# **MVVM之DataBinding学习笔记**

# 前言

  ...想了半分钟，好像并没有啥需要特别声明的，能翻到这篇文章的人，相信也早已了解了DataBinding是个什么东西，所以还是简单粗暴点儿，就当给自己留个笔记...开撸吧！

# 配置

* 确保sdk的support包更新到了最新版
* 在对应module的build.gradle文件中进行如下配置（需AS版本1.5以上）

android {

...

dataBinding {

enabled = true

}

}

# 基本使用

## 数据绑定

### Activity

* **先上layout代码**

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<!--如果本页不需要绑定数据，data标签可以省略-->

<data class="DataBingMain">

<import type="lxf.widget.util.AppUtils"/>

<variable

name="user"

type="lxf.androiddemos.model.UserEntity"/>

</data>

<LinearLayout

xmlns:tools="http://schemas.android.com/tools"

tools:context="lxf.androiddemos.ui.DatabindingActivity"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:padding="16dp"

android:background="@mipmap/bg\_robot"

android:orientation="vertical"

>

<TextView

style="@style/text\_base"

android:text="@{`name:`+user.name}"/>

<TextView

style="@style/text\_base"

android:text="@{user.sex}"/>

<TextView

style="@style/text\_base"

android:text="@{String.valueOf(user.age)}"/>

<TextView

style="@style/text\_base"

android:text="@{user.initType(user.type)}"/>

</LinearLayout>

</layout>

1. layout的编写方式改变：新增layout和data标签，layout为根布局，包含data和ui布局两部分。data即要绑定的数据model，ui布局即我们以前写法中的根布局。
2. data标签：
   * 使用DataBinding编写布局，系统会自动生成一个继承ViewDataBinding类，而class属性可以指定这个类的名字，如果不指定，则会根据xml的名字自动生成。
   * variable可以设置多个。
   * 通俗的讲，name即变量名，可以在下面直接引用，同时会在自动生成的ViewDataBinding类中自动生成setXXX和getXXX方法，用来绑定数据。
   * type即我们的数据model。
   * 支持import，导入后可以在@{}中直接使用，方式同java。
3. @{}语法中支持大部分的java操作，当然最好不要写太复杂的语句，如果有这个需求，可以在java类中写一个方法，然后在此调用：
   * 运算符： + - / \* % () && || & | ^ >> >>> << == > < >= <=
   * 字符串拼接 + (注意字符串要用``括起来，esc下面那个键)
   * instanceof
   * 方法调用
   * res资源访问
   * 数组访问 []
   * 三目运算 表达式1 ? 表达式2 : 表达式3
   * 聚合判断 表达式1 ?? 表达式2 （表达式1为null，则返回表达式2）
   * 等等...

**activity代码**  
　　通过DataBindingUtil.setContentView方法代替原来的setContentView。

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

//1.获取ViewDataBinding对象

DataBingMain dataBinding = DataBindingUtil.setContentView(this, R.layout.activity\_databinding);

//2.获取数据

UserEntity user = new UserEntity();

user.setName("lxf");

user.setSex("man");

user.setAge(25);

user.setType(1);

//3.绑定数据

dataBinding.setUser(user);

//dataBinding.setVariable(BR.user,user);

}

**model代码**

package lxf.androiddemos.model;

public class UserEntity {

private String name;

private String sex;

private int age;

private int type;

public String initType(int type){

String result;

switch (type){

case 1:

result = "程序猿";

break;

case 2:

result = "程序猿的天敌";

break;

default:

result = "无业游民";

break;

}

return result;

}

//setter getter方法略

}

activity和model代码很简单，就不需要解释了。

### Fragment

　　看到这里应该有个疑问：fragment中没有setContentView方法，该怎么办？  
　　所幸DataBinding库还提供了另外一个初始化布局的方法：DataBindingUtil.inflate()。

