

## 24 | HTTP网络编程与JSON解析

2019-08-22 陈航

Flutter核心技术与实战

[进入课程 >](#)



**讲述：陈航**

时长 14:27 大小 13.24M



你好，我是陈航。

在上一篇文章中，我带你一起学习了 Dart 中异步与并发的机制及实现原理。与其他语言类似，Dart 的异步是通过事件循环与队列实现的，我们可以使用 Future 来封装异步任务。而另一方面，尽管 Dart 是基于单线程模型的，但也提供了 Isolate 这样的“多线程”能力，这使得我们可以充分利用系统资源，在并发 Isolate 中搞定 CPU 密集型的任务，并通过消息机制通知主 Isolate 运行结果。

异步与并发的一个典型应用场景，就是网络编程。一个好的移动应用，不仅需要有良好的界面和易用的交互体验，也需要具备和外界进行信息交互的能力。而通过网络，信息隔离的客户端与服务端间可以建立一个双向的通信通道，从而实现资源访问、接口数据请求和提交、上传下载文件等操作。

为了便于我们快速实现基于网络通道的信息交换实时更新 App 数据，Flutter 也提供了一系列的网络编程类库和工具。因此在今天的分享中，我会通过一些小例子与你讲述在 Flutter 应用中，如何实现与服务端的数据交互，以及如何将交互响应的数据格式化。

## Http 网络编程

我们在通过网络与服务端数据交互时，不可避免地需要用到三个概念：定位、传输与应用。

其中，**定位**，定义了如何准确地找到网络上的一台或者多台主机（即 IP 地址）；**传输**，则主要负责在找到主机后如何高效且可靠地进行数据通信（即 TCP、UDP 协议）；而**应用**，则负责识别双方通信的内容（即 HTTP 协议）。

我们在进行数据通信时，可以只使用传输层协议。但传输层传递的数据是二进制流，如果没有应用层，我们无法识别数据内容。如果想要使传输的数据有意义，则必须要用到应用层协议。移动应用通常使用 HTTP 协议作应用层协议，来封装 HTTP 信息。

在编程框架中，一次 HTTP 网络调用通常可以拆解为以下步骤：

1. 创建网络调用实例 client，设置通用请求行为（如超时时间）；
2. 构造 URI，设置请求 header、body；
3. 发起请求，等待响应；
4. 解码响应的内容。

当然，Flutter 也不例外。在 Flutter 中，Http 网络编程的实现方式主要分为三种：dart:io 里的 HttpClient 实现、Dart 原生 http 请求库实现、第三方库 dio 实现。接下来，我依次为你讲解这三种方式。


## HttpClient

HttpClient 是 dart:io 库中提供的网络请求类，实现了基本的网络编程功能。

接下来，我将和你分享一个实例，对照着上面提到的网络调用步骤，来演示 HttpClient 如何使用。

在下面的代码中，我们创建了一个 HttpClient 网络调用实例，设置了其超时时间为 5 秒。随后构造了 Flutter 官网的 URI，并设置了请求 Header 的 user-agent 为 Custom-UA。

然后发起请求，等待 Flutter 官网响应。最后在收到响应后，打印出返回结果：

 复制代码

```
1 get() async {
2   // 创建网络调用示例，设置通用请求行为（超时时间）
3   var httpClient = HttpClient();
4   httpClient.idleTimeout = Duration(seconds: 5);
5
6   // 构造 URI，设置 user-agent 为 "Custom-UA"
7   var uri = Uri.parse("https://flutter.dev");
8   var request = await httpClient.getUrl(uri);
9   request.headers.add("user-agent", "Custom-UA");
10
11  // 发起请求，等待响应
12  var response = await request.close();
13
14  // 收到响应，打印结果
15  if (response.statusCode == HttpStatus.ok) {
16    print(await response.transform(utf8.decoder).join());
17  } else {
18    print('Error: \nHttp status ${response.statusCode}');
19  }
20 }
```

可以看到，使用 HttpClient 来发起网络调用还是相对比较简单。

这里需要注意的是，由于网络请求是异步行为，因此在 **Flutter 中，所有网络编程框架都是以 Future 作为异步请求的包装**，所以我们需要使用 await 与 async 进行非阻塞的等待。当然，你也可以注册 then，以回调的方式进行相应的事件处理。

