

入门指南

欢迎来到 Sisk 文档！

最后，什么是 Sisk Framework？它是一个开源的轻量级库，使用 .NET 构建，旨在成为最小化、灵活和抽象的。它允许开发人员快速创建互联网服务，几乎不需要任何配置。Sisk 使您的现有应用程序能够拥有一个托管的 HTTP 模块，完整且可丢弃。

Sisk 的价值观包括代码透明度、模块化、性能和可扩展性，并且可以处理各种类型的应用程序，例如 Restful、JSON-RPC、Web-sockets 等。

其主要功能包括：

资源	描述
路由	支持前缀、自定义方法、路径变量、值转换器等的路径路由器。
请求处理器	也称为中间件，提供一个接口来构建自己的请求处理器，与请求之前或之后的操作一起工作。
压缩	使用 Sisk 轻松压缩响应内容。
Web sockets	提供接受完整 Web sockets 的路由，用于读取和写入客户端。
服务器发送事件	提供向支持 SSE 协议的客户端发送服务器事件的功能。
日志记录	简化日志记录。记录错误、访问、定义按大小轮换日志、同一日志的多个输出流等。
多主机	为多个端口创建 HTTP 服务器，每个端口都有自己的路由器，每个路由器都有自己的应用程序。
服务器处理器	扩展您自己的 HTTP 服务器实现。使用扩展、改进和新功能进行自定义。

第一步

Sisk 可以在任何 .NET 环境中运行。在本指南中，我们将教您如何使用 .NET 创建 Sisk 应用程序。如果您尚未安装它，请从 [这里](#) 下载 SDK。

在本教程中，我们将介绍如何创建项目结构、接收请求、获取 URL 参数和发送响应。本指南将重点介绍使用 C# 构建一个简单的服务器。您也可以使用您喜欢的编程语言。

NOTE

您可能对快速入门项目感兴趣。请查看 [此存储库](#) 以获取更多信息。

创建项目

让我们将项目命名为"My Sisk Application"。一旦您设置了 .NET，您可以使用以下命令创建项目：

```
dotnet new console -n my-sisk-application
```

接下来，导航到您的项目目录，并使用 .NET 实用工具安装 Sisk：

```
1 cd my-sisk-application  
2 dotnet add package Sisk.HttpServer
```

您可以在 [这里](#) 找到在项目中安装 Sisk 的其他方法。

现在，让我们创建一个 HTTP 服务器实例。对于这个示例，我们将配置它以监听端口 5000。

构建 HTTP 服务器

Sisk 允许您一步一步地手动构建应用程序，因为它路由到 `HttpServer` 对象。然而，这可能对于大多数项目来说并不方便。因此，我们可以使用构建器方法，它使得让应用程序启动变得更容易。

Program.cs

C#

```
1 class Program  
2 {  
3     static async Task Main(string[] args)  
4     {  
5         using var app = HttpServer.CreateBuilder()
```

```
6     .UseListeningPort("http://localhost:5000/")
7     .Build();
8
9     app.Router.MapGet("/", request =>
10    {
11        return new HttpResponseMessage()
12        {
13            Status = 200,
14            Content = new StringContent("Hello, world!")
15        };
16    });
17
18    await app.StartAsync();
19 }
20 }
```

了解 Sisk 的每个重要组件至关重要。稍后在本文档中，您将了解更多关于 Sisk 工作原理的信息。

手动（高级）设置

您可以在文档的 [此部分](#) 中了解每个 Sisk 机制的工作原理，它解释了 HttpServer、Router、ListeningPort 和其他组件之间的行为和关系。

安装

您可以通过 Nuget、dotnet cli 或 [其他选项](#) 安装 Sisk。您可以通过在开发者控制台中运行以下命令轻松设置 Sisk 环境：

```
dotnet add package Sisk.HttpServer
```

此命令将在您的项目中安装 Sisk 的最新版本。

Native AOT 支持

[.NET Native AOT](#) 允许发布本机 .NET 应用程序，这些应用程序是自给自足的，不需要在目标主机上安装 .NET 运行时。另外，Native AOT 提供诸如：

- 应用程序大小大大减小
- 初始化速度大大提高
- 内存消耗降低

Sisk Framework 本质上允许几乎所有功能使用 Native AOT，而无需对源代码进行改造以适应 Native AOT。

不支持的功能

然而，Sisk 使用反射，尽管很少，为某些功能提供支持。下面提到的功能可能在本机代码执行期间部分可用或完全不可用：

- [自动扫描模块](#) 的路由器：此资源扫描执行程序集中的嵌入类型，并注册符合 [路由器模块](#) 的类型。此资源需要可以在程序集修剪期间排除的类型。

Sisk 中的所有其他功能都与 AOT 兼容。通常会找到一个或多个方法，这些方法会产生 AOT 警告，但是相同的方法，如果没有在此处提及，具有重载，可以传递类型、参数或类型信息，以帮助 AOT 编译器编译对象。

部署 Sisk 应用

部署 Sisk 应用的过程包括将您的项目发布到生产环境中。虽然这个过程相对简单，但有一些细节需要注意，以避免对部署的基础设施造成安全和稳定性的问题。

理想情况下，在进行所有可能的测试后，您应该准备好将应用程序部署到云端。

发布您的应用

发布 Sisk 应用或服务是生成适合生产环境的二进制文件。在这个例子中，我们将编译二进制文件以便在安装了 .NET Runtime 的机器上运行。

您需要在机器上安装 .NET SDK 来构建应用程序，并在目标服务器上安装 .NET Runtime 来运行应用程序。您可以在[这里](#)学习如何在 Linux 服务器上安装 .NET Runtime，[这里](#)学习如何在 Windows 上安装，[这里](#)学习如何在 Mac OS 上安装。

在项目所在的文件夹中，打开终端并使用 .NET 发布命令：

```
$ dotnet publish -r linux-x64 -c Release
```

这将在 bin/Release/publish/linux-x64 中生成二进制文件。

NOTE

如果您的应用程序使用 Sisk.ServiceProvider 包，您应该将 service-config.json 文件复制到主机服务器中，连同 dotnet publish 生成的所有二进制文件。您可以预先配置文件，包括环境变量、监听端口和主机，以及其他服务器配置。

下一步是将这些文件传输到将要托管应用程序的服务器。

之后，给二进制文件授予执行权限。假设我们的项目名称是 "my-app"：

```
1  $ cd /home/htdocs
2  $ chmod +x my-app
3  $ ./my-app
```

运行应用程序后，检查是否有任何错误消息。如果没有产生错误消息，那么您的应用程序正在运行。

此时，应用程序可能无法从外部网络访问，因为尚未配置访问规则，例如防火墙。我们将在下一步中考虑这一点。

您应该有应用程序监听的虚拟主机地址。这是手动在应用程序中设置的，并取决于您如何实例化 Sisk 服务。

如果您 **不** 使用 Sisk.ServiceProvider 包，您应该在定义 HttpServer 实例的地方找到它：

```
1  HttpServer server = HttpServer.Emit(5000, out HttpServerConfiguration config, out var host,
2  out var router);
// sisk 应该监听 http://localhost:5000/
```

手动关联 ListeningHost：

```
config.ListeningHosts.Add(new ListeningHost("https://localhost:5000/", router));
```

或者，如果您使用 Sisk.ServiceProvider 包，在您的 service-config.json 中：

```
1  {
2      "Server": { },
3      "ListeningHost": {
4          "Ports": [
5              "http://localhost:5000/"
6          ]
7      }
8  }
```

从这里，我们可以创建一个反向代理来监听您的服务并使流量在开放网络上可用。

代理您的应用

代理您的服务意味着不直接将 Sisk 服务暴露在外部网络中。这是一种常见的服务器部署做法，因为：

- 允许您在应用程序中关联 SSL 证书；
- 创建访问规则以避免过载；
- 控制带宽和请求限制；
- 为您的应用程序分离负载均衡器；
- 防止基础设施故障造成的安全损害。

您可以通过反向代理服务器，如 [Nginx](#) 或 [Apache](#)，或使用 HTTP-over-DNS 隧道，如 [Cloudflare](#)，来提供您的应用程序。

另外，请记得正确解析代理的转发头，以便通过 [转发解析器](#) 获取客户端信息，例如 IP 地址和主机。

创建隧道、配置防火墙并运行应用程序后，下一步是为您的应用程序创建一个服务。

NOTE

在非 Windows 系统上，直接在 Sisk 服务中使用 SSL 证书是不可能的。这是 HttpListener 的实现方式，这是 Sisk 中 HTTP 队列管理的核心模块，这种实现方式在操作系统之间有所不同。如果您 [将证书与 IIS 中的虚拟主机关联](#)，则可以在 Sisk 服务中使用 SSL。对于其他系统，强烈推荐使用反向代理。

创建服务

创建服务将使您的应用程序始终可用，即使在服务器实例重启或发生不可恢复的崩溃后。

在这个简单的教程中，我们将使用前一个教程的内容作为示例，以保持服务始终活跃。

1. 访问服务配置文件所在的文件夹：

```
cd /etc/systemd/system
```

2. 创建您的 `my-app.service` 文件并包含以下内容：

my-app.service

INI

```
1 [Unit]
2 Description=<description about your app>
3
4 [Service]
5 # 设置将启动服务的用户
6 User=<user which will launch the service>
7
8 # ExecStart 路径不是相对于 WorkingDirectory 的。
9 # 将其设置为可执行文件的完整路径
10 WorkingDirectory=/home/htdocs
11 ExecStart=/home/htdocs/my-app
12
13 # 设置服务在崩溃后始终重启
```

```
14  Restart=always
15  RestartSec=3
16
17  [Install]
18  WantedBy=multi-user.target
```

3. 重启服务管理器模块：

```
$ sudo systemctl daemon-reload
```

4. 从文件名启动新创建的服务并检查是否正在运行：

```
1  $ sudo systemctl start my-app
2  $ sudo systemctl status my-app
```

5. 现在，如果您的应用程序正在运行 ("Active: active")，请启用服务以便在系统重启后继续运行：

```
$ sudo systemctl enable my-app
```

现在，您已经准备好向所有人展示您的 Sisk 应用。

使用 SSL

在开发中使用 SSL 可能是必要的，尤其是在需要安全的环境中，例如大多数 Web 开发场景。Sisk 构建在 `HttpListener` 之上，`HttpListener` 不支持原生的 HTTPS，只支持 HTTP。然而，有一些变通方法可以让你在 Sisk 中使用 SSL。请参见以下内容：

通过 IIS 在 Windows 上

- 可用平台：Windows
- 工作量：中等

如果你在 Windows 上，你可以使用 IIS 来启用 HTTP 服务器的 SSL。为了使其生效，建议你先按照 [此教程](#) 进行操作，如果你想让你的应用程序监听其他主机而不是 "localhost"。

要使其生效，你必须通过 Windows 功能安装 IIS。IIS 对 Windows 和 Windows Server 用户免费。要在你的应用程序中配置 SSL，请准备好 SSL 证书，即使它是自签名的。接下来，你可以查看 [如何在 IIS 7 或更高版本上设置 SSL](#)。

通过 mitmproxy

- 可用平台：Linux、macOS、Windows
- 工作量：简单

mitmproxy 是一个拦截代理工具，允许开发人员和安全测试人员检查、修改和记录客户端（例如 Web 浏览器）和服务器之间的 HTTP 和 HTTPS 流量。你可以使用 **mitmdump** 实用程序在客户端和 Sisk 应用程序之间启动反向 SSL 代理。

1. 首先，在你的机器上安装 [mitmproxy](#)。
2. 启动你的 Sisk 应用程序。对于这个例子，我们将使用 8000 端口作为不安全的 HTTP 端口。
3. 启动 mitmproxy 服务器以监听安全端口 8001：

```
mitmdump --mode reverse:http://localhost:8000/ -p 8001
```

就这样！你可以通过 `https://localhost:8001/` 访问你的应用程序。你的应用程序不需要运行才能启动 `mitmdump`。

或者，你可以在你的项目中添加对 [mitmproxy 帮助程序](#) 的引用。这仍然需要在你的计算机上安装 mitmproxy。

通过 Sisk.SslProxy 包

- 可用平台：Linux、macOS、Windows
- 工作量：简单

Sisk.SslProxy 包是启用 Sisk 应用程序 SSL 的一种简单方法。然而，它是一个 **非常实验性** 的包。使用这个包可能会不稳定，但你可以成为少数人中的一员，他们将为使这个包可行和稳定做出贡献。要开始使用，你可以安装 Sisk.SslProxy 包：

```
dotnet add package Sisk.SslProxy
```

NOTE

你必须在 Visual Studio 包管理器中启用 "启用预发布包" 来安装 Sisk.SslProxy。

再次提醒，这是一个实验项目，所以不要考虑将其投入生产。

目前，Sisk.SslProxy 可以处理大多数 HTTP/1.1 功能，包括 HTTP Continue、Chunked-Encoding、WebSockets 和 SSE。请参阅 [这里](#) 了解更多关于 SslProxy 的信息。

在 Windows 上配置命名空间预留

Sisk 与 HttpListener 网络接口一起工作，将虚拟主机绑定到系统以侦听请求。

在 Windows 上，此绑定有一些限制，只允许将 localhost 绑定为有效主机。当尝试侦听另一个主机时，服务器会抛出访问被拒绝错误。此教程解释了如何授予在系统上侦听任何主机的授权。

Namespace Setup.bat

BATCH

```
1 @echo off
2
3 :: 在这里插入前缀，不要包含空格或引号
4 SET PREFIX=
5
6 SET DOMAIN=%ComputerName%\%USERNAME%
7 netsh http add urlacl url=%PREFIX% user=%DOMAIN%
8
9 pause
```

在 PREFIX 中，是服务器将要侦听的前缀（“侦听主机→端口”）。它必须以 URL 方案、主机、端口和末尾斜杠的格式编写，例如：

Namespace Setup.bat

BATCH

```
SET PREFIX=http://my-application.example.test/
```

这样，您就可以通过以下方式在应用程序中侦听：

Program.cs

C#

```
1 class Program
2 {
3     static async Task Main(string[] args)
4     {
5         using var app = HttpServer.CreateBuilder()
6             .UseListeningPort("http://my-application.example.test/")
7             .Build();
8
9         app.Router.MapGet("/", request =>
10        {
```

```
11     return new HttpResponseMessage()
12     {
13         Status = 200,
14         Content = new StringContent("Hello, world!")
15     };
16 });
17
18     await app.StartAsync();
19 }
20 }
```

变更日志

每对 Sisk 进行的更改都会通过变更日志记录。你可以在 [这里](#) 查看所有 Sisk 版本的变更日志。

常见问题

关于 Sisk 的常见问题。

Sisk 是开源的吗？

完全开源。Sisk 使用的所有源代码都已发布并经常更新在 [GitHub](#) 上。

是否接受贡献？

只要贡献符合 [Sisk 哲学](#)，所有贡献都非常欢迎！贡献不仅限于代码！您可以通过文档、测试、翻译、捐赠和帖子等方式贡献。

Sisk 是否有资金支持？

不。目前没有任何组织或项目为 Sisk 提供资金支持。

我可以在生产环境中使用 Sisk 吗？

绝对可以。该项目已经开发超过三年，并在商业应用中进行了大量测试，这些应用已经投入生产。Sisk 被用于重要的商业项目作为主要基础设施。

已经编写并提供了在不同系统和环境中 [部署](#) 的指南。

Sisk 是否具有身份验证、监控和数据库服务？

不。Sisk 不具有这些功能。它是一个用于开发 HTTP 网络应用程序的框架，但它仍然是一个最小的框架，仅提供应用程序运行所需的功能。

您可以使用任何第三方库来实现所需的所有服务。Sisk 被设计为无关、灵活和与任何东西一起工作。

为什么我应该使用 Sisk 而不是 <框架>？

我不知道。您告诉我。

Sisk 被创建以填补 .NET 中 HTTP 网络应用程序的通用场景。已建立的项目，例如 ASP.NET，解决了各种问题，但具有不同的偏见。与较大的框架不同，Sisk 需要用户了解他们正在做什么和构建什么。基本的 Web 开发和 HTTP 协议知识对于使用 Sisk 至关重要。

Sisk 更接近 Node.js 的 Express，而不是 ASP.NET Core。它是一个高级抽象，允许您创建具有所需 HTTP 逻辑的应用程序。

我需要学习什么才能使用 Sisk？

您需要了解：

- Web 开发 (HTTP、Restful 等)
- .NET

仅此而已。拥有这两个主题的基本知识，您可以在几个小时内使用 Sisk 开发一个高级应用程序。

我可以使用 Sisk 开发商业应用程序吗？

绝对可以。

Sisk 是在 MIT 许可证下创建的，这意味着您可以在任何商业项目中使用 Sisk，无论是商业还是非商业，均无需专有许可证。

我们要求的是，在您的应用程序中，您需要在某个地方添加一个关于使用的开源项目的通知，并且 Sisk 在其中。

路由

Router 是构建服务器的第一步。它负责存储 Route 对象，这些对象是将 URL 和其方法映射到服务器执行的操作的端点。每个操作负责接收请求并将响应发送回客户端。

路由是路径表达式（“路径模式”）和它们可以监听的 HTTP 方法的对。当请求发送到服务器时，它将尝试找到匹配接收到的请求的路由，然后调用该路由的操作并将结果响应发送回客户端。

有多种方式在 Sisk 中定义路由：它们可以是静态的、动态的或自动扫描的，通过属性定义或直接在 Router 对象中定义。

```
1 Router mainRouter = new Router();
2
3 // 将 GET / 映射到以下操作
4 mainRouter.MapGet("/", request => {
5     return new HttpResponse("Hello, world!");
6 });
```

