沟通需求和标准，谈妥价格

签合同，收取50%费用

需求分析-框架搭建-模块切分

数据库初步设计

项目验收交付后，支付剩余费用。

只有主动才能获取到机会

What why when how

**基础**

**1．变量命名**

变量名称一定要有意义，看到变量的名字，就知道变量在程序中的作用。

C#命名规范：

Camel：多用于变量或者字段命名，第一个单词的首字母小写，其余每个单词的首字母大写。

给字段命名，前面必须加下划线。\_highSchoolStudent

Pascal：多用于函数命名，每个单词的首字母都要大写 HighSchoolStudent()

返回值，接受变量的名称也要有意义。int max= GetMax();

**2．进程**

一个应用程序就是一个进程，而一个进程又是由多个线程组成的。

进程帮助我们在内存中分配应用程序执行所需要的空间，可以通过进程来直接操作应用程序。

打开进程：Process.Start(“notepad”)

**3．面向对象**

面向对象编程好处：让程序具有扩展性。

类：决定了对象将要拥有的属性和行为，万物皆对象。

--字段：存储数据

--属性：保护字段 get set 取属性值时，执行 get 属性赋值时，执行set

--函数：描述对象的行为

--构造函数：初始化对象，给对象的每个字段赋值。

自动属性，可以在构造函数中对字段进行保护。

面向对象三大特征。

封装：将一堆很难理解的功能封装到一个函数里，可以对函数重复的调用。

作用：减少了大量的冗余代码 ；

将复杂功能封装起来，对外提供一个简单接口，不需知道怎么实现，会使用就ok了。

继承：减少了类中的冗余代码，让类与类产生了关系，为多态打下了基础

特性：单根性：一个子类只能有一个父类

传递性：爷爷类 爹类 儿子类

多态：就是把所有的子类对象都当做父类来看。

实现多态一般使用三种方法

①虚方法 virtual override 父类加virtual 子类加 override

父类能创建对象，成员有意义时，使用虚方法

②抽象类 abstract override

父类不能创建对象，成员没有意义时，使用抽象类

③接口 interface

定义：多态就是指不同对象收到相同消息时，会产生不同行为，同一个类在不同的场合下表现出不同的特征。

作用：把不同的子类对象都当做父类来看，可以屏蔽不同子类对象之间的差异，写出通用的代码，以适应需求的不断变化。

④继承实现 里氏转换：

1、子类可以赋值给父类

2、如果父类中装的是子类对象，则可以将这个父类转换为对应的子类对象

案例：animal 父类，cat，dog，pig 等子类

父类没有办法确定子类如何去实现，使用抽象方法，而抽象成员只能存在于抽象类中，父类是抽象类

子类继承一个抽象父类，该子类必须重写这个抽象类中的所有抽象成员。 override 父类的方法

实现多态：都是声明父类对象，指向子类对象实例

Animal[] a = { new Cat(), new Dog(), new Pig() }；

里斯转换关键字

1、is：返回bool类型，指示是否可以做这个转换

2、as：如果转换成功，则返回对象实例，否则返回null

为什么要有这个关键字：

Person person = new Student();

Teacher t = (Teacher)person; //强转，有时报无法转换异常

Student s = person as Student;

里式转换作用：我们可以将所有的子类都当做是父类来看，针对父类进行编程，写出通用的代码，适应需求的不断改变。

**4、关键字**

new干什么用

答：1、创建对象：干三件事，①在堆中开辟空间 ②在开辟的堆空间中创建对象 ③调用对象的构造函数

不想让调用构造函数，将构造函数私有化。private

2、隐藏父类的成员

子类成员起名与父类一样时，会把父类成员隐藏掉(覆盖掉)，在访问修饰符后加new 彻底隐藏父类成员。

this 干什么用：

答：1. 代表当前类的对象

局部变量优先级高于全局变量。

2. 显示的调用自己的构造函数

base 干什么用

答：1. 显示调用父类的构造函数，只是指向父类的引用而已。

2. 调用父类的成员

**5、抽象类**

实现多态

方法的签名：指的是方法的参数与返回值

用abstract 修饰的类

抽象类特征：

①.抽象成员必须标记为abstract,并且不能有任何实现（连大括号都不能有）。

②.抽象成员必须在抽象类中。

③.抽象类不能被实例化(没有意义)

④.子类继承抽象类后，必须把父类中的所有抽象成员都重写。（除非子类也是一个抽象类，则可以不重写）

⑤.抽象成员的访问修饰符不能是private(子类需要继承，拿不到了)

⑥.在抽象类中可以包含实例成员。并且抽象类的实例成员可以不被子类实现

⑦.抽象类是有构造函数的。虽然不能被实例化。（每一个类都有构造函数，抽象类构造函数是给子类调用的）

⑧.如果父类的抽象方法中有参数，那么。继承这个抽象父类的子类在重写父类的方法的时候必须传入对应的参数。

如果抽象父类的抽象方法中有返回值，那么子类在重写这个抽象方法的时候 也必须要有返回值。

如果父类中的方法有默认的实现，并且父类需要被实例化，这时可以考虑将父类定义成一个普通类，用虚方法来实现多态。

⑨父类中有构造函数，子类构造函数必须显示调用

如果父类中的方法没有默认实现，父类也不需要被实例化，则可以将该类定义为抽象类。

例：移动设备 U盘，移动硬盒，MP3 设备不同，读写方式不不能定义死

**6、接口**

理解接口：接口是一种能力；

接口也是一种规范

如果你继承了这个接口，就必须按照接口的要求来实现这个接口。

接口特点

接口是一种规范。也是一种能力。

只要一个类继承了一个接口，这个类就必须实现这个接口中所有的成员

接口的功能要单一

接口是为了多态。

接口不能被实例化。也就是说，接口不能new(不能创建对象)，跟抽象类是一样的，因为创建对象也没意义。

接口中的成员不能加“访问修饰符”，接口中的成员访问修饰符为public,不能修改。

类中的成员默认的访问修饰符是private ，而接口中默认的访问修饰符是public

接口中的成员不能有任何实现（“光说不做”，只是定义了一组未实现的成员）。

接口中只能有方法、属性、索引器、事件，不能有“字段”和构造函数。

接口与接口之间可以继承

接口并不能去继承一个类，而类可以继承接口 （接口只能继承于接口，而类既可以继承接口，也可以继承类）

实现接口的子类必须实现该接口的全部成员。

一个类可以同时继承一个类并实现多个接口，如果一个子类同时继承了父类A，并实现了接口IA,那么语法上A必须写在IA的前面。

class MyClass:A,IA{}，因为类是单继承的。

显示实现接口的目的：解决方法的重名问题

什么时候显示的去实现接口：当继承的接口中的方法和参数一摸一样的时候，要是用显示的实现接口

当一个抽象类实现接口的时候，需要子类去实现接口。

**7、索引器**

让对象以索引的方式操作数组

Int[] I = {0, 1, 3, 55, 99};

Dictionary<string,string> dic = new Dictionary<string,string>(); //键值对集合

**8、访问修饰符**

public private internal protected protected internal

public:在哪都可以访问

private:私有的，只能在当前类的内部进行访问

internal:只能在当前程序集(项目)中访问。

protected：受保护的，可以在当前类以及该类的子类中访问

注意点：

1、能够修饰类的访问修饰符只有两个：public internal(默认就是internal)。

2、在同一个项目中，public的权限跟internal是一样的。

3、子类的访问权限不能高于父类。

**9、添加引用和导入命名空间**

命名空间就是类的"文件夹"。类就是"文件夹"中的文件。需要导入命名空间

添加引用：如果我需要在一个项目中，访问另一个项目中的某一个类

步骤一：添加对另一个项目的引用

步骤二：导入命名空间

**10、类库 （类的仓库）**

.exe 程序集和.dll 程序集（没有Main函数）

.dll（类库）文件，我们使用类库来帮助我们封装一些常用的功能（）

类库好处：我写完了发给你，你同样能够使用，前提是添加引用

**11、静态和非静态**

静态成员：被static修饰

实例成员：不被static修饰（实例即对象）

静态成员先于实例成员被加载到内存中，只有创建了对象（实例化），才有实例成员。

静态类中只能存在静态成员 （Console是典型的静态成员）

调用：

静态成员调用：类名.静态成员名;

实例成员调用：实例名.实例成员;(实例就是我们的对象)

非静态类中也可以有静态成员，静态成员中不能有非静态成员。

为什么要在普通类中有静态成员？

如果一些成员方法，会被别人翻来覆去调用，可以考虑定义静态类，省得还得创建对象。

什么时候使用静态？

①作为工具类

②在整个项目中资源共享，正因为它是资源共享的，所以静态成员必须要等到整个项目都结束的时候才会被资源释放。

在我们的项目中应该尽可能的少使用静态成员。

**12、设计模式**

设计模式就是用来专门解决一些特定的问题。

建筑行业最早应用到设计模式这个概念

1、注册一个公司

2、招兵买马

3、投标买地

4、安排施工队开始施工

5、卖楼

①单例设计模式

在整个程序中，我们要保证对象必须是唯一的。（两次打开qq，第二次打开的是我第一次打开的）

实现：

第一步：构造函数私有化

第二步：声明一个静态字段，作为全局唯一的单例对象

第三步：声明一个静态函数，返回全局唯一的对象

②简单工厂设计模式

核心：把所有的子类都当做父类来看待

练习：

提示用户分别的输入两个数字：

再输入运算符：返回一个计算的父类，并调用方法得到结果。

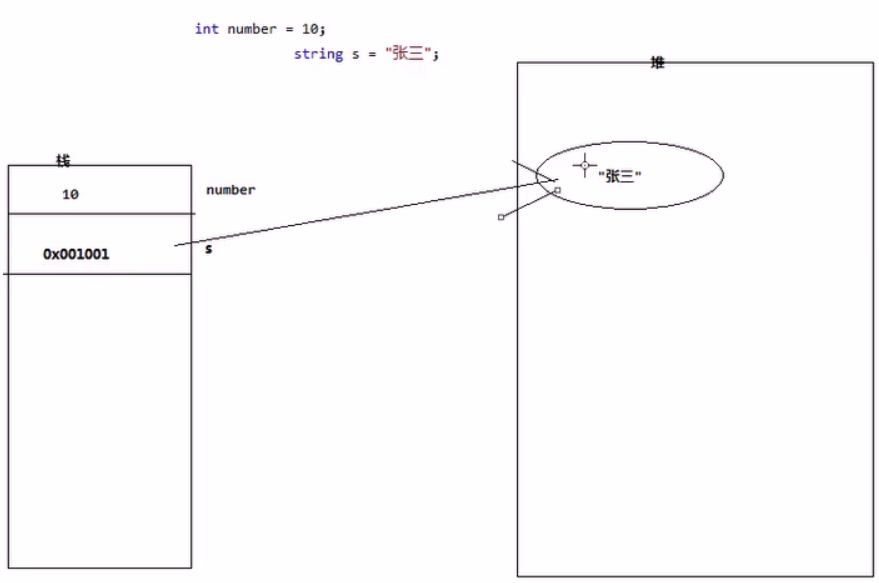
Add Sub Cheng Chu

**13、值类型和引用类型**

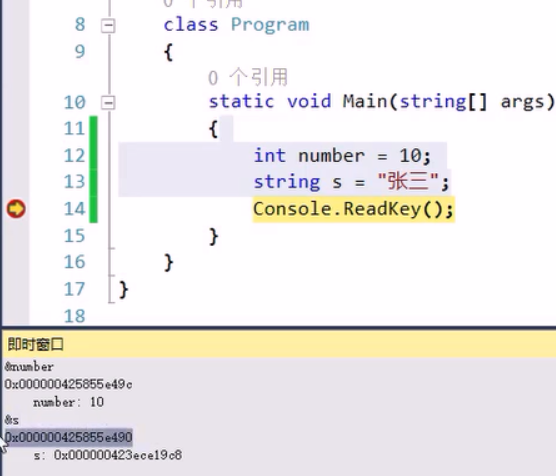
值类型：int double char bool decimal struct enum

引用类型：string 数组 自定义类 接口 委托

①值类型的值存储在内存的栈上，引用类型的值存储在堆中。栈上存储数据的效率要高于堆。



注意：代码查看在内存中地址 在即时窗口中输入 &+变量名 回车



②值传递：把值类型作为参数传递，把值本身进行传递。

注意：ref可以把值传递改变为引用传递。（把栈地址传给形参了）

③引用传递：把引用类型的值作为参数传递，传递的是引用（堆中的地址）。

④实参与形参：内存中两块不同的空间

⑤值类型均继承自System.ValueType valuetype继承自object

**14、字符串的学习**

①重要特性：

不可变性：我们不管对字符串做什么操作，都会在内存中产生一块新的实例。如果对字符串进行大量操作，会在堆中产生大量的空间（垃圾），只能通过GC干掉

驻留池：声明一个字符串后，又声明一个新字符串，首先去内存中找有没有这个字符串。如果有，直接将指向给它，节省空间。如果没有，开辟一个新空间。

可以将一个字符串看作是char类型的只读数组。可以通过下标访问 “aaaa”[2]

注意：对字符串进行操作必须接收它的返回值，因为不可变性，对它操作都会产生新的空间。

②== 与 Equals 比较？

对于string 类型而言，不管是==还是Equals 比较的都是值本身

Equals 默认比较的是地址。我们自己定义的类中如果用到Equals ，都会将Equals 进行重写，使之按照我们自己的需求进行比较。

③对象.ToString()

能够拿到这个对象所在类的命名空间

**15. GC**

Garbage Collection垃圾回收 ，

工作原理：每隔一段时间，会扫描整个内存，如果发现有空间没有被指向。则立即把它销毁。

内存工作繁忙时，不一定会马上销毁，导致内存卡 ，尽量不要大量操作字符串。

手动 GC.Collect(); //但是微软强烈不建议我们使用该函数

**16、StringBuilder**

它就是拼接字符串的一个工具，拼成完成后，还是需要将它转回字符串。

Stopwatch sp = new Stopwatch();

sp.Start();

sp.Stop();

sp.Elapsed; //显示间隔时间

**17、ref 、out 、params 参数**

ref参数侧重于将一个值带到函数中进行改变，再将改变后的值带出去。ref参数在函数内不用赋值，在函数外必须为ref参数赋值(值是从外部传来的)。

out参数侧重于在函数中返回多个值，且必须在方法内为其赋值。外部接收的(定义)的值没必要赋初值(值从函数内传来的)。

params可变参数，必须是形参列表中最后一个元素。

传参方式：

传类型一致的数组 Test(“zhangsan”, new int[]{1, 2, 3, 4})

传给数组类型一致的参数 Test(“zhangsan”, 1, 2, 3, 4)

**18、集合**

非泛型集合：

ArrayList

Hashtable

ArrayList不建议使用原因：list.add(100); //给的是指类型，而add中保存的是object类型，导致有装箱和拆箱的过程，增加了耗时和占用更多内存。

泛型集合

List<T>

Dictionary<Tkey,Tvalue> （字典）

List<T>常用的函数

Add()：添加单个元素

AddRange():添加一个集合

Insert():插入一个元素

InsertRange():插入一个集合

Remove():移除指定的元素

RemoveAt():根据下标移除元素

RemoveRange():移除一定范围内的元素

ToArray():集合转换成数组

ToList():数组转换成集合

什么是初始化器？

var list = new List<string>(){ ‘a’, ‘b’, ‘c’};

**19、装箱和拆箱**

装箱：值类型---->引用类型

拆箱：引用类型--->值类型

装箱的时候拿什么类型装的箱，拆的时候，就得拿什么类型去拆。

我们判断是否发生了拆箱或者装箱，首先要判断这两种数据类型是否存在继承关系。如

int n = 0;

