Android 网络编程三「进阶」

原文链接：https://shimo.im/doc/qLjNJ8gKwSs9D6GC链接在保存文档时自动生成，点击可以查看该文档的最新版本

我们上一篇介绍了基本的网络封装，但是只是以一些初级的知识来顺便给大家说明下封装的一些原则以及如何进行所谓的封装，今天我们就来进行更深层次的封装。

我们知道，Volley 发起一个 get 请求，并传递一些参数是这样的：

```
RequestQueue mQueue = Volley.newRequestQueue(context);

StringRequest stringRequest = new StringRequest("http://stormzhang.com/?param1=value1¶m2=value2", listener, errorListener);

mQueue.add(stringRequest);
```

上述 url 中 "?" 后面的部分即为参数部分，每一个参数对用 "&" 进行连接，这是 get 约定的一种传递参数的方法，也叫 query string。

那我们再看看 post 方法如何传递参数：

```
RequestQueue mQueue = Volley.newRequestQueue(context);

StringRequest stringRequest = new StringRequest(Method.POST, "http://stormzhang.com/", listener, errorListener) {

    @Override

    protected Map<String, String> getParams() throws AuthFailureError {

        Map<String, String> map = new HashMap<String, String>();

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

        map.put("param2", "value2");
```

```
        return map;

    }

};

mQueue.add(stringRequest);
```

这里有两个问题：

1. 每次都要这样传递参数太累了，太多的重复性代码；
2. get 与 post 方法传递的参数不一致，而实际使用中 get 与 post 方法对调用者来说应该没多大区别；

所以，我们这里肯定要做进一步的封装，在上篇文章的基础封装上，要让功能更加强大，使它使用在各种场景下。

参数的封装

在做进一步的封装之前，我们首先得先考虑下我们的实际使用场景，我们大部分网络请求都是以 json 数据格式来约定的，不管是返回的数据也好，还是请求的参数也好，除了简单的 String 类型的 key、value 对参数传递，我们很可能会传递一些 json 格式的参数，如：

```
{ "user":{

        "name":"stormzhang",

        "age":18,

        "desc":"handsome"

    }

}
```

所以我们自己封装的参数类型要满足最起码的 json 格式的参数。

PS：这里稍微提一点，我相信肯定有人搞不清 Volley 是如何传递 json 格式的参数的，Volley 中有个 JsonRequest 就是默认支持传递 json 格式的参数，其实就是把 json 先转成 json 格式的字符串进行传递的，注意，json 跟 json 格式的字符串是有本质区别的，比如

```
{"age":18}
```

这是一个 json 数据，以大括号结尾，如果把它转成字符串是这样的：

```
"{\"age\":18}"
```

这本质上是个字符串，只不过是 json 格式化的字符串。而传递给服务端本质上就是传递这样

的一种字符串，只需要在 Header 中声明：

```
{"Content-Type":"application/json; charset=utf-8"}
```