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,

Bundle savedInstanceState) {

ViewDataBinding binding = DataBindingUtil.inflate(inflater,R.layout.fragment\_blank,container,false);

return binding.getRoot();

}

　　xml布局的写法同activity。

## 列表绑定

  在此已RecyclerView为例。

### 单布局

**先看item布局：**

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<data>

<variable

name="item"

type="lxf.androiddemos.model.MainRecyclerItem"/>

</data>

<android.support.v7.widget.CardView

android:layout\_width="match\_parent"

android:layout\_height="60dp"

android:layout\_marginLeft="16dp"

android:layout\_marginRight="16dp"

android:layout\_marginTop="16dp"

app:cardElevation="5dp"

android:onClick="@{item.onItemClick}"

>

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_gravity="center"

android:textSize="18sp"

android:textColor="@color/text\_green\_bg"

android:text="@{item.content}"/>

</android.support.v7.widget.CardView>

</layout>

  这里值得我们关注的有两个地方，一个就是TextView上的数据绑定，一个是父布局上的onClick属性，可以通过这种方式来设置item点击事件，说白了其实就是调用MainRecyclerItem中的一个方法，我们可以通过getter方法很方便的知道当前item的具体数据，具体实现请往下看。  
　　**model实体类**

public class MainRecyclerItem {

public static final String[] items = new String[]{"ViewDragHelper", "自定义Behavior", "二维码", "DataBinding"};

private String content;

public void onItemClick(View view) {

Intent intent = null;

switch (getContent()) {

case "ViewDragHelper":

intent = new Intent(view.getContext(), ViewDragHelperActivity.class);

break;

case "自定义Behavior":

intent = new Intent(view.getContext(), BehaviorActivity.class);

break;

case "二维码":

intent = new Intent(view.getContext(), ZxingActivity.class);

break;

case "DataBinding":

intent = new Intent(view.getContext(), DatabindingActivity.class);

break;

}

if (intent != null)

view.getContext().startActivity(intent);

}

public String getContent() {

return content;

}

public void setContent(String content) {

this.content = content;

}

}

**adapter**

class RecyclerBindingViewHolder extends RecyclerView.ViewHolder {

ViewDataBinding binding;

private RecyclerBindingViewHolder(ViewDataBinding binding) {

super(binding.getRoot());

this.binding = binding;

}

static RecyclerBindingViewHolder createViewHolder(ViewGroup parent, int layoutId) {

ViewDataBinding binding = DataBindingUtil.inflate(LayoutInflater.from(parent.getContext()),layoutId,parent,false);

return new RecyclerBindingViewHolder(binding);

}

public abstract class BaseRecyclerBindingAdapter extends RecyclerView.Adapter<RecyclerBindingViewHolder> implements ChangeDataLinstener{

protected List<Object> mData;

public BaseRecyclerBindingAdapter(List<Object> list) {

mData = (list != null) ? list : new ArrayList<>();

}

@Override

public RecyclerBindingViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {

return RecyclerBindingViewHolder.createViewHolder(parent,getItemLayoutId(viewType));

}

@Override

public void onBindViewHolder(RecyclerBindingViewHolder holder, int position) {

//绑定数据

holder.binding.setVariable(getItemVariableId(),mData.get(position));

holder.binding.executePendingBindings();

}

@Override

public int getItemCount() {

return mData.size();

}

public void setmData(List<Object> mData) {

this.mData = mData;

notifyDataSetChanged();

}

//item布局id

public abstract int getItemLayoutId(int viewType);

//对应item布局里面data标签中的name，会自动生成一个BR.xxx属性，类似于R文件

public abstract int getItemVariableId();

}

　　我们这里把adapter写成了一个抽象类，如果没有什么很奇葩的要求，可以算一个通用adapter了，可以看到它没有任何的findviewbyid和set数据，一切都在布局中封装好了，实现非常的简洁。

