在开发过程中，网络是我们很重要的一部分，因此我们就以网络框架或者说网络模块开始。在这个框架开发过程中，我会整理开发思路、以及遇到一些设计问题时会有怎么样的考虑、解决方案，当然这只是我个人的观点，大家也可以有自己的实现。除了网络框架，后续的系列还想更新ImageLoader框架、ORM框架，如果有时间也会增加动画框架和微博开发的系列文章。当然这些框架只是一些简单的框架基础，本人水平、时间有限，而且已经有现成、成熟的很多框架，我们在这里只是以重复造轮子的态度去学习轮子构建过程，从而达到能够造轮子的地步。至于很多细节的问题，我们这里就补过多讨论了，如果有兴趣，各位可以自行研究。

最后，我们暂且把这个框架命名为Simple\_Net\_Framework，下面我们一起进入主题吧。

## 基本结构

SimpleNet框架的基本结构类似于Volley,包括一些命名上也有跟Volley一致。它主要分为四个部分，最上面的部分为Request，即各种请求类型。例如返回的数据类型为json的对应为JsonRequest，返回数据字符串的为StringRequest，如果需要上传文件，那么你需要使用MultipartRequest，该请求只支持小文件的上传，如果上传的文件过大则会产生OOM。

第二部分为消息队列，消息队列维护了提交给网络框架的请求列表，并且根据相应的规则进行排序。默认情况下更具优先级和进入队列的顺序来执行，该队列使用的是线程安全的PriorityBlockingQueue<E>，因为我们的队列会被并发的访问，因此需要保证访问的原子性。

第三部分是Executor，也就是网络的执行者。该Executor继承自Thread，在run方法中循环访问第二部分的请求队列，请求完成之后将结果投递给UI线程。为了更好的控制请求队列，例如请求排序、取消等操作，这里我们并没有使用线程池来操作，而是自行管理队列和Thread的形式，这样整个结构也变得更为灵活。

第四部分则是Response投递类，在第三部分的Executor中执行网络请求，Executor是Thread，但是我们并不能在主线程中更新UI，因此我们使用

ResponseDelivery来封装Response的投递，保证Response执行在UI线程。

每个部分职责都相对单一，这样便于日后的升级和维护。

框架分析

图1中看起来有点像是分层架构，其实不是，这个图更多的是表达了它的逻辑顺序，而不是结构。而在我们的应用开发中，分层架构是一个重要的手段，如图2所示。

图2

但在开发过程中，我们往往会把UI和业务层耦合起来，因为它们的关系太密切了，分解起来并不是那么容易。高手能够把复杂的事情简单化，而分解就是简单化的重要手段，分解这个过程在开发过程中我们成为重构。但是如何分离UI和业务层也是本人最近想学习的，如果各位有好的解决方案，还希望多多指教。

那么我们就引入了一个分层概念，为了便于理解你也可以按照如图1的结构来加深理解。那么分层有什么优缺点呢？

优点：

1、复杂问题分解简单化，每一层负责自己的实现，并向外提供服务；

2、职责分离，复杂的系统都有很多人员进行开发，这些功能开发的管理和集成是个很严重的问题，分层设计实现之后，每层只需定义好自己的对外接口，其他依赖层服务的就可以进行开发；

3、每一层对其他层都是独立的，对外隐藏实现细节，上层无需知道下层的细节，只需调用接口即可；

4、有利于标准化。

缺点：

1、分层之后对于领域业务的修改有可能需要修改很多层；

2、过多的层次影响性能。

如上所说，我们的Simple\_Net\_Framework并不是分层的，而是简单的模块化，但是理论基础都是类似的，依赖于抽象而不依赖于实现、单一职责......这里引入分层的概念，这是便于理解，同时也是希望大家在开发过程中能够尽量保证模块的内聚性、耦合性。

