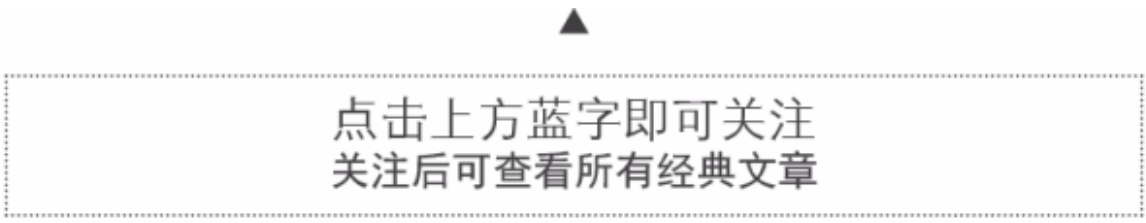
叶应是叶



5月28日，科沃斯机器人股份有限公司（股票简称：科沃斯，股票代码：603486）正式登陆上交所。据悉，科沃斯本次发行4010万股，发行价格20.02元/股，募集资金总额8.03亿元，募集资金总额扣除发行费用后，将用于年产400万台家庭服务机器人项目、机器人互联网生态圈项目和国际市场营销项目。

本篇来自 **叶应是叶** 的投稿，分享了Android DataBinding 从入门到进阶，一起来看看！希望大家喜欢。

叶应是叶 的博客地址：

https://www.jianshu.com/u/9df45b87cfdf

DataBinding是谷歌官方发布的一个框架，顾名思义即数据绑定，是MVVM模式在Android上的一种实现，用于降低布局和逻辑的耦合性，使代码逻辑更加清晰.MVVM相对于MVP，其实就是将Presenter层替换成了ViewModel层.DataBinding能够省去我们一直以来的findViewById（）步骤，大量减少Activity内的代码，数据能够单向或双向绑定到layout文件中，有助于防止内存泄漏，而且能自动进行空检测以避免空指针异常

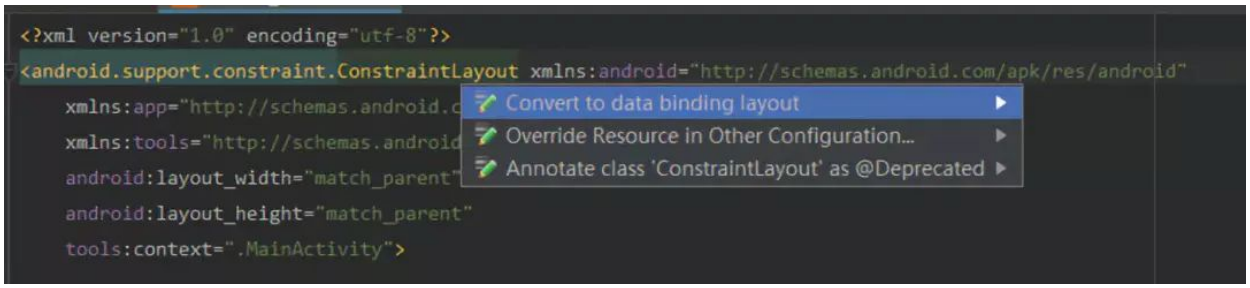
启用DataBinding的方法是在对应Model的build.gradle文件里加入以下代码，同步后就能引入对DataBinding的支持

```
android {

    dataBinding {

        enabled = true

    }

}
```

基础入门

启用DataBinding后，这里先来看下如何在布局文件中绑定指定的变量。

打开布局文件，选中根布局的ViewGroup，按住Alt +回车键，点击“ 转换为数据绑定布局 ”，就可以生成DataBinding需要的布局规则。




```
<?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"

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

    <data>
```

```
    </data>

    <android.support.constraint.ConstraintLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        tools:context=".MainActivity">

    </android.support.constraint.ConstraintLayout>

</layout>
```

和原始布局的区别在于多出了一个布局标签将原始布局包裹了起来，数据标签用于声明要用变量以及变量类型，要实现MVVM的ViewModel就需要把数据（Model）与UI（View）进行绑定，data标签的作用就像一个桥梁搭建了View and Model之间的通道。

这里先来声明一个模型

```
package com.leavesc.databinding_demo.model;

/**

 * 作者：叶应是叶

 * 时间：2018/5/16 20:20

 * 描述：https://github.com/leavesC
```

```
 */

public class User {

    private String name;

    private String password;

    . . .

}
```

在数据标签里声明要使用到的变量名，类的全路径

```
<data>

        <variable

            name="userInfo"

            type="com.leavesc.databinding_demo.model.User" />

    </data>
```

如果用户类型要多处用到，也可以直接将之后导入进来，这样就不用每次都指明整个包名路径了，而java.lang.*包中的类会被自动导入，所以可以直接使用

```
<data>

        <import type="com.leavesc.databinding_demo.model.User"/>
```

```
        <variable
            name="userInfo"
            type="User"/>
    </data>
```

