# OkHttp：Java 平台上的新一代 HTTP 客户端

原文地址: <https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html>

在 Java 程序中经常需要用到 HTTP 客户端来发送 HTTP 请求并对所得到的响应进行处理。比如屏幕抓取（screen scraping）程序通过 HTTP 客户端来访问网站并解析所得到的 HTTP 文档。在 Java 服务端程序中也可能需要使用 HTTP 客户端来与第三方 REST 服务进行集成。随着微服务（microservices）的流行，HTTP 成为不同服务之间的标准集成方式。HTTP 客户端的重要性也日益显著。在 Java 平台上，Java 标准库提供了 HttpURLConnection 类来支持 HTTP 通讯。不过 HttpURLConnection 本身的 API 不够友好，所提供的功能也有限。大部分 Java 程序都选择使用 Apache 的开源项目 HttpClient 作为 HTTP 客户端。Apache HttpClient 库的功能强大，使用率也很高，基本上是 Java 平台中事实上的标准 HTTP 客户端。本文介绍的是由 Square 公司开发的 OkHttp，是一个专注于性能和易用性的 HTTP 客户端。

## OkHttp 简介

OkHttp 库的设计和实现的首要目标是高效。这也是选择 OkHttp 的重要理由之一。OkHttp 提供了对最新的 HTTP 协议版本 HTTP/2 和 SPDY 的支持，这使得对同一个主机发出的所有请求都可以共享相同的套接字连接。如果 HTTP/2 和 SPDY 不可用，OkHttp 会使用连接池来复用连接以提高效率。OkHttp 提供了对 GZIP 的默认支持来降低传输内容的大小。OkHttp 也提供了对 HTTP 响应的缓存机制，可以避免不必要的网络请求。当网络出现问题时，OkHttp 会自动重试一个主机的多个 IP 地址。

在 Java 程序中使用 OkHttp 非常简单，只需要在 Maven 的 POM 文件中添加[代码清单 1](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 1. OkHttp %E7%9A%84 Maven %E4%BE%9D%E8%B5%96%E5%A3%B0%E6%98%8E) 中的依赖即可。目前 OkHttp 的最新版本是 2.5.0。

##### 清单 1. OkHttp 的 Maven 依赖声明

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>   <groupId>com.squareup.okhttp</groupId>   <artifactId>okhttp</artifactId>   <version>2.5.0</version>  </dependency> |

## HTTP 连接

虽然在使用 OkHttp 发送 HTTP 请求时只需要提供 URL 即可，OkHttp 在实现中需要综合考虑 3 种不同的要素来确定与 HTTP 服务器之间实际建立的 HTTP 连接。这样做的目的是为了达到最佳的性能。

首先第一个考虑的要素是 URL 本身。URL 给出了要访问的资源的路径。比如 URL http://www.baidu.com 所对应的是百度首页的 HTTP 文档。在 URL 中比较重要的部分是访问时使用的模式，即 HTTP 还是 HTTPS。这会确定 OkHttp 所建立的是明文的 HTTP 连接，还是加密的 HTTPS 连接。

第二个要素是 HTTP 服务器的地址，如 baidu.com。每个地址都有对应的配置，包括端口号，HTTPS 连接设置和网络传输协议。同一个地址上的 URL 可以共享同一个底层 TCP 套接字连接。通过共享连接可以有显著的性能提升。OkHttp 提供了一个连接池来复用连接。

第三个要素是连接 HTTP 服务器时使用的路由。路由包括具体连接的 IP 地址（通过 DNS 查询来发现）和所使用的代理服务器。对于 HTTPS 连接还包括通讯协商时使用的 TLS 版本。对于同一个地址，可能有多个不同的路由。OkHttp 在遇到访问错误时会自动尝试备选路由。

当通过 OkHttp 来请求某个 URL 时，OkHttp 首先从 URL 中得到地址信息，再从连接池中根据地址来获取连接。如果在连接池中没有找到连接，则选择一个路由来尝试连接。尝试连接需要通过 DNS 查询来得到服务器的 IP 地址，也会用到代理服务器和 TLS 版本等信息。当实际的连接建立之后，OkHttp 发送 HTTP 请求并获取响应。当连接出现问题时，OkHttp 会自动选择另外的路由进行尝试。这使得 OkHttp 可以自动处理可能出现的网络问题。当成功获取到 HTTP 请求的响应之后，当前的连接会被放回到连接池中，提供给后续的请求来复用。连接池会定期把闲置的连接关闭以释放资源。

