探索Android开源框架与源码解析

1. OkHttp源码解析篇

使用

1. 添加依赖

```
implementation 'com.squareup.okhttp3:okhttp:3.14.9'
```

2. 常用请求方法

1. 同步GET请求

- 执行请求的操作是阻塞式的，直到http响应返回

1. 创建OkHttpClient对象

1. 直接创建

```
val client = OkHttpClient()
```

2. 通过Builder模式创建

```
val client = OkHttpClient.Builder()
    .build()
```

2. 创建Request对象

```
val request = Request.Builder()
    .url("https://www.baidu.com")
    .get()
    .build()
```

3. 将request封装成call对象

```
val call = client.newCall(request)
```

4. 调用call.execute发送同步请求

```
val response = call.execute()
if (response.isSuccessful) {
    log(response.body()?.string())
} else {
    log(IOException("Unexpected code $response").message)
}
```

- 注意：需要在子线程调用，发送请求后，当前线程就会进入阻塞状态，直到收到响应

```
lifecycleScope.launch {
    withContext(Dispatchers.IO) {
        getSync()
    }
}
```

- 别忘了添加网络请求权限

```
<uses-permission android:name="android.permission.INTERNET" />
```

- 如果是非https请求，可能会报错：java.net.UnknownServiceException: CLEARTEXT communication to。。。
- CLEARTEXT，就是明文的意思，在Android P系统的设备上，如果应用使用的是非加密的明文流量的http网络请求，则会导致该应用无法进行网络请求， https则不会受影响，同样地，如果应用嵌套了webView，webView也只能使用https请求;
- 解决该异常需要改为https请求，或者在 AndroidManifest.xml文件的Application标签中加入 android:usesCleartextTraffic="true"

2. 异步get请求

- 执行请求的操作是非阻塞式的，执行结果通过接口回调方式告知调用者
- 前三步是一样的，第四步调用异步方法 call.enqueue

```
val client = OkHttpClient()
val request = Request.Builder()
    .url("https://www.baidu.com")
    .get()
    .build()
val call = client.newCall(request)
//调用异步方法enqueue
call.enqueue(object : Callback {
    override fun onFailure(call: Call, e: IOException) {
        log("onFailure:${e.message}")
        runOnUiThread { tv.text = e.message }
    }

    override fun onResponse(call: Call, response: Response) {
        val result = response.body()?.string()
        log("onResponse:${result}")
        runOnUiThread { tv.text = "onResponse${result}" }
    }

})
```

- 注意：回调方法onResponse，onFailure是在 子线程/工作线程 中执行的, 所以onResponse中使用了runOnUiThread来更新UI;

3. 异步POST请求提交键值对

- 多了一步创建FormBody，为POST请求的参数

```
val client = OkHttpClient()
//创建FormBody
```

```
val formBody = FormBody.Builder()
    .add("k", "wanAndroid")
    .build()
val request = Request.Builder()
    .url("https://www.wanandroid.com/article/query/0/json")
    .post(formBody)
    .build()
val call = client.newCall(request)
call.enqueue(object : Callback {
    override fun onFailure(call: Call, e: IOException) {
        log("onFailure:${e.message}")
        runOnUiThread { tv.text = e.message }
    }

    override fun onResponse(call: Call, response: Response) {
        val result = response.body()?.string()
        log("onResponse:${result}")
        runOnUiThread { tv.text = "onResponse${result}" }
    }
})
```

4. Post方式提交流（上传文件）

```
private fun postFile() {
    val client = OkHttpClient()
    //获取要上传的文件
    val file=File(externalCacheDir,"ljy.txt")
    //创建RequestBody：
    val requestBody=RequestBody.create(
        MediaType.parse("text/x-markdown; charset=utf-8"),
        file
    )
    val request=Request.Builder()
        .url("https://api.github.com/markdown/raw")
        .post(requestBody)
        .build()
    client.newCall(request).enqueue(object : Callback{
        override fun onFailure(call: Call, e: IOException) {
            log("onFailure:${e.message}")
        }

        override fun onResponse(call: Call, response: Response) {
            log("onResponse:${ response.body()?.string()}")
        }
    })
}
```

- 需要在AndroidManifest.xml中添加读写权限, 和运行时权限申请

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.WRITE_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED
```

```
    && ActivityCompat.checkSelfPermission(this,
Manifest.permission.READ_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED
) {
    this@MainActivity.requestPermissions(arrayOf(
            Manifest.permission.WRITE_EXTERNAL_STORAGE,
            Manifest.permission.READ_EXTERNAL_STORAGE), 10001)
} else {
    ...
}

override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out
String>, grantResults: IntArray
) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    if (requestCode == 10001) {
        ...
    }
}
```

5. 异步下载文件

```
private fun downloadFile() {
    val client = OkHttpClient()
    val url = "https://pic3.zhimg.com/v2-
dc32dcddfd7e78e56cc4b6f689a24979_xl.jpg"
    val request = Request.Builder()
        .url(url)
        .build()
    client.newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) {
            log("onFailure:${e.message}")
        }

        override fun onResponse(call: Call, response: Response) {
            val inputStream = response.body()?.byteStream()
            val fileOutputStream = FileOutputStream(File(externalCacheDir,
"ljy.jpg"))
            val buffer = ByteArray(2048)
            var len: Int
            while (inputStream?.read(buffer).also { len = it ?: -1 } != -1) {
                fileOutputStream.write(buffer, 0, len)
            }
            fileOutputStream.flush()
            log("文件下载成功")
        }
    })
}
```

6. Post提交表单

- 有时会上传文件同时还需要传其他类型的字段

```
private fun sendMultipart() {
    val client = OkHttpClient()
    val file = File(externalCacheDir, "ljy.jpg")
```

```
    val requestBody: RequestBody = MultipartBody.Builder()
        .setType(MultipartBody.FORM)
        .addFormDataPart("name", "ljy")
        .addFormDataPart("age", "18")
        .addFormDataPart(
            "image", "header.jpg",
            RequestBody.create(MediaType.parse("image/png"), file)
        )
        .build()
    val request: Request = Request.Builder()
        .header("Authorization", "Client-ID " + "...")
        .url("https://api.imgur.com/3/image")
        .post(requestBody)
        .build()
    client.newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) {
            log("onFailure:${e.message}")
        }

        override fun onResponse(call: Call, response: Response) {
            log("onResponse:${response.body()?.string()}")
        }
    })
}
```

3.常用设置

1. 设置超时时间

```
val client = OkHttpClient.Builder()
    .connectTimeout(30,TimeUnit.SECONDS)
    .readTimeout(60,TimeUnit.SECONDS)
    .writeTimeout(90,TimeUnit.SECONDS)
    .build()
```

2. 设置缓存

```
//设置缓存路径和大小，及缓存拦截器
val client = OkHttpClient.Builder()
    .addNetworkInterceptor(CacheInterceptor())
    .cache(
        Cache(
            File(cacheDir, "httpCache2"),
            100 * 1024 * 1024L
        )
    ).build()

//缓存拦截器
class CacheInterceptor : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        var request: Request = chain.request()
        val var10000: Response
        val response: Response
        if (NetUtil.isNetworkAvailable(this@OkHttpDemoActivity)) {
            //如果有网，返回一个30内有效的响应，则30秒内同一请求会直接从缓存中读取
```

```
            response = chain.proceed(request)
            //构建maxAge = 30秒的CacheControl
            val cacheControl = CacheControl.Builder()
                .maxAge(30, TimeUnit.SECONDS)
                .build()
                .toString()
            var10000 = response.newBuilder()
                .removeHeader("Pragma")
                .removeHeader("Cache-Control") //填入30秒的CacheControl
                .header("Cache-Control", cacheControl)
                .build()
        } else {
            //如果没网，用原来的请求重新构建一个强制从缓存中读取的请求
            request = request.newBuilder()
                .cacheControl(CacheControl.FORCE_CACHE)
                .build()
            var10000 = chain.proceed(request)
        }
        return var10000
    }
}
```

- OkHttpClient.cache的入参Cache构造函数如下：

```
public Cache(File directory, long maxSize) {
    this(directory, maxSize, FileSystem.SYSTEM);
}

Cache(File directory, long maxSize, FileSystem fileSystem) {
    this.cache = DiskLruCache.create(fileSystem, directory, VERSION, ENTRY_COUNT,
maxSize);
}
```

- 可以看到也是用的DiskLruCache；

3. 设置失败重试

```
val client = OkHttpClient.Builder()
    .retryOnConnectionFailure(true)
    .build()
```

4. 持久化cookie

```
//添加三方库依赖
implementation 'com.zhy:okhttputils:2.6.2'
//持久化cookie,保持session会话：
val cookieJar = new CookieJarImpl(new
PersistentCookieStore(CommonModule.getAppContext()))
val client = OkHttpClient.Builder()
    .cookieJar(cookieJar)
    .build()
```

源码解析

Request

1. Request.Builder()构造方法如下，method默认是GET

```
public Builder() {
  this.method = "GET";
  this.headers = new Headers.Builder();
}

public Request build() {
  if (url == null) throw new IllegalStateException("url == null");
  return new Request(this);
}

//Request构造方法
Request(Builder builder) {
    this.url = builder.url;
    this.method = builder.method;
    this.headers = builder.headers.build();
    this.body = builder.body;
    this.tags = Util.immutableMap(builder.tags);
}
```

1. Request.BUilder的post方法如下：

```
public Builder post(RequestBody body) {
  return method("POST", body);
}

public Builder method(String method, @Nullable RequestBody body) {
  if (method == null) throw new NullPointerException("method == null");
  if (method.length() == 0) throw new IllegalArgumentException("method.length()
== 0");
  if (body != null && !HttpMethod.permitsRequestBody(method)) {
    throw new IllegalArgumentException("method " + method + " must not have a
request body.");
  }
  if (body == null && HttpMethod.requiresRequestBody(method)) {
    throw new IllegalArgumentException("method " + method + " must have a request
body.");
  }
  this.method = method;
  this.body = body;
  return this;
}
```

OkHttpClient

1. OkHttpClient构造方法实现如下:

```
public OkHttpClient() {
    this(new Builder());
```

```
}

// builder的构造方法中提供了默认值：
public Builder() {
    dispatcher = new Dispatcher();
    protocols = DEFAULT_PROTOCOLS;
    connectionSpecs = DEFAULT_CONNECTION_SPECS;
    eventListenerFactory = EventListener.factory(EventListener.NONE);
    proxySelector = ProxySelector.getDefault();
    if (proxySelector == null) {
    proxySelector = new NullProxySelector();
    }
    cookieJar = CookieJar.NO_COOKIES;
    socketFactory = SocketFactory.getDefault();
    hostnameVerifier = OkHostnameVerifier.INSTANCE;
    certificatePinner = CertificatePinner.DEFAULT;
    proxyAuthenticator = Authenticator.NONE;
    authenticator = Authenticator.NONE;
    connectionPool = new ConnectionPool();
    dns = Dns.SYSTEM;
    followSslRedirects = true;
    followRedirects = true;
    retryOnConnectionFailure = true;
    callTimeout = 0;
    connectTimeout = 10_000;
    readTimeout = 10_000;
    writeTimeout = 10_000;
    pingInterval = 0;
}

public OkHttpClient build() {
    return new OkHttpClient(this);
}
```

1. OkHttpClient.newCall

