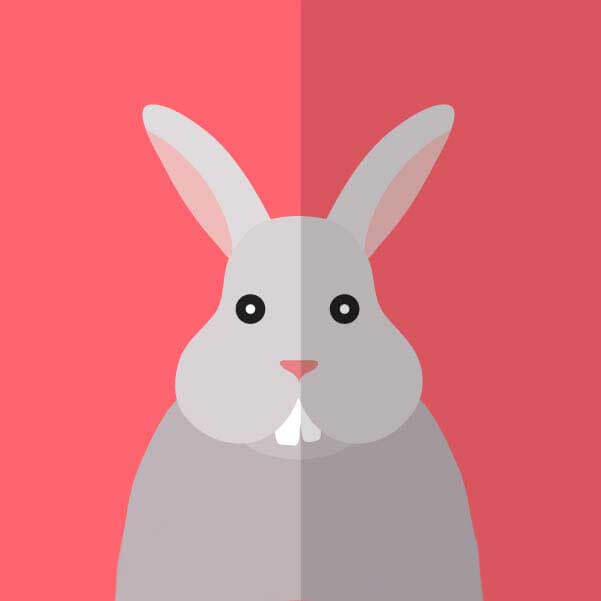
**Android：Activity与Fragment通信(99%)完美解决方案**

[](https://www.jianshu.com/u/2ce7b74b592b)

[牛晓伟](https://www.jianshu.com/u/2ce7b74b592b) 关注

2016.01.27 15:28\* 字数 2616 阅读 33384评论 55喜欢 315

**前言**

最近一直在想着能否有一种更好的方案来解决：Android中Activity与Fragment之间通信的问题，什么叫**更好**呢，就是能让Fragment的复用性高，性能还有好（不用反射），代码还要好维护，不需要为每对Activity和Fragment之间定义接口而发愁。

先简单说下Javascript这门语言吧，或许有人就会问：咱们不是聊Android的java问题吗？怎么话题转到JavaScript了。因为我的解决方案的启发是从它来的，没兴趣的朋友可以略过。最近在学习javascript这门语言，同时自己搞Android（java）开发也有5年多时间了，所以在学习js的过程中，就会惯性的把这两者进行比较。

与java语言的 **严谨** 相比 Javascript是一门**"放荡不羁"、"不拘小节"**（宽泛）的语言。  
为什么要用**"放荡不羁"**这个词呢，下面是它的一个解释：

放荡不羁 [fàng dàng bù jī][解释] 羁:约束。放纵任性，不加检点，不受约束。

因为我觉得这个词更能充分的体现js弱类型的特点。  
在给变量赋值时 可以这样写:

var a = 1;

还可以这样写:

var b = '123';

var o = new Object();

甚至还可以这样写:

var fun = new function(){};

fun1 = new function(){};

可以把任何类型的值赋给一个变量，也可以不加var关键字来声明一个变量，是不是很任性，很不拘束啊。

**"不拘小节"**主要体现了JavaScript的语法更宽泛、更简单的特点: 比如：

js代码:

//函数声明不需要定义返回值，参数前面不需要有类型出现，

//函数体里面就可以有返回值

function max(a,b){ return a > b? a:b; }

/\* \*可以传递任意多个参数，在java里面根本不可以 \*/

function print(){

var len = arguments.length;

for(var i = 0; i < len; i++){

console.log(arguments[i]);

}

}

相应java代码:

int max(int a, int b){

return a> b? a:b;

}

/\* \*传递任意多个Object类型的参数 \*/

void print(Object... args){

for (int i = 0; i < args.length; i++){

System.out.println(args[i]);

}

}

上面的代码说明了JavaScript在声明函数时，不会有像java那么严格的规定，语法不拘小节，语法更简单（这里没有说java不好的意思）。

**启发点**

JavaScript中有一个重要的点（万事万物皆对象），函数也不列外，并且**函数可以作为另外一个函数的参数**，如:

js代码：

//遍历一个数组如果是它是数组，就把它乘以10再输出

var array = [1,2, '你好' , '不' ,31,15];

//数组的each方法接收一个函数

testArray.each( function( value ){

typeof value == 'number' ? alert( value \*10 ):null;

}) ;