### 多布局

　　如果我们想加一个头部文件，可以这样：

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="item"

type="lxf.androiddemos.model.MainRecyclerHeader"/>

</data>

<TextView

android:layout\_width="match\_parent"

android:layout\_height="80dp"

android:gravity="center"

android:text="@{item.header}"

android:background="@color/text\_orange\_bg">

</TextView>

</layout>

　　注意和之前的item布局有个相通之处，就是data标签的name属性值是一样的,然后通过getItemViewType实现我们的不同布局即可。

datas = new ArrayList<>();

MainRecyclerHeader header = new MainRecyclerHeader();

header.setHeader("我是头部文件");

datas.add(header);for (int i = 0; i < MainRecyclerItem.items.length; i++) {

MainRecyclerItem item = new MainRecyclerItem();

item.setContent(MainRecyclerItem.items[i]);

datas.add(item);

}

BaseRecyclerBindingAdapter bindingAdapter = new BaseRecyclerBindingAdapter(datas) {

@Override

public int getItemLayoutId(int viewType) {

return viewType;

}

@Override

public int getItemVariableId() {

return BR.item;//对应item布局里面data标签中的name

}

@Override

public int getItemViewType(int position) {

if (position == 0)

return R.layout.header\_recycler\_main;

else

return R.layout.item\_recycler\_main;

}

};

## 事件绑定

　　事件绑定说白了，其实就是一种特殊的变量绑定，或者说是一个方法的调用。

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<!--如果本页不需要绑定数据，data标签可以省略-->

<data class="DataBingMain">

<variable

name="user"

type="lxf.androiddemos.model.UserEntity"/>

<variable

name="util"

type="lxf.androiddemos.test.TestUtil"/>

</data>

<LinearLayout

...

>

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:onClick="@{util.onBtnClick}"

/>

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:onTextChanged="@{util.onTextChanged}"

/>

</LinearLayout>

</layout>

public class TestUtil {

public void onBtnClick(View view){

Toast.makeText(view.getContext(),"onBtnClick",Toast.LENGTH\_SHORT).show();

}

public void onTextChanged(CharSequence s, int start, int before, int count){

System.out.println(s);

}

}

　　需要注意的是通过onclick这种方式绑定的事件，实现方法中一定要传入view参数（类似于传统的onClick方法），否则编译会报错。  
　　同样的官方文档提到，你也可以用这种方式来绑定一些比较偏门的监听，比如上面的onTextChanged，方法参数必须与传统的onTextChanged参数一模一样，否则编译报错，这种方式可以使你只监听onTextChanged一个方法，而非TextWatcher的三个方法，另外EditText本身是没有android:onTextChanged这个属性的，具体实现原理需要先理解一下什么是databinding的自定义属性，会在后文提到。

# 进阶使用

## 数据更新

　　在很多情况下，我们需要动态去设置相关数据，DataBinding为我们提供了两种方式来实现它。  
　　**方法一**

1. 实体类继承BaseObservable，或者自己实现Observable
2. 在需要刷新的属性的get方法上添加@Bindable注解，此时会自动生成BR类。（这里有个坑，很多时候BR文件不会自动生成，此时需要重启AS...请让我先默默地日一波dog）
3. 在相应的set方法里调用notifyPropertyChanged(BR.xxx)进行刷新。

package lxf.androiddemos.model;

import android.databinding.BaseObservable;import android.databinding.Bindable;import android.view.View;

import lxf.androiddemos.BR;

public class UserEntity extends BaseObservable{

private String name;

private String sex;

private int age;

private int type;

...

public void addAge(View view) {

setAge(getAge() + 1);

}

@Bindable

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;// notifyChange();//刷新所有可刷新数据

notifyPropertyChanged(BR.age);

