# 数据绑定介绍

数据绑定库编写声明式的布局，减少应用中逻辑以及布局所需要的“胶水代码”。

数据绑定库提供了方便与通用性－它是一个支持库，可以在Android 2.1（API级别7+）以上的平台使用。

为了使用数据绑定，gradle插件的版本必须是1.5.0-alpha1以上。

# 中文翻译

<http://yanghui.name/blog/2016/02/17/data-binding-guide/>

<https://blog.gokit.info/post/android-data-binding/>

Github上给出了比较具体代码的讲解：<https://github.com/liangfeidotme/MasteringAndroidDataBinding>

# 搭建开发环境

在Module的build.gradle中增加以下配置：

android {  
    ...  
    dataBinding {  
        enabled = true  
    }  
}

注意：即使Module没有使用DataBinding，但是Module依赖的库使用了DataBinding，则Module也需要进行这个配置。

如果使用Kotlin语言开发，则还需要在Module中配置这个插件：

apply **plugin**: **'kotlin-kapt'**

# 布局和绑定表达式

DataBinding库会自动生成将布局中的视图与数据对象绑定所需要的类。

示例布局：

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.firstName}"/>  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.lastName}"/>  
   </LinearLayout>  
</layout>

## 数据对象

data class User(val firstName: String, val lastName: String)

这种类型的对象具有永不更改的数据。在应用程序中，通常只读取一次数据，此后再也不会更改。

经试验，也可以使用var修饰，在onCreate方法中，设置绑定数据后再改变绑定的数据的属性，运行时看到的效果也是改变之后的。如果设置单击屏幕时再改变属性，则看不到显示效果的改变，说明在onCreate中设置绑定数据时并不是实时设置View的显示内容的。

## 绑定数据

DataBinding库会为每个布局文件生成一个绑定类，默认情况下，类的基于布局名称自动生成，如布局文件为：activity\_main.xml，则对应生成的绑定类为：ActivityMainBinding，即布局的每个下划线分隔的部分开头大写，删除下划线，最后加Binding。

数据绑定示例代码：

override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    val binding: ActivityMainBinding = DataBindingUtil.setContentView(this, R.layout.activity\_main)  
    binding.user = User("Test", "User")  
}

注：不需要调用setContentView方法了。

也可以通过下面的方式获取绑定对象：

val binding: ActivityMainBinding = ActivityMainBinding.inflate(getLayoutInflater())

注：经试验，使用这种方式看不一绑定的内容。后来查看到别人使用时，还需要这样调用才可以：setContentView(binding.getRoot())

注：按Ctrl再点击绑定类是跳转不到绑定类的源代码的，会跳转到绑定类对应的布局文件。绑定类生成的保存位置为：appModule\build\generated\data\_binding\_base\_class\_source\_out\debug\dataBindingGenBaseClassesDebug\out\包名目录\module小写名目录\databinding\ActivityMainBinding.java

如果用在Fragment、ListView或RecyclerView，则使用绑定类的inflate()方法，或者DataBindingUtil的inflate()方法类来完成数据绑定，如下：

val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)  
// or  
val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false)

在Fragment中的使用示例：

@Override

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

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

fragmentLoginBinding.user = new User("Test", "User");

return fragmentBlankBinding.getRoot();

}

在生成的绑定类中可以直接访问布局里带id的控件，如：binding.tvUserName

## 表达式语言

可以在表达式语言中使用以下运算符和关键字：

* 数学运算 + - / \* %
* 字符串连接 +
* 逻辑运算 && ||
* 二进制 & | ^
* 一元运算 + - ! ~
* 位移 >> >>> <<
* 比较 == > < >= <= (请注意， "<" 必须转义为 "&lt;")
* instanceof
* 分组 ()
* 文字 字符, 字符串, 数字, null
* Cast
* 方法调用
* 字段访问
* 数组访问 []
* 三元运算 ?:

示例代码：

android:text="@{String.valueOf(index + 1)}"  
android:visibility="@{age > 13 ? View.GONE : View.VISIBLE}"  
android:transitionName='@{"image\_" + id}'

缺少的操作：

* this
* super
* new
* 显示泛型调用

## 空合并运算符

android:text="@{user.displayName ?? user.lastName}"

如果displayName不为空，则使用displayName，否则使用lastName，在功能上等于：

android:text="@{user.displayName != null ? user.displayName : user.lastName}

## 避免空指针异常

生成的数据绑定代码自动检查null值，并避免空指针异常。例如@{user.name}，如果user为null，则为user.name指定默认值null，如user.age是int类型，则使用默认值为0。

在绑定事件时，如果没有传入事件对象，或者为null，则方法不会被执行，不会出现空指针异常。

## 集合

常用集合，比如arrays、lists、sparse lists和maps，可以使用 [ ] 操作符来进行方便的访问

<data>  
    <import type="android.util.SparseArray"/>  
    <import type="java.util.Map"/>  
    <import type="java.util.List"/>  
    <variable name="list" type="List&lt;String>"/>  
    <variable name="sparse" type="SparseArray&lt;String>"/>  
    <variable name="map" type="Map&lt;String, String>"/>  
    <variable name="index" type="int"/>  
    <variable name="key" type="String"/>  
</data>  
…  
android:text="@{list[index]}"  
…  
android:text="@{sparse[index]}"  
…  
android:text="@{map[key]}"

对于map，也可以使用object.key的方式访问，如上面例子中的@{map[key]}可以替换为@{map.key}

## 字符串文字

可以使用单引号将属性值引起来，这样就可以在表达式中使用双引号了，如下：

android:text='@{map["firstName"]}'

如果使用双引号将属性值引起来，则表达式中的字符串需要使用反引用引起来 `，如下：

android:text="@{map[`firstName`]}"

如果值是一个int，则需要使用String.valueOf()方法转换成String，如下:

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

## 资源

注：网上说这里有Bug，因为DataBinding库会把demen解析成float类型，而这里需要的是一个资源id，如R.dimen.largePadding，这是一个int变量，解析成float就找不到这个资源了。

android:padding="@{large? @dimen/largePadding : @dimen/smallPadding}"

格式字符串和复数形式可以通过提供参数来评估：

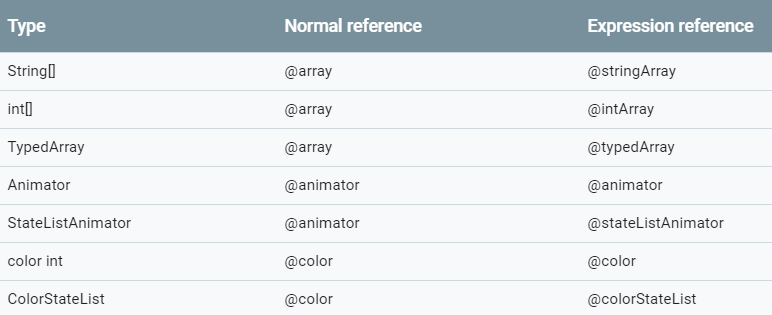
<string name="nameFormat">名字: %s 性别: %s</string>

android:text="@{@string/nameFormat(firstName, lastName)}"  
android:text="@{@plurals/banana(bananaCount)}"

当复数带有多个参数时，应传递所有参数：

Have an orange  
Have %d oranges  
  
android:text="@{@plurals/orange(orangeCount, orangeCount)}"

