`Restsharp` v107

Quick start | RestSharp

Introduction

前言

WARNING

警告

RestSharp v107 changes the library API surface and its behaviour significantly. We advise looking at v107 docs to understand how to migrate to the latest version of RestSharp.

RestSharp v107显著地改变了API库的表现及其行为。我们建议大家查看v107号文档以理解如何通过定期查看最新版本说明。

The main purpose of RestSharp is to make synchronous and asynchronous calls to remote resources over HTTP. As the name suggests, the main audience of RestSharp are developers who use REST APIs. However, RestSharp can call any API over HTTP, as long as you have the resource URI and request parameters that you want to send comply with W3C HTTP standards.

RestSharp重要目的是通过Http接口实现对远程资源的同步或异步调用。正如名字所示，RestSharp主要面向那些使用REST APIs的群体。

One of the main challenges of using HTTP APIs for .NET developers is to work with requests and responses of different kinds and translate them to complex C# types. RestSharp can take care of serializing the request body to JSON or XML and deserialize the response. It can also form a valid request URI based on different parameter kinds - path, query, form or body.

NET开发人员使用HTTP API的主要挑战之一是处理不同类型的请求和响应，并将其转换为复杂的C类型。RestSharp可以将请求体序列化为JSON或XML，并反序列化响应。它还可以基于不同的参数类型（路径、查询、表单或主体）形成有效的请求URI。

#Getting Started

开始

Before you can use RestSharp in your application, you need to add the NuGet package. You can do it using your IDE or the command line:

如果你的应用需要使用RestSharp，你可以添加NuGet包。你可以使用IDE添加或者在命令行输入以下命令：

```
dotnet add package RestSharp
```

#Basic Usage

基本用法

If you only have a few number of one-off requests to make to an API, you can use RestSharp like so:

如果只需要向API发送少量或一次性请求，你可以使用RestSharp，例如：

```
using RestSharp;
using RestSharp.Authenticators;

var client = new RestClient("https://api.twitter.com/1.1") {
    Authenticator = new HttpBasicAuthenticator("username", "password")
};
var request = new RestRequest("statuses/home_timeline.json");
var response = await client.GetAsync(request, cancellationToken);
```

It will return a `RestResponse` back, which contains all the information returned from the remote server. You have access to the headers, content, HTTP status and more.

它将返回一个“response”，其中包含从远程服务器返回的所有信息。您可以访问标题、内容、HTTP状态等。

We recommend using the generic overloads like `Get<T>` to automatically deserialize the response into .NET classes.

我们建议使用“Get”之类的泛型重载将响应自动反序列化为.NET类。

recommend：推荐，overloads： 重载， automatically [ˌɔːtəˈmætɪkli]： 自动地，deserialize： 反序列化

For example:

举个栗子🌰：

```
using RestSharp;
using RestSharp.Authenticators;

var client = new RestClient("https://api.twitter.com/1.1");
client.Authenticator = new HttpBasicAuthenticator("username", "password");

var request = new RestRequest("statuses/home_timeline.json", DataFormat.Json);

var timeline = await client.GetAsync<HomeTimeline>(request, cancellationToken);
```

The most important difference, however, that async methods that are named after HTTP methods return the `Task<T>` instead of `Task<IRestResponse<T>>`. Because it means that you won't get an error response if the request fails, those methods throw an exception.

然而，最重要的区别是，以HTTP方法命名的异步方法返回`Task<T>`而不是`Task<IRestResponse<T>`。因为这意味着如果请求失败，您将不会得到错误响应，所以这些方法会引发异常。

All `ExecuteAsync` overloads, however, behave in the same way as `Execute` and return the `IRestResponse` or `IRestResponse<T>`.

然而，所有`ExecuteAsync`重载的行为方式都与`Execute`相同，并返回`IRESResponse`或`IRESresponse<T>`。

Read here about how RestSharp handles exceptions.

阅读此处关于RestSharp如何处理异常。

#Content type

#内容类型

RestSharp supports sending XML or JSON body as part of the request. To add a body to the request, simply call `AddJsonBody` or `AddXmlBody` method of the `IRestRequest` instance.

RestSharp支持将XML或JSON正文作为请求的一部分发送。要将主体添加到请求中，只需调用`IRestRequest`实例的`AddJsonBody`或`AddXmlBody`方法。

There is no need to set the `Content-Type` or add the `DataFormat` parameter to the request when using those methods, RestSharp will do it for you.

使用这些方法时，无需设置`Content Type`或向请求中添加`DataFormat`参数，RestSharp将为您完成此操作。

RestSharp will also handle both XML and JSON responses and perform all necessary deserialization tasks, depending on the server response type. Therefore, you only need to add the `Accept` header if you want to deserialize the response manually.

RestSharp还将处理XML和JSON响应，并根据服务器响应类型执行所有必要的反序列化任务。因此，如果要手动反序列化响应，只需添加`Accept`标头。

For example, only you'd only need these lines to make a request with JSON body:

例如，您只需要这些行就可以使用JSON主体发出请求：

```
var request = new RestRequest("address/update").AddJsonBody(updatedAddress);
var response = await client.PostAsync<AddressUpdateResponse>(request);
```

Read more about serialization and deserialization here.

在此处阅读有关序列化和反序列化的更多信息。

#Response

#响应

When you use `ExecuteAsync`, you get an instance of `RestResponse` back that has the `Content` property, which contains the response as string. You can find other useful properties there, like `StatusCode`, `ContentType` and so on. If the request wasn't successful, you'd get a response back with `IsSuccessful` property set to `false` and the error explained in the `ErrorException` and `ErrorMessage` properties.

当您使用`ExecuteAsync`时，您会得到一个具有`Content`属性的`RestResponse`实例，该实例包含了string类型的响应。您可以在那里找到其他有用的属性，如`StatusCode`、`ContentType`等。如果请求未成功，您将返回一个响应，其中`IsSuccessful`属性设置为`false`，错误在`ErrorException`和`ErrorMessage`属性中解释。

When using typed `ExecuteAsync<T>`, you get an instance of `RestResponse<T>` back, which is identical to `RestResponse` but also contains the `T Data` property with the deserialized response.

当使用类型化的`ExecuteAsync<T>`时，返回一个`RestResponse<T>`，它与`RestResponse`相同但也包含反序列化响应的`T Data`属性。

None of `ExecuteAsync` overloads throw if the remote server returns an error. You can inspect the response and find the status code, error message, and, potentially, an exception.

如果远程服务器返回错误，`ExecuteAsync`重载都不会抛出。您可以检查响应并找到状态代码、错误消息以及可能的异常。

Extensions like `GetAsync<T>` will not return the whole `RestResponse<T>` but just a deserialized response. These extensions will throw an exception if the remote server returns an error. The exception will tell you what status code was returned by the server.

像`GetAsync<T>`这样的扩展不会返回整个`restreponse<T>`，而只是一个反序列化的响应。如果远程服务器返回错误，这些扩展将引发异常。异常将告诉您服务器返回了什么状态代码。

Last Updated: 2022/7/21 17:09:28

最后更新：2022/7/21 17:09:28

Contributors: Marcel Juen

贡献者：Marcel Juen

Usage

使用说明

Recommended usage

推荐使用

RestSharp works best as the foundation for a proxy class for your API. Each API would most probably require different settings for `RestClient`. Hence, a dedicated API class (and its interface) gives you sound isolation between different `RestClient` instances and make them testable.

RestSharp最适合作为API代理类的基础。每个API很可能需要不同的`RestClient`设置。因此，一个专用的API类（及其接口）可以在不同的`RestClient`实例之间提供良好的隔离，并是他们具备可测试性。

Essentially, RestSharp is a wrapper around `HttpClient` that allows you to do the following:

本质上，RestSharp是`HttpClient`的包装器，允许您执行以下操作：

- Add default parameters of any kind (not just headers) to the client, once
- 将任何类型的默认参数（不只是头参数）添加到客户端一次
- Add parameters of any kind to each request (query, URL segment, form, attachment, serialized body, header) in a straightforward way
- 以简单的方式向每个请求添加任何类型的参数（查询、URL段、表单、附件、序列化正文、标头）
- Serialize the payload to JSON or XML if necessary
- 在必要情况下，将负载序列化为JSON或XML
- Set the correct content headers (content type, disposition, length, etc.)
- 设置正确的内容标题（内容类型、配置、长度等）
- Handle the remote endpoint response
- 处理远程端点响应
- Deserialize the response from JSON or XML if necessary

- 如果需要，从JSON或XML反序列化响应

For example, let's look at a simple Twitter API v2 client, which uses OAuth2 machine-to-machine authentication. For it to work, you would need to have access to the Twitter Developers portal, a project, and an approved application inside the project with OAuth2 enabled.

例如，让我们看看一个简单的TwitterAPI v2客户端，它使用OAuth2机器对机器身份验证。要使其工作，您需要访问Twitter开发者门户、一个项目和项目内部的一个已批准的应用程序，并启用OAuth2。

#Authenticator

#验证程序

Before we can call the API itself, we need to get a bearer token. Twitter exposes an endpoint `https://api.twitter.com/oauth2/token`. As it follows the OAuth2 conventions, the code can be used to create an authenticator for some other vendors.

在调用API本身之前，我们需要获得一个载体。Twitter公开了一个端点`https://api.twitter.com/oauth2/token`.由于遵循OAuth2约定，该代码可用于为其他一些供应商创建验证器。

First, we need a model for deserializing the token endpoint response. OAuth2 uses snake case for property naming, so we need to decorate model properties with `JsonPropertyName` attribute:

首先，我们需要一个反序列化令牌端点响应的模型。OAuth2使用snake case进行属性命名，因此我们需要使用`JsonPropertyName`属性装饰模型属性：

```
record TokenResponse {
    [JsonPropertyName("token_type")]
    public string TokenType { get; init; }
    [JsonPropertyName("access_token")]
    public string AccessToken { get; init; }
}
```

Next, we create the authenticator itself. It needs the API key and API key secret to call the token endpoint using basic HTTP authentication. In addition, we can extend the list of parameters with the base URL to convert it to a more generic OAuth2 authenticator.

接下来，我们创建验证器本身。它需要API密钥和API密钥密钥来使用基本HTTP身份验证调用令牌端点。此外，我们可以使用基本URL扩展参数列表，以将其转换为更通用的OAuth2身份验证器。

The easiest way to create an authenticator is to inherit from the `AuthanticatorBase` base class:

创建验证器的最简单方法是从`AuthanticatorBase`基类继承：

```
public class TwitterAuthenticator : AuthenticatorBase {
    readonly string _baseUrl;
    readonly string _clientId;
    readonly string _clientSecret;

    public TwitterAuthenticator(string baseUrl, string clientId, string
clientSecret) : base("") {
        _baseUrl      = baseUrl;
        _clientId     = clientId;
        _clientSecret = clientSecret;
    }
```

```
    protected override async ValueTask<Parameter>
GetAuthenticationParameter(string accessToken) {
        var token = string.IsNullOrEmpty(Token) ? await GetToken() : Token;
        return new HeaderParameter(KnownHeaders.Authorization, token);
    }
}
```

During the first call made by the client using the authenticator, it will find out that the `Token` property is empty. It will then call the `GetToken` function to get the token once and reuse the token going forward.

