

本章目录

- [文件操作](#)
- [Http请求-HttpClient](#)
- [Http请求-Dio package](#)
- [WebSocket](#)
- [使用Socket API](#)
- [Json转Model](#)

文件操作

Dart的IO库包含了文件读写的相关类，它属于Dart语法标准的一部分，所以通过Dart IO库，无论是Dart VM下的脚本还是Flutter，都是通过Dart IO库来操作文件的，不过和Dart VM相比，Flutter有一个重要差异是文件系统路径不同，这是因为Dart VM是运行在PC或服务器操作系统下，而Flutter是运行在移动操作系统中，他们的文件系统会有一些差异。

APP目录

Android和iOS的应用存储目录不同，`PathProvider` 插件提供了一种平台透明的方式来访问设备文件系统上的常用位置。该类当前支持访问两个文件系统位置：

- **临时目录:** 可以使用 `getTemporaryDirectory()` 来获取临时目录；系统可随时清除的临时目录（缓存）。在iOS上，这对应于 `NSTemporaryDirectory()` 返回的值。在Android上，这是 `getCacheDir()` 返回的值。
- **文档目录:** 可以使用 `getApplicationDocumentsDirectory()` 来获取应用程序的文档目录，该目录用于存储只有自己可以访问的文件。只有当应用程序被卸载时，系统才会清除该目录。在iOS上，这对应于 `NSDocumentDirectory`。在Android上，这是 `AppData` 目录。
- **外部存储目录:** 可以使用 `getExternalStorageDirectory()` 来获取外部存储目录，如SD卡；由于iOS不支持外部目录，所以在iOS下调用该方法会抛出 `UnsupportedError` 异常，而在Android下结果是android SDK中 `getExternalStorageDirectory` 的返回值。

一旦你的Flutter应用程序有一个文件位置的引用，你可以使用[dart:io](#) API来执行对文件系统的读/写操作。有关使用Dart处理文件和目录的详细内容可以参考Dart语言文档，下面我们看一个简单的例子。

示例

我们还是以计数器为例，实现在应用退出重启后可以恢复点击次数。这里，我们使用文件来保存数据：

1. 引入PathProvider插件；在 `pubspec.yaml` 文件中添加如下声明：

```
path_provider: ^0.4.1
```

添加后，执行 `flutter packages get` 获取一下，版本号可能随着时间推移会发生变化，读者可以使用最新版。

2. 实现：

```
import 'dart:io';
import 'dart:async';
import 'package:flutter/material.dart';
import 'package:path_provider/path_provider.dart';

class FileOperationRoute extends StatefulWidget {
  FileOperationRoute({Key key}) : super(key: key);

  @override
  _FileOperationRouteState createState() => new
  _FileOperationRouteState();
}

class _FileOperationRouteState extends State<FileOperationRoute>
{
  int _counter;

  @override
  void initState() {
    super.initState();
    // 从文件读取点击次数
    _readCounter().then((int value) {
      setState(() {
        _counter = value;
      });
    });
  }
}
```

```

    });
}

Future<File> _getLocalFile() async {
    // 获取应用目录
    String dir = (await
getApplicationDocumentsDirectory()).path;
    return new File('$dir/counter.txt');
}

Future<int> _readCounter() async {
    try {
        File file = await _getLocalFile();
        // 读取点击次数 (以字符串)
        String contents = await file.readAsString();
        return int.parse(contents);
    } on FileSystemException {
        return 0;
    }
}

Future<Null> _incrementCounter() async {
    setState(() {
        _counter++;
    });
    // 将点击次数以字符串类型写到文件中
    await (await _getLocalFile()).writeAsString('$_counter');
}

@override
Widget build(BuildContext context) {
    return new Scaffold(
        appBar: new AppBar(title: new Text('文件操作')),
        body: new Center(
            child: new Text('点击了 $_counter 次'),
        ),
        floatingActionButton: new FloatingActionButton(
            onPressed: _incrementCounter,
            tooltip: 'Increment',
            child: new Icon(Icons.add),
        ),
    );
}
}

```

上面代码比较简单，不再赘述，需要说明的是，本示例只是为了演示文件读

写，而在实际开发中，如果要存储一些简单的数据，使用shared_preferences插件会比较简单。

注意，Dart IO库操作文件的API非常丰富，但本书不是介绍Dart语言的，故不详细说明，读者需要的话可以自行学习。

通过HttpClient发起HTTP请求

Dart IO库中提供了HttpRequest的一些类，我们可以直接使用HttpClient来发起请求。使用HttpClient发起请求分为五步：

1. 创建一个HttpClient

```
HttpClient httpClient = new HttpClient();
```

2. 打开Http连接，设置请求头

```
HttpClientRequest request = await httpClient.getUrl(uri);
```

这一步可以使用任意Http method，如 `httpClient.post(...)`、`httpClient.delete(...)` 等。如果包含Query参数，可以在构建uri时添加，如：

```
Uri uri=Uri(scheme: "https", host: "flutterchina.club",
  queryParameters: {
    "xx":"xx",
    "yy":"dd"
  });
```

通过HttpClientRequest可以设置请求header，如：

```
request.headers.add("user-agent", "test");
```

如果是post或put等可以携带请求体方法，可以通过HttpClientRequest对象发送request body，如：

```
String payload="...";  
request.add(utf8.encode(payload));  
//request.addStream(_inputStream); //可以直接添加输入流
```

3. 等待连接服务器

```
HttpClientResponse response = await request.close();
```

这一步完成后，请求信息就已经发送给服务器了，返回一个 HttpClientResponse 对象，它包含响应头（header）和响应流（响应体的 Stream），接下来就可以通过读取响应流来获取响应内容。

4. 读取响应内容

```
String responseBody = await  
response.transform(utf8.decoder).join();
```