如果存在进口的类名相同的情况，可以使用别名指定别名

```
<data>
        <import type="com.leavesc.databinding_demo.model.User" />
        <import
            alias="TempUser"
            type="com.leavesc.databinding_demo.model2.User" />
        <variable
            name="userInfo"
            type="User" />
        <variable
            name="tempUserInfo"
```

```
            type="TempUser" />

    </data>
```

这里声明了一个用户类型的变量userInfo，我们要做的就是使这个变量与两个TextView控件挂钩，通过设置userInfo的变量值同时使TextView显示相应的文本。完整的布局代码如下所示

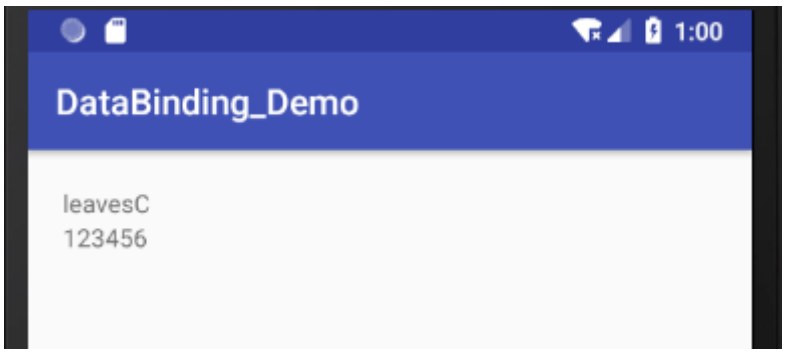
```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <import type="com.leavesc.databinding_demo.model.User" />
        <variable
            name="userInfo"
            type="User" />
    </data>
    <LinearLayout
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        android:layout_margin="20dp"
        android:orientation="vertical"
        tools:context="com.leavesc.databinding_demo.Main2Activity">
        <TextView
            android:id="@+id/tv_userName"
            ...
            android:text="@{userInfo.name}" />
        <TextView
            ...
            android:text="@{userInfo.password}" />
    </LinearLayout>
</layout>
```

通过@{userInfo.name}使TextView引用到相关的变量，DataBinding会将映射到相应的getter方法，之后可以在Activity中通过DataBindingUtil设置布局文件，省略原先

Activity的setContentView()方法，并为变量userInfo赋值

```
private User user;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMain2Binding activityMain2Binding =
DataBindingUtil.setContentView(this, R.layout.activity_main2);
        user = new User("leavesC", "123456");
        activityMain2Binding.setUserInfo(user);
    }
```



由于@{userInfo.name}在布局文件中并没有明确的值，所以在预览视图中什么都不会显示，不便于观察文本的大小和字体颜色等属性，此时可以为之设定默认值（文本内容或者是字体大小等属性都适用），默认值将只在预览视图中显示，且默认值不能包含引号

```
android:text="@{userInfo.name,default=defaultValue}"
```

此外，也可以通过ActivityMain2Binding直接获取到指定ID的控件

```
activityMain2Binding.tvUserName.setText("leavesC");
```

每个数据绑定布局文件都会生成一个绑定类，ViewDataBinding的实例名是根据布局文件名来生成，将之改为首字母大写的驼峰命名法来命名，并省略布局文件名包含的下划线。控件的获取方式类似，但首字母小写。也可以通过如下方式自定义ViewDataBinding的实例名

```
<data class="CustomBinding">


</data>
```

此外，绑定在表达式中会根据需要生成一个名为context的特殊变量，context的值是根视图的getContext()方法报道查看的Context对象，context变量会被具有该名称的显式变量声明所覆盖。Databinding同样是支持在Fragment和RecyclerView中使用。例如，可以看Databinding在Fragment中的使用

```
    @Override

    public View onCreateView(@NonNull LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState) {

        FragmentBlankBinding fragmentBlankBinding =
DataBindingUtil.inflate(inflater, R.layout.fragment_blank, container, false);

        fragmentBlankBinding.setHint("Hello");

        return fragmentBlankBinding.getRoot();

    }
```

以上实现数据绑定的方式，每当绑定的变量发生变化的时候，都需要重新向ViewDataBinding传递新的变量值才能刷新UI。接下来看如何实现自动刷新UI

单向数据绑定

实现数据变化自动驱动UI刷新的方式有三种：BaseObservable，ObservableField，ObservableCollection

- **BaseObservable**

一个纯净的ViewModel类被更新后，并不会让UI自动更新。而数据绑定后，我们自然会希望数据变更后UI会即时刷新，Observable就是为此而生的概念。BaseObservable提供了notifyChange（）和notifyPropertyChanged（）两个方法，前者会刷新所有的值域，后者则只更新对应的BR的标志，该BR的生成通过注释@Bindable生成，可以通过BR notify特定属性关联的视图

```
/**

 * 作者：叶应是叶

 * 时间：2018/5/16 20:54

 * 描述：

 */

public class Goods extends BaseObservable {

    //如果是 public 修饰符，则可以直接在成员变量上方加上 @Bindable 注解

    @Bindable

    public String name;

    //如果是 private 修饰符，则在成员变量的 get 方法上添加 @Bindable 注解

    private String details;

    private float price;

    public Goods(String name, String details, float price) {

        this.name = name;
```

```
        this.details = details;
        this.price = price;
    }
    public void setName(String name) {
        this.name = name;
        //只更新本字段
        notifyPropertyChanged(com.leavesc.databinding_demo.BR.name);
    }
    @Bindable
    public String getDetails() {
        return details;
    }
    public void setDetails(String details) {
        this.details = details;
        //更新所有字段
        notifyChange();
    }
    public float getPrice() {
        return price;
    }
    public void setPrice(float price) {
```

```
        this.price = price;
    }
}
```

