Okhttp3 Analysis

@Tim Qi

JDK Socket

HttpURLConnection

HttpClient

Okhttp

xutils 网络部分

volley

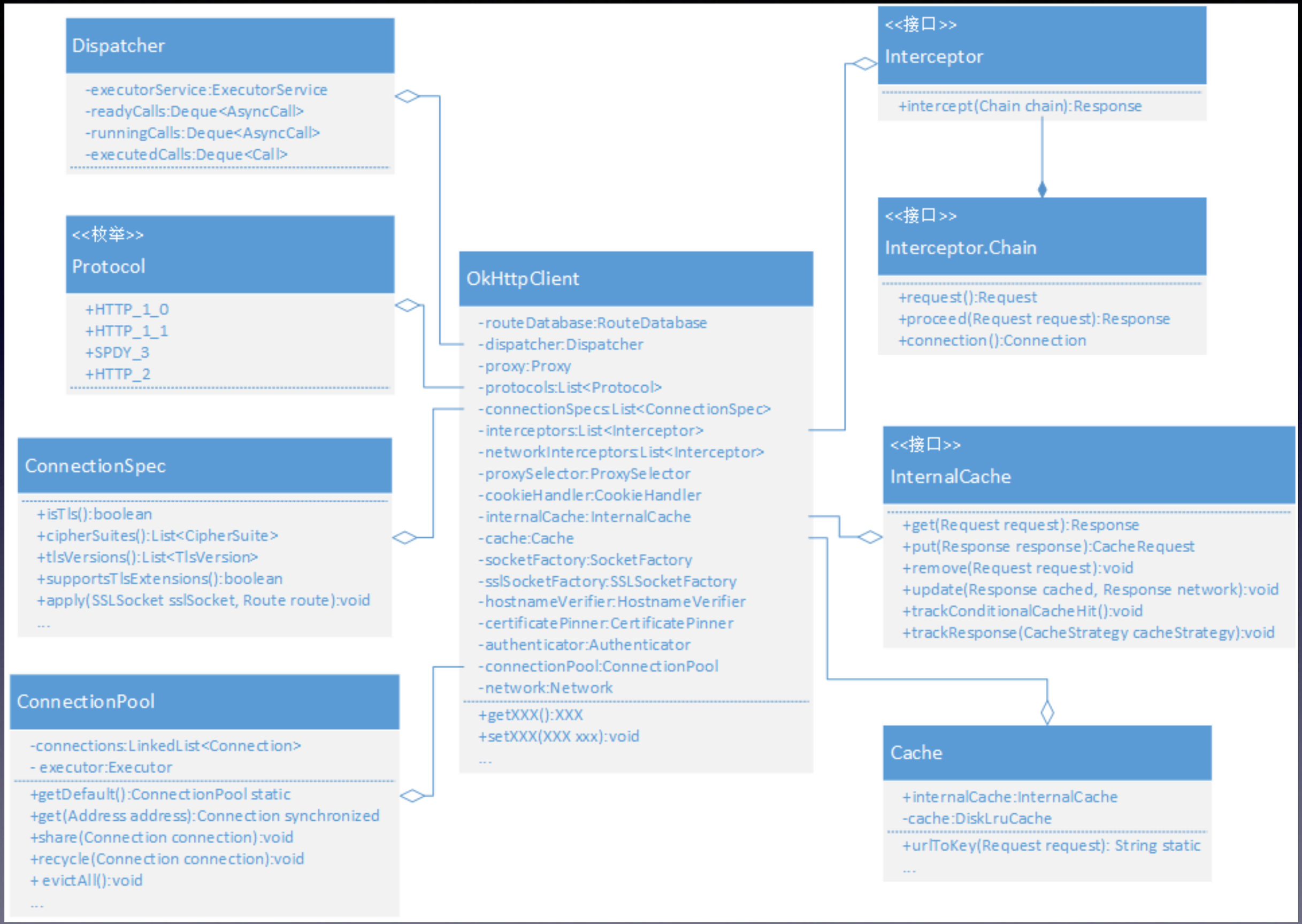
android-async-http

Retrofit

...

