _goodsLiveData
    private val _goodsLiveData =
        MutableLiveData<Call<GoodsResponse>>()

    val categoryLiveData: LiveData<Call<CategoryResponse>>
```

```

        get() = _categoryLiveData
    private val _categoryLiveData =
MutableLiveData<Call<CategoryResponse>>()

    val balanceResponseLiveData: LiveData<Call<BalanceResponse>>
        get() = _balanceLiveData
    private val _balanceLiveData =
MutableLiveData<Call<BalanceResponse>>()

    fun getGoods(cmd: String) {
        _goodsLiveData.value = Repository.getGoods(cmd)
    }

    fun getAllCategory(cmd: String){
        _categoryLiveData.value = Repository.getAllCategory(cmd)
    }

    fun getBalance(){
        _balanceLiveData.value = Repository.getBalance(username)
    }
}

```

*MainViewModel*和*LoginViewModel*最大的不同就是——在登录界面我们不需要展示出从网络层拿回来的数据；而在主界面我们需要把种类信息、商品名称展示在Recycler上，并且我们也需要这些信息去加载图片。

既然我们需要把数据展示在Recycler上，就需要用到RecycleView的Adapter，就需要用到List去传递数据，所以我们必须要在*MainViewModel*里定义两个List

```

val goodsList = ArrayList<Goods>()
val categoryList = ArrayList<Category>()

```

Adapter的写法就比较固定，这里我们看一下两个Adapter

首先是GoodsAdapter，主要的操作就是，当用户点击了每一项，都要打开商品详情的Activity：

```

class GoodsAdapter(val context: Context, val goodsList:
List<Goods>): RecyclerView.Adapter<GoodsAdapter.ViewHolder>(){

```

```
    inner class ViewHolder(view: View):  
RecyclerView.ViewHolder(view){  
    val goodsImage: ImageView =  
view.findViewById(R.id.goods_image)  
    val goodsName: TextView = view.findViewById(R.id.goods_name)  
}  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType:  
Int): ViewHolder {  
    val view =  
LayoutInflater.from(context).inflate(R.layout.goods_item, parent,  
false)  
    val holder = ViewHolder(view)  
    holder.itemView.setOnClickListener {  
        val mainActivity = context as MainActivity  
        val position = holder.adapterPosition  
        val goods = goodsList[position]  
        val intent = Intent(context,  
GoodsActivity::class.java).apply {  
            putExtra("goods_name", goods.goods_name)  
            putExtra("goods_storage", goods.goods_storage)  
            putExtra("goods_price", goods.goods_price)  
            putExtra("goods_category", goods.goods_category)  
            putExtra("goods_id", goods.goods_id)  
            putExtra("username",  
mainActivity.viewModel.username)  
        }  
        context.startActivity(intent)  
    }  
    return holder  
}  
//    return ViewHolder(view)  
}  
  
    override fun onBindViewHolder(holder: ViewHolder, position: Int)  
{  
    val goods = goodsList[position]  
    holder.goodsName.text = goods.goods_name  
    val id = "ic_goods_${goods.goods_id}"  
  
    Glide.with(context).load(Command.getImageByReflect(id)).into(holder.  
goodsImage)  
}
```

```
    }

    override fun getItemCount() = goodsList.size
}
```

intent的部分特别简单，就不多说了。重要的是这个Glide的改变。我们之前只加载一个图片，而现在我们对于不同的商品能展示不同的图片了。这里我们定义了一个getImageByReflect函数，这个函数就是通过反射去获取真正的图片id：

```
fun getImageByReflect(imageName: String): Int{
    var field: Class<*>
    var res = 0
    try {
        field = Class.forName("com.example.spreadshop.R\$drawable")
        res = field.getField(imageName).getInt(field)

    }catch (e: java.lang.Exception){
        e.printStackTrace()
        Log.d("SpreadShopTest", "getImageByReflect exception!")
    }

    return res
}
```

我们都知道，在drawable下创建了文件ic\_test.png，那么对应的就会在R.drawable下新建一个Int类型的变量叫做ic\_test。我们这个函数的目的就是通过前者去寻找后者。其中需要注意的是这个内部类的写法：

com.example.spreadshop.R\\$drawable在java中，我们只需要这么写：  
com.example.spreadshop.R\\$drawable，但是kotlin增加了字符串嵌套变量的操作，所以要转义一下。经过这么一个转换，我们就能按照goods\_id去加载手机中已经存好的照片(这样其实是不对的，但是也没办法，我们还不知道怎么在MySQL中存图片)

接下来是CategoryAdapter：

```
class CategoryAdapter(val context: Context, val categoryList:
List<Category>): RecyclerView.Adapter<CategoryAdapter.ViewHolder>(){

    inner class ViewHolder(view: View):
```

```
RecyclerView.ViewHolder(view){
    val categoryName: TextView =
view.findViewById(R.id.category_name)
    val categoryImage: ImageView =
view.findViewById(R.id.category_image)
}

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): ViewHolder {
    val view =
LayoutInflater.from(context).inflate(R.layout.category_item, parent,
false)
    val holder = ViewHolder(view)
    holder.itemView.setOnClickListener {
        val position = holder.adapterPosition
        val category = categoryList[position]
        val mainActivity = context as MainActivity

        mainActivity.viewModel.getGoods(Command.getCategoryGoods(category
.y.category))
    }
    return holder
//    return ViewHolder(view)
}

    override fun onBindViewHolder(holder: ViewHolder, position: Int)
{
    val category = categoryList[position]
    holder.categoryName.text = category.category
    Log.d("SpreadShopTest", "category name:
${category.category}")
    val id: Int
    when(category.category){
        "手机" -> {
            Log.d("SpreadShopTest", "category id: 手机")
            id = R.drawable.ic_phone
        }
        "衣服" -> {
            Log.d("SpreadShopTest", "category id: 衣服")
            id = R.drawable.ic_cloth
        }
        "裤子" -> {
    }
}
```

```

        Log.d("SpreadShopTest", "category id: 裤子")
        id = R.drawable.ic_trousers
    }
    else -> {
        Log.d("SpreadShopTest", "category id: else")
        id = R.drawable.test_maotai
    }
}

Log.d("SpreadShopTest", "val id: $id")

Glide.with(context).load(id).into(holder.categoryImage)
}

override fun getItemCount() = categoryList.size
}

```

这里的变化就是，我们要对每一项设置监听器：当点击这个类别时，就发起按着这个类别去找商品的请求。而返回类型是GoodsResponse，那么自然就会刷新下面的GoodsRecyclerView了

另外，我们不是将这个RecyclerView改成了横着的吗？只需要这么一句话：

```

val categoryLayoutManager = GridLayoutManager(this, 1)
categoryLayoutManager.orientation = LinearLayoutManager.HORIZONTAL
bindingMain.categoryRecycler.layoutManager = categoryLayoutManager
val categoryAdapter = CategoryAdapter(this,
mainViewModel.categoryList)
bindingMain.categoryRecycler.adapter = categoryAdapter

```

就是第二行中的这句话，将方向变成了水平的

*MainActivity*的代码模块结构与*LoginActivity*比较相似的部分就不赘述了，我们着重看一些不同点

首先我们需要设置好两个Adapter

```

// 设置goodsRecycler的adapter
// 由于ArrayList的构造函数，所以goodsList不为空
val goodsLayoutManager = GridLayoutManager(this, 2)

```

```
bindingMain.goodsRecycler.layoutManager = goodsLayoutManager
val goodsAdapter = GoodsAdapter(this, mainViewModel.goodsList)
bindingMain.goodsRecycler.adapter = goodsAdapter

// 设置categoryRecycler的adapter
val categoryLayoutManager = GridLayoutManager(this, 1)
categoryLayoutManager.orientation = LinearLayoutManager.HORIZONTAL
bindingMain.categoryRecycler.layoutManager = categoryLayoutManager
val categoryAdapter = CategoryAdapter(this,
mainViewModel.categoryList)
bindingMain.categoryRecycler.adapter = categoryAdapter
```

接着我们需要在isGotGoodsLiveData.observe()最后添加这样一段代码：

```
runOnUiThread {
    goodsAdapter.notifyDataSetChanged()
    bindingMain.swipeRefresh.isRefreshing = false
}
```

在isGotCategoryLiveData.observe()最后添加这样一段代码：

```
runOnUiThread {
    categoryAdapter.notifyDataSetChanged()
}
```

`notifyDataSetChanged()` 的作用就是告诉对应的Adapter数据已经改变，让它们显示新的数据。`bindingMain.swipeRefresh.isRefreshing = false` 的作用则是我们在上滑刷新商品时，刷新成功后让刷新图标不再显示。

另外，我们[前面](#)提到过`LoginActivity`在跳转到`MainActivity`时往`intent`里传了一个用户名，这个用户名在这里有什么用呢？我们在`MainActivity`中需要拿到这个用户名，用这个用户名去获取账户余额，并将用户名和余额显示在侧滑菜单的对应位置上，对应下面这段代码：

```
mainViewModel.username =
intent.getStringExtra("username").toString()

if(mainViewModel.username != "") mainViewModel.getBalance()

if(bindingMain.navView.headerCount > 0){
```

```
    val header = bindingMain.navView.getHeaderView(0)
    val uname = header.findViewById<TextView>(R.id.user_name)
    uname.text = "user name: ${mainViewModel.username}"
}
```

然后就是侧滑菜单的具体实现：

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when(item.itemId){
        android.R.id.home ->
        bindingMain.drawerLayout.openDrawer(GravityCompat.START)
    }
    return true
}
```

当我们点击home按钮或者侧滑之后，它就会打开drawerLayout，唤起我们的侧滑菜单

接着我们看看SearchView搜索的逻辑代码。由于我们将它放在了Toolbar中作为菜单的一项，那么它的出生自然要在onCreateOptionsMenu方法中了：

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.toolbar, menu)

    val searchItem = menu?.findItem(R.id.search_edit)
    //searchEdit = findViewById(R.id.search_edit)
    //searchEdit = MenuItemCompat.getActionView(searchItem) as
    SearchView
    searchEdit = searchItem?.actionView as SearchView

    searchEdit.isSubmitButtonEnabled = true
    searchEdit.imeOptions = EditorInfo.IME_ACTION_SEARCH

    searchEdit.setOnQueryTextListener(object :
    SearchView.OnQueryTextListener{
        override fun onQueryTextSubmit(query: String?): Boolean {
            if(query != null){
                mainViewModel.getGoods(Command.getSearchGoods(query))
            }else{

```

```
        Log.d("SpreadShopTest", "SearchView:  
query is null")  
    }  
  
    inputMethodManager.hideSoftInputFromWindow(searchEdit.windowToken,  
InputMethodManager.HIDE_NOT_ALWAYS)  
    return true  
}  
  
    override fun onQueryTextChange(newText: String?):  
Boolean {  
    if(newText == ""){  
  
        mainViewModel.getGoods(Command.GET_RECOMMAND)  
    }  
    return true  
}  
}  
}
```

我们可以通过这两行代码从menu中获取到SearchView的实例：

```
val searchItem = menu?.findItem(R.id.search_edit)  
searchEdit = searchItem?.actionView as SearchView
```

注释掉的是Java中已经过时的写法。接下来是对这个SearchView的参数进行一些设置。这里给出参考的一些网站：

[Android的SearchView详解 \(wjhsh.net\)](http://wjhsh.net)

[详细解读Android中的搜索框 \(三\) —— SearchView - developer\\_Kale - 博客园 \(cnblogs.com\)](http://www.cnblogs.com/developer_kale/p/3754007.html)

然后我们进行两个比较重要的设置：点击提交按钮发生的事，以及清空输入框发生的事。这里的代码很好看懂，唯一要注意的是在点击提交按钮后，要使用 `hideSoftInputFromWindow` 关闭输入法，这样能增加用户体验。

最后我们还需要两个intent去打开GoodsActivity和OrderActivity，打开GoodsActivity的intent我们在GoodsAdapter中已经实现了。而打开

*OrderActivity*的intent，我们将它写在侧滑菜单的监听器中：

```
bindingMain.navView.setNavigationItemSelectedListener {
    when(it.itemId){
        R.id.nav_mybag -> Log.d("SpreadShopTest", "You clicked
mybag")
        R.id.nav_order -> {
            Log.d("SpreadShopTest", "You clicked myorder")
            val intent = Intent(this@MainActivity,
OrderActivity::class.java).apply {
                putExtra("username", mainViewModel.username)
            }
            startActivity(intent)
        }
        R.id.nav_contact -> Log.d("SpreadShopTest", "You clicked
Contact")
        R.id.nav_logout -> Log.d("SpreadShopTest", "You clicked
logout")
    }
    bindingMain.drawerLayout.closeDrawers()
    true
}
```

当我们点击My Order后，就会通过intent去打开*OrderActivity*

## 3.3 商品详情页

接下来是商品的详情页。这部分其实和《第一行代码》的12.7是一个模子，所以不重要的部分就一步带过了。

### 3.3.1 前端

首先是前端代码：

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#fff">
```

```
        android:fitsSystemWindows="true"
    >

    <com.google.android.material.appbar.AppBarLayout
        android:id="@+id/app_bar"
        android:layout_width="match_parent"
        android:layout_height="250dp"
        android:fitsSystemWindows="true"
    >

    <com.google.android.material.appbar.CollapsingToolbarLayout
        android:id="@+id/collapsing_toolbar"
        android:layout_width="match_parent"
        android:layout_height="match_parent"

        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:contentScrim="@color/teal_200"
        app:layout_scrollFlags="scroll|exitUntilCollapsed">

        <ImageView
            android:id="@+id/goods_image_detail"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scaleType="centerCrop"
            app:layout_collapseMode="parallax"
        />

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar_detail"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_collapseMode="pin"
        />

    </com.google.android.material.appbar.CollapsingToolbarLayout>
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"

        app:layout_behavior="@string/appbar_scrolling_view_behavior">
```

```
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <com.google.android.material.card.MaterialCardView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="15dp"
        android:layout_marginLeft="15dp"
        android:layout_marginRight="15dp"
        android:layout_marginTop="35dp"
        app:cardCornerRadius="4dp"
    >

    <TextView
        android:id="@+id/goods_text_detail"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="10dp"
    />
    </com.google.android.material.card.MaterialCardView>
    </LinearLayout>

</androidx.core.widget.NestedScrollView>

<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/buy_btn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:src="@drawable/ic_backup"
    app:layout_anchor="@+id/app_bar"
    app:layout_anchorGravity="bottom|end"
    android:contentDescription="buy goods"
/>

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

这里用到了可折叠式标题栏，具体效果是我们在商品详情页上/下滑动时，商品图片会以渐变的形式收入Toolbar或者从Toolbar抽出

### 3.3.2 后端

然后是GoodsActivity和GoodsViewModel。没错，这里的逻辑又是比较复杂的，我们从Activity一步一步说起

首先是展示Home键，这个键默认就是个返回的箭头，所以不用动：

```
setSupportActionBar(bindingGoods.toolbarDetail)
supportActionBar?.setDisplayHomeAsUpEnabled(true)
```

然后是一些ViewModel的设置：

```
goodsViewModel =
ViewModelProvider(this).get(GoodsViewModel::class.java)

goodsViewModel.setGoodsName(intent.getStringExtra("goods_name") ?: "null")
goodsViewModel.setGoodsCategory(intent.getStringExtra("goods_category") ?: "null")
goodsViewModel.setGoodsPrice(intent.getIntExtra("goods_price", 999))
goodsViewModel.setGoodsStorage(intent.getIntExtra("goods_storage", -1))
goodsViewModel.setGoodsId(intent.getIntExtra("goods_id", -1))
goodsViewModel.username = intent.getStringExtra("username") ?: ""
goodsViewModel.isSetFullyLiveData.value = true
```

都是我们从intent拿到的数据，放到了当前Activity的ViewModel层。当所有的LiveData都设置完成后，将`isSetFullyLiveData`置为true。没错，这和我们前面的代码很相似。也就是`isGotxxxLiveData`这样的逻辑。那么一旦设置了这个值，就该调用这个方法了：

```
goodsViewModel.isSetFullyLiveData.observe(this){
    if(it == true){
        bindingGoods.collapsingToolbar.title =
        goodsViewModel.goodsNameLiveData.value
        val imgId =
```

```
"ic_goods_${goodsViewModel.goodsIdLiveData.value}"  
  
Glide.with(this).load(Command.getImageByReflect(imgId)).into(binding  
Goods.goodsImageDetail)  
    bindingGoods.goodsTextDetail.text = generateGoodsDetail()  
} else {  
    Log.d("SpreadShopTest", "isSetFullyLiveData: false")  
}  
}  
}
```

这样我们就能将标题，图片，详细信息等乱七八糟的信息都加载上了。这里用到的`generateGoodsDetail`函数是这样的：

```
private fun generateGoodsDetail(): String{  
    val res = StringBuilder()  
    res.appendLine("goods_name:  
${goodsViewModel.goodsNameLiveData.value}")  
    res.appendLine("goods_category:  
${goodsViewModel.goodsCategoryLiveData.value}")  
    res.appendLine("goods_price:  
${goodsViewModel.goodsPriceLiveData.value}")  
    res.appendLine("goods_storage:  
${goodsViewModel.goodsStorageLiveData.value}")  
    val sb = goodsViewModel.goodsNameLiveData.value?.repeat(500)  
    res.append(sb)  
    return res.toString()  
}
```

接下来是购买功能，我们打算用这个`FloatButton`去当购买按键，当点击之后，弹出一个窗口询问你购买的数量。而这时我们恰好找到了一个非常好用的第三方库——`XPopup`：

[XPopup: 🔥 XPopup2.0版本重磅来袭，2倍以上性能提升，带来可观的动画性能优化和交互细节的提升！！！功能强大，交互优雅，动画丝滑的通用弹窗！可以替代Dialog, PopupWindow, PopupMenu, BottomSheet, DrawerLayout, Spinner等组件，自带十几种效果良好的动画，支持完全的UI和动画自定义！\(Powerful and Beautiful Popup, can absolutely replace Dialog, PopupWindow, PopupMenu, BottomSheet, DrawerLayout, Spinner. With built-in animators, very easy to custom popup view.\) \(gitee.com\)](#)

安装的时候遇到一个小问题，就是gradle7.0之后所有的仓库引入都放到了settings.properties中了：

