开源库

- Retrofit
- RabbitMQ
- OkHttp
- RxJava
- Glide
- EventBus

Retrofit

Retrofit 知识

	Retrofit 简介
介绍	一个 RESTful 的 HTTP 网络请求框架（基于 OKHTTP）
作者	Square
功能	1.基于 okhttp & 遵循 Restful API 设计风格； 2.通过注解配置网络请求参数； 3.支持同步 & 异步网络请求； 4.支持多种数据的解析 & 序列化格式（Gson、Json、XML、Protobuf）； 5.提供对 RxJava 支持。
优点	1.功能强大：支持同步 & 异步、支持多种数据的解析 & 序列化格式、支持 RxJava； 2.简洁易用：通过注解配置网络请求参数、采用大量设计模式简化使用； 3.可扩展性好：功能模块高度封装、解耦彻底，如自定义 Converters （自定义转换器）等等。
应用场景	任何网络请求的需求场景都应优先选择（特别是后台 API 遵循 Resful API 设计风格 & 项目中使用到 RxJava）。

特别注意：

- 准确来说，Retrofit 是一个 RESTful 的 HTTP 网络请求框架的封装。
- 原因：网络请求的工作本质上是 OkHttp 完成，而 Retrofit 仅负责网络请求接口的封装。

- App 应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数、Header、Url 等信息，之后由 OkHttp 完成后续的请求操作。
- 在服务端返回数据之后，OkHttp 将原始的结构交给 Retrofit，Retrofit 根据用户的需求对结果进行解析。

与其他开源请求库对比

除了 Retrofit，如今 Android 中主流的网络请求框架有：

- Android-Asynv-Http
- Volley
- OkHttp

网络请求开源库对比：

网络请求库/对比	android-async-http	Volley	OkHttp	Retrofit
作者	Loopj	Google	Square	Square
面世时间	1（最早）	2	3	4（最晚）
人们使用情况（GitHub start书）	2	1（最多）	3	4（最少）
功能	1.基于 HttpClient； 2.在 UI 线程外、异步处理 Http 请求； 3.在匿名回调中处理请求结果，callback 使用了 Android 的 Handler 发送消息机制在创建它的线程中执行； 4.自动智能请求重试； 5.持久化 cookie 存储，保存 cookie 到你的应用程序的 SharedPreferences。	1.基于 HttpUrlConnection； 2.封装了 URL 图片加载框架，支持图片加载； 3.网络请求的排序、优先级处理； 4.缓存； 5.多级别取消请求； 6.Activity 和生命周期的联动（Activity 结束时同时取消所有网络请求）。	1.高性能 Http 请求库，可把它理解成一个封装之后的类似 HttpUrlConnection 的一个东西，属于同级并不属于上述两种； 2.支持 SPDY，共享同一个 Socket 来处理同一个服务器的所有请求； 3.支持 http 2.0、websocket； 4.支持同步、异步； 5.封装了线程池、数据转换、参数使用、错误处理等； 6.无缝的支持 GZIP 来减少数据流量； 7.缓存响应数据来减少重复的网络请求； 8.能从很多常用的连接问题中自动恢复； 9.解决了代理服务器问题和 SSL 握手失败问题。	1.基于 OkHttp； 2.RESTful API 设计风格； 3.支持同步、异步； 4.通过注解配置请求，包括请求方法、请求参数、请求头、返回值等； 5.可以搭配多种 Converter （转换器）将获得的数据解析 & 序列化，支持 Gson(默认)、Jackson、Protobuf 等； 6.提供对 RxJava 的支持。
性能	1.作者已经停止对该项目维护； 2.Android 5.0 后不推荐使用 HttpClient；所以不推荐在项目中使用。	1.可扩展性好：可支持 HttpClient、HttpUrlConnection 和 OkHttp。	1.基于 NIO 和 Okio，所以性能更好：请求、处理速度快（IO：阻塞式；NIO：非阻塞式；Okio 是 Square 公司基于 IO 和 NIO 基础上做的一个更简单、高效处理数据流的一个库）。	1.性能最好，处理最快； 2.扩展性差，高度封装所带来的必然后果；解析数据都是使用的统一的 converter（转换器），如果服务器不能给出统一的 API 的形式，将很难进行处理。
开发者使用	1.作者已经停止对该项目维护； 2.Android 5.0 后不推荐使用 HttpClient；所以不推荐在项目中使用。	1.封装性好：简单易用。	1. API 调用更加简单、方便； 2.使用时需要进行多一层封装。	1.简洁易用（RestfulAPI 设计分割）； 2.代码简洁（更加高度的封装性和注解用法）； 3.解耦的更彻底、职责更细分； 4.易与其他框架联合使用（RxJava）； 5.使用方法较多，原理复杂，存在一定门槛。

应用场景	1.作者已经停止对该项目维护； 2.Android 5.0 后不推荐使用 HttpClient；所以不推荐在项目中使用。	1.适合轻量级网络交互：网络请求频繁、传输数据量小； 2.不能进行大数据量的网络操作（比如下载视频、音频），所以不适合用来上传文件。	1.重量级网络交互场景：网络请求频繁、传输数据量大（其实会更推荐 Retrofit，反正 Retrofit 是基于 Okhttp 的）。	1.任何场景下优先选择，特别是：后台 Api 遵循 RESTful 的风格 & 项目中使用 RxJava。
备注		Volley 的 request 和 response 都是把数据放到 byte 数组里，不支持输入输出流，把数据放到数组中，如果大文件多了，数组就会非常的大且多，消耗内存，所以不如直接返回 Stream 那样具备可操作性，比如下载一个大文件，不可能把整个文件都缓存内存之后再写到文件里。	Android 4.4 的源码中可以看到 HttpURLConnection 已经替换成 OkHttp 实现了，所以有理由相信 OkHttp 的强大。	

Retrofit 源码分析

Retrofit 的主要原理是利用了 Java 的动态代理技术，把 ApiService 的方法调用集中到了 InvocationHandler.invoke，再构建了 ServiceMethod 、OkHttpClient，返回 callAdapter.adapter 的结果。

Retrofit 的最大特点就是解耦。

基本使用

- 创建 Retrofit 对象

```
//构建 OkHttpClient 对象
OkHttpClient okHttpClient = new OkHttpClient.Builder()
                .connectTimeout(10, TimeUnit.SECONDS) // 连接超时
                .readTimeout(10, TimeUnit.SECONDS) // 读取超时
                .retryOnConnectionFailure(true) // 是否重试
                .writeTimeout(10, TimeUnit.SECONDS) // 写入超时
                .build();
// 创建 Retrofit 对象，外观模式
Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("http://test")
        .addConverterFactory(GsonConverterFactory.create(new
GsonBuilder().create())) // 解析使用 GsonConverterFactory
                .addCallAdapterFactory(RxJavaCallAdapterFactory.create()) // 返
回使用 RxJavaCallAdapterFactory
                .client(okHttpClient) // 请求使用 OkHttpClient
                .build();
```

- 定义 API

```
public interface ApiService {
    @GET("data/")
    Observable<BaseResponse> getMessage(@Path("page") int page);

    public static class BaseResponse {

        /**
```

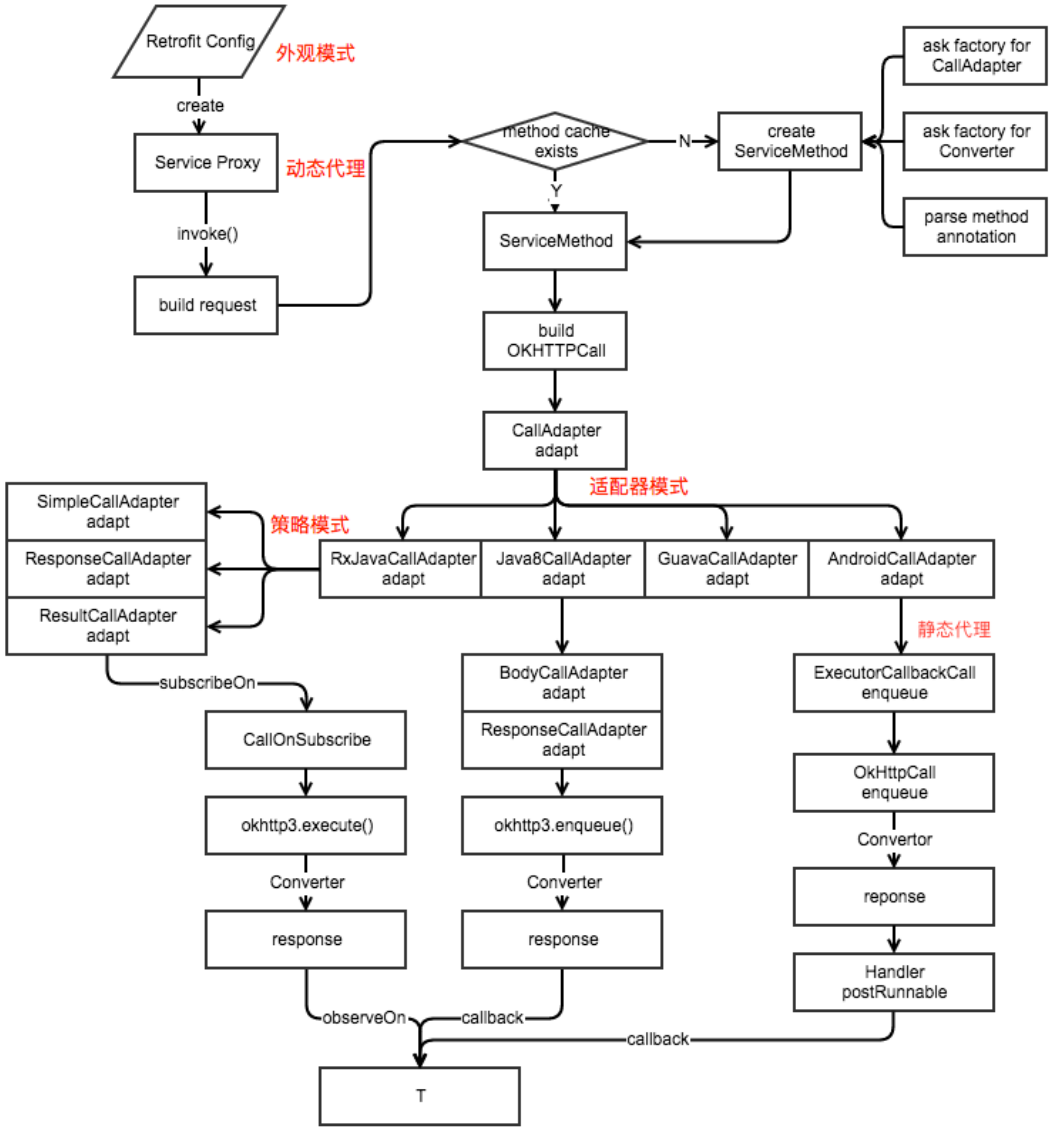
```
         * 业务错误码
         */
        @SerializedName("F_responseNo")
        public int responseNo;

        /**
         * 业务错误描述
         */
        @SerializedName("F_responseMsg")
        public String responseMsg;

        @Override
        public String toString() {
            return "BaseResponse{" +
                    "responseNo=" + responseNo +
                    ", responseMsg='" + responseMsg + '\'' +
                    '}';
        }
    }
}
```

