

宝贵建议请发送至：[wangzhenyang@itcast.cn](mailto:wangzhenyang@itcast.cn)



黑马程序员  
itheima.com 上海

—编程，始于黑马

# Android 课程同步笔记

Beta 0.02 版

By 阳哥

1. 数据准备 (★★)	2
2. Fragment 实现底部切换菜单 (★★)	5
2.1 布局文件的实现	5
2.2 主业务逻辑的实现	7
3. 智慧北京之左侧侧拉栏 (★★)	10
3.1 创建智慧北京工程	10
3.2 xUtils 简介	11
3.3 ViewUtils 简介	12
3.4 左侧侧拉栏的实现	14
4. 智慧北京之 ViewPager 实现 tab 切换菜单 (★★★★)	16
4.1 主界面框架的实现	16
4.2 Android 触摸事件分发机制	22
4.3 主界面框架的改进	39
5. 请求网络 (★★)	41
5.1 客户端请求网络	41
5.2 服务端的搭建	45
6. Android 布局文件优化 (★★★)	46
6.1 merge 的使用	47
6.2 ViewStub 的使用	51
6.3 include 的使用	52

## Android 智慧北京-02 数据准备

### 1.数据准备 (★★)

智慧北京新闻客户端请求的数据都是来自后台服务器，Android 手机作为客户端。客户端和服务端之间的数据交换是以 json 格式为基础的。因此这一节，为了方便模拟数据，我将把后台服务器搭建好同时将后台的主要 json 数据结构解释一下。

◆ 数据下载地址：<http://pan.baidu.com/s/1ntn5AEP>

下载好的数据文件夹打开以后其内容预览：

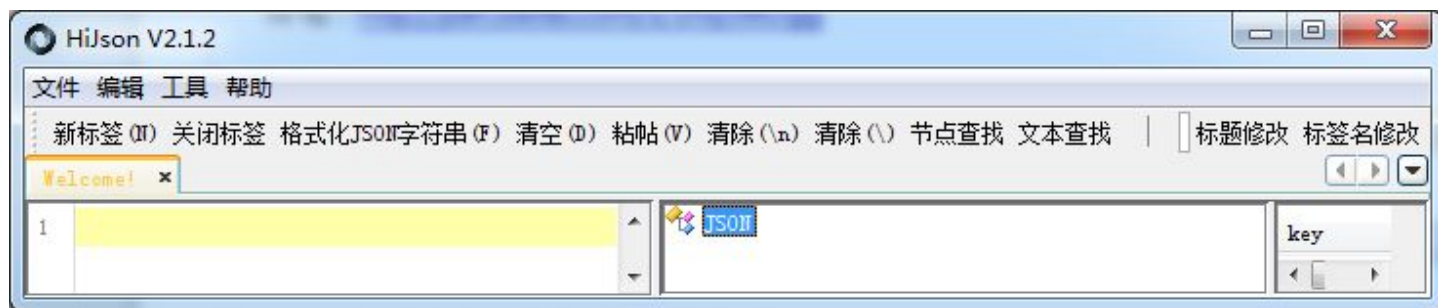
名称	修改日期	类型	大小
10002	2015/1/13 13:58	文件夹	
10003	2015/1/13 13:58	文件夹	
10004	2015/1/13 13:58	文件夹	
10005	2015/1/13 13:58	文件夹	
10006	2015/1/13 13:58	文件夹	
10007	2015/1/13 13:59	文件夹	
10008	2015/1/13 13:59	文件夹	
10009	2015/1/13 13:59	文件夹	
10010	2015/1/13 13:59	文件夹	
10011	2015/1/13 13:59	文件夹	
10012	2015/1/13 13:59	文件夹	
10014	2015/1/13 13:59	文件夹	
10091	2015/1/13 13:59	文件夹	
10093	2015/1/13 13:59	文件夹	
10094	2015/1/13 13:59	文件夹	
10095	2015/1/13 13:59	文件夹	
10105	2015/1/13 13:59	文件夹	
META-INF	2015/1/13 13:59	文件夹	
WEB-INF	2015/1/13 13:59	文件夹	
categories.json	2014/10/10 8:55	JSON 文件	2 KB
comment_1.json	2014/10/10 8:55	JSON 文件	1 KB
common.json	2014/10/10 8:55	JSON 文件	1 KB
index.jsp	2014/10/10 8:55	JSP 文件	1 KB

◆ 查看 json 数据文件软件 HiJSON 下载地址：

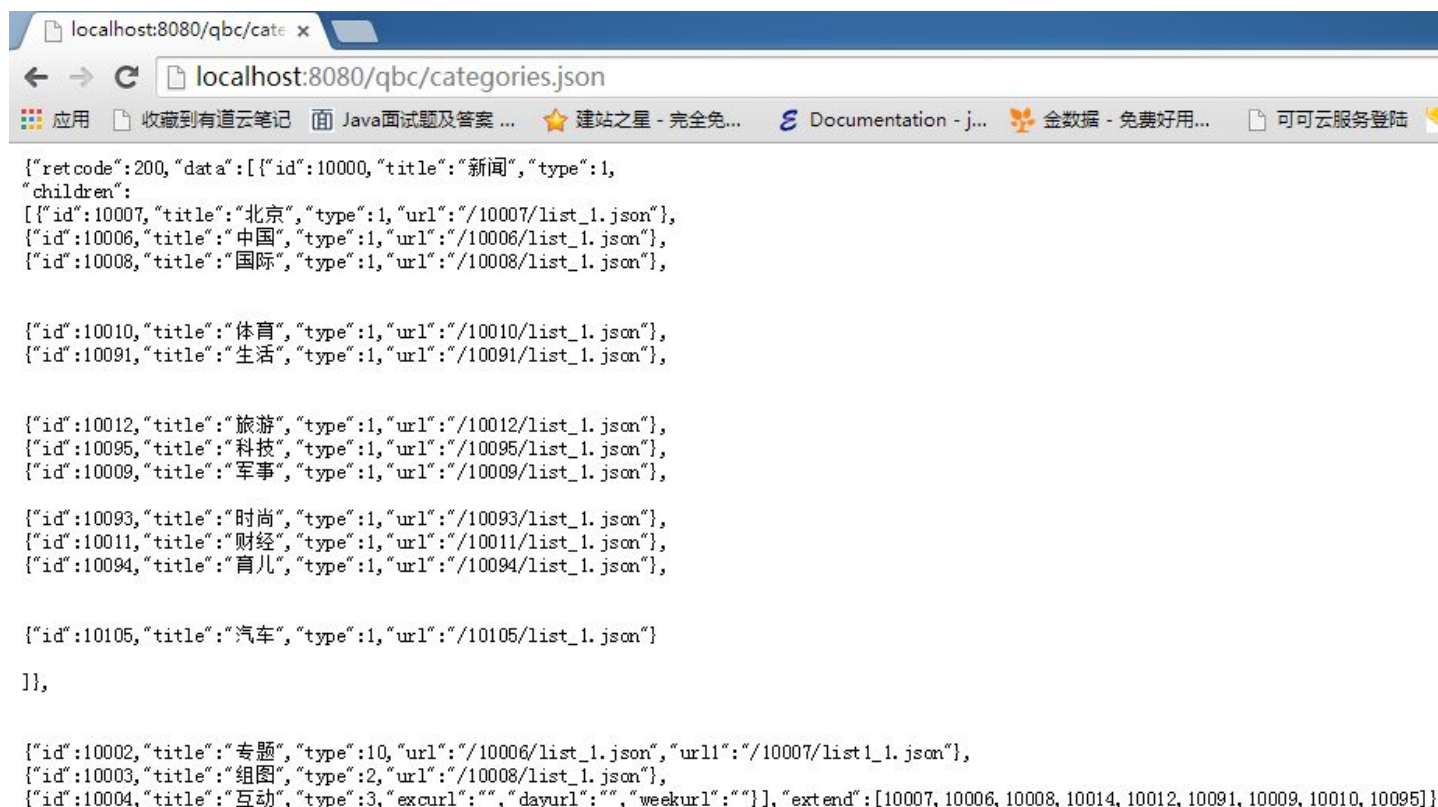
32 位：<http://pan.baidu.com/s/1hq5WDgg>

64 位：<http://pan.baidu.com/s/1ntkNNAh>

该软件小的一个工具软件，无需安装。打开以后预览图如下：



下载好的数据需要放到 tomcat 中，并运行 tomcat 服务器。服务器运行起来后在浏览器中输入地址，访问 json 数据。访问成功截图如下（我用的是 chrome 浏览器，如果是其他浏览器可能是文件的下载而不是自己将 json 打开）。

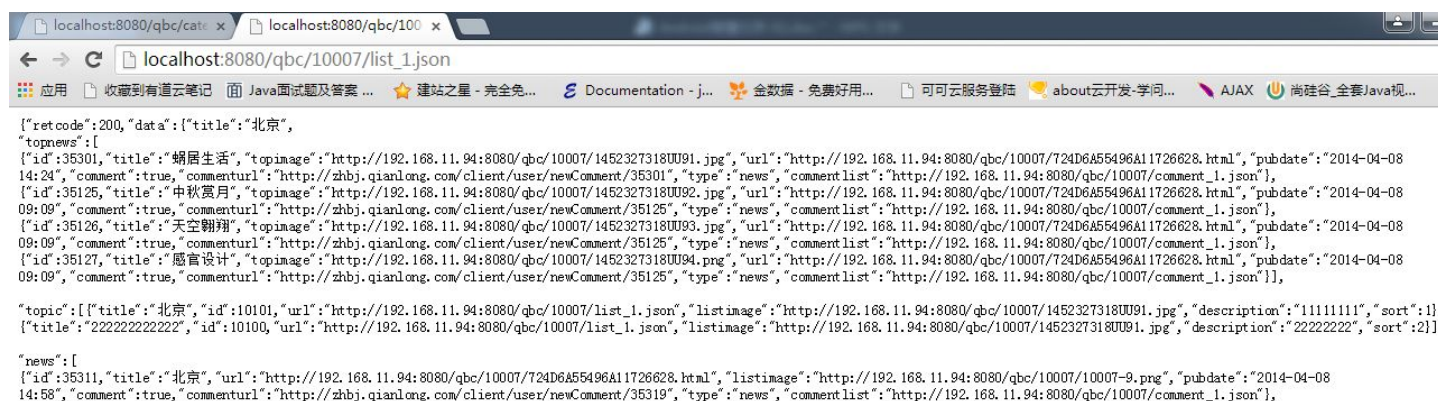


**Tips**：关于上面数据的说明。

◆ 上面的北京、中国、国际、体育等等对应如下实际运行图的新闻标题分类。



◆ 每一个标题中有一个 `url:/10007/list_1.json`，是一个链接，链接到一个地址，该地址展示的内容就是改标题下的新闻内容和对应的图片。我们直接在浏览器中输入该 url 查看到的数据内容如下所示：



注意上图中的 10007 文件夹下的 json 文件里面包含了 ip 地址，这个 ip 地址大家在使用的时候一定要记得修改成自己本地电脑的 IP 地址，不然 Android 模拟器访问不到（如果只在模拟器上使用把 ip 改成 10.0.2.2 也行，注：可不是 localhost）。