```
dependencyResolutionManagement {  
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)  
    repositories {  
        google()  
        mavenCentral()  
        maven {url 'https://jitpack.io'} // XPopup的依赖  
    }  
}
```

这个库非常好用，所以这里直接给出所有的代码了，一看就能看懂：

```
bindingGoods.buyBtn.setOnClickListener {  
  
    XPopup.Builder(this@GoodsActivity).asInputConfirm("Buying  
${goodsViewModel.goodsNameLiveData.value}", "Please enter the  
purchase quantity: ", "1", object: OnInputConfirmListener{  
    override fun onConfirm(text: String?) {  
        if(goodsViewModel.isSetFullyLiveData.value  
== true && text != null && text != "" && text.isDigitsOnly()){  
  
            goodsViewModel.requestOrder(goodsViewModel.username,  
            goodsViewModel.goodsIdLiveData.value!!, text.toInt())  
            }else if(text == ""){  
  
            goodsViewModel.requestOrder(goodsViewModel.username,  
            goodsViewModel.goodsIdLiveData.value!!, 1)  
            Log.d("SpreadShopTest", "Only Buy  
One!")  
            }else{  
            Log.d("SpreadShopTest", "XPopInput  
return Nothing, buy fail")  
            return  
            }  
        }  
    }).show()  
}
```

这里我们还很贴心的给用户设置了一个默认值1，表示不输入默认购买一件，并且XPopup正好还支持提示，所以在提示里打上"1"就好了

接着我们要处理OrderResponse：

```
goodsViewModel.orderLiveData.observe(this){
    it.enqueue(object : Callback<OrderResponse>{
        override fun onResponse(
            call: Call<OrderResponse>,
            response: Response<OrderResponse>
        ) {
            val orderResponse = response.body()
            if(orderResponse != null){
                val order = orderResponse.order
                if(orderResponse.success){
                    XPopup.Builder(this@GoodsActivity).asConfirm("Order Info",
                    order.message) {
                        val mainActivity = SpreadShopApplication.context as
MainActivity

                    mainActivity.mainViewModel.getGoods(Command.GET_RECOMMAND)
                        this@GoodsActivity.finish()
                    }.show()
                }else{
                    Log.d("SpreadShopTest", "orderResponse.success
is fail, msg: ${order.message}")
                    XPopup.Builder(this@GoodsActivity).asConfirm("Failed!!!",
                    order.message)
                } {
                    Toast.makeText(
                        this@GoodsActivity,
                        "buy failed over",
                        Toast.LENGTH_SHORT
                    ).show()
                }.show()
            }
        }else{
            Log.d("SpreadShopTest", "orderResponse is null")
        }
    }
}

override fun onFailure(call: Call<OrderResponse>, t:
```

```
Throwable) {
    t.printStackTrace()
    Log.d("SpreadShopTest", "orderResponse on failure")
}
})
}
```

这里唯一需要注意的是这句话：

```
val mainActivity = SpreadShopApplication.context as MainActivity
mainActivity.mainViewModel.getGoods(Command.GET_RECOMMAND)
this@GoodsActivity.finish()
```

当购买成功，用户点击确定后，我们首先再发起一次获取推荐请求，然后关闭当前Activity。这样购买完自动会回到商城页面，并且数据也是最新的。而这个获取到>MainActivity的实例在《第一行代码》中的14.1有讲。但为啥这里获得的context就恰好是>MainActivity呢？我们之后在实现记住密码功能时得到了这个问题的[答案](#)

结项时我们还是觉得这中写法有问题，我们只需要在>MainActivity中的对应方法中设置，当GoodsActivity被显示在最上层时就发起这个请求即可，没必要这么麻烦

## 3.4 订单表

### 3.4.1 前端

这部分的前端非常简单，用到的都是前面提到的知识，这里直接给出：

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

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/order_toolbar"
        >
```

```
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="@color/material_dynamic_neutral60"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
    />

<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/order_refresh"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/order_recycler"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
</LinearLayout>
```

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="5dp"
    >

    <TextView
        android:id="@+id/order_username"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />

    <TextView
        android:id="@+id/order_id"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
```

```
<TextView
    android:id="@+id/order_time"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>

<TextView
    android:id="@+id/order_goods_name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>

<TextView
    android:id="@+id/order_goods_num"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="-----"
/>

</LinearLayout>
```

### 3.4.2 后端

后端部分也是老生常谈，设置Adapter，加载，把各种数据在RecyclerView上一显示，就完事了

*OrderActivity* :

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import android.view.MenuItem
import androidx.lifecycle.ViewModelProvider
import androidx.recyclerview.widget.LinearLayoutManager
```

```
import com.example.spreadshop.R
import com.example.spreadshop.databinding.ActivityOrderBinding
import com.example.spreadshop.logic.model.MyOrderResponse
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response

class OrderActivity : AppCompatActivity() {

    private lateinit var bindingOrder: ActivityOrderBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        bindingOrder = ActivityOrderBinding.inflate(layoutInflater)
        setContentView(bindingOrder.root)

        setSupportActionBar(bindingOrder.orderToolbar)
        supportActionBar?.setDisplayHomeAsUpEnabled(true)

        val orderViewModel = ViewModelProvider(this)
        [OrderViewModel::class.java]
        orderViewModel.username =
        intent.getStringExtra("username").toString()
        if(orderViewModel.username != ""){
            Log.d("SpreadShopTest", "OrderActivity: getting My
Order, username=${orderViewModel.username}")
            orderViewModel.getMyOrder()
        }else{
            Log.d("SpreadShopTest", "OrderActivity: get fail")
        }

        val orderLayoutManager = LinearLayoutManager(this)
        bindingOrder.orderRecycler.layoutManager =
        orderLayoutManager
        val orderAdapter = OrderAdapter(this,
        orderViewModel.orderList)
        bindingOrder.orderRecycler.adapter = orderAdapter

        orderViewModel.myOrderResponseLiveData.observe(this){
            it.enqueue(object : Callback<MyOrderResponse>{
                override fun onResponse(
                    call: Call<MyOrderResponse>,
```

```
        response: Response<MyOrderResponse>
    ) {
    val myOrderResponse = response.body()
    if(myOrderResponse != null){
        if(myOrderResponse.success){
            val list = myOrderResponse.orderList
            orderViewModel.orderList.clear()
            orderViewModel.orderList.addAll(list)
            orderViewModel.isGotOrderLiveData.value
            = true
        }
        for (order in list){
            Log.d("SpreadShopTest",
            order.goods_name)
        }
    }else{
        Log.d("SpreadShopTest", "My Order fail")
    }
}else{
    Log.d("SpreadShopTest", "My OrderResponse is
null")
}
}

override fun onFailure(call: Call<MyOrderResponse>,
t: Throwable) {
    Log.d("SpreadShopTest", "My Order on failure")
    t.printStackTrace()
}
}

orderViewModel.isGotOrderLiveData.observe(this){
    if(it == true){
        Log.d("SpreadShopTest", "is got order live data
true")
        runOnUiThread {
            orderAdapter.notifyDataSetChanged()
            bindingOrder.orderRefresh.isRefreshing = false
        }
    }
}
```

```

        bindingOrder.orderRefresh.setOnRefreshListener {
            orderViewModel.getMyOrder()
        }

    }// end onCreate

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when(item.itemId){
            android.R.id.home -> {
                finish()
                return true
            }
        }
        return super.onOptionsItemSelected(item)
    }
}

```

*OrderAdapter :*

```

import android.content.Context
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.example.spreadshop.R
import com.example.spreadshop.logic.model.Bag

class OrderAdapter(val context: Context, val orderList: List<Bag>): RecyclerView.Adapter<OrderAdapter.ViewHolder>() {
    inner class ViewHolder(view: View): RecyclerView.ViewHolder(view){
        val username: TextView =
        view.findViewById(R.id.order_username)
        val orderId: TextView = view.findViewById(R.id.order_id)
        val orderTime: TextView = view.findViewById(R.id.order_time)
        val orderGoodsName: TextView =
        view.findViewById(R.id.order_goods_name)
        val orderGoodsNum: TextView =
        view.findViewById(R.id.order_goods_num)
    }
}

```

```

    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view =
LayoutInflater.from(context).inflate(R.layout.order_item, parent,
false)
        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val order = orderList[position]
        Log.d("SpreadShopTest", "OrderAdapter:
name=${order.goods_name}")
        holder.username.text = "username: ${order.account_name}"
        holder.orderId.text = "order id: ${order.order_id}"
        holder.orderTime.text = "order time: ${order.order_time}"
        holder.orderGoodsName.text = "goods name:
${order.goods_name}"
        holder.orderGoodsNum.text = "goods num:
${order.goods_number}"
    }

    override fun getItemCount() = orderList.size
}

```

*OrderViewModel :*

```

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import com.example.spreadshop.logic.Repository
import com.example.spreadshop.logic.model.Bag
import com.example.spreadshop.logic.model.MyOrderResponse
import retrofit2.Call

class OrderViewModel: ViewModel() {
    val myOrderResponseLiveData: LiveData<Call<MyOrderResponse>>
        get() = _myOrderResponse
    private val _myOrderResponse =

```

```
MutableLiveData<Call<MyOrderResponse>>()

var username = ""

val orderList = ArrayList<Bag>()

val isGotOrderLiveData = MutableLiveData<Boolean>()

fun getMyOrder(){
    _myOrderResponse.value = Repository.getMyOrder(username)
}

}
```

## 3.5 其他

**Spread Zhao:** 用户端开发的过程中遇到了许多问题，上面展示的只是最终成型的版本。中间遇到了大大小小的问题，在这里我将我开发的日志展示出来以供参考。另外，上面的许多细节介绍其实也是引用了我的日志。

work\_on\_spreadshop

# 2022-10-20

昨天在写Retrofit的时候，使用了json传递数据，对象是拿到了，但是返回的都是空。后来才突然想起来， **data class里对应的成员名要和json中完全一致才可以！**

这是要传的json：

```
{
    "status": "ok", "query": "北京",
    "places": [
        {
            "name": "北京市", "location": {
                "lat": 39.9041999, "lng": 116.4073963},
            "formatted_address": "中国?京市"
        },
        {
            "name": "北京西站", "location": {
                "lat": 39.89491, "lng": 116.322056},
            "formatted_address": "中国?北京市西站"
        }
    ]
}
```

```
        "formatted_address": "中国 ?京市 丰台区 莲花池东路  
118号"},  
  
        {"name": "北京南站", "location":  
        {"lat": 39.865195, "lng": 116.378545},  
        "formatted_address": "中国 ?京市 丰台区 永外大街车站  
路12号"},  
  
        {"name": "北京站(地铁站)", "location":  
        {"lat": 39.904983, "lng": 116.427287},  
        "formatted_address": "中国 ?京市 东城区 2号线"}  
    ]  
}
```

那么下面就是对应的 `data class` :

```
data class PlaceResponse(val status: String, val places:  
List<Place>)  
  
data class Place(val name: String, val location: Location,  
@SerializedName("formatted_address") val address: String)  
  
data class Location(val lng: String, val lat: String)
```

由于JSON中一些字段的命名可能与Kotlin的命名规范不太一致，因此这里使用了 `@SerializedName` 注解的方式，来让JSON字段和Kotlin字段之间建立映射关系。

---

记录一下 `BindingView` 的写法，以下是郭神的文章：

[\(29条消息\) kotlin-android-extensions插件也被废弃了？扶我起来\\_guolin的博客-CSDN博客\\_kotlin-android-extensions废弃](#)

然后是Retrofit中Service的快速创建。如果不快速创建的话，正常写法是：

```
val retrofit = Retrofit.Builder()  
    .baseUrl("http://10.0.2.2/")  
    .addConverterFactory(GsonConverterFactory.create())  
  
    .build()
```

```
val appService = retrofit.create(AppService::class.java)
```

如果每用到一次和10.0.2.2的连接就要写一堆这些，烦都烦死了。所以我们需要简化一下。在逻辑层的网络包，也就是.logic.model下新建ServiceCreator单例类：

```
object ServiceCreator {  
  
    private const val BASE_URL = "http://10.0.2.2/"  
  
    private val retrofit = Retrofit.Builder()  
        .baseUrl(BASE_URL)  
  
        .addConverterFactory(GsonConverterFactory.create())  
        .build()  
  
    //这里fun后面的<T>表示泛型声明，表明参数中有泛型  
    fun <T> create(serviceClass: Class<T>): T =  
        retrofit.create(serviceClass)  
}
```

这样新建Service就只需要这样写：

```
val appService = ServiceCreator.create(AppService::class.java)
```

如果还是觉得麻烦，还可以再加一个函数：

```
inline fun <reified T> create(): T = create(T::class.java)
```

这样新建Service只需要这样写：

```
val appService = ServiceCreator.create<AppService>()
```

这里使用了Kotlin泛型实化。比如，如果定义一个函数：`fun getGenericType() = T::class.java`这里会产生语法错误。例如，假设我们创建了一个List<String>集合，虽然在编译时期只能向集合中添加字符串类型的元素，但是在运行时期JVM并不能知道它本来只打算包含哪种类型的元素，只能识别出来它是个List。所有基于JVM的语言，它们的泛型功能都是通过类型擦除机制来实现的，其中当然也包括了Kotlin。这种机制使得我

们不可能使用 `a is T` 或者 `T::class.java` 这样的语法，因为 `T` 的实际类型在运行的时候已经被擦除了。因此，在编译时的泛型无法被具体化成某个具体的类

但是，如果这样写：`inline fun <reified T> getGenericType() = T::class.java`

加上 `inline` 和 `reified`，就可以在编译时通过了。

另外，郭神的《第一行代码》中的500页，第10.6讲也是在说这个事情。

---

在使用 `LiveData` 去保存登录系统的用户名和密码的时候，出现了这样的问题。我们的 `LoginViewModel` 是这样的：

```
class LoginViewModel(uname: String, passwd: String): ViewModel() {

    val username: LiveData<String>
        get() = _username
    private val _username = MutableLiveData<String>()

    val password: LiveData<String>
        get() = _password
    private val _password = MutableLiveData<String>()

    init{
        _username.value = uname
        _password.value = passwd
    }

    fun setUsername(uname: String){
        _username.value = uname
    }

    fun setPassword(passwd: String){
        _password.value = passwd
    }

}
```

这种写法是为了保存好数据的封装性。下划线开头的是真正的变量，而对应

的没下划线的就是提供给外部访问的。而因为它们是LiveData类型，不可变。所以从外部是无法访问的，只能通过set方法来改变值。然后是在对应的LoginActivity中使用Retrofit来发起登陆请求。而这个请求的参数毫无疑问就是用户和密码。那么它们从哪儿来呢？如果不适用ViewModel + LiveData的话，就是定义在程序内部的变量，并从EditText处拿到值。也就是这样：

```
username = bindingLogin.accountEdit.text.toString()
password = bindingLogin.passwordEdit.text.toString()
```

但是如今我们使用了LiveData，自然这些变量全要作为LiveData存到ViewModel中。那么在从EditText那里拿值的操作就变成了这样：

```
loginViewMode.setUsername(bindingLogin.accountEdit.text.toString())
loginViewMode.setPassword(bindingLogin.passwordEdit.text.toString())
```

接下来就是真正使用Retrofit的Service接口去发请求了。这里遇到的问题是：由于数据被放在了LiveData中，而它的value字段有可能为空，但是kotlin中的变量类型默认不为空。所以我们只能采取下列写法：

```
if(loginViewMode.username.value != null &&
loginViewMode.password.value != null){
    val username = loginViewMode.username.value
    val password = loginViewMode.password.value
    //Smart cast to 'String' is impossible, because
    'loginViewMode.username.value' is a complex expression
    loginResultService.getLoginResult(username!!, password!!)
    .enqueue(object: Callback<LoginResult> {
        override fun onResponse(
            call: Call<LoginResult>,
            response: Response<LoginResult>
        ) {
            Log.d("SpreadShop", "on Response")
            val loginResult = response.body()
            if(loginResult != null){
                Log.d("SpreadShop",
                    "loginResult.message: ${loginResult.message}")
            }
        }
    })
}
```

```
        if(loginResult.success){
            Log.d("SpreadShop", "Login Success")
        }else{
            Log.d("SpreadShop", "Login Fail")
        }
    }else{
        Log.d("SpreadShop", "LoginResult is Null")
    }
}

override fun onFailure(call: Call<LoginResult>, t: Throwable) {
    Log.d("SpreadShop", "on Failure")
    t.printStackTrace()
}
}
}
```

其实改变只有第一行if语句中的判断。因为我们在这里处理成了必须非空，然后才能在其中断言这两个变量是非空的。另外真正传参的时候还是要加上`!!`非空断言。

好吧，我上面的写法很傻，下面给出整个的登录按钮的注册监听：

```
bindingLogin.loginBtn.setOnClickListener {

    loginViewModel.setUsername(bindingLogin.accountEdit.text.toString())
    loginViewModel.setPassword(bindingLogin.passwordEdit.text.toString())

    val username = loginViewModel.username.value
    val password = loginViewModel.password.value

    val loginResultService =
ServiceCreator.create<LoginResultService>()

    if(username != "" && password != ""){
        Log.d("SpreadShopTest", "username: $username")
        Log.d("SpreadShopTest", "password: $password")
    }
}

//Smart cast to 'String' is impossible, because
```

```
'loginViewMode.username.value' is a complex expression
        loginResultService.getLoginResult(username!!,
password!!).enqueue(object: Callback<LoginResult> {
    override fun onResponse(
        call: Call<LoginResult>,
        response: Response<LoginResult>
    ) {
        Log.d("SpreadShopTest", "on
Response")
        val loginResult = response.body()
        if(loginResult != null){
            Log.d("SpreadShopTest",
"loginResult.message: ${loginResult.message}")
            if(loginResult.success){
                Log.d("SpreadShopTest", "Login Success")
            }else{
                Log.d("SpreadShopTest", "Login Fail")
            }
        }else{
            Log.d("SpreadShopTest",
"LoginResult is Null")
        }
    }

    override fun onFailure(call:
Call<LoginResult>, t: Throwable) {
        Log.d("SpreadShopTest", "on
Failure")
        t.printStackTrace()
    }
}
}else{
    Log.d("SpreadShopTest", "usernameEmpty:
${username == ""}")
    Log.d("SpreadShopTest", "passwordEmpty:
${password == ""}")
}
}

// end bindingLogin.loginBtn.setOnClickListener
```