```
@Override public Call newCall(Request request) {
    return RealCall.newRealCall(this, request, false /* for web socket */);
}
//其内部调用的RealCall.newRealCall：
static RealCall newRealCall(OkHttpClient client, Request originalRequest, boolean
forWebSocket) {
    // Safely publish the Call instance to the EventListener.
    RealCall call = new RealCall(client, originalRequest, forWebSocket);
    call.transmitter = new Transmitter(client, call);
    return call;
}

//RealCall构造方法如下：
private RealCall(OkHttpClient client, Request originalRequest, boolean
forWebSocket) {
    this.client = client;
    this.originalRequest = originalRequest;
    this.forWebSocket = forWebSocket;
}
```

Call

1. Call.execute同步请求方法源码

```
// Call的方法要看RealCall中的实现，dispatcher主要负责保存和移除同步请求
 @Override public Response execute() throws IOException {
    //判断executed，确保同一个HTTP请求只执行一次
    synchronized (this) {
      if (executed) throw new IllegalStateException("Already Executed");
      executed = true;
    }
    transmitter.timeoutEnter();
    transmitter.callStart();
    try {
       //调用dispatcher的executed将请求加入到同步请求队列中
      client.dispatcher().executed(this);
      //通过拦截器链获取response
      return getResponseWithInterceptorChain();
    } finally {
      //回收同步请求
      client.dispatcher().finished(this);
    }
}
```

1. Call.enqueue 异步请求方法源码

```
//RealCall中的实现:
@Override public void enqueue(Callback responseCallback) {
    synchronized (this) {
       //确保call只执行一次
      if (executed) throw new IllegalStateException("Already Executed");
      executed = true;
    }
    transmitter.callStart();
    client.dispatcher().enqueue(new AsyncCall(responseCallback));
}
```

- 可以看到他们都调用了dispatcher的方法

Dispatcher任务调度

- 用于控制并发的请求，主要维护了以下变量

```
/**
最大并发请求数
 */
private int maxRequests = 64;
/**
每个主机最大请求数
 */
private int maxRequestsPerHost = 5;
/**
消费者线程池
 */
private ExecutorService executorService;
```

```
/**
将要运行的异步请求队列
 */
private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();
/**
正在运行的异步请求队列
 */
private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();
/**
正在运行的同步请求队列
 */
private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();
```

- dispatcher的executed方法如下：

```
synchronized void executed(RealCall call) {
    //将请求加入到同步请求队列中
    runningSyncCalls.add(call);
}
```

- dispatcher().finished用于回收同步请求，实现如下：

```
void finished(RealCall call) {
    finished(runningSyncCalls, call);
}

private <T> void finished(Deque<T> calls, T call) {
    Runnable idleCallback;
    synchronized (this) {
       //移除同步请求
      if (!calls.remove(call)) throw new AssertionError("Call wasn't in-
flight!");
      idleCallback = this.idleCallback;
    }

    boolean isRunning = promoteAndExecute();

    if (!isRunning && idleCallback != null) {
      idleCallback.run();
    }
}
```

- dispatcher的enqueue方法如下：

```
void enqueue(AsyncCall call) {
    synchronized (this) {
       //将请求加入到准备好的异步请求队列中
      readyAsyncCalls.add(call);
      // Mutate the AsyncCall so that it shares the AtomicInteger of an existing
running call to
      // the same host.
      if (!call.get().forWebSocket) {
        //通过host查找已经存在的Call
        AsyncCall existingCall = findExistingCallWithHost(call.host());
```

```
        //如果存在则复用callsPerHost
        if (existingCall != null) call.reuseCallsPerHostFrom(existingCall);
      }
    }
    promoteAndExecute();
}
```

- 其入参AsyncCall是RealCall的内部类,构造函数入参就是我们传入的callback，并在execute方法中调用callback，而在NamedRunnable的run中调用了execute方法

```
final class AsyncCall extends NamedRunnable {
     ...
    void executeOn(ExecutorService executorService) {
      assert (!Thread.holdsLock(client.dispatcher()));
      boolean success = false;
      try {
        executorService.execute(this);
        success = true;
      } catch (RejectedExecutionException e) {
        InterruptedIOException ioException = new InterruptedIOException("executor
rejected");
        ioException.initCause(e);
        transmitter.noMoreExchanges(ioException);
        responseCallback.onFailure(RealCall.this, ioException);
      } finally {
        if (!success) {
          client.dispatcher().finished(this); // This call is no longer running!
        }
      }
    }

    @Override protected void execute() {
      boolean signalledCallback = false;
      transmitter.timeoutEnter();
      try {
        Response response = getResponseWithInterceptorChain();
        signalledCallback = true;
        responseCallback.onResponse(RealCall.this, response);
      } catch (IOException e) {
        if (signalledCallback) {
          // Do not signal the callback twice!
          Platform.get().log(INFO, "Callback failure for " + toLoggableString(),
e);
        } else {
          responseCallback.onFailure(RealCall.this, e);
        }
      } catch (Throwable t) {
        cancel();
        if (!signalledCallback) {
          IOException canceledException = new IOException("canceled due to " +
t);
          canceledException.addSuppressed(t);
          responseCallback.onFailure(RealCall.this, canceledException);
        }
        throw t;
```

```
      } finally {
        client.dispatcher().finished(this);
      }
    }
}

public abstract class NamedRunnable implements Runnable {
  protected final String name;

  public NamedRunnable(String format, Object... args) {
    this.name = Util.format(format, args);
  }

  @Override public final void run() {
    String oldName = Thread.currentThread().getName();
    Thread.currentThread().setName(name);
    try {
      execute();
    } finally {
      Thread.currentThread().setName(oldName);
    }
  }

  protected abstract void execute();
}
```

- 上面AsyncCall的execute中，在最后的finally中也调用了finished用于回收异步请求

```
void finished(AsyncCall call) {
    call.callsPerHost().decrementAndGet();
    finished(runningAsyncCalls, call);
}
```

- finished和异步中都调用了promoteAndExecute方法，其实现如下

```
private boolean promoteAndExecute() {
    assert (!Thread.holdsLock(this));

    //遍历准备好的异步请求队列，放到可执行的list和正在运行的队列中：
    List<AsyncCall> executableCalls = new ArrayList<>();
    boolean isRunning;
    synchronized (this) {
      for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {
        AsyncCall asyncCall = i.next();

        if (runningAsyncCalls.size() >= maxRequests) break; // Max capacity.
        if (asyncCall.callsPerHost().get() >= maxRequestsPerHost) continue; //
Host max capacity.

        i.remove();
        asyncCall.callsPerHost().incrementAndGet();
        executableCalls.add(asyncCall);
        runningAsyncCalls.add(asyncCall);
      }
```

```
      //重新计算待执行的同步异步请求数量
      isRunning = runningCallsCount() > 0;
    }

    //遍历可执行的AsyncCall list，调用executeOn方法传入线程池执行
    for (int i = 0, size = executableCalls.size(); i < size; i++) {
      AsyncCall asyncCall = executableCalls.get(i);
      asyncCall.executeOn(executorService());
    }

    return isRunning;
}

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

public synchronized int runningCallsCount() {
    return runningAsyncCalls.size() + runningSyncCalls.size();
}
```

异步请求的调用顺序：

1. 使用者调用Call.enqueue(Callback)；
2. Call.enqueue中调用了client.dispatcher().enqueue(new AsyncCall(responseCallback))；
3. dispatcher().enqueue调用promoteAndExecute；
4. promoteAndExecute中会遍历readyAsyncCalls，放到executableCalls和runningAsyncCalls中，并调用runningCallsCount重新计算待执行的同步异步请求数量，然后遍历executableCalls，调用asyncCall.executeOn(executorService())；
5. asyncCall.executeOn中调用executorService.execute(this)，其中this为runnable类型的asyncCall，最后会调用其run方法；
6. NamedRunnable的run方法中调用了execute方法，asyncCall中实现了execute方法；
7. asyncCall.execute中调用了 Response response = getResponseWithInterceptorChain()，并调用callback,最终调用dispatcher().finished；
8. dispatcher().finished中又调用了promoteAndExecute方法，直到队列中的请求都执行完毕；

拦截器链

- 拦截器是okhttp中一个强大的机制，可以实现网络监听，请求及响应重写，请求失败重试等功能；
- 上面的同步请求异步请求源码中都有调用getResponseWithInterceptorChain方法，其代码如下

```
Response getResponseWithInterceptorChain() throws IOException {
    // Build a full stack of interceptors.
    //创建一系列拦截器，并放入list中
    List<Interceptor> interceptors = new ArrayList<>();
    interceptors.addAll(client.interceptors());
    //1. 重试和失败重定向拦截器
```

```
    interceptors.add(new RetryAndFollowUpInterceptor(client));
    //2. 桥接适配拦截器（如补充请求头，编码方式，压缩方式）
    interceptors.add(new BridgeInterceptor(client.cookieJar()));
    //3. 缓存拦截器
    interceptors.add(new CacheInterceptor(client.internalCache()));
    //4. 连接拦截器
    interceptors.add(new ConnectInterceptor(client));
    if (!forWebSocket) {
      interceptors.addAll(client.networkInterceptors());
    }
    //5. 网络io流拦截器
    interceptors.add(new CallServerInterceptor(forWebSocket));

    //创建拦截器链chain，并执行chain.proceed方法
    Interceptor.Chain chain = new RealInterceptorChain(interceptors, transmitter,
null, 0,
        originalRequest, this, client.connectTimeoutMillis(),
        client.readTimeoutMillis(), client.writeTimeoutMillis());

    boolean calledNoMoreExchanges = false;
    try {
      Response response = chain.proceed(originalRequest);
      if (transmitter.isCanceled()) {
        closeQuietly(response);
        throw new IOException("Canceled");
      }
      return response;
    } catch (IOException e) {
      calledNoMoreExchanges = true;
      throw transmitter.noMoreExchanges(e);
    } finally {
      if (!calledNoMoreExchanges) {
        transmitter.noMoreExchanges(null);
      }
    }
}
```

- 上方法创建一系列拦截器，并放入list中，再创建拦截器链RealInterceptorChain，并执行chain.proceed方法
- proceed方法实现如下：

```
@Override public Response proceed(Request request) throws IOException {
    return proceed(request, transmitter, exchange);
}

public Response proceed(Request request, Transmitter transmitter, @Nullable
Exchange exchange)
      throws IOException {
    ...

