**第一章 NET Core与异步编程**

2020年1月6日

9:55

# 第一部分、微服务

Microservices Architecture

什么是微服务？

一种架构模式，它提倡将单一的应用程序划分为一组小的服务，服务之间互相协调，互相配合。每个服务运行再独立的进程，服务之间采用轻量级的沟通机制互相沟通。（基于HTTP的RESTful API）

每个服务都围绕具体业务构建，能独立部署再生产环境，预发布环境等

适合大型互联网应用业务开发（电商、）

特征

微服务和IaaS（基础设施即服务）结合

微服务和PaaS（平台即服务）结合

微服务和SaaS(容器即服务)结合

# 第二部分、NET Core平台简介和安装

为了应对云计算和微服务等新一代运维和开发的需要，微软推出了新一代的.NET Core平台来满足服务实现需要

NET Core平台是适用于Windows、Mac OSX和Linux操作系统的免费、开源托管的计算机软件平台，是微软开发的第一个官方版本。

**一、.NET Core 平台组成**

.NET Core运行时

ASP.NET Core运行时

.NET Core CLI工具和语言编译器

dotnet工具

**二、安装**

**1、独立安装**

下载：从微软官方网站（https://dotnet.microsoft.com/download）上下载.NET Core SDK安装包，再在当前计算机中安装。

安装完毕后，在命令提示符下使用 dotnet –version 命令查看.NET Core SDK版本号，如果有版本号表示安装成功

因为.NET Core更新迭代较快，建议安装并使用.NET Core长期支持稳定版，目前.NET Core的使用版本长期支持稳定版是3.x。

**2、VS安装**

Visual Studio 2017中只支持.NET Core SDK 2.2，Visual Studio 2019中才能支持.NET Core SDK3.0。



**三、使用vs创建core应用（可以选择创建如下应用）**

REJ€TN(.NET Сосе) 
соте 
Илих macOS Window•s 

ASP.NET core web 
.NET core .NET Framework Windowss Linuxn macos ASP.NET core web wel 
AZ React Redux, VVC 
C# 
Linux macOS Windows 
Web 



gvc 
.NET ASP.NET 
Linux 
macOS Windows Web 

# 第三部分 异步编程

**一、NET异步编程模式**

异步编程模型（Asynchronous Programming Model，APM）

基于事件的异步模式（Event-based Asynchronous Pattern，EAP）

基于任务的异步模式（Task-based Asynchronous Pattern，TAP）

**二、基于任务的异步模式**

基于任务的异步模式是基于

using System.Threading;

using System.Threading.Tasks;

为了简化异步编程的过程，.NET Framework 4.5在C#中引入了await和async关键字

使用async关键字定义的方法通常包含一个或多个await关键字

await仅可用于由async关键字修改的异步方法中，使用await运算符的任务通常是实现基于任务的异步模式的方法调用返回，返回值包括Task、Task<TResult>、ValueTask和ValueTask<TResult>对象的法。

微服务选择

1、跨平台语言 如.net Core

<https://dotnet.microsoft.com/download/dotnet-core/3.1>

2、协议RestFull API ,grpc

3、部署Docker

微软文档

<https://docs.microsoft.com/zh-cn/dotnet/>

第二章 NET Core基础

2020年8月11日

7:54

# 第一部分、项目文档

注意：Core仅支持 web api和 web mvc，不支持web form框架

在ASP.NET Core Web应用程序项目使用**appsettings.json配置文件**。。

JavaScript、图片和样式表文件都存放在**wwwroot文件夹**下对应的子文件夹中。

**路由配置、中间件配置、依赖配置都在Startup.cs文件中**

# 第二部分、ASP.NET的启动过程

**Core应用是独立的，本质上是一个独立的控制台应用程序，不必须由IIS内部托管运行。其有一个内部的自托管（Self-Hosted）的web服务器，这个服务器叫做Kestrel Web服务器。支持.net core的所有平台和版本均支持。**

**一、Startup类**

从Program.cs文件可以看到，Startup类是应用程序的入口。

**可以设置配置服务。**

**可以配置请求管道，该请求管道将用于处理应用程序的所有请求。**

**1、Configure()方法：**

**Configure()指定ASP.Net Core应用程序如何相应每一个HTTP请求**

实现功能可以通过管道配置来实现，管道配置可以封装到中间件中并通过扩展方法添加到IApplicationBuilder上。（IApplicationBuilder是Configure()方法必须的一个参数）

每个User扩展方法都会把一个中间件请求添加到请求管道。

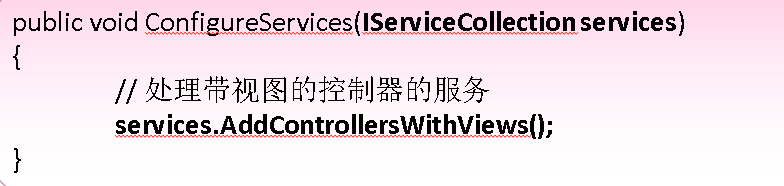
这样应用程序所有请求都会通过扩展的中间件进行处理。

public void 'WebHostEnvirpnment eny) 
IfiJ 

**2、ConfigureService()方法用来配置用于应用程序内的服务。**

ConfigureServices()方法通过参数获取一个实现IServiceCollection接口的实例。

IServiceCollection实例包含很多Add{Service}扩展方法，通过依赖注入可将服务加入Web服务容器，使其在应用程序中可用。



注意：先在ConfigureService()方法注入依赖关系，使其在应用程序中可用，然后在Configure()方法中调用扩展方法，添加到请求管道来处理所有的应用程序请求