特点

- 支持 HTTP 2 / SPDY
- socket 自动选择路线，自动重连
- socket 连接池，减少握手次数
- 队列线程池，高效写并发
- Interceptors
- 基于Headers的缓存策略


Dispatcher
-executorService:ExecutorService
-readyCalls:Deque<AsyncCall>
-runningCalls:Deque<AsyncCall>
-executedCalls:Deque<Call>
<<接口>>
Interceptor
+intercept(Chain chain):Response
<<枚举>>
Protocol
+HTTP_1_0
+HTTP_1_1
+SPDY_3
+HTTP_2
OkHttpClient
-routeDatabase:RouteDatabase
-dispatcher:Dispatcher
-proxy:Proxy
-protocols:List<Protocol>
-connectionSpecs:List<ConnectionSpec>
-interceptors:List<Interceptor>
-networkInterceptors:List<Interceptor>
-proxySelector:ProxySelector
-cookieHandler:CookieHandler
-internalCache:InternalCache
-cache:Cache
-socketFactory:SocketFactory
-sslSocketFactory:SSLSocketFactory
-hostnameVerifier:HostnameVerifier
-certificatePinner:CertificatePinner
-authenticator:Authenticator
-connectionPool:ConnectionPool
-network:Network
+getXXX():XXX
+setXXX(XXX xxx):void
...
<<接口>>
Interceptor.Chain
+request():Request
+proceed(Request request):Response
+connection():Connection
ConnectionSpec
+isTls():boolean
+cipherSuites():List<CipherSuite>
+tlsVersions():List<TlsVersion>
+supportsTlsExtensions():boolean
+apply(SSLSocket sslSocket, Route route):void
...
<<接口>>
InternalCache
+get(Request request):Response
+put(Response response):CacheRequest
+remove(Request request):void
+update(Response cached, Response network):void
+trackConditionalCacheHit():void
+trackResponse(CacheStrategy cacheStrategy):void
ConnectionPool
-connections:LinkedList<Connection>
- executor:Executor
+getDefault():ConnectionPool static
+get(Address address):Connection synchronized
+share(Connection connection):void
+recycle(Connection connection):void
+ evictAll():void
...
Cache
+internalCache:InternalCache
-cache:DiskLruCache
+urlToKey(Request request): String static
...

- 任务调度
- 缓存处理
- 链接管理

线程池

- 为什么要开线程
- 使用单线程有什么缺点

```
executorService = new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60, TimeUnit.SECONDS,
    new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));
```

- int corePoolSize: 最小并发线程数
- int maximumPoolSize: 最大线程数
- long keepAliveTime: 保活时间
- TimeUnit unit: 时间单位，一般用秒
- BlockingQueue<Runnable> workQueue: 工作队列
- ThreadFactory threadFactory: 单个线程的工厂