在的setName（）方法中更新的只是本字段，而setDetails（）方法中更新的是所有字段。添加两个按钮用于改变货物变量的三个属性值，由此可以看出两个通知方法的区别。当前涉及的按钮点击事件绑定，在下面也会讲到

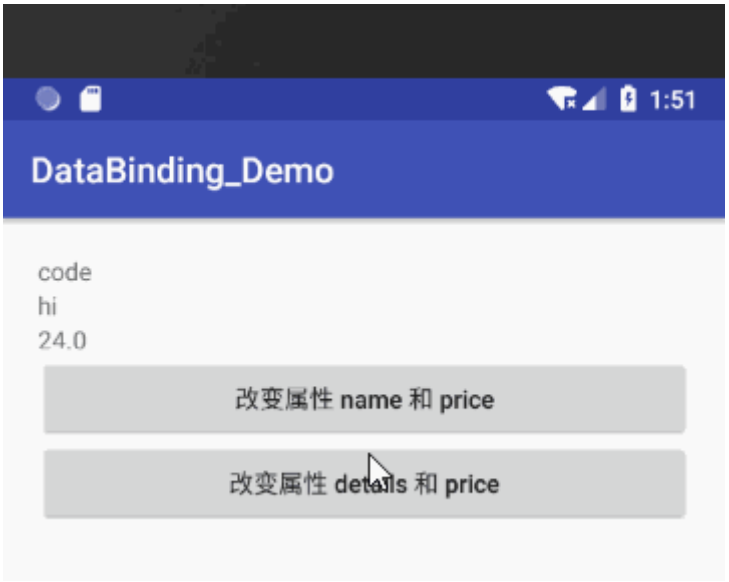
```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <import type="com.leavesc.databinding_demo.model.Goods" />
        <import
type="com.leavesc.databinding_demo.Main3Activity.GoodsHandler" />
        <variable
            name="goods"
            type="Goods" />
        <variable
```

```
            name="goodsHandler"

            type="GoodsHandler" />

    </data>

    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:orientation="vertical"

        android:padding="20dp"

        tools:context=".Main3Activity">

        <TextView

            . . .

            android:text="@{goods.name}" />

        <TextView

            . . .

            android:text="@{goods.details}" />
```

```
        <TextView
            ...
            android:text="@{String.valueOf(goods.price)}" />
        <Button
            ...
            android:onClick="@{()->goodsHandler.changeGoodsName()}"
            android:text="改变属性 name 和 price"
            android:textAllCaps="false" />
        <Button
            ...
            android:onClick="@{()->goodsHandler.changeGoodsDetails()}"
            android:text="改变属性 details 和 price"
            android:textAllCaps="false" />
    </LinearLayout>
```

```
</layout>

/**

 * 作者：叶应是叶

 * 时间：2018/5/16 21:07

 * 描述：

 */

public class Main3Activity extends AppCompatActivity {

    private Goods goods;

    private ActivityMain3Binding activityMain3Binding;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main3);

        activityMain3Binding = DataBindingUtil.setContentView(this,
R.layout.activity_main3);
```

```
        goods = new Goods("code", "hi", 24);
        activityMain3Binding.setGoods(goods);
        activityMain3Binding.setGoodsHandler(new GoodsHandler());
    }

    public class GoodsHandler {
        public void changeGoodsName() {
            goods.setName("code" + new Random().nextInt(100));
            goods.setPrice(new Random().nextInt(100));
        }

        public void changeGoodsDetails() {
            goods.setDetails("hi" + new Random().nextInt(100));
            goods.setPrice(new Random().nextInt(100));
        }
    }
}
```



可以看到，name视图的刷新没有同时刷新价格视图，而细节视图刷新的同时也刷新了价格视图。实现了可观察接口的类允许注册一个监听器，当可观察对象的属性更改时就会通知这个监听器，此时就需要用到OnPropertyChangedCallback。当中propertyId就用于标识特定的字段。

```
goods.addOnPropertyChangedCallback(new Observable.OnPropertyChangedCallback()
{

                @Override

                public void onPropertyChanged(Observable sender, int
propertyId) {

                    if (propertyId ==
com.leavesc.databinding_demo.BR.name) {

                        Log.e(TAG, "BR.name");

                    } else if (propertyId ==
com.leavesc.databinding_demo.BR.details) {

                        Log.e(TAG, "BR.details");
```

```
                } else if (propertyId ==
com.leavesc.databinding_demo.BR._all) {

                    Log.e(TAG, "BR._all");

                } else {

                    Log.e(TAG, "未知");

                }

            }

        });
```