    // Call the next interceptor in the chain.
    RealInterceptorChain next = new RealInterceptorChain(interceptors,
transmitter, exchange,
        index + 1, request, call, connectTimeout, readTimeout, writeTimeout);
    Interceptor interceptor = interceptors.get(index);
```

```
    Response response = interceptor.intercept(next);
   ...
    return response;
}
```

- 其核心代码就是上面几行，创建下一个拦截器链，调用interceptors.get(index)取得当前拦截器，并执行interceptor.intercept方法得到response返回；
- getResponseWithInterceptorChain中传入的index为0，则当前拦截器就是RetryAndFollowUpInterceptor，那么我们来看看他的intercept方法是如何实现的

RetryAndFollowUpInterceptor

- RetryAndFollowUpInterceptor的intercept方法代码如下

```
 @Override public Response intercept(Chain chain) throws IOException {
    Request request = chain.request();
    //还记得interceptor.intercept(next)么，所以这里的realChain是下一个拦截器链
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    Transmitter transmitter = realChain.transmitter();

    int followUpCount = 0;
    Response priorResponse = null;
    while (true) {
      transmitter.prepareToConnect(request);

      if (transmitter.isCanceled()) {
        throw new IOException("Canceled");
      }

      Response response;
      boolean success = false;
      try {
        //调用下一个拦截器链的proceed方法
        response = realChain.proceed(request, transmitter, null);
        success = true;
      } catch (RouteException e) {
        // The attempt to connect via a route failed. The request will not have
been sent.
        if (!recover(e.getLastConnectException(), transmitter, false, request))
{
          throw e.getFirstConnectException();
        }
        continue;
        //当发生IOException或者RouteException时会执行recover方法
      } catch (IOException e) {
        // An attempt to communicate with a server failed. The request may have
been sent.
        boolean requestSendStarted = !(e instanceof
ConnectionShutdownException);
        if (!recover(e, transmitter, requestSendStarted, request)) throw e;
        continue;
      } finally {
        // The network call threw an exception. Release any resources.
        if (!success) {
          transmitter.exchangeDoneDueToException();
```

```
        }
      }

      // Attach the prior response if it exists. Such responses never have a
body.
      if (priorResponse != null) {
        response = response.newBuilder()
            .priorResponse(priorResponse.newBuilder()
                    .body(null)
                    .build())
            .build();
      }

      Exchange exchange = Internal.instance.exchange(response);
      Route route = exchange != null ? exchange.connection().route() : null;
      Request followUp = followUpRequest(response, route);

      if (followUp == null) {
        if (exchange != null && exchange.isDuplex()) {
          transmitter.timeoutEarlyExit();
        }
        return response;
      }

      RequestBody followUpBody = followUp.body();
      if (followUpBody != null && followUpBody.isOneShot()) {
        return response;
      }

      closeQuietly(response.body());
      if (transmitter.hasExchange()) {
        exchange.detachWithViolence();
      }

      //重试次数判断
      if (++followUpCount > MAX_FOLLOW_UPS) {
        throw new ProtocolException("Too many follow-up requests: " +
followUpCount);
      }

      request = followUp;
      priorResponse = response;
    }
}
```

- recover方法代码如下

```
private boolean recover(IOException e, Transmitter transmitter,
      boolean requestSendStarted, Request userRequest) {
    // The application layer has forbidden retries.
    if (!client.retryOnConnectionFailure()) return false;

    // We can't send the request body again.
    if (requestSendStarted && requestIsOneShot(e, userRequest)) return false;
```

```
    // This exception is fatal.
    if (!isRecoverable(e, requestSendStarted)) return false;

    // No more routes to attempt.
    if (!transmitter.canRetry()) return false;

    // For failure recovery, use the same route selector with a new connection.
    return true;
}
```

- RetryAndFollowUpInterceptor的intercept方法中调用下一个拦截器链的proceed方法获取response， 并在while (true) 循环中根据异常结果或响应结果判断是否要进行重新请求，如当发生IOException或者RouteException时会执行recover方法， 并且通过++followUpCount > MAX_FOLLOW_UPS判断最大重试次数，超出则直接跳出循环；
- 由RealInterceptorChain.proceed可知会继续调用下一个拦截器的intercept方法，由getResponseWithInterceptorChain中顺序可知下一个拦截器就是BridgeInterceptor
- 那么来继续看一下BridgeInterceptor的intercept方法

BridgeInterceptor

- BridgeInterceptor的intercept方法如下

```
@Override public Response intercept(Chain chain) throws IOException {
    Request userRequest = chain.request();
    Request.Builder requestBuilder = userRequest.newBuilder();

    //补充RequestBody的请求头
    RequestBody body = userRequest.body();
    if (body != null) {
      MediaType contentType = body.contentType();
      if (contentType != null) {
        requestBuilder.header("Content-Type", contentType.toString());
      }

      long contentLength = body.contentLength();
      if (contentLength != -1) {
        requestBuilder.header("Content-Length", Long.toString(contentLength));
        requestBuilder.removeHeader("Transfer-Encoding");
      } else {
        requestBuilder.header("Transfer-Encoding", "chunked");
        requestBuilder.removeHeader("Content-Length");
      }
    }

    if (userRequest.header("Host") == null) {
      requestBuilder.header("Host", hostHeader(userRequest.url(), false));
    }

    if (userRequest.header("Connection") == null) {
      requestBuilder.header("Connection", "Keep-Alive");
    }

    // If we add an "Accept-Encoding: gzip" header field we're responsible for
also decompressing
    // the transfer stream.
```

```
    boolean transparentGzip = false;
    if (userRequest.header("Accept-Encoding") == null &&
userRequest.header("Range") == null) {
      transparentGzip = true;
      requestBuilder.header("Accept-Encoding", "gzip");
    }

    List<Cookie> cookies = cookieJar.loadForRequest(userRequest.url());
    if (!cookies.isEmpty()) {
      requestBuilder.header("Cookie", cookieHeader(cookies));
    }

    if (userRequest.header("User-Agent") == null) {
      requestBuilder.header("User-Agent", Version.userAgent());
    }

    //调用下一个拦截器链的proceed方法
    Response networkResponse = chain.proceed(requestBuilder.build());


    //补充响应头

    HttpHeaders.receiveHeaders(cookieJar, userRequest.url(),
networkResponse.headers());

    Response.Builder responseBuilder = networkResponse.newBuilder()
        .request(userRequest);

    if (transparentGzip
        && "gzip".equalsIgnoreCase(networkResponse.header("Content-Encoding"))
        && HttpHeaders.hasBody(networkResponse)) {
        //Response.body的输入流转为GzipSource，以解压的方式读取流数据
      GzipSource responseBody = new GzipSource(networkResponse.body().source());
      Headers strippedHeaders = networkResponse.headers().newBuilder()
          .removeAll("Content-Encoding")
          .removeAll("Content-Length")
          .build();
      responseBuilder.headers(strippedHeaders);
      String contentType = networkResponse.header("Content-Type");
      responseBuilder.body(new RealResponseBody(contentType, -1L,
Okio.buffer(responseBody)));
    }

    return responseBuilder.build();
}
```

- BridgeInterceptor的intercept中，先各种判断对RequestBody的请求头进行补充，将其转化为能够进行网络访问的请求，然后调用下一个拦截器链的proceed方法获取response,再对respone的响应头进行补充，如设置cookieJar，gzip解压，将请求回来的响应response转化为用户可用的response;
- 调用下一个拦截器链的proceed，又会调用下一个拦截器的intercept方法，下一个拦截器为CacheInterceptor

CacheInterceptor

- CacheInterceptor的intercept方法如下

```
@Override public Response intercept(Chain chain) throws IOException {

    //尝试获取缓存的Response
    Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;

    long now = System.currentTimeMillis();

    //CacheStrategy缓存策略，维护了networkRequest 和 cacheResponse
    //根据时间获取缓存策略，其内部会结合时间等条件返回对应的缓存测试略
    CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(),
cacheCandidate).get();
    Request networkRequest = strategy.networkRequest;
    Response cacheResponse = strategy.cacheResponse;

    if (cache != null) {
      cache.trackResponse(strategy);
    }

    if (cacheCandidate != null && cacheResponse == null) {
      closeQuietly(cacheCandidate.body()); // The cache candidate wasn't
applicable. Close it.
    }

    //如果禁止网络访问又没有缓存，则直接new一个失败的Response
    // If we're forbidden from using the network and the cache is insufficient,
fail.
    if (networkRequest == null && cacheResponse == null) {
      return new Response.Builder()
          .request(chain.request())
          .protocol(Protocol.HTTP_1_1)
          .code(504)
          .message("Unsatisfiable Request (only-if-cached)")
          .body(Util.EMPTY_RESPONSE)
          .sentRequestAtMillis(-1L)
          .receivedResponseAtMillis(System.currentTimeMillis())
          .build();
    }

    //如果不需要网络访问，则直接返回缓存的response
    // If we don't need the network, we're done.
    if (networkRequest == null) {
      return cacheResponse.newBuilder()
          .cacheResponse(stripBody(cacheResponse))
          .build();
    }

    //需要网络访问，则调用下一个拦截器链的proceed获取response
    Response networkResponse = null;
    try {
```

```
      networkResponse = chain.proceed(networkRequest);
    } finally {
      // If we're crashing on I/O or otherwise, don't leak the cache body.
      if (networkResponse == null && cacheCandidate != null) {
        closeQuietly(cacheCandidate.body());
      }
    }

    //如果我们本地有缓存的Response
    // If we have a cache response too, then we're doing a conditional get.
    if (cacheResponse != null) {
      //服务器返回304，则直接返回本地缓存的response
      if (networkResponse.code() == HTTP_NOT_MODIFIED) {
        Response response = cacheResponse.newBuilder()
            .headers(combine(cacheResponse.headers(),
networkResponse.headers()))
            .sentRequestAtMillis(networkResponse.sentRequestAtMillis())

.receivedResponseAtMillis(networkResponse.receivedResponseAtMillis())
            .cacheResponse(stripBody(cacheResponse))
            .networkResponse(stripBody(networkResponse))
            .build();
        networkResponse.body().close();

        // Update the cache after combining headers but before stripping the
        // Content-Encoding header (as performed by initContentStream()).
        cache.trackConditionalCacheHit();
        cache.update(cacheResponse, response);
        return response;
      } else {
        closeQuietly(cacheResponse.body());
      }
    }


    Response response = networkResponse.newBuilder()
        .cacheResponse(stripBody(cacheResponse))
        .networkResponse(stripBody(networkResponse))
        .build();

    //如果有缓存，则对缓存进行更新
    if (cache != null) {
      if (HttpHeaders.hasBody(response) && CacheStrategy.isCacheable(response,
networkRequest)) {
        // Offer this request to the cache.
        CacheRequest cacheRequest = cache.put(response);
        return cacheWritingResponse(cacheRequest, response);
      }

      //如果不是get请求则移除缓存
      if (HttpMethod.invalidatesCache(networkRequest.method())) {
        try {
          cache.remove(networkRequest);
        } catch (IOException ignored) {
          // The cache cannot be written.
```

```
        }
      }
    }

    return response;
}
```

- CacheInterceptor的intercept中对用不用缓存和对缓存是否更新进行了各种判断，如果用网络请求也会调用下一个拦截器链的proceed方法获取response,
- 那么下一个拦截器就是ConnectInterceptor

ConnectInterceptor

- ConnectInterceptor的intercept方法如下, 正式开启okhttp的网络请求

```
@Override public Response intercept(Chain chain) throws IOException {
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    Request request = realChain.request();
    Transmitter transmitter = realChain.transmitter();

    // We need the network to satisfy this request. Possibly for validating a
conditional GET.
    boolean doExtensiveHealthChecks = !request.method().equals("GET");
    Exchange exchange = transmitter.newExchange(chain, doExtensiveHealthChecks);

    return realChain.proceed(request, transmitter, exchange);
  }
```

- 上面调用transmitter.newExchange获取Exchange，并调用下一个拦截器链的proceed传给下一个拦截器,获取response,newExchange方法如下

```
Exchange newExchange(Interceptor.Chain chain, boolean doExtensiveHealthChecks) {
    ...
    ExchangeCodec codec = exchangeFinder.find(client, chain,
doExtensiveHealthChecks);
    Exchange result = new Exchange(this, call, eventListener, exchangeFinder,
codec);
    ...
}
```

- 上面调用了exchangeFinder.find获取ExchangeCodec, 其中通过findHealthyConnection得到RealConnection，再return RealConnection.newCode