某些资源需要显式类型评估，如下表所示：



例如，要使用一个Sting[]类型的数组资源时，正常使用是这样的：@array/citys，在绑定表达式中要这样使用：@{@stringArray/citys}

## 默认值

**android:text="@{user.lastName, default = `unknow`}"**/>

## 调用对象方法静态方法

可以在绑定表达式中调用对象的方法，或者静态变量的方法，如下：

public class StringUtils {

public static String toUpperCase(String str) {

return str.toUpperCase();

}

}

<import type="com.leavesc.databinding\_demo.StringUtils" />

<TextView

android:layout\_width="match\_parent"

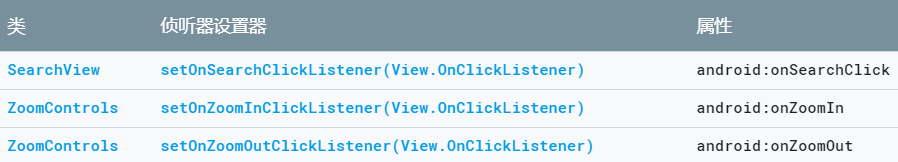
android:layout\_height="wrap\_content"

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

## 事件处理

事件属性名称由侦听器方法的名称确定（比如OnClickListener有onClick（）方法，则事件的属性为android:onClick），但有一些例外。

对于click事件，有一些专门的事件处理程序需要一些属性，而不是android:onClick避免冲突。您可以使用以下属性来避免这些类型的冲突：



您可以使用以下机制来处理事件：

* [方法引用](https://developer.android.com/topic/libraries/data-binding/expressions#method_references)：在表达式中，您可以引用符合侦听器方法签名的方法。当表达式对方法引用求值时，数据绑定将方法引用和所有者对象包装在侦听器中，然后在目标视图上设置该侦听器。如果表达式的计算结果为null，则数据绑定不会创建侦听器而是设置一个null监听器代替。
* [侦听器绑定](https://developer.android.com/topic/libraries/data-binding/expressions#listener_bindings)：这是事件发生时评估的lambda表达式。数据绑定始终创建一个侦听器，并在视图上进行设置。调度事件后，侦听器将评估lambda表达式。

### 方法引用

事件可以直接绑定到处理程序方法，类似于[android:onClick](https://developer.android.com/reference/android/view/View.html#attr_android:onClick) 可以将其分配给活动中的方法的方式。与该[View](https://developer.android.com/reference/android/view/View.html)onClick属性相比，一个主要优点 是在编译时处理表达式，因此，如果该方法不存在或其签名不正确，则会收到编译时错误。

方法引用和侦听器绑定之间的主要区别在于，实际的侦听器实现是在绑定数据时创建的，而不是在事件触发时创建的。如果您希望在事件发生时评估表达式，则应使用[监听器绑定](https://developer.android.com/topic/libraries/data-binding/expressions#listener_bindings)。

要将事件分配给其处理程序，请使用常规绑定表达式，该值是要调用的方法名称。例如，考虑以下示例布局数据对象：

class MyHandlers {  
    fun onClickFriend(view: View) { ... }  
}

绑定表达式可以将视图的点击侦听器分配给 onClickFriend()方法，如下所示：

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
   <data>  
       <variable name="handlers" type="com.example.MyHandlers"/>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.firstName}"  
           android:onClick="@{handlers::onClickFriend}"/>  
   </LinearLayout>  
</layout>

注意：表达式中方法的签名必须与侦听器对象中方法的签名完全匹配。

### 监听器绑定

侦听器绑定是事件发生时运行的绑定表达式。它们类似于方法引用，但是它们使您可以运行任意数据绑定表达式。适用于Gradle 2.0版及更高版本的Android Gradle插件提供了此功能。

在方法引用中，方法的参数必须与事件侦听器的参数匹配。在侦听器绑定中，只有您的返回值必须与侦听器的期望返回值匹配（除非期望返回void）。例如，考虑具有该onSaveClick() 方法的以下presenter类：

class Presenter {  
    fun onSaveClick(task: Task){}  
}

然后，可以将click事件绑定到onSaveClick()方法，如下所示：

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
    <data>  
        <variable name="task" type="com.android.example.Task" />  
        <variable name="presenter" type="com.android.example.Presenter" />  
    </data>  
    <LinearLayout android:layout\_width="match\_parent" android:layout\_height="match\_parent">  
        <Button android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  
        android:onClick="@{() -> presenter.onSaveClick(task)}" />  
    </LinearLayout>  
</layout>

注：() -> 的功用就是创建一个OnClickListener，这里的Lambda省略了接口方法中的参数，这是允许的，在创建的OnClickListener的onClick方法的代码体中会包含这行代码：presenter.onSaveClick(task)

在表达式中使用回调时，数据绑定将自动创建必要的侦听器并将其注册为事件。当视图触发事件时，数据绑定将评估给定的表达式。像在常规绑定表达式中一样，在评估这些侦听器表达式时，您仍然可以获得数据绑定的null和线程安全性。

在上面的示例中，我们尚未定义view传递给的参数[onClick(View)](https://developer.android.com/reference/android/view/View.OnClickListener.html#onClick(android.view.View))。侦听器绑定为侦听器参数提供了两种选择：您可以忽略该方法的所有参数，也可以全部命名。如果您想命名参数，则可以在表达式中使用它们。例如，上面的表达式可以编写如下：

android:onClick="@{(view) -> presenter.onSaveClick(task)}"

或者，如果要在表达式中使用参数，则可以按以下方式工作：

class Presenter {  
    fun onSaveClick(view: View, task: Task){}  
}

android:onClick="@{(theView) -> presenter.onSaveClick(theView, task)}"

您可以使用具有多个参数的lambda表达式：

class Presenter {  
    fun onCompletedChanged(task: Task, completed: Boolean){}  
}

<CheckBox android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  
      android:onCheckedChanged="@{(cb, isChecked) -> presenter.completeChanged(task, isChecked)}" />

如果监听的事件有返回值，则声明的方法也需要对应的返回值，示例如下：

class Presenter {  
    fun onLongClick(view: View, task: Task): Boolean { }  
}

android:onLongClick="@{(theView) -> presenter.onLongClick(theView, task)}"

## 导入、变量和includes

### 导入

<data>  
    <import type="android.view.View"/>  
</data>

<TextView  
   android:text="@{user.lastName}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"  
   android:visibility="@{user.isAdult ? View.VISIBLE : View.GONE}"/>

类型别名：

当类名冲突时，可以将其中一个类重命名为别名：

<import type="android.view.View"/>  
<import type="com.example.real.estate.View"  
        alias="Vista"/>

导入的类型可用作变量和表达式中的类型引用。以下示例显示User并List用作变量的类型：

<data>  
    <import type="com.example.User"/>  
    <import type="java.util.List"/>  
    <variable name="user" type="User"/>  
    <variable name="userList" type="List&lt;User>"/>  
</data>

<TextView  
   android:text="@{((User)(user.connection)).lastName}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

<data>  
    <import type="com.example.MyStringUtils"/>  
    <variable name="user" type="com.example.User"/>  
</data>  
…  
<TextView  
   android:text="@{MyStringUtils.capitalize(user.lastName)}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

java.lang.\*会自动导入

### 变量

变量名不能包含\_，未实验过完。DataBinding会生成setXXX变量的方法。

<data>  
    <import type="android.graphics.drawable.Drawable"/>  
    <variable name="user" type="com.example.User"/>  
    <variable name="image" type="Drawable"/>  
    <variable name="note" type="String"/>  
</data>