## 2.Fragment 实现底部切换菜单 (★★)

如下图所示，我们将通过 Fragment 实现其菜单功能。



底部菜单功能直接在第一天的《引导页》工程中的 MainActivity.java 中编写。

### 2.1 布局文件的实现

我们的布局整体是一个 LinearLayout，里面分为两个部分，上面黑色的为一个 FrameLayout，用来替换 Fragment，

下面的菜单式 RadioGroup 控件。每一个菜单末 RadioGroup 中的子控件 RadioButton。

布局文件名为 frag\_home.xml，其代码清单如下所示：

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



```
<FrameLayout
android:id="@+id/layout_content"
android:layout_width="match_parent"
android:layout_height="0dip"
android:layout_weight="1.0"
/>
<RadioGroup
    android:id="@+id/main_radio"
    android:layout_width="fill_parent"
    android:layout_height="60dp"
    android:layout_gravity="bottom"
    android:background="@drawable/bottom_tab_bg"
    android:gravity="center_vertical"
    android:orientation="horizontal"
    android:paddingTop="2dp" >

    <RadioButton
        android:id="@+id/rb_function"
        style="@style/main_tab_bottom"
        android:drawableTop="@drawable/icon_function"
        android:text="@string/tab_function" />

    <RadioButton
        android:id="@+id/rb_news_center"
        style="@style/main_tab_bottom"
        android:drawableTop="@drawable/icon_newscenter"
        android:text="@string/tab_news_center" />

    <RadioButton
        android:id="@+id/rb_smart_service"
        style="@style/main_tab_bottom"
        android:drawableTop="@drawable/icon_smartservice"
        android:text="@string/tab_smart_service" />

    <RadioButton
        android:id="@+id/rb_gov_affairs"
        style="@style/main_tab_bottom"
        android:drawableTop="@drawable/icon_govaffairs"
        android:text="@string/tab_gov_affairs" />

    <RadioButton
        android:id="@+id/rb_setting"
        style="@style/main_tab_bottom"
```

```
        android:drawableTop="@drawable/icon_setting"
        android:text="@string/tab_setting" />
    </RadioGroup>

</LinearLayout>
```

## 2.2 主业务逻辑的实现

在 MainActivity.java 中实现底部菜单的核心业务，其代码清单如下：

```
public class MainActivity extends FragmentActivity {
    private FrameLayout layout_content;
    private RadioGroup radioGroup;
    protected int index;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.frag_home);

        layout_content = (FrameLayout) findViewById(R.id.layout_content);
        radioGroup = (RadioGroup) findViewById(R.id.main_radio);

        radioGroup.setOnCheckedChangeListener(new OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                switch (checkedId) {
                    case R.id.rb_function:
                        //首页，在此次创建一个索引，根据索引再去获取 fragment 适配器中 fragment
                        index = 0;
                        break;
                    case R.id.rb_news_center:
                        index = 1;
                        //新闻中心
                        break;
                    case R.id.rb_smart_service:
                        index = 2;
                        //智慧服务
                        break;
                }
            }
        });
    }
}
```

对象



```
        case R.id.rb_gov_affairs:
            index = 3;
            //政务信息
            break;
        case R.id.rb_setting:
            index = 4;
            //设置
            break;
    }
    //通过 Fragment 数据适配器去获取对应索引的 fragment 对象，核心实现就是去调用
getItem(index) 方法
    Fragment fragment = (Fragment)
fragmentStatePagerAdapter.instantiateItem(layout_content, index);
    //通过索引指定的 fragment 去替换指定的帧布局
    fragmentStatePagerAdapter.setPrimaryItem(layout_content, 0, fragment);
    //提交一个事物
    fragmentStatePagerAdapter.finishUpdate(layout_content);
    }
});
radioGroup.check(R.id.rb_function);
}

FragmentStatePagerAdapter fragmentStatePagerAdapter = new
FragmentStatePagerAdapter(getSupportFragmentManager()) {
    //fragment 对象的个数
    @Override
    public int getCount() {
        return 5;
    }
    //获取相应的 fragment 对象
    @Override
    public Fragment getItem(int arg0) {
        BaseFragment baseFragment = null;
        switch (arg0) {
            case 0:
                baseFragment = new FunctionFragment();
                break;
            case 1:
                baseFragment = new NewsCenterFragment();
                break;
            case 2:
                baseFragment = new SmartServiceFragment();
```

```

        break;
    case 3:
        baseFragment = new GovAffairsFragment();

        break;
    case 4:
        baseFragment = new SettingFragment();
        break;
    }
    return baseFragment;
}
};
}

```

**Tips** : 因为不同的 Fragment 表现形式类似的，因此上面代码抽取出了 BaseFragment 类，其代码清单如下：

```

public class BaseFragment extends Fragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return super.onCreateView(inflater, container, savedInstanceState);
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
    }

    //设置 Menu 是否可见，menuVisible 如果为 true 则当期 fragment 对象 onCreateView 返回的
    //view 可见，否则就不可见
    @Override
    public void setMenuVisibility(boolean menuVisible) {
        if(getView()!=null){
            getView().setVisibility(menuVisible?View.VISIBLE:View.GONE);
        }
    }
}

```



**Tips** 在主业务逻辑中用到了 5 个 Fragment 类继承了 BaseFragment，分别为

这 5 个 Fragment 都比较简单，代码也一样。

```
public class SmartServiceFragment extends BaseFragment {  
  
    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container,  
        Bundle savedInstanceState) {  
        TextView textView = new TextView(getActivity());  
        textView.setText("智慧服务");  
        return textView;  
    }  
}
```

## 3.智慧北京之左侧侧拉栏 (★★)

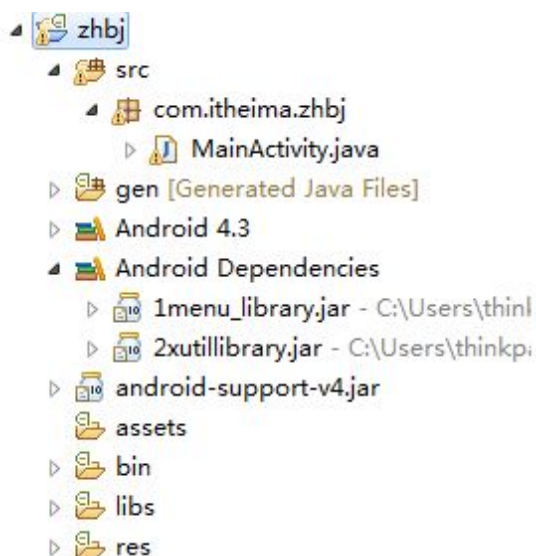
上面的独立小工程演示了 Fragment 如何实现菜单功能，可以作为基础知识来学习。从这一节开始，将正式创建智慧北京工程，首先完成的是左侧侧拉栏的实现。

左侧侧拉栏依旧使用 SlidingMenu 实现，而中间的菜单则不再使用 Fragment 实现，改为用 ViewPager 实现。

### 3.1 创建智慧北京工程

创建智慧北京工程，并引入 SlidingMenu 库工程和 Xutils 库工程。

新创建好的工程目录结构如下：



**Tips:** 在上面我们用到了 xUtils 库工程，xUtils 是目前最流行的开源 Android 库。下面将对 xUtils 作简单的介绍。

## 3.2 xUtils 简介

◆ xUtils 下载地址：<https://github.com/wyouflf/xUtils>

◆ xUtils 简介

1) xUtils 包含了很多实用的 android 工具。

2) xUtils 支持大文件上传，更全面的 http 请求协议支持(10 种谓词)，拥有更加灵活的 ORM，更多的事件注解

支持且不受混淆影响...

3) xUtils 最低兼容 android 2.2 (api level 8)

◆ xUtils 四大模块

1) DbUtils 模块

① android 中的 orm 框架，一行代码就可以进行增删改查；

② 支持事务，默认关闭；

③ 可通过注解自定义表名，列名，外键，唯一性约束，NOT NULL 约束，CHECK 约束等（需要混淆的时候请注解表名和列名）；

- ④支持绑定外键，保存实体时外键关联实体自动保存或更新；
- ⑤自动加载外键关联实体，支持延时加载；
- ⑥支持链式表达查询，更直观的查询语义，参考下面的介绍或 sample 中的例子。

## 2 ) ViewUtils 模块

- ①android 中的 ioc 框架，完全注解方式就可以进行 UI，资源和事件绑定；
- ②新的事件绑定方式，使用混淆工具混淆后仍可正常工作；
- ③目前支持常用的 20 种事件绑定，参见 ViewCommonEventListener 类和包 com.lidroid.xutils.view.annotation.event。

## 3 ) HttpUtils 模块

- ①支持同步，异步方式的请求；
- ②支持大文件上传，上传大文件不会 oom；
- ③支持 GET，POST，PUT，MOVE，COPY，DELETE，HEAD，OPTIONS，TRACE，CONNECT 请求；
- ④下载支持 301/302 重定向，支持设置是否根据 Content-Disposition 重命名下载的文件；

## 4 ) BitmapUtils 模块

- ①加载 bitmap 的时候无需考虑 bitmap 加载过程中出现的 oom 和 android 容器快速滑动时候出现的图片错位等现象；
- ②支持加载网络图片和本地图片；
- ③内存管理使用 lru 算法，更好的管理 bitmap 内存；
- ④可配置线程加载线程数量，缓存大小，缓存路径，加载显示动画等...

在本节中我们将用到的是 xUtils 的 ViewUtils 模块。

## 3.3 ViewUtils 简介

完全注解方式就可以进行 UI 绑定和事件绑定。

无需 findViewById 和 setClickListener 等。

```
// xUtils 的 view 注解要求必须提供 id，以使代码混淆不受影响。
@Inject(R.id.textView)
TextView textView;
//@Inject(R.id.textView, parentId=R.id.parentView)//TextView textView;

@ResInject(id = R.string.label, type = ResourceType.String)
private String label;
// 取消了之前使用方法名绑定事件的方式，使用 id 绑定不受混淆影响// 支持绑定多个 id @OnClick({R.id.id1,
R.id.id2, R.id.id3})// or @OnClick(value={R.id.id1, R.id.id2, R.id.id3}, parentId={R.id.pid1,
R.id.pid2, R.id.pid3})// 更多事件支持参见 ViewCommonEventListener 类和包
com.lidroid.xutils.view.annotation.event。
@OnClick(R.id.test_button)
public void testButtonClick(View v) { // 方法签名必须和接口中的要求一致
    ...
}...//在 Activity 中注入：
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ViewUtils.inject(this); //注入 view 和事件
    ...
    textView.setText("some text...");
    ...
}...//在 Fragment 中注入：
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
{
    View view = inflater.inflate(R.layout.bitmap_fragment, container, false); // 加载 fragment
    布局
    ViewUtils.inject(this, view); //注入 view 和事件
    ...
}...//在 PreferenceFragment 中注入：
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    ViewUtils.inject(this, getPreferenceScreen()); //注入 view 和事件
    ...
}
// 其他重载
```



```
// inject(View view);
// inject(Activity activity)
// inject(PreferenceActivity preferenceActivity)
// inject(Object handler, View view)
// inject(Object handler, Activity activity)
// inject(Object handler, PreferenceGroup preferenceGroup)
// inject(Object handler, PreferenceActivity preferenceActivity)
```