再看Simple\_Net\_Framework，Request是一个抽象的泛型类，泛型类型就是返回的Response类型，例如StringRequest就是继承自Request<String>。第二部分的RequestQueue依赖于Request，Request是抽象的，因此任何Request的子类都可以传递到请求队列中来，它依赖的是抽象Request，而不是具体的某个实现，因此保证了可扩展性。你可以自己实现自己所需的Request，例如大文件的上传Request。同理，第三部分的NetworkExecutor也只是依赖于Request抽象，但这里又引入了一个类型HttpStack，这个网络请求的真正执行者，有HttpClientStack和HttpUrlConnStack，两者分别为Apache的HttpClient和java的HttpURLConnection，关于这两者的区别请参考:Android访问网络，使用HttpURLConnection还是HttpClient？(http://blog.csdn.net/guolin\_blog/article/details/12452307)。HttpStack也是一个抽象，具体使用HttpClient还是HttpURLConnection则由运行系统版本来定，HttpStackFactory会根据系统版本给框架返回对应的HttpStack。最后的ResponseDelivery比较简单了，只是通过Handler将结果投递给UI线程执行，也就是执行RequestListener的onComplete方法，此时网络执行完成，用户即可在该方法中更新UI或者相关的其他的操作。

下面我们再看看SimpleNet的工程结构，如图3所示。

这就是Simple\_Net\_Framework框架的基本结构了

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

在[教你写Android网络框架之基本架构](http://blog.csdn.net/bboyfeiyu/article/details/42740443" \t "http://blog.csdn.net/bboyfeiyu/article/details/_blank)一文中我们已经介绍了SimpleNet网络框架的基本结构，今天我们就开始从代码的角度来开始切入该网络框架的实现，在剖析的同时我们会分析设计思路，以及为什么要这样做，这样做的好处是什么。这样我们不仅学到了如何实现网络框架，也会学到设计一个通用的框架应该有哪些考虑，这就扩展到框架设计的范畴，通过这个简单的实例希望能给新人一些帮助。当然这只是一家之言，大家都可以有自己的实现思路。

正如你所看到的，这系列博客是为新人准备的，如果你是高手，请忽略。

在框架开发当中，很重要的一点就是抽象。也就是面向对象中重要的一条原则: 依赖倒置原则，简单来说就是要依赖抽象，而不依赖具体。这样就使得我们的框架具有可扩展性，同时也满足了开闭原则，即对扩展开放，对修改关闭。针对于我们的网络框架来说，最重要的抽象就是Reqeust类、Response类，因此今天我们就从两个类开始切入。最后我们再引入网络框架中的请求队列（RequestQueue），这是SimpleNet中的中枢神经，所有的请求都需要放到该队列，然后等待着被执行。请求队列就像工厂中的流水线一样，而网络请求就像流水线上的待加工的产品。执行网络请求的对象就类似工厂中的工人，在自己的岗位上等待流水线上传递过来的产品，然后对其加工，加工完就将产品放到其他的位置。它们角色对应关系参考图1，如对SimpleNet的一些角色不太清楚可参考[教你写Android网络框架之基本架构](http://blog.csdn.net/bboyfeiyu/article/details/42740443" \t "http://blog.csdn.net/bboyfeiyu/article/details/_blank)一文。

## Request类

既然网络框架，那么我们先从网络请求类开始。前文已经说过，既然是框架，那么就需要可扩展性。因此注定了Request是抽象，而不是具体。而对于网络请求来说，用户得到的请求结果格式是不确定，比如有的服务器返回的是json,有的返回的是xml，有的直接是字符串。但是对于Http Response来说，它的返回数据类型都是Stream，也就是我们得到的原始数据都是二进制的流。所以在Request基类中我们必须预留方法来解析Response返回的具体类型，虽然返回的类型不同，但是他们的处理逻辑是一样的，因此我们可把Request作为泛型类，它的泛型类型就是它的返回数据类型，比如Request<String>，那么它的返回数据类型就是String类型的。另外还有请求的优先级、可取消等，我们这里先给出核心代码，然后再继续分析。



上述代码Request<T>为抽象类，T则为该请求Response的数据格式。这个T是请求类中的一个比较重要的点，不同的人有不同的需求，即请求Reponse的数据格式并不是都是一样的，我们必须考虑到请求返回类型的多样性，用泛型T来表示返回的数据格式类型，然后Request子类覆写对应的方法实现解析Response的数据格式，最后调用请求Listener将请求结果执行在UI线程，这样整个请求就完成了。

每个Request都有一个序列号，该序列号由请求队列生成，标识该请求在队列中的序号，该序号和请求优先级决定了该请求在队列中的排序，即它在请求队列的执行顺序。每个请求有请求方式，例如"POST"、"GET"，这里我们用枚举来代替，具名类型比单纯的字符串更易于使用。每个Request都可以添加Header、Body参数 （ 关于请求参数的格式可以参考 四种常见的 POST 提交数据方式），并且可以取消。抽象类封装了通用的代码，只有可变的部分是抽象函数，这里只有parseResponse这个函数。