```
 public ExchangeCodec find(OkHttpClient client, Interceptor.Chain chain, boolean
doExtensiveHealthChecks) {
    ...
    RealConnection resultConnection = findHealthyConnection(connectTimeout,
readTimeout,
      writeTimeout, pingIntervalMillis, connectionRetryEnabled,
doExtensiveHealthChecks);
    return resultConnection.newCodec(client, chain);
    ...
}
```

- findHealthyConnection又调用了findConnection方法, findConnection方法代码如下, 其中通过连接池或 new RealConnection获取RealConnection，并调用了其connect方法
- 源码很长，下面只是列出了关键步骤

```
private RealConnection findConnection(int connectTimeout, int readTimeout, int
writeTimeout,
      int pingIntervalMillis, boolean connectionRetryEnabled) throws IOException
{
    boolean foundPooledConnection = false;
    RealConnection result = null;
    ...
    //先从连接池中取
    if (result == null) {
        // Attempt to get a connection from the pool.
        if (connectionPool.transmitterAcquirePooledConnection(address,
transmitter, null, false)) {
          foundPooledConnection = true;
          result = transmitter.connection;
       }
       ...
    }
    ...
    //连接池中有就不用继续搞了
    if (result != null) {
      // If we found an already-allocated or pooled connection, we're done.
      return result;
    }

   ...
   //连接池没有就new一个，并调用connect方法
    result = new RealConnection(connectionPool, selectedRoute);

    // Do TCP + TLS handshakes. This is a blocking operation.
    result.connect(connectTimeout, readTimeout, writeTimeout,
pingIntervalMillis,
        connectionRetryEnabled, call, eventListener);
    //添加到连接池中
    connectionPool.routeDatabase.connected(result.route());
    ...
    return result;
}
```

CallServerInterceptor

- 最后来看看CallServerInterceptor的intercept

```
@Override public Response intercept(Chain chain) throws IOException {
    RealInterceptorChain realChain = (RealInterceptorChain) chain;
    Exchange exchange = realChain.exchange();
    Request request = realChain.request();

    long sentRequestMillis = System.currentTimeMillis();
    //向socket中写入请求头信息
    exchange.writeRequestHeaders(request);
```

```
    boolean responseHeadersStarted = false;
    Response.Builder responseBuilder = null;
    if (HttpMethod.permitsRequestBody(request.method()) && request.body() !=
null) {
      // If there's a "Expect: 100-continue" header on the request, wait for a
"HTTP/1.1 100
      // Continue" response before transmitting the request body. If we don't get
that, return
      // what we did get (such as a 4xx response) without ever transmitting the
request body.
      if ("100-continue".equalsIgnoreCase(request.header("Expect"))) {
        exchange.flushRequest();
        responseHeadersStarted = true;
        exchange.responseHeadersStart();
        responseBuilder = exchange.readResponseHeaders(true);
      }

      if (responseBuilder == null) {
        if (request.body().isDuplex()) {
          // Prepare a duplex body so that the application can send a request
body later.
          exchange.flushRequest();
          BufferedSink bufferedRequestBody = Okio.buffer(
              exchange.createRequestBody(request, true));
          //写入body信息
          request.body().writeTo(bufferedRequestBody);
        } else {
          // Write the request body if the "Expect: 100-continue" expectation was
met.
          BufferedSink bufferedRequestBody = Okio.buffer(
              exchange.createRequestBody(request, false));
          request.body().writeTo(bufferedRequestBody);
          bufferedRequestBody.close();
        }
      } else {
        exchange.noRequestBody();
        if (!exchange.connection().isMultiplexed()) {
          // If the "Expect: 100-continue" expectation wasn't met, prevent the
HTTP/1 connection
          // from being reused. Otherwise we're still obligated to transmit the
request body to
          // leave the connection in a consistent state.
          exchange.noNewExchangesOnConnection();
        }
      }
    } else {
      exchange.noRequestBody();
    }

    //请求结束
    if (request.body() == null || !request.body().isDuplex()) {
      exchange.finishRequest();
    }

    if (!responseHeadersStarted) {
```

```
      exchange.responseHeadersStart();
    }

    //读取响应头
    if (responseBuilder == null) {
      responseBuilder = exchange.readResponseHeaders(false);
    }

    Response response = responseBuilder
        .request(request)
        .handshake(exchange.connection().handshake())
        .sentRequestAtMillis(sentRequestMillis)
        .receivedResponseAtMillis(System.currentTimeMillis())
        .build();

    int code = response.code();
    if (code == 100) {
      // server sent a 100-continue even though we did not request one.
      // try again to read the actual response
      response = exchange.readResponseHeaders(false)
          .request(request)
          .handshake(exchange.connection().handshake())
          .sentRequestAtMillis(sentRequestMillis)
          .receivedResponseAtMillis(System.currentTimeMillis())
          .build();

      code = response.code();
    }

    exchange.responseHeadersEnd(response);

    //读取响应body
    if (forWebSocket && code == 101) {
      // Connection is upgrading, but we need to ensure interceptors see a non-
null response body.
      response = response.newBuilder()
          .body(Util.EMPTY_RESPONSE)
          .build();
    } else {
      response = response.newBuilder()
          .body(exchange.openResponseBody(response))
          .build();
    }

    if ("close".equalsIgnoreCase(response.request().header("Connection"))
        || "close".equalsIgnoreCase(response.header("Connection"))) {
      exchange.noNewExchangesOnConnection();
    }

    if ((code == 204 || code == 205) && response.body().contentLength() > 0) {
      throw new ProtocolException(
          "HTTP " + code + " had non-zero Content-Length: " +
response.body().contentLength());
    }
```

```
    return response;
}
```

2. Retrofit使用及源码解析

- Retrofit是目前Android最优秀的网络封装框架，是对OkHttp网络请求库的封装
- App应用程序通过Retrofit请求网络，实际上是使用Retrofit接口层封装请求参数，之后由OkHttp完成后续的请求操作；服务器数据返回后，OkHttp将原始的结果交给Retrofit，根据用户需求对结果进行解析；

使用

简单使用

添加依赖

- retrofit2内置了OkHttp,所以无需再单独添加OkHttp依赖了

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

创建Retrofit实例

```
val baseUrl = "https://api.github.com/"
val okHttpClient = OkHttpClient.Builder()
    .connectTimeout(30, TimeUnit.SECONDS)
    .readTimeout(60, TimeUnit.SECONDS)
    .writeTimeout(90, TimeUnit.SECONDS)
    .build()
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    .client(okHttpClient)
    .build()
```

创建返回数据的类

```
class RepoList {
    @SerializedName("items") val items:List<Repo> = emptyList()
}

data class Repo(
    @SerializedName("id") val id: Int,
    @SerializedName("name") val name: String,
    @SerializedName("description") val description: String,
    @SerializedName("stargazers_count") val starCount: String,
)
```

创建网络请求接口

```
interface ApiService {
    @GET("search/repositories?sort=stars&q=Android")
    fun searRepos(@Query("page") page: Int, @Query("per_page") perPage: Int):
Call<RepoList>
}
```

创建网络请求接口实例

```
val apiService = retrofit.create(ApiService::class.java)
```

调用接口实例方法获取Call

```
val call = apiService.searRepos(1, 5)
```

发送网络请求

1.同步请求

```
val response: Response<RepoList> = call.execute()
if (response.isSuccessful) {
    val repo = response.body()
    LjyLogUtil.d(repo.toString())
} else {
    LjyLogUtil.d("code=${response.code()}, msg=${response.message()}")
    LjyLogUtil.d(IOException("Unexpected code $response").message)
}
```

2.异步请求

```
call.enqueue(object : Callback<RepoList> {
    override fun onResponse(call: Call<RepoList>, result: Response<RepoList>) {
        if (result.body() != null) {
            val repoList: RepoList = result.body()!!
            for (it in repoList.items) {
                LjyLogUtil.d("${it.name}_${it.starCount}")
                LjyLogUtil.d(it.description)
            }
        }
    }

    override fun onFailure(call: Call<RepoList>, t: Throwable) {
        LjyLogUtil.d("onFailure:${t.message}")
    }
})
```

注解类型

1. 网络请求方法

@GET, @POST, @PUT, @DELETE, @HEAD, @PATCH，@OPTIONS

- 分别对应 HTTP中的网络请求方式
- 注解的value属性用来设置相对/完整url, 如果是完整url则可以覆盖创建Retrofit实例时的baseUrl
- Retrofit把网络请求的URL分成了两部分设置，一是创建Retrofit实例时设置的baseUrl，另一半是网络请求方法注解的value设置或@Url中设置的部分，

```
@GET("api/items")
fun getRepos(): Call<RepoList>

@GET("https://api.github.com/api/items")
fun getRepos(): Call<RepoList>
```

@HTTP

- 替换以上注解的作用 及 更多的功能拓展

```
/**
 * method：网络请求的方法（区分大小写）
 * path：网络请求地址路径
 * hasBody：是否有请求体
 */
@HTTP(method = "GET", hasBody = false)
fun getRepos(@Url url: String):  Call<RepoList>

@HTTP(method = "GET", path = "api/items/{userId}", hasBody = false)
fun getRepos2(@Path("userId") userId: String) :  Call<RepoList>
```

2. 标记

@FormUrlEncoded

- 表示请求体(RequestBody)是Form表单

```
@FormUrlEncoded
@POST("api/search")
fun searchRepo( @Field("name") repoName:String): Call<RepoList>
```

@Multipart

- 表示请求体是一个支持文件上传的Form表单
- 具体使用见下面的@Part部分

@Streaming

- 表示返回的数据以流的形式返回，适用于返回数据较大的场景（若没有使用该注解，默认是吧数据全部载入内存中）

```
@Streaming
@GET
fun downloadFile(@Url url: String?): Call<ResponseBody>
```

3. 网络请求参数

@Path

- URL地址的缺省值

```
@GET("api/items/{userId}/repos")
fun getItem(@Path("userId") userId: String): Call<Repo>
//在发起请求时， {userId} 会被替换为方法的参数 userId（被@Path注解的参数）
```

@Url

- 直接传入一个请求的url，和@GET，@POST等注解的value属性设置url类似,但是通过参数传入显然更灵活一点

```
@FormUrlEncoded
@POST("api/search")
fun searchRepo(@Url url: String, @Field("name") repoName: String):
Call<RepoList>
```

@Header & @Headers

- 使用场景：@Header用于添加不固定的请求头，@Headers用于添加固定的请求头
- 使用方式：@Header作用于方法的参数；@Headers作用于方法

```
@Streaming
@GET
fun downloadFile(@Header("RANGE") start:String , @Url url: String?):
Call<ResponseBody>

@Headers("Content-Type: application/json;charset=UTF-8")
@POST("api/search")
fun searchRepo2(@Body params: Map<String, Any>): Call<RepoList>
```

- 添加header还可以通过上一篇介绍过的okHttp拦截器实现

```
val okHttpClient = OkHttpClient.Builder()
    .addInterceptor(object : Interceptor{
        override fun intercept(chain: Interceptor.Chain): okhttp3.Response {
            val request=chain.request().newBuilder()
                .addHeader("Content-Type", "application/x-www-form-urlencoded;
charset=UTF-8")
                .addHeader("Accept-Encoding", "gzip, deflate")
                .addHeader("Connection", "keep-alive")
                .addHeader("Accept", "*/*")
                .addHeader("Cookie", "add cookies here")
                .build()
            return chain.proceed(request)
        }
    })
    .build()
```

@Query & @QueryMap

- 用于 @GET 方法的查询参数