## http

HttpClient 使用方式虽然简单，但其接口却暴露了不少内部实现细节。比如，异步调用拆分得过细，链接需要调用方主动关闭，请求结果是字符串但却需要手动解码等。

http 是 Dart 官方提供的另一个网络请求类，相比于 HttpClient，易用性提升了不少。同样，我们以一个例子来介绍 http 的使用方法。

首先，我们需要将 http 加入到 pubspec 中的依赖里：

```
1 dependencies:
2   http: '>=0.11.3+12'
```

在下面的代码中，与 HttpClient 的例子类似的，我们也是先后构造了 http 网络调用实例和 Flutter 官网 URI，在设置 user-agent 为 Custom-UA 后，发出请求，最后打印请求结果：

```
1 httpGet() async {
2   // 创建网络调用示例
3   var client = http.Client();
4
5   // 构造 URI
6   var uri = Uri.parse("https://flutter.dev");
7
8   // 设置 user-agent 为 "Custom-UA"，随后立即发出请求
9   http.Response response = await client.get(uri, headers : {"user-agent" : "Custom-UA"})
10
11   // 打印请求结果
12   if(response.statusCode == HttpStatus.ok) {
13     print(response.body);
14   } else {
15     print("Error: ${response.statusCode}");
16   }
17 }
```

可以看到，相比于 HttpClient，http 的使用方式更加简单，仅需一次异步调用就可以实现基本的网络通信。

## dio

HttpClient 和 http 使用方式虽然简单，但其暴露的定制化能力都相对较弱，很多常用的功能都不支持（或者实现异常繁琐），比如取消请求、定制拦截器、Cookie 管理等。因此对于复杂的网络请求行为，我推荐使用目前在 Dart 社区人气较高的第三方 dio 来发起网络请求。

接下来，我通过几个例子来和你介绍 dio 的使用方法。与 http 类似的，我们首先需要把 dio 加到 pubspec 中的依赖里：

```
1 dependencies:
2   dio: '>2.1.3'
```

在下面的代码中，与前面 HttpClient 与 http 例子类似的，我们也是先后创建了 dio 网络调用实例、创建 URI、设置 Header、发出请求，最后等待请求结果：

```
1 void getRequest() async {
2   // 创建网络调用示例
3   Dio dio = new Dio();
4
5   // 设置 URI 及请求 user-agent 后发起请求
6   var response = await dio.get("https://flutter.dev", options:Options(headers: {"user-a{
7
8   // 打印请求结果
9   if(response.statusCode == HttpStatus.ok) {
10    print(response.data.toString());
11  } else {
12    print("Error: ${response.statusCode}");
13  }
14 }
```

这里需要注意的是，创建 URI、设置 Header 及发出请求的行为，都是通过 dio.get 方法实现的。这个方法 options 参数提供了精细化控制网络请求的能力，可以支持设置 Header、超时时间、Cookie、请求方法等。这部分内容不是今天分享的重点，如果你想深入理解的话，可以访问其[API 文档](#)学习具体使用方法。

对于常见的上传及下载文件需求，dio 也提供了良好的支持：文件上传可以通过构建表单 FormData 实现，而文件下载则可以使用 download 方法搞定。

在下面的代码中，我们通过 FormData 创建了两个待上传的文件，通过 post 方法发送至服务端。download 的使用方法则更为简单，我们直接在请求参数中，把待下载的文件地址和本地文件名提供给 dio 即可。如果我们需要感知下载进度，可以增加 onReceiveProgress 回调函数：

```

1 // 使用 FormData 表单构建待上传文件
2 FormData formData = FormData.from({
3   "file1": UploadFileInfo(File("./file1.txt"), "file1.txt"),
4   "file2": UploadFileInfo(File("./file2.txt"), "file1.txt"),
5
6 });
7 // 通过 post 方法发送至服务端
8 var responseY = await dio.post("https://xxx.com/upload", data: formData);
9 print(responseY.toString());
10
11 // 使用 download 方法下载文件
12 dio.download("https://xxx.com/file1", "xx1.zip");
13
14 // 增加下载进度回调函数
15 dio.download("https://xxx.com/file1", "xx2.zip", onReceiveProgress: (count, total) {
16   //do something
17 });