要了解路由可以做什么，我们需要了解请求可以做什么。HttpRequest 将包含所有需要的信息。Sisk 还包括一些额外的功能，可以加快整体开发速度。

对于服务器接收到的每个操作，都会调用类型为 RouteAction 的委托。该委托包含一个参数，持有 HttpRequest 对象，该对象包含有关请求的所有必要信息。从该委托返回的对象必须是 HttpResponse 或通过 隐式响应类型 映射到它的对象。

匹配路由

当请求发送到 HTTP 服务器时，Sisk 搜索满足请求路径表达式的路由。该表达式始终在路由和请求路径之间进行测试，而不考虑查询字符串。

此测试没有优先级，并且仅限于单个路由。当没有路由与该请求匹配时，返回 Router.NotFoundErrorHandler 响应给客户端。当路径模式匹配，但 HTTP 方法不匹配时，发送 Router.MethodNotAllowedErrorHandler 响应给客户端。

Sisk 检查路由碰撞的可能性，以避免这些问题。当定义路由时，Sisk 将查找可能与正在定义的路由碰撞的可能路由。该测试包括检查路径和路由设置为接受的方法。

使用路径模式创建路由

您可以使用各种 `SetRoute` 方法定义路由。

```
1 // SetRoute 方式
2 mainRouter.SetRoute(RouteMethod.Get, "/hey/<name>", (request) =>
3 {
4     string name = request.RouteParameters["name"].GetString();
5     return new HttpResponseMessage($"Hello, {name}");
6 });
7
8 // Map* 方式
9 mainRouter.MapGet("/form", (request) =>
10 {
11     var formData = request.GetFormData();
12     return new HttpResponseMessage(); // 空 200 ok
13 });
14
15 // Route.* 帮助方法
16 mainRouter += Route.Get("/image.png", (request) =>
17 {
18     var imageStream = File.OpenRead("image.png");
19
20     return new HttpResponseMessage()
21     {
22         // StreamContent 内部
23         // 流在发送响应后被释放。
24         Content = new StreamContent(imageStream)
25     };
26 });
27
28 // 多个参数
29 mainRouter.MapGet("/hey/<name>/surname/<surname>", (request) =>
30 {
31     string name = request.RouteParameters["name"].GetString();
32     string surname = request.RouteParameters["surname"].GetString();
33
34     return new HttpResponseMessage($"Hello, {name} {surname}!");
35 });

// 多个参数
mainRouter.MapGet("/hey/<name>/surname/<surname>", (request) =>
{
    string name = request.RouteParameters["name"].GetString();
    string surname = request.RouteParameters["surname"].GetString();

    return new HttpResponseMessage($"Hello, {name} {surname}!");
});
```

`RouteParameters` 属性的 `HttpResponse` 包含有关请求路径变量的所有信息。

每个发送到服务器的路径在执行路径模式测试之前都会被规范化，遵循以下规则：

- 所有空段都从路径中删除，例如：`////foo//bar` 变为 `/foo/bar`。
- 路径匹配是 **区分大小写** 的，除非 `Router.MatchRoutesIgnoreCase` 设置为 `true`。

[Query](#) 和 [RouteParameters](#) 属性的 [HttpRequest](#) 返回 [StringValueCollection](#) 对象，其中每个索引属性返回非空 [StringValue](#)，可以用作选项/单子将其原始值转换为托管对象。

以下示例读取路由参数“id”并从中获取 [Guid](#)。如果参数不是有效的 Guid，抛出异常，并在服务器不处理 [Router.CallbackErrorHandler](#) 时返回 500 错误给客户端。

```
1 mainRouter.SetRoute(RouteMethod.Get, "/user/<id>", (request) =>
2 {
3     Guid id = request.RouteParameters["id"].GetGuid();
4 });
```

NOTE

路径的尾部 / 在请求路径和路由路径中都被忽略，即，如果您尝试访问定义为 /index/page 的路由，您也可以使用 /index/page/ 访问它。

您还可以通过启用 [ForceTrailingSlash](#) 标志强制 URL 以 / 结尾。

使用类实例创建路由

您还可以使用反射和 [RouteAttribute](#) 属性动态定义路由。这样，具有此属性的类的实例将在目标路由器中定义其路由。

要将方法定义为路由，它必须用 [RouteAttribute](#) 标记，例如该属性本身或 [RouteGetAttribute](#)。该方法可以是静态的、实例的、公共的或私有的。当使用 [SetObject\(type\)](#) 或 [SetObject<TType>\(\)](#) 方法时，实例方法将被忽略。

Controller/MyController.cs

C#

```
1 public class MyController
2 {
3     // 将匹配 GET /
4     [RouteGet]
5     HttpResponse Index(HttpRequest request)
6     {
7         HttpResponse res = new HttpResponse();
8         res.Content = new StringContent("Index!");
9         return res;
10    }
11
12    // 静态方法也可以
13    [RouteGet("/hello")]
14    static HttpResponse Hello(HttpRequest request)
15    {
```

```
16     HttpResponseMessage res = new HttpResponseMessage();
17     res.Content = new StringContent("Hello world!");
18     return res;
19 }
20 }
```

以下行将定义 MyController 的 Index 和 Hello 方法作为路由，因为它们都被标记为路由，并且提供了类的实例，而不是其类型。如果提供的是其类型而不是实例，则仅定义静态方法。

```
1 var myController = new MyController();
2 mainRouter.SetObject(myController);
```

从 Sisk 0.16 版开始，可以启用 AutoScan，它将搜索实现 RouterModule 的用户定义类，并将其自动关联到路由器。这不支持 AOT 编译。

```
mainRouter.AutoScanModules<ApiController>();
```

上述指令将搜索所有实现 ApiController 的类型，但不包括该类型本身。两个可选参数指示方法将如何搜索这些类型。第一个参数表示将在其中搜索类型的程序集，第二个参数表示定义类型的方式。

正则路由

您可以将路由标记为使用正则表达式进行解释，而不是使用默认的 HTTP 路径匹配方法。

```
1 Route indexRoute = new Route(RouteMethod.Get, @"\/[a-z]+\/", "My route", IndexPage, null);
2 indexRoute.UseRegex = true;
3 mainRouter.SetRoute(indexRoute);
```

或者使用 [RegexRoute](#) 类：

```
1 mainRouter.SetRoute(new RegexRoute(RouteMethod.Get, @"\/[a-z]+\/", request =>
2 {
3     return new HttpResponseMessage("hello, world");
4 }));
```

您还可以从正则表达式模式中捕获组到 [HttpRequest.RouteParameters](#) 内容中：

```
1  public class MyController
2  {
3      [RegexRoute(RouteMethod.Get, @"/uploads/(?<filename>.*\.(jpeg|jpg|png))")]
4      static HttpResponseMessage RegexRoute(HttpRequest request)
5      {
6          string filename = request.RouteParameters["filename"].GetString();
7          return new HttpResponseMessage().WithContent($"Acessing file {filename}");
8      }
9 }
```

路由前缀

您可以使用 `RoutePrefix` 属性为类或模块中的所有路由添加前缀，并将前缀设置为字符串。

请参见以下使用 BREAD 体系结构（浏览、读取、编辑、添加和删除）的示例：

```
1  [RoutePrefix("/api/users")]
2  public class UsersController
3  {
4      // GET /api/users/<id>
5      [RouteGet]
6      public async Task<HttpResponse> Browse()
7      {
8          ...
9      }
10
11     // GET /api/users
12     [RouteGet("/{id}")]
13     public async Task<HttpResponse> Read()
14     {
15         ...
16     }
17
18     // PATCH /api/users/<id>
19     [RoutePatch("/{id}")]
20     public async Task<HttpResponse> Edit()
```

```
21     {
22     ...
23 }
24
25     // POST /api/users
26     [RoutePost]
27     public async Task<HttpResponse> Add()
28     {
29     ...
30 }
31
32     // DELETE /api/users/<id>
33     [RouteDelete("/<id>")]
34     public async Task<HttpResponse> Delete()
35     {
36     ...
37 }
38 }
```

在上面的示例中，`HttpResponse` 参数已省略，以便通过全局上下文 `HttpContext.Current` 使用。请参阅下一节以获取更多信息。

无请求参数的路由

路由可以在不需要 `HttpRequest` 参数的情况下定义，并且仍然可以在请求上下文中获取请求及其组件。

让我们考虑一个 `ControllerBase` 抽象，它作为 API 的所有控制器的基础，并且该抽象提供 `Request` 属性来获取当前线程的 `HttpRequest`。

Controller/Base.cs

C#

```
1  public abstract class ControllerBase
2  {
3      // 从当前线程获取请求
4      public HttpRequest Request { get => HttpContext.Current.Request; }
5
6      // 下面的行从当前 HTTP 会话获取数据库，或者如果不存在则创建一个新的。
7      public DbContext Database { get => HttpContext.Current.RequestBag.GetOrAdd<DbContext>
8          () ; }
9 }
```

并且所有其后代都可以使用不带请求参数的路由语法：

Controller/UsersController.cs

C#

```
1 [RoutePrefix("/api/users")]
2 public class UsersController : ControllerBase
3 {
4     [RoutePost]
5     public async Task<HttpResponse> Create()
6     {
7         // 从当前请求读取 JSON 数据
8         UserCreationDto? user = JsonSerializer.DeserializeAsync<UserCreationDto>
9         (Request.Body);
10        ...
11        Database.Users.Add(user);
12
13        return new HttpResponse(201);
14    }
}
```

有关当前上下文和依赖注入的更多详细信息，请参阅 [依赖注入](#) 教程。

任意方法路由

您可以定义一个路由，以便仅通过其路径匹配，并跳过 HTTP 方法。这可以在路由回调内部对方法进行验证时很有用。

```
1 // 将匹配 / 的任何 HTTP 方法
2 mainRouter.SetRoute(RouteMethod.Any, "/", callbackFunction);
```

任意路径路由

任意路径路由将测试从 HTTP 服务器接收的任何路径，并且仅限于路由方法。如果路由方法为 RouteMethod.Any 且路由在其路径表达式中使用 [Route.AnyPath](#)，则此路由将监听来自 HTTP 服务器的所有请求，并且无法定义其他路

由。

```
1 // 下面的路由将匹配所有 POST 请求
2 mainRouter.SetRoute(RouteMethod.Post, Route.AnyPath, callbackFunction);
```

忽略大小写路由匹配

默认情况下，路由与请求的解释是区分大小写的。要使其忽略大小写，请启用此选项：

```
mainRouter.MatchRoutesIgnoreCase = true;
```

这也将为使用正则表达式匹配的路由启用 `RegexOptions.IgnoreCase` 选项。

未找到 (404) 回调处理程序

您可以为请求不匹配任何已知路由时创建自定义回调。

```
1 mainRouter.NotFoundErrorHandler = () =>
2 {
3     return new HttpResponseMessage(404)
4     {
5         // 自 0.14 版以来
6         Content = new HtmlContent("<h1>Not found</h1>")
7         // 旧版本
8         Content = new StringContent("<h1>Not found</h1>", Encoding.UTF8, "text/html")
9     };
10};
```

方法不允许 (405) 回调处理程序

您还可以为请求匹配其路径但不匹配方法时创建自定义回调。

```
1 mainRouter.MethodNotAllowedErrorHandler = (context) =>
2 {
3     return new HttpResponseMessage(405)
4     {
5         Content = new StringContent($"Method not allowed for this route.")
6     };
7 }
```

内部错误处理程序

路由回调可以在服务器执行期间抛出错误。如果不正确处理，可能会终止 HTTP 服务器的整体功能。路由器具有一个回调，当路由回调失败并防止服务中断时将被调用。

此方法仅在 [ThrowExceptions](#) 设置为 `false` 时可访问。

```
1 mainRouter.CallbackErrorHandler = (ex, context) =>
2 {
3     return new HttpResponseMessage(500)
4     {
5         Content = new StringContent($"Error: {ex.Message}")
6     };
7 }
```

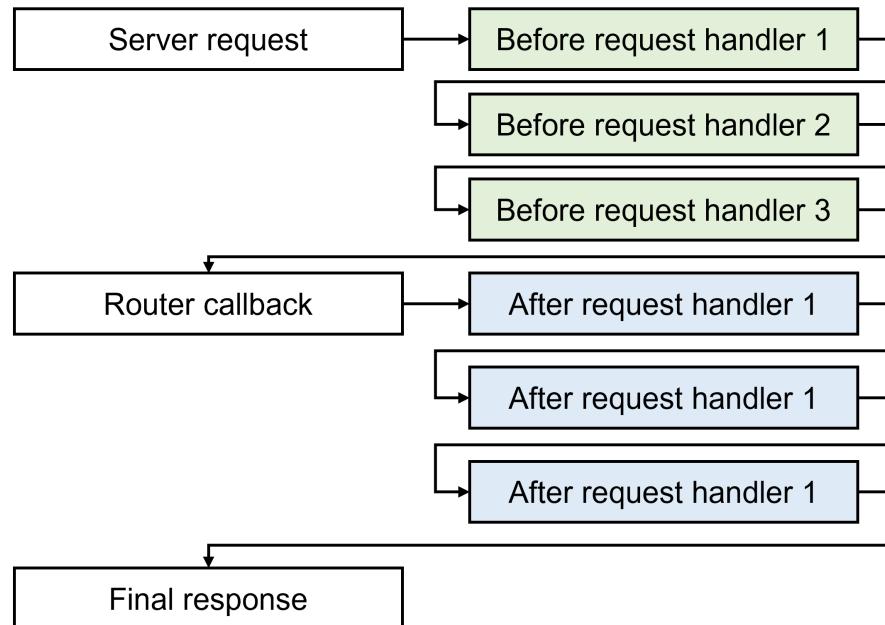
请求处理

请求处理器，也称为“中间件”，是运行在请求在路由器上执行之前或之后的函数。它们可以为每个路由或每个路由器定义。

请求处理器有两种类型：

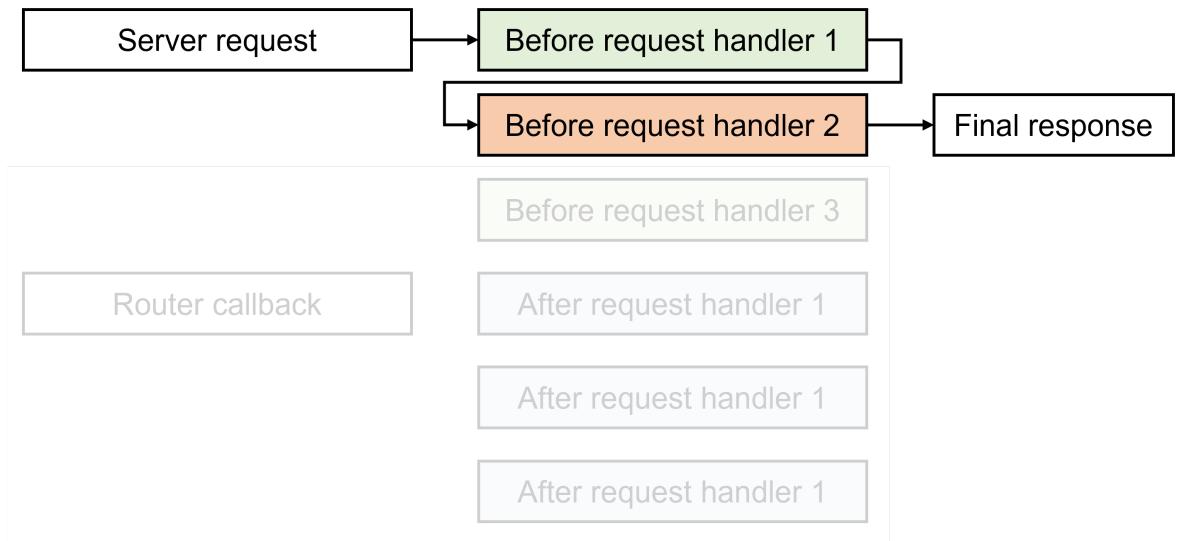
- **BeforeResponse**：定义请求处理器将在调用路由器操作之前执行。
- **AfterResponse**：定义请求处理器将在调用路由器操作之后执行。在此上下文中发送 HTTP 响应将覆盖路由器操作的响应。

两种请求处理器都可以覆盖实际路由器回调函数的响应。另外，请求处理器可以用于验证请求，例如身份验证、内容或其他信息，例如存储信息、日志或可以在响应之前或之后执行的其他步骤。



这样，请求处理器可以中断整个执行过程并在完成周期之前返回响应，丢弃过程中的所有其他内容。

示例：假设用户身份验证请求处理器未能对其进行身份验证。它将防止请求生命周期继续并挂起。如果这发生在第二个请求处理器中，第三个及以后的处理器将不会被评估。



创建请求处理器

要创建请求处理器，我们可以创建一个继承 [IRequestHandler](#) 接口的类，以以下格式：

Middleware/AuthenticateUserRequestHandler.cs

C#

```

1  public class AuthenticateUserRequestHandler : IRequestHandler
2  {
3      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
4          RequestHandlerExecutionMode.BeforeResponse;
5
6      public HttpResponse? Execute(HttpContext context)
7      {
8          if (request.Headers.Authorization != null)
9          {
10              // 返回 null 表示请求周期可以继续
11              return null;
12          }
13          else
14          {
15              // 返回 HttpResponse 对象表示此响应将覆盖相邻的响应。
16              return new HttpResponse(System.Net.HttpStatusCode.Unauthorized);