}

}

**方法二**

1. 实体类继承BaseObservable，或者自己实现Observable
2. 使用ObservableField<>，泛型可以填入自己需要的类型，注意必须要初始化。对于基本数据类型也可以直接使用ObservableBoolean, ObservableByte, ObservableChar, ObservableShort, ObservableInt, ObservableLong, ObservableFloat, ObservableDouble和ObservableParcelable。
3. 通过set和get方法为ObservableField设值和取值

package lxf.androiddemos.model;

import android.databinding.BaseObservable;import android.databinding.ObservableField;import android.view.View;

public class UserEntity extends BaseObservable{

public ObservableField<String> address = new ObservableField<>();

public void changeAddress(View view){

address.set("change:" + address.get());

}

}

## 自定义属性绑定适配器和回调

### 自动寻找setter

　　DataBinding在遇到属性绑定时，会自动去寻找该属性的set方法，找到就会调用，找不到就报错。

<ImageView

android:layout\_width="50dp"

android:layout\_height="50dp"

app:imageResource="@{R.mipmap.ic\_launcher}"/>

　　比如上面这段代码，我们知道ImageView中是没有imageResource这个属性的，但是有setImageResource(int resId)方法，因此这段代码是可以正常运行的。利用这种特性，可以为一些自定义控件增加setter方法，使其支持DataBinding。

### @BindingMethods

　　当xml属性名称和源码中set方法名称不一致时，可以通过这种方式来进行绑定。先看一个官方的实现：

@BindingMethods({

...

@BindingMethod(type = TextView.class, attribute = "android:inputType", method = "setRawInputType"),

...

})

　　这段代码的意思就是将TextView的android:inputType属性绑定到setRawInputType方法，其实也可以通俗的认为是为原本的setter方法起了一个别名。

### @BindingAdapter

　　很多时候，源码中并没有提供set方法，比如ImageView，我们希望通过设置url来达到加载图片的目的，我们可以通过@BindAdapter来实现。

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<!--如果本页不需要绑定数据，data标签可以省略-->

<data class="DataBingMain">

<import type="lxf.androiddemos.R"/>

<variable

name="user"

type="lxf.androiddemos.model.UserEntity"/>

</data>

<LinearLayout

...

>

<ImageView

android:layout\_width="100dp"

android:layout\_height="100dp"

app:url="@{user.img}"

app:placeHolder="@{R.mipmap.ic\_launcher}"/>

</LinearLayout>

</layout>

public class UserEntity extends BaseObservable{

private String img;

@BindingAdapter(value = {"url","placeHolder"},requireAll = false)

public static void setImage(ImageView imageView ,String url,int placeHolder){

Glide.with(imageView.getContext()).load(url).placeholder(placeHolder).into(imageView);

}

}

这里有几点需要注意：

* xml文件中一定不要忘记各种类的import（除java.lang包外均需导入），否则你一定会碰到databinding程序包不存在这个错误。
* 设置@BindAdapter注解的方法需要是static的，否则编译也会报错。
* 你可以把这个方法设置在一个专门的工具类中，不是说必须要在这个model实体类里。
* @BindAdapter包含value和requireAll两个属性，value是一个String[]，包含你自定义的属性。requireAll意思是是否需要设置你在value中声明的全部属性，默认为true。如果设定为false，那么没赋值的自定义属性会传默认值。

　　到这里，我们来回头看一下之前在**事件绑定**中留下的那个坑——onTextChanged，其实这是官方提前为我们封装好的：

@BindingAdapter(value = {"android:beforeTextChanged", "android:onTextChanged",

"android:afterTextChanged", "android:textAttrChanged"}, requireAll = false)

public static void setTextWatcher(TextView view, final BeforeTextChanged before,

final OnTextChanged on, final AfterTextChanged after,

final InverseBindingListener textAttrChanged) {

final TextWatcher newValue;

if (before == null && after == null && on == null && textAttrChanged == null) {

newValue = null;

} else {

newValue = new TextWatcher() {

@Override

public void beforeTextChanged(CharSequence s, int start, int count, int after) {

if (before != null) {

before.beforeTextChanged(s, start, count, after);