```
@GET("search/repositories?sort=stars&q=Android")
fun searRepos(@Query("page") page: Int, @Query("per_page") perPage: Int):
Call<RepoList>

@GET("search/repositories?sort=stars&q=Android")
fun searRepos(@QueryMap params: Map<String, Any>): Call<RepoList>
```

@Field & @FieldMap

- 发送 Post请求 时提交请求的表单字段，与 @FormUrlEncoded 注解配合使用

```
@FormUrlEncoded
@POST("api/search")
fun searchRepo(@Url url: String, @Field("name") repoName: String):
Call<RepoList>

@FormUrlEncoded
@POST("api/search")
fun searchRepo(@Url url: String, @FieldMap params: Map<String, Any>):
Call<RepoList>
```

@Part & @PartMap

- 发送 Post请求 时提交请求的表单字段, 与 @Multipart 注解配合使用
- 与@Field的区别：功能相同，但携带的参数类型更加丰富，包括数据流，所以适用于 有文件上传的场景

```
@POST("upload/imgFile")
@Multipart
fun uploadImgFile(
    @Part("userId") userId: RequestBody?,
    @PartMap partMap: Map<String, RequestBody?>,
    @Part("file") file: MultipartBody.Part
): Call<ResponseBody>

@Multipart
@POST("upload/files")
fun uploadFiles(
    @Part("userId") userId: RequestBody?,
    @Part files: List<MultipartBody.Part>
): Call<ResponseBody>

//使用:
val userId: RequestBody = RequestBody.create(MediaType.parse("multipart/form-
data"), "1111")

val paramsMap: MutableMap<String, RequestBody> = HashMap()
paramsMap["userId"] = RequestBody.create(MediaType.parse("text/plain"),
"123456")
paramsMap["userName"] = RequestBody.create(MediaType.parse("text/plain"),
"jinYang")
paramsMap["taskName"] = RequestBody.create(MediaType.parse("text/plain"), "新建派
单")

val imgFile=File(externalCacheDir, "ljy.jpg")
val requestFile: RequestBody =
    RequestBody.create(MediaType.parse("multipart/form-data"),imgFile )

val partFile = MultipartBody.Part.createFormData("imageUrl", imgFile.name,
requestFile)

apiService.uploadImgFile(userId,paramsMap,partFile)
```

```
val imgFile1=File(externalCacheDir, "ljy1.jpg")
val imgFile2=File(externalCacheDir, "ljy2.jpg")
val imgFile3=File(externalCacheDir, "ljy3.jpg")
val imageFiles= arrayOf(imgFile1,imgFile2,imgFile3)
val parts = ArrayList<MultipartBody.Part>(imageFiles.size)
for (i in imageFiles.indices) {
    val file: File = imageFiles[i]
    parts[i] = MultipartBody.Part.createFormData(
        "file_$i", file.name, RequestBody.create(
            MediaType.parse("image/*"), file
        )
    )
}

apiService.uploadFiles(userId,parts)
```

@Body

- 以 Post方式 传递 自定义数据类型 给服务器,如果提交的是一个Map，那么作用相当于 @Field

```
@Headers("Content-Type: application/json;charset=UTF-8")
@POST("api/add")
fun addRepo(@Body repo: Repo): Call<Boolean>

@Headers("Content-Type: application/json;charset=UTF-8")
@POST("api/add")
fun addRepo2(@Body params: Map<String, Any>): Call<Boolean>

@Headers("Content-Type: application/json;charset=UTF-8")
@POST("api/add")
fun addRepo3(@Body body: RequestBody): Call<Boolean>

@FormUrlEncoded
@POST("api/add")
fun addRepo4(@Body body: FormBody): Call<Boolean>

//使用:
val repo = Repo(1, "name", "info", "20")
apiService.addRepo(repo)

val map: MutableMap<String, Any> = HashMap()
map["key"] = "value"
apiService.addRepo2(map)

val body: RequestBody = RequestBody
    .create(MediaType.parse("application/json; charset=utf-8"), repo.toString())
apiService.addRepo3(body)

val formBody = FormBody.Builder()
    .add("key", "value")
    .build()
apiService.addRepo4(formBody)
```

数据解析器 & 请求适配器

- Retrofit支持多种数据解析方式和网络请求适配器，使用时需要在Gradle添加依赖

数据解析器

- 默认情况下Retrofit只支持将HTTP的响应体转换换为Call，有了Converter就可以把ResponseBody替换成其他类型， 如我们常用的GsonConverterFactory，下面列出官方给我们提供的Converter;

1. 添加依赖

```
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'//Gson的支持 [常用] [可选]
implementation 'com.squareup.retrofit2:converter-simplexml:2.9.0'//simplexml的支持 [可选]
implementation 'com.squareup.retrofit2:converter-jackson:2.9.0'//jackson的支持 [可选]
implementation 'com.squareup.retrofit2:converter-protobuf:2.9.0'//protobuf的支持 [可选]
implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'//moshi的支持 [可选]
implementation 'com.squareup.retrofit2:converter-wire:2.9.0'//wire的支持 [可选]
implementation 'com.squareup.retrofit2:converter-scalars:2.9.0'//String的支持 [可选]
```

2. 创建Retrofit实例时添加

```
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    .addConverterFactory(GsonConverterFactory.create())
    .addConverterFactory(JacksonConverterFactory.create())
    .addConverterFactory(SimpleXmlConverterFactory.create())
    .addConverterFactory(ProtoConverterFactory.create())
    .addConverterFactory(ScalarsConverterFactory.create())
    .build()
```

3. 自定义Converter

- 自定义之前我们先来看看官方是如何实现的, 以平时最常用的GsonConverterFactory为例

```
//继承Converter.Factory
public final class GsonConverterFactory extends Converter.Factory {
  //静态的create方法
  public static GsonConverterFactory create() {
    return create(new Gson());
  }


  public static GsonConverterFactory create(Gson gson) {
    if (gson == null) throw new NullPointerException("gson == null");
    return new GsonConverterFactory(gson);
  }

  private final Gson gson;

  private GsonConverterFactory(Gson gson) {
```

```
    this.gson = gson;
  }

  //重写responseBodyConverter方法，将响应体交给GsonResponseBodyConverter处理
  @Override
  public Converter<ResponseBody, ?> responseBodyConverter(
      Type type, Annotation[] annotations, Retrofit retrofit) {
    TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
    return new GsonResponseBodyConverter<>(gson, adapter);
  }

  //重写requestBodyConverter方法，将请求体交给GsonRequestBodyConverter处理
  @Override
  public Converter<?, RequestBody> requestBodyConverter(
      Type type,
      Annotation[] parameterAnnotations,
      Annotation[] methodAnnotations,
      Retrofit retrofit) {
    TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
    return new GsonRequestBodyConverter<>(gson, adapter);
  }
}

//处理响应体的Converter，实现Converter<ResponseBody, T>接口
final class GsonResponseBodyConverter<T> implements Converter<ResponseBody, T> {
  private final Gson gson;
  private final TypeAdapter<T> adapter;

  GsonResponseBodyConverter(Gson gson, TypeAdapter<T> adapter) {
    this.gson = gson;
    this.adapter = adapter;
  }

   //重写convert方法，将ResponseBody通过Gson转为自定义的数据模型类
  @Override
  public T convert(ResponseBody value) throws IOException {
    JsonReader jsonReader = gson.newJsonReader(value.charStream());
    try {
      T result = adapter.read(jsonReader);
      if (jsonReader.peek() != JsonToken.END_DOCUMENT) {
        throw new JsonIOException("JSON document was not fully consumed.");
      }
      return result;
    } finally {
      value.close();
    }
  }
}

//处理请求体的Converter，实现Converter<T, RequestBody>接口
final class GsonRequestBodyConverter<T> implements Converter<T, RequestBody> {
  private static final MediaType MEDIA_TYPE = MediaType.get("application/json;
charset=UTF-8");
  private static final Charset UTF_8 = Charset.forName("UTF-8");
```

```
  private final Gson gson;
  private final TypeAdapter<T> adapter;

  GsonRequestBodyConverter(Gson gson, TypeAdapter<T> adapter) {
    this.gson = gson;
    this.adapter = adapter;
  }

   //重写convert方法，通过Gson将自定义的数据模型类转换为RequestBody
  @Override
  public RequestBody convert(T value) throws IOException {
    Buffer buffer = new Buffer();
    Writer writer = new OutputStreamWriter(buffer.outputStream(), UTF_8);
    JsonWriter jsonWriter = gson.newJsonWriter(writer);
    adapter.write(jsonWriter, value);
    jsonWriter.close();
    return RequestBody.create(MEDIA_TYPE, buffer.readByteString());
  }
}
```

- 那么我们来自己试试吧：
- 例1：返回格式为Call

```
//1. 自定义StringConverter，实现Converter
class StringConverter : Converter<ResponseBody, String> {
    companion object {
        val INSTANCE = StringConverter()
    }

    @Throws(IOException::class)
    override fun convert(value: ResponseBody): String {
        return value.string()
    }
}

//2. 自定义StringConverterFactory，用来向Retrofit注册StringConverter
class StringConverterFactory : Converter.Factory() {
    companion object {
        private val INSTANCE = StringConverterFactory()
        fun create(): StringConverterFactory {
            return INSTANCE
        }
    }

    // 只实现从ResponseBody 到 String 的转换，所以其它方法可不覆盖
    override fun responseBodyConverter(
        type: Type,
        annotations: Array<Annotation?>?,
        retrofit: Retrofit?
    ): Converter<ResponseBody, *>? {
        return if (type === String::class.java) {
            StringConverter.INSTANCE
        } else null
        //其它类型不处理，返回null
    }
```

```
}

3. 使用
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    // 自定义的Converter一定要放在官方提供的Converter前面
    //addConverterFactory是有先后顺序的，多个Converter都支持同一种类型，只有第一个才被使用
    .addConverterFactory(StringConverterFactory.create())
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

- 例2：ResponseBody转换为Map

```
class MapConverterFactory : Converter.Factory() {
    companion object {
        fun create(): MapConverterFactory {
            return MapConverterFactory()
        }
    }

    override fun responseBodyConverter(
        type: Type,
        annotations: Array<Annotation>,
        retrofit: Retrofit
    ): Converter<ResponseBody, *> {
        return MapConverter()
    }

    class MapConverter : Converter<ResponseBody, Map<String, String>> {
        @Throws(IOException::class)
        override fun convert(body: ResponseBody): Map<String, String> {
            val map: MutableMap<String, String> = HashMap()
            val content = body.string()
            val keyValues = content.split("&").toTypedArray()
            for (i in keyValues.indices) {
                val keyValue = keyValues[i]
                val splitIndex = keyValue.indexOf("=")
                val key = keyValue.substring(0, splitIndex)
                val value = keyValue.substring(splitIndex + 1, keyValue.length)
                map[key] = value
            }
            return map
        }
    }
}
```

请求适配器

- Converter是对于Call中T的转换，而CallAdapter则可以对Call转换，下面列出官方给我们提供的CallAdapter;

1. 添加依赖

```
implementation 'com.squareup.retrofit2:adapter-rxjava2:2.9.0'//RxJava支持 [常用]
[可选]
implementation 'com.squareup.retrofit2:adapter-java8:2.9.0'//java8支持 [可选]
implementation 'com.squareup.retrofit2:adapter-guava:2.9.0'//guava支持 [可选]
```