```

```
17         }
18     }
}
```

在上面的示例中，我们指示如果请求中存在 `Authorization` 标头，则应继续并调用下一个请求处理器或路由器回调，否则将返回未经授权的响应。

每当请求处理器返回 `null` 时，表示请求必须继续并调用下一个对象或以路由器的响应结束周期。

将请求处理器与单个路由关联

您可以为路由定义一个或多个请求处理器。

Router.cs

C#

```
1 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage, "", new IRequestHandler[]
2 {
3     new AuthenticateUserRequestHandler(),           // before request handler
4     new ValidateJsonContentRequestHandler(),        // before request handler
5     //                                               -- 方法 IndexPage 将在此处执行
6     new WriteToLogRequestHandler()                 // after request handler
7});
```

或者创建一个 `Route` 对象：

Router.cs

C#

```
1 Route indexRoute = new Route(RouteMethod.Get, "/", "", IndexPage, null);
2 indexRoute.RequestHandlers = new IRequestHandler[]
3 {
4     new AuthenticateUserRequestHandler()
5 };
6 mainRouter.SetRoute(indexRoute);
```

将请求处理器与路由器关联

您可以定义一个全局请求处理器，它将在路由器上的所有路由上运行。

Router.cs

C#

```
1 mainRouter.GlobalRequestHandlers = new IRequestHandler[]
2 {
3     new AuthenticateUserRequestHandler()
4 };
```

将请求处理器与属性关联

您可以将请求处理器定义为方法属性，连同路由属性。

Controller/MyController.cs

C#

```
1 public class MyController
2 {
3     [RouteGet("/")]
4     [RequestHandler<AuthenticateUserRequestHandler>]
5     static HttpResponse Index(HttpServletRequest request)
6     {
7         return new HttpResponse() {
8             Content = new StringContent("Hello world!")
9         };
10    }
11 }
```

请注意，需要传递所需的请求处理器类型，而不是对象实例。这样，请求处理器将由路由器解析器实例化。您可以用 [ConstructorArguments](#) 属性传递类构造函数的参数。

示例：

Controller/MyController.cs

C#

```
1 [RequestHandler<AuthenticateUserRequestHandler>("arg1", 123, ...)]
2 public HttpResponse Index(HttpServletRequest request)
3 {
4     return res = new HttpResponse() {
5         Content = new StringContent("Hello world!")
6     };
7 }
```

您还可以创建自己的属性，它实现了 RequestHandler：

Middleware/Attributes/AuthenticateAttribute.cs

C#

```
1 public class AuthenticateAttribute : RequestHandlerAttribute
2 {
3     public AuthenticateAttribute() : base(typeof(AuthenticateUserRequestHandler),
4 ConstructorArguments = new object?[] { "arg1", 123, ... })
5     {
6     ;
7     }
}
```

并将其用作：

Controller/MyController.cs

C#

```
1 [Authenticate]
2 static HttpResponse Index(HttpServletRequest request)
3 {
4     return res = new HttpResponse() {
5         Content = new StringContent("Hello world!")
6     };
7 }
```

跳过全局请求处理器

在路由器上定义全局请求处理器后，您可以在特定路由上忽略此请求处理器。

Router.cs

C#

```
1 var myRequestHandler = new AuthenticateUserRequestHandler();
2 mainRouter.GlobalRequestHandlers = new IRequestHandler[]
3 {
4     myRequestHandler
5 };
6
7 mainRouter.SetRoute(new Route(RouteMethod.Get, "/", "My route", IndexPage, null)
8 {
9     BypassGlobalRequestHandlers = new IRequestHandler[]
10 {
11         myRequestHandler, // ok: 与全局请求处理器中相同的实例
12         new AuthenticateUserRequestHandler() // wrong: 不会跳过全局请求处理器
13     }
14 });
15 );
```

NOTE

如果您要跳过请求处理器，则必须使用与之前实例化的相同的引用跳过。创建另一个请求处理器实例将不会跳过全局请求处理器，因为其引用将更改。请记住在 GlobalRequestHandlers 和 BypassGlobalRequestHandlers 中使用相同的请求处理器引用。

请求

请求是表示 HTTP 请求消息的结构体。`[HttpRequest] (/api/Sisk.Core.Http.HttpRequest)` 对象包含在整个应用程序中处理 HTTP 消息的有用函数。

HTTP 请求由方法、路径、版本、头部和正文组成。

在本文档中，我们将教你如何获取这些元素。

获取请求方法

要获取收到请求的方法，可以使用 `Method` 属性：

```
1 static HttpResponse Index(HttpServletRequest request)
2 {
3     HttpMethod requestMethod = request.Method;
4     ...
5 }
```

此属性返回一个由 `HttpMethod` 对象表示的请求方法。

NOTE

与路由方法不同，此属性不服务于 `RouteMethod.Any` 项。相反，它返回真实的请求方法。

获取请求 URL 组件

你可以通过请求的某些属性获取 URL 的各种组件。以下 URL 为例：

```
http://localhost:5000/user/login?email=foo@bar.com
```

组件名称	描述	组件值
Path	获取请求路径。	/user/login
FullPath	获取请求路径和查询字符串。	/user/login?email=foo@bar.com
FullUrl	获取完整的 URL 请求字符串。	http://localhost:5000/user/login? email=foo@bar.com
Host	获取请求主机。	localhost
Authority	获取请求主机和端口。	localhost:5000
QueryString	获取请求查询。	?email=foo@bar.com
Query	获取请求查询的命名值集合。	{StringValueCollection object}
IsSecure	判断请求是否使用 SSL (true) 或不使用 (false)。	false

你也可以使用 [HttpRequest.Uri](#) 属性，它将上述所有内容包含在一个对象中。

获取请求正文

某些请求包含正文，例如表单、文件或 API 事务。你可以通过属性获取请求正文：

```

1 // 以字符串形式获取请求正文，使用请求编码作为编码器
2 string body = request.Body;
3
4 // 或以字节数组获取
5 byte[] bodyBytes = request.RawBody;
6
7 // 或者，你可以流式读取。
8 Stream requestStream = request.GetRequestStream();

```

还可以通过属性 [HasContents](#) 判断请求是否有正文，以及通过 [IsContentAvailable](#) 判断 HTTP 服务器是否已完全接收来自远程点的内容。

不能通过 [GetRequestStream](#) 多次读取请求内容。如果使用此方法读取，[RawBody](#) 和 [Body](#) 中的值也将不可用。在请求上下文中不需要显式释放请求流，它会在创建它的 HTTP 会话结束时被释放。你还可以使用 [HttpRequest.RequestEncoding](#) 属性获取最佳编码来手动解码请求。

服务器对读取请求内容有限制，适用于 `HttpRequest.Body` 和 `HttpRequest.RawBody`。这些属性会将整个输入流复制到一个与 `HttpRequest.ContentLength` 相同大小的本地缓冲区。

如果发送的内容大于用户配置中定义的 `HttpServerConfiguration.MaximumContentSize`，服务器会返回状态码 413 Content Too Large 的响应。除此之外，如果未配置限制或限制过大，服务器在客户端发送的内容超过 `Int32.MaxValue` (2 GB) 并尝试通过上述属性访问内容时，将抛出 `OutOfMemoryException`。你仍然可以通过流式处理来处理内容。

NOTE

虽然 Sisk 允许这样做，但始终建议遵循 HTTP 语义来创建你的应用程序，并且不要在不允许的方法中获取或提供内容。阅读关于 [RFC 9110 "HTTP Semantics"](#) 的内容。

获取请求上下文

HTTP Context 是一个专属的 Sisk 对象，用于存储 HTTP 服务器、路由、路由器和请求处理程序信息。你可以使用它来在这些对象难以组织的环境中进行组织。

`RequestBag` 对象包含从请求处理程序传递到另一个点的存储信息，并可在最终目的地消费。此对象也可被在路由回调后运行的请求处理程序使用。

TIP

此属性也可通过 `HttpRequest.Bag` 属性访问。

Middleware/AuthenticateUserRequestHandler.cs

C#

```
1  public class AuthenticateUserRequestHandler : IRequestHandler
2  {
3      public string Identifier { get; init; } = Guid.NewGuid().ToString();
4      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
5          RequestHandlerExecutionMode.BeforeResponse;
6
7      public HttpResponse? Execute(HttpRequest request, HttpContext context)
8      {
9          if (request.Headers.Authorization != null)
10         {
11             context.RequestBag.Add("AuthenticatedUser", new User("Bob"));
12             return null;
13         }
14     }
15 }
```

```
14         else
15     {
16         return new HttpResponse(System.Net HttpStatusCode.Unauthorized);
17     }
18 }
}
```

上述请求处理程序将在请求包中定义 `AuthenticatedUser`，并可在最终回调中稍后消费：

Controller/MyController.cs

C#

```
1 public class MyController
2 {
3     [RouteGet("/")]
4     [RequestHandler<AuthenticateUserRequestHandler>]
5     static HttpResponse Index(HttpServletRequest request)
6     {
7         User authUser = request.Context.RequestBag["AuthenticatedUser"];
8
9         return new HttpResponse() {
10             Content = new StringContent($"Hello, {authUser.Name}!")
11         };
12     }
13 }
```

你还可以使用 `Bag.Set()` 和 `Bag.Get()` 辅助方法按其类型单例获取或设置对象。

Middleware/Authenticate.cs

C#

```
1 public class Authenticate : RequestHandler
2 {
3     public override HttpResponse? Execute(HttpServletRequest request, HttpContext context)
4     {
5         request.Bag.Set<User>(authUser);
6     }
7 }
```

Controller/MyController.cs

C#

```
1 [RouteGet("/")]
2 [RequestHandler<Authenticate>]
3 public static HttpResponse GetUser(HttpServletRequest request)
4 {
5     var user = request.Bag.Get<User>();
```

```
6     ...
7 }
```

获取表单数据

你可以使用以下示例中的 [NameValueCollection](#) 获取表单数据的值：

Controller/Auth.cs

C#

```
1 [RoutePost("/auth")]
2 public HttpResponse Index(HttpServletRequest request)
3 {
4     var form = request.GetFormContent();
5
6     string? username = form["username"];
7     string? password = form["password"];
8
9     if (AttemptLogin(username, password))
10    {
11        ...
12    }
13 }
```

获取 multipart 表单数据

Sisk 的 HTTP 请求允许你获取上传的 multipart 内容，例如文件、表单字段或任何二进制内容。

Controller/Auth.cs

C#

```
1 [RoutePost("/upload-contents")]
2 public HttpResponse Index(HttpServletRequest request)
3 {
4     // 以下方法将整个请求输入读取到
5     // MultipartObjects 数组中
6     var multipartFormDataObjects = request.GetMultipartFormContent();
```

```
7
8     foreach (MultipartObject uploadedObject in multipartFormDataObjects)
9     {
10         // Multipart 表单数据提供的文件名。
11         // 如果对象不是文件，则返回 null。
12         Console.WriteLine("File name : " + uploadedObject.Filename);
13
14         // Multipart 表单数据对象字段名。
15         Console.WriteLine("Field name : " + uploadedObject.Name);
16
17         // Multipart 表单数据内容长度。
18         Console.WriteLine("Content length : " + uploadedObject.ContentLength);
19
20         // 根据每个已知内容类型的文件头确定图像格式。
21         // 如果内容不是已识别的常见文件格式，此方法将返回 MultipartObjectCommonFormat.Unknown
22         Console.WriteLine("Common format : " + uploadedObject.GetCommonFileFormat());
23     }
24 }
```

你可以阅读更多关于 Sisk [Multipart form objects](#) 及其方法、属性和功能。

检测客户端断开

自 Sisk v1.15 版本起，框架提供了一个 CancellationToken，当客户端与服务器之间的连接在收到响应之前提前关闭时会抛出。此令牌可用于检测客户端不再需要响应并取消长时间运行的操作。

```
1   router.MapGet("/connect", async (HttpRequest req) =>
2   {
3       // 从请求获取断开令牌
4       var dc = req.DisconnectToken;
5
6       await LongOperationAsync(dc);
7
8       return new HttpResponseMessage();
9   });

```

此令牌并不兼容所有 HTTP 引擎，每个都需要实现。

服务器发送事件支持

Sisk 支持 [Server-sent events](#)，允许以流的方式发送块并保持服务器与客户端之间的连接。

调用 `HttpRequest.GetEventSource` 方法将把 `HttpRequest` 放入其监听器状态。从此，HTTP 请求的上下文将不再期望 `HttpResponse`，因为它会覆盖服务器端事件发送的包。

发送所有包后，回调必须返回 `Close` 方法，它将向服务器发送最终响应并指示流已结束。

无法预测将发送的所有包的总长度，因此无法使用 `Content-Length` 标头确定连接结束。

大多数浏览器默认不支持发送除 GET 方法之外的 HTTP 标头或方法。因此，在使用需要特定请求头的事件源请求的请求处理程序时，请小心，因为它们可能不会有这些头。

此外，大多数浏览器在客户端收到所有包后未调用 `EventSource.close` 方法时会重新启动流，导致服务器端无限额外处理。为避免此类问题，通常会发送一个最终包，指示事件源已完成发送所有包。

下面的示例显示浏览器如何与支持服务器端事件的服务器通信。

sse-example.html

HTML

```
1  <html>
2      <body>
3          <b>Fruits:</b>
4          <ul></ul>
5      </body>
6      <script>
7          const evtSource = new EventSource('http://localhost:5555/event-source');
8          const eventList = document.querySelector('ul');
9
10         evtSource.onmessage = (e) => {
11             const newElement = document.createElement("li");
12
13             newElement.textContent = `message: ${e.data}`;
14             eventList.appendChild(newElement);
15
16             if (e.data === "Tomato") {
17                 evtSource.close();
18             }
19         }
20     </script>
21 </html>
```

并逐步向客户端发送消息：

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      [RouteGet("/event-source")]
4      public async Task<HttpResponse> ServerEventsResponse(HttpRequest request)
5      {
6          var sse = await request.GetEventSourceAsync();
7
8          string[] fruits = new[] { "Apple", "Banana", "Watermelon", "Tomato" };
9
10         foreach (string fruit in fruits)
11         {
12             await serverEvents.SendAsync(fruit);
13             await Task.Delay(1500);
14         }
15
16         return serverEvents.Close();
17     }
18 }
```

运行此代码时，我们期望得到类似以下的结果：



Fruits:

解析代理 IP 和主机

Sisk 可以与代理一起使用，因此 IP 地址可能会被代理端点替换，从客户端到代理的事务中。

你可以在 Sisk 中使用 [forwarding resolvers](#) 定义自己的解析器。

头部编码

头部编码可能会成为某些实现的一个问题。在 Windows 上，UTF-8 头部不受支持，因此使用 ASCII。Sisk 内置了编码转换器，可用于解码错误编码的头部。

此操作成本高且默认禁用，但可以在 [NormalizeHeadersEncodings](#) 标志下启用。

响应

响应表示HTTP请求的对象，是服务器发送给客户端的HTTP响应，用来指示请求的资源、页面、文档、文件或其他对象。

一个HTTP响应由状态、头部和内容组成。

在本文档中，我们将教您如何使用Sisk构建HTTP响应。

设置HTTP状态

自HTTP/1.0以来，HTTP状态列表保持不变，Sisk支持所有这些状态。

```
1  HttpResponse res = new HttpResponse();
2  res.Status = System.Net HttpStatusCode.Accepted; //202
```

或使用流畅语法：

```
1  new HttpResponse()
2  .WithStatus(200) // 或
3  .WithStatus(HttpStatusCode.OK) // 或
4  .WithStatus(HttpStatusCodeInformation.OK);
```

您可以在此处[查看](#)可用的`HttpStatusCode`的完整列表。您也可以通过使用`HttpStatusCodeInformation`结构提供自己的状态代码。

正文和内容类型

Sisk支持本地.NET内容对象来发送响应正文。例如，您可以使用[StringContent](#)类发送JSON响应：

```
1  HttpResponseMessage res = new HttpResponseMessage();
2  res.Content = new StringContent(myJson, Encoding.UTF8, "application/json");
```

服务器将始终尝试从您在内容中定义的内容中计算 Content-Length，如果您没有在头部明确定义。如果服务器无法从响应内容中隐含地获取Content-Length头部，响应将使用分块编码发送。

您也可以通过发送`StreamContent`或使用`GetResponseStream`方法来流式传输响应。

响应头部

您可以添加、编辑或删除响应中发送的头部。下面示例演示了如何向客户端发送重定向响应：

```
1  HttpResponseMessage res = new HttpResponseMessage();
2  res.StatusCode = HttpStatusCode.Moved;
3  res.Headers.Add(HttpKnownHeaderNames.Location, "/login");
```

或使用流畅语法：

```
1  new HttpResponseMessage(301)
2  .WithHeader("Location", "/login");
```

当您使用`HttpHeaderCollection`的`Add`方法时，您正在添加一个头部到请求中，而不改变已经发送的头部。`Set`方法用指定的值替换具有相同名称的头部。`HttpHeaderCollection`的索引器内部调用`Set`方法来替换头部。

发送Cookie

Sisk有方法方便地在客户端定义Cookie。通过此方法设置的Cookie已经URL编码并符合RFC-6265标准。

```
1  HttpResponseMessage res = new HttpResponseMessage();
2  res.SetCookie("cookie-name", "cookie-value");
```

或使用流畅语法：

```
1 new HttpResponse(301)
2 .WithCookie("cookie-name", "cookie-value", expiresAt:
DateTime.Now.Add(TimeSpan.FromDays(7)));
```

还有其他[更完整的版本](#)相同的メソッド。

分块响应

您可以设置传输编码为分块，以发送大响应。