# 第三部分、跨域请求

相同的源是指如果两个url地址具有相同的协议，主机名和端口，则它们具有相同的源。

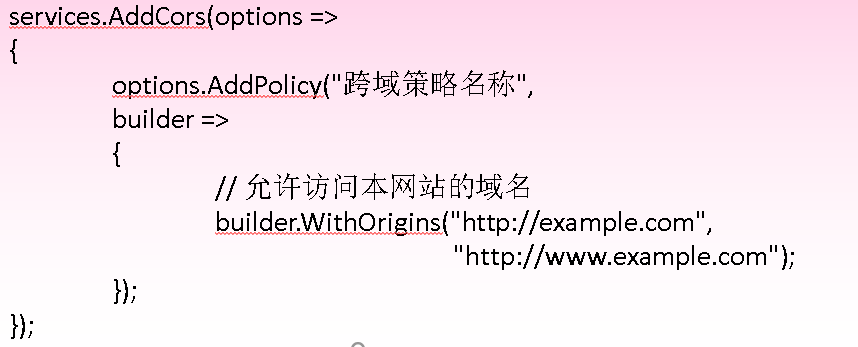
出于对客户访问安全的考虑，浏览器会阻止网页向其他网站（即其他区网络域）发送请求，此限制称为同源策略

同源是指如果两个URL地址具有相同协议、主机名和端口，则它们具有相同的源。

如果需要读取其他网站中的数据，需要允许其他站点对开发者的应用进行跨域请求，如在微服务中调用其他服务。

**一、实现步骤**

**1、在ConfigureServices()方法调用AddCors()方法添加跨域策略**



**builder对象可以设置的cors策略选项**

|  |  |
| --- | --- |
| 策略名 | 说明 |
| AllowAnyOrigin | 允许所有来源请求，不安全 |
| AllowAnyHeader | 允许任何请求标头 |
| AllowAnyMethod | 允许任何HTTP方法 |
| WithOrigins | 允许特定来源的CORS请求 |
| WithHeaders | 允许在CORS请求中发送特定标头 |
| WithExposedHeaders | 设置公开的响应标头 |
| AllowCredentials | 允许所有来源跨域请求中的凭据 |

**2、在Configure()方法中通过CORS中间件将CORS策略应用到所有应用终结点，语法代码如下：**

* + public void IWebHostEnvironmentenv) 
    app. UseCors(); 
    end oints.Ma 

**二、实践 在Core API提供数据的时候，需要允许跨域请求**

public class Startup

{

...省略代码...

readonly string MyCorsPolicy = "CorsPolicy";// 声明跨域策略名称

public void ConfigureServices(IServiceCollection services) {

services.AddCors(options =>

{ // 添加跨域服务

options.AddPolicy(MyCorsPolicy, builder =>

{ // 声明跨域策略

builder.AllowAnyOrigin().AllowAnyHeader().WithMethods("GET", "POST", "HEAD", "PUT", "DELETE", "OPTIONS");

});

});

services.AddControllers();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env){

...省略代码...

app.UseCors();// 使用允许跨域请求

app.UseEndpoints(endpoints =>{ // 所有请求都应用跨域请求策略

endpoints.MapControllers().RequireCors(MyCorsPolicy);

});

}

}

**三、访问api**

**1、在vue中使用ajaox**

<link rel="stylesheet" href="./css/antd.min.css">

<script src="./js/vue.min.js"></script>

<script src="./js/antd.min.js"></script>

<script src="./js/axios.min.js"></script>

created: function () {

var config = {

method: 'get',

url: '<https://localhost:44311/api/travel>',

headers: {

'Content-Type': 'application/x-www-form-urlencoded'

}

};

\_this=this;

axios(config)

.then(function (response) {

\_this.tlist=response.data;

console.info(response.data);

})

.catch(function (error) {

\_this.tlist=["未能连接到服务器"];

});

}

# 第四部分 依赖注入

**一、概念**

依赖注入（Dependency injection，DI）是一种实现对象及其合作者或依赖项之间松散耦合的技术。

* + 商 品 信 息 类 
    1 、 创 建 商 品 信 息 对 象 
    商 品 类 别 类 
    2 ． 看 商 品 信 息 对 象 是 否 有 依 赖 对 象 需 要 注 入 
    3 、 有 需 要 容 器 创 建 商 品 类 别 对 象 
    并 注 入 到 商 品 信 息 对 象 
    4 ． 容 器 管 畦 这 些 对 象 的 生 命 周 期 
    客 户 端 类 
    直 接 从 D 《 容 器 获 取 商 品 信 息 对 象 

通常类会通过它们的构造函数声明其依赖关系，允许它们遵循显示依赖原则，这种方法被称为“构造函数注入”

**二、实现步骤**

**1、使用接口或基类抽象化依赖关系实现**

先创建一个名为IProductService的商品服务接口，其中定义获取全部商品信息的GetProductList()方法

* + public interface IProductService 
    GetProductList(); 

**2.再创建ProductService类来实现IProductService商品服务接口的GetProductList()方法初始化商品信息列表**

* + public class ProductService : IProductService 
    public 
    return new List<Product> 
    new Product{ ProductlDF1, 
    new Product{ ProductlD=2, 
    new Product{ ProductlD=3, ProductName—" 
    itJ5007RIOOo 

**3、注册服务容器中的依赖关系，ASP.NET Core提供了一个内置的服务容器IServiceProvider，服务在应用的Startup.ConfigureServices()方法中注册**

* + public class Startup 
    public void ConfigureServices(lServiceCollection services) 
    nt<UProductService, 

**4、将服务注入到使用它的类的构造函数中**

* + public class Sho Controller : Controller 
    'Productservice_service; 
    pu b lic ShopController(JProductService se rv ice) 
    service = service; 
    public IActionResult Index() 
    — _service.GetProductList(); 
    return View(); 

第三章 NET Core高级

2020年8月11日

10:26

# 第一部分 ASP.NET Core 配置

**一、概述**

ASP.NET Core Web应用程序项目中包含三个配置文件：

launchSettings.json是启动配置文件。

appsettings.json是程序运行时应用的配置文件。

appsettings.Development.json是程序调试时（即开发时）应用的配置文件。

**二、步骤(读取配置文件)**

ASP.NET Core应用的依赖注入容器中，提供了实现IConfiguration的对象，可以注入到MVC的控制器中，然后通过IConfiguration对象读取配置信息。

