@:Databinding

Databinding

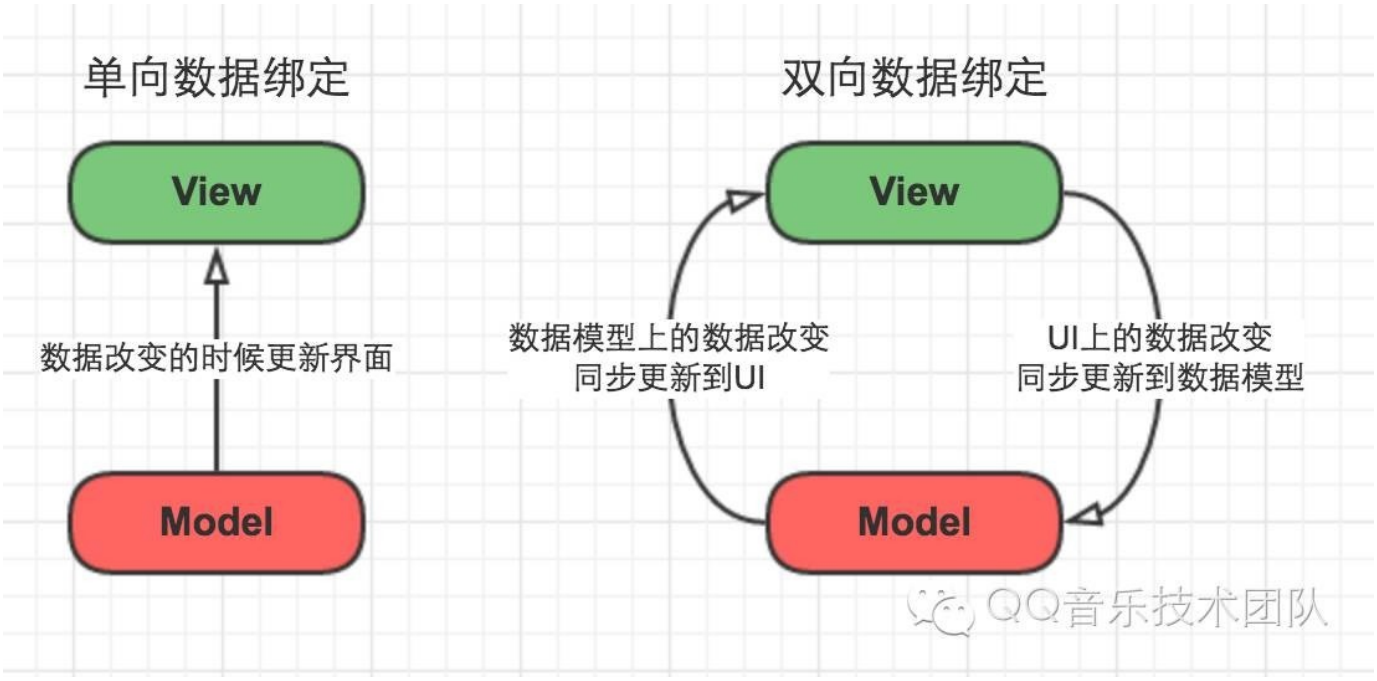
介绍

Databinding数据绑定，简单的说，就是通过某种机制，把代码中的数据和xml(UI)绑定起来，双方都能对数据进行操作，并且在数据发生变化的时候，可以自动刷新数据。

数据绑定方式

单向绑定
单向绑定就是说数据的流向是单方面的，只能从代码流向UI；

双向绑定
双向绑定的数据流向是双向的，当业务代码中的数据改变时，UI上的数据能够得到刷新；当用户通过UI交互编辑了数据时，数据的变化也能自动的更新到业务代码中的数据上。



优势

1.性能很好，因为没有反射，而且性能比直接findViewById要高。
2.谷歌原生支持
3.代码简洁
4.减少代码线程切换更新UI
5.自动检查空指针

局限性

1.与某些技术冲突。比如dragger2，插件化技术，热修复技术。
2.databinding依赖与XML布局，databinding插件会根据xml文件内容生成代码，因此无法支持在代码中动态生成View
3.Databinding库要求安卓的gradle插件最低是1.5.0
4.Databinding要求安卓平台2.1（Api 7+）
5.Android Studio版本要在1.3之上

使用介绍

配置

在app级别的build.gradle文件中加入DataBinding元素，点击同步即可

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

DataBinding的布局文件格式

activity_main.layout

```
<?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>
```