```
1 HttpResponse res = new HttpResponse();
2 res.SendChunked = true;
```

使用分块编码时，Content-Length头部会自动省略。

响应流

响应流是一种托管方式，允许您分段发送响应。这是比使用HttpResponse对象更底层的操作，因为它们需要您手动发送头部和内容，然后关闭连接。

此示例打开文件的只读流，将流复制到响应输出流，并不加载整个文件到内存中。这对于服务中型或大文件非常有用。

```
1 // 获取响应输出流
2 using var fileStream = File.OpenRead("my-big-file.zip");
3 var responseStream = request.GetResponseStream();
4
5 // 设置响应编码使用分块编码
6 // 并且您不应该在分块编码时发送content-length头部
7 responseStream.SendChunked = true;
8 responseStream.SetStatus(200);
9 responseStream.SetHeader(HttpKnownHeaderNames.ContentType, contentType);
10
11 // 将文件流复制到响应输出流
```

```
12     fileStream.CopyTo(responseStream.ResponseStream);
13
14     // 关闭流
15     return responseStream.Close();
```

GZip、Deflate和Brotli压缩

您可以使用Sisk中的HTTP内容压缩发送压缩内容的响应。首先，将您的[HttpContent](#)对象封装在以下压缩器之一中，以向客户端发送压缩响应。

```
1   router.MapGet("/hello.html", request => {
2     string myHtml = "...";
3
4     return new HttpResponse () {
5       Content = new GZipContent(new HtmlContent(myHtml)),
6       // 或 Content = new BrotliContent(new HtmlContent(myHtml)),
7       // 或 Content = new DeflateContent(new HtmlContent(myHtml)),
8     };
9   });
10 };
```

您也可以将这些压缩内容与流一起使用。

```
1   router.MapGet("/archive.zip", request => {
2
3     // 不要在这里应用“using”。HttpServer将在发送响应后丢弃您的内容。
4     // after sending the response.
5     var archive = File.OpenRead("/path/to/big-file.zip");
6
7     return new HttpResponse () {
8       Content = new GZipContent(archive)
9     };
10 });
11 };
```

使用这些内容时，Content-Encoding头部会自动设置。

自动压缩

可以通过[EnableAutomaticResponseCompression](#)属性自动压缩HTTP响应。此属性自动将路由器中的响应内容封装在请求接受的可压缩内容中，前提是响应不是从[CompressedContent](#)继承的。

对于请求，只有一个可压缩的内容被选择，按照以下顺序：

- [BrotliContent](#) (br)
- [GZipContent](#) (gzip)
- [DeflateContent](#) (deflate)

如果请求指定接受这些压缩方法之一，则响应将自动压缩。

隐式响应类型

除了[HttpResponse](#)之外，您还可以使用其他返回类型，但需要配置路由器如何处理每种类型的对象。

概念是始终返回引用类型并将其转换为有效的[HttpResponse](#)对象。返回[HttpResponse](#)的路由不会进行任何转换。

值类型（结构）不能用作返回类型，因为它们与[RouterCallback](#)不兼容，因此必须将它们封装在[ValueResult](#)中才能在处理程序中使用。

考虑以下不使用[HttpResponse](#)作为返回类型的路由器模块的示例：

```
1  [RoutePrefix("/users")]
2  public class UsersController : RouterModule
3  {
4      public List<User> Users = new List<User>();
5
6      [RouteGet]
7      public IEnumerable<User> Index(HttpContext request)
8      {
9          return Users.ToArray();
10     }
11
12     [RouteGet("<id>")]
13     public User View(HttpContext request)
14     {
15         int id = request.RouteParameters["id"].GetInteger();
```

```

16     User dUser = Users.First(u => u.Id == id);
17
18     return dUser;
19 }
20
21 [RoutePost]
22 public ValueResult<bool> Create(HttpRequest request)
23 {
24     User fromBody = JsonSerializer.Deserialize<User>(request.Body)!;
25     Users.Add(fromBody);
26
27     return true;
28 }
29 }
```

因此，现在需要在路由器中定义如何处理每种类型的对象。对象始终是处理程序的第一个参数，输出类型必须是有效的HttpResponse。此外，路由的输出对象永远不应为空。

对于ValueResult类型，不需要指示输入对象是ValueResult，而只需要T，因为ValueResult是其原始组件的反射对象。

类型关联不会比较已注册的内容与路由器回调返回的对象类型。相反，它检查路由器结果的类型是否可分配给已注册的类型。

注册Object类型的处理程序将退回到所有先前未验证的类型。值处理程序的插入顺序也很重要，因此注册Object处理程序将忽略所有其他特定于类型的处理程序。始终首先注册特定的值处理程序以确保顺序。

```

1 Router r = new Router();
2 r.SetObject(new UsersController());
3
4 r.RegisterValueHandler<ApiResult>(apiResult =>
5 {
6     return new HttpResponseMessage() {
7         Status = apiResult.Success ? HttpStatusCode.OK : HttpStatusCode.BadRequest,
8         Content = apiResult.GetHttpContent(),
9         Headers = apiResult.GetHeaders()
10    };
11 });
12 r.RegisterValueHandler<bool>(bvalue =>
13 {
14     return new HttpResponseMessage() {
15         Status = bvalue ? HttpStatusCode.OK : HttpStatusCode.BadRequest
16    };
17 });
18 r.RegisterValueHandler<IEnumerable<object>>(enumerableValue =>
19 {
```

```

20     return new HttpResponseMessage(string.Join("\n", enumerableValue));
21   );
22
23   // 注册对象的值处理器必须是最后一个
24   // 值处理器，用作后备
25   r.RegisterValueHandler<object>(fallback =>
26   {
27     return new HttpResponseMessage() {
28       Status = HttpStatusCode.OK,
29       Content = JsonContent.Create(fallback)
30     };
31   });

```

关于可枚举对象和数组的说明

实现了`IEnumerable`的隐式响应对象在通过定义的值处理器转换之前，会通过`ToArray()`方法读取到内存中。为了发生这种情况，`IEnumerable`对象被转换为对象数组，响应转换器将始终接收`Object[]`而不是原始类型。

考虑以下场景：

```

1  using var host = HttpServer.CreateBuilder(12300)
2  .UseRouter(r =>
3  {
4    r.RegisterValueHandler<IEnumerable<string>>(stringEnumerable =>
5    {
6      return new HttpResponseMessage("String array:\n" + string.Join("\n", stringEnumerable));
7    });
8    r.RegisterValueHandler<IEnumerable<object>>(stringEnumerable =>
9    {
10      return new HttpResponseMessage("Object array:\n" + string.Join("\n", stringEnumerable));
11    });
12    r.MapGet("/", request =>
13    {
14      return (IEnumerable<string>)["hello", "world"];
15    });
16  })
17  .Build();

```

在上面的示例中，`IEnumerable<string>`转换器**永远不会被调用**，因为输入对象始终是`Object[]`，并且不能转换为`IEnumerable<string>`。但是，接收`IEnumerable<object>`的转换器将接收其输入，因为其值是兼容的。

如果您需要实际处理将被枚举的对象类型，则需要使用反射来获取集合元素的类型。所有可枚举对象（列表、数组和集合）都由HTTP响应转换器转换为对象数组。

实现了 [IAsyncEnumerable](#) 的值如果启用了 [ConvertIAsyncEnumerableIntoEnumerable](#) 属性，则由服务器自动处理，类似于 [IEnumerable](#)。异步枚举被转换为阻塞枚举器，然后转换为同步对象数组。

日志

你可以配置 Sisk 自动写入访问日志和错误日志。可以定义日志轮转、扩展和频率。

[LogStream](#) 类提供了一种异步写日志并保持在可等待写队列中的方式。

本文将向你展示如何为你的应用程序配置日志记录。

基于文件的访问日志

日志写入文件时会打开文件，写入行文本，然后在每行写完后关闭文件。此过程是为了保持日志写入的响应性。

Program.cs

C#

```
1  class Program
2  {
3      static async Task Main(string[] args)
4      {
5          using var app = HttpServer.CreateBuilder()
6              .UseConfiguration(config => {
7                  config.AccessLogsStream = new LogStream("logs/access.log");
8              })
9              .Build();
10
11         ...
12
13         await app.StartAsync();
14     }
15 }
```

上述代码会将所有传入请求写入 `logs/access.log` 文件。请注意，如果文件不存在会自动创建，但其前置文件夹不会。无需手动创建 `logs/` 目录，因为 `LogStream` 类会自动创建它。

基于流的日志记录

你可以将日志文件写入 `TextWriter` 对象实例，例如 `Console.Out`，通过在构造函数中传入 `TextWriter` 对象：

Program.cs

C#

```
1  using var app = HttpServer.CreateBuilder()
2      .UseConfiguration(config => {
3          config.AccessLogsStream = new LogStream(Console.Out);
4      })
5      .Build();
```

对于基于流的日志中的每条消息，都会调用 `TextWriter.Flush()` 方法。

访问日志格式化

你可以通过预定义变量自定义访问日志格式。考虑以下行：

```
config.AccessLogsFormat = "%dd/%dmm/%dy %tH:%ti:%ts %tz %ls %ri %rs://%ra%rz%rq [%sc %sd] %lin -
> %lou in %lmsms [%{user-agent}]";
```

它将写出类似以下的消息：

```
29/mar./2023 15:21:47 -0300 Executed ::1 http://localhost:5555/ [200 OK] 689B → 707B in 84ms
[Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/111.0.0.0 Safari/537.36]
```

你可以按下表描述的格式来格式化日志文件：

Value	代表含义	示例
%dd	月份中的日期（两位数字）	05
%dmmm	月份全名	July
%dmm	月份缩写（三个字母）	Jul
%dm	月份编号（两位数字）	07

Value	代表含义	示例
%dy	年份 (四位数字)	2023
%th	12 小时制小时数	03
%tH	24 小时制小时数 (HH)	15
%ti	分钟 (两位数字)	30
%ts	秒 (两位数字)	45
%tm	毫秒 (三位数字)	123
%tz	时区偏移 (UTC 总小时数)	+03:00
%ri	客户端远程 IP 地址	192.168.1.100
%rm	HTTP 方法 (大写)	GET
%rs	URI 方案 (http/https)	https
%ra	URI 权威 (域名)	example.com
%rh	请求主机	www.example.com
%rp	请求端口	443
%rz	请求路径	/path/to/resource
%rq	查询字符串	?key=value&another=123
%sc	HTTP 响应状态码	200
%sd	HTTP 响应状态描述	OK
%lin	请求的可读大小	1.2 KB
%linr	请求的原始大小 (字节)	1234
%lou	响应的可读大小	2.5 KB
%lour	响应的原始大小 (字节)	2560
%lms	毫秒级 elapsed 时间	120
%ls	执行状态	Executed
%{header-name}	表示请求的 header-name 头部。	Mozilla/5.0 (platform; rv:gecko [...]

Value	代表含义	示例
%{res-name}	表示响应的 res-name 头部。	

轮转日志

你可以配置 HTTP 服务器在日志文件达到一定大小时将其轮转为压缩的 .gz 文件。大小会按你定义的阈值定期检查。

```
1 LogStream errorLog = new LogStream("logs/error.log")
2     .ConfigureRotatingPolicy(
3         maximumSize: 64 * SizeHelper.UnitMb,
4         dueTime: TimeSpan.FromHours(6));
```

上述代码会每六小时检查一次 LogStream 的文件是否已达到 64MB 限制。如果是，文件将被压缩为 .gz 文件，然后 access.log 被清理。

在此过程中，写入文件会被锁定，直到文件被压缩并清理完毕。此期间所有待写入的行将排队等待压缩结束。

此功能仅适用于基于文件的 LogStreams。

错误日志

当服务器没有将错误抛给调试器时，它会在有错误时将错误转发到日志写入。你可以通过以下方式配置错误写入：

```
1 config.ThrowExceptions = false;
2 config.ErrorsLogsStream = new LogStream("error.log");
```

此属性仅在错误未被回调或 Router.CallbackErrorHandler 捕获时才会写入日志。

服务器写入的错误始终包含日期时间、请求头（不包括正文）、错误跟踪以及内部异常跟踪（如果有）。

其他日志实例

你的应用程序可以拥有零个或多个 LogStreams，没有限制可以拥有多少日志通道。因此，可以将应用程序的日志定向到除默认 AccessLog 或 ErrorLog 之外的文件。

```
1 LogStream appMessages = new LogStream("messages.log");
2 appMessages.WriteLine("Application started at {0}", DateTime.Now);
```

扩展 LogStream

你可以扩展 LogStream 类以编写自定义格式，兼容当前的 Sisk 日志引擎。下面的示例允许通过 Spectre.Console 库将彩色消息写入 Console：

CustomLogStream.cs

C#

```
1 public class CustomLogStream : LogStream
2 {
3     protected override void WriteLineInternal(string line)
4     {
5         base.WriteLineInternal($"[{DateTime.Now:g}] {line}");
6     }
7 }
```

另一种自动为每个请求/响应写入自定义日志的方法是创建一个 [HttpServerHandler](#)。下面的示例更完整。它将请求和响应正文以 JSON 写入 Console。对于调试请求非常有用。此示例使用 ContextBag 和 HttpServerHandler。

Program.cs

C#

```
1 class Program
2 {
3     static async Task Main(string[] args)
4     {
5         var app = HttpServer.CreateBuilder(host =>
6         {
7             host.UseListeningPort(5555);
8             host.UseHandler<JsonMessageHandler>();
```

```

9         });
10
11         app.Router += new Route(RouteMethod.Any, "/json", request =>
12     {
13
14             return new HttpResponseMessage()
15                 .WithContent(JsonContent.Create(new
16                     {
17                         method = request.Method.Method,
18                         path = request.Path,
19                         specialMessage = "Hello, world!!"
20                     }));
21
22         await app.StartAsync();
23     }
24 }
```

JsonMessageHandler.cs

C#

```

1  class JsonMessageHandler : HttpServerHandler
2  {
3
4      protected override void OnHttpRequestOpen(HttpRequest request)
5      {
6
7          if (request.Method != HttpMethod.Get && request.Headers["Content-
8 Type"]?.Contains("json", StringComparison.InvariantCultureIgnoreCase) = true)
9          {
10
11              // At this point, the connection is open and the client has sent the
12 header specifying
13
14              // that the content is JSON. The line below reads the content and leaves it
15 stored in the request.
16
17              //
18
19              // If the content is not read in the request action, the GC is likely to
20 collect the content
21
22              // after sending the response to the client, so the content may not be
23 available after the response is closed.
24
25              //
26
27              _ = request.RawBody;
28
29
30              // add hint in the context to tell that this request has an json body on it
31              request.Bag.Add("IsJsonRequest", true);
32
33      }
34
35
36      protected override async void OnHttpRequestClose(HttpServerExecutionResult result)
37      {
38
39          string? requestJson = null,
```

```

28             responseJson = null,
29             responseMessage;
30
31         if (result.Request.Bag.ContainsKey("IsJsonRequest"))
32     {
33             // reformats the JSON using the CypherPotato.LightJson library
34             var content = result.Request.Body;
35             requestJson = JsonValue.Deserialize(content, new JsonOptions() { WriteIndented
36 = true }).ToString();
37         }
38
39         if (result.Response is {} response)
40     {
41             var content = response.Content;
42             responseMessage = $"{(int)response.Status}
43 {HttpStatusInformation.GetStatusCodeDescription(response.Status)}";
44
45             if (content is HttpContent httpContent &&
46                 // check if the response is JSON
47                 httpContent.Headers.ContentType?.MediaType?.Contains("json",
48 StringComparison.InvariantCultureIgnoreCase) = true)
49             {
50                 string json = await httpContent.ReadAsStringAsync();
51                 responseJson = JsonValue.Deserialize(json, new JsonOptions() {
52 WriteIndented = true }).ToString();
53             }
54         }
55         else
56     {
57             // gets the internal server handling status
58             responseMessage = result.Status.ToString();
59         }
60
61         StringBuilder outputMessage = new StringBuilder();
62
63         if (requestJson != null)
64     {
65             outputMessage.AppendLine("-----");
66             outputMessage.AppendLine($">> {result.Request.Method} {result.Request.Path}");
67
68             if (requestJson is not null)
69                 outputMessage.AppendLine(requestJson);
70         }
71
72         outputMessage.AppendLine($"<<< {responseMessage}");
73
74         if (responseJson is not null)
75             outputMessage.AppendLine(responseJson);

```

```
        outputMessage.AppendLine("-----");

        await Console.Out.WriteLineAsync(outputMessage.ToString());
    }
}
```

服务器发送事件

Sisk 支持开箱即用地通过服务器发送事件发送消息。您可以创建可 disposable 和持久的连接，在运行时获取连接并使用它们。

此功能有一些受到浏览器限制的限制，例如只能发送文本消息，无法永久关闭连接。服务器端关闭的连接将由客户端每 5 秒（某些浏览器为 3 秒）尝试重新连接。

这些连接对于在不每次由客户端请求信息的情况下，从服务器向客户端发送事件非常有用。

创建 SSE 连接

SSE 连接的工作方式与常规 HTTP 请求类似，但不是发送响应并立即关闭连接，而是保持连接打开以发送消息。

调用 [HttpRequest.GetEventSource\(\)](#) 方法，请求将进入等待状态，同时创建 SSE 实例。

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3       using var sse = req.GetEventSource();
4
5       sse.Send("Hello, world!");
6
7       return sse.Close();
8   });
9 }
```

