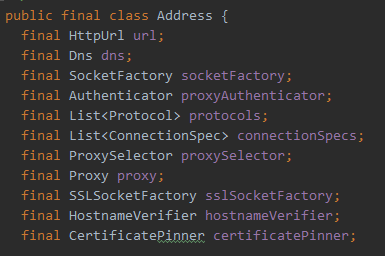
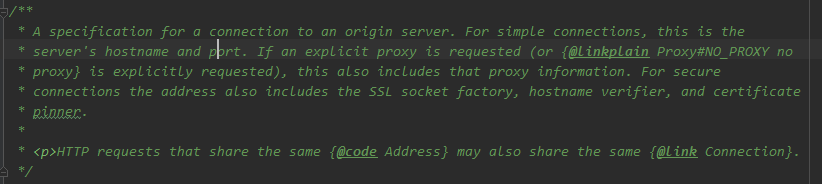
2016/7/17

|  |  |  |
| --- | --- | --- |
| 类名 |  |  |
| Address |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Address类，看起来只是用来保存连接用的





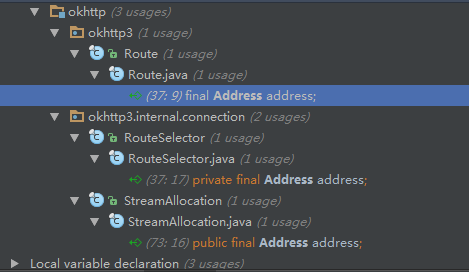
Address指定了到服务器的连接是怎么样的一条连接，类似于一个Connection Bean

对于简单的连接来说只需要主机和端口，对于使用代理的连接则会引入代理的信息，对于SSL加密的连接就会引入SSL工厂信息。

SocketFactory SSLSocketFactory HostnameVerfier Proxy和ProxySelector都是java或javax的库函数。

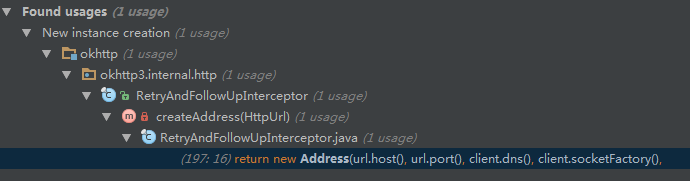
当然如何判断2个Adress是相同的，只能通过确认每个成员函数都相同来确保equal

通过搜索发现



有三个地方用到了这个Address

实际只有一个地方有用到



是RetryAndFollowUpInterceptor.java的createAddress函数

Address大概就是这样。

下面来看看HttpUrl.java这个类是做什么的

开头的注释简要的介绍了如何使用HttpUrl及相基本的HTTP知识

举例:

**\* HttpUrl url = new HttpUrl.Builder()**

**\* .scheme("https")**

**\* .host("www.google.com")**

**\* .addPathSegment("search")**

**\* .addQueryParameter("q", "polar bears")**

**\* .build();**

**\* System.out.println(url);**

打印出

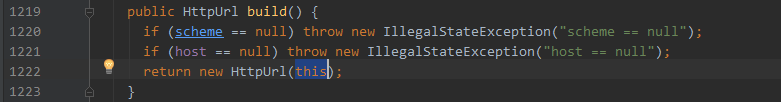
**\* https://www.google.com/search?q=polar%20bears**

可以看到HttpUrl可以拼接协议，用记名/密码，主机，路径以及查询参数

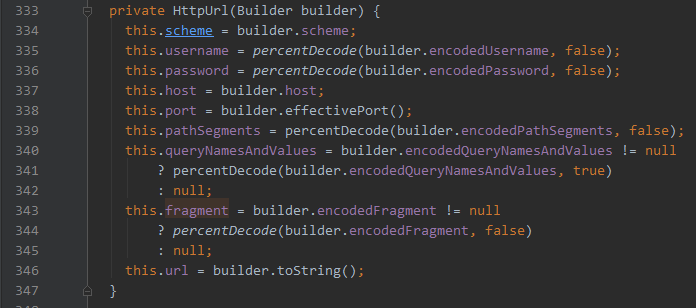


可以看到这个Builder内部类有点长

但是我们只要关注它的build()函数，返回是一个HttpUrl



再细看下HttpUrl就有通过Builder来构建的函数，这样就产生了一个HttpUrl实例，这就叫装饰者吧



另外一个比较重要的就是newBuilder()函数，可以让HttpUrl从一个链接生成Builer

\* HttpUrl url = HttpUrl.parse("http://who-let-the-dogs.out").newBuilder()

\* .addPathSegment("\_Who?\_")

\* .query("\_Who?\_")

\* .fragment("\_Who?\_")

\* .build();

\* System.out.println(url);

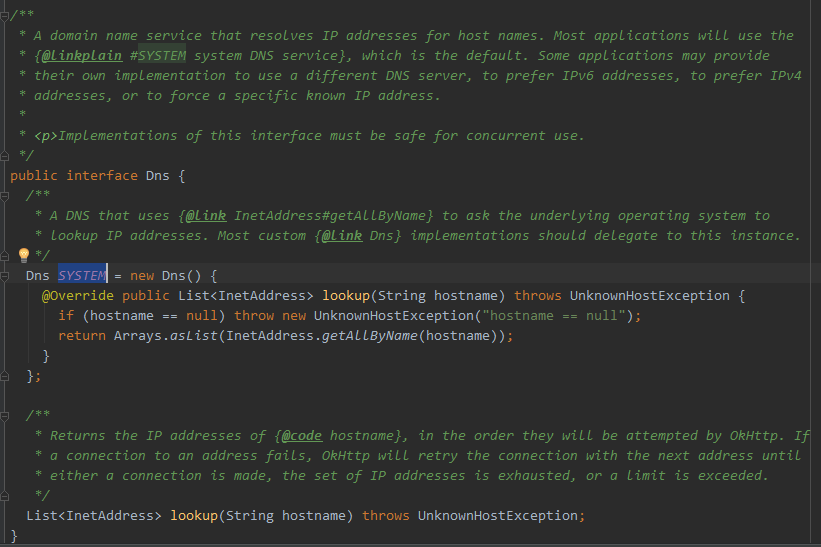
\*

\* http://who-let-the-dogs.out/\_Who%3F\_?\_Who?\_#\_Who?\_

大概就是parse生成一个HttpUrl对象，将解析后URL保存到这个对象对，再由这个HttpUrl的newBuiler生成Builer，最后再由Builer的build生成回HttpUrl对象。

再接下来挑个软柿子来捏捏

那就是Dns.java吧！加注释不到50行，但这里有个小技巧



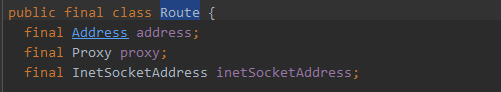
比如在OkHttpClient.java初始化时就可以直接通过SYSTEM直接new一个匿名类出来，方便。



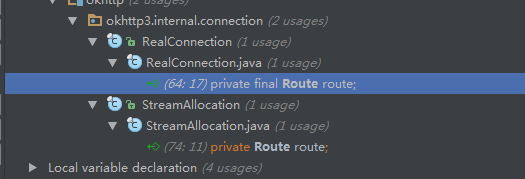
DNS解析用的就是系统自带的解析方式，解析完后直接放到List里面。

下一个就看看Route吧

这个不到100行，其实也是个Bean而已，用来存储



这三个变量，Address应该知道是用来存储连接的，有些成员变量也和Address里的类似，让我们看一下都在哪里用了



是RealConnection.java和StreamAllocation.java

简单看下RealConnection.java，这里初始化了自身的route



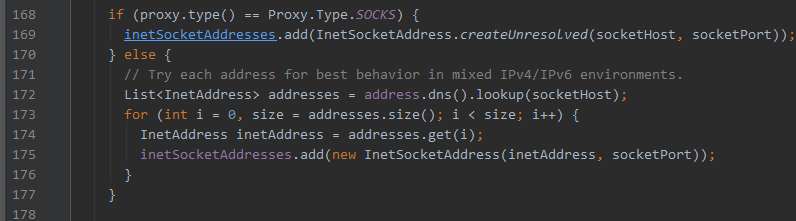
但是是谁赋值了？很好，只有一处使用到了RealConnection.java的route赋值



那就再看下StreamAllocation.java

我们突然发现这个类是一个很大的类，很核心

但是和Route有关的可能应该是RouteSelector.java，可以直接推测这个就是用来选择如何连接服务器的，通过代理，还是DNS解析出来的IP列表。看一个小片断