这就告诉了服务端传递的是一个 json 格式的字符串，然后以 utf-8 进行编解码，然后服务端拿到这些字符串会自动转成 json 数据。

毫无疑问，通过上篇文章，相信大家的第一印象就是封装一个类，来单独处理参数，我们姑且取名 JsonParams，类中大概是如下这样：

```
public class JsonParams {

    protected final JSONObject params = new JSONObject();

    public JsonParams() {

    }

    public void put(String key, String value) {
        if (key != null && value != null) {
            try {
                params.put(key, value);
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
    }

    public void put(String key, boolean value) {
        if (key != null) {
```

```
        try {
            params.put(key, value);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}

public void put(String key, int value) {
    if (key != null) {
        try {
            params.put(key, value);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}

public void put(String key, float value) {
    if (key != null) {
        try {
            params.put(key, value);
        } catch (JSONException e) {
            e.printStackTrace();
        }
```

```
    }
}

public void put(String key, String[] value) {
    if (key != null && value != null) {
        try {
            params.put(key, new JSONArray(Arrays.asList(value)));
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}

public void put(String key, JSONArray value) {
    if (key != null && value != null) {
        try {
            params.put(key, value);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}

public void put(String key, JSONObject value) {
    if (key != null && value != null) {
```

```
        try {
            params.put(key, value);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}

public JsonParams remove(String key) {
    params.remove(key);
    return this;
}

public JSONObject toJson() {
    return params;
}

public String toString() {
    return params.toString();
}

private String getEncodedParamString() {
    StringBuilder result = new StringBuilder();
    Iterator it = params.keys();
    while (it.hasNext()) {
```

```
            if(result.length() > 0)
                result.append("&");

            String key = (String) it.next();
            String value = params.optString(key);
            result.append(URLEncoder.encode(key));
            result.append("=");
            result.append(URLEncoder.encode(value));
        }
        return result.toString();
    }



    /**
     * 把参数转成 query string
     * @return String
     */
     public String toQueryString(String url) {
        String paramString = getEncodedParamString();
        if (TextUtils.isEmpty(paramString)) {
            return url;
        }

        if (url.indexOf("?") == -1) {
            url += "?" + paramString;
```

```
        } else {
            url += "&" + paramString;
        }
        return url;
    }
}
```

代码就不过多解释了，其实很简单，主要就是各种数据类型的 put 方法，值得一提的是最后一个方法 getEncodedParamString()，我们如果想让 get 跟 post 支持同样的传递参数的用法，那么我们需要把 JsonParams 转化成 query string 的形式，其实逻辑就是把一个 json 格式的数据每一个 key 拼接成 key1=value&key2=value2 的形式，然后做一些编解码。而 toQueryString() 方法主要负责把原本的请求 url 和 参数进行拼接，比如如果一个 url 是 http://stormzhang.com ，那么就需要把这个 url 与参数用 "?" 进行连接，而如果一个 url 本身就带参数，如 http://stormzhang.com/?params1=value ，那么就需要把这个 url 与参加用 "&" 进行拼接就 ok 了。

好了，JsonParams 我们封装好了，接下来我们怎么把它应用到我们的 Request 中呢?

封装 Request

我们上面说了，我们的 Request 约定了是 JsonRequest，但是面向对象的第一大特性就是继承，我们应该给所有 Request 定义一个基类，在基类中处理一些公用的逻辑，如传递一些 App 版本号、手机型号、系统版本等信息，已经跟我们自定义的 JsonParams 进行关联起来。

所以，我们先来定义一个 BaseJsonRequest，如下:

```
public class BaseJsonRequest extends JsonRequest<String> {

    private JsonParams mJsonParams;

    public BaseJsonRequest(int method, String url,
                           JsonParams jsonParams,
                           Response.Listener<String> listener,
```

```
                    Response.ErrorListener errorListener) {
    super(method,
            url,
            null == jsonParams ? null : jsonParams.toString(),
            listener, errorListener);
}


/**
 * 所有和 api 无关的一些基本参数放在 header 里传递
 */
@Override
public Map<String, String> getHeaders() throws AuthFailureError {
    Map<String, String> headers = new HashMap<String, String>();
    headers.put("App-Version", Config.getVersionName());
    headers.put("App-Device", "Android");
    headers.put("Version-Code", Config.getVersionCode());
    return headers;
}


/**
 * 需要实现父类的抽象方法
 */
@Override
protected Response<String> parseNetworkResponse(NetworkResponse response) {
    if (response == null) return null;
    String parsed = null;
```

```
        try {
            parsed = new String(response.data,
HttpHeaderParser.parseCharset(response.headers));
        } catch (Exception e) {
            parsed = new String(response.data);
        }
        return Response.success(parsed,
HttpHeaderParser.parseCacheHeaders(response));
    }
}
```

看到没有，其实很简单，主要只做了两个处理，一个就是构造方法中，把 JsonParams 作为参数传递过去，另一个就是在基类的 Header 中放一些通用的参数，这个就根据自己的需求去自行添加更多了。