在上面的代码中，我们创建了一个 SSE 连接并发送一条“Hello, world”消息，然后从服务器端关闭 SSE 连接。

NOTE

当关闭服务器端连接时，默认情况下客户端将尝试再次连接，该连接将重新启动，永久执行该方法。

通常，在从服务器关闭连接时，转发终止消息以防止客户端尝试再次连接是很常见的。

追加头部

如果您需要发送头部，可以在发送任何消息之前使用 [HttpRequestEventSource.AppendHeader](#) 方法。

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3       using var sse = req.GetEventSource();
4       sse.AppendHeader("Header-Key", "Header-value");
5
6       sse.Send("Hello!");
7
8       return sse.Close();
9   });

```

注意，必须在发送任何消息之前发送头部。

等待失败连接

连接通常在服务器由于可能的客户端断开连接而无法发送消息时终止。因此，连接会自动终止，类的实例将被丢弃。

即使重新连接，类的实例也不会工作，因为它与之前的连接相关联。在某些情况下，您可能需要在稍后需要此连接，而不想通过路由的回调方法管理它。

为此，我们可以用标识符标识 SSE 连接，并在稍后使用它，甚至在路由的回调之外。此外，我们使用 [WaitForFail](#) 标记连接，以便在不终止路由和自动终止连接的情况下。

KeepAlive 中的 SSE 连接将等待发送错误（由断开连接引起）以恢复方法执行。也可以为此设置超时。在超时后，如果未发送任何消息，则连接将被终止，执行将恢复。

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3       using var sse = req.GetEventSource("my-index-connection");
4
5       sse.WaitForFail(TimeSpan.FromSeconds(15)); // 在 15 秒内无任何消息时终止连接
6
7       return sse.Close();
8   });

```

上述方法将创建连接，处理它并等待断开连接或错误。

```
1  HttpRequestEventSource? evs = server.EventSources.GetByIdentifier("my-index-connection");
2  if (evs != null)
3  {
4      // 连接仍然活跃
5      evs.Send("Hello again!");
6  }
```

上面的代码片段将尝试查找新创建的连接，如果存在，则向其发送消息。

所有已标识的活动服务器连接都可在集合 `HttpServer.EventSources` 中。该集合仅存储活动和已标识的连接。关闭的连接将从集合中删除。

NOTE

值得注意的是，`KeepAlive` 由可能以无法控制的方式连接到 Sisk 的组件（例如，Web 代理，HTTP 内核或网络驱动程序）建立的限制，并且它们在一段时间内后关闭空闲连接。

因此，通过发送周期性 ping 或延长连接关闭前的时间来保持连接打开非常重要。阅读下一节以更好地了解发送周期性 ping。

设置连接 ping 策略

Ping 策略是一种自动向客户端发送周期性消息的方法。此功能允许服务器在不将连接无限期保持打开的情况下，了解客户端何时已断开与该连接的连接。

```
1  [RouteGet("/sse")]
2  public HttpResponse Events(HttpRequest request)
3  {
4      using var sse = request.GetEventSource();
5      sse.WithPing(ping =>
6      {
7          ping.DataMessage = "ping-message";
8          ping.Interval = TimeSpan.FromSeconds(5);
9          ping.Start();
10     });
11
12     sse.KeepAlive();
13
14 }
```

```
    return sse.Close();
}
```

在上面的代码中，每 5 秒将向客户端发送一个新的 ping 消息。这将保持 TCP 连接活跃，防止由于不活动而关闭。此外，当发送消息失败时，连接将自动关闭，释放连接使用的资源。

查询连接

您可以使用谓词搜索活动连接的标识符，以便能够广播，例如。

```
1  HttpRequestEventSource[] evs = server.EventSources.Find(es => es.StartsWith("my-
2  connection-"));
3  foreach (HttpRequestEventSource e in evs)
4  {
5      e.Send("Broadcasting to all event sources that starts with 'my-connection-'");
}
```

您还可以使用 [All](#) 方法获取所有活动 SSE 连接。

Web Sockets

Sisk 支持 WebSockets，例如接收和发送消息给客户端。

此功能在大多数浏览器中运行良好，但在 Sisk 中仍处于实验阶段。若发现任何 bug，请在 GitHub 上报告。

异步接收消息

WebSocket 消息按顺序接收，排队直到被 `ReceiveMessageAsync` 处理。该方法在超时、操作被取消或客户端断开时返回无消息。

一次只能进行一次读取和写入操作，因此在等待 `ReceiveMessageAsync` 的消息时，无法向已连接的客户端写入。

```
1  router.MapGet("/connect", async (HttpRequest req) =>
2  {
3      using var ws = await req.GetWebSocketAsync();
4
5      while (await ws.ReceiveMessageAsync(timeout: TimeSpan.FromSeconds(30)) is {
6          receivedMessage)
7          {
8              string msgText = receivedMessage.GetString();
9              Console.WriteLine("Received message: " + msgText);
10
11              await ws.SendAsync("Hello!");
12      }
13
14      return await ws.CloseAsync();
15});
```

同步接收消息

下面的示例展示了如何使用同步 WebSocket，而不需要异步上下文，在接收消息、处理后完成 socket 的使用。

```

1  router.MapGet("/connect", async (HttpRequest req) =>
2  {
3      using var ws = await req.GetWebSocketAsync();
4      WebSocketMessage? msg;
5
6      askName:
7          await ws.SendAsync("What is your name?");
8          msg = await ws.ReceiveMessageAsync();
9
10     if (msg is null)
11         return await ws.CloseAsync();
12
13     string name = msg.GetString();
14
15     if (string.IsNullOrEmpty(name))
16     {
17         await ws.SendAsync("Please, insert your name!");
18         goto askName;
19     }
20
21     askAge:
22         await ws.SendAsync("And your age?");
23         msg = await ws.ReceiveMessageAsync();
24
25         if (msg is null)
26             return await ws.CloseAsync();
27
28         if (!Int32.TryParse(msg?.GetString(), out int age))
29         {
30             await ws.SendAsync("Please, insert an valid number");
31             goto askAge;
32         }
33
34         await ws.SendAsync($"You're {name}, and you are {age} old.");
35
36         return await ws.CloseAsync();
37 });

```

Ping 策略

类似于 Server Side Events 中的 ping 策略，你也可以配置 ping 策略，以在无活动时保持 TCP 连接打开。

```
1 ws.PingPolicy.Start(  
2     dataMessage: "ping-message",  
3     interval: TimeSpan.FromSeconds(10));
```

放弃语法

HTTP 服务器可以用于监听来自操作的回调请求，例如 OAuth 身份验证，并在接收到该请求后可以被丢弃。这在需要后台操作但不想为其设置整个 HTTP 应用的情况下非常有用。

以下示例展示了如何在端口 5555 上创建一个监听 HTTP 服务器，并等待下一个上下文：

```
1  using (var server = HttpServer.CreateListener(5555))
2  {
3      // 等待下一个 HTTP 请求
4      var context = await server.WaitNextAsync();
5      Console.WriteLine($"请求路径: {context.Request.Path}");
6  }
```

`WaitNext` 函数等待下一个已完成的请求处理上下文。一旦获得此操作的结果，服务器已经完全处理了请求并将响应发送给客户端。

依赖注入

通常，会为请求的生命周期内持续存在的成员和实例分配内存，例如数据库连接、已验证的用户或会话令牌。实现这一点的一种可能方法是通过 `HttpContext.RequestBag`，它创建一个在整个请求生命周期内持续存在的字典。

该字典可以被 [请求处理器](#) 访问，并在整个请求过程中定义变量。例如，验证用户的请求处理器将用户设置在 `HttpContext.RequestBag` 中，在请求逻辑中，可以通过 `HttpContext.RequestBag.Get<User>()` 来检索该用户。

以下是一个示例：

RequestHandlers/AuthenticateUser.cs

C#

```
1  public class AuthenticateUser : IRequestHandler
2  {
3      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
4          RequestHandlerExecutionMode.BeforeResponse;
5
6      public HttpResponse? Execute(HttpRequest request, HttpContext context)
7      {
8          User authenticatedUser = AuthenticateUser(request);
9          context.RequestBag.Set(authenticatedUser);
10         return null; // advance to the next request handler or request logic
11     }
12 }
```

Controllers/HelloController.cs

C#

```
1  [RouteGet("/hello")]
2  [RequestHandler<AuthenticateUser>]
3  public static HttpResponse SayHello(HttpContext request)
4  {
5      var authenticatedUser = request.Bag.Get<User>();
6      return new HttpResponseMessage()
7      {
8          Content = new StringContent($"Hello {authenticatedUser.Name}!")
9      };
10 }
```

这是对此操作的初步示例。`User` 实例是在专门用于身份验证的请求处理器中创建的，并且所有使用此请求处理器的路由都保证在其 `HttpContext.RequestBag` 实例中将有一个 `User`。

可以通过诸如 `GetOrAdd` 或 `GetOrAddAsync` 之类的方法来定义获取实例的逻辑，当实例尚未在 `RequestBag` 中定义时。

从 1.3 版本开始，引入了静态属性 `HttpContext.Current`，允许访问当前执行的 `HttpContext` 的请求上下文。这使得可以在当前请求之外暴露 `HttpContext` 的成员，并在路由对象中定义实例。

以下示例定义了一个控制器，该控制器具有通常由请求上下文访问的成员。

Controllers/Controller.cs

C#

```
1  public abstract class Controller : RouterModule
2  {
3      public DbContext Database
4      {
5          get
6          {
7              // 创建一个 DbContext 或获取现有的一个
8              return HttpContext.Current.RequestBag.GetOrAdd(() => new DbContext());
9          }
10     }
11
12     // 如果属性在请求包中没有定义 User 时访问，将抛出异常
13     public User AuthenticatedUser { get => HttpContext.Current.RequestBag.Get<User>(); }
14
15     // 也支持暴露 HttpRequest 实例
16     public HttpRequest Request { get => HttpContext.Current.Request; }
17 }
```

并定义继承自控制器的类型：

Controllers/PostsController.cs

C#

```
1  [RoutePrefix("/api/posts")]
2  public class PostsController : Controller
3  {
4      [RouteGet]
5      public IEnumerable<Blog> ListPosts()
6      {
7          return Database.Posts
8              .Where(post => post.AuthorId = AuthenticatedUser.Id)
9              .ToList();
10     }
11
12     [RouteGet("<id>")]
13     public Post GetPost()
```

```

14     {
15         int blogId = Request.RouteParameters["id"].GetInteger();
16
17         Post? post = Database.Posts
18             .FirstOrDefault(post => post.Id == blogId && post.AuthorId
19 = AuthenticatedUser.Id);
20
21         return post ?? new HttpResponseMessage(404);
22     }
}

```

对于上面的示例，您需要在路由器中配置一个[值处理程序](#)，以便路由器返回的对象转换为有效的 [HttpResponse](#)。

请注意，方法不带有 `HttpRequest request` 参数，如其他方法中所示。这是因为，从 1.3 版本开始，路由器支持两种类型的委托用于路由响应：[RouteAction](#)，这是默认的委托，它接收一个 `HttpRequest` 参数，以及 [ParameterlessRouteAction](#)。`HttpRequest` 对象仍然可以通过静态 `HttpContext` 的 [Request](#) 属性访问。

在上面的示例中，我们定义了一个可处置对象 `DbContext`，并且我们需要确保在 HTTP 会话结束时处置所有在 `DbContext` 中创建的实例。为此，我们可以使用两种方法来实现这一点。一种方法是创建一个在路由器操作之后执行的[请求处理器](#)，另一种方法是通过自定义[服务器处理器](#)。

对于第一种方法，我们可以直接在继承自 `RouterModule` 的 `OnSetup` 方法中内联创建请求处理器：

Controllers/PostsController.cs

C#

```

1  public abstract class Controller : RouterModule
2  {
3      ...
4
5      protected override void OnSetup(Router parentRouter)
6      {
7          base.OnSetup(parentRouter);
8
9          HasRequestHandler(RequestHandler.Create(
10              execute: (req, ctx) =>
11              {
12                  // 获取请求处理器上下文中定义的 DbContext 并处置它
13                  ctx.RequestBag.GetOrDefault<DbContext>()?.Dispose();
14                  return null;
15              },
16              executionMode: RequestHandlerExecutionMode.AfterResponse));
17      }
18 }

```

TIP

从 Sisk 1.4 版本开始，引入了属性 `HttpServerConfiguration.DisposeDisposableContextValues`，它默认启用，用于定义 HTTP 服务器是否应在 HTTP 会话关闭时处置上下文包中的所有 `IDisposable` 值。

上述方法将确保在 HTTP 会话结束时处置 `DbContext`。您可以为需要在响应结束时处置的其他成员执行此操作。

对于第二种方法，您可以创建一个自定义的 [服务器处理程序](#)，它将在 HTTP 会话结束时处置 `DbContext`。

Server/Handlers/ObjectDisposerHandler.cs

C#

```
1 public class ObjectDisposerHandler : HttpServerHandler
2 {
3     protected override void OnHttpRequestClose(HttpServerExecutionResult result)
4     {
5         result.Context.RequestBag.GetOrDefault<DbContext>()?.Dispose();
6     }
7 }
```

并在应用程序生成器中使用它：

Program.cs

C#

```
1 using var host = HttpServer.CreateBuilder()
2     .UseHandler<ObjectDisposerHandler>()
3     .Build();
```

这是处理代码清理并将请求的依赖项与将使用的模块类型分离的一种方法，减少了路由器操作中重复的代码量。这是一种类似于在 ASP.NET 等框架中使用依赖注入的做法。

流式内容

Sisk 支持读取和发送流式内容到和从客户端。这一功能对于在请求的生命周期中序列化和反序列化内容的内存开销非常有用。

请求内容流

小内容会自动加载到 HTTP 连接缓冲区内存中，快速加载到 `HttpRequest.Body` 和 `HttpRequest.RawBody`。对于较大的内容，可以使用 `HttpRequest.GetRequestStream` 方法来获取请求内容读取流。

值得注意的是，`HttpRequest.GetMultipartFormContent` 方法会将整个请求内容读入内存，因此对于读取大内容可能不太有用。

考虑以下示例：

Controller/UploadDocument.cs

C#

```
1 [RoutePost ( "/api/upload-document/<filename>" )]
2 public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4     var fileName = request.RouteParameters [ "filename" ].GetString ();
5
6     if (!request.HasContents) {
7         // 请求没有内容
8         return new HttpResponse ( HttpStatusInformation.BadRequest );
9     }
10
11    var contentStream = request.GetRequestStream ();
12    var outputFileName = Path.Combine (
13        AppDomain.CurrentDomain.BaseDirectory,
14        "uploads",
15        fileName );
16
17    using (var fs = File.Create ( outputFileName )) {
18        await contentStream.CopyToAsync ( fs );
19    }
20}
```

```
21     return new HttpResponseMessage () {
22         Content = JsonContent.Create ( new { message = "文件发送成功。" } )
23     };
24 }
```

在上面的示例中，`UploadDocument` 方法读取请求内容并将内容保存到文件中。除了 `Stream.CopyToAsync` 使用的读取缓冲区外，不会进行任何额外的内存分配。上面的示例消除了对非常大文件的内存分配压力，可以优化应用程序性能。

一个良好的做法是在可能耗时的操作中始终使用 [CancellationToken](#)，例如发送文件，因为它取决于客户端和服务器之间的网络速度。

可以通过以下方式调整 `CancellationToken`：

Controller/UploadDocument.cs

C#

```
1 // 下面的取消令牌将在 30 秒超时时抛出异常。
2 CancellationTokenSource copyCancellation = new CancellationTokenSource ( delay:
3 TimeSpan.FromSeconds ( 30 ) );
4
5 try {
6     using (var fs = File.Create ( outputFileName )) {
7         await contentStream.CopyToAsync ( fs, copyCancellation.Token );
8     }
9 }
10 catch (OperationCanceledException) {
11     return new HttpResponseMessage ( HttpStatusCode.BadRequest ) {
12         Content = JsonContent.Create ( new { Error = "上传超出了最大上传时间 (30 秒)。" } )
13     };
14 }
```

响应内容流

发送响应内容也是可能的。目前，有两种方法可以做到这一点：通过 `HttpRequest.GetResponseStream` 方法和使用 `StreamContent` 类型的内容。

考虑一个需要提供图像文件的场景。可以使用以下代码：

Controller/ImageController.cs

C#

```

1 [RouteGet ( "/api/profile-picture" )]
2 public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4     // 示例方法来获取个人资料图片
5     var profilePictureFilename = "profile-picture.jpg";
6     byte[] profilePicture = await File.ReadAllBytesAsync ( profilePictureFilename );
7
8     return new HttpResponse () {
9         Content = new ByteArrayContent ( profilePicture ),
10        Headers = new () {
11            ContentType = "image/jpeg",
12            ContentDisposition = $"inline; filename={profilePictureFilename}"
13        }
14    };
15 }

```

上面的方法每次读取图像内容时都会进行内存分配。如果图像很大，这可能会导致性能问题，并且在峰值情况下，甚至可能导致内存过载和服务器崩溃。在这些情况下，缓存可能很有用，但它不会消除问题，因为仍然需要为该文件保留内存。缓存可以缓解每次请求都需要分配内存的压力，但对于大文件来说，它是不够的。

通过流式传输发送图像可以解决这个问题。与其读取整个图像内容，不如创建一个文件的读取流，并使用一个小缓冲区将其复制到客户端。

通过 GetResponseStream 方法发送

`HttpRequest.GetResponseStream` 方法创建一个对象，允许将 HTTP 响应的块作为内容流准备好时发送。这种方法更为手动，需要在发送内容之前定义状态、头部和内容大小。