由于我们并不需要将username和password在屏幕上显示出来，所以根本不需要livedata的observe方法。所以在赋值完之后，直接使用就好。还有一点，字符串为空并不是`== null`。这里的空可不是c语言里的空指针，它是有实际内存的，只不过值为空。

## 2022-10-22

今天将这个项目彻底换成了MVVM架构，全部采用livedata，并且不使用协程去实现。原因就是我想彻底搞清楚MVVM架构的整体思路，而协程虽然能学到很多高级的Kotlin用法，但是对于自己思维的限制实在是太大了。如果用协程的话，我目前只知道抄书，所以想要打破一下这个局限。

首先从登陆这个功能开始。首先的首先，是它的请求数据对象：

```
data class LoginResult(val username: String, val passwd: String,  
val message: String, val success: Boolean)
```

确实，这里考虑到了返回的状态，也就是`success`成员。但是在传递Goods的时候当时并没有考虑到这一点，所以我们最后更改了那部分代码。这里`success`就是用来确定我的登陆是否成功，是密码错误还是账号不存在。

然后，就是登陆的Retrofit协议接口了：

```
interface LoginResultService {  
    @GET("login")  
    fun getLoginResult(@Query("username") username: String,  
                      @Query("password") password: String): Call<LoginResult>  
}
```

这里使用了`@Query`注解，这种注解是专门用来匹配下面的http请求的：

```
http://localhost:8080/login?username=spreadzhao&password=1234
```

我们向`getLoginResult`函数中传递参数`spreadzhao`和`1234`，就能得到上面的http请求地址。

然后，是登陆的网络层接口，这里的操作是关键，也是和书中不一样的地方：

```
private val loginService =
```

```
ServiceCreator.create<LoginResultService>()

fun getLoginResult(username: String, password: String) =
    loginService.getLoginResult(username, password)
```

在书中的MVVM架构里，网络层的代码非常复杂，又是高阶扩展函数又是挂起函数，其实都是为了使用协程而设计的。而我们不想去研究协程，那么改怎么办呢？我最一开始的代码是这样的：

```
private val goodsService = ServiceCreator.create<GoodsService>()

// 这里为什么能拿到cmd这个变量？是因为lambda表达式？
fun getGoods(cmd: String) =
    goodsService.getGoods(cmd).enqueue(object: Callback<List<Goods>>{
        override fun onResponse(call: Call<List<Goods>>, response:
        Response<List<Goods>>) {
            val list = response.body()
            if(list != null){
                Log.d("SpreadShopTest", "[\$cmd]list is not null")
            }else{
                Log.d("SpreadShopTest", "[\$cmd]list is null")
            }
        }

        override fun onFailure(call: Call<List<Goods>>, t: Throwable)
        {
            t.printStackTrace()
            Log.d("SpreadShopTest", "[showall]on failure")
        }
    })
```

由于我的Login代码被删掉了，所以这里给出当时写的获取商品的代码，其实是一样的。

可以看到，我当时的设想是在这里直接得到相应的对象，并做相应的处理。这样经过仓库层再一包装，确实可以做到在Activity中一句话发起请求：

```
Repository.getLoginResult(username!!, password!!)
```

但是这种写法在我仔细想了想后，终于发现了问题：LiveData哪儿去了？首先，我们的目的是获取一个LoginResult类型的数据。但是如果按照我上面的写法去做的话，**那么这个数据在到达网络层就已经结束了！**因为我在网络层就已经处理了响应的代码。而我们的目的是将这个数据最终传递到ViewModel层，因为这才是MVVM架构的意义(拆分UI层减轻负担)。

另一个问题是，我在确定登陆成功之后，要打开MainAcitivity，而这个操作一定是由LoginActivity来做的。但是按照我这种写法，这个操作居然交给了网络层去做？显然这是不正确的。

由于以上两点，我们得出结论：

- 返回的对象必须是LiveData；
- 返回的对象必须传递到ViewModel层，然后由LoginActivity去观察这个LiveData的变化，观察到变化之后再去处理响应操作(打开MainActivity或者提示登录失败)。

由此，我们才有了如今的这种写法，这也是我的**灵光一现**。而就是这个灵光一现，让我对于Retrofit的理解又深了几分。首先，是这个enqueue函数，到底是谁执行的？答案是Call<T>类型。那么既然如此，**我只需要将这个Call<T>对象的返回值层层传递到ViewModel层，然后将它包裹成LiveData**，上面的两个要求不就都实现了吗？！因此，我才有了这样的代码：

```
private val loginService =  
    ServiceCreator.create<LoginResultService>()  
  
fun getLoginResult(username: String, password: String) =  
    loginService.getLoginResult(username, password)
```

可以看到，我只是调用了loginService.getLoginResult()函数去得到返回值然后再返回给上层。那么这个返回值的类型毫无疑问就是Call<LoginResult>类型。另外，第一行这种写法在昨天的日记中也已经提到。

有了网络层，接下来就该是仓库层了。而这里的代码也很简单，不用像书上

还要用 `LiveData` 函数(书中使用这个函数也是因为协程)：

```
fun getLoginResult(username: String, password: String) =  
    SpreadShopNetwork.getLoginResult(username, password)
```

依然是将这个返回值继续接力传递一下就行了，不需要任何其他操作。因为仓库层主要的功能是从网络获取数据和从本地获取数据。

接下来到了比较关键的 `ViewModel` 层，这里的代码就得说道说道了：

```
class LoginViewModel: ViewModel() {  
  
    val username: LiveData<String>  
        get() = _username  
    private val _username = MutableLiveData<String>()  
  
    val password: LiveData<String>  
        get() = _password  
    private val _password = MutableLiveData<String>()  
  
    val loginResult: LiveData<Call<LoginResult>>  
        get() = _loginResult  
    private val _loginResult = MutableLiveData<Call<LoginResult>>()  
(  
  
    fun setUsername(uname: String){  
        _username.value = uname  
    }  
  
    fun setPassword(passwd: String){  
        _password.value = passwd  
    }  
  
    fun getLoginResult(username: String, password: String){  
        _loginResult.value = Repository.getLoginResult(username,  
password)  
    }  
}  
  
val loginResult: LiveData<Call<LoginResult>>这句话之前的代码我们在
```

之前已经解释过了，接下来我们主要研究研究这段代码：

```
val loginResult: LiveData<Call<LoginResult>>
    get() = _loginResult
private val _loginResult = MutableLiveData<Call<LoginResult>>()
```

结构和前面其实一模一样。唯一要强调的就是LiveData中包含的类型：

Call<LoginResult>。这个类型其实就是在通过网络层，仓库层一次次传递回来的值。其实从根本上讲，它就是这个函数的返回值：

```
@GET("login")
fun getLoginResult(@Query("username") username: String,
@Query("password") password: String): Call<LoginResult>
```

因此我们要做的，就是将从仓库层拿来的Call<LoginResult>实体包裹进这个LiveData中，然后在LoginActivity里观察它就好了，而最终在ViewModel层的封装就是这样的：

```
fun getLoginResult(username: String, password: String){
    _loginResult.value = Repository.getLoginResult(username,
password)
}
```

因此当\_loginResult.value发生变化时，我们观察的loginResult就会跟着发生改变，因此这样就会触发LoginActivity中的Observer函数了。接下来我们就给出LoginActivity中的代码。首先是点击登录按钮之后的事件：

```
bindingLogin.loginBtn.setOnClickListener {
    loginViewMode.setUsername(bindingLogin.accountEdit.text.to
String())
    loginViewMode.setPassword(bindingLogin.passwordEdit.text.to
String())
    val username = loginViewMode.username.value
    val password = loginViewMode.password.value
    if(username != "" && password != ""){
        Log.d("SpreadShopTest", "username: $username")
```

```
        Log.d("SpreadShopTest", "password: $password")
        loginViewModel.getLoginResult(username!!, password!!)

    }else{
        Log.d("SpreadShopTest", "usernameEmpty:
${username == ""}")
        Log.d("SpreadShopTest", "passwordEmpty:
${password == ""}")
    }

}// end bindingLogin.loginBtn.setOnClickListener
```

和我们[之前的代码](#)一对比，简直不要再简洁！首先，由于使用了网络层，而`loginResultService`的[实例就在其中](#)，并且已经被包装到了`ViewModel`层，所以我们在这里只需要调用`loginViewModel`的`getLoginResult`方法就自动会一层层走下去。而这样做的最终结果就是，`loginViewModel`中的`_loginResult.value`变成了`Repository.getLoginResult`的返回值。而接下来我们要做的另一件是，就是在`LoginActivity`中去观察它的变化了：

```
loginViewModel.loginResult.observe(this){
    it.enqueue(object: Callback<LoginResult>{
        override fun onResponse(
            call: Call<LoginResult>,
            response: Response<LoginResult>
        ) {
            Log.d("SpreadShopTest", "on Response")
            val loginResult = response.body()
            if(loginResult != null){
                Log.d("SpreadShopTest", "loginResult.message:
${loginResult.message}")
                if(loginResult.success){
                    Log.d("SpreadShopTest", "Login Success")
                    val intent = Intent(this@LoginActivity,
MainActivity::class.java)
                    startActivity(intent)
                }else{
                    Log.d("SpreadShopTest", "Login Fail")
                }
            }else{
                Log.d("SpreadShopTest", "LoginResult is Null")
            }
        }
    })
}
```

```
        }
    }

    override fun onFailure(call: Call<LoginResult>, t: Throwable) {
        Log.d("SpreadShopTest", "on Failure")
        t.printStackTrace()
    }
}

}
```

到了这一步，我们也算是终于完成了[之前的两个要求](#)：

- LoginResult作为LiveData封装到了ViewModel中
- 由LoginActivity去处理登陆的操作，而不是由网络层

另外补充一点，我们并没有使用[switchMap\(\)](#)函数，是因为这个LiveData对象并不是来自外部，是我们ViewModel本身的，只不过是对它的[value](#)字段进行赋值而已。**更加详细的描述在郭神的《第一行代码》中的13.4.2节。**

---

之前也[提到过](#)，我更改了Goods的请求对象，原本我们返回的对象是[Call<List<Goods>>](#)，但是我们没有考虑过查询失败的情况。因此我们重新确定了数据模型，在上面进一步封装。下面给出数据模型的代码和网络接口的代码即可，剩下的修改就不多赘述了：

```
data class GoodsResponse(val success: Boolean, val goods: List<Goods>)

data class Goods(val goods_id: Int, val goods_name: String, val goods_category: String, val goods_storage: Int, val goods_price: Int)
```

```
@GET("searchgoods")
fun getGoods(@Query("cmd") cmd: String): Call<GoodsResponse>
```

在修改之后，执行的时候不管怎么执行都是报错，而我很确定已经把所有的[List<Goods>](#)都改成了[GoodsResponse](#)。后来鉴定为编译器抽风，只需要Rebuild一下就好了。

接下来，是对于MainActivity的Material Design设计。首先是主题的更换，在`res/values/themes.xml`文件和`res/values-night/themes.xml`文件中，将style的parent更改成

`Theme.MaterialComponents[.Light].NoActionBar`，这样就能够将主题更换成Material Design了，如果不更换的话，会出现崩溃的现象。另外，我们既然改成了`NoActionBar`，就需要我们自己去实现ActionBar了。在`activity_main.xml`中，就是如下的代码：

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

<!--
    CoordinatorLayout是加强版的FrameLayout，它专为MaterialDesign设计，能够监听其中控件的变化。-->

<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="@color/purple_200"

        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
        />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

```
<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:menu="@menu/nav_menu"
    app:headerLayout="@layout/nav_header"
    />

</androidx.drawerlayout.widget.DrawerLayout>
```

我们将 `ToolBar` 包裹在了 `CoordinatorLayout` 中，这两个都是 Material Design 的组件。另外下面还有一个 `Navigation View`，这也是我们的侧滑菜单的主要组件。

需要强调的有两点。一是我们最外层的组件：`DrawerLayout`，它能实现窗口想抽屉一样拉开。在所有的子组件中都会有一个 `android:layout_gravity` 属性，这个就是决定当前组件从哪个地方拉出来的选项。`start` 表示根据语言判断。而这个 `DrawerLayout` 和 `Navigation View` 组合起来使用就能够实现侧滑菜单；第二点就是 `Navigation View` 中的 `app:menu` 属性和 `app:headerLayout` 属性。每一个 `Navigation View` 都由一个标题和一个菜单组成。而这两个文件就是我们接下来要介绍的。

首先是 `nav_menu`，它在 `res/menu/nav_menu.xml`。menu 文件夹里所有的视图都是菜单。`toolbar` 就是上面工具栏专用的菜单；`nav_menu` 就是 `Navigation View` 专用的菜单。

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<!--
    group 表示这些项被分成一个组
    checkableBehavior="single" 表示同时只能选中一个
-->
<group android:checkableBehavior="single">
    <item
        android:id="@+id/nav_mybag"
        android:title="My Bag"
    />
    <item
        android:id="@+id/nav_myprofile"
        android:title="My Profile"
    />
</group>
<group android:checkableBehavior="single">
    <item
        android:id="@+id/nav_myorder"
        android:title="My Order"
    />
    <item
        android:id="@+id/nav_myfavor"
        android:title="My Favor"
    />
</group>
<group android:checkableBehavior="single">
    <item
        android:id="@+id/nav_mysetting"
        android:title="My Setting"
    />
</group>
</menu>
```

```
        android:id="@+id/nav_order"
        android:title="My Order"
    />

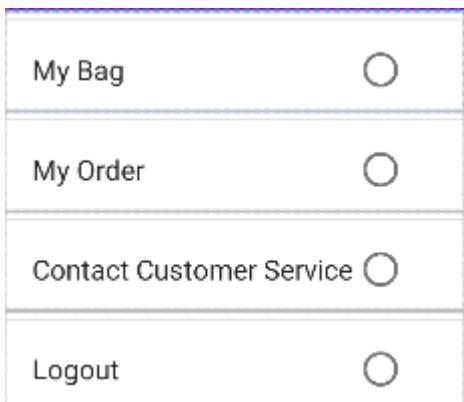
    <item
        android:id="@+id/nav_contact"
        android:title="Contact Customer Service"
    />

    <item
        android:id="@+id/nav_logout"
        android:title="Logout"
    />

</group>

</menu>
```

这样的话，侧滑菜单的选项就是这样的：



然后就是headerLayout了，它位于 `res/layout/nav_header.xml`：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:padding="10dp"
    android:background="@color/material_dynamic_primary40"
    android:layout_height="200dp">

    <de.hdodenhof.circleimageview.CircleImageView
        android:id="@+id/user_icon"
```

```
        android:layout_width="70dp"
        android:layout_height="70dp"
        android:src="@drawable/nav_user"
        android:layout_centerInParent="true"
    />

    <TextView
        android:id="@+id/app_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="Spread Shop"
        android:textColor="@color/black"
        android:textSize="14sp"
    />

    <TextView
        android:id="@+id/user_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@+id/app_name"
        android:textColor="@color/material_dynamic_neutral60"
        android:text="user name: null"
    />

</RelativeLayout>
```

需要注意的是，这里的 `de.hdodenhof.circleimageview.CircleImageView` 是我们引入的第三方库，专门用来把图片切成圆形。不妨就在这里列出项目所有的依赖吧：

```
dependencies {

    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.5.1'
    implementation 'com.google.android.material:material:1.5.0'
    implementation
    'androidx.constraintlayout:constraintlayout:2.1.4'
    implementation 'com.squareup.retrofit2:retrofit:2.6.1'
    implementation 'com.squareup.retrofit2:converter-gson:2.6.1'
    implementation 'com.google.android.material:material:1.1.0'
```

```
        implementation 'de.hdodenhof:circleimageview:3.0.1'
        testImplementation 'junit:junit:4.13.2'
        androidTestImplementation 'androidx.test.ext:junit:1.1.3'
        androidTestImplementation 'androidx.test.espresso:espresso-
core:3.4.0'
    }
```

接下来还有一点，就是我们主界面的[toolbar](#)的菜单。既然是菜单文件，肯定是位于[res/menu](#)文件夹下了，我们就叫它 `toolbar.xml` 吧：

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

<!--
    menu文件夹里的所有试图都是菜单
    toolbar就是上面工具栏专用的菜单
    而nav_menu就是Navigation View专用的菜单
-->

<!--
    always: 永远显示再Toolbar中，空间不够不显示
    ifRoom: 屏幕够就显示，不够显示再菜单中
    never: 永远显示在菜单
-->

<item
    android:id="@+id/backup"
    android:icon="@drawable/ic_backup"
    android:title="Backup"
    app:showAsAction="always"
    />

<item
    android:id="@+id/delete"
    android:icon="@drawable/ic_delete"
    android:title="Delete"
    app:showAsAction="ifRoom"
    />

<item
    android:id="@+id/settings"
    android:icon="@drawable/ic_settings"
    android:title="Settings"
    app:showAsAction="never"
    />
```

```
        android:icon="@drawable/ic_settings"
        android:title="Settings"
        app:showAsAction="never"
    />
</menu>
```

非常好理解，就不用过多赘述了。有了所有的前端代码，接下来就是在MainActivity中将它们显示出来并注册点击事件。我们还是先从NavigationView开始，在MainActivity中需要想给按钮注册监听器一样给NavigationView里的菜单子项注册监听事件：