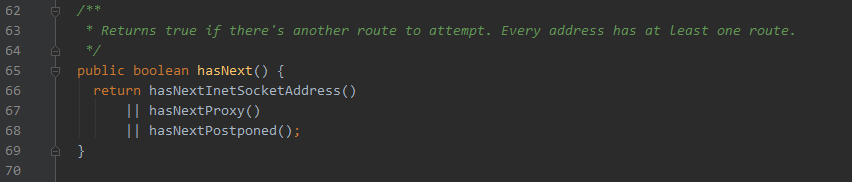
这里就有使用到之前DNS接口lookup

几个reset函数对Proxy列表和ip地址列表进行了初始化





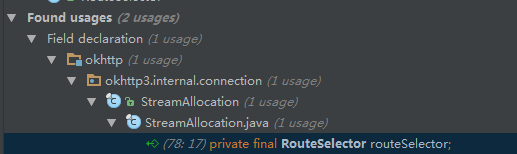
当然RouteSelector.java是用来选择连接方式的话，使用方式就是



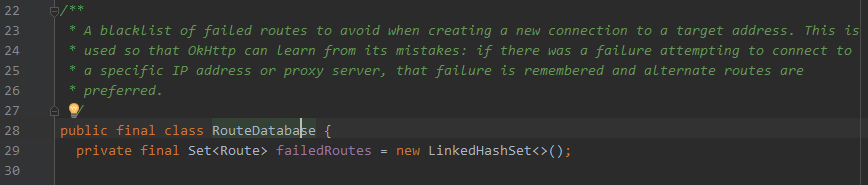


就是这2个函数

我们发现，只有StreamAllocation.java有使用到了RouteSelector.java那就从这里出发看下吧！



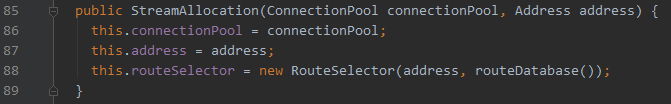
还没等我们开始，马上就有一个RouteDatabase.java的类，这个类是做什么的？写的比较清楚了



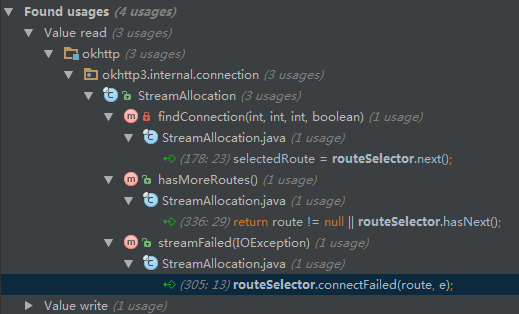
一个集合就是用来保存失败的Route.

再返回来看一下RotueSelector.java里面也是有RouteDatabase的

既然这样看下StreamAllocation.java吧



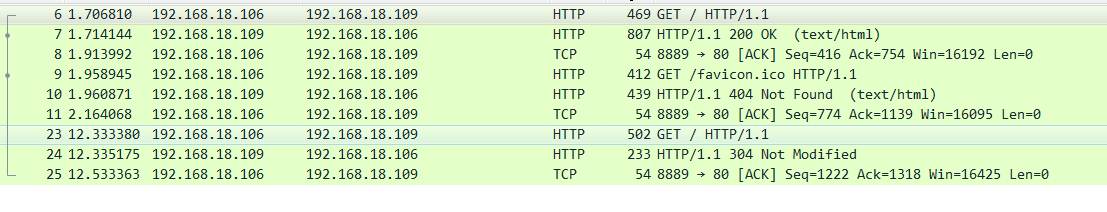
现在还对这个类还一无所知，还是看下routeSelector是做了哪些事吧



很简单，第一个是查找下一个Route，第二个是看还有没有其它的Route，最后一个是连接不通时设置Route的状态，加到Set里面。

现在还没有说到SSL加密，Cookie的使用，线程池的使用，多文件上传，Response的读取。现在先从最基本的同步请求开始：

当然还是要先捉下HTTP包



这个是chrome访问同一个页面时全新的第一次和第二次带缓存的不同，第二次返回的是Status 304

让我们先看下这个HTTP的GET发了哪些给服务器



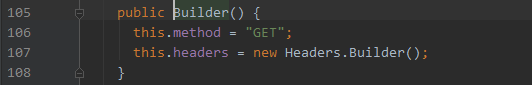
发现HTTP请求不过是一行一行的数据而已以\r\n(0x0d,0x0a)结尾。

现开始分析下Request是如何到Response回来的

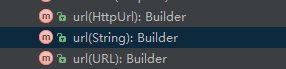


实际使用很简单，基本不必多说，因为是同步的也没有线程池的使用

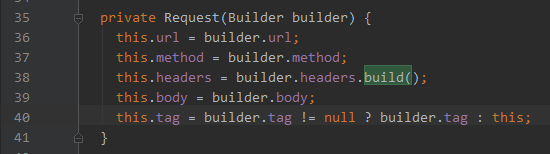
Builder之前也是见过



Builder生成method及headers的头



url有三个参数，实际调用的是url(HttpUrl)，url(Stinrg)通过HttpUrl.parse生成HttpUrl对象，保存到Builer里，最后再由Builder.build()从this生成Request.



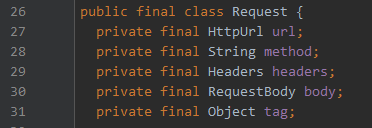
这里有个headers，跳到Headers.java那儿看



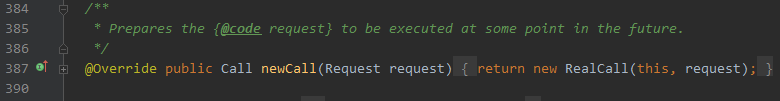
可以发现name和value只在放在String[] 数组里，偶数是name，奇数是value,类的里其它函数只是把这个namesAndValues数组转换成不可变的数据类型。而builder.headers.build()只是将Header.Builder的List<String>交给Headers.java去生成之前数组而已。

当然Header.Builder也有自己的一些方法，不过都是添加，删除检查的操作，不是核心。

好吧，到这里Request就有基本的信息了



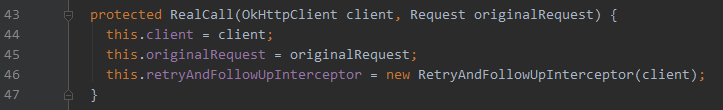
可以用来发送请求了！

现在就要到clinet.newCall()看下这个函数做了什么事  


newCall生成一个RealCall，并把client的本身和request传进去

继续

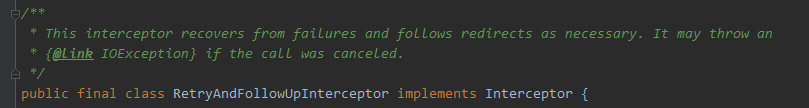
现在到了一个全新的类RealCall.java



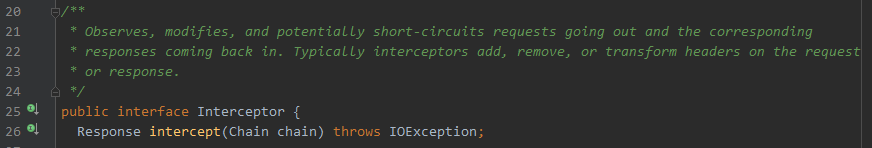
没什么好说的，前面2个，但是第三个参数



必须要展开看看



原来这个拦截器是用来从错误中恢复或者重定向用的，不过现在还不知道他是怎么实现的



实现的接口中要实现intercept()这个函数，看起来是要拦截某个Chain 链。

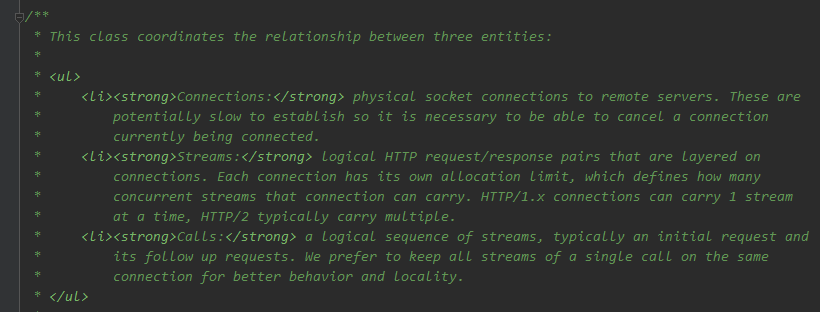
那既然都见到了RetryAndFollowUpInterceptor,那先大概浏览下这个类实现的intercept做了什么



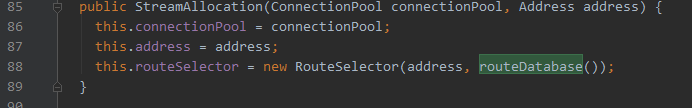
108行，很可能就是我们之前传入的Request



这个StreamAllocation是什么？之前也遇到过但是没细看，在这儿里就要深入了解一下了，下面是这个类的简要说明



1. 和Connection有关，且是实际和Server打交道的Socket
2. 和Stream有关，但是是基于Connection之上，虚拟出来的的stream
3. 和Call有关，按顺序使用发送request，这个不是特别明白

