Retrofit

32、Retrofit的作用和特点

> 1. Square公司开发的针对Android网络请求的框架
> 2. 基于OkHttp
> 3. 通过运行时注解的方式提供功能。
> 4. 支持NIO(new io)
> 5. 默认使用Gson解析

33、Retrofit的基本集成方法

```
//gradle.build
compile 'com.squareup.retrofit2:retrofit:2.1.0'
compile 'com.squareup.retrofit2:converter-gson:2.1.0'
compile 'com.squareup.retrofit2:converter-scalars:2.1.0'
compile 'com.squareup.retrofit2:converter-jackson:2.1.0'
compile 'com.squareup.retrofit2:converter-moshi:2.1.0'
compile 'com.squareup.retrofit2:converter-protobuf:2.1.0'
compile 'com.squareup.retrofit2:converter-wire:2.1.0'
compile 'com.squareup.retrofit2:converter-simplexml:2.1.0'
```

34、Retrofit的注解分类

> 1. HTTP请求方法注解 ： 8种-GET,POST,DELETE,HEAD,PATCH,OPTIONS和 HTTP ； HTTP 可以替换上面7种方法。
> 2. 标记类注解 ： 3种-FormUrlEncoded、Multipart、Streaming
> 3. 参数类注解 ： Header、Headers、Body、Path、Field、FieldMap、Part、PartMap、Query和QueryMap
>
> - Streaming 代表响应的数据以 流的形式 返回，如果不使用，会默认把全部数据加载到 内存 中，所以下载 大文件 需要加上该 Streaming注解

35、Retrofit的Get使用方法

> 1-定义接口

```
//Retrofit的请求URL是拼接而成：  = baseUrl + 网络接口中的URL
public interface GitHubService{
    @GET("users/{user}/repos")
    Call<List<Repo>> listRepos(@Path("user") String user);
}
```

> 2-Get请求

```
//1. 构建Retrofit和实例化接口
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.githuba.com/")
        .build();
GitHubService service = retrofit.create(GitHubService.class);
//2. 获取到指令，并在合适的时机去执行
Call<List<Repo>> repos = service.listRepos("feather");
//3. 同步调用
try {
    Response<List<Repo>> responseData = repos.execute();
    List<Repo> data1 = responseData.body();
} catch (IOException e) {
    e.printStackTrace();
}
//4. 异步调用
repos.enqueue(new Callback<List<Repo>>() {
    @Override
    public void onResponse(Call<List<Repo>> call, Response<List<Repo>> response) {
        List<Repo> data2 = response.body();
        //TODO 成功后进行处理
    }

    @Override
    public void onFailure(Call<List<Repo>> call, Throwable t) {
        //TODO 失败
    }
});
```

36、Retrofit的baseUrl和请求接口内URL(path)的整合规则

- path 是绝对路径的形式：

> path = "/apath", baseUrl = "http://host:port/a/b"
> Url = "http://host:port/apath"

- path 是相对路径，baseUrl 是目录形式：

> path = "apath", baseUrl = "http://host:port/a/b/"
> Url = "http://host:port/a/b/apath"

- path 是相对路径，baseUrl 是文件形式(没有目录结尾的/)：

> path = "apath", baseUrl = "http://host:port/a/b"
> Url = "http://host:port/a/apath"

- path 是完整的 Url：

> path = "http://host:port/aa/apath", baseUrl = "http://host:port/a/b"
> Url = "http://host:port/aa/apath"

37、Retrofit的参数类注解：Query

> 1. 用于在path后面附加上请求参数，如“？name=wch”

```
//1. 接口
public interface GitHubService{
    @GET("/")
    Call<String> getAgeByName(@Query("name") String name);
}
//2. 设置参数
Call<String> ageString = service.getAgeByName("wch");
//3. 会生成"https://www.baidu.com/?name=wch"这样的URL
```

38、Retrofit的@QueryMap-URL携带参数数量不定

> xxx/news?newsid=10&...

```
public interface GitHubService{
    @GET("news")
    Call<String> getNews(@Query("newsid") String newsID, @QueryMap Map<String, String> map);
}
//1. 传入一个固定参数，传入不定参数组成的HashMap
HashMap<String, String> paramsMap = new HashMap<>();
paramsMap.put("type", "event");
paramsMap.put("date", "2018/3/1");
Call<String> news = service.getNews("10", paramsMap);
```