- 获取 API 实例

```
// 获取 API 实例
ApiService service = retrofit.create(ApiService.class);
// 调用 getMessage 的方法请求数据
Observable<ApiService.BaseResponse> observable = service.getMessage(1);
...
```

Retrofit 就这样经过简单的配置后就可以向服务器请求数据了，超级简单。

流程图



源码分析

Retrofit.create 方法（创建 API 实例）分析

Retrofit 的 create 方法作为 Retrofit 的入口。

```
  public <T> T create(final Class<T> service) {
    // 验证接口是否合理
    Utils.validateServiceInterface(service);
    // 默认 false
    if (validateEagerly) {
      eagerlyValidateMethods(service);
    }
    // 动态代理
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[]
{ service },
        new InvocationHandler() {
          // 平台的抽象，指定默认的 CallbackExecutor CallAdapterFactory 用，这里
Android 平台是 Android（还有 Java8 和 ios）
```

```
          private final Platform platform = Platform.get();
          // ApiService 中的方法调用会走到这里
          @Override public Object invoke(Object proxy, Method method, @Nullable
Object[] args)
              throws Throwable {
            // If the method is a method from Object then defer to normal
invocation.
            // Object 的方法不管
            if (method.getDeclaringClass() == Object.class) {
              return method.invoke(this, args);
            }
            // java8 的默认方法，Android 暂不支持默认方法，所以暂时也需要管
            if (platform.isDefaultMethod(method)) {
              return platform.invokeDefaultMethod(method, service, proxy,
args);
            }
            // 重点
            // 为 Method 生成一个 ServiceMethod
            ServiceMethod<Object, Object> serviceMethod =
                (ServiceMethod<Object, Object>) loadServiceMethod(method); // 4
            // 再包装成 OkHttpCall
            OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod,
args); // 请求 5
            return serviceMethod.callAdapter.adapt(okHttpCall);
          }
        });
  }
```

从 create 方法中可以看出，Retrofit 的主要原理是利用了 Java 的**动态代理**技术创建了 API 实例，把 ApiService 的 方法调用集中到了 InvocationHandler.invoke，再构建了 ServiceMethod、OkHttpCall，返回 callAdapter.adapt() 的结果。

也就是当调用前面写的 ApiService Interface 中的请求方法，会被 proxy 拦截，调用 InvocationHandler.invoke 的方法：

```
    @GET("data/")
    Observable<BaseResponse> getMessage(@Path("page") int page);
```

动态代理技术就是动态生成接口的实例类（当然生成实现类有缓存机制），并创建其实例（称之为代理），代理把对接口的调用转发给 InvocationHandler 实例，而在 InvocationHandler 的实现中，处理执行真正的逻辑（例如再次转发给真正的实现类对象），还可以进行一些有用的操作，例如统计执行时间，进行初始化和清理、对接口调用进行检查等。

在 invoke 方法处理方法调用的过程中，如果调用的是 Object 的方法，例如 equals、toString，那就直接调用。如果是 default ，就调用 default 方法。而真正重要的代码只有三行：

```
ServiceMethod<Object, Object> serviceMethod =
                (ServiceMethod<Object, Object>) loadServiceMethod(method);
// 再包装成 OkHttpCall
OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod, args); // 请求
return serviceMethod.callAdapter.adapt(okHttpCall);
```

ServiceMethod 是接口方法的抽象，主要负责解析它对应的 method 的各种参数（它有各种如 parseHeaders 的方法），比如注解（@GET）、入参，另外还负责获取 callAdapter、responseConverter 等 Retrofit 配置，好为后面的 okhttp3/Request 做好参数准备，它的 toRequest 为 OkHttp 提供 Request，toResponse 将请求结果转换为想要的数据类，可以说它承载了后续 Http 请求所需的一切参数。总的来说就是 ServiceMethod 类的作用就是把对接口方法的调用转为一次 HTTP 调用。

一个 ServiceMethod 对象对应于一个 API interface 的一个方法。

总结

1. Retrofit 是使用动态代理 Proxy 对定义的接口进行处理的，当调用接口的方法时，会在动态代理的 InvocationHandler # invoke 方法对请求进行处理。
2. ServiceMethod 会解析接口的方法，将方法的注解解析为请求的 Request，根据用户设置配置生成具体的 CallAdapter、ResponseConverter，将请求的结果使用 ResponseConverter 转为合适的 R 对象。 ServiceMethod 类的作用就是把对接口方法的调用转为一次 HTTP 调用，而且 ServiceMethod 有缓存，减少了一定的消耗。
3. 实际是调用了 call 的 exercute()（同步） 或者 enqueue()（异步） 来完成请求。OkHttpCall 算是 OkHttp 的包装类，用它跟 OkHttp 对接。会在 OkHttpCall 中将 OkHttp 的 response 包装成 retrofit 标准下的 response，再使用 RespouseConverter 转成想要的 R 对象。默认是 OkHttpClient，当然还可以扩展一个新的 Call ，比如 HttpUrlConnectionCall。
4. Retrofit 提供了很多的 ConverterFactory，比如 Gson、Jackson、xml、protobuff 等等，需要什么，就配置相对应的工厂，在 Service 方法上声明泛型具体类型就可以了。
5. 生成的 CallAdapter 有四个工厂，分别对应不同的平台：RxJava、Java8、Guava 还有一个 Retrofit 默认的。简单来说就是用来将 Call 转成 T 的一个策略。因为这里具体请求时耗时操作，所以需要 CallAdapter 去管理线程。比如 RxJava 会根据调用方法的返回值，如 Response < T > | Result < T > | Observable < T >，生成不同的 CallAdapter。实际上就是对 RxJava 的回调方式做封装。比如将 response 再拆解为 success 和 error 等。

RabbitMQ

* RabbitMQ 知识

什么是 MQ

MQ 全称为 Message Queue，即消息队列。本质是个队列，FIFO 先进先出，只不过队列中存放的内容是 message 而已。

其主要用途：不同进程 process / 线程 thread 之间通信。

为什么会产生消息队列？原因：

1. 不同进程（ process ）之间传递消息时，两个进程之间耦和程度过高，改动一个进程，引发必须修改另一个进程，为了隔离这两个进程，在两进程间抽离出一层（一个模块），所有两进程之间传递

的消息，都必须通过消息队列来传递，单独修改一个进程，不会影响另一个。

2. 不同进程（process）之间传递消息时，为了实现标准化，将消息的格式规范化了，并且某一个进程接收的消息太多，一下子无法处理完，并且也有先后顺序，必须对收到的消息进行排队，因此诞生了事实上的消息队列。

MQ 框架非常多，比较流行的有 RabbitMQ、ActiveMq、ZeroMq、Kafka，以及阿里开源的 RocketMQ。

消息队列（MQ）是一种应用程序对应用程序的通信方法，也就是信息中间件。应用程序通过读写出入队列的消息（针对应用程序的数据）来通信，而无需专用连接来链接它们。消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常适用于诸如远程过程调用的技术。排队指的是应用程序通过队列来通信。队列的使用除去了接收和发送应用程序同时执行的要求。

消息发出后可以立即返回，由消息系统来确保消息的可靠传递。消息发布者只管把消息发布到 MQ 中而不用管谁来取，消息使用者只管从 MQ 中取消息而不管消息是谁发布的。这样发布者和使用者都不用知道对方的存在。

MQ 的模型：所有 MQ 产品从模型抽象上来说都是一样的过程，消费者（consumer）订阅某个队列。生产者 (producer) 创建消息，然后发布到队列（queue）中，最后将消息发送到监听的消费者。

MQ 常用于业务解耦的情况，其他常见使用场景包括最终一致性、广播、错峰流控等等。

RabbitMQ

RabbitMQ 则是 MQ 的一种开源实现，遵循 AMQP（Advanced Message Queue，高级消息队列协议）协议，特点是消息转发是非同步并且可靠的。

RabbitMQ 作为一个消息代理，主要用来处理应用程序之间消息的存储与转发，可让消费者和生产者解耦，消息是基于二进制的。它提供了可靠的消息机制和灵活的消息路由，并支持消息集群和分布式部署，常用于应用解耦、耗时任务队列、流量削锋等场景。在易用性、扩展性、高可用性等方面表现不俗。

大数据处理场景需要 kafka，如果需要较高性能和确认机制，数据的可靠性和活跃的社区，支持消息的持久化于中间件的高可用部署，就选择 RabbitMQ 来作为应用的中间件。

AMQP

AMQP 即 Advanced Message Queuing Protocol，一个提供统一消息服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。

消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

AMQP 的主要特征是面向消息、队列、路由（包括点对点和发布 / 订阅）、可靠性、安全。

基于此协议的客户端与消息中间件可传递消息，并不受客户端 / 中间件不同产品、不同开发语言等条件的限制。

Erlang

RabbitMQ 是使用 Erlang 语言开发的。

Erlang 是一种通用的面向并发的编程语言，是一个结构化、动态类型编程语言，内建并行计算支持。

最初是由爱立信专门为通信应用设计的，比如控制交换机或者变换协议等，因此非常适合于构建分布式，实时软并行计算系统。

使用 Erlang 编写出的应用运行时通常由成千上万个轻量级进程组成，并通过消息传递相互通讯。

进程间上下文切换对于 Erlang 来说仅仅只是一两个环节，比起 C 程序的线程切换要高效的多得多了。

功能

1. 应用解耦

   mq 基于数据的接口层，将耦合的引用来分解开，两边都实现这个接口，这样就允许独立的修改或者扩展两边的处理过程，只要两边遵守相同的接口约束即可。

2. 流量削峰

   在高并发、大流向的场景下，RabbitMQ 可以减少突发访问压力，不会因为突发的超时负荷要求而崩溃。

3. 异步通信

   通过把消息发送给消息中间件，将不是实时的业务异步处理。

特点

1. 可靠性（Reliability）

   RabbitMQ 使用一些机制来保证可靠性，如持久性、传输确认及发布确认等。

2. 灵活的路由（Flexible Routing）

   在消息进入队列之前，通过交换器来路由消息。对于典型的路由功能，RabbitMQ 提供了一些内置的交换器来实现。针对更复杂的路由功能，可以将多个交换器绑定在一起，也可以通过插件机制来实现自己的交换器。