}

}

@Override

public void onTextChanged(CharSequence s, int start, int before, int count) {

if (on != null) {

on.onTextChanged(s, start, before, count);

}

if (textAttrChanged != null) {

textAttrChanged.onChange();

}

}

@Override

public void afterTextChanged(Editable s) {

if (after != null) {

after.afterTextChanged(s);

}

}

};

}

final TextWatcher oldValue = ListenerUtil.trackListener(view, newValue, R.id.textWatcher);

if (oldValue != null) {

view.removeTextChangedListener(oldValue);

}

if (newValue != null) {

view.addTextChangedListener(newValue);

}

}

　　可以看到，当on!=null时，会调用传统的onTextChanged方法。

### @BindingConversion

　　方法注释，当自定义的属性和setter方法中需要的参数类型不符时进行转换。

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:background="@{`#de325e`}"

/>

　　上面这种写法，直接编译是会报错的，因为setBackground接收的是一个Drawable对象，而我们传入的是个string，所以我们此处可以用@BindingConversion来转换一下（PS：我知道传统写法是可以直接传字符串颜色值的，我只是举个简单例子）。

@BindingConversion

public static Drawable colorToDrawable(String color){

return new ColorDrawable(Color.parseColor(color));

}

　　DataBinding在碰到这种参数类型不对的问题时，会自动去检索看看有没有相关的@BindingConversion方法，如果有的话则会调用，需要注意，这个方法也需要是static的。

## 接口回调

### model的回调

　　当属性值变化时的回调。

user.addOnPropertyChangedCallback(new Observable.OnPropertyChangedCallback() {

@Override

public void onPropertyChanged(Observable sender, int propertyId) {

if (propertyId == BR.age){

Toast.makeText(getApplicationContext(),"age刷新了",Toast.LENGTH\_SHORT).show();

}

}

});

### ViewDatabinding的回调

　　当ViewDataBinding执行executePendingBindings()尺寸必须再次评估时的回调。可以设置一些view的展示动画等。

dataBinding.addOnRebindCallback(new OnRebindCallback() {

@Override

public boolean onPreBind(ViewDataBinding binding) {

return super.onPreBind(binding);

}

@Override

public void onCanceled(ViewDataBinding binding) {

super.onCanceled(binding);

}

@Override

public void onBound(ViewDataBinding binding) {

super.onBound(binding);

}

});

## 双向绑定

### 基本数据

　　双向绑定意思不仅数据绑定UI，同时UI更新时可以刷新数据，语法为@={},举个例子：

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:onTextChanged="@{util.onTextChanged}"

android:text="@={user.address}"

/>

　　这样在程序运行后，editText会自动显示user.address的初始值，改变editText，则user.address也会同步改变，可以想象，如果我们将user.address绑定另一个TextView，则TextView的内容会跟随editText的变化而变化。

### 隐式属性监听

<ImageView

android:layout\_width="100dp"

android:layout\_height="100dp"

android:visibility="@{checkbox.checked?View.VISIBLE:View.GONE}"

app:placeHolder="@{R.mipmap.ic\_launcher}"

app:url="@{user.img}" />

<CheckBox

android:id="@+id/checkbox"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content" />

　　可以看ImageView的visibility属性，通过CheckBox的checked属性来控制自身的显示和隐藏，这是官方给出的一种支持，同时官方还支持下面这些属性：

* **AbsListView** android:selectedItemPosition
* **CalendarView** android:date
* **CompoundButton** android:checked
* **DatePicker** android:year, android:month, android:day (yes, these are synthetic, but we had a listener, so we thought you’d want to use them)
* **NumberPicker** android:value
* **RadioGroup** android:checkedButton
* **RatingBar** android:rating
* **SeekBar** android:progress
* **TabHost** android:currentTab (you probably don’t care, but we had the listener)
* **TextView** android:text
* **TimePicker** android:hour, android:minute (again, synthetic, but we had the listener)