2. 创建Retrofit实例时添加

```
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .addCallAdapterFactory(Java8CallAdapterFactory.create())
    .addCallAdapterFactory(GuavaCallAdapterFactory.create())
    .build()
```

3. 自定义CallAdapter

- 同样的我们来看看官方的RxJava2CallAdapterFactory是如何实现的

```
//适配器工厂类，继承CallAdapter.Factory
public final class RxJava2CallAdapterFactory extends CallAdapter.Factory {
  //静态的create方法
  public static RxJava2CallAdapterFactory create() {
    return new RxJava2CallAdapterFactory(null, false);
  }

  ...
  //Rxjava的调度器scheduler
  private final @Nullable Scheduler scheduler;
  private final boolean isAsync;
  //构造方法
  private RxJava2CallAdapterFactory(@Nullable Scheduler scheduler, boolean
isAsync) {
    this.scheduler = scheduler;
    this.isAsync = isAsync;
  }

  //重写get方法，返回RxJava2CallAdapter
  @Override
  public @Nullable CallAdapter<?, ?> get(
      Type returnType, Annotation[] annotations, Retrofit retrofit) {
    Class<?> rawType = getRawType(returnType);
    ...
    return new RxJava2CallAdapter(
        responseType, scheduler, isAsync, isResult, isBody, isFlowable, isSingle,
isMaybe, false);
  }
}

//适配器类，实现CallAdapter接口
final class RxJava2CallAdapter<R> implements CallAdapter<R, Object> {
  private final Type responseType;
  RxJava2CallAdapter(Type responseType,....){this.responseType =
responseType;....}
```

```
  ...
  //重写responseType方法
  @Override
  public Type responseType() {
    return responseType;
  }

  //重写adapt方法，通过 RxJava2 将Call转换为Observable
  @Override
  public Object adapt(Call<R> call) {
    Observable<Response<R>> responseObservable =
        isAsync ? new CallEnqueueObservable<>(call) : new CallExecuteObservable<>
(call);

    Observable<?> observable;
    if (isResult) {
      observable = new ResultObservable<>(responseObservable);
    } else if (isBody) {
      observable = new BodyObservable<>(responseObservable);
    } else {
      observable = responseObservable;
    }

    if (scheduler != null) {
      observable = observable.subscribeOn(scheduler);
    }

    if (isFlowable) {
      return observable.toFlowable(BackpressureStrategy.LATEST);
    }
    if (isSingle) {
      return observable.singleOrError();
    }
    if (isMaybe) {
      return observable.singleElement();
    }
    if (isCompletable) {
      return observable.ignoreElements();
    }
    return RxJavaPlugins.onAssembly(observable);
  }
}
```

- 然后我们自己试试搞一个

```
//1. 自定义Call
class LjyCall<T>(private val call: Call<T>) {
    @Throws(IOException::class)
    fun get(): T? {
        return call.execute().body()
    }
}
//2. 自定义CallAdapter
class LjyCallAdapter<R>(private val responseType: Type) : CallAdapter<R, Any> {
    override fun responseType(): Type {
```

```
        return responseType
    }

    override fun adapt(call: Call<R>): Any {
        return LjyCall(call)
    }
}
//3. 自定义CallAdapterFactory
class LjyCallAdapterFactory : CallAdapter.Factory() {
    companion object {
        private val INSTANCE = LjyCallAdapterFactory()
        fun create(): LjyCallAdapterFactory {
            return INSTANCE
        }
    }

    override fun  get(
        returnType: Type,
        annotations: Array<Annotation>,
        retrofit: Retrofit
    ): CallAdapter<R, Any>? {
        // 获取原始类型
        val rawType = getRawType(returnType)
        if (rawType == LjyCall::class.java && returnType is ParameterizedType) {
            val callReturnType = getParameterUpperBound(0, returnType)
            return LjyCallAdapter(callReturnType)
        }
        return null
    }
}
//4. 使用
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    //也是放到前面，有先后顺序
    .addCallAdapterFactory(LjyCallAdapterFactory.create())
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build()
```

源码解析

- 源码地址: square/retrofit

Retrofit & Builder

Builder的构造方法

- Retrofit实例通过Builder（建造者）模式创建, 那么我们来看看Builder的构造方法

```
public static final class Builder {
    //平台类型对象
    private final Platform platform;
    //网络请求工厂，默认使用OkHttpClient，生产网络请求器（Call）
    private @Nullable okhttp3.Call.Factory callFactory;
    //url的基地址，注意这里的类型是HttpUrl，而非String
    private @Nullable HttpUrl baseUrl;
```

```
    //数据转换器工厂集合
    private final List<Converter.Factory> converterFactories = new ArrayList<>
();
    //适配器工厂集合
    private final List<CallAdapter.Factory> callAdapterFactories = new
ArrayList<>();
    //回调方法执行器，在 Android 上默认是封装了 handler 的 MainThreadExecutor
    private @Nullable Executor callbackExecutor;
    private boolean validateEagerly;
    public Builder() {
      this(Platform.get());
    }
    Builder(Platform platform) {
      this.platform = platform;
    }
    ...
}
```

- 可以看到构造函数中platform是通过Platform.get()获取的，那么来看看Platform.get()代码的实现,

```
class Platform {
  private static final Platform PLATFORM = findPlatform();

  static Platform get() {
    return PLATFORM;
  }

  private static Platform findPlatform() {
    return "Dalvik".equals(System.getProperty("java.vm.name"))
        ? new Android() //
        : new Platform(true);
  }
  ...
}
```

- 很明显我们需要的是Android的实现，它的defaultCallbackExecutor返回封装了Handler的MainThreadExecutor， 其作用是可以从工作线程切换到UI线程

```
static final class Android extends Platform {
    Android() {
      super(Build.VERSION.SDK_INT >= 24);
    }

    @Override
    public Executor defaultCallbackExecutor() {
      return new MainThreadExecutor();
    }

   ...

    static final class MainThreadExecutor implements Executor {
      private final Handler handler = new Handler(Looper.getMainLooper());

      @Override
```

```
      public void execute(Runnable r) {
        handler.post(r);
      }
    }
}
```

Builder.build()

- 生成Retrofit实例最后需要调用build, 那么我们来看看该方法的实现

```
public Retrofit build() {

      //可以看到baseUrl是必须设置的
      if (baseUrl == null) {
        throw new IllegalStateException("Base URL required.");
      }

      //callFactory 默认使用OkHttpClient
      okhttp3.Call.Factory callFactory = this.callFactory;
      if (callFactory == null) {
        callFactory = new OkHttpClient();
      }

      Executor callbackExecutor = this.callbackExecutor;
      if (callbackExecutor == null) {
        //这里就是对应上面Android的defaultCallbackExecutor，返回封装了Handler的
MainThreadExecutor，
        callbackExecutor = platform.defaultCallbackExecutor();
      }

      //数据解析器的集合：
      // Make a defensive copy of the adapters and add the default Call adapter.
      List<CallAdapter.Factory> callAdapterFactories = new ArrayList<>
(this.callAdapterFactories);

callAdapterFactories.addAll(platform.defaultCallAdapterFactories(callbackExecuto
r));

      //适配器的集合：
      // Make a defensive copy of the converters.
      List<Converter.Factory> converterFactories =
          new ArrayList<>(
              1 + this.converterFactories.size() +
platform.defaultConverterFactoriesSize());

      // Add the built-in converter factory first. This prevents overriding its
behavior but also
      // ensures correct behavior when using converters that consume all types.
      converterFactories.add(new BuiltInConverters());
      converterFactories.addAll(this.converterFactories);
      converterFactories.addAll(platform.defaultConverterFactories());

      return new Retrofit(
          callFactory,
          baseUrl,
```

```
          unmodifiableList(converterFactories),
          unmodifiableList(callAdapterFactories),
          callbackExecutor,
          validateEagerly);
}
```

- 上面可以看到baseUrl是必须设置的，那么再来看看其有何要求呢

```
public Builder baseUrl(String baseUrl) {
    Objects.requireNonNull(baseUrl, "baseUrl == null");
    return baseUrl(HttpUrl.get(baseUrl));
}

public Builder baseUrl(HttpUrl baseUrl) {
    Objects.requireNonNull(baseUrl, "baseUrl == null");
    List<String> pathSegments = baseUrl.pathSegments();
    //如果host后面由路径，则必须以’/‘结尾
    if (!"".equals(pathSegments.get(pathSegments.size() - 1))) {
      throw new IllegalArgumentException("baseUrl must end in /: " + baseUrl);
    }
    this.baseUrl = baseUrl;
    return this;
}

  /**
   * Returns a list of path segments like {@code ["a", "b", "c"]} for the URL
{@code
   * http://host/a/b/c}. This list is never empty though it may contain a single
empty string.
   *
   * <p><table summary="">
   *   <tr><th>URL</th><th>{@code pathSegments()}</th></tr>
   *   <tr><td>{@code http://host/}</td><td>{@code [""]}</td></tr>
   *   <tr><td>{@code http://host/a/b/c"}</td><td>{@code ["a", "b", "c"]}</td>
</tr>
   *   <tr><td>{@code http://host/a/b%20c/d"}</td><td>{@code ["a", "b c", "d"]}
</td></tr>
   * </table>
   */
  public List<String> pathSegments() {
    return pathSegments;
  }
```

- 上面build方法中说了，callFactory 默认使用OkHttpClient，可能从命名上看并不是一个类型， 但是如果我们看看OkHttpClient源码就会发现它实现了Call.Factory;

```
public class OkHttpClient implements Cloneable, Call.Factory, WebSocket.Factory
{
    ...
}
```

Retrofit的构造方法

- 看过了Builder再来看看我们的主角Retrofit，先来看看其变量和构造方法

```
public final class Retrofit {
   //网络请求配置对象的集合（对网络请求接口中方法注解进行解析后得到的对象）
  private final Map<Method, ServiceMethod<?>> serviceMethodCache = new
ConcurrentHashMap<>();

   //下面几个在Builder中都有过介绍了
  final okhttp3.Call.Factory callFactory;
  final HttpUrl baseUrl;
  final List<Converter.Factory> converterFactories;
  final List<CallAdapter.Factory> callAdapterFactories;
  final @Nullable Executor callbackExecutor;
  final boolean validateEagerly;

  Retrofit(
      okhttp3.Call.Factory callFactory,
      HttpUrl baseUrl,
      List<Converter.Factory> converterFactories,
      List<CallAdapter.Factory> callAdapterFactories,
      @Nullable Executor callbackExecutor,
      boolean validateEagerly) {
    this.callFactory = callFactory;
    this.baseUrl = baseUrl;
    this.converterFactories = converterFactories; // Copy+unmodifiable at call
site.
    this.callAdapterFactories = callAdapterFactories; // Copy+unmodifiable at
call site.
    this.callbackExecutor = callbackExecutor;
    this.validateEagerly = validateEagerly;
  }
  ...
}
```