ICompareable i = n; //装箱操作（接口与值类型之间的拆箱与装箱）

方法重载时，如果具有该类型的重载，那么就不叫拆箱与装箱

特殊案例：

int n = 0;

object o = n;

n=100;

Console.log(n+ ‘,’ + (int)o); 方法里面默认使用了String.Concat()方法

经历了，3次装箱，1次拆箱 ……

**20、编码格式**

将字符串以怎样的形式保存为二进制。

ascii 256 西方

6000 GB2312 国内

GBK GB18030

ISO（国际标准）

Unicode：编码集合，数量比较大，占空间。

utf-16

utf-8

浏览器中看是什么编码格式？

更多工具—>编码

出现乱码的原因：保存文件时采用的编码格式与打开该文件时采取的编码格式不一致。

文本文件：拖到txt中还能看得懂得就是文本文件。.txt .html .csv

.txt 文件，选择“文件” 下 “另存为”，可看txt文件编码格式

**21、文件操作**

相对路径：

绝对路径：

File类

使用前需要导入该类所在的命名空间

该类为静态类

只能操作小文件：因为他的读取形式是一次性读取。

读写数据要转码的： Encoding.Defualt.GetBytes(str) 系统默认编码格式转成bytes

Encoding.UTF8.GetString(buffer) utf-8转成…

方法：

操作文件

Exist(path) 判断指定的文件是否存在，路径是相对路径(.exe 所在路径)

Create(path) 创建 路径是相对路径

Move(oldUrl, newUrl) 剪切

Copy(oldUrl, newUrl) 复制 newUrl 不能是现有的文件

Delete() 删除 路径是相对路径

读取文件数据 ：File 一般只操作小文件。

ReadAllLines(绝对路径，Encoding.Default); //当前操作系统默认编码格式

默认采取的的编码格式是utf-8

返回数组

操作每一行数据时，用它

ReadAllText(绝对路径)

返回字符串

默认采取的编码格式是utf-8

读取过来看一看时，用它

ReadAllBytes(绝对路径) buffer 接收返回值

返回字节

看不懂，转成字符串 string str = Encoding.UTF8.GetString(buffer) 指定编码格式，转

写入文件数据

WriteAllBytes(路径， buffer)

WriteLine(“aaa”)

WriteAllLines(路径，“c”)

②Directory 类

操作文件夹

CreateDirectory：创建一个新的文件夹

Delete ：删除 存在保护机制，内部有文件，不允许删除。全删可以加个true

Move ： 剪切 剪切文件夹，只能C盘在C盘，D盘在D盘

Exist ：判断指定文件夹是否存在

GetFiles(“”) 获取指定目录下所有文件的全路径，要是匹配，在后面加参数“\*.mp3”

GetDirectory() 获取指定目录下第一目录所有文件夹

如果要获得目录下所有文件夹，只能使用递归调用每个字文件夹

③FileStream 操作字节

缓冲区：在内存中开了一个空间，在读时数据时，不往硬盘中写入，当缓冲区读满时，统一向硬盘中写入。

对硬盘形成一个缓冲作用，硬盘的读写次数有上限，尽量减少对硬盘的操作。

静态类，智能提示没有了，关了重新开就好了

④StreamReader 操作字符（文本文件）

⑤StreamWriter 操作字符（文本文件）

⑥ 文件流与文件(File)有什么区别？

文件：读取数据是一次性读取 （读取小文件）

文件流：读取文件给一个大小，切分成好多块，每次读取一小块。（读取大文件），文件流的资源GC并不会帮助释放。

**22. MD5加密**

明文保存

加密保存

不可逆的加密算法，简单数字可能被解密，复杂的密码很难破解。

ToString(“x2”) 与 GetString() 方法比较

x 为 16进制

x2 为 每次都是两位数

**23. 递归函数**

调用函数本身

**24.正则表达式**

就是一个模板，我们可以使用这个模板，在字符串中找到我们想要的数据。

常用三种情况：

判断是否匹配：Regex.IsMatch(“字符串”, “正则表达式”)

字符串提取：

Regex.Match(“字符串”, 要提取字符串的正则表达式) // 只能提取一次

Regex.Matches() //循环提取所有，匹配的字符串

字符串替换： Regex.Replace(“字符串”，”正则”，”替换内容”)

注意：正则表达式是操作字符串的。

组成：限定符，元字符，常用的表达式

**基础\_Ado.Net**

**1.Ado.Net**

是一种数据访问技术，使应用程序可以连接到数据库，并以各种方式来操作其中的数据。

本质上是一个com组件库，在.Net中访问数据，优先选择的数据访问接口。

在System.Data.dll中，System.Data命名空间下，提供不同Ado.Net类

ADO ActiveX Data Object 早期开发人员用来访问数据的组件。Ado.Net 逐步取代Ado

Ado.Net 和 ADO 是两种不同的方法。技术上 Ado oledb接口 基于COM技术；Ado.Net 拥有自己的接口，基于.Net体系架构。

**Ado.Net组成**：

①DataSet 非连接的核心组件，独立于任何数据源的数据访问，适用于多种不同的数据源。（数据在内存中的一个临时缓存）

②Data Provider(数据提供程序) 用于连接数据库、执行命令、检索结果

如：用于操作SQLServer的数据提供程序，System.Data.SqlClient.dll等(oledb、odbc、oracle)

**数据提供程序包括对象** 针对介绍SqlServer数据库

Connection 提供与数据源的链接 SqlConnection

Command 执行数据库命令的对象 SqlCommand

DataReader 从数据源中提供快速的，只读的数据流 SqlDataReader

DataAdapter 提供DataSet对象与数据源的桥梁

④**Ado.net 访问数据库步骤 共五步**

链接到数据库，打开链接，创建执行命令对象，执行命令，关闭链接

**2.** Ado.Net**连接字符串**

一组被格式化的键值对。告诉数据源在哪里，数据库名称，提供什么样的访问信任级别，其它相关信息

链接成功后，可以看到链接的版本等信息。

①Sqlserver 身份认证 又称为安全链接

Data Source 数据源 Initial Catalog 数据库名 User Id 账号 Password 密码

本地：Data Source=.;Initial Catalog=TestBase;User Id=sa;Password=123456;

远程：Data Source=198.2.7.9,3306; Initial Catalog=TestBase;User Id=sa;Password=123456;

简写方式：Data Source 简写 server Initial Catalog 简写 database User Id 简写 uid Password 简写 pwd

②Window 身份认证 又称为可信链接

Data Source 数据源 Initial Catalog 数据库名 Integrated Security=True/SSPI (或者Trusted\_Connection=True)

本地：Data Source=.;Initial Catalog=TestBase; Integrated Security=SSPI;

③构建方式

方式一：手写

方式二：SqlConnectionStringBuilder 类来构建，通过一些属性来购建

通常放在配置文件中connectionString，providerName 为了区分是那种数据库。也可以放到appsettings 中

<add name=”connStr” connectionString=”server=.;database=TestBase;uid=sa;pwd=123456;” providerName=”System.Data.SqlClient”>

通过ConfigurationManager 来读取

3.SqlConnection 类 连接对象

继承自DbConnection 抽象基类，不能被实例化。提供与SqlServer的链接

属性：ConnectionString 链接字符串属性、

Database 要连接的数据库名称、

DataSource 服务器即数据源名称、

State 链接状态 closed 已关闭 Open 打开的 Executing 正在执行命令 Fetching 正在检索 Broken中断

ConnectionTimeout 等待时间，默认15秒

方法：conn.Open() // 打开链接

conn.Close() // 关闭链接，关闭后，连接字符串还存在，还可以再打开

conn.Dispose() // 释放链接，释放后，连接字符串已清空，需重新设置

conn.CreateCommand() // 创建命令对象

SqlConnection conn = new SqlConnection(connStr);

conn.Open();

using(){} //只要对象继承于IDisposable接口，就可以使用using(){} 自动释放对象。 相当于try finally 没有catch抓取异常

只要离开using 代码块，会自动调用disposed 释放链接

4.连接池

为什么？

平常连接过程非常耗时耗力，经历几个阶段，建立物理连接，与服务器初次握手，分析链接字符串，身份验证等等…

连接池是为了重复利用已有链接

是什么？

是用于存放一定数量的、与数据库服务器的物理链接的容器。需要的时候，从容器里取出一条空闲的链接，而不是创建新的连接

作用：减少了链接数据库的开销，从而提高应用程序的性能

连接池分类：同一时刻同一应用程序域可以有多个不同类型的连接池。

用什么来标识区分的：如进程、应用程序域、连接字符串 window标识共同组成的签名来标识区分

对同一程序域来说，由连接字符串来区分。打开一条连接如果这条连接的类型签名与现有的连接不匹配，就会创建一个新的连接池。反之则不会创建，它们会共用同一个连接池

如何分配：根据链接请求的类型找到与它相匹配的连接池，尽力的分配一条空闲的连接。有空闲的连接，返回这条连接。

如果已用完，就会创建一个新连接添加到连接池中，如果已达到最大连接数，等待，直到有空闲的连接可用。

移除无效连接：不能正确的连接到数据库服务器的连接，连接池存储的与数据库服务器的连接数量是有限的。

无效连接应当移除，浪费连接池空间。连接池管理器会自动处理无效连接的移除问题。

回收连接：释放使用完的连接，应当及时关闭或释放。Conn对象的close或Dispose 都能让连接回到连接池。

Ado.net 默认是启用连接池的，第一次连接请求后就会存在连接池

连接字符串是可以控制连接池的行为的。控制有三个属性

三个属性：Max Pool Size：最大连接数 100

Min Pool Size：最小连接数 0

Pooling ：是否启用连接池 默认为true

Stopwatch 类，用于对比启用连接池和不启用连接池的耗时

测试：测试连接池启用与否的耗时？

测试：连接池类别区分测试

5.SqlCommand 类 执行命令类

命令 ---T-SQL 语句或存储过程

SqlCommand是什么？

对sqlServer数据库执行sql语句或存储过程的封装类。对象即ado.net中执行数据命令的对象

属性：Connection 要使用的链接

CommandText 要执行的SQL语句或存储过程名

CommandType CommandType.Text 默认 ---执行的是一个sql语句

CommandType.StoreProcedure ---执行的是一个存储过程(如果是存储过程，必须设置)

Parameters 对象的命令参数集合，默认为空集合

Transaction 获取或设置要在其中执行的事务

创建方式：

通过构造函数

通过conn.CreatCommand() 方式创建执行命令对象

执行命令方法：

①cmd.ExecuteNonQuery(); **同步方法，对应的还有异步的方法**

执行sql语句或存储过程并返回受影响的行数

命令类型：插入、更新、删除---DNL操作。执行之前，conn链接状态一定要打开

②cmd.ExecuteScalar();

执行查询语句或查询过程，返回查询结果集中的第一行第一列的值，忽略其他行或列

命令类型：查询---DQL操作

适用：做查询，返回一个值，记录数、数据运算而出的结果

有时，插入数据后，想返回自动生成的标识列的值，可以使用它，如insert into user values(‘小明’)；select @@identity;

③cmd.ExecuteReader();

查询，返回一个对象：SqlDataReader，这个对象是一个读取数据的数据流。

实时读取，游标 指针。读取方式较固定，只进不出，只读。

dr 读取数据整个过程，conn必须保持open状态，如果关闭，是无法读取数据的。

while(dr.Read()) // 是否可以前进到下一条记录

{

Int userId = int.Parse(dr[“UserId”].ToString())

}

dr 读取数据过程中，要即时保存，因为他是读一条，丢一条

适用：只是读取数据，不做修改的情况。适用于数据量较小的。因为读取占用链接，如果数据量大，会造成程序速率低

CommandBehavior.CloseConnection dr与链接关联起来

连接使用原则：最晚打开，最早关闭

count(1) 伪列，有多少数据，查询出多少行

6.SqlParameter 参数

表示sqlcommand对象的参数，或与DataSet中列的映射

预防sql注入，解决转义对程序的影响

常用属性：DbType 参数的SqlDbType (数据类型，数据库的类型而言)

Direction 参数的类型：输入 输出 输入输出、返回值类型 默认输入类型

ParameterName 参数的名称

Size 最大大小，字节为单位

Value 参数的值

SqlValue 作为sql类型的参数的值

构造方法：

command 添加参数的方式：

cmd.Parameters.Add(); // 添加单个参数

cmd.Parameters.AddWithValue(“@UserName”,”xiaoming”) // 添加单个参数

cmd.Parameters.AddRange(params); // 添加单个参数

①输入参数：一般参数化sql语句或存储过程，默认使用的参数---输入参数

以下三种：用在存储过程里

②输出参数

output 程序中是可以接收到存储过程里需要返回的值，而不用return 的

Direction:Output

create proc GetDeptName

@DeptId int,

@DeptName nvarchar(50) output ---输出参数

as

begin

select @DeptName=DeptName from DeptInfos

where DeptId=@DeptId

end

③输入输出参数，也叫双向参数

它需要传入值，也输出值。不用return。也用output 标识

Direction:InputOutput

需设置value值

create proc GetDeptName

@DeptName nvarchar(50) output ---输出参数

as

begin

select @DeptName=DeptName from DeptInfos

where DeptName like ‘%’+ @DeptId+’%’

end

④返回值参数

它不出现在存储过程中 只可以返回的int类型，通过return 返回

Direction:ReturnValue

create proc GetUserAge

@UserId int

as

begin

declare @age int

select @Age=age from DeptInfos

where UserId=@UserId

return @age

end

存储过程执行：exec GetUserName 31;

7.SqlDataReader

定义：提供一种sqlserver 数据库中读取只进的行流的方式

特点：快速的、轻量的、只读的，遍历访问每一行数据的数据流。向一个方向，一行一行的，不能向后读取，不能修改数据

缺点：不灵活，只适合数据小的情况，读取数据，一直占用链接

读取方式：Read() 获取第一行的数据，指针又指向下一行的数据，再次调用Read() 方法

当调用Read()方法返回false时，就表示不再有数据行

注意：链接对象一致保持open状态，如果连接关闭，是不能读取数据的。

使用完成后应该马上close()关闭，不然reader对象会一直占用连接的

创建方式：是不能直接构建的，com.ExecuteReader()来创建。

cmd. ExecuteReader(CommanBehaviour.CloseConnection) 的好处，关闭reader对象时，就会自动关闭连接。

读取时尽量使用与数据库字段类型相匹配的方法来取得对应的值，会减少因类型不一致而增加类型转换操作性能损耗

没有读取到末尾就要关闭reader对象时，先调用cmd.Cancel()，然后再调用reader.Close()

cmd.ExbuteReader()获取存储过程的返回值或输出参数，先调用reader.Close()，然后才能获取参数的值。

重用属性：

Connection：获取与Reader对象相关的SqlConnection

FiedCount 当前行中的列数

HasRows reader是否包含一行还是多行

IsClosed reader对象是否已关闭

Item[int] 列序号，给定列号的情况，获取指定列的值dr[1] 是object类型

Item[string] 列名，获取指定列的值

常用方法

Close() 关闭dr

GetInt32(列序号) 根据数据类型相匹配的方法

GetFieldType(i) 获取数据类型的Type

GetName() 获取指定列的列名

GetOrdinal(列名) 获取指定列名的列序号

Read() 使dr前进到吓一跳记录

NextResult() 使dr前进到下一条记录

读取

Read()