```

有时，我们的页面由多个并行的请求响应结果构成，这就需要等待这些请求都返回后才能刷新界面。在 dio 中，我们可以结合 Future.wait 方法轻松实现：

```

1 // 同时发起两个并行请求
2 List<Response> responseX= await Future.wait([dio.get("https://flutter.dev"),dio.get("ht
3
4 // 打印请求 1 响应结果
5 print("Response1: ${responseX[0].toString()}");
6 // 打印请求 2 响应结果
7 print("Response2: ${responseX[1].toString()}");

```

此外，与 Android 的 okHttp 一样，dio 还提供了请求拦截器，通过拦截器，我们可以在请求之前，或响应之后做一些特殊的操作。比如可以为请求 option 统一增加一个 header，或是返回缓存数据，或是增加本地校验处理等等。

在下面的例子中，我们为 dio 增加了一个拦截器。在请求发送之前，不仅为每个请求头都加上了自定义的 user-agent，还实现了基本的 token 认证信息检查功能。而对于本地已经缓存了请求 uri 资源的场景，我们可以直接返回缓存数据，避免再次下载：

```
1 // 增加拦截器
2 dio.interceptors.add(InterceptorsWrapper(
3   onRequest: (RequestOptions options){
4     // 为每个请求头都增加 user-agent
5     options.headers["user-agent"] = "Custom-UA";
6     // 检查是否有 token，没有则直接报错
7     if(options.headers['token'] == null) {
8       return dio.reject("Error: 请先登录 ");
9     }
10    // 检查缓存是否有数据
11    if(options.uri == Uri.parse('http://xxx.com/file1')) {
12      return dio.resolve(" 返回缓存数据 ");
13    }
14    // 放行请求
15    return options;
16  }
17 ));
18
19 // 增加 try catch, 防止请求报错
20 try {
21   var response = await dio.get("https://xxx.com/xxx.zip");
22   print(response.data.toString());
23 }catch(e) {
24   print(e);
25 }
```

需要注意的是，由于网络通信期间有可能会异常（比如，域名无法解析、超时等），因此我们需要使用 try-catch 来捕获这些未知错误，防止程序出现异常。

除了这些基本的用法，dio 还支持请求取消、设置代理，证书校验等功能。不过，这些高级特性不属于本次分享的重点，故不再赘述，详情可以参考 dio 的[GitHub 主页](#)了解具体用法。


## JSON 解析

移动应用与 Web 服务器建立好了连接之后，接下来的两个重要工作分别是：服务器如何结构化地去描述返回的通信信息，以及移动应用如何解析这些格式化的信息。

### 如何结构化地描述返回的通信信息？

在如何结构化地去表达信息上，我们需要用到 JSON。JSON 是一种轻量级的、用于表达由属性值和字面量组成对象的数据交换语言。

一个简单的表示学生成绩的 JSON 结构，如下所示：

 复制代码

```
1 String jsonString = '''
2 {
3   "id":"123",
4   "name":" 张三 ",
5   "score" : 95
6 }
7 ''';
```

需要注意的是，由于 Flutter 不支持运行时反射，因此并没有提供像 Gson、Mantle 这样自动解析 JSON 的库来降低解析成本。在 Flutter 中，JSON 解析完全是手动的，开发者要做的事情多了一些，但使用起来倒也相对灵活。


接下来，我们就看看 Flutter 应用是如何解析这些格式化的信息。

## 如何解析格式化的信息？

所谓手动解析，是指使用 dart:convert 库中内置的 JSON 解码器，将 JSON 字符串解析成自定义对象的过程。使用这种方式，我们需要先将 JSON 字符串传递给 JSON.decode 方法解析成一个 Map，然后把这个 Map 传给自定义的类，进行相关属性的赋值。