## 请求，响应与调用

HTTP 客户端所要执行的任务很简单，接受 HTTP 请求并返回响应。每个 HTTP 请求包括 URL，HTTP 方法（如 GET 或 POST），HTTP 头和请求的主体内容等。HTTP 请求的响应则包含状态代码（如 200 或 500），HTTP 头和响应的主体内容等。虽然请求和响应的交互模式很简单，但在实现中仍然有很多细节要考虑。OkHttp 会对收到的请求进行一定的处理，比如增加额外的 HTTP 头。同样的，OkHttp 也可能在返回响应之前对响应做一些处理。例如，OkHttp 可以启用 GZIP 支持。在发送实际的请求时，OkHttp 会加上 HTTP 头 Accept-Encoding。在接收到服务器的响应之后，OkHttp 会先做解压缩处理，再把结果返回。如果 HTTP 响应的状态代码是重定向相关的，OkHttp 会自动重定向到指定的 URL 来进一步处理。OkHttp 也会处理用户认证相关的响应。

OkHttp 使用调用（Call）来对发送 HTTP 请求和获取响应的过程进行抽象。[代码清单 2](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 2. OkHttp %E6%9C%80%E5%9F%BA%E6%9C%AC%E7%9A%84 HTTP %E8%AF%B7%E6%B1%82) 中给出了使用 OkHttp 发送 HTTP 请求的基本示例。首先创建一个 OkHttpClient 类的对象，该对象是使用 OkHttp 的入口。接着要创建的是表示 HTTP 请求的 Request 对象。通过 Request.Builder 这个构建帮助类可以快速的创建出 Request 对象。这里指定了 Request 的 url 为 http://www.baidu.com。接着通过 OkHttpClient 的 newCall 方法来从 Request 对象中创建一个 Call 对象，再调用 execute 方法来执行该调用，所得到的结果是表示 HTTP 响应的 Response 对象。通过 Response 对象中的不同方法可以访问响应的不同内容。如 headers 方法来获取 HTTP 头，body 方法来获取到表示响应主体内容的 ResponseBody 对象。

##### 清单 2. OkHttp 最基本的 HTTP 请求

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class SyncGet {     public static void main(String[] args) throws IOException {      OkHttpClient client = new OkHttpClient();        Request request = new Request.Builder()              .url("http://www.baidu.com")              .build();        Response response = client.newCall(request).execute();      if (!response.isSuccessful()) {          throw new IOException("服务器端错误: " + response);      }        Headers responseHeaders = response.headers();      for (int i = 0; i < responseHeaders.size(); i++) {          System.out.println(responseHeaders.name(i) + ": " + responseHeaders.value(i));      }        System.out.println(response.body().string());     }  } |

## HTTP 头处理

HTTP 头是 HTTP 请求和响应中的重要组成部分。在创建 HTTP 请求时需要设置一些 HTTP 头。在得到 HTTP 的响应之后，也会需要对其中包含的 HTTP 头进行解析。从代码的角度来说，HTTP 头的数据结构是 Map<String, List<String>>类型。也就是说，对于每个 HTTP 头，可能有多个值。但是大部分 HTTP 头都只有一个值，只有少部分 HTTP 头允许多个值。OkHttp 采用了简单的方式来区分这两种类型，使得对 HTTP 头的使用更加简单。

在设置 HTTP 头时，使用 header(name, value) 方法来设置 HTTP 头的唯一值。对同一个 HTTP 头，多次调用该方法会覆盖之前设置的值。使用 addHeader(name, value) 方法来为 HTTP 头添加新的值。在读取 HTTP 头时，使用 header(name) 方法来读取 HTTP 头的最近出现的值。如果该 HTTP 头只有单个值，则返回该值；如果有多个值，则返回最后一个值。使用 headers(name) 方法来读取 HTTP 头的所有值。

[代码清单 3](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 3. HTTP %E5%A4%B4%E8%AE%BE%E7%BD%AE%E5%92%8C%E8%AF%BB%E5%8F%96%E7%9A%84%E7%A4%BA%E4%BE%8B) 中使用 header 方法设置了 User-Agent 头的值，并添加了一个 Accept 头的值。在进行解析时，通过 header 方法来获取 Server 头的单个值，通过 headers 方法来获取 Set-Cookie 头的所有值。

##### 清单 3. HTTP 头设置和读取的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Headers {     public static void main(String[] args) throws IOException {      OkHttpClient client = new OkHttpClient();        Request request = new Request.Builder()              .url("http://www.baidu.com")              .header("User-Agent", "My super agent")              .addHeader("Accept", "text/html")              .build();        Response response = client.newCall(request).execute();      if (!response.isSuccessful()) {          throw new IOException("服务器端错误: " + response);      }        System.out.println(response.header("Server"));      System.out.println(response.headers("Set-Cookie"));     }  } |