3. 扩展性、消息集群（Clustering）

   多个 RabbitMQ 节点可以组成一个集群，形成一个逻辑 Broker。也可以根据实际业务情况动态地扩展集群中节点。

4. 高可用性（Highly Avaliable Queues）

   队列可以在集群中的机器上设置镜像，使得在部分节点出现问题的情况下队列依然可用。

5. 多种协议（Multi-protocol）

   RabbitMQ 除了原生支持 AMQP 协议，还支持 STOMP、MQTT 等多种消息中间件协议。

6. 多语言客户端（Many Clients）

   RabbitMQ 几乎支持所有常用语言，比如 Java、Python、Ruby、PHP、C#、JavaScript、.NET 等。

7. 管理界面（Management UI）

   RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息、集群中的节点等。

8. 跟踪机制（Tracing）

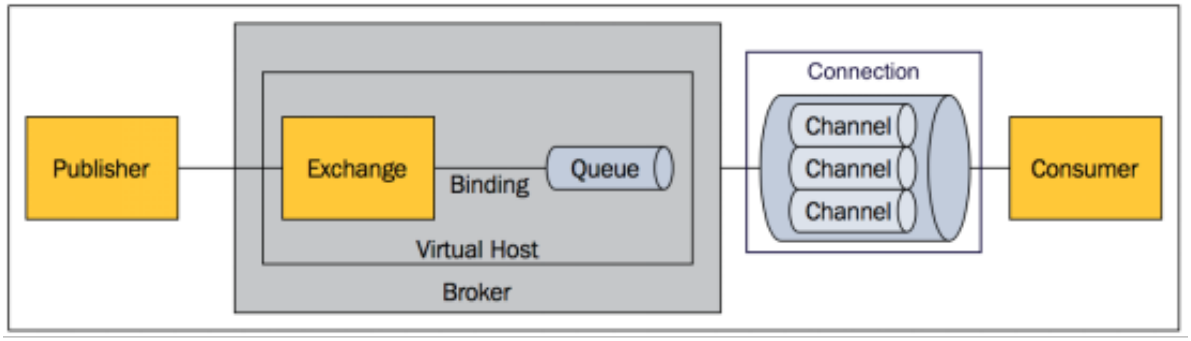
   如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。

9. 插件机制（Plugin System）

RabbitMQ 提供了许多插件，以实现从多方面进行扩展，当然也可以编写自己的插件。

基本定义

RabbitMQ 比 MQ 模型有更加详细的模型概念：



Broker 服务

RabbitMQ 服务器，提供一种传输服务，接收客户端连接，实现 AMQP 消息队列和路由功能的进程。

它的角色就是维护一条从 Publisher 到 Consumer 的路线，保证数据能够按照指定的方式进行传输。

用户与权限设置就是依附于 Broker。

Publisher 生产者

数据的发送方，消息生产者，就是一个向交换器发布消息的客户端应用程序。

Message 消息

消息，消息是不具名的，由 Header 和 Body 组成，Header 是由生产者添加的各种属性的集合，包括 Message 是否被持久化（delivery-mode）、由哪个 Message Queue 接收（routing-key）、优先级是多少（priority）等，Body 是真正传输的数据，是不透明的，内容格式为 byte[] 。

Consumer 消费者

数据的接收方。消息的消费者，就是接收消息的程序。

当有 Message 到达时，RabbitMQ 把它发送给它的某个订阅者即 Consumer。当然可能会把同一个 Message 发送给很多的 Consumer。在这个 Message 中，只有 body，header 已经被删除了。对于 Consumer 来说，它是不知道谁发送的这个消息的，就是协议本身不支持。但是如果 Publish 发送的 bidy 中包含了 Publish 的信息就另当别论了。

一个队列可以绑定多个消费者，但是只有其中的一个消费者会消费消息。

生产者 Producer 和消费者 Consumer 都是 RabbitMQ 的客户端，Producer 负责发送消息，Consumer 负责消费消息。

Virtual Host 虚拟主机

虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。一个 broker 里可以有多个 Virtual Host ，用作不同用户的权限分离。

一个 Virtual Host 里面可以有若干个 Exchange 和 Queue，主要用于权限控制，隔离应用。

每个 virtual host 本质上都是一个 RabbitMQ Server，拥有它自己的 queue、exchange 和 binds rule 等等，这就保证了可以在多个不同的 Application 中使用 RabbitMQ。

Connection 连接

TCP 连接，对于 RabbitMQ 而言，其实就是一个位于客户端和 Broker 之间 TCP 连接。Publisher 和 Consumer 都是通过 TCP 连接到 RabbitMQ Server 的。

Channel 信道

信道，多路复用连接中的一条独立的双向数据流通道。它建立在 TCP 连接中，数据流动都是在 Channel 中进行的。也就是说，一般情况是程序起始建立 TCP 连接，第二步就是建立这个 Channel。

AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这个动作都是通过信道完成。

引入信道的原因：

1. RabbitMQ 之间使用 TCP 连接，每次发布消息都要连接 TCP，建立和关闭 TCP （三次握手和四次挥手）都是有代价的，频繁的建立关闭 TCP 连接对于系统的性能有很大的影响，导致连接资源严重浪费，造成服务器性能瓶颈。
2. TCP 的连接数也有限制，这也限制了系统处理高并发的能力，如果使用 TCP 连接，高峰期每秒成千上万的连接造成资源浪费。
3. Channel 的原理一个进程一条通道，多条进程多条通道公用一条 TCP 连接，一条 TCP 连接可以容纳无限的 channel，不会有性能瓶颈。

仅仅创建了客户端到 Broker 之间的连接 Connection 后，客户端还是不能发送消息的，需要在 Connection 的基础上创建 Channel，AMQP 协议规定只有通过 Channel 才能执行 AMQP 的命令，一个 Connection 可以包含多个 Channel，每个 Channel 代表一个会话任务。

Queue 队列

消息队列，提供了 FIFO（先进先出） 的处理机制，具有缓存消息的能力。RabbitMQ 中，队列消息可以设置为持久化、临时或者自动删除。是 RabbitMQ 的内部消息，用于存储消息。

队列是消息载体，每个消息都会被投入到一个或多个队列，队列会保存消息直到发送给消费者，它是消息的容器，也是消息的终点。

队列是先进先出的，默认情况下先存储的消息先被处理。

设置为临时队列，Queue 中的数据在系统重启之后就会丢失。

设置为自动删除的队列，当不存在用户连接到 server，队列中的数据会被自动删除。

RabbitMQ 中的消息都只能存储在 Queue 中，生产者生产消息并最终投递到 Queue 中，消费者可以从 Queue 中获取消息并消费。

多个消费者可以订阅同一个 Queue，这时 Queue 中的消息会被平均分摊给多个消费者进行处理，而不是每一个消费者都收到所有的消息并处理。

Binding 绑定

绑定，它的作用就是把 exchange 和 queue 按照与规则绑定起来。

绑定用于消息队列和交换器之间的关联。

Exchange 和 Queue 的绑定可以是多对多的关系。

一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

Routing Key 路由键

路由关键字，exchange 根据这个关键字进行消息投递。

Exchange 交换器

信息交换器，路由消息，用来接收生产者发送的消息并将这些消息路由给服务器中的队列，可以根据应用场景的不同选择合适的交换机。

向 RabbitMQ 发送消息，实际上是把消息发到交换器上，再由交换器根据相关路由规则发到特定队列上，在队列上监听的消费者就可以进行消费了。所以生产者发送消息时会经有交换器（ Exchange ）来决定要给哪个队列（ Queue ）。ExchangeType 决定了 Exchange 路由消息的行为。

一个 Exchange 可以和多个 Queue 进行绑定，和 Queue 一样，Exchange 也可设置持久化、临时或者自动删除。

如果需要精准路由到队列，或者对消息进行单一维度分类可以使用 direct 类型交换器；如果需要广播消息，可以使用 fanout 类型交换器；如果对消息进行多维度分类，可以使用 topic 交换器；如果消息归类的逻辑包含了较多的 AND/OR 逻辑判断，可以使用 header 交换器（开发中很少用到 header 交换器）。

消息发送到没有队列绑定的交换机时，消息将丢失，因为交换机没有存储消息的能力，消息只能存在于队列中。

由 Exchange 、Queue、RoutingKey 三个才能决定一个从 Exchange 到 Queue 的唯一的线路。

Exchange 和 Queue 是在 RabbitMQ Server（ 也叫做 Broker ）端，Producer 和 Consumer 在应用端。

* 提高 RabbitMQ 传输消息数据的可靠性途径

*** RabbitMQ 消息幂等性**

OkHttp

OkHttp 基础知识

OkHttp 是一个用于进行 Http / Http2 通信的客户端，并且同时适用于 Android 和 Java。OkHttp 是一个用于进行 Http / Http2 通信的客户端，并且同时适用于 Android 和 Java。

特点

OkHttp 是一个非常强大而有效的网络架构，其主要特点在于：

- 对于同一个主机的所有请求，允许其在 Http /Http2 上共享同一个套接字，这就避免了重复的 TCP 连接带来的 3 次握手的时间。
- 对于 Http 协议，其支持连接池用于减少请求延迟。
- 数据都使用了 gzip 压缩传输，从而减少网络传输 size 的大小。
- 对响应进行缓存，避免缓存有效期内重复的网络请求。
- 弱网情况下，在连接失败后，OkHttp 会自动进行重试，特别是有备用地址时还会通过备用地址进行连接。而安全上，其支持新一代的 TLS 功能、SNL 和 ALPN，如果服务器不支持的化则会自动降级到 TLS 1.0。

OkHttp 的使用是很简单的，它在 request/reponse API 上采用了链式 Builder 的设计模式，使得它具备一旦构建便不可修改性。

OkHttp 还支持同步和异步请求。其实网络请求的实现原理上也是一次 I/O 通信，并且还是同步的 I/O。

使用

Http Get

```
// 创建 okHttpClient 实例
OkHttpClient mOkHttpClient = new OkHttpClient();
// 通过链式 Builder 设计提的 Builder 创建一个Request
final Request request = new Request.Builder()
             .url("https://github.com/hongyangAndroid")
             .build();
// new call
Call call = mOkHttpClient.newCall(request);
// 请求加入调度
// 异步请求
call.enqueue(new Callback()
        {
            // 请求失败
            @Override
            public void onFailure(Request request, IOException e)
            {
            }
      // 请求成功
            @Override
```

```
            public void onResponse(final Response response) throws IOException
            {
                String htmlStr =  response.body().string();
            }
        });
```

Http Post

```
Request request = buildMultipartFormRequest(
        url, new File[]{file}, new String[]{fileKey}, null);
// 构建 Body
FormEncodingBuilder builder = new FormEncodingBuilder();
builder.add("username","张三");
// 构建 Request，对于 post 请求，除了设置 url 还需要设置 post(body)
Request request = new Request.Builder()
                    .url(url)
                    .post(builder.build())
                    .build();
//执行一个异步请求。
 mOkHttpClient.newCall(request).enqueue(new Callback(){});
```

post 的时候，参数是包含在请求体中的，通过 FormEncodingBuilder 添加多个 String 键值对，然后去构造 RequestBody，最后完成 Request 的构造。