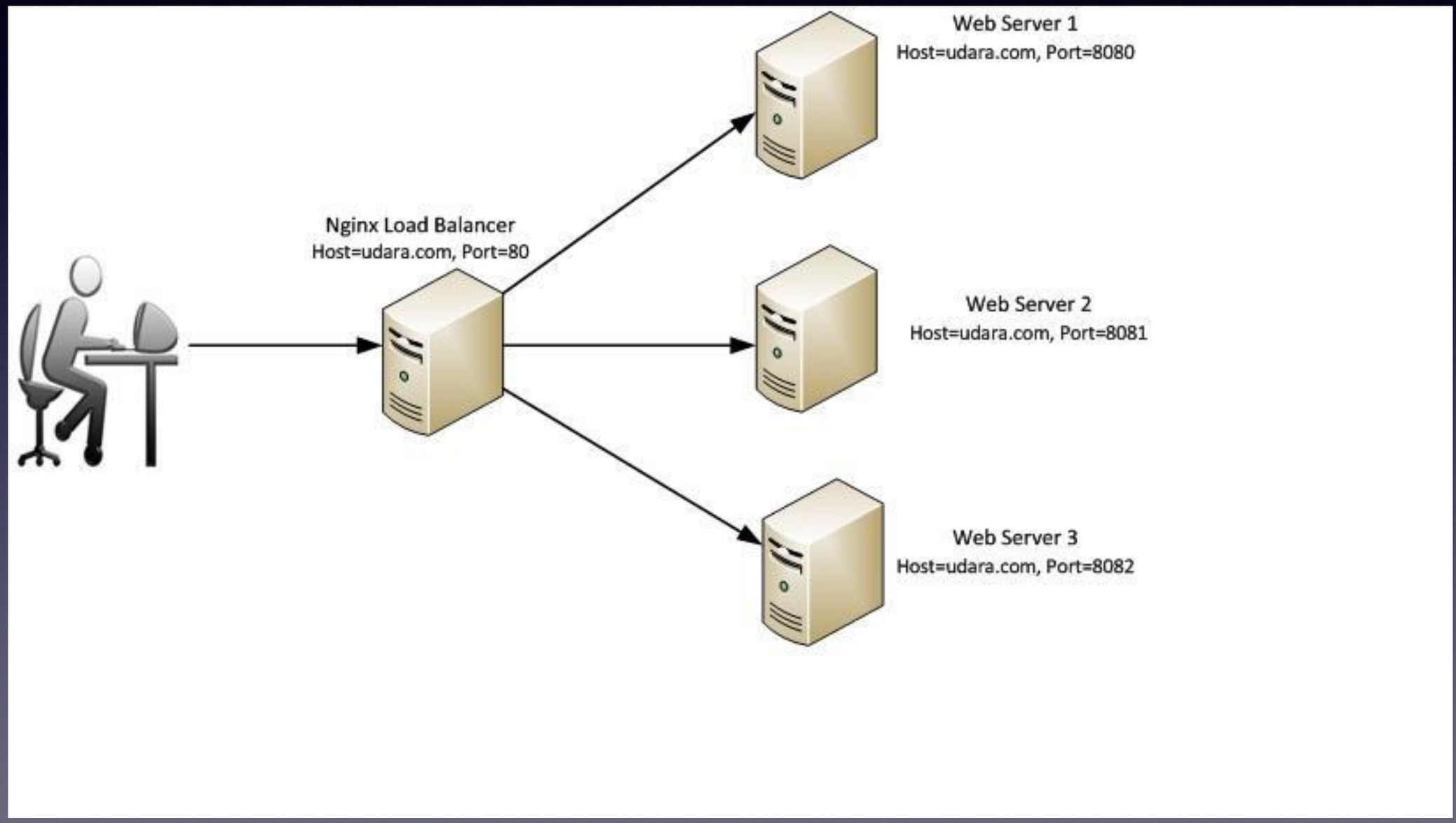
Java 线程池

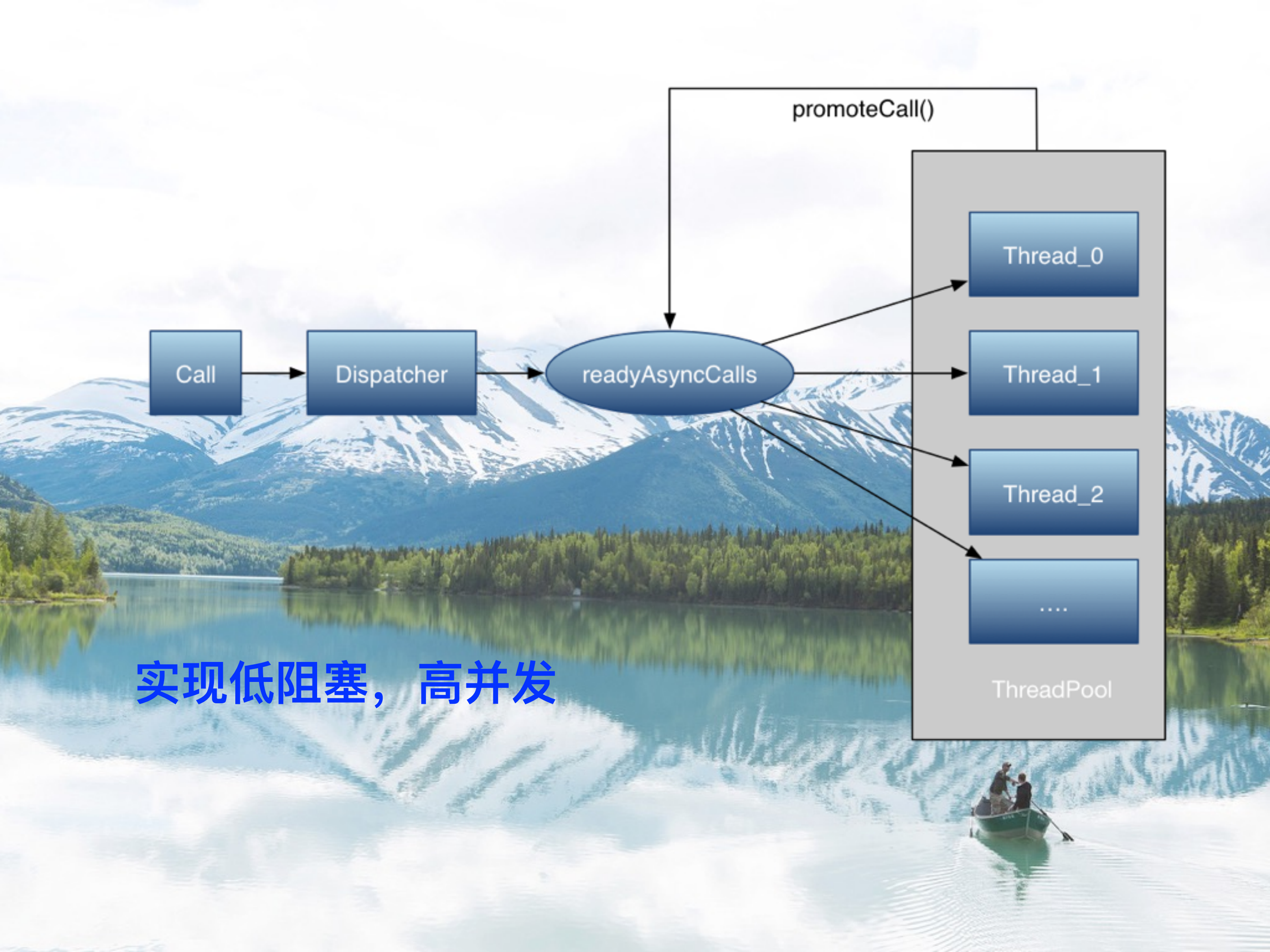
- FixedThreadPool
- CachedThreadPool
- ScheduledThreadPool
- SingleThreadPool

```java
public synchronized ExecutorService executorService() {
  if (executorService == null) {
    executorService = new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));
  }
  return executorService;
}
```

- corePoolSize，maximumPoolSize，keepAliveTime，unit，workQueue，threadFactory
- 在Okhttp中，构建了一个阀值为[0, Integer.MAX_VALUE]的线程池，它不保留任何最小线程数，随时创建更多的线程数，当线程空闲时只能活60秒，它使用了一个不存储元素的阻塞工作队列，一个叫做"OkHttp Dispatcher"的线程工厂

反向代理模型


Web Server 1
Host=udara.com, Port=8080
Nginx Load Balancer
Host=udara.com, Port=80
Web Server 2
Host=udara.com, Port=8081
Web Server 3
Host=udara.com, Port=8082


promoteCall()
Call
Dispatcher
readyAsyncCalls
Thread_0
Thread_1
Thread_2
....
ThreadPool

