满分家园Android平台

开发手册

目录

[一、 工程结构 4](#_Toc397006245)

[二、 重要约定 7](#_Toc397006246)

[三、 基本开发框架 8](#_Toc397006247)

[1. DAO层 9](#_Toc397006248)

[1) 概述 9](#_Toc397006249)

[2) 网络异步Dao接口-BaseNetDao 9](#_Toc397006250)

[3) 本地同步Dao接口-BaseDbDao 10](#_Toc397006251)

[2. BEAN层 11](#_Toc397006252)

[1) 数据bean 11](#_Toc397006253)

[2) 显示bean 12](#_Toc397006254)

[3. SERVICE层 13](#_Toc397006255)

[1) 概述 13](#_Toc397006256)

[2) 生命周期 13](#_Toc397006257)

[3) 基本功能 13](#_Toc397006258)

[4) 关于本地缓存(待定) 14](#_Toc397006259)

[5) 任务异步处理框架 14](#_Toc397006260)

[4. VIEW层 15](#_Toc397006261)

[1) 概述 15](#_Toc397006262)

[2) 异步任务处理 15](#_Toc397006263)

[3) 通用信息提示 16](#_Toc397006264)

[5. VIEW-LIST基础组件 16](#_Toc397006265)

[1) 概述 16](#_Toc397006266)

[2) 特性 17](#_Toc397006267)

[3) SEARCH框架 17](#_Toc397006268)

[4) BaseFragmentActivity 18](#_Toc397006269)

[5) BaseFragment 18](#_Toc397006270)

[6) BaseListFragment 18](#_Toc397006271)

[7) BaseListCheckFragment 19](#_Toc397006272)

[8) BaseListIdxAbleFragment 19](#_Toc397006273)

[9) BaseListCheckIdxAbleFragment 20](#_Toc397006274)

[10) BaseListDetailIdxAbleFragment 21](#_Toc397006275)

[11) BaseComnActivity 22](#_Toc397006276)

[12) BaseListActivity 22](#_Toc397006277)

[6. APP容器 23](#_Toc397006278)

[1) MyApplication基类 23](#_Toc397006279)

[2) App引导 23](#_Toc397006280)

[3) 通用登录服务 23](#_Toc397006281)

[四、 基础工具/运行框架 24](#_Toc397006282)

[1. 统一网络通讯 24](#_Toc397006283)

[1) 概述 24](#_Toc397006284)

[2) 同步调用接口 24](#_Toc397006285)

[3) 异步调用接口 24](#_Toc397006286)

[4) 序列化工具 25](#_Toc397006287)

[2. 统一通讯数据包 25](#_Toc397006288)

[1) 数据包格式 26](#_Toc397006289)

[2) 数据块格式 26](#_Toc397006290)

[3) 数据包解析器 27](#_Toc397006291)

[3. 统一文件操作 27](#_Toc397006292)

[4. N+1配置框架 28](#_Toc397006293)

[1) 概述 28](#_Toc397006294)

[2) Uconfig定义 28](#_Toc397006295)

[3) 调用接口 29](#_Toc397006296)

[5. 序列生成模块 29](#_Toc397006297)

[6. 统一编码模块 30](#_Toc397006298)

[7. 统一登录模块 31](#_Toc397006299)

[8. 统一程序升级框架 31](#_Toc397006300)

[9. 客户端/服务器数据同步机制 32](#_Toc397006301)

[1) 客户端 32](#_Toc397006302)

[2) 服务端 33](#_Toc397006303)

[10. 本地数据库版本升级框架 33](#_Toc397006304)

[11. 头像/图片显示模块 35](#_Toc397006305)

[12. 统一错误收集模块 36](#_Toc397006306)

# 工程结构

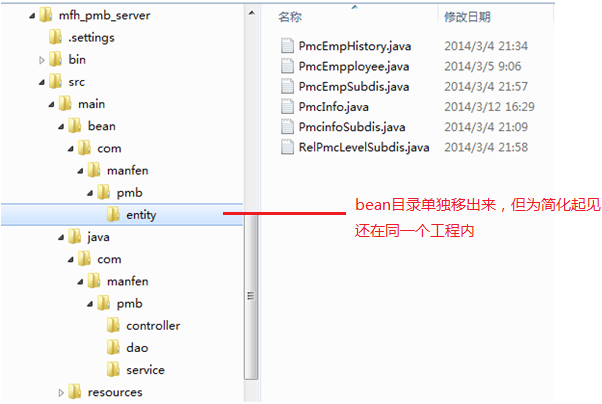


1. 因为都是java语言，最小面一层是服务器端和客户端都可以共用的代码，但这部分代码注意运行在不同的虚拟机上，必须保证兼容性。
2. 在业务层，针对同一业务，服务器端和客户端有相同的bean也可以重用，这部分代码可以放在服务端，然后客户端通过代码link的方式加入到本工程的src集中，如图：

这些bean都继承自MfhEntity基类，为了避免相互干扰，客户端也可以构造一个同名同包和具有相同基本属性的基类，确保这些bean代码可以正确编译。

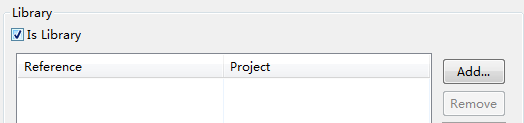


服务器端工程源码实际结构示意如下：

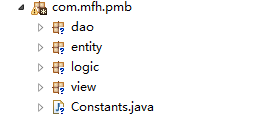


1. 除了最上层工程，下面基础工程都采用Is Library选项:

在eclipse-》windos->Preference->android中：



1. 工程内目录结构，分成四个目录，每个目录下有自己的Constants定义：



其中logic目录代表前端逻辑层（对应与后台java的service目录，偏界面逻辑和简单业务逻辑）

1. 已有工程说明：

mfh-owner-client:面向业主和注册粉丝的app

mfh-pmb-client: 物业公司移动端，面向物业公司人员

mfh-cc-client:面向满分客服人员的，春晨公司内部app

mfh-business-client:面向周边商家的导购平板app(还未开发)