我们通过读取响应流来获取服务器返回的数据，在读取时我们可以设置编码格式，这里是 utf8。

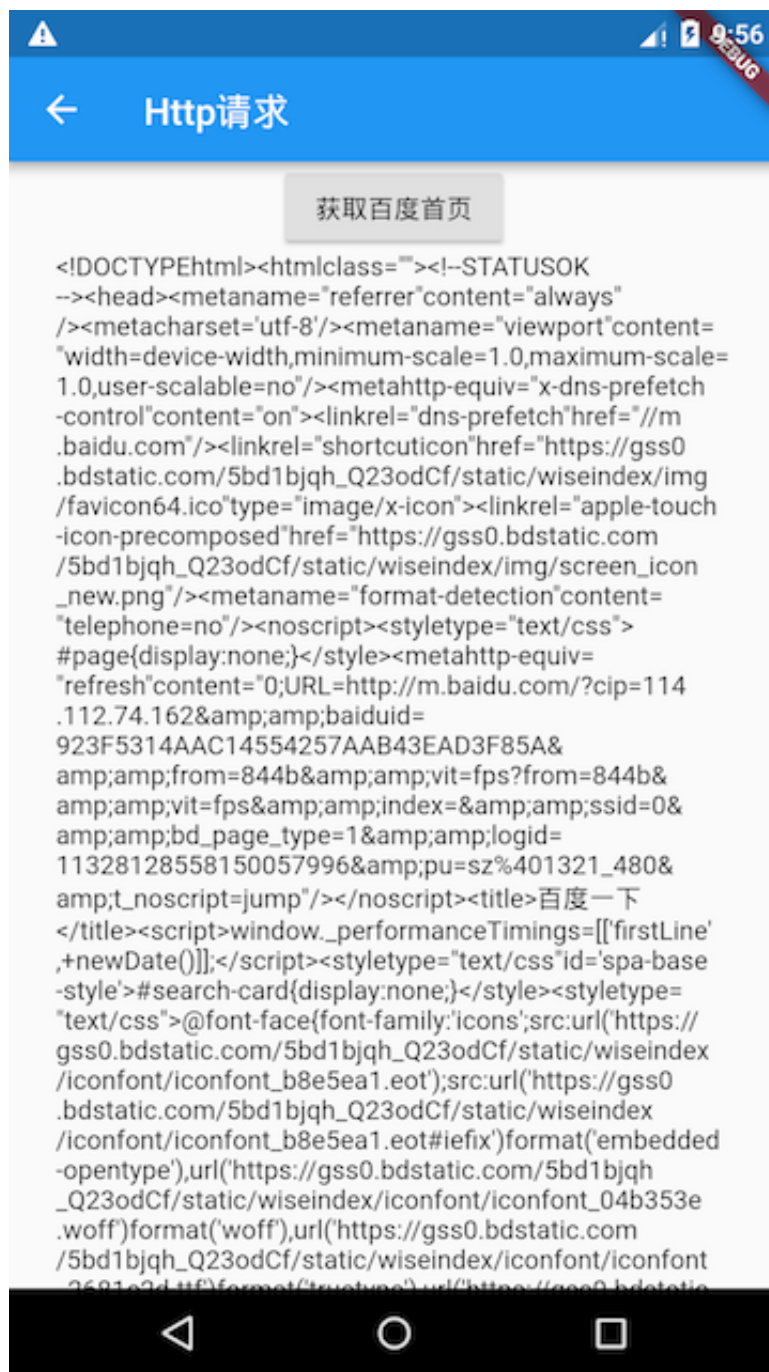
5. 请求结束，关闭 HttpClient

```
httpClient.close();
```

关闭 client 后，通过该 client 发起的所有请求都会中止。

示例

我们实现一个获取百度首页 html 的例子，示例效果如下：



点击“获取百度首页”按钮后，会请求百度首页，请求成功后，我们将返回内容显示出来并在控制台打印响应header，代码如下：

```
import 'dart:convert';
import 'dart:io';

import 'package:flutter/material.dart';

class HttpTestRoute extends StatefulWidget {
```

```

class HttpTestRoute extends StatefulWidget {
  @override

  _HttpTestRouteState createState() => new _HttpTestRouteState();
}

class _HttpTestRouteState extends State<HttpTestRoute> {
  bool _loading = false;
  String _text = "";

  @override
  Widget build(BuildContext context) {
    return ConstrainedBox(
      constraints: BoxConstraints.expand(),
      child: SingleChildScrollView(
        child: Column(
          children: <Widget>[
            RaisedButton(
              child: Text("获取百度首页"),
              onPressed: _loading ? null : () async {
                setState(() {
                  _loading = true;
                  _text = "正在请求...";
                });
                try {
                  // 创建一个HttpClient
                  HttpClient httpClient = new HttpClient();
                  // 打开Http连接
                  HttpClientRequest request = await
httpClient.getUrl(
                      Uri.parse("https://www.baidu.com"));
                  // 使用iPhone的UA
                  request.headers.add("user-agent", "Mozilla/5.0
(iPhone; CPU iPhone OS 10_3_1 like Mac OS X) AppleWebKit/603.1.30
(KHTML, like Gecko) Version/10.0 Mobile/14E304 Safari/602.1");
                  // 等待连接服务器 (会将请求信息发送给服务器)
                  HttpClientResponse response = await
request.close();

                  // 读取响应内容
                  _text = await
response.transform(utf8.decoder).join();
                  // 输出响应头
                  print(response.headers);

                  // 关闭client后, 通过该client发起的所有请求都会中止。
                  httpClient.close();

                } catch (e) {
                  _text = "连接失败, 重试";
                }
              },
            ),
          ],
        ),
      ),
    );
  }
}

```

```

        _text = 请求失败: $e ;
    } finally {
        setState(() {
            _loading = false;
        });
    }
}

),
Container(
    width: MediaQuery.of(context).size.width-50.0,
    child: Text(_text.replaceAll(new RegExp(r"\s"),
""))
)
),
),
);
}
}

```

控制台输出:

```

I/flutter (18545): connection: Keep-Alive
I/flutter (18545): cache-control: no-cache
I/flutter (18545): set-cookie: .... //有多个, 省略...
I/flutter (18545): transfer-encoding: chunked
I/flutter (18545): date: Tue, 30 Oct 2018 10:00:52 GMT
I/flutter (18545): content-encoding: gzip
I/flutter (18545): vary: Accept-Encoding
I/flutter (18545): strict-transport-security: max-age=172800
I/flutter (18545): content-type: text/html; charset=utf-8
I/flutter (18545): tracecode: 00525262401065761290103018, 00522983

```

HttpClient配置

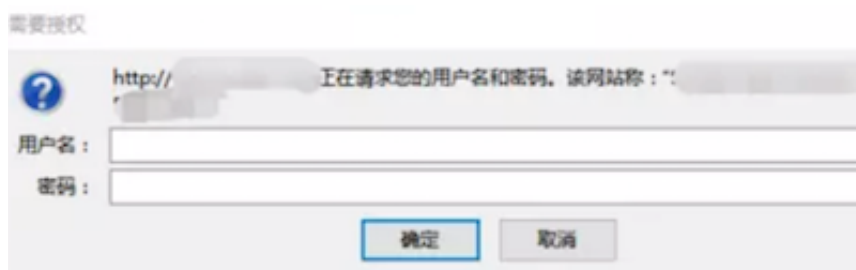
HttpClient有很多属性可以配置，常用的属性列表如下：

属性	含义
idleTime out	对应请求头中的keep-alive字段值，为了避免频繁建立连接，http Client在请求结束后会保持连接一段时间，超过这个阈值后才会关闭连接。
connecti onTimeo ut	和服务器建立连接的超时，如果超过这个值则会抛出SocketExce ption异常。
maxCon nections PerHost	同一个host，同时允许建立连接的最大数量。
autoUnc ompress	对应请求头中的Content-Encoding，如果设置为true，则请求头中Content-Encoding的值为当前HttpClient支持的压缩算法列表，目前只有"gzip"
userAge nt	对应请求头中的User-Agent字段。