Retrofit的create方法

- 使用中我们创建了retrofit实例后，会调用其create方法生成接口的动态代理对象,代码如下

```
public <T> T create(final Class<T> service) {
    validateServiceInterface(service);
    return (T)
        //通过动态代理创建接口的实例
        Proxy.newProxyInstance(
            //参数1：classLoader
            service.getClassLoader(),
            //参数2：接口类型数组
            new Class<?>[] {service},
            //参数3：实现了InvocationHandler的代理类
            new InvocationHandler() {
              private final Platform platform = Platform.get();
              private final Object[] emptyArgs = new Object[0];

              @Override
```

```
              public @Nullable Object invoke(Object proxy, Method method,
@Nullable Object[] args)
                  throws Throwable {
                // If the method is a method from Object then defer to normal
invocation.
                if (method.getDeclaringClass() == Object.class) {
                  return method.invoke(this, args);
                }
                args = args != null ? args : emptyArgs;
                return platform.isDefaultMethod(method)
                    ? platform.invokeDefaultMethod(method, service, proxy, args)
                    : loadServiceMethod(method).invoke(args);
              }
            });
}
```

- 可以看到其invoke方法中最后调用了loadServiceMethod方法，其代码如下，就是将method解析为ServiceMethod，并加入到serviceMethodCache中缓存

```
ServiceMethod<?> loadServiceMethod(Method method) {
    ServiceMethod<?> result = serviceMethodCache.get(method);
    if (result != null) return result;

    synchronized (serviceMethodCache) {
      //先在缓存map中获取
      result = serviceMethodCache.get(method);
      if (result == null) {
        //取不到新建一个并加入缓存
        result = ServiceMethod.parseAnnotations(this, method);
        serviceMethodCache.put(method, result);
      }
    }
    return result;
}
```

- 上面代码的解析工作实际是调用 ServiceMethod.parseAnnotations，通过 RequestFactory 完成对注解的解析的

```
abstract class ServiceMethod<T> {
  static <T> ServiceMethod<T> parseAnnotations(Retrofit retrofit, Method method)
{
    RequestFactory requestFactory = RequestFactory.parseAnnotations(retrofit,
method);
    Type returnType = method.getGenericReturnType();
    ...
    return HttpServiceMethod.parseAnnotations(retrofit, method, requestFactory);
  }
+
  abstract @Nullable T invoke(Object[] args);
}
```

- 那么再来看看RequestFactory.parseAnnotations中干了点啥吧

```
final class RequestFactory {
```

```
    static RequestFactory parseAnnotations(Retrofit retrofit, Method method) {
        return new Builder(retrofit, method).build();
    }
    ...
    static final class Builder {
        Builder(Retrofit retrofit, Method method) {
          this.retrofit = retrofit;
          this.method = method;
          //获取网络请求方法的注解：如@GET,@POST@HTTP
          this.methodAnnotations = method.getAnnotations();
          //获取网络请求方法参数的类型
          this.parameterTypes = method.getGenericParameterTypes();
          //获取网络请求参数的注解,如@Url,@Path,@Query等
          this.parameterAnnotationsArray = method.getParameterAnnotations();
        }

        //build方法
        RequestFactory build() {
          //解析网络请求方法的注解
          for (Annotation annotation : methodAnnotations) {
            parseMethodAnnotation(annotation);
          }
          ...
          //解析网络请求方法参数的类型和注解
          int parameterCount = parameterAnnotationsArray.length;
          parameterHandlers = new ParameterHandler<?>[parameterCount];
          for (int p = 0, lastParameter = parameterCount - 1; p < parameterCount;
p++) {
            parameterHandlers[p] =
                parseParameter(p, parameterTypes[p],
parameterAnnotationsArray[p], p == lastParameter);
          }
          ...
          return new RequestFactory(this);
        }

        //解析网络请求方法的注解，看到下面的是不是很眼熟，有点豁然开朗了
        private void parseMethodAnnotation(Annotation annotation) {
          if (annotation instanceof DELETE) {
            parseHttpMethodAndPath("DELETE", ((DELETE) annotation).value(),
false);
          } else if (annotation instanceof GET) {
            parseHttpMethodAndPath("GET", ((GET) annotation).value(), false);
          } else if (annotation instanceof HEAD) {
            parseHttpMethodAndPath("HEAD", ((HEAD) annotation).value(), false);
          } else if (annotation instanceof PATCH) {
            parseHttpMethodAndPath("PATCH", ((PATCH) annotation).value(), true);
          } else if (annotation instanceof POST) {
            parseHttpMethodAndPath("POST", ((POST) annotation).value(), true);
          } else if (annotation instanceof PUT) {
            parseHttpMethodAndPath("PUT", ((PUT) annotation).value(), true);
          } else if (annotation instanceof OPTIONS) {
            parseHttpMethodAndPath("OPTIONS", ((OPTIONS) annotation).value(),
false);
          } else if (annotation instanceof HTTP) {
```

```
            HTTP http = (HTTP) annotation;
            parseHttpMethodAndPath(http.method(), http.path(), http.hasBody());
          } else if (annotation instanceof retrofit2.http.Headers) {
            String[] headersToParse = ((retrofit2.http.Headers)
annotation).value();
            if (headersToParse.length == 0) {
              throw methodError(method, "@Headers annotation is empty.");
            }
            headers = parseHeaders(headersToParse);
          } else if (annotation instanceof Multipart) {
            if (isFormEncoded) {
              throw methodError(method, "Only one encoding annotation is
allowed.");
            }
            isMultipart = true;
          } else if (annotation instanceof FormUrlEncoded) {
            if (isMultipart) {
              throw methodError(method, "Only one encoding annotation is
allowed.");
            }
            isFormEncoded = true;
          }
        }

        //解析网络请求方法参数的类型和注解
        private @Nullable ParameterHandler<?> parseParameter(
            int p, Type parameterType, @Nullable Annotation[] annotations,
boolean allowContinuation) {
          ParameterHandler<?> result = null;
          if (annotations != null) {
            for (Annotation annotation : annotations) {
              ParameterHandler<?> annotationAction =
                  parseParameterAnnotation(p, parameterType, annotations,
annotation);
              ...
              result = annotationAction;
            }
          }
          ...
          return result;
        }

        //解析网络请求方法参数的类型和注解
        @Nullable
        private ParameterHandler<?> parseParameterAnnotation(
            int p, Type type, Annotation[] annotations, Annotation annotation) {
              //判断参数的注解注解
              if (annotation instanceof Url) {
                ...
                gotUrl = true;
                //判断参数的类型
                if (type == HttpUrl.class
                    || type == String.class
                    || type == URI.class
```

```
                    || (type instanceof Class &&
"android.net.Uri".equals(((Class<?>) type).getName()))) {
                  return new ParameterHandler.RelativeUrl(method, p);
                }
                ...
              } else if (annotation instanceof Path) {
                ...
                gotPath = true;
                Path path = (Path) annotation;
                String name = path.value();
                validatePathName(p, name);
                Converter<?, String> converter = retrofit.stringConverter(type,
annotations);
                return new ParameterHandler.Path<>(method, p, name, converter,
path.encoded());
              } else if (annotation instanceof Query) {
                ...
              } else if (annotation instanceof QueryName) {
                ...
              } else if (annotation instanceof QueryMap) {
                ...
              } else if (annotation instanceof Header) {
                ...
              } else if (annotation instanceof HeaderMap) {
                ...
              } else if (annotation instanceof Field) {
                ...
              } else if (annotation instanceof FieldMap) {
                ...
              } else if (annotation instanceof Part) {
                ...
              } else if (annotation instanceof PartMap) {
                ...
              }
              return null; // Not a Retrofit annotation.
            }
    }
    ...
}
```

- ServiceMethod.parseAnnotations中最后是调用HttpServiceMethod.parseAnnotations获取ServiceMethod实例的

```
 static <ResponseT, ReturnT> HttpServiceMethod<ResponseT, ReturnT>
parseAnnotations(
      Retrofit retrofit, Method method, RequestFactory requestFactory) {
    boolean isKotlinSuspendFunction = requestFactory.isKotlinSuspendFunction;
    ...
    //获取方法注解
    Annotation[] annotations = method.getAnnotations();
    Type adapterType;
    if (isKotlinSuspendFunction) {
      ...
    } else {
      //网络请求方法的返回值类型就是请求适配器的类型
```

```
      adapterType = method.getGenericReturnType();
    }
    //请求适配器
    CallAdapter<ResponseT, ReturnT> callAdapter =
        createCallAdapter(retrofit, method, adapterType, annotations);
    Type responseType = callAdapter.responseType();
    ...
    //数据解析器
    Converter<ResponseBody, ResponseT> responseConverter =
        createResponseConverter(retrofit, method, responseType);
    //从retrofit获取请求工厂，默认的话是OkHttpClient
    okhttp3.Call.Factory callFactory = retrofit.callFactory;
    if (!isKotlinSuspendFunction) {
      return new CallAdapted<>(requestFactory, callFactory, responseConverter,
callAdapter);
    }
    ...
  }

  //CallAdapted继承了HttpServiceMethod
 static final class CallAdapted<ResponseT, ReturnT> extends
HttpServiceMethod<ResponseT, ReturnT> {
    private final CallAdapter<ResponseT, ReturnT> callAdapter;

    CallAdapted(
        RequestFactory requestFactory,
        okhttp3.Call.Factory callFactory,
        Converter<ResponseBody, ResponseT> responseConverter,
        CallAdapter<ResponseT, ReturnT> callAdapter) {
      super(requestFactory, callFactory, responseConverter);
      this.callAdapter = callAdapter;
    }

    //adapt方法的实现，调用callAdapter的adapt方法
    @Override
    protected ReturnT adapt(Call<ResponseT> call, Object[] args) {
      return callAdapter.adapt(call);
    }
  }
```

- 到这里我们就知道了Retrofit的create方法中的loadServiceMethod(method).invoke(args)，实际是调用的HttpServiceMethod的invoke方法
- 那么我们看一下HttpServiceMethod的invoke方法，其中调用的adapt方法就是上面CallAdapted的adapt，传入的call类型为OkHttpCall；

```
  @Override
  final @Nullable ReturnT invoke(Object[] args) {
    Call<ResponseT> call = new OkHttpCall<>(requestFactory, args, callFactory,
responseConverter);
    return adapt(call, args);
  }
```

- 从上面代码可以知道Retrofit中的Call实际是用的OkHttpCall

OkHttpCall 的同步请求方法 execute

- 从上面Retrofit的create方法我们知道，下面代码中的call.execute实际是调用了OkHttpCall的execute方法

```
val apiService = retrofit.create(ApiService::class.java)
val call = apiService.searRepos(1, 5)
val response: Response<RepoList> = call.execute()
if (response.isSuccessful) {
    val repo = response.body()
    LjyLogUtil.d(repo.toString())
} else {
    LjyLogUtil.d(IOException("Unexpected code $response").message)
}
```

- OkHttpCall.execute代码如下，其中创建的一个okHttp库的Call对象，到这就说明了Retrofit中的网络请求实际是交给okHttp处理的;

```
    @Override
    public Response<T> execute() throws IOException {
        okhttp3.Call call;
        synchronized (this) {
          if (executed) throw new IllegalStateException("Already executed.");
          executed = true;
          //创建okhttp3.Call
          call = getRawCall();
        }

        if (canceled) {
          call.cancel();
        }

        return parseResponse(call.execute());
    }

    @GuardedBy("this")
    private okhttp3.Call getRawCall() throws IOException {
        okhttp3.Call call = rawCall;
        if (call != null) return call;
        ...
        return rawCall = createRawCall();
        ...
    }
    private okhttp3.Call createRawCall() throws IOException {
        okhttp3.Call call = callFactory.newCall(requestFactory.create(args));
        if (call == null) {
          throw new NullPointerException("Call.Factory returned null.");
        }
        return call;
    }
```