实现低阻塞，高并发

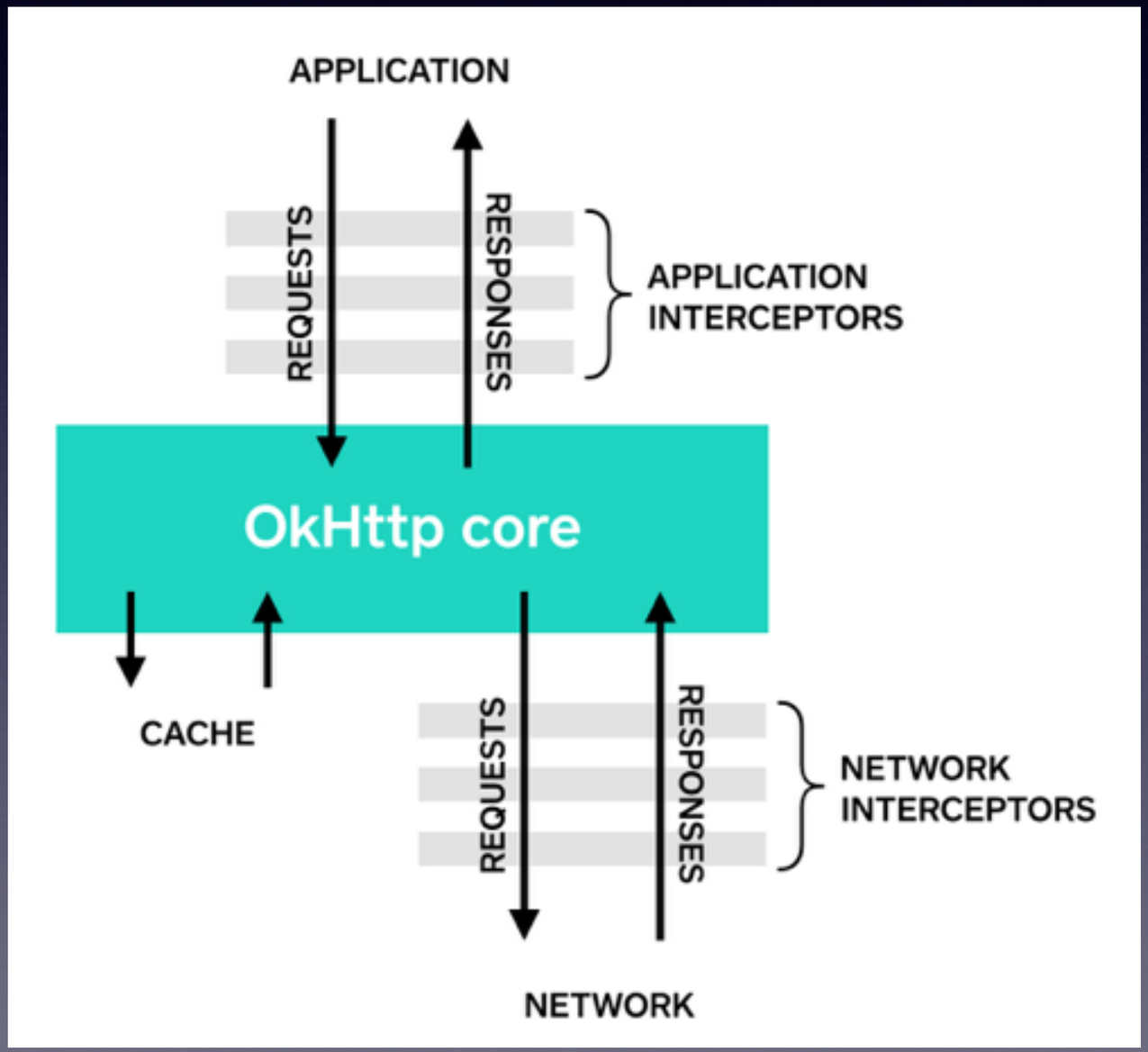
生产者消费者模型

Example

- Synchronous Get

- Asynchronous Get

```
synchronized void enqueue(AsyncCall call) {
  if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {
    runningAsyncCalls.add(call);
    executorService().execute(call);
  } else {
    readyAsyncCalls.add(call);                                    入队操作
  }
}
```