以上面表示学生成绩的 JSON 结构为例，我来和你演示手动解析的使用方法。

首先，我们根据 JSON 结构定义 Student 类，并创建一个工厂类，来处理 Student 类属性成员与 JSON 字典对象的值之间的映射关系：

 复制代码

```
1 class Student{
2   // 属性 id, 名字与成绩
3   String id;
4   String name;
5   int score;
6   // 构造方法
7   Student({
8     this.id,
9     this.name,
10    this.score
```




```

11     });
12     //JSON 解析工厂类，使用字典数据为对象初始化赋值
13     factory Student.fromJson(Map<String, dynamic> parsedJson){
14         return Student(
15             id: parsedJson['id'],
16             name : parsedJson['name'],
17             score : parsedJson ['score']
18         );
19     }
20 }

```

数据解析类创建好了，剩下的事情就相对简单了，我们只需要把 JSON 文本通过 `JSON.decode` 方法转换成 Map，然后把它交给 Student 的工厂类 `fromJson` 方法，即可完成 Student 对象的解析：


 复制代码

```

1 loadStudent() {
2     //jsonString 为 JSON 文本
3     final jsonResponse = json.decode(jsonString);
4     Student student = Student.fromJson(jsonResponse);
5     print(student.name);
6 }

```

在上面的例子中，JSON 文本所有的属性都是基本类型，因此我们直接从 JSON 字典取出相应的元素为对象赋值即可。而如果 JSON 下面还有嵌套对象属性，比如下面的例子中，Student 还有一个 teacher 的属性，我们又该如何解析呢？


 复制代码

```

1 String jsonString = '''
2 {
3     "id": "123",
4     "name": " 张三 ",
5     "score" : 95,
6     "teacher": {
7         "name": " 李四 ",
8         "age" : 40
9     }
10 }
11 ''';


```

这里，teacher 不再是一个基本类型，而是一个对象。面对这种情况，我们需要为每一个非基本类型属性创建一个解析类。与 Student 类似，我们也需要为它的属性 teacher 创建一个解析类 Teacher：

 复制代码


```
1 class Teacher {
2     //Teacher 的名字与年龄
3     String name;
4     int age;
5     // 构造方法
6     Teacher({this.name,this.age});
7     //JSON 解析工厂类，使用字典数据为对象初始化赋值
8     factory Teacher.fromJson(Map<String, dynamic> parsedJson){
9         return Teacher(
10             name : parsedJson['name'],
11             age : parsedJson ['age']
12         );
13     }
14 }
```

然后，我们只需要在 Student 类中，增加 teacher 属性及对应的 JSON 映射规则即可：

 复制代码

```
1 class Student{
2     ...
3     // 增加 teacher 属性
4     Teacher teacher;
5     // 构造函数增加 teacher
6     Student({
7         ...
8         this.teacher
9     });
10     factory Student.fromJson(Map<String, dynamic> parsedJson){
11         return Student(
12             ...
13             // 增加映射规则
14             teacher: Teacher.fromJson(parsedJson ['teacher'])
15         );
16     }
17 }
```

完成了 teacher 属性的映射规则添加之后，我们就可以继续使用 Student 来解析上述的 JSON 文本了：

 复制代码

```
1 final jsonResponse = json.decode(jsonString); // 将字符串解码成 Map 对象
2 Student student = Student.fromJson(jsonResponse); // 手动解析
3 print(student.teacher.name);
```

可以看到，通过这种方法，无论对象有多复杂的非基本类型属性，我们都可以创建对应的解析类进行处理。

不过到现在为止，我们的 JSON 数据解析还是在主 Isolate 中完成。如果 JSON 的数据格式比较复杂，数据量又大，这种解析方式可能会造成短期 UI 无法响应。对于这类 CPU 密集型的操作，我们可以使用上一篇文章中提到的 compute 函数，将解析工作放到新的 Isolate 中完成：

 复制代码

```
1 static Student parseStudent(String content) {
2   final jsonResponse = json.decode(content);
3   Student student = Student.fromJson(jsonResponse);
4   return student;
5 }
6 doSth() {
7   ...
8   // 用 compute 函数将 json 解析放到新 Isolate
9   compute(parseStudent, jsonString).then((student)=>print(student.teacher.name));
10 }
```

通过 compute 的改造，我们就不用担心 JSON 解析时间过长阻塞 UI 响应了。

## 总结

好了，今天的分享就到这里了，我们简单回顾一下主要内容。

首先，我带你学习了实现 Flutter 应用与服务端通信的三种方式，即 HttpClient、http 与 dio。其中 dio 提供的功能更为强大，可以支持请求拦截、文件上传下载、请求合并等高级

