# OkHttp

## RealCall

1、RealCall实现了Call接口

public interface Call {

Request request();

Response execute() throws IOException;

void enqueue(Callback var1);

void cancel();

boolean isExecuted();

boolean isCanceled();

public interface Factory {

Call newCall(Request var1);

}

}

2、execute源码

public Response execute() throws IOException {

synchronized(this) {

if(this.executed) {

throw new IllegalStateException("Already Executed");

}

this.executed = true;

}

Response var2;

try {

this.client.dispatcher().executed(this);

Response result = this.getResponseWithInterceptorChain(false);

if(result == null) {

throw new IOException("Canceled");

}

var2 = result;

} finally {

this.client.dispatcher().finished(this);

}

return var2;

}

3、enqueue源码

void enqueue(Callback responseCallback, boolean forWebSocket) {

synchronized(this) {

if(this.executed) {

throw new IllegalStateException("Already Executed");

}

this.executed = true;

}

this.client.dispatcher().enqueue(  
new RealCall.AsyncCall(responseCallback, forWebSocket));

}

4、

## Dispatcher

1、分发器内部重要字段介绍

private int maxRequests = 64;//最大同时连接数

private int maxRequestsPerHost = 5;//同一个主机的最大连接数

private ExecutorService executorService;//线程池

private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque();//异步调用的等待队列

private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque();//正在运行的异步调用队列

private final Deque<RealCall> runningSyncCalls = new ArrayDeque();//正在运行的同步调用队列

2、execute以及与execute对应的finished源码

synchronized void executed(RealCall call) {

this.runningSyncCalls.add(call);//仅仅将其添加到同步调用队列中，说明真正的执行代码其实是在getResponseWithInterceptorChain函数中

}

synchronized void finished(Call call) {

if(!this.runningSyncCalls.remove(call)) {

throw new AssertionError("Call wasn\'t in-flight!");

}//对于同步执行的任务来说，只需要将这个Call的实例移出runningSyncCalls队列即可

}

3、enqueue以及与enqueue对应的finished源码

synchronized void enqueue(AsyncCall call) {

if(this.runningAsyncCalls.size() < this.maxRequests   
&& this.runningCallsForHost(call) < this.maxRequestsPerHost) {

this.runningAsyncCalls.add(call);

this.executorService().execute(call);//满足限制条件就执行

} else {

this.readyAsyncCalls.add(call);//否则添加到等待队列中

}

}

synchronized void finished(AsyncCall call) {

if(!this.runningAsyncCalls.remove(call)) {

throw new AssertionError("AsyncCall wasn\'t running!");

} else {

this.promoteCalls();//对于异步执行的AsyncCall来说，还需要额外的操作，使得线程可以执行其他尚未执行的AsyncCall

}

}

private void promoteCalls() {

if(this.runningAsyncCalls.size() < this.maxRequests) {//正在运行的任务数量小于同时最大任务数量，否则就没有拉去新任务的必要

if(!this.readyAsyncCalls.isEmpty()) {//等待队列不为空才能进行拉取

Iterator i = this.readyAsyncCalls.iterator();

do {

if(!i.hasNext()) {//或者readyAsyncCalls为空时返回

return;

}

AsyncCall call = (AsyncCall)i.next();

if(this.runningCallsForHost(call) < this.maxRequestsPerHost){

i.remove();

this.runningAsyncCalls.add(call);//加到正在执行队列

this.executorService().execute(call);//执行

}

} while(this.runningAsyncCalls.size() < this.maxRequests);//直到填同时执行的数量与maxRequests相同

}

}

}

4、getResponseWithInterceptorChain源码

private Response getResponseWithInterceptorChain(boolean forWebSocket) throws IOException {

RealCall.ApplicationInterceptorChain chain =   
new RealCall.ApplicationInterceptorChain(0,   
this.originalRequest, forWebSocket);

return chain.proceed(this.originalRequest);

}

## AsyncCall

1、异步调用的任务，继承自NamedRunnable(封装了Runnable接口的抽象类)，与RealCall完全不同，RealCall实现了Call接口，而AsyncCall继承了NamedRunnable抽象类，两者具有同名方法execute。并且AsyncCall是RealCall的内部类

2、execute方法源码

protected void execute() {

boolean signalledCallback = false;

try {

Response e =   
RealCall.this.getResponseWithInterceptorChain(this.forWebSocket);//异步非阻塞的真正获取相应的方法在这里调用

if(RealCall.this.canceled) {

signalledCallback = true;

this.responseCallback.onFailure(RealCall.this,   
new IOException("Canceled"));//回调

} else {

signalledCallback = true;

this.responseCallback.onResponse(RealCall.this, e); //回调

}

} catch (IOException var6) {

if(signalledCallback) {

Internal.logger.log(Level.INFO, "Callback failure for " +   
RealCall.this.toLoggableString(), var6);

} else {

this.responseCallback.onFailure(RealCall.this, var6); //回调

}

} finally {

RealCall.this.client.dispatcher().finished(this);

}

}

## ApplicationInterceptorChain

1、ApplicationInterceptorChain是RealCall的内部类，实现了接口Chain

public interface Interceptor {

Response intercept(Interceptor.Chain var1) throws IOException;

public interface Chain {

Request request();

Response proceed(Request var1) throws IOException;

Connection connection();

}

}

* Chain接口是Interceptor的内部接口，只是这样组织而已，不要太在意
* Chain实现是由该类库提供的，即ApplicationInterceptorChain
* Interceptor的实现可以由用户自己提供，例如

class MyTestInterceptor1 implements Interceptor {

public Response intercept(Chain chain) throws IOException {

Request request = chain.request();

//拦截调用之前执行的动作

Response response = chain.proceed(request);