**1、appsettings.json**

"SiteConfig": { 
"Name": " 商 城 后 台 管 理 " ， 
"Copyright": " 版 权 所 有 ： 商 城 软 件 开 发 组 " ， 
“ ． 省 略 代 码 ． 

**2、因为SiteConfig是自定义的配置，所以需要创建对应配置类来保存配置数据**

namespace SettingDemo.Models 
public class SiteConfig 
public string Name { get; set; } 
public string Copyright { get; set; } 

**3、在Controllers中创建ShopController控制器，并在ShopController控制器构造方法中注入IConfiguration的对象，通过IConfiguration.GetSection()方法读取配置键。使用Get<T>()方法绑定配置的值到SiteConfig对象中。**

using Microsoft.Extensions.Configuration; 
using SettingDemo. Models; 
public class ShopController : Controller 
private 'Configuration config; 
public ShopController(lConfiguration configuration) // 
config = configuration; 
public IActionResult Index() 
SiteConfig siteConfig = 
return View(siteConfig); 

4、注意，修改json文件的编码格式为utf-8

0 在 浏 览 器 中 显 示 的 结 果 有 中 文 乱 码 ， 是 因 为 appsettings.json 的 文 档 编 码 是 
GB2312 ， 这 里 需 要 通 过 记 事 本 或 N 。 tepad + + 等 工 具 将 文 档 编 码 转 换 为 
UTF-8 或 UTF-8 BOMO 

# 第二部分 ASP.NET Core 日志

**一、概述**

记录日志一般有**两个目的：**

故障定位

显示程序运行状态

**ILogger<TCategoryName>对象扩展方法见下表**,按严重性从低到高排列。

|  |
| --- |
| LogTrace() 跟踪级别 |
| LogDebug() 调试级别 |
| LogInformation()信息级别 |
| LogWarning() 警告级别 |
| LogError() 错误级别 |
| LogCritical() 严重级别 |

**二、实例 步骤**

public class ShopController : Controller 
private config: 
private 
public ShopControIIer (IConfiguration configuration, logger 
config configuration; 
this. logger = 
this. 
public IActionResuIt Index() 
logger. gii'arning 
1 te nflg Sit nftg config. 
return Viev(siteConfig) 
et ect10nCSiteConfig"). 

不仅在后台，页面中也可以调用

ShopCon troller.cs 
Index.cshrmlt X 
MyCoreWeb. Models. Si teConEig; 
•using MyCoreXeb. Controllers; 
'using Microsoft. Extensions. Logging: 
'inject logger: 
— "Index 
logger. cshtmIüE") ; 
you sure you vant to delete 
aoosett•nas.lson 

**三、运行实例并观察效果**

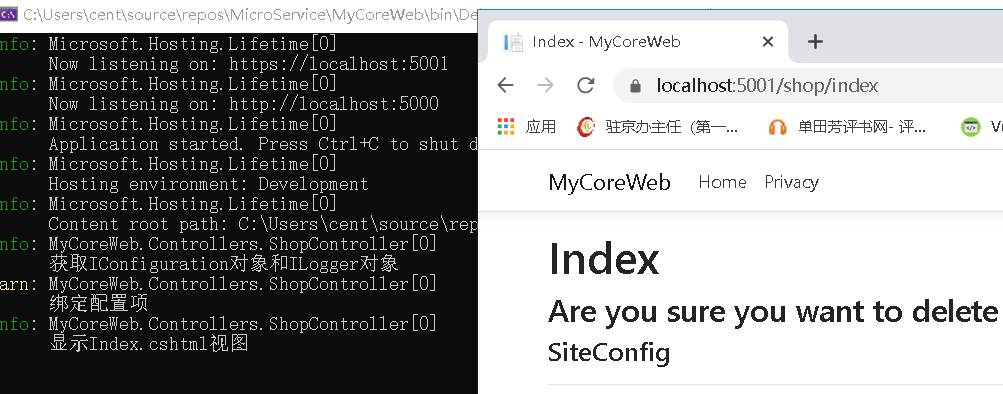
两种方式：

IIS Express 
IIS Express 
MyCoreWeb 
Azure Dev Spaces 
Web Chrome) 

**1、选择运行方式 IIS Express,运行页面，在vs的输出中可以看到日志**

NyCoreNeb. Controllers. ShopController: Information: 
N'ßoreveb. Control I ers. ShopControIIer : 
Warning: 
Information: h filndex. cshtnlnæ 
MyCoreYeb. Control I ers. ShopControIIer : 

**2、选择MycoreWeb可以在运行页面的同时，启动控制台，在控制台中观察日志**



# 第三部分 EF Core的使用

**一、管理Nuget程序包，安装以下包**

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.SqlServer：Sql Server数据库EF提供程序。

Microsoft.EntityFrameworkCore.Design：设计时使用到的EF共享库。

Microsoft.EntityFrameworkCore.Tools：EF的NuGet包管理器命令工具。

**二、Code First 模式**

**1、创建模型类**

public class Product 
public int productID{ get; set; 
public string ProductName { get; set;} // 
public decimal ProductPrice { get; set; V/ 
public Category get; set; 

**2、创建数据上下文类文件：重写ShopContext类的OnConfiguring()方法来设置数据库连接字符串，然后在类中添加Products和Categories集合属性键字段。**

using MicrosofVEntityFramewprkCpre; 
using EFCoreDemo.Models• 
namespace EFCoreDemo 
public class Sho Context:DbContext 
public DbSet<Product> Products { get; set; 
protected override void 
optionsBuilder.UseSqlServer( 
" Da ta n itial Cata " ) ; 

**3、生成数据库**

打开Visual Studio的“工具”-“NuGet包管理器” -“程序包管理控制台”

输入以下命令

**Add-Migration**：创建准备迁移的框架代码。

**Update-Database**：将任何挂起的迁移应用到数据库。

执行命令成功后，在SSMS（SQL Server Management Sudio）工具就可以查看到ShopDB数据库

**三、DataBase First 模式**

**1、在项目中生成ShopDB对应的实体模型和数据上下类**

语法

**Scaffold-DbContext '连接字符串' 数据提供程序名称 -OutputDir 结果输出文件夹 -Context 数据上下文名 [-DataAnnotations]**

Scaffold-DbContext 'Data Source=.\MSSQLSERVER2019;Initial Catalog=TRDB;User ID=sa;Password=123456' Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Context TRDBContext