### 3.4 左侧侧拉栏的实现

1

在 3.1 节中创建的工程中国，修改 MainActivity 类。其代码清单如下：

```
public class MainActivity extends SlidingFragmentActivity {
    private SlidingMenu slidingMenu;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.content);
        setBehindContentView(R.layout.menu_frame);
        slidingMenu = getSlidingMenu();
        slidingMenu.setMode(SlidingMenu.LEFT);
        slidingMenu.setBehindOffsetRes(R.dimen.slidingmenu_offset);
        slidingMenu.setShadowDrawable(R.drawable.shadow);
        slidingMenu.setShadowWidthRes(R.dimen.shadow_width);
        slidingMenu.setTouchModeAbove(SlidingMenu.TOUCHMODE_FULLSCREEN);

        MenuFragment menuFragment = new MenuFragment();
        getSupportFragmentManager()
            .beginTransaction()
            .replace(R.id.menu, menuFragment, "MENU")
            .commit();
    }
}
```

2

编写 MenuFragment 类。

```
public class MenuFragment extends BaseFragment {

    @Override
    public View initView(LayoutInflater inflater) {
        TextView textView = new TextView(getActivity());
```

```

        textView.setText("左侧侧拉菜单");
        return textView;
    }

    @Override
    public void initData(Bundle savedInstanceState) {

    }
}

```

3

MenuFragment 继承自 BaseFragment，BaseFragment 代码清单如下：

```

public abstract class BaseFragment extends Fragment {
    public View view;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        view = initView(inflater);
        return view;
    }
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        initData(savedInstanceState);
        super.onActivityCreated(savedInstanceState);
    }

    public abstract View initView(LayoutInflater inflater);
    public abstract void initData(Bundle savedInstanceState);
}

```

4

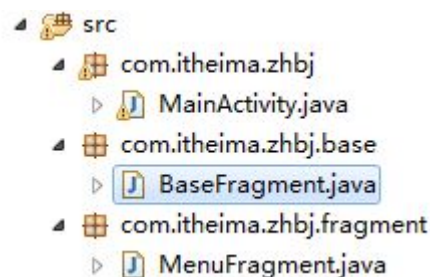
修改 AndroidManifest.xml 文件，修改应用主题为全屏显示

```

<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Light.NoTitleBar.Fullscreen" >
    <activity

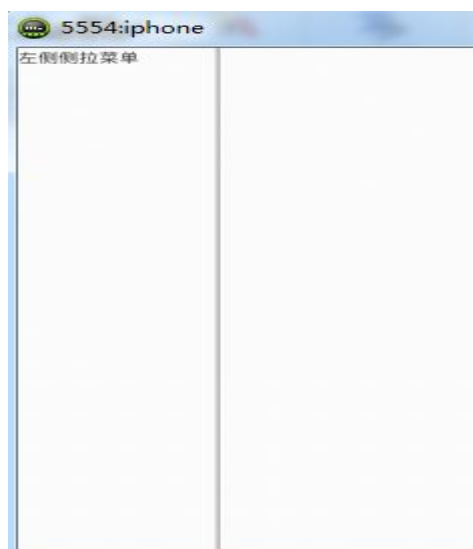
```

上面工程 src 工程目录结构如下图所示：



5

运行上面工程，效果如下：



## 4.智慧北京之 ViewPager 实现 tab 切换菜单(★★★★)

在该节中我们将实现智慧北京主界面以及 tab 切换效果。我们的整体思路是首先建立一个 HomeFragment，然后在 Fragment 中使用 ViewPager。

### 4.1 主界面框架的实现

1

在如下图包中创建 HomeFragment，继承 BaseFragment。



HomeFragment 代码清单如下：

```
public class HomeFragment extends BaseFragment {
    //使用 xUtils 中的 ViewUtils 进行对象的注入操作
    @ViewInject(R.id.layout_content)
    private ViewPager viewPager;
    @ViewInject(R.id.main_radio)
    private RadioGroup main_radio;
    private List<BasePager> pagersList;
    @Override
    public View initView(LayoutInflater inflater) {
        view = inflater.inflate(R.layout.frag_home, null);
        //注入是基于反射的原理实现的，首先告诉 ViewUtils 要反射的类和相应的 view 对象，这样才能完成 findViewById 操作
        ViewUtils.inject(this, view);
        return view;
    }

    @Override
    public void initData(Bundle savedInstanceState) {
        //给 RadioGroup 设置监听事件
        main_radio.setOnCheckedChangeListener(new OnCheckedChangeListener() {

            @Override
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                switch (checkedId) {
                    case R.id.rb_function:
                        //选中不同的 radioButton 时完成相应的 viewPager 的显示
                        viewPager.setCurrentItem(0);
                        break;
                    case R.id.rb_news_center:
                        viewPager.setCurrentItem(1);
                        break;
                    case R.id.rb_smart_service:
                        viewPager.setCurrentItem(2);

                        break;
                    case R.id.rb_gov_affairs:
                        viewPager.setCurrentItem(3);

                        break;
                    case R.id.rb_setting:
                        viewPager.setCurrentItem(4);

                        break;
                }
            }
        });
    }
}
```

```
        default:
            break;
    }
}
});
//给 RadioGroup 设置默认选中的对象为首页
main_radio.check(R.id.rb_function);
//创建一个集合，用于存储不同的页
pagersList = new ArrayList<BasePager>();
pagersList.add(new FunctionPager(getActivity()));
pagersList.add(new NewsCenterPager(getActivity()));
pagersList.add(new SmartServicePager(getActivity()));
pagersList.add(new GovAffairsPager(getActivity()));
pagersList.add(new SettingsPager(getActivity()));
//给 viewPager 设置适配器
viewPager.setAdapter(new MyAdapter());
}
class MyAdapter extends PagerAdapter{

    @Override
    public int getCount() {
        return pagersList.size();
    }

    @Override
    public boolean isViewFromObject(View arg0, Object arg1) {
        return arg0==arg1;
    }

    @Override
    public Object instantiateItem(ViewGroup container, int position) {
        View view = pagersList.get(position).getRootView();
        container.addView(view);
        return view;
    }

    @Override
    public void destroyItem(ViewGroup container, int position, Object object) {
        View view = pagersList.get(position).getRootView();
        container.removeView(view);
    }
}
```

上面的 HomeFragment 类用到了 frag\_home.xml 布局文件，其代码清单如下：

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

    <android.support.v4.view.ViewPager
        android:id="@+id/layout_content"
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:layout_weight="1.0" >
    </android.support.v4.view.ViewPager>

    <RadioGroup
        android:id="@+id/main_radio"
        android:layout_width="fill_parent"
        android:layout_height="60dp"
        android:layout_gravity="bottom"
        android:background="@drawable/bottom_tab_bg"
        android:gravity="center_vertical"
        android:orientation="horizontal"
        android:paddingTop="2dp" >

        <RadioButton
            android:id="@+id/rb_function"
            style="@style/main_tab_bottom"
            android:drawableTop="@drawable/icon_function"
            android:text="@string/tab_function" />

        <RadioButton
            android:id="@+id/rb_news_center"
            style="@style/main_tab_bottom"
            android:drawableTop="@drawable/icon_newscenter"
            android:text="@string/tab_news_center" />

        <RadioButton
            android:id="@+id/rb_smart_service"
            style="@style/main_tab_bottom"
            android:drawableTop="@drawable/icon_smartservice"
            android:text="@string/tab_smart_service" />
    </RadioGroup>
</LinearLayout>
```



```

<RadioButton
    android:id="@+id/rb_gov_affairs"
    style="@style/main_tab_bottom"
    android:drawableTop="@drawable/icon_govaffairs"
    android:text="@string/tab_gov_affairs" />

<RadioButton
    android:id="@+id/rb_setting"
    style="@style/main_tab_bottom"
    android:drawableTop="@drawable/icon_setting"
    android:text="@string/tab_setting" />
</RadioGroup>
</LinearLayout>

```

2

在 HomeFragment 中我们使用了 5 种 BasePager 类，分别是 FunctionPager、NewsCenterPager 等，这五种 Pager 继承了 BasePager，BasePager 代码清单如下：

```

public abstract class BasePager {
    protected Context context;
    protected View view;
    public BasePager(Context context){
        this.context = context;
        view = initView();
    }
    public View getRootView(){
        return view;
    }
    public abstract View initView();
    public abstract void initData();
}

```

5 种 Pager 功能都是类似了，因此这里只给出 FunctionPager 代码清单：

```

public class FunctionPager extends BasePager {
    public FunctionPager(Context context) {
        super(context);
    }

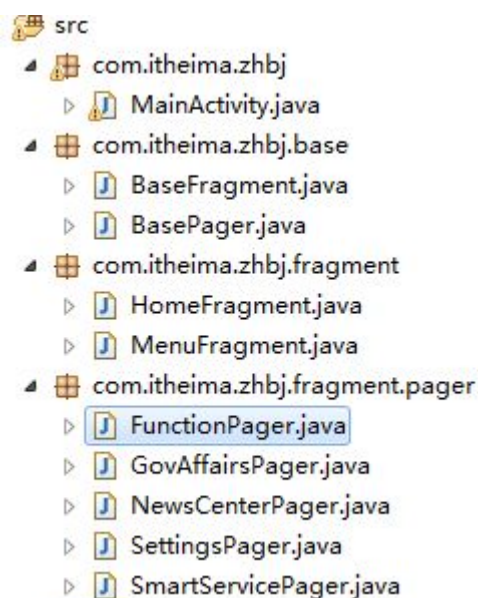
    @Override
    public View initView() {

```

```
        TextView textView = new TextView(context);
        textView.setText("首页");
        return textView;
    }

    @Override
    public void initData() {
        // TODO Auto-generated method stub
    }
}
```

此时工程的 src 目录结构为：



3

在我们的 MainActivity.java 中我们得将 HomeFragment 引入进来。

因为 MainActivity.java 之前已经创建，这里只需要在其 onCreate 方法中添加如下代码：

```
//引入主界面
HomeFragment homeFragment = new HomeFragment();
getSupportFragmentManager()
    .beginTransaction()
    .replace(R.id.content_frame, homeFragment, "HOME")
    .commit();
```

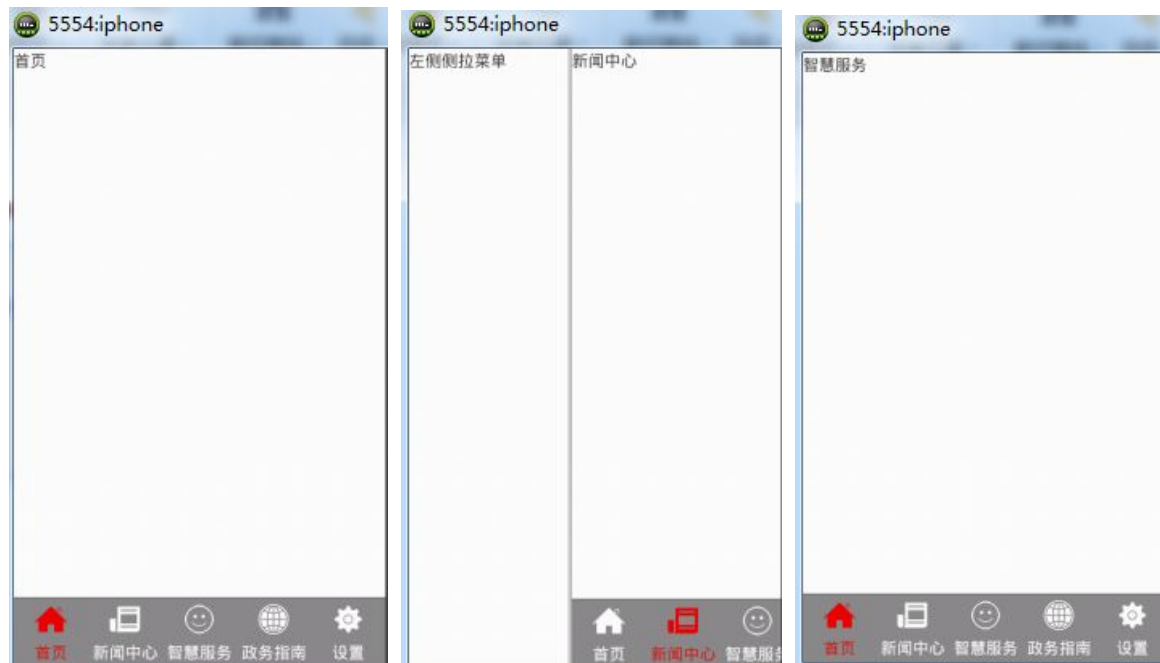
上面清单中的 R.id.content\_frame 是 MainActivity 布局文件中 FrameLayout 的 id。

MainActivity 的布局文件 content.xml 文件清单如下：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/content_frame"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
</FrameLayout>
```