当我看到上面JavaScript中函数的用法时我眼前一亮，为啥我不可以借鉴之来解决android中activity与fragment通信的问题呢？

**Fragment的使命**

先让我们聊聊Fragment为什么出现，这对于我们解决Activity与Fragment的通信有帮助。一个新事物的产生总是为了解决旧事物存在的问题，Fragment是android3.0的产物，在android3.0之前解决手机、平板电脑的适配问题是很头疼的，对ActivityGroup有印象的朋友，应该能深深的体会到ActivityGroup包裹的多个Activity之间切换等一系列的性能问题。由此Fragment诞生了。个人总结的Fragment的使命：

* 解决手机、平板电脑等各种设备的适配问题
* 解决多个Activity之间切换性能问题
* 模块化，因为模块化导致复用的好处

**Fragment的使用**

Fragment是可以被包裹在多个不同Activity内的，同时一个Activity内可以包裹多个Fragment，Activity就如一个大的容器，它可以管理多个Fragment。所有Activity与Fragment之间存在依赖关系。

**Activity与Fragment通信方案**

上文提到Activity与Fragment之间是存在依赖关系的，因此它们之间必然会涉及到通信问题，解决通信问题必然会涉及到对象之间的引用。因为Fragment的出现有一个重要的使命就是：模块化，从而提高复用性。若达到此效果，Fragment必须做到高内聚，低耦合。

现在大家动动脚趾都能想到的解决它们之间通信的方案有：handler，广播，EvnetBus，接口等（或许还有别的方案，请大家多多分享），那我们就聊下这些方案。

**handler方案:**

先上代码

public class MainActivity extends FragmentActivity{

//声明一个Handler

public Handler mHandler = new Handler(){

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

...相应的处理代码

}

}

...相应的处理代码

}

public class MainFragment extends Fragment{

//保存Activity传递的handler

private Handler mHandler;

@Override

public void onAttach(Activity activity) {

super.onAttach(activity);

//这个地方已经产生了耦合，若还有其他的activity，这个地方就得修改

if(activity instance MainActivity){

mHandler = ((MainActivity)activity).mHandler;

}

}

...相应的处理代码

}

该方案存在的缺点：

* Fragment对具体的Activity存在耦合，不利于Fragment复用
* 不利于维护，若想删除相应的Activity，Fragment也得改动
* 没法获取Activity的返回数据
* handler的使用个人感觉就很不爽（不知大家是否有同感)

**广播方案：**

具体的代码就不写了，说下该方案的缺点：

* 用广播解决此问题有点大材小用了，个人感觉广播的意图是用在一对多，接收广播者是未知的情况
* 广播性能肯定会差（不要和我说性能不是问题，对于手机来说性能是大问题）
* 传播数据有限制（必须得实现序列化接口才可以）  
  暂时就想到这些缺点，其他的缺点请大家集思广益下吧。

**EventBus方案：**

具体的EventBus的使用可以自己搜索下，个人对该方案的看法：

* EventBus是用反射机制实现的，性能上会有问题（不要和我说性能不是问题，对于手机来说性能是大问题）
* EventBus难于维护代码
* 没法获取Activity的返回数据

**接口方案**

我想这种方案是大家最易想到，使用最多的一种方案吧，具体上代码：

//MainActivity实现MainFragment开放的接口

public class MainActivity extends FragmentActivity implements FragmentListener{

@override

public void toH5Page(){ }

...其他处理代码省略

}

public class MainFragment extends Fragment{

public FragmentListener mListener;

//MainFragment开放的接口

public static interface FragmentListener{

//跳到h5页面

void toH5Page();

}

@Override

public void onAttach(Activity activity) {

super.onAttach(activity);

//对传递进来的Activity进行接口转换

if(activity instance FragmentListener){

mListener = ((FragmentListener)activity);

}

}

...其他处理代码省略

}

这种方案应该是既能达到复用，又能达到很好的可维护性，并且性能也是杠杠的。但是唯一的一个遗憾是假如项目很大了，Activity与Fragment的数量也会增加，这时候为每对Activity与Fragment交互定义交互接口就是一个很头疼的问题（**包括为接口的命名，新定义的接口相应的Activity还得实现，相应的Fragment还得进行强制转换**）。 想看更好的解决方案请看下面章节。

**大招来也**

设计模式里经常提到的一个概念就是封装变化，同时受javascript中的函数的参数可以是函数对象的启发下，我有了下面的想法，先上代码：[代码地址](https://link.jianshu.com/?t=https://github.com/niuxiaowei/ActivityCommWithFragment/)

/\*\* \* + Created by niuxiaowei on 2016/1/20.