在编译时检查变量类型，因此，如果变量实现 [Observable](https://developer.android.com/reference/androidx/databinding/Observable.html)或是 [可观察的集合](https://developer.android.com/topic/libraries/data-binding/observability.html#observable_collections)，则应在类型中反映出来。如果变量是未实现Observable接口的基类或接口，则不会观察到变量。

当针对各种配置（例如，横向或纵向）有不同的布局文件时，将合并变量。这些布局文件之间不得有冲突的变量定义。

对于每个所描述的变量，生成的绑定类都有一个setter和getter。变量将使用默认的托管代码值，直到调用setter为止（null对于引用类型，0对于int，falsefor boolean等等）。

根据需要Databinding库会生成一个名为context的变量，用于绑定表达式，该变量来自根View的getContext()方法，示例如下：

**android:text="@{context.getApplicationInfo().toString()}"**

如果我们自己在data标签中也声明了一个名为context的变量，则会覆盖DataBinding库提供的context变量。

当一个对象拥有getXXX方法时，使用时当成属性一样来访问，如obj.xxx，而不是obj.getXXX()，当在表达式中使用一个对象内的属性时，会分别尝试直接调用、getter、ObservableField.get()

### Includes

includes一个布局时，可能需要把不前布局中的变量传递给include进来的布局中，使用app命名空间的属性来进行传递，如下：

<?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">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <include layout="@layout/name"  
           bind:user="@{user}"/>  
       <include layout="@layout/contact"  
           bind:user="@{user}"/>  
   </LinearLayout>  
</layout>

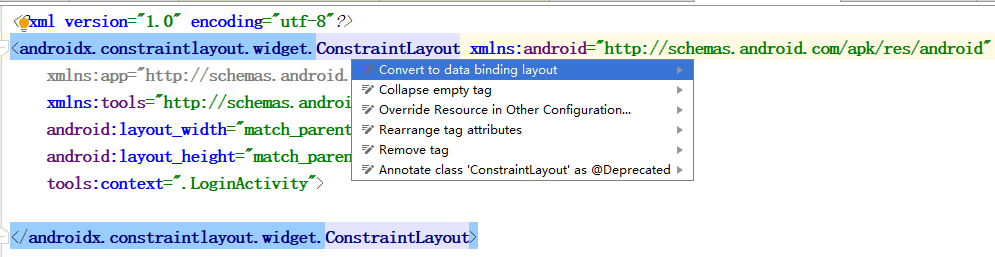
注：这要求在name.xml和contact.xml中也要声明有同样变量名的user声明。也就是说要在多个布局之间传递变量，就需要在这多个布局里面都声明相同类型且名字一样的变量。

数据绑定不支持包含作为merge元素的直接子级。例如，不支持以下布局：

<?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">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <merge><!-- Doesn't work -->  
       <include layout="@layout/name"  
           bind:user="@{user}"/>  
       <include layout="@layout/contact"  
           bind:user="@{user}"/>  
   </merge>  
</layout>

# 快速转换布局为DataBinding

光标定位到普通布局的根标签上，按Alt + 回车，选择“Convert to data binding layout”，如下：



转换后如下：



# 使用可观察的数据对象

使得普通的Java类，当对象的属性修改时，UI不会自动更新，所以需要使用可观察数据对象，这样DataBinding库就可以观察到数据的变化并自动更新UI。

有三种不同类型的可观察类： [对象](https://developer.android.com/topic/libraries/data-binding/observability#observable_objects)，[字段](https://developer.android.com/topic/libraries/data-binding/observability#observable_fields)和 [集合](https://developer.android.com/topic/libraries/data-binding/observability#observable_collections)。

## 可观察字段

创建一个类实现Observable接口需要做一些工作量，如果你的类仅具有一些属性，这么做是不值得的，在这种情况下，可以使用泛型ObservableField类和以下特定于原始类型的类来使字段可观察：

* ObservableBoolean
* ObservableByte
* ObservableChar
* ObservableShort
* ObservableInt
* ObservableLong
* ObservableFloat
* ObservableDouble
* ObservableParcelable

可观察字段是具有单个字段的自包含可观察对象。原始版本避免在访问操作期间装箱和拆箱。要使用此机制，请使用public finalJava编程语言创建一个属性，或使用Kotlin 创建一个只读属性，如以下示例所示：

class User {  
    val firstName = ObservableField<String>()  
    val lastName = ObservableField<String>()  
    val age = ObservableInt()  
}

在代码中访问这些属性时王仲甫和get()和set()，如下：

**mUser**.**lastName**.set(**"Even"**)  
**mUser**.**firstName**.set(**"Dai"**)  
**val** firstName: String? = **mUser**.**firstName**.get()  
**val** age: Int = **mUser**.**age**.get()

在布局中使用这些属性的值时不需要调用get()方法，如下：

<**TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.firstName}"**/>  
<**TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.lastName}"**/>

**注意：**Android Studio 3.1及更高版本允许您用LiveData对象替换可观察字段，这为您的应用程序提供了更多好处。有关更多信息，请参见[使用LiveData通知UI有关数据更改](https://developer.android.com/topic/libraries/data-binding/architecture.html#livedata)

## 可观察的集合

一些应用程序使用动态结构来保存数据。可观察的集合允许使用key访问这些结构。

当key是一个引用类型时[ObservableArrayMap](https://developer.android.com/reference/androidx/databinding/ObservableArrayMap.html) 类是有用的，如下面的例子：

private val user = ObservableArrayMap<String, Any>().apply {  
    put("firstName", "Google")  
    put("lastName", "Inc.")  
    put("age", 17)  
}

这里的key为“firstName”、“lastName”、“age”都是引用类型

当指定key的value发生改变时地，View就会自动更新。

在布局中可能通过map.key的方式获取value

<data>  
    <import type="androidx.databinding.ObservableMap"/>  
    <variable name="user" type="ObservableMap&lt;String, Object>"/>  
</data>  
…  
<TextView  
    android:text="@{user.lastName}"  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>  
<TextView  
    android:text="@{String.valueOf(1 + (Integer)user.age)}"  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>

注：此处user.age要想用于数学运算，则必须强转为Integer，否则编译时会报错，因为编译时把age当Object看，Object不能与1相加。

在当key是一个int类型时[ObservableArrayList](https://developer.android.com/reference/androidx/databinding/ObservableArrayList.html) 类是有用的，如下所示：

注：这里说的key指定的是数组的索引，因为在绑定表达式中可以这样方法数组：@{list[0]}，这里的 0就是一个key，它是一个int类型。

所以map[key]、list[key]，当key是一引用类型时使用map，当key是一int类型时使用list。

val userArray = ObservableArrayList<Any>().apply {  
    add("Google")  
    add("Inc.")  
    add(17)  
}

<data>  
    <import type="android.databinding.ObservableList"/>  
    <import type="com.example.my.app.Fields"/>  
    <variable name="user" type="ObservableList&lt;Object>"/>  
</data>  
…  
<TextView  
    android:text='@{user[Fields.LAST\_NAME]}'  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>  
<TextView  
    android:text='@{String.valueOf(1 + (Integer)user[Fields.AGE])}'  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>