```
bindingMain.navView.setNavigationItemSelectedListener {
    when(it.itemId){
        R.id.nav_mybag -> Log.d("SpreadShopTest", "You clicked
mybag")
        R.id.nav_order -> Log.d("SpreadShopTest", "You clicked
myorder")
        R.id.nav_contact -> Log.d("SpreadShopTest", "You clicked
Contact")
        R.id.nav_logout -> Log.d("SpreadShopTest", "You clicked
logout")
    }
    bindingMain.drawerLayout.closeDrawers()
    true
}
```

不管点击了任何按钮，最终都要调用`closeDrawers()`方法关闭所有的侧滑菜单。

好了，NavigationView已经做完了！但是更重要的是接下来的Toolbar。因为只有有了Toolbar我们的程序才看起来会完整一些。首先，由于我们在[前面](#)已经删掉了原本的ActionBar，所以我们需要设置新的ActionBar为我们自己定义的Toolbar：

```
setSupportActionBar(bindingMain.toolbar)
```

接下来，是展示左上角的侧滑菜单按钮。首先需要调用`getSupportActionBar`方法来得到这个ActionBar的实例(其实就是一个Toolbar)，然后将home图标显示出来并设置上我们自己的icon：

```
/*
```

```
supportActionBar?.let{
    it.setDisplayHomeAsUpEnabled(true)
    it.setHomeAsUpIndicator(R.drawable.ic_menu)
}
/**/

//let和apply都可以
supportActionBar?.apply {
    setDisplayHomeAsUpEnabled(true)
    setHomeAsUpIndicator(R.drawable.ic_menu)
}
```

然后，我们还要做两件事：给Toolbar的menu菜单的子项注册监听事件；将Toolbar的menu菜单显示出来。这两件事要分别重写两个函数，代码不多，直接展示了：

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when(item.itemId){
        android.R.id.home ->
        bindingMain.drawerLayout.openDrawer(GravityCompat.START)
        R.id.backup -> Log.d("SpreadShopTest", "you clicked
backup")
        R.id.delete -> Log.d("SpreadShopTest", "you clicked
delete")
        R.id.settings -> Log.d("SpreadShopTest", "you clicked
settings")
    }
    return true
}

override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.toolbar, menu)
    return true
}
```

`openDrawer()`有很多重载函数，可以自己到源码中看一看。好了，今天的所有进展就到这里了。

# 2022-10-23

一个挺吓人的小问题，我修改了MainActivity的名字，然后觉得不妥又改回

来了。但是在运行的时候报了一大堆错。好在下面这个网址和我的情况一样，看起来并不是改名字的问题，而是我引入了下面的依赖而引用了 androidx 库而导致的冲突：

```
implementation 'androidx.recyclerview:recyclerview:1.0.0'
```

所以根据这个网站：

[编译报错Duplicate class](#)

[android.support.v4.app.INotificationSideChannel found in modules  
classes - 北海南竹 - 博客园 \(cnblogs.com\)](#)

在 `gradle.properties` 里添加这个选项就好了：

```
android.enableJetifier=true
```

另外这段代码的解释在下面这个网站：

[\(29条消息\) \[Android\]\[踩坑\]gradle中配置android.useAndroidX与  
android.enableJetifier使应用对support库的依赖自动转换为androidx的依  
赖\\_Ryan ZHENG的博客-CSDN博客\\_enablejetifier](#)

根据 [Android官网](#) 介绍：

`android.useAndroidX=true` 表示“Android插件会使用对应的AndroidX库，而非Support库”；未设置时默认为 `false`；

`android.enableJetifier=true` 表示Android插件会通过重写其二进制文件来自动迁移现有的第三方库，以使用AndroidX依赖项；未设置时默认为 `false`；

当然，这两个属性对Android Studio版本是有要求的： **3.2**

同时，由于Android Studio的版本对gradle、gradle plugin的最低版本也有限制，因此可以认为对gradle及其插件也是有要求的：

- 将 Android studio 升级到 3.2 及以上； (我使用的3.6.3)
- Gradle 插件版本改为 4.6 及以上； (我使用的4.6)
- `compileSdkVersion` 版本升级到 28 及以上； (我使用的30)
- `buildToolsVersion` 版本改为 28.0.2 及以上； (我使用的30.0.3)

---

今天我们将 `MainActivity` 进行了大改，在 `DrawerLayout` 的基础上添加了很

多东西，从 `activity_main.xml` 开始看，对比之前的 `CoordinatorLayout`，添加了很多东西：

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
```

```
<!--
```

CoordinatorLayout是加强版的FrameLayout，它专为MaterialDesign设计，

能够监听其中控件的变化。-->

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
<!--
```

AppBarLayout可以让RecyclerView不遮挡Toolbar，  
使用app:layout\_behavior来指定

```
-->
```

```
<com.google.android.material.appbar.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
```

```
<androidx.appcompat.widget.Toolbar
```

```
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="@color/purple_200"
```

```
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
```

```
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
    />
```

```
</com.google.android.material.appbar.AppBarLayout>
```

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/category_recycler"
    android:layout_width="match_parent"
    android:layout_height="200dp"

    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    />

<!--
    想让谁实现下拉刷新功能，就把谁放到SwipeRefreshLayout里，
    记得添加依赖
    recyclerView里面的layout_behavior搬到外面
-->

<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/swipe_refresh"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="250dp"
    app:layout_anchor="@+id/category_recycler"
    app:layout_anchorGravity="bottom"

    >

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/goods_recycler"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />

    </androidx.swiperefreshlayout.widget.SwipeRefreshLayout>

</androidx.coordinatorlayout.widget.CoordinatorLayout>

<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:menu="@menu/nav_menu"
    app:headerLayout="@layout/nav_header"
    />
```

```
</androidx.drawerlayout.widget.DrawerLayout>
```

多出来的东西全都在CoordinatorLayout中，分别是AppBarLayout、RecyclerView和SwipeRefreshLayout，并且SwipeRefreshLayout中又嵌入了一个RecyclerView。需要注意的是SwipeRefreshLayout中的 `android:layout_marginTop="250dp"` 属性，它在上方流出了一个空白，这样它才不会遮挡上面的 `category_recycler`。 **肯定有更好的解决方法，但是我目前没有找到。**

最重要的其实是这两个RecyclerView的东西，我们先从Category开始说起。这是为了加载返回的所有商品的属性的列表，也就是商城里常见的“商品分类”。这个RecyclerView中的项是 `category_item.xml`：

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        >

        <LinearLayout
            android:orientation="horizontal"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            >

            <ImageView
                android:id="@+id/category_image"
                android:layout_width="300dp"
                android:layout_height="100dp"
                android:scaleType="fitCenter"
                />

            <TextView
                android:id="@+id/category_name"
                >
            </TextView>
        </LinearLayout>
    </LinearLayout>
</MaterialCardView>
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="test"
        android:textColor="@color/black"
        android:textSize="16sp"
    />

</LinearLayout>

</LinearLayout>
</com.google.android.material.card.MaterialCardView>
```

非常简单，没什么好说的。唯一的特点是图片的  
`android:scaleType="fitCenter"`这个属性，它是让图片从中间开始进行合适的缩放。

接下来自然是创建对应的Adapter来加载它了。创建  
`ui/goods/CategoryAdapter`类：

```
class CategoryAdapter(val context: Context, val categoryList:
List<Category>): RecyclerView.Adapter<CategoryAdapter.ViewHolder>
() {

    inner class ViewHolder(view: View):
    RecyclerView.ViewHolder(view){
        val categoryName: TextView =
        view.findViewById(R.id.category_name)
        val categoryImage: ImageView =
        view.findViewById(R.id.category_image)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): ViewHolder {
        val view =
        LayoutInflater.from(context).inflate(R.layout.category_item,
        parent, false)
        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position:
```

```

Int) {
    val category = categoryList[position]
    holder.categoryName.text = category.category
    Log.d("SpreadShopTest", "category name: ${category.category}")
    val id: Int
    when(category.category){
        "手机" -> {
            Log.d("SpreadShopTest", "category id: 手机")
            id = R.drawable.ic_phone
        }
        "衣服" -> {
            Log.d("SpreadShopTest", "category id: 衣服")
            id = R.drawable.ic_cloth
        }
        "裤子" -> {
            Log.d("SpreadShopTest", "category id: 裤子")
            id = R.drawable.ic_trousers
        }
        else -> {
            Log.d("SpreadShopTest", "category id: else")
            id = R.drawable.test_maotai
        }
    }
}

Log.d("SpreadShopTest", "val id: $id")

Glide.with(context).load(id).into(holder.categoryImage)
}

override fun getItemCount() = categoryList.size
}

```

这里我们使用了Glide插件去加载图片，因此我再给一遍当前项目中的所有依赖：

```

dependencies {
    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.5.1'

    implementation 'com.google.android.material:material:1.5.0'
}

```

```
implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
implementation 'com.squareup.retrofit2:retrofit:2.6.1'
implementation 'com.squareup.retrofit2:converter-gson:2.6.1'
implementation 'com.google.android.material:material:1.1.0'
implementation 'de.hdodenhof:circleimageview:3.0.1'
implementation 'androidx.recyclerview:recyclerview:1.0.0'
implementation
'androidx.swiperefreshlayout:swiperefreshlayout:1.0.0'
implementation 'com.github.bumptech.glide:glide:4.9.0'
testImplementation 'junit:junit:4.13.2'
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-
core:3.4.0'
}
```

这是之前的依赖。

然后就是在MainActivity的categoryLiveData中去观察变量，并在相应处理中设置Adapter，这样就能在界面上显示拿到的结果了：

```
mainViewModel.categoryLiveData.observe(this){
    it.enqueue(object: Callback<CategoryResponse>{
        override fun onResponse(
            call: Call<CategoryResponse>,
            response: Response<CategoryResponse>
        ) {
            val categoryResponse = response.body()
            if(categoryResponse != null){
                if(categoryResponse.success){
                    Log.d("SpreadShopTest", "category success")
                    val list = categoryResponse.categories

                    // category recycler
                    val layoutManager =
                        GridLayoutManager(this@MainActivity, 1)
                    bindingMain.categoryRecycler.layoutManager =
                        layoutManager
                    val adapter =
                        CategoryAdapter(this@MainActivity, list)
                    bindingMain.categoryRecycler.adapter =

```

```
adapter

    for(category in list){
        Log.d("SpreadShopTest", "category:
$category")
    }
}else{
    Log.d("SpreadShopTest", "Category fail")
}
}else{
    Log.d("SpreadShopTest", "category is null")
}
}

override fun onFailure(call: Call<CategoryResponse>, t:
Throwable) {
    t.printStackTrace()
    Log.d("SpreadShopTest", "category on failure")
}
}
} // end mainViewModel.categoryLiveData.observe
```

对于商品的处理和Category其实一模一样，**唯一的区别是显示上的一些细节**。所以我还是按照`xml -> Adapter -> MainActivity`的顺序直接给出相应代码：

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!--
        一张Material卡片，之后这一张张卡片都会添加到
        recyclerView当中

        centerCrop: 让图片保持原有比例填充满ImageView,
        并将超出屏幕的部分裁剪掉-->
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        >

        <ImageView
            android:id="@+id/goods_image"
            android:layout_width="match_parent"
            android:layout_height="100dp"
            android:scaleType="centerCrop"
        />

        <TextView
            android:id="@+id/goods_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:layout_margin="5dp"
            android:text="test"
            android:textSize="16sp"
        />

    </LinearLayout>
</com.google.android.material.card.MaterialCardView>
```

```
class GoodsAdapter(val context: Context, val goodsList:
List<Goods>): RecyclerView.Adapter<GoodsAdapter.ViewHolder>(){

    inner class ViewHolder(view: View):
    RecyclerView.ViewHolder(view){
        val goodsImage: ImageView =
        view.findViewById(R.id.goods_image)
        val goodsName: TextView =
        view.findViewById(R.id.goods_name)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): ViewHolder {
        val view =
        LayoutInflater.from(context).inflate(R.layout.goods_item, parent,
        false)
        return ViewHolder(view)
    }
}
```

```
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val goods = goodsList[position]
        holder.goodsName.text = goods.goods_name
        // 所有商品暂时都是茅台的图片

        Glide.with(context).load(R.drawable.test_maotai).into(holder.goodsImage)
    }

    override fun getItemCount() = goodsList.size
}
```

```
mainViewModel.goodsLiveData.observe(this){
    it.enqueue(object : Callback<GoodsResponse>{
        override fun onResponse(call: Call<GoodsResponse>, response: Response<GoodsResponse>) {
            val goodsResponse = response.body()
            if(goodsResponse != null){
                Log.d("SpreadShopTest", "goodsResponse is not null")
                if(goodsResponse.success){
                    Log.d("SpreadShopTest", "goodsResponse success!")
                    val list = goodsResponse.goods

                    // goods recycler
                    val layoutManager =
                        GridLayoutManager(this@MainActivity, 2)
                    bindingMain.goodsRecycler.layoutManager =
                        layoutManager
                    val adapter = GoodsAdapter(this@MainActivity,
                        list)
                    bindingMain.goodsRecycler.adapter = adapter

                    for(goods in list){
                        Log.d("SpreadShopTest", "goods: ${goods.goods_name}")
                    }
                }else{

```

```
        Log.d("SpreadShopTest", "goodsResponse
fail!")
    }
}else{
    Log.d("SpreadShopTest", "goodsResponse is null")
}
}

override fun onFailure(call: Call<GoodsResponse>, t:
Throwable) {
    t.printStackTrace()
    Log.d("SpreadShopTest", "goodsResponse on failure")
}
})
} // end mainViewModel.goodsLiveData.observe
```

## 2022-10-25

这次主要是做了下拉刷新的逻辑端操作，并且已经测试成功。需要注意的是，本次的更新改动比较大，将原来的Retrofit处理响应的操作整个更换了。

之前我们的思路是：如果接到Retrofit的响应数据，就在响应操作中设置RecyclerView的adapter之类的。但是，这种思路是**完全错误的！**首先，我们每发一次请求，都会调一次这个函数，那么这个函数每次都会新建一个adapter，而RecyclerView通常都是一个adapter用到底；其次，我们将adapter在这里设置成局部变量，在外部又如何能够通知数据发生了改变？

因此，我们首先需要将Adapter移到响应处理的外面，也就是直接包含在Activity的onCreate方法里：

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // 打开MainActivity之后立刻发起请求获取所有的种类
    mainViewModel.getAllCategory(Command.GET_ALL_CATEGORY)
    // 打开之后获取一次推荐商品
    mainViewModel.getGoods(Command.GET_RECOMMAND)
```

```
// 设置goodsRecycler的adapter
// 由于ArrayList的构造函数，所以goodsList不为空
val goodsLayoutManager = GridLayoutManager(this, 2)
bindingMain.goodsRecycler.layoutManager =
goodsLayoutManager
    val goodsAdapter = GoodsAdapter(this,
mainViewModel.goodsList)
    bindingMain.goodsRecycler.adapter = goodsAdapter

...
}

// end onCreate
```

可以看到，我们在Activity创建的时候，就将goodsRecycler的LayoutManager和Adapter设置好。需要注意的是adapter中的第二个参数：`mainViewModel.goodsList`。这个是在MainViewModel中定义的一个`ArrayList`，专门用来存放最终获取到的结果。那么可想而知，在相应处理函数中我们就需要更新这个`ArrayList`了：

```
mainViewModel.goodsLiveData.observe(this){
    it.enqueue(object : Callback<GoodsResponse>{
        override fun onResponse(call:
Call<GoodsResponse>, response: Response<GoodsResponse>) {
            val goodsResponse = response.body()
            if(goodsResponse != null){
                Log.d("SpreadShopTest",
"goodsResponse is not null")
                if(goodsResponse.success){
                    Log.d("SpreadShopTest",
"goodsResponse success!")
                }
            }
        }
    })
    val list =
    goodsResponse.goods

    mainViewModel.goodsList.clear()

    mainViewModel.goodsList.addAll(list)

    mainViewModel.isGotLiveData.value = true
} else{
```

```
        Log.d("SpreadShopTest",
"goodsResponse fail!")
    }
}else{
    Log.d("SpreadShopTest",
"goodsResponse is null")
}
}

override fun onFailure(call: Call<GoodsResponse>,
t: Throwable) {
    t.printStackTrace()
    mainViewModel.goodsList.clear()
    mainViewModel.isGotLiveData.value = false
    Log.d("SpreadShopTest", "goodsResponse on
failure")
}
}
// end mainViewModel.goodsLiveData.observe
```

可以看到，如果我们成功得到了返回的`Callback<List<Goods>>`，我们就执行如下代码：

```
val list = goodsResponse.goods
mainViewModel.goodsList.clear()
mainViewModel.goodsList.addAll(list)
mainViewModel.isGotLiveData.value = true
```

首先将`MainViewModel`的`goodsList`清空，然后将我们得到的新数据全部放进去，最后设置其中的标志位为`true`；而如果我们没有拿回响应数据，就是这样的处理：

```
t.printStackTrace()
mainViewModel.goodsList.clear()
mainViewModel.isGotLiveData.value = false
Log.d("SpreadShopTest", "goodsResponse on failure")
```

只是清空列表，然后将标志位设置为`false`。这两个标志位的设置非常重  
要，它会触发接下来的一个操作：

```
mainViewModel.isGotLiveData.observe(this){
```

```
        if(it == true){
            Log.d("SpreadShopTest", "got live data!")
        }else{
            Log.d("SpredShopTest", "didn't got live data")
            Toast.makeText(this@MainActivity, "refresh
failed",
                        Toast.LENGTH_SHORT).show()
        }

        runOnUiThread {
            goodsAdapter.notifyDataSetChanged()
            bindingMain.swipeRefresh.isRefreshing = false
        }
    }
}
```

一旦标志位设置，我们就起一个新线程，告知adapter数据发生了改变，并取消刷新的进度条。通过这样设置，就能实现即使刷新失败也不会让那个圈圈一直转了。另外，刷新布局的监听器只需要做这样一件事：

```
bindingMain.swipeRefresh.setOnRefreshListener {
    mainViewModel.getGoods(Command.GET_RECOMMAND)
}
```