可以发现，有些属性只是为了更方便的设置请求头，对于这些属性，你完全可以通过HttpRequest直接设置header，不同的是通过HttpClient设置的对整个httpClient都生效，而通过HttpRequest设置的只对当前请求生效。

HTTP请求认证

Http协议的认证（Authentication）机制可以用于保护非公开资源。如果Http服务器开启了认证，那么用户在发起请求时就需要携带用户凭据，如果你在浏览器中访问了启用Basic认证的资源时，浏览就会弹出一个登录框，如：



我们先看看Basic认证的基本过程：

1. 客户端发送http请求给服务器，服务器验证该用户是否已经登录验证过了，如果没有的话，服务器会返回一个401 Unauthorized给客户端，并且在响应header中添加一个“WWW-Authenticate”字段，例如：

```
WWW-Authenticate: Basic realm="admin"
```

其中"Basic"为认证方式，realm为用户角色的分组，可以在后台添加分组。

2. 客户端得到响应码后，将用户名和密码进行base64编码（格式为用户名:密码），设置请求头Authorization，继续访问：

```
Authorization: Basic YXXFISDJFISJFGIJIJG
```

服务器验证用户凭据，如果通过就返回资源内容。

注意，Http的方式除了Basic认证之外还有：Digest认证、Client认证、Form Based认证等，目前Flutter的HttpClient只支持Basic和Digest两种认证方式，这两种认证方式最大的区别是发送用户凭据时，对于用户凭据的内容，前者只是简单的通过Base64编码（可逆），而后者会进行哈希运算，相对来说安全一点点，但是为了安全起见，无论是采用Basic认证还是Digest认证，都应该在Https协议下，这样可以防止抓包和中间人攻击。

HttpClient关于Http认证的方法和属性：

1. `addCredentials(Uri url, String realm, HttpClientCredentials credentials)`

该方法用于添加用户凭据,如:

```
httpClient.addCredentials(_uri,
    "admin",
    new HttpClientBasicCredentials("username","password"), //Basic
    认证凭据
);
```

如果是Digest认证, 可以创建Digest认证凭据:

```
HttpClientDigestCredentials("username","password")
```

2. `authenticate(Future<bool> f(Uri url, String scheme, String realm))`

这是一个setter, 类型是一个回调, 当服务器需要用户凭据且该用户凭据未被添加时, httpClient会调用此回调, 在这个回调当中, 一般会调用 `addCredential()` 来动态添加用户凭证, 例如:

```
httpClient.authenticate=(Uri url, String scheme, String realm)
async{
    if(url.host=="xx.com" && realm=="admin"){
        httpClient.addCredentials(url,
            "admin",
            new HttpClientBasicCredentials("username","pwd"),
        );
        return true;
    }
    return false;
};
```

一个建议是, 如果所有请求都需要认证, 那么应该在HttpClient初始化时就调用 `addCredentials()` 来添加全局凭证, 而不是去动态添加。

代理

可以通过 `findProxy` 来设置代理策略，例如，我们要将所有请求通过代理服务器（192.168.1.2:8888）发送出去：

```
client.findProxy = (uri) {  
    // 如果需要过滤uri，可以手动判断  
    return "PROXY 192.168.1.2:8888";  
};
```

`findProxy` 回调返回值是一个遵循浏览器PAC脚本格式的字符串，详情可以查看API文档，如果不需要代理，返回"DIRECT"即可。

在APP开发中，很多时候我们需要抓包来调试，而抓包软件(如charles)就是一个代理，这时我们就可以将请求发送到我们的抓包软件，我们就可以在抓包软件中看到请求的数据了。

有时代理服务器也启用了身份验证，这和http协议的认证是相似的，HttpClient提供了对应的Proxy认证方法和属性：

```
set authenticateProxy(  
    Future<bool> f(String host, int port, String scheme, String  
    realm));  
void addProxyCredentials(  
    String host, int port, String realm, HttpClientCredentials  
    credentials);
```

他们的使用方法和上面“HTTP请求认证”一节中介绍的 `addCredentials` 和 `authenticate` 相同，故不再赘述。

证书校验

Https中为了防止通过伪造证书而发起的中间人攻击，客户端应该对自签名或非CA颁发的证书进行校验。HttpClient对证书校验的逻辑如下：

1. 如果请求的Https证书是可信CA颁发的，并且访问host包含在证书的domain列表中(或者符合通配规则)并且证书未过期，则验证通过。
2. 如果第一步验证失败，但在创建HttpClient时，已经通过SecurityContext将证书添加到证书信任链中，那么当服务器返回的证书在信任链中的话，则验证通过。
3. 如果1、2验证都失败了，如果用户提供了 `badCertificateCallback` 回调，则会调用它，如果回调返回 `true`，则允许继续链接，如果返回 `false`，则终止链接。

综上所述，我们的证书校验其实就是提供一个 `badCertificateCallback` 回调，下面通过一个示例来说明。

示例

假设我们的后台服务使用的是自签名证书，证书格式是PEM格式，我们将证书的内容保存在本地字符串中，那么我们的校验逻辑如下：

```
String PEM="XXXXXX";// 可以从文件读取
...
httpClient.badCertificateCallback=(X509Certificate cert, String
host, int port){
    if(cert.pem==PEM){
        return true; // 证书一致，则允许发送数据
    }
    return false;
};
```

`X509Certificate` 是证书的标准格式，包含了证书除私钥外所有信息，读者可以自行查阅文档。另外，上面的示例没有校验host，是因为只要服务器返回的证书内容和本地的保存一致就已经能证明是我们的服务器了（而不是中间人），host验证通常是为了防止证书和域名不匹配。

对于自签名的证书，我们也可以将其添加到本地证书信任链中，这样证书验证时就会自动通过，而不会再走到 `badCertificateCallback` 回调中：

```
SecurityContext sc=new SecurityContext();  
//file为证书路径  
sc.setTrustedCertificates(file);  
//创建一个HttpClient  
HttpClient httpClient = new HttpClient(context: sc);
```

注意，通过 `setTrustedCertificates()` 设置的证书格式必须为PEM或PKCS12，如果证书格式为PKCS12，则需将证书密码传入，这样则会在代码中暴露证书密码，所以客户端证书校验不建议使用PKCS12格式的证书。

总结

值得注意的是，HttpClient提供的这些属性和方法最终都会作用在请求header里，我们完全可以通过手动去设置header来实现，之所以提供这些方法，只是为了方便开发者而已。另外，Http协议是一个非常重要的、使用最多的网络协议，每一个开发者都应该对http协议非常熟悉。