Fields是自己创建的一个类，用字段保存了List中的一些索引，如下：

**public interface** Fields {  
 **int *FIRST\_NAME*** = 0;  
 **int *LAST\_NAME*** = 1;  
 **int *AGE*** = 2;  
}

所以也可以直接在代码中使用android:text='@{user[0]}'来访问数组

## 可观察对象

实现[Observable](https://developer.android.com/reference/androidx/databinding/Observable.html)是需要写一定的代码量的。为了简化开发，数据绑定库提供了[BaseObservable](https://developer.android.com/reference/androidx/databinding/BaseObservable.html)，示例如下：

class User : BaseObservable() {  
  
    @get:Bindable  
    var firstName: String = ""  
        set(value) {  
            field = value  
            notifyPropertyChanged(BR.firstName)  
        }  
  
    @get:Bindable  
    var lastName: String = ""  
        set(value) {  
            field = value  
            notifyPropertyChanged(BR.lastName)  
        }  
}

注：

这个BR有好几个的，导包时要选对，使用androidx.databinding.library.baseAdapters.BR，这里的BR.firstName和BR.lastName是根据User对象的属性生成的。

BR.firstName是一个int型变量，为什么会生成这个变量，是因为在firstName上加了注解：@get:Bindable，意思为Bindable注解是加入到getFirstName方法上的，有了这个注解，Data binding库就会根据getFirstName()方法生成BR.firstName变量，如果没有加这个注解，就不会生成这个变量

notifyPropertyChanged(BR.lastName) 功能为通知lastName属性发了改变了，则UI会自动刷新绑定这个属性的View。还有另一 个方法notifyChange()，用于通知所有的UI更新，而不是BR.lastName指定的某个UI。

数据绑定生成一个BR在模块包中命名的类，该类包含用于数据绑定的资源的ID。该 [Bindable](https://developer.android.com/reference/androidx/databinding/Bindable.html)注释产生的一个条目BR编译期间类文件。如果不能更改数据类的基类，则 [Observable](https://developer.android.com/reference/androidx/databinding/Observable.html)可以使用[PropertyChangeRegistry](https://developer.android.com/reference/androidx/databinding/PropertyChangeRegistry.html) 对象来实现接口， 以高效地注册和通知侦听器。

## 注册属性修改监听器

实现了 Observable 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");

}

}

});

# 生成的绑定类

数据绑定库生成绑定类，这些绑定类用于访问布局的变量和视图。该页面显示了如何创建和定制生成的绑定类。

生成的绑定类将布局变量与布局中的视图链接。绑定类的名称和包可以 [自定义](https://developer.android.com/topic/libraries/data-binding/generated-binding#custom_binding_class_names)。所有生成的绑定类都从[ViewDataBinding](https://developer.android.com/reference/androidx/databinding/ViewDataBinding.html)该类继承 。

为每个布局文件生成一个绑定类。默认情况下，类的名称基于布局文件的名称，将其转换为Pascal大小写并向其添加Binding后缀。上面的布局文件名是， activity\_main.xml因此相应的生成类是 ActivityMainBinding。此类包含从布局属性（例如，user变量）到布局视图的所有绑定，并且知道如何为绑定表达式分配值。

## 创建一个绑定对象

膨胀布局后，应尽快创建绑定对象，以确保在使用布局内的表达式将视图绑定到视图之前，不会修改视图层次结构。将对象绑定到布局的最常见方法是在绑定类上使用静态方法。您可以使用inflate()绑定类的方法来扩展视图层次结构并将对象绑定到该视图层次结构，如以下示例所示：

override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    val binding: MyLayoutBinding = MyLayoutBinding.inflate(layoutInflater)  
}

存在的替代版本inflate()，需要一个方法[ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html)对象中除了[LayoutInflater](https://developer.android.com/reference/android/view/LayoutInflater.html)对象，显示在下面的例子：

val binding: MyLayoutBinding = MyLayoutBinding.inflate(getLayoutInflater(), viewGroup, false)

如果使用其他机制对布局进行了膨胀，则可以单独绑定它，如下所示：

val binding: MyLayoutBinding = MyLayoutBinding.bind(viewRoot)

有时，绑定类型无法预先知道。在这种情况下，可以使用[DataBindingUtil](https://developer.android.com/reference/androidx/databinding/DataBindingUtil.html)类创建绑定 ，如以下代码片段所示：

val viewRoot = LayoutInflater.from(this).inflate(layoutId, parent, attachToParent)  
val binding: ViewDataBinding? = DataBindingUtil.bind(viewRoot)

如果使用的数据绑定内的物品[Fragment](https://developer.android.com/reference/android/app/Fragment.html)， [ListView](https://developer.android.com/reference/android/widget/ListView.html)或[RecyclerView](https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.html)适配器，可能更喜欢使用 [inflate()](https://developer.android.com/reference/androidx/databinding/DataBindingUtil.html#inflate(android.view.LayoutInflater,%20int,%20android.view.ViewGroup,%20boolean,%20android.databinding.DataBindingComponent)) 的绑定的类或方法的 [DataBindingUtil](https://developer.android.com/reference/androidx/databinding/DataBindingUtil)类，如在下面的代码示例：

val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)  
// or  
val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false)

## 具有ID的View

数据绑定库为每个在布局中具有ID的视图在绑定类中创建一个不可变字段。例如，数据绑定库从以下布局创建类型的firstName和lastName字段TextView：

<layout xmlns:android="http://schemas.android.com/apk/res/android">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.firstName}"  
   android:id="@+id/firstName"/>  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.lastName}"  
  android:id="@+id/lastName"/>  
   </LinearLayout>  
</layout>

该库可以单次从视图层次结构中提取包含ID的视图。这种机制比[findViewById()](https://developer.android.com/reference/android/app/Activity.html#findViewById(int))查找views要快。

ID并不是没有数据绑定就没有必要，但是在某些情况下仍然需要从代码访问视图。

## 变量

数据绑定库为布局中声明的每个变量生成访问器方法（setter/getter）：

<data>  
   <import type="android.graphics.drawable.Drawable"/>  
   <variable name="user" type="com.example.User"/>  
   <variable name="image" type="Drawable"/>  
   <variable name="note" type="String"/>  
</data>

## ViewStubs

与普通视图不同，[ViewStub](https://developer.android.com/reference/android/view/ViewStub.html)对象从一个不可见的视图开始。当它们变为可见或被明确告知要膨胀时，它们会通过膨胀另一个布局来替换自身。

因为ViewStub本质上从视图层次结构中消失，所以绑定对象中的视图也必须消失以允许被垃圾回收声明。由于视图是最终视图 [ViewStubProxy](https://developer.android.com/reference/androidx/databinding/ViewStubProxy.html)，因此ViewStub在生成的绑定类中，对象将代替，从而使您可以访问ViewStub存在的，并且可以在膨胀后访问膨胀的视图层次结构ViewStub。

当膨胀另一个布局时，必须为新布局建立一个绑定。因此，ViewStubProxy必须监听ViewStub [OnInflateListener](https://developer.android.com/reference/android/view/ViewStub.OnInflateListener.html)并在需要时建立绑定。由于在给定时间只能存在一个侦听器，因此 ViewStubProxy允许您设置OnInflateListener建立绑定后调用的。

## 立即绑定

当变量或可观察对象发生更改时，绑定将安排在下一帧之前更改。但是，有时绑定必须立即执行。要强制执行，请使用[executePendingBindings()](https://developer.android.com/reference/androidx/databinding/ViewDataBinding.html#executePendingBindings()) 方法。

## 高级绑定

### 动态变量

有时，特定的绑定类是未知的。例如，[RecyclerView.Adapter](https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.Adapter.html)针对任意布局的操作不知道特定的绑定类。在调用[onBindViewHolder()](https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.Adapter.html#onBindViewHolder(VH,%20int))方法期间，它仍然必须分配绑定值。

