27.开源框架

27.1.Okhttp

1.exectute执行

1.1.真正的请求交给了 RealCall 类

exectute()方法执行RealCall的execute方法

client.dispatcher().enqueue(new AsyncCall(responseCallback));

利用dispatcher调度器enqueueAsyncCall，并通过回调（Callback）获取服务器返回的结果

1.2 Dispatcher

Dispatcher将call 加入到队列中，然后通过线程池来执行call

Dispatcher是一个任务调度器，它内部维护了三个双端队列： readyAsyncCalls：准备运行的异步请求 runningAsyncCalls：正在运行的异步请求 runningSyncCalls：正在运行的同步请求

新来的请求放队尾，执行请求从对头部取。

1.3 线程池

这不是一个newCachedThreadPool吗？没错，除了最后一个threadFactory参数之外与newCachedThreadPool一毛一样，只不过是设置了线程名字而已，用于排查问题。

阻塞队列用的SynchronousQueue，它的特点是不存储数据，当添加一个元素时，必须等待一个消费线程取出它，否则一直阻塞。

通常用于需要快速响应任务的场景，在网络请求要求低延迟的大背景下比较合适，

采用责任链的模式来使每个功能分开，每个Interceptor自行完成自己的任务

2.拦截器

利用Builder模式配置各种参数，例如：超时时间、拦截器等

retryAndFollowUpInterceptor

失败和重定向拦截器

当请求内部抛出异常时，判定是否需要重试

当响应结果是3xx重定向时，构建新的请求并发送请求

BridgeInterceptor

封装request和response拦截

负责把用户构造的请求转换为发送到服务器的请求

把服务器返回的响应转换为用户友好的响应

CacheInterceptor

当在OkHttpClient中配置了缓存，则将这个Resposne缓存起来

ConnectInterceptor——连接服务，负责和服务器建立连接 这

负责了Dns解析和Socket连接

CallServerInterceptor

传输http的头部和body数据

3.addInterceptor 和 addNetworkdInterceptor区别

在OkHttpClient.Builder的构造方法有两个参数，使用者可以通过addInterceptor 和 addNetworkdInterceptor 添加自定义的拦截器

加拦截器的顺序可以知道 Interceptors 和 networkInterceptors 刚好一个在 RetryAndFollowUpInterceptor 的前面，一个在后面

责任链调用图可以分析出来，假如一个请求在 RetryAndFollowUpInterceptor 这个拦截器内部重试或者重定向了 N 次，那么其内部嵌套的所有拦截器也会被调用N次，同样 networkInterceptors 自定义的拦截器也会被调用 N 次。而相对的 Interceptors 则一个请求只会调用一次，所以在OkHttp的内部也将其称之为 Application Interceptor。

4.责任链模式

https://juejin.cn/post/6844903792073261063#heading-12

将处理者和请求者进行解耦

多个对象都有机会处理请求，将这些对象连成一个链，将请求沿着这条链传递。

在请求到达时，拦截器会做一些处理（比如添加参数等），然后传递给下一个拦截器进行处理。

5.缓存怎么处理

https://juejin.cn/post/6844903552339410958#heading-4

使用OkHttp的缓存

定义一个网络拦截器

Http协议 缓存的控制是通过首部的Cache-Control来控制

only-if-cache: 表示直接获取缓存数据，若没有数据返回，则返回504

有网络时访问服务器

无网络时返回缓存数据

1.自定义Interceptor,重写intercept设置header *2.OkHttpClient .cache// 设置缓存路径和缓存容量 *addNetworkInterceptor设置自定义缓存

不使用OkHttp的缓存

if (NetworkUtil.isConnected(mContext)) { response = chain.proceed(newRequest); saveCacheData(response); // 保存缓存数据 } else { // 不执行chain.proceed会打断责任链，即后面的拦截器不会被执行 response = getCacheData(chain.request().url()); // 获取缓存数据 }

6.Okhttp连接池

连接池是为了解决频繁的进行建立Sokcet连接（TCP三次握手）和断开Socket（TCP四次分手）

socket复用有何标准

get

1.http协议

1.在http 1.x协议下，所有的请求的都是顺序的，正在写入数据的socket无法被另一个请求复用 2.http2.0协议使用了多路复用技术，允许同一个socket在同一个时候写入多个流数据

http1.x协议下当前socket没有其他流正在读写时可以复用，否则不行，http2.0对流数量没有限制。

2.域名和http和ssl协议配置需要匹配

put

在连接池中找连接的时候会对比连接池中相同host的连接。

如果在连接池中找不到连接的话，会创建连接，创建完后会存储到连接池中。

27.2.Glide

2.1 Glide怎么绑定生命周期

https://juejin.cn/post/6844903647877267463#heading-1

Glide.with(Activity activity)的方式传入页面引用

1.创建无UI的Fragment，并绑定到当前activity

2.builder模式创建RequestManager， 将fragment的lifecycle传入，这样Fragment和RequestManager就建立了联系