网络操作

Dio http库

通过上一节介绍，我们可以发现直接使用HttpClient发起网络请求是比较麻烦的，很多事情得我们手动处理，如果再涉及到文件上传/下载、Cookie管理等就会非常繁琐。幸运的是，Dart社区有一些第三方http请求库，用它们来发起http请求将会简单的多，本节我们介绍一下目前人气较高的dio库。

dio是一个强大的Dart Http请求库，支持Restful API、FormData、拦截器、请求取消、Cookie管理、文件上传/下载、超时等。

引入

引入dio:

```
dependencies:  
  dio: ^x.x.x #请使用pub上的最新版本
```

导入并创建dio实例：

```
import 'package:dio/dio.dart';  
Dio dio = new Dio();
```

接下来就可以通过 dio实例来发起网络请求了，注意，一个dio实例可以发起多个http请求，一般来说，APP只有一个http数据源时，dio应该使用单例模式。

示例

发起 GET 请求：

```
Response response;  
response=await dio.get("/test?id=12&name=wendu")  
print(response.data.toString());
```

对于 GET 请求我们可以将query参数通过对象来传递，上面的代码等同于：

```
response=await dio.get("/test",data:{"id":12,"name":"wendu"})  
print(response.data.toString());
```

发起一个 POST 请求：

```
response=await dio.post("/test",data:{"id":12,"name":"wendu"})
```

发起多个并发请求：

```
response= await Future.wait([dio.post("/info"),dio.get("/token")]);
```

下载文件：

```
response=await dio.download("https://www.google.com/",_savePath);
```

发送 FormData：

```
FormData formData = new FormData.from({  
    "name": "wendux",  
    "age": 25,  
});  
response = await dio.post("/info", data: formData)
```

如果发送的数据是FormData，则dio会将请求header的 `contentType` 设为“multipart/form-data”。

通过FormData上传多个文件:

```
FormData formData = new FormData.from({  
    "name": "wendux",  
    "age": 25,  
    "file1": new UploadFileInfo(new File("./upload.txt"),  
        "upload1.txt"),  
    "file2": new UploadFileInfo(new File("./upload.txt"),  
        "upload2.txt"),  
    // 支持文件数组上传  
    "files": [  
        new UploadFileInfo(new File("./example/upload.txt"),  
            "upload.txt"),  
        new UploadFileInfo(new File("./example/upload.txt"),  
            "upload.txt")  
    ]  
});  
response = await dio.post("/info", data: formData)
```

值得一提的是，dio内部仍然使用HttpClient发起的请求，所以代理、请求认证、证书校验等和HttpClient是相同的，我们可以在 `onHttpClientCreate` 回调中设置，例如：


```
dio.onHttpClientCreate = (HttpClient client) {  
  // 设置代理  
  client.findProxy = (uri) {  
    return "PROXY 192.168.1.2:8888";  
  };  
  // 校验证书  
  httpClient.badCertificateCallback=(X509Certificate cert, String  
host, int port){  
    if(cert.pem==PEM){  
      return true; // 证书一致, 则允许发送数据  
    }  
    return false;  
  };  
};
```

注意, `onHttpClientCreate` 会在当前dio实例内部需要创建HttpClient时调用, 所以通过此回调配置HttpClient会对整个dio实例生效, 如果你想针对某个应用请求单独的代理或证书校验策略, 可以创建一个新的dio实例即可。

怎么样, 是不是很简单, 除了这些基本的用法, dio还支持请求配置、拦截器等, 官方资料比较详细, 故本书不再赘述, 详情可以参考dio主页:

<https://github.com/flutterchina/dio>。

使用WebSockets

Http协议是无状态的, 只能由客户端主动发起, 服务端再被动响应, 服务端无法向客户端主动推送内容, 并且一旦服务器响应结束, 链接就会断开(见注解部分), 所以无法进行实时通信。WebSocket协议正是为解决客户端与服务端实时通信而产生的技术, 现在已经被主流浏览器支持, 所以对于Web开发者来说应该比较熟悉了, Flutter也提供了专门的包来支持WebSocket协议。

注意: Http协议中虽然可以通过keep-alive机制使服务器在响应结束后链接会保持一段时间, 但最终还是会断开, keep-alive机制主要是用于避免在同一台服务器请求多个资源时频繁创建链接, 它本质上是支持链接复用的技术, 而非用于实时通信, 读者需要知道这两者的区别。

WebSocket协议本质上是一个基于tcp的协议，它是先通过HTTP协议发起一条特殊的http请求进行握手后，如果服务端支持WebSocket协议，则会进行协议升级。WebSocket会使用http协议握手后创建的tcp链接，和http协议不同的是，WebSocket的tcp链接是个长链接（不会断开），所以服务端与客户端就可以通过此TCP连接进行实时通信。有关WebSocket协议细节，读者可以看RFC文档，下面我们重点看看Flutter中如何使用WebSocket。

在接下来例子中，我们将连接到由[websocket.org](https://www.websocket.org)提供的测试服务器。服务器将简单地返回我们发送给它的相同消息！

步骤

1. 连接到WebSocket服务器。
2. 监听来自服务器的消息。
3. 将数据发送到服务器。
4. 关闭WebSocket连接。

1. 连接到WebSocket服务器

[web_socket_channel](#) package 提供了我们需要连接到WebSocket服务器的工具。

该package提供了一个 `WebSocketChannel` 允许我们既可以监听来自服务器的消息，又可以将消息发送到服务器的方法。

在Flutter中，我们可以创建一个 `WebSocketChannel` 连接到一台服务器：

```
final channel = new
IOWebSocketChannel.connect('ws://echo.websocket.org');
```

2. 监听来自服务器的消息

现在我们建立了连接，我们可以监听来自服务器的消息，在我们发送消息给测试服务器之后，它会返回相同的消息。

我们如何收取消息并显示它们？在这个例子中，我们将使用一个 `StreamBuilder` Widget来监听新消息，并用一个Text Widget来显示它们。

```
new StreamBuilder(  
  stream: widget.channel.stream,  
  builder: (context, snapshot) {  
    return new Text(snapshot.hasData ? '${snapshot.data}' : '');  
  },  
);
```

工作原理

`WebSocketChannel` 提供了一个来自服务器的消息Stream。

该 `Stream` 类是 `dart:async` 包中的一个基础类。它提供了一种方法来监听来自数据源的异步事件。与 `Future` 返回单个异步响应不同，`Stream` 类可以随着时间推移传递很多事件。

该 `StreamBuilder` Widget将连接到一个Stream，并在每次收到消息时通知Flutter重新构建界面。

3. 将数据发送到服务器

为了将数据发送到服务器，我们会 `add` 消息给 `WebSocketChannel` 提供的sink。

```
channel.sink.add('Hello!');
```