在客户端使用身份验证器进行的第一次调用期间，它将发现"令牌"属性为空。然后它将调用`GetToken`函数来获取令牌一次，并继续重用令牌。

Now, we need to implement the `GetToken` function in the class:

现在，我们需要在类中实现`GetToken`函数：

```
async Task<string> GetToken() {
    var options = new RestClientOptions(_baseUrl);
    using var client = new RestClient(options) {
        Authenticator = new HttpBasicAuthenticator(_clientId, _clientSecret),
    };

    var request = new RestRequest("oauth2/token")
        .AddParameter("grant_type", "client_credentials");
    var response = await client.PostAsync<TokenResponse>(request);
    return $"{response!.TokenType} {response!.AccessToken}";
}
```

As we need to make a call to the token endpoint, we need our own short-lived instance of `RestClient`. Unlike the actual Twitter client, it will use the `HttpBasicAuthenticator` to send the API key and secret as the username and password. The client then gets disposed as we only use it once.

当我们需要调用令牌端点时，我们需要自己的`RestClient`临时实例。与实际的Twitter客户端不同，它将使用`HttpBasicAuthenticator`发送API密钥和密码作为用户名和密码。然后客户机被处理掉，因为我们只使用它一次。

Here we add a POST parameter `grant_type` with `client_credentials` as its value. At the moment, it's the only supported value.

这里我们添加了一个POST参数`grant_type`，其值为`client_credentials`。目前，它是唯一受支持的值。

The POST request will use the `application/x-www-form-urlencoded` content type by default.

默认情况下，POST请求将使用`application/x-www-form-urlencoded`内容类型。

#API client

#API 客服端

Now, we can start creating the API client itself. Here we start with a single function that retrieves one Twitter user. Let's being by defining the API client interface:

现在，我们可以开始创建API客户端本身了。这里我们从一个函数开始，该函数检索一个Twitter用户。让我们通过定义API客户端接口：

```
public interface ITwitterClient {
    Task<TwitterUser> GetUser(string user);
}
```

As the function returns a `TwitterUser` instance, we need to define it as a model:

由于函数返回一个`TwitterUser`实例，我们需要将其定义为一个模型：

```
public record TwitterUser(string Id, string Name, string Username);
```

When that is done, we can implement the interface and add all the necessary code blocks to get a working API client.

完成后，我们可以实现该接口并添加所有必要的代码块，以获得一个工作的API客户端。

The client class needs the following:

客户端类需要以下内容：

- A constructor, which accepts API credentials to pass to the authenticator
- 构造函数，它接受API凭据以传递给身份验证器
- A wrapped `RestClient` instance with the Twitter API base URI pre-configured
- 预先配置了Twitter API基本URI的包装`RestClient`实例
- The `TwitterAuthenticator` that we created previously as the client authenticator
- 我们之前创建的`TwitterAuthenticator`是客户端身份验证器
- The actual function to get the user
- 获取用户的实际函数

```
public class TwitterClient : ITwitterClient, IDisposable {
    readonly RestClient _client;

    public TwitterClient(string apiKey, string apiKeySecret) {
        var options = new RestClientOptions("https://api.twitter.com/2");

        _client = new RestClient(options) {
            Authenticator = new TwitterAuthenticator("https://api.twitter.com",
apiKey, apiKeySecret)
        };
    }

    public async Task<TwitterUser> GetUser(string user) {
        var response = await
_client.GetJsonAsync<TwitterSingleObject<TwitterUser>>(
            "users/by/username/{user}",
            new { user }
        );
        return response!.Data;
```

```
    }

    record TwitterSingleObject<T>(T Data);

    public void Dispose() {
        _client?.Dispose();
        GC.SuppressFinalize(this);
    }
}
```

The code above includes a couple of things that go beyond the "basics", and so we won't cover them here:

上面的代码包含了一些超出"基础"的内容，因此我们将不在这里介绍它们：

- The API client class needs to be disposable, so that it can dispose of the wrapped `HttpClient` instance
- API客户端类需要是一次性的，这样它就可以处理包装好的`HttpClient`实例
- Twitter API returns wrapped models. In this case, we use the `TwitterSingleObject` wrapper. In other methods, you'd need a similar object with `T[] Data` to accept collections
- Twitter API返回包装模型。在本例中，我们使用`TwitterSingleObject`包装器。在其他方法中，需要具有`T[] Data`的类似对象来接受集合

You can find the full example code in this gistopen in new window.

您可以在新窗口的gistopen中找到完整的示例代码。

Such a client can and should be used *as a singleton*, as it's thread-safe and authentication-aware. If you make it a transient dependency, you'll keep bombarding Twitter with token requests and effectively half your request limit.

这样的客户端可以而且应该*作为单例*使用，因为它是线程安全的，并且支持身份验证。如果你让它成为一个暂时的依赖，你会不断地用令牌请求轰炸Twitter，有效地降低了你的请求限制。

You can, for example, register it in the DI container:

例如，您可以在DI容器中注册它：

```
services.AddSingleton<ITwitterClient>(
    new TwitterClient(
        Configuration["Twitter:ApiKey"],
        Configuration["Twitter:ApiKeySecret"]
    )
);
```

#Create a request

#创建请求

Before making a request using `RestClient`, you need to create a request instance:

在使用`RestClient`发出请求之前，您需要创建一个请求实例：

```
var request = new RestRequest(resource); // resource is the sub-path of the
client base path
```

The default request type is `GET` and you can override it by setting the `Method` property. You can also set the method using the constructor overload:

默认的请求类型是`GET`，您可以通过设置`Method`属性来设置它。

```
var request = new RestRequest(resource, Method.Post);
```

After you've created a `RestRequest`, you can add parameters to it. Below, you can find all the parameter types supported by RestSharp.

创建`RestRequest`后，可以向其添加参数。在下面，您可以找到RestSharp支持的所有参数类型。

#Http Header

#Http头参数

Adds the parameter as an HTTP header that is sent along with the request. The header name is the parameter's name and the header value is the value.

将参数添加为随请求一起发送的HTTP标头。标题名称是参数的名称，标题值是值。

Content-Type

内容类型

RestSharp will use the correct content type by default. Avoid adding the `Content-Type` header manually to your requests unless you are absolutely sure it is required. You can add a custom content type to the body parameter itself.

默认情况下，RestSharp将使用正确的内容类型。避免将`Content Type`标头手动添加到请求中，除非您绝对确定它是必需的。可以将自定义内容类型添加到主体参数本身。

#Get or Post

`GetOrPost` behaves differently based on the method. If you execute a GET call, RestSharp will append the parameters to the Url in the form `url?name1=value1&name2=value2`.

`GetOrPost`基于函数而表现不同。如果执行GET调用，RestSharp将以“Url”的形式将参数附加到`url?name1=value1&name2=value 2`。

On a POST or PUT Requests, it depends on whether you have files attached to a Request. If not, the Parameters will be sent as the body of the request in the form `name1=value1&name2=value2`. Also, the request will be sent as `application/x-www-form-urlencoded`.

对于POST或PUT请求，这取决于您是否将文件附加到请求。如果没有，参数将作为请求主体以`name1=value1&name2=value 2`的形式发送。此外，请求将以`application/x-www-form-urlencoded`的形式发送。

In both cases, name and value will automatically be url-encoded.

在这两种情况下，名称和值将自动进行url编码。

If you have files, RestSharp will send a `multipart/form-data` request. Your parameters will be part of this request in the form:

如果您有文件，RestSharp将发送一个`multipart/form-data`请求。您的参数将在以下表格中作为此请求的一部分：

```
Content-Disposition: form-data; name="parameterName"

ParameterValue
```

#AddObject

You can avoid calling `AddParameter` multiple times if you collect all the parameters in an object, and then use `AddObject`. For example, this code:

如果收集对象中的所有参数，然后使用`AddObject`，则可以避免多次调用`AddParameter`。例如，以下代码：

```
var params = new {
    status = 1,
    priority = "high",
    ids = new [] { "123", "456" }
};
request.AddObject(params);
```

is equivalent to:

相当于：

```
request.AddParameter("status", 1);
request.AddParameter("priority", "high");
request.AddParameter("ids", "123,456");
```

Remember that `AddObject` only works if your properties have primitive types. It also works with collections of primitive types as shown above.

请记住，`AddObject`仅在属性具有基元类型时有效。它还可以处理基元类型的集合，如上所示。

If you need to override the property name or format, you can do it using the `RequestProperty` attribute. For example:

如果需要重写属性名称或格式，可以使用`RequestProperty`属性。例如：

```
public class RequestModel {
    // override the name and the format
    [RequestAttribute(Name = "from_date", Format = "d")]
    public DateTime FromDate { get; set; }
}

// add it to the request
request.AddObject(new RequestModel { FromDate = DateTime.Now });
```

In this case, the request will get a GET or POST parameter named `from_date` and its value would be the current date in short date format.

在这种情况下，请求将获得一个名为`from_date`的GET或POST参数，其值将是短日期格式的当前日期。

#Url Segment

#Url 段

Unlike `GetOrPost`, this `ParameterType` replaces placeholder values in the `RequestUrl`:

与`GetOrPost`不同，此`ParameterType`替换了`RequestUrl`中的占位符值：

```
var request = new RestRequest("health/{entity}/status")
    .AddUrlSegment("entity", "s2");
```

When the request executes, RestSharp will try to match any `{placeholder}` with a parameter of that name (without the `{}`) and replace it with the value. So the above code results in `health/s2/status` being the url.

当请求执行时，RestSharp将尝试将任何`{placeholder}`与该名称的参数（不带`{}`）匹配，并将其替换为值。因此，上面的代码导致`health/s2/status`是url。

#Request Body

#请求主体

If this parameter is set, its value will be sent as the body of the request.

如果设置了此参数，则其值将作为请求正文发送。

We recommend using `AddJsonBody` or `AddXmlBody` methods instead of `AddParameter` with type `BodyParameter`. Those methods will set the proper request type and do the serialization work for you.