基于 Http 的文件上传

```
// 文件
File file = new File(Environment.getExternalStorageDirectory(),
"balabala.mp4");
// 构建 Body
RequestBody fileBody = RequestBody.create(MediaType.parse("application/octet-
stream"), file);

RequestBody requestBody = new MultipartBuilder()
     .type(MultipartBuilder.FORM) // 表单上传
     .addPart(Headers.of(
          "Content-Disposition",
              "form-data; name=\"username\""),
          RequestBody.create(null, "张三"))
     .addPart(Headers.of(
         "Content-Disposition",
         "form-data; name=\"mFile\";
         filename=\"wjd.mp4\""), fileBody)
     .build();

Request request = new Request.Builder()
    .url("http://192.168.1.103:8080/okHttpServer/fileUpload")
    .post(requestBody)
```

```
    .build();
// 发送异步请求
Call call = mOkHttpClient.newCall(request);
call.enqueue(new Callback()
{
    //...
});
```

通过 MultipartBuilder 的 addPart 方法可以添加键值对或者文件。

OkHttp 设置自定义拦截器

OkHttp 的使用是支持用户自定义拦截器的，而且自定义的拦截器会最先执行，并最后处理响应结果。

自定义拦截器

自定义拦截器主要的逻辑就是：

1. 实现 Interceptor 接口，重写 intercept(Interceptor.Chain chain) 方法，在使用责任链的时候，可以调用自定义拦截器的处理。
2. 调用 Response response = chain.proceed(request) 调用下一个拦截器，并获取响应结果。

下面是一个 log 拦截器的实现，在 intercept 的操作分为三部分：

1. 获取请求信息，打印
2. 调用下一个拦截器
3. 获取响应信息，打印

```
    /**
     * 打印日志使用
     */
    public static final class LoggerInterceptor implements Interceptor {
        private String tag;

        public LoggerInterceptor(String tag) {
            this.tag = tag;
        }

        // 重写 intercept 方法
        @Override
        public Response intercept(Chain chain) throws IOException {
            // 1.获取请求信息，打印
            Request request = chain.request();

            long t1 = System.nanoTime();
            LogUtil.i(tag, String.format("Sending request %s on %s%n%s\n%s",
                    request.url(), chain.connection(), request.headers(),
request.body()));
            // 2. 调用下一个拦截器
```

```
            // response 就是响应信息
            Response response = chain.proceed(request);
            // 3. 获取响应信息，打印
            long t2 = System.nanoTime();
            LogUtil.i(tag, String.format("Received response for %s in
%.1fms%n%s\n%s",
                    response.toString(), (t2 - t1) / 1e6d,
response.headers(),response.body()));
            return response;
        }
    }
```

使用自定义拦截器

使用自定义拦截器是通过 OkHttpCilent.Builder() 来配置的，有两种方式：

1. addInterceptor()

```
client = new OkHttpClient.Builder()
        .addInterceptor(new LoggerInterceptor())
        .build();
```

addInterceptor() 方法无需担心中间响应，例如重定向和重试。即使从缓存提供 HTTP 响应，也会被调用一次。遵守应用程序的原始意图，不关心 OkHttp 注入的标头，例如 If-None-Match，允许短路不被调用 chain.proceed()，也允许重试并多次调用 chain.proceed()。

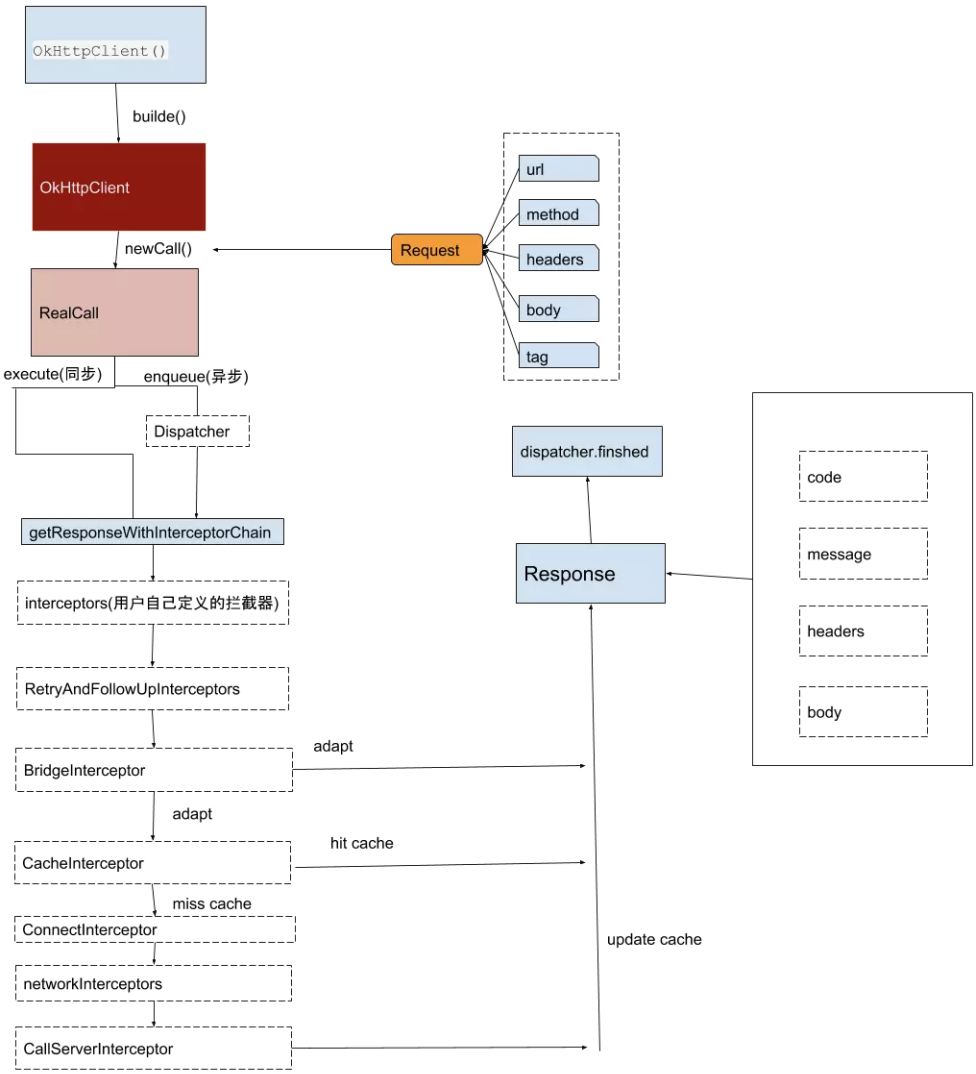
addNetworkInterceptor()

```
client = new OkHttpClient.Builder()
        .addNetworkInterceptor(new LoggerInterceptor())
        .build();
```

addNetworkInterceptor() 方法能够对诸如重定向和重试之类的中间响应进行操作，在读取缓存时不会被调用到，可以观察具体的请求数据，就像通过网络传输数据一样带有请求的访问 Connection。

addInterceptor 和 addNetworkInterceptor 主要的区别是 addInterceptor 是最先执行的拦截器，addNetworkInterceptor 是在 ConnectInterceptor 之后执行的拦截器。

OkHttp 源码解析



简述 OkHttp 的执行流程：

1. OkHttpClient 实现了 Call.Factory，负责为 Request 创建 call；
2. RealCall 为 Call 的具体实现，其 enqueue() 异步请求接口通过 Dispatcher() 调度器利用 ExcutorService 实现，而最终进行网络请求时和同步的 execute() 接口一致，都是通过 getResponseWithInterceptorChain() 函数实现。
3. getResponseWithInterceptorChain() 中利用 Interceptor 链条，分层实现缓存、透明压缩、网络 IO 等功能，最终将响应数据返回给用户。
4. OkHttp 的实现采用了责任链模式，它包含了一些命令对象和一系列的处理对象，每一个处理对象决定它能处理哪些命令对象，它也知道如何将它不能处理的命令对象传递给该链中的下一个处理对象，该模式还描述了往该处理链的末尾添加新的处理对象的方法。

OkHttp 里面的拦截器有：

1. 在配置 OkHttpClient 时设置的 interceptors（addInterceptor）。
2. 负责失败重试以及重定向的 RetryAndFollowUpInterceptor。
3. 负责把用户构造的请求转换为发送服务器的请求、把服务器返回的响应转换为用户友好的响应的 BridgeInterceptor。
4. 负责读取缓存直接返回、更新缓存的 CacheInterceptor。

5. 负责和服务器建立连接的 ConnectInterceptor。
6. 配置 OkHttpClient 时设置的 networkInterceptors（addNetworkInterceptor）。
7. 负责向服务器发送请求数据、从服务器读取响应数据的 CallServerInterceptor。

位置决定了功能，最后一个 CallServerInterceptor 负责和服务器实际通讯，重定向、缓存等一定是在实际通讯之前的。

RxJava

RxJava 就是一个实现异步操作的库。

RxJava 最大的优点就是简洁。

异步操作很关键的一点是程序的简洁性，因为在调度过程比较复杂的情况下，异步代码经常会既难写也难被读懂。Android 创造的 AsyncTask 和 Handler，其实都是为了让异步代码更加简洁。RxJava 的优势也是简洁，但它的简洁与众不同在于：随着程序逻辑变得越来越复杂，它依然能够保持简洁。

源码分析

RxJava 是响应式编程（Reactive Extensions）在 JVM 平台上的实现，即用 Java 语言实现的一套基于观察者模式的异步编程接口。

RxJava 是使用观察者模式实现的。

RxJava 中观察者模式

RxJava 有四个基本概念：Observable（可观察者，即被观察者）、Observer（观察者）、subscribe（订阅）、事件。Observable 和 Observer 通过 subscribe() 方法实现订阅关系，从而 Observable 可以在需要的时候发出事件来通知 Observer。

与传统观察者模式不同，RxJava 的事件回调方法除了普通的 onNext 之外，还定义了两个特殊的事件：onCompleted() 和 onError()。

- onCompleted()：事件队列完结。RxJava 不仅把每个事件单独处理，还会把它们看做一个队列。RxJava 规定，当不会再有新的 onNext() 发出时，需要触发 onCompleted() 方法作为标志。
- onError()：事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出。
- 在一个正确运行的事件序列中，onCompleted() 和 onError() 有且只有一个，并且是事件序列中的最后一个。需要注意的是，omCompleted() 和 onError() 二者也是互斥的，即在队列中调用了其中一个，就不应该再调用另一个。并且只要 onCompleted() 和 onError() 中有一个调用了，都会中止 onNext() 的调用。