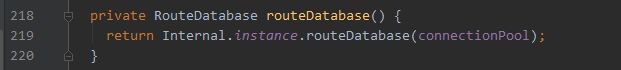


createAddress自然是返回Address，说过他更像是一个Connection Bean而不是“地址”的概念。

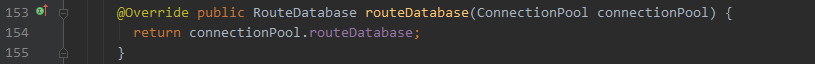
而connectionPool想必就是说明中提到的实际的socketPool吧

但是那个routeDatabase()是什么东西 ？我们知道RouteDatabase是用来保存失败不可用的Route

routeDatabase实际调用的是Internal那个routeDatabase



而这个routeDatabase又是调用的是OkHttpClinet中实例化的instance的接口



看起来只是返回connectionPool的routeDatabase而已。

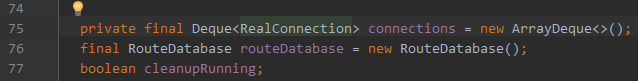
可以看到new的这个StreamAllocation用的是client的connectionPool()



而很奇怪的一点是routeDatabase返回的是本对象的connectionPool.routeDatabase，这样不是多此一举吗？

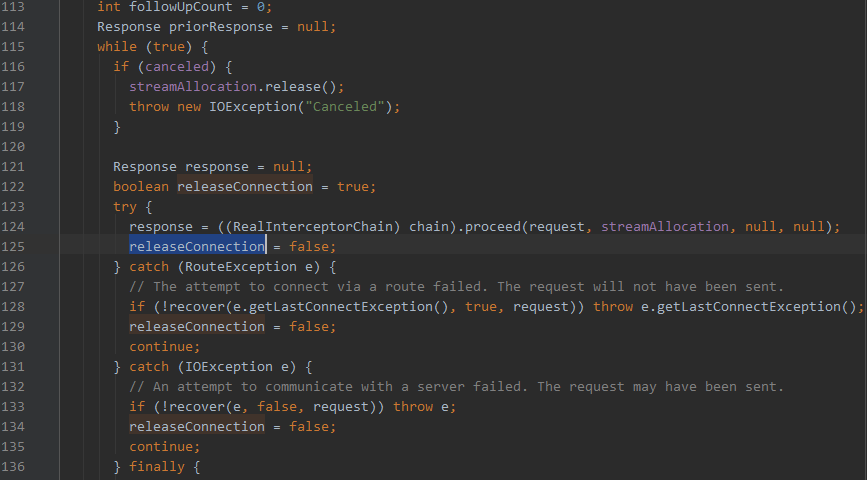
不过，这是初始化完了，但是很重要的ConnectionPool还没有说明

ConnectionPool.java里面放的是RealConnection双向队列



这儿ConnectionPool提供了基本的get,put，查看空闲连接的大小等方法，但是实际上如何运用我们还不是很清楚，不过我们还是要接着往下走

我们看到这儿有个while，不错，这个和RetryAndFollowUpInterceptor这个名字还是对得上的，while的话就可以不断地尝试



一开始就把request通过proceed往下传，如果有RouteException的话就会调用 recover恢复

2016-07-18

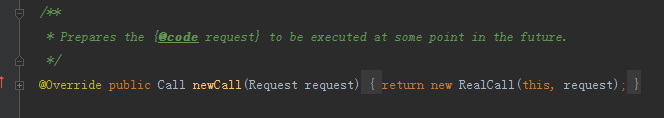
现在还很模糊的是StreamAllocation.java这个是做什么用的，怎么用的，它与其它类间是如何交互的，我们不清楚，另外还有之前的Chain又是什么？

我们先回到最原始的代码

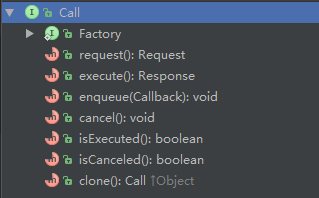


Response一定就是我们发出request的响应，现在的关键就在execute里，如果execute明白了，就明白了同步调用是什么样的，API很简单让我们跳进去看下。

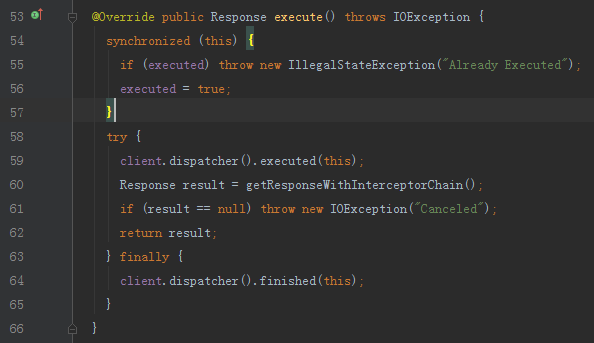
Client的newCall实际上是调用了RealCall



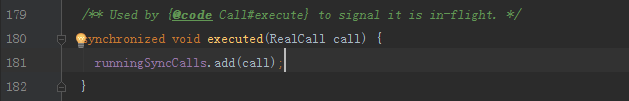
RealCall实现了Call接口，就有了以下



RealCall重载了execute(),先设置这个RealCall的executed为true后将RealCall放到dispatcher中执行

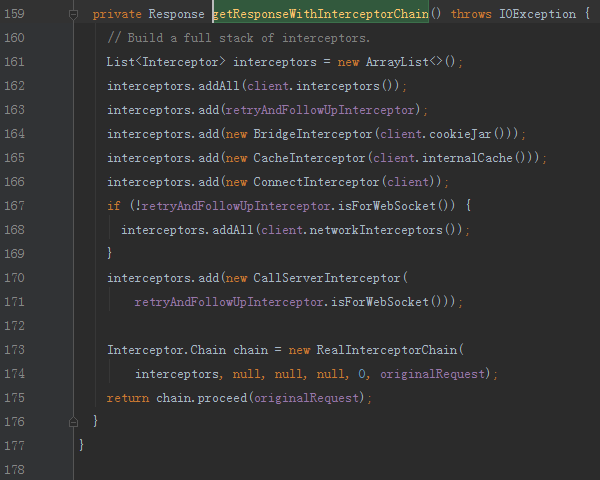


但是我们发现dispatcher只是将RealCall放到deque中并没有做什么。**不知道是什么原因。**



继续往下

getResponseWithInterceptorChain()跳进去看下

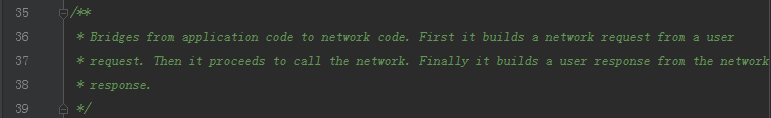


这里我们看到了retryAndFollowUpInterceptor，不算client的拦截器的话，他就是第一个，应该也是第一个，因为要起到重连，路由的作用。

到这儿，就应该是责任者模式了，定义一条Chain，通过Chain上的拦截器对request进行处理，看来一个同步请求就需要这么多个拦截器来处理啊

retryAndFollowUpInterceptor我们知道大概作用了，那一个一个往下看！

BridgeInterceptor.java



说得大概知道是什么意思，他们处理request并往chain传递，并将Chain传回来的response再进行加工，最后返回给上一级拦截器。

这个Chain有点类这个形状

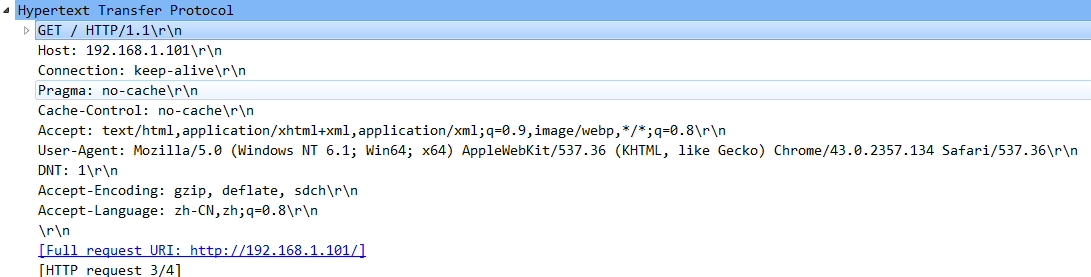
A->B->C->D Send Request and Receive Response ->C->B->A

A发送请求给B，会处理B得会回来的响应，一层嵌套一层

我们看到BridgeInterceptor对request的处理就是处理下Http Header



完成的结果应该是添加下面的Header参数



处理完成后



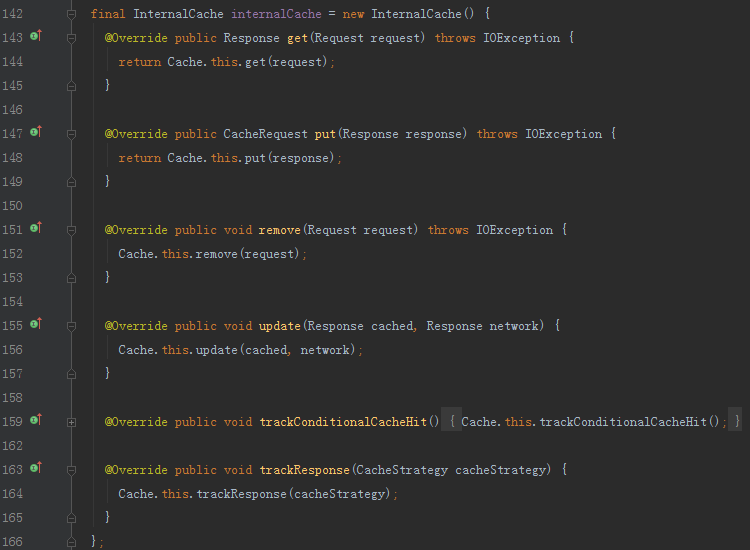
直接将新生成的request送入下一个interceptor

不过这个名字叫Bridge感觉有点奇怪啊

CacheInterceptor.java接下来就是这个类

CacheInterceptor这个类实现的功能就是根据配置和缓存的设置来看是需要返回失败的Resonpse或是Cache Response或者需要重新请求网络时，往下一级Chain提交。当返回结果OK时，会重新更新Cache.