~~mfh-ha-client:面向房产经纪人app~~

~~mfh-tax-client:税务专管员app~~

# 重要约定

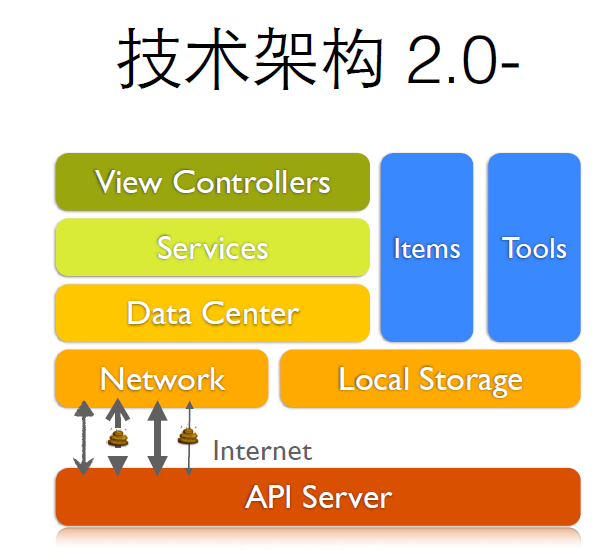
1. 源码全部采用utf-8编码。
2. 使用android原生控件，符合android推荐的操作风格
3. 文字等常量统一定义在xml文件中
4. 命名规范：因android资源目录下不能建子目录，所以按"模块名\_主词\_谓词",不要用大小写分割了，全部小写，用下划线分割; 如全局公用模块中的用户列表布局文件名：

comn\_user\_list.xml

1. 像素点统一采用dip，就是把屏幕的高分成标准的480分，宽分成320分，一分为一个dip。
2. 尽量多开发fragment，这样可以容易被重用，少开发activity和fragmentActivity。通过fragmentActivity可以方便地集成fragment。fragment之间通信通过activity进行，相互之间避免直接通信；fragment可以通过sendBroadcast给activity发消息，activity通过通过registerReceiver()注册一个内部类作为接收者来接收。
3. 同理activity之间参数传入和回传通过intent和startActivityForRuesult实现。activity之间方法调用也要尽量避免直接方法调用，可以通过broadCast和reciever机制实现。
4. 牢记android主线程即UI线程只有一个，是单线程。所有耗时的操作都应转入后台异步处理。我们提供了MyMultiAsyncTask()类负责在一个activity或fragment中进行多任务异步处理。

# 基本开发框架

分层开发，主要是分离出logic和dao层。



-----引自三号包厢技术沙龙

## DAO层

### 概述

* 此层的作用在于数据读取和ORMapping，为统一Dao模式，分为异步dao和同步dao两种模式，其中通过网络读取数据采用异步dao，通过本地数据库读取采用同步dao，分别继承自BaseNetDao、BaseDbDao两个类。***在我们的实际应用场景中，需要混合两种编程***，如历史消息这块应存在本地数据库以便于快速查询，业务部分可以通过网络实时读取。关于网络通讯和协议部分详细参见“[统一网络通讯](#_统一网络通讯)”章节和“[统一通讯数据包](#_统一通讯数据包)”章节。
* 序列化/反序列化采用阿里开源的FastJson，开源，体积小巧，速度很快。详细参见“[序列化工具](#_序列化工具)”章节。
* 从本地数据库读取数据时，为了方便编程，引入类似于服务器端的ORMapping技术，此处使用了afinal开源框架，并对其进行了适当修改，使其支持继承类、pageInfo查询分页等高级特性。

详见com.mfh.comna.dao包

### 网络异步Dao接口-BaseNetDao

本质上把从后台服务器读取数据也模拟成一次dao操作。参见com.mfh.comna.dao.IAsyncDao<T, PK>,其中对dao涉及的bean类型和主键类型做了泛型。

以查询结果集接口为例：

/\*\*

\* 执行结果集查询

\* **@param** params 查询参数

\* **@param** callBack 查询结果处理函数

\* **@param** factUrl 查询接口url，可以为空，代表使用默认

\* **@author**zhangyz created on 2014-3-10

\*/

**publicvoid** query(AjaxParams params, QueryRsProcessor<T> callBack, String... factUrl);

* 因为是异步接口，故调用者需要提供回调函数callBack以实现对查询到的结果集进行处理。callBack需要继承QueryRsProcessor接口，支持传递分页参数。
* 调用者从BaseNetDao继承实现自己的dao子类，至少需要实现initUrlInfo方法
* 若在bean中可能存在日期类型的属性，反序列化时需要知道日期格式，故可以调用setStrDataFormat()方法进行设置，默认为JSON.DEFFAULT\_DATE\_FORMAT （"yyyy-MM-dd HH:mm:ss"）,此时无须设置。

### 本地同步Dao接口-BaseDbDao

使用同步dao时代表使用本地数据库，android本身内置了sqllite数据库引擎。本地dao模块基于sqllite进行了封装。使用本地数据库目的是为了支持离线操作和加速访问，有下面几点设计策略：

* 一些表可以直接在本地放一份，支持离线查询，包括:
  + 全局公用的码表，基础数据表等，本身数据也小
  + 消息等历史数据
  + 业务表，只本地存储租户自己的数据，因此可能还是小的。（有个阀值，即使自己数据很大也不离线存储）
* 一些本地写操作先写在本地库，然后再异步上传到服务器上。
* 两端数据同步策略(备用)：
* 每个表增加一个时间戳字段updateDate~~和是否修改字段updateFlag~~,新增和修改都要记录更改这个时间戳的值；
* 每个表的主键字段值分为两个部分：设备号+序列号，通过设备号可以知晓是不是自己产生的。设备号预先分配的；
* 先上传再下载，不考虑需要数据合并的场景；
* 每台设备通过一个游标表记录每张表的时间戳同步位置，包括下载游标位置和上传游标位置。

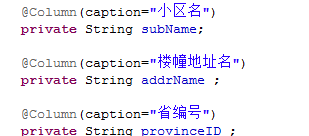
使用方法：

* 系统启动时需要先调用BaseDbDao.initDao(context, dbName)进行一次初始化；
* 编写自己的子类，实现initTableChName()方法告知框架表名和中文名，实现initPojoClass()方法告知框架bean的类型。
* 若表主键是自增的，且可以跨手机终端唯一的，则务必从BaseSeqAbleDao子类继承，该类封装实现了自增主键的实现，且可以覆盖getSequenceArea()方法，告知框架本表的主键自增序列级别，参见enum对象SeqArea，分成从万到千亿几个级别。
* 为了支持跨手机唯一，需要为每个手机本地分配每个序列号级别的范围db.idp0~db.idpn,如同样的对于“万”级别，A手机生成的序列范围是10000~20000，B手机是20001~30000。配置项的管理参见[统一配置框架](#_N+1配置框架)。

## BEAN层

### 数据bean

* bean对象都位于entity包下，与java后台的bean完全一致，因此应尽量和java后台重用bean对象。
* 所有客户端bean对象应从MfhEntity基类继承，并根据其主键类型实现相应的IObject接口，如IStringId、ILongId接口等。
* bean对象本身支持注解，可以对某个属性定义不同名的column名（数据库字段名）和中文描述，其中中文描述在显示一个bean对象的详细信息时可以用到。 如，采用**com.mfh.comn.annotations.Column**注解.