## POST 请求

HTTP POST 和 PUT 请求可以包含要提交的内容。只需要在创建 Request 对象时，通过 post 和 put 方法来指定要提交的内容即可。[代码清单 4](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 4. HTTP POST %E8%AF%B7%E6%B1%82%E7%9A%84%E5%9F%BA%E6%9C%AC%E7%A4%BA%E4%BE%8B) 中通过 RequestBody 的 create 方法来创建媒体类型为 text/plain 的内容并提交。

##### 清单 4. HTTP POST 请求的基本示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class PostString {     public static void main(String[] args) throws IOException {      OkHttpClient client = new OkHttpClient();      MediaType MEDIA\_TYPE\_TEXT = MediaType.parse("text/plain");      String postBody = "Hello World";        Request request = new Request.Builder()              .url("http://www.baidu.com")              .post(RequestBody.create(MEDIA\_TYPE\_TEXT, postBody))              .build();        Response response = client.newCall(request).execute();      if (!response.isSuccessful()) {          throw new IOException("服务器端错误: " + response);      }        System.out.println(response.body().string());     }  } |

以 String 类型提交内容只适用于内容比较小的情况。当请求内容较大时，应该使用流来提交。[代码清单 5](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 5. %E4%BD%BF%E7%94%A8%E6%B5%81%E6%96%B9%E6%B3%95%E6%8F%90%E4%BA%A4 HTTP POST %E8%AF%B7%E6%B1%82%E7%9A%84%E7%A4%BA%E4%BE%8B) 中给了使用流方式来提交内容的示例。这里创建了 RequestBody 的一个匿名子类。该子类的 contentType 方法需要返回内容的媒体类型，而 writeTo 方法的参数是一个 BufferedSink 对象。我们需要做的就是把请求的内容写入到 BufferedSink 对象即可。

##### 清单 5. 使用流方法提交 HTTP POST 请求的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | public class PostStream {     public static void main(String[] args) throws IOException {      OkHttpClient client = new OkHttpClient();      final MediaType MEDIA\_TYPE\_TEXT = MediaType.parse("text/plain");      final String postBody = "Hello World";        RequestBody requestBody = new RequestBody() {          @Override          public MediaType contentType() {              return MEDIA\_TYPE\_TEXT;          }            @Override          public void writeTo(BufferedSink sink) throws IOException {              sink.writeUtf8(postBody);          }     @Override          public long contentLength() throws IOException {              return postBody.length();          }      };        Request request = new Request.Builder()              .url("http://www.baidu.com")              .post(requestBody)              .build();        Response response = client.newCall(request).execute();      if (!response.isSuccessful()) {          throw new IOException("服务器端错误: " + response);      }        System.out.println(response.body().string());     }  } |

如果所要提交的内容来自本地文件，则不需要额外的流操作，只需要通过 RequestBody 的 create 方法，并把 File 类型的对象作为参数传入即可。

如果需要模拟 HTML 中的表单提交，可以通过 FormEncodingBuilder 来创建请求内容，见[代码清单 6](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 6. %E8%A1%A8%E5%8D%95%E6%8F%90%E4%BA%A4%E7%A4%BA%E4%BE%8B)。

##### 清单 6. 表单提交示例

|  |  |
| --- | --- |
| 1  2  3 | RequestBody formBody = new FormEncodingBuilder()              .add("query", "Hello")              .build(); |

如果需要模拟 HTML 中的文件上传功能，可以通过 MultipartBuilder 来创建 multipart 请求内容。[代码清单 7](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 7. %E6%8F%90%E4%BA%A4 multipart %E8%AF%B7%E6%B1%82%E7%9A%84%E7%A4%BA%E4%BE%8B) 中的 multipart 请求添加了一个表单属性 title 和一个文件 file。

##### 清单 7. 提交 multipart 请求的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | MediaType MEDIA\_TYPE\_TEXT = MediaType.parse("text/plain");  RequestBody requestBody = new MultipartBuilder()      .type(MultipartBuilder.FORM)      .addPart(              Headers.of("Content-Disposition", "form-data; name=\"title\""),              RequestBody.create(null, "测试文档"))      .addPart(              Headers.of("Content-Disposition", "form-data; name=\"file\""),              RequestBody.create(MEDIA\_TYPE\_TEXT, new File("input.txt")))      .build(); |

