Okhttp3基本使用

Jdqm (/u/a04b98b0f913) ＋关注
2018.03.28 11:50* 字数 1549 阅读 81647 评论 16 喜欢 71
(/u/a04b98b0f913)

I.简介

HTTP是现代应用常用的一种交换数据和媒体的网络方式，高效地使用HTTP能让资源加载更快，节省带宽。OkHttp是一个高效的HTTP客户端，它有以下默认特性：

- 支持HTTP/2，允许所有同一个主机地址的请求共享同一个socket连接
- 连接池减少请求延时
- 透明的GZIP压缩减少响应数据的大小
- 缓存响应内容，避免一些完全重复的请求

当网络出现问题的时候OkHttp依然坚守自己的职责，它会自动恢复一般的连接问题，如果你的服务有多个IP地址，当第一个IP请求失败时，OkHttp会交替尝试你配置的其他IP，OkHttp使用现代TLS技术(SNI, ALPN)初始化新的连接，当握手失败时会回退到TLS 1.0。

> note: OkHttp 支持 Android 2.3 及以上版本Android平台， 对于 Java, JDK 1.7及以上.

对于Okhttp3的源码阅读预计会写3篇文章来总结：

- 1.Okhttp的基本使用 (https://www.jianshu.com/p/da4a806e599b)
- 2.Okhttp主流程源码分析 (https://www.jianshu.com/p/b0353ed71151)
- 3.Okhttp3架构分析，主要通过一些流程图类展现 (https://www.jianshu.com/p/9deec36f2759)

II.使用

OkHttp的使用是非常简单的. 它的请求/响应 API 使用构造器模式builders来设计，它支持阻塞式的同步请求和带回调的异步请求。

Download OkHttp3

```
implementation 'com.squareup.okhttp3:okhttp:3.10.0'
```

当你看到这的时候，可能最新的稳定版已经不是 `3.10.0` 了，你需要移步官方GitHub来查看最新版本。 官方地址 https://github.com/square/okhttp (https://github.com/square/okhttp)，另外不要忘了在清单文件声明访问Internet的权限，如果使用 `DiskLruCache` ，那还得声明写外存的权限。

1.1. 异步GET请求

-new OkHttpClient;
-构造Request对象；
-通过前两步中的对象构建Call对象；
-通过Call#enqueue(Callback)方法来提交异步请求；

```
String url = "http://wwww.baidu.com";
OkHttpClient okHttpClient = new OkHttpClient();
final Request request = new Request.Builder()
        .url(url)
        .get()//默认就是GET请求，可以不写
        .build();
Call call = okHttpClient.newCall(request);
call.enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: ");
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, "onResponse: " + response.body().string());
    }
});
```

异步发起的请求会被加入到 `Dispatcher` 中的 `runningAsyncCalls` 双端队列中通过线程池来执行。

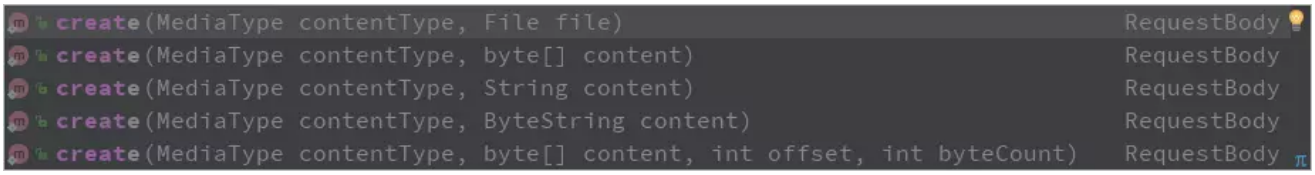
1.2. 同步GET请求

前面几个步骤和异步方式一样，只是最后一部是通过 `Call#execute()` 来提交请求，注意这种方式会阻塞调用线程，所以在Android中应放在子线程中执行，否则有可能引起ANR异常， `Android3.0` 以后已经不允许在主线程访问网络。

```
String url = "http://wwww.baidu.com";
OkHttpClient okHttpClient = new OkHttpClient();
final Request request = new Request.Builder()
        .url(url)
        .build();
final Call call = okHttpClient.newCall(request);
new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            Response response = call.execute();
            Log.d(TAG, "run: " + response.body().string());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}).start();
```

