Android网络框架源码分析二---Retrofit

楚云之南 (/u/a79bd8f1db7c) + 关注

2016.01.03 23:11* 字数 3019 阅读 6654 评论 19 喜欢 68 赞赏 1

(/u/a79bd8f1db7c)

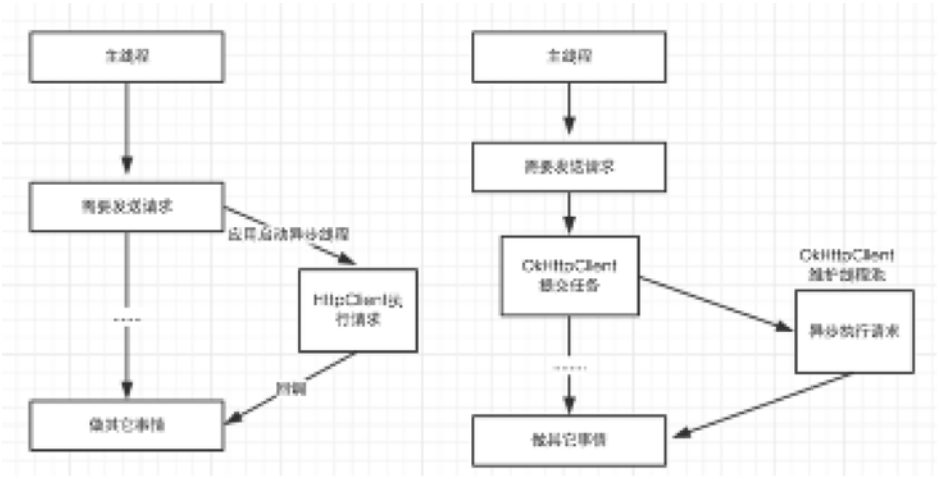
前面分析了 Volley 的代码，读者可能已经发现了基本上就是分析几个任务队列的处理逻辑和工作线程(网络工作线程和缓存工作线程)。 Volley 中的工作线程是自己使用线程数组来维护的，那么就有可能存在线程由于异常退出之后，没有下一个工作线程补充的风险(线程池可以弥补这个缺陷)。不管怎么样，个人觉得 Volley 代码是比较简洁、高效的，而且也比较适合阅读，建议大家花个小半天搂一眼源码。

ok，我们接着说今天的主题。俗话说社会在进步，人类在发展，精彩的故事发生地让人目不暇接，就让我们刚想站起来为 Volley 鼓掌时， Retrofit 又适时的开源了，并且适时地发布了2.0版本。在 Retrofit 2.0 中，最大的改动莫过于减小库的体积，首先， Retrofit 2.0 去掉了对所有的 HTTP 客户端的兼容，而钟情于 OkHttpClient 一个，极大地减少了各种适配代码,原因一会儿说;其次，拆库，比如将对 RxJava 的支持设置为可选(需要额外引入库)；再比如将各个序列化反序列化转换器支持设置为可选(需要额外引入库); 这次升级的其它内容，可以戳这里，中文版的哦， Retrofit 2.0升级演讲PPT总结 (https://link.jianshu.com?t=https://realm.io/cn/news/droidcon-jake-wharton-simple-http-retrofit-2/?utm_source=tuicool&utm_medium=referral)

> 注： 本文余下部分中出现的HttpClient均代指apache 的HttpClient实现

> 注： 对于2.0抛弃HttpClient和HttpURLConnection，为了减小库体积是一方面，另外一个重要的原因作为一个专门为Android&Java 应用量身打造的Http栈，OkHttpClient越来越受到各个大的开源项目的青睐，毕竟它本身就是开源的。

此外， OkHttpClient 与 HttpClient 相比，它最大的改进是自带工作线程池，所以上层应用无需自己去维护复杂的并发模型，而 HttpClient 仅仅提供了一个线程安全的类，所以还需要上层应用去处理并发的逻辑(实际上 Volley 一部分工作就是干这个)。从这个角度来说， Retrofit 得益于 OkHttpClient 的优势，较之于 Volley 是一种更加先进的网络框架。



Paste_Image.png

由于 Retrofit 不需要去关心并发工作线程的维护，所以它可以全力关注于如何精简发送一个请求的代价，实际上使用 Retrofit 发送一个请求实在是太easy了！