Data Source=.;Initial Catalog=BookShopDB;Integrated Security=True

Scaffold-DbContext 'Data Source=.;Initial Catalog=BookShopDB;Integrated Security=True' Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Context BookShopDBContext

Scaffold-DbContext 'Data Source=.;lnitial Catalog=ShopDB; Integrated Security=True,•' 
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Context ShopContext 

**2、在ASP.NET Core Web应用程序中的appsettings.json中配置连接字符串**

"ConnectionStrings": { 
"ShopDB": "Data Source=.;lnitial Catalog=ShopDB;User Id=sa;Password=123456" 

**3、在Startup.cs中的ConfigureServices()方法中，添加数据上下文对象依赖注入服务，并从appsettings.json文件中读取连接字符串配置，代码如下：**

public class Startup 
public void ConfigureServices(lServiceCollection services) 
services .Add Db Context<Shop 
DB" )); 
services.AddControllersWithViews(); 

**4、在Controllers文件夹中创建ShopController控制器类，在该类的构造方法中获取依赖注入的数据上下文对象，在Index()方法中读取全部商品类别和商品信息，代码如下：其中黑色字体为依赖注入**

public class ShopController : Controller{ 
ShopContext shopContext; 
public ShopController(ShopContext context){ 
shopContext = context; 
public IActionResult Index() 
ViewBag.Categories = shopContext.Categories.ToList(); 
ViewBag.Products = shopContext.Products.ToList(); h 
return View(); 

第四章 gRPC（数据传输）

2020年8月11日

15:01

# 第一部分 概念

**一、定义**

gRPC（google remote procedure call）是一种跨平台，跨语言，使用http/2作为通信协议的数据传输技术。

**二、适用场景**

高性能轻量级微服务/多语言混合开发/点对点通信/网络受限

**三、不适用场景**

浏览器可访问的API/广播实时通信/进程间通信

**四、核心是Protobuf协议文件，类似JSON或者XML数据传输格式和规范**

|  |  |  |  |
| --- | --- | --- | --- |
| **比较项目** | **XML** | **JSON** | **Protobuf** |
| **数据结构支持** | **复杂结构** | **简单结构** | **较复杂结构** |
| **数据保存格式** | 文本 | 文本 | 二进制 |
| **数据保存大小** | 大 | 一般 | 小 |
| **解析效率** | 慢 | 一般 | 快 |
| **开发难度** | 繁琐 | 简单 | 简单 |
| **学习成本** | 低 | 低 | 低 |
| **适用范围** | 数据交换 | 数据交换 | 数据交换 |

**五、数据类型**

int32/int64/bool/float/double/string/bytes 字节数组

**六、字段修饰符**

required 必须字段 /optional 可选字段/repeated 重复字段

# 第二部分 创建gRPC服务

**一、新建vs gRPC项目**

Protos 
greet.proto 
ecu Greetersen,'ice.cs 

**二、适用NuGet包管理工具安装相关的工具和程序包**

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.SqlServer：Sql Server数据库EF提供程序。

Microsoft.EntityFrameworkCore.Design：设计时使用到的EF共享库。

Microsoft.EntityFrameworkCore.Tools：EF的NuGet包管理器命令工具。

**三、数据库优先模式反向生成上下文和实体类**

语句执行

**Scaffold-DbContext '连接字符串' 数据提供程序名称 -OutputDir 结果输出文件夹 -Context 数据上下文名 [-DataAnnotations]**

Scaffold-DbContext 'Data Source=.\MSSQLSERVER2019;Initial Catalog=TRDB;User ID=sa;Password=123456' Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Context TRDBContext

**四、写.proto文件**

**添加新建项，协议缓冲区文件**

user-proto 
2 
3 
4 
5 
6 
7 
8 
9 
10 
11 
12 
13 
14 
15 
16 
17 
18 
19 
20 
21 
22 
syntax = "prot03" . 
New gRpC. P rot os " ; 
option csharp_namespace — " 
import "google/protobuf/empty. proto" , 
package Shopping 
•service UserManager{ 
rpc GetÜserByID (UserRequest) returns (UserResponse) ; 
rpc GetUserList (google. protobuf. Empty) returns (UserListResponse) ; 
•message User { 
int32 id=l, 
string name=2; 
—message UserRequest { 
int32 id—l; 
- message UserResponse( 
User user—I; 
—message UserListResponse( 
repeated User 

# 五、添加服务引用，然后生成一下项目，vs会自动生成对应的代理类

gRPC 
greet 
user 
Protos\greet.proto 
Protos\user.proto 

# 

# 

# 六、NuGet包管理，安装System.Data.SqlClient程序级，然后编写Service文件

UserSen,'ice.cs 
—using 
ewg Rpc 
2 
4 
6 
8 
10 
11 
12 
13 
14 
15 
16 
17 
18 
19 
20 
21 
22 
23 
24 
25 
26 
27 
28 
29 
30 
31 
32 
33 
34 
35 
36 
37 
38 
39 
40 
41 
42 
43 
44 
45 
NewgRpc. Services-Ll serService 
Grpc. Core; 
NewgRpc. Protos ; 
System; 
System. Collections. Generic; 
System. Linq; 
System. Threading. Tasks 
NewgRDc. Models 
Google. Protobuf. WellKnownTypes ; 
GetUserByIOO_lserRequest request, ServerCaIICon 
using 
using 
uslng 
using 
using 
using 
us Ing 
—namespace NewgRpc. Services 
public class UserService: UserManager. UserManagerBase 
public override Task<UserResponse> GetUserByID(UserRequest request, ServerCa11Context context) 
var result new UserResponse() 
using (TRDBContext db-new TRDBContext ()) 
Users user = db. Users. FirstOrDefau1t(u u. Userld 
result. User = new User 
Id = user. Userld, 
Name = user. CserName 
//esult. User. Id = user. Cserld; 
request. Id) ; 
//esult. User. Name = user. UserName; 
return Task. FromResu1t (result) ; 
public override Task(ÜserLi stResponse> GetUserList (Empty request, 
var result = new UserListResponse ; 
using (TRDBContext db = new TRDBContext ()) 
— db. Users. ToList ( ) ; 
List<Csers) userlist 
foreach (Users u in userlist) 
ServerCallContext context) 
result. Users. Add(new User (Id=u. Userld, Name—u. CserName}) ; 
return Task. FromResult (result) ; 