数据量小的情况，高效

8.DataTable

概念

DataSet：数据在内存中的缓存（数据库）。是Ado.Net核心组件。

DataTable：内存中的一个表（数据库中的一个表）

应用：DataSet中的成员使用中，它可以独立创建与使用。结合DataAdapter使用。

架构：列和约束来表示，

列 即 DataColurmn

行 即 DataRow

构造：DataTable() DataTable(表名)

常用属性

Columns 列集合

Row 行集合

Contraints 约束的集合

DataSet 所属的DataSet

PrimaryKey 主键(列的数组)

TableName 表名

方法

AcceptChanges() 提交更改

RejectChanges() 回滚更改

Clear() 清空数据

Copy() 复制架构和数据

Clone() 只复制架构，不包含数据

Load(IDataReader) 通过提供的IDataReader，用某个数据源的值来填充DataTable

Merge(DataTable) 合并，将指定的dt合并到当前dt里

NewRow() 创建一个DataRow，与dt具有相同的架构

Reset() 将dt重置到最初状态

Select() 获取dt所有的行的数组（条件、排序）

9.DataSet

概念：数据在内存中的缓存---内存中的数据库

Ado.Net 中的核心组件，将数据库中的数据加载到内存中。可以断开连接

成员：由一组DataTable 组成。DataRelation 相互关联，一起实施了数据的完整性

引用：三种，结合DataAdapter使用，

DataAdapter 将数据填充到DataSet中

DataAdapter 将DataSet中的更改提交到数据库

XML文档或文本加载到DataSet中

作用：ds将数据加载到内存中来执行，提高了数据访问的速度。提高硬盘数据的安全性，程序运行的速度和稳定性

特性：独立性，不依赖于任何数据，支持离线和获取数据时链接。数据视图

创建：DataSet() DataSet(名称)

常用属性

DataSetName ds名称

Tables ds 的 table集合

Ralations ds 的表关系

常用方法：

ds.AcceptChanges() 提交更改

ds.RejectChanges() 回滚

ds.Clear() 清除所有表中所有行的数据

ds.Copy() 复制结构和数据

ds.Clone() 赋值结构，不包含数据

ds.Merge() 合并

ds.Reset() 重置

ds.Load(IDataReader) 将数据源的值填充到ds

10.DataRelation

关系：一对一 一个表列数太多，拆分成两张表

一对多 如一个部门表对应多个员工表 父----子

多对多 中间表建立，如权限分配

创建约束：限制数据 规则，阻止

关联： 相互读取，

使用：通过关系，相互读取

11.SqlDataAdapter

是什么？

适配器（桥接器），DataSet数据之间用于检索和保存数据的桥梁

SqlDataAdapter类 用于填充DataSet以及更新数据源的一组数据库命令和一个数据库连接

是DataSet和SQLServer之间的桥接器

是如何提供桥接的？

Fill 方法，可以将sqlserver 查询出来的数据填充到DataSet里，Update 更改提交到数据库，从而使数据保持一致。

和SqlConnection、SqlCommand 一起使用，提高访问速度

4个重要属性

SelectCommand 查询 SqlCommand

InsertCommand 向数据源插入记录

UpdateCommand 更新数据库的记录

DeleteCommand 删除库中记录

SqlDataAdapter 对数据的操作也是建立在SqlCommand基础之上的。

4中创建方式

构造方式

带command 的构造方式

带查询语句与链接对象的构造方式

带查询语句与链接字符串的构造方式

Adapter 填充 DataSet/DataTable Fill 填充到ds 或 dt中，链接可以不open

表名映射要在填充之前进行设置

连接与断开两种方式填充数据的执行机制

Fill 断开式：没有显示打开conn。----打开conn，da来做这件事---获取数据填充到ds/dt---自动关闭

断开式给我们的错觉是断开的，没有与数据库建立连接，只是这个过程都是由da在Fill过程中完成的。

连接式：显示打开连接，Fill前后conn都open。Da保证conn状态执行fill前后一致。

断开式与连接式区别：连接式填充过程比断开式速度快。

推荐选择连接式：可以一次性加载到内存中，加载完毕后，可以断开conn，不必一直占用连接。速率更快

更新方式 Update(ds/dt/rows) DataRow 有RowState 状态。Unchanged Added Modified Deleted 四个状态

更新时，需配置command执行命令

Modified----以修改的---da.UpdateCommand

Added 已添加 da.InsertCommand

Deleted 已删除 da.DeletedCommand

配置对应的命令属性的方式

方式一：SqlCommandBuilder 自动为da 配置相关的命令

方式二：手动配置SqlCommand

12.Adapter 与 Reader的比较

相同点

执行查询，将数据读取出来，一个或多个

不同点

Reader 提供一种从数据库中读取数据的流的方式

Adapter Daset 与数据库之间的一个桥接器。Fill Update

性能对比：速度 占用内存 连接状态 适用数据量 读取方式 是否可读

Reader Adapter

速度 块 慢（数据量小的情况，速度快慢不明显）

数据量 小 da

内存 小 大

连接 一直占用 断开与连接

读取方式 从头读到尾 一次性加载到内存中（一个或多个）

一条一条读 任意读取

读一条丢一条

只向前不能后退

即时存储，不灵活 灵活

只读，不能修改 可读可写

当不要求随意读取，不修改，数据量小的情况下，sqlDataReader --- 速度快，占用内存小

当要求随意读取，可以修改，数据量大的情况下，sqDataAdapter --- 灵活性大，占用内存大，不一直占用连接

13.Ado.net 调用数据库事务

事务

将一组操作当做一个操作来执行，一致性提交。有回滚，结果：要么一致性提交，要么回滚

执行过程：如果成功，提交。如果失败，回滚。插入过程中出现异常，因为并未提交到数据库，维护数据的完整性。

事务特性：一致性、原则性、独立性、永久性

程序中如何调用事务？事务可以放在存储过程里

create proc AddUserByTran

@UserName varchar(50),

@UserPwd varchar(50),

@Age int,

@DeptName nvarchar(50)

begin

begin tran

begin try

-- 插入部门信息

Insert into deptInfos(DeptName) values (@DeptName);

declare @deptId int

select @deptId=@@identity

-- 插入用户信息

Insert into UserInfos values(@userName,@userPwd,@age,@deptId)

commit tran -- 提交

return 1；

end try

begin catch

rollback tran;

return 0;

end catch

end

go

Ado.Net 如何调用事务？

实质：调用一个普通的存储过程，里边的逻辑是一段事务代码

批量导入---用数据库事务，不可能传入几千行参数，不具备可行性

Ado.Net SqlTransaction 开启事务，循环生成操作，传入不同的参数值。

与数据库事务有何不同？

Ado.Net中一切操作都是由SqlCommand来操作的

开启事务：conn.BeginTransaction(); 将事务赋值给 cmd.Transaction

事务中一般执行的都是增删改。创建事务是，还可以设置隔离级别

为什么要封装通用类？

数据交互：增删改查，相同的事：连接 T-SQL命令。Command 执行命令，选择执行方式得到相应的结果。关闭连接

每一个操作都要经过这一过程，代码都杂乱无章，不清晰。

改善：将一些相同的逻辑，封装起来达到通用的目的。代码重利用，提高效率，不再考虑交互过程，逻辑清晰

封装内容：

1.配置连接字符串，建立连接，连接释放---using()

2.SqlCommand三种执行方法，增删改 ExecuteNonQuery() 查询，返回一个值ExecuteScalar()。ExecuteReader() sqlDataReader

3.SqlDataAdapter填充ds dt

4.SqlTransaction CommandInfo:T-SQL 参数数组 CommandType

14.Dapper

unchecked 解决编译时溢出问题，如int类型值是有范围的，超出需放到unchecked{} 代码块中

unsafe 代码块包裹，用到指针

//&取地址运算符，去除存放变量的地址，跨函数传递变量值

\*简接运算符，取出地址中的值。

unsafe{

int i=10;

PointTest(&i);

}

unsage void PointTest(int \*p){}

Dapper 包，基础封装都在这个包里，相当一个复杂的SqlHelper

Dapper.Contrib 扩展包，引入后，可以做单表等操作，不用写sql的操作。

使用缓存：浏览器端缓存，ResponseCache，换浏览器缓存失效

服务器端缓存，services.AddResponseCaching(); app.UseResponseCaching();

MemoryCache 缓存，必须进入到控制器中，到控制器中看是不是有键值

不加特性的话，类名加s会出错。

Dapper 中的特性，也就四五个。Table、Key、Write(false)写入时禁止往里面写入、Computed更新时不更新该字段 等等

core 设置跨域 services.AddCors(); app.UseCors();

15.Dapper 与 vue html页面

讲解封装dapper和使用页面的案例

**基础\_ASP.NET MVC**

1、MVC 与 WebForm 项目结构比较

2、项目发布IIS

依赖于windows---依赖IIS—无法跨平台

2、安装IIS作为服务器（本地）

3、启动或关闭window功能

4、安装Interface Infamation Service（建议全部安装）

5、Window服务器版（window云服务器）---通过服务器管理期来增加角色和功能

准备发布：

1、打开IIS

2、添加一个网站。物理路径指向—项目的根目录，设定一个端口号（建议在5000---10000之间）

3、直接访问

发布：

邮件鼠标，选择发布按钮---设置一些配置，可以发布了

问题：权限问题，设置网站所在物理路径的权限

1.增加账号IUser---设置权限为完全控制

2.增加账号IIS\_IUsers 设置权限为完全控制

3、项目发布IIS

依赖于windows---依赖IIS—无法跨平台

6、路由

生活中路由---路由器---转发信息

Asp.net mvc Route 怎么转发请求？

客户端发起请求---到大iis---转发到程序集---经过一个路由匹配---转发到匹配的控制器---匹配的Action中取处理

路由规则是通过key,value 方式存储的，url 的规则通过正则表达式进行匹配

如果声明多个路由，name名称不能重复

路由的匹配是去掉整个url中协议的部分，IP以及端口号

忽略路由：mvc 之前都是使用webForm即aspx页面；iis通过后缀来匹配转发到webForm处理流程。Mvc出现后，iis打了一个补丁，如果监听到是mvc请求，就给添加一个后缀；再后来iis升级后可以支持mvc了，不需要通过后缀来匹配了。

之前的补丁（添加后缀）就没有必要了，为了兼容之前的版本，来一个忽略。

7、mvc插件式开发

控制器可以在其他地方的，因为网站在启动的时候需要做初始化，扫描控制器然后保存控制器；

扫描控制器的规则：看是否继承自Controller父类。当强求来了，要使用控制器实例的时候，从存储介质中取出来然后再初始化。

可以把控制器部分（逻辑计算部分可以独立开，放到一个类库中）

把一个大的业务逻辑做拆分；每一个类库分别处理部分内容；每一个内容当做一个插件来完成；大项目一模块一一独立出来做一个插件；部署的时候，只要把所有的插件加载进来，就可以做到多个插件组合成了一个大平台

热插拔---网站没有停止；发布的时候如果有新功能发布，不需要停止服务器，可以吧一个新的插件发布上去。

步骤：准备好插件；触发一个增量初始化的动作，注册路由规则，扫描控制器

8、Razor语法 混编

在视图中可以写入各种html标签

cshtml 并不是一个html页，而是一个cs文件，可以识别html各种标签。cs类型后台代码的文件。Mvc可以由后台程序员全权完成整个项目开发。

cshtml即可以写html，也可以写后台代码。后台代码需要包裹在@{} 块中

@if 判断可以包裹html 标签

@for 中 a标签href 要连接到第三方网站，需要指明协议http等。如果没有指定，会认为是当前网站中的某个Action

@{} 中如果想要展示没有标签的文本，可再前面加上@：或<text></text>

使用partial 页 （局部视图）

方式一：@{ Html.RenderPartial(“partialPage”,”这里是partial 页面”) }

后面值传入到页面的值，@Model string 接收

方式二：@Html.RenderPartial(“partialPage”,”这里是partial 页面”)

重新生成新页面

方式三：@Html.Action(“ChildAction”,”Second”, new {name=” Html.RenderAction”})

返回的是字符串，放入当前位置，需要经过Action处理

方式四：@{Html.RenderAction(“ChildAction”,”Second”, new {name=”Html.RenderAction”});}

在指定位置添加一个view,返回void 需要放入大括号，需要经过action 处理

方式三和方式四ChildActionOnly 特性，只能被子请求访问，不能独立访问

在嵌入页面中需要用到js，需要放到下方包裹中

@section script{

<script></script>

}

扩展：后台代码生成页面标签元素。 通过mvc的TagBuilder类 HtmlHelper类 创建，返回MvcHtmlString 类对象。

@Html.ActionLink() // 用来生成html中的a标签

@Html.RouteLink() // 用来生成html中的a标签

@Html.TextBox() // 生成input 标签

9、MVC 页面布局

新建的项目中，默认的页面展示其实是分为上中下三个模板 header body footer

请求不同的页面，只是中间的body部分变化，header和footer部分没有任何变化。

默认情况下，cshtml页面的加载：都需要加载一个ViewStart；ViewStart把cshtml嵌套在内；ViewStart可以删除

在ViewStart中使用了layout=”引入母版页”

母版页：一个页面的布局结构

母版页提供一个@RenderBody()，区域占位符；需要展示页面的时候，把页面嵌套在母版页的@RenderBody()位置，形成一个完整的页面。

母版页就是一个公共的区域：如果每个内容页都需要使用到的文件 js css，可以直接引入到母版页，内容页就可以使用了

指定母版与加载机制

我们来看\_ViewStart.cshtml页面

@{

Layout = “~/Views/Shared/\_Layout.cshtml”;

}

这句代码指定了默认的母版的位置，当前应用程序根目录下<”~”的含义>的View/Shared/\_Layout.cshtml

除非特殊的情况，比如视图是Partial视图，或显示的在视图中添加以下代码指示不适用母版

@{

Layout=null;

}

10、JS Css 捆绑加载

Bundle 捆绑压缩

为了能够把多个js/多个css 捆绑起来，统一调用

可以不断的Include，就可以把多个js或多个css 捆绑在一起；页面中调用的时候，可以只调用一个，却达到调用多个的目的。

捆绑还可以把js/css 压缩，体积更小，降低加载的消耗。

@Styles.Render(“~/Content/css”)

@Scripts.Render(“~/bundles/modernizr”)

出现异常：一般模板页引用js 文件放到最下方，而内容页中添加js 代码后 提示 $ 不存在（母版页已引入）

解决：在模板页中引入js代码下方，添加@RenderSection(“script”, required:false)

内容面中使用js 使用如下模板

@sections scripts{

<script></script>

}

11、连接数据做数据交互 EF6

在分层架构后，为了做到层与层之间的解耦；需要依赖注入，就必须要使用到IOC容器了

Unity

准备工作：抽象和细节；

抽象：接口/抽象类

细节：普通类

Unity：开始使用

nuget 引入Unity；初始化UnityContainer；注册抽象与细节；获取对象实例

IOC容器的作用：创建对象；本质就是一个工厂；

**依赖注入**：构建某一个对象的时候，如果对象A依赖对象B。在构建对象A的时候自动将对象B构建好，然后传递进来。

分三种方式：构造函数注入 + 属性注入 + 方法注入

构造函数传递：构造函数注入

属性传递 ：属性注入，需要在属性上面标记一个特性Dependency

方法参数传递：在构造完了对象之后，根据需要去执行某一个方法，也会根据这个方法需要的参数，自动构造出对象，然后传递给方法的参数。也需要标记一个特性 InjectionMethod

12、IOC容器整合MVC

通常通过配置文件整合IOC容器到MVC

第一步：nuget 引入Unity.Container + Unity.Abstractions + Unity.Configuration + Unity.Interception