工作原理

`WebSocketChannel` 提供了一个 `StreamSink`，它将消息发给服务器。

`StreamSink` 类提供了给数据源同步或异步添加事件的一般方法。

4. 关闭WebSocket连接

在我们使用 `WebSocket` 后，要关闭连接：

```
channel.sink.close();
```

完整的例子

```
import 'package:flutter/material.dart';
import 'package:web_socket_channel/io.dart';

class WebSocketRoute extends StatefulWidget {
  @override
  _WebSocketRouteState createState() => new _WebSocketRouteState();
}

class _WebSocketRouteState extends State<WebSocketRoute> {
  TextEditingController _controller = new TextEditingController();
  IOWebSocketChannel channel;
  String _text = "";

  @override
  void initState() {
    // 创建websocket连接
    channel = new
IOWebSocketChannel.connect('ws://echo.websocket.org');
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("WebSocket(内容回显)"),
      ),
      body: new Padding(
        padding: const EdgeInsets.all(20.0),
        child: new Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            new Form(
              child: new TextFormField(
                controller: _controller,
                decoration: new InputDecoration(labelText: 'Send a
message'),
            ),
            new StreamBuilder(
              stream: channel.stream,
              builder: (context, snapshot) {
                // 网络不通会走到这
                if (snapshot.hasError) {
```

```

        _text = "网络不通...";
    } else if (snapshot.hasData) {
        _text = "echo: "+snapshot.data;
    }
    return new Padding(
        padding: const EdgeInsets.symmetric(vertical:
24.0),
        child: new Text(_text),
    );
    },
    ),
    ),
    floatingActionButton: new FloatingActionButton(
        onPressed: _sendMessage,
        tooltip: 'Send message',
        child: new Icon(Icons.send),
    ),
);
}

void _sendMessage() {
    if (_controller.text.isNotEmpty) {
        channel.sink.add(_controller.text);
    }
}

@override
void dispose() {
    channel.sink.close();
    super.dispose();
}
}

```

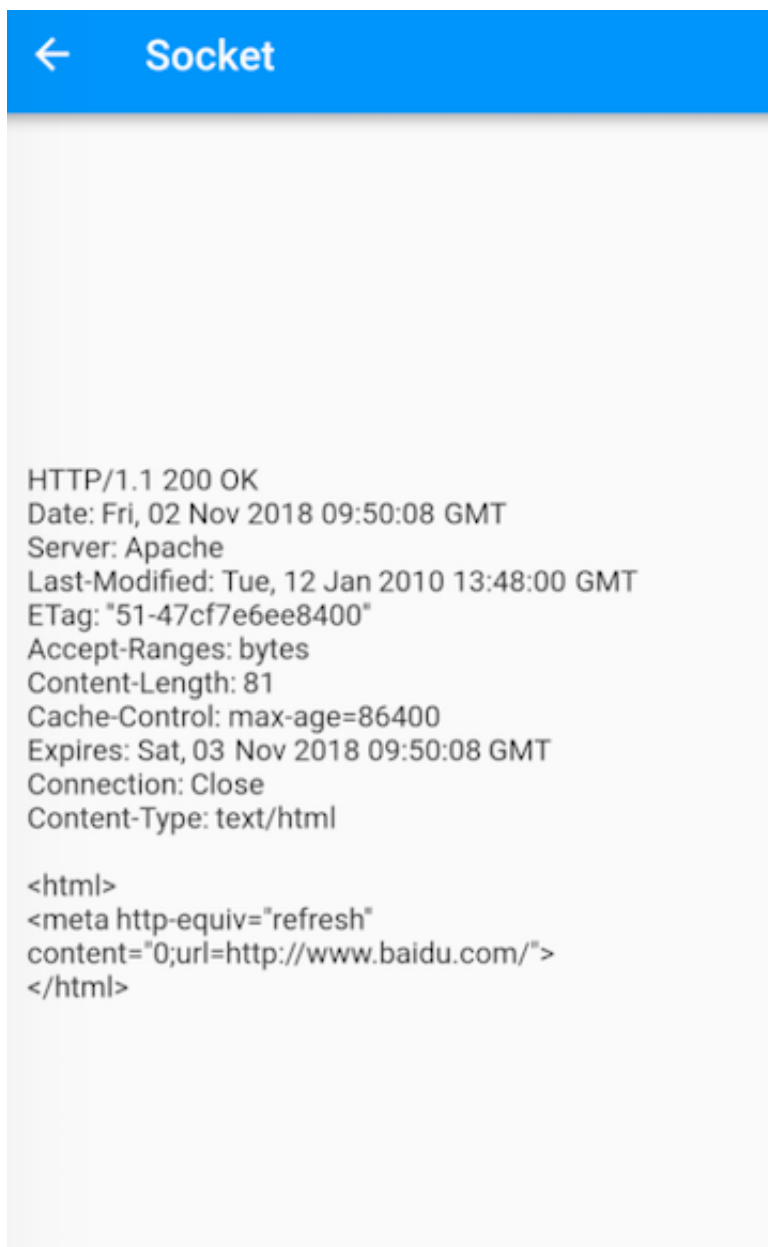
上面的例子比较简单，不再赘述。我们现在思考一个问题，假如我们想通过WebSocket传输二进制数据应该怎么做（比如要从服务器接收一张图片）？我们发现 `StreamBuilder` 和 `Stream` 都没有指定接收类型的参数，并且在创建WebSocket链接时也没有相应的配置，貌似没有什么办法.....其实很简单，要接收二进制数据仍然使用 `StreamBuilder`，因为WebSocket中所有发送的数据使用帧的形式发送，而帧是有固定格式，每一个帧的数据类型都可以通过Opcode字段指定，它可以指定当前帧是文本类型还是二进制类型（还有其它类型），所以客户端在收到帧时就已经知道了其数据类型，所以flutter完全可以在收到数据后解析出正确的类型，所以就无需开发者去关心，当服务器传输的数据是指定为二进制时，`StreamBuilder` 的 `snapshot.data` 的类型就是 `List<int>`，是文本时，则为 `String`。

Socket

我们之前介绍的Http协议和WebSocket协议都属于应用层协议，除了它们，应用层协议还有很多如：SMTP、FTP等，它们都是通过Socket实现的。其实，操作系统中提供的原生网络请求API是标准的，在C语言的Socket库中，它主要提供了端到端建立链接和发送数据的基础API，而高级编程语言中的Socket库其实都是对操作系统的socket API的一个封装。所以，如果我们需要自定义协议或者想直接来控制管理网络链接、又或者我们觉得自带的HttpClient不好用想重新实现一个，这时我们就需要使用Socket。Flutter的Socket API在dart io包中，下面我们看一个使用Socket实现简单http请求的示例，以请求百度首页为例：

```
_request() async{
  // 建立连接
  var socket=await Socket.connect("baidu.com", 80);
  // 根据http协议，发送请求头
  socket.writeln("GET / HTTP/1.1");
  socket.writeln("Host:baidu.com");
  socket.writeln("Connection:close");
  socket.writeln();
  await socket.flush(); // 发送
  // 读取返回内容
  _response =await socket.transform(utf8.decoder).join();
  await socket.close();
}
```