然后就简单了，我们上一篇提到的 StormClient 类就会是如下这样的：

```
public class StormClient {

    public static final String QA = "http://stormzhang.cn";
    public static final String PRODUCTION = "http://stormzhang.com";

    public static String getHost() {
        return BuildConfig.DEBUG ? QA : PRODUCTION;
    }

    public static String getAbsoluteUrl(String url) {
        return getHost() + url;
    }
```

```
    /**
     * 不带参数的 get 方法
     */
    public static void get(String url, HttpCallback callback) {
        get(url, null, callback);
    }

    /**
     * 带参数的 get 方法
     */
    public static void get(String url, JsonParams params, HttpCallback callback) {
        RequestManager.sendRequest(Request.Method.GET, getAbsoluteUrl(url), params,
callback);
    }

    /**
     * 不带参数的 post 方法
     */
    public static void post(String url, HttpCallback callback) {
        post(url, null, callback);
    }

    /**
     * 不带参数的 post 方法
     */
```

```
    public static void post(String url, JsonParams params, HttpCallback callback) {

        RequestManager.sendRequest(Request.Method.POST, getAbsoluteUrl(url), params,
callback);

    }

}
```

我们再来看下 RequestManager 类：

```
public class RequestManager {



    public static RequestQueue mQueue =
Volley.newRequestQueue(MyApplication.getContext());



    public static void addRequest(Request<?> request) {

        mQueue.add(request);

    }



    public static void sendRequest(int method, String url, JsonParams jsonParams,
HttpCallback callback) {

        String fullURL = url;

        // 注意下面代码，用来处理 get 方法的参数

        if (Request.Method.GET == method && jsonParams != null) {

            fullURL = jsonParams.toQueryString(fullURL);

        }



        BaseJsonRequest request = new BaseJsonRequest(method, fullURL, jsonParams,
callback, callback);

        RequestManager.addRequest(request);

    }
```

```
}
```

上面代码加了点注释，比较简单，相信大家应该都能看得懂。

用起来就很简单了：

```
// 不管是 get、post，传递参数是一样的

JsonParams params = new JsonParams();

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

params.put("param2", "value2");

StormClient.get("/api/v1/home", params, new HttpCallback() {});
```

至此，我们平时用到的 get、post 请求的最常用的使用场景基本都满足了，请对照这两篇文章，仔细消化整个封装的过程，最好能把代码亲自敲一遍，运行看下效果。

你以为这就完了么？并没有，网络封装还有一个比较重要的点：缓存。

缓存

我们知道网络请求一定涉及到缓存，缓存用得好可以提升用户体验，虽然一些网络框架默认支持 ETag、时间缓存等，但是这都是在联网状况下发生的，是为了减少请求次数，缓解服务器压力而生的。我们想要的是从客户端的用户体验出发，用户如果请求网络失败，那么我们应该默认显示上一次的请求结果，而这样的需求，就需要我们自己来实现。

缓存数据的方式要很多种，有 Sqlite、File、SharedPreference，最简单的做法是使用 SharedPreference，但是我还是建议缓存跟 SharedPreference 分开，所以我建议使用文件来做缓存，所以，我们先定义一个 FileCache 类：

```
public class FileCache {


    // 设置缓存目录名称

    private static final String CACHE_NAME = "volley";


    public static ACache cache = ACache.get(MyApplication.getContext(), CACHE_NAME);
```

```
    public static void put(String key, String value) {
        cache.put(key, value);
    }

    public static String get(String key) {
        return cache.getAsString(key);
    }

    public static void clear() {
        cache.clear();
    }
}
```

这个类的内容很简单，其中涉及到一个 ACache 类，其实这是一个很简单的开源缓存文件，支持设置缓存路径，缓存大小，缓存数量等，因为不是本文重点，就不多说了，具体用法见这里：https://github.com/yangfuhai/ASimpleCache