## 响应缓存

OkHttp 可以对 HTTP 响应的内容在磁盘上进行缓存。在进行 HTTP 请求时，如果该请求的响应已经被缓存而且没有过期，OkHttp 会直接使用缓存中的响应内容，而不需要真正的发出 HTTP 请求到远程服务器。在创建缓存时需要指定一个磁盘目录和缓存的大小。在[代码清单 8](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 8. %E8%AE%BE%E7%BD%AE%E5%93%8D%E5%BA%94%E7%BC%93%E5%AD%98%E7%9A%84%E7%A4%BA%E4%BE%8B) 中，创建出 Cache 对象之后，通过 OkHttpClient 的 setCache 进行设置。通过 Response 对象的 cacheResponse 和 networkResponse 方法可以得到缓存的响应和从实际的 HTTP 请求得到的响应。如果该请求的响应来自实际的网络请求，则 cacheResponse 方法的返回值为 null；如果响应来自缓存，则 networkResponse 的返回值为 null。OkHttp 在进行缓存时会遵循 HTTP 协议的要求，因此可以通过标准的 HTTP 头 Cache-Control 来控制响应的缓存时间。

##### 清单 8. 设置响应缓存的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class CacheResponse {     public static void main(String[] args) throws IOException {      int cacheSize = 100 \* 1024 \* 1024;      File cacheDirectory = Files.createTempDirectory("cache").toFile();      Cache cache = new Cache(cacheDirectory, cacheSize);      OkHttpClient client = new OkHttpClient();      client.setCache(cache);        Request request = new Request.Builder()              .url("http://www.baidu.com")              .build();        Response response = client.newCall(request).execute();      if (!response.isSuccessful()) {          throw new IOException("服务器端错误: " + response);      }        System.out.println(response.cacheResponse());      System.out.println(response.networkResponse());     }  } |

## 用户认证

OkHttp 提供了对用户认证的支持。当 HTTP 响应的状态代码是 401 时，OkHttp 会从设置的 Authenticator 对象中获取到新的 Request 对象并再次尝试发出请求。Authenticator 接口中的 authenticate 方法用来提供进行认证的 Request 对象，authenticateProxy 方法用来提供对代理服务器进行认证的 Request 对象。[代码清单 9](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 9. %E7%94%A8%E6%88%B7%E8%AE%A4%E8%AF%81%E7%9A%84%E7%A4%BA%E4%BE%8B) 中新的 Request 对象中添加了 HTTP 基本认证的 Authorization 头。

##### 清单 9. 用户认证的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | OkHttpClient client = new OkHttpClient();  client.setAuthenticator(new Authenticator() {  public Request authenticate(Proxy proxy, Response response) throws IOException {      String credential = Credentials.basic("user", "password");      return response.request().newBuilder()              .header("Authorization", credential)              .build();  }    public Request authenticateProxy(Proxy proxy, Response response)  throws IOException {      return null;  }  }); |

## 异步调用

OkHttp 除了支持常用的同步 HTTP 请求之外，还支持异步 HTTP 请求调用。在使用同步调用时，当前线程会被阻塞，直到 HTTP 请求完成。当同时发出多个 HTTP 请求时，同步调用的性能会比较差。这个时候通过异步调用可以提高整体的性能。

[代码清单 10](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 10. %E5%BC%82%E6%AD%A5%E8%B0%83%E7%94%A8%E7%9A%84%E7%A4%BA%E4%BE%8B) 给出了异步调用的示例。在通过 newCall 方法创建一个新的 Call 对象之后，不是通过 execute 方法来同步执行，而是通过 enqueue 方法来添加到执行队列中。在调用 enqueue 方法时需要提供一个 Callback 接口的实现。在 Callback 接口实现中，通过 onResponse 和 onFailure 方法来处理响应和进行错误处理。

##### 清单 10. 异步调用的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class AsyncGet {     public static void main(String[] args) throws IOException {      OkHttpClient client = new OkHttpClient();        Request request = new Request.Builder()              .url("http://www.baidu.com")              .build();        client.newCall(request).enqueue(new Callback() {          public void onFailure(Request request, IOException e) {              e.printStackTrace();          }            public void onResponse(Response response) throws IOException {              if (!response.isSuccessful()) {                  throw new IOException("服务器端错误: " + response);              }                System.out.println(response.body().string());          }  });     }  } |

## 取消调用

当一个 HTTP 调用执行之后，可以通过 Call 接口的 cancel 方法来取消该请求。当一个调用被取消之后，等待该请求的响应的代码会收到 IOException。同步和异步调用都可以被取消。如果需要同时取消多个请求，可以在创建请求时通过 RequestBuilder 的 tag 方法来为请求添加一个标签。在需要时可以通过 OkHttpClient 的 cancel 方法来取消拥有一个标签的所有请求。[代码清单 11](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 11. %E5%8F%96%E6%B6%88 HTTP %E8%AF%B7%E6%B1%82%E7%9A%84%E7%A4%BA%E4%BE%8B) 中的所有请求都设置了标签 website，可以通过 cancel 方法来全部取消。