可以看到，使用Socket需要我们自己实现Http协议细节，本例只是一个简单示例，没有处理重定向、cookie等。本示例完整代码参考示例demo，运行后如下：



可以看到响应内容分两个部分，第一部分是响应头，第二部分是响应体，服务端可以根据请求信息动态来输出响应体。由于本示例请求头比较简单，所以响应体和浏览器中访问的会有差别，读者可以补充一些请求头(如user-agent)来看看输出的变化。

Json Model

在实战中，后台接口往往会返回一些结构化数据，如JSON、XML等，如之前我们请求Github API的示例，它返回的数据就是JSON格式的字符串，为了方便我们在代码中操作JSON，我们先将JSON格式的字符串转为Dart对象，这个可以通过 `dart:convert` 中内置的JSON解码器`json.decode()` 来实现，该方法可以根据JSON字符串具体内容将其转为List或Map，这样我们就可以通过他们来查找所需的值，如：

```
// 一个JSON格式的用户列表字符串
String jsonStr='[{"name":"Jack"}, {"name":"Rose"}]';
// 将JSON字符串转为Dart对象(此处是List)
List items=json.decode(jsonStr);
// 输出第一个用户的姓名
print(items[0]["name"]);
```

通过`json.decode()` 将JSON字符串转为List/Map的方法比较简单，它没有外部依赖或其它的设置，对于小项目很方便。但当项目变大时，这种手动编写序列化逻辑可能变得难以管理且容易出错，例如有如下JSON:

```
{
  "name": "John Smith",
  "email": "john@example.com"
}
```

我们可以通过调用 `json.decode` 方法来解码JSON，使用JSON字符串作为参数:

```
Map<String, dynamic> user = json.decode(json);

print('Howdy, ${user['name']}!');
print('We sent the verification link to ${user['email']}.');
```

由于 `json.decode()` 仅返回一个 `Map<String, dynamic>`，这意味着直到运行时我们才知道值的类型。通过这种方法，我们失去了大部分静态类型语言特性：类型安全、自动补全和最重要的编译时异常。这样一来，我们的代码可能会变得非常容易出错。例如，当我们访问 `name` 或 `email` 字段时，我们输入的很快，导致字段名打错了。但由于这个JSON在map结构中，所以编译器不知道这个错误的字段名，所以编译时不会报错。

其实，这个问题在很多平台上都会遇到，而也早就有了好的解决方法即“Json Model化”，具体做法就是，通过预定义一些与Json结构对应的Model类，然后在请求到数据后再动态根据数据创建出Model类的实例。这样一来，在开发阶段我们使用的是Model类的实例，而不再是Map/List，这样访问内部属性时就不会发生拼写错误。例如，我们可以通过引入一个简单的模型类(Model class)来解决前面提到的问题，我们称之为 `User` 。在User类内部，我们有：

- 一个 `User.fromJson` 构造函数, 用于从一个map构造出一个 `User` 实例 map structure
- 一个 `toJson` 方法, 将 `User` 实例转化为一个map.

这样，调用代码现在可以具有类型安全、自动补全字段（name和email）以及编译时异常。如果我们将拼写错误字段视为 `int` 类型而不是 `String` ，那么我们的代码就不会通过编译，而不是在运行时崩溃。

user.dart

```
class User {
  final String name;
  final String email;

  User(this.name, this.email);

  User.fromJson(Map<String, dynamic> json)
    : name = json['name'],
      email = json['email'];

  Map<String, dynamic> toJson() =>
    <String, dynamic>{
      'name': name,
      'email': email,
    };
}
```

现在，序列化逻辑移到了模型本身内部。采用这种新方法，我们可以非常容易地反序列化user.

```
Map userMap = json.decode(json);
var user = new User.fromJson(userMap);

print('Howdy, ${user.name}!');
print('We sent the verification link to ${user.email}.');
```

要序列化一个user，我们只是将该 User 对象传递给该 json.encode 方法。我们不需要手动调用 toJson 这个方法，因为`JSON.encode`内部会自动调用。

```
String json = json.encode(user);
```

这样，调用代码就不用担心JSON序列化了，但是，Model类还是必须的。在实践中，User.fromJson 和 User.toJson 方法都需要单元测试到位，以验证正确的行为。

另外，实际场景中，JSON对象很少会这么简单，嵌套的JSON对象并不罕见，如果有什么能为我们自动处理JSON序列化，那将会非常好。幸运的是，有！

自动生成Model

尽管还有其他库可用，但在本书中，我们介绍一下官方推荐的[json_serializable package](#)包。它是一个自动化的源代码生成器，可以在开发阶段为我们生成JSON序列化模板，这样一来，由于序列化代码不再由我们手写和维护，我们将运行时产生JSON序列化异常的风险降至最低。

在项目中设置json_serializable

要包含 json_serializable 到我们的项目中，我们需要一个常规和两个开发依赖项。简而言之，开发依赖项是不包含在我们的应用程序源代码中的依赖项，它是开发过程中的一些辅助工具、脚本，和node中的开发依赖项相似。

pubspec.yaml

```
dependencies:
  # Your other regular dependencies here
  json_annotation: ^2.0.0

dev_dependencies:
  # Your other dev_dependencies here
  build_runner: ^1.0.0
  json_serializable: ^2.0.0
```

在您的项目根文件夹中运行 `flutter packages get`（或者在编辑器中点击“Packages Get”）以在项目中使用这些新的依赖项。

以json_serializable的方式创建model类

让我们看看如何将我们的 `User` 类转换为一个 `json_serializable`。为了简单起见，我们使用前面示例中的简化JSON model。

user.dart

```
import 'package:json_annotation/json_annotation.dart';

// user.g.dart 将在我们运行生成命令后自动生成
part 'user.g.dart';

/// 这个标注是告诉生成器，这个类是需要生成Model类的
@JsonSerializable()

class User{
  User(this.name, this.email);

  String name;
  String email;
  // 不同的类使用不同的mixin即可
  factory User.fromJson(Map<String, dynamic> json) =>
    _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

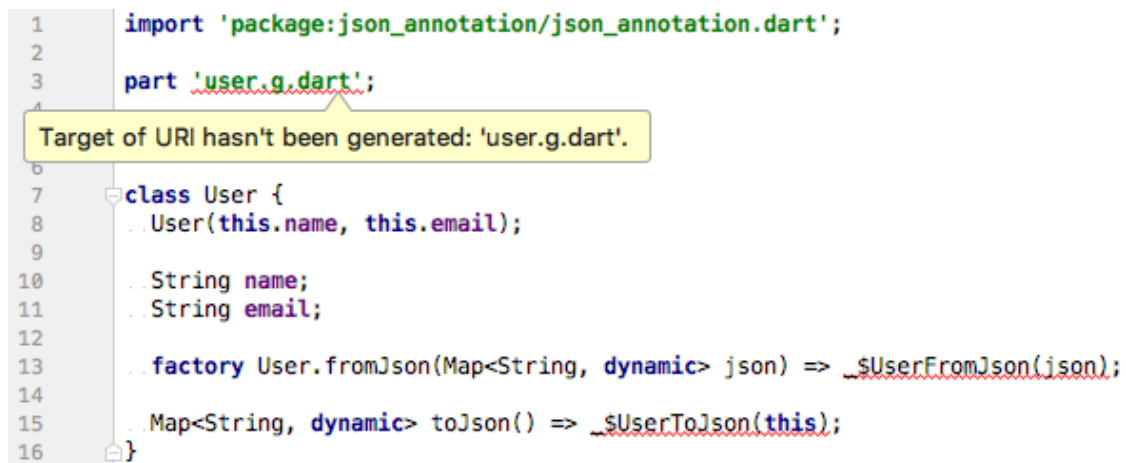
有了上面的设置，源码生成器将生成用于序列化 `name` 和 `email` 字段的JSON代码。

如果需要，自定义命名策略也很容易。例如，如果我们正在使用的API返回带有 `_snake_case_` 的对象，但我们想在我们的模型中使用 `_lowerCamelCase_`，那么我们可以使用 `@JsonKey` 标注：