# 七、配置Startup.cs文件

**引用命名空间**





# 

# 

# 第三部分 创建gRPC客户端

**一、创建Core mvc项目**

**二、复制服务端的Protos文件夹过来，然后添加服务引用，选择客户端，成功后，生成一下项目，自动生成代理类。**

gRPC 
URL 
O URL 

# 

# 三、编写控制器

oller.cs -v x 
c? Susing 
using 
using 
using 
using 
using 
using 
•i % gRPCClien t.Con trollers.usercontroller 
a client 
System; 
System. Collections. Generic ; 
System. Lina; 
System. Threading. Tasks ; 
Grpc. Net. Client ; 
Microsoft. AspNetCore. Mvc : 
NewgRpc. Protos ; 
8 namespace gRPCClient. Control lers 
public class UserContr011er : Controller 
UserManager. UserManagerC1ient p.lient — 
public User-Controller() 
null; 
var channel GrpcChannel. // I : 5001 
client — new UserManager. UseruanagerClient (channel) ; 
public IActionResult Index() 
var resp client. GetCserList (new Google. Protobuf. WellKnownTypes. Empty ()) ; 
var users — resp. Users. ToList() ; 
return View (users) 
var userRequest — new UserRequest ; 
userReQuest. Id =(int)id; 
— cl ient. GetCserByID (userRequest) ; 
var resp — 
var user = resp. User; 
return View (user) ; 

# 五、添加视图

# 列表

Index.cshtml 
2 
4 
6 
8 
9 
10 
11 
12 
13 
16 
17 
18 
19 
20 
21 
22 
23 
24 
25 
26 
x 
— "Index"; 
8<table table—striped table—hover") 
<thead> 
<tr> 
< / thead> 
<tbody> 
@foreach (var item in Model ) 
< tr> 
Id</td> 
tern. 
href="/user/detail/@item. 
< / t body) 
(/table> 

# 

# 详情

Detail.cshtrnl 
2 
3 
4 
5 
6 
8 
9 
— "Detail"; 
(hl >Detail 
uodel. ld 
CM0de1.Name 

# 

# 第四部分

第五章 **SignalR**

2020年8月11日

16:01

第六章 Redis基础与应用

2020年8月11日

16:01

# 第一部分 基础

**一、概述**

Redis是用ANSI C编写的开源的支持网络的，基于内存的，可持久性的**键值对存储数据库**，**是NoSql的一员。**

**不适合存储数据量太大和数据访问频率太低的场合。**

**适用于：缓存、计算器和限速器、排行榜、好友关系、简单消息队列、会话Session共享**

**二、启动服务**

**切换到解压或者安装文件夹**

C:\>cd Program Files\Redis

命令

redis-server redis.windows.conf

默认服务器端口号6379，本地远程访问127.0.0.1，可以通过修改该 redis.windows.conf文件来修改该端口号。

**三、redis-cli命令，启动客户端工具，quit退出**

**本地 redis-cli**

**远程 redis-cli -h 127.0.0.1 -p 6379 -a "mypass"**

set 键 值 写入

get 键 读出数据

# 第二部分 常用命令

**一、连接命令**

|  |  |
| --- | --- |
| 命令 | 说明 |
| AUTH password | 验证密码是否正确 |
| ECHO message | 打印字符串 |
| PING | 查看服务是否允许连接 |
| QUIT | 关闭当前连接 |
| SELECT index | 切换到指定的数据库 |

redis-cli 进入客户端

PING 查看是否允许连接，响应PONG

CONFIG get requirepass //查看密码 其中requirepass无效

CONFIG get requirepass //设置密码

AUTH "123456" //登录

ECHO "new password" //打印消息

QUIT 退出客户端

**二、键命令**

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 说明 | 命令 | 说明 |
| DEL key | 删除键 | PERSIST key | 移除键的过期时间 |
| DUMP key | 序列化指定的键 | PTTL key | 返回指定键的剩余的过期时间 |
| EXISTS key | 检查指定的键是否存在 | TTL key | 返回指定键的剩余生存时间 |
| EXPIRE key seconds | 以秒设置指定键过期时间 | RANDOMKEY | 从当前数据库中随机返回一个键 |
| PEXPIRE key milliseconds | 以毫秒设置键过期时间 | RENAME key newkey | 修改键的名称 |
| KEYS pattern | 查找所有符合给定模式的键 | TYPE key | 返回键所储存的值的类型 |
| MOVE key db | 将键移动到指定的数据库 | select index | 选中或者创建库 |