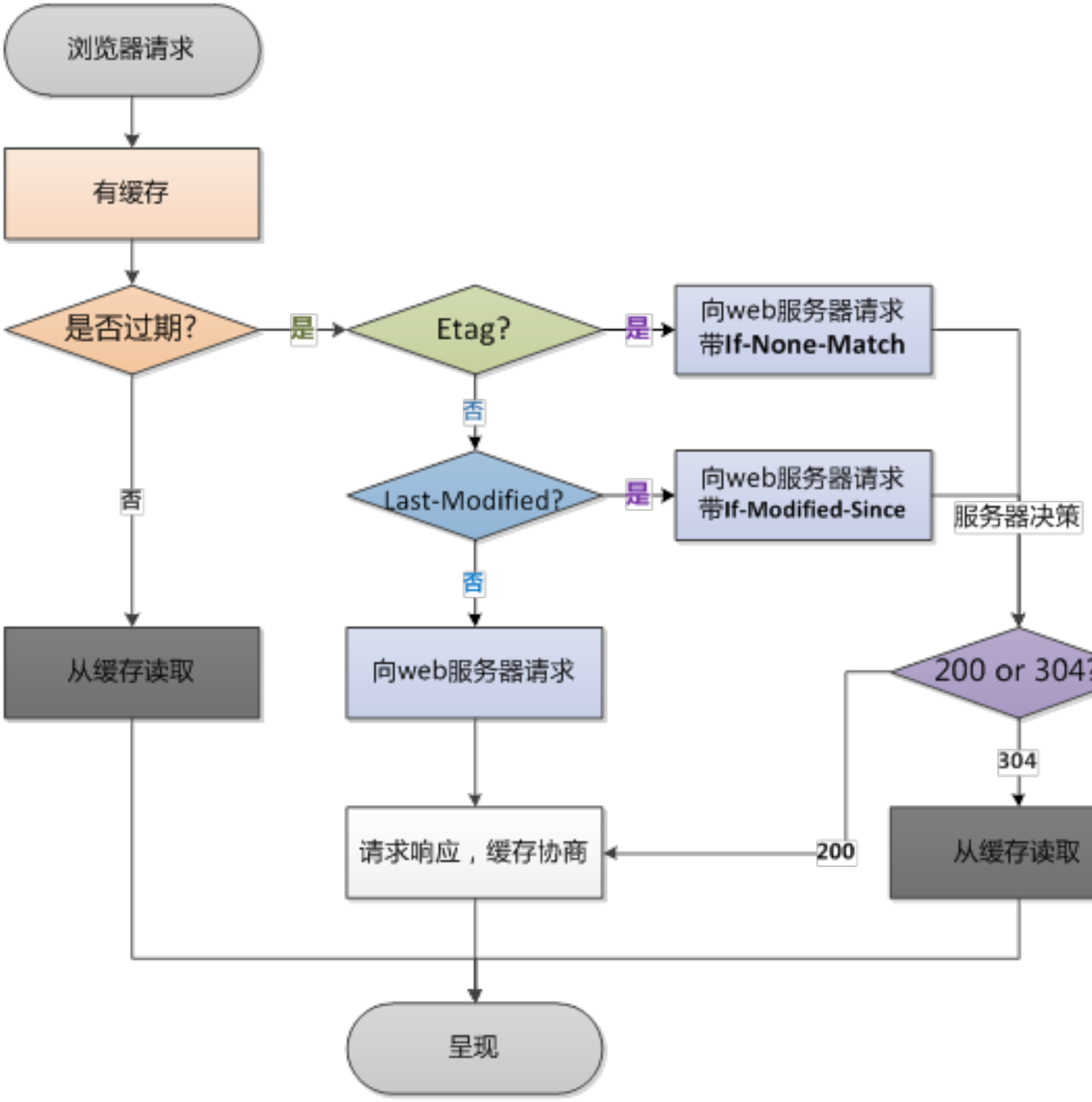
```
@Override protected void execute() {
  boolean signalledCallback = false;
  try {
    Response response = getResponseWithInterceptorChain(forWebSocket);
    if (canceled) {
      signalledCallback = true;
      responseCallback.onFailure(RealCall.this, new IOException("Canceled"));
    } else {
      signalledCallback = true;
      responseCallback.onResponse(RealCall.this, response);
    }
  } catch (IOException e) {
    if (signalledCallback) {
      // Do not signal the callback twice!
      logger.log(Level.INFO, "Callback failure for " + toLoggableString(), e);
    } else {
      responseCallback.onFailure(RealCall.this, e);
    }
  } finally {
    client.dispatcher().finished(this);
  }
}
```

AsyncCall

```
private void promoteCalls() {
  if (runningAsyncCalls.size() >= maxRequests) return; // Already running max capacity.
  if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

  for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {
    AsyncCall call = i.next();

    if (runningCallsForHost(call) < maxRequestsPerHost) {
      i.remove();
      runningAsyncCalls.add(call);
      executorService().execute(call);
    }

    if (runningAsyncCalls.size() >= maxRequests) return; // Reached max capacity.
  }
}
```