注意，本文并不是使用 `Retrofit` 的帮助文档，建议先看 `Retrofit` 的文档和 `OkHttp` 的文档，这些对于理解余下部分很重要。

使用Retrofit发送一个请求

假设我们要从这个地址 `http://www.exm.com/search.json?key=retrofit` 中获取如下Json返回：

```
{
  "data": [
             {
               "title":"Retrofit使用简介",
               "desc":"Retrofit是一款面向Android和Java的HttpClient",
               "link":"http://www.exm.com/retrofit"
             },
             {
                "title":"Retrofit使用简介",
                "desc":"Retrofit是一款面向Android和Java的HttpClient",
                "link":"http://www.exm.com/retrofit"
             },
             {
               "title":"Retrofit使用简介",
               "desc":"Retrofit是一款面向Android和Java的HttpClient",
               "link":"http://www.exm.com/retrofit"
             }
        ]
}
```

1.引入依赖

```
compile 'com.squareup.retrofit:retrofit:2.0.0-beta2'
//gson解析
compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'
```

2.配置Retrofit

```
Retrofit retrofit = new Retrofit.Builder()
                    .baseUrl("http://www.exm.com")
                    .addConverterFactory(GsonConverterFactory.create())
                    .client(new OkHttpClient())
                    .build();
```

3.新建Model类 SearchResult来解析结果

4.新建请求接口
Retrofit使用注解来定义一个请求，在方法上面指定请求的方法等信息，在参数中指定参数等信息。

```
public interface RestApi {
        @GET("/search.json")
        Call<List<SearchResult>> search(
            @Query("key") String key
             );

            //可以定义其它请求
           @GET("/something.json")
            Call<SomeThing> dosomething(
                   @Query("params") long params
                    .......
                    .......
              );

}
```

5.发送请求，我们可以发送同步请求(阻塞当前线程)和异步请求，并在回调中处理请求结果。

```
RestApi restApi = retrofit.create(RestApi.class);
Call<List<SearchResult>> searchResultsCall = resetApi.search("retrofit");
//Response<List<SearchResult> searchResults = searchResultsCall.execute();   同步方法
searchResultsCall.enqueue(new Callback<List<SearchResult>>() {
             @Override
             public void onResponse(Response<List<SearchResult>> response, Retrofit retro
                 content.setText(response.body().toString());
             }

             @Override
             public void onFailure(Throwable t) {
                 content.setText("error");
             }
         });
```

Retrofit源码分析

`Retrofit` 整个项目中使用了动态代理和静态代理，如果你不太清楚代理模式，建议先 `google` 一下，如果对于 `Java` 的动态代理原理不是太熟悉，强烈建议先看：这篇文章-戏说代理和Java动态代理 (https://www.jianshu.com/p/0d919e54eef0)
ok，下面按照我们使用 `Retrofit` 发送请求的步骤来：

```
RestApi restApi = retrofit.create(RestApi.class); //产生一个RestApi的实例
```

输入一个接口，直接输出一个实例。

> 这里岔开说一句，现在随便在百度上搜一下Java动态代理，出来一堆介绍AOP(面向切面编程)和Spring，导致一部分人本末倒置，认为动态代理几乎等于AOP，甚至有些人认为动态代理是专门在一个函数执行前和执行后添加一个操作，比如统计时间(因为现在几乎所有介绍动态代理的地方都有这个例子)，害人不浅。实际上动态代理是JDK提供的API，并不是由这些上层建筑决定的，它还可以做很多别的事情，Retrofit中对动态代理的使用就是佐证。

搂一眼这里的源码,再次建议，如果这里代码看不明白，先看看上面提到的那篇文章：

```
public <T> T create(final Class<T> service) {
  //返回一个动态代理类的实例
  return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] { service },
    //这个InvocationHandler是关键所在，以后调用restapi接口中的方法都会被发送到这里
    new InvocationHandler() {
      private final Platform platform = Platform.get();

      @Override
      public Object invoke(Object proxy, Method method, Object... args)
          throws Throwable {
        /* 如果是Object的方法，如toString()等，直接调用InvocationHandler的方法，
         *注意，这里其实是没有任何意义的，因为InvocationHandler其实是一个命令传送者
         *在动态代理中，这些方法是没有任何语义的，所以不需要在意
         */
        if (method.getDeclaringClass() == Object.class) {
          return method.invoke(this, args);
        }
       //对于Java8的兼容，在Android中不需要考虑
        if (platform.isDefaultMethod(method)) {
          return platform.invokeDefaultMethod(method, service, proxy, args);
        }
        //返回一个Call对象
        return loadMethodHandler(method).invoke(args);
      }
    });
}
```