```
// 显式关联JSON字段名与Model属性的对应关系
@JsonKey(name: 'registration_date_millis')
final int registrationDateMillis;
```

运行代码生成程序

`json_serializable` 第一次创建类时，您会看到与下图类似的错误。



```
1 import 'package:json_annotation/json_annotation.dart';
2
3 part 'user.g.dart';
4
5
6
7 class User {
8   ..User(this.name, this.email);
9
10  ..String name;
11  ..String email;
12
13  ..factory User.fromJson(Map<String, dynamic> json) => .._UserFromJson(json);
14
15  ..Map<String, dynamic> toJson() => .._UserToJson(this);
16 }
```

这些错误是完全正常的，这是因为Model类的生成代码还不存在。为了解决这个问题，我们必须运行代码生成器来为我们生成序列化模板。有两种运行代码生成器的方法：

一次性生成

通过在我们的项目根目录下运行：

```
flutter packages pub run build_runner build
```

这触发了一次性构建，我们可以在需要时为我们的Model生成json序列化代码，它通过我们的源文件，找出需要生成Model类的源文件（包含 `@JsonSerializable` 标注的）来生成对应的.g.dart文件。一个好的建议是将所有Model类放在一个单独的目录下，然后在该目录下执行命令。

虽然这非常方便，但如果我们不需要每次在Model类中进行更改时都要手动运行构建命令的话会更好。

持续生成

使用`_watcher_`可以使我们的源代码生成的过程更加方便。它会监视我们项目中文件的变化，并在需要时自动构建必要的文件，我们可以通过 `flutter packages pub run build_runner watch` 在项目根目录下运行来启动`_watcher_`。只需启动一次观察器，然后它就会在后台运行，这是安全的。

自动化生成模板

上面的方法有一个最大的问题就是要为每一个json写模板，这是比较枯燥的。如果有一个工具可以直接根据JSON文本生成模板，那我们就能彻底解放双手了。笔者自己用dart实现了一个脚本，它可以自动生成模板，并直接将JSON转为Model类，下面我们看看怎么做：

1. 定义一个"模板的模板"，名为"template.dart"：

```
import 'package:json_annotation/json_annotation.dart';
%t
part '%s.g.dart';
@JsonSerializable()
class %s {
  %s();

  %s
  factory %s.fromJson(Map<String,dynamic> json) =>
    _$sFromJson(json);
  Map<String, dynamic> toJson() => _$sToJson(this);
}
```

模板中的“%t”、“%s”为占位符，将在脚本运行时动态被替换为合适的导入头和类名。

2. 写一个自动生成模板的脚本(mo.dart)，它可以根据指定的JSON目录，遍历生成模板，在生成时我们定义一些规则：
 - 如果JSON文件名以下划线“_”开始，则忽略此JSON文件。
 - 复杂的JSON对象往往会出现嵌套，我们可以通过一个特殊标志来手动指定嵌套的对象（后面举例）。

脚本我们通过Dart来写，源码如下：

```
import 'dart:convert';
import 'dart:io';
import 'package:path/path.dart' as path;
const TAG="\$";
const SRC="./json"; //JSON 目录
const DIST="lib/models/"; //输出model目录

void walk() { //遍历JSON目录生成模板
  var src = new Directory(SRC);
  var list = src.listSync();
  var template=new File("./template.dart").readAsStringSync();
  File file;
  list.forEach((f) {
    if (FileSystemEntity.isFileSync(f.path)) {
      file = new File(f.path);
      var paths=path.basename(f.path).split(".");
      String name=paths.first;
      if(paths.last.toLowerCase()!="json"||name.startsWith("_"))
return ;
      if(name.startsWith("_")) return;
      //下面生成模板
      var map = json.decode(file.readAsStringSync());
      //为了避免重复导入相同的包，我们用Set来保存生成的import语句。
      var set= new Set<String>();
      StringBuffer attrs= new StringBuffer();
      (map as Map<String, dynamic>).forEach((key, v) {
        if(key.startsWith("_")) return ;
        attrs.write(getType(v,set,name));
        attrs.write(" ");
        attrs.write(key);
        attrs.writeln(";");
        attrs.write("    ");
      });
      String className=name[0].toUpperCase()+name.substring(1);
      var dist=format(template,
[name,className,className,attrs.toString(),
                                className,className,className]);
      var _import=set.join(";\\r\\n");
      _import+=_import.isEmpty?"":";";
      dist=dist.replaceFirst("%t",_import );
      //将生成的模板输出
      new File("$DIST$name.dart").writeAsStringSync(dist);
    }
  });
}
```

```
String changeFirstChar(String str, [bool upper=true] ){
    return (upper?
str[0].toUpperCase():str[0].toLowerCase())+str.substring(1);
}
```

// 将JSON类型转为对应的dart类型

```
String getType(v,Set<String> set,String current){
    current=current.toLowerCase();
    if(v is bool){
        return "bool";
    }else if(v is num){
        return "num";
    }else if(v is Map){
        return "Map<String,dynamic>";
    }else if(v is List){
        return "List";
    }else if(v is String){ // 处理特殊标志
        if(v.startsWith("$TAG[]")){
            var className=changeFirstChar(v.substring(3),false);
            if(className.toLowerCase()!=current) {
                set.add('import "$className.dart"');
            }
            return "List<${changeFirstChar(className)}>";
        }else if(v.startsWith(TAG)){
            var fileName=changeFirstChar(v.substring(1),false);
            if(fileName.toLowerCase()!=current) {
                set.add('import "$fileName.dart"');
            }
            return changeFirstChar(fileName);
        }
        return "String";
    }else{
        return "String";
    }
}
```

// 替换模板占位符

```
String format(String fmt, List<Object> params) {
    int matchIndex = 0;
    String replace(Match m) {
        if (matchIndex < params.length) {
            switch (m[0]) {
                case "%s":
                    return params[matchIndex++].toString();
            }
        } else {

```