- **ObservableField**

继承于Observable类相对来说限制有点高，且也需要进行通知操作，因此为了简单起见可以选择使用ObservableField .ObservableField 可以理解为官方对BaseObservable中字段的注解和刷新等操作的封装，官方原生提供了对基本数据类型的封装，例如ObservableBoolean，ObservableByte，ObservableChar，ObservableShort，ObservableInt，ObservableLong，ObservableFloat，ObservableDouble以及ObservableParcelable，也可以通过ObservableField泛型来申明其他类型。

```
/**

 * 作者：叶应是叶

 * 时间：2018/5/13 21:33

 * 描述：

 */
```

```
public class ObservableGoods {

    private ObservableField<String> name;

    private ObservableFloat price;

    private ObservableField<String> details;

    public ObservableGoods(String name, float price, String details) {

        this.name = new ObservableField<>(name);

        this.price = new ObservableFloat(price);

        this.details = new ObservableField<>(details);

    }

    ```

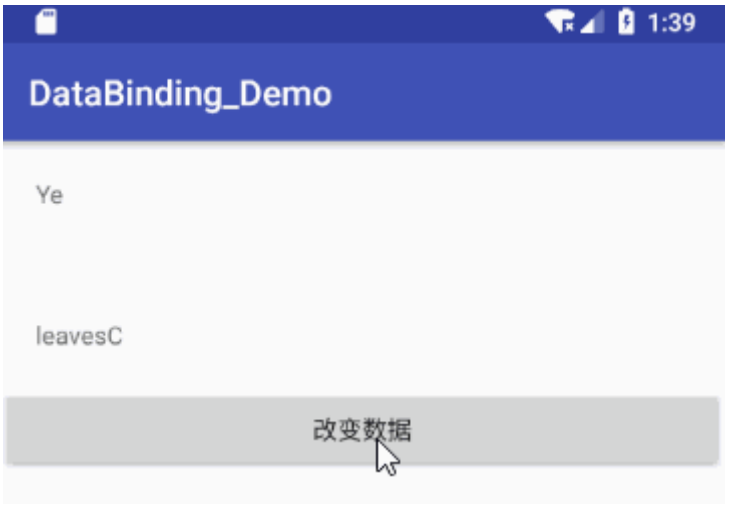
}
```

对ObservableGoods属性值的改变都会立即触发UI刷新，概念上与Observable区别不大，具体效果可看下面提供的源代码，这里不再赘述。

- **ObservableCollection**

dataBinding也提供了包装类用于替代原生的List和Map，分别是ObservableList和ObservableMap，当其包含的数据发生变化时，绑定的视图也会随之进行刷新。

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

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <import type="android.databinding.ObservableList"/>
        <import type="android.databinding.ObservableMap"/>
        <variable
            name="list"
            type="ObservableList&lt;String&gt;"/>
        <variable
            name="map"
            type="ObservableMap&lt;String,String&gt;"/>
        <variable
            name="index"
            type="int"/>
        <variable
```

```
            name="key"

            type="String"/>

    </data>

    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:orientation="vertical"

        tools:context="com.leavesc.databinding_demo.Main12Activity">

        <TextView

            . . .

            android:padding="20dp"

            android:text="@{list[index],default=xx}"/>

        <TextView

            . . .
```

```
                android:layout_marginTop="20dp"

                android:padding="20dp"

                android:text="@{map[key],default=yy}"/>

            <Button

                . . .

                android:onClick="onClick"

                android:text="改变数据"/>

        </LinearLayout>

</layout>

private ObservableMap<String, String> map;

        @Override

        protected void onCreate(Bundle savedInstanceState) {

            super.onCreate(savedInstanceState);

            ActivityMain12Binding activityMain12Binding =
DataBindingUtil.setContentView(this, R.layout.activity_main12);
```

```
        map = new ObservableArrayMap<>();

        map.put("name", "leavesC");

        map.put("age", "24");

        activityMain12Binding.setMap(map);

        ObservableList<String> list = new ObservableArrayList<>();

        list.add("Ye");

        list.add("leavesC");

        activityMain12Binding.setList(list);

        activityMain12Binding.setIndex(0);

        activityMain12Binding.setKey("name");

    }

    public void onClick(View view) {

        map.put("name", "leavesC,hi" + new Random().nextInt(100));