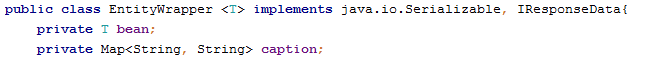
有了caption，在进行详细信息显示时可以动态生成，而不用把界面写死，如图左边一列的中文名即可以来自于此处的caption：



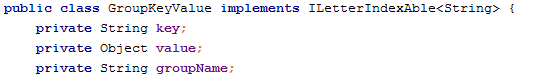
### 显示bean

数据bean和界面上显示bean进行区分，不要混为一谈。数据bean严格与数据库字段对应，引用其他表的外键也都采用id，而不是name；各属性的java类型与字段的数据库类型严格对应。界面上显示用bean的是组装和翻译后的。显示相关的bean具体参见：

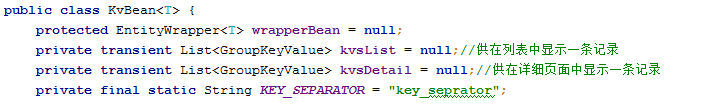
EntityWrapper<T>：



GroupKeyValue：



和将两者联合的KvBean<T>



## SERVICE层

### 概述

Service类位于logic包下，基类为com.mfh.comna.logic.BaseService和comnService，所有子类都应从这两个类之一继承。Service类介于dao层和view层中间，起到控制和逻辑处理的作用。相对服务器端来说，android端的service相对轻量，无事务和多线程的困扰。

一个service也可以包含多个dao，对多个dao的数据结果进行再组装供view显示。通常一个service对应一个dao类，故在service类定义了三个泛型，即bean类型、主键类型和dao类型：BaseService<T, PK, D extends IDao<T, PK>>

### 生命周期

每个service类应设计成单例模式，即在系统运行期间同一个类只能一个对象存在。在java服务器端可以通过spring之类的容器对service进行管理，在android可以简单地通过下面服务工厂创建service：

com.mfh.comna.logic.ServiceFactory.getService(Class<T> serviceClass, Context… context),

其中第一个参数便是service类名；第二个参数可选为android上下文，便于统一进行错误提示、进度显示等涉及到界面的操作。

相应的，dao类一般在service类中进行创建，故dao类也是单例的。

### 基本功能

BaseService基类提供了下面几个属性供子类直接使用，简化编程：

* 统一配置项对象uconfig，提供从多样化、N+1型的配置文件中读取某配置项的功能。详细参见“[统一配置框架](#_统一配置框架)”章节。
* 日志记录对象logger，基于slf4j日志门户框架，在android上引入了slf4j-android.jar包。业务代码所有捕获异常的地方应采用logger记录异常,如 logger.error(….), 避免使用System.out.print()。
* 数据访问对象dao，如果service子类本身需要多个dao对象，那么这个属性指的是主dao。Dao对象有service负责创建，但子类需要实现Class<D> getDaoClass()方法提供dao对象的类名。关于dao的操作参见“[Dao层](#_Dao层)”章节。
* 数据编码访问对象codeService，提供对字典类（支持多级）数据的统一访问接口，具体参见“[统一编码模块](#_统一编码模块)”章节。

### 关于本地缓存(待定)

读数据访问层的混合封装：

* 优先从内存缓存读取；
* 无内存缓存则从本地数据库读取；
* 无本地数据库查看有无本地缓存文件；
* 没有本地缓存最后再从网络读取，并根据需要将结果进行缓存。

缓存失效策略有根据过期时间定期清理（离线情况下不清理）、

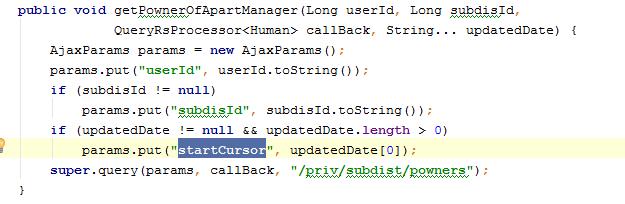
根据服务端某推送事件清理、永久缓存（需要手工清理）

### 5)客户端/服务器数据同步机制

客户端由于需要进行离线存储，往往需要将服务器端数据下载同步到本地，特别需要进行增量同步以避免数据量过大时同步过荷。两端进行增量同步的关键是需要传递一个游标，用于表征此处数据下载同步的起点。游标的类型和值多样，具体由服务器端进行解释，客户端只需记录一个字符串来存储该游标值即可。

①客户端

1. 提供了CursorUtil来支持客户端本地针对某张表同步游标的存储,通过调用getLastUpdate()方法可以获取游标值。
2. 在通过网络请求服务器时，在参数中加入startCursor参数，用以传递游标值，如：



1. 在获取到数据后，从结果集中再获取新的游标，把获取到的最大游标值再存储到本地，目前这一步还未通用化，需要每个dao模块自行书写；可以调用CursorUtil下面三个方法以便于此项操作：

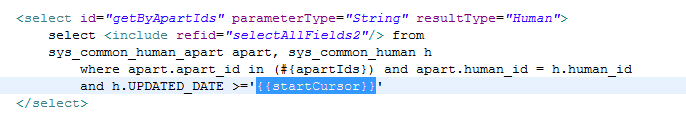
1)、处理结果集之前调用：cursorUtil.beginRecordCursor();

2)、针对结果集中的每条记录调用：cursorUtil.nextRecordCurosr(…)

3)、最后调用: cursorUtil.commitCursor();

②服务端

服务器端需要配合支持startCursor参数，接收到该参数后，需要一路传递到dao模块中。服务端框架已经做了该事，具体是通过pageInfo将游标参数保存在里面，然后通过已有的mybatis分页插件将该参数直接传递到dao层。在dao的映射文件里，应通过关键字{{startCursor}}来为该游标值占位：如：



其他无需再做额外事情。

③数据拉取模式

目前的数据同步都是由客户端发起从服务器端拉取数据，但具体来说可以分为两种：

* 1. 定时拉取，即采用定时器按预定时间间隔主动拉取数据。
  2. 事件拉取, 由一定的事件触发，例如收到服务器端更新通知、其他模块通知等等，此时可以采用broadcast机制(实现优先级不高)。
  3. 调用拉取，由调用者主动调用拉取数据方法，在拉取完成后需要回调调用者告诉数据拉取完毕;例如在翻屏时。