基本实现

Observer 即观察者，它决定事件触发的时候将有怎样的行为。RxJava 中的 Observer 接口的实现方式：

```
        Observer<String> observer = new Observer<String>() {
            @Override
            public void onCompleted() {
                Log.d(TAG, "onCompleted");
            }
```

```
        @Override
        public void onError(Throwable e) {
            Log.d(TAG, "onError");
        }

        @Override
        public void onNext(String s) {
            Log.d(TAG, "onNext");
        }
    };
```

除了 Observer 接口之外，RxJava 还内置了一个实现了 Observer 的抽象类：Subscriber。

Subscriber 对 Observer 接口进行了一些扩展，但他们的基本使用方式是完全一样的：

```
    Subscriber<String> subscriber = new Subscriber<String>() {
        @Override
        public void onCompleted() {
            Log.d(TAG, "onCompleted");
        }

        @Override
        public void onError(Throwable e) {
            Log.d(TAG, "onError");
        }

        @Override
        public void onNext(String s) {
            Log.d(TAG, "onNext");
        }
    };
```

不仅基本使用方式一样，实质上，在 RxJava 的 subscribe 过程中，Observer 也总是会先被转换成一个 Subscriber 再使用。

Observer 和 Subscriber 的区别

如果只是使用基本功能，选择 Observer 和 Subscriber 是完全一样的。它们的区别对于使用者来说主要有两点：

1. onStart()：这是 Subscriber 增加的方法。

   它会在 subscribe 刚开始，而事件还未发送之前被调用，可以用于做一些准备工作。例如数据的清零或重置。

   这是一个可选方法，默认情况下它的实现为空。

   需要注意的是，如果对准备工作的线程有要求（例如弹出一个显示进度的对相框，这必须在主线程执行），onStart() 就不适用了，因为它总是在 subscribe 所发生的线程被调用，而不能指定线程。

   要在指定的线程来做准备工作，可以使用 doOnSubscribe() 方法。

2. unsubscribe()：这是 Subscriber 所实现的另一个接口 Subscription 的方法，用于取消订阅。

   在这个方法被调用后，Subscriber 将不再接受事件。

   一般在这个方法调用前，可以使用 isUnsubscribed() 先判断一下状态。

   unsubscribe() 这个方法很重要，因为在 subscribe() 之后，Observable 会持有 Subscriber 的引用，这个引用如果不能及时被释放，将有内存泄漏的风险。

   所以最好保持一个原则：要在不再使用的时候尽快在合适的地方（例如 onPause()、onStop() 等方法中）调用 unsubscribe() 来解除引用关系，以避免内存泄漏的发生。

RxJava 的基本订阅流程

一个简单的 RxJava 的使用：

```
        Observable.create(new Observable.OnSubscribe<String>() {
            @Override
            public void call(Subscriber<String> subscriber) {
                subscriber.onNext("next");
                subscriber.onCompleted();
            }
        })
        .subscribe(new Subscriber<String>() {
          @Override
          public void onCompleted() {
            Log.d(TAG, "onCompleted");
          }

          @Override
          public void onError(Throwable e) {
            Log.d(TAG, "showQuestionView onError");
          }

          @Override
          public void onNext(String string) {
            Log.d(TAG, "onNext string:"+string);
          }
        });
```

- Observable.create() 方法返回了一个 Observable 实例对象，并且将参数 OnSubscribe< T > f 存储为成员 onSubscribe。
- subscriber() 实际就做了 4 件事情：
  1. 调用 Subscriber.onStart() 。
  2. 如果 subscriber 不是 SafeSubscriber 类型，将传入的 Subscriber 转化为 SafeSubscriber，这是为了保证 onCompleted 或 onError 调用的时候会中止 onNext() 的调用，而将 subscriber 作为 SafeSubscriber 的 actual 成员。
  3. 调用 Observable 中的 OnSubscribe.call(Subscriber)。在这里，事件发送的逻辑开始运行。从这也可以看出，在 RxJava 中，Observable 并不是在创建的时候就立即开始发送事件，而是在它被订阅的时候，即当 subscribe() 方法执行的时候开始运行。

4. 被转化后的 SafeSubscriber 作为 Subscription 返回。这是为了方便 unsubscribe()。

- 通过 SafeSubscriber 中的布尔变量 done 来做标记保证 onCompleted() 和 onError() 二者的互斥性，即在队列中调用了其中一个，就不应该再调用另一个。并且只要 onCompleted() 和 onError() 中有一个调用了，都会中止 onNext() 的调用。

基本订阅流程总结

方法的主导只要由 Observable（被观察者） 来，在创建 Observable 的时候，会将 OnSubscribe(订阅操作)传给 Observable(被观察者) 作为成员变量，在调用 subscribe 的方法（订阅）时，将 Subscriber (观察者)作为参数传入，调用 onSubscribe 的 call 方法来处理订阅的事件，OnSubscribe 的 call 方法中调用 Subcriber 的相关方法来通知观察者。

概括

1. RxJava 主要采用的是观察者模式，Observable 作为被观察者，负责接收原始的 Observable 发出的事件，并在处理后发送给 Observer，Observer 作为观察者。
2. Observable 并不是在创建的时候就立即开始发送事件，而是在它被订阅的时候，也就是 subscribe() 方法执行的时候开始。
3. subscribe() 方法里会调用 OnSubscribe#call 方法，在 OnSubscribe 的 call 方法会把消息传递给观察者 Subscriber。

线程切换源码分析

RxJava 进行异步非常简单，只需要使用 subscribeOn 和 observeOn 这两个操作符即可。

subscribeOn 操作 OnSubscribe （订阅操作）的运行线程。

observeOn 操作观察者的运行线程。一般都是主线程，也就是 UI 线程。

subscribeOn 流程分析

简单使用

subscribeOn(Schedulers.computation()) 方法让 OnSubscribe()（订阅操作） 运行在计算线程。

简单使用：

```
        Thread th=Thread.currentThread();
        System.out.println("onResume Tread name:"+th.getName()); //out:onResume
Tread name:main
        Observable.create(new Observable.OnSubscribe<String>() { //
OnSubscribe1
            @Override
            public void call(Subscriber<? super String> subscriber) {
                Log.d(TAG, "call subscriber:" + subscriber );
                Thread th=Thread.currentThread();
                System.out.println("call Tread name:"+th.getName()); //out:call
Tread name:RxComputationScheduler-1
                subscriber.onNext("Hello");
                subscriber.onCompleted();
            }
        }) //Observable1
```

```
                .subscribeOn(Schedulers.computation())
                .subscribe(new Subscriber<String>() {
                    @Override
                    public void onCompleted() {
                        Log.d(TAG, "onCompleted");
                    }

                    @Override
                    public void onError(Throwable e) {
                        Log.d(TAG, "onError");
                    }

                    @Override
                    public void onNext(String s) {
                        Thread th=Thread.currentThread();
                        System.out.println("onNext Tread name:"+th.getName());
//out:onNext Tread name:RxComputationScheduler-1
                        Log.d(TAG, "onNext s:" + s);
                    }
                });
```

总结

subscribeOn 就是 create + OperatorSubscribeOn 实现。

从调用 OperatorSubscribeOn 的 call 方法，自己实现的 OnSubscribe1 对象的 call() 方法是在指定线程中运行，所以如果设置一个 subscribeOn 会导致 OnSubscribe1 对象的 call() 方法在指定线程中运行，而且 subscribeOn() 方法的 OnSubscribe1 只是指调用 subscribeOn() 方法的 Observable 对象，之后的 Observable 对象是没有用的。而 Subscriber 的 onNext() 方法也在指定线程运行，是因为在 call 中调用的时候没有切换线程，所以 onNext() 方法也在指定线程中运行。

observeOn 流程分析

简单使用

```
        Observable.create(new Observable.OnSubscribe<String>() { // Observable
1
            @Override
            public void call(Subscriber<? super String> subscriber) {
                Log.d(TAG, "call subscriber:" + subscriber );
                Thread th=Thread.currentThread();
                System.out.println("call Tread name:"+th.getName());
                subscriber.onNext("Hello");
                subscriber.onCompleted();
            }
        })
                // 订阅操作的运行线程
                .subscribeOn(Schedulers.computation())
                // 观察者运行线程
                .observeOn(AndroidSchedulers.mainThread())
```

```
                .subscribe(new Subscriber<String>() { // Subscriber 1
                    @Override
                    public void onCompleted() {
                        Log.d(TAG, "onCompleted");
                    }

                    @Override
                    public void onError(Throwable e) {
                        Log.d(TAG, "onError");
                    }

                    @Override
                    public void onNext(String s) {
                        Thread th=Thread.currentThread();
                        System.out.println("onNext Tread name:"+th.getName());
                        Log.d(TAG, "onNext s:" + s);
                    }
                });
```

AndroidSchedulers.mainThread() 返回的是什么?

AndroidSchedulers.mainThread 就是通过向主线程的 MessageQueue 中发消息，主线程的 Looper 会从 MessageQueue 取出来进行消费，处理消息也就到了主线程。

总结

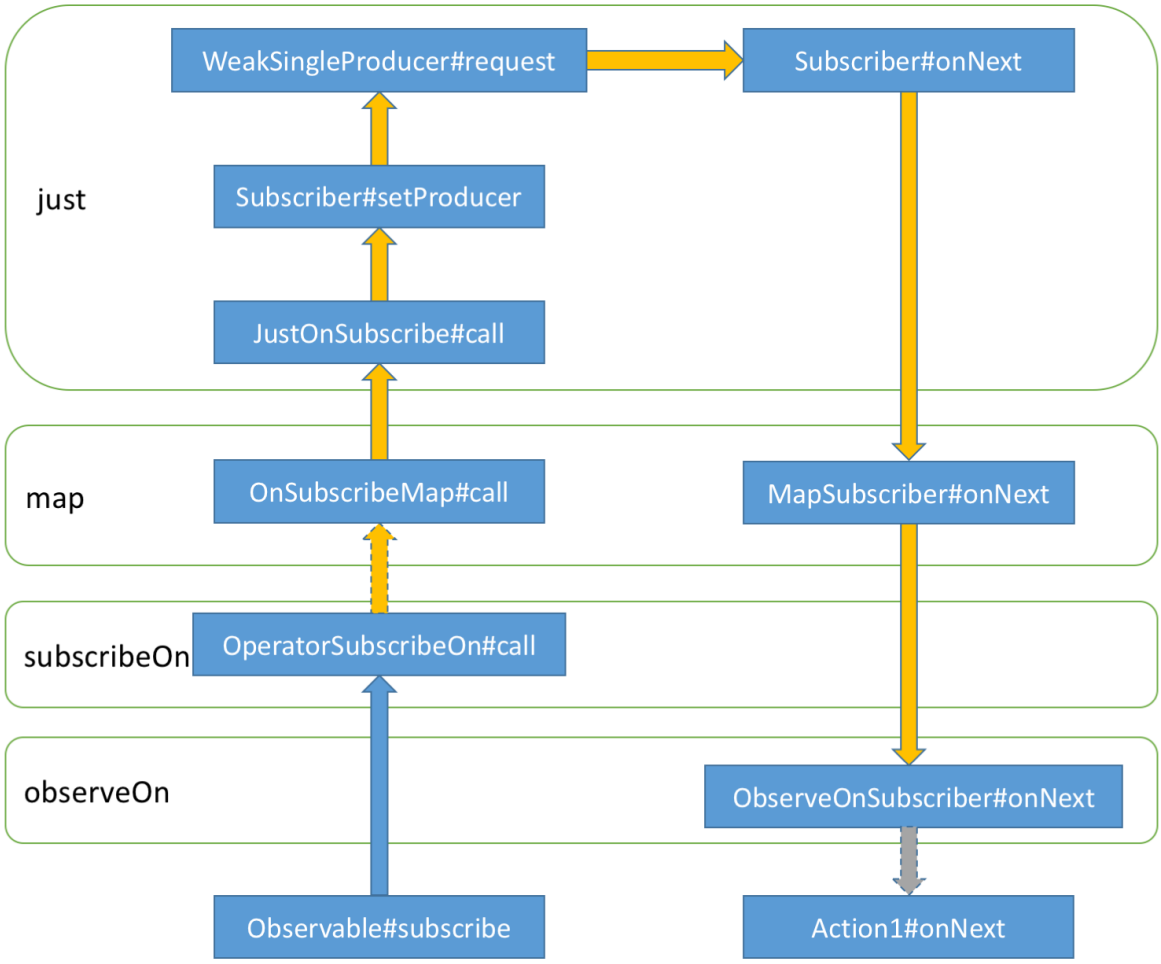
observeOn 就是 lift + OperatorObserveOn 实现。

```
public class Observable<T> {
    public final <R> Observable<R> lift(final Operator<? extends R, ? super T>
operator) {
        return new Observable<R>(new OnSubscribeLift<T, R>(onSubscribe,
operator));
    }
}
```