DataBinding的布局文件和原来的布局稍有些不同，它的根布局是一个layout标签，里边包含一个data标签和原来的根布局。
data标签中可以import类名

```
    <data>
        <import type="com.example.peiyu_wang.databinding.entity.StudentObserve" />

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

        <variable
            name="stu"
            type="StudentObserve" />

        <variable
            name="clicklistener"
            type="OnClickListener" />

    </data>
```

```
<variable name="user" type="com.example.User"/>
```

在data标签中的user变量在你的布局文件中使用。

```
<TextView android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@{user.firstName}"/>
```

布局文件中的表达式要写在属性中，并写成"@{}"形式。这个例子中，TextView中的文字被设置成为了user的firstName属性。
点击build->makeProject

> 此时DataBinding就会自动生成ViewDataBinding类的子类，类名根据layout文件的名称而生成。如现在的layout文件名是activity_main.xml，因此生成的类名是ActivityMainBinding。能不能自定义生成类名?当然可以。

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

> android:text="@{user.firstName}",这里写的是user.firstName,其实是调用了user.getFirstName方法。更多介绍请看实现原理。。。

布局写好后，回到MainActivity

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activi
    User user = new User("Test", "User");
    binding.setUser(user);
}
```

这样就完成了DataBinding的数据绑定，现在运行你应用程序就可以得到你设置的结果。

DataBind + 事件

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <import type="com.example.peiyu_wang.databinding.entity.StudentObserve" />
        <import type="android.view.View.OnClickListener" />
        <variable
            name="stu"
            type="StudentObserve" />
        <variable
            name="clicklistener"
            type="OnClickListener" />
    </data>
    <LinearLayout xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        tools:context="com.example.peiyu_wang.databinding.MainActivity">
        <TextView
            android:layout_width="match_parent"
            android:layout_height="40dp"
            android:gravity="center"
            android:text="@{stu.name}"
            android:textSize="16sp" />
        <TextView
            android:layout_width="match_parent"
            android:layout_height="40dp"
            android:gravity="center"
            android:text="@{stu.address}"
            android:textSize="16sp" />
        <!--事件绑定-->
        <Button
            android:id="@+id/change"
            android:layout_width="match_parent"
            android:layout_height="50dp"
            android:onClick="@{clicklistener::onClick}"
            android:text="change Text" />
    </LinearLayout>
</layout>
```

可以看到xml中定义类Button的点击事件，。其实就是调用类clicklistener.onClick(View v)方法，此时不一定需要时View.OnClickListener

BindingAdapter

android:text="@{user.firstName}"这种情况下
DataBinding框架的处理过程分成三步

>
> 1.对binding表达式求值
> 2.寻找合适的BindingAdapter，如果找到，就调用它的方法

> 3.如果没有找到合适的BindingAdapter，就在View上寻找合适的方法调用

因此默认情况下，会调用android命名空间下的View.setText进行更新数据，那么此时我想进行这样的一个操作，当TextView里面的内容没有变化的时候，就不进行更新，更新也一样嘛。因此我们重新设置我们自己的BindingAdapter，此时在User中定义自己的处理android:text的处理方法

```
@BindingAdapter("android:text")
    public static void setText(TextView view, CharSequence text) {
        final CharSequence oldText = view.getText();
        if (text == oldText || (text == null && oldText.length() == 0)) {
            return;
        }
        if (text instanceof Spanned) {
            if (text.equals(oldText)) {
                return; // No change in the spans, so don't set anything.
            }
        } else if (!haveContentsChanged(text, oldText)) {
            return; // No content changes, so don't set anything.
        }
        view.setText(text);
    }
```

DataBinding UI自动更新数据

ActivityMainBinding.invalidateAll()

此时的数据对象，无需其他修饰.如：

```
public class Student {
    private String name;
    private String address;
    private String company;
    public Student() {
    }
    public Student(String name, String address, String company) {
        this.name = name;
        this.address = address;
        this.company = company;
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public String getCompany() {
        return company;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public void setCompany(String company) {
        this.company = company;
    }
}
```

在改变数据后，调用对应ViewDataBinding子类的invalidateAll()方法。此时需要手动调用刷新

将数据类继承自android.databinding.BaseObservable

```
public class StudentObserve extends BaseObservable {
    private String name;
    private String address;
    private String phone;
    public StudentObserve() {
    }
    public StudentObserve(String name, String address, String phone) {
        this.name = name;
        this.address = address;
        this.phone = phone;
    }
    @Bindable
    public String getName() {
        return name;
    }
    @Bindable
    public String getAddress() {
        return address;
    }
    @Bindable
    public String getPhone() {
        return phone;
    }
    public void setName(String name) {
        this.name = name;
        notifyPropertyChanged(BR.name);
    }
    public void setAddress(String address) {
        this.address = address;
        notifyPropertyChanged(BR.address);
    }
    public void setPhone(String phone) {
        this.phone = phone;
        notifyPropertyChanged(BR.phone);
    }
}
```