FileCache 文件就是做了简单点的封装而已，当然，实际开发中你可能会有其他需求，只需要在这个类做处理就好。

然后，我们只需要在 BaseJsonRequest 类里加上一点简单的处理，就可以把缓存存储起来：

```
public class BaseJsonRequest extends JsonRequest<String> {

    private String mUrl;

    public BaseJsonRequest(int method, String url,
                           JsonParams jsonParams,
                           Response.Listener<String> listener,
```

```
                    Response.ErrorListener errorListener) {
    super(method,
            url,
            null == jsonParams ? null : jsonParams.toString(),
            listener, errorListener);
    this.mUrl = url;
}

/**
 * 所有和 api 无关的一些基本参数放在 header 里传递
 */
@Override
public Map<String, String> getHeaders() throws AuthFailureError {
    Map<String, String> headers = new HashMap<String, String>();
    headers.put("App-Device", "Android");
    return headers;
}

/**
 * 需要实现父类的抽象方法
 */
@Override
protected Response<String> parseNetworkResponse(NetworkResponse response) {
    if (response == null) return null;
    String parsed = null;
    try {
```

```
            parsed = new String(response.data,
HttpHeaderParser.parseCharset(response.headers));
        } catch (Exception e) {
            parsed = new String(response.data);
        }
        // 加入缓存
        FileCache.put(mUrl, parsed);
        return Response.success(parsed,
HttpHeaderParser.parseCacheHeaders(response));
    }
}
```

可以看到上述黑字，只是在返回数据之前先把结果存储起来，这样默认就把所有请求的数据缓存起来了，默认缓存的 key 是请求的 url，接下来我们希望看到的是在请求失败之后，能默认显示缓存数据，同样，我们只需要在 HttpCallback 类做点变通处理：

```
public class HttpCallback implements Response.Listener<String>,
Response.ErrorListener {

    private String mCache;

    public HttpCallback() {

    }

    @Override
    public void onResponse(String response) {
        ok(response);
    }
```

```
    @Override
    public void onErrorResponse(VolleyError error) {
        String errorMessage = error.getMessage();
        if (mCache != null) {
            ok(mCache);
        }
        fail(errorMessage);
    }

    public void ok(String response) {

    }

    public void fail(String errorMessage) {

    }

    public void setCache(String cache) {
        this.mCache = cache;
    }
}
```

我们给 HttpCallback 新增了一个 setCache 方法，然后在 onErrorResponse 方法中加上上述一句代码，就这一个很小的变通，就可以在网络访问失败的时候，回调 ok() 方法，然后传递缓存数据。

接下来，在所有请求的r入口 RequestManager 也只需加上两行代码：

```
public class RequestManager {



    public static RequestQueue mQueue =
Volley.newRequestQueue(MyApplication.getContext());



    public static void addRequest(Request<?> request) {

        mQueue.add(request);

    }



    public static void sendRequest(int method, String url, JsonParams jsonParams,
HttpCallback callback) {

        String fullURL = url;

        // 注意下面代码，用来处理 get 方法的参数

        if (Request.Method.GET == method && jsonParams != null) {

            fullURL = jsonParams.toQueryString(fullURL);

        }



        // 设置请求失败支持缓存

        String cache = FileCache.get(fullURL);

        if (TextUtils.isEmpty(cache)) {

            callback.setCache(cache);

        }



        BaseJsonRequest request = new BaseJsonRequest(method, fullURL, jsonParams,
callback, callback);

        RequestManager.addRequest(request);

    }
```

```
}
```

在把请求加入队列之前，先检测下有没有该 url 对应的缓存，如果有，那么在回调中预先设置进去，ok，大功告成，你所有调用的方法没有做任何变动，我们只在我们之前封装的一些类中做了点简单的处理，就可以让所有请求失败的时候，直接显示了缓存的数据，这一切都是我们之前做了封装的好处。

mock 数据

到目前为止，看似我们封装的网络请求很完美，基本上满足了我们实际开发的大部分场景，但是，我们在实际开发中前端、后端是同时进行，很多时候前端有等后端的情况，其实这对开发效率是极大的影响，理想的开发是前端、后端在根据需求定好 api 文档之后，前、后端就可以并行进行开发了，这个时候前端就需要一个数据 mock 框架。