2.1. POST方式提交String

这种方式与前面的区别就是在构造Request对象时，需要多构造一个RequestBody对象，用它来携带我们要提交的数据。在构造 `RequestBody` 需要指定 `MediaType` ，用于描述请求/响应 `body` 的内容类型，关于 `MediaType` 的更多信息可以查看 RFC 2045 (https://tools.ietf.org/html/rfc2045)，RequstBody的几种构造方式：



request_body

```
MediaType mediaType = MediaType.parse("text/x-markdown; charset=utf-8");
String requestBody = "I am Jdqm.";
Request request = new Request.Builder()
        .url("https://api.github.com/markdown/raw")
        .post(RequestBody.create(mediaType, requestBody))
        .build();
OkHttpClient okHttpClient = new OkHttpClient();
okHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: " + e.getMessage());
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, response.protocol() + " " +response.code() + " " + response.messag
        Headers headers = response.headers();
        for (int i = 0; i < headers.size(); i++) {
            Log.d(TAG, headers.name(i) + ":" + headers.value(i));
        }
        Log.d(TAG, "onResponse: " + response.body().string());
    }
});
```

响应内容

```
http/1.1 200 OK
Date:Sat, 10 Mar 2018 05:23:20 GMT
Content-Type:text/html;charset=utf-8
Content-Length:18
Server:GitHub.com
Status:200 OK
X-RateLimit-Limit:60
X-RateLimit-Remaining:52
X-RateLimit-Reset:1520661052
X-CommonMarker-Version:0.17.4
Access-Control-Expose-Headers:ETag, Link, Retry-After, X-GitHub-OTP, X-RateLimit-Limi

Access-Control-Allow-Origin:*
Content-Security-Policy:default-src 'none'
Strict-Transport-Security:max-age=31536000; includeSubdomains; preload
X-Content-Type-Options:nosniff
X-Frame-Options:deny
X-XSS-Protection:1; mode=block
X-Runtime-rack:0.019668
Vary:Accept-Encoding
X-GitHub-Request-Id:1474:20A83:5CC0B6:7A7C1B:5AA36BC8
onResponse: <p>I am Jdqm.</p>
```

2.2 POST方式提交流

```
RequestBody requestBody = new RequestBody() {
    @Nullable
    @Override
    public MediaType contentType() {
        return MediaType.parse("text/x-markdown; charset=utf-8");
    }

    @Override
    public void writeTo(BufferedSink sink) throws IOException {
        sink.writeUtf8("I am Jdqm.");
    }
};

Request request = new Request.Builder()
        .url("https://api.github.com/markdown/raw")
        .post(requestBody)
        .build();
OkHttpClient okHttpClient = new OkHttpClient();
okHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: " + e.getMessage());
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, response.protocol() + " " +response.code() + " " + response.messag
        Headers headers = response.headers();
        for (int i = 0; i < headers.size(); i++) {
            Log.d(TAG, headers.name(i) + ":" + headers.value(i));
        }
        Log.d(TAG, "onResponse: " + response.body().string());
    }
});
```

2.3. POST提交文件

```
MediaType mediaType = MediaType.parse("text/x-markdown; charset=utf-8");
OkHttpClient okHttpClient = new OkHttpClient();
File file = new File("test.md");
Request request = new Request.Builder()
        .url("https://api.github.com/markdown/raw")
        .post(RequestBody.create(mediaType, file))
        .build();
okHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: " + e.getMessage());
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, response.protocol() + " " +response.code() + " " + response.messag
        Headers headers = response.headers();
        for (int i = 0; i < headers.size(); i++) {
            Log.d(TAG, headers.name(i) + ":" + headers.value(i));
        }
        Log.d(TAG, "onResponse: " + response.body().string());
    }
});
```

2.4. POST方式提交表单

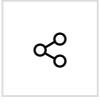
```
OkHttpClient okHttpClient = new OkHttpClient();
RequestBody requestBody = new FormBody.Builder()
        .add("search", "Jurassic Park")
        .build();
Request request = new Request.Builder()
        .url("https://en.wikipedia.org/w/index.php")
        .post(requestBody)
        .build();

okHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: " + e.getMessage());
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        Log.d(TAG, response.protocol() + " " +response.code() + " " + response.messag
        Headers headers = response.headers();
        for (int i = 0; i < headers.size(); i++) {
            Log.d(TAG, headers.name(i) + ":" + headers.value(i));
        }
        Log.d(TAG, "onResponse: " + response.body().string());
    }
});
```