4

运行上面工程到模拟器，效果如下：



经过使用我们发现上面的效果有些不尽人意的地方，当我们按住屏幕往左滑动的时候我们的 ViewPager 切换了，但是 RadioGroup 依然停留在初始位置，这是 ViewPager 的预加载机制导致的，为了解决这个问题我们先介绍 Android 中控件的触摸分发机制，然后再修改我们的代码。

## 4.2 Android 触摸事件分发机制

Android 中的事件分为按键事件和触摸事件，这里我们对触摸事件进行阐述。

Touch 事件是由一个 ACTION\_DOWN，n 个 ACTION\_MOVE，一个 ACTION\_UP 组成 onClick，onLongClick，onScroll 等事件。Android 中的控件都是继承 View 这个基类的，而控件分为两种：一种是继承 View 不能包含其他控件的控件；一种是继承 ViewGroup 可以包含其他控件的控件，暂且称为容器控件，比如 ListView，GridView，LinearLayout 等。

这里先对几个函数讲解下。

❖ **public boolean dispatchTouchEvent (MotionEvent ev)**   这个方法分发 TouchEvent

❖ **public boolean onInterceptTouchEvent(MotionEvent ev)** 这个方法拦截 TouchEvent

❖ **public boolean onTouchEvent(MotionEvent ev)**       这个方法处理 TouchEvent

其中 view 类中有 dispatchTouchEvent 和 onTouchEvent 两个方法，ViewGroup 继承 View，而且还新添了一个 onInterceptTouchEvent 方法。Activity 中无 onInterceptTouchEvent 方法，但有另外两种方法。我们可以发现上面 3 个方法都是返回 boolean，那各代表什么意思呢？

❖ **public boolean dispatchTouchEvent (MotionEvent ev)**

**Activity** 中解释：

Called to process touch screen events.You can override this to intercept all touch screen events before they are dispatched to the window. Be sure to call this implementation for touch screen events that should be handled normally.

#### Parameters

ev	The touch screen event.
----	-------------------------

#### Returns

· ❖ boolean Return true if this event was consumed.

它会被调用处理触摸屏事件，可以重写覆盖此方法来拦截所有触摸屏事件在这些事件分发到窗口之前。通常应该处理触摸屏事件，一定要调用这个实现。当返回值为 true 时，表示这个事件已经被消费了。

❖ **public boolean onInterceptTouchEvent (MotionEvent ev)**

Implement this method to intercept all touch screen motion events. This allows you to watch events as they are dispatched to your children, and take ownership of the current gesture at any point.

Using this function takes some care, as it has a fairly complicated interaction

with [View.onTouchEvent\(MotionEvent\)](#), and using it requires implementing that method as well as this one in the correct way. Events will be received in the following order:

1. You will receive the down event here.
2. The down event will be handled either by a child of this viewgroup, or given to your own `onTouchEvent()` method to handle; this means you should implement `onTouchEvent()` to return true, so you will continue to see the rest of the gesture (instead of looking for a parent view to handle it). Also, by returning true from `onTouchEvent()`, you will not receive any following events in `onInterceptTouchEvent()` and all touch processing must happen in `onTouchEvent()` like normal.
3. For as long as you return false from this function, each following event (up to and including the final up) will be delivered first here and then to the target's `onTouchEvent()`.
4. If you return true from here, you will not receive any following events: the target view will receive the same event but with the action [ACTION\\_CANCEL](#), and all further events will be delivered to your `onTouchEvent()` method and no longer appear here.

#### Parameters

<i>ev</i>	The motion event being dispatched down the hierarchy.
-----------	---

#### Returns

· Return true to steal motion events from the children and have them dispatched to this ViewGroup through onTouchEvent(). The current target will receive an ACTION\_CANCEL event, and no further messages will be delivered here.

基本意思就是：

1. ACTION\_DOWN 首先会传递到 onInterceptTouchEvent()方法

2.如果该 ViewGroup 的 onInterceptTouchEvent()在接收到 down 事件处理完成之后 return false 那么后续的 move, up 等事件将继续会先传递给该 ViewGroup，之后才和 down 事件一样传递给最终的目标 view 的 onTouchEvent() 处理。

3.如果该 ViewGroup 的 onInterceptTouchEvent()在接收到 down 事件处理完成之后 return true 那么后续的 move, up 等事件将不再传递给 onInterceptTouchEvent() 而是和 down 事件一样传递给该 ViewGroup 的 onTouchEvent() 处理，注意，目标 view 将接收不到任何事件。

4.如果最终需要处理事件的 view 的 onTouchEvent()返回了 false，那么该事件将被传递至其上一层次的 view 的 onTouchEvent()处理。

5.如果最终需要处理事件的 view 的 onTouchEvent()返回了 true，那么后续事件将可以继续传递给该 view 的 onTouchEvent()处理。

### **Android touch 事件传递机制：**

我们可以看看 android 源代码：

Activity.java 中



```
public boolean dispatchTouchEvent(MotionEvent ev) {  
    if (ev.getAction() == MotionEvent.ACTION_DOWN) {  
        onUserInteraction();  
    }  
    if (getWindow().superDispatchTouchEvent(ev)) {  
        return true;  
    }  
    return onTouchEvent(ev);  
}
```

暂且不管 onUserInteraction 方法因为它只是一个空方法如果你没实现的话。

getWindow().superDispatchTouchEvent(ev)。其中 getWindow()返回的是 PhoneWindow。

PhoneWindow.java:

```
public boolean superDispatchTouchEvent(MotionEvent event) {  
    return super.dispatchTouchEvent(event);  
}
```

此函数调用 super.dispatchTouchEvent(event),Activity 的 rootview 是 PhoneWindow.DecorView,它继承

FrameLayout。通过 super.dispatchTouchEvent 把 touch 事件派发给各个 Activity 的是子 view。同时我可以看到，

如果子 view 拦截了事件，则不会执行 onTouchEvent 函数。

ViewGroup.java 中 dispatchTouchEvent 方法：

由于代码过长这里就不贴出来了，但也知道它返回的是

return target.dispatchTouchEvent(ev);

这里 target 指的是所分发的目标，可以是它本身，也可以是它的子 View。

ViewGroup.java 中的 onInterceptTouchEvent 方法：

```
public boolean onInterceptTouchEvent(MotionEvent ev) {  
    return false;  
}
```

默认情况下返回 false，即不拦截 touch 事件。

View.java 中的 dispatchTouchEvent 方法

```
/**
 * Pass the touch screen motion event down to the target view, or this
 * view if it is the target.
 *
 * @param event The motion event to be dispatched.
 * @return True if the event was handled by the view, false otherwise.
 */
public boolean dispatchTouchEvent(MotionEvent event) {
    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onTouchEvent(event, 0);
    }

    if (onFilterTouchEventForSecurity(event)) {
        //noinspection SimplifiableIfStatement
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnTouchListener != null && (mViewFlags & ENABLED_MASK)
== ENABLED
            && li.mOnTouchListener.onTouch(this, event)) {
            return true;
        }

        if (onTouchEvent(event)) {
            return true;
        }
    }

    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);
    }
    return false;
}
```

这里我们很清楚可以知道如果 if 条件不成立则 dispatchTouchEvent 的返回值是 onTouchEvent 的返回值

View.java 中的 onTouchEvent 方法

```
/**
 * Implement this method to handle touch screen motion events.
 *
 * @param event The motion event.
 * @return True if the event was handled, false otherwise.
 */
public boolean onTouchEvent(MotionEvent event) {
    final int viewFlags = mViewFlags;

    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (event.getAction() == MotionEvent.ACTION_UP && (mPrivateFlags &
PFLAG_PRESSED) != 0) {
            setPressed(false);
        }
        // A disabled view that is clickable still consumes the touch
        // events, it just doesn't respond to them.
        return (((viewFlags & CLICKABLE) == CLICKABLE ||
            (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE));
    }

    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }

    if (((viewFlags & CLICKABLE) == CLICKABLE ||
        (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
        switch (event.getAction()) {
            case MotionEvent.ACTION_UP:
                boolean prepressed = (mPrivateFlags & PFLAG_PREPRESSED) != 0;
                if ((mPrivateFlags & PFLAG_PRESSED) != 0 || prepressed) {
                    // take focus if we don't have it already and we should in
                    // touch mode.
                    boolean focusTaken = false;
                    if (isFocusable() && isFocusableInTouchMode() && !isFocused())
{
                        focusTaken = requestFocus();
                    }

                    if (prepressed) {
                        // The button is being released before we actually
                        // showed it as pressed. Make it show the pressed
```