例如，我们返回的数据格式是Json，那么我们构建一个子类叫做JsonRequest，示例代码如下。



可以看到，实现一个请求类还是非常简单的，只需要覆写parseResponse函数来解析你的请求返回的数据即可。这样就保证了可扩展性，比如后面如果我想使用这个框架来做一个ImageLoader,那么我可以创建一个ImageRequest,该请求返回的类型就是Bitmap，那么我们只需要覆写parseResponse函数，然后把结果转换成Bitmap即可。

这里引入了Response类，这个Response类存储了请求的状态码、请求结果等内容，我们继续往下看。

Response类

每个请求都对应一个Response，但这里的问题是这个Response的数据格式我们是不知道的。我们写的是框架，不是应用。框架只是构建一个基本环境，并且附带一些比较常用的类，比如这里的JsonRequest。但是重要的一点是可以让用户自由、简单的扩展以实现他的需求。对于Response类来说，我们最重要的一点就是要确定请求结果的数据格式类型。我们都知道，HTTP实际上是基于TCP协议，而TCP协议又是基于Socket，Socket实际上操作的也就是输入、输出流，输出流是向服务器写数据，输入流自然是从服务器读取数据。因此我们在Response类中应该使用InputStream存储结果或者使用更为易于使用的字节数组，这里我们使用字节数组来存储。我们来看Response类。



这个类很简单，只是继承了BasicHttpResponse，然后将输入流转换成字节数组，然后包装了几个常用的方法，主要是为了使用简单吧。我们将结果存储为字节数组，这样可以用户可以很方便的将结果转换为String、bitmap等数据类型，如果直接存储的是InputStream，那么在很多时候用户需要在外围将InputStream先转换为字节数组，然后再转换为最终的格式，例如InputStream转为String类型。这也是为什么我们这里选用byte[]而不用InputStream的原因。

请求队列

网络请求队列也比较简单，实际上就是内部封装了一个优先级队列，在构建队列时会启动几个NetworkExecutor ( 子线程 )来从请求队列中获取请求，并且执行请求。请求队列会根据请求的优先级进行排序，这样就保证了一些优先级高的请求得到尽快的处理，这也就是为什么Request类中实现了Comparable接口的原因。如果优先级一致的情况下，则会根据请求加入到队列的顺序来排序，这个序号由请求队列生成，这样就保证了优先级一样的情况下按照FIFO的策略执行。



这里引入了一个HttpStack，这是一个接口，只有一个函数。该接口定义了执行网络请求的抽象，代码如下：



>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

今天我们就来剖析另外几个特别重要的角色，即NetworkExecutor、HttpStack以及ResponseDelivery，它们分别对应的功能是网络请求线程、Http执行器、Response分发，这三者是执行http请求和处理Response的核心。

我们再来回顾一下，SimpleNet各个角色的分工合作。首先用户需要创建一个请求队列，然后将各个请求添加到请求队列中。多个NetworkExecutor （ 实质上是一个线程 ）共享一个消息队列，在各个NetworkExecutor中循环的取请求队列中的请求，拿到一个请求，然后通过HttpStack来执行Http请求，请求完成后最终通过ResponseDelivery将Response结果分发到UI线程，保证请求回调执行在UI线程，这样用户就可以直接在回调中更新UI。执行流程如图1.

NetworkExecutor

作为SimpleNet中的“心脏”，NetworkExecutor起着非常重要的作用。之所以称之为“心脏”，是由于NetworkExecutor的功能是源源不断地从请求队列中获取请求，然后交给HttpStack来执行。它就像汽车中的发动机，人体中的心脏一样，带动着整个框架的运行。

NetworkExecutor实质上是一个Thread，在run方法中我们会执行一个循环，不断地从请求队列中取得请求，然后交给HttpStack，由于比较简单我们直接上代码吧。



在启动请求队列时，我们会启动指定数量的NetworkExecutor （ 参考 教你写Android网络框架之Request、Response类与请求队列）。在构造NetworkExecutor时会将请求队列以及HttpStack注入进来，这样NetworkExecutor就具有了两大元素，即请求队列和HttpStack。然后在run函数的循环中不断地取出请求，并且交给HttpStack