```

        throw new Exception("Missing parameter for string
format");
    }
    throw new Exception("Invalid format string: " +
m[0].toString());
    }
    return fmt.replaceAllMapped("%s", replace);
}

void main(){
    walk();
}

```

3. 写一个shell([mo.sh](#)), 将生成模板和生成model串起来:

```

dart mo.dart
flutter packages pub run build_runner build --delete-
conflicting-outputs

```

至此，我们的脚本写好了，我们在根目录下新建一个json目录，然后把user.json移进去，然后在lib目录下创建一个models目录，用于保存最终生成的Model类。现在我们只需要一句命令即可生成Model类了：

```
./mo.sh
```

运行后，一切都将自动执行，现在好多了，不是吗？

嵌套JSON

我们定义一个person.json内容修改为：


```
{
  "name": "John Smith",
  "email": "john@example.com",
  "mother": {
    "name": "Alice",
    "email": "alice@example.com"
  },
  "friends": [
    {
      "name": "Jack",
      "email": "Jack@example.com"
    },
    {
      "name": "Nancy",
      "email": "Nancy@example.com"
    }
  ]
}
```

每个Person都有 name 、 email 、 mother 和 friends 四个字段，由于 mother 也是一个Person，朋友是多个Person(数组)，所以我们期望生成的Model是下面这样：

```
import 'package:json_annotation/json_annotation.dart';
part 'person.g.dart';

@JsonSerializable()
class Person {
  Person();

  String name;
  String email;
  Person mother;
  List<Person> friends;

  factory Person.fromJson(Map<String,dynamic> json) =>
    _$PersonFromJson(json);
  Map<String, dynamic> toJson() => _$PersonToJson(this);
}
```

这时，我们只需要简单修改一下JSON，添加一些特殊标志，重新运行mo.sh即可：

```
{
  "name": "John Smith",
  "email": "john@example.com",
  "mother": "$person",
  "friends": "$[]person"
}
```

我们使用美元符“\$”作为特殊标志符(如果与内容冲突，可以修改mo.dart中的 TAG 常量，自定义标志符)，脚本在遇到特殊标志符后会先把相应字段转为相应的对象或对象数组，对象数组需要在标志符后面添加数组符“[]”，符号后面接具体的类型名，此例中是person。其它类型同理，加入我们给User添加一个Person类型的boss 字段：

```
{
  "name": "John Smith",
  "email": "john@example.com",
  "boss": "$person"
}
```

重新运行mo.sh，生成的user.dart如下：

```
import 'package:json_annotation/json_annotation.dart';
import "person.dart";
part 'user.g.dart';

@JsonSerializable()

class User {
  User();

  String name;
  String email;
  Person boss;

  factory User.fromJson(Map<String,dynamic> json) =>
    _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

可以看到， boss 字段已自动添加，并自动导入了“person.dart”。

使用IDE插件生成model

目前Android Studio(或IntelliJ)有一个[插件](#)，它可以自动将Json转为model，该插件会对嵌套Json也会生成model。这个特性在有些时候可能会引起重定义，如两个Json都内嵌了一个user的对象时，会导致user model在不同的文件中会被定义两次，需要开发者手动去重。

FAQ

很多人可能会问Flutter中有没有像Java开发中的Gson/Jackson一样的Json序列化类库？答案是没有！因为这样的库需要使用运行时反射，这在Flutter中是禁用的。运行时反射会干扰Dart的_tree shaking_，使用_tree shaking_，可以在release版中“去除”未使用的代码，这可以显著优化应用程序的大小。由于反射会默认应用到所有代码，因此_tree shaking_会很难工作，因为在启用反射时很难知道哪些代码未被使用，因此冗余代码很难剥离，所以Flutter中禁用了Dart的反射功能，而正因如此也就无法实现动态转化Model的功能。

FutureBuilder

由于网络请求是异步的，所以在实战中，我们通常会在请求的过程中弹出一个加载框，等到请求结束后再来渲染最终页面，如果发生错误提示错误信息。在Flutter中我们虽然完全可以手动去做这些事，如发现请求状态发生变化时再调用 `setState()` 去重建UI，但由于这是一个固定的模式，Flutter提供了 `FutureBuilder` Widget专门来处理在异步任务的不同过程构建不同的UI元素。

示例

我们通过Github开放的API来请求flutterchina组织下的所有公开的开源项目，实现：

1. 在请求阶段弹出loading
2. 请求结束后，如果请求失败，则展示错误信息；如果成功，则将项目名称列表展示出来。

代码如下：

```

class _FutureBuilderRouteState extends State<FutureBuilderRoute> {
  Dio _dio = new Dio();

  @override
  Widget build(BuildContext context) {

    return new Container(
      alignment: Alignment.center,
      child: FutureBuilder(
        future:
        _dio.get("https://api.github.com/orgs/flutterchina/repos"),
        builder: (BuildContext context, AsyncSnapshot snapshot) {
          // 请求完成
          if (snapshot.connectionState == ConnectionState.done) {
            Response response = snapshot.data;
            // 发生错误
            if (snapshot.hasError) {
              return Text(snapshot.error.toString());
            }
            // 请求成功，通过项目信息构建用于显示项目名称的ListView
            return ListView(
              children: response.data.map<Widget>((e) =>
                ListTile(title: Text(e["full_name"]))
              ).toList(),
            );
          }
          // 请求未完成时弹出loading
          return CircularProgressIndicator();
        }
      ),
    );
  }
}

```

代码中的AsyncSnapshot代表一个异步任务快照，它的 `connectionState` 属性代表当前异步任务的状态，开发者可以通过判断该状态得到异步任务的进度。值得注意的是，AsyncSnapshot会将最近一次请求的结果缓存，如果您需要通过FutureBuilder发起多次异步任务时，一定要注意，比如下面的代码在多次请求时就不能正常工作：

```

FutureBuilder(
  future:
  _dio.get("https://api.github.com/orgs/flutterchina/repos"),
  builder: (BuildContext context, AsyncSnapshot snapshot) {
    // 请求成功, 返回项目列表
    if(snapshot.hasData){
      Response response = snapshot.data;
      return ListView(
        children: response.data.map<Widget>((e) =>
          ListTile(title: Text(e["full_name"]))
        ).toList(),
      );
    }else if(snapshot.hasError){ // 发生错误
      return Text(snapshot.error.toString());
    }
    // 请求未完成时弹出loading
    return CircularProgressIndicator();
  }
)

```

上面代码在首次请求时没有问题，但在第二次请求时loading就弹不出来了，原因是snapshot会缓存上次请求结果，所以在第二次请求的过程中，依然会显示第一次的请求结果，直到第二次请求结束后才会更新。其实要避免这个问题很容易，我们记住“状态优先”原则就行，状态改变时我们应该首先判断请求状态，然后在判断是否成功，就像上面的第一个示例那样。