手动消费，避免死锁

- 任务调度
- 缓存处理
- 链接管理


浏览器请求
有缓存
是否过期?
是
Etag?
是
向web服务器请求
带If-None-Match
否
Last-Modified?
是
向web服务器请求
带If-Modified-Since
否
否
服务器决策
从缓存读取
向web服务器请求
200 or 304?
304
请求响应，缓存协商
200
从缓存读取
呈现

缓存

是方便用户快速的获取值的一种储存方式。小到与CPU同频的昂贵的缓存颗粒，内存，硬盘，网络，CDN反代缓存，DNS递归查询，OS页面置换，都可以看作缓存

缓存

- 缓存载体与持久载体总是相对的，容量远远小于持久容量，成本高于持久容量，速度高于持久容量
- 需要一种页面置换算法(page replacement algorithm)将旧页面去掉换成新的页面，如LRU，FIFO，LFU，NRU
- 如果没有命中缓存，就需要从原始地址获取，这个步骤叫做“回源”

networkRequest	cacheResponse	result
null	null	only-if-cached(表明不进行网络请求，且缓存不存在或者过期，一定会返回503错误)
null	non-null	不进行网络请求，而且缓存可以使用，直接返回缓存，不用请求网络
non-null	null	需要进行网络请求，而且缓存不存在或者过期，直接访问网络
non-null	non-null	Header中含有`ETag/Last-Modified`标签，需要在`条件请求`下使用，还是需要访问网络

OkHttp中使用了CacheStrategy实现缓存，它根据之前的缓存结果与当前将要发送Request的header进行策略分析，并得出是否进行请求的结论

Map In Java

- HashMap
- HashTable
- LinkedHashMap
- TreeMap

不计扩容的时间复杂度

	HashMap	LinkedHashMap	TreeMap
Performance get/set	O(1)	O(1)	O(logN)
Implement	Array	Link + Array	Red-Black Tree
Iteration	unpredictable	put/accessOrder	Comparable `<Key>`

在OkHttp中，使用FileSystem作为缓存载体（磁盘相对于网络的缓存），使用LRU作为页面置换算法（封装了LinkedHashMap）

Okhttp 缓存的主要对象

- FileSystem：Okio封装
- DiskLruCache.Editor：同步锁，高度封装fs
- DiskLruCache.Entry: 维护key对应的多个文件
- Cache.Entry: 封装Response对应流
- DiskLruCache：文件的创建，清理，读取。内部有清理线程池，LinkedHashMap（即LruCache）
- Cache：对上层提供透明的get / put操作

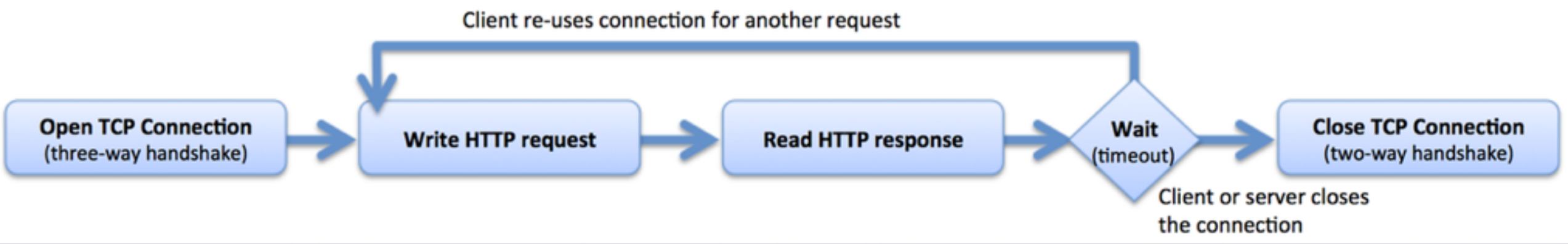
- 任务调度
- 缓存处理
- 链接管理

链接过程


Open TCP Connection
(three-way handshake)
Write HTTP request
Read HTTP response
Close TCP Connection
(two-way handshake)

缺点

- 建立连接的时延过长


Client re-uses connection for another request
Open TCP Connection
(three-way handshake)
Write HTTP request
Read HTTP response
Wait
(timeout)
Close TCP Connection
(two-way handshake)
Client or server closes
the connection