我们建议使用`AddJsonBody`或`AddXmlBody`方法，而不是类型为`BodyParameter`的`AddParameter`。这些方法将设置正确的请求类型并为您执行序列化工作。

#AddStringBody

If you have a pre-serialized payload like a JSON string, you can use `AddStringBody` to add it as a body parameter. You need to specify the content type, so the remote endpoint knows what to do with the request body. For example:

如果您有一个预序列化的负载（如JSON字符串），则可以使用`AddStringBody`将其添加为body参数。您需要指定内容类型，以便远程端点知道如何处理请求主体。例如：

```
const json = "{ data: { foo: \"bar\" } }";
request.AddStringBody(json, ContentType.Json);
```

You can specify a custom body content type if necessary. The `contentType` argument is available in all the overloads that add a request body.

如有必要，可以指定自定义正文内容类型。`contentType`参数在添加请求体的所有重载中都可用。

#AddJsonBody

When you call `AddJsonBody`, it does the following for you:

当您调用`AddJsonBody`时，它会为您执行以下操作：

- Instructs the RestClient to serialize the object parameter as JSON when making a request
- 指示RestClient在发出请求时将对象参数序列化为JSON

- Sets the content type to `application/json`
- 将内容类型设置为`application/json`
- Sets the internal data type of the request body to `DataType.Json`
- 将请求主体的内部数据类型设置为`DataType.Json`

WARNING

警告

Do not send JSON string or some sort of `JObject` instance to `AddJsonBody`; it won't work! Use `AddStringBody` instead.

Here is the example:

不要将JSON字符串或某种`JObject`实例发送到`AddJsonBody`；应用对此并不支持！改用`AddStringBody`。
下面是一个例子：

```
var param = new MyClass { IntData = 1, StringData = "test123" };
request.AddJsonBody(param);
```

#AddXmlBody

When you call `AddXmlBody`, it does the following for you:

当您调用`AddXmlBody`时，它会为您执行以下操作：

- Instructs the RestClient to serialize the object parameter as XML when making a request
- 指示RestClient在发出请求时将对象参数序列化为XML
- Sets the content type to `application/xml`
- 将内容类型设置为`application/xml`
- Sets the internal data type of the request body to `DataType.Xml`
- 将请求主体的内部数据类型设置为`DataType.Xml`

WARNING

警告

Do not send XML string to `AddXmlBody`; it won't work!

不要将XML字符串发送到`AddXmlBody`；应用对此并不支持！

#Query String

`QueryString` works like `GetOrPost`, except that it always appends the parameters to the url in the form `url?name1=value1&name2=value2`, regardless of the request method.

`QueryString`的工作方式类似于`GetOrPost`，只是它总是以 url 的形式将参数附加到`url?name1=value1&name2=value 2`，而不考虑请求方法。

Example:

例子：

```
var client = new RestClient("https://search.me");
var request = new RestRequest("search")
    .AddParameter("foo", "bar");
var response = await client.GetAsync<SearchResponse>(request);
```

It will send a `GET` request to `https://search.me/search?foo=bar")`.

它会向`https://search.me/search?foo=bar")`发送一个`GET`请求。

For `POST`-style requests you need to add the query string parameter explicitly:

对于`POST`样式的请求，需要显式添加查询字符串参数：

```
request.AddQueryParameter("foo", "bar");
```

In some cases, you might need to prevent RestSharp from encoding the query string parameter. To do so, set the `encode` argument to `false` when adding the parameter:

在某些情况下，您可能需要防止RestSharp对查询字符串参数进行编码。为此，在添加参数时，将`encode`参数设置为`false`：

```
request.AddQueryParameter("foo", "bar/fox", false);
```

#Making a call

#进行通话

Once you've added all the parameters to your `RestRequest`, you are ready to make a request.

将所有参数添加到`RestRequest`后，就可以发出请求了。

`RestClient` has a single function for this:

`RestClient`有一个功能：

```
public async Task<RestResponse> ExecuteAsync(
    RestRequest request,
    CancellationToken cancellationToken = default
)
```

You can also avoid setting the request method upfront and use one of the overloads:

您还可以避免预先设置请求方法，并使用以下重载之一：

```
Task<RestResponse> ExecuteGetAsync(RestRequest request, CancellationToken
cancellationToken)
Task<RestResponse> ExecutePostAsync(RestRequest request, CancellationToken
cancellationToken)
Task<RestResponse> ExecutePutAsync(RestRequest request, CancellationToken
cancellationToken)
```

When using any of those methods, you will get the response content as string in `response.Content`.

当使用这些方法中的任何一种时，您将在`response.content`中获得字符串形式的响应内容。

RestSharp can deserialize the response for you. To use that feature, use one of the generic overloads:

RestSharp可以为您反序列化响应。要使用该功能，请使用以下通用重载之一：

```
Task<RestResponse<T>> ExecuteAsync<T>(RestRequest request, CancellationToken
cancellationToken)
Task<RestResponse<T>> ExecuteGetAsync<T>(RestRequest request, CancellationToken
cancellationToken)
Task<RestResponse<T>> ExecutePostAsync<T>(RestRequest request, CancellationToken
cancellationToken)
Task<RestResponse<T>> ExecutePutAsync<T>(RestRequest request, CancellationToken
cancellationToken)
```

All the overloads that return `RestResponse` or `RestResponse<T>` don't throw an exception if the server returns an error. Read more about it here.

如果服务器返回错误，则所有返回`Resresponse`或`Response<T>`的重载不会引发异常。点击此处了解更多信息.

If you just need a deserialized response, you can use one of the extensions:

如果只需要反序列化响应，可以使用以下扩展之一：

```
Task<T> GetAsync<T>(RestRequest request, CancellationToken cancellationToken)
Task<T> PostAsync<T>(RestRequest request, CancellationToken cancellationToken)
Task<T> PutAsync<T>(RestRequest request, CancellationToken cancellationToken)
Task<T> HeadAsync<T>(RestRequest request, CancellationToken cancellationToken)
Task<T> PatchAsync<T>(RestRequest request, CancellationToken cancellationToken)
Task<T> DeleteAsync<T>(RestRequest request, CancellationToken cancellationToken)
```

Those extensions will throw an exception if the server returns an error, as there's no other way to float the error back to the caller.

如果服务器返回错误，这些扩展将抛出异常，因为没有其他方法将错误浮回调用方式。

#JSON requests

#JSON 返回体

To make a simple `GET` call and get a deserialized JSON response with a pre-formed resource string, use this:

要进行一个简单的`GET`调用，并使用预先形成的资源字符串获取反序列化的JSON响应，请使用以下命令：

```
var response = await client.GetJsonAsync<TResponse>("endpoint?foo=bar",
cancellationToken);
```

You can also use a more advanced extension that uses an object to compose the resource string:

您还可以使用更高级的扩展，该扩展使用对象来组成资源字符串：

```
var client = new RestClient("https://example.org");
var args = new {
    id = "123",
    foo = "bar"
};
// Will make a call to https://example.org/endpoint/123?foo=bar
var response = await client.GetJsonAsync<TResponse>("endpoint/{id}", args,
cancellationToken);
```

It will search for the URL segment parameters matching any of the object properties and replace them with values. All the other properties will be used as query parameters.

它将搜索与任何对象属性匹配的URL段参数，并将其替换为值。所有其他属性将用作查询参数。

Similar things are available for `POST` requests.

类似的事情也适用于`POST`请求。

```
var request = new CreateOrder("123", "foo", 10100);
// Will post the request object as JSON to "orders" and returns a
// JSON response deserialized to OrderCreated
var result = client.PostJsonAsync<CreateOrder, OrderCreated>("orders", request,
cancellationToken);
```

```
var request = new CreateOrder("123", "foo", 10100);
// Will post the request object as JSON to "orders" and returns a
// status code, not expecting any response body
var statusCode = client.PostJsonAsync("orders", request, cancellationToken);
```

The same two extensions also exist for `PUT` requests (`PutJsonAsync`);

同样的两个扩展也存在于`PUT`请求（`PutJsonAsync`）；

#JSON streaming APIs

For HTTP API endpoints that stream the response data (like Twitter search streamopen in new window) you can use RestSharp with `StreamJsonAsync<T>`, which returns an `IAsyncEnumerable<T>`:

对于传输响应数据的HTTP API端点（如Twitter search streamopen in new window)您可以将RestSharp与`StreamJsonAsync<T>`一起使用，它返回一个`iasyncnumerable<T>`：

```
public async IAsyncEnumerable<SearchResponse> SearchStream(
    [EnumeratorCancellation] CancellationToken cancellationToken = default
) {
    var response = _client.StreamJsonAsync<TwitterSingleObject<SearchResponse>>(
        "tweets/search/stream", cancellationToken
    );

    await foreach (var item in response.WithCancellation(cancellationToken)) {
        yield return item.Data;
    }
}
```

The main limitation of this function is that it expects each JSON object to be returned as a single line. It is unable to parse the response by combining multiple lines into a JSON string.

该函数的主要限制是，它期望每个JSON对象作为单行返回。它无法通过将多行组合成JSON字符串来解析响应。

#Uploading files

#上传文件

To add a file to the request you can use the `RestRequest` function called `AddFile`. The main function accepts the `FileParameter` argument:

要向请求中添加文件，可以使用名为`AddFile`的`RestRequest`函数。主函数接受`FileParameter`参数：

```
request.AddFile(fileParameter);
```

You can instantiate the file parameter using `FileParameter.Create` that accepts a bytes array, or `FileParameter.FromFile`, which will load the file from disk.

您可以使用接受字节数组的`FileParameter.Create`或`FileParameter.FromFile`实例化文件参数，后者将从磁盘加载文件。

There are also extension functions that wrap the creation of `FileParameter` inside:

还有一些扩展函数将`FileParameter`的创建封装其中：

```
// Adds a file from disk
AddFile(parameterName, filePath, contentType);

// Adds an array of bytes
AddFile(parameterName, bytes, fileName, contentType);

// Adds a stream returned by the getFile function
AddFile(parameterName, getFile, fileName, contentType);
```

Remember that `AddFile` will set all the necessary headers, so please don't try to set content headers manually.

请记住，`AddFile`将设置所有必要的标题，因此请不要尝试手动设置内容标题。

#Downloading binary data

#下载二进制数据

There are two functions that allow you to download binary data from the remote API.

有两个函数允许您从远程API下载二进制数据。

First, there's `DownloadDataAsync`, which returns `Task<byte[]>`. It will read the binary response to the end, and return the whole binary content as a byte array. It works well for downloading smaller files.

首先是`DownloadDataAsync`，它返回`Task<byte[]>`。它将读取二进制响应到末尾，并将整个二进制内容作为字节数组返回。它适用于下载较小的文件。

For larger responses, you can use `DownloadStreamAsync` that returns `Task<Stream>`. This function allows you to open a stream reader and asynchronously stream large responses to memory or disk.