这样我们只要下拉一刷新，就会发起请求，就会调用enqueue函数，就会将标志位设置成一个新的值，就会触发标志LiveData的observe，就会通知adapter数据发生了改变。经过这环环相扣的操作，从后端到前端都实现了数据更新操作。最后补充一下MainViewModel更新完的代码：

```
class MainViewModel: ViewModel() {

    val goodsList = ArrayList<Goods>()

    val isGotLiveData = MutableLiveData<Boolean>()

    val goodsLiveData: LiveData<Call<GoodsResponse>>
        get() = _goodsLiveData
    private val _goodsLiveData =
        MutableLiveData<Call<GoodsResponse>>()
```

```
val categoryLiveData: LiveData<Call<CategoryResponse>>
    get() = _categoryLiveData
private val _categoryLiveData =
    MutableLiveData<Call<CategoryResponse>>()

fun getGoods(cmd: String) {
    _goodsLiveData.value = Repository.getGoods(cmd)
}

fun getAllCategory(cmd: String){
    _categoryLiveData.value = Repository.getAllCategory(cmd)
}
}
```

以及整个MainActivity的流程代码：

```
class MainActivity : AppCompatActivity() {

    private lateinit var bindingMain: ActivityMainBinding
    private lateinit var bindingNavHeader: NavHeaderBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        bindingMain = ActivityMainBinding.inflate(layoutInflater)
        bindingNavHeader =
            NavHeaderBinding.inflate(layoutInflater)
        setContentView(bindingMain.root)

        setSupportActionBar(bindingMain.toolbar)

        //let和apply都可以
        supportActionBar?.apply {
            setDisplayHomeAsUpEnabled(true)
            setHomeAsUpIndicator(R.drawable.ic_menu)
        }

        val username = intent.getStringExtra("username")
        Log.d("SpreadShopTest", "Log in username: $username")

        if(bindingMain.navView.headerCount > 0){

            val header = bindingMain.navView.getHeaderView(0)
        }
    }
}
```

```
        val uname = header.findViewById<TextView>(R.id.user_name)
        uname.text = "user name: $username"
    }

    val mainViewModel = ViewModelProvider(this)

    .get(MainViewModel::class.java)

    // 打开MainActivity之后立刻发起请求获取所有的种类
    mainViewModel.getAllCategory(Command.GET_ALL_CATEGORY)

    // 打开之后获取一次推荐商品
    mainViewModel.getGoods(Command.GET_RECOMMAND)

    // 设置goodsRecycler的adapter
    // 由于ArrayList的构造函数，所以goodsList不为空
    val goodsLayoutManager = GridLayoutManager(this, 2)
    bindingMain.goodsRecycler.layoutManager =
    goodsLayoutManager
    val goodsAdapter = GoodsAdapter(this,
    mainViewModel.goodsList)
    bindingMain.goodsRecycler.adapter = goodsAdapter

    mainViewModel.goodsLiveData.observe(this){
        it.enqueue(object : Callback<GoodsResponse>{
            override fun onResponse(call:
Call<GoodsResponse>, response: Response<GoodsResponse>){
                val goodsResponse = response.body()
                if(goodsResponse != null){
                    Log.d("SpreadShopTest", "goodsResponse is
not null")
                    if(goodsResponse.success){
                        Log.d("SpreadShopTest",
"goodsResponse success!")
                    }
                    val list = goodsResponse.goods
                    mainViewModel.goodsList.clear()
                    mainViewModel.goodsList.addAll(list)
                    mainViewModel.isGotLiveData.value =
                }
            }
        })
    }
}
```

```
        true

        }else{
            Log.d("SpreadShopTest",
"goodsResponse fail!")
        }
    }else{
        Log.d("SpreadShopTest", "goodsResponse is
null")
    }
}

override fun onFailure(call: Call<GoodsResponse>,
t: Throwable) {
    t.printStackTrace()
    mainViewModel.goodsList.clear()
    mainViewModel.isGotLiveData.value = false
    Log.d("SpreadShopTest", "goodsResponse on
failure")
}
}
} // end mainViewModel.goodsLiveData.observe

mainViewModel.categoryLiveData.observe(this){
    it.enqueue(object: Callback<CategoryResponse>{
        override fun onResponse(
            call: Call<CategoryResponse>,
            response: Response<CategoryResponse>
        ) {
            val categoryResponse = response.body()
            if(categoryResponse != null){
                if(categoryResponse.success){
                    Log.d("SpreadShopTest", "category
success")
                    val list =
categoryResponse.categories

                    // category recycler
                    val categoryLayoutManager =
GridLayoutManager(this@MainActivity, 1)
                }
            }
        }
    })
}
```

```
bindingMain.categoryRecycler.layoutManager =  
  
categoryLayoutManager  
  
        val adapter =  
  
CategoryAdapter(this@MainActivity, list)  
  
        bindingMain.categoryRecycler.adapter  
= adapter  
  
    }else{  
        Log.d("SpreadShopTest", "Category  
fail")  
    }  
}else{  
    Log.d("SpreadShopTest", "category is  
null")  
}  
}  
  
    override fun onFailure(call:  
Call<CategoryResponse>, t: Throwable) {  
    t.printStackTrace()  
    Log.d("SpreadShopTest", "category on  
failure")  
}  
}  
}  
} // end mainViewModel.categoryLiveData.observe  
  
  
mainViewModel.isGotLiveData.observe(this){  
    if(it == true){  
        Log.d("SpreadShopTest", "got live data!")  
    }else{  
        Log.d("SpredShopTest", "didn't got live data")  
        Toast.makeText(this@MainActivity, "refresh  
failed", Toast.LENGTH_SHORT).show()  
    }  
}
```

```
        runOnUiThread {
            goodsAdapter.notifyDataSetChanged()
            bindingMain.swipeRefresh.isRefreshing = false
        }
    }
    bindingMain.navView.setNavigationItemSelected {
        when(it.itemId){
            R.id.nav_mybag -> Log.d("SpreadShopTest", "You
        clicked mybag")
            R.id.nav_order -> Log.d("SpreadShopTest", "You
        clicked myorder")
            R.id.nav_contact -> Log.d("SpreadShopTest", "You
        clicked Contact")
            R.id.nav_logout -> Log.d("SpreadShopTest", "You
        clicked logout")
        }
        bindingMain.drawerLayout.closeDrawers()
        true
    }

    bindingMain.swipeRefresh.setOnRefreshListener {
        mainViewModel.getGoods(Command.GET_RECOMMAND)
    }
}

}// end onCreate

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when(item.itemId){
        android.R.id.home ->
        bindingMain.drawerLayout.openDrawer(GravityCompat.START)
        R.id.backup -> Log.d("SpreadShopTest", "you clicked
    backup")
        R.id.delete -> Log.d("SpreadShopTest", "you clicked
    delete")
        R.id.settings -> Log.d("SpreadShopTest", "you clicked
    settings")
    }
    return true
}

override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.toolbar, menu)
}
```

```
        return true
    }

}
```

注：此时Category的操作流程还没改，最后要改成和goods一样的模式。

## 2022-10-26

今天我感觉并没有费多少功夫，但是今天对我项目的改变是最大的！



先来嘚瑟一下~~，我们按着昨天的思路来，将下拉刷新处理完毕之后，首先我们要做的是更改这个MaterialCardView，因为它之前实在是太丑了。在这里我就直接展示所有和商品信息相关的xml代码了：

首先是activity\_main.xml中的SwipeRefreshLayout：

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/swipe_refresh"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="110dp"
```

```
    app:layout_anchor="@+id/category_recycler"
    app:layout_anchorGravity="bottom"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    >

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/goods_recycler"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />

</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

可以看到，我们只是把对上面的留白改成了110dp。接下来是这个 RecyclerView中展示的卡片项目：

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:rippleColor="@color/material_dynamic_neutral90"
    android:layout_margin="5dp"
    >

    <!--
        app:strokeColor="@color/material_dynamic_neutral70"
        app:strokeWidth="5dp"      app:cardElevation="8dp"
        app:cardCornerRadius="8dp"-->

    <!--
        一张Material卡片，之后这一张张卡片都会添加到
        recyclerView当中

        centerCrop: 让图片保持原有比例填充满ImageView,
        并将超出屏幕的部分裁剪掉-->
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        >

        <ImageView
            android:id="@+id/goods_image"
            android:layout_width="match_parent"
            android:layout_height="100dp"
            android:scaleType="centerCrop"
        />

        <TextView
            android:id="@+id/goods_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
            android:layout_margin="5dp"
            android:text="test"
            android:textSize="16sp"
            android:gravity="center"

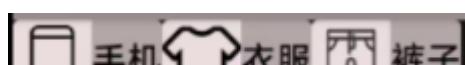
            android:background="@color/material_dynamic_neutral70"
            android:textColor="@color/black"
        />

    </LinearLayout>
</com.google.android.material.card.MaterialCardView>
```

最重要的是CardView中的layout\_height属性，我之前就是写的match\_parent，结果一张卡片直接和屏幕一样高。接下来，是展示类别的RecyclerView：

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/category_recycler"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
/>
```

这个的高度只有50dp，所以才能像这样短小精悍：





然后就是其中展示的类别卡片了：

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    app:rippleColor="@color/material_dynamic_neutral90"
    >

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        >

        <ImageView
            android:id="@+id/category_image"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_gravity="center_vertical"
            android:scaleType="fitCenter"
            android:background="@color/material_dynamic_neutral90"
            />

        <TextView
            android:id="@+id/category_name"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="1"
            android:layout_gravity="center"
            android:text="test"
            android:gravity="center"
            android:textColor="@color/black"
            android:textSize="16sp"
            >
    
```

```
        android:background="@color/material_dynamic_neutral70"
        />

    </LinearLayout>

</com.google.android.material.card.MaterialCardView>
```

这里的精髓还是在最外层布局： width是wrap， 所以横向只会包住图片和文字的宽度；而height是match， 而它的parent正好就是上面的 RecyclerView， 而它的高度是50dp， 所以卡片的高度也是50dp。另外这里 TextView的这两个参数： layout\_width 和 layout\_weight 表示它会在宽度上占掉所有图片剩下来的空间。

另外， 你还能看到， 我在标题栏加了一个搜索栏， 这用的是官方提供的 SearchView。它被当成菜单项加到了Toolbar当中：

```
<com.google.android.material.appbar.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="@color/material_dynamic_neutral60"

        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
        app:layout_scrollFlags="scroll|enterAlways|snap"
    />

</com.google.android.material.appbar.AppBarLayout>
```

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
```

```
<!--
    menu文件夹里的所有试图都是菜单
-->
```

toolbar就是上面工具栏专用的菜单

而nav\_menu就是Navigation View专用的菜单

-->

<!--

always: 永远显示再Toolbar中, 空间不够不显示

ifRoom: 屏幕够就显示, 不够显示再菜单中

never: 永远显示在菜单

-->

```
<item
    android:id="@+id/search_edit"
    android:icon="@drawable/ic_search"
    app:actionViewClass="android.widget.SearchView"
    app:showAsAction="always|collapseActionView"
    android:title="search_edit"
/>
</menu>
```

app:actionViewClass就是制定当前的item到底是什么类型的。接下来，我就将按照如下顺序来展示我所有的前端代码的逻辑部分：

- GoodsAdapter实现
- CategoryAdapter实现
- SearchView实现

首先是GoodsAdapter，主要的操作就是，当我点击了每一项，都要打开商品详情的Activity：

```
class GoodsAdapter(val context: Context, val goodsList:
List<Goods>): RecyclerView.Adapter<GoodsAdapter.ViewHolder>(){

    inner class ViewHolder(view: View):
    RecyclerView.ViewHolder(view){
        val goodsImage: ImageView =
        view.findViewById(R.id.goods_image)
        val goodsName: TextView =
        view.findViewById(R.id.goods_name)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): ViewHolder {
```

```
    val view =
LayoutInflater.from(context).inflate(R.layout.goods_item, parent,
false)
    val holder = ViewHolder(view)
    holder.itemView.setOnClickListener {
        val mainActivity = context as MainActivity
        val position = holder.adapterPosition
        val goods = goodsList[position]
        val intent = Intent(context,
GoodsActivity::class.java).apply {
            putExtra("goods_name", goods.goods_name)
            putExtra("goods_storage", goods.goods_storage)
            putExtra("goods_price", goods.goods_price)
            putExtra("goods_category", goods.goods_category)
            putExtra("goods_id", goods.goods_id)
            putExtra("username",
mainActivity.viewModel.username)
        }
        context.startActivity(intent)
    }
    return holder
//        return ViewHolder(view)
}

override fun onBindViewHolder(holder: ViewHolder, position:
Int) {
    val goods = goodsList[position]
    holder.goodsName.text = goods.goods_name
    val id = "ic_goods_${goods.goods_id}"

    Glide.with(context).load(Command.getImageByReflect(id)).into(holder.goodsImage)
}

override fun getItemCount() = goodsList.size
}
```

intent的部分特别简单，就不多说了。重要的是这个Glide的改变。我们之前只加载一个图片，而现在我们对于不同的商品能展示不同的图片了。这里我

们定义了一个getImageByReflect函数，这个函数就是通过反射去获取真正

的图片id：

```
fun getImageByReflect(imageName: String): Int{
    var field: Class<*>
    var res = 0
    try {
        field =
        Class.forName("com.example.spreadshop.R\\$drawable")
        res = field.getField(imageName).getInt(field)

    }catch (e: java.lang.Exception){
        e.printStackTrace()
        Log.d("SpreadShopTest", "getImageByReflect exception!")
    }

    return res
}
```

我们都知道，在`drawable`下创建了文件`ic_test.png`，那么对应的就会在`R.drawable`下新建一个Int类型的变量叫做`ic_test`。我们这个函数的目的就是通过前者去寻找后者。其中需要注意的是这个内部类的写法：

`com.example.spreadshop.R\\$drawable`在java中，我们只需要这么写：  
`com.example.spreadshop.R$drawable`，但是kotlin增加了字符串嵌套变量的操作，所以要转义一下。经过这么一个转换，我们就能按照`goods_id`去加载手机中已经存好的照片(这样其实是不对的，但是也没办法，我们还不知道怎么在MySQL中存图片)。

接下来是CategoryAdapter：

```
class CategoryAdapter(val context: Context, val categoryList:
List<Category>): RecyclerView.Adapter<CategoryAdapter.ViewHolder>
() {
    inner class ViewHolder(view: View):
    RecyclerView.ViewHolder(view){
        val categoryName: TextView =
        view.findViewById(R.id.category_name)
        val categoryImage: ImageView =
        view.findViewById(R.id.category_image)
    }
}
```

```
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view =
            LayoutInflator.from(context).inflate(R.layout.category_item,
            parent, false)
        val holder = ViewHolder(view)
        holder.itemView.setOnClickListener {
            val position = holder.adapterPosition
            val category = categoryList[position]
            val mainActivity = context as MainActivity

            mainActivity.mainViewModel.getGoods(Command.getCategoryGoods(category.category))
        }
        return holder
    }
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val category = categoryList[position]
        holder.categoryName.text = category.category
        Log.d("SpreadShopTest", "category name: ${category.category}")
        val id: Int
        when(category.category){
            "手机" -> {
                Log.d("SpreadShopTest", "category id: 手机")
                id = R.drawable.ic_phone
            }
            "衣服" -> {
                Log.d("SpreadShopTest", "category id: 衣服")
                id = R.drawable.ic_cloth
            }
            "裤子" -> {
                Log.d("SpreadShopTest", "category id: 裤子")
                id = R.drawable.ic_trousers
            }
            else -> {
                Log.d("SpreadShopTest", "category id: else")
                id = R.drawable.test_maotai
            }
        }
    }
}
```

```
    }

    Log.d("SpreadShopTest", "val id: $id")

    Glide.with(context).load(id).into(holder.categoryImage)
}

override fun getItemCount() = categoryList.size
}
```

这里的变化就是，我们要对每一项设置监听器：当点击这个类别时，就发起按着这个类别去找商品的请求。而返回类型是GoodsResponse，那么自然就会刷新下面的GoodsRecyclerView了。

另外，我们不是将这个RecyclerView改成了横着的吗？只需要这么一句话：

```
val categoryLayoutManager = GridLayoutManager(this, 1)
categoryLayoutManager.orientation =
LinearLayoutManager.HORIZONTAL
bindingMain.categoryRecycler.layoutManager =
categoryLayoutManager
val categoryAdapter = CategoryAdapter(this,
mainViewModel.categoryList)
bindingMain.categoryRecycler.adapter = categoryAdapter
```

就是第二行中的这句话，将方向变成了水平的。

然后是这个SearchView的逻辑代码。由于我们将它放在了Toolbar中作为菜单的一项，那么它的出生自然要在onCreateOptionsMenu方法中了：

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.toolbar, menu)

    val searchItem = menu?.findItem(R.id.search_edit)
    //searchEdit = findViewById(R.id.search_edit)
    //searchEdit = MenuItemCompat.getActionView(searchItem)
    as SearchView
    searchEdit = searchItem?.actionView as SearchView
```

```
        searchEdit.isSubmitButtonEnabled = true
        searchEdit.imeOptions = EditorInfo.IME_ACTION_SEARCH

        searchEdit.setOnQueryTextListener(object :
SearchView.OnQueryTextListener{
            override fun onQueryTextSubmit(query: String?): Boolean {
                if(query != null){

mainViewModel.getGoods(Command.getSearchGoods(query))
                }else{
                    Log.d("SpreadShopTest",
"SearchView: query is null")
                }
            }

inputMethodManager.hideSoftInputFromWindow(searchEdit.windowToken
, InputMethodManager.HIDE_NOT_ALWAYS)
                return true
            }

            override fun onQueryTextChange(newText: String?): Boolean {
                if(newText == ""){

mainViewModel.getGoods(Command.GET_RECOMMAND)
                }
                return true
            }
        })
        return true
    }
}
```

我们可以通过这两行代码从menu中获取到SearchView的实例：