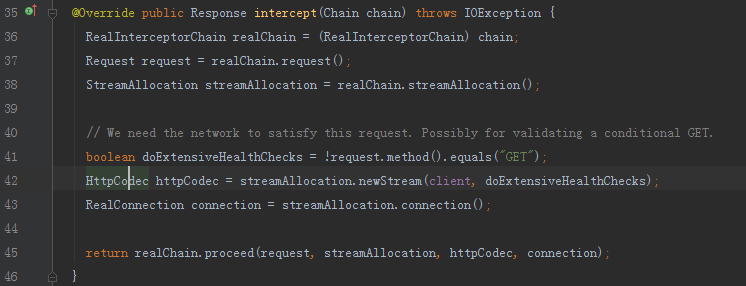
实际的internalCache在Cache.java中实现 了



因为是比较独立的一个类，暂不详细展开

继续看下Chain的下一个！

下一个是ConnectInterceptor.java，好爽，只有这么几行，但是这几行绝对不简单

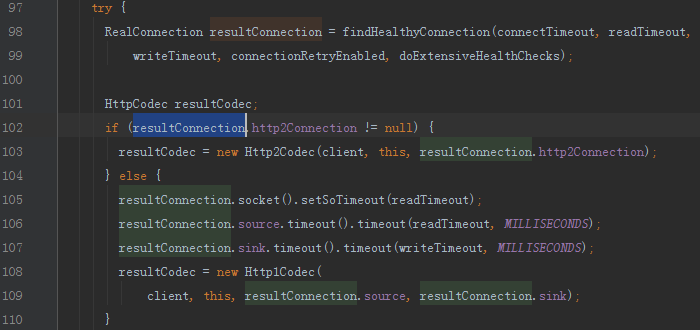


到目前为止proceed里的参数在这里全部集合完毕！

要说request，connection还眼熟一点,httpCodec是Http编码器？什么鬼！

还有newStream做了什么事？

先看newStream吧！在newStram中resultCode需要一个输入输出流，也就是source和sink，而这个source和sink就必须 在findHealthyConnection中去查找



findHealthyConnection实际调用的是StreamAllocaion.java的findConnection

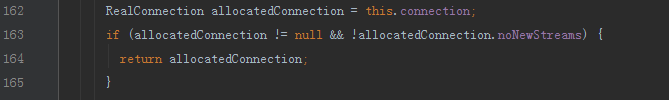
它不仅可以查找现在connectionPool是否有可用的connection，还能在没有合适的 connection时创建RealConnection，并放到ConnectionPool里供下次调用。

这里可以有需要想清楚一点

一个socket可以由多个Stream复用，比如socket是到服务器的一个连接，但是多个Request可以访问同一个服务器的多个资源，这里就是复用同一个socket，多个Stream.

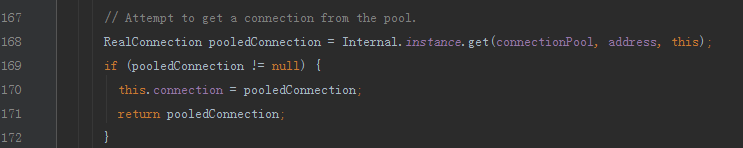
这个想法是合理的，但是要看下代码的实现 。

在findConnection中，首先查看上次发分配的Connection能不能用



如果上次分配的Connection不为空且可以用于新的Stream那么直接返回。这样做估计是缓存吧，因为okHttpClient极有可能一直重复访问同一个资源,这样就不需要一直重新从pool里取或重新分配。那么时候会重用这个Connection呢，从StreamAllocaiton的生命周期来看，只有本次Request才有效，但是生次生效的Connection可以放到Pool里下次再用，所以缓存起来的RealConnection只有是重连或者跳转时才用得上，因此也不需要检查Address，这个不错。

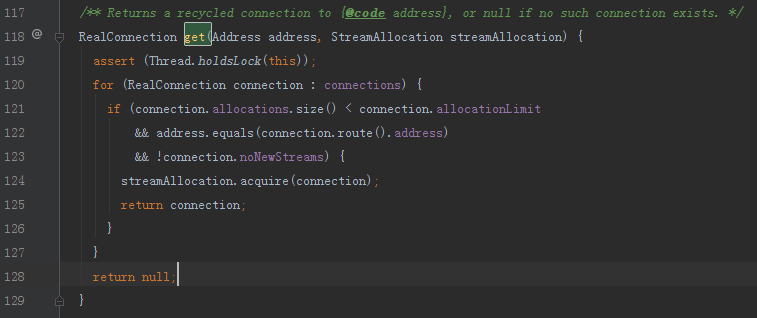
如果上次分配的Connection不合适则从pool里找一个合适的。第一次需要重新分配，并加入到pool中，其它Request就可以复用这个RealConnection了。