HttpStack

HttpStack只是一个接口，只有一个performRequest函数，也就是执行请求。

HttpStack是网络请求的真正执行者，有HttpClientStack和HttpUrlConnStack，两者分别为Apache的HttpClient和java的HttpURLConnection，关于这两者的区别请参考:Android访问网络，使用HttpURLConnection还是HttpClient？ 默认情况下，我们会根据api版本来构建对应的HttpStack，当然用户也可以自己实现一个HttpStack，然后通过SimpleNet的工厂函数传递进来。



在购置请求队列时会传递HttpStack，如果httpStack为空,则由HttpStackFactory根据api版本生成对应的HttpStack。即api 9以下是HttpClientStack, api 9 及其以上则为HttpUrlConnStack。



HttpClientStack和HttpUrlConnStack分别就是封装了HttpClient和HttpURLConnection的http请求，构建请求、设置header、设置请求参数、解析Response等操作。针对于这一层，我们没有给出一个抽象类，原因是HttpClient和HttpURLConnection并不属于同一个类族，他们的行为虽然都很相似，但是其中涉及到的一些类型却是不同的。这里我们给出HttpUrlConnStack的示例，最近比较忙，因此写的配置比较简单，有需要的同学自己优化了。

## ResponseDelivery

在HttpStack的performRequest函数中，我们会返回一个Response对象，该对象包含了我们请求对应的Response。关于Response类你不太了解的可以参考[教你写Android网络框架之Request、Response类与请求队列](http://blog.csdn.net/bboyfeiyu/article/details/43015859" \t "http://blog.csdn.net/bboyfeiyu/article/details/_blank)。我们在NetworkExecutor中执行http请求的最后一步会将结果分发给UI线程，主要工作其实就是将请求的回调执行到UI线程，以便用户可以更新UI等操作。



不管是从缓存中获取还是从网络上获取，我们得到的都是一个Response对象，最后我们通过ResponseDelivery对象将结果分发给UI线程。

ResponseDelivery其实就是封装了关联了UI线程消息队列的Handler，在deliveryResponse函数中将request的deliveryResponse执行在UI线程中。既然我们有了关联了UI线程的Handler对象，那么直接构建一个Runnable，在该Runnable中执行request的deliveryResponse函数即可。在Request类的deliveryResponse中，又会调用parseResponse解析Response结果，返回的结果类型就是Request<T>中的T，这个T是在Request子类中指定，例如JsonRequest，那么返回的Response的结果就是JSONObject。这样我们就得到了服务器返回的json数据，并且将这个json结果通过回调的形式传递给了UI线程。用户就可以在该回调中更新UI了。

这其中主要就是抽象和泛型，写框架很多时候泛型是很重要的手段，因此熟悉使用抽象和泛型是面向对象开发的重要一步。

这样，整个请求过程就完成了。下面我们总结一下这个过程。

不同用户的服务器返回的数据格式是不一致的，因此我们定义了Request<T>泛型基类，泛型T就是返回的数据格式类型。比如返回的数据格式为json，那对应的请求就是JsonRequest,泛型T为JSONObject，在JsonRequest中覆写parseResponse函数，将得到的Response中的原始数据转换成JSONObject。然后将请求放到队列中，NetworkExecutor将请求分发给HttpStack执行，执行完成之后得到Response对象，最终ResponseDelivery将结果通过请求回调投递到UI线程。

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

我们从基本结构到代码实现，剖析了一个简单的网络框架应该是怎样运作的，以及在面对各式各样的需求时应该如何对代码做出处理，在深入了解网络框架的同时学习到一些简单的面向对象设计原则。正如第一篇博文所说，SimpleNet框架参照的是Volley实现，甚至有一些类名也是一样的。我们的目标并不是要重新发明轮子，而是以学习轮子制作的过程来达到提升自我的目的。SimpleNet只是一个简单的网络框架实现，没有经过严格的测试以及市场检验，不建议大家在项目中使用，当然如果你觉得没有什么问题，在经过测试的情况下也可以运用在自己的项目中。

请求配置与https

在执行http请求时，我们经常需要对http请求进行配置，例如超时配置和https配置等。SimpleNet在这里只做出了简单的配置，如有更多的需求则请自行实现。由于HttpClient和HttpURLConnection所属的类族是不一样的，他们对于Https的配置并没有一个公共的类型，因此这里没有进行抽象，而是针对两个HttpClient和HttpURLConnection创建来两个配置类，其中HttpClientConfig是HttpClientStack的配置类，而HttpUrlConnConfig则是HttpUrlConnStack的配置类。