39、Retrofit的的POST请求和@Field&FieldMap

> 1. POST 请求作用：能将 表单(附带各种数据：字符串、媒体类型等) 上传给服务器，而不是附加在 URL中 。
> 2. @FormUrlEncoded 指明是 表单请求
> 3. @Field 指明(key-value)中 key的内容 ， 参数填写的是 具体的value
> 4. @FieldMap 和 @QueryMap 使用方法类似。

```
//1. 接口
public interface GitHubService{
    ...
    @FormUrlEncoded
    @POST("getIpInfo.php")
    Call<String> getIp(@Field("ip") String first);
}
//2. POST请求：将信息都放到数据段中
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://ip.taobao.com/service/")
        .addConverterFactory(GsonConverterFactory.create()) //返回的数据用GSON解析
        .build();
GitHubService service = retrofit.create(GitHubService.class);
Call<String> ipCall = service.getIp("59.108.54.37");
ipCall.enqueue(new Callback<String>() {
    @Override
    public void onResponse(Call<String> call, Response<String> response) {
        Toast.makeText(context, "返回数据:" + response.body(), Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onFailure(Call<String> call, Throwable t) {
        Toast.makeText(context, "onFailure!", Toast.LENGTH_SHORT).show();
    }
});
```

40、Retrofit传输JSON字符串-@Body

> 1. POST 中通过 @Body 可以将 JSON字符串 作为请求体
> 2. Retrofit会将 @Body标注的对象 转换为字符串。

```
public interface GitHubService{
    @POST("getIpInfo.php")
    Call<String> getIpByJSON(@Body Ip ip);
}
public class Ip{
    private String ip;
    public Ip(String ip){
        this.ip = ip;
    }
}
...
//1. 自动将类转换为JSON字符串
Call<String> ipCall = service.getIpByJSON(new Ip("59.108.54.37"));
...
```

41、Retrofit的文件上传(@Part/@PartMap)

```
//1-@Multipart表示允许多个@Part
public interface FileUploadService {
    @Multipart
    @POST("upload")
    Call<ResponseBody> upload(@Part("description") RequestBody description,
                              @Part MultipartBody.Part file);
    //多个文件上传，不演示了
    @Multipart
    @POST("upload")
    Call<ResponseBody> multiUpload(@Part("description") RequestBody description,
                              @PartMap Map<String, RequestBody > files);
}
//2-上传的具体实现
        //...
        //1. 需要上传的文件
        File file = new File("failename");
        //2. 第一个参数: 用来传递简单的键值对(这里是描述符)
        String descriptionString = "This is a description";
        RequestBody description = RequestBody.create(MediaType.parse("multipart/form-data"), de
        //3. 第二个参数: 需要上传的文件组成的
        RequestBody requestFile = RequestBody.create(MediaType.parse("application/otcet-stream"
        MultipartBody.Part body = MultipartBody.Part.createFormData("aFile", file.getName(), re
        //4. 调用通过Upload方法进行文件上传
        Call<ResponseBody> call = service.upload(description, body);
        call.enqueue(new Callback<ResponseBody>() {
            @Override
            public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {
                System.out.println("success");
            }
            @Override
            public void onFailure(Call<ResponseBody> call, Throwable t) {
                t.printStackTrace();
            }
        });
```

42、Http请求中消息头的作用

> 1. 消息头 中可以添加一些 特殊的信息
> 2. 能防止防止攻击、过滤掉不安全的访问或者进行加密以进行权限验证
> 3. `Retrofit` 中通过 `@Headr` 能添加消息头。
> 4. `Retrofit` 中有 静态添加和动态添加 两种方法

43、Retrofit中@Header/@Headers的使用

> 1. `@Headers` 用于静态添加。
> 2. `@Header` 用于注解参数，用于动态添加。

```
public interface someService{
    //1. 静态添加一个消息头
    @GET("news")
    @Headers("Accept-Encoding: application/json")
    Call<ResponseBody> getNews1();
    //2. 静态添加多个消息头
    @GET("news")
    @Headers({"Accept-Encoding: application/json", "User-Agent: MoonRetrofit"})
    Call<ResponseBody> getNews2();
    //3. 动态添加消息头
    @GET("news")
    Call<ResponseBody> getNews3(@Header("Location") String location);
}
```