this.connection = pooledConnection; 这句就是用来缓存这次分配的。

另外这个其实调用的是pool本身的函数,传入的参数是Address和这次Request的StreamAllocation

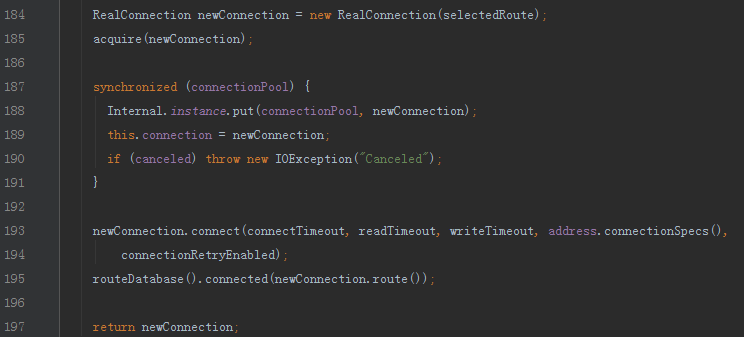
ConnectionPool.java的get函数如下



看上去挺正常一个循环，等等，那个acquire是什么鬼？像是用来释放连接用的。

暂时管不着，那就是找到一个Address匹配且可以接受新的Stream的链接，返回！

当然如果在ConnectionPool没有找到一个合适的连接就需要重新new一个。



New完后需要将新的Connection放到pool里面，更新当前的connection缓存

那个connect对newConnection做了什么？

在RealConnection.java里主要从

connect->buildConnection->ConnectSocket->establishProtocal，当然

最主要的是保存了



这样一个输入输出流就创建好了

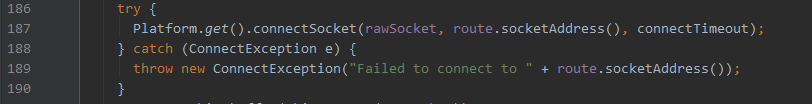
当然这里还有个疑问就是那个route是怎么一回事？

应该有个地方会再更新noNewStream的，要不然路由不通时会怎么做？这个现在还不到，因为现在只是创建了Socket但还没有去进行连接。今天就到这里吧！

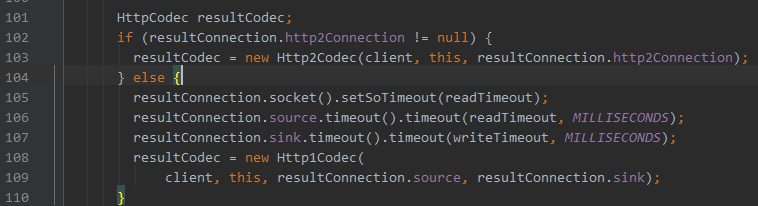
其实己经有去连接了，不过这个socket是非阻塞的，所以有一个超时

这个connect就在

RealConnection中的connectSocket

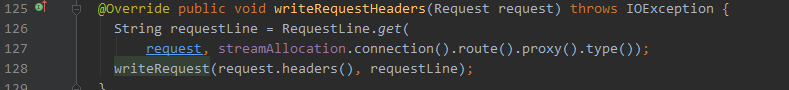


让我们先回到StreamAllocation.java的newStream中

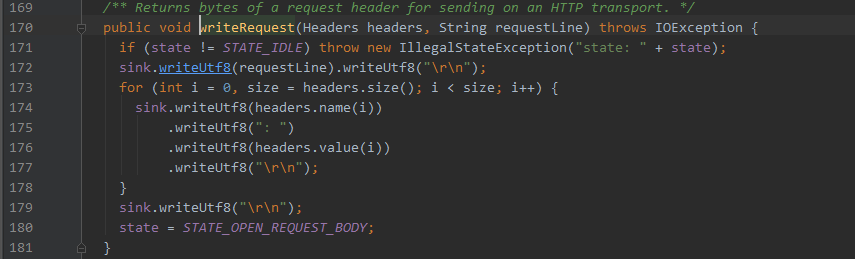


最后返回的是HttpCodec，不过这里可能是Http1Codec也有可能是Http2Codec

暂时先看Http1Codec吧，浏览下，发现一个writeRequestHeaders

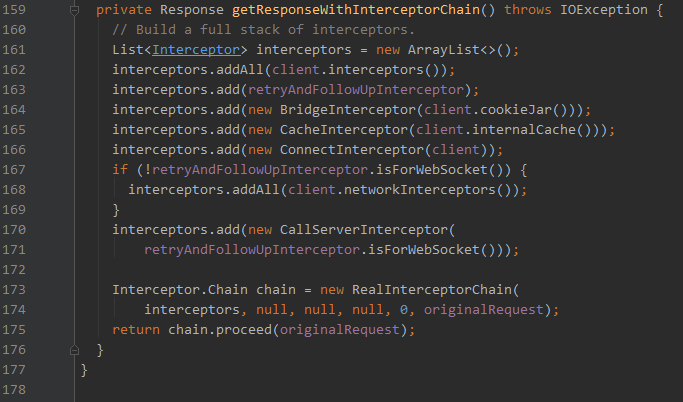


这个应该就是写Header吧，看他怎么写的



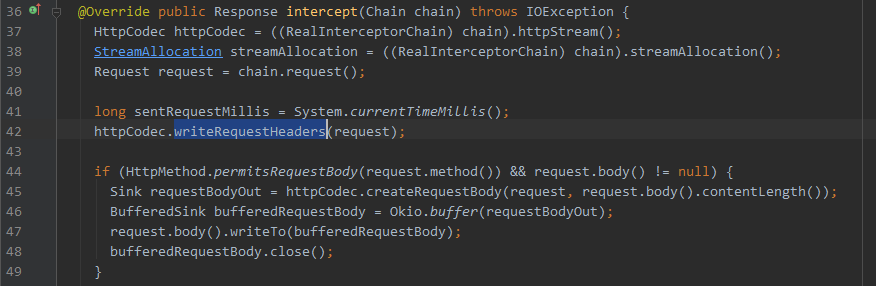
至此也明白了是通过刚才的RealConnection创建的sink和source 来写数据的。

Codec可以读写Request Header，写Request Body，读Response Body等一系列操作。现在万事俱备，回到



这里，看下networkIntercepteors我们在一开始都没有设置，那直接 跳到最后一个拦截器吧！

爽，这个拦截器也不长！

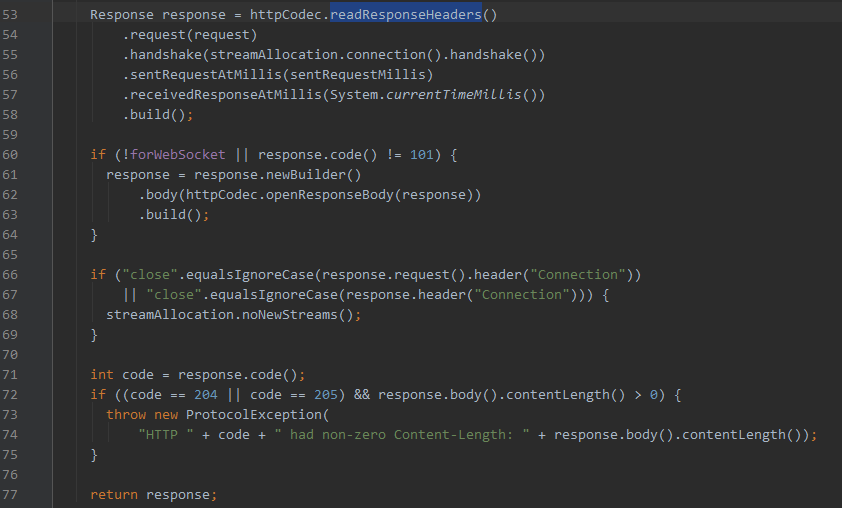


一进来就调用了我们刚刚提过的writeRequestHeader写入header

接下来就判断是不是需要传输request body，比如post,get等

如果需要写就作一个缓冲的OKio进行body数据写

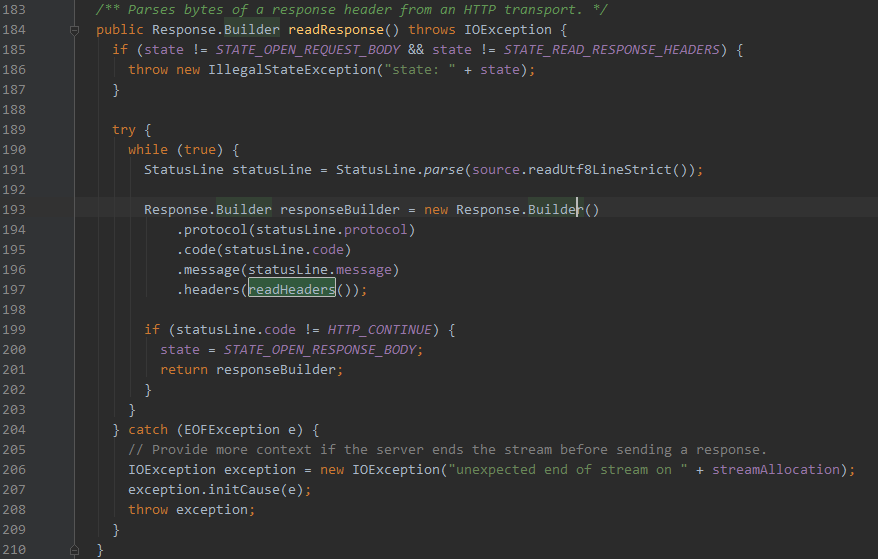
Flush数据，没什么好说的，接下来就是要读取Response回来



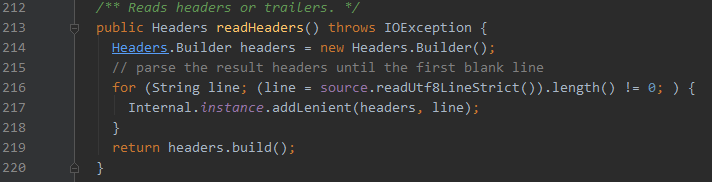
readResponse



一个循环，不断地读取source的回复



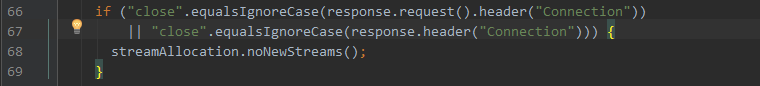
其实这儿有2个循环，另一个循环在readHeaders()中，看起来是一个Response可以分为多个Response回复



如果只有一个的话，那就通过readHeaders全部处理掉了

我们在这里跳过WebSocket

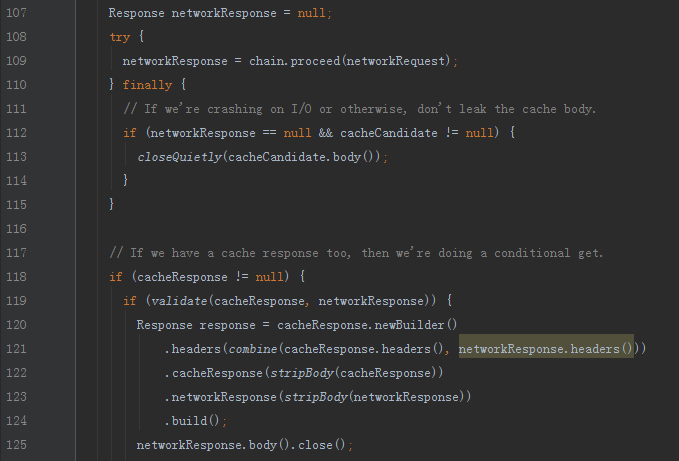
如何request中或 response中Connection有包含close则设置streamAllocation的Streams状态