    }
```



双向数据绑定

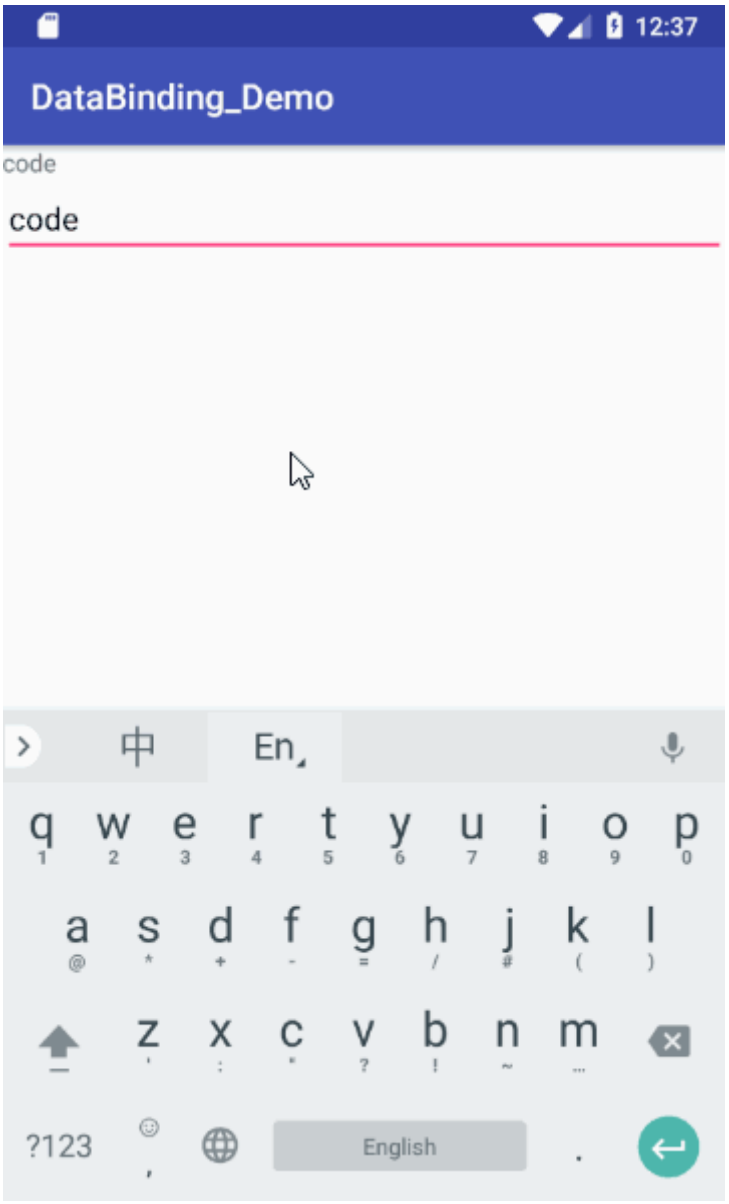
双向绑定的意思即为当数据改变时同时使视图刷新，而视图改变时也可以同时改变数据。

看以下例子，当EditText的输入内容改变时，会同时同步到变量goods，绑定变量的方式比单向绑定多了一个等号：android:text="@={goods.name}"

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <import type="com.leavesc.databinding_demo.model.ObservableGoods"/>
        <variable
            name="goods"
            type="ObservableGoods" />
    </data>
    <LinearLayout
```

```
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical"
            tools:context=".Main10Activity">
            <TextView
                ...
                android:text="@{goods.name}" />
            <EditText
                ...
                android:text="@={goods.name}" />
        </LinearLayout>
</layout>
public class Main10Activity extends AppCompatActivity {
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            ActivityMain10Binding activityMain10Binding =
DataBindingUtil.setContentView(this, R.layout.activity_main10);
            ObservableGoods goods = new ObservableGoods("code", "hi", 23);
```

```
        activityMain10Binding.setGoods(goods);

    }

}
```



事件绑定

严格意义上来说，事件绑定也是一种变量绑定，设置只不过的变量的英文回调接口而已

事件绑定柯林斯用于以下多种回调事件

- android:onClick
- android:onLongClick
- android:afterTextChanged
- android:onTextChanged

- …

在Activity内部新建一个UserPresenter类来声明onClick（）和afterTextChanged（）事件相应的回调方法

```
public class UserPresenter {

        public void onUserNameClick(User user) {

            Toast.makeText(Main5Activity.this, "用户名：" +
user.getName(), Toast.LENGTH_SHORT).show();

        }

        public void afterTextChanged(Editable s) {

            user.setName(s.toString());

            activityMain5Binding.setUserInfo(user);

        }

        public void afterUserPasswordChanged(Editable s) {

            user.setPassword(s.toString());

            activityMain5Binding.setUserInfo(user);

        }

}
```

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

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

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

```
    <data>

        <import type="com.leavesc.databinding_demo.model.User" />

        <import
type="com.leavesc.databinding_demo.MainActivity.UserPresenter" />

        <variable

            name="userInfo"

            type="User" />

        <variable

            name="userPresenter"

            type="UserPresenter" />

    </data>

    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:layout_margin="20dp"

        android:orientation="vertical"

        tools:context="com.leavesc.databinding_demo.MainActivity">

        <TextView

            . . .
```

```
                android:onClick="@{()->userPresenter.onUserNameClick(userInfo)}"
                android:text="@{userInfo.name}" />
            <TextView
                ...
                android:text="@{userInfo.password}" />
            <EditText
                ...
                android:afterTextChanged="@{userPresenter.afterTextChanged}"
                android:hint="用户名" />
            <EditText
                ...
  android:afterTextChanged="@{userPresenter.afterUserPasswordChanged}"
                android:hint="密码" />
        </LinearLayout>
</layout>
```