在下面的示例中，RecyclerView绑定到的所有布局都具有一个 item变量。该BindingHolder对象具有一个getBinding()返回[ViewDataBinding](https://developer.android.com/reference/androidx/databinding/ViewDataBinding.html)基类的方法 。

override fun onBindViewHolder(holder: BindingHolder, position: Int) {  
    item: T = items.get(position)  
    holder.binding.setVariable(BR.item, item);  
    holder.binding.executePendingBindings();  
}

**注意：**数据绑定库生成一个**BR**在模块包中命名的类，该类包含用于数据绑定的资源的ID。在上面的示例中，库自动生成**BR.item**变量。

## 后台线程

您可以在后台线程中更改数据模型，只要它不是集合即可。数据绑定可在评估过程中定位每个变量/字段，以避免任何并发问题。

## 自定义绑定类名称

默认情况下，绑定类基于布局文件的名称生成，从大写字母开始，删除下划线（\_），大写以下字母，并在单词**Binding后加**后缀。该类放在 databinding模块包下面的包中。例如，布局文件 contact\_item.xml生成ContactItemBinding类。如果模块包为com.example.my.app，则将绑定类放置在 com.example.my.app.databinding包中。

通过调整元素的class属性，可以将绑定类重命名或放置在不同的包中 data。例如，以下布局在当前模块ContactItem的databinding包中生成绑定类：

<data class="ContactItem">  
    …  
</data>

您可以通过在类名中添加句点作为前缀来在另一个包中生成绑定类。以下示例在模块包中生成绑定类：

<data class=".ContactItem">  
    …  
</data>

您还可以在要生成绑定类的位置使用完整的程序包名称。以下示例ContactItem在com.example包中创建绑定类 ：

<data class="com.example.ContactItem">  
    …  
</data>

# 绑定适配器

绑定适配器负责进行适当的框架调用以设置值。一个示例是设置属性值，例如调用[setText()](https://developer.android.com/reference/android/widget/TextView.html#setText(char[],%20int,%20int))方法。另一个示例是设置事件侦听器，如调用[setOnClickListener()](https://developer.android.com/reference/android/view/View.html#setOnClickListener(android.view.View.OnClickListener))方法。

数据绑定库允许您指定调用的方法来设置值，提供自己的绑定逻辑以及使用适配器指定返回对象的类型。

## 设置属性值

每当绑定值更改时，生成的绑定类都必须使用绑定表达式在视图上调用setter方法。您可以允许数据绑定库自动确定方法，显式声明该方法或提供自定义逻辑以选择一种方法。

### 自动方法选择

对于名为example的属性，库将自动尝试查找setExample(arg)接受兼容类型作为参数的方法。不考虑属性的名称空间，在搜索方法时仅使用属性名称和类型。

例如，给定android:text="@{user.name}"表达式，该库将寻找一个setText(arg)接受user.getName()所返回类型的方法。如果返回类型user.getName()为String，则库将寻找一个setText()接受String参数的方法。如果表达式返回a int，则库将搜索setText() 接受int参数的方法。表达式必须返回正确的类型，必要时可以转换返回值。

即使给定名称不存在任何属性，数据绑定也有效。然后，您可以使用数据绑定为任何设置器创建属性。例如，支持类[DrawerLayout](https://developer.android.com/reference/androidx/drawerlayout/widget/DrawerLayout.html)没有任何属性，但是有很多设置器。以下布局分别自动将 [setScrimColor(int)](https://developer.android.com/reference/androidx/drawerlayout/widget/DrawerLayout.html#setScrimColor(int))和[setDrawerListener(DrawerListener)](https://developer.android.com/reference/androidx/drawerlayout/widget/DrawerLayout.html#setDrawerListener(android.support.v4.widget.DrawerLayout.DrawerListener))方法用作app:scrimColor和app:drawerListener属性的设置器 ：

<android.support.v4.widget.DrawerLayout  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"  
    app:scrimColor="@{@color/scrim}"  
    app:drawerListener="@{fragment.drawerListener}">

### 指定自定义方法名称

一些属性的设置器名称不匹配。在这些情况下，可以使用[BindingMethods](https://developer.android.com/reference/androidx/databinding/BindingMethods.html) 注释将属性与设置器关联 。该注释与一个类一起使用，并且可以包含多个[BindingMethod](https://developer.android.com/reference/androidx/databinding/BindingMethod.html)注释，每个重命名方法都包含 一个注释。绑定方法是可以添加到应用程序中任何类的注释。在下面的示例中，android:tint属性与[setImageTintList(ColorStateList)](https://developer.android.com/reference/android/widget/ImageView.html#setImageTintList(android.content.res.ColorStateList))方法关联，而不与setTint()方法关联：

@BindingMethods(value = [  
    BindingMethod(  
        type = android.widget.ImageView::class,  
        attribute = "android:tint",  
        method = "setImageTintList")])

大多数时候，您不需要在Android框架类中重命名setter。这些属性已经使用名称约定实现，用于自动查找匹配方法。

### 提供自定义逻辑

一些属性需要自定义绑定逻辑。例如，该android:paddingLeft属性没有关联的设置器。而是setPadding(left, top, right, bottom)提供了该方法。带有[BindingAdapter](https://developer.android.com/reference/androidx/databinding/BindingAdapter.html) 注释的静态绑定适配器方法使您可以自定义如何调用属性的setter。

Android框架类的属性已经BindingAdapter 创建了注释。例如，以下示例显示了该paddingLeft属性的绑定适配器：

@BindingAdapter("android:paddingLeft")  
fun setPaddingLeft(view: View, padding: Int) {  
    view.setPadding(padding,  
                view.getPaddingTop(),  
                view.getPaddingRight(),  
                view.getPaddingBottom())  
}

参数类型很重要。第一个参数确定与属性关联的视图的类型。第二个参数确定给定属性在绑定表达式中接受的类型。

绑定适配器对于其他类型的定制很有用。例如，可以从工作线程调用自定义加载程序以加载映像。

发生冲突时，您定义的绑定适配器将覆盖Android框架提供的默认适配器。

您还可以具有接收多个属性的适配器，如以下示例所示：

@BindingAdapter("imageUrl", "error")  
fun loadImage(view: ImageView, url: String, error: Drawable) {  
    Picasso.get().load(url).error(error).into(view)  
}

您可以在布局中使用适配器，如以下示例所示。请注意，它@drawable/venueError指的是您应用中的资源。围绕资源@{}使其成为有效的绑定表达式。

<ImageView app:imageUrl="@{venue.imageUrl}" app:error="@{@drawable/venueError}" />

**注意：**数据绑定库出于匹配目的会忽略自定义名称空间。

该适配器被称为如果同时imageUrl和error用于一个[ImageView](https://developer.android.com/reference/android/widget/ImageView.html)对象，imageUrl是一个字符串，error是一个[Drawable](https://developer.android.com/reference/android/graphics/drawable/Drawable.html)。如果希望在设置任何属性时调用适配器，则可以将适配器的可选[requireAll](https://developer.android.com/reference/androidx/databinding/BindingAdapter.html#requireAll()) 标志设置 为false，如以下示例所示：

@BindingAdapter(value = ["imageUrl", "placeholder"], requireAll = false)  
fun setImageUrl(imageView: ImageView, url: String?, placeHolder: Drawable?) {  
    if (url == null) {  
        imageView.setImageDrawable(placeholder);  
    } else {  
        MyImageLoader.loadInto(imageView, url, placeholder);  
    }  
}