```
val searchItem = menu?.findItem(R.id.search_edit)
searchEdit = searchItem?.actionView as SearchView
```

我注释掉的是java中已经过时的写法。接下来是对这个SearchView的参数进行一些设置。这里我就引入一些网站了：

[Android的SearchView详解 \(wjhsh.net\)](http://wjhsh.net)

### [详细解读Android中的搜索框（三）——SearchView - developer\\_Kale - 博客园\(cnblogs.com\)](#)

然后我们进行两个比较重要的设置：点击提交按钮发生的事，以及清空输入框发生的事。这里的代码很好看懂，唯一要注意的是在点击提交按钮后，要使用 `hideSoftInputFromWindow` 关闭输入法，这样能增加用户体验。

---

补充一点在NavigationView更新用户名的代码。如果直接使用binding去获取这个TextView并设置值是无效的，经过搜索找到了这种实现很像java的代码：

```
mainViewModel.username =  
    intent.getStringExtra("username").toString()  
    if(bindingMain.navView.headerCount > 0){  
        val header = bindingMain.navView.getHeaderView(0)  
        val uname = header.findViewById<TextView>(R.id.user_name)  
        uname.text = "user name: ${mainViewModel.username}"  
    }  
}
```

目前还并不清楚这么写起效果和不这么写没效果的原因，有待研究。

---

接下来是商品的详情页。这部分其实和《第一行代码》的12.7是一个模子，所以不重要的部分就一步带过了。首先是前端代码：

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.coordinatorlayout.widget.CoordinatorLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:fitsSystemWindows="true"  
>  
  
<com.google.android.material.appbar.AppBarLayout  
    android:id="@+id/app_bar"  
    android:layout_width="match_parent"
```

```
        android:layout_height="250dp"
        android:fitsSystemWindows="true"
    >

<com.google.android.material.appbar.CollapsingToolbarLayout
        android:id="@+id/collapsing_toolbar"
        android:layout_width="match_parent"
        android:layout_height="match_parent"

        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:contentScrim="@color/teal_200"
        app:layout_scrollFlags="scroll|exitUntilCollapsed">

    <ImageView
            android:id="@+id/goods_image_detail"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scaleType="centerCrop"
            app:layout_collapseMode="parallax"
        />

    <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar_detail"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_collapseMode="pin"
        />

</com.google.android.material.appbar.CollapsingToolbarLayout>
</com.google.android.material.appbar.AppBarLayout>

<androidx.core.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"

        app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <LinearLayout
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
```

```
<com.google.android.material.card.MaterialCardView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="15dp"
    android:layout_marginLeft="15dp"
    android:layout_marginRight="15dp"
    android:layout_marginTop="35dp"
    app:cardCornerRadius="4dp"
    >

    <TextView
        android:id="@+id/goods_text_detail"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="10dp"
        />
    </com.google.android.material.card.MaterialCardView>
    </LinearLayout>

</androidx.core.widget.NestedScrollView>

<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/buy_btn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:src="@drawable/ic_backup"
    app:layout_anchor="@+id/app_bar"
    app:layout_anchorGravity="bottom|end"
    android:contentDescription="buy goods"
    />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

然后是GoodsActivity和GoodsViewModel。没错，这里的逻辑又是比较复杂的，我们从Activity一步一步说起。

首先是展示Home键，这个键默认就是个返回的箭头，所以不用动：

```
setSupportActionBar(bindingGoods.toolbarDetail)
```

```
supportActionBar?.setDisplayHomeAsUpEnabled(true)
```

然后是一些ViewModel的设置：

```
goodsViewModel =  
ViewModelProvider(this).get(GoodsViewModel::class.java)  
  
goodsViewModel.setGoodsName(intent.getStringExtra("goods_name")  
?: "null")  
goodsViewModel.setGoodsCategory(intent.getStringExtra("goods_cate  
gory") ?: "null")  
goodsViewModel.setGoodsPrice(intent.getIntExtra("goods_price",  
9999))  
goodsViewModel.setGoodsStorage(intent.getIntExtra("goods_storage"  
, -1))  
goodsViewModel.setGoodsId(intent.getIntExtra("goods_id", -1))  
goodsViewModel.username = intent.getStringExtra("username") ?: ""  
goodsViewModel.isSetFullyLiveData.value = true
```

都是我们从intent拿到的数据，放到了当前Activity的ViewModel层。当所有的LiveData都设置完成后，将`isSetFullyLiveData`置为true。没错，这和我们前面的代码很相似。也就是`isGotGoodsLiveData`和`isGotCategoryLiveData`这样的逻辑(本文章中原来叫`isGotLiveData`，后来改了名字)。那么一旦设置了这个值，就该调用这个方法了：

```
goodsViewModel.isSetFullyLiveData.observe(this){  
    if(it == true){  
        bindingGoods.collapsingToolbar.title =  
        goodsViewModel.goodsNameLiveData.value  
        val imgId =  
        "ic_goods_${goodsViewModel.goodsIdLiveData.value}"  
  
        Glide.with(this).load(Command.getImageByReflect(imgId)).into(bind  
        ingGoods.goodsImageDetail)  
        bindingGoods.goodsTextDetail.text = generateGoodsDetail()  
    }else{  
        Log.d("SpreadShopTest", "isSetFullyLiveData: false")  
    }  
}
```

这样我们就能将标题，图片，详细信息等乱七八糟的信息都加载上了。这里

用到的generateGoodsDetail函数是这样的：

```
private fun generateGoodsDetail(): String{
    val res = StringBuilder()
    res.appendLine("goods_name:
    ${goodsViewModel.goodsNameLiveData.value}")
    res.appendLine("goods_category:
    ${goodsViewModel.goodsCategoryLiveData.value}")
    res.appendLine("goods_price:
    ${goodsViewModel.goodsPriceLiveData.value}")
    res.appendLine("goods_storage:
    ${goodsViewModel.goodsStorageLiveData.value}")
    val sb = goodsViewModel.goodsNameLiveData.value?.repeat(500)
    res.append(sb)
    return res.toString()
}
```

非常简单，就不多赘述了。

---

接下来是购买功能，我打算用这个FloatingActionButton去当购买按键，当点击之后，弹出一个窗口询问你购买的数量。而这时我恰好找到了一个非常好用的第三方库——XPopup：

[XPopup: 🔥 XPopup2.0版本重磅来袭，2倍以上性能提升，带来可观的动画性能优化和交互细节的提升！！！功能强大，交互优雅，动画丝滑的通用弹窗！可以替代Dialog, PopupWindow, PopupMenu, BottomSheet, DrawerLayout, Spinner等组件，自带十几种效果良好的动画，支持完全的UI和动画自定义！\(Powerful and Beautiful Popup, can absolutely replace Dialog, PopupWindow, PopupMenu, BottomSheet, DrawerLayout, Spinner. With built-in animators, very easy to custom popup view.\) \(gitee.com\)](#)

安装的时候遇到一个小问题，就是gradle7.0之后所有的仓库引入都放到了settings.properties中了：

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
```

```
        google()
        mavenCentral()
        maven {url 'https://jitpack.io'} // XPopup的依赖
    }}
```

这玩意儿非常好用，所以我直接给所有的代码了，一看就能看懂：

```
bindingGoods.buyBtn.setOnClickListener {

    XPopup.Builder(this@GoodsActivity).asInputConfirm("Buying
${goodsViewModel.goodsNameLiveData.value}", "Please enter the
purchase quantity: ", "1", object: OnInputConfirmListener{
        override fun onConfirm(text: String?) {

if(goodsViewModel.isSetFullyLiveData.value == true && text != null && text != "" && text.isDigitsOnly()){

    goodsViewModel.requestOrder(goodsViewModel.username,
    goodsViewModel.goodsIdLiveData.value!!, text.toInt())
        }else if(text == ""){

    goodsViewModel.requestOrder(goodsViewModel.username,
    goodsViewModel.goodsIdLiveData.value!!, 1)
        Log.d("SpreadShopTest", "Only Buy
One!")
        }else{
            Log.d("SpreadShopTest",
"XPopInput return Nothing, buy fail")
            return
        }
    }
}).show()
}
```

这里还很贴心的给用户设置了一个默认值1，表示不输入默认购买一件，并且XPopup正好还支持提示，所以在提示里打上"1"就好了。

```
goodsViewModel.orderLiveData.observe(this){
    it.enqueue(object : Callback<OrderResponse>{
        override fun onResponse(
            call: Call<OrderResponse>,
```

```
        response: Response<OrderResponse>
    ) {
    val orderResponse = response.body()
    if(orderResponse != null){
        val order = orderResponse.order
        if(orderResponse.success){

XPopup.Builder(this@GoodsActivity).asConfirm("Order Info",
order.message) {
    val mainActivity = SpreadShopApplication.context
    as MainActivity

mainActivity.mainViewModel.getGoods(Command.GET_RECOMMAND)
    this@GoodsActivity.finish()
}.show()
}else{
    Log.d("SpreadShopTest",
"orderResponse.success is fail, msg: ${order.message}")

XPopup.Builder(this@GoodsActivity).asConfirm("Failed!!!",
order.message
) {
    Toast.makeText(
        this@GoodsActivity,
        "buy failed over",
        Toast.LENGTH_SHORT
    ).show()
}.show()
}
}else{
    Log.d("SpreadShopTest", "orderResponse is null")
}
}

override fun onFailure(call: Call<OrderResponse>, t:
Throwable) {
    t.printStackTrace()
    Log.d("SpreadShopTest", "orderResponse on failure")
}
}
}
```

这里唯一需要注意的是这句话：

```
val mainActivity = SpreadShopApplication.context as MainActivity
mainActivity.mainViewModel.getGoods(Command.GET_RECOMMAND)
this@GoodsActivity.finish()
```

当购买成功，用户点击确定后，我们首先再发起一次获取推荐请求，然后关闭当前Activity。这样购买完自动会回到商城页面，并且数据也是最新的。而这个获取到mainActivity的实例在《第一行代码》中的14.1有讲。**但是我还是不太清楚，为啥这里获得的context就恰好是MainActivity呢？**

最后给一下GoodsViewModel的代码：

```
class GoodsViewModel: ViewModel() {
    val goodsNameLiveData = MutableLiveData<String>()
    val goodsStorageLiveData = MutableLiveData<Int>()
    val goodsPriceLiveData = MutableLiveData<Int>()
    val goodsCategoryLiveData = MutableLiveData<String>()
    val goodsIdLiveData = MutableLiveData<Int>()

    var username = ""

    val orderLiveData: LiveData<Call<OrderResponse>>
        get() = _orderLiveData
    private val _orderLiveData =
        MutableLiveData<Call<OrderResponse>>()

    val isSetFullyLiveData = MutableLiveData<Boolean>()

    fun setGoodsName(n: String){
        goodsNameLiveData.value = n
    }

    fun setGoodsStorage(s: Int){
        goodsStorageLiveData.value = s
    }

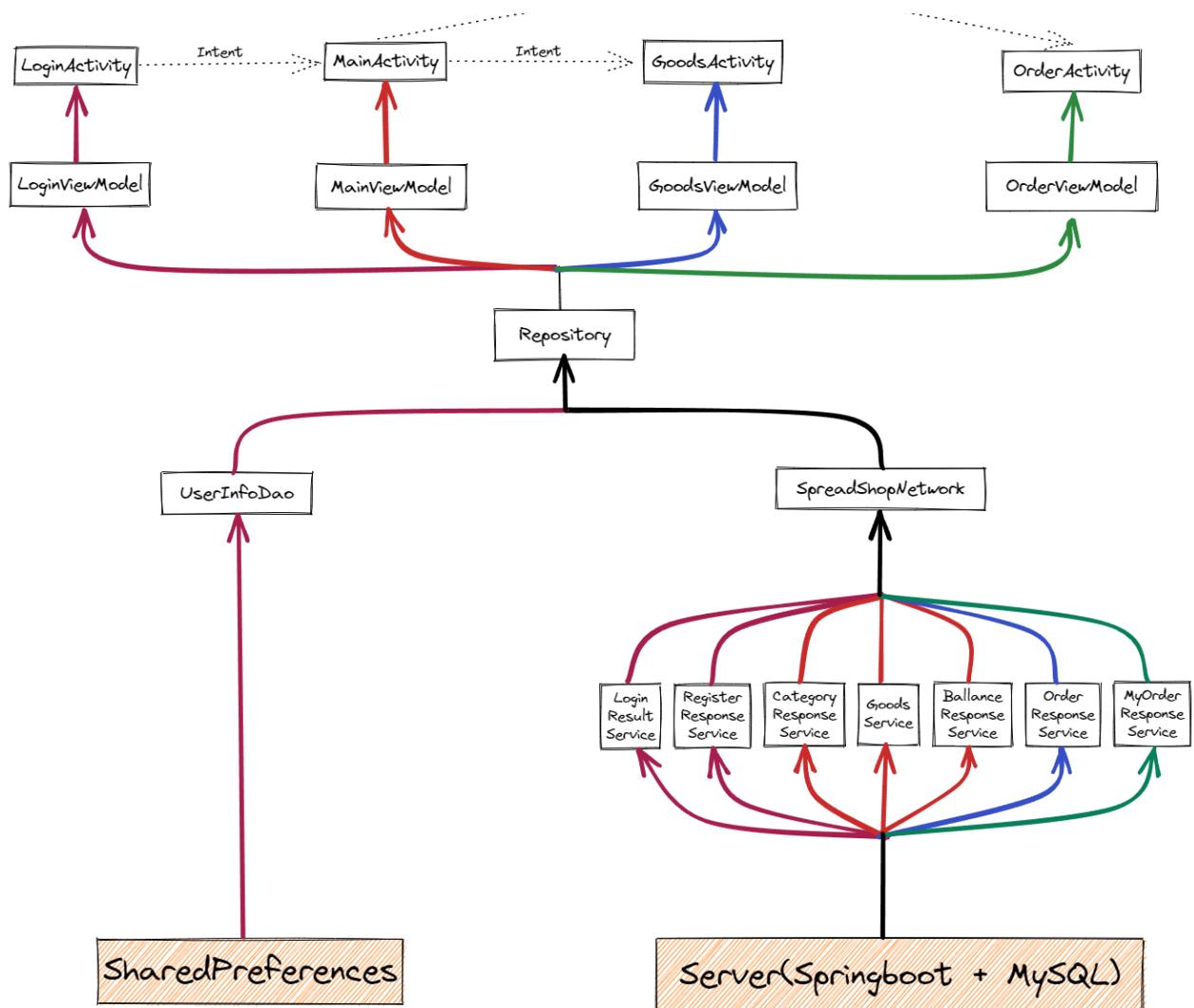
    fun setGoodsPrice(p: Int){
        goodsPriceLiveData.value = p
    }
}
```

```
fun setGoodsCategory(c: String){  
    goodsCategoryLiveData.value = c  
}  
  
fun setGoodsId(i: Int){  
    goodsIdLiveData.value = i  
}  
  
fun requestOrder(username: String, goods_id: Int, number: Int){  
    _orderLiveData.value = Repository.requestOrder(username,  
    goods_id, number)  
}  
}
```

## 2022-10-27

结项了！！！最后的操作，只不过是在已经有的技术基础上加了亿点功能而已。所以这里我给出了整个项目的MVVM架构的图：





实线箭头表示数据的流动方向；虚线箭头表示打开关系；每个箭头的颜色表示了数据是由哪个类掌管的，从Server或者SharedPreferences开始按着一个颜色走才能走通，黑色表示公共路径。

另外遇到一个小插曲，就是实现记住密码功能的时候。一开始我是按照《第一行代码》中天气预报程序里保存搜索过的城市那样做的。其中使用了一个获取全局Context的方式，也就是之前做购买成功自动返回MainActivity并发起网络请求功能时用到的技术。但是这个技术在这里居然行不通，只要一调用程序就会崩溃。之所以我这么做会这样，而书中却不会，最根本的原因就是：书中的**context是在Fragment中获取的，而我是在Activity中获取的**。在Fragment中获取时，Activity已经创建好了，所以这样的代码是没问题的：

```
override fun onCreate() {
    super.onCreate()
    context = applicationContext
```

}

但是我现在做的操作是：在Activity的`onCreate`方法中去调用`applicationContext`方法，显然是不可能完成的，也就导致了`context`没有被初始化。所以，在Activity中想要将自己这个`context`传递出去，还是老老实实把`this`当参数传出去吧：

```
if(loginViewMode.isUserInfoSaved(this)){
    val uinfo = loginViewMode.getSavedUserInfo(this)
    // 给输入框设置值要用setText不能用语法糖
    bindingLogin.accountEdit.setText(uinfo.username)
    bindingLogin.passwordEdit.setText(uinfo.password)
    bindingLogin.rememberPwd.isChecked = true
}
```

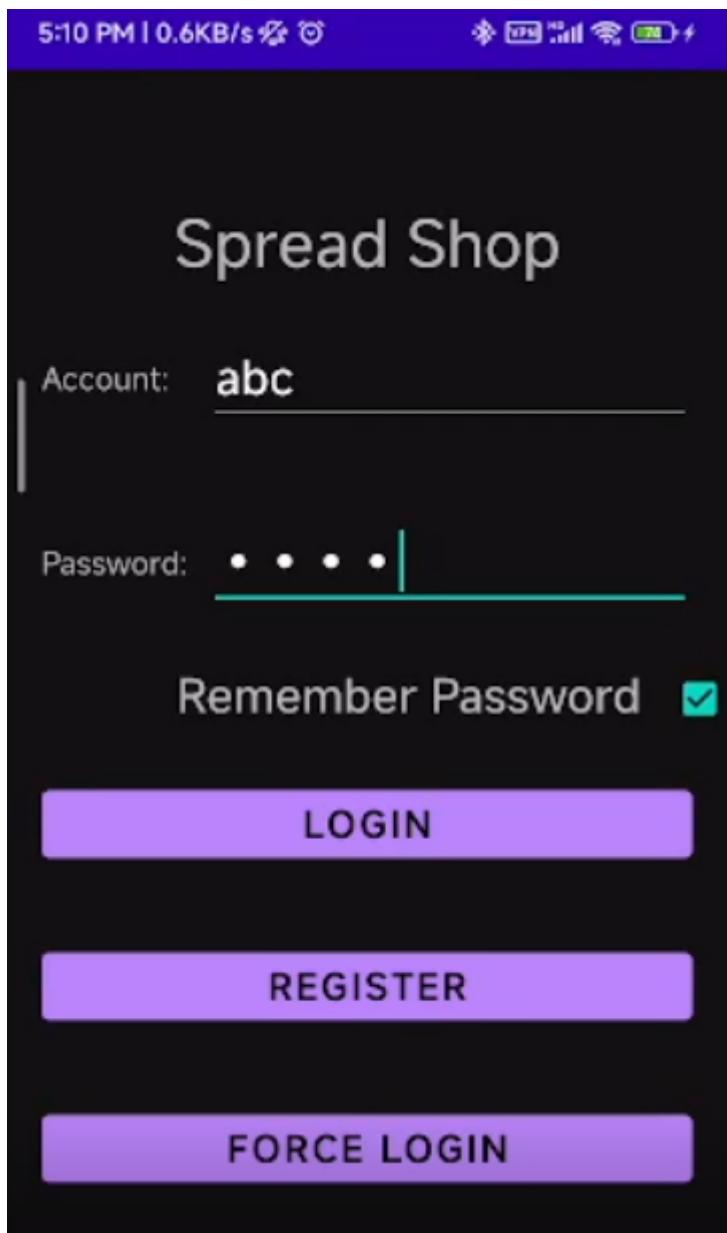
---

如果没有意外的话，**SpreadShop**这个项目就到此为止了，我的日记也会在此截止了，但是我对**Kotlin**和**Android**的探索会一直持续下去！！！