\* 各种方法集合的类，可以把一个方法类以key-value的形式放入本类，

\* 可以通过key值来调用相应的方法 \*/

public class Functions {

//带参数方法的集合，key值为方法的名字

private HashMap<String,FunctionWithParam> mFunctionWithParam ;

//无参数无返回值的方法集合，同理key值为方法名字

private HashMap<String,FunctionNoParamAndResult> mFunctionNoParamAndResult ;

/\*\* \* 基础方法类 \*/

public static abstract class Function{

//方法的名字，用来做调用，也可以理解为方法的指针

public String mFunctionName;

public Function(String functionName){

this.mFunctionName = functionName;

}

}

/\*\* \* 带有参数没有返回值的方法

\* @param <Param> 参数 \*/

public static abstract class FunctionWithParam<Param> extends Function{

public FunctionWithParam(String functionName) {

super(functionName);

}

public abstract void function(Param param);

}

/\*\* \* 没有参数和返回值的方法 \*/

public static abstract class FunctionNoParamAndResult extends Function{

public FunctionNoParamAndResult(String functionName) {

super(functionName);

}

public abstract void function();

}

/\*\* \* 添加带参数的函数

\* @param function {@link com.niu.myapp.myapp.view.util.Functions.FunctionWithParam}

\* @return \*/

public Functions addFunction(FunctionWithParam function){

if(function == null){

return this;

}

if(mFunctionWithParam == null){

mFunctionWithParam = new HashMap<>(1);

}

mFunctionWithParam.put(function.mFunctionName,function);

return this;

}

/\*\* \* 添加带返回值的函数

\* @param function {@link com.niu.myapp.myapp.view.util.Functions.FunctionWithResult}

\* @return \*/

public Functions addFunction(FunctionNoParamAndResult function){

if(function == null){ return this; }

if(mFunctionNoParamAndResult == null){

mFunctionNoParamAndResult = new HashMap<>(1);

}

mFunctionNoParamAndResult.put(function.mFunctionName,function);

return this;

}

/\*\* \* 根据函数名，回调无参无返回值的函数

\* @param funcName \*/

public void invokeFunc(String funcName) throws FunctionException {

FunctionNoParamAndResult f = null;

if(mFunctionNoParamAndResult != null){

f = mFunctionNoParamAndResult.get(funcName);

if(f != null){ f.function(); }

}

if(f == null){ throw new FunctionException("没有此函数"); }

}

/\*\* \* 调用具有参数的函数

\* @param funcName

\* @param param

\* @param <Param> \*/

public <Param> void invokeFunc(String funcName,Param param)throws FunctionException{

FunctionWithParam f = null;

if(mFunctionWithParam != null){

f = mFunctionWithParam.get(funcName);

if(f != null){ f.function(param); }

}

}

}

**设计思路：**

**1. 用一个类来模拟Javascript中的一个Function**

**Function**就是此类，它是一个基类，每个**Functioon**实例都有一个mFuncName 既然是方法（或者函数）它就有有参数和无参数之分  
**FunctionWithParam<Param>**是Function的子类，代表有参数的方法类，方法参数通过泛型解决  
**FunctionNoParamAndResult**是Function的子类，代表无参无返回值的方法类

**2. 一个可以存放多个方法（或者函数）的类**

**Functions**类就是此类,下面简单介绍下Functions有4个主要方法：

* addFunction(FunctionNoParamAndResult function) 添加一个无参无返回值的方法类
* addFunction(FunctionWithParam function) 添加一个有参无返回值的方法类
* invokeFunc(String funcName) 根据funcName调用一个方法
* invokeFunc(String funcName,Param param) 根据funcName调用有参无返回值的方法类

**使用举例：**[**代码地址**](https://link.jianshu.com/?t=https://github.com/niuxiaowei/ActivityCommWithFragment/)

每个app都有的基础activity(BaseActivity)

public abstract class BaseActivity extends FragmentActivity {

/\*\*

\* 为fragment设置functions，具体实现子类来做

\* @param fragmentId \*/

public void setFunctionsForFragment(

int fragmentId){

}

}

其中的一个activity：

public class MainActivity extends BaseActivity {

@Override public void setFunctionsForFragment(int fragmentId) {

super.setFunctionsForFragment(fragmentId);

switch (fragmentId) {

case R.id.fragment\_main:

FragmentManager fm = getSupportFragmentManager();

BaseFragment fragment = (BaseFragment) fm.findFragmentById(fragmentId);

//开始添加functions

fragment.setFunctions(new Functions()

.addFunction(new Functions.FunctionNoParamAndResult(MainFragment.FUNCTION\_NO\_PARAM\_NO\_RESULT) {