Controller/ImageController.cs

C#

```

1 [RouteGet ( "/api/profile-picture" )]
2 public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4     var profilePictureFilename = "profile-picture.jpg";
5
6     // 在这种发送形式中，必须在发送内容之前定义状态和头部
7     var requestStreamManager = request.GetResponseStream ();
8
9     requestStreamManager.SetStatus ( System.Net.HttpStatusCode.OK );
10    requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentType, "image/jpeg" );
11    requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentDisposition, $"inline;
12    filename={profilePictureFilename}" );
13
14    using (var fs = File.OpenRead ( profilePictureFilename )) {

```

```
15  
16     // 在这种发送形式中，必须在发送内容之前定义内容大小  
17     requestStreamManager.SetContentLength ( fs.Length );  
18  
19     // 如果不知道内容大小，可以使用分块编码来发送内容  
20     requestStreamManager.SendChunked = true;  
21  
22     // 然后，将内容写入输出流  
23     await fs.CopyToAsync ( requestStreamManager.ResponseStream );  
24 }
```

通过 StreamContent 发送内容

StreamContent 类允许将数据源作为字节流发送内容。这种发送形式更为简单，消除了之前的要求，甚至允许使用压缩编码 来减少内容大小。

Controller/ImageController.cs

C#

```
1 [RouteGet ( "/api/profile-picture" )]  
2 public HttpResponse UploadDocument ( HttpRequest request ) {  
3  
4     var profilePictureFilename = "profile-picture.jpg";  
5  
6     return new HttpResponse () {  
7         Content = new StreamContent ( File.OpenRead ( profilePictureFilename ) ),  
8         Headers = new () {  
9             ContentType = "image/jpeg",  
10            ContentDisposition = $"inline; filename=\\"{profilePictureFilename}\\""  
11        }  
12    };  
13 }
```

⊗ IMPORTANT

在这种类型的内容中，不要将流封装在 `using` 块中。内容将在 HTTP 服务器完成内容流时自动丢弃，无论是否有错误。

在 Sisk 中启用 CORS (跨域资源共享)

Sisk 有一个工具，在公开服务时处理 [跨域资源共享 \(CORS\)](#) 非常有用。此功能不是 HTTP 协议的一部分，而是由 W3C 定义的 Web 浏览器的特定功能。此安全机制阻止网页向与提供网页的域不同的域发起请求。服务提供者可以允许某些域访问其资源，或者仅允许一个域。

同源

要将资源识别为“同源”，请求必须在其请求中标识 [Origin](#) 头：

```
1 GET /api/users HTTP/1.1
2 Host: example.com
3 Origin: http://example.com
4 ...
```

远程服务器必须以与请求的 origin 相同的值返回一个 [Access-Control-Allow-Origin](#) 头：

```
1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: http://example.com
3 ...
```

此验证是 **显式** 的：主机、端口和协议必须与请求的完全相同。查看示例：

- 服务器响应其 Access-Control-Allow-Origin 为 https://example.com：
 - https://example.net - 域不同。
 - http://example.com - 协议不同。
 - http://example.com:5555 - 端口不同。
 - https://www.example.com - 主机不同。

在规范中，只有语法被允许用于请求和响应的两个头。URL 路径被忽略。如果是默认端口（HTTP 为 80，HTTPS 为 443），则端口也会被省略。

```
1 Origin: null
2 Origin: <scheme>://<hostname>
```

启用 CORS

本机上，你可以在 [ListeningHost](#) 中使用 [CrossOriginResourceSharingHeaders](#) 对象。

你可以在初始化服务器时配置 CORS：

```

1  static async Task Main(string[] args)
2  {
3      using var app = HttpServer.CreateBuilder()
4          .UseCors(new CrossOriginResourceSharingHeaders(
5              allowOrigin: "http://example.com",
6              allowHeaders: ["Authorization"],
7              exposeHeaders: ["Content-Type"]))
8          .Build();
9
10     await app.StartAsync();
11 }
```

上述代码将为 **所有响应** 发送以下头：

```

1  HTTP/1.1 200 OK
2  Access-Control-Allow-Origin: http://example.com
3  Access-Control-Allow-Headers: Authorization
4  Access-Control-Expose-Headers: Content-Type
```

这些头需要为所有响应发送给 Web 客户端，包括错误和重定向。

你可能会注意到 [CrossOriginResourceSharingHeaders](#) 类有两个类似的属性：[AllowOrigin](#) 和 [AllowOrigins](#)。请注意，一个是复数，另一个是单数。

- **AllowOrigin** 属性是静态的：仅你指定的 origin 将被发送给所有响应。
- **AllowOrigins** 属性是动态的：服务器检查请求的 origin 是否包含在此列表中。如果找到，则将其发送给该 origin 的响应。

通配符和自动头

或者，你可以在响应的 origin 中使用通配符（ * ）来指定允许任何 origin 访问资源。然而，此值不允许用于具有凭据（授权头）的请求，并且此操作将导致错误（[CORSNotSupportingCredentials](#)）。

你可以通过在 `AllowOrigins` 属性中显式列出允许的 origin，或在 `AllowOrigin` 的值中使用 `AutoAllowOrigin` 常量来解决此问题。此魔法属性将为 `Origin` 头的相同值定义 `Access-Control-Allow-Origin` 头。

你还可以使用 `AutoFromRequestMethod` 和 `AutoFromRequestHeaders` 来实现类似 `AllowOrigin` 的行为，自动根据发送的头进行响应。

```
1  using var host = HttpServer.CreateBuilder()
2      .UseCors(new CrossOriginResourceSharingHeaders(
3
4          // 根据请求的 Origin 头进行响应
5          allowOrigin: CrossOriginResourceSharingHeaders.AutoAllowOrigin,
6
7          // 根据 Access-Control-Request-Method 头或请求方法进行响应
8          allowMethods: [CrossOriginResourceSharingHeaders.AutoFromRequestMethod],
9
10         // 根据 Access-Control-Request-Headers 头或发送的头进行响应
11         allowHeaders: [CrossOriginResourceSharingHeaders.AutoFromRequestHeaders]))
```

其他应用 CORS 的方式

如果你正在处理 `service providers`，可以覆盖配置文件中定义的值：

```
1  static async Task Main(string[] args)
2  {
3      using var app = HttpServer.CreateBuilder()
4          .UsePortableConfiguration( ... )
5          .UseCors(cors => {
6              // 将覆盖配置文件中定义的 origin
7              cors.AllowOrigin = "http://example.com";
8          })
9          .Build();
10
11     await app.StartAsync();
12 }
```

在特定路由上禁用 CORS

UseCors 属性可用于所有路由和所有路由属性，并可通过以下示例禁用：

```
1 [RoutePrefix("api/widgets")]
2 public class WidgetController : Controller {
3
4     // GET /api/widgets/colors
5     [RouteGet("/colors", UseCors = false)]
6     public IEnumerable<string> GetWidgets() {
7         return new[] { "Green widget", "Red widget" };
8     }
9 }
```

在响应中替换值

你可以在路由器操作中显式替换或删除值：

```
1 [RoutePrefix("api/widgets")]
2 public class WidgetController : Controller {
3
4     public IEnumerable<string> GetWidgets(HttpContext request) {
5
6         // 删除 Access-Control-Allow-Credentials 头
7         request.Context.OverrideHeaders.AccessControlAllowCredentials = string.Empty;
8
9         // 替换 Access-Control-Allow-Origin
10        request.Context.OverrideHeaders.AccessControlAllowOrigin = "https://contorso.com";
11
12        return new[] { "Green widget", "Red widget" };
13    }
14 }
```

预检请求

预检请求是客户端在实际请求之前发送的 [OPTIONS](#) 方法请求。

Sisk 服务器将始终以 `200 OK` 和适用的 CORS 头响应请求，然后客户端可以继续实际请求。此条件仅在请求存在路由且 [RouteMethod](#) 明确配置为 `Options` 时不适用。

全局禁用 CORS

无法做到这一点。若不使用 CORS，请不要配置它。

JSON-RPC 扩展

Sisk 有一个实验性的 JSON-RPC 2.0 API 模块，允许您创建更简单的应用程序。该扩展严格实现了 JSON-RPC 2.0 传输接口，并提供通过 HTTP GET、POST 请求和 WebSockets 的传输。

您可以通过 Nuget 安装该扩展，使用以下命令。请注意，在实验/测试版本中，您需要在 Visual Studio 中启用搜索预发布包的选项。

```
dotnet add package Sisk.JsonRpc
```

传输接口

JSON-RPC 是一种无状态、异步的远程过程调用（RDP）协议，使用 JSON 进行单向数据通信。JSON-RPC 请求通常由一个 ID 标识，响应由相同的 ID 发送。并非所有请求都需要响应，这些被称为“通知”。

JSON-RPC 2.0 规范[规范](#) 详细解释了传输的工作原理。该传输与其使用位置无关。Sisk 通过 HTTP 实现该协议，遵循 JSON-RPC over HTTP[的规定](#)，部分支持 GET 请求，完全支持 POST 请求。WebSockets 也被支持，提供异步消息通信。

JSON-RPC 请求类似于：

```
1  {
2      "jsonrpc": "2.0",
3      "method": "Sum",
4      "params": [1, 2, 4],
5      "id": 1
6 }
```

成功响应类似于：

```
1  {
2      "jsonrpc": "2.0",
3      "result": 7,
4
5 }
```

```
"id": 1  
}
```

JSON-RPC 方法

以下示例显示如何使用 Sisk 创建 JSON-RPC API。一个数学运算类执行远程操作并将序列化的响应发送给客户端。

Program.cs

C#

```
1  using var app = HttpServer.CreateBuilder(port: 5555)  
2      .UseJsonRPC((sender, args) =>  
3      {  
4          // 添加所有标记为 WebMethod 的方法到 JSON-RPC 处理器  
5          args.Handler.Methods.AddMethodsFromType(new MathOperations());  
6  
7          // 将 /service 路由映射到处理 JSON-RPC POST 和 GET 请求  
8          args.Router.MapPost("/service", args.Handler.Transport.HttpPost);  
9          args.Router.MapGet("/service", args.Handler.Transport.HttpGet);  
10  
11         // 在 GET /ws 创建一个 WebSocket 处理器  
12         args.Router.MapGet("/ws", request =>  
13         {  
14             var ws = request.GetWebSocket();  
15             ws.OnReceive += args.Handler.Transport.WebSocket;  
16  
17             ws.WaitForClose(timeout: TimeSpan.FromSeconds(30));  
18             return ws.Close();  
19         });  
20     })  
21     .Build();  
22  
23     await app.StartAsync();
```

MathOperations.cs

C#

```
1  public class MathOperations  
2  {  
3      [WebMethod]  
4      public float Sum(float a, float b)  
5      {
```

```
6         return a + b;
7     }
8
9     [WebMethod]
10    public double Sqrt(float a)
11    {
12        return Math.Sqrt(a);
13    }
14 }
```

上面的示例将 `Sum` 和 `Sqrt` 方法映射到 JSON-RPC 处理器，这些方法将在 `GET /service`、`POST /service` 和 `GET /ws` 中可用。方法名称不区分大小写。

方法参数将自动反序列化为其特定类型。使用带有命名参数的请求也是支持的。JSON 序列化由 [LightJson](#) 库执行。当类型不能正确反序列化时，您可以为该类型创建一个特定的 [JSON 转换器](#) 并稍后将其与 `JsonSerializerOptions` 关联起来。

您还可以直接在方法中获取 JSON-RPC 请求的原始 `$.params` 对象。

MathOperations.cs

C#

```
1 [WebMethod]
2 public float Sum(JsonArray|JsonObject @params)
3 {
4     ...
5 }
```

为此，`@params` 必须是方法中唯一的参数，且名称必须为 `params`（在 C# 中，`@` 用于转义此参数名称）。

参数反序列化适用于命名对象和位置数组。例如，以下方法可以通过以下两种请求调用：

```
1 [WebMethod]
2 public float AddUserToStore(string apiKey, User user, UserStore store)
3 {
4     ...
5 }
```

对于数组，参数的顺序必须遵循。

```
1 {
2     "jsonrpc": "2.0",
3     "method": "AddUserToStore",
4     "params": [
5         "1234567890",
```

```
6         {
7             "name": "John Doe",
8             "email": "john@example.com"
9         },
10        {
11            "name": "My Store"
12        }
13    ],
14    "id": 1
15
16 }
```

自定义序列化器

您可以在 `JsonRpcHandler.JsonSerializerOptions` 属性中自定义 JSON 序列化器。在此属性中，您可以启用 [JSON5](#) 以用于反序列化消息。虽然这不是 JSON-RPC 2.0 的一部分，但 JSON5 是 JSON 的一个扩展，允许更易读和更具可读性的写作。

Program.cs

C#

```
1  using var host = HttpServer.CreateBuilder( 5556 )
2      .UseJsonRPC( ( o, e ) => {
3
4          // 使用一个标准化的名称比较器。该比较器仅比较名称中的字母和数字，并丢弃其他符号。例如：
5          // foo_bar10 = FooBar10
6          e.Handler.JsonSerializerOptions.PropertyNameComparer = new
7          JsonSanitizedComparer();
8
9          // 启用 JSON5 以用于 JSON 解释器。即使激活此选项，普通 JSON 仍然被允许
10         e.Handler.JsonSerializerOptions.SerializationFlags =
11         LightJson.Serialization.JsonSerializationFlags.Json5;
12
13         // 将 POST /service 路由映射到 JSON-RPC 处理器
14         e.Router.MapPost( "/service", e.Handler.Transport.HttpPost );
15     }
16     .Build();
17
18
19 host.Start();
```

SSL 代理

① WARNING

此功能是实验性的，不应在生产环境中使用。如果您想让 Sisk 与 SSL 协作，请参阅 [此文档](#)。

Sisk SSL 代理是一个模块，提供了一个 HTTPS 连接，用于 Sisk 中的 `ListeningHost`，并将 HTTPS 消息路由到不安全的 HTTP 上下文。该模块是为使用 [HttpListener](#) 运行的服务提供 SSL 连接而构建的，因为 `HttpListener` 不支持 SSL。

代理在同一应用程序中运行，并监听 HTTP/1.1 消息，将它们以相同的协议转发给 Sisk。目前，此功能是高度实验性的，可能不稳定到不能在生产环境中使用。

目前，`SslProxy` 支持几乎所有 HTTP/1.1 功能，例如 `keep-alive`、分块编码、`WebSockets` 等。对于打开到 SSL 代理的连接，会创建一个到目标服务器的 TCP 连接，并将代理转发到已建立的连接。

`SslProxy` 可以与 `HttpServer.CreateBuilder` 一起使用，如下所示：

```
1  using var app = HttpServer.CreateBuilder(port: 5555)
2      .UseRouter(r =>
3      {
4          r.MapGet("/", request =>
5          {
6              return new HttpResponseMessage("Hello, world!");
7          });
8      })
9      // 添加 SSL 到项目
10     .UseSsl(
11         sslListeningPort: 5567,
12         new X509Certificate2(@".\ssl.pfx", password: "12345")
13     )
14     .Build();
15
16 app.Start();
```

您必须为代理提供一个有效的 SSL 证书。为了确保证书被浏览器接受，请记得将其导入到操作系统中，以便它能够正常工作。

基本身份验证

基本身份验证包添加了一个请求处理器，能够处理基本身份验证方案，并且只需进行很少的配置和努力即可在您的 Sisk 应用程序中使用。基本 HTTP 身份验证是一种最小的输入形式，通过用户 ID 和密码对请求进行身份验证，会话完全由客户端控制，并且没有身份验证或访问令牌。



更多关于基本身份验证方案的信息，请参阅 [MDN 规范](#)。

安装

要开始使用，请在您的项目中安装 Sisk.BasicAuth 包：

```
> dotnet add package Sisk.BasicAuth
```

您可以在 [Nuget 存储库](#) 中查看更多关于如何在您的项目中安装它的信息。

创建身份验证处理器

您可以控制整个模块或个别路由的身份验证方案。为此，让我们首先编写第一个基本身份验证处理器。

在以下示例中，连接到数据库，检查用户是否存在以及密码是否有效，然后将用户存储在上下文包中。

```
1  public class UserAuthHandler : BasicAuthenticateRequestHandler
2  {
3      public UserAuthHandler() : base()
4      {
5          Realm = "要进入此页面，请提供您的凭据。";
6      }
7
8      public override HttpResponseMessage? OnValidating(BasicAuthenticationCredentials credentials,
```

```

9     HttpContext context)
10    {
11        DbContext db = new DbContext();
12
13        // 在这种情况下，我们使用电子邮件作为用户 ID 字段，因此我们将使用电子邮件查找用户。
14        User? user = db.Users.FirstOrDefault(u => u.Email == credentials.UserId);
15        if (user == null)
16        {
17            return base.CreateUnauthorizedResponse("抱歉！没有找到此电子邮件的用户。");
18        }
19
20        // 验证此用户的凭据密码是否有效。
21        if (!user.ValidatePassword(credentials.Password))
22        {
23            return base.CreateUnauthorizedResponse("无效的凭据。");
24        }
25
26        // 将已登录的用户添加到 HTTP 上下文中
27        // 并继续执行
28        context.Bag.Add("loggedUser", user);
29        return null;
30    }
}

```

因此，只需将此请求处理程序与我们的路由或类关联即可。

```

1  public class UsersController
2  {
3      [RouteGet("/")]
4      [RequestHandler(typeof(UserAuthHandler))]
5      public string Index(HttpContext request)
6      {
7          User loggedUser = (User)request.Context.RequestBag["loggedUser"];
8          return "您好， " + loggedUser.Name + "！";
9      }
10 }