对于较大的响应，您可以使用`DownloadStreamAsync`，它返回`Task<Stream>`。此函数允许您打开流读取器，并将大型响应异步流传输到内存或磁盘。

#Blazor support

#Blazor支持

Inside a Blazor webassembly app, you can make requests to external API endpoints. Microsoft examples show how to do it with `HttpClient`, and it's also possible to use RestSharp for the same purpose.

在Blazor webassembly应用程序中，您可以向外部API端点发出请求。微软的例子展示了如何使用`HttpClient`，也可以使用RestSharp实现同样的目的。

You need to remember that webassembly has some platform-specific limitations. Therefore, you won't be able to instantiate `RestClient` using all of its constructors. In fact, you can only use `RestClient` constructors that accept `HttpClient` or `HttpMessageHandler` as an argument. If you use the default parameterless constructor, it will call the option-based constructor with default options. The options-based constructor will attempt to create an `HttpMessageHandler` instance using the options provided, and it will fail with Blazor, as some of those options throw thw "Unsupported platform" exception.

您需要记住，webassembly有一些特定于平台的限制。因此，您将无法使用其所有构造函数实例化`RestClient`。实际上，只能使用接受`HttpClient`或`HttpMessageHandler`作为参数的`RestClient`构造函数。如果使用默认的无参数构造函数，它将使用默认选项调用基于选项的构造函数。基于选项的构造函数将尝试使用提供的选项创建`HttpMessageHandler`实例，但在Blazor中会失败，因为其中一些选项会引发“不受支持的平台”异常。

Here is an example how to register the `RestClient` instance globally as a singleton:

下面是一个如何将`RestClient`实例全局注册为单例的示例：

```
builder.Services.AddSingleton(new RestClient(new HttpClient()));
```

Then, on a page you can inject the instance:

然后，在页面上可以注入实例：

```
@page "/fetchdata"
@using RestSharp
@inject RestClient _restClient
```

And then use it:

之后会启用下列代码：

```
@code {
    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync() {
```

```
        forecasts = await _restClient.GetJsonAsync<WeatherForecast[]>
("http://localhost:5104/weather");
    }

    public class WeatherForecast {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    }
}
```

In this case, the call will be made to a WebAPI server hosted at `http://localhost:5104/weather`. Remember that if the WebAPI server is not hosting the webassembly itself, it needs to have a CORS policy configured to allow the webassembly origin to access the API endpoint from the browser.

在这种情况下，将调用位于的WebAPI服务器`http://localhost:5104/weather`. 请记住，如果WebAPI服务器不是webassembly本身的宿主，则需要配置CORS策略，以允许WebAssemblyOrigin从浏览器访问API端点。

Help us by improving this page! open in new window

帮助我们改进此页面！在新窗口中打开

Last Updated: 2022/7/21 17:09:28

Contributors: Marcel Juen

Serialization

序列化

RestSharp has JSON and XML serializers built in.

RestSharp内置了JSON和XML序列化程序。

TIP

小提示

The default behavior of RestSharp is to swallow deserialization errors and return `null` in the `Data` property of the response. Read more about it in the Error Handling.

RestSharp的默认行为是吞下反序列化错误，并在响应的“Data”属性中返回“null”。在错误处理中了解更多信息.

#JSON

The default JSON serializer uses `System.Text.Json`, which is a part of .NET since .NET 6. For earlier versions, it is added as a dependency. There are also a few serializers provided as additional packages.

默认的JSON序列化程序使用`System.Text.Json`，从.NET 6开始就是.NET的一部分。对于早期版本，它是作为依赖项添加的。还有一些序列化程序作为附加包提供。

By default, RestSharp will use `JsonSerializerDefaults.Web` configuration. If necessary, you can specify your own options:

默认情况下，RestSharp将使用`JsonSerializerDefaults.Web`。如有必要，您可以指定自己的选项：

```
client.UseSystemTextJson(new JsonSerializerOptions {...});
```

#XML

The default XML serializer is `DotNetXmlSerializer`, which uses `System.Xml.Serialization` library from .NET.

默认的XML序列化程序是使用了.Net中`System.Xml.Serialization`库的`DotNetXmlSerializer`模块。

In previous versions of RestSharp, the default XML serializer was a custom RestSharp XML serializer. To make the code library size smaller, that serializer is now available as a separate package `RestSharp.Serializers.Xml` open in new window. You can add it back if necessary by installing the package and adding it to the client:

在以前版本的RestSharp中，默认的XML序列化程序是自定义的RestSharp XML序列化程序。为了减小代码库的大小，该序列化程序现在可以作为一个单独的包`RestSharp.Serializers.Xml`在新窗口中打开. 如果需要，您可以通过安装软件包并将其添加到客户端来将其添加回来：

```
client.UseXmlSerializer();
```

As before, you can supply three optional arguments for a custom namespace, custom root element, and if you want to use `SerializeAs` and `DeserializeAs` attributed.

如前所述，您可以为自定义名称空间、自定义根元素提供三个可选参数，如果您想使用`SerializeAs`和`DeserializeAs`属性。

#NewtonsoftJson (aka Json.Net)

The `NewtonsoftJson` package is the most popular JSON serializer for .NET. It handles all possible scenarios and is very configurable. Such a flexibility comes with the cost of performance. If you need speed, keep the default JSON serializer.

`NewtonsoftJson`包是.NET中最流行的JSON序列化程序。它处理所有可能的场景，并且可配置性极强。这种灵活性伴随着性能成本。如果需要速度，请保留默认的JSON序列化程序。

RestSharp support Json.Net serializer via a separate package `RestSharp.Serializers.NewtonsoftJson` open in new window.

RestSharp支持Json。Net序列化程序通过单独的包`RestSharp.Serializers.NewtonsoftJson`在新窗口中打开.

WARNING

警告

Please note that `RestSharp.Newtonsoft.Json` package is not provided by RestSharp, is marked as obsolete on NuGet, and no longer supported by its creator.

请注意RestSharp不提供`RestSharp.Newtonsoft.Json`包，在NuGet上标记为已过时，其创建者不再支持它。

Use the extension method provided by the package to configure the client:

使用包提供的扩展方法配置客户端：

```
client.UseNewtonsoftJson();
```

The serializer configures some options by default:

序列化程序默认配置一些选项：

```
JsonSerializerSettings DefaultSettings = new JsonSerializerSettings {
    ContractResolver     = new CamelCasePropertyNamesContractResolver(),
    DefaultValueHandling = DefaultValueHandling.Include,
    TypeNameHandling     = TypeNameHandling.None,
    NullValueHandling    = NullValueHandling.Ignore,
    Formatting           = Formatting.None,
    ConstructorHandling  = ConstructorHandling.AllowNonPublicDefaultConstructor
};
```

If you need to use different settings, you can supply your instance of `JsonSerializerSettings` as a parameter for the extension method.

如果需要使用不同的设置，可以将`JsonSerializerSettings`实例作为扩展方法的参数提供。

#Custom

#约定

You can also implement your custom serializer. To support both serialization and deserialization, you must implement the `IRestSerializer` interface.

您还可以实现自定义序列化程序。要同时支持序列化和反序列化，必须实现`IRestSerializer`接口。

Here is an example of a custom serializer that uses `System.Text.Json`:

下面是一个使用`System.Text.Json`的自定义序列化程序示例：

```
public class SimpleJsonSerializer : IRestSerializer {
    public string Serialize(object obj) => JsonSerializer.Serialize(obj);

    public string Serialize(Parameter bodyParameter) =>
Serialize(bodyParameter.Value);

    public T Deserialize<T>(IRestResponse response) =>
JsonSerializer.Deserialize<T>(response.Content);

    public string[] SupportedContentTypes { get; } = {
        "application/json", "text/json", "text/x-json", "text/javascript",
"*+json"
    };

    public string ContentType { get; set; } = "application/json";

    public DataFormat DataFormat { get; } = DataFormat.Json;
}
```

The value of the `SupportedContentTypes` property will be used to match the serializer with the response `Content-Type` headers.

`SupportedContentTypes` 属性的值将用于将序列化程序与响应 `Content-Type` 标头匹配。

The `ContentType` property will be used when making a request so the server knows how to handle the payload.

`ContentType` 属性将在发出请求时使用，以便服务器知道如何处理有效负载。

Help us by improving this page! open in new window

Last Updated: 2022/7/21 17:09:28

Contributors: Marcel Juen

Authenticators

证书

RestSharp includes authenticators for basic HTTP (Authorization header), NTLM and parameter-based systems.

RestSharp包括用于基本HTTP（授权头）、NTLM和基于参数的系统的认证器。

#Basic Authentication

#基本认证

The `HttpBasicAuthenticator` allows you pass a username and password as a basic `Authorization` header using a base64 encoded string.

`HttpBasicAuthenticator` 允许您使用base64编码字符串将用户名和密码作为基本的 `Authorization` 头传递。

```
var client = new RestClient("http://example.com");
client.Authenticator = new HttpBasicAuthenticator("username", "password");
```

#OAuth1

For OAuth1 authentication the `OAuth1Authenticator` class provides static methods to help generate an OAuth authenticator.

对于OAuth1身份验证，`OAuth1Authenticator` 类提供静态方法来帮助生成OAuth身份验证器。

#Request token

#请求令牌

This method requires a `consumerKey` and `consumerSecret` to authenticate.

此方法需要 `consumerKey` 和 `consumerSecret` 进行身份验证。

```
var client = new RestClient("http://example.com");
client.Authenticator = OAuth1Authenticator.ForRequestToken(consumerKey,
consumerSecret);
```

#Access token

#访问令牌

This method retrieves an access token when provided `consumerKey`, `consumerSecret`, `oauthToken`, and `oauthTokenSecret`.

当提供`consumerKey`、`consumerSecret`、`oauthToken`和`oauthTokenSecret`时，此方法检索访问令牌。

```
client.Authenticator = OAuth1Authenticator.ForAccessToken(
    consumerKey, consumerSecret, oauthToken, oauthTokenSecret
);
```

This method also includes an optional parameter to specify the `OAuthSignatureMethod`.

此方法还包括一个可选参数，用于指定`OAuthSignatureMethod`。

```
client.Authenticator = OAuth1Authenticator.ForAccessToken(
    consumerKey, consumerSecret, oauthToken, oauthTokenSecret,
    OAuthSignatureMethod.PlainText
);
```

#0-legged OAuth

The same access token authenticator can be used in 0-legged OAuth scenarios by providing `null` for the `consumerSecret`.

通过为`consumerSecret`提供`null`，可以在0-legged OAuth场景中使用相同的访问令牌验证器。

```
client.Authenticator = OAuth1Authenticator.ForAccessToken(
    consumerKey, null, oauthToken, oauthTokenSecret
);
```

#OAuth2

RestSharp has two very simple authenticators to send the access token as part of the request.