第二部：添加Unity.Config 配置文件（设置成始终复制），在Unity 节点下配置需要的服务，注册抽象和具体之间的关系；设置为始终复制。

第三步：读入配置文件，把配置文件的信息，注册到UnityContainer容器中

第四步：就可以获取实例了。

问题：使用IOC，初始化很代码很多，每次使用都需要初始化IOC容器？

封装一个Factory类，将代码放到静态构造函数里，进程开始时，会自动调用静态构造函数，封装获取ioc容器方法即可。

静态构造函数，进程中只执行一次。

13、MVC + IOC支持AOP扩展

AOP面向切面编程：

可以在不修改之前业务逻辑的基础上，动态的增加新的功能。

可以在执行某一个行为之前做点什么事儿？查询一个用户；在查询用户之前，检查一下参数 和 记录一下日志

Unity 支持AOP

第一步：Nuget 引入 Unity.Interception + Unity.Interception.Configuraton 程序集

第二步：增加配置文件的2个节点

第三步：增加需要的AOP动作，新增一个记录日志类，该类必须实现 InterceptionBehavior 接口

14、如何做到控制器支持构造函数注入？

IOC容器就是用来创建对象的实例的；通过依赖注入；

调用MVC的某一个Action(方法)一定要先实例化控制器。如果能够使用IOC容器来创建控制器的实例，就可以直接通过依赖注入把我们需要的Service给注入进来。

经过源码解读：

Mvc的控制器其实是通过一个控制器工厂来创建的。控制器既然由工厂来创建，如果我们给换一个工厂，就可以自定义工厂如何去创建控制器的实例；默认获取的控制器工厂是DefaultControllerFactory；如果我们需要换一个工厂，就可以把这个DefaultControllerFactory替换掉，就可以自己定义创建控制器的动作；

如何让我们自己定义的控制器工厂在框架中生效？

做到了自己定义的控制器工厂生效了，请求来了以后，会先到控制器工厂，让后通过工厂去实例化控制器。

15、Asp.net mvc 对AOP的支持

Filter:

Action 对于方法AOP的支持

Result 对于结果AOP的支持

AuthorizationFilter 授权对于aop的支持

Exception 异常---通用aop来支持

以上四种支持AOP的技术手段：通过特性Attribute来支持的

AOP：在不修改之前的业务逻辑基础上，动态的添加一些功能

**知识点一：ActionFilter**

ActionFilterAttribute 是一个抽象类，不能直接使用，用来让我们扩展的

IActionFilter 是一个接口，用来让我们扩展的

扩展ActionFilterAttribute:

第一步：新建一个特性，以Attribute结尾，继承ActionFilterAttribute

第二部：覆写其中的四个方法

第三步：标记在某一个Action方法上

第四步：请求来了，OnActionExecuting --- 进入方法 --- OnActionExecuted --- OnResultExecuting ---渲染视图 --- OnResultExecuted;

扩展IActionFilter

第一步：新建一个类，以Attribute结尾，继承Attribute，实现了接口IActionFilter

第二部：实现接口，实现方法

第三步：标记

第四步：请求来了以后，发现并没有进入到扩展的ActionFilter中去，为什么？

因为框架的内部，不是判断是否是特性，不是通过Attribute来判断；而是判断是不是一个FilterAttribute---MvcFilter

**知识点二：ActionFilter 的应用 + Filter多种注册**

请求进入到Action之前或者是请求经过了Action之后---扩展业务逻辑

1.记录日志

2.提高性能方面---压缩

压缩：浏览器发起请求到服务器，服务器响应结果给浏览器；响应的数据肯定存在大小区别；如果能够在响应数据上，把数据压缩，数据变小了，性能可以提高；

压缩条件：需要浏览器和服务器同时支持；

浏览器发起请求---服务器看浏览器支持哪些压缩格式，服务器根据浏览器支持的压缩格式做压缩，通知标记给响应结果，告诉浏览器已经使用了哪个格式做了压缩。浏览器就知道使用哪种格式做解压然后展示结果。

**知识点三：注册方式**

方式一：标记在Actioin 上，只对当前Action生效

方式二：标记在控制器上，可以让控制器下所有的Action都生效

方式三：全局注册，对项目中的所有方法都生效

**知识点四：ResultFilter**

结果的Filter：在生成结果之前或者在生成结果之后，可以插入一些自己的业务逻辑。

1.可以通过ActionFilterAttribute继承扩展---请求来了以后，先进入到OnResultExecuting---视图渲染---OnResultExecuted

2.可以通过实现IResultFilter接口来扩展

引用场景：跟视图有关，如果想要在视图生成之前或者之后做点什么，就可以考虑使用ResultFilter

**知识点五：ExceptionFilter**

用来处理异常的Filter，常规的解决异常—try-catch 包裹；再处理

方式一：HandleErrorAttribute默认实现

方式二：IException接口

难道异常都能捕捉到吗？并不是都能捕捉到；

异常的补充：

在MvcApplication中增加方法：Application\_Error

只要响应不是200，就可以被这里捕捉到

protected void Application\_Error(object sender, EventArgs e)

{

Exception exception = Server.GetLastError();

Response.Write(“System is Error…..”);

Server.ClearError();

}

**知识点六：AuthorizeAttribute**

为了验证权限而存在的，系统开发中必不可少

1.登录授权---给服务器用户名和密码，服务器验证用户名和密码。把用户信息写入到Session，在返回的时候，把SessionID返回到客户端，写入到Cookie中

2.再次访问的时候，验证是否有权限---浏览器会自动的带上SessionID，服务器就要找SessionID，如果找到了，就表示登陆过；就验证通过了；否则就跳转到登录页；

**知识点七：匿名**

如果我需要让所有的Action都支持权限验证？全局注册；---是所有的Action都生效的

因为全部都生效了，全方位覆盖了；

会有误伤---其实有部分的Action是需要不验证的；

**知识点八：退出**

其实就是把登录后写入的session数据清除掉

16、Asp.net mvc 管道

**一、Http请求的全流程**

1.用户浏览器输入地址

2.DNS解析（域名供应商）：将输入的网址解析成IP+端口号

3.请求到达服务器Server:IP在互联网定位服务器，端口确定进程，端口还可以带有协议信息，用于穿透防火墙

4.HTTP.SYS服务接收HTTP请求，交给IIS

5.IIS将请求发到对应的程序处理

6.请求包装成HttpWorkerRequest对象

7.传到AppDomain的HttpRuntime入口

8.开始管道模型

9.IIS接收返回的数据流，给HTTP.SYS

10.HTTP.SYS将数据给客户端（浏览器）

**二、**

**三、AspNET\_ISAPI如何把请求转交给Asp.net 管道**

HttpContext：上下文

HttpRuntime：Http运行时

在Asp.net中，任何一个请求的处理都需要让IHttpHandler来处理

**四、管道处理模型的核心设计**

到HttpApplication;

可能需要经过很多个环境的处理

1.验证权限

2.缓存的判断

3.异常处理

….

有多个环节的处理，如果再HttpApplication中直接写死；请求来了以后，中间任何一个环节都是需要这样处理的；且不能扩展；

如果我们希望能走做到扩展？怎么做？

框架：---有固定处理环节，也有不定的处理环节；

使用的是观察者模式；

把固定的操作封装在一起，把不定的处理环节，封装在一个事件中，就可以在初始化的时候，给事件注册动作；请求来了以后，就可以去执行事件+固定的动作；形成一个执行链；固定的动作是会执行；事件也会执行；不定的动作就可以通过注册事件；把需要执行的动作给加入到处理环节。

**五、HttpModule**

在框架中，事件动作的注册究竟是谁来完成呢？

HttpModule：管道

自定义CustomHttpMoudle

1.新建一个CustomHttpModule类，实现IHttpMoudle接口

2.实现方法---在方法中，给几个事件注册动作

3.要让CustomHttpModule，生效---增加配置文件（本地电脑有一个默认配置文件C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config\WebConfig .NET 框架的默认配置，默认情况都是按照这个配置来执行的---慎重修改）

4.可以在我们的项目中的Web.config中进行配置

<system.webServer>

<modules>

<add name=”扩展全称,项目名称”/>

</modules>

</system.webServer>

5.任何一次请求来了以后，都会执行我们注册的这个CustomHttpMoudle中的注册动作；

6.这里的这个配置其实是一种请求级的配置扩展；任何一次请求来都要经过这里的处理。

**六、HttpModule 扩展和Global事件**

只要是请求级的处理环节，都可以通过HttpModule来扩展；

1.记录日志 --- 统计访问量

2.缓存

3.拦截请求

MVC --- mvc其实是Asp.net 管道中的一个扩展。

在httpModule中发布一个事件；事件执行的动作，可以配置成在Global中的一个方法

Global中的一个方法：命名：配置HttpModule配置文件的时候给定的一个名称\_在HttpModule中事件的名称。

在执行HttpMoudle中的事件的时候，就会触发Global中的CustomHttpModuleRichard\_CustomHttpMoudleEvent方法

**1. 进阶**

1.**方法汇总**

**静态方法**

特点：一旦创建直到应用程序结束才会结束（生命周期）；全局的；效率高；

使用场景：用户登录信息、系统配置信息、系统设置信息、SQLHelper类

注意：静态的东西创建多了，占用内存会很大。所以，不是必要不要创建。

调用：类名.方法。不用实例化；静态方法调用非静态方法，不能直接调用，要间接调用(把所在类实例化后调用)

**构造方法**

用处：初始化对象；或初始化数据。在内存中开辟一块空间。用于保存数据。

特点：默认是一个无参数构造方法，可多个并重载

**析构方法**

作用：释放对象，GC垃圾回收在调用 ~Dispose(){}

垃圾回收机制：回收非托管资源。平常类放在内存里，不用手动释放，垃圾回收机制会自动释放。

系统级资源，需要手动释放

using(SqlConnection sql = new SqlConnection()){} 这是一个语法糖

finally{}

close 清空，但不释放等GC处理。Dispose() 完全释放

Dispose()需要实现IDisposable接口

Close()和Dispose()区别 Close()方法关闭对象，没有完全释放；Dispose()方法完全释放，使用时需重新创建。

99%情况下不需要自己编写经典方法。

**虚方法 virtual 子类override 重写方法**

作用：允许子类/派生类对父类方法进行重写，实现和父类不同的功能，方便扩展。参数和返回值类型必须一样。

特点：好维护

**抽象方法 abstract override 重写**

抽象方法必须写在抽象类里，并且加个abstract 关键词。只是定义；不能new；不带方法体，子类一定要实现。

使用场合：强制性一定要实现的。

注意：抽象方法与接口区别？

抽象类(本质上是类)单继承；接口可以多继承。

抽象类里可以写普通方法，虚方法；接口只能写规范，不能写实现

使用场合：抽象类常用于不会经常改动、然后抽象范围大一些的事件。

接口适用于经常修改，只是一个规范的地方。

**扩展方法**

定义：在非泛型静态类里的静态方法，this 指向调用对象

使用场合：扩展方法，在linq链式编程。

调用密封类(sealed)里中的对象，属性，或方法(扩展密封类)。扩展接口，linq链式调用。

**2.泛型**

泛型：把类型做到通用，代表动态

object 类型做数据转换的时候有拆箱、装箱操作有性能损耗，尽量少用

泛型集合 Dictionary<k,v> 键值对(泛型字典) 尖括号一般都是泛型，key值不能重复

Dictionary<int,string> dic = new Dictionary<int, string>();

List<T>

**自定义泛型**：最大优点就是通用性。

**泛型约束**

public static void Show<T>(T t) where T: new() // 约束一：约束，表示T类型必须带一个无参数的构造函数，我们可以实例化它。

如接口就没有无参数的构造函数。

约束二：struct 值类型约束，struct/int、double、bool、枚举

约束三：class 引用类型约束，数组、类、接口、委托、object、字符串

约束四：自定义类型约束

多个约束时，new()一定要写在最后面

**协变逆变**

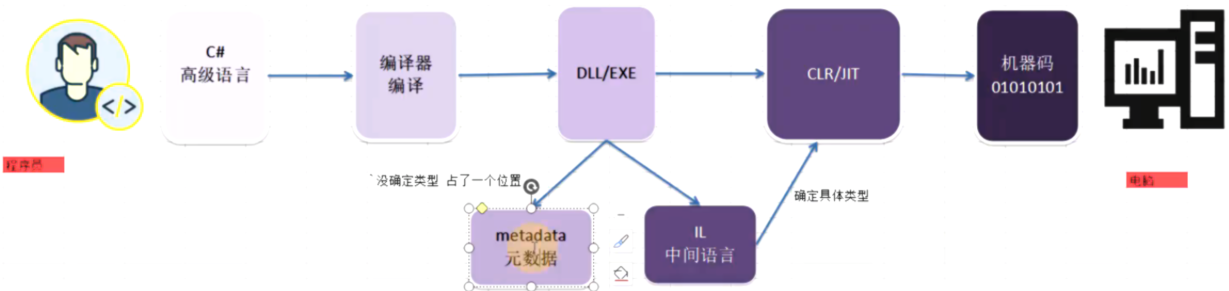
协变逆变是针对接口和委托来说的，离开了他们就没有这种说法。

out 关键字是协变，in 代表是逆变

应用场景：在知道自己或者别人以后有用到父类通过子类实例化，或子类通过父类实例化的情况化使用out或in 关键字

**总结**：泛型的最大优点就是通用性。使用灵活，减少通用代码

用处：让泛型类、泛型方法、泛型接口、泛型委托更通用。减少代码量。



exe/dll 主要区别就是exe文件有入口

元数据：描述exe/dll文件的一个数据清单。通过反射来操作或获取元数据

**3.反射** Reflection

Type 就是反射的类型

定义：读取元数据(dll)文件描述信息的一个类库。

作用：提供用户动态的能力

操作对象：就是

**使用场景**

更新程序时(更新自己的dll)

使用别人的dll文件(这种可以读取别人的私有的东西)

哪些地方用到了：asp.net MVC、ORM、LOC、AOP 几乎所有的框架都会使用反射。

**为什么要用反射区操作类、方法、特性、属性等操作？**

①因为我们需要动态，②读取私有的对象

定义：就是一个操作metadata的一个类库(可以把反射当成一个小工具，用来读取或者操作元数据的)

获取type方式：

方式一：使用Assembly获取(获取命名空间下面的)

方式二：通过对象实例获取 obj.GetType();

方式三：通过引用类型来获取 typeof(Student)

注意：路径一定要在本项目中。

**反射创建对象**

Assembly assembly = Assembly.load(‘Ant.DB.SqlServer.dll’); // 加载dll 文件

Type type = assembly.GetType(“Ant.DB.SqlServer.SQLServerHelper”); // 获取类型，找到项目下类(完整类型名称)

Object odbHelper = Activator.CreateInstance(type); // 将类型给它，创建实例对象。和new 创建实例一样。

IDBHelper dbHelper = odbHelper as IDBHelper; //类型转换(as 转换不报错，类型不对返回null)。创建这个接口为了通用。

**反射创建对象带参数**

Assembly assembly = Assembly.LoadFrom(‘Ant.DB.SqlServer.dll’);

Type type = assembly.GetType(“Ant.DB.SqlServer.SQLServerHelper”); // type.GetConstructors() 获取类的所有构造方法

foreach(ConstructorInfo ci in type.GetConstructors()) // 获取到所有的构造方法

{

foreach(var para in ci.GetParameters()){} // 获取到构造方法的所有参数类型

}

Object odbHelper = Activator.CreateInstance(type, new object[]{“Ant编程”}); // 调用带有string 类型参数的构造函数，类型相同的构造方法