**KEYS \* 显示当前库所有键名称**

**EXPIRE msg 60 设置国旗世间60苗**

**MOVE msg 1 移动到库1**

**SELECT 1 选择库1**

**EXISTS msg 判断键是否存在**

**DUMP msg 序列化指定的键msg**

**TTL name 返回指定键的剩余实践**

**PERSIST msg 移除键的过期时间**

**TYPE msg 返回键存储的数据类型**

**RANDOMKEY 随机返回一个键**

**GET msg 读取**

**DEL msg 删除**

**rename key newkey 修改键名称**

**三、发布与订阅**

**打开二个客户端 ，**

**第一个客户端**

**语法：subscribe channel(监听的频道)**

**subscribe redischat**

**第二客户端**

**语法**

**publish 频道 “内容”**

**publish redisChat "abc"**

**publish redisChat "def"**

• redis•cli 
icrosoft Windows 10.0. 18363.476) 
(c) 2019 Microsoft Corporationo 
: /redis 
: s—cl i 
127. O. O. auth "123456" 
127. O. O. SUBSCRIBE redisChat 
(press Ctrl—C to quit) 
eading messages... 
x 
- redis•cli 
icrosoft Windows 10.0. 18363. 476) 
(c) 2019 Microsoft Corporationo 
: \Users\Administrator>cd /redis 
x 
1) subscribe 
"redi sChat" 
(integer) 1 
message 
"redi sChat " 
"Redis is a great 
"redi sChat 
"Learn redis soon 
127. O. O. auth "123456" 
127. O. O. PUBLISH redisChat 
aching technique 
(integer) I 
127. O. O. PUBLISH redisChat 
(integer) 1 
127. o. o. 
Redis 
"Redis is a great c 
"Learn redis soon 
Redis 
caching technique 

# 第三部分 数据类型

**一、数据类型**

字符串/哈希表/列表/集合/有序集合

**二、字符串 存储一个字符串**

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 说明 | 命令 | 说明 |
| SET key value | 设置指定键的值 | INCR key | 键值增一 |
| GET key | 获取指定键的值 | INCRBY key increment | 键值加上指定值 |
| GETRANGE key start end | 返回键中字符串值的子字符 | INCRBYFLOAT key increment | 键值加上指定浮点值 |
| GETSET key value | 将指定键设新值，并返回键的旧值 | DECR key | 键值减一 |
| MGET key1 [key2..] | 获取所有指定键值 | DECRBY key decrement | 键值减去指定值 |
| STRLEN key | 返回键所储存的字符串值的长度 | APPEND key value | 追加到该键值的末尾 |

MSET name "XPhone R" price 5999.99 clicked 1 设置多个字符串键值

INCR clicked 值+1

INCR clicked 值+1

MGET name price clicked 获取多个键值

STRLEN name 获取长度

GETRANGE name 0 6 截取从0起，不包含6

APPEND name " 10" 值追加字符串

MGET name price clicked 获取多个键值

**三、哈希表类型 存储多个键值对**

**字典结构数据， 表字段和字段值的映射，字段值只能是字符串。 Redis的任何数据类型都不支持嵌套**

|  |  |
| --- | --- |
| 命令 | 说明 |
| HSET key field value | 将哈希表键中的字段的值设为value |
| HMSET key field1 value1 [field2 value2 ] | 同时将多个（字段-字段值）对设置到哈希表键中 |
| HGET key field | 获取存储在哈希表中指定字段的值 |
| HMGET key field1 [field2] | 获取所有给定字段的值 |
| HGETALL key | 获取在哈希表中指定键的所有字段和值 |
| HEXISTS key field | 查看哈希表键中，指定的字段是否存在 |
| HKEYS key | 获取所有哈希表中的字段 |
| HLEN key | 获取哈希表中字段的数量 |
| HVALS key | 获取哈希表中所有值 |
| HINCRBY key field increment | 为哈希表键中的指定字段的整数值加上增量 |
| HDEL key field1 [field2] | 删除一个或多个哈希表字段 |

**四、列表类型（有序，可以用下标，存储多个字符串）**

|  |  |
| --- | --- |
| 命令 | 说明 |
| LPUSH key value1 [value2] | 将一个或多个值插入到列表头部 |
| RPUSH key value1 [value2] | 在列表尾部添加一个或多个值 |
| LPOP key | 移出并获取列表的第一个元素 |
| RPOP key | 移除列表的最后一个元素，返回值为移除的元素 |
| LLEN key | 获取列表长度 |
| LSET key index value | 通过索引设置列表元素的值 |
| LINDEX key index | 通过索引获取列表中的元素 |
| LREM key count value | 移除列表元素 |
|  |  |

**五、集合类型 没有顺序，可以进行并、交、差运算，不能用下标访问，不允许重复成员**

|  |  |
| --- | --- |
| 命令 | 说明 |
| SADD key member1 [member2] | 向集合添加一个或多个成员 |
| SREM key member1 [member2] | 移除集合中一个或多个成员 |
| SMEMBERS key | 返回集合中的所有成员 |
| SISMEMBER key member | 判断元素成员是否是集合键的成员 |
| SUNION key1 [key2] | 返回所有给定集合的并集 |
| SINTER key1 [key2] | 返回给定所有集合的交集 |
| SDIFF key1 [key2] | 返回给定所有集合的差集 |

**六、有序集合类型 ，有序，不允许重复成员**

|  |  |
| --- | --- |
| 命令 | 说明 |
| ZADD key score1 member1 [score2 member2] | 向有序集合添加一个或多个成员，或者更新已存在成员的分数 |
| ZCARD key | 获取有序集合的成员数 |
| ZCOUNT key min max | 计算在有序集合中指定区间分数的成员数 |
| ZRANGE key start stop [WITHSCORES] | 通过索引区间返回有序集合指定区间内的成员 |
| ZREVRANGE key start stop [WITHSCORES] | 返回有序集中指定区间内的成员，通过索引，分数从高到低 |
| ZREVRANGEBYSCORE key max min [WITHSCORES] | 返回有序集中指定分数区间内的成员，分数从高到低排序 |
| ZINCRBY key increment member | 有序集合中对指定成员的分数加上增量increment |
| ZREVRANK key member | 返回有序集合中指定成员的排名，有序集成员按分数值递减排序 |
| ZSCORE key member | 返回有序集中，成员的分数值 |

# 第四部分 内存回收

**一、查看内存消耗 info memory**

|  |  |
| --- | --- |
| 属性 | 说明 |
| used\_memory | Redis分配器分配的内存总量，即Redis存储的所有数据所占的内存 |
| used\_memory\_human | 以可读的形式返回使用内存量 |
| used\_memory\_rss | Redis进程占用的物理内存总量 |
| used\_memory\_peak | 使用内存的峰值 |
| used\_memory\_peak\_human | 可读格式返回used\_memory\_peak |
| used\_memory\_lua | Lua引擎消耗的内存大小 |
| mem\_fragmentation\_ratio | used\_memory\_rss/used\_memory比值，内存碎片率 |
| mem\_allocator | Redis所使用的内存分配器，默认jemalloc |