RestSharp有两个非常简单的验证器，将访问令牌作为请求的一部分发送。

`OAuth2UriQueryParameterAuthenticator` accepts the access token as the only constructor argument, and it will send the provided token as a query parameter `oauth_token`.

`OAuth2UriQueryParameterAuthenticator`接受访问令牌作为唯一的构造函数参数，并将提供的令牌作为查询参数`oauth_token`发送。

`OAuth2AuthorizationRequestHeaderAuthenticator` has two constructors. One only accepts a single argument, which is the access token. The other constructor also allows you to specify the token type. The authenticator will then add an `Authorization` header using the specified token type or `OAuth` as the default token type, and the token itself.

`OAuth2AuthorizationRequestHeaderAuthenticator`有两个构造函数。只接受一个参数，即访问令牌。另一个构造函数还允许您指定令牌类型。然后，验证器将使用指定的令牌类型或`OAuth`作为默认令牌类型以及令牌本身添加`Authorization`标头。

For example:

举个例子：

```
client.Authenticator = new OAuth2AuthorizationRequestHeaderAuthenticator(
    token, "Bearer"
);
```

The code above will tell RestSharp to send the bearer token with each request as a header. Essentially, the code above does the same as the sample for `JwtAuthenticator` below.

上面的代码将告诉RestSharp发送承载令牌，并将每个请求作为报头。本质上，上面的代码与下面的`JwtAuthenticator`示例相同。

As those authenticators don't do much to get the token itself, you might be interested in looking at our sample OAuth2 authenticator, which requests the token on its own.

由于这些验证器对获取令牌本身没有太大作用，您可能有兴趣查看我们的示例OAuth2验证器，它自己请求令牌。

#JWT

The JWT authentication can be supported by using `JwtAuthenticator`. It is a very simple class that can be constructed like this:

可以通过使用"JwtAuthenticator"来支持JWT认证。它是一个非常简单的类，可以这样构造：

```
var authenticator = new JwtAuthenticator(myToken);
```

For each request, it will add an `Authorization` header with the value `Bearer <your token>`.

对于每个请求，它将添加一个值为`Bearer <your token>`的`Authorization`标头。

As you might need to refresh the token from, you can use the `SetBearerToken` method to update the token.

由于您可能需要从中刷新令牌，因此可以使用`SetBearerToken`方法更新令牌。

#Custom Authenticator

#自定义验证器

You can write your own implementation by implementing `IAuthenticator` and registering it with your RestClient:

您可以通过实现`IAuthenticator`并将其注册到RestClient来编写自己的实现：

```
var client = new RestClient();
client.Authenticator = new SuperAuthenticator(); // implements IAuthenticator
```

The `Authenticate` method is the very first thing called upon calling `RestClient.Execute` or `RestClient.Execute<T>`. The `Authenticate` method is passed the `RestRequest` currently being executed giving you access to every part of the request data (headers, parameters, etc.)

`Authenticate`方法是调用`RestClient.Execute`或`RestClient.Execute<T>`时调用的第一个方法。`Authenticate`方法被传递给当前正在执行的`RestRequest`，允许您访问请求数据的每个部分（头、参数等）

You can find an example of a custom authenticator that fetches and uses an OAuth2 bearer token here.

您可以在这里找到一个获取并使用OAuth2承载令牌的自定义验证器示例.

Help us by improving this page! open in new window

Last Updated: 2022/7/21 17:09:28

Contributors: Marcel Juen

Error handling

错误处理

If there is a network transport error (network is down, failed DNS lookup, etc), or any kind of server error (except 404), `RestResponse.ResponseStatus` will be set to `ResponseStatus.Error`, otherwise it will be `ResponseStatus.Completed`.

如果存在网络传输错误（网络关闭、DNS查找失败等），或任何类型的服务器错误（404除外），`RestResponse.ResponseStatus` 将设置为`ResponseStatus.Error`，否则将为`ResponseStatus.Completed`。

If an API returns a 404, `ResponseStatus` will still be `Completed`. If you need access to the HTTP status code returned you will find it at `RestResponse.StatusCode`. The `Status` property is an indicator of completion independent of the API error handling.

如果API返回404，`ResponseStatus` 仍将保持 `Completed` 状态。如果您需要访问返回的HTTP状态代码，您可以在`RestResponse.StatusCode`中找到它。`Status`属性是独立于API错误处理的完成指示符。

Normally, RestSharp doesn't throw an exception if the request fails.

通常，如果请求失败，RestSharp不会抛出异常。

However, it is possible to configure RestSharp to throw in different situations, when it normally doesn't throw in favour of giving you the error as a property.

然而，可以将RestSharp配置为在不同的情况下抛出，当它通常不抛出时，将错误作为属性提供给您。

Property 属性	Behavior 表现
`FailOnDeserializationError`	Changes the default behavior when failed deserialization results in a successful response with an empty `Data` property of the response. Setting this property to `true` will tell RestSharp to consider failed deserialization as an error and set the `ResponseStatus` to `Error` accordingly. 当失败的反序列化导致响应`Data`属性为空时，更改默认行为。将此属性设置为`true`将告诉RestSharp将失败的反序列化视为错误，并相应地将`ResponseStatus`设置为`error`。

Property 属性	Behavior 表现
`ThrowOnDeserializationError`	Changes the default behavior when failed deserialization results in empty `Data` property of the response. Setting this property to `true` will tell RestSharp to throw when deserialization fails. 当失败的反序列化导致响应的空`Data`属性时，更改默认行为。将此属性设置为`true`将告诉RestSharp在反序列化失败时抛出。
`ThrowOnAnyError`	Setting this property to `true` changes the default behavior and forces RestSharp to throw if any errors occurs when making a request or during deserialization. 将此属性设置为`true`将更改默认行为，并在发出请求或反序列化期间发生任何错误时强制RestSharp抛出。

Those properties are available for the `RestClient` instance and will be used for all request made with that instance.

这些属性可用于`RestClient`实例，并将用于该实例的所有请求。

WARNING

警告

Please be aware that deserialization failures will only work if the serializer throws an exception when deserializing the response. Many serializers don't throw by default, and just return a `null` result. RestSharp is unable to figure out why `null` is returned, so it won't fail in this case. Check the serializer documentation to find out if it can be configured to throw on deserialization error.

请注意，只有在序列化程序在反序列化响应时引发异常时，反序列化失败才会起作用。默认情况下，许多序列化程序不抛出，只返回`null`结果。RestSharp无法找出返回`null`的原因，因此在这种情况下它不会失败。请查看序列化程序文档，了解是否可以将其配置为引发反序列化错误。

There are also slight differences on how different overloads handle exceptions.

不同的重载处理异常的方式也略有不同。

Asynchronous generic methods `GetAsync<T>`, `PostAsync<T>` and so on, which aren't a part of `RestClient` interface (those methods are extension methods) return `Task<T>`. It means that there's no `RestResponse` to set the response status to error. We decided to throw an exception when such a request fails. It is a trade-off between the API consistency and usability of the library. Usually, you only need the content of `RestResponse` instance to diagnose issues and most of the time the exception would tell you what's wrong.

异步泛型方法`GetAsync<T>`、`PostAsync<T>`等等，它们不是`RestClient`接口的一部分（这些方法是扩展方法），返回`Task<T>`。这意味着没有`RestResponse`将响应状态设置为错误。当这样的请求失败时，我们决定抛出一个异常。这是API一致性和库可用性之间的权衡。通常，您只需要`RestResponse`实例的内容来诊断问题，大多数情况下，异常会告诉您出了什么问题。

Below you can find how different extensions deal with errors. Note that functions, which don't throw by default, will throw exceptions when `ThrowOnAnyError` is set to `true`.

下面您可以找到不同的扩展如何处理错误。请注意，默认情况下不抛出的函数在`ThrowOnAnyError`设置为`true`时将抛出异常。

Function	Throws on errors
`ExecuteAsync`	No
`ExecuteGetAsync`	No
`ExecuteGetAsync<T>`	No
`ExecutePostAsync`	No
`ExecutePutAsync`	No
`ExecuteGetAsync<T>`	No
`ExecutePostAsync<T>`	No
`ExecutePutAsync<T>`	No
`GetAsync`	Yes
`GetAsync<T>`	Yes
`PostAsync`	Yes
`PostAsync<T>`	Yes
`PatchAsync`	Yes
`PatchAsync<T>`	Yes
`DeleteAsync`	Yes
`DeleteAsync<T>`	Yes
`OptionsAsync`	Yes
`OptionsAsync<T>`	Yes
`HeadAsync`	Yes
`HeadAsync<T>`	Yes

In addition, all the functions for JSON requests, like `GetJsonAsync` and `PostJsonAsyn` throw an exception if the HTTP call fails.

此外，如果HTTP调用失败，JSON请求的所有函数，如 `GetJsonAsync` 和 `PostJsonAsyn`，都会引发异常。

Help us by improving this page! open in new window

Last Updated: 2022/7/21 17:09:28

Contributors: Marcel Juen

RestSharp v107

The latest version of RestSharp is v107. It's a major upgrade, which contains quite a few breaking changes.

RestSharp的最新版本是v107。这是一次重大升级，其中包含了许多突破性的更改。

The most important change is that RestSharp stop using the legacy `HttpWebRequest` class, and uses well-known 'HttpClient' instead. This move solves lots of issues, like hanging connections due to improper `HttpClient` instance cache, updated protocols support, and many other problems.

最重要的变化是RestSharp停止使用遗留的`HttpWebRequest`类，而是使用著名的“HttpClient”。这一举措解决了许多问题，如由于不正确的`HttpClient`实例缓存导致的连接挂起、更新的协议支持以及许多其他问题。

Another big change is that `SimpleJson` is retired completely from the code base. Instead, RestSharp uses `JsonSerializer` from the `System.Text.Json` package, which is the default serializer for ASP.NET Core.

另一个重大变化是`SimpleJson`完全从代码库中退出。相反，RestSharp使用 ASP.NET Core默认序列化工具`System.Text.Json`中的`JsonSerializer`包。

Finally, most of the interfaces are now gone.

最后，大多数接口现在都消失了。

#Brief migration guide

#简要迁移指南

#RestClient and options

#RestClient和选项

The `IRestClient` interface is deprecated. You will be using the `RestClient` class instance.

`IRestClient`接口已弃用。您将使用`RestClient`类实例。

Most of the client options are moved to `RestClientOptions`. If you can't find the option you used to set on `IRestClient`, check the options, it's probably there.