@Override

public void function() {

Toast.makeText(MainActivity.this, "成功调用无参无返回值方法", Toast.LENGTH\_LONG).show();

}

}).

addFunction(new Functions.FunctionWithParam<Integer>(MainFragment.FUNCTION\_HAS\_PARAM\_NO\_RESULT) {

@Override

public void function(Integer o) {

Toast.makeText(MainActivity.this, "成功调用有参无返回值方法 参数值=" + o, Toast.LENGTH\_LONG).show(); } }));

}

}

}

每个app都会有的基础fragment（BaseFragment）

public abstract class BaseFragment extends Fragment {

protected BaseActivity mBaseActivity;

/\*\* \* 函数的集合 \*/

protected Functions mFunctions;

/\*\* \* activity调用此方法进行设置Functions

\* @param functions \*/

public void setFunctions(Functions functions){

this.mFunctions = functions;

}

@Override

public void onAttach(Activity activity) {

super.onAttach(activity);

//呼叫activity进行回调方法的设置

if(activity instanceof BaseActivity){

mBaseActivity = (BaseActivity)activity;

mBaseActivity.setFunctionsForFragment(getId());

}

}

}

MainActivity对应的MainFragment

public class MainFragment extends BaseFragment {

/\*\* \* 没有参数没有返回值的函数 \*/

public static final String FUNCTION\_NO\_PARAM\_NO\_RESULT = "FUNCTION\_NO\_PARAM\_NO\_RESULT";

/\*\* \* 有参数没有返回值的函数 \*/

public static final String FUNCTION\_HAS\_PARAM\_NO\_RESULT = "FUNCTION\_HAS\_PARAM\_NO\_RESULT";

@Override

public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

mBut1 = (Button) getView().findViewById(R.id.click1);

mBut3 = (Button) getView().findViewById(R.id.click3);

mBut1.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

try {

//调用无参无返回值的方法

mFunctions.invokeFunc(

FUNCTION\_NO\_PARAM\_NO\_RESULT);

} catch (FunctionException e) {

e.printStackTrace();

}

}

});

mBut3.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

try {

//调用有参无返回值的方法

mFunctions.invokeFunc(

FUNCTION\_HAS\_PARAM\_NO\_RESULT, 100);

} catch (FunctionException e) {

e.printStackTrace(); }

}

});

}

看到这您是不是觉得已经结束了，当然是没有了，因为还有2个问题没解决。方法返回值和方法接收多个参数的问题。

**方法返回值的问题**

上代码：[代码地址](https://link.jianshu.com/?t=https://github.com/niuxiaowei/ActivityCommWithFragment/)

/\*\* \* 有返回值，没有参数的方法

\* @param <Result> \*/

public static abstract class FunctionWithResult<Result> extends Function{

public FunctionWithResult(String functionName) {

super(functionName);

}

public abstract Result function();

}

/\*\* \* 带有参数和返回值的 方法

\* @param <Result>

\* @param <Param> \*/

public static abstract class FunctionWithParamAndResult<Result,Param> extends Function{

public FunctionWithParamAndResult(String functionName) {

super(functionName);

}

public abstract Result function(Param data);

}

**FunctionWithResult<Result>**无参数有返回值的方法类  
**FunctionWithParamAndResult<Result,Param>** 有参数也有返回值的方法类  
在Functions类中定义添加和调用这2种方法类的 相应方法。

**其次是方法含有多个参数的问题**

在解决此问题时我想了很多办法（比如怎样引入多个泛型，但最终以失败告终，希望有看了这篇文章的朋友可以多提下宝贵意见）。然后我就想到了用Bundle来解决多参数的问题，把多个参数放到Bundle中，但是在往Bundle中塞入数据时得有一个对应的key值，生成key值以及记住key值（记住key值是为了从Bundle中取数据）是一个繁琐的事。同时Bundle不能传递非序列化对象。所以就封装了一个FunctionParams类解决以上问题，请看类的实现: [代码地址](https://link.jianshu.com/?t=https://github.com/niuxiaowei/ActivityCommWithFragment/)

/\*\* \* 函数的参数，当函数的参数涉及到多个值时，可以用此类，

\* 此类使用规则：存参数与取参数的顺序必须一致，

\* 比如存参数顺序是new

\*FunctionParamsBuilder().putString("a").putString("b").putInt(100);

\*取的顺序也是: functionParams.getString(),

\*functionParams.getString(), functionParams.getInt(); \*/

public static class FunctionParams {

private Bundle mParams = new Bundle(1);

private int mIndex = -1;

private Map mObjectParams = new HashMap(1);