**二、设置Redis可使用给的最大内存，默认没有上限 config set maxmemory 6gb**

**三、内存溢出控制策略**

|  |  |
| --- | --- |
| 策略 | 说明 |
| noeviction | 默认策略，不会删除任何数据，拒绝所有写入操作并返回客户端错误信息 |
| volatile-lru | 根据LRU算法只对设置有超时属性的Key执行删除操作，如果没有可删除的Key，则回退到noeviction策略 |
| allkeys-lru | 针对所有的key，根据LRU算法执行删除操作直到回收到足够内存空间 |
| allkeys-random | 随机删除所有的键，直到腾出足够空间 |
| volatile-random | 针对带有过期属性的键，进行删除操作，直到腾出足够空间 |
| volatile-ttl | 根据键值对象的ttl属性，删除最近将要过期数据。如果没有，则回退到noviction策略 |

**设置策略语法**

CONFIG set maxmemory-policy {policy}

示例5.11 修改内存淘汰机制策略为volatile-lru

CONFIG set maxmemory-policy volatile-lru

第七章 Core项目使用Redis

2020年8月12日

15:59

# 第一部分 ASP.NET Core分布式缓存

**一、概念**

在ASP.NET Core中提供多种缓存数据的方法，而**在云计算和微服务架构中常用Redis作为缓存服务器来使用**

在.NET Core中提供方便访问Redis的软件包，通过NuGet软件包管理工具安装

**二、mvc项目，实现对Redis的 增 删 读**

**步骤**

**1、创建mvc项目**

**2、管理 NuGet包，安装**

Microsoft.Extensions.Caching.Redis 适用于.NET Core 2.2

Microsoft.Extensions.Caching.StackExchangeRedis 适用于.NET Core3.0以上版本

二者的使用方法相同。

3、使用第三方包，见第三部分

# 第二部分 Redis缓存Session

# 第三部分 第三方Redis客户端软件包

StackExchange.Redis包是开源版本，没有任何限制，因此本书主要介绍StackExchange.Redis包的使用。

• ReadDate0 
•using 
us Ing 
us Ing 
using 
using 
using 
using 
using 
• gRPCCIient.ControllersRedisDataControIIer 
System; 
System. Col lections. Generic ; 
System. Lina; 
System. Threading. Tasks; 
Microsoft. AspNetCore. Mvc; 
StackExchange. Redis 
Newtonsoft. J son; 
NewgRpc. Protos ; 
•namespace gRPCC1ient. Controllers 
public class RedisDataController 
ConnectionMultip1exer 
IDatabase db; 
public RedisDataController() 
. Control ler 
redis = ConnectionMultip1exer. Connect ("127. 0. 0. 1:6379") ; 
db = redis. GetDatabase (0) ; 
public IActionResult WriteDate () 
User u = new User() ( Id 
= 1, Name = 
User u2 new Cser() [ Id — 2, Name ¯ 
JsonConvert. SerializeObject (u) ; 
string jsonl — 
string json2 = JsonConvert. SerializeObject (u2) ; 
db. HashSet ("myprods", "myprodl", jsonl) 
db. HashSet ("myprods", "myprod2", json2) 
ViewBag. msg — 
return View() 
public IActionResult ReadDate() 
// if (db. KeyExists ("myprodl")) 
string result — db. ; 
User u = JsonConvert. (result) : 
// ViewBag. msg — "key myprodl MJffL"J: "+db. StringGet ("myprodl"). ToString () ; 
ViewBag. msg = u; 
ul i st—new ; 
foreach (string item in db. HashKeys ("myprods")) 
string result — 
db. HashGet ("myprods", 
item) ; 
ulist. Add (JsonConvert. Deserial (result)) ; 
return View(ulist) : 
public IActionResult RemoveDate () 
if (db. 
ViewBag. msg = 
else 
ViewBag.msg - 
return View() ; 

# 第四部分 Redis事务

第八章 Docker容器基础与应用

2020年8月11日

16:02

**通过命令关闭Hyper-V（控制面板关闭Hyper-V起不到决定性作用，要彻底关闭Hyper-V）**

**以管理员身份运行Windows Powershell (管理员)（Windows键+X）**

运行下面命令

bcdedit /set hypervisorlaunchtype off

**步骤三：重启电脑，即可正确运行VMware Workstation**

**使用docker则相反** bcdedit /set hypervisorlaunchtype auto

容器和Docker

2020年1月4日

16:10

# 

# 容器和Docker

容器化是一种软件开发方法

通过使用容器化的开发方法，应用程序或服务以及它们的依赖和配置（抽 象为Mainfest 文件并部署）可打包在一起成为一个容器镜像。容器化的应用程序可以作为独立的单元 进行测试，也可以作为容器镜像实例，部署到主机操作系统（OS）上。

容器还可让应用程序在一个共享的操作系统上实现彼此隔离。容器化的应用程序运行在容器主机 （Container Host）中，容器主机运行在OS（Windows 或Linux）上。因此容器比虚拟机镜像更轻便。

每个容器可以运行完整的Web 应用或服务.

