GIT工具使用

• 服务器搭建（截图见PPT）

• 客户端安装及使用

• 常见操作

• 注意事项：

• 链接的处理https://admin@localhost:8443/r/takeout.git

1、admin@删除

2、localhost换成ip地址

• 由于服务器存在安全证书

客户端如果想要链接上必须对此问题进行处理，即不做安全校验

Cmd中输入命令：git config --global --bool --add http.sslVerify false

• 工程提交时需要设置忽略文档信息（原则：在本地能够生产的文件都不需要提交）

• 代码更新时需要先从服务器获取最新的更新，然后拉取代码。

•

MVC转换到MVP

MVC模式

M：model

V：view

C：controller

在Android中默认按MVC的方式构建项目，V—layout，C—Activity。

登陆案例代码分析。

MVP模式

M：model

V：view

P：presenter

我们这里把与业务有关的代码都放到了P中。

解耦和方式

• 利用配置文件，使用反射获取到需要加载的对象。

• 设计模式：单例、工厂、观察者……

Dagger2

配置

• apt插件的安装

project的gradle中添加

// Add plugin https://bitbucket.org/hvisser/android-apt

buildscript {

dependencies {

classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'

}

}

2、apt插件的使用

modle的gradle中添加

// Apply plugin

apply plugin: 'com.neenbedankt.android-apt'

3、添加dagger2的依赖

// Add Dagger dependencies

dependencies {

compile 'com.google.dagger:dagger:2.6'

apt 'com.google.dagger:dagger-compiler:2.6'

}

注解使用

@Module、@Provides、@Component、@Inject

@Module：标注对象创建的容器

@Provides：标识需要创建的对象

@Component：将创建好的对象与需要注入的对象组合到一起

使用的设计模式

1、单例

2、Builder建造者模式：用于复杂对象的创建，比如：AlertDialog.Builder

3、工厂模式

ButterKnife工具使用

配置依赖：**compile 'com.jakewharton:butterknife:5.1.1'**

安装插件：在plugins中搜索butterknife,安装Android ButterKnife Zelezny

控件初始化操作：

Retorfit工具使用

主页: https://github.com/square/retrofit

配置:

添加Retrofit依赖:

compile **'com.google.code.gson:gson:2.2.4'**compile **'com.squareup.retrofit2:retrofit:2.1.0'**compile **'com.squareup.retrofit2:converter-gson:2.1.0'**

使用步骤:

请求方法:@GET / @POST

URL处理:测试链接http://localhost:8080/TakeoutService/login?username="itheima"&password="bj"

替换原则：

1、@Path - 替换参数

@GET("/group/{id}/users")

public Call<List<User>> groupList(@Path("id") int groupId);

2、@Query - 添加查询参数

@GET("/group/{id}/users")

public Call<List<User>> groupList(@Path("id") int groupId, @Query("sort") String sort);

3、@QueryMap - 如果有多个查询参数，把它们放在Map中

@GET("/group/{id}/users")

public Call<List<User>> groupList(@Path("id") int groupId, @QueryMap Map<String, String> options);

创建Retrofit对象, 并发起请求.示例代码:

// 构建Retrofit实例

Retrofit retrofit = new Retrofit.Builder().

baseUrl(API2).

addConverterFactory(GsonConverterFactory.create()).

build();

// 构建接口的实现类

IpAPI weatherAPI = retrofit.create(IpAPI.class);

// 调用接口定义的方法

Call<IPModel> weatherCall = weatherAPI.getWeather("8.8.8.8");

// 异步执行请求

weatherCall.enqueue(new Callback<IPModel>() {

@Override

public void onResponse(Call<IPModel> call, Response<IPModel> response) {

IPModel model = response.body();

System.out.println("country:" + model.getData().getCountry());

}

@Override

public void onFailure(Call<IPModel> call, Throwable t) {

System.out.println(t.toString());

}

});

Ormlite工具使用

配置

compile **'com.j256.ormlite:ormlite-android:5.0'**

使用

常用注解

@DatabaseTable(tableName = "t\_user")

指定实体和表的一一对应关系

@DatabaseField()

指定属性与表中列的一一对应关系

常用配置说明：

主键：id = true

自增主键：generatedId = true

列名：columnName = "name"

是否为空：canBeNull = true

外键：foreign = true

示例：

@DatabaseTable(tableName = **"t\_address"**)  
**public class** AddressBean {  
 @DatabaseField(id = **true**)  
 **private int \_id**;  
 @DatabaseField(canBeNull = **false**)  
 **private** String **goodsAddress**;  
 @DatabaseField(canBeNull = **false**)  
 **private** String **village**;  
 @DatabaseField(canBeNull = **false**,foreign = **true**,columnName = **"user\_id"**,foreignColumnName = **"\_id"**)  
 **private** UserBean **user**;