3.RequestManager实现LifecycleListener 是一个接口,回调中处理请求

2.1 Glide缓存机制内存缓存，磁盘缓存

https://juejin.cn/post/6844904002551808013#comment

Glide的缓存机制，主要分为2种缓存，一种是内存缓存，一种是磁盘缓存。

之所以使用内存缓存的原因是：防止应用重复将图片读入到内存，造成内存资源浪费。

之所以使用磁盘缓存的原因是：防止应用重复的从网络或者其他地方下载和读取数据

具体来讲，缓存分为加载和存储：

内存缓存分为弱引用和lru缓存

弱引用是缓存正在使用的图片，避免内存泄漏

将缓存图片的时候，写入顺序

弱引用缓存-》Lru算法缓存-》磁盘缓存中

当加载一张图片的时候，获取顺序

弱引用缓存-》Lru算法缓存-》磁盘缓存

2.3关于LruCache

LruCache 内部用LinkHashMap存取数据

LinkedHashMap继承于HashMap，它使用了一个双向链表来存储Map中的Entry顺序关系，这种顺序有两种，一种是LRU顺序，一种是插入顺序

LruCache中将LinkedHashMap的顺序设置为LRU顺序来实现LRU缓存

每次调用get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。

调用put插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除。

2.4 Glide与Picasso的区别

27.3.LruCache的原理是什么？

```
LruCache的实现需要两个数据结构：双向链表和哈希表。
双向链表用于记录元素被塞进cache的顺序，然后淘汰最久未使用的元素。
哈希表用于直接记录元素的位置，即用O(1)的时间复杂度拿到链表的元素。

get的操作逻辑：根据传入的key(图片url的MD5值)去哈希表里拿到对应的元素，如果元素存在，就把元素挪到链表的尾部。
put的操作逻辑：首先判断key是否在哈希表里面，如果在的话就去更新值，并把元素挪到链表的尾部。
            如果不在哈希表里，说明是一个新的元素。这时候需要去判断此时cache的容量了，
            如果超过了最大的容量，就淘汰链表头部的元素，再将新的元素插入链表的尾部，如果没有超过最大容量，
            直接在链表尾部追加新的元素。
```

为啥要用linkedHashMap的数据结构？？ HashMap是无序的，当我们希望有顺序地去存储key-value时，就需要使用LinkedHashMap了.

27.4.Glide如何绑定生命周期

27.5. Retrofit

1.通过建造者模式构建一个Retrofit实例

2.通过Retrofit对象的create方法返回一个Service的动态代理对象

3.调用service的方法的时候解析接口注解

4 .调用Okhttp的网络请求方法,通过 回调执行器 切换线程（子线程 ->>主线程）

动态代理

运行时创建的代理类，在委托类的方法前后去做一些事情

在运行过程中，会在虚拟机内部创建一个Proxy的类。通过实现InvocationHandler的接口，来代理委托类的函数。

使用动态代理来对接口中的注释进行解析，解析后完成OkHttp的参数构建。

优点

代理类原始类脱离联系,在原始类和接口未知的时候 就确定代理类的行为

27.6 LeakCanary

1.四种引用

JVM通过垃圾回收器对这四种引用做不同的处理

1.强引用

指向的对象任何时候都不会被回收,垃圾回收器宁愿抛出OOM也不会对该对象进行回收

2.软引用

但是如果内存空间不足 ，才回去回收软引用中的对象.

3.弱引用

当发生垃圾回收时，不管当前内存是否足够，都会将弱引用关联的对象进行回收。

4.虚引用

虚引用必须和引用队列一同使用

ReferenceQueue

如果软/弱/虚引用中的对象被回收，那么软/弱/虚引用就会被 JVM加入关联的引用队列ReferenceQueue中

是说我们可以通过监控引用队列来判断Reference引用的对象是否被回收，从而执行相应的方法。

1.了Application类提供的registerActivityLifecycleCallback(ActivityLifecycleCallbacks callback)方法来注册ActivityLifecycleCallbacks回调，这样就能对当前应用程序中所有的Activity的生命周期事件进行集中处理，当监听到Activity 或 Fragment onDestroy() 时，把他们放到一个弱引用WeakReference 中。

2.把弱引用WeakReference 关联到一个引用队列ReferenceQueue。（如果弱引用关联的对象被回收，则会把这个弱引用加入到ReferenceQueue中）。

3.延时5秒检测ReferenceQueue中是否存在当前弱引用对象。

4.如果检测不到说明可能发生泄露，通过gcTrigger.runGc()手动掉用GC。 遍历ReferenceQueue中所有的记录，当未回收对象个数大于5个时,dump heap获取内存快照hprof文件。

6.使用Shark解析hprof文件,Hprof.open()把heapDumpFile转换成Hprof对象，

7.根据heap中的对象关系图HprofHeapGraph获取泄露对象的objectIds

8.找出内存泄漏对象到GC roots的最短路径

9.输出分析结果展示到页面。