数据会变化的字段用@Bindable注解修饰getXXX方法，点击build->makeProject后，会自动在BR类中添加这个会修改的字段，使用常量表示，在setXXX方法中，表示要更新的字段传入。这样指定更新某个部分，而不是全部都更新，因此避免了数据没有改变那部分也重新赋值的消耗，性能更好。可以想想为什么@Bindable要修饰在getXXX方法而不是setXXX方法？可以根据android:text="@{user.firstName}",这里写的是user.firstName,其实是调用了user.getFirstName方法进行猜想，UI数据的来源是getXXX方法，这些数据是要更新的，因此我们在数据Model中表示要更新的部分，就是用@Bindable修饰getXXX方法。可以有点难理解。

使用ObservableField

有这么一种情况，如果一个数据类只有一个或者两个字段需要更新的，而且继承BaseObservable，显然没有必要，因此Google针对单个成员变量定义的Observe--BaseObservable.

```
public ObservableField<String> name = new ObservableField<String>();

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

public ObservableField<String> phone=new ObservableField<String>();
```

后两种只需要修改数据，UI就会自动更新。不需要像第一种那样需要手动调用对应ViewDataBinding子类的invalidateAll()方法.

DataBinding + RecycleView

1.原本Holder持有View对象，因为使用了DataBinding,所有不需要持有View，只需要持有ViewDataBinding对象

```
public class BaseViewHolder<T extends ViewDataBinding> extends RecyclerView.ViewHolde
    private T dataBing;

    public BaseViewHolder(View itemView) {
        super(itemView);
    }

    public T getDataBing() {
        return dataBing;
    }

    public void setDataBing(T dataBing) {
        this.dataBing = dataBing;
    }
}
```

RecycleViewAdapter

```
public class BaseRecycleViewAdapter<T, K extends ViewDataBinding> extends RecyclerVie
    protected List<T> lists;     //数据源
    protected int resouceId;     //布局ID
    protected int variableId;  //布局内VariableId，就是使用BR类自动生成的常量int，只想layout

    public BaseRecycleViewAdapter(List<T> lists, int resouceId, int variableId) {
        this.lists = lists;
        this.resouceId = resouceId;
        this.variableId = variableId;
    }

    /**
     * 创建绑定数据的ViewHolder(实际上就相当于初始化出来界面)
     *
     * @param parent
     * @param viewType
     * @return
     */
    @Override
    public BaseViewHolder<K> onCreateViewHolder(ViewGroup parent, int viewType) {
```

```
        K itemBing = DataBindingUtil.inflate(LayoutInflater.from(parent.getContext())
        BaseViewHolder<K> holder = new BaseViewHolder<K>(itemBing.getRoot());//初始化v
        holder.setDataBing(itemBing);
        return holder;
    }

    @Override
    public void onBindViewHolder(final BaseViewHolder<K> holder, final int position)
        T data = lists.get(position);//获取数据
        holder.getDataBing().setVariable(variableId, data);//赋值
        if (listener != null) {//设置单击事件
            holder.itemView.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View view) {
                    listener.onItemClick(holder.getDataBing(), position);
                }
            });
            holder.itemView.setOnLongClickListener(new View.OnLongClickListener() {
                @Override
                public boolean onLongClick(View view) {
                    listener.onLongItemClick(holder.getDataBing(), position);
                    return false;
                }
            });
        }
        holder.getDataBing().executePendingBindings();//刷新界面
    }
    @Override
    public int getItemCount() {
        return lists == null ? 0 : lists.size();
    }
    //自定义item单击事件
    protected OnItemClickListener listener;

    public void setListener(OnItemClickListener listener) {
        this.listener = listener;
    }
    public interface OnItemClickListener {
        public void onItemClick(ViewDataBinding dataBinding, int position);
        public void onLongItemClick(ViewDataBinding dataBinding, int position);
    }
}
```

参考资料

http://www.jianshu.com/p/9fb720f405a7
http://www.jianshu.com/p/7c8b484cda91
http://www.jianshu.com/p/686bfc58bbb0
http://www.jianshu.com/p/de4d50b88437
http://www.jianshu.com/p/ad170ed79324