所谓 mock 数据，也就是所谓的假数据，也就是客户端在做 UI 开发的时候需要一些数据来填充，这个时候后端功能又没有实现好，为了解决这种场景，mock 的概念诞生了，如果我们有一个 mock 的框架，客户端自己模拟假数据填充，而无需等待后端，等到后端真正开发完成了，再进行简单的 api 连通的确认，这会极大的提升开发效率。

废话不多说，我们同样，只需要简单的做些处理就可以使我们的这套网络框架支持 mock，先看 HttpCallback：

```
public class HttpCallback implements Response.Listener<String>,
Response.ErrorListener {



    private String mCache;

    private String mMock;



    public HttpCallback() {



    }



    @Override

    public void onResponse(String response) {

        ok(response);

        if (BuildConfig.DEBUG && mMock != null) {
```

```
        mock(mMock);
    }
}

@Override
public void onErrorResponse(VolleyError error) {
    String errorMessage = error.getMessage();
    if (mCache != null) {
        ok(mCache);
    }
    fail(errorMessage);

    if (BuildConfig.DEBUG && mMock != null) {
        mock(mMock);
    }
}

public void ok(String response) {
}

public void fail(String errorMessage) {
}

public void mock(String mock) {
}
```

```
    public void setMock(String mock) {
        this.mMock = mock;
    }

    public void setCache(String cache) {
        this.mCache = cache;
    }
}
```

上述代码黑色部分是我们新增的部分，可以看到，我们仍然只是做了很小的处理，只这一步就满足了我们的需求，我们接下来看下调用方法：

```
public class MainActivity extends AppCompatActivity {

    private TextView contentText;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        contentText = (TextView) findViewById(R.id.text_content);

        HttpCallback callback = new HttpCallback() {
            public void ok(String reponse) {
                initUI(reponse);
            }
```

```
            public void fail(String response) {

            }


            public void mock(String mock) {
                initUI(mock);
            }
        };


        JsonParams params = new JsonParams();
        params.put("param1", "value1");
        params.put("param2", "value2");
        callback.setMock("stormzhang is handsome!");
        StormClient.get("", params, callback);
    }


    private void initUI(String response) {
        contentText.setText(response);
    }
}
```

可以看到，使用起来只需要先调用 setMock() 方法，把你要传递的假数据传递过去就 ok，为了演示方便，我只传递了字符串，这块你可以根据你的需求去定制，比如单独写一个 Mock 类专门放一些假数据，又或者把假数据放在一个个的 json 文件，放在 assets 目录里都可以，调用的时候只需要增加一个 mock() 方法就可以使用了，简单方便，正式环境可以移除掉这部分代码，当然如果发布版本忘记移除也没关系，因为我们在 HttpCallback 声明了 debug 环境才会起作用，release 环境 mock() 方法并不会响应。是不是觉得酷毙了？这就是封装的魅力。

以上是根据实际使用场景的一些高级封装，内容与代码都有点多，请大家耐心点消化，最好能理解了并且运行一遍体验下，看懂了不代表你真的理解了，只有自己亲自实践了才能更深刻的理解。另外，文章中的一些代码我觉得很详细了，但是考虑大家水平不一致，过几天我会单独

给大家写一个 demo 出来，顺便把源码发给你们，加深你们的理解与认识，真心希望大家看后对网络、对封装有个全新的认识，也希望大家能应用到你们的实际工作中。

但是我知道，很多人可能看懂了我这三篇文章，也理解了封装的一些基本原则，但是实际工作中自己仍然无从下手，这个是正常的，如果仅仅三篇文章就能教会每个人封装的话，那编程也太简单了，编程是需要大量的实践与总结的，不用着急，你在实际工作中多实践，多多思考，可能突然的某一天你就会顿悟，原来封装不过如此而已！

stormzhang
2016.12.26