**注意：**发生冲突时，您的绑定适配器将覆盖默认的数据绑定适配器。

绑定适配器方法可以选择在其处理程序中采用旧值。采用旧值和新值的方法应首先声明属性的所有旧值，然后再声明新值，如以下示例所示：

@BindingAdapter("android:paddingLeft")  
fun setPaddingLeft(view: View, oldPadding: Int, newPadding: Int) {  
    if (oldPadding != newPadding) {  
        view.setPadding(padding,  
                    view.getPaddingTop(),  
                    view.getPaddingRight(),  
                    view.getPaddingBottom())  
    }  
}

事件处理程序只能与具有一个抽象方法的接口或抽象类一起使用，如以下示例所示：

@BindingAdapter("android:onLayoutChange")  
fun setOnLayoutChangeListener(  
        view: View,  
        oldValue: View.OnLayoutChangeListener?,  
        newValue: View.OnLayoutChangeListener?  
) {  
    if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.HONEYCOMB) {  
        if (oldValue != null) {  
            view.removeOnLayoutChangeListener(oldValue)  
        }  
        if (newValue != null) {  
            view.addOnLayoutChangeListener(newValue)  
        }  
    }  
}

在您的布局中使用此事件处理程序，如下所示：

<View android:onLayoutChange="@{() -> handler.layoutChanged()}"/>

当一个侦听器有多种方法时，必须将其拆分为多个侦听器。例如，[View.OnAttachStateChangeListener](https://developer.android.com/reference/android/view/View.OnAttachStateChangeListener.html)有两种方法：[onViewAttachedToWindow(View)](https://developer.android.com/reference/android/view/View.OnAttachStateChangeListener.html#onViewAttachedToWindow(android.view.View))和[onViewDetachedFromWindow(View)](https://developer.android.com/reference/android/view/View.OnAttachStateChangeListener.html#onViewDetachedFromWindow(android.view.View))。该库提供了两个接口来区分它们的属性和处理程序：

// Translation from provided interfaces in Java:  
@TargetApi(Build.VERSION\_CODES.HONEYCOMB\_MR1)  
interface OnViewDetachedFromWindow {  
    fun onViewDetachedFromWindow(v: View)  
}  
  
@TargetApi(Build.VERSION\_CODES.HONEYCOMB\_MR1)  
interface OnViewAttachedToWindow {  
    fun onViewAttachedToWindow(v: View)  
}

因为更改一个侦听器也会影响另一个侦听器，所以您需要一个适用于这两个属性或两者都适用的适配器。您可以在批注中设置 [requireAll](https://developer.android.com/reference/androidx/databinding/BindingAdapter.html#requireAll()) 为false，以指定并非必须为每个属性都分配一个绑定表达式，如以下示例所示：

@BindingAdapter(  
        "android:onViewDetachedFromWindow",  
        "android:onViewAttachedToWindow",  
        requireAll = false  
)  
fun setListener(view: View, detach: OnViewDetachedFromWindow?, attach: OnViewAttachedToWindow?) {  
    if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.HONEYCOMB\_MR1) {  
        val newListener: View.OnAttachStateChangeListener?  
        newListener = if (detach == null && attach == null) {  
            null  
        } else {  
            object : View.OnAttachStateChangeListener {  
                override fun onViewAttachedToWindow(v: View) {  
                    attach.onViewAttachedToWindow(v)  
                }  
  
                override fun onViewDetachedFromWindow(v: View) {  
                    detach.onViewDetachedFromWindow(v)  
                }  
            }  
        }  
  
        val oldListener: View.OnAttachStateChangeListener? =  
                ListenerUtil.trackListener(view, newListener, R.id.onAttachStateChangeListener)  
        if (oldListener != null) {  
            view.removeOnAttachStateChangeListener(oldListener)  
        }  
        if (newListener != null) {  
            view.addOnAttachStateChangeListener(newListener)  
        }  
    }  
}

上面的示例比普通示例稍微复杂一点，因为[View](https://developer.android.com/reference/android/view/View.html)该类使用[addOnAttachStateChangeListener()](https://developer.android.com/reference/android/view/View.html#addOnAttachStateChangeListener(android.view.View.OnAttachStateChangeListener))and [removeOnAttachStateChangeListener()](https://developer.android.com/reference/android/view/View.html#removeOnAttachStateChangeListener(android.view.View.OnAttachStateChangeListener))方法代替的setter方法 [OnAttachStateChangeListener](https://developer.android.com/reference/android/view/View.OnAttachStateChangeListener.html)。该android.databinding.adapters.ListenerUtil 班可以帮助跟踪以前的听众，让他们可以在绑定的适配器中删除。

通过注释的接口OnViewDetachedFromWindow和 OnViewAttachedToWindow用@TargetApi(VERSION\_CODES.HONEYCOMB\_MR1)，数据绑定代码发生器知道，在Android 3.1（API级12）和更高，由支持的相同版本上运行时，听者应该只产生[addOnAttachStateChangeListener()](https://developer.android.com/reference/android/view/View.html#addOnAttachStateChangeListener(android.view.View.OnAttachStateChangeListener))方法。

## 对象转换

### 自动对象转换

当一个[Object](https://developer.android.com/reference/java/lang/Object.html)从绑定表达式返回时，库将选择用于设置属性值的方法。将其转换Object 为所选方法的参数类型。在使用[ObservableMap](https://developer.android.com/reference/androidx/databinding/ObservableMap.html)该类存储数据的应用程序中，此行为很方便 ，如以下示例所示：

<TextView  
   android:text='@{userMap["lastName"]}'  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content" />

**注意：**您也可以使用**object.key**符号在map中引用一个值。例如，**@{userMap["lastName"]}**在上面的示例中，可以将替换为 **@{userMap.lastName}**。

### 自定义转化

在某些情况下，需要在特定类型之间进行自定义转换。例如，android:background视图的属性需要一个[Drawable](https://developer.android.com/reference/android/graphics/drawable/Drawable.html)，但是color指定的值是一个整数。下面的示例显示了一个期望使用的属性Drawable，但是提供了一个整数：

<View  
   android:background="@{isError ? @color/red : @color/white}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

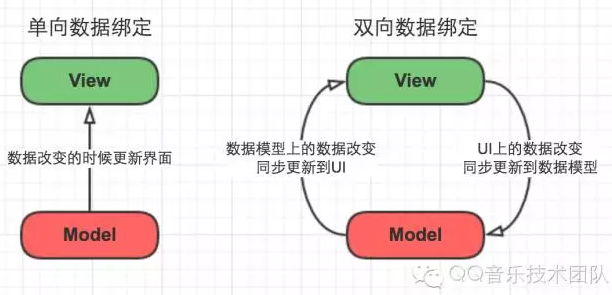
只要Drawable期望a并返回整数，int则应将其转换为a [ColorDrawable](https://developer.android.com/reference/android/graphics/drawable/ColorDrawable.html)。可以使用带有[BindingConversion](https://developer.android.com/reference/androidx/databinding/BindingConversion.html) 注释的静态方法来完成转换 ，如下所示：

@BindingConversion  
fun convertColorToDrawable(color: Int) = ColorDrawable(color)

但是，绑定表达式中提供的值类型必须一致。不能在同一表达式中使用不同的类型，如以下示例所示：

<View  
   android:background="@{isError ? @drawable/error : @color/white}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