state

```
        // state now (before scheduling the click) to ensure
        // the user sees it.
        setPressed(true);
    }

    if (!mHasPerformedLongPress) {
        // This is a tap, so remove the longpress check
        removeLongPressCallback();

        // Only perform take click actions if we were in the pressed

        if (!focusTaken) {
            // Use a Runnable and post this rather than calling
            // performClick directly. This lets other visual state
            // of the view update before click actions start.
            if (mPerformClick == null) {
                mPerformClick = new PerformClick();
            }
            if (!post(mPerformClick)) {
                performClick();
            }
        }
    }

    if (mUnsetPressedState == null) {
        mUnsetPressedState = new UnsetPressedState();
    }

    if (prepressed) {
        postDelayed(mUnsetPressedState,
            ViewConfiguration.getPressedStateDuration());
    } else if (!post(mUnsetPressedState)) {
        // If the post failed, unpress right now
        mUnsetPressedState.run();
    }
    removeTapCallback();
}
break;

case MotionEvent.ACTION_DOWN:
    mHasPerformedLongPress = false;

    if (performButtonActionOnTouchDown(event)) {
```

```
        break;
    }

    // Walk up the hierarchy to determine if we're inside a scrolling
container.
    boolean isInScrollingContainer = isInScrollingContainer();

    // For views inside a scrolling container, delay the pressed
feedback for
    // a short period in case this is a scroll.
    if (isInScrollingContainer) {
        mPrivateFlags |= PFLAG_PREPRESSED;
        if (mPendingCheckForTap == null) {
            mPendingCheckForTap = new CheckForTap();
        }
        postDelayed(mPendingCheckForTap,
ViewConfiguration.getTapTimeout());
    } else {
        // Not inside a scrolling container, so show the feedback right
away

        setPressed(true);
        checkForLongClick(0);
    }
    break;

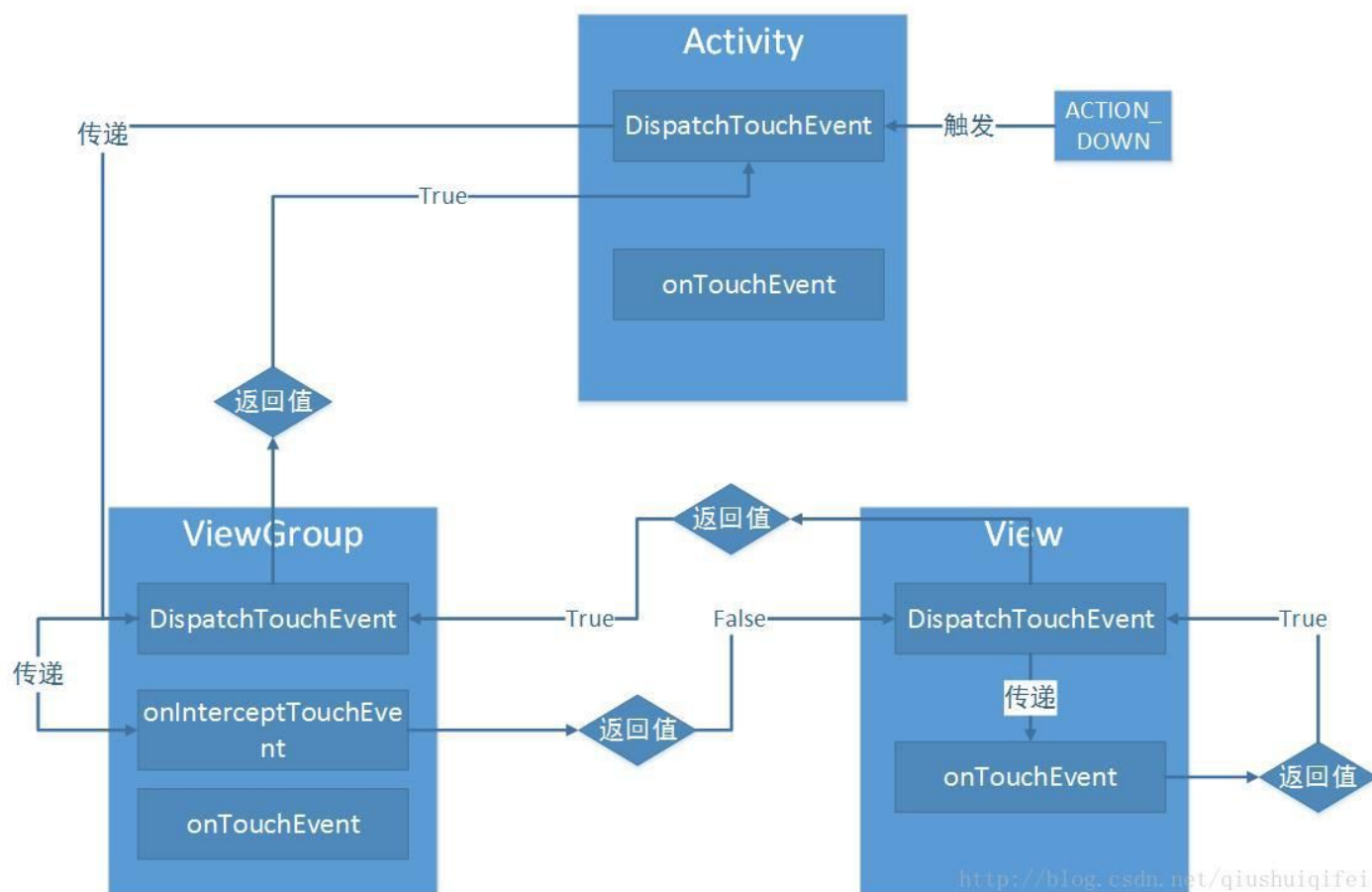
case MotionEvent.ACTION_CANCEL:
    setPressed(false);
    removeTapCallback();
    removeLongPressCallback();
    break;

case MotionEvent.ACTION_MOVE:
    final int x = (int) event.getX();
    final int y = (int) event.getY();

    // Be lenient about moving outside of buttons
    if (!pointInView(x, y, mTouchSlop)) {
        // Outside button
        removeTapCallback();
        if ((mPrivateFlags & PFLAG_PRESSED) != 0) {
            // Remove any future long press/tap checks
            removeLongPressCallback();
        }
    }
}
```

```
        setPressed(false);
    }
    }
    break;
}
return true;
}
return false;
}
```

所以很容易得到触摸事件默认处理流程（以 ACTION\_DOWN 事件为例）：



当触摸事件 ACTION\_DOWN 发生之后，先调用 Activity 中的 dispatchTouchEvent 函数进行处理，紧接着 ACTION\_DOWN 事件传递给 ViewGroup 中的 dispatchTouchEvent 函数，接着 ViewGroup 中的 dispatchTouchEvent 中的 ACTION\_DOWN 事件传递到调用 ViewGroup 中的 onInterceptTouchEvent 函数，此函数负责拦截 ACTION\_DOWN 事件。由于 ViewGroup 下还包含子 View，所以默认返回值为 false，即不拦截此



ACTION\_DOWN 事件。如果返回 false 则 ACTION\_DOWN 事件继续传递给其子 view。由于子 view 不是 ViewGroup 的控件，所以 ACTION\_DOWN 事件接着传递到 onTouchEvent 进行处理事件。此时消息的传递基本上结束。从上可以分析，motionEvent 事件的传递是采用隧道方式传递。隧道方式，即从根元素依次往下传递直到最内层子元素或在中间某一元素中由于某一条件停止传递。

接下来继续分析，事件的处理。刚才 ACTION\_DOWN 事件传递到 view 的 onTouchEvent 函数中处理了，默认是返回 true，接着 view 的 dispatchTouchEvent 返回 true，再接着 ViewGroup 的 dispatchTouchEvent 返回 true，最后 Activity 的 dispatchTouchEvent 返回 true。我们发现，motionEvent 事件的处理采用冒泡方式。冒泡方式，从最内层子元素依次往外传递直到根元素或在中间某一元素中由于某一条件停止传递。

下图为程序调试结果：

ACTION\_DOWN 事件输出：

com.qyl.dragdemo	ActivitydispatchTouchEvent	Actiondown
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	Actiondown
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	Actiondown
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	false
com.qyl.dragdemo	DragIcondispatchTouchEvent	Actiondown
com.qyl.dragdemo	DragIconOnTouchEvent	ACTION_DOWN
com.qyl.dragdemo	DragIconOnTouchEvent	true
com.qyl.dragdemo	DragIcondispatchTouchEvent	true
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	true
com.qyl.dragdemo	ActivitydispatchTouchEvent	true

ACTION\_MOVE 事件输出：

com.qyl.dragdemo	ActivitydispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	false
com.qyl.dragdemo	DragIcondispatchTouchEvent	Actionmove
com.qyl.dragdemo	DragIconOnTouchEvent	ACTION_MOVE
com.qyl.dragdemo	DragIconOnTouchEvent	true
com.qyl.dragdemo	DragIcondispatchTouchEvent	true
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	true
com.qyl.dragdemo	ActivitydispatchTouchEvent	true

现在我们来做一些改变，就接着以 ACTION\_DOWN 为例

### 情况一：

我们在 View 中 onTouchEvent 中 ACTION\_DOWN 返回 **false**，输出结果如下：

```
com.qyl.dragdemo ActivitydispatchTouchEvent Actiondown
com.qyl.dragdemo LinearLayoutdispatchTouchEvent Actiondown
com.qyl.dragdemo LinearLayoutonInterceptTouchEvent Actiondown
com.qyl.dragdemo LinearLayoutonInterceptTouchEvent false
com.qyl.dragdemo DragIcondispatchTouchEvent Actiondown
com.qyl.dragdemo DragIconOnTouchEvent ACTION_DOWN
com.qyl.dragdemo DragIcondispatchTouchEvent false
com.qyl.dragdemo LinearLayoutonTouchEvent Actiondown
com.qyl.dragdemo LinearLayoutonTouchEvent true
com.qyl.dragdemo LinearLayoutdispatchTouchEvent true
com.qyl.dragdemo ActivitydispatchTouchEvent true
com.qyl.dragdemo ActivitydispatchTouchEvent Actionmove
com.qyl.dragdemo LinearLayoutdispatchTouchEvent Actionmove
com.qyl.dragdemo LinearLayoutonTouchEvent Actionmove
com.qyl.dragdemo LinearLayoutonTouchEvent true
com.qyl.dragdemo LinearLayoutdispatchTouchEvent true
com.qyl.dragdemo ActivitydispatchTouchEvent true
```

可以发现 ACTION\_DOWN 事件传递到上层的 ViewGroup 的 onTouchEvent，同时返回 true，说明事件被 ViewGroup 消费了。同时之后的 touch 事件（ACTION\_MOVE 等）不再传递给 view，只传递到 ViewGroup，由 ViewGroup 的 onTouchEvent 函数处理 touch 事件。同时 onInterceptTouchEvent 也不再调用。

### 情况二：

我们在 View 中 onTouchEvent 中 ACTION\_MOVE 返回 **false**，输出结果如下：

com.qyl.dragdemo	ActivitydispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	false
com.qyl.dragdemo	DragIcondispatchTouchEvent	Actionmove
com.qyl.dragdemo	DragIconOnTouchEvent	ACTION_MOVE
com.qyl.dragdemo	DragIcondispatchTouchEvent	false
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	false
com.qyl.dragdemo	ActivityonTouchEvent	Actionmove
com.qyl.dragdemo	ActivityonTouchEvent	false
com.qyl.dragdemo	ActivitydispatchTouchEvent	false

由于 view 未消费此 ACTION\_MOVE 事件，按照原理来说应该是将事件处理冒泡到 ViewGroup 去处理，但结果却是 Activity 处理的。我们知道，触摸事件首先发生的就是 ACTION\_DOWN 事件，我们在 onInterceptTouchEvent 所解释就可以发现 ACTION\_DOWN 与 ACTION\_MOVE 等事件有区别，ACTION\_DOWN 事件作为起始事件，它的重要性是要超过 ACTION\_MOVE 和 ACTION\_UP 的，如果发生了 ACTION\_MOVE 或者 ACTION\_UP，那么一定曾经发生了 ACTION\_DOWN。也就是说 ACTION\_DOWN 事件被 view 消费了，而 ACTION\_MOVE 事件没被消费，传递到 ViewGroup，由于之前 ViewGroup 没处理 ACTION\_DOWN 事件，所以它也不处理 ACTION\_MOVE。但 Activity 却不一样，它可以接受所有事件。

### 情况三：

这次在 ViewGroup 中的 onInterceptTouchEvent 中 ACTION\_DOWN 返回 **true**

结果如下：

com.qyl.dragdemo	ActivitydispatchTouchEvent	Actiondown
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	Actiondown
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	Actiondown
com.qyl.dragdemo	LinearLayoutonTouchEvent	Actiondown
com.qyl.dragdemo	LinearLayoutonTouchEvent	true
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	true
com.qyl.dragdemo	ActivitydispatchTouchEvent	true