提交表单时，使用 `RequestBody` 的实现类 `FormBody` 来描述请求体，它可以携带一些经过编码的 `key-value` 请求体，键值对存储在下面两个集合中：

```
private final List<String> encodedNames;
private final List<String> encodedValues;
```

2.5. POST方式提交分块请求

MultipartBody 可以构建复杂的请求体，与HTML文件上传形式兼容。多块请求体中每块请求都是一个请求体，可以定义自己的请求头。这些请求头可以用来描述这块请求，例如它的 `Content-Disposition` 。如果 `Content-Length` 和 `Content-Type` 可用的话，他们会被自动添加到请求头中。



```
private static final String IMGUR_CLIENT_ID = "...";
private static final MediaType MEDIA_TYPE_PNG = MediaType.parse("image/png");

private void postMultipartBody() {
    OkHttpClient client = new OkHttpClient();


    // Use the imgur image upload API as documented at https://api.imgur.com/endpoint
    MultipartBody body = new MultipartBody.Builder("AaB03x")
            .setType(MultipartBody.FORM)
            .addPart(
                    Headers.of("Content-Disposition", "form-data; name=\"title\""),
                    RequestBody.create(null, "Square Logo"))
            .addPart(
                    Headers.of("Content-Disposition", "form-data; name=\"image\""),
                    RequestBody.create(MEDIA_TYPE_PNG, new File("website/static/logo-
            .build();

    Request request = new Request.Builder()
            .header("Authorization", "Client-ID " + IMGUR_CLIENT_ID)
            .url("https://api.imgur.com/3/image")
            .post(body)
            .build();

    Call call = client.newCall(request);
    call.enqueue(new Callback() {
        @Override
        public void onFailure(Call call, IOException e) {

        }

        @Override
        public void onResponse(Call call, Response response) throws IOException {
            System.out.println(response.body().string());

        }

    });
}
```

III.拦截器-interceptor

OkHttp的拦截器链可谓是其整个框架的精髓，用户可传入的 `interceptor` 分为两类：
①一类是全局的 `interceptor` ，该类 `interceptor` 在整个拦截器链中最早被调用，通过 `OkHttpClient.Builder#addInterceptor(Interceptor)` 传入；
②另外一类是非网页请求的 `interceptor` ，这类拦截器只会在非网页请求中被调用，并且是在组装完请求之后，真正发起网络请求前被调用，所有的 `interceptor` 被保存在 `List<Interceptor> interceptors` 集合中，按照添加顺序来逐个调用，具体可参考 `RealCall#getResponseWithInterceptorChain()` 方法。通过 `OkHttpClient.Builder#addNetworkInterceptor(Interceptor)` 传入；

这里举一个简单的例子，例如有这样一个需求，我要监控App通过 `OkHttp` 发出的所有原始请求，以及整个请求所耗费的时间，针对这样的需求就可以使用第一类全局的 `interceptor` 在拦截器链头去做。