数据同步层级

数据分成几个同步层次，一般有：本地数据库-》后台服务器

Service层面

1. BaseService设置统一设置数据定时同步。在BaseService中initAllSyncDataLogic()开启后台ComnaAndroidService定时同步数据， 在ComnaAndroidService中定时调用BaseService的syncDataFromFrontToEnd(int fromLayerIndex)， fromLayerIndex从哪一层开始同步数据（比如数据层有三层， 如果传参为1 就从第一层向上同步）
2. 在BaseService中syncDataFromFrontToEnd(int fromLayerIndex)中的逻辑对应的是开启异步任务调用同步策略类的syncDataFromFrontToEnd(int fromLayerIndex)实现方法。
3. 整个数据同步模块改成组装模式，不是继承模式，以支持更好的灵活性和重用性;考虑到最多三层，可以定义如下抽象基类:

abstract class DataSyncStrategy (策略类)

abstract class TwoLayerStrategy extends DataSyncStrategy

abstract class ThreeLayerStrategy extends DataSyncStrategy

对于声明有三层数据层的子类来说应实现下面方法；

syncDataFromOne\_two()

syncDataFromTwo\_three()

最后在BaseService中要求子类提供不同的DataSyncStrategy策实现即可， 即在BaseService子类的getDateSyncStrategy中返回当前Service的数据同步策略实现类。

View层面

1. 在BaseFragment中 getService 获取当前页面的数据接口Service， 然后在初始化的时候调用BaseService initAllSyncDataLogic()开启同步数据。
2. 在readPageList从最近一层读取数据。
3. 当第一次进界面而且没有数据的时候会出发同步数据。
4. 在界面数据翻页滑到最后的时候也会开启同步数据。

使用

1. BaseService子类中返回数据实现策略实例

@Override

**public** DataSyncStrategy getDataSyncStrategy() {

DataSyncStrategy dataSyncStrategy = **new** TwoLayerStrategy(){

@Override

**publicvoid** syncDataFromOne\_Two() {

**if** (ls != **null**&& ls.getLoginName() != **null**)

setUtil = **new** MemberSetUtil(ls.getLoginName());

**else**

**return**;

String cursorValue = setUtil.getLastUpdate();

List<EntityWrapper<Human>> humans =

**new** ArrayList<EntityWrapper<Human>>();

List<EntityWrapper<Human>> temp = **null**;

temp = netDao.getSellSupportsByPowner(ls.getMySubdisIds(), "");

**if** (temp != **null**) humans.addAll(temp);

saveMember(humans);

}

};

**return** dataSyncStrategy;

}

1. BaseListFragment中返回界面的Service

@Override

**public** BaseService getService() {

MemberService service =

ServiceFactory.getService(MemberService.**class**, getActivity());

**return** service;

}

### 6) 任务异步处理框架

安卓系统本身提供了很好的任务异步处理框架AsyncTask，但也有使用不便之处，例如对异常、连续多任务、多入口任务等场景支持不好。我们提供了如下封装：

* 单任务异步处理：MyAsyncTask，支持从AsyncTask继承，支持异常、日志、等待框显示等额外功能
* 多入口任务异步处理，MyMultiAsyncTask，支持在任务执行时传递一个任务号，以区别是哪个异步任务；
* 连续多任务处理，MultiContinuousTask，执行连续的多个网络任务。每个任务异步执行完毕后，再顺序启动下一个任务，直至全部任务执行结束；
* 多任务网络查询，MultiTaskWithPage，从MultiContinuousTask继承，实现连续的多个查询任务，每个查询任务进一步再支持多次分页查询。

## VIEW层

### 概述

View层最复杂，涉及到android各类界面控件、布局，以及activity和fragment。View进一步关联各类界面资源，如布局文件、图标、菜单等。

在activity和fragment里创建service对象。一般界面控件显示数据时都会通过其绑定的adapter获取数据，因此从service里获取到的数据会传递给adapter进而触发界面重绘，从而形成完整的闭环。

在实践中，为便于重用，应该尽量把fragment作为view层的入口，避免直接使用activity；Fragment-》android控件-》service-》dao形成一个高内聚易重用的业务模块。然后通过FragmentActive可以组装多个Fragment形成更复杂的业务操作界面。

### 异步任务处理

需要牢记android主线程即UI线程只有一个，我们代码的主流程是单线程执行的，这样很安全。所有耗时的操作如复杂计算、本地I/O操作、网络I/O操作都应转入后台异步处理，避免直接在fragment或activity代码里面执行。

为便于执行异步任务，特别在一个view里有多种异步任务需要启动执行，我们提供了com.mfh.comna.logic.MyMultiAsyncTask里，对android的异步任务类（AsyncTask）再做包装，支持多项异步任务、启动前显示等待框、统一异常处理封装并提示出错消息等基础操作。因为是多任务，故每个任务的参数和返回值类型可能不同，不能统一定义，所以都采用Object作为输入参数和输出结果泛型。

为方便使用，在我们提供的fragment或activity基类里，直接提供了下面三类方法进行完整的异步任务处理：

* 提供doAsyncTask(….)等系列方法供直接启动异步任务；
* 子类需要继承实现doInBackground()方法进行具体的后台处理逻辑
* 子类需要继承实现onPostExecute()方法进行后台任务完成后的界面重绘操作。

### 通用信息提示

系统提供了易用的信息提示工具com.mfh.comna.utils.DialogUtil，可以直接显示提示消息框、对话确认框等，以提供统一的提示框风格。为便于使用，在我们提供的fragment或activity基类里，直接提供了下面几种方法供调用：

* showHint，提示消息，稍后自动关闭
* showMessage，提示消息，点击确定后关闭
* showYesNoDialog，显示典型的多按钮的询问对话框

## VIEW-LIST基础组件

### 概述

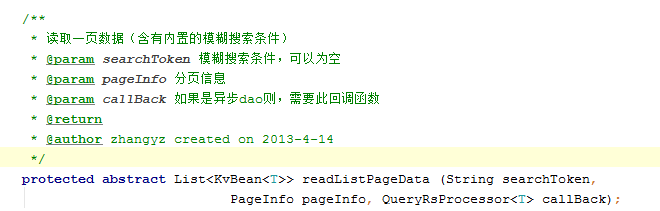
针对几种常见的界面组件，系统提供了基类供继承。在这些基类中都实现了上述异步任务处理、消息提示等公用方法，不再一一赘述。几个基础组件的继承关系如下：



### 特性

* 重型组件，已经与service、dao集成，可以直接从网络下载数据到本地，然后从本地读取数据显示到列表上。这些操作都是异步和稳定的。要实现一个后台对象的列表显示，子类只需定义后台url数据接口、定义bean类和实现Item绘制即可。
* 支持翻页和滚屏，一次只读取一页数据，与dao已经集成好；
* 支持上翻或下翻两个方向查看历史数据；
* 集成了搜索功能；
* 良好的扩展性，数据和布局层面都允许子类进行定制。