大多数客户端选项都移动到`RestClientOptions`。如果找不到用于在`IRestClient`上设置的选项，请检查选项，它可能在那里。

This is how you can instantiate the client using the simplest possible way:

这是如何使用最简单的方式实例化客户端：

```
var client = new RestClient("https://api.myorg.com");
```

For customizing the client, use `RestClientOptions`:

要自定义客户端，请使用`RestClientOptions`：

```
var options = new RestClientOptions("https://api.myorg.com") {
    ThrowOnAnyError = true,
    Timeout = 1000
};
var client = new RestClient(options);
```

You can still change serializers and add default parameters to the client.

您仍然可以更改序列化程序并向客户端添加默认参数。

#RestClient lifecycle

#RestClient生命周期

Do not instantiate `RestClient` for each HTTP call. RestSharp creates a new instance of `HttpClient` internally, and you will get lots of hanging connections, and eventually exhaust the connection pool.

不要为每个HTTP调用实例化`RestClient`。RestSharp在内部创建了一个新的`HttpClient`实例，您将获得大量挂起的连接，并最终耗尽连接池。

If you use a dependency-injection container, register your API client as a singleton.

如果使用依赖注入容器，请将API客户端注册为单例。

#Body parameters#主体参数

Beware that most of the code generators, like Postman C# code gen, generate code for RestSharp before v107, and that code is broken. Such code worked mostly due to obscurity of previous RestSharp versions API. For example, Postman-generated code tells you to add the content-type header, and the accept header, which in many cases is an anti-pattern. It also posts JSON payload as string, where RestSharp provides you with serialization and deserialization of JSON out of the box.

请注意，大多数代码生成器（如Postman C#code gen）都是在v107之前为RestSharp生成代码的，并且该代码已被破坏。这类代码之所以能正常工作，主要是因为以前的RestSharp版本API晦涩难懂。例如，Postman生成的代码告诉您添加内容类型头和accept头，在许多情况下，这是一种反模式。它还将JSON负载作为字符串发布，其中RestSharp为您提供了JSON的即时序列化和反序列化。

Therefore, please read the Usage page and follow our guidelines when using RestSharp v107+.

因此，请阅读用法使用RestSharp v107+时，请翻页并遵循我们的指南。

Some of the points to be aware of:

需要注意的一些要点：

- `AddParameter("application/json", ..., ParameterType.RequestBody)` won't work, use `AddBody` instead, or better, `AddJsonBody`.
- `AddParameter("application/json", ..., ParameterType.RequestBody)` 已经弃用, 请改用 `AddBody` 替代或更加`AddJsonBody`.
- `AddJsonBody("{ foo: 'bar' }")` won't work (and it never worked), use `AddStringBody`. `AddJsonBody` is for serializable objects, not for strings.
- `AddJsonBody("{ foo: 'bar' }")` 已弃用， 请使用`AddStringBody`。`AddJsonBody` 用于可序列化对象，而不是字符串。

- If your `AddParameter(something, something, ParameterType.RequestBody)` doesn't work, try `AddBody` as it will do its best to figure out what kind of body you're adding.
- 如果您的`AddParameter（something，somethine，ParameterType.RequestBody）`不起作用，请尝试`AddBody`，因为它会尽最大努力找出您要添加的主体类型。

#Headers

Lots of code out there that uses RestSharp has lines like:

很多使用RestSharp的代码都有这样的行：

```
request.AddHeader("Content-Type", "application/json");
request.AddHeader("Accept", "application/json");
```

This is completely unnecessary, and often harmful. The `Content-Type` header is the content header, not the request header. It might be different per individual part of the body when using multipart-form data, for example. RestSharp sets the correct content-type header automatically, based on your body format, so don't override it. The `Accept` header is set by RestSharp automatically based on registered serializers. By default, both XML and JSON are supported. Only change the `Accept` header if you need something else, like binary streams, or plain text.

这是完全不必要的，而且往往是有害的。`Content-Type`报头是内容报头，而不是请求报头。例如，当使用多部分表单数据时，身体的每个部分可能不同。RestSharp根据您的正文格式自动设置正确的内容类型标题，因此不要覆盖它。`Accept`标头由RestSharp根据注册的序列化程序自动设置。默认情况下，支持XML和JSON。只有在需要其他内容（如二进制流或纯文本）时才更改`Accept`标题。

#Making requests

#发出请求

The `IRestRequest` interface is deprecated. You will be using the `RestRequest` class instance.

`IRestRequest`接口已弃用。您将使用`RestRequest`类实例。

You can still create a request as before:

您仍然可以像以前一样创建请求：

```
var request = new RestRequest();
```

Adding parameters hasn't changed much, except you cannot add cookie parameters to the request. It's because cookies are added to the `HttpMessageHandler` cookie container, which is not accessible inside the request class.

添加参数并没有太大变化，只是不能将cookie参数添加到请求中。这是因为cookie被添加到`HttpMessageHandler`cookie容器中，在请求类中无法访问该容器。

```
var request = new RestRequest()
    .AddQueryParameter("foo", "bar")
    .AddJsonBody(someObject);
```

Quite a few options previously available via `IRestRequest` are now in `RestClientOptions`. It's also because changing those options forced us to use a different HTTP message handler, and it caused hanging connections, etc.

以前通过`IRestRequest`提供的许多选项现在都在`RestClientOptions`中。这也是因为更改这些选项迫使我们使用不同的HTTP消息处理程序，并导致连接挂起等。

When you got a request instance, you can make a call:

当您收到请求实例时，您可以进行调用：

```
var request = new RestRequest()
    .AddQueryParameter("foo", "bar")
    .AddJsonBody(someObject);
var response = await client.PostAsync<MyResponse>(request, cancellationToken);
```

All the synchronous methods are gone. If you absolutely must call without using `async` and `await`, use `GetAwaiter().GetResult()` blocking call.

所有的同步方法都成为过去了。如果绝对必须在不使用`异步`和`等待`的情况下调用，请使用`GetAwaiter().GetResult()`阻塞调用。

The `IRestResponse` interface is deprecated. You get an instance of `RestResponse` or `RestResponse<T>` in return.

不推荐使用`IRestResponse`接口。您将得到一个`RestResponse`或`RestResponse<T>`的实例。

You can also use a simplified API for making POST and PUT requests:

您还可以使用简化的API进行POST和PUT请求：

```
var request = new MyRequest { Data = "foo" };
var response = await client.PostAsync<MyRequest, MyResponse>(request,
cancellationToken);
// response will be of type TResponse
```

This way you avoid instantiating `RestRequest` explicitly.

这样可以避免显式实例化`RestRequest`。

#Using your own HttpClient

#使用您自己的HttpClient

`RestClient` class has two constructors, which accept either `HttpClient` or `HttpMessageHandler` instance.

`RestClient`类有两个构造函数，它们接受`HttpClient`或`HttpMessageHandler`实例。

This way you can use a pre-configured `HttpClient` or `HttpMessageHandler`, customized for your needs.

通过这种方式，您可以使用预先配置的`HttpClient`或`HttpMessageHandler`，根据您的需要进行定制。

#Default serializers

#默认序列化程序

For JSON, RestSharp will use `JsonSerializer` from the `System.Text.Json` package. This package is now referenced by default, and it is the only dependency of the RestSharp NuGet package.

对于JSON，RestSharp将使用`System.Text.Json`的`JsonSerializer`包。This package is now referenced by default, and it is the only dependency of the RestSharp NuGet package.

The `Utf8` serializer package is deprecated as the package is not being updated.

`Utf8`序列化程序包已被弃用，因为该包未被更新。

For XML requests and responses RestSharp uses `DotNetXmlSerializer` and `DotNetXmlDeserializer`. Previously used default `XmlSerializer`, `XmlDeserializer`, and `XmlAttrobuteDeserializer` are moved to a separate package `RestSharp.Serializers.Xml`.

对于XML请求和响应，RestSharp使用`DotNetXmlSerializer`和`DotNetXmlDeserializer`。以前使用的默认`XmlSerializer`，`XmlDeserializer`和`XmlAttrobuteDeserializer`将移动到单独的包`XmlAttrobuteDeserializer`。

#NTLM authentication

#NTLM身份验证

The `NtlmAuthenticator` is deprecated.

`NtlmAuthenticator`已弃用。

NTLM authenticator was doing nothing more than telling `WebRequest` to use certain credentials. Now with RestSharp, the preferred way would be to set the `Credentials` or `UseDefaultCredentials` property in `RestClientOptions`.

NTLM验证器只不过是告诉`WebRequest`使用某些凭据。现在使用RestSharp，首选方法是在`RestClientOptions`中设置`Credentials`或`UseDefaultCredentials`属性。

The reason to remove it was that all other authenticators use `AuthenticatorBase`, which must return a parameter. In general, any authenticator is given a request before its made, so it can do something with it. NTLM doesn't work this way, it needs some settings to be provided for `HttpClientHandler`, which is set up before the `HttpClient` instance is created, and it happens once per RestClient instance, and it cannot be changed per request.

删除它的原因是所有其他验证器都使用`AuthenticatorBase`，它必须返回一个参数。一般来说，任何验证器在发出请求之前都会收到一个请求，因此它可以对其进行处理。NTLM不是以这种方式工作的，它需要为`HttpClientHandler`提供一些设置，该设置是在创建`HttpClient`实例之前设置的，每个RestClient实例都会发生一次，并且不能根据请求进行更改。

#Delegating handlers

#委托处理程序

You can easily build your own request/response pipeline, as you would with `HttpClient`. RestClient will create an `HttpMessageHandler` instance for its own use, using the options provided. You can, of course, provide your own instance of `HttpMessageHandler` as `RestSharpClient` has a constructor that accepts a custom handler and uses it to create an `HttpClient` instance. However, you'll be on your own with the handler configuration in this case.

您可以轻松构建自己的请求/响应管道，就像使用 `HttpClient` 一样。RestClient将使用提供的选项创建一个 `HttpMessageHandler` 实例供自己使用。当然，您可以提供自己的 `HttpMessageHandler` 实例，因为 `RestSharpClient` 有一个接受自定义处理程序的构造函数，并使用它创建 `HttpClient` 实例。

If you want to build a *pipeline*, use delegating handlersopen in new window. For example, you can use `HttpTracer` to debug your HTTP callsopen in new window like this:

如果要构建*管道*，请在新窗口中使用委托handlersopen。例如，您可以这般使用 `HttpTracer` 来在新窗口中调试您的HTTP callsopen：

```
var options = new RestClientOptions(_server.Url) {
    ConfigureMessageHandler = handler =>
        new HttpTracerHandler(handler, new ConsoleLogger(),
HttpMessageParts.All)
};
var client = new RestClient(options);
```

#Recommended usage

#推荐用法

`RestClient` should be thread-safe. It holds an instance of `HttpClient` and `HttpMessageHandler` inside. Do not instantiate the client for a single call, otherwise you get issues with hanging connections and connection pooling won't be possible.