### 自定义双向绑定

　　双向绑定其实就是正向绑定+反向绑定，前面讲的全部是正向绑定，截下来我们来看看怎么定义反向绑定。

#### 绑定方法（@InverseBindingMethods）

　　首先先来了解几个名词：

* @InverseBindingMethods：其实就是元素为@InverseBindingMethod的一个数组，用来注解**类**。

@Target(ElementType.TYPE)public @interface InverseBindingMethods {

InverseBindingMethod[] value();

}

@InverseBindingMethod：反向绑定方法，用来确定怎么去监听view属性的变化和回调哪一个getter方法。包含以下4个属性：

* + type：包含attribute的view类型。
  + attribute:支持双向绑定的属性（string格式）。
  + event:可以省略，用来通知DataBinding系统attribute已经改变，默认为attribute + "AttrChanged"。（UI通知数据）
  + method:可以省略，用来从view获取数据的方法，不设定的话会自动寻找"is" 或 "get" + attribute方法。（数据刷新）

　　event调用时机需要通过@BindingAdapter进行设置。

* InverseBindingListener：反向绑定监听器，当使用双向绑定时，会在你的layout自动生成的binding类中自动生成一个InverseBindingListener的实现（拗口吗？好像有一点点。。不理解的可以去看看源码）。

　　看完这几个名词是不是已经凌乱了？（话说我当时也差点哭了。。），我们来看个官方例子消化一下：

@InverseBindingMethods({

@InverseBindingMethod(type = CompoundButton.class, attribute = "android:checked"),

})//1.这里需要双向绑定的是checked属性，event和method都省略了。public class CompoundButtonBindingAdapter {

...

//2.设置什么时候调用event

@BindingAdapter(value = {"android:onCheckedChanged", "android:checkedAttrChanged"},

requireAll = false)

public static void setListeners(CompoundButton view, final OnCheckedChangeListener listener,

final InverseBindingListener attrChange) {

if (attrChange == null) {

view.setOnCheckedChangeListener(listener);

} else {

view.setOnCheckedChangeListener(new OnCheckedChangeListener() {

@Override

public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {

if (listener != null) {

listener.onCheckedChanged(buttonView, isChecked);

}

attrChange.onChange();

}

});

}

}

}

//3.我们在layout中使用双向绑定

<CheckBox

android:id="@+id/checkbox"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:checked="@={user.checked}"/>

//4.layout的binding类中自动生成的InverseBindingListener实现。

// Inverse Binding Event Handlers

private android.databinding.InverseBindingListener checkboxandroidCheck = new android.databinding.InverseBindingListener() {

@Override

public void onChange() {//这段逻辑其实就是用来更新user实体类中的checked字段的

// Inverse of user.checked.get()

// is user.checked.set((java.lang.Boolean) callbackArg\_0)

boolean callbackArg\_0 = checkbox.isChecked();//其实就是method

// localize variables for thread safety

// user.checked != null

boolean checkedUserObjectnul = false;

// user.checked

android.databinding.ObservableField<java.lang.Boolean> checkedUser = null;

// user

lxf.androiddemos.model.UserEntity user = mUser;

// user.checked.get()

java.lang.Boolean CheckedUser1 = null;

// user != null

boolean userObjectnull = false;

userObjectnull = (user) != (null);

if (userObjectnull) {

checkedUser = user.checked;

checkedUserObjectnul = (checkedUser) != (null);

if (checkedUserObjectnul) {

checkedUser.set((java.lang.Boolean) (callbackArg\_0));

}

}

}

};

　　整个反向绑定的流程下来其实就是：

1. 定义需要反向绑定的属性（checked），并配置event（checkedAttrChanged）和method（isChecked）。
2. 系统会自动根据event找到对应的方法（setLinstener），配置好调用时机。
3. 开发者在layout中使用双向绑定。
4. 自动在binding类中生成一个InverseBindingListener的实现。