**私有构造方法**

正常私有构造函数用new 关键字无法实例化

Object odbHelper = Activator.CreateInstance(type, true); //加个true就可以创建私有的构造函数

该功能用在我们的单例模式里(一个对象只能创建一次)。这个也叫反射破坏单例

**反射创建泛型类**

Type type = assembly.GetType(“Ant.DB.SqlServer.SQLServerHelper`3”); // 创建泛型的格式，返单引号和泛型个数

Type makeType = type.MakeGenericType(new Type[]{typeof(int), typeof(string), typeof(double)});

Object o = Activator.CreateInstance(makeType);

**反射调用方法**

普通都，创建对象后可直接调用。如dbHelper.Add(model); 等

Object o = Activator.CreateInstance(type);

情况一：无参方法调用

MethodInfo methodAdd = Type.GetMethod(“Add”); // 方法名

methodAdd.Invoke(o, null); // 调用无参数方法

情况二：有参方法调用

MethodInfo methodAdd = Type.GetMethod(“Add”, new Type[]{typeof(string), typeof(int)}); // 方法名

methodAdd.Invoke(o, new object[]{“Ant编程”, 3});

情况三：无参数重载方法

MethodInfo methodAdd = Type.GetMethod(“Add”, new Type[]{}); // 方法名

methodAdd.Invoke(o, null);

情况四：静态方法调用，对象可以为空，直接给方法名称和参数类型

**反射与MVC 和 AOP**

dll文件名称 + 类型名称 + 方法名称(就可以拿到我们的方法)

反射在mvc中的一些缺点：加个httpPost等解决。其实是在网络请求中的一些问题，不是反射的问题

AOP：是面向切面编程，是oop对象技术的一种补充。如一个方法功能实现前实现一点东西，或之后实现点中西。

**反射调用私有方法**

MethodInfo methodRemove = type.GetMethod(“Remove”, BindingFlags.Instance|BindingFlags.NonPublic);

methodRemove.Invoke(o, new object[]{“Ant编程”});

**反射调用泛型方法**

情况一：普通类里泛型方法的调用

MethodInfo methodInfo = type.GetMethod(“Test”); 找到调用的方法

var method = methodInfo.MakeGenericMethod(new Type[]{typeof(int), typeof(string)})； 确定方法的参数类型和个数

method.invoke(o, new object[]{1, “Ant”})

情况二：泛型类里面的泛型方法调用

Type type = assembly.GetType(“Ant.DB.SqlServer.SQLServerHelper`3”); // 创建泛型的格式，返单引号和泛型个数

Type makeType = type.MakeGenericType(new Type[]{typeof(int), typeof(string), typeof(double)}); // 确定泛型方法参数类型

MethodInfo methodInfo = type.GetMethod(“Test”); 找到调用的方法

var method = methodInfo.MakeGenericMethod(new Type[]{typeof(int), typeof(string)})； 确定方法的参数类型和个数

method.invoke(o, new object[]{1, “Ant”})

**反射操作字段和属性**

type.GetProperty()

foreach(var prop in type.GetProperties())

{

If(prop.Name.Equals(“StudentName”))

{

Prop.SetValue(obj, “Ant”)

}

}

方法二：

PropertyInfo[] propertyInfos = type.GetProperties(); 查找所有的属性

PropertyInfo id = type.GetProperty(“Id”);

**反射案例：完善SQLHelper类 反射 + 泛型**

Public T Find<T>(int id)

{

Type type = typeof(T);

var sql = $”select {string.Join(‘,’,type.GetProperties().Select(r=>r.Name)) } from {type.Name} where Id={id}”;

var row; // 从数据库取出的dataRow

object obj = Activator.CreateInstance(type);

foreach(var prop in type.GetProperties())

{

prop.SetValue(obj, row[prop.Name]);

}

return obj as T;

}

**4.特性**

定义：就是一个类，继承自Attribute就是特性。像贴标签，贴上标签就产生了新的功能。使用场景：数据验证

分类：默认特性、自定义特性

自定义的特性的方法必须通过反射使用

特性：默认特性，默认在一个类上引用一次。

如果要多次使用，在特性上添加[AttributeUsage(AttributeTarge.Field, AllowedMutiple=true)]特性。特性的特性

如果自定义的特性需要继承，使用AttributeUsage Inherited=true 默认值为true

如果想给自定义特性增加范围，使用AttributeUsage AttributeTargets.Class 等

系统自带

Obsolete 标识特性是否可用

AttributeUsage 标识自定义特性是否有特殊功能

DebuggerStepThrogh

AOP：切面编程，执行代码前或执行后添加一些东西。

**调用 和反射一起使用**

Type type = typeof(UseAttribute);

object[] attributes = type.GetCustomAttributes(true); // 找出所有特性

或者 object[] attributes = type.GetCustomAttributes(typeof(DefindAttribute) ,true);

foreach(obj item in attributes)

{

DefindAttribute defind = item as DefindAttribute;

If(defind != null)

{

Console.WriteLine(defind.ShowInfo);

}

}

总结：特性使用后不会调用的，通过反射进行使用。一个[] 中可以标记多个特性，通过逗号分隔

三步：定义，使用，查找出来

特性还可以标定特性，AttributeUsege(AttributeTarge.Field, AllowedMutiple=true) 该特性用来限制特性的。标在字段上，标多个

案例：查找一个特性

public static boo Validate<T>(T t)

{

Type type = t.GetType();

foreach(var prop in type.GetProperties())

{

if(prop.IsDefind(typeof(AbstracValidateAttribute), true))

{

object obj =prop.GetValue(t);

foreach(var attribute in prop.GetCustomAttributes(typeof(AbstractValidateAttribute), true))

{

if(!attribute.Validate(obj)) // 如果成功了 就继续验证，否则直接返回

{

Return false;

}

}

}

}

return true;

}

// 验证

var student = new Student();

student.Validate();

**5.委托**

委托：本质上是一个类，使用delegate 声明。继承自multicastDelegate

委托声明可以在类内部，也可以在命名空间外。委托操作的目标是方法

委托规范：传递的方法返回值与参数必须与委托一致。

委托作用：其实就是代理，做了一层包装，屏蔽了方法的真实调用

保存的是方法的指针。指向一个方法。类似前端的回调函数。

调用委托的时候这个方法就立即被执行。

可以保存一个或多个方法的信息。用来传递方法，主要用于实现代码的解耦。

委托背景：通过调用委托来调用目标对象方法，保护目标对象的安全细节（替目标对象调用方法）

**泛型委托**

做到通用性

Action<string> action = new Action<string>(Method1); // 官方版本 不带返回值

action(“我是不带返回值的泛型委托调用”);

Func<string, string > func = new Func<string, string>(Method3); // 官方版本 带返回值

func(“我是带返回值的泛型委托调用”)

**多播委托**

【1】每个委托都是继承自MulticastDelegate，也就是每个委托都是多播委托。按照顺序执行多个委托方法

【2】多个委托，形成委托链

【3】带返回值的多波委托只能取到最后一个方法的值

【4】多播委托可以用加减号来操作方法的增加或减少

【5】匿名方法时，不起作用，

【6】添加相同方法时，添加的是相同的实例。

第一种：无返回值

//Action action = new Action(MethodTest)

//action = (Action)MulticastDelegate.Combine(action, new Action(MethodTest2)); // 加上一个

//action =(Action)MulticastDelegate.Remove(action, new Action(MethodTest2)); // 减去一个

简写：

Action action = MethodTest;

action += MethodTest2; // 加上一个

action -= MethodTest2; // 减去一个

第二种：有返回值

Fun<string> func = ()=>{return “我是lamda”};

func + = ()=>{return “我是func1”};

func -= ()=>{return “我是func1”}; // 注意：匿名函数，无法删除，反而把这个加上了。即对象的指向没找到

var str = func();

遍历执行委托链中每个委托

foreach(Action item in action.GetInvocationList())

{

item();

}

**6.事件**

事件：表面上看，是委托上面的引用

从IL文件看，它是一个委托的方法

事件发布：定义的委托事件

事件订阅：把目标类的方法条件赋值到委托

事件触发：调用委托对象

委托的安全版本，定义事件类外部，是不能使用=号来操作，只能使用+=。

在定义事件类的外部不能调用事件。

事件就是在委托的前面增加一个event 关键字。

delegate void StudentDelegate;

public event StudentDelegate event;

public void Invoke()

{

event?.Invoke();

}

【1】定义一个委托

【2】定义一个调用和定义事件的类，为了安全

【3】订阅事件 即绑定事件调用的方法，

【4】调用事件

class Publisher // 事件发布者，事件的定义和调用，触发事件也可以写在里面

{

public event EventHandler<EventArgs> CustomEvent;

public void DoSomthing()

{

OnCustomEvent(new EventArgs())

}

protected virtual void OnCustomEvent(EventArgs e) // 在受保护的虚方法中包装事件的调用，允许子类重写调用行为

{

CustomEvent?.Invoke(this, e);

}

}

class Subscriber // 事件订阅者，事件方法编写和订阅功能

{

private void HanderCustomEvent(object sender, EventArgs e)

{

// 在这里实现功能

}

}

**7. Lambda**

就是一个匿名方法。

**8. Linq 扩展方法+lambda**

语言继承查询—用来做查询的一些操作类库

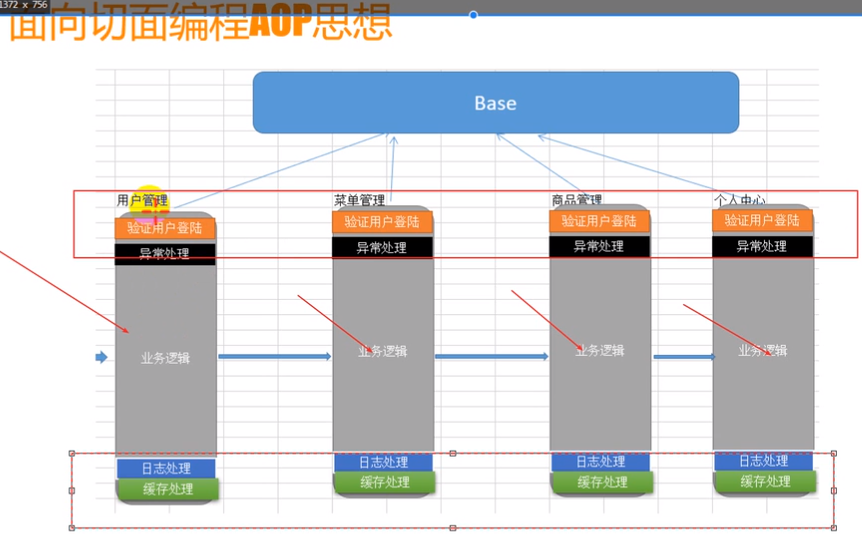
核心就是对数据源的操作。

9 AOP 思想

POP 面向过程编程，按照顺序

OOP 面向对象编程，万物皆对象，最小粒度只能到类。

面向切面编程AOP，是面向对象编程的一个扩展。可以将对象粒度更小花



装饰器模式实现AOP

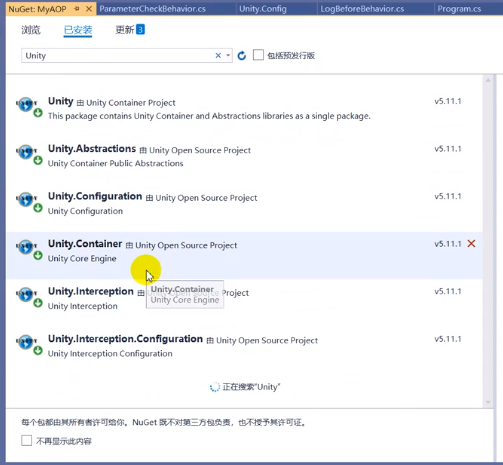
代理模式实现AOP

AOP 静态实现：装饰器模式/代理模式

AOP 动态实现：(Remoting)（Emit）

Unity 动态实现AOP

第一步：nuget 引用Unity，图片中几个都安装



UnityConfigAOP.Show(); // 通过Unity + 配置文件来实现

10. 加密

MD5 加密

不可逆加密：原文—加密—密文，密文无法解密出原文

11. QuartZ 调度服务

什么是定时服务？

答：在指定的时间点去做某件事，循环往复！ 。

QuartZ 定时任务框架，包含三个角色

Ischeduler(某一单元，流程)

ITrigger 从某一时刻开始做事

Ijob 做什么事？

**使用步骤**

nuget添加引用 QuartZ

**传递参数**

jobDetail.JobDataMap.Add()

trigger.JobDataMap.Add

要注意，使用MergedJobDataMap有覆盖，后者为准

如果我上一次执行的结果需要作为下一次执行的参数呢？

[PersistJobDataAfterExecution]执行后的保留作业数据链式传参

**常用定时策略**

SimpleTrigger:从什么事件开始，间隔多久执行重复操作。可以限制最大次数

Cron：表达式的方式，可以灵活定制时间规则。

**2. 异步多线程**

进程、线程、同步、异步的概念

回顾委托，开始异步

异步多线程的三大特点：不卡界面，方法块，无序性

从运行模型开始

计算机的概念：

**进程**：程序在服务器上运行时，占据的计算资源合集，称之为进程。

好处是进程之间不会相互干扰，但是进程间的通信比较困难(分布式)

**线程**：程序执行的最小单位，响应操作的最小的执行流。

线程也包含自己的计算资源，线程属于进程，一个进程可以有多个线程。

**多线程**：一个进程里面，有多个线程并发执行。

**1. Thread**

**多线程Thread**：类，是。NetFramework 对线程对象的抽象封装

通过Thread去完成的操作，最终是通过向操作系统请求得到的执行流。

常用：

CurrentThread：当前线程，任何操作执行都是线程完成的，运行当前这句话的线程。

ManagerThreadId：是.Net 平台给Thread起的名字，就是个int值，尽量不重复。

Thread.CurrentThread.ManagerThreadId;

**同步单线程方法**：按顺序执行，每次调用完成后才能进入下一行，是同一个线程完成的。

**异步多线程特点**：发起调用，不等待结束就直接进入下一行(主线程)；动作会由一个新线程来执行(子线程)。并发了

**注意**：任何的异步多线程都离不开委托delegate。

Action 是.net 框架的一个内置委托,委托实例化action后

执行三种方式：action.Invoke(“aaa”); 同步 action(“aaa”); 同步 action.BeginInvoke(“aaa”,null,null); 异步

