1. OkHttp

1.1.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、

1.2. 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);
}
```

1.3. 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);
            }
        }
```

1.4. 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
        }
    }
```

2. Log4j

2.1. 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>
```

2.2. 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。

2.2.1. 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

2.2.2. 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)

2.2.3. 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>
```