如图，子类需要实现的关键接口，从中可以看出一些特性



### SEARCH框架

android本身内置了一套比较完善的搜索框架，我们将其与我们的view基类做了良好集成, 实现在activity或fragment中无缝支持搜索。具体参见com.mfh.comna.logic.search.MySearchAdapter。

使用方法：

* acitivity或fragment支持搜索，首先需要使用android自带的searchView，可以在actionbar上或页面上，这个自己去放置就行了。
* 然后在onCreate（）方法创建searchAdapter时，有doSearch和genSuggestionProvider方法需要实现，后者可以返回null；

adapter = **new** MySearchAdapter(**this**) {

@Override

**publicvoid** doSearch(String queryToken) {

//找到本activity包含的fragment列表设置搜索串，并通知其重新读取数//据并刷新,要求dbdao层支持传入的搜索串。

fmt.setSearchToken(queryToken);

fmt.doLoadAndRefreshStart();

//或本身就是BaseListActivity：

setSearchToken(queryToken);

doLoadAndRefresh();

}

@Override

**public** MySuggestionProvider genSuggestionProvider() {

// new MySuggestionProvider (this, CursorAdapter.NO\_SELECTION){}

**returnnull**;

}

};

* 然后需要明确调用initSearchView方法。若：
* 来自于actionbar,在onCreateOptionsMenu中：

SearchView searchView = (SearchView)menu.getItem(0).getActionView()；

adapter.initSearchView(searchView, “queryHint”);

* 来自于页面，在onCreate中：

SearchView searchView = (SearchView)this.findViewById(R.id.indiSearchView)

adapter.initSearchView(searchView, “queryHint”);

* 若改变配置可以再adapter.configSearchView()方法。

### BaseFragmentActivity

自定义的FragmentActive基类,所有自行开发的FragmentActive都应从该类继承。内部支持下面多个BaseFragment的管理和切换。

### BaseFragment

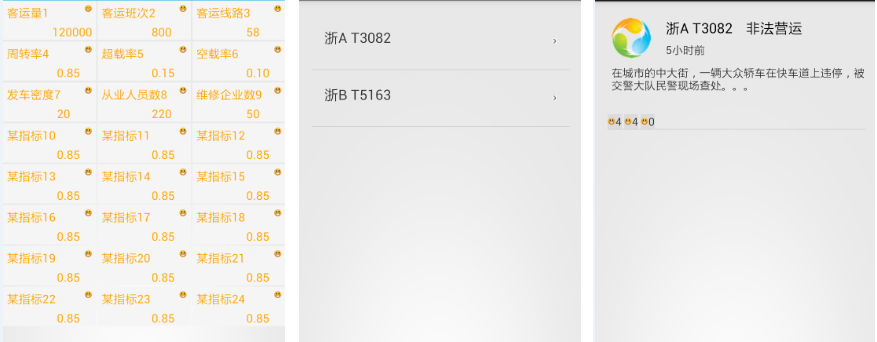
在我们系统中使用的Fragment基类，所有fragment都应从该类继承实现。

### BaseListFragment

继承自BaseFragment，实现基本列表功能的Fragment。下面几种都是本列表组件可以实现的效果。

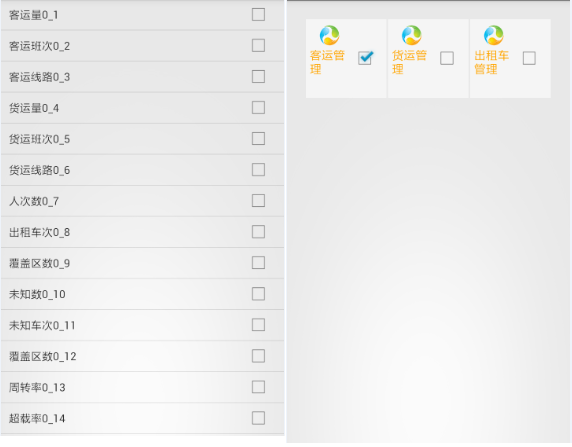
支持翻页，后台重新按页读取数据，前台进行展示。

支持往下翻页读取新数据，也支持往上翻页读取数据。特别是消息对话列表就应采用这个。参见new PageInfo(boolean fromLast, int pageSize)构造函数。



### BaseListCheckFragment

继承自BaseListFragment，进一步实现列表中每个条目支持选择框（单选或多选）的功能。如图：



### BaseListIdxAbleFragment

继承自BaseListFragment，进一步实现列表支持分组显示，并且可提供索引滑块快速定位。对应的适配器需要采用LetterPageAdapter。



### BaseListCheckIdxAbleFragment

继承自BaseListIdxAbleFragment,在其基础上再针对每个条目也添加了支持选择框的功能（包括单项或多选）。对应的适配器需要采用LetterPageCheckAdapter。



### BaseListDetailIdxAbleFragment

继承自BaseListIdxAbleFragment,前者实现多条记录的分组索引显示，本类则可以实现对单条记录详细信息的显示，按照key-value分两列显示，当单条记录属性本身也多时，同时也支持对这些属性分组显示。



或者不分组显示：



### BaseComnActivity

在我们系统中使用的普通Activity基类，所有普通的activity都需要从该类继承。

### BaseListActivity

从BaseComnActivity继承，实现了基本的列表功能。

## APP容器

### MyApplication基类

自定义应用类，我们所有的andriod应用都应从该类继承，以便进行统一的环境初始化工作以及提供全局缓存服务等。

在AndroidManifest.xml中进行如下配置：

<application android:name="com.mfh.comna.MyApplication"

….

</application>

### App引导

参见InitService和InitAcitvity,InitActivity。主要功能是更新检测和升级。

* 统一初始化, 本地数据环境准备
* 网络环境检查
* 版本检查和自动升级- com.mfh.app.update.AppUpdate
  + 版本检测、程序下载、升级提示和升级
  + 支持前台模式和后台模式
  + 下一步：增量升级。。。
* 启动主应用，最后引导进入主Acitivity:参见startMainActivity(), 一般是登录页面。

### 通用登录服务

com.mfh.app.login.LoginService

* 提供登录相关和当前用户相关的标准接口
* 本地存储登录会话标识，下次无须再次登录
* 会话超时后自动登录

# 基础工具/运行框架

## 统一网络通讯

### 概述

支持同步网络调用和异步网络调用后台服务的http接口。基于Afinal开源工程中的FinalHttp组件二次封装而来。

上述网络dao组件BaseNetDao本身便是基于异步网络调用组件基础上封装起来的;因此涉及到后台数据增、删、改、查的网络操作还是统一通过BaseNetDao操作，无须使用本底层模块。