缺点

- 由于 TCP 拥塞机制，当带宽一定是，存在过多的僵尸连接会影响其它客户端访问速度
- 服务器的并发限制
- 可能用于 DDOS

不过这些缺点都不太要紧

连接池相关的对象

- Call，封装 HTTP 请求
- Connection，对 jdk 中Socket 物理连接的封装，内部有 List<WeakReference<StreamAllocation>> 的引用
- StreamAllocation，Connection 被上层代码引用的次数
- ConnectionPool，Socket 连接池，对连接进行缓存和管理
- Deque，双端队列，用来存储 connection

StreamAllocation

- 用来跟踪流的调用，避免因为僵尸连接带来的内存泄漏
- 提供 acquire，release 接口来改变引用计数值
- 管理 Stream，Connection 的分配和 ConnectionPool 的交互

连接过程

- 选择线路与自动重连（RouteSelector）
- 连接 Socket 链路 （RealConnection）
- 释放 Socket

如果 Proxy 为 null

- 则设置为 Proxy.NO_PROXY
- 通过 DNS 查找到 InetSocketAddress，结果为一个数组，作为自动重连的来源
- 调用 next() 知道查询到为止
- 如果 next() 不能枚举出结果，则抛出异常

如果 Proxy 为 HTTP

- 设置 socket ip 为代理地址 ip
- 设置 socket 端口为代理地址端口
- 如果 next() 不能枚举出结果，则抛出异常

路线选择的迷之缩进

```
public Route next() throws IOException {
  // Compute the next route to attempt.
  if (!hasNextInetSocketAddress()) {
    if (!hasNextProxy()) {
      if (!hasNextPostponed()) {
        throw new NoSuchElementException();
      }
      return nextPostponed();
    }
    lastProxy = nextProxy();
  }
  lastInetSocketAddress = nextInetSocketAddress();

  Route route = new Route(address, lastProxy, lastInetSo
  if (routeDatabase.shouldPostpone(route)) {
    postponedRoutes.add(route);
    // We will only recurse in order to skip previously
    return next();
  }

  return route;
```

连接 Socket（RealConnection）

- 如果连接池中已经存在则立即连接，否则进行下一步分配 （StreamAllocation）
- 根据路线（Route），调用 Platform.get().connectSocket 选择当前平台最好的库尝试 socket 连接
- 将成功的 RealConnection 放入连接池中
- 如果存在 TLS，则进行 SSL 版本证书认证
- 构造 HttpStream 并维护 socket 连接

StreamAllocation 新建 Stream

```
public HttpStream newStream(int connectTimeout, int readTimeout, int writeTimeout,
    boolean connectionRetryEnabled, boolean doExtensiveHealthChecks)
    throws RouteException, IOException {
  try {
    RealConnection resultConnection = findHealthyConnection(connectTimeout, readTime
        writeTimeout, connectionRetryEnabled, doExtensiveHealthChecks);

    HttpStream resultStream;
    if (resultConnection.framedConnection != null) {
      resultStream = new Http2xStream(this, resultConnection.framedConnection);
    } else {
      resultConnection.socket().setSoTimeout(readTimeout);
      resultConnection.source.timeout().timeout(readTimeout, MILLISECONDS);
      resultConnection.sink.timeout().timeout(writeTimeout, MILLISECONDS);
      resultStream = new Http1xStream(this, resultConnection.source, resultConnectio
    }
```

释放 Socket

- 尝试从连接池中删除
- 如果连接池中没有命中，则调用 socket 的关闭

ConnectionPool 初始化

```
hostnameVerifier = OkHostnameVerifier.INSTANCE;
certificatePinner = CertificatePinner.DEFAULT;
proxyAuthenticator = Authenticator.NONE;
authenticator = Authenticator.NONE;
connectionPool = new ConnectionPool();
dns = Dns.SYSTEM;
followSslRedirects = true;
followRedirects = true;
retryOnConnectionFailure = true;
```

ConnectionPool 结构

- 名为 “OkHttp ConnectionPool” 的 ThreadPool 做 Connection 的回收
- Deque<Connection>，提供 get / put / remove
- RouteBase，记录链接失败的 Route

Connection自动回收的实现

引用计数法