我们可以看到 `Retrofit.create()` 之后，返回了一个接口的动态代理类的实例，那么我们调用这个代理类的方法时，调用自然就被发送到我们定义的 `InvocationHandler` 中，所以调用 `Call<List<SearchResult>> searchResultsCall = resetApi.search("retrofit");` 时，直接调用到 `InvocationHandler` 的 `invoke` 方法，下面是invoke此时的上下文：

```
@Override
   public Object invoke(Object proxy, Method method, Object... args)
       throws Throwable {
       //proxy对象就是你在外面调用方法的resetApi对象
       //method是RestApi中的函数定义，
       //据此，我们可以获取定义在函数和参数上的注解，比如@GET和注解中的参数
       //args,实际参数，这里传送的就是字符串"retrofit"

      //这里必然是return 一个Call对象
      return loadMethodHandler(method).invoke(args);
   }
```

此时， invoke 的返回必然是一个 Call ， Call 是 Retrofit 中对一个 Request 的抽象,由此，大家应该不难想象到 loadMethodHandler(method).invoke(args); 这句代码应该就是去解析接口中传进来的注解，并生成一个 OkHttpClient 中对应的请求，这样我们调用 searchResultsCall 时，调用 OkHttpClient 走网络即可。确实， Retrofit 的主旋律的确就是这样滴。

> 注意，实际上 Call,CallBack 这种描述方式是在 OkHttp 中引入的， Retrofit 底层使用 OkHttp 所以也是使用这两个类名来抽象一个网络请求和一个请求回来之后的回调。总体来看，Retrofit中的 Call Callback 持有一个 OkHttp 的 Call Callback ，将对 Retrofit 中的各种调用转发到 OkHttp 的类库中，实际上这里就是静态代理啦，因为我们会定义各种代理类，比如 OkHttpCall

> 注 下文中如不无明确指出，则所有的 Call,CallBack 都是 Retrofit 中的类

MethodHandler类

MethodHandler 类，它是 Retrofit 中最重要的抽象了，每一个 MethodHandler 对应于本例的 RestApi 中的一个每个方法代表的请求以及和这个请求相关其它配置，我们来看看吧。

```
//这个OkHttp的工厂，用于产生一个OkHttp类库中的Call，实际上就是传入配置的Builder的OkHttpClient
private final okhttp3.Call.Factory callFactory;
//通过Method中的注解和传入的参数，组建一个OkHttp的Request
private final RequestFactory requestFactory;
//用于对Retrofit中的Call进行代理
private final CallAdapter<?> callAdapter;
//用于反序列化返回结果
private final Converter<ResponseBody, ?> responseConverter;

// 返回一个Call对象
Object invoke(Object... args) {
    return callAdapter.adapt(new OkHttpCall<>(callFactory, requestFactory, args, responseCor
}
```

在 Retrofit中 通过添加 Converter.Factory 来为 Retrofit 添加请求和响应的数据编码和解析。所以我们可以添加多个 Converter.Factory 为 Retrofit 提供处理不同数据的功能。

CallAdapter 可以对执行的Call进行代理，这里是静态代理。我们也可以通过添加自己的CallAdapter来作一些操作，比如为请求加上缓存：

```
  new CallAdapter.Factory {
      @Override
       public <R> Call<R> adapt(Call<R> call) {  return call;}
  }

class CacheCall implements Call  {
    Call delegate;
     CacheCall(Call call) {
         this.delegate = call;
    }

    @Override
    public void enqueue(Callback<T> callback) {
        //查看是否有缓存，否则直接走网络
        if(cached) {
            return cache;
        }
        this.delegate.enqueue(callback);
    }
}
```

至此，我们在调用 `resetApi.search("retrofit");` 时，实际上调用的层层代理之后的 `OkHttpCall` ,它是 `MethodHandler` 中invoke的时候塞入的。看看 `OkHttpCall` 中的代码吧：

```
@Override
public void enqueue(final Callback<T> callback) {
  okhttp3.Call rawCall;
  try {
    //创建一个okhttp的Call
    rawCall = createRawCall();
  } catch (Throwable t) {
    callback.onFailure(t);
    return;
  }
 //直接调用okhttp的入队操作
rawCall.enqueue(new okhttp3.Callback() {
  private void callFailure(Throwable e) {
    try {
      callback.onFailure(e);
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }

  private void callSuccess(Response<T> response) {
    try {
      callback.onResponse(response);
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }

  @Override
  public void onFailure(Request request, IOException e) {
    callFailure(e);
  }

  @Override
   public void onResponse(okhttp3.Response rawResponse) {
    Response<T> response;
    try {
      //解析结果
      response = parseResponse(rawResponse);
    } catch (Throwable e) {
      callFailure(e);
      return;
    }
    callSuccess(response);
  }
});
```