好了到了这里就需要往回回溯了！

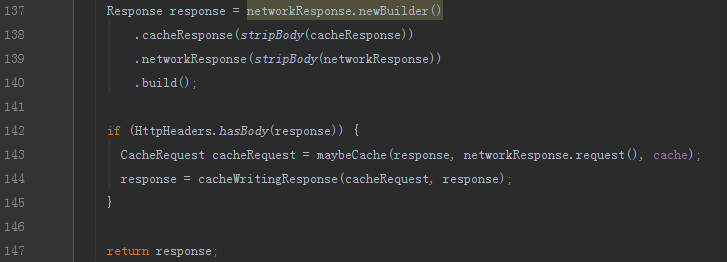
20160719

在CallServerInterceptor.java后，直接就跳回了CacheInterceptor.java

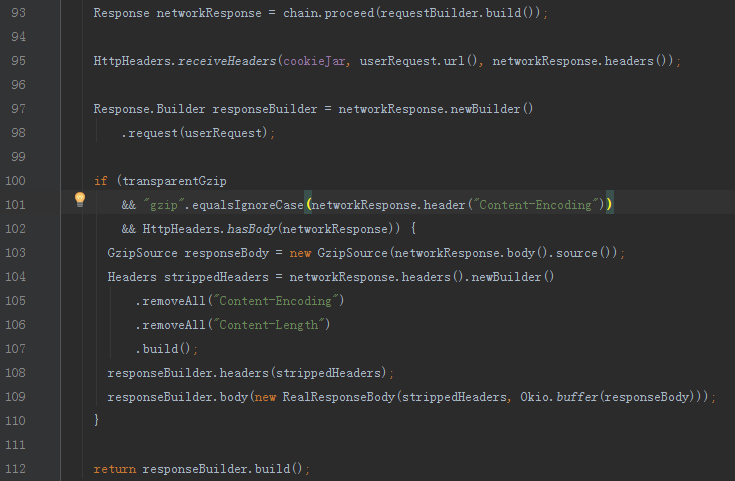


可以看到在得到Response后，还需要通过和CacheResponse进行比较，如果当前的Response比缓存的Response还要旧那就使用cacheResponse进行build真正的Response.

如果没有cache或者缓存太旧，那么就使用回来的Response，并且如果有带body就更新缓存。

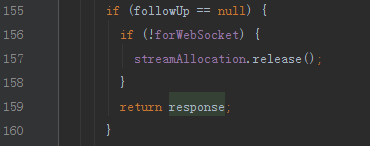


Return response后回到BridgeInterceptor.java



首先如果有cookie就先保存cookie到cookieJar,略去中间的转换，我们就到了responseBuiler.boyd(….)这里，这里使用strippedHeadder和一个Okio.buffer来生成response的body

如果没有异常，我们的会回到RetryAndFollowUpInterceptor，检查不需要跳转的话，就返回了。



好的到这里，基本就回到了

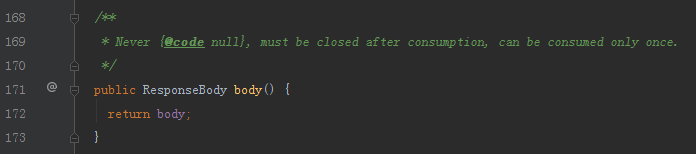


Execute()返回response后，调用了body().string()，这个后我们刚才

BridegIntercepotr.java里有看到

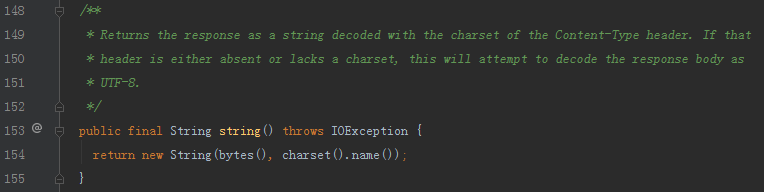
跳过去看下！

Response里的body()只是简单返回设置的body而已

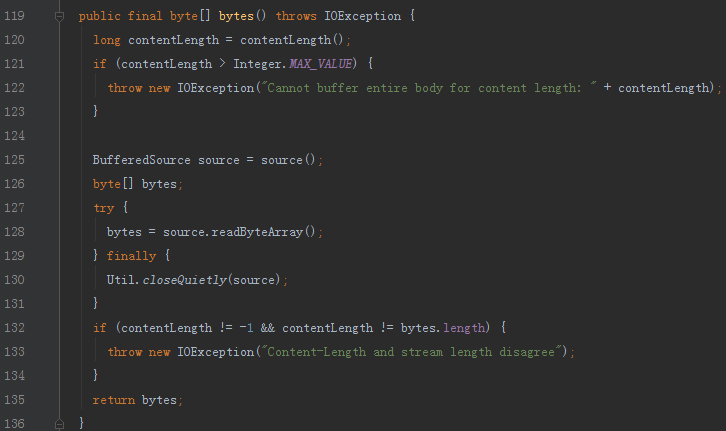


我们实际调用的是body()的string函数

看下ResponseBody.java，这是一个抽象类，实际类是RealResponseBody.java



bytes()函数从source里将数据完全读出，并通过String转换成相应的字符集，方便！



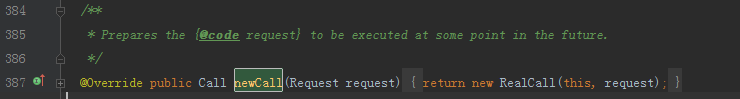
这样，一次同步的调用的大概流程知道了，但是对于细节的处理可能还不到位。像HttpUrl，Route，Sink/Source，FollowUp, Connection管理都不是特别清楚，甚至异步我们也还没讲，OkHttp这张大网才刚刚有点脉络！

现在看下异步的时候是怎么做到的吧！

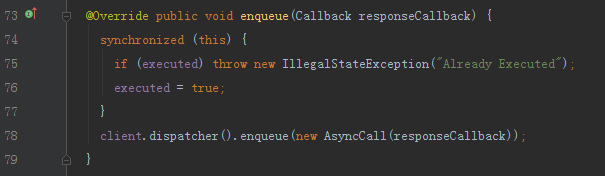
https://github.com/square/okhttp/wiki/Recipes



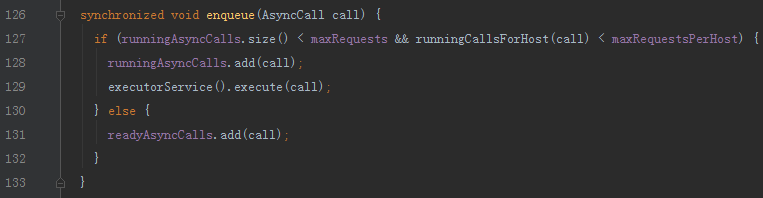
看起来只有newCall之后使用了enqueue，并注册了一个匿名回调对象



还是调用的RealCall,不过这次是enqueue，只是将responseCallback放到AsyncCall中再放到dispatcher中

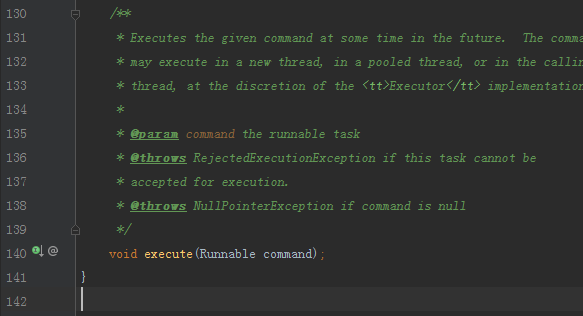


Enqueu后放到runningAsyncCall中，如果有可以执行的线程

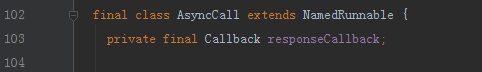


这里做了2个限制，一个是总的异步调用不能超过maxRequests，同一个主机的call也不能超过maxRequestPerHost.

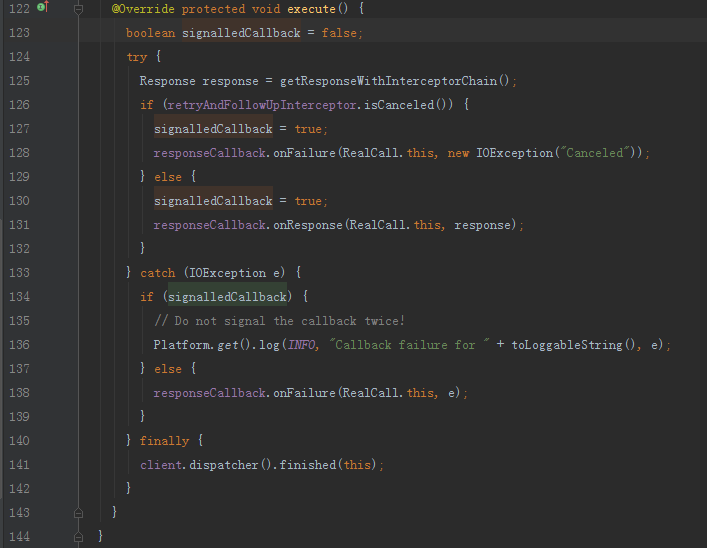
从线程池里取出一个线程来执行一个Runnable命令(Executor.java)k



很明显，AsyncCall实现了这个接口，最终调用的就是AsyncCall的execute函数



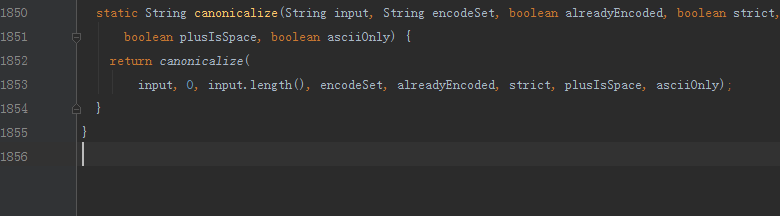
看一下这个execute吧！



竟然也是使用同一个getResponseWithInterceptorChain，好吧，那就省事多了！

这个多了一个Cancel，好吧，简单！

虽然现在大概明白了OkHttp处理Http请求的方式，但是对于真的要写出一个好用的Http请求库真不是件容易的事，请看 HttpUrl.java



1856行，除掉注释一半，也有近1000行！而这只是为处理URL而已，一个好用的库只在实现基本功能是不行的，细节，好用的细节才是让一个库发扬光大的原因所在！

所以，如何实现OkHttp in C++11就是个很大的挑战！因为光字符处理可能就很头痛了。

好吧，先让我们回过头来，再一个一个类地过一遍！

既然提到了HttpUrl那就从这里开始讲吧，这应该是HTTP最最最最最基本的知识了吧！

算了就当是一个翻译吧！翻译一下这个代码的注释！以下内容挺长的，可以略过，不过简直就是一篇小文章了。

\* <h3>What's in a URL?</h3>

\*

\* A URL has several components.