```

或者使用 [RouterModule](#) 类：

```

1  public class UsersController : RouterModule
2  {
3      public ClientModule()
4      {
5          // 现在此类中的所有路由都将由 UserAuthHandler 处理。
6          base.HasRequestHandler(new UserAuthHandler());
}

```

```
7     }
8
9     [RouteGet( "/")]
10    public string Index(HttpRequest request)
11    {
12        User loggedUser = (User)request.Context.RequestBag["loggedUser"];
13        return "您好, " + loggedUser.Name + "!";
14    }
15 }
```

备注

基本身份验证的主要责任由客户端承担。存储、缓存控制和加密都在客户端本地处理。服务器只接收凭据并验证是否允许访问。

请注意，此方法不是最安全的，因为它将大量责任放在客户端上，客户端可能难以跟踪和维护其凭据的安全性。另外，密码必须在安全连接上下文（SSL）中传输，因为它们没有内置加密。请求头的简短拦截可能会暴露用户的访问凭据。

对于生产环境中的应用程序，请选择更强大的身份验证解决方案，并避免使用太多现成的组件，因为它们可能无法适应项目的需求，最终会使其暴露于安全风险之中。

服务提供者

服务提供者是一种将 Sisk 应用程序移植到不同环境的方式，使用一个可移植的配置文件。该功能允许您在不修改应用程序代码的情况下更改服务器的端口、参数和其他选项。该模块依赖于 Sisk 构造语法，可以通过 `UsePortableConfiguration` 方法进行配置。

一个配置提供者是通过 `IConfigurationProvider` 实现的，它提供了一个配置读取器，可以接受任何实现。默认情况下，Sisk 提供了一个 JSON 配置读取器，但也存在一个用于 INI 文件的包。您也可以创建自己的配置提供者并将其注册为：

```
1  using var app = HttpServer.CreateBuilder()
2      .UsePortableConfiguration(config =>
3      {
4          config.WithConfigReader<MyConfigurationReader>();
5      })
6      .Build();
```

如前所述，默认的提供者是一个 JSON 文件。默认情况下，文件名为 `service-config.json`，并在运行进程的当前目录中搜索，而不是可执行文件目录。

您可以选择更改文件名，以及 Sisk 应该在哪里查找配置文件，使用：

```
1  using Sisk.Core.Http;
2  using Sisk.Core.Http.Hosting;
3
4  using var app = HttpServer.CreateBuilder()
5      .UsePortableConfiguration(config =>
6      {
7          config.WithConfigFile("config.toml",
8              createIfDontExists: true,
9              lookupDirectories:
10                 ConfigurationFileLookupDirectory.CurrentDirectory |
11                 ConfigurationFileLookupDirectory.AppDirectory);
12      })
13      .Build();
```

上面的代码将在运行进程的当前目录中查找 `config.toml` 文件。如果找不到，它将在可执行文件所在的目录中搜索。如果文件不存在，`createIfDontExists` 参数将被尊重，创建一个空文件在最后测试的路径（基于 `lookupDirectories`），并在控制台中抛出一个错误，防止应用程序初始化。

TIP

您可以查看 INI 配置读取器和 JSON 配置读取器的源代码，以了解如何实现 `IConfigurationProvider`。

从 JSON 文件读取配置

默认情况下，Sisk 提供了一个配置提供者，用于从 JSON 文件读取配置。该文件遵循一个固定的结构，包含以下参数：

```
1  {
2      "Server": {
3          "DefaultEncoding": "UTF-8",
4          "ThrowExceptions": true,
5          "IncludeRequestIdHeader": true
6      },
7      "ListeningHost": {
8          "Label": "My sisk application",
9          "Ports": [
10             "http://localhost:80/",
11             "https://localhost:443/" // 配置文件也支持注释
12         ],
13         "CrossOriginResourceSharingPolicy": {
14             "AllowOrigin": "*",
15             "AllowOrigins": [ "*" ], // 新增于 0.14
16             "AllowMethods": [ "*" ],
17             "AllowHeaders": [ "*" ],
18             "MaxAge": 3600
19         },
20         "Parameters": {
21             "MySqlConnection": "server=localhost;user=root;"
22         }
23     }
24 }
```

从配置文件创建的参数可以在服务器构造函数中访问：

```
1  using var app = HttpServer.CreateBuilder()
2      .UsePortableConfiguration(config =>
3      {
4          config.WithParameters(paramCollection =>
```

```

5         {
6             string databaseConnection = paramCollection.GetValueOrDefault("MySqlConnection");
7         });
8     }
9     .Build();

```

每个配置读取器提供了一种读取服务器初始化参数的方式。一些属性被指示在进程环境中而不是在配置文件中定义，例如敏感的 API 数据、API 密钥等。

配置文件结构

JSON 配置文件由以下属性组成：

属性	必需	描述
Server	必 需	代表服务器本身及其设置。
Server.AccessLogsStream	可 选	默认为 <code>console</code> 。指定访问日志输出流。可以是文件名、 <code>null</code> 或 <code>console</code> 。
Server.ErrorsLogsStream	可 选	默认为 <code>null</code> 。指定错误日志输出流。可以是文件名、 <code>null</code> 或 <code>console</code> 。
Server.MaximumContentLength	可 选	默认为 <code>0</code> 。指定最大内容长度（以字节为单位）。零表示无限。
Server.IncludeRequestIdHeader	可 选	默认为 <code>false</code> 。指定是否应发送 <code>X-Request-Id</code> 标头。
Server.ThrowExceptions	可 选	默认为 <code>true</code> 。指定是否应抛出未处理的异常。设置为 <code>false</code> 时为生产环境， <code>true</code> 时为调试环境。
ListeningHost	必 需	代表服务器监听主机。
ListeningHost.Label	可	代表应用程序标签。

属性	必需	描述
	选	
ListeningHost.Ports	必需	代表一个字符串数组，匹配 ListeningPort 语法。
ListeningHost.CrossOriginResourceSharingPolicy	可选	设置应用程序的 CORS 标头。
ListeningHost.CrossOriginResourceSharingPolicy.AllowCredentials	可选	默认为 <code>false</code> 。指定 <code>Allow-Credentials</code> 标头。
ListeningHost.CrossOriginResourceSharingPolicy.ExposeHeaders	可选	默认为 <code>null</code> 。此属性期望一个字符串数组。指定 <code>Expose-Headers</code> 标头。
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigin	可选	默认为 <code>null</code> 。此属性期望一个字符串。指定 <code>Allow-Origin</code> 标头。
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigins	可选	默认为 <code>null</code> 。此属性期望一个字符串数组。指定多个 <code>Allow-Origin</code> 标头。请参阅 AllowOrigins 以获取更多信息。
ListeningHost.CrossOriginResourceSharingPolicy.AllowMethods	可选	默认为 <code>null</code> 。此属性期望一个字符串数组。指定 <code>Allow-Methods</code> 标头。
ListeningHost.CrossOriginResourceSharingPolicy.AllowHeaders	可选	默认为 <code>null</code> 。此属性期望一个字符串数组。指定 <code>Allow-Headers</code> 标头。
ListeningHost.CrossOriginResourceSharingPolicy.MaxAge	可选	默认为 <code>null</code> 。此属性期望一个整数。指定 <code>Max-Age</code> 标头（以秒为单位）。
ListeningHost.Parameters	可选	指定提供给应用程序设置方法的属性。

INI 配置提供程序

Sisk 有一种方法可以获取启动配置，而不仅仅是 JSON。事实上，任何实现 [IConfigurationReader](#) 的管道都可以与 [PortableConfigurationBuilder.WithConfigurationPipeline](#)一起使用，从任何文件类型中读取服务器配置。

[Sisk.IniConfiguration](#) 包提供了一个基于流的 INI 文件读取器，它不会为常见的语法错误抛出异常，并且具有简单的配置语法。这个包可以在 Sisk 框架之外使用，为需要高效 INI 文档读取器的项目提供了灵活性。

安装

要安装包，可以从以下开始：

```
$ dotnet add package Sisk.IniConfiguration
```

你也可以安装核心包，它不包括 INI [IConfigurationReader](#)，也不包括 Sisk 依赖项，只包括 INI 序列化器：

```
$ dotnet add package Sisk.IniConfiguration.Core
```

使用主包，你可以在代码中使用它，如下面的示例所示：

```
1  class Program
2  {
3      static HttpServerHostContext Host = null!;
4
5      static void Main(string[] args)
6      {
7          Host = HttpServer.CreateBuilder()
8              .UsePortableConfiguration(config =>
9              {
10                  config.WithConfigFile("app.ini", createIfDontExists: true);
11
12                  // 使用 IniConfigurationReader 配置读取器
13                  config.WithConfigurationPipeline<IniConfigurationReader>();
14              })
15              .UseRouter(r =>
```

```

16         {
17             r.MapGet("/", SayHello);
18         })
19     .Build();
20
21     Host.Start();
22 }
23
24     static HttpResponse SayHello(HttpRequest request)
25     {
26         string? name = Host.Parameters["name"] ?? "world";
27         return new HttpResponse($"Hello, {name}!");
28     }
29 }
```

上面的代码将在进程的当前目录 (CurrentDirectory) 中查找一个 app.ini 文件。INI 文件的内容如下：

```

1 [Server]
2 # 支持多个监听地址
3 Listen = http://localhost:5552/
4 Listen = http://localhost:5553/
5 ThrowExceptions = false
6 AccessLogsStream = console
7
8 [Cors]
9 AllowMethods = GET, POST
10 AllowHeaders = Content-Type, Authorization
11 AllowOrigin = *
12
13 [Parameters]
14 Name = "Kanye West"
```

INI 风格和语法

当前实现风格：

- 属性和节名是 **大小写不敏感** 的。
- 属性名和值是 **修剪** 的，除非值被引号括起来。
- 值可以用单引号或双引号括起来。引号内可以包含换行符。
- 支持使用 `#` 和 `;` 的注释。**尾随注释也是允许的**。

- 属性可以有多个值。

详细信息，Sisk 中使用的 INI 解析器的“风格”文档可以在 [这里](#) 找到。

使用以下 INI 代码作为示例：

```

1  One = 1
2  Value = 这是一个值
3  Another value = "这个值
4      有一个换行符"
5  ; 下面的代码有一些颜色
6  [some section]
7  Color = Red
8  Color = Blue
9  Color = Yellow ; 不要使用黄色

```

解析它：

```

1 // 从字符串解析 INI 文本
2 IniDocument doc = IniDocument.FromString(iniText);
3
4 // 获取一个值
5 string? one = doc.Global.GetOne("one");
6 string? anotherValue = doc.Global.GetOne("another value");
7
8 // 获取多个值
9 string[]? colors = doc.GetSection("some section")?.GetMany("color");

```

配置参数

节和名称	允许多个值	描述
Server.Listen	是	服务器监听地址/端口。
Server.Encoding	否	服务器默认编码。
Server.MaximumContentLength	否	服务器最大内容长度（以字节为单位）。
Server.IncludeRequestIdHeader	否	指定是否应发送 X-Request-Id 标头。

节和名称	允许多个值	描述
Server.ThrowExceptions	否	指定是否应抛出未处理的异常。
Server.AccessLogsStream	否	指定访问日志输出流。
Server.ErrorsLogsStream	否	指定错误日志输出流。
Cors.AllowMethods	否	指定 CORS Allow-Methods 标头值。
Cors.AllowHeaders	否	指定 CORS Allow-Headers 标头值。
Cors.AllowOrigins	否	指定多个 Allow-Origin 标头，逗号分隔。 AllowOrigins 有更多信息。
Cors.AllowOrigin	否	指定一个 Allow-Origin 标头。
Cors.ExposeHeaders	否	指定 CORS Expose-Headers 标头值。
Cors.AllowCredentials	否	指定 CORS Allow-Credentials 标头值。
Cors.MaxAge	否	指定 CORS Max-Age 标头值。

手动（高级）设置

在本节中，我们将创建一个没有任何预定义标准的 HTTP 服务器，以完全抽象的方式。这里，您可以手动构建您的 HTTP 服务器的功能。每个 ListeningHost 都有一个路由器，一个 HTTP 服务器可以有多个 ListeningHost，每个 ListeningHost 指向不同的主机和端口。

首先，我们需要了解请求/响应概念。它非常简单：对于每个请求，必须有一个响应。Sisk 也遵循这个原则。让我们创建一个方法，响应一个“Hello, World！”消息，指定状态代码和头部。

```
1 // Program.cs
2 using Sisk.Core.Http;
3 using Sisk.Core.Routing;
4
5 static HttpResponse IndexPage(HttpRequest request)
6 {
7     HttpResponse indexResponse = new HttpResponse
8     {
9         Status = System.Net HttpStatusCode.OK,
10        Content = new HtmlContent(@"
11            <html>
12                <body>
13                    <h1>Hello, world!</h1>
14                </body>
15            </html>
16        ")
17    };
18
19    return indexResponse;
20 }
```

下一步是将此方法与一个 HTTP 路由关联起来。

路由器

路由器是请求路由的抽象，作为服务的请求和响应之间的桥梁。路由器管理服务路由、函数和错误。

一个路由器可以有多个路由，每个路由可以在该路径上执行不同的操作，例如执行函数、提供页面或提供服务器资源。

让我们创建我们的第一个路由器，并将我们的 `PageIndex` 方法与索引路径关联起来。

```
1 Router mainRouter = new Router();
2
3 // SetRoute 将所有索引路由与我们的方法关联起来。
4 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage);
```

现在我们的路由器可以接收请求并发送响应。然而，`mainRouter` 不绑定到任何主机或服务器，因此它不能单独工作。下一步是创建我们的 `ListeningHost`。

Listening Hosts 和 Ports

一个 `ListeningHost` 可以托管一个路由器和多个监听端口，用于同一个路由器。一个 `ListeningPort` 是 HTTP 服务器将监听的前缀。

这里，我们可以创建一个 `ListeningHost`，它指向两个端点，用于我们的路由器：

```
1 ListeningHost myHost = new ListeningHost
2 {
3     Router = new Router(),
4     Ports = new ListeningPort[]
5     {
6         new ListeningPort("http://localhost:5000/")
7     }
8 };
```

现在我们的 HTTP 服务器将监听指定的端点，并将其请求重定向到我们的路由器。

服务器配置

服务器配置负责大部分 HTTP 服务器自身的行为。在此配置中，我们可以将 `ListeningHosts` 关联到我们的服务器。

```
1 HttpServerConfiguration config = new HttpServerConfiguration();
2 config.ListeningHosts.Add(myHost); // 将我们的 ListeningHost 添加到此服务器配置
```

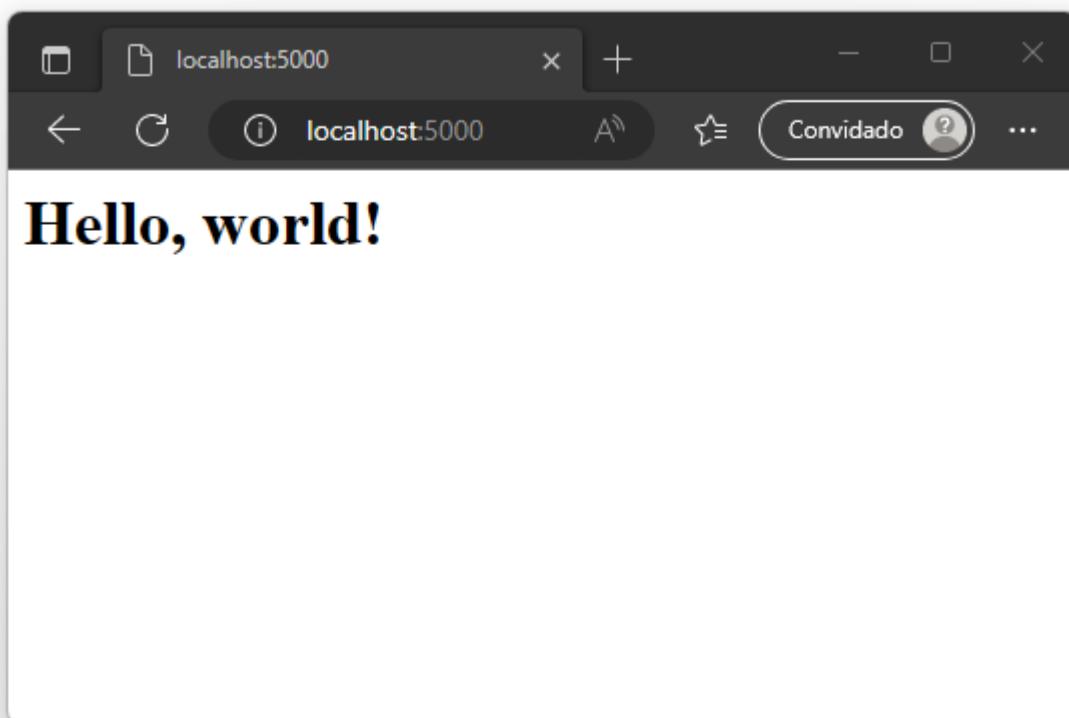
接下来，我们可以创建我们的 HTTP 服务器：

```
1 HttpServer server = new HttpServer(config);
2 server.Start(); // 启动服务器
3 Console.ReadKey(); // 防止应用程序退出
```