##### 清单 11. 取消 HTTP 请求的示例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class CancelRequest {  private OkHttpClient client = new OkHttpClient();  private String tag = "website";    public void sendAndCancel() {      sendRequests(Lists.newArrayList(              "http://www.baidu.com",              "http://www.163.com",              "http://www.sina.com.cn"));      client.cancel(this.tag);  }    public void sendRequests(List<String> urls) {      urls.forEach(url -> {          client.newCall(new Request.Builder()                      .url(url)                      .tag(this.tag)                      .build())                  .enqueue(new SimpleCallback());      });  }    private static class SimpleCallback implements Callback {        public void onFailure(Request request, IOException e) {          e.printStackTrace();      }        public void onResponse(Response response) throws IOException {          System.out.println(response.body().string());      }  }    public static void main(String[] args) throws IOException {      new CancelRequest().sendAndCancel();  }  } |

## 拦截器

拦截器是 OkHttp 提供的对 HTTP 请求和响应进行统一处理的强大机制。拦截器在实现和使用上类似于 Servlet 规范中的过滤器。多个拦截器可以链接起来，形成一个链条。拦截器会按照在链条上的顺序依次执行。 拦截器在执行时，可以先对请求的 Request 对象进行修改；再得到响应的 Response 对象之后，可以进行修改之后再返回。

[代码清单 12](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 12. %E8%AE%B0%E5%BD%95%E8%AF%B7%E6%B1%82%E5%92%8C%E5%93%8D%E5%BA%94%E4%BF%A1%E6%81%AF%E7%9A%84%E6%8B%A6%E6%88%AA%E5%99%A8) 中的拦截器 LoggingInterceptor 用来记录 HTTP 请求和响应的相关信息。Interceptor 接口只包含一个方法 intercept，其参数是 Chain 对象。Chain 对象表示的是当前的拦截器链条。通过 Chain 的 request 方法可以获取到当前的 Request 对象。在使用完 Request 对象之后，通过 Chain 对象的 proceed 方法来继续拦截器链条的执行。当执行完成之后，可以对得到的 Response 对象进行额外的处理。

##### 清单 12. 记录请求和响应信息的拦截器

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class LoggingInterceptor implements Interceptor {  public Response intercept(Chain chain) throws IOException {      Request request = chain.request();        long t1 = System.nanoTime();      System.out.println(String.format("发送请求: [%s] %s%n%s",              request.url(), chain.connection(), request.headers()));        Response response = chain.proceed(request);        long t2 = System.nanoTime();      System.out.println(String.format("接收响应: [%s] %.1fms%n%s",              response.request().url(), (t2 - t1) / 1e6d, response.headers()));        return response;  }  } |

OkHttp 中的拦截器分成应用和网络拦截器两种。应用拦截器对于每个 HTTP 响应都只会调用一次，可以通过不调用 Chain.proceed 方法来终止请求，也可以通过多次调用 Chain.proceed 方法来进行重试。网络拦截器对于调用执行中的自动重定向和重试所产生的响应也会被调用，而如果响应来自缓存，则不会被调用。应用和网络拦截器的添加方式见[代码清单 13](https://www.ibm.com/developerworks/cn/java/j-lo-okhttp/index.html" \l "_%E6%B8%85%E5%8D%95 13. %E6%B7%BB%E5%8A%A0%E5%BA%94%E7%94%A8%E5%92%8C%E7%BD%91%E7%BB%9C%E6%8B%A6%E6%88%AA%E5%99%A8)。

##### 清单 13. 添加应用和网络拦截器

|  |  |
| --- | --- |
| 1  2  3 | client.interceptors().add(new LoggingInterceptor()); //添加应用拦截器    client.networkInterceptors().add(new LoggingInterceptor()); //添加网络拦截器 |

## 小结

OkHttp 作为一个简洁高效的 HTTP 客户端，可以在 Java 和 Android 程序中使用。相对于 Apache HttpClient 来说，OkHttp 的性能更好，其 API 设计也更加简单实用。本文对 OkHttp 进行了详细的介绍，包括同步和异步调用、HTTP GET 和 POST 请求处理、用户认证、响应缓存和拦截器等。对于新开发的 Java 应用，推荐使用 OkHttp 作为 HTTP 客户端。