#### 绑定适配器（@InverseBindingAdapter）

1. 下面再来看个新名词...( ╯□╰ )：
2. @InverseBindingAdapter：反向绑定适配器，用来注解**方法**。只包含attribute和event两个属性，含义同上：
   * attribute：支持双向绑定的属性（string格式）。
   * event：可以省略，用来通知DataBinding系统attribute已经改变，默认为attribute + "AttrChanged"。需要通过@BindingAdapter进行设置调用时机。

　　@InverseBindingAdapter注解的方法本身就相当于获取数据的getter方法（类似于@BindingAdapter注解的方法本身就相当于setter方法）。

　　官方案例（双向绑定android:text）：

//1.这一步相当于做了两个操作：确定绑定的属性和event；指定getter方法@InverseBindingAdapter(attribute = "android:text", event = "android:textAttrChanged")

public static String getTextString(TextView view) {

return view.getText().toString();

}//2.根据event找到对应方法，配置event的调用时机。 @BindingAdapter(value = {"android:beforeTextChanged", "android:onTextChanged",

"android:afterTextChanged", "android:textAttrChanged"}, requireAll = false)

public static void setTextWatcher(TextView view, final BeforeTextChanged before,

final OnTextChanged on, final AfterTextChanged after,

final InverseBindingListener textAttrChanged) {

final TextWatcher newValue;

if (before == null && after == null && on == null && textAttrChanged == null) {

newValue = null;

} else {

newValue = new TextWatcher() {

...

@Override

public void onTextChanged(CharSequence s, int start, int before, int count) {

if (on != null) {

on.onTextChanged(s, start, before, count);

}

if (textAttrChanged != null) {

textAttrChanged.onChange();

}

}

....

};

}

...

}

//3.使用双向绑定

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:onTextChanged="@{util.onTextChanged}"

android:text="@={user.address}" />

//4.binding类中自动生成InverseBindingListener的实现。private android.databinding.InverseBindingListener mboundView10androidT = new android.databinding.InverseBindingListener() {

@Override

public void onChange() {

// Inverse of user.address.get()

// is user.address.set((java.lang.String) callbackArg\_0)

java.lang.String callbackArg\_0 = android.databinding.adapters.TextViewBindingAdapter.getTextString(mboundView10);//getter方法

// localize variables for thread safety

// user.address != null

boolean addressUserObjectnul = false;

// user

lxf.androiddemos.model.UserEntity user = mUser;

// user.address

android.databinding.ObservableField<java.lang.String> addressUser = null;

// user.address.get()

java.lang.String AddressUser1 = null;

// user != null

boolean userObjectnull = false;

userObjectnull = (user) != (null);

if (userObjectnull) {

addressUser = user.address;

addressUserObjectnul = (addressUser) != (null);

if (addressUserObjectnul) {

addressUser.set((java.lang.String) (callbackArg\_0));

}

}

}

};

　　一大堆的代码看下来，其实绑定方法和绑定适配器两种方法的最终效果是一样的，实现过程也是大同小异，这里就不赘述了，和上面的绑定方法基本一致。

#### 比葫芦画瓢

　　我们来自定义实现这样一个效果，点击改变自定义view的颜色，同时将色值在另一个TextView中展示出来（虽然没什么卵用，仅仅当个案例吧），效果图如下（请自觉忽略其他的东西。。）：

实现过程：

//1.自定义ColorPicker，并为color属性添加getter和setter方法public class ColorPicker extends View {

...

private String mColor;

public String getColor() {

return mColor;

}

public void setColor(String mColor) {

this.mColor = mColor;

paint.setColor(Color.parseColor(mColor));

invalidate();

}

...