多线程写法不难，但是使用好很难。

**2. 核心基础**

比较一：

同步单线程方法卡画面--- 主（UI）线程忙于计算，所以不能响应

异步多线程方法不卡画面--- 计算任务交给子线程，主（UI）线程已经闲置，可以响应别的操作

bs 开发：用户注册发邮件/发短信/写日志等。

比较二：

同步单线程方法慢 ---- 只有一个线程计算 1个线程耗费13000毫秒

异步多线程方法块 ---- 因为多个线程并发计算 5个线程耗费4269 毫秒， 性能只有3倍提升

多线程就是用资源换性能，但不是线性增长。线程并不是越多越好。

a 多线程的协调管理额外额外成本 --- 项目经理

b 资源也有上限的 --- 5两车只有3条到

**问题：**如果beginInvoke 上万次，理论上开启了多少个线程。。。？

TreadPool 是由上限的。

Thread -- 是真的可以启动1w个？当然电脑抗不住，死机了。

比较三：无序性，不可预测性

启动无序，执行事件不确定。即使是同一个线程同一个任务耗时也可能不一样。跟操作系统的调度策略有关。

cpu分片(计算能力太强，1s分拆1000份儿，宏观上就变成了并发的)。哪任务执行过程就得看运气了。

但是线程的优先级可以影响操作系统的调度。

启动无序原因：几乎同一时间向操作系统请求线程，也是个需要cpu处理的请求，也看运气了。

因为线程是操作系统资源，clr只能去申请，具体是什么顺序，无法掌控

**案例**：电商下订单：增加订单表-日志-发邮件-生成支付-物流通知

多线程，有顺序要求，等待500ms执行下一个动作，顺序测试100次都没问题，上线都没问题。。。但是偶尔，一个月总有几次顺序错了，而且无法重现

随着用户增加，数据量累计，数据库压力变大，服务器硬件资源不够，都会影响到执行效率，各种情况都有，所以都错了

使用多线程是，不要通过延时等方法去掌握顺序，不要试图“风骚的多模式”掌控顺序

**3. 解决无序性**

解决多线程的不可预测性

任何异步多线程都是跟委托相关的。

Action<string> action = this.UpdateDB;

action.Invoke(“aaa”); // 同步

action.BeginInvoke(“aaa”,null,null); // 异步，不阻塞，启动一个新的线程解决。

使用场景：用户点击按钮，希望业务操作要做，但是不要卡界面。

案例一：不允许没有监控的项目上线--- 需要在业务操作后记录下日志

解决：AsyncCallback callback = ar=>{ ar.AsyncState; }, beginInvoke第三个参数 会成为这个属性。可以是字符串、对象等…

将callback 放到BeginInvoke 第二个参数上 action.BeginInvoke(“aaa”, callback,null); 第三个参数可以是对象等

案例二：用户必须确定操作完成，才能返回 --- 上传文件，只有成功之后才能预览

一方面文件上传，完成后才预览；另一方面，还希望有个进度提示--- 只有主线程才能操作页面

解决：IAsyncResult

IAsyncResult asyncResult = action.BeginInvoke(“aaa”, null,null); // 启动子线程完成计算

while(!asyncResult.IsCompleted) // 属性，用来描述异步动作是否完成。异步动作完成后会修改这个属性为true;

{}

真实开发中，一开始可以读取到文件的size，然后就直接获取已经上传好的size，相除就是进度。

案例三：信号量等待，应用场景：多个逻辑，都实现了再往下走

var asyncResult = action.BeginInvoke(“aaa”, null, null);

asyncResult.AsyncWaitHandle.WaitOne(); // 阻塞当前线程，直到接收到信号量，从asyncResult 出发。 页面无延迟

waitOne(-1) // 一直等待

waitOne(1000); // 最多等1000毫秒，超过了就不等了。用来做超时控制，

微服务架构，一个操作需要调用5个接口，如果某个接口很慢，会影响整个流程，可以做超时控制，超时就换接口或者放弃或给个结果。

**有返回值**

Func<int> func = this.RemoveService();

IAsyncResult asyncResult = func.BeginInvoke(null, null); // 异步调用结果，描述异步操作的

Int iResult = func.EndInvoke(asyncResult); // 真实返回值

如果想要获取异步调用的真实返回值，只能EndInvoke

Func<string> func = ()=> DateTime.Now.ToString();

String sResult = func.EndInvoke(func.BeginInvoke(null,null)); // 返回值类型，根据委托来

EndInvoke另一种写法

IAsyncResult asynResult = func.BeginInvoke(ar=>{

Int iResult = func.EndInvoke(ar);

}, null)

即endinvoke 即可以在主线程获取，也可以在子线程获取。注意：一次异步，endinvoke 只能操作一次。

总结：现在我们能简单控制线程顺序，把某个动作安排在某个动作之后，但是还没有吧多个线程之间安排等待。

**4. 异步多线程**

版本线程对比

ThreadStart start = ()=>{

Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

}

Thread thread = new Thread(start);

thread.Start(); //1.1 版本，最原始的线程就启动了

优势：Thread api 特别多(丰富)，thread.Start(); Suspend();Resume(); Join(); Abort(); ResetAbort(); 等

缺点：其实玩不好，因为线程资源是操作系统管理的，申请处理后，响应并不灵敏

Thread 启动线程是没有控制的(没有上限)，可能导致死机，真有可能开始1万个线程。

.netFramework 2.0 新clr 提出ThreadPool：池化资源管理设计思想，池化就是一个容器，容器提前申请5个线程，程序需要使用线程，直接找容器获取，用完只有再放回容器(控制状态)，避免频繁的申请和销毁；容器自己还会根据限制的数量去申请和释放。

WaitCallback callback = o=>{

Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

}

ThreadPool.QueueUserWorkItem(callback);

ThreadPool.SetMax… 设置最大数量等。。

优势：线程复用；可以限制最大线程数量

缺点：API 又太少了，在线程等待和控制特别弱。影响使用

Task 被称为多线程的最佳实践

Action action = ()=>{

Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

}

Task task = new Task(action);

task.Start();

优点：Task 线程全是线程池线程；提供了丰富的API

Parallel.Invoke(()=>{},()=>{},()=>{}); // 多线程，主线程也参与运算。节约一个线程。能控制最大并发数量(并行编程)

通过参数ParallelOptions.MaxDegreeOfParallelism 指定最大并发数量。

**5. Task 解析**

Console.WriteLine(“aaaa”);….多个

Task.Run(()=> this.Coding(“aaa”,”a1”));

Task.Run(()=> this.Coding(“aaa”,”a1”));

// 除了明白场景，还要知道线程问题，有严格时间限制、先后顺序的，只能单线程

下面用多线程为了提高效率，因为这些任务是可以独立并发执行的。

案例：一个数据库查询10条数据需要10s，能不能多线程？ 不能，这是个不可分割的任务。

一个操作要查询数据库，一个要调用接口，要读硬盘文件？ 可以，因为任务彼此不干扰。

需求：即需要多线程提升性能，又需要多线程全部完成后才能执行的操作。

List<Task> taskList = new List<Task>();

taskList.Add(Task.Run(()=> this.Coding(“aaa”,”a1”)));

taskList.Add(Task.Run(()=> this.Coding(“aaa”,”a1”)));

Task.WaitAll(taskList.ToArray());

阻塞当前线程(主线程)，直到全部任务结束，主线程被阻塞，所以卡界面

Task.WaitAny(taskList.ToArray()); // 阻塞当前线程，直到任一任务结束，主线程被阻塞，所以卡界面。

卡界面解决：

开启一个子线程，将WaitAny 和 WaitAll 包进去

Task.Run(()=>{

// 将上面两个包进去

})

// 不推荐做法，尽量不要线程套线程，有更好的方法解决；这里全是字线程完成的，不能直接操作界面。

如：TaskFactory taskFactory = new TaskFactory();

taskFactory.ContinueWhenAll(taskList.ToArray(), tArray=>{ // 全部任务完成后，启动一个新的task来完成后续动作

});

taskFactory.ContinueWhenAny(taskList.ToArray(), tArray=>{ // 任一任务完成后，启动一个新的task来完成后续动作

})

continue 的后续线程，又能是新线程，可能是刚完成任务的线程，还可能是同一个线程。不可能是主线程。

线程是不可预测的，所以几个动作是不可预测的。

四个方法学号了，就可以完成90%的多线程业务场景。

**6. 多线程安全**

问题一：循环中启动线程

for(int i=0;i<5;i++){ 加个 k. int k=I;

Task.Run(()=>{

Console.WriteLine(Thread.CurrentThread.ManagedThreadId + “-” + I + “-” + k);

})

}

输出都是5.

原因：循环第一次，动作还没有启动，当i为5时线程才开始。所以都是5了。

解决方法，加个k，其实有5个k。声明周期内k。

注意：多线程去访问同一个集合，一般是没问题的。多线程问题都是出在修改一个对象的时候出现的，如上面的i。

如： 遍历10000 开启多线程，添加如list中，多线程之后，结果小于10000个，就有数据丢失了

原因：多线程操作同一个集合，list是个数组结构，在内存上是连续摆放的，加入同一时刻，去增加一个数据，都是操作同一个内存位置，2个cpu同时发了命令，内存先执行一个再执行一个，就出现覆盖。

**多线程安全问题**：一段代码，单线程执行和多线程执行结果不一致，就表名有线程安全问题。

**解决线程安全问题**

方法一：加锁：能解决线程安全问题

private static readonly object LOCK = new object();

for(var i=0;i<10000;i++){

Task.Run(()=>{

lock(LOCK)

{

list.Add(i);

}

})

}

加锁就是单线程化，lock 就是保证方法块儿任意时刻只有一个线程能进去，其他线程就排队。

Lock 原理：就是个语法糖。 等价于Monitor.Enter(LOCK) …….. Monitor.Exit(LOCK);

锁定一个内存引用地址，所以不能是值类型，也不能是null。

子线程占用引用表示锁状态，被用，当前引用被释放后，其它子线程才能再来占用这个引用。执行。

多线程，子线程异常不会返回给主线程，所以子线程最好加个try catch

**LOCK 相关测试**

情况一：锁的修饰符，private 改成public 并被想要并行运行的方法同时使用

假如我是希望主线程和方法是并发的，如果共用一个锁变量，就会出现相互阻塞

锁不同的变量，才能并发的。所以要private

情况二：方法和锁都不用静态。使用方法实例化

初始化两个对象，调用方法(方法内循环遍历线程)

结果，可以并发。不同的实例里面，都是不同的字段。所以要static.

情况三：object 改成string，方法外启用另一个循环线程

结果，不能并发。锁定的是内存引用，字符串是享元的，堆里面只有一个“静待花开”引用。指向了同一个引用地址。

情况四：封装到泛型类中

相同类型不能并发--- 泛型类，在类型参数相同时，是同一个类

不同类型的能并发 泛型类，在类型参数不同时，是不同类。

情况五：lock(this) 当前实例

不同实例中，this是不同的，初始化能并发

情况六：lock(this) 调用递归，会不会死锁

不会死锁。 只有一个线程，锁谁去呢

**7. await/async**

C#5.0新出现的

写法

public async Task DoSomething(){

//return Task.Run(()=>{ Console.WriteLine(“….”)}); // 原来的

await Task.Run(()=>{});

}

是一个语法糖，不是一个全新的异步多线程使用方式。本身并不会产生新的线程，但是依托于Task而存在，所以程序执行时也是有多线程的

**普通方法**

public void NoReturn() // 没有返回值，在方法里面开启了一个Task线程

{

Task.Run(()=>{

Console.WriteLine(Tread.CurrentThread.ManagedThreadId);

})

}

async 可以随便添加，可以不用await

await 只能出现在task前面，但是方法必须声明async，不能单独出现

**asyn/await**

// 没有返回值

public async Task NoReturn()

{

Task task = Task.Run(()=>{

Console.WriteLine(Tread.CurrentThread.ManagedThreadId);

})

await task;

}

有返回值的

public async Task<long> NoReturn() // 没有返回值

{

long = 0;

await Task.Run(()=>{

long = 1;

Console.WriteLine(Tread.CurrentThread.ManagedThreadId);

})

return long;

}

**总结：**await/async 之后，原本没有返回值的，可以返回Task

原本返回 X类型的，可以返回Task<X>

一般来说，尽量不要void， 因为不能再await

**分析：**没有返回值的，

task.ContinueWith(t=> Console.WriteLine(‘’))// task 完成之后执行的回调动作。回调线程id都有可能

主线程遇到await，直接返回，干自己的事。

可以认为，加了await 就等于将await后面的代码，包装成一个回调，其实回调的线程id有多种可能性。

**总结：**回调理论：可以用同步代码的形式去写异步。

常规编程下，我们是同步方法，一个线程，按顺序执行

为了并发，为了不阻塞线程，会用异步多线程方法，但是执行顺序又是很奇怪的，经常要等待才能顺序执行。

await/async 之后，就可以用同步的方式编写异步编程。

await/async 只是一个语法糖，深入IL，了解原理。

**反编译解析**

IL里面没有await 和async，就是常规代码

首先实例化状态，状态是-1，然后去执行task前面的，启动线程去执行task

判断完成没？没有就把状态重置到0，让线程递归调用下；自己回去了，回去继续自己的使命。

子线程递归回去再执行状态机动作，包括完成自己的后续动作

如果再遇到await ，又把状态重置为-1，然后再继续。

状态机状态切换+递归，就能支持无限层级await。

**3. 爬虫**

是个应用程序，自动提取网页的程序，通过数据筛选、过滤，得到有用的信息，一般是为搜索引擎服务或者作为内容来源。

做资源性网站等必备技能。

特点：Application + WebReqeust + Filter +Data +Threads

作用：

做个内容站—小说/电影/动漫

数据搜索爬虫--- 招标数据爬虫、淘宝数据、招聘信息

竞品分析—抓取竞争对手数据

怎么写爬虫步骤？

下载网页(模拟请求 HttpWebRequest)

②解析网页，筛选数据(正则，SubString+Replace)，第三方工具包HtmlAgilityPack支持Xpath(本质是正则)

③抓取分页数据，经过分析每一页数据获取的路径仅仅只是一个page的值不一样

获取所有页数据就可以得到最大的页数，然后循环每一页

第一步：先找最大页数

第二部：循环最大页数，拼接请求路径，返回数据，解析数据

如何抓取ajax 请求的数据

匹配页面和请求url

第二步：获取请求的数据

第三步：解析数据：根据json格式数据建立实体 通过HashTable

HashTable hashTable = JsonConvert.DeserialiseObject<HashTable>(str);

如何提高爬取速率？多线程

TaskFactory taskFactory = new TaskFactory();

TaskFactory.StartNew(放入方法)

需要限制线程数。

另一种限制线程数量 paralle

**6. 手写ORM框架**

技术栈：泛型+ 反射 + 特性 + ADO.NET + sql server 读写分离配置以及框架 和分布式事务

写代码的一层封装，面向对象操作

把数据库给你搬到程序里面封装一一映射，通过操作类对象，就可完成对数据库的操作，不用关心sql，不用写sql

底层还是基于Ado.Net + sql 语句实现的

**第一节**

从 ado.net 到通用数据库访问层

泛型+ 反射 + ado.net 完成通用主键查询 & 全表查询

特性attribute完成映射

需求一：一个方法满足不同的表的查询。 泛型，最好泛型约束 BaseModel

知识：基于Generic 完成类型通用

基于泛型约束保证类型正确

需求二：根据不同的类型去完成不同的查询，甚至数据绑定

知识点：基于反射完成sql动态拼装

基于反射Relection完成数据动态绑定

使用无参数构造函数约束，直接可以T t = new T();

可空值类型：Int 值有可能是null 的，映射是异常问题。数据库是null时，查询出来是DbNull，不能直接赋值给Nullable

ORM：在通用数据访问基础上，还有各种高阶特性，如映射、缓存、延迟、事务等。

需求三：为区分映射表和其它业务表区别，映射表后加Model 。又比如改了数据库的表&字段，不想改代码

实体与数据库的表&字段，不完全一致

知识点：特性Attribute 提供额外信息，完成映射

需求四：属性映射，拼装和赋值都需要，有点麻烦？

知识点：上下都需要映射一次，select state as status

开发不建议使用存储过程，不好维护，不好调试等等。一般维护人员爱用存储过程

赋值，粘贴，改。多了就是代码重复，一定要花时间去优化，会获得真实的进步

问题：实际项目中值类型和可控类型该怎么选？

数据库设计环节，个人倾向于非null，没有意义的就给默认值，如int 给-1，date给最小值

**第二节**

泛型+ 反射 +ado.net 完成通用数据插入

放sql注入null处理

泛型缓存完成性能提升

委托完成代码重用

问题：自增---特殊字符---sql注入

需求一：一些列不需要插入

知识点：特性Attribute提供额外信息，完成过滤

问题二：sql 注入，在不知不觉中，执行了以外的sql。 ’);select \* from user where 1=1;-- 或truncate table user;等