}

@DatabaseTable(tableName = **"t\_user"**)  
**public class** UserBean {  
 @DatabaseField(id = **true**)  
 **private int \_id**;

@ForeignCollectionField(eager = **true**)  
**private** ForeignCollection<AddressBean> **addressList**;

}

DBHelper管理

与原声的DBHelper相同，主要负责数据库的创建和表的维护。需要继承OrmLiteSqliteOpenHelper类。

使用工具：TableUtils完成表的相关操作，如创建、清空、删除

项目整合

包结构划分。项目比较复杂时，大家开始动手完成代码前必须要想清除，代码是放在哪里的。

业务层联网操作

依据接口文档中给出的服务器回复信息，设置Retorfit。由于服务器回复的是Json文件，并且服务器地址一般情况下不会变动，所以Retorfit的创建可以在业务层基类中完成

**new** Retrofit.Builder().  
 baseUrl(Constant.***HOME***).  
 addConverterFactory(GsonConverterFactory.*create*()).  
 build();

从接口文档中我们知道，回复的数据格式是统一的，我们仅仅封装一个对象就可以满足访问网络的需求

与之对应的API接口设置一个就能满足需求。

不同的模块在处理时只需要增加对应的方法即可，方法返回的类型是相同的。所以我们自需要创建ResponseInfoAPI的实现类对象就能满足需求。

ResponseInfoAPI responseInfoAPI = **retrofit**.create(ResponseInfoAPI.**class**);

可以集成到业务的基类中。

联网回复结果处理

当获取到服务器返回数据后会出发设置好的Callback,两个方法如下：

**public void** onResponse(Call<ResponseInfo> call, Response<ResponseInfo> response)

public void onFailure(Call<ResponseInfo> call, Throwable t)

我们需要对回复的结果做进一步处理，首先必须要判断code值，如果为0表示当前请求操作服务器处理成功，返回用户想要数据，如果不为0表示服务器处理该请求出现问题，比如：用户名或密码输入错误。这个信息我们需要统一展示给用户。所以我们需要对两个方法进行统一处理。在onResponse中需要

ResponseInfo body = response.body();  
**if**(**"0"**.equals(body.getCode())) {  
 *//* 服务器处理成功，可以解析*data*数据了parseDestInfo(body.getData());  
}**else**{  
 String error=**errorInfo**.get(body.getCode());  
 onFailure(call,**new** RuntimeException(error));  
}

如果出现服务器处理错误会出发onFailure方法，同时由于网络问题也会触发该方法，我们需要对出发来源进行区分，可以定义一个自己的异常，封装服务器返回错误提示信息，展示给用户，如果是网络问题则提示：请检查网络，或服务器忙等。代码如下（这里使用了RuntimeException）

public void onFailure(Call<ResponseInfo> call, Throwable t) {

if(t instanceof RuntimeException){

showError(((RuntimeException)t).getMessage());

}

showError("服务器忙，请稍后重试……");

}

业务层数据库操作

数据库需要使用到DBHelper，所以我们在DBHelper类中实现了单例，这样就不用每次都创建了。

需要重点注意的问题，操作DBHelper时需要传递上下文，我们需要使用生命周期较长的Application作为参数，不能使用某个具体的Activity或Fragment对象。

公共布局处理

首先完成MainActivity，布局比较简单，我们经常实现，所以layout大家可以直接拷贝了。

问题是：很多项目中都会有这种需求，但Item的数量不等，我们不得不调整代码，有通用的实现方式吗?

注意我们Layou的底部导航部分内容。

这样进行布局的目的是我们刻意批量的处理各种事件。

比如：底部导航每个Item都有自己的点击事件，那么我们只需要获取到底部导航容器，对他的孩子（不包含孙子），循环一次就可以设置好点击事件。代码如下：

**int** childCount = **switcherContainer**.getChildCount();  
**for** (**int** i = 0; i < childCount; i++) {  
 **switcherContainer**.getChildAt(i).setOnClickListener(**switherOcl**);  
}

在某个孩子被点击后我们可以处理以下工作：

• 修改对应状态，达到切换图片背景和文字颜色的效果

• 切换Fragment，我们可以把用到Fragment按顺序添加到集合中，当点击某个按钮后，会获取到期再底部容器中的具体位置信息，位置从0开始，而Fragment集合也是从0开始基数，所以他们的位置是一一对应的。

这里边有一个让人非常恼火的问题，每个Item都有两张不同的图片，我们需要逐个完成选择器的编写，这里面给大家介绍一个插件，他可以帮助我们生产这部分代码，只要你每组图片的命名按照其规定要求就能够帮助我们生产选择器。

命名规则可以查询提供的官方文档。

操作截图：

Fragment切换

将每个底部容器Item对应的Fragment放到一个容器中，注意Fragment的添加顺序，需要与Item的顺序保持一致，然后我们就可以通过Item的index找到对应的Fragment进行切换了。