}//2.自定义反向绑定@InverseBindingMethods({

@InverseBindingMethod(type = ColorPicker.class,attribute = "color")

})public class ColorPickerAdapter {

@BindingAdapter(value = {"colorAttrChanged"},requireAll = false)

public static void setListener(ColorPicker picker, final InverseBindingListener attrChange){

if (attrChange!=null){

picker.setOnColorChangeListener(new ColorPicker.OnColorChangeListener() {

@Override

public void onColorChange(ColorPicker picker, String color) {

//...

attrChange.onChange();

}

});

}

}

}//3.在layout中使用双向绑定

<lxf.androiddemos.test.ColorPicker

android:layout\_width="100dp"

android:layout\_height="100dp"

app:color="@={user.color}" />

　　上面给出了关键代码，刚接触DataBinding的萌新如果理解不了可以去文末下载Demo看看，只是一个很简单的案例，应该没什么问题。  
　　接下来我们用@InverseBindingAdapter来实现同样的效果：

public class ColorPickerAdapter {

@InverseBindingAdapter(attribute = "color")

public static String getColor(ColorPicker picker){

return picker.getColor();

}

@BindingAdapter(value = {"colorAttrChanged"},requireAll = false)

public static void setListener(ColorPicker picker, final InverseBindingListener attrChange){

if (attrChange!=null){

picker.setOnColorChangeListener(new ColorPicker.OnColorChangeListener() {

@Override

public void onColorChange(ColorPicker picker, String color) {

//...

attrChange.onChange();

}

});

}

}

}

　　另外关于一些情况下双向绑定存在的死循环问题，只要在setter方法中判断一下新老值不同即可。

## 依赖注入

　　DataBindingComponent，一般用于一个@BindingAdapter方法需要有多种实现时（比如说测试。。），我们来看一下前面那个修改年龄age的例子：

//原来的方式

@BindingAdapter(value = {"url","placeHolder"},requireAll = false)

public static void setImage(ImageView imageView , String url, int placeHolder){

ImgLoadUtil.load(imageView,url,placeHolder);

}

//运用DataBindingComponent//1.如果需要多种实现，可以先建一个抽象的adapter，注意方法为非静态的public abstract class AppAdapter {

@BindingAdapter(value = {"url","placeHolder"},requireAll = false)

public abstract void setImage(ImageView imageView , String url, int placeHolder);

}//2.添加抽象adapter的实现，这里我们只写了一个public class ImgAdapter extends AppAdapter {

@Override

public void setImage(ImageView imageView, String url, int placeHolder) {

ImgLoadUtil.load(imageView,url,placeHolder);

}

}public class Img2Adapter extends AppAdapter {

@Override

public void setImage(ImageView imageView, String url, int placeHolder) {

...

}

}//3.添加DataBindingComponent的实现（非静态的@BindingAdapter注解方法会自动在DataBindingComponent中生成相应的getter方法）。public class MyComponent implements android.databinding.DataBindingComponent {

@Override

public AppAdapter getAppAdapter() {

return new ImgAdapter();

}

}public class My2Component implements android.databinding.DataBindingComponent {

@Override

public AppAdapter getAppAdapter() {

return new Img2Adapter();

}

}//4.Activity中调用

//DataBingMain dataBinding = DataBindingUtil.setContentView(this, R.layout.activity\_databinding);

//DataBindingUtil.setDefaultComponent(new MyComponent());

DataBingMain dataBinding = DataBindingUtil.setContentView(this, R.layout.activity\_databinding,new MyComponent());

　　最终效果是一模一样的。

# 遇到的坑

Error:(8, 36) 错误: 程序包lxf.androiddemos.databinding不存在

遇到这种情况一般都是xml中的问题，比如data标签中引入的包名不对，或者是布局里面使用了什么错误的属性，等等。。。数据量大的时候，这种错误一般比较难找，简直就是日了dog。

需要更新数据时，为getter方法设置@Bindable，很多时候BR文件不会生成，需要重启AS，默默地再日一波dog。

* 最好不要使用clean project，否则R文件和BR文件会被清掉，R文件会自动重新生成，至于BR文件...那只dog，麻烦你再过来一下。