例如配置https时，httpClient的SSLSocketFactory所在的包为org.apache.http.conn.ssl.SSLSocketFactory；而HttpURLConnection的SSLSocketFactory所在的包却是javax.net.ssl.SSLSocketFactory。这是apache和Android团队的不同实现，因此不好做出抽象层，我们这里使用两个配置类来进行配置。

使用HttpClient时配置https请参考httpclient中使用HTTPS的方法，使用HttpUrlConnStack执行https请求时配置https请参考Android网络编程——https 不验证证书方式（信任所有证书）。

例如，在低于api 9时，用户可以通过HttpClientConfig来配置SSLSocketFactory，然后在执行请求时会获取配置类的SSLSocketFactory来设置HttpClient。



在HttpClientStack中执行请求时，会判断是否是https请求，如果是则需要获取配置类中配置的SSLSocketFactory，代码如下 :

[java] view plaincopy在CODE上查看代码片派生到我的代码片

01./\*\*

02. \* 如果是https请求,则使用用户配置的SSLSocketFactory进行配置.

03. \*

04. \* @param request

05. \*/

06.private void configHttps(Request<?> request) {

07. SSLSocketFactory sslSocketFactory = mConfig.getSocketFactory();

08. if (request.isHttps() && sslSocketFactory != null) {

09. Scheme sch = new Scheme("https", sslSocketFactory, 443);

10. mHttpClient.getConnectionManager().getSchemeRegistry().register(sch);

11. }

12.}

HttpUrlConnStack的https设置也是类似的，就不给出了。由于没有服务器，对于Https的配置本人并没有经过测试，如有问题请自行调试了。

Response缓存

在某些情况下，数据并不会每次都需要从服务端获取，因此我们添加了Response缓存。这样就可以避免不必要的请求浪费流量，也可以提升用户体验。用户可以通过Request的setShouldCache(boolean shouldCache)方法来设置是否缓存该请求的Response，如果是true那么则缓存，否则不缓存。

在执行请求时，会判断是否缓存该请求的Response,如果是，那么会将该Response缓存到内存中。如果该请求开启了缓存，那么在请求前会判断是否含有缓存，如果有缓存则直接取缓存结果，没有缓存才从服务端获取。如下是NetworkExecutor中执行网络请求的代码

Response缓存

针对于缓存，我们添加了一个简单的缓存接口。该接口是一个泛型接口，key和value的类型都是泛型。设计为泛型是因为我们在后续的框架中还会使用，后续的ImageLoader框架将以SimpleNet框架为基础来构建一个图片加载框架，其中也会用到缓存接口，但是它的类型却是不一样的，因此我们使用泛型来保证它的扩扩展性。



针对于网络请求的缓存，我们的实现是LruMemCache类，该类将Response请求结果按照LRU的规则进行缓存。代码如下 :



缓存类对象是各个NetworkExecutor共享的，它定义在NetworkExecutor中

这样，多个NetworkExecutor就可以拥有同一份缓存了。

结束语

Android网络框架到此就已经结束了，由于公司项目原因，很多东西写得比较随意，但不管写得好与烂，总归是一份善意的分享。正如上文所说，我们的目的并不是重新发明轮子，而是在学习发明轮子的过程中提升自己的技术能力、设计能力、抽象能力。或者说我们在学习实现这个网络框架的过程中能够让自己有所领悟，能够对“面向接口编程，而不是面向实现编程”这句话的含义有所理解。正如郭林所说的，写代码到了一定阶段，更重要的是要读代码。从优秀的代码中学习优秀的设计与实现，这是从码农到工程师的一个必经阶段。

当你能够体会到软件之美时，说明你已经到了一个新的阶段。你能够体会到设计之美、架构之美，在你的眼中优秀的设计是美的，而不只是一堆乱七八糟却能实现功能的代码，你能够在其中得到满足。每个人都是从菜鸟慢慢进步而来的，什么事都不是一蹴而就。本人写这一系列的博文，只是希望在自我提升的同时也能够帮助到一些需要帮助的同行，虽然本人水平很有限，但是能将自己的体会分享给别人，甚至说是帮到他人，那就是这个系列的初衷了。