能力。因此，我推荐你在实际项目中使用 dio 的方式。

然后，我和你分享了 JSON 解析的相关内容。JSON 解析在 Flutter 中相对比较简单，但由于不支持反射，所以我们只能手动解析，即：先将 JSON 字符串转换成 Map，然后再把这个 Map 给到自定义类，进行相关属性的赋值。

如果你有原生 Android、iOS 开发经验的话，可能会觉得 Flutter 提供的 JSON 手动解析方案并不好用。在 Flutter 中，没有像原生开发那样提供了 Gson 或 Mantle 等库，用于将 JSON 字符串直接转换为对应的实体类。而这些能力无一例外都需要用到运行时反射，这是 Flutter 从设计之初就不支持的，理由如下：

1. 运行时反射破坏了类的封装性和安全性，会带来安全风险。就在前段时间，Fastjson 框架就爆出了一个巨大的安全漏洞。这个漏洞使得精心构造的字符串文本，可以在反序列化时让服务器执行任意代码，直接导致业务机器被远程控制、内网渗透、窃取敏感信息等操作。
2. 运行时反射会增加二进制文件大小。因为搞不清楚哪些代码可能会在运行时用到，因此使用反射后，会默认使用所有代码构建应用程序，这就导致编译器无法优化编译期间未使用的代码，应用安装包体积无法进一步压缩，这对于自带 Dart 虚拟机的 Flutter 应用程序是难以接受的。

反射给开发者编程带来了方便，但也带来了许多难以解决的新问题，因此 Flutter 并不支持反射。而我们要做的就是，老老实实地手动解析 JSON 吧。

我把今天分享所涉及到的知识点打包到了[GitHub](#)中，你可以下载下来，反复运行几次，加深理解与记忆。

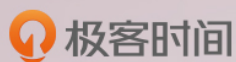
## 思考题

最后，我给你留两道思考题吧。

1. 请使用 dio 实现一个自定义拦截器，拦截器内检查 header 中的 token：如果没有 token，需要暂停本次请求，同时访问"<http://xxxx.com/token>"，在获取新 token 后继续本次请求。
2. 为以下 Student JSON 写相应的解析类：

```
1 String jsonString = '''
2 {
3     "id":"123",
4     "name":" 张三 ",
5     "score" : 95,
6     "teachers": [
7         {
8             "name": " 李四 ",
9             "age" : 40
10        },
11        {
12            "name": " 王五 ",
13            "age" : 45
14        }
15    ]
16 }
17 ''';
```

欢迎你在评论区给我留言分享你的观点，我会在下一篇文章中等待你！感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。



# Flutter 核心技术与实战

来自 Google 的高性能跨平台开发框架

陈航

美团点评高级技术专家



新版升级：点击「👉 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言 (6)

写留言



给我点阳光就灿烂

2019-08-22

如何进行socket通信

展开 ▾

1

2



ls

2019-08-24

还是不大理解 await 和 async，当发起网络请求的时候，不会阻塞主线程吗。当调用 async 的函数时，是另外起了一个线程去等待网络请求返回？这个异步怎么理解好。

作者回复: 网络请求和I/O是另一个平行世界（操作系统）里并发完成的，flutter只是触发了他们的启动而已

1

1



吴小安

2019-08-23

这个反射引起的安全问题在移动端和前端现在有解决？

展开 ▾

作者回复: 在可用性和安全的综合衡量下，一般是采用类的黑名单，避免反序列化有安全风险的类

1

1



右手边

2019-08-22

老师您好，JSON 的解析确实有点复杂了，真的没有类似Gson那样方便的库么？

作者回复: 运行时的没有，有一些开发期的IDE插件可以简化JSON解析的代码

1

1



**许童童**

2019-08-22

习惯了用js直接获取属性，这样先定义模型确实会更废时间一些，但先定义模型其实会更规范、出bug更容易调试，也要求开发人员不能随便变更数据结构。

展开 ▾



**江厚宏**

2019-08-22

老师能不能介绍一下反序列化工具，比如json\_serializable和 built\_value，建议用哪一个，如果遇到泛型，该如何处理

展开 ▾