\*

\* <h4>Scheme</h4>

\*

\* <p>Sometimes referred to as <i>protocol</i>, A URL's scheme describes what mechanism should be

\* used to retrieve the resource. Although URLs have many schemes ({@code mailto}, {@code file},

\* {@code ftp}), this class only supports {@code http} and {@code https}. Use {@link URI

\* java.net.URI} for URLs with arbitrary schemes.

\*

\* <h4>Username and Password</h4>

\*

\* <p>Username and password are either present, or the empty string {@code ""} if absent. This class

\* offers no mechanism to differentiate empty from absent. Neither of these components are popular

\* in practice. Typically HTTP applications use other mechanisms for user identification and

\* authentication.

\*

\* <h4>Host</h4>

\*

\* <p>The host identifies the webserver that serves the URL's resource. It is either a hostname like

\* {@code square.com} or {@code localhost}, an IPv4 address like {@code 192.168.0.1}, or an IPv6

\* address like {@code ::1}.

\*

\* <p>Usually a webserver is reachable with multiple identifiers: its IP addresses, registered

\* domain names, and even {@code localhost} when connecting from the server itself. Each of a

\* webserver's names is a distinct URL and they are not interchangeable. For example, even if {@code

\* http://square.github.io/dagger} and {@code http://google.github.io/dagger} are served by the same

\* IP address, the two URLs identify different resources.

\*

\* <h4>Port</h4>

\*

\* <p>The port used to connect to the webserver. By default this is 80 for HTTP and 443 for HTTPS.

\* This class never returns -1 for the port: if no port is explicitly specified in the URL then the

\* scheme's default is used.

\*

\* <h4>Path</h4>

\*

\* <p>The path identifies a specific resource on the host. Paths have a hierarchical structure like

\* "/square/okhttp/issues/1486" and decompose into a list of segments like ["square", "okhttp",

\* "issues", "1486"].

\*

\* <p>This class offers methods to compose and decompose paths by segment. It composes each path

\* from a list of segments by alternating between "/" and the encoded segment. For example the

\* segments ["a", "b"] build "/a/b" and the segments ["a", "b", ""] build "/a/b/".

\*

\* <p>If a path's last segment is the empty string then the path ends with "/". This class always

\* builds non-empty paths: if the path is omitted it defaults to "/". The default path's segment

\* list is a single empty string: [""].

\*

\* <h4>Query</h4>

\*

\* <p>The query is optional: it can be null, empty, or non-empty. For many HTTP URLs the query

\* string is subdivided into a collection of name-value parameters. This class offers methods to set

\* the query as the single string, or as individual name-value parameters. With name-value

\* parameters the values are optional and names may be repeated.

\*

\* <h4>Fragment</h4>

\*

\* <p>The fragment is optional: it can be null, empty, or non-empty. Unlike host, port, path, and

\* query the fragment is not sent to the webserver: it's private to the client.

\*

\* <h3>Encoding</h3>

\*

\* <p>Each component must be encoded before it is embedded in the complete URL. As we saw above, the

\* string {@code cute #puppies} is encoded as {@code cute%20%23puppies} when used as a query

\* parameter value.

\*

\* <h4>Percent encoding</h4>

\*

\* <p>Percent encoding replaces a character (like {@code \ud83c\udf69}) with its UTF-8 hex bytes

\* (like {@code %F0%9F%8D%A9}). This approach works for whitespace characters, control characters,

\* non-ASCII characters, and characters that already have another meaning in a particular context.

\*

\* <p>Percent encoding is used in every URL component except for the hostname. But the set of

\* characters that need to be encoded is different for each component. For example, the path

\* component must escape all of its {@code ?} characters, otherwise it could be interpreted as the

\* start of the URL's query. But within the query and fragment components, the {@code ?} character

\* doesn't delimit anything and doesn't need to be escaped. <pre> {@code

\*

\* HttpUrl url = HttpUrl.parse("http://who-let-the-dogs.out").newBuilder()

\* .addPathSegment("\_Who?\_")

\* .query("\_Who?\_")

\* .fragment("\_Who?\_")

\* .build();

\* System.out.println(url);

\* }</pre>

\*

\* This prints: <pre> {@code

\*

\* http://who-let-the-dogs.out/\_Who%3F\_?\_Who?\_#\_Who?\_

\* }</pre>

\*

\* When parsing URLs that lack percent encoding where it is required, this class will percent encode

\* the offending characters.

\*

\* <h4>IDNA Mapping and Punycode encoding</h4>

\*

\* <p>Hostnames have different requirements and use a different encoding scheme. It consists of IDNA

\* mapping and Punycode encoding.

\*

\* <p>In order to avoid confusion and discourage phishing attacks, <a

\* href="http://www.unicode.org/reports/tr46/#ToASCII">IDNA Mapping</a> transforms names to avoid

\* confusing characters. This includes basic case folding: transforming shouting {@code SQUARE.COM}

\* into cool and casual {@code square.com}. It also handles more exotic characters. For example, the

\* Unicode trademark sign (™) could be confused for the letters "TM" in {@code http://ho™mail.com}.

\* To mitigate this, the single character (™) maps to the string (tm). There is similar policy for

\* all of the 1.1 million Unicode code points. Note that some code points such as "\ud83c\udf69" are

\* not mapped and cannot be used in a hostname.

\*

\* <p><a href="http://ietf.org/rfc/rfc3492.txt">Punycode</a> converts a Unicode string to an ASCII

\* string to make international domain names work everywhere. For example, "σ" encodes as "xn--4xa".

\* The encoded string is not human readable, but can be used with classes like {@link InetAddress}

\* to establish connections.

\*

\* <h3>Why another URL model?</h3>

\*

\* <p>Java includes both {@link URL java.net.URL} and {@link URI java.net.URI}. We offer a new URL

\* model to address problems that the others don't.

\*

\* <h4>Different URLs should be different</h4>

\*

\* <p>Although they have different content, {@code java.net.URL} considers the following two URLs

\* equal, and the {@link Object#equals equals()} method between them returns true:

\*

\* <ul>

\* <li>http://square.github.io/

\* <li>http://google.github.io/

\* </ul>

\*

\* This is because those two hosts share the same IP address. This is an old, bad design decision

\* that makes {@code java.net.URL} unusable for many things. It shouldn't be used as a {@link

\* java.util.Map Map} key or in a {@link Set}. Doing so is both inefficient because equality may

\* require a DNS lookup, and incorrect because unequal URLs may be equal because of how they are

\* hosted.

\*

\* <h4>Equal URLs should be equal</h4>

\*

\* <p>These two URLs are semantically identical, but {@code java.net.URI} disagrees:

\*

\* <ul>

\* <li>http://host:80/

\* <li>http://host

\* </ul>

\*

\* Both the unnecessary port specification ({@code :80}) and the absent trailing slash ({@code /})

\* cause URI to bucket the two URLs separately. This harms URI's usefulness in collections. Any

\* application that stores information-per-URL will need to either canonicalize manually, or suffer

\* unnecessary redundancy for such URLs.

\*

\* <p>Because they don't attempt canonical form, these classes are surprisingly difficult to use

\* securely. Suppose you're building a webservice that checks that incoming paths are prefixed

\* "/static/images/" before serving the corresponding assets from the filesystem. <pre> {@code

\*

\* String attack = "http://example.com/static/images/../../../../../etc/passwd";

\* System.out.println(new URL(attack).getPath());

\* System.out.println(new URI(attack).getPath());

\* System.out.println(HttpUrl.parse(attack).encodedPath());

\* }</pre>

\*

\* By canonicalizing the input paths, they are complicit in directory traversal attacks. Code that

\* checks only the path prefix may suffer!

\* <pre> {@code

\*

\* /static/images/../../../../../etc/passwd

\* /static/images/../../../../../etc/passwd

\* /etc/passwd

\* }</pre>

\*

\* <h4>If it works on the web, it should work in your application</h4>

\*

\* <p>The {@code java.net.URI} class is strict around what URLs it accepts. It rejects URLs like

\* "http://example.com/abc|def" because the '|' character is unsupported. This class is more

\* forgiving: it will automatically percent-encode the '|', yielding "http://example.com/abc%7Cdef".