方法引用的方式与调用函数的方式类似，既可以选择保持事件回调方法的签名一致：@{userPresenter.afterTextChanged}，此时方法名可以不一样，但方法参数和返回值必须和原始的回调函数保持一致。也可以引用不遵循默认签名的函数：@{()->userPresenter.onUserNameClick(userInfo)}，这里用到了Lambda表达式，这样就可以不

遵循默认的方法签名，将userInfo对象直接传回点击方法中。此外，也可以使用方法引用::的形式来进行事件绑定

使用类方法

首先定义一个静态方法

```
public class StringUtils {

    public static String toUpperCase(String str) {

        return str.toUpperCase();

    }
```

```
}
```

在数据标签中导入该工具类

```
<import type="com.leavesc.databinding_demo.StringUtils" />
```

然后就可以像对待一般的函数一样来调用了

```
<TextView

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:onClick="@{()->userPresenter.onUserNameClick(userInfo)}"

        android:text="@{StringUtils.toUpperCase(userInfo.name)}" />
```

运算符

- **基础运算符**

DataBinding支持在布局文件中使用以下运算符，表达式和关键字

- 算术+ - / *%
- 字符串合并+
- 逻辑&& ||
- 二元&| ^
- 一元+ - ！∽
- 移位>> >>> <<
- 比较==> <> = <=
- INSTANCEOF
- 分组（ ）
- 字符，字符串，数字，空
- 投
- 方法调用
- 实地访问
- Array访问[]

- 三元？：

目前不支持以下操作

- 这个
- 超
- 新
- 显示泛型调用

此外，DataBinding还支持以下几种形式的调用

Null Coalescing

空合并运算符??会取第一个不为空的值作为返回值

```
<TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="@{()->userPresenter.onUserNameClick(userInfo)}"
        android:text="@{StringUtils.toUpperCase(userInfo.name)}" />
```

等价于

```
android:text="@{user.name != null ? user.name : user.password}"
```

属性控制

可以通过变量值来控制查看的属性

```
<TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:text="可见性变化"
        android:visibility="@{user.male   ? View.VISIBLE : View.GONE}" />
```

避免空指针异常

DataBinding也会自动帮助我们避免空指针异常。例如，如果“@ {userInfo.password}”中userInfo为null的话，userInfo.password会被赋值为默认值null，而不会抛出空指针异常

include 和 viewStub

- **include**

对于包括的布局文件，一样是支持通过dataBinding来进行数据绑定，此时一样需要在待包括的布局中依然使用布局标签，声明需要使用到的变量

view_include.xml

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <import type="com.leavesc.databinding_demo.model.User" />
        <variable
            name="userInfo"
            type="User" />
    </data>
    <android.support.constraint.ConstraintLayout
        android:layout_width="match_parent"
```

```
            android:layout_height="wrap_content"
            android:background="#acc">
            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:padding="20dp"
                android:text="@{userInfo.name}" />
        </android.support.constraint.ConstraintLayout>
</layout>
```

在主布局文件中将相应的变量传递给包括布局，从而使两个布局文件之间共享同一个变量

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:bind="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools">
        <data>
            <import type="com.leavesc.databinding_demo.model.User" />
            <variable
                name="userInfo"
```

```
            type="User" />

    </data>

    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:orientation="vertical"

        tools:context=".Main6Activity">

        <include

            layout="@layout/view_include"

            bind:userInfo="@{userInfo}" />

    </LinearLayout>

</layout>
```

- **viewStub**

dataBinding一样支持ViewStub布局，在布局文件中引用viewStub布局

```
<ViewStub

        android:id="@+id/view_stub"

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:layout="@layout/view_stub"/>
```

获取到ViewStub对象，由此就可以来控制ViewStub的可见性

```
ActivityMain6Binding activityMain6Binding = DataBindingUtil.setContentView(this,
R.layout.activity_main6);

      View view = activityMain6Binding.viewStub.getViewStub().inflate();
```

如果需要为ViewStub绑定变量值，则ViewStub文件一样要使用布局标签进行布局，主布局文件使用自定义的绑定命名空间将变量传递给ViewStub

```
<ViewStub

            android:id="@+id/view_stub"

            android:layout_width="match_parent"

            android:layout_height="wrap_content"

            android:layout="@layout/view_stub"

            bind:userInfo="@{userInfo}" />
```

如果在xml中没有使用bind:userInfo="@{userInf}"对ViewStub进行数据绑定，则可以等到当ViewStub 膨胀时再绑定变量，此时需要为ViewStub设置setOnInflateListener回调函数，在回调函数中进行数据绑定

```
activityMain6Binding.viewStub.setOnInflateListener(new ViewStub.OnInflateListener() {

                @Override

                public void onInflate(ViewStub stub, View inflated) {

                    //如果在 xml 中没有使用 bind:userInfo="@{userInf}" 对 viewStub 进行数据绑定

                    //那么可以在此处进行手动绑定

                    ViewStubBinding viewStubBinding = DataBindingUtil.bind(inflated);

                    viewStubBinding.setUserInfo(user);
```

```
                Log.e(TAG, "onInflate");
            }
        });
```