解决：数据清洗，转码

参数化

ORM(其实就是参数化)

问题三：今天代码在多线程中有线程安全问题吗？ 没有调用的方法都是人家的，我们只是做了个封装。

问题四：泛型缓存，每次都要动态拼装sql，尤其是通过特性去映射，能不能缓存一次，后面重用。

不同的类型—需要缓存一个不同的sql---就推荐用泛型缓存。

知识点：静态初始化：第一次使用某一类型的类，静态字段和构造函数会在方法之前执行。第二次可以重用第一次参数。

泛型类的类型参数T,不同的时候，其实会产生一个全新的类，所以，静态字段，静态构造函数都会重新执行一次。（运行时）

泛型缓存跟字典缓存有什么区别？

泛型缓存是基于内存相关的缓存。

字典缓存：灵活方便，数据的保存以可为准。性能问题---性能主要是数据超过1w(大概)以上有下降。

泛型缓存：缺点，只能跟类型相关 Company 只能存一个，不适合更新。有点：性能好，读取的性能

类型对应的，不需要更新的，用泛型缓存。没有现成安全问题

有更新的，用地点缓存

Hash存储---增删改查性能都差不多，但是最怕数据太多，多了有损耗

**第三节**

泛型+反射+ado.net完成通用数据更新

委托完成代码复用

通过数据验证功能实现

表达式目录树完成批量更新

问题：字段将截断…(字符串长度超出)

数据库有规则，不应该留到数据库校验—还有的时候有业务规则(state 只能是012) –

数据入库前一定得检测，不能相信客户端检测。

一站式通用数据校验：集中校验，规则各部相同。只能一一提供，额外提供一点格式信息？

特性提供额外信息，然后通过特性校验！

**第四节**

泛型+反射+ado.net完成通用数据删除

表达式目录树完成批量删除，扩展orm

需求：希望删除状态等于3的。O/RM是不支持的。State==3 或id>10 and or 添加多种，个数也不确定，该怎么封装？

需要一种共同语言，让上端调用者和封装的类库能够彼此交流

Expression：就是一个很好的沟通语言，表达式目录树

表达式目录树怎么解析成sql语句。表达式目录树其实就是个二叉树数据结构。向获取的是叶节点，只能去一个个访问。

因为不知道节点深度，需要用到递归

visit 解析入口。会识别当前目录树的类型，如果是lambda，会把body再次交给visit递归；如果是其它类型，会调用对应的visit方法。

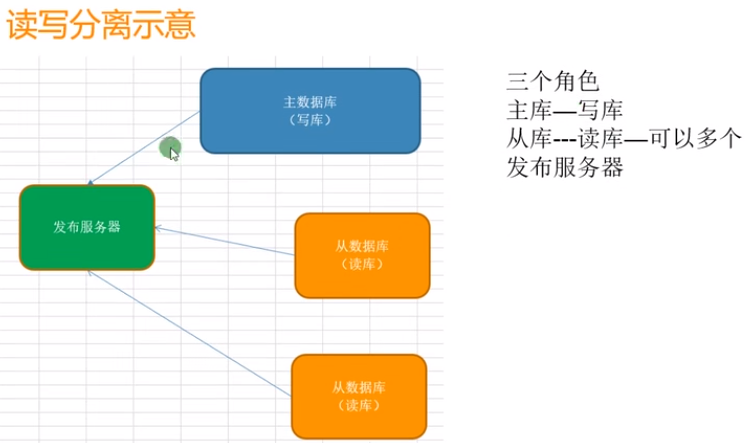
就这样不断的去visit，找到全部的叶节点。而且也找到对应的关系。

**第五节**

Sqlserver 的读写分离配置—负载均衡

框架支持读写分离

分布式事务探讨



二八原则：80%是查询，20%是增删改。

什么是读写分离？

为什么要读写分离？

读写分离怎么操作？

三个角色：主库、从库、发布服务器。

步骤：

①复制->配置分发->指定分发服务器->快照文件夹(配置文件夹共享)->

②发布：复制下的 本地发布->事务发布->选中所有表->当前账号-

③订阅：新建从库，订阅发布。主库修改后，从库的数据也跟着变化。有延时

主数据做增删改操作之后，会把操作日志推送给发布服务器（不影响增删改操作），从库监听发布服务器，有更新，同步从库，能完成同步但是有延迟。

配置发布服务器 --- 注意服务器名称统一

新建发布 --- 注意共享路径

新建订阅 --- 可以订阅多个

主从库代码

配置多个链接，one-write-n-read

sqlConnectionPool and dispatch

访问层提供默认链接并接受指定

三个或者更多的从库，需要从中合适的选择一个库，负载均衡策略(轮询)

链接字符串轮序，随机，权重

主从库在不同的服务器上，配置有高低，不同从库的任务不同(10%,30%,60%)

权重：不同的从库承担不同的任务。首先得配置，读取配置，然后分别是1,3,6

能不能根据数据库的压力来配置？

一定得有个实时获取压力的途径

方法a 可以根据查询的响应时间

方法b 还得途径去获取硬件信息

方法c 实时获取下数据库连接数

主从库同步延迟问题？

测试之后，7s完成了数据同步。

主库发布服务器与从库都是一台服务器，延迟是相对高一些。但是数据同步基本上是按秒算的。

3秒的延迟，怎么办？

适用于能接受延迟的操作(只要保证增删改时数据不错就行)，其实大部分业务都能接受脏数据

不能接受的呢？那你就查主库

还有一招叫缓存/nosql---增删改了同步主库，也同步下nosql，短时间缓存

为什么有了缓存还要分主从库，因为缓存只能拦截部分，还有很多请求

**50 .net core**

特点：跨平台、自托管、开源、高性能

依赖环境：以前开发依赖环境是windows, .net 环境依赖为Nuget，自己配置

1.执行过程

– 基于Kestrel 的控制台应用程序



代理服务器：.net core 使用iis做反向代理与用nginx有区别呢

2.中间件执行过程

①http管道，HttpContent 上下文

②三种写法

写法一：Run

写法二：Use

app.UseHttpsRedirection()

app.UseStaticFiles() //开启调用静态文件的权限

app.UseCookiePolicy();

app.UseAuthentication();

app.UseSession();

app.UseMvc();

写法三：Map & MapThen

③中间件与模块处理程序的区别

3.中间件执行过程

源于配置 Configure Services、ApplicationSettings

4.必学知识

Linq

Mvc

Razor

Rest

Route

DI/Ioc

AOP

Filter&Attributes

Cache

ORM

5.sdk 环境

①环境

Host：托管环境

Runtime：

SDK：

注释：版本必须匹配

②IDE

vs 2015

vs 2017

vs 2019

vs code

6. 常用命令

dotnet new -- 创建

dotnet build -- 编译

dotnet clean -- 清理

dotnet help -- 命令帮助

dotnet publish -- 发布

dotnet run F5 -- 运行

dotnet watch run --

7. 结构

①properties 启动配置

②wwwroot 根目录

③依赖项 其中SDK，为基本环境

④Programe 整个程序的入口

应用程序使用进程内HTTP服务器实现运行。服务器实现侦听HTTP请求，并将它们作为一组组成HttpContext 的请求特性呈现给应用程序，也就是Request和Response

Vs 2019 提供源代码查看，需要在工具里开启。

⑤Startup 配置服务，配置中间件文档，启动mvc，打开web服务

8.

properties 启动配置

9.

properties 启动配置

10. 两种托管模式，进程内，进程外

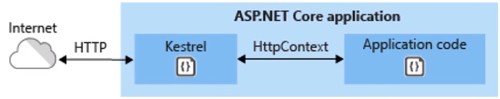
properties 启动配置

**51 .net core dcoker**

1.脚本启动？

细思极恐---控制台程序---WebServer ------Kestrel

网站是怎么诞生的？ 由Http协议（数据格式）---浏览器发起Http请求(一段文本)---发送到服务器（ip:port）---得有个进程监听端口---按照Http协议解析得到有效信息---然后才是由我们的代码处理具体请求---处理完响应结果---按照协议格式拼装字符串---返还给浏览器---浏览器解析下，将内容渲染呈现。



以前：端口监听、协议解析等都是iis服务器做的

2.Kestrel

是个精简高效的HttpServer，以包（dll）形式提供，自身不能单独运行。内部封装了对libuv的调用，作为I/O底层，屏蔽各系统底层实现差异。是跨平台的！

现在：端口监听、协议解析等由Kestrel 处理。

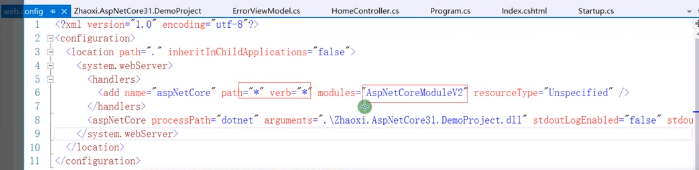
Kestrel虽好，但是还不够用，只是一个简单版的---生产等环境不能直接使用。Kestrel---IIS---Nginx---Docker

3.IIS发布

步骤一：发布后部署

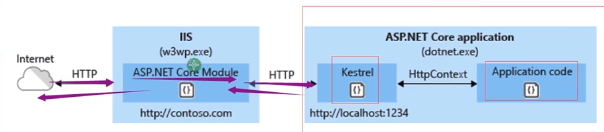
步骤二：安装AspNetCoreModule

托管问题---编译目录页可以部署，缺少(发布时会生成)web.config



**web.config作用**：iis把所有的请求都交给AspNetCoreModule。处理的方式 是下一行，用dotnet 将dll程序跑起来。

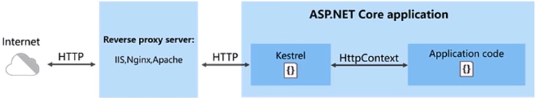
.net core项目：编译目录和发布的文件目录几乎是一样的，就是发布时会多一个web.config



iis 在这里只是充当一个反向代理的角色。解析和响应还是有代码来处理的。Iis做反向代理使用于windows系统，大部分时候用Nginx做反向代理。

4.Nginx + Kestrel

配置Nginx反向代理。注意：kestrel 属于不完整版的webServer，前面一定要Nginx做反向代理的。



为什么使用：反向代理--- nginx ---集群，负载均衡(一套代码，共同承载即集群)



推荐使用nginx 进行反向代理

负载均衡策略：集群---反向代理---负载均衡，现在是标配

5.docker部署

Docker-compose批量处理

**1.从微服务到docker容器技术**



虚拟机---容器诞生

1 IT发展快，项目越来越负载，N多个进程协作

2 摩尔定律---硬件发展飞快

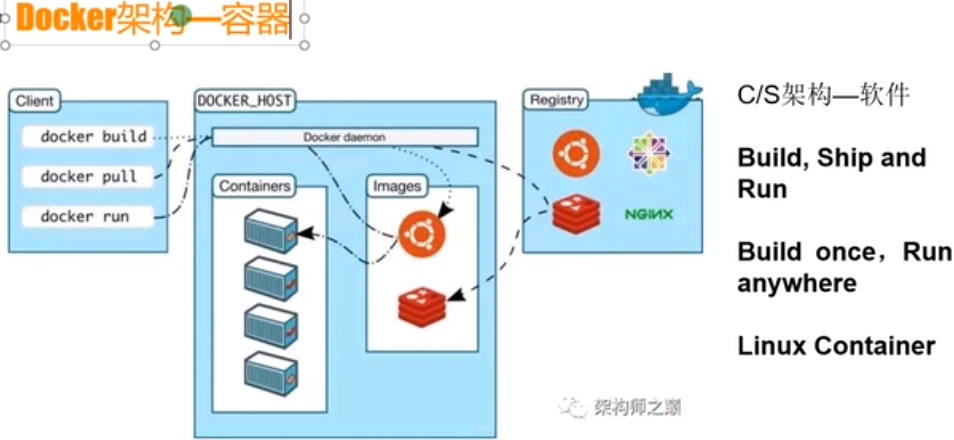
但是多进程不适合放在一起---不同的进程依赖环境不一样---能不能隔离一下？

把大的物理服务器拆分成多个小的麻雀（五脏俱全）---提高利用率/互不干扰。但虚拟机----浪费资源，启动慢



物理机--- 虚拟机 --- 容器技术

**2.多方式构建docker容器实例**



Docker 就是一个应用程序，在服务器上安装docker，跑起来就是一个Docker daemon 进程。还有一个Client客户端

客户端可以执行命令操作，命令发给docker进程

Docker 三个核心概念：容器实例、镜像、registry或叫docker hub

容器实例：独立的运行环境(像虚拟机)，包含代码等，只包含一个进程。容器实例化都是由镜像启动的

镜像：docker的价值优势都在镜像上（或较拍照），可以通过镜像实现多个实例，镜像不仅有代码，还有依赖环境

Registry：远端

注意：所有实例都是由镜像起来的

镜像：docker image 是一个极度精简版的Linux程序运行环境。

自己做的镜像都是基于dockerfile构建的。

步骤：选中.net core web 项目---右键 添加 – 选 Docker支持 --- 选 Linux 后生成Dockerfile 文件。

一步步的搭建Asp.Net Core集群+Nginx支持访问。

3.docker-compose批处理

**51. JWT 权限验证**

特点：跨平台、自托管、开源、高性能

1. API 保护

防泄漏

防攻击

收益话

2. API 保护

**52. 微服务架构**

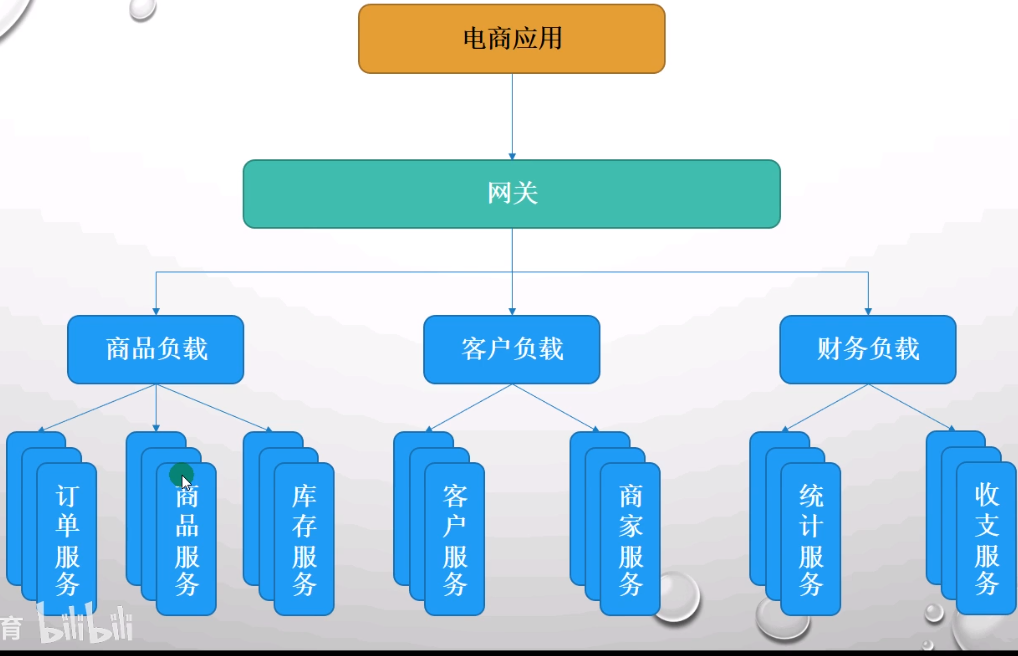
**分布式**：就是把业务拆分为一个又一个的步骤，比如做饭，选菜，掌勺等分开每个部分做自己的那部分业务。