通过com.mfh.comna.net.NetFactory工厂类获取http网络通讯对象。在NetFactory中对网络通讯的*“超时参数”、“字符集”、“重试次数”*等实际应用中需要考虑的重要参数进行了统一设置。

### 同步调用接口

AjaxParams ajaxParam = **new** AjaxParams();//传递参数

params.put("param1", "value1");

String ret = (String)NetFactory.getHttp().**getSync**(“http://l92....”, ajaxParam);

注意：若调用者本身已经使用了android的异步调用框架来进行网络通讯，则通讯本身使用这个同步调用方式即可。

### 异步调用接口

AjaxParams ajaxParam = **new** AjaxParams();//传递参数

params.put("param1", "value1");

NetFactory.getHttp().get([http://192.168...](http://192.168.../), ajaxParam, new AjaxCallBack<T>() {

public void onSuccess(T t){//成功后回调

};

public void onFailure (Throwable t, String strMsg){//失败后回调

};

);

为了提供统一的出错处理机制和异步调用过程中的信息提示（如展现“处理中…”提示框），我们进一步提供了com.mfh.comna.logic.AsyncTaskCallBack类供使用，调用者可以从该类进行继承编写网络回调函数;该类统一提供了调用进度条显示（可选）、出错后统一提示（可选）、出错后日志记录和统一收集等。

new AsyncTaskCallBack(Context… context) {

protected void doSuccess (T rawValue) {//调用成功执行，内部无须考虑异常

}

}

进一步地为了和服务器端的统一通讯数据包配合，你还可以使用进一步从AsyncTaskCallBack继承的子类**NormalNetTask**，基于NormalNetTask编写回调函数。我们强烈建议android端与服务器端采用下述的统一通讯数据包进行通讯，然后客户端网络操作回调函数都基于NormalNetTask进行。

### 序列化工具

网络调用一般都涉及到需要对传递的对象和接收到的对象进行序列化和反序列化，此处引入了阿里开源的fastJson，适合在android环境上运行，体积小，速度快。

* 序列化：com.alibaba.fastjson.Json.toJSONString(…..)

含有日期属性时可以指定日期格式，使用下面方法：

com.alibaba.fastjson.Json.toJSONStringWithDateFormat(…..)

* 反序列化：com.alibaba.fastjson.Json.ParseObject(…),或toJavaObject(….)直接生成java对象。美中不足的是反序列化接口中未提供日期格式设置，为此修改了一下代码，可以通过线程变量设置一下：

DefaultJSONParser.dateFormatPatternThread.set(“yyyy-MM-dd….”);

## 统一通讯数据包

通讯数据包统一采用JSON格式，未来可扩展也支持XML格式，默认为JSON格式。我们约定，整个系统前后台所有通讯涉及到的数据包采用统一格式。

### 数据包格式

数据包结构如下，分成三段：

* 返回码
* 返回码描述
* 数据块（具体携带的数据，又分为多种类型）

Json格式为：

{code:1, msg: 成功, data: …..}

对应的android端java对象为：com.mfh.comn.net.**ResponseBody**，其中data类型为下面所述的IResponseData类型。

详见com.mfh.comn.net包。

### 数据块格式

数据块即上述data标签后的部分，又分成下面几种类型,统一实现com.mfh.comn.net.data.IResponseData接口。

* 空值，即无data部分数据；
* 简单值对象,参见RspValue<T>对象，如String、Integer、Long、Date等，其中data序列化后的字串样式需要双方自行协商；完整格式如：

{ data:helloWorld}, 或：

{ data:{val:helloWorld}}

* 查询数据结果集，参见RspQueryResult <T>对象，支持对某实体的查询结果列表：

{total:12,rows:[

{prop1:value1, prop2:value2, prop3:value3,…},

{ prop1:value1, prop2:value2, prop3:value3,...}

]} 其中total和rows都是关键字

* 查询编码结果集：

{deepType:0, levelNum:2, levels:[{…},{…}],

options:[

{ code:1,hasChild:false,levelName:codeType,value:待签},

{ code:2,hasChild:false,levelName:codeType,value:租约中},

{ code:3,hasChild:false,levelName:codeType,value:已售}

]}

* Map<String, String>数据，参见RspMap对象

{props:{key1:value1,key2:value2, ....} 其中props是关键字

* 单bean，参见RspBean<T>对象,

{prop1:value1,prop2:value2, prop3:value3, ...}

* 单bean+caption， 对应com.mfh.comn.bean.EntityWrapper<T>对象，支持对bean的某些属性值有中文描述，适合界面显示，如下面的prop2属性除了有value2属性值外，value2对应的中文描述为valueCaption2：

{

bean:{ prop1:value1,prop2:value2, prop3:value3, ...},

caption:{prop2:valueCaption2}

}

* 混合结果，上述几种数据块格式的任意组合，参见RspMixData对象。

### 数据包解析器

针对json格式的使用JsonParser：

JsonParser parser = new JsonParser();

ResponseBody resp = parser.parser(rawValue, pojoClass);

或resp = parser.parser(rawValue, pojoClass, dataFormat);

针对xml格式的使用XmlParser。

## 统一文件操作

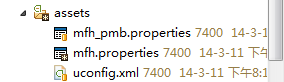
系统提供了针对android本地文件访问的基础对象com.dinsc.comn.dao.BaseFao，主要是统一管理应用程序使用的根目录，避免存放混乱，调用BaseFao.getFileDir(String relativeDirName)即可。

## N+1配置框架

### 概述

配置文件是整个应该比不可少的组成部分，配置文件也分为多个层次，比如默认配置文件(相当于出厂配置)，最终配置文件（相当于客户化配置）等等，可以分成多层，后一层的配置可以覆盖或合并前一层的配置，整个配置文件体系称为N+1配置。

配置文件本身格式可以是xml格式或property文件格式。我们约定android端配置文件统一放在assert目录下。如图：



其中uconfig.xml是整个N+1配置的入口定义文件，其他两个.properties文件分别代表不同用途的配置，每个用途是一个domain。

### Uconfig定义

Uconfig.xml中的定义如下：

<configs>

<classuconfig=*"com.mfh.comna.cfg.DefaultUConfig"*>

<itemtype=*"properties"*>com.mfh.comna.cfg.PropertiesConfiguration</item>

<itemtype=*"xml"*>com.mfh.comna.cfg.XMLConfiguration</item>

</class>

**<configdomain="COMMON"description="终端公共配置项"merge-rule="default">**

**<defaulttype="properties"location="mfh.properties"/>**

**<extendstype="properties"location="intern:custom/mfh.properties"/>**

**</config>**

**<configdomain="APP"description="物业管理应用配置项"merge-rule="default">**

**<defaulttype="properties"location="mfh\_pmb.properties"/>**

**<extendstype="properties"location="intern:custom/mfh\_pmb.properties"/>**

**</config>**

</configs>

上述文件中，

Domain用以标识不同的配置用途，一个domain对应一组配置文件，后面在通过接口获取某个配置项时也需要传递domain参数。

关于merge-rule合并规则有如下几种：

* order: 对default location 和 extends location 按照顺序查找, 找到第一个就返回
* replace: 对default location 和 extends location 按照 1 + N 合并(返回最后一个N配置),于order正好相反
* merge: 对default location 和 extends location 读取为配置列表(仅针对XML, 合并Document)