FunctionParams(Bundle mParams,Map mObjectParams){

this.mParams = mParams;

this.mObjectParams = mObjectParams;

}

public <Param> Param getObject(Class<Param> p){

if(mObjectParams == null){ return null; }

return p.cast(mObjectParams.get((mIndex++) + "")); }

/\*\* \* 获取int值

\* @return \*/

public int getInt(){

if(mParams != null){

return mParams.getInt((mIndex++) + ""); } return 0;

}

/\*\* \* 获取int值

\* @param defalut

\* @return \*/

public int getInt(int defalut){

if(mParams != null){

return mParams.getInt((mIndex++) + "");

}

return defalut;

}

/\*\* \* 获取字符串

\* @param defalut \* @return \*/

public String getString(String defalut){

if(mParams != null){

return mParams.getString((mIndex++) + "");

}

return defalut;

}

/\*\* \* 获取字符串 \* @return \*/

public String getString(){

if(mParams != null){

return mParams.getString((mIndex++) + "");

} return null;

}

/\*\* \* 获取Boolean值

\* @return 默认返回false \*/

public boolean getBoolean(){

if(mParams != null){

return mParams.getBoolean((mIndex++) + "");

} return false;

}

/\*\* \* 该类用来创建函数参数 \*/

public static class FunctionParamsBuilder{

private Bundle mParams ;

private int mIndex = -1;

private Map mObjectParams = new HashMap(1);

public FunctionParamsBuilder(){ }

public FunctionParamsBuilder putInt(int value){

if(mParams == null){

mParams = new Bundle(2);

}

mParams.putInt((mIndex++) + "", value);

return this;

}

public FunctionParamsBuilder putString(String value){

if(mParams == null){

mParams = new Bundle(2);

}

mParams.putString((mIndex++) + "", value);

return this;

}

public FunctionParamsBuilder putBoolean(boolean value){

if(mParams == null){ mParams = new Bundle(2); }

mParams.putBoolean((mIndex++) + "", value);

return this;

}

public FunctionParamsBuilder putObject(Object value){

if(mObjectParams == null){

mObjectParams = new HashMap(1);

}

mObjectParams.put((mIndex++) + "", value);

return this;

}

public FunctionParams create(){

FunctionParams instance = new FunctionParams(mParams,mObjectParams); return instance;

}

}

}

**FunctionParams**封装了取参数的功能，比如：

public <Param> Param getObject(Class<Param> p){

if(mObjectParams == null){ return null; }

return p.cast(mObjectParams.get((mIndex++) + ""));

}

取对象参数的功能，不需要传人key值，只需要传人需要即将取出来的类的Class实例即可

**FunctionParamsBuilder**类，看它的名字就知道是用了设计模式里的Builder（构建）模式。该类是用来存放参数的，当所有的参数都存放完毕后调用create()方法创建一个FunctionParams对象事物都是有两面性的，有缺点就有优点，只不过是在某些场合下优点大于缺点，还是反之。  
**FunctionParams**解决了以上提到的Bundle传递多参数种种不便的问题，但同时**FunctionParams**也有一个缺点就是存参数的顺序与取参数的顺序一定要一致，比如：

//存的顺序 new

FunctionParamsBuilder().putString("1").putInt(2)

.putBoolean(true).create();

//取的顺序

functionParams.getString();

functionParams.getInt();

functionParams.getBoolean();

但是这种缺点函数的定义来看也不是缺点。

Activity与Fragment之间的通信是通过Functions的，即把变化的部分封装在Functions是类中，Functions起一个桥梁作用。

此方案优点：

* Fragment与Activity的耦合性几乎没有
* 性能也好（没用反射）
* 可以从Activity获取返回数据
* 扩展性好（新增加的成对的Activity与Fragment之间的通信只需做以下几步：  
  1.新增加Activity只需要覆盖BaseActivity中的 setFunctionsForFragment(int fragmentId) 方法，把相应的回调函数加入。  
  2.相应的Fragment定义函数key值即可）

**总结**

简单总结为以下几点：

* Fragment的使命
* Activity与Fragment之间通信的解决方案（handler，广播，EventBus，接口）的优缺点。
* 我自己关于Activity与Fragment之间通信的解决方案（Functions），其实解决的主要是Fragment调用Activity的方案。

希望大家能多提宝贵意见，多交流。[代码地址](https://link.jianshu.com/?t=https://github.com/niuxiaowei/ActivityCommWithFragment/)

本人微信：704451290