将 onSubscribe （也就是 onSubscribe1 ）与 operator (也就是 OperatorObserveOn)作为参数，创建 onSubscribeList，当前的 Observable 的 onSubscribe 成了 OnSubscribeLift 对象，而 onSubscribe1 成为 OnSubscribeLift 的 parent 变量，而 OperatorObserverOn 成为 OnSubscribeLift 的 operator 变量。

ObserveOn() 方法会生成 OperatorObserveOn 对象，并且将其设置为 Observable 的 onSubscribe 对象，并且将下游的 Subscriber 作为对象进行封装，在调用 onNext()、onError()、onComplete() 方法时通过向主线程发送 message 消息，在主线程中处理消息，从而确保 Subscriber 的 onNext()、onError()、onComplete() 运行在主线程。

完整过程



subscribeOn() 方法会使用 OperatorSubscribeOn 类作为 Observable 的 onSubscribe 对象，将上游的 Observable 进行封装，从而确保上游的 OnSubscribe 的 call() 方法运行在指定线程。
ObserveOn() 方法会使用 OperatorObserveOn 类作为 Observable 的 onSubscribe 对象，将下游的 Subscriber 进行封装，从而确保 Subscriber 的 onNext()、onError()、onComplete() 运行在指定的线程。

* 操作符源码分析

RxJava1 与 RxJava2 的对比

1. 接口变化

RxJava 2.x 拥有了新的特性，其依赖于 4 个基本接口，它们分别是：

- Publisher
- Subscriber
- Subscription
- Processor

其中最核心的莫过于 Publisher 和 Subscriber。Publisher 可以发出一系列的事件，而 Subscriber 负责和处理这些事件。

其中用的比较多的自然是 Publisher 的 Flowable，它支持背压。

很明显，RxJava 2.x 最大的改动就是对于 backpressure 的处理，为此将原来的 Observable 拆分成了新的 Observable 和 Flowable，同时其他相关部分也同时进行了拆分。

2. 背压概念

异步环境下产生的问题：同步环境下会等待一件事处理完后再进行下一步，而异步环境下是处理完一件事，未等它得出结果接着处理下一步，在获得结果之后进行回调，再处理结果。

发送和处理速度不统一：例如生产者生产的产品放置到缓存队列中，供消费者消费。若生产者生产的速度大于消费者消耗的速度，则会出现缓存队列溢出的问题。

背压是一种流速控制即解决策略，例如背压中的丢弃策略，一旦发现缓存队列已满，为了整个过程顺利进行，则会丢弃最新产生的产品，避免溢出，因此背压也是一种流速控制的解决策略。

* RxJava2 背压源码分析

Glide

Picasso 比 Glide 更加简洁和轻量，Glide 比 Picasso 功能更为丰富。

Glide 3 的用法

在 Activity 中使用 Glide 显示图片：

```
Glide.with(Context context).load(Strint url).into(ImageView imageView);
```

首先调用 Glide.with() 方法用于创建一个加载图片的实例。

with() 方法可以接收 Context、Activity 或者 Fragment 类型的参数。也就是说选择的范围非常广，不管是在 Activity 还是 Fragment 中调用 with() 方法，都可以直接传 this。那如果调用的地方既不在 Activity 中也不在 Fragment 中也没有关系，可以获取当前应用程序的 ApplicationContext，传入到 with() 方法当中。

注意 with() 方法中传入的实例会决定 Glide 加载图片的生命周期，如果传入的是 Activity 或者 Fragment 的实例，那么当这个 Activity 或 Fragment 被销毁的时候，图片加载也会停止。如果传入的是 ApplicaationContext，那么只有当应用程序被杀掉的时候，图片加载才会停止。

load() 方法用于指定待加载的图片资源。Glide 支持加载各种各样的图片资源，包括网络图片、本地图片、应用资源、二进制流、Uri 对象等的。因此 load() 方法也有很多个方法重载。

into() 方法中传入图片显示的 ImageView 的实例，将图片显示在这个 ImageView 上。into() 方法不仅仅是只能接收 ImageView 类型的参数，还支持很多更丰富的用法。

Glide 的关键三步是：先 with()、再 load()、最后 into()。

- 加载占位图：placeholder
- 异常占位图：error

Glide 是支持 GIF 图片的，而 Picasso 是不支持加载 GIF 图片的。

- 指定图片大小：override

实际上，使用 Glide 在绝大多数情况下都是不需要指定图片大小的。

Glide 从来都不会直接将图片的完整尺寸全部加载到内存中，而是用多少加载多少。Glide 会自动判断 ImageView 的大小，然后只将这么大的图片像素加载到内存当中，从而节省内存开支。

所以说 Glide 在绝大多数情况下都是不需要指定图片大小的，因为 Glide 会自动根据 ImageView 的大小来决定图片的大小。

- 关闭硬件缓存：diskCacheStrategy(DiskCacheStrategy.NONE)

  Glide 有非常强大的缓存机制，第一次加载的时候会把图片缓存下来，下次加载的时候将会直接从缓存中读取，不会再去网络下载，因而加载的速度非常快。

Glide 执行基本流程源码分析 1

with

可以看到，with() 方法的重载方法非常多，既可以传入 Activity，也可以传入 Fragment 或者是 Context。但是每一个 with() 方法重载的代码都非常简单：

1. 先调用 RequestManagerRetriever 的静态 get() 方法得到一个 RequestManagerRetriever 对象，这个静态 get() 方法就是一个单例实现。
2. 再调用 RequestManagerRetriever 的实例 get() 方法，去获取 RequestManager 对象。

所以 with() 方法返回的就是一个 RequestManager 对象。

RequestManagerRetriever 类中有很多个 get() 方法的重载，Context 参数、Activity 参数、Fragment 参数等等，但是实际上只有两种情况而已：

1. 传入 Application 类型的参数。
2. 传入非 Application 类型的参数。

传入 Application 参数的 get() 方法：如果在 Glide.with() 方法中传入的是一个 Application 对象，那么这里就会调用带有 Context 参数的 get() 方法重载，然后会调用 getApplicationManager() 方法来获取一个 RequestManager 对象。其实这是最简单的一种情况，因为 Application 对象的生命周期即应用程序的生命周期，因此 Glide 并不需要做什么特殊的处理，它自动就是和应用程序的生命周期是同步的，如果应用程序关闭的话，Glide 的加载也会同时终止。

传入非 Application 参数的 get() 方法：在使用 Glide.with() 方法中不管传入的是 Activity、FragmeActivity、v4 包下的 Fragment 还是 app 包下的 Fragment，最终的流程都是一样的，那就是调用 getSupportRequestManagerFragment 方法向当前的 Activity 当中添加一个隐藏的 Fragment。为什么要添加一个隐藏的 Fragment？因为 Glide 需要知道加载的生命周期。Glide 并没有办法知道 Activity 的生命周期，于是 Glide 就是用了添加隐藏 Fragment 这种小技巧，因为 Fragment 的生命周期和 Activity 是同步的，如果 Activity 被销毁了，Fragment 是可以监听到的，这样 Glide 就可以捕获这个事件并停止图片加载了。

load

ModelLoader 是一个工厂接口，只有一个 getResourceFetcher() 方法。而 getResourceFacher() 方法用于获取 DataFetcher 对象，DaraFetcher 对象可以获取解码资源的数据。

load() 方法会先调用其父类 GenericRequestBuilder 的 load() 方法，然后将自己返回，也就是说，最终 load() 方法返回的其实就是一个 DrawableRequestBuilder 对象，并且 DrawableRequestBuilder 类中有一个 into() 方法。

Glide 执行基本流程源码分析 2

into() 方法实现在 DrawableRequestBuilder 的父类 GenericRequestBuilder 类中。

into() 方法的实现代码很长，只跟着请求网络图片的代码看 Glide 执行流程，并且在这里分为三部分来解析：

1. 网络请求；
2. 解析网络结果；
3. 将图片显示到界面上。

网络请求

1. 工厂模式，根据不同的图片类型（GlideDrawable、Bitmap、Drawable），创建不同的 target（GlideDrawableImageViewTarget、BitmapImageViewTarget、DrawableImageViewTarget）
2. into 方法中创建一个请求（Request request = buildRequest(target)），并执行这个请求（requestTracker.runRequest(request);）
3. runRequest 有一个简单的逻辑判断，就是先判断 Glide 当前是不是处于暂停状态，如果不是暂停状态就调用 Request 的 begin() 方法来执行 Request，否则的话就先将 Request 添加到待执行队列里面，等暂停状态解除了之后再执行。
4. 在 beigin() 方法中，具体的图片加载是由 onSizeReady() 和 target.getSize() 两个方法来完成的。这里分为两种情况，一种是使用了 override() API 为图片指定了一个固定的宽高，一种是没有指定。如果指定了的话，就会执行 onSizeReady() 方法。如果没指定的话，就会执行 target.getSize() 方法。而 target.getSize() 方法的内部会根据 ImageView 的 layout_width 和 layout_height 值做一系列的计算，来算出图片应该的宽高，在计算完之后，它也会调用 onSizeReady() 方法。也就是说，不管是哪种情况，最终都会调用到 onSizeReady() 方法。
5. 在 onSizeReady() 方法中先是处理缓存，如果有缓存，就调用回调的 onResourceReady(cached) 方法，并返回，如果没有缓存就先构建了一个 EngineJob，它的主要作用就是用来开启线程的，为后面的异步加载图片做准备。接着创建了一个 DecodeJob 对象，主要用来对图片进行解码的，然后创建了一个 EngineRunnable 对象，并且调用了 EngineJob 的 start() 方法来运行 EngineRunnable 对象，这实际上就是让 EngineRunnable 的 run() 方法在子线程当中执行了。
6. 调用到了HttpUrlFetcher 的 load 方法进行网络请求（urlConnection.connect()），返回了一个 InputStream，服务器返回的数据还没有开始读。回到 ImageVideoFetcher 的 loadData() 方法中，在这个方法的最后一行，创建了一个 ImageVideoWrapper 对象，并将得到的 InputStream 作为参数传了进去。