缺省规则: replace

location可以是：

* assets:代表是assets目录下,是默认位置；
* intern:代表android程序私有目录；
* extern:使用外部存储目录下的;
* raw:代表res/raw目录下的。

### 调用接口

首先初始化（若android应用从ComnApplication继承可以省略此步）

ConfigsParseHelper.init(configAlias, stream);

UConfigCache uconfig = MyApplication.getUconfig();//先获取配置对象

uconfig.getDomain("domainName").getString(“configItemName”);

或

uconfig.getDomain("domainName").getString(“configItemName”, “defaultName”);

其中uconfig对象在上述BaseService基类中已经获取，作为一个基本属性存在了。

## 序列生成模块

提供多序列生成器，包括UUID、整形序列、长整形序列、字符序列，并且序列生成可以跨手机唯一。

使用时通过service工厂获取序列服务：

com.mfh.comna.seq.SequenceService seqService =

ServiceFactory.getSequenceService();

然后可以进行如下调用：

seqService.getUUId();

public long getNextSeqLongValue(String sequeceName);

public String getNextSeqStringValue(String sequeceName);

为了支持跨手机生成的序列也是唯一的，需要在domain为COMMON的配置文件中作如下配置：

#use for uuid key

db.idchar=0001

#use for int-long key

db.idp0=10000

db.idp1=100000

db.idp2=1000000

db.idp3=10000000

注意：在上述BaseSeqAbleDao基类中已经集成了序列服务，提供了本地数据库操作时表主键的自动生成服务。

## 统一编码模块

BaseService中提供了codeService成员变量，用于统一访问各类编码。但每个具体的servcie里应负责初始化自己管辖范围内的编码提供者，或initService进行统一初始化，如：

codeServcie.init(String domain,....)或add(.....)

同时工程中的建议建立一个CodeDomain常量类统一定义各个编码大类常量。

## 统一登录和授权模块

登录后台：

com.mfh.comna.login.LoginService接口

实现类：

com.mfh.comna.login.logic.MfhLoginService：

1、支持同步和异步登录两种方法。其中异步登录是正常的登录方法，而同步登录则是用于任何网络调用发生异常后检测是会话超时时自动进行登录然后重新发起调用请求。参见HttpHandle中的makeRequestWithRetries方法，要求Java服务器端在会话超时返回时设置“needLogin”head头，如：

response.addHeader("needLogin", "true");

2、调用登录方法后可以获取sessionId，此后任何方法调用都应在url中加上该参数。

本地登录框架（需要本地数据库，备用）

PrivSession.initPrivSession().login(...);

之后：

PrivSession.getPrivSession().getCurrentUserId();

## 统一程序升级框架

程序升级是app应用必不可少的特性，后台java服务本身提供了升级服务：

1. 版本检测接口,提供apk的名字作为参数

/app/update/version?apk=….

后台在{app.fileServer.default}/apk目录下存放着两个文件，一个是以apk命名的json文件，里面描述了apk文件的版本信息；一个是apk文件本身。如版本文件：pmbClient.json，里面内容假设如下：

{

versionName:1.0.7, //版本描述

versionCode:7, //版本描述

apkName:mfh-pmb-client-7.apk //该版本对应的apk文件名

}

然后在同名目录下放置真正的apk文件mfh-pmb-client-7.apk。

1. 升级包下载接口,如：

/app/update/download/mfh-pmb-client-7.apk

客户端需要按下面几步配合进行：

AppUpdate update = new AppUpdate(this, apkName);

//执行版本检测

update.checkServVersionCode(null);

考虑到升级过程比较耗时，故update对象支持是否按异步模式进行，默认为异步。

为了方便起见，这些逻辑都封装在了InitActivity类里，子类只需继承**getUpdateParamToken**()方法即可。

3、增量升级的实现

升级工具类在 com.mfh.comna.utils 中的UpdateUtil类。

实现原理：分析两个版本apk文件的差异，获得差异文件。 差异文件再 和老版本的apk合并，生成新版本apk。

使用：调用UpdateUtil类public static void makeNewApk(Context context,String downPath,String newApkPath) ，方法介绍参见类中的描述。

此过程可能比较费时， 需要在子线程中执行。

优点：用户体验较好，节省流量。

缺点：不方便管理， 服务器需要多个版本的的差异文件，app中也需要加更多的逻辑。

## 本地数据库版本升级框架

Android程序一旦开始使用本地数据库，随着程序的升级，后面就会面临本地数据库版本也可能需要升级，例如增加一张表或给已有表增加一个字段等等。为了尊重客户数据价值，不建议将本地数据库删除重建，因此有必要提供一个平滑的本地数据库升级方案：

在InitService里，有如下代码启动了本地数据库自动升级过程：

UpgradeConfigParseHelper helper = new UpgradeConfigParseHelper();

SqlliteUpgradeSupport support = new SqlliteUpgradeSupport();

helper.doDbUpdate(uconfig.getDomain(UConfig.CONFIG\_DBUPGRADE), support);

具体要求在uconfig.xml统一配置文件里增加关于本地数据库升级的配置域，域名即为UConfig.CONFIG\_DBUPGRADE，如：

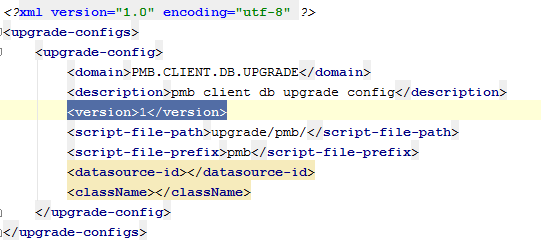
<config domain="DB.UPGRADE" description="平台数据库升级配置" merge-rule="merge">