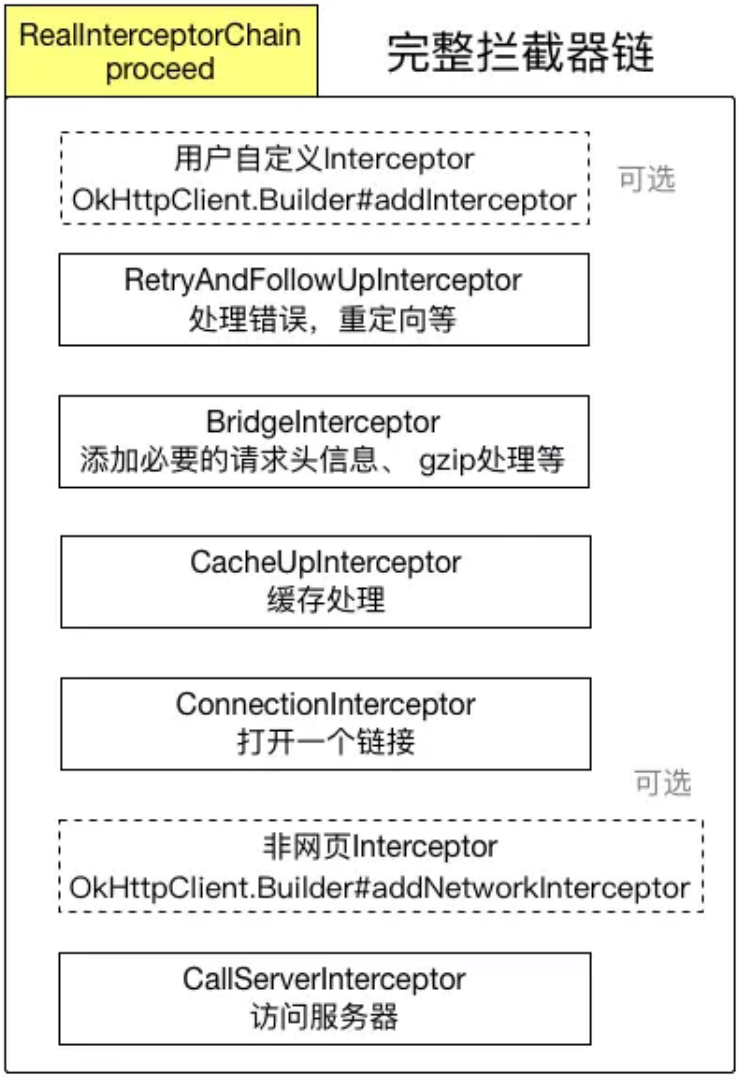
```
OkHttpClient okHttpClient = new OkHttpClient.Builder()
        .addInterceptor(new LoggingInterceptor())
        .build();
Request request = new Request.Builder()
        .url("http://www.publicobject.com/helloworld.txt")
        .header("User-Agent", "OkHttp Example")
        .build();
okHttpClient.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        Log.d(TAG, "onFailure: " + e.getMessage());
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        ResponseBody body = response.body();
        if (body != null) {
            Log.d(TAG, "onResponse: " + response.body().string());
            body.close();
        }
    }
});
```

```
public class LoggingInterceptor implements Interceptor {
    private static final String TAG = "LoggingInterceptor";

    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();

        long startTime = System.nanoTime();
        Log.d(TAG, String.format("Sending request %s on %s%n%s",
                request.url(), chain.connection(), request.headers()));

        Response response =  chain.proceed(request);

        long endTime = System.nanoTime();
        Log.d(TAG, String.format("Received response for %s in %.1fms%n%s",
                response.request().url(), (endTime - startTime) / 1e6d, response.hea

        return response;
    }
}
```

针对这个请求，打印出来的结果

```
Sending request http://www.publicobject.com/helloworld.txt on null
User-Agent: OkHttp Example

Received response for https://publicobject.com/helloworld.txt in 1265.9ms
Server: nginx/1.10.0 (Ubuntu)
Date: Wed, 28 Mar 2018 08:19:48 GMT
Content-Type: text/plain
Content-Length: 1759
Last-Modified: Tue, 27 May 2014 02:35:47 GMT
Connection: keep-alive
ETag: "5383fa03-6df"
Accept-Ranges: bytes
```

注意到一点是这个请求做了重定向，原始的 `request url` 是 `http://www.publicobject.com/helloworld.tx` ，而响应的 `request url` 是 `https://publicobject.com/helloworld.txt` ，这说明一定发生了重定向，但是做了几次重定向其实我们这里是不知道的，要知道这些的话，可以使用 `addNetworkInterceptor()` 去做。更多的关于 `interceptor` 的使用以及它们各自的优缺点，请移步OkHttp官方说明文档 (https://github.com/square/okhttp/wiki/Interceptors)。



完整interceptor-chain

IV. 自定义dns服务

Okhttp默认情况下使用的是系统

V.其他

1. 推荐让 `OkHttpClient` 保持单例，用同一个 `OkHttpClient` 实例来执行你的所有请求，因为每一个 `OkHttpClient` 实例都拥有自己的连接池和线程池，重用这些资源可以减少延时和节省资源，如果为每个请求创建一个 `OkHttpClient` 实例，显然就是一种资源的浪费。当然，也可以使用如下的方式来创建一个新的 `OkHttpClient` 实例，它们共享连接池、线程池和配置信息。

```
OkHttpClient eagerClient = client.newBuilder()
    .readTimeout(500, TimeUnit.MILLISECONDS)
    .build();
Response response = eagerClient.newCall(request).execute();
```

2. 每一个Call（其实现是RealCall）只能执行一次，否则会报异常，具体参见

   `RealCall#execute()`