`RestClient` 应该是线程安全的。它在内部保存了 `HttpClient` 和 `HttpMessageHandler` 的实例。不要为单个调用实例化客户端，否则会出现挂起连接的问题，连接池将不可能实现。

Do create typed API clients for your use cases. Use a single instance of `RestClient` internally in such an API client for making calls. It would be similar to using typed clients using `HttpClient`, for example:

为您的用例创建类型化API客户端。在这样的API客户端中，在内部使用单个 `RestClient` 实例进行调用。这类似于使用 `HttpClient` 的类型化客户端，例如：

```
public class GitHubClient {
    readonly RestClient _client;

    public GitHubClient() {
        _client = new RestClient("https://api.github.com/")
            .AddDefaultHeader(KnownHeaders.Accept,
"application/vnd.github.v3+json");
    }

    public Task<GitHubRepo[]> GetRepos()
        => _client.GetAsync<GitHubRepo[]>("users/aspnet/repos");
}
```

Do not use one instance of `RestClient` across different API clients.

不要跨不同的API客户端使用一个`RestClient`实例。

This documentation contains the complete example of a Twitter API client, which you can use as a reference.

本文档包含Twitter API客户端的完整示例，您可以将其用作参考。

#Presumably solved issues

#预计问题处理

The next RestSharp version presumably solves the following issues:

下一个RestSharp版本可能会解决以下问题：

- Connection pool starvation
- 连接池不足
- Hanging open TCP connections
- 挂起打开的TCP连接
- Improper handling of async calls
- 异步调用处理不当
- Various `SimpleJson` serialization quirks
- 各种`SimpleJson`序列化怪症
- HTTP/2 support
- HTTP/2支持
- Intermediate certificate issue
- 中间证书颁发
- Uploading large files (use file parameters with `Stream`)
- 上传大文件（使用带有`流`的文件参数）
- Downloading large files (use `DownloadFileStreamAsync`)
- 下载大文件（使用`DownloadFileStreamAsync`）

#Deprecated interfaces

#弃用的接口

The following interfaces are removed from RestSharp:

从RestSharp中删除了以下接口：

- `IRestClient`
- `IRestRequest`
- `IRestResponse`
- `IHttp`

#Motivation

#原因

All the deprecated interfaces had only one implementation in RestSharp, so those interfaces were abstracting nothing. It is now unclear what was the purpose for adding those interfaces initially.

所有不推荐的接口在RestSharp中只有一个实现，因此这些接口没有抽象任何内容。现在还不清楚最初添加这些接口的目的是什么。

What about mocking it, you might ask? The answer is: what would you do if you use a plain `HttpClient` instance? It doesn't implement any interface for the same reason - there's nothing to abstract, and there's only one implementation. We don't recommend mocking `RestClient` in your tests when you are testing against APIs that are controlled by you or people in your organisation. Test your clients against the real thing, as REST calls are I/O-bound. Mocking REST calls is like mocking database calls, and lead to a lot of issues in production even if all your tests pass against mocks.

你可能会问，嘲笑它怎么样？答案是：如果使用普通的`HttpClient`实例，您会怎么做？出于同样的原因，它没有实现任何接口-没有什么可抽象的，只有一个实现。当您针对您或您组织中的人员控制的API进行测试时，我们不建议在测试中模拟`HttpClient`。针对实际情况测试您的客户机，因为REST调用是I/O绑定的。模拟REST调用类似于模拟数据库调用，即使所有测试都通过了模拟，也会在生产中导致许多问题。

As mentioned in Recommended usage, we advise against using `RestClient` in the application code, and advocate wrapping it inside particular API client classes. Those classes would be under your control, and you are totally free to use interfaces there. If you absolutely must mock, you can mock your interfaces instead.

如推荐用法中所述，我们建议不要在应用程序代码中使用`RestClient`，并主张将其包装在特定的API客户端类中。这些类将在您的控制下，您完全可以在那里自由使用接口。如果您确实必须模拟，那么您可以模拟接口。

#Mocking

#模拟

Mocking an infrastructure component like RestSharp (or HttpClient) is not the best idea. Even if you check that all the parameters are added correctly to the request, your "unit test" will only give you a false sense of safety that your code actually works. But, you have no guarantee that the remote server will accept your request, or if you can handle the actual response correctly.

模仿RestSharp（或HttpClient）这样的基础设施组件不是最好的主意。即使您检查了所有参数是否正确添加到请求中，您的“单元测试”也只会给您一个错误的安全感，即您的代码实际工作。但是，您不能保证远程服务器会接受您的请求，或者您是否能够正确处理实际响应。

The best way to test HTTP calls is to make some, using the actual service you call. However, you might still want to check if your API client forms requests in a certain way. You might also be sure about what the remote server responds to your calls with, so you can build a set of JSON (or XML) responses, so you can simulate remote calls.

测试HTTP调用的最佳方法是使用您调用的实际服务进行一些测试。但是，您可能仍然希望检查您的API客户端是否以某种方式形成请求。您还可以确定远程服务器用什么响应您的调用，因此可以构建一组JSON（或XML）响应，以便模拟远程调用。

It is perfectly doable without using interfaces. As RestSharp uses `HttpClient` internally, it certainly uses `HttpMessageHandler`. Features like delegating handlers allow you to intercept the request pipeline, inspect the request, and substitute the response. You can do it yourself, or use a library like MockHttpopen in new window. They have an example provided in the repository README, so we have changed it for RestClient here:

这在不使用接口的情况下是完全可行的。由于RestSharp在内部使用`HttpClient`，它当然使用`HttpMessageHandler`。委托处理程序等功能允许您拦截请求管道、检查请求并替换响应。您可以自己做，也可以使用类似MockHttpopen in new window的库。他们在存储库自述文件中提供了一个示例，因此我们在这里为RestClient更改了它：

```
var mockHttp = new MockHttpMessageHandler();

// Setup a respond for the user api (including a wildcard in the URL)
mockHttp.When("http://localhost/api/user/*")
        .Respond("application/json", "{'name' : 'Test McGee'}"); // Respond with
JSON

// Instantiate the client normally, but replace the message handler
var client = new RestClient(...) { ConfigureMessageHandler = _ => mockHttp };

var request = new RestRequest("http://localhost/api/user/1234");
var response = await client.GetAsync(request);

// No network connection required
Console.Write(response.Content); // {'name' : 'Test McGee'}
```

#Reference

#参考

Below, you can find members of `IRestClient` and `IRestRequest` with their corresponding status and location in the new API.

在下面，您可以找到`IRestClient`和`IRestRequest`的成员及其在新API中的相应状态和位置。

`IRestClient` member	Where is it now?
`CookieContainer`	`RestClient`

`IRestClient` member	Where is it now?
`AutomaticDecompression`	`RestClientOptions`, changed type
`MaxRedirects`	`RestClientOptions`
`UserAgent`	`RestClientOptions`
`Timeout`	`RestClientOptions`, `RestRequest`
`Authenticator`	`RestClient`
`BaseUrl`	`RestClientOptions`
`Encoding`	`RestClientOptions`
`ThrowOnDeserializationError`	`RestClientOptions`
`FailOnDeserializationError`	`RestClientOptions`
`ThrowOnAnyError`	`RestClientOptions`
`PreAuthenticate`	`RestClientOptions`
`BaseHost`	`RestClientOptions`
`AllowMultipleDefaultParametersWithSameName`	`RestClientOptions`
`ClientCertificates`	`RestClientOptions`
`Proxy`	`RestClientOptions`
`CachePolicy`	`RestClientOptions`, changed type
`FollowRedirects`	`RestClientOptions`
`RemoteCertificateValidationCallback`	`RestClientOptions`
`Pipelined`	Not supported
`UnsafeAuthenticatedConnectionSharing`	Not supported
`ConnectionGroupName`	Not supported
`ReadWriteTimeout`	Not supported
`UseSynchronizationContext`	Not supported
`DefaultParameters`	`RestClient`
`UseSerializer(Func<IRestSerializer> serializerFactory)`	`RestClient`
`UseSerializer<T>()`	`RestClient`
`Deserialize<T>(IRestResponse response)`	`RestClient`

`IRestClient` member	Where is it now?
`BuildUri(IRestRequest request)`	`RestClient`
`UseUrlEncoder(Func<string, string> encoder)`	Extension
`UseQueryEncoder(Func<string, Encoding, string> queryEncoder)`	Extension
`ExecuteAsync<T>(IRestRequest request, CancellationToken cancellationToken)`	`RestClient`
`ExecuteAsync<T>(IRestRequest request, Method httpMethod, CancellationToken cancellationToken)`	Extension
`ExecuteAsync(IRestRequest request, Method httpMethod, CancellationToken cancellationToken)`	Extension
`ExecuteAsync(IRestRequest request, CancellationToken cancellationToken)`	Extension
`ExecuteGetAsync<T>(IRestRequest request, CancellationToken cancellationToken)`	Extension
`ExecutePostAsync<T>(IRestRequest request, CancellationToken cancellationToken)`	Extension
`ExecuteGetAsync(IRestRequest request, CancellationToken cancellationToken)`	Extension
`ExecutePostAsync(IRestRequest request, CancellationToken cancellationToken)`	Extension
`Execute(IRestRequest request)`	Deprecated
`Execute(IRestRequest request, Method httpMethod)`	Deprecated
`Execute<T>(IRestRequest request)`	Deprecated
`Execute<T>(IRestRequest request, Method httpMethod)`	Deprecated
`DownloadData(IRestRequest request)`	Deprecated
`ExecuteAsGet(IRestRequest request, string httpMethod)`	Deprecated
`ExecuteAsPost(IRestRequest request, string httpMethod)`	Deprecated
`ExecuteAsGet<T>(IRestRequest request, string httpMethod)`	Deprecated
`ExecuteAsPost<T>(IRestRequest request, string httpMethod)`	Deprecated
`BuildUriWithoutQueryParameters(IRestRequest request)`	Removed
`ConfigureWebRequest(Action<HttpWebRequest> configurator)`	Removed

`IRestClient` member	Where is it now?
`AddHandler(string contentType, Func<IDeserializer> deserializerFactory)`	Removed
`RemoveHandler(string contentType)`	Removed
`ClearHandlers()`	Removed