BindingAdapter

dataBinding提供了BindingAdapter这个注解用于支持自定义属性，或者是修改原有属性。注解值可以是已有的xml属性，例如android:src，android:text等，也可以自定义属性然后在xml中使用。

例如，对于一个ImageView，我们希望在某个变量值发生变化时，可以动态改变显示的图片，此时可以通过BindingAdapter来实现。

需要先定义一个静态方法，为之添加BindingAdapter注解，注解值是为ImageView控件自定义的属性名，而该静态方法的两个参数可以这样来理解：当ImageView控件的url属性值发生变化时，dataBinding就会将ImageView的实例以及新的url值传递给loadImage（）方法，从而可以在此动态改变ImageView的相关属性。

```
@BindingAdapter({"url"})
    public static void loadImage(ImageView view, String url) {
        Log.e(TAG, "loadImage url : " + url);
    }
```

在xml文件中关联变量值，当中，绑定这个名称可以自定义

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

```
    <data>
        <import type="com.leavesc.databinding_demo.model.Image" />
        <variable
            name="image"
            type="Image" />
    </data>
    <android.support.constraint.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".Main8Activity">
        <ImageView
            android:id="@+id/image"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/ic_launcher_background"
            bind:url="@{image.url}" />
    </android.support.constraint.ConstraintLayout>
</layout>
```

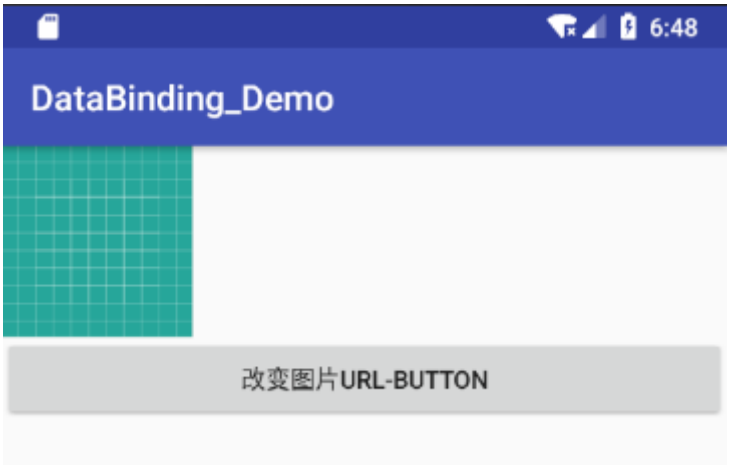
BindingAdapter更加强大的一点是可以覆盖Android原先的控件属性。例如，可以设定每一个Button的文本都要加上后缀：“ - Button”

```
@BindingAdapter("android:text")
```

```
    public static void setText(Button view, String text) {
        view.setText(text + "-Button");
    }
```

```
<Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="@{()->handler.onClick(image)}"
        android:text='@{"改变图片Url"}'/>
```

这样，整个工程中使用到了“android:text”这个属性的控件，其显示的文本就会多出一个后缀



BindingConversion

dataBinding还支持数据进行转换，或者进行类型转换。

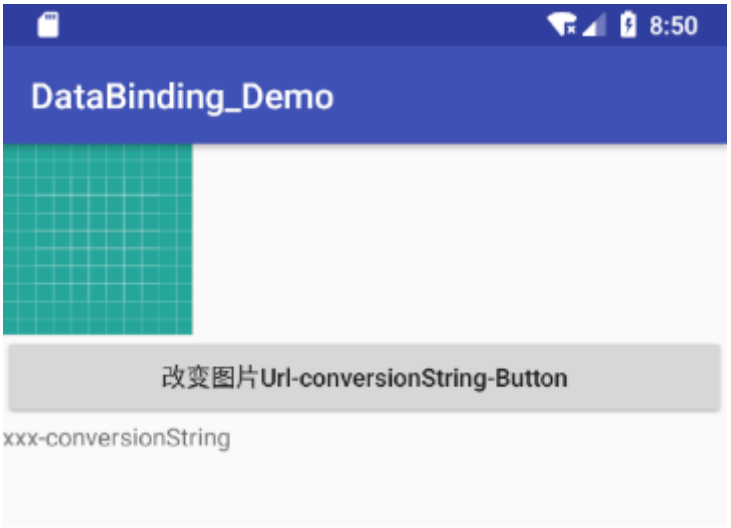
与BindingAdapter类似，以下方法会将布局文件中所有以@{String}方式引用到的String类型变量加上后缀-conversionString

```
@BindingConversion
    public static String conversionString(String text) {
```

```
        return text + "-conversionString";
    }
```

xml文件

```
<TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text='@{"xxx"}'
            android:textAllCaps="false"/>
```