# 将布局视图绑定到架构组件



AndroidX库包含[Architecture Architecture](https://developer.android.com/topic/libraries/architecture/index.html)，您可以使用它们来设计健壮，可测试和可维护的应用程序。数据绑定库可与体系结构组件无缝协作，以进一步简化UI的开发。应用程序中的布局可以绑定到架构组件中的数据，这已经帮助您管理UI控制器的生命周期并通知有关数据中的更改。

该页面显示了如何将架构组件合并到您的应用程序中，以进一步增强使用数据绑定库的好处。

## 使用LiveData通知UI有关数据更改

您可以将[LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData)对象用作数据绑定源，以将数据更改自动通知UI。有关此体系结构组件的更多信息，请参见[LiveData概述](https://developer.android.com/topic/libraries/architecture/livedata)。

不同于实施对象 [Observable](https://developer.android.com/reference/androidx/databinding/Observable.html)-如 [观察到的领域](https://developer.android.com/topic/libraries/data-binding/observability.html#observable_fields) - LiveData 对象了解订阅数据的变化观察员的生命周期。这些知识带来了很多好处，这些[好处在使用LiveData的好处中进行了说明](https://developer.android.com/topic/libraries/architecture/livedata.html#the_advantages_of_using_livedata)。在Android的Studio版本3.1及更高版本，可以取代 [观察到的领域](https://developer.android.com/topic/libraries/data-binding/observability.html#observable_fields) 与LiveData您的数据对象绑定代码。

要将LiveData对象与绑定类一起使用，您需要指定生命周期所有者来定义LiveData对象的范围。以下示例在实例化绑定类之后将活动指定为生命周期所有者：

class ViewModelActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        // Inflate view and obtain an instance of the binding class.  
        val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)  
  
        // Specify the current activity as the lifecycle owner.  
        binding.setLifecycleOwner(this)  
    }  
}

您可以使用[ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel) 组件（如[使用ViewModel来管理UI相关的数据中](https://developer.android.com/topic/libraries/data-binding/architecture#viewmodel)所述）将数据绑定到布局。在ViewModel组件中，您可以使用LiveData对象来转换数据或合并多个数据源。以下示例显示了如何转换中的数据ViewModel：

class ScheduleViewModel : ViewModel() {  
    val userName: LiveData  
  
    init {  
        val result = Repository.userName  
        userName = Transformations.map(result) { result -> result.value }  
    }  
}

## 使用ViewModel管理与UI相关的数据

数据绑定库与[ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel)组件无缝 协作，这些组件公开了布局观察到的数据并对更改做出反应。通过将 ViewModel组件与数据绑定库一起使用，可以将UI逻辑移出布局并移入组件中，这更易于测试。数据绑定库可确保在需要时将视图与数据源绑定和解除绑定。剩下的大部分工作就是确保您公开了正确的数据。有关此体系结构组件的更多信息，请参见[ViewModel Overview](https://developer.android.com/topic/libraries/architecture/viewmodel.html)。

要将ViewModel组件与数据绑定库一起使用，必须实例化从ViewModel类继承的组件， 获取绑定类的实例，然后将ViewModel组件分配给绑定类中的属性。以下示例显示如何将组件与库一起使用：

class ViewModelActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        // Obtain the ViewModel component.  
        UserModel userModel = ViewModelProviders.of(getActivity())  
                                                  .get(UserModel.class)  
  
        // Inflate view and obtain an instance of the binding class.  
        val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)  
  
        // Assign the component to a property in the binding class.  
        binding.viewmodel = userModel  
    }  
}

在布局中，ViewModel使用绑定表达式将组件的属性和方法分配给相应的视图，如以下示例所示：

<CheckBox  
    android:id="@+id/rememberMeCheckBox"  
    android:checked="@{viewmodel.rememberMe}"  
    android:onCheckedChanged="@{() -> viewmodel.rememberMeChanged()}" />

## 使用Observable ViewModel可以更好地控制绑定适配器

您可以使用[ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel.html) 实现的 [Observable](https://developer.android.com/reference/androidx/databinding/Observable.html)组件，以将数据更改通知其他应用程序组件，类似于使用 [LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData)对象的方式。

在某些情况下，即使您失去了的生命周期管理功能，也可能更喜欢使用[ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel.html)实现[Observable](https://developer.android.com/reference/androidx/databinding/Observable.html) 接口的 组件而不是使用LiveData对象LiveData。使用ViewModel实现的组件，Observable您可以更好地控制应用程序中的绑定适配器。例如，此模式使您可以在数据更改时更好地控制通知，它还允许您指定自定义方法来设置双向数据绑定中的属性值。

要实现可观察的ViewModel组件，必须创建一个从[ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel.html)该类继承并实现该[Observable](https://developer.android.com/reference/androidx/databinding/Observable.html) 接口的 类。当观察者使用[addOnPropertyChangedCallback()](https://developer.android.com/reference/androidx/databinding/Observable.html#addOnPropertyChangedCallback(android.databinding.Observable.OnPropertyChangedCallback)) 和 [removeOnPropertyChangedCallback()](https://developer.android.com/reference/androidx/databinding/Observable.html#removeOnPropertyChangedCallback(android.databinding.Observable.OnPropertyChangedCallback)) 方法订阅或取消订阅通知时，可以提供自定义逻辑 。您还可以提供在[notifyPropertyChanged()](https://developer.android.com/reference/androidx/databinding/BaseObservable.html#notifyPropertyChanged(int)) 方法的属性更改时运行的自定义逻辑 。以下代码示例演示如何实现可观察的 ViewModel：

/\*\*  
 \* A ViewModel that is also an Observable,  
 \* to be used with the Data Binding Library.  
 \*/  
open class ObservableViewModel : ViewModel(), Observable {  
    private val callbacks: PropertyChangeRegistry = PropertyChangeRegistry()  
  
    override fun addOnPropertyChangedCallback(  
            callback: Observable.OnPropertyChangedCallback) {  
        callbacks.add(callback)  
    }  
  
    override fun removeOnPropertyChangedCallback(  
            callback: Observable.OnPropertyChangedCallback) {  
        callbacks.remove(callback)  
    }  
  
    /\*\*  
     \* Notifies observers that all properties of this instance have changed.  
     \*/  
    fun notifyChange() {  
        callbacks.notifyCallbacks(this, 0, null)  
    }  
  
    /\*\*  
     \* Notifies observers that a specific property has changed. The getter for the  
     \* property that changes should be marked with the @Bindable annotation to  
     \* generate a field in the BR class to be used as the fieldId parameter.  
     \*  
     \* @param fieldId The generated BR id for the Bindable field.  
     \*/  
    fun notifyPropertyChanged(fieldId: Int) {  
        callbacks.notifyCallbacks(this, fieldId, null)  
    }  
}

# 双向数据绑定

使用单向数据绑定，您可以在属性上设置一个值，并设置一个对该属性的更改做出反应的侦听器：

<CheckBox  
    android:id="@+id/rememberMeCheckBox"  
    **android:checked="@{viewmodel.rememberMe}"  
    android:onCheckedChanged="@{viewmodel.rememberMeChanged}"**  
/>

双向数据绑定为该过程提供了捷径：

<CheckBox  
    android:id="@+id/rememberMeCheckBox"  
    **android:checked="@={viewmodel.rememberMe}"**  
/>

“=”符号的功能就是实现双向绑定，使用双向相等，如对象属性改变时更新ui，并监听ui数据变化时保存到对象属性

为了对后勤数据的变化做出反应，您可以将布局变量实现为Observable，通常是 [BaseObservable](https://developer.android.com/reference/androidx/databinding/BaseObservable)，并使用[@Bindable](https://developer.android.com/reference/androidx/databinding/Bindable)批注，如以下代码片段所示：

class LoginViewModel : **BaseObservable** {  
    // val data = ...  
  