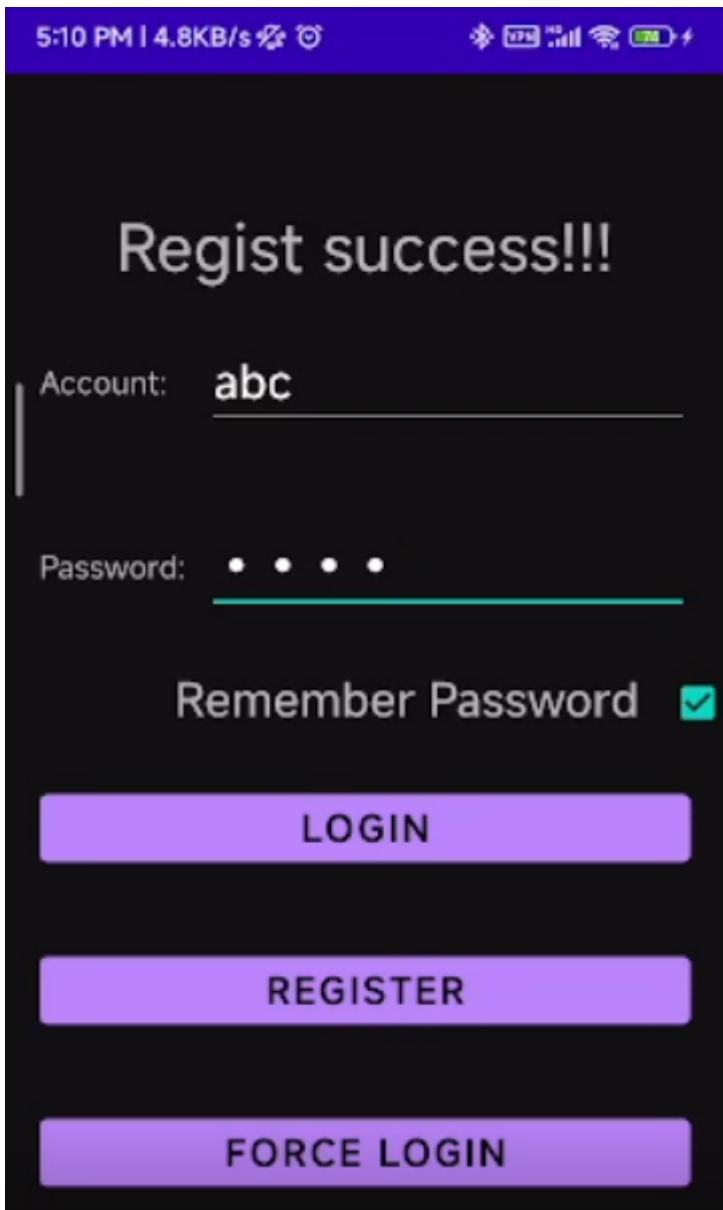
## 4. 项目演示

### 4.1 注册

现在我们注册一个用户abc，密码就设为asdf吧



点击REGISTER按钮，弹出Regist success!!!，表明注册成功，我们就可以用这个账号密码登录了！



## 4.2 登录

输入用户名和密码，点击点击*LOGIN*按钮即可发出登录请求。这里我们先展示几个登录失败的情况：

1. 输入的用户名未注册
2. 密码错误

# Unknown username

Account: abc

Password: • • •

Remember Password

**LOGIN**

**REGISTER**

**FORCE LOGIN**

# Wrong password for user [abc]

Account: abc

Password: • • • •

Remember Password

**LOGIN**

**REGISTER**

**FORCE LOGIN**

只有当你输入已注册的用户名和其对应的正确密码时，才能登录上我们的平台，进入主界面

## ≡ Goods



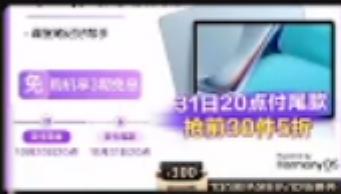
chair



cloth



computer



flat2



stationery12



chair9



earphone5



toy6



watch6



watch5

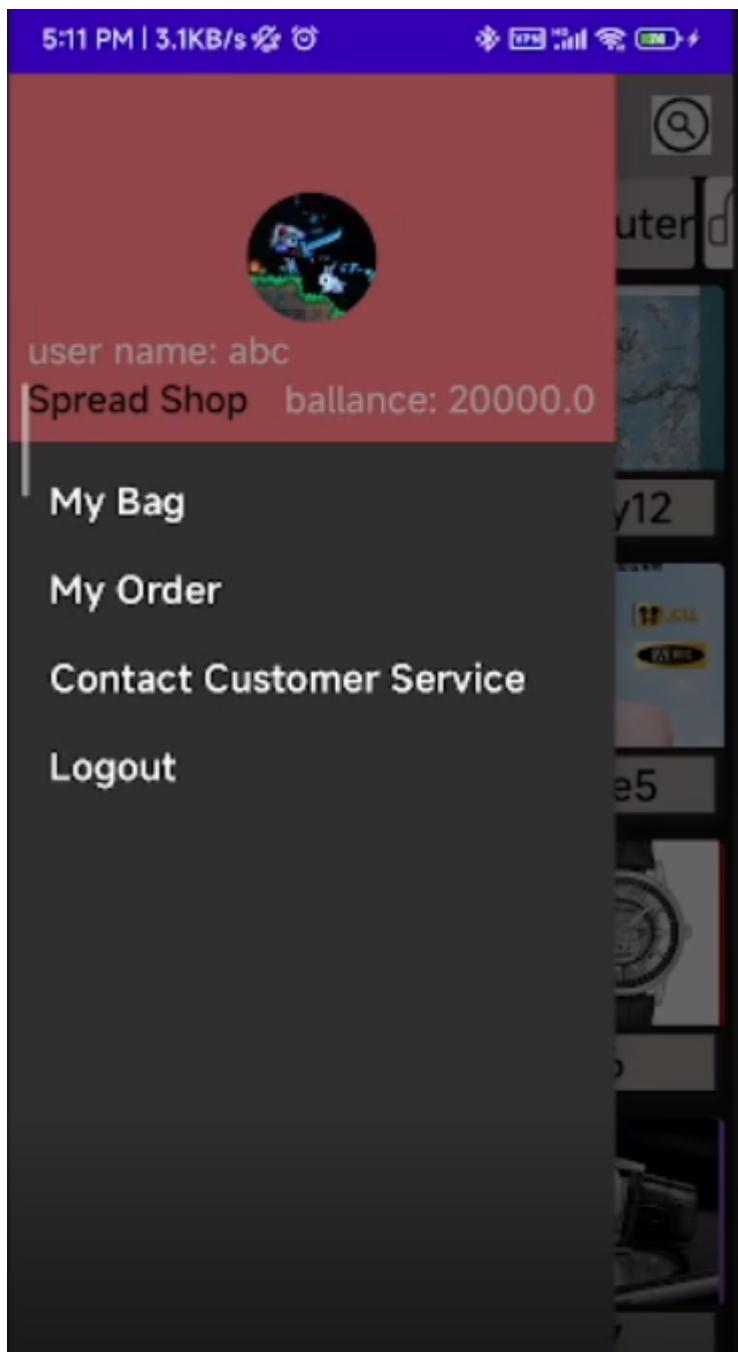


watch7

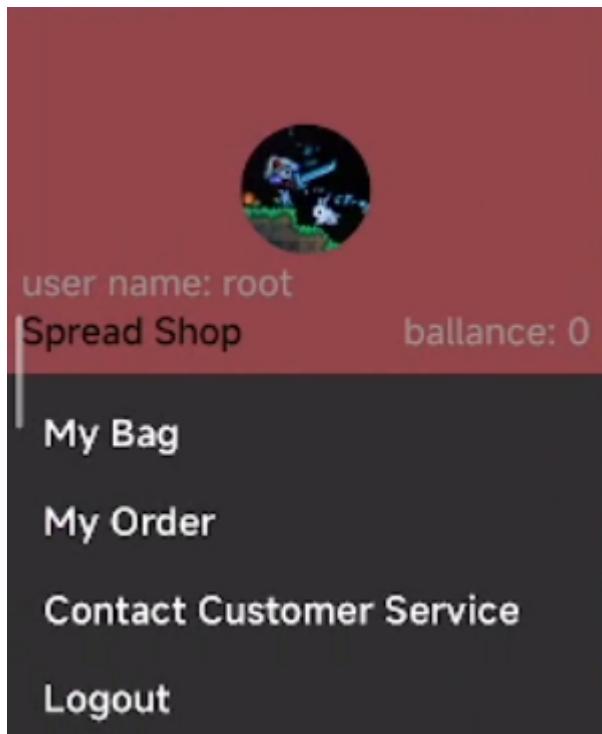


真无线降噪耳机 3

点击左上角的书页按钮，可以进入到侧滑菜单，这里会展示账户信息和余额

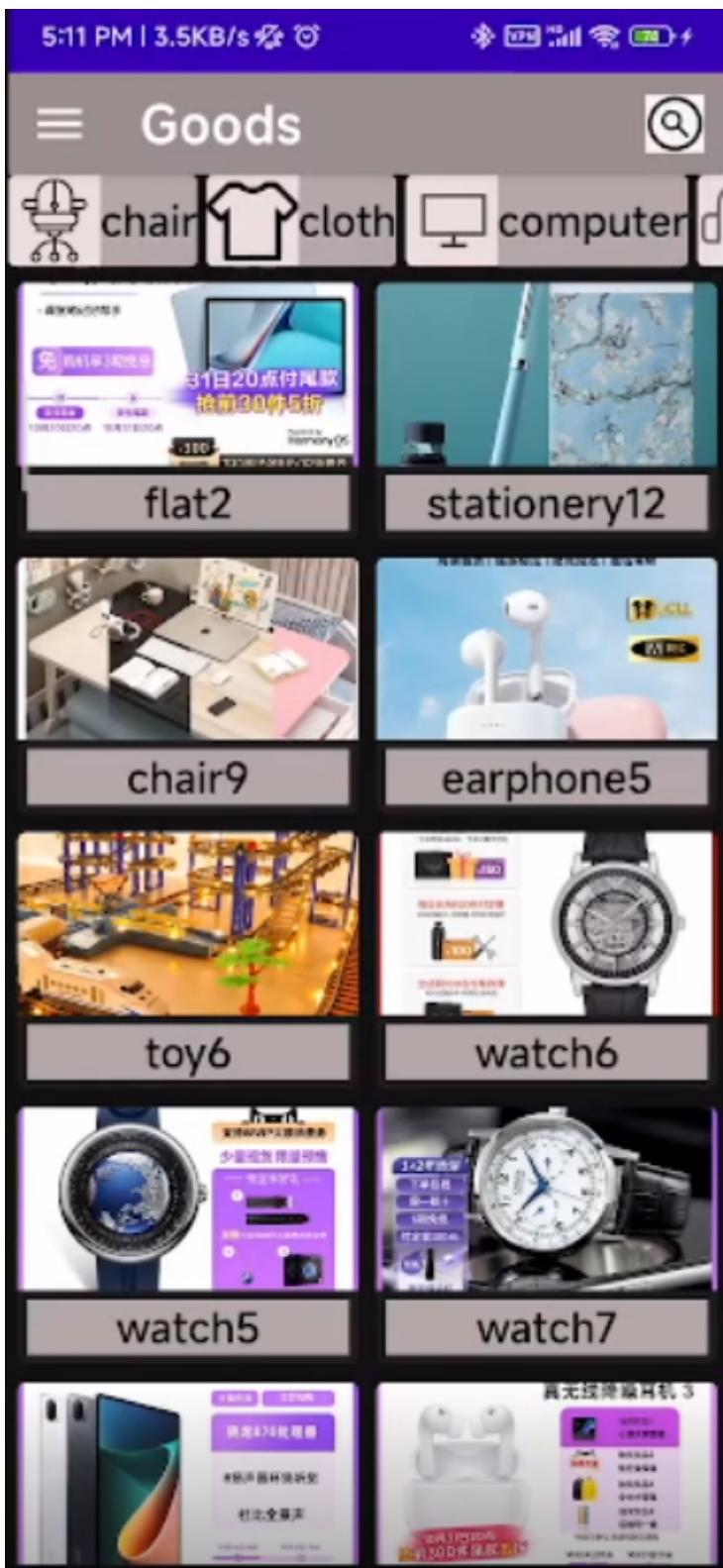


**FORCE LOGIN**则是我们在测试时用的功能，类似于常见的游客登录功能，不需要账号和密码，但登录后用户名为root且不会分配账户余额，这点可以在侧滑菜单查看



另外不得不提的是记住密码的功能，在登录时勾选*Remember Password*后点击*LOGIN*按钮，退出重启程序，它会记住用户名和密码

### 4.3 展示商品



主要分为两部分，我们可以看到，Toolbar下的第一横排是商品的种类信息，可以横向滑动查看，可以点击查看该分类的全部商品

### Goods



chair



cloth



computer



chair1



chair2



chair3



chair4



chair5



chair6



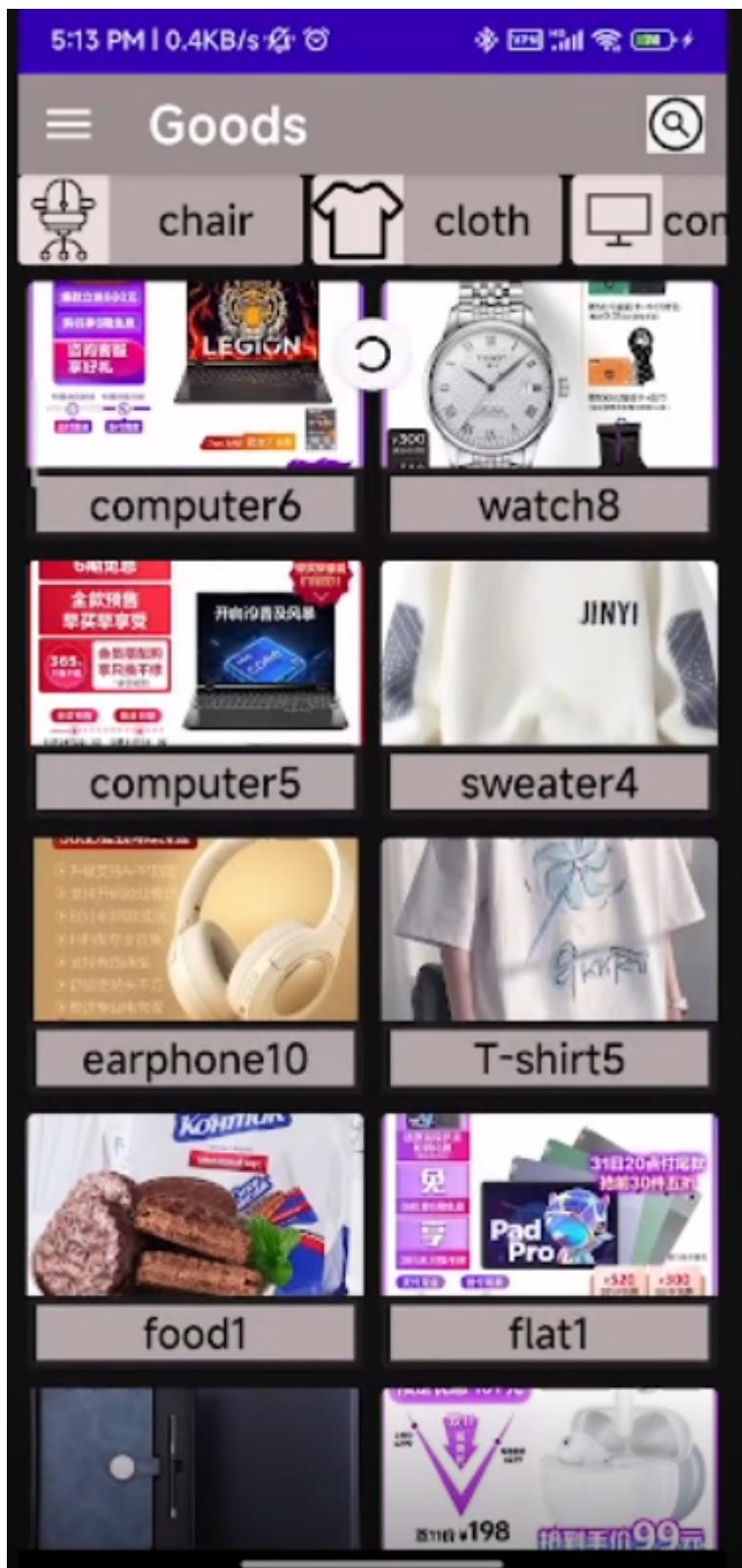
chair7



chair8

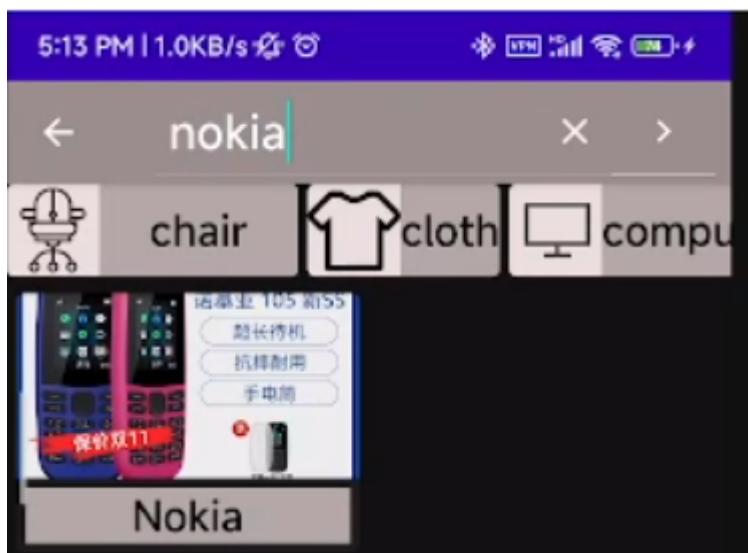
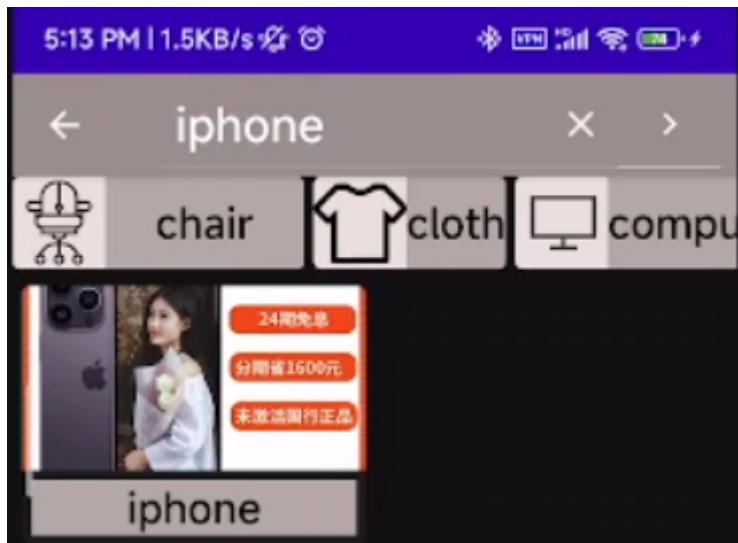