解析请求结果

1. 从 ImageVideoFetcher 的 loadData() 方法返回到 DecodeJob 的 decodeSource() 方法中，在得到了这个 ImageVideoWrapper 对象之后，紧接着又将这个对象传入到了 decodeFromSourceData() 当中，去解码这个对象。
2. 在 DecodeJob 的 decodeFromSourceData() 方法中，调用了 loadProvider.getSourceDecoder().decode() 方法来进行解析。loadProvider 就是在 onSizeReady() 方法中得到的 FixedLoadProvider，而 getSourceDecoder() 得到的则是一个 GifBitmapWrapperResourceDecoder 对象，也就是要调用这个对象的 decode() 方法来对图片进行解码。

3. 在 decode 方法中调用 decodeStream() 方法，decodeStream() 方法中会先从流中读取 2 个字节的数据，来判断这张图是 GIF 图还是普通的静图，如果是 GIF 图就调用 decodeGifWrapper() 方法来进行解码，如果是普通的静图就调用 decodeBitmapWrapper() 方法来进行解码。而 decodeBitmapWrapper() 方法调用了 bitmapDecoder.decode() 方法。这个 bitmapDecoder 是一个 ImageVideoBitmapDecoder 对象。
4. ImageVideoBitmapDecoder 的 decode() 方法先调用了 source.getStream() 来获取服务器返回的 InputStream，然后调用了 streamDecoder.decode() 方法进行解码。stramDecode 是一个 StreamBitmapDecoder 对象。
5. StreamBitmapDecoder 的 decode 方法又去调用了 DownSampler 的 decode() 方法。
6. 在 DownSampler 的 decode() 方法中，对服务器返回的 InputStream 的读取，以及转换为图片格式全都在这里了。当然这里其实处理了很多的逻辑，包括对图片的压缩，甚至还有旋转、圆角等逻辑处理。decode() 方法执行之后，会返回一个 Bitmap 对象，那么图片在这里其实也就已经被解析出来了，剩下的工作就是如何让这个 Bitmap 显示到界面上。
7. 回到 StreamBitmapDecoder 当中，decode() 方法返回的是一个 Resource< Bitmap > 对象。而从 DownSampler 中得到的是一个 Bitmap。因此在 StreamBitmapDecoder 的 decode 方法中又调用了 BitmapResource.obtain() 方法，将 Bitmap 对象包装成了 Resource< Bitmap > 对象。
8. BitmapResource 的源码也非常简单，经过这样一层包装之后，如果还需要获取 Bitmap，只需要调用 Resource< Bitmap > 的 get() 方法就可以了。

   然后一层一层的向上返回，StreamBitmapDecoder 会将值返回到 ImageVideoBitmapDecoder 当中，而 ImageVideoBitmapDecoder 又会将值返回到 GifBitmapWrapperResourceDecoder 的 decodeBitmapWrapper() 方法当中。

   ```
   而 GifBitmapWrapperResourceDecoder 的 decodeBitmapWrapper() 方法中又将 Resource< Bitmap > 封装到了一个 GifBitmapWrapper 对象当中，并且返回的也是一个 GifBitmapWrapper 对象。

   GifBitmapWrapper 就是既能封装 GIF，又能封装 Bitmap，从而保证了不管什么类型的图片 Glide 都能从容应对。
   ```

9. GifBitmapWrapper 类比较简单，就是分别对 gifResource 和 bitmapResource 做了一层封装而已。

   然后这个 GifBitmapWrapper 对象会一直向上返回，返回到 GifBitmapWrapperResourceDecoder 最外层的 decode() 方法的时候，会对它再做一次封装，将 GifBitmapWrapper 封装到了一个 GifBitmapWrapperResource 对象当中，最终返回的是一个 Respurce< GifBitmapWrapper > 对象。这个 GifBitmapResource 和 BitmapResource 是相似的，它们都实现了 Resource 接口，都可以通过 get() 方法来获取封装起来的具体内容。

将图片展示到界面上

1. 接着回到 DecodeJob 当中，它的 decodeFromSourceData() 方法返回的是一个 Resource< T > 对象，其实也就是 Resource< GifBitmapWrapper > 对象了。接着继续向上返回，最终返回到 DecodeJob 的 decodeFromSource() 方法当中，接着调用 transformEncodeAndTranscode() 方法， 而且 decodeFromSource() 方法最终返回的是一个 Resource< Z > 对象，注意传入的参数是 Resource< T >。

2. GifBitmapWrapperDrawableTranscoder 的核心作用就是用来转码的。因为 GifBitmapWrapper 是无法直接显示到 ImageView 上面的，只有 Bitmap 或者 Drawable 才能显示到 ImageView 上。因此，这里的 transcode() 方法先从 Resource< GifBitmapWrapper > 中取出 GifBitmapWrapper 对象，然后再从 GifBitmapWrapper 中取出 Resource< Bitmap >对象。

   接下来做了一个判断，如果 Resource< Bitmap > 为空，那么说明此时加载的是 GIF 图，直接调用 getGifResource() 方法将图片取出即可，因为 Glide 用于加载 GIF 图片使用的是 GifDrawable 这个类，它本身就是一个 Drawable 对象了。而如果 Resource< Bitmap > 不为空，那么就需要再做一次转码，将 Bitmap 转换成 Drawable 对象才行，因为要保证静图和动图的类型一致性，方便逻辑上处理的。

   ```
   这里又进行了一次转码，是调用的 GlideBitmapDrawableTranscoder 对象的 transcode() 方法。
   ```

3. 在 EngineRunnable 的 run 方法中 decode() 方法执行之后最终得到了 Resource< GlideDrawable > 对象，那么接下来就是如何将它显示出来了。

   在 EngineRunnable 的 run 方法中调用了 onLoadComplete() 方法，表示图片加载已经完成了。

4. EngineRunnable 的 onLoadComplete() 方法中调用了manager.onResourceReady(resource)，这个 manager 就是 EngineJob 对象，因此这里实际上调用的是 EngineJob 的 onResourceReady() 方法。

5. EngineJob 的 onResourceReady() 方法中使用 Handler 发出了一条 MSG_COMPLETE 消息，那么在 MainThreadCallback 的 handleMessage() 方法中就会收到这条消息。从这里开始，所有的逻辑又回到主线程当中进行了，因为很快就需要更新 UI 了。

   在 MainThreadCallback 的 handleMessage 方法中，如果收到完成的消息，就会调用 EngineJob 的 handleResultOnMainThread 方法。

6. 在 handleResultOnMainThread() 方法通过一个循环调用了所有 ResourceCallback 的 onResourceReady() 方法，而 ResourceCallback 是通过 addCallback() 方法向 cbs 集合中去添加的 ResourceCallback 。

   而 addCallback() 方法是在 Engine 的 load() 方法中调用的。在 Engine 的 load() 方法里调用了 EngineJob 的 addCallback() 方法来注册的一个 ResourceCallback。而 Engine.load() 方法的 ResourceCallback 参数是在 GenericRequest 的 onSizeReady() 方法中调用 engine.load() 方法的时候传入的 GenericRequest 本身对象。GenericRequest 本身就实现了 ResourceCallback 的接口，所以 handleResultOnMainThread() 方法中调用的 ResourceCallback 的 onResourceReady 就是 GenericRequest 的 onResourceReady 方法。

7. 首先在第一个 onResourceReady() 方法当中，调用 resource.get() 方法获取到了封装的图片对象，也就是 GlideBitmapDrawable 对象，或者是 GifDrawable 对象，然后将这个值传入到了第二个 onResourceReady() 方法当中，并且调用了 target.onResourceReady() 方法。

   而这个 target 是在 into() 方法的最后一行调用了 glide.buildImageViewTarget() 方法来构建出一个 Target，而这个 Target 就是一个 GlideDrawableImageViewTarget 对象。

8. 在 GlideDrawableImageViewTarget 的 onResourceReady() 方法中做了一些逻辑处理，包括如果是 GIF 图片的话，就调用 resource.start() 方法开始播放图片。其中还调用了 super.onResourceReady() 方法，GlideDrawableImageViewTraget 的父类是 ImageViewTarget。

9. 可以看到，在 ImageViewTarget 的 onResourceReady() 方法当中调用了 setResource() 方法，而 ImageViewTarget 的 setResource() 方法是一个抽象方法，具体的实现还是在子类那边实现的。
10. 调用了 view.setImageDrawable() 方法，而这个 view 就是使用 Glide 时传递进来的 ImageView。这样图片就显示出来了。

总结

对 Glide 的基本执行流程总结如下：

1. with() 方法如果传入的不是 Application 的 Context，就会向当前的 Context 添加一个隐藏的 fragment，这主要是为了知道加载的生命周期，而 with() 会返回一个 RequestManager 对象。
2. load() 方法会创建 Glide 的实例，并调用了 Glide 的初始化，对缓存、下载等对象进行初始化，注册好 ModelLoader 的工厂模式，创建 DrawableTypeRequest 对象并返回，DrawableTypeRequest 包含 streamModelLoader 和 fileDescriptorModelLoader 两个 ModelLoader。

   ModelLoader 是一个工厂接口，主要目标是：1. 将特定的模型转换为可解码为资源的数据类型；2. 允许模型与视图的纬度组合以获得特定大小的资源。有一个接口是 getResourceFetcher() 返回 DataFetcher。

   DataFetcher 是用于延迟检索加载资源的数据的接口。方法 loadData() 用于异步从解码资源中获取数据。
3. into() 方法主要分为三个部分：1. 请求数据；2. 将请求的数据解析转换为图片格式；3. 将图片显示到 ImageView 上。请求数据是使用 Engine 来开启线程来调用 Fetcher 接口的 loadData 来获取数据的。数据转码是通过 ResourceDecoder 接口的 decoder() 方法来实现的，将数据转换为图片或者 Gif 格式，并将结果封装为 Resouce 对象。将图片显示到 ImageView 的过程是先使用 ResourceTranscode 接口的 transcode() 方法将 Resouce 包含的 Bitmap 转换为可以显示的 Drawable 格式，转换完成后使用 Handle 进入 UI 线程，然后显示图片。