//拦截调用之后执行的动作

return response;

}

}

2、proceed源码

public Response proceed(Request request) throws IOException {

if(this.index < RealCall.this.client.interceptors().size()) {//①

RealCall.ApplicationInterceptorChain chain =   
RealCall.this.new ApplicationInterceptorChain(this.index + 1,   
request, this.forWebSocket);//这里为什么需要生成一个新的Chain的实例，因为index字段是final的，至于为什么要final目前暂不清楚

Interceptor interceptor =   
(Interceptor)RealCall.this.client.interceptors().get(this.index);//获取当前index所指向的拦截器

Response interceptedResponse = interceptor.intercept(chain);//进行拦截，在拦截过程中会继续回调chain.process方法，直到①处的判断为false，才会进行②处的语句

if(interceptedResponse == null) {

throw new NullPointerException("application interceptor " +  
 interceptor + " returned null");

} else {

return interceptedResponse;

}

} else {

return RealCall.this.getResponse(request, this.forWebSocket);//②

}

}

# Log4j

## 依赖项配置

1、使用maven可以添加如下依赖

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.6.6</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.6</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

## 资源文件log4j.properties的配置

1、Log4j由三个重要的组件构成：日志信息的优先级，日志信息的输出目的地，

日志信息的输出格式

1. 日志信息的优先级从高到低有ERROR、WARN、 INFO、DEBUG，分别用来指定这条日志信息的重要程度
2. 日志信息的输出目的地指定了日志将打印到控制台还是文件中
3. 而输出格式则控制了日志信息的显示内容

2、可以完全不使用配置文件，而是在代码中配置Log4j环境。但是，使用配置文件将使应用程序更加灵活。Log4j支持两种配置文件格式，一种是XML格式的文件，一种是Java特性文件(键=值)。下面我们介绍使用Java特性文件做为配置文件的方法

3、Log级别

1. ALL Level是最低等级的，用于打开所有日志记录
2. DEBUG Level：指出细粒度信息事件对调试应用程序是非常有帮助的
3. INFO level：表明消息在粗粒度级别上突出强调应用程序的运行过程
4. WARN level：表明会出现潜在错误的情形
5. ERROR level：指出虽然发生错误事件，但仍然不影响系统的继续运行
6. FATAL level：指出每个严重的错误事件将会导致应用程序的退出
7. OFF Level是最高等级的，用于关闭所有日志记录

* Log4j建议只使用四个级别，优先级从高到低分别是ERROR、WARN、INFO、DEBUG。通过在这里定义的级别，您可以控制到应用程序中相应级别的日志信息的开关。比如在这里定义了INFO级别，则应用程序中所有DEBUG级别的日志信息将不被打印出来，也是说大于等于的级别的日志才输出。

### 配置根Logger

1、配置根Logger，其语法为：

log4j.rootLogger = [ level ] , appenderName, appenderName, …

1. level是日志记录的优先级，分为OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL或者自定义的级别。Log4j建议只使用四个级别，优先级从高到低分别是ERROR、WARN、INFO、DEBUG。通过在这里定义的级别，可以控制到应用程序中相应级别的日志信息的开关。比如在这里定义了INFO级别，则应用程序中所有DEBUG级别的日志信息将不被打印出来
2. appenderName就是指B日志信息输出到哪个地方。可以同时指定多个输出目的地

### 配置日志信息输出目的地Appender

1、语法如下

log4j.appender.appenderName = fully.qualified.name.of.appender.class

log4j.appender.appenderName.option1 = value1

…

log4j.appender.appenderName.option = valueN

2、其中，Log4j提供的appender有以下几种

org.apache.log4j.ConsoleAppender(控制台)

org.apache.log4j.FileAppender(文件)

org.apache.log4j.DailyRollingFileAppender(每天产生一个日志文件)

org.apache.log4j.RollingFileAppender(文件大小到达指定尺寸的时候产生一个新的文件)

org.apache.log4j.WriterAppender(将日志信息以流格式发送到任意指定的地方)

### 配置日志信息的格式

1、其语法为

log4j.appender.appenderName.layout = fully.qualified.name.of.layout.class

log4j.appender.appenderName.layout.option1 = value1

…

log4j.appender.appenderName.layout.option = valueN

2、其中，Log4j提供的layout有以下几种

org.apache.log4j.HTMLLayout(以HTML表格形式布局)

org.apache.log4j.PatternLayout(可以灵活地指定布局模式)

org.apache.log4j.SimpleLayout(包含日志信息的级别和信息字符串)

org.apache.log4j.TTCCLayout(包含日志产生的时间、线程、类别等等信息)

3、Log4J采用类似C语言中的printf函数的打印格式格式化日志信息

%%：输出一个"%"字符

%c：输出所属的类目，通常就是所在类的全名

%d：输出日志时间点的日期或时间，默认格式为ISO8601，也可以在其后指定格式，比如：%d{yyyy-MM-dd HH:mm:ss}，输出类似：2017-03-22 18:14:34,829

%F：输出日志消息产生时所在的文件名称

%l：输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。举例：Testlog4.main(TestLog4.java:10)

%L：输出代码中的行号

%m：输出代码中指定的消息,产生的日志具体信息

%n：输出一个回车换行符，Windows平台为"rn"，Unix平台为"n"

%p：输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL

%r：输出自应用启动到输出该log信息耗费的毫秒数

%t：输出产生该日志事件的线程名

%x：输出和当前线程相关联的NDC(嵌套诊断环境),尤其用到像java servlets这样的多客户多线程的应用中

Quartz

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>${quartz.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${spring.version}</version>

</dependency>

<!-- Transaction dependency is required with Quartz integration -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>