<default type="xml"

location="assets:upgrade/pmb/pmb\_db\_upgrade.xml"/>

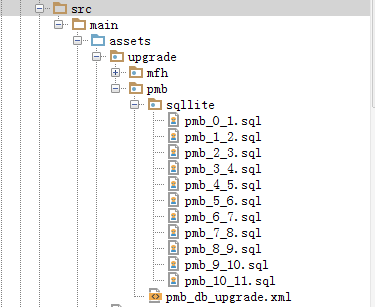
</config>

显然具体的升级配置文件在assert目录下的pmb\_db\_upgrade.xml文件中进行配置，每当需要进行一次升级时，将其中的version指定到指定版本即可,如图：



然后在对应的升级脚本文件目录下(assets/upgrade/pmb/sqllite/)放置具体的sql升级脚本文件，文件命名为：pmb\_0\_1.sql,此处的pmb即为上述配置的前缀，0代表上一版本，1代表当前要升级到的版本。Sql文件中每一条sql语句以分号结尾，可以提供多条sql语句一起升级。

如图：



其中：pmb\_7\_8.sql打开如下：

alter table emb\_session add channelkey VARCHAR;

alter table tb\_work\_order add flowCode VARCHAR;

alter table tb\_work\_order add area VARCHAR;

alter table tb\_work\_order add mobile VARCHAR;

alter table tb\_work\_order add sourceType VARCHAR;

alter table tb\_work\_order add ownerId INTEGER;

alter table tb\_work\_order add isRead INTEGER;

--增加通讯录 human表 updateDate字段

alter table emb\_member add updatedDate DATETIME;

## 头像/图片显示模块

提供了很好用的FineImgView类, 用于头像/图像显示的自定义类。图像显示在android中陷阱很多，特别是显示网络上的一个图片链接。此处对图像显示进行了封装，支持如下有用特性：

* 支持先从网络（网络地址通过配置项serverUrlconfigItem指定）上下载存到本地(本地目录名在变量localDir中指定);
* 然后再显示；以后就直接从本地进行显示了;
* 支持图片过大时使用缩略图显示避免内存溢出;
* 支持内存缓存，对生成的bitmap进行缓存可以大大加快显示速度。

## 统一错误收集模块

收集客户端运行的错误信息，提交到服务端，便于收集运行日志和排错。待定。

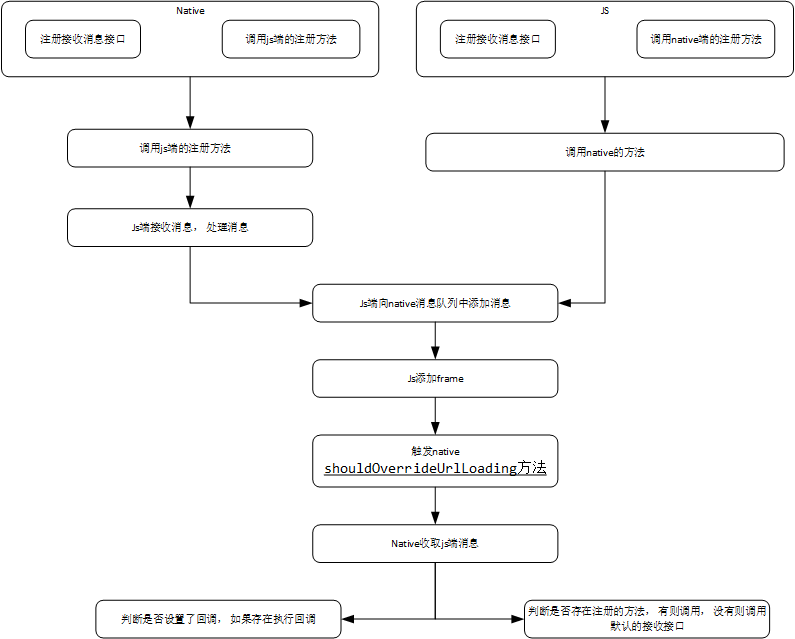
## 统一配置模块

获取服务器配置项， 通过ServerConfig 在InitActivity 中初始化, 访问服务器配置数据保存到本地。 本地需要的时候直接从ServerConfig中获取， ServerConfig采取的是单例。

## JS桥模块

### 原理

1. 流程图



1. 详细说明

①Native调用JS示例，首先JS端注册一个方法，

bridge.registerHandler(“methodName”,  
 new WebViewJavascriptBridge.WVJBHandler() {  
 @Override  
 public void handle(String data, WebViewJavascriptBridge.WVJBResponseCallback responseCallback) {  
 //*TODO，处理Native端发送的数据*  
 }  
 });

然后Native调用对应的JS端注册的方法，将数据用json字符串传入,并提供回调处理JS端的响应结果。

bridge.callHandler(“methodName”, “jsonData”,  
 new WebViewJavascriptBridge.WVJBResponseCallback() {  
 @Override  
 public void callback(String responseData) {  
 //*TODO，处理JS端返回的响应结果*

}  
 });

②JS调用Native端原理类似，是由JS端调用Native注册的方法，发送数据并处理响应结果。

### 2. Native端使用

1.配置XML，添加自定义WebView控件。

<com.mfh.comna.api.web.HybridWebView  
 android:id="@+id/webview"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:scrollbars="none"/>

1. 初始化WebView 和 JS桥。

@Bind(R.id.webview)  
HybridWebView mWebView;

WebViewJavascriptBridge bridge = new WebViewJavascriptBridge(this, mWebView, new UserServerHandler(), webviewDelegate);

class UserServerHandler implements WebViewJavascriptBridge.WVJBHandler{  
 @Override  
 public void handle(String data, WebViewJavascriptBridge.WVJBResponseCallback responseCallback) {

//*TODO，*  
 }  
 }

WebViewDelegate webviewDelegate = new WebViewDelegate(){  
 @Override  
 public void onPageFinished(WebView view, String url) {  
 }  
 @Override  
 public void onReceivedError(WebView view, int errorCode, String description, String failingUrl) {  
 }  
 @Override  
 public boolean shouldOverrideUrlLoading(WebView view, String url) {  
 return false;  
 }  
 @Override  
 public void onReceivedTitle(WebView view, String title) { }  
 @Override  
 public void onReceivedIcon(WebView view, Bitmap icon) {  
 }  
 @Override  
 public void onProgressChanged(WebView view, int newProgress) { }  
 };

1. 注册Native方法

bridge.registerHandler(“methodName”,  
 new WebViewJavascriptBridge.WVJBHandler() {  
 @Override  
 public void handle(String data, WebViewJavascriptBridge.WVJBResponseCallback responseCallback) {

//*TODO，*

}  
 });

1. 调用JS方法

bridge.callHandler(“methodName”, “jsonData”,  
 new WebViewJavascriptBridge.WVJBResponseCallback() {  
 @Override  
 public void callback(String responseData) {  
 //*TODO，* }  
 });