\* This kind behavior is consistent with web browsers. {@code HttpUrl} prefers consistency with

\* major web browsers over consistency with obsolete specifications.

\*

\* <h4>Paths and Queries should decompose</h4>

\*

\* <p>Neither of the built-in URL models offer direct access to path segments or query parameters.

\* Manually using {@code StringBuilder} to assemble these components is cumbersome: do '+'

\* characters get silently replaced with spaces? If a query parameter contains a '&amp;', does that

\* get escaped? By offering methods to read and write individual query parameters directly,

\* application developers are saved from the hassles of encoding and decoding.

\*

\* <h4>Plus a modern API</h4>

\*

\* <p>The URL (JDK1.0) and URI (Java 1.4) classes predate builders and instead use telescoping

\* constructors. For example, there's no API to compose a URI with a custom port without also

\* providing a query and fragment.

\*

\* <p>Instances of {@link HttpUrl} are well-formed and always have a scheme, host, and path. With

\* {@code java.net.URL} it's possible to create an awkward URL like {@code http:/} with scheme and

\* path but no hostname. Building APIs that consume such malformed values is difficult!

\*

\* <p>This class has a modern API. It avoids punitive checked exceptions: {@link #parse parse()}

\* returns null if the input is an invalid URL. You can even be explicit about whether each

\* component has been encoded already.

\*/

中文如下

什么是URL？

URL由几个部分组成

1. 协议类型

协议类型代表使用什么协议。一个URL的协议类型决定了它以什么样的机制去获取资源。尽管URL拥有许多协议，如mailto,file,ftp等，但这个类(HttpUrl)只支持http和https。Java.net.URI可以支持URL的各种协议。

1. 用户名和密码

用户名和密码既可以出现在URL中，也可以是空字符串。这个类并没有区分空字符串和没有用户名密码。这些字段在实际中用的并不多。典型的 HTTP应用就使用其它机制来实现用户认证和授权。

1. 主机

主机指定了提供URL资源的网页服务器。它可以是一个像square.com一样的主机名，一个像192.168.0.1的IP地址，或是像::1这样的一个IPV6地址。

通常，一个网页服务器有多种身份标识，它自己的IP地址，注册的域名，甚至是localhost（如果要连接服务器本身的话）。每个网页服务器的名字是一个独一无二的URL，它们之间是不是能相互替代的。例如，尽管<http://square.github.io/dagger>和<http://google.github.io/dagger>都是由一个相同的IP的主机提供服务，但是这2个URL提供的却是不同的资源。

1. 端口

端口是用来连接服务器用的。默认时HTTP使用80端口，HTTPS使用443端口。这个类对于端口不会返回-1：如果没有显式地在URL中指定协议类型的端口的话，就会使用默认值 。

1. 路径

路径指明了到主机的资源。路径的等级架构用起来像/square/okhttp/issues/1488，并且这会被解析成不同的段,就像[“square”,”okhttp”,”issues”,”1486”]。

这个类提供了按段来组合和提取路径折方法。它把一个列表中的段用/组合起来，就形成了一个路径。例如对于一个段列表[“a”,”b”]可以产生”/a/b”，[“a”,”b”,””]则产生”/a/b/”。

如果路径的最后一段是一个空的字符串那么这个路径就会经”/”结束。这个类产生一个非空的路径，如果路径被忽略也会产生默认的路径”/”。默认的路径的段列表就是一个空的字符串[“”]。

1. 查询

查询选项是可选的。它可以是null，空的或是非空。对于许多HTTP的URL来说，查询字符串可以被分割成键值对。这个类提供了以单空符串设置查询的方式或单独设置键值对的方式。在使用键值对时，值是可选的，键可以重复。

1. 片段

片段是可选的，它可以是null，空或是非空。不像是主机，端口，路径和查询。片段并不会发送给网页服务器。它是客户端私有的。

编码

每个部分在被嵌入一个完整的URL都必须进行编码。就像我们看到的一样，在作为查询参数值时，字符串”cute #puppies”会被编码成”cute%20%23puppies”。

1. 百分号编码

百分号编码将一个UTF8编码的十六进制字符替换为百分号编码如\ud83c\udf69换为%F0%9F%8D%A9。这个方法对空格，控制字符，非ASCII字符及在其它特定场合下有其它意义的字符都有效。

百分号编码在URL中都会用到，除了主机名。每个部分编码方式也会与其它部份不同。例如，路径部分必须将”？”进行转义，不然就会被认为是一个URL查询。但是在查询和片段里，”?”字符并不会有任何限制，不需要转义。

HttpUrl url = HttpUrl.parse("http://who-let-the-dogs.out").newBuilder()

.addPathSegment("\_Who?\_")

.query("\_Who?\_")

.fragment("\_Who?\_")

.build();

System.out.println(url);

}

http://who-let-the-dogs.out/\_Who%3F\_?\_Who?\_#\_Who?\_

当解析到一个缺少百分号编码的URL时，这个类会将错误的字符串进行百分号编码。

1. IDNA 映射和Punycode编码

主机名有一个不同的需求，并使用一个不同的编码协议。它由IDNA映射和Punycode编码组成。

为了避免混淆和抑制钓鱼攻击

IDNA映射将主机名转换为不易混淆的字符。这包含基本的情形：将大小写的转换为小写字符(SQUARE.COM->square.com)。它还会处理额外的字符，例如，Unicode商标TM(右上角)可能会和TM混淆，http://ho™mail.com。为了避免这种情形，就将单字符TM映射到字符串TM。相处的策略适用于1.1百万的Unicode字符。但是请注意，有些字符如”\ud83c\udf69”是不能够被映射，不能在主机名里使用的。

Punycode将Unicode字符串转化为ASCII字符来让国际域名都能在各地工作。例如,“o”被编码为”xn—4xa”。这个编码的字符串人类不可费耶诺德 ，但是能够用来进行建立连接。

为什么会有另外一个URL模型？

Java引入了URL和URI。我们提供一个新的URL模型来解决它们不能解决的问题。

1. 不同的URL的资源就应该不同

尽管它们拥有不同的内容，java.net.URL 却会认为下面的2个URL是相同的，并且用equals()方法来比较会返回true;

<http://square.github.io/>

<http://google.github.io/>

这是因为这两个主机共享同一个IP地址，这是一个错误的决定，它导致了URL在很多时间不可用。它不能够用来放到一个Map中或一个Set中。这样做的话效率很低，因为需要进行DNS解析，并且是不正确的，因为不相同的URL可能是相同的，这取决于它们是如何配置的。

1. 相同的URL的资源就应该相同的

下面2个URL语义上是相同的，但是URI却不这么认为

http://host:80/

http://host

对于不需要的端口80和缺少的/导致URI将2个URL区分开来。这损害了URI在集合中的易用性。任何储存了消息-URL的应用程序需要人工选择，不然就会有冗余。

因为它们并不尝试使用权威的方式，这些类用起来很难保证安全。猜想你现在正在布置一个网页服务器，它会在从文件系统响应之前检查进入路径的前缀。

String attack = "http://example.com/static/images/../../../../../etc/passwd";

System.out.println(new URL(attack).getPath());

System.out.println(new URI(attack).getPath());

System.out.println(HttpUrl.parse(attack).encodedPath());

通过检查输入的路径，它们可能会串通一起来导致攻击。

/static/images/../../../../../etc/passwd

/static/images/../../../../../etc/passwd

/etc/passwd

1. 如果它在网页上工作，那么在你的应用程序上也要能工作

Java.net.URI类对它能接受的URL也是很严格。它拒绝像”<http://example.com/abc|def>一样的链接 ，因为|号是不支持的。我们这个类是更加宽容的一个类，它会自动地使用百分号编码将|进行编码，产生<http://example.com/abc%7Cdef>。这样的行为和浏览器类似，HttpUrl更喜欢和主流的浏览器兼容，而不是过时的标准。

1. 路径和查询应该可以被提取

内建的URL模型都没有提供直接访问路径段和查询参数的方法。人工使用StringBuilder来组装这些部分真的很笨拙:使用”+”来安静地替换空格？如果一个查询参数包含”&amp;”，这会被转义吗？通过提供各种各样的读写方法，来直接读写查询参数，应用程序开发者可以从让人困扰的编码解码中解脱出来。

1. 加一个现代的API

URL和URI在没有开建前就使用了固定的方式。例如，没有API可以单独组成更新自定义端口和提供查询和片段。

HttpUrl是一个设计良好，并且拥有协议类型，主机和路径。在URL中，也可以创建一个/的URL如http:/ 有协议和路径，但没主机名。创建一个能处理这样的乱的格式的API真是够难的！

这个类有一个现代的API，它避免了让人痛苦的异常，parse()会返回空如果输入是一个不合法的URL。并且你可以清楚地知每个部分是不是己经被编码过了！

---------------------------翻译完成-------------------------------