### 主机操作系统（OS）（Windows 或Linux）->一个或多个容器主机（Container Host）->容器化的应用程序

# 什么是Docker？

Docker 是一个开源项目，这个项目旨在通过把应用程序打包为可移植、自给自足的容器（可运行在云 端或本地），实现应用程序的自动化部署。Docker 也是一家专门促进和发展这门技术的公司，它和多 家云提供商、Windows、Linux 提供商都有合作，微软也包括在内。

在Windows 和 Linux 上，Docker 镜像容器可以本地化运行。但是Windows 镜像只能运行在Windows 主机上，Linux 镜像只能运行在Linux 主机上，这意味着镜像需要托管到服务器或虚拟机上。

容器镜像是一种打包方法，可以把应用程序或服务打包，然后用可靠、可重复的方式部署。可以说， Docker 不仅是一门技术，还是一种方法论，一种工作流程。

# Docker术语

容器镜像（Container image）：一个包含了创建容器所需所有依赖和信息的“包”。创建好的镜像是不可变的。

容器（Container）：Docker 镜像的实例。

标签（Tag）：用于镜像的标记或标签，让不同镜像或同一镜像的不同版本（依赖于版本号或目标环 境）可以区分开来。

Dockerfile：一个文本文件，其中包含创建镜像的指令。

构建（Build）：构建容器镜像的操作。可基于Dockerfile 提供的信息和上下文，以及构建这个镜像的 文件夹中其他文件来构建一个容器镜像。我们可以使用 Docker 命令（docker build）来构建镜像

仓库（Repository，Repo）：相关 Docker 镜像的集合，用标签来表示镜像版本。有些仓库会包含某 个特定镜像的多个变体，例如，一个镜像包含若干 SDK（更加重量级），一个镜像只包含运行时（更加 轻量级）等。这些变体可以用标签来标记。一个仓库可包含多个平台变体，例如一个 Linux 镜像和一个 Windows 镜像

注册表（Registry）：为仓库提供访问服务。对于大多数公共镜像来说，默认的注册表是Docker Hub （它隶属于Docker 公司）。一个注册表通常包含来自于多个团队的仓库。企业通常会创建私有注册表 来存储和管理他们创建的镜像。Azure 容器注册表也是一种注册表。

Docker Hub：公用的注册表，可用来上传镜像并让它们协同工作。Docker Hub 提供了Docker 镜像托 管服务、公有或私有注册表、构建触发器和Web hook，可以和GitHub 与Bitbucket 无缝集成。

Azure 容器注册表（Azure Container Registry）：一种和Docker 镜像配合使用的公共资源，它的组 件位于Azure 中。它提供了一个与用户的Azure 部署紧密相关的注册表，我们可以使用Azure Active Directory 组和权限对其进行访问控制。

Docker 可信注册表（Docker Trusted Registry）：一种Docker 注册表服务（来自Docker 公司）， 可以在本地安装，所以它一般在企业内部数据中心和网络中使用。对于需要在企业内部进行管理的私有 镜像来说，这种方式更加方便。Docker 可信注册表包含在Docker Datacenter 产品中。详细信息可以 参考Docker 可信注册表（DTR）。

Docker 社区版(CE)：针对 Windows 和 macOS的开发工具集，用于在本地构建，运行和测试容器。 Docker CE for Windows 提供了Linux 容器和Windows 容器开发环境。Windows 上的Linux Docker 主 机是基于Hyper-V 虚拟机的。而Windows 容器主机是直接基于Windows 的。Docker CE for Mac 基于 Apple 的Hypervisor 框架和xhyve hypervisor，它在 macOS 上提供了一个Linux Docker 主机虚拟机。 Docker CE for Windows 和 Docker CE for Mac 替代了基于Oracle VirtualBox 的 Dock

集群（Cluster）：一组Docker 主机可通过一个虚拟的 Docker 主机暴露，让应用程序可以扩展为多个 服务实例，分布到集群内的多个主机上。Docker 集群可以使用 Docker Swarm、Mesosphere DC/OS、 Kubernetes 和Azure Service Fabric 来创建。（如果使用Docker Swarm 管理集群，通常使用Swarm 这 个术语来代表集群。）

编排引擎（Orchestrator）：用来简化集群和Docker 主机管理的工具。编排引擎可以让我们通过一个 命令行工具（CLI）或者一个图形化的界面来管理镜像，容器和主机。我们可以管理容器网络、配置、 负载均衡、服务发现、高可用性、Docker 主机配置等。编排引擎负责在一组节点上运行、分发、扩展 和修复工作负载。通常来说，编排引擎产品和提供集群基础设施的产品（如 Mesosphere DC/OS、 Kubernetes、Docker Swarm 和Azure Service Fabric）是相同的。

为Docker 容器选择.NET Core 或.NET Framework

2020年1月4日

17:13

# 如下这些情况，应使用.NET Core 和Linux 或Windows 容器来开发容器化的Docker 服务端应用程序：

• 需要跨平台。例如，需要同时使用 Linux 容器和Windows 容器。

• 应用程序架构是基于微服务的。

• 需要容器快速启动，希望容器尽量保持轻量级，以获得更高“密度”，也就是说，为了节约成 本，希望在每个硬件单元上承载更多容器。

简言之，在创建一个新的容器化.NET 应用程序时，应该把.NET Core 作为默认选择。.NET Core 有很多 优势，并且它和容器的方法论和工作方式完美匹配。

# 如下这些情况，应该使用.NET Framework 来开发容器化的Docker 服务端应用程序：

• 应用程序目前使用了.NET Framework，并且高度依赖于Windows。

• 需要使用.NET Core 不支持的Windows API。

• 需要使用.NET Core 不支持的第三方的.NET 库或NuGet 包。

下表概括总结了应该使用.NET Framework 或.NET Core 的决策因素。请务必注意，Linux 容器必须使用 基于Linux 的 Docker 主机（虚拟机或服务器），而 Windows 容器必须使用基于 Windows Server 的 Docker 主机（虚拟机或服务器）。

# 官方.NET Docker 镜像

官方的.NET Docker 镜像是由微软创建和优化的Docker 镜像。它们都公开发布在 Docker Hub 上的 Microsoft 仓库中。每个仓库可包含多个镜像，分别提供了不同的.NET 版本，操作系统类型和版本（ Linux Debian、Linux Alpine、Windows Nano Server、Windows Server Core 等）。

对于.NET 仓库来说，微软的愿景是尽量构建小而专的仓库。一个仓库代表一种具体场景，或者一个需 要完成的具体工作。例如，microsoft/aspnetcore 镜像应该在Docker 上使用ASP.NET Core 的场景下使 用，因为这些镜像做了额外优化，可以让容器更快地启动。

基于容器和微服务的应用架构

2020年1月4日

17:22