它直接把事件发送给 ViewGroup 的 onTouchEvent 处理，此后不再拦截事件直接到 ViewGroup 中的 onTouchEvent 处理。



**情况四：**

在 ViewGroup 中的 onInterceptTouchEvent 中 ACTION\_MOVE 返回 **true**

结果如下：

com.qyl.dragdemo	ActivitydispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutonInterceptTouchEvent	Actionmove
com.qyl.dragdemo	DragIcondispatchTouchEvent	Action_CANCEL
com.qyl.dragdemo	DragIconOnTouchEvent	ACTION_CANCEL
com.qyl.dragdemo	DragIconOnTouchEvent	true
com.qyl.dragdemo	DragIcondispatchTouchEvent	true
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	true
com.qyl.dragdemo	ActivitydispatchTouchEvent	true
com.qyl.dragdemo	ActivitydispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutonTouchEvent	Actionmove
com.qyl.dragdemo	LinearLayoutonTouchEvent	true
com.qyl.dragdemo	LinearLayoutdispatchTouchEvent	true
com.qyl.dragdemo	ActivitydispatchTouchEvent	true

ACTION\_MOVE 被 ViewGroup 拦截了，上次处理 ACTION\_DOWN 的 view 则会收到 ACTION\_CANCEL 事件，之后 ViewGroup 不再拦截后续事件，事件直接在 ViewGroup 中的 onTouchEvent 处理。

还有很多情况，这里不一一列出了。

我发现重写了 onTouchEvent 函数就无法获取 onClick 和 onLongClick 事件。接下来讨论当重写了 onTouchEvent，android 是如何区分 onClick，onLongClick 事件的。搞清楚此问题对于如何响应 UI 各种事件是很重要的，例如类似 android 桌面的应用程序图标，可以点击，然后长按拖动。

Android 中 onclick，onLongClick 都是由 ACTION\_DOWN，ACTION\_UP 组成。如果在同一个 View 中 onTouchEvent、onclick、onLongClick 都进行了重写。onTouchEvent 最先捕获 ACTION\_DOWN、ACTION\_UP 等单元事件。接下来才可能发生 onClick、onLongClick 事件。一个 onclick 事件是由 ACTION\_DOWN 和 ACTION\_UP 组成的。一个 onLongClick 事件至少有一个 ACTION\_DOWN。那 android 具体是怎么实现的呢，可以看源代码：

View.java 中：

上面我已经展示了 onTouchEvent 方法，但由于过长我折叠了一部分代码，现在展开

```
if (((viewFlags & CLICKABLE) == CLICKABLE ||
    (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
```

这个 if 条件内执行就是 click 事件处理及 longClick 事件处理。先看 ACTION\_DOWN 事件

```
case MotionEvent.ACTION_DOWN:
    if (mPendingCheckForTap == null) {
        mPendingCheckForTap = new CheckForTap();
    }
    mPrivateFlags |= PREPRESSED;
    mHasPerformedLongPress = false;
    postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());
    break;
```

我们看到有个 postDelayed 方法，此方法意思为延时把线程插入到消息队列。即 ACTION\_DOWN 后触发一个 postDelayed 方法。mPendingCheckForTap 属于 CheckForTap 的实例。

```
private final class CheckForTap implements Runnable {
    public void run() {
        mPrivateFlags &= ~PREPRESSED;
        mPrivateFlags |= PRESSED;
        refreshDrawableState();
        if ((mViewFlags & LONG_CLICKABLE) == LONG_CLICKABLE) {
            postCheckForLongClick(ViewConfiguration.getTapTimeout());
        }
    }
}
```

在里面开启一个线程当为 LONG\_CLICKABLE，调用 postCheckForLongClick 方法。

```
private void postCheckForLongClick(int delayOffset) {
    mHasPerformedLongPress = false;

    if (mPendingCheckForLongPress == null) {
        mPendingCheckForLongPress = new CheckForLongPress();
    }
    mPendingCheckForLongPress.rememberWindowAttachCount();
    postDelayed(mPendingCheckForLongPress,
        ViewConfiguration.getLongPressTimeout() - delayOffset);
}
```

再看 mPendingCheckForLongPress 这个线程。

```
class CheckForLongPress implements Runnable {  
  
    private int mOriginalWindowAttachCount;  
  
    public void run() {  
        if (isPressed() && (mParent != null)  
            && mOriginalWindowAttachCount == mWindowAttachCount) {  
            if (performLongClick()) {  
                mHasPerformedLongPress = true;  
            }  
        }  
    }  
  
    public void rememberWindowAttachCount() {  
        mOriginalWindowAttachCount = mWindowAttachCount;  
    }  
}
```

<http://blog.csdn.net/qiushuiqifei>

当上面一系列条件全都符合的情况就调用 performLongClick 方法。

```
/**  
 * Call this view's OnLongClickListener, if it is defined. Invokes the context menu  
 * if the OnLongClickListener did not consume the event.  
 *  
 * @return True there was an assigned OnLongClickListener that was called, false  
 * otherwise is returned.  
 */  
public boolean performLongClick() {  
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_LONG_CLICKED);  
  
    boolean handled = false;  
    if (mOnLongClickListener != null) {  
        handled = mOnLongClickListener.onLongClick(View.this);  
    }  
    if (!handled) {  
        handled = showContextMenu();  
    }  
    if (handled) {  
        performHapticFeedback(HapticFeedbackConstants.LONG_PRESS);  
    }  
    return handled;  
}
```

<http://blog.csdn.net/qiushuiqifei>

此方法就调用我们熟悉的 onLongClick 函数。



至此 onLongClick 事件已经分析完。紧接着看 ACTION\_UP 事件

```
case MotionEvent.ACTION_UP:
    boolean prepressed = (mPrivateFlags & PREPRESSED) != 0;
    if ((mPrivateFlags & PRESSED) != 0 || prepressed) {
        // take focus if we don't have it already and we should in
        // touch mode.
        boolean focusTaken = false;
        if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
            focusTaken = requestFocus();
        }

        if (!mHasPerformedLongPress) {
            // This is a tap, so remove the longpress check
            removeLongPressCallback();

            // Only perform take click actions if we were in the pressed state
            if (!focusTaken) {
                // Use a Runnable and post this rather than calling
                // performClick directly. This lets other visual state
                // of the view update before click actions start.
                if (mPerformClick == null) {
                    mPerformClick = new PerformClick();
                }
                if (!post(mPerformClick)) {
                    performClick();
                }
            }
        }
    }
}
```

<http://blog.csdn.net/qiushuiqifei>

直接关注 performClick 函数：

```
public boolean performClick() {
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);

    if (mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        mOnClickListener.onClick(this);
        return true;
    }

    return false;
}
```

<http://blog.csdn.net/qiushuiqifei>

这里我们同样看到了我们熟悉的 onClick 方法。



所以 android 这种机制是保证了此 onClick 和 onLongClick 能与 onTouchEvent 并存。接下来考虑 onClick 与 onLongClick 是否并存，其实这个问题前面已经阐述了。只要此事件没被消费，它还会接着传递下去。从上面知道 onLongClick 是在单独的线程执行，发生在 ACTION\_UP 之前。OnClick 发生在 ACTION\_UP 之后，也就是说，如果在 onLongClick 返回 false，onClick 就会发生，而 onLongClick 返回 true，则代表此事件已经被消费。OnClick 不再发生。

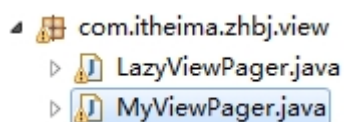
如果多次设置 onclick 事件，则最顶层的 onclick 覆盖掉底层 onclick 事件；多次设置 onLongClick 事件，则只执行底层 view 的 onLongClick 方法。当 ACTION\_DOWN 调用之后返回 false。

可以看到 ACTION\_DOWN 被消费了，所以不会让上层处理了。

## 4.3 主界面框架的改进

在 4.2 节我们介绍了 Android 的事件分发机制，那么这一节我们将对我们在 4.1 节创建的架构做改进。改进主要有两个方面，一是解决 ViewPager 的预加载问题，而是解决滑动事件切换 ViewPager 的问题。

我们在 src 目录下创建 com.itheima.zhbj.view 包，如下图所示。



在该包下引入第三方的 LazyViewPager.java 类，该类通过 `DEFAULT_OFFSCREEN_PAGES` 参数来配置 ViewPager 预加载的页数，我们将其设置为 0，则代表不预加载页面。同时 LazyViewPager 其实是一个自定义的类继承了 ViewGroup 类。

```
public class LazyViewPager extends ViewGroup {  
    private static final String TAG = "LazyViewPager";  
    private static final boolean DEBUG = false;  
  
    private static final boolean USE_CACHE = false;  
  
    private static final int DEFAULT_OFFSCREEN_PAGES = 0; //定义默认预加载个数
```

LazyViewPager 下载地址：<http://pan.baidu.com/s/1pJv9gQR>

有了 LazyViewPager 以后我们还需要自定义 MyViewPager 继承 LazyViewPager，同时覆写父类的 onInterceptTouchEvent 方法,在该方法中返回 false，将时间传播给终止掉。

MyViewPager 代码清单如下：

```
public class MyViewPager extends LazyViewPager {

    public MyViewPager(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public MyViewPager(Context context) {
        super(context);
    }

    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        //向下传递
        return false;
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) { //直接 return 不管是 true 还是 false 的话 都不能滑动
        return false;
    }

}
```

将 HomeFragment.java 中使用了 ViewPager 修改为自定义的 MyViewPager 类，同时修改其布局文件 frag\_home.xml 中 ViewPager 控件。

frag\_home.xml 修改内容如下：

```
<!-- <android.support.v4.view.ViewPager
    android:id="@+id/layout_content"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1.0" >
</android.support.v4.view.ViewPager> -->
<com.itheima.zhbj.view.MyViewPager
    android:id="@+id/layout_content"
```

```
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:layout_weight="1.0" >
    </com.itheima.zhbj.view.MyViewPager>
```

HomeFragment.java 修改内容如下：

```
//使用 xUtils 中的 ViewUtils 进行对象的注入操作
// @ViewInject(R.id.layout_content)
// private ViewPager viewPager;
// @ViewInject(R.id.layout_content)
private MyViewPager myViewPager;

//                viewPager.setCurrentItem(0);
//                myViewPager.setCurrentItem(0);

//给 viewPager 设置适配器
//    viewPager.setAdapter(new MyAdapter());
//    myViewPager.setAdapter(new MyAdapter());
```

修改好以上几处代码和布局文件后，再次运行该项目于模拟器，发现已经成功解决了预加载和滑动事件被激发的问题。

## 5.请求网络 (★★)

### 5.1 客户端请求网络

我们应用的布局已经完成，接下来当软件启动的时候应该初始化数据，这些数据主要是通过访问网络来实现的。

我们修改 HomeFragment.java，给自定义的 ViewPager 对象设置监听事件，当切换到不同的 tab 菜单时请求相应的网络获取数据。在 onPageSelected 方法中调用 BasePager 的 initData ( ) 方法。

```
myViewPager.setOnPageChangeListener(new OnPageChangeListener() {
    public void onPageSelected(int position) {
        BasePager basePager = pagersList.get(position);
        basePager.initData();
    }
    public void onPageScrolled(int position, float positionOffset, int
```

```
ositionOffsetPixels) {  
    // TODO Auto-generated method stub  
  
}  
  
@Override  
public void onPageScrollStateChanged(int state) {  
    // TODO Auto-generated method stub  
  
}  
});  
//默认进来时新闻中心  
BasePager basePager = pagersList.get(1);  
basePager.initData();
```

软件进入主界面的时候默认加载新闻中心数据，因此我们需要直接调用新闻中心 Pager 的 initData 方法。

NewsCenter.java 代码需要覆写 initData 方法，在该方法中实现网络的请求过程，initData 方法实现如下：

```
@Override  
public void initData() {  
    HttpUtils httpUtils = new HttpUtils(5000);  
    httpUtils.send(HttpMethod.GET, HMApi.NEWS_CENTER_CATEGORIES, null, new  
RequestCallBack<String>() {  
  
        @Override  
        public void onSuccess(ResponseInfo<String> responseInfo) {  
            Log.i(TAG, responseInfo.result);  
            Gson gson = new Gson();  
            NewCenter newCenter = gson.fromJson(responseInfo.result,  
NewCenter.class);  
            //TODO:  
        }  
    }  
}
```