}

如此一来， `OkHttpClient` 的回调也被引导到我们的 `Callback` 上来，整个流程就已经走通了。

总结

终于到了总结的时候了，一般来说，这都是干货的时候，哈哈～

Volley对比优势

1. 缓存处理； Volley 自己就提供了一套完整的缓存处理方案，默认使用文件存储到磁盘中，并且提供了 TTL SOFTTTL 这么体贴入微的机制；一个 Request 可能存在两个 Response，对于需要显示缓存，再显示网络数据的场景真是爽的不要不要的。而 Retrofit 中并没有提供任何和缓存相关的方案。
2. 代码简单，可读性高。 Volley 的代码是写的如此的直接了当，让你看起代码来都不需要喝口茶，这样的好处是我们我们需要修改代码时不太容易引入bug....囧
3. 同一个请求如果同时都在发送，那么实际上只会有一个请求真正发出去， 其它的请求会等待这个结果回来，算小小优化吧。实际上这种场景不是很多哈，如果有，可能是你代码有问题...
4. 请求发送的时候提供了优先级的概念，但是是只保证顺序出去，不保证顺序回来，然并卵。
5. 支持不同的Http客户端实现，默认提供了 HttpClient 和 HttpUrlConnection 的实现，而 Retrofit 在2.0版本之后，锁死在 OkHttp 上了。

   6.支持请求取消

Retrofit

1.发送请求真简单，定义一个方法就可以了，这么简单的请求框架还有谁？ Volley？
2.较好的可扩展性， Volley 中每一个新建一个 Request 时都需要指定一个父类，告知序列化数据的方式，而 Retrofit 中只需要在配置时，指定各种转换器即可。 CallAdapter 的存在，可以使你随意代理调用的 Call ，不错不错。。。
3. OkHttpClient 自带并发光环，而 Volley 中的工作线程是自己维护的，那么就有可能存在线程由于异常退出之后，没有下一个工作线程补充的风险(线程池可以弥补这个缺陷)，这在 Retrofit 中不存在，因为即使有问题，这个锅也会甩给 OkHttp ，嘿嘿
4.支持请求取消

再次说明的是， Retrofit 没有对缓存提供任何额外支持，也就是说它只能通过 HTTP 的 Cache control 做文件存储，这样就会有一些问题：
1.需要 Server 通过 Cache control 头部来控制缓存，需要修改后台代码
2.有些地方比较适合使用数据库来存储，比如关系存储，此时， Retrofit 就无能为力了
3.缓存不在我们的控制范围之内，而是完全通过 OkHttp 来管理，多少有些不便，比如我们要删除某一个指定的缓存，或者更新某一个指定缓存，代码写起来很别扭(自己hack请求头里面的 cache contrl)

而在我们项目的实际使用过程中，缓存是一个比较重要的角色， Retrofit 对缓存的支持度不是很好，真是让人伤心。。。
但是，我们还是觉得在使用中 Retrofit 真心比较方便，容易上手，通过注解代码可读性和可维护性提升了N个档次，几乎没有样板代码(好吧，如果你觉得每个请求都需要定义一个方法，那这也算。。)，所以最后的决定是选择Retrofit。

有人说了， Volley 中的两次响应和缓存用起来很happy怎么办？嗯，我们会修改 Retrofit ，使它支持文件存储和 ORM 存储，并将 Volley 的缓存 网络两次响应回调移接过来，这个项目正在测试阶段，待我们项目做完小白鼠，上线稳定之后，我会把代码开源，大家敬请关注。

> 更新 2016.061.06
> Volley 官方文档上明确说鸟，只适用于轻量级请求，而不适用数据量较大的请求，比如下载一个50M的文件，原因是 Volley 默认在请求回来之后，就把数据从IO流里面搞到内存里面了，50M。。。
> Retrofit 是直接把流甩出来了，你上面怎么处理是你的事情，所以这里Retrofit还是更加灵活