可以看到，对于Button来说，BindingAdapter和BindingConversion同时生效了，而BindingConversion的优先级要高些。此外，BindingConversion也可以用于转换属性值的类型。

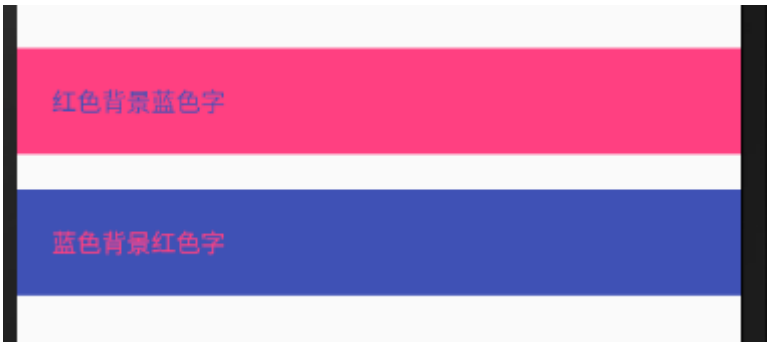
看以下布局，此处在向background和textColor两个属性赋值时，直接就使用了字符串，按正常情况来说这自然是会报错的，但有了BindingConversion后就可以自动将字符串类型的值转为的需要Drawable状语从句：Color了

```
<TextView
            android:layout_width="match_parent"
```

```
            android:layout_height="wrap_content"

            android:background='@{"红色"}'

            android:padding="20dp"

            android:text="红色背景蓝色字"

            android:textColor='@{"蓝色"}'/>

        <TextView

            android:layout_width="match_parent"

            android:layout_height="wrap_content"

            android:layout_marginTop="20dp"

            android:background='@{"蓝色"}'

            android:padding="20dp"

            android:text="蓝色背景红色字"

            android:textColor='@{"红色"}'/>
```

```
@BindingConversion
    public static Drawable convertStringToDrawable(String str) {
        if (str.equals("红色")) {
            return new ColorDrawable(Color.parseColor("#FF4081"));
        }
        if (str.equals("蓝色")) {
            return new ColorDrawable(Color.parseColor("#3F51B5"));
        }
```

```
        return new ColorDrawable(Color.parseColor("#344567"));
    }

    @BindingConversion
    public static int convertStringToColor(String str) {
        if (str.equals("红色")) {
            return Color.parseColor("#FF4081");
        }
        if (str.equals("蓝色")) {
            return Color.parseColor("#3F51B5");
        }
        return Color.parseColor("#344567");
    }
```



Array、List、Set、Map ...

dataBinding也支持在布局文件中使用数组，Lsit，Set和Map，且在布局文件中都可以通过list[index]形式来获取元素。

而为了和变量标签的尖括号区分开，在声明Lsit <String>之类的数据类型时，需要使用尖括号的转义字符。

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <import type="java.util.List" />
        <import type="java.util.Map" />
        <import type="java.util.Set" />
        <import type="android.util.SparseArray" />
        <variable
            name="array"
            type="String[]" />
        <variable
            name="list"
            type="List&lt;String&gt;" />
        <variable
            name="map"
            type="Map&lt;String, String&gt;" />
        <variable
            name="set"
            type="Set&lt;String&gt;" />
        <variable
```

```
            name="sparse"
            type="SparseArray&lt;String&gt;" />
        <variable
            name="index"
            type="int" />
        <variable
            name="key"
            type="String" />
    </data>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        tools:context=".Main7Activity">
        <TextView
            ...
            android:text="@{array[1]}" />
        <TextView
            ...
            android:text="@{sparse[index]}" />
```

```
        <TextView
            ...
            android:text="@{list[index]}" />
        <TextView
            ...
            android:text="@{map[key]}" />
        <TextView
            ...
            android:text='@{map["leavesC"]}' />
        <TextView
            ...
            android:text='@{set.contains("xxx")?"xxx":key}' />
    </LinearLayout>
</layout>
```

资源引用

dataBinding支持对尺寸和字符串这类资源的访问

- **dimens.xml**

```
<dimen name="paddingBig">190dp</dimen>
<dimen name="paddingSmall">150dp</dimen>
```

- **strings.xml**

```
<string name="format">%s is %s</string>
<data>
```

```
        <variable
            name="flag"
            type="boolean" />
    </data>
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingLeft="@{flag ? @dimen/paddingBig:@dimen/paddingSmall}"
        android:text='@{@string/format("leavesC", "Ye")}'
        android:textAllCaps="false" />
```

对DataBinding的介绍到这里也就结束，当然，肯定还有些遗落的知识点，不过大体上我自认也已经讲得很清楚了，剩下的就留待日后补充了。这里提供了上述示例代码的下载，或者你能在GitHub上给个Star？

我的GitHub主页地址如下：

https://github.com/leavesC

项目地址如下：

https://github.com/leavesC/DataBinding_Demo

欢迎长按下图 -> 识别图中二维码

或者 扫一扫 关注我的公众号

阅读原文