**Tips**：上面用到了一个用于处理 JSON 的开源框架，GSON。

Gson ( 又称 Google Gson ) 是 Google 公司发布的一个开放源代码的 Java 库，主要用途为序列化 Java 对象为 JSON 字符串，或反序列化 JSON 字符串成 Java 对象。

Gson 下载地址：<http://pan.baidu.com/s/16jbUu>

**Tips**：我们将请求的 URL 地址封装在一个类中，HMApi，其代码清单如下：

```
public class HMApi {  
    public static final String BASE_URL = "http://10.0.2.2:8080/qbc";  
    // public static final String BASE_URL = "http://192.168.56.1:8080/qbc";  
    public final static String NEWS_CENTER_CATEGORIES = BASE_URL + "/categories.json";  
}
```

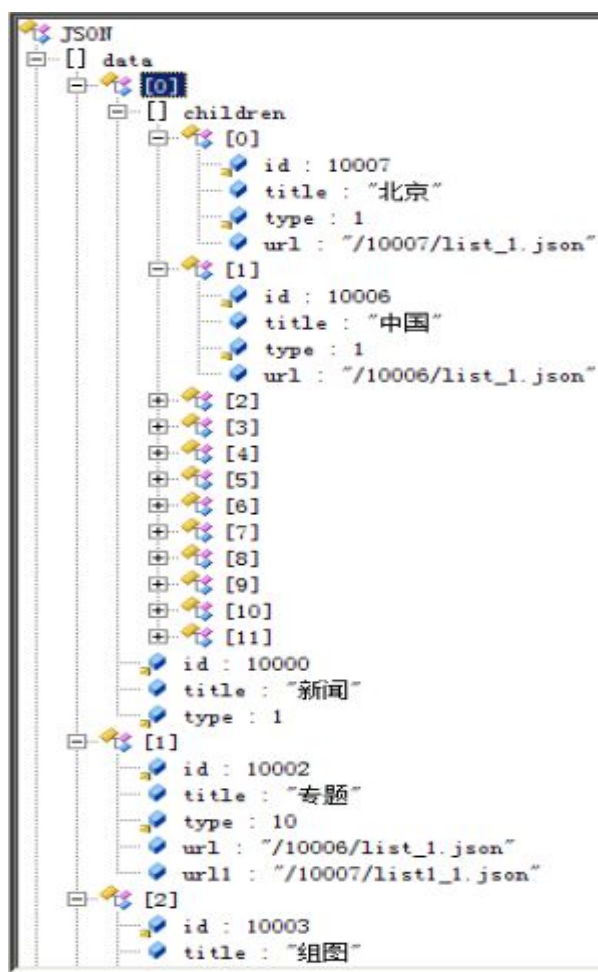
上面地址 10.0.2.2 是 Android 内置的用于模拟器访问本地服务器的地址。

**Tips**：在 Gson 的使用中有如下代码：

```
NewCenter newCenter = gson.fromJson(responseInfo.result, NewCenter.class);
```

其中 responseInfo.result 是服务器返回的 json 字符串，NewCenter 是我们自定义的一个 javaBean，这就代码的意思就是将 json 字符串转换成 NewCenter 对象，我们必须保证 NewCenter 的属性必须跟 JSON 的属性一致。

请求 <http://10.0.2.2:8080/qbc/categories.json> 返回的 json 数据格式如下图所示：



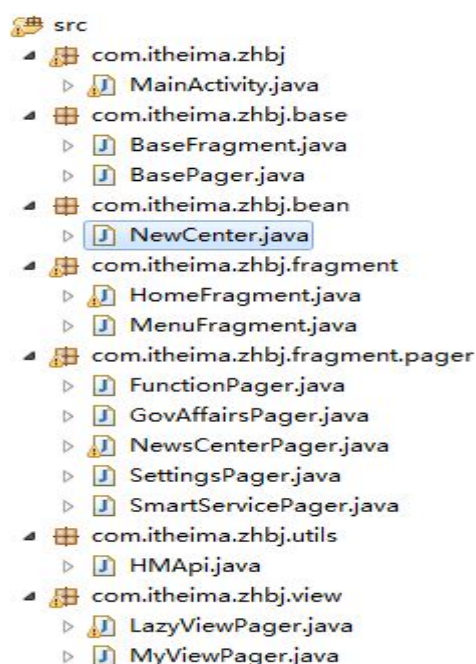
NewCenter.java 代码清单如下所示：

```
public class NewCenter {
    public List<NewCenterItem> data;
    public List<String> extend;
    public String retcode;

    public class NewCenterItem {
        public List<Children> children;
        public String id;
        public String title;
        public String type;
        public String url;
        public String url1;
        public String dayurl;
        public String excurl;
        public String weekurl;
    }

    public class Children {
        public String id;
        public String title;
        public String type;
        public String url;
    }
}
```

上面步骤完成后的 src 目录结构如下：

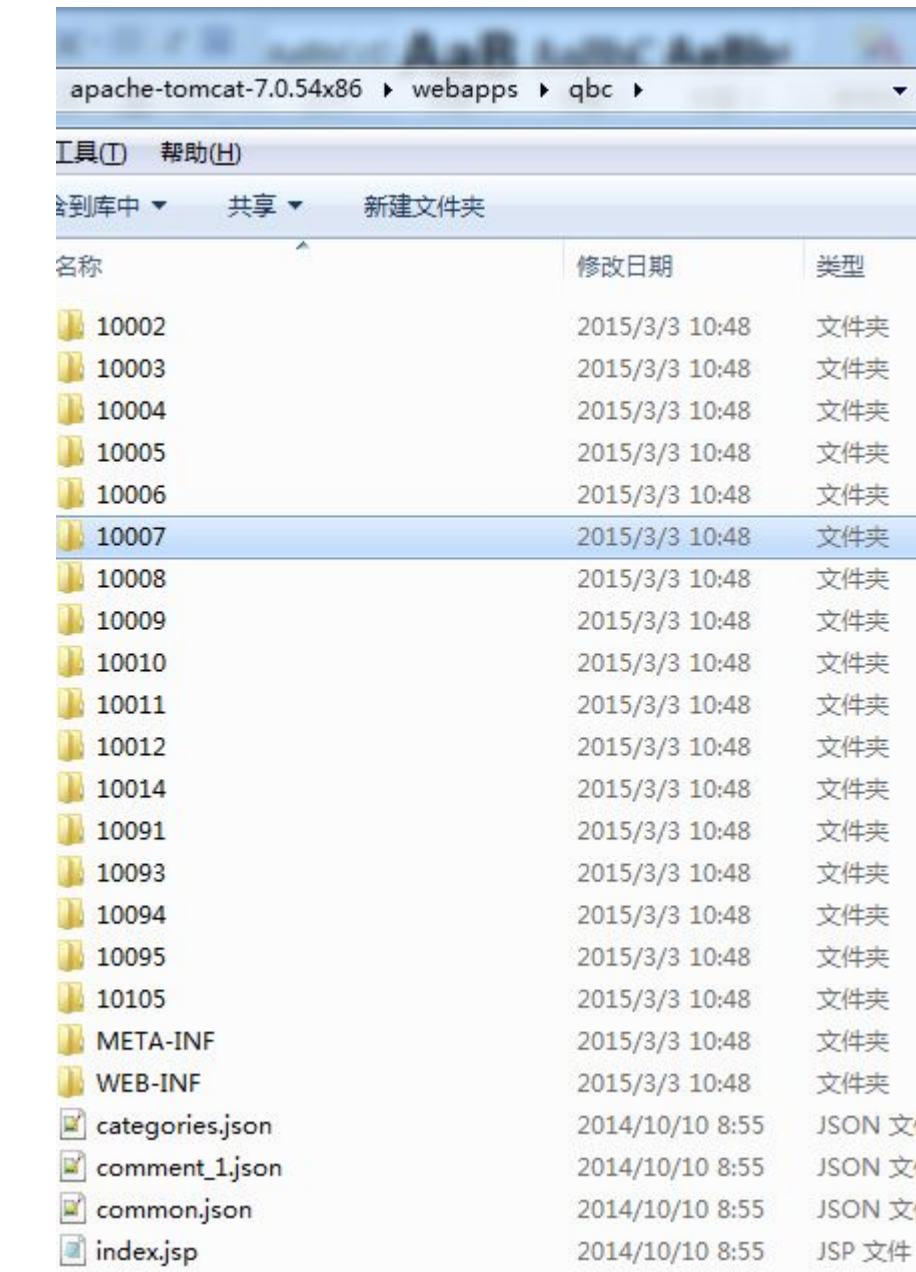




## 5.2 服务端的搭建







服务端工程下载地址：<http://pan.baidu.com/s/1o6l1PRO>

将下载好的数据文件夹放到 tomcat 下的 webapps 目录下，作为一个 web 工程发布，本人数据在 tomcat 中的结构如下图所示：



为了能让我们的模拟器能访问到数据，我能需要将 1002/1003...10105 文件夹中的 list\_1.json 和 list\_2.json 文件内容的 url 地址改为我们本机的 ip 地址。



	724D6A55496A11726628_files	2015/3/3 10:48	文件夹	
	724D6A55496A11726628.html	2014/10/10 8:55	HTML 文档	11 KB
	1452327318UU91.jpg	2014/10/10 8:55	JPEG 图像	21 KB
	1452327318UU92.jpg	2014/10/10 8:55	JPEG 图像	18 KB
	list_1.json	2015/3/6 10:55	JSON 文件	5 KB
	list_2.json	2014/10/11 10:18	JSON 文件	5 KB

## 6.Android 布局文件优化 (★★★)

要实现一个好的布局，不只是实现了、显示出来就完了，不管层次，堆砌代码也可以实现功能，但是这显然违背了 Android 布局设计的原则。可能你会说，Android 布局设计哪有什么原则，我可以明确告诉你，当然有，只要有利于提高最终效果的方法、意识，我们都可以把它提升为原则。

如果我们的布局嵌套布局，当达到一定程度的时候可能会报 `java.lang.StackOverflowError` 错误，如下图所示：

```
java.lang.StackOverflowError
android.text.TextUtils.getChars(TextUtils.java:101)
android.graphics.Canvas.drawText(Canvas.java:1346)
android.text.Layout.draw(Layout.java:553)
android.widget.TextView.onDraw(TextView.java:6195)
android.view.View.draw(View.java:7312)
android.view.ViewGroup.drawChild(ViewGroup.java:1797)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:1432)
android.view.ViewGroup.drawChild(ViewGroup.java:1795)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:1432)
android.view.View.draw(View.java:7315)
android.view.ViewGroup.drawChild(ViewGroup.java:1797)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:1432)
android.view.View.draw(View.java:7315)
android.view.ViewGroup.drawChild(ViewGroup.java:1797)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:1432)
android.view.View.buildDrawingCache(View.java:7012)
android.view.View.getDrawingCache(View.java:6800)
android.view.ViewGroup.drawChild(ViewGroup.java:1649)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:1432)
android.widget.AbsListView.dispatchDraw(AbsListView.java:1950)
android.widget.ListView.dispatchDraw(ListView.java:3384)
android.view.View.draw(View.java:7315)
```