Glide 缓存机制源码分析

Glide 的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide 又将它分成了两个模块，一个是内存缓存，一个是硬件缓存。

这两个缓存模块的作用各不相同，内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存和硬盘缓存的相互结合才构成了 Glide 极佳的图片缓存效果。

缓存 key

既然是缓存功能，就必然会有用于进行缓存的 Key。那么 Glide 的缓存 Key 是怎么生成的呢？Glide 的缓存 Key 生成规则非常繁琐，决定缓存 Key 的参数竟然有 10 个之多。不过逻辑还是比较简单的。

生成缓存 Key 的代码在 Engine 类的 load() 方法当中开始。

在 lode() 方法里面调用了 fetcher.getId() 方法获得了一个 id 字符串，这个字符串就是要加载的图片的唯一标识，比如说如果是一张网络上的图片的话，那么这个 id 就是这张图片的 url 地址。

接下来将这个 id 连同着 signature、width、height 等等 10 个参数一起传入到 EngineKeyFactory 的 buildKey() 方法当中，从而构建出了一个 EngineKey 对象，这个 EngineKey 也就是 Glide 中的缓存 Key 了。

可见，决定缓存 Key 的条件非常多，即使用 override() 方法改变了一下图片的 width 或者 height，也会生成一个完全不同的缓存 Key。

EngineKey 的源码主要就是重写了 enquals() 和 hashCode() 方法，保证只有传入 EngineKey 的所有参数都相同的情况下才认为是同一个 EngineKey 对象。

内存缓存

从内存缓存中获取

有了缓存 Key，接下来就开始进行缓存了，先看内存缓存。

在默认情况下，Glide 自动就是开启内存缓存的。也就是说，当使用 Glide 加载了一张图片之后，这张图片就会被缓存到内存当中，主要在它还没从内存中被清除之前，下次使用 Glide 再加载这张图片都会直接从内存当中读取，而不用重新从网络或硬盘上读取了，这样无疑就可以大幅度提升图片的加载效率。比如再一个 RecyclerView 当中反复上下滑动，RecyclerView 中只要是 Glide 加载过的图片都可以直接从内存当中迅速读取并展示出来，从而大大提升了用户体验。

而 Glide 最为人性化的是，甚至不需要编写任何额外的代码就能自动享受到这个极为便利的内存缓存功能，因为 Glide 默认就已经将它开启了。

Glide 也提供了接口来关闭 Glide 的默认内存缓存：

```
Glide.with(this)
     .load(url)
     .skipMemoryCache(true)
     .into(imageView);
```

只需要调用 skipMemoryCache() 方法并传入 true，就标识禁用了 Glide 的内存缓存功能。

Glide 内存缓存的实现是使用的 LruCache 算法，LruCache 算法（Least Recently Used）也叫近期最少使用算法。它的主要算法原理就是把最近使用的对象用强引用存储在 LinkedHashMap 中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。Glide 还结合了一种弱引用的机制，共同完成了内存缓存功能。

就是如果能从内存缓存当中读取到要加载的图片，那么就直接进行回调，如果读取不到的话，才会开启线程执行后面的图片加载逻辑。

写入内存缓存

在图片加载完成之后，会在 EngineJob 当中通过 Handler 发送一条消息将执行逻辑切回到主线程当中，从而执行 handleResultOnMainThread() 方法。

handleResultOnMainThread() 方法中通过 EngineResourceFactory 构建出了一个包含图片资源的 EngineResource 对象，然后将这个对象回调到了 Engine 的 onEngineJobComplete() 方法当中。

Engine 的 onEngineJobComplete() 方法当中，回调回来的 EngineResource 被 put 到了 activeResources 方法，也就是在这里写入的缓存。

LruCache 缓存

EngineResource 中的一个引用机制：观察刚才的 handleResultOnMainThread() 方法，有调用 EngineResource 的 acquire() 方法，接着又调用了 EngineResource 的 release() 方法。其实，EngineResource 是用一个 acquired 变量用来记录图片被引用的次数，调用 acquire() 方法会让变量加 1，调用 release() 方法会让变量减 1。

当 acquired 变量大于 0 的时候，说明图片正在使用中，也就应该放在 activeResources 弱引用缓存当中，而经过 release() 之后，如果 acquired 变量等于 0 了，说明图片已经不再被使用了，那么此时会调用 listener 的 onResourceReleased() 方法来释放资源，这个 listener 就是 Engine 对象。

Engine的onResourceReleased中会将缓存图片从 activeResources 中删除，然后再将它 put 到 LruResourceCache 当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用 LruCache 来进行缓存的功能。

硬盘缓存

从磁盘缓存中读取

禁止 Glide 对图片进行硬件缓存使用的代码是：

```
Glide.with(this)
     .load(url)
     .diskCacheStrategy(DiskCacheStrategy.NONE)
     .into(imageView);
```

调用 diskCacheStrategy() 方法并传入 DiskCacheStrategy.NONE，就可以禁用掉 Glide 的硬盘缓存功能了。

这个 diskCacheStrategy() 方法基本就是 Glide 硬盘缓存功能的一切，它可以接收四种参数：

- DiskCacheStrategy.NONE：表示不缓存任何内容。
- DiskCacheStrategy.SOURCE：表示只缓存原始图片。
- DiskCacheStrategy.RESULT：表示只缓存转换过后的图片（默认选项）。
- DiskCacheStrategy.ALL：表示既缓存原始图片，也缓存转换过后的图片。

用 Glide 去加载一张图片的时候，Glide 默认并不会将原始图片展示出来，而是会对图片进行压缩和转换。总之就是经过种种一系列操作之后得到的图片，就叫转换过后的图片，而 Glide 默认情况下在硬盘缓存的就是转换过后的图片，通过调用 diskCacheStrategy() 方法则可以改变这一默认行为。

和内存缓存类似，硬盘缓存的实现也是使用 LruCache 算法，而且 Google 还提供了一个现成的工具类 DiskLruCache。

Glide 开启线程来加载图片后会执行 EngineRunnable 的 run 方法，run() 方法中又会调用一个 decode() 方法。

EngineRunnable 的 decode() 方法中，会分为两种情况，一种是调用 decodeFromCache() 方法从硬盘缓存当中读取图片，一种是调用 decodeFromSource() 方法读取原始图片。

默认情况下 Glide 会优先从缓存当中读取，只有缓存中不存在要读取的图片时，才会去读取原始图片。

EngineRunnable 的 decodeFromSource() 方法中会先去调用 DecodeJob 的 decodeResultFromCache() 方法来获取缓存，如果获取不到，会再调用 decodeSourceFromCache() 方法获取缓存，这两个方法的区别其实就是 DiskCacheStrategy.RESULT 和 DiskCacheStrategy.SOURCE 这两个参数的区别。

DecodeJob的decodeResultFromCache和decodeSourceFromCache方法都是调用了 loadFromCache() 方法从缓存当中读取数据，如果是 decodeResultFromCache() 方法就直接将数据解析并返回，如果是 decodeSourceFromCache() 方法，还要调用一下 transformEncodeAndTranscode() 方法先将数据转换一下再解析并返回。

这两个方法中在调用 loadFromCache() 方法时传入的参数却不一样，一个传入的是 resultKey，另外一个却又调用了 resultKey 的 getOriginalKey() 方法。Glide 的缓存 Key 是由 10 个参数共同组成的，包括图片的 width、height 等等。但如果要缓存的原始图片，其实并不需要这么多的参数，因为不用对图片做任何的变化。

getOriginalKey只使用了 id 和 signature 这两个参数来构成缓存 Key。

DecodeJob的loadFromCache方法调用 getDiskCache() 方法获取到的就是 Glide 自己编写的 DiskLruCache 工具类的实例，然后调用它的 get() 方法并把缓存 Key 传入，就能得到硬件缓存的文件了。如果文件为空就返回 null，如果文件不为空则将它解码成 Resource 对象后返回即可。

写入磁盘缓存

在 EngineRunnable 的 decode() 方法中，在没有缓存的情况下，会调用 decodeFromSource() 方法来读取原始图片，而 deocderFromSource() 方法调用了 DecodeJob 的 decodeFromSource() 方法。

DecodeJob的decodeFromSource方法中调用了 decodeSource() 方法解析原图片，调用 transformEncodeAndTranscode() 则是用来对图片进行转码和解码的。

在 decodeSource() 方法中会先调用 fetcher 的 loadData() 方法读取图片数据，然后调用 decodeFromSourceData() 方法来对图片进行解码。

在 decodeFromSourceData() 方法中先判断是否允许缓存原始图片，如果允许的话又会调用 cacheAndDecodeSourceData() 方法。而在这个方法中同样调用了 getDiskCache() 方法来获取 DiskLruCache 实例，接着调用它的 put() 方法就可以写入磁盘缓存了，注意原始图片的缓存 Key 使用的 result.getOriginalKey()。

原始图片的缓存写入就是这么简单。接着来看 transformEncodeAndTranscode() 方法如何写入转换过后的图片缓存。

在 transformEncodeAndTranscode() 方法中先是调用 transform() 方法来对图片进行转换，然后在 writeTransformedToCache() 方法中将转换过后的图片写入到硬盘缓存中，调用的同样是 DiskLruCache 实例的 put() 方法，不过这里用的缓存 Key 是 resultKey。

*** Glide 回调与监听**

*** Glide 图片变化功能**

Glide 自定义模块功能

自定义模块的基本用法

首先需要定义一个自己的模块类，并让它实现 GlideModule 接口，如下所示：

```
public class MyGlideModule implements GlideModule {
    @Override
    public void applyOptions(Context context, GlideBuilder builder) {
    }

    @Override
    public void registerComponents(Context context, Glide glide) {
    }
}
```

可以看到，在 MyGlideModule 类当中，重写了 applyOptions() 和 registerComponents() 方法，这两个方法分别就是用来更改 Glide 配置以及替换 Glide 组件的。只需要在这两个方法中加入具体的逻辑，就能实现更改 Glide 配置或者替换 Glide 组件的功能了。

不过，目前 Glide 还无法识别自定义的 MyGlideModule，如果想要让它生效，还得在 AndroidManifest.xml 文件当中加入如下配置才行：

```
<manifest>

    ...

    <application>

        <meta-data
            android:name="com.example.glidetest.MyGlideModule"
            android:value="GlideModule" />

        ...

    </application>
</manifest>
```

在 < application > 标签中加入一个 meta-data 配置项，其中 android:name 指定成自定义的这个 MyGlideModule 了。

Glide 使用的是基于原生 HttpURLConnection 进行订制的 HTTP 通讯组件。

*** Glide 带进度的图片加载**

*** Glide 4 的使用**