- 上面execute中通过getRawCall获取okhttp3.Call，getRawCall中又通过createRawCall中的callFactory.newCall创建okhttp3.Call，newCall的入参是通过requestFactory.create创建的请求对象

OkHttpCall 的异步请求方法 enqueue

- 上面看过了同步请求的过程，异步请求也是一样通过okHttp.Call进行异步请求的

```
  @Override
  public void enqueue(final Callback<T> callback) {
    Objects.requireNonNull(callback, "callback == null");

    okhttp3.Call call;
    Throwable failure;

    synchronized (this) {
      if (executed) throw new IllegalStateException("Already executed.");
      executed = true;

      call = rawCall;
      failure = creationFailure;
      if (call == null && failure == null) {
        try {
          call = rawCall = createRawCall();
        } catch (Throwable t) {
          throwIfFatal(t);
          failure = creationFailure = t;
        }
      }
    }

    if (failure != null) {
      callback.onFailure(this, failure);
      return;
    }

    if (canceled) {
      call.cancel();
    }

    call.enqueue(
        new okhttp3.Callback() {
          @Override
          public void onResponse(okhttp3.Call call, okhttp3.Response rawResponse)
{
            Response<T> response;
            try {
              response = parseResponse(rawResponse);
            } catch (Throwable e) {
              throwIfFatal(e);
              callFailure(e);
              return;
            }

            try {
              callback.onResponse(OkHttpCall.this, response);
            } catch (Throwable t) {
              throwIfFatal(t);
              t.printStackTrace(); // TODO this is not great
```

```
            }
          }

          @Override
          public void onFailure(okhttp3.Call call, IOException e) {
            callFailure(e);
          }

          private void callFailure(Throwable e) {
            try {
              callback.onFailure(OkHttpCall.this, e);
            } catch (Throwable t) {
              throwIfFatal(t);
              t.printStackTrace(); // TODO this is not great
            }
          }
        });
  }
```

OkHttpCall 的 parseResponse方法

- 通过上面代码我们发现OkHttpCall的同步异步请求都调用了parseResponse方法，其代码如下，其中通过 T body = responseConverter.convert(catchingBody)，用数据解析器对响应体进行解析， 这个responseConverter就是ServiceMethod的build方法调用createResponseConverter方法返回的Converter；

```
  Response<T> parseResponse(okhttp3.Response rawResponse) throws IOException {
    ResponseBody rawBody = rawResponse.body();

    // Remove the body's source (the only stateful object) so we can pass the
response along.
    rawResponse =
        rawResponse
            .newBuilder()
            .body(new NoContentResponseBody(rawBody.contentType(),
rawBody.contentLength()))
            .build();

    int code = rawResponse.code();
    if (code < 200 || code >= 300) {
      try {
        // Buffer the entire body to avoid future I/O.
        ResponseBody bufferedBody = Utils.buffer(rawBody);
        return Response.error(bufferedBody, rawResponse);
      } finally {
        rawBody.close();
      }
    }

    if (code == 204 || code == 205) {
      rawBody.close();
      return Response.success(null, rawResponse);
    }
```

```
    ExceptionCatchingResponseBody catchingBody = new
ExceptionCatchingResponseBody(rawBody);
    try {
      T body = responseConverter.convert(catchingBody);
      return Response.success(body, rawResponse);
    } catch (RuntimeException e) {
      // If the underlying source threw an exception, propagate that rather than
indicating it was
      // a runtime exception.
      catchingBody.throwIfCaught();
      throw e;
    }
  }
```

Retrofit中的设计模式

Builder（建造者）模式

- 将复杂对象的构建和表示相分离，使复杂对象的构建简单化
- 防止构造方法参数过多，造成使用者使用不便，通过链式调用不同方法设置不同参数

```
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    .client(okHttpClient)
    .addConverterFactory(GsonConverterFactory.create())
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build()
```

- 关于建造者模式如需了解更多内容可以查看Android设计模式-2-建造者模式

工厂模式

- 将“类实例化的操作”与“使用对象的操作”分开，降低耦合，易于扩展，有利于产品的一致性

```
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    .client(okHttpClient)
    // 自定义的Converter一定要放在官方提供的Converter前面
    .addConverterFactory(StringConverterFactory.create())
    .addConverterFactory(MapConverterFactory.create())
    .addConverterFactory(GsonConverterFactory.create())
    .addCallAdapterFactory(LjyCallAdapterFactory.create())
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build()
```

- 关于工厂模式如需了解更多内容可以查看：
-
    - Android设计模式-4.1-简单工厂模式
    - Android设计模式-4.2-工厂方法模式
    - Android设计模式-5-抽象工厂模式

策略模式

— 策略类之间可以自由切换，由于策略类都实现同一个接口，所以使它们之间可以自由切换。

- 易于扩展，增加一个新的策略只需要添加一个具体的策略类即可，基本不需要改变原有的代码，符合“开闭原则“

```
val retrofit = Retrofit.Builder()
    .baseUrl(baseUrl)
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .addCallAdapterFactory(Java8CallAdapterFactory.create())
    .addCallAdapterFactory(GuavaCallAdapterFactory.create())
    .build()
```

- 关于策略模式如需了解更多内容可以查看Android设计模式-6-策略模式

观察者模式

- 定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新

```
//call为被观察者，Callback为观察者
call.enqueue(object : Callback<RepoList> {
    override fun onResponse(call: Call<RepoList>, result: Response<RepoList>) {
        if (result.body() != null) {
            val repoResult: RepoList = result.body()!!
            for (it in repoResult.items) {
                LjyLogUtil.d("${it.name}_${it.starCount}")
            }
        }
    }

    override fun onFailure(call: Call<RepoList>, t: Throwable) {
        LjyLogUtil.d("onFailure:${t.message}")
    }

})
```

- 关于观察者模式如需了解更多内容可以查看Android设计模式-11-观察者模式

适配器模式

- 定义一个包装类，用于包装不兼容接口的对象
- 可以让没有关联的类一起运行，提高了类的复用

```
final class DefaultCallAdapterFactory extends CallAdapter.Factory {
  private final @Nullable Executor callbackExecutor;

  DefaultCallAdapterFactory(@Nullable Executor callbackExecutor) {
    this.callbackExecutor = callbackExecutor;
  }

  @Override
  public @Nullable CallAdapter<?, ?> get(
      Type returnType, Annotation[] annotations, Retrofit retrofit) {
```

```
    ...
    final Type responseType = Utils.getParameterUpperBound(0, (ParameterizedType)
returnType);

    final Executor executor =
        Utils.isAnnotationPresent(annotations, SkipCallbackExecutor.class)
            ? null
            : callbackExecutor;

    return new CallAdapter<Object, Call<?>>() {
      @Override
      public Type responseType() {
        return responseType;
      }

      @Override
      public Call<Object> adapt(Call<Object> call) {
        return executor == null ? call : new ExecutorCallbackCall<>(executor,
call);
      }
    };
  }
}
```

- 适配器模式在Android中最常见的使用就是listView，recycleView的Adapter
- 关于适配器模式如需了解更多内容可以查看Android设计模式-19-适配器模式

装饰模式

- 动态扩展一个实现类的功能，装饰类和被装饰类可以独立发展，不会相互耦合

```
//ExecutorCallbackCall是装饰者，而里面真正去执行网络请求的还是OkHttpCall
static final class ExecutorCallbackCall<T> implements Call<T> {
    final Executor callbackExecutor;
    final Call<T> delegate;

    ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {
      this.callbackExecutor = callbackExecutor;
      this.delegate = delegate;
    }

    @Override
    public void enqueue(final Callback<T> callback) {
      Objects.requireNonNull(callback, "callback == null");

      delegate.enqueue(
          new Callback<T>() {
            @Override
            public void onResponse(Call<T> call, final Response<T> response) {
              callbackExecutor.execute(
                  () -> {
                    if (delegate.isCanceled()) {
                      // Emulate OkHttp's behavior of throwing/delivering an
IOException on
                      // cancellation.
```

```
                    callback.onFailure(ExecutorCallbackCall.this, new
IOException("Canceled"));
                  } else {
                    callback.onResponse(ExecutorCallbackCall.this, response);
                  }
                });
            }

            @Override
            public void onFailure(Call<T> call, final Throwable t) {
              callbackExecutor.execute(() ->
callback.onFailure(ExecutorCallbackCall.this, t));
            }
          });
    }
}
```

- 关于装饰模式如需了解更多内容可以查看Android设计模式-20-装饰模式

外观模式

- 为复杂的模块或子系统提供外界访问的接口
- 通过创建一个统一的类，用来包装子系统中一个或多个复杂的类
- Retrofit类就是Retrofit框架提供给我们的外观类

```
public final class Retrofit {
    private final Map<Method, ServiceMethod<?>> serviceMethodCache = new
ConcurrentHashMap<>();

    final okhttp3.Call.Factory callFactory;
    final HttpUrl baseUrl;
    final List<Converter.Factory> converterFactories;
    final List<CallAdapter.Factory> callAdapterFactories;
    final @Nullable Executor callbackExecutor;
    final boolean validateEagerly;
    ...
}
```

- 关于外观模式如需了解更多内容可以查看Android设计模式-22-外观模式

代理模式

- 也称委托模式，间接访问目标对象, 分为静态代理和动态代理

```
//Retrofit.create
public <T> T create(final Class<T> service) {
    validateServiceInterface(service);
    return (T)
        //通过动态代理创建接口的实例
        Proxy.newProxyInstance(
            //参数1：classLoader
            service.getClassLoader(),
            //参数2：接口类型数组
            new Class<?>[] {service},
            //参数3：实现了InvocationHandler的代理类
```

```
            new InvocationHandler() {
             ...
            });
}
```

1. 静态代理

```
abstract class AbsObject {
    abstract fun doSomething()
}

class RealObject : AbsObject() {
    override fun doSomething() {
        LjyLogUtil.d("RealObject.doSomething")
    }
}

class ProxyObject(private val realObject: RealObject) : AbsObject() {
    override fun doSomething() {
        LjyLogUtil.d("before RealObject")
        realObject.doSomething()
        LjyLogUtil.d("after RealObject")
    }
}

//使用
val realObject = RealObject()
val proxyObject = ProxyObject(realObject)
proxyObject.doSomething()
```

2. 动态代理

```
interface Subject {
    fun doSomething()
}

class Test : Subject {
    override fun doSomething() {
        LjyLogUtil.d("Test.doSomething")
    }
}

class DynamicProxy(private val target: Subject) : InvocationHandler {
    override fun invoke(proxy: Any?, method: Method?, args: Array<out Any>?):
Any? {
        LjyLogUtil.d("Proxy: ${proxy?.javaClass?.name}")
        LjyLogUtil.d("before target")
        //Kotlin中数组转为可变长参数，通过前面加*符号
        val invoke = method!!.invoke(target, *(args ?: emptyArray()))
        LjyLogUtil.d("after target")
        return invoke
    }
}
```

```
//使用:
val test = Test()
val myProxy = DynamicProxy(test)
val subject: Subject =
    Proxy.newProxyInstance(
        test.javaClass.classLoader,
        test.javaClass.interfaces,
        myProxy
    ) as Subject
subject.doSomething()
LjyLogUtil.d("subject.className:" + subject.javaClass.name)
```

- 关于代理模式如需了解更多内容可以查看Android设计模式-17-代理模式