    **@Bindable**  
    fun getRememberMe(): Boolean {  
        return data.rememberMe  
    }  
  
    fun setRememberMe(value: Boolean) {  
        // 避免无限循环  
        if (data.rememberMe != value) {  
            data.rememberMe = value  
  
            // 对变化做出反应  
            saveData()  
  
            // 通知观察者有一个新的值  
            notifyPropertyChanged(BR.rememberMe)  
        }  
    }  
}

注：实验证明，在setRemember方法中不做if判断，也并不会出现无限循环。

因为可绑定属性的getter方法被调用getRememberMe()，所以属性的相应setter方法会自动使用名为setRememberMe()。

有关使用BaseObservable和的更多信息@Bindable，请参见[使用可观察的数据对象](https://developer.android.com/topic/libraries/data-binding/observability)。

## 使用自定义属性的双向数据绑定

该平台为[最常见的双向属性](https://developer.android.com/topic/libraries/data-binding/two-way#two-way-attributes)和更改侦听器提供了双向数据绑定实现，您可以将其用作应用程序的一部分。如果要对自定义属性使用双向数据绑定，则需要使用 [@InverseBindingAdapter](https://developer.android.com/reference/androidx/databinding/InverseBindingAdapter) 和 [@InverseBindingMethod](https://developer.android.com/reference/androidx/databinding/InverseBindingMethod) 批注。

例如，如果要在自定义视图MyView的time"属性启用双向数据绑定，请完成以下步骤：

1、使用以下方法注释设置初始值并在值更改时更新的方法@BindingAdapter：

**@BindingAdapter("time")**  
@JvmStatic fun setTime(view: MyView, newValue: Time) {  
    // 重要：避免无限循环  
    if (view.time != newValue) {  
        view.time = newValue  
    }  
}

注：这里的判断是为了预防无限循环，如：数据改变了设置给View，View更新了又导致把新数据设置给Data对象，设置Data后又会更新View，一直死循环。

2、注释使用以下方法从视图中读取值的方法 @InverseBindingAdapter：

**@InverseBindingAdapter("time")**  
@JvmStatic fun getTime(view: MyView) : Time {  
    return view.getTime()  
}

此时，DataBinding知道数据更改时要做什么（它调用带有 [@BindingAdapter](https://developer.android.com/reference/androidx/databinding/BindingAdapter)注释的方法），以及当View的属性改变时调用什么（它调用[InverseBindingListener](https://developer.android.com/reference/androidx/databinding/InverseBindingListener)）。但是，DataBinding不知道属性何时改变。

为此，您需要在视图上设置一个侦听器。它可以是与自定义视图关联的自定义侦听器，也可以是通用事件，例如失去焦点或更改文本。将@BindingAdapter注释添加到设置侦听器以更改属性的方法：

**@BindingAdapter("app:timeAttrChanged")**  
@JvmStatic fun setListeners(  
        view: MyView,  
        **attrChange: InverseBindingListener**  
) {  
    // Set a listener for click, focus, touch, etc.  
}

侦听器包含一个InverseBindingListeneras作为参数。您可以使用 InverseBindingListener告诉数据绑定系统属性已更改。然后，系统可以开始调用使用注释的方法 @InverseBindingAdapter，依此类推。

**注意：**每个双向绑定都会生成一个*合成事件属性*。此属性与基本属性具有相同的名称，但包含后缀 **"AttrChanged"**。合成事件属性允许库创建带注释的方法，该方法**@BindingAdapter**用于将事件侦听器与的适当实例相关联**View**。

实际上，此侦听器包括一些非平凡的逻辑，包括用于单向数据绑定的侦听器。有关示例，请参见用于文本属性更改的适配器 [TextViewBindingAdapter](https://android.googlesource.com/platform/frameworks/data-binding/+/refs/heads/studio-master-dev/extensions/baseAdapters/src/main/java/androidx/databinding/adapters/TextViewBindingAdapter.java#352)。

## 转换器

如果绑定到[View](https://developer.android.com/reference/android/view/View)对象的变量在显示之前需要进行格式化，转换或更改，则可以使用Converter对象。

例如，拿一个EditText显示日期的对象：

<EditText  
    android:id="@+id/birth\_date"  
    android:text="@={Converter.dateToString(viewmodel.birthDate)}"  
/>

该viewmodel.birthDate属性包含类型的值Long，因此需要使用转换器对其进行格式化。

因为使用的是双向表达式，所以在这种情况下，还需要有一个*逆转换器*以使库知道如何将用户提供的字符串转换回支持数据类型Long。通过将[@InverseMethod](https://developer.android.com/reference/androidx/databinding/InverseMethod)注释添加到一个转换器中并让该注释引用反向转换器来完成此过程。以下代码片段中显示了此配置的示例：

object Converter {  
    **@InverseMethod("stringToDate")**  
    fun dateToString(  
        view: EditText, oldValue: Long,  
        value: Long  
    ): String {  
        // Converts long to String.  
    }  
  
    fun stringToDate(  
        view: EditText, oldValue: String,  
        value: String  
    ): Long {  
        // Converts String to long.  
    }  
}

## 使用双向数据绑定的无限循环

使用双向数据绑定时，请注意不要引入无限循环。当用户更改属性时，将使用注释的方法 @InverseBindingAdapter，并将该值分配给backing属性。反过来，这将调用使用注释的方法 @BindingAdapter，这将触发对使用注释的方法的另一次调用@InverseBindingAdapter，依此类推。

因此，通过在使用注释的方法中比较新值和旧值来打破可能的无限循环非常重要@BindingAdapter。

## 双向属性

当您使用下表中的属性时，该平台为双向数据绑定提供内置支持。有关平台如何提供此支持的详细信息，请参见相应的绑定适配器的实现：



# 掘金上的

## 相关连接

<https://juejin.im/post/5b02cf8c6fb9a07aa632146d#heading-2>

<https://mp.weixin.qq.com/s?__biz=MzI1NjEwMTM4OA==&mid=2651232170&idx=1&sn=f4d7eb8f35ebf3b13696562ca3172bac&chksm=f1d9eac9c6ae63df357c3a96aa0218b5d66237c5411de5b34cd24ddb7a1d258b34444966d8c6&scene=0#rd>

Data Binding高级篇：<http://blog.zhaiyifan.cn/2016/07/06/android-new-project-from-0-p8/>

## 双向数据绑定

## 引用其他控件的值

<**TextView  
 android:id="@+id/tv\_first\_name"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.firstName}"**/>  
  
<**TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{tvFirstName.text}"**/>

注：这里第二个TextView引用了第一个TextView的值，第一个TextView的id为“tv\_first\_name”，在引用它的值时，如果是带下划线的id名，则需要去掉下划线，且从第二个单词开始单词首字母大写，否则编译通不过。

## 使用Class

如果想用Class作为参数传递, 那么该Class不能直接通过静态导入来使用. 需要作为字段常量来使用

## 函数回调

DataBinding还支持在XML中绑定函数参数类型, 并且还是Lambda和高阶函数类型, 这点比Java还先进.

### 对象

即直接将对象作为和属性等同的方式在XML使用. 这就必须先手动创建一个对象. 稍显麻烦.