上面的问题说明白点儿就是“嵌套层次过深，导致栈溢出”。最直接的解决方法就是缩减层级，而且缩减层级还有一个重要的好处就是提高页面加载速度，因为 Android 中的布局是嵌套加载的，多一层布局就要耗费很长的加载时间。这时候我们在 xml 布局文件中可以巧妙的使用 `merge` 节点，下面我会结合 `hierarchyviewer` 工具来分别讲述一下

merge,ViewStub,include 在布局优化中的作用。

## 6.1 merge 的使用

顾名思义，就是合并、融合的意思。使用它可以有效的将某些符合条件的多余的层级优化掉。使用 merge 的场合

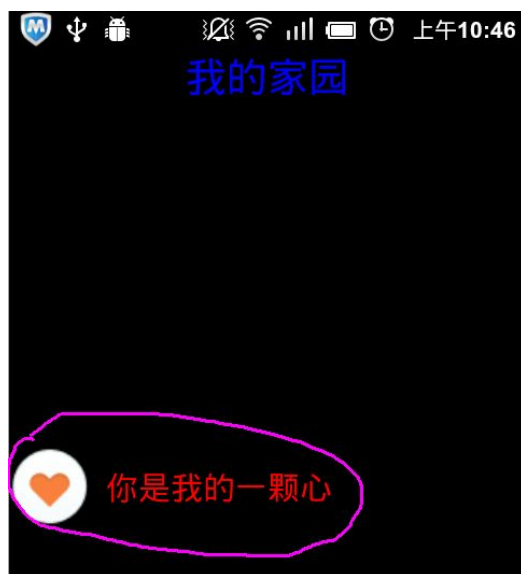
主要有两处：

(1) 自定义 View 中使用，父元素尽量是 FrameLayout，当然如果父元素是其他布局，而且不是太复杂的情况下也是可以使用的

(2) Activity 中的整体布局，根元素需要是 FrameLayout

下面这个例子将融合这两种情况，来展示如何缩减布局层次。

总体显示界面如图所示：



其中粉红色圈住的部分为我们的自定义 View。

整个界面的布局 layout\_mergedemo.xml 如下：

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
```

```

android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textSize="25sp"
    android:textColor="#FF0000FF"
    android:gravity="center_horizontal"
    android:text="我的家园"/>
<com.example.myandroiddemo.MyItemView
    android:id="@+id/myitem_view"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center" />
</FrameLayout>

```

自定义布局 view\_item.xml 如下：

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal" >
    <ImageView
        android:id="@+id/view_item_img"
        android:layout_width="50dp"
        android:layout_height="50dp"
        android:scaleType="fitXY"/>
    <TextView
        android:id="@+id/view_item_txt"
        android:layout_width="fill_parent"
        android:layout_height="50dp"
        android:gravity="center_vertical"
        android:layout_marginLeft="10dp"
        android:textSize="20sp"
        android:textColor="#FFFF0000"/>
</LinearLayout>

```

自定义 View 使用下面来解析布局：

```

public class MyItemView extends LinearLayout {
    .....
    private void initView(Context context) {
        mContext = context;
    }
}

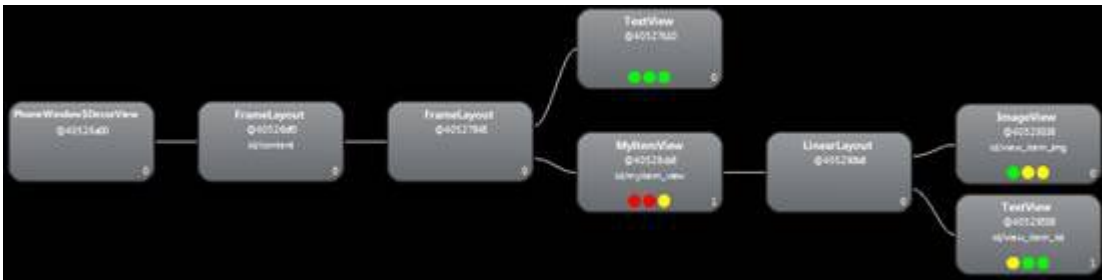
```

```

View view = LayoutInflater.from(mContext).inflate(R.layout.view_item, this,
true);
mMyItemImg = (ImageView)view.findViewById(R.id.view_item_img);
mMyItemText = (TextView)view.findViewById(R.id.view_item_txt);
}
.....
}

```

整个功能开发完成之后，使用 hierarchyviewer 来看一下布局层次：



我们发现简单的一个功能竟然使用了六层布局，包括每个 Window 自动添加的 PhoneWindow\$DecorView 和 FrameLayout ( id/content )。明显可以看到这个布局存在冗余，比如第二层和第三层的 FrameLayout，比如第四层的 MyItemView(LinearLayout 子类)和第五层的 LinearLayout，都可以缩减。

好了，该我们的 merge 标签大显身手了。将布局这样修改：

简化 layout\_mergedemo.xml：

```

<merge xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools">
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textSize="25sp"
    android:textColor="#FF0000FF"
    android:gravity="center_horizontal"
    android:text="我的家园"/>
<com.example.myandroiddemo.MyItemView
    android:id="@+id/myitem_view"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center" />
</merge>

```

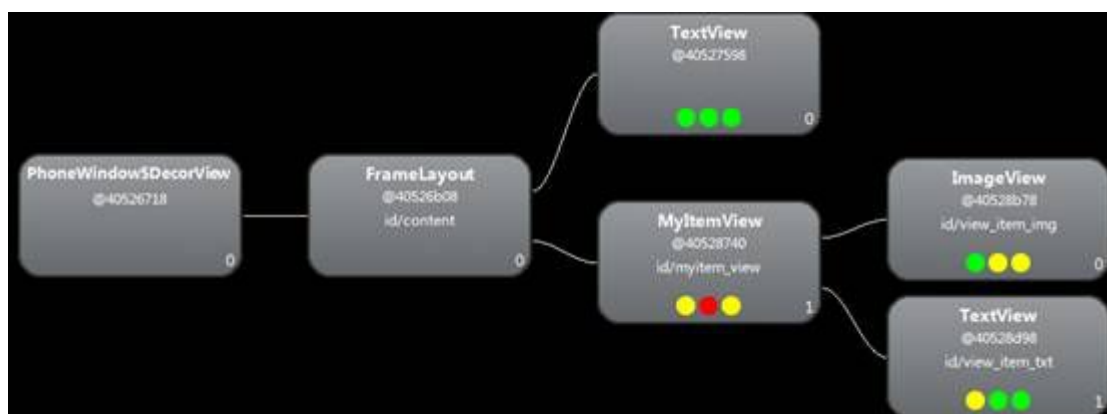
简化 view\_item.xml：

```
<merge xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools" >
    <ImageView
        android:id="@+id/view_item_img"
        android:layout_width="50dp"
        android:layout_height="50dp"
        android:scaleType="fitXY"/>
    <TextView
        android:id="@+id/view_item_txt"
        android:layout_width="fill_parent"
        android:layout_height="50dp"
        android:gravity="center_vertical"
        android:layout_marginLeft="10dp"
        android:textSize="20sp"
        android:textColor="#FFFF0000"/>
</merge>
```

因为这里 merge 代替的布局元素为 LinearLayout，而不是 FrameLayout，所以我们需要在自定义布局代码中将 LinearLayout 的属性添加上，比如垂直或者水平布局，比如背景色等，此处设置为水平布局：

```
private void initView(Context context) {
    setOrientation(LinearLayout.HORIZONTAL);
    .....
}
```

修改后我们再来看看布局层次吧：



我们可以看到只有四层了，加载速度也明显加快，特别是如果布局比较复杂，子 View 较多的情况下，合理使用 merge 能大大提高程序的速度和流畅性。

但是使用 merge 标签还是有一些限制的，具体有以下几点：

- ( 1 ) merge 只能用在布局 XML 文件的根元素
- ( 2 ) 使用 merge 来 inflate 一个布局时，必须指定一个 ViewGroup 作为其父元素，并且要设置 inflate 的 attachToRoot 参数为 true。（参照 inflate(int, ViewGroup, boolean)）
- ( 3 ) 不能在 ViewStub 中使用 merge 标签。最直观的一个原因就是 ViewStub 的 inflate 方法中根本没有 attachToRoot 的设置

## 6.2 ViewStub 的使用

其实就是一个轻量级的页面，我们通常使用它来做预加载处理，来改善页面加载速度和提高流畅性，ViewStub 本身不会占用层级，它最终会被它指定的层级取代。

比如有如下布局文件，layout\_loading.xml 布局文件：

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">
    <ViewStub
        android:id="@+id/viewstub"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"/>
    <NetErrAndLoadView
        android:id="@+id/start_loading_lay"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</merge>
```

这个页面是相当轻量级的，所以导致动画加载速度非常快、而且流畅。等页面切换动画完成之后，我们再指定 ViewStub 的资源，来加载实际的页面，这个资源就是实际要加载的页面的布局文件。比如要加载 MainActivity 的布局文件 layout\_main.xml，onCreate 实现如下：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.layout_loading);
}
```



```
mLoadHandler = new Handler();
mLoadingView = (NetErrAndLoadView)findViewById(R.id.start_loading_lay);
mLoadingView.startLoading();
mViewStub = (ViewStub)findViewById(R.id.viewstub);
mViewStub.setLayoutResource(R.layout.layout_main);

mLoadHandler.postDelayed(new Runnable() {
    @Override
    public void run() {
        mViewStub.inflate();
        mLoadingView.hide();
    }
}, 500);
}
```

上面的 500 单位是 ms，就是延迟加载的时间。上面使用的是动态添加 ViewStub 指向布局资源的方法 ( mViewStub.setLayoutResource(R.layout.layout\_main)) ,当然根据需要可以直接在 ViewStub 的布局块儿中设置，需要设置 ViewStub 标签下的 layout 属性(android:layout="@layout/ layout\_main")。

ViewStub 也是有少许缺点的，下面所说：

- 1、ViewStub 只能 Inflate 一次，之后 ViewStub 对象会被置为空。按句话说，某个被 ViewStub 指定的布局被 Inflate 后，就不能够再通过 ViewStub 来控制它了。所以它不适用 于需要按需显示隐藏的情况。
- 2、ViewStub 只能用来 Inflate 一个布局文件，而不是某个具体的 View，当然也可以把 View 写在某个布局文件中。如果想操作一个具体的 view，还是使用 visibility 属性吧。
- 3、ViewStub 中不能嵌套 merge 标签。不过这些确定都无伤大雅，我们还是能够用 ViewStub 来做很多事情。

## 6.3 include 的使用

制造这个标签纯粹是为了布局重用，在项目中通常会存在一些布局公用的部分，比如自定义标题，我们不需要把一份代码 Ctrl C, Ctrl V 的到处都是，严重违背程序简洁化、模块儿化的设计思想。简单说一下如何调用吧，比如现在



有一个公用布局 head.xml，如果在其他的布局中引用，只需要这样添加 include 标签：

```
<include android:layout_width="fill_parent" android:layout_height="wrap_content"
layout="@Layout/head" />
```

这个标签下的布局在父布局刷新时候同样是会加载的，所以它即不能像 merge 那样缩减层级，也不能像 ViewStub 那样轻量级实现延迟加载，它就是用来布局重用的。

**至此，本文档完！**

2015 年 3 月 6 日 星期五 17:41:28

北京市海淀区中关村软件园国际软件大厦