`IRestRequest` member	Where is it now?
`AlwaysMultipartFormData`	`RestRequest`
`JsonSerializer`	Deprecated
`XmlSerializer`	Deprecated
`AdvancedResponseWriter`	`RestRequest`, changed signature
`ResponseWriter`	`RestRequest`, changed signature
`Parameters`	`RestRequest`
`Files`	`RestRequest`
`Method`	`RestRequest`
`Resource`	`RestRequest`
`RequestFormat`	`RestRequest`
`RootElement`	`RestRequest`
`DateFormat`	`XmlRequest`
`XmlNamespace`	`XmlRequest`
`Credentials`	Removed, use `RestClientOptions`
`Timeout`	`RestRequest`
`ReadWriteTimeout`	Not supported
`Attempts`	`RestRequest`
`UseDefaultCredentials`	Removed, use `RestClientOptions`
`AllowedDecompressionMethods`	Removed, use `RestClientOptions`
`OnBeforeDeserialization`	`RestRequest`
`OnBeforeRequest`	`RestRequest`, changed signature

`IRestRequest` member	Where is it now?
`Body`	Removed, use `Parameters`
`AddParameter(Parameter p)`	`RestRequest`
`AddFile(string name, string path, string contentType)`	Extension
`AddFile(string name, byte[] bytes, string fileName, string contentType)`	Extension
`AddFile(string name, Action<Stream> writer, string fileName, long contentLength, string contentType)`	Extension
`AddFileBytes(string name, byte[] bytes, string filename, string contentType)`	Extension `AddFile`
`AddBody(object obj, string xmlNamespace)`	Deprecated
`AddBody(object obj)`	Extension
`AddJsonBody(object obj)`	Extension
`AddJsonBody(object obj, string contentType)`	Extension
`AddXmlBody(object obj)`	Extension
`AddXmlBody(object obj, string xmlNamespace)`	Extension
`AddObject(object obj, params string[] includedProperties)`	Extension
`AddObject(object obj)`	Extension
`AddParameter(string name, object value)`	Extension
`AddParameter(string name, object value, ParameterType type)`	Extension
`AddParameter(string name, object value, string contentType, ParameterType type)`	Extension
`AddOrUpdateParameter(Parameter parameter)`	Extension
`AddOrUpdateParameters(IEnumerable<Parameter> parameters)`	Extension
`AddOrUpdateParameter(string name, object value)`	Extension
`AddOrUpdateParameter(string name, object value, ParameterType type)`	Extension
`AddOrUpdateParameter(string name, object value, string contentType, ParameterType type)`	Extension
`AddHeader(string name, string value)`	Extension
`AddOrUpdateHeader(string name, string value)`	Extension

`IRestRequest` member	Where is it now?
`AddHeaders(ICollection<KeyValuePair<string, string>> headers)`	Extension
`AddOrUpdateHeaders(ICollection<KeyValuePair<string, string>> headers)`	Extension
`AddCookie(string name, string value)`	Extension
`AddUrlSegment(string name, string value)`	Extension
`AddUrlSegment(string name, string value, bool encode)`	Extension
`AddUrlSegment(string name, object value)`	Extension
`AddQueryParameter(string name, string value)`	Extension
`AddQueryParameter(string name, string value, bool encode)`	Extension
`AddDecompressionMethod(DecompressionMethods decompressionMethod)`	Not supported
`IncreaseNumAttempts()`	Made internal

Help us by improving this page! open in new window

Last Updated: 2022/7/21 17:09:28

Contributors: Marcel Juen

Get Help

获取帮助

Got issues, questions, suggestions? Please read this page carefully to understand how you can get help working with RestSharp.

有问题、问题或建议吗？请仔细阅读本页，了解如何使用RestSharp获得帮助。

#Questions

#问题

The most effective way to resolve questions about using RestSharp is StackOverflow.

解决有关使用RestSharp的问题的最有效方法是StackOverflow。

RestSharp has a large user base. Tens of thousands of projects and hundreds of thousands of developers use RestSharp on a daily basis. So, asking questions on **StackOverflow** with restsharpopen in new window tag would most definitely lead you to a solution.

RestSharp拥有庞大的用户群。数万个项目和数十万开发人员每天都在使用RestSharp。因此，使用restsharpopen在新窗口中在**StackOverflow**上提问标签绝对会引导您找到解决方案。

WARNING

警告

Please do not use GitHub issues to ask question about using RestSharp.

请不要使用GitHub问题来询问有关使用RestSharp的问题。

#Discussions

#讨论

We have a mail listopen in new window at Google Groups dedicated to discussions about using RestSharp, feature proposals and similar topics. It is perfectly fine to ask questions about using RestSharp at that group too.

我们有一个谷歌邮箱，专门讨论使用RestSharp、功能建议和类似主题的小组。在该组中也可以询问有关使用RestSharp的问题。

#Bugs and issues

#错误和问题

The last resort to get help when you experience some unexpected behaviour, a crash or anything else that you consider a bug, is submitting an issue at our GitHub repository.

当您遇到一些意外行为、崩溃或任何其他您认为是错误的情况时，获得帮助的最后手段是在我们的GitHub存储库中提交问题。

WARNING

警告

Do not ignore our contribution guidelines, otherwise you risk your issue to be closed without being considered. Respect the maintainers, be specific and provide as many details about the issue as you can.

不要忽视我们的贡献指南，否则您的问题可能会在未经考虑的情况下被关闭。尊重维护人员，做到具体，并尽可能提供有关问题的详细信息。

Ensure you provide the following in the issue:

确保在问题中提供以下内容：

- Expected behaviour
- 预期行为
- Actual behaviour
- 实际行为
- Why do you think it is an issue, not a misunderstanding
- 你为什么认为这是一个问题，而不是误解
- How the issue can be reproduced: a repository or at least a code snippet
- 如何重现问题：存储库或至少是代码片段
- If RestSharp unexpectedly throws an exception, provide the stack trace
- 如果RestSharp意外抛出异常，请提供堆栈跟踪

#Contributing

#贡献

Although issues are considered as contributions, we strongly suggest helping the community by solving issues that you experienced by submitting a pull request.

尽管问题被视为贡献，但我们强烈建议通过解决提交拉取请求时遇到的问题来帮助社区。

Here are contribution guidelines:

以下是代码贡献指南：

- Make each pull request atomic and exclusive; don't send pull requests for a laundry list of changes.
- 使每个拉取请求具有原子性和排他性；不要发送更改列表的拉取请求。
- Even better, commit in small manageable chunks.
- 优先，以可管理的小块提交。
- Use the supplied `.editorconfig` file to format the code.
- 使用提供的`.editorconfig`文件以格式化代码。
- Any change must be accompanied by a unit test covering the change.
- 任何变更都必须附有涵盖变更的单元测试。
- New tests are preferred to use FluentAssertions.
- 新的测试优选使用FluentAssertions。
- No regions.
- 没有区域。
- No licence header for tested.
- 未测试许可证标题。
- Code must build for .NET Standard 2.0, .NET 5, and .NET 6.
- 代码必须为.NET Standard 2.0、.NET 5和.NET 6构建。
- Test must run on .NET 6.
- 测试必须在.NET 6上运行。
- Use `autocrlf=true` (`git config --global core.autocrlf true`)
- 使用`autocrlf=true` (`git config --global core.autocrlf true`)

#Sponsor

#赞助商

You can also support maintainers and motivate them by contributing financially at Open Collectiveopen in new window.

您还可以在打开Collectiveopen的新窗口中上为维护人员提供支持，并通过提供资金激励他们.

#Common issues

#共同问题

Before opening an issue on GitHub, please check the list of known issues below.

在GitHub上打开问题之前，请查看下面的已知问题列表。

#Content type

#内容类型

One of the mistakes developers make when using RestSharp is setting the `Content-Type` header manually. Remember that in most of the usual scenarios setting the content type header manually is not required, and it might be harmful.

开发人员在使用RestSharp时犯的一个错误是手动设置`Content-Type`头。请记住，在大多数常见场景中，不需要手动设置内容类型标头，这可能是有害的。

RestSharp sets the content type header automatically based on the request type. You might want to override the request body content type, but the best way to do it is to supply the content type to the body parameter itself. Functions for adding the request body to the request have overloads, which accept content type. For example

RestSharp根据请求类型自动设置内容类型标头。您可能希望重写请求正文内容类型，但最好的方法是将内容类型提供给正文参数本身。用于将请求主体添加到请求的函数具有接受内容类型的重载。例如

```
request.AddStringBody(jsonString, ContentType.Json);
```

#Setting the User Agent

#设置用户代理

Setting the user agent on the request won't work when you use `AddHeader`.

使用`AddHeader`时，在请求上设置用户代理将不起作用。

Instead, please use the `RestClientOptions.UserAgent` property.

相反地，请使用`RestClientOptions.UserAgent`属性。

#Empty response

#空响应

We regularly get issues where developers complain that their requests get executed and they get a proper raw response, but the `RestResponse<T>` instance doesn't have a deserialized object set.

我们经常遇到这样的问题：开发人员抱怨他们的请求被执行了，他们得到了正确的原始响应，但`RestResponse<T>`实例没有反序列化的对象集。

In other situations, the raw response is also empty.

在其他情况下，原始响应也是空的。

All those issues are caused by the design choice to swallow exceptions that occur when RestSharp makes the request and processes the response. Instead, RestSharp produces so-called *error response*.

所有这些问题都是由于RestSharp发出请求并处理响应时出现的吞咽异常的设计选择造成的。相反，RestSharp生成所谓的*错误响应*。

You can check the response status to find out if there are any errors. The following properties can tell you about those errors:

您可以检查响应状态以确定是否存在任何错误。以下属性可以告诉您这些错误：

- `IsSuccessful`
- `ResponseStatus`
- `StatusCode`
- `ErrorMessage`
- `ErrorException`

It could be that the request was executed and you got `200 OK` status code back and some content, but the typed `Data` property is empty.

可能是请求已执行，您得到了`200 OK`状态代码和一些内容，但类型化的`Data`属性为空。

In that case, you probably got deserialization issues. By default, RestSharp will just return an empty (`null`) result in the `Data` property. Deserialization errors can be also populated to the error response. To do that, set the `client.FailOnDeserializationError` property to `true`.

在这种情况下，您可能会遇到反序列化问题。默认情况下，RestSharp只会在`Data`属性中返回空（`null`）结果。反序列化错误也可以填充到错误响应中。为此，请将`client.FailOnDeserializationError`属性设置为`true`。

It is also possible to force RestSharp to throw an exception.

也可以强制RestSharp抛出异常。

If you set `client.ThrowOnDeserializationError`, RestSharp will throw a `DeserializationException` when the serializer throws. The exception has the internal exception and the response.

如果你设置了`client.ThrowOnDeserializationError`,，RestSharp将在序列化程序抛出时抛出`DeserializationException`。

Finally, by setting `ThrowOnAnyError` you can force RestSharp to re-throw any exception that happens when making the request and processing the response.

最后，通过设置`ThrowOnAnyError`，您可以强制RestSharp重新抛出发出请求和处理响应时发生的任何异常。