下面的部分则是我们随机展示的推荐商品，可以纵向滑动查看，同时可以通过上滑刷新出新的一批推荐商品



## 4.4 搜索商品

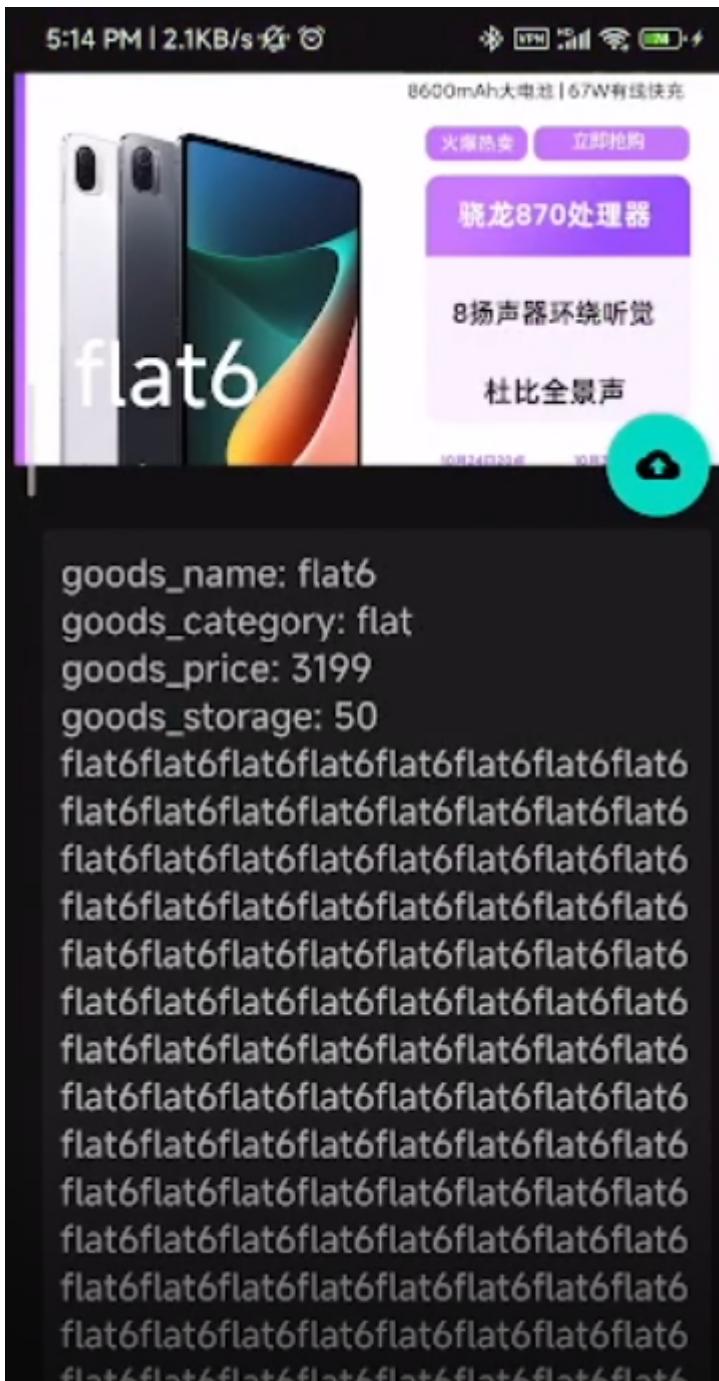
点击右上角的搜索按钮，程序会唤醒手机输入法，允许用户通过商品名称搜索商品，输入要搜索的商品名称点击>按钮或者输入法输入回车，即可发起搜索请求



另外，当搜索框清空或者点击X按钮之后，程序会自动展示一批新的推荐商品

## 4.5 商品详情页

点击任何一个商品，就可以跳转到该商品的详情页

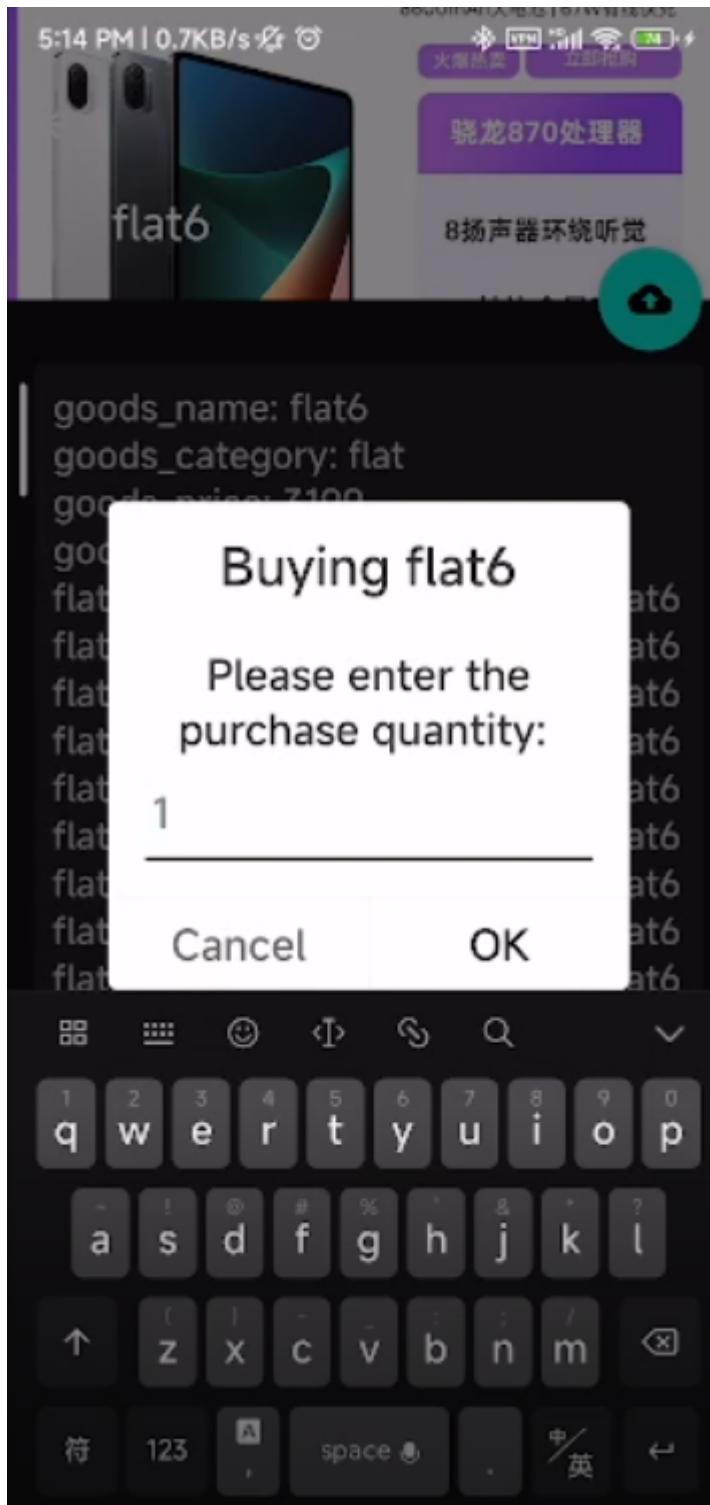


这里会展示出商品图片、名称、种类、价格和库存，最下方是商品的其他介绍，由于我们这里不知道该写些什么，就重复填充了商品的名称仅供测试

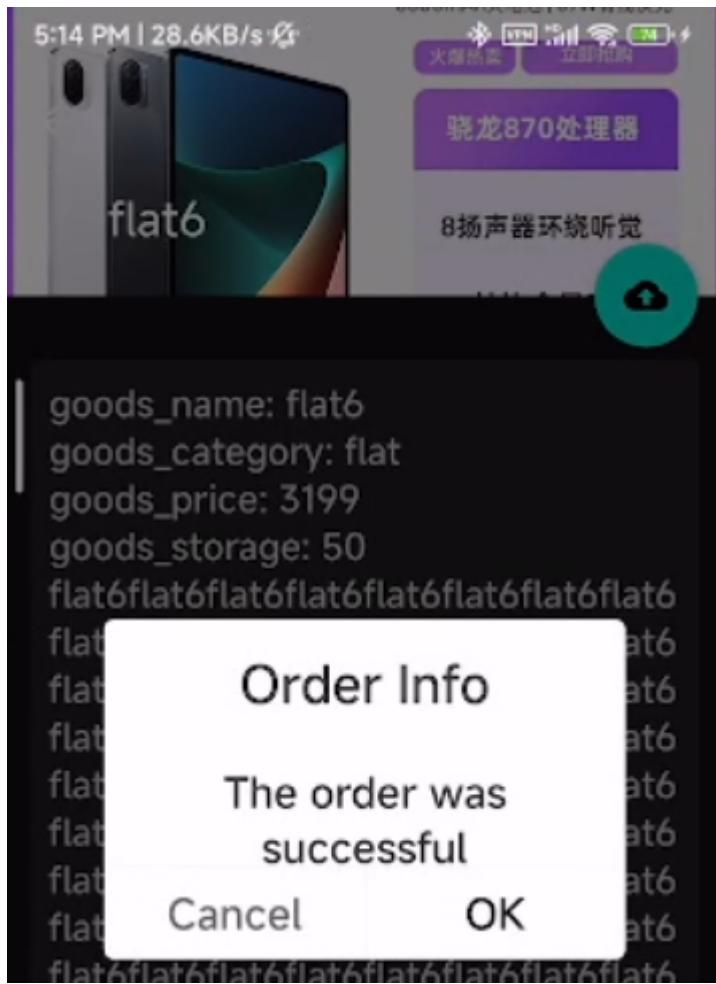
不得不提的是，当你上滑的时候，商品图片会以一个渐变动画的方式缩到顶部的Toolbar中，反之同理，非常美观

## 4.6 购买商品

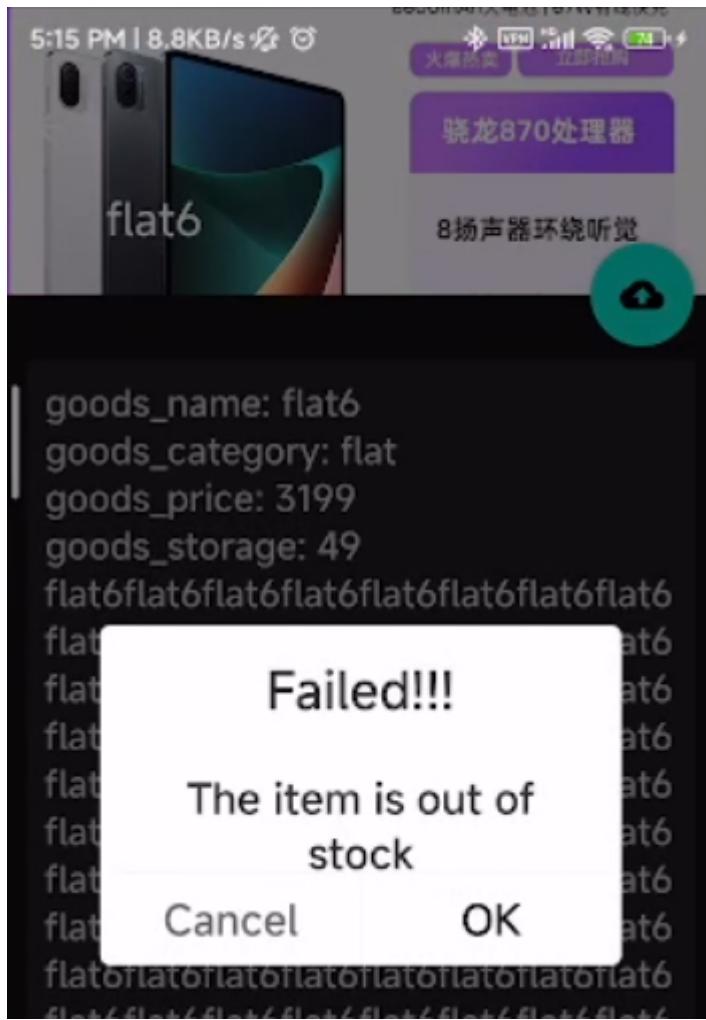
点击商品详情页中的绿色按钮，会弹出购买界面让用户确认购买的数量，默认为1



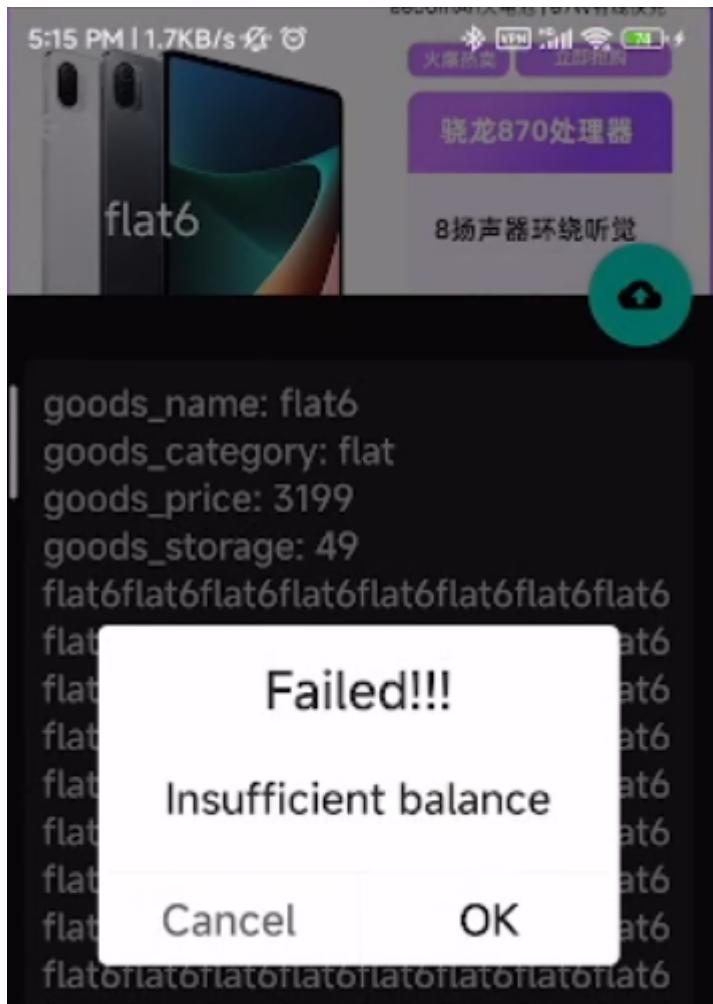
这里展示几种购买的情况：



购买成功！点击OK会返回主界面，对应的账户余额和商品库存也会发生变化



购买失败！商品库存不足，无法满足用户的需求，点击OK会返回商品详情页



购买失败！用户余额不足，点击OK会返回商品详情页

## 4.7 展示订单

点击侧滑菜单的My Order，可以展示用户所有订单的信息，同样支持下拉刷新

## ← My Order

order time: 2022-10-22T15:41:21

goods name: iphone

goods num: 3

---

username: spreadzhao

order id: 3

order time: 2022-10-22T15:41:58

goods name: HUAWEI

goods num: 2

---

username: spreadzhao

order id: 5

order time: 2022-10-27T11:52:23

goods name: toy3

goods num: 1

---

username: spreadzhao

order id: 7

order time: 2022-10-27T13:55:40

goods name: computer9

goods num: 1

---

username: spreadzhao

order id: 15

order time: 2022-10-30T21:33:04

goods name: earphone7

goods num: 5