现在我们可以编译我们的可执行文件，并使用以下命令运行我们的 HTTP 服务器：

```
dotnet watch
```

在运行时，打开您的浏览器，并导航到服务器路径，您应该会看到：



请求生命周期

以下是通过一个 HTTP 请求示例解释的整个请求生命周期。

- **接收请求：**每个请求都会在请求本身和将要发送给客户端的响应之间创建一个 HTTP 上下文。这个上下文来自 Sisk 中的内置监听器，可以是 [HttpListener](#)、[Kestrel](#) 或 [Cadente](#)。
 - 外部请求验证：验证 [HttpServerConfiguration.RemoteRequestsAction](#) 的请求。
 - 如果请求是外部的，并且该属性是 `Drop`，则在不向客户端发送响应的情况下关闭连接，
`HttpServerExecutionStatus = RemoteRequestDropped`。
 - 转发解析器配置：如果配置了 [ForwardingResolver](#)，它将在请求的原始主机上调用 [OnResolveRequestHost](#) 方法。
 - DNS 匹配：在解析主机并配置了多个 [ListeningHost](#) 的情况下，服务器将查找对应的主机。
 - 如果没有 [ListeningHost](#) 匹配，返回 400 Bad Request 响应给客户端，
`HttpServerExecutionStatus = DnsUnknownHost` 状态返回给 HTTP 上下文。
 - 如果 [ListeningHost](#) 匹配，但其 [Router](#) 尚未初始化，返回 503 Service Unavailable 响应给客户端，
`HttpServerExecutionStatus = ListeningHostNotReady` 状态返回给 HTTP 上下文。
 - 路由器绑定：将 [ListeningHost](#) 的路由器与接收的 HTTP 服务器关联。
 - 如果路由器已经与另一个 HTTP 服务器关联，这是不允许的，因为路由器正在使用服务器的配置资源，会抛出 [InvalidOperationException](#)。这只会在 HTTP 服务器初始化期间发生，而不是在创建 HTTP 上下文期间。
 - 预定义头部：
 - 如果配置了，预定义响应中的 `X-Request-Id` 头部。
 - 如果配置了，预定义响应中的 `X-Powered-By` 头部。
 - 内容大小验证：验证请求内容是否小于 [HttpServerConfiguration.MaximumContentLength](#)，仅当其大于零时。
 - 如果请求发送的 `Content-Length` 大于配置的，返回 413 Payload Too Large 响应给客户端，
`HttpServerExecutionStatus = ContentTooLarge` 状态返回给 HTTP 上下文。
- **路由操作：**服务器为接收的请求调用路由器。
 - 如果路由器找不到匹配请求的路由：
 - 如果 [Router.NotFoundErrorHandler](#) 属性配置了，调用该操作，并将操作的响应转发给 HTTP 客户端。
 - 如果前面的属性为 `null`，返回默认的 404 Not Found 响应给客户端。
 - 如果路由器找到匹配的路由，但路由的方法不匹配请求的方法：
 - 如果 [Router.MethodNotAllowedErrorHandler](#) 属性配置了，调用该操作，并将操作的响应转发给 HTTP 客户端。
 - 如果前面的属性为 `null`，返回默认的 405 Method Not Allowed 响应给客户端。

- 如果请求的方法是 `OPTIONS`：
 - 路由器仅当没有路由匹配请求方法（路由的方法不是显式 `RouteMethod.Options`）时，返回 `200 Ok` 响应给客户端。
- 如果 `HttpServerConfiguration.ForceTrailingSlash` 属性启用了，匹配的路由不是正则表达式，请求路径不以 `/` 结尾，且请求方法是 `GET`：
 - 返回 `307 Temporary Redirect` HTTP 响应给客户端，`Location` 头部包含路径和查询，以相同的位置但在结尾添加 `/`。
- 为所有配置的 HTTP 服务器处理程序调用 `OnContextBagCreated` 事件。
- 执行所有全局 `IRequestHandler` 实例，具有 `BeforeResponse` 标志。
 - 如果任何处理程序返回非 `null` 响应，则该响应转发给 HTTP 客户端，且上下文关闭。
 - 如果此步骤中抛出错误，并且 `HttpServerConfiguration.ThrowExceptions` 禁用了：
 - 如果 `Router.CallbackErrorHandler` 属性启用了，调用它，并将结果响应返回给客户端。
 - 如果前面的属性未定义，返回空响应给服务器，服务器根据抛出的异常类型转发响应，通常为 `500 Internal Server Error`。
- 执行所有在路由中定义的 `IRequestHandler` 实例，具有 `BeforeResponse` 标志。
 - 如果任何处理程序返回非 `null` 响应，则该响应转发给 HTTP 客户端，且上下文关闭。
 - 如果此步骤中抛出错误，并且 `HttpServerConfiguration.ThrowExceptions` 禁用了：
 - 如果 `Router.CallbackErrorHandler` 属性启用了，调用它，并将结果响应返回给客户端。
 - 如果前面的属性未定义，返回空响应给服务器，服务器根据抛出的异常类型转发响应，通常为 `500 Internal Server Error`。
- 调用路由器的操作，并将其转换为 HTTP 响应。
 - 如果此步骤中抛出错误，并且 `HttpServerConfiguration.ThrowExceptions` 禁用了：
 - 如果 `Router.CallbackErrorHandler` 属性启用了，调用它，并将结果响应返回给客户端。
 - 如果前面的属性未定义，返回空响应给服务器，服务器根据抛出的异常类型转发响应，通常为 `500 Internal Server Error`。
- 执行所有全局 `IRequestHandler` 实例，具有 `AfterResponse` 标志。
 - 如果任何处理程序返回非 `null` 响应，则处理程序的响应替换前一个响应，并立即转发给 HTTP 客户端。
 - 如果此步骤中抛出错误，并且 `HttpServerConfiguration.ThrowExceptions` 禁用了：
 - 如果 `Router.CallbackErrorHandler` 属性启用了，调用它，并将结果响应返回给客户端。
 - 如果前面的属性未定义，返回空响应给服务器，服务器根据抛出的异常类型转发响应，通常为 `500 Internal Server Error`。
- 执行所有在路由中定义的 `IRequestHandler` 实例，具有 `AfterResponse` 标志。
 - 如果任何处理程序返回非 `null` 响应，则处理程序的响应替换前一个响应，并立即转发给 HTTP 客户端。
 - 如果此步骤中抛出错误，并且 `HttpServerConfiguration.ThrowExceptions` 禁用了：
 - 如果 `Router.CallbackErrorHandler` 属性启用了，调用它，并将结果响应返回给客户端。
 - 如果前面的属性未定义，返回空响应给服务器，服务器根据抛出的异常类型转发响应，通常为 `500 Internal Server Error`。
- **处理响应：**准备响应后，服务器为发送给客户端做准备。

- 根据 [ListeningHost.CrossOriginResourceSharingPolicy](#) 的配置，在响应中定义跨源资源共享（CORS）头部。
- 发送响应的状态代码和头部给客户端。
- 发送响应内容给客户端：
 - 如果响应内容是 [ByteArrayContent](#) 的后代，则直接将响应字节复制到响应输出流中。
 - 如果前面的条件不满足，响应被序列化为流并复制到响应输出流中。
- 关闭流并丢弃响应内容。
- 如果 [HttpServerConfiguration.DisposeDisposableContextValues](#) 启用了，丢弃所有在请求上下文中定义的继承自 [IDisposable](#) 的对象。
- 为所有配置的 HTTP 服务器处理程序调用 [OnHttpRequestClose](#) 事件。
- 如果服务器抛出了异常，为所有配置的 HTTP 服务器处理程序调用 [OnException](#) 事件。
- 如果路由允许访问日志记录，并且 [HttpServerConfiguration.AccessLogsStream](#) 不为 null，写入一行日志到日志输出中。
- 如果路由允许错误日志记录，并且存在异常，并且 [HttpServerConfiguration.ErrorsLogsStream](#) 不为 null，写入一行日志到错误日志输出中。
- 如果服务器通过 [HttpServer.WaitNext](#) 等待请求，释放互斥锁，且上下文变得可用于用户。

转发解析器

转发解析器是一个帮助器，帮助解码通过请求、代理、CDN 或负载均衡器识别客户端的信息。当您的 Sisk 服务运行通过反向或正向代理时，客户端的 IP 地址、主机和协议可能与原始请求不同，因为这是从一个服务到另一个服务的转发。这个 Sisk 功能允许您在处理请求之前控制和解析此信息。这些代理通常提供有用的头部来识别其客户端。

目前，使用 [ForwardingResolver](#) 类，可以解析客户端 IP 地址、主机和使用的 HTTP 协议。在 Sisk 1.0 版本之后，服务器不再有标准实现来解码这些头部，因为安全原因从服务到服务不同。

例如，`X-Forwarded-For` 头部包含有关转发请求的 IP 地址的信息。这个头部由代理使用，以携带一系列信息到最终服务，并包括所有使用的代理的 IP 地址，包括客户端的真实地址。问题是：有时很难识别客户端的远程 IP 地址，并且没有特定的规则来识别这个头部。强烈推荐阅读以下头部的文档：

- 阅读关于 `X-Forwarded-For` 头部的信息 [这里](#)。
- 阅读关于 `X-Forwarded-Host` 头部的信息 [这里](#)。
- 阅读关于 `X-Forwarded-Proto` 头部的信息 [这里](#)。

ForwardingResolver 类

这个类有三个虚拟方法，允许为每个服务实现最合适的解决方案。每个方法负责通过代理解析请求的信息：客户端的 IP 地址、请求的主机和使用的安全协议。默认情况下，Sisk 将始终使用原始请求的信息，而不解析任何头部。

下面的例子展示了如何使用这个实现。这个例子通过 `X-Forwarded-For` 头部解析客户端的 IP 地址，并在请求中转发多个 IP 地址时抛出错误。

✖️ IMPORTANT

不要在生产代码中使用这个例子。始终检查实现是否适合使用。在实现之前阅读头部文档。

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          using var host = HttpServer.CreateBuilder()
6              .UseForwardingResolver<Resolver>()
7              .UseListeningPort(5555)
```

```
8         .Build();
9
10        host.Router.SetRoute(RouteMethod.Any, Route.AnyPath, request =>
11            new HttpResponse("Hello, world!!!"));
12
13        host.Start();
14    }
15
16    class Resolver : ForwardingResolver
17    {
18        public override IPAddress OnResolveClientAddress(HttpContext request,
19        IPEndPoint connectingEndpoint)
20        {
21            string? forwardedFor = request.Headers.XForwardedFor;
22            if (forwardedFor is null)
23            {
24                throw new Exception("X-Forwarded-For 头部缺失。");
25            }
26            string[] ipAddresses = forwardedFor.Split(',');
27            if (ipAddresses.Length != 1)
28            {
29                throw new Exception("X-Forwarded-For 头部中有太多地址。");
30            }
31
32            return IPAddress.Parse(ipAddresses[0]);
33        }
34    }
}
```

HTTP 服务器处理器

在 Sisk 0.16 版本中，我们引入了 `HttpServerHandler` 类，该类旨在扩展 Sisk 的整体行为，并为 Sisk 提供额外的事件处理程序，例如处理 HTTP 请求、路由器、上下文包等。

该类集中了整个 HTTP 服务器和请求的生命周期中发生的事件。HTTP 协议没有会话，因此无法在请求之间保留信息。Sisk 目前提供了一种方式，允许您实现会话、上下文、数据库连接和其他有用的提供程序，以帮助您的工作。

请参阅 [此页面](#) 以了解每个事件的触发位置和其目的。你也可以查看 [HTTP 请求的生命周期](#) 以了解请求发生了什么以及事件在哪里触发。HTTP 服务器允许你同时使用多个处理程序。每个事件调用都是同步的，即它将阻塞当前线程，直到与该函数关联的所有处理程序都执行并完成。

与 `RequestHandlers` 不同，它们不能应用于某些路由组或特定路由。相反，它们应用于整个 HTTP 服务器。你可以在你的 HTTP 服务器处理程序中应用条件。此外，每个 Sisk 应用程序都定义了每个 `HttpServerHandler` 的单例，因此每个 `HttpServerHandler` 只有一个实例。

使用 `HttpServerHandler` 的一个实用示例是自动在请求结束时释放数据库连接。

```
1 // DatabaseConnectionHandler.cs
2
3 public class DatabaseConnectionHandler : HttpServerHandler
4 {
5     public override void OnHttpRequestClose(HttpServerExecutionResult result)
6     {
7         var requestBag = result.Request.Context.RequestBag;
8
9         // 检查请求是否在其上下文包中定义了 DbContext
10        if (requestBag.IsSet<DbContext>())
11        {
12            var db = requestBag.Get<DbContext>();
13            db.Dispose();
14        }
15    }
16 }
17
18 public static class DatabaseConnectionHandlerExtensions
19 {
20     // 允许用户从 HTTP 请求创建 DbContext 并将其存储在其请求包中
21     public static DbContext GetDbContext(this HttpRequest request)
22     {
23         var db = new DbContext();
24         return request.SetContextBag<DbContext>(db);
```

```
25     }
26 }
```

上面的代码中，`GetDbContext` 扩展方法允许直接从 `HttpRequest` 对象创建连接上下文并将其存储在其请求包中。未释放的连接可能会在运行数据库时引起问题，因此在 `OnHttpRequestClose` 中终止它。

你可以在你的构建器中或直接使用 [HttpServer.RegisterHandler](#) 注册处理程序到 HTTP 服务器。

```
1 // Program.cs
2
3 class Program
4 {
5     static void Main(string[] args)
6     {
7         using var app = HttpServer.CreateBuilder()
8             .UseHandler<DatabaseConnectionHandler>()
9             .Build();
10
11         app.Router.SetObject(new UserController());
12         app.Start();
13     }
14 }
```

这样，`UserController` 类就可以使用数据库上下文：

```
1 // UserController.cs
2
3 [RoutePrefix("/users")]
4 public class UserController : ApiController
5 {
6     [RouteGet()]
7     public async Task<HttpResponse> List(HttpContext request)
8     {
9         var db = request.GetDbContext();
10        var users = db.Users.ToArray();
11
12        return JsonOk(users);
13    }
14
15    [RouteGet("<id>")]
16    public async Task<HttpResponse> View(HttpContext request)
17    {
18        var db = request.GetDbContext();
19
20        var userId = request.GetQueryValue<int>("id");
```

```

21         var user = db.Users.FirstOrDefault(u => u.Id == userId);
22
23         return JsonOk(user);
24     }
25
26     [RoutePost]
27     public async Task<HttpResponse> Create(HttpRequest request)
28     {
29         var db = request.GetDbContext();
30         var user = JsonSerializer.Deserialize<User>(request.Body);
31
32         ArgumentNullException.ThrowIfNull(user);
33
34         db.Users.Add(user);
35         await db.SaveChangesAsync();
36
37         return JsonMessage("User added.");
38     }
39 }
```

上面的代码使用了 ApiController 中的 JsonOk 和 JsonMessage 方法，该方法继承自 RouterController：

```

1 // ApiController.cs
2
3 public class ApiController : RouterModule
4 {
5     public HttpResponse JsonOk(object value)
6     {
7         return new HttpResponse(200)
8             .WithContent(JsonContent.Create(value, null, new JsonSerializerOptions()
9                 {
10                     PropertyNameCaseInsensitive = true
11                 }));
12     }
13
14     public HttpResponse JsonMessage(string message, int statusCode = 200)
15     {
16         return new HttpResponse(statusCode)
17             .WithContent(JsonContent.Create(new
18                 {
19                     Message = message
20                 }));
21     }
22 }
```

开发人员可以使用此类实现会话、上下文和数据库连接。提供的代码展示了一个使用 `DatabaseConnectionHandler` 的实用示例，自动在每个请求结束时释放数据库连接。

集成非常简单，处理程序在服务器设置期间注册。`HttpServerHandler` 类提供了一个强大的工具集，用于在 HTTP 应用程序中管理资源和扩展 `Sisk` 行为。

每个服务器多个监听主机

Sisk Framework一直支持在每个服务器上使用多个主机，即单个HTTP服务器可以监听多个端口，每个端口都有自己的路由器和服务运行在上面。

这样，使用Sisk就可以轻松地分离责任并管理单个HTTP服务器上的服务。下面的例子展示了创建两个 ListeningHosts，每个监听不同的端口，具有不同的路由器和动作。

阅读[手动创建您的应用](#)以了解有关此抽象的详细信息。

```
1  static void Main(string[] args)
2  {
3      // 创建两个监听主机，每个都有自己的路由器和监听端口
4      //
5      ListeningHost hostA = new ListeningHost();
6      hostA.Ports = [new ListeningPort(12000)];
7      hostA.Router = new Router();
8      hostA.Router.SetRoute(RouteMethod.Get, "/", request => new
9      HttpResponseMessage().WithContent("来自主机A的问候! "));
10
11     ListeningHost hostB = new ListeningHost();
12     hostB.Ports = [new ListeningPort(12001)];
13     hostB.Router = new Router();
14     hostB.Router.SetRoute(RouteMethod.Get, "/", request => new
15     HttpResponseMessage().WithContent("来自主机B的问候! "));
16
17     // 创建一个服务器配置并添加两个监听主机
18     //
19     HttpServerConfiguration configuration = new HttpServerConfiguration();
20     configuration.ListeningHosts.Add(hostA);
21     configuration.ListeningHosts.Add(hostB);
22
23     // 创建一个使用指定配置的HTTP服务器
24     //
25     HttpServer server = new HttpServer(configuration);
26
27     // 启动服务器
28     server.Start();
29
30     Console.WriteLine("尝试访问主机A在 {0}", server.ListeningPrefixes[0]);
31     Console.WriteLine("尝试访问主机B在 {0}", server.ListeningPrefixes[1]);
32
```

```
    Thread.Sleep(-1);  
}
```