项目随着用户量，设备量增多，并发量上升，已经不足以支撑整个业务系统，这时候可以采用硬件+软件方式来解决这样的问题。

通过提升硬件(服务配置+带宽)，开销较大。分布式就是解决这个问题的。

**集群**：就像搬砖，一个人不够，就多分配给你几个人，你们都是干一样的事情。从广义理解，它其实就是分布式。

1、微服务流程



负载：由consul

网管：由Ocelot

服务：由webapi

2、配置负载consul

consul 下载之后不需要搭建，直接在文件目录下，启动cmd，执行consul.exe agent -dev 即可启动consul 的服务

可通过localhost:8500 回车，通过默认端口号访问consul 服务

程序 nuget 安装 Consul 服务包，扩展中间件ConsulManager，注册服务，使用服务后，打开后台程序

dotnet StudyDemo4\_UserService.dll --urls="http://\*:19611" --ip="127.0.0.1" --port=19611 (关闭用ctrl+c)

程序关闭后，存在一个问题，在consul 服务中依然存在。需要引入健康检查，后台程序中添加Heart 控制器，

引入swagger

nuget 安装 Swashbuckle.AspNetCore

注册服务services.AddSwaggerGen(r => r.SwaggerDoc("v1", new OpenApiInfo{

Title = "user api",

Version = "v1"

}));

使用服务app.UseSwagger();

app.UseSwaggerUI(r => { r.SwaggerEndpoint("/swagger/v1/swagger.json", "user api"); });

查看接口文档：http://localhost:19611/swagger/index.html

3、配置网管Ocelot

nuget 安装Ocelot，有个坑，版本要15.0.6，最新版的获取不到服务

dotnet StudyDemo4\_OcelotGetWay.dll --urls="http://\*:5916" --ip="127.0.0.1" --port=5916 //网管启动服务

通过微服务的网关访问微服务，现在可通过网关直接访问服务了,还没用consul

nuget 安装Ocelot.Provider.Consul，版本也用15.0.6

测试：访问网关---> consul ---->服务 成功

微服务架构突破了开发语言的限制

4、网关中调用多项目swagger

①启动consul服务：consul.exe agent -dev

②启动服务项目：dotnet StudyDemo4\_UserService.dll --urls="http://\*:19611" --ip="127.0.0.1" --port=19611 //启动两个服务

③启动网关服务：dotnet StudyDemo4\_OcelotGetWay.dll --urls="http://\*:5916" --ip="127.0.0.1" --port=5916

添加product服务项目，在网关中添加swagger显示

Nuget 安装consul，然后安装Swashbuckle.AspNetCore

④dotnet StudyDemo4\_ProductService.dll --urls="http://\*:9166" --ip="127.0.0.1" --port=9166

给网关请求路由地址加上/user 或/product，出现问题，请求swagger 异常。需配置swagger路由json文件的路由规则

nuget给网关项目安装Swashbuckle.AspNetCore

为了在网关中拿到两个服务的swagger，配置两个服务的路由，在网关的路由配置中不需要/swagger 前缀

5、限制1分钟内超过5次访问

通过过滤器，添加后放到微服务上有问题。过滤器是每个服务内的过滤。

解决思路，通过cookie解决或通过redis，或通过网关解决。

**53. NPOI**

**名词介绍**

整个excel： 工作簿

Sheet 页：页签，一个工作簿对应多个sheet 页

表格：对应一个Sheet

行、列、单元格

**常规操作**

导出一个Excel：其实就是要生成一个Excel文件，Excel文件对应的文件流

导入一个Excel：读取一个文件，读取文件流，需要从文件流中读取我们需要的各种数据。解析excel数据

HSSFWorkbook book = new HSSFWorkbook(); // 创建工作簿

ISheet sheet = book.CreateSheet(“Sheet1”); // 创建页签

IRow head = sheet.CreateRow(0); // 创建行，从0 开始

ICell cell = head.CreateCell(0); // 创建单元格，从0 开始，即第一行第一列

// 文件流需要释放，所以放到using中

using(FileStream file = new FileStream(“D:aaa.xls”, FileMode.Create)) // 创建excel 文件

{

book.Write(file); // 将数据写入文件

}

**导出excel**

【1】数据写到工作簿中的哪个sheet页中

【2】生成的excel—考虑表头放在哪个位置

【3】直接集合中的某个对象来直接生成—如果对象是一个实体，实体中有多少个属性就表示有多少列。

目的：做到业务系统不需要考虑其他，只需要按照规则来给定数据即可，就可以生成excel出来。

扩展实体类：sheet页名称、标题所在行 index、每个sheet的数据 如：List<Student>

列标题：可以通过添加特性来 设计标题文本 ([Title(Title=”学生名称”)])

导出场景：

方式一：写入Response 二进制流

方式二：调用框架的File 方法

方式三：扩展IActionResult 方法 ExcelResult 方法

**导入excel**

本质：目的是把excel 文件提交到服务器，然后把excel 文件中的数据信息读取出来，再处理数据信息

Excel 文件的解析

1.Excel 文件 --- 文件流 fileStream MemoryStream byte[] ---IWorkBook,就可以使用NPOI来解析内部数据

2.就是解析一个IWorkbook --- 读取内部数据

导入场景：

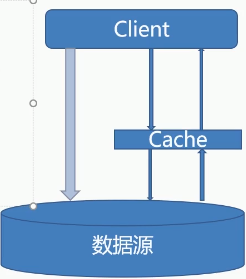
form 表单导入

js 导入，可以做到无刷新导入，可以使用组件

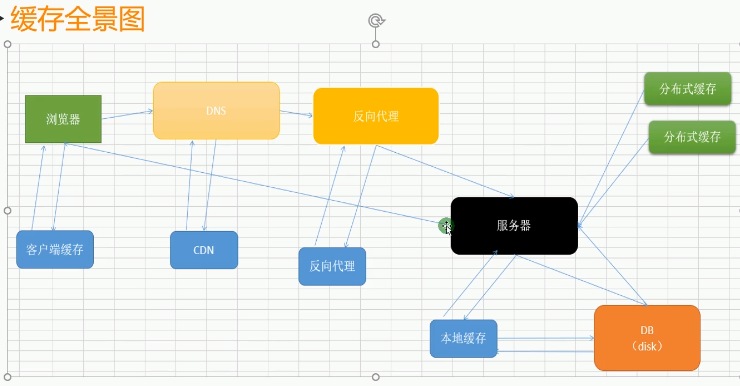
**54. 多级缓存**

系统性优化的第一步就是缓存。缓存是一种技术效果，把数据结果存在某个介质中，下次直接重用。

大数据，高并发



反向代理是服务端的一个保护、屏蔽



传统说的服务器压力大，比如加载一个页面，可能会发送上百个请求。

在命令行中启动项目： dotnet Web.dll --urls=http://\*:5177 --port=5177

1.**客户端缓存策略**

客户端缓存 直接缓存在客户端本地；分两种

MemoryCache：浏览器内存缓存

DiskCahe：硬盘缓存-临时文件夹

好处：缩短网络路径，加快响应数据；拦截请求，降低服务器压力；让服务器减少对请求的响应。

**浏览器首次请求**

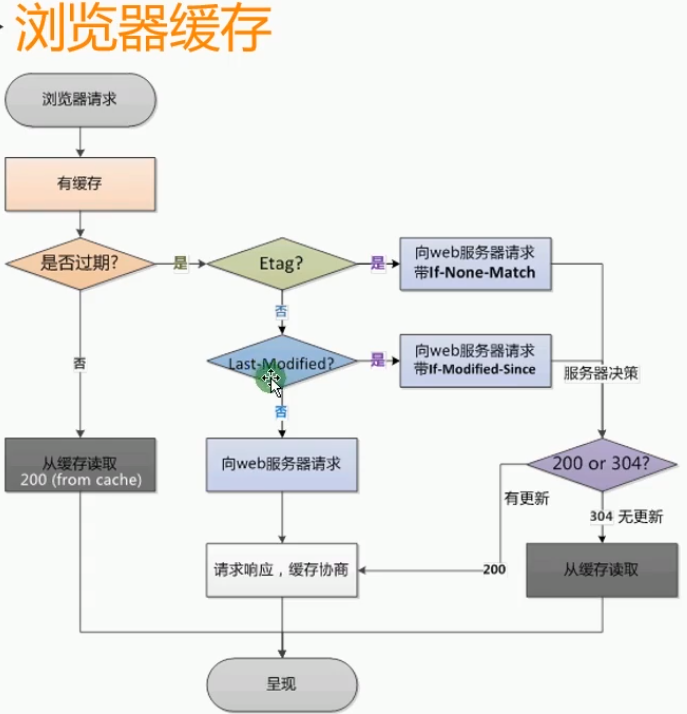


缓存—存时

Http协议(请求/响应式模型)，传输的是文本---服务器解析文本，进行增删改查---响应有个ResponseHeader---告诉浏览器如何使用(压缩/缓存)---缓存控制就是Header---浏览器就缓存数据了(key-value 方式存储)

再次请求时拿着url做key，内存中有则获取数据，缓存效果就实现了

**第二次请求**



Etag是个MD5摘要---如果缓存过期，发个请求带上摘要且比对摘要，200(有更新)/304(无更新)

Ctrl + F5 强制刷新，所有请求重新获取

F5 刷新页面，可看缓存

Cache-Control + Etag(通过响应头设置缓存时间)

**常规缓存**：直接缓存，有效期内可以重用。有缓存，设置有效期，通常是很长如24小时。超过24小时，E-Tag304 还是缓存。有改动 200 重新获取

如果修改后，希望立即生效？ 一行css变灰——有的网站有延迟—缓存---强刷

特使情况，想立即生效？ 只有一招：换url，加个版本号。

**动态页面缓存**

添加缓存协商方式：

①[ResponseCache(Duration=60)] // 在函数上加上这个特性，产生60秒缓存。

②base.HttpContext.Response.Headers[“Cache-Control”]=”public,max-age=60”;

**客户端缓存总结**：缓存在客户端本地，浏览器内存中或本地硬盘

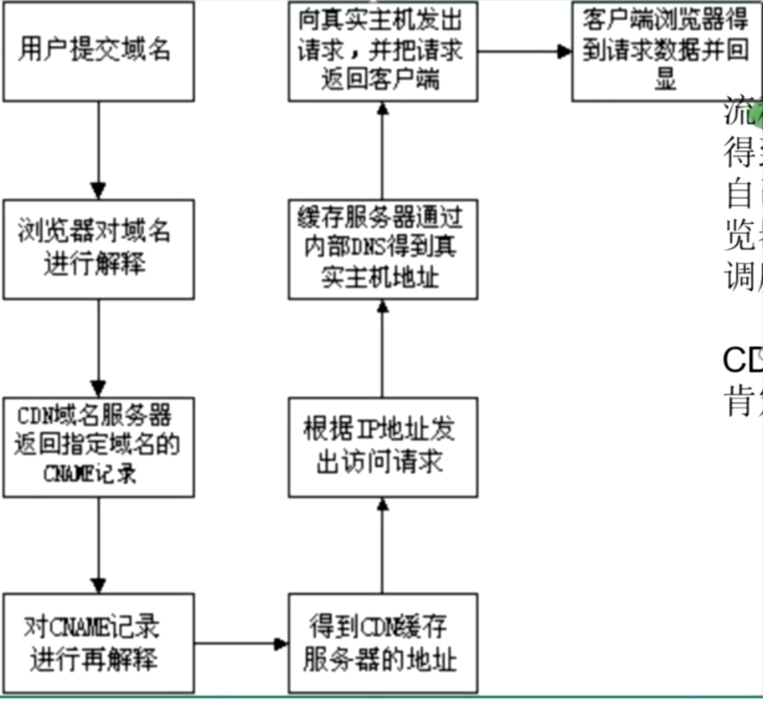
好处：快速获取，加快响应速度；减少请求，降低服务器压力

生效范围：仅对当前用户，当前浏览器生效

实现方式：通过制定HttpHeader 完成缓存的协商

更新策略：时效 + Etag + Version 版本号

**2.CDN缓存**



用户提交域名---浏览器对域名进行解释---CND域名服务器返回制定域名的CName记录---对CNanme记录进行解释---

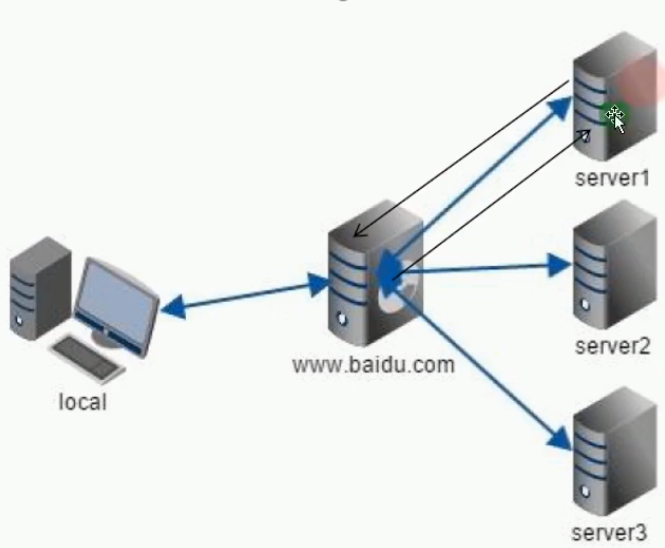
得到CND缓存服务器的地址---根据IP地址发出访问请求---缓存服务器通过内部DNS得到真实主机地址---向真实主机发出请求，并把请求返回客户端---客户端浏览器得到请求数据并回显

CDN怎么知道缓存哪些东西？也是根据httpHeader

在哪里缓存？DNS离用户最近的地方，缓存就效果好。对一批用户生效的，一般用来存储体积大的东西

CND缓存其实是代理缓存。（第三方的）正向代理

**3.nginx反向代理缓存**

代码发布在server3上

**配置代理**

Nginx使用：步骤一 监听(listen 5177)

步骤二 转发请求给地址(location / { proxy\_pass <http://CacheDemo> })

步骤三 设置转发地址 upstream CacheDemo{ server localhost:5177;} // 配置集群，直接添加就可以

start nginx 开启nginx

nginx -s reload 优雅重启

步骤四 执行缓存路径，当前盘符，必须写在最上面

proxy\_cache\_path /Proejct/nginx-1.1.7.8/data levels=1:2 key\_zone=web\_cache:50m inactive=10m max\_size=1g;

第五步 缓存策略

Location /First/{

proxy\_story off;

proxy\_redirect off;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header Host $http\_host;

proxy\_pass <http://CacheDemo/First/>;

proxy\_cache web\_cache;

proxy\_cache\_valid 200 304 2m;

proxy\_cache\_key $scheme$proxy\_host$request\_uri; #url 地址做key

}

中间的是nginx，没有代码，只是代发请求。

配置conf/nginx.conf 文件，启用start nginx

配置缓存分两步：指定缓存路径；缓存策略；

分析：就是降低了服务器压力，直接返回。url 改变，重新请求

80% 的请求集中在20% 的页面上。

**4.本地缓存与分布式缓存 nginx + redis**

两者都是服务器缓存，大部分时候都是为了缓存数据库，降低数据库压力。

.net core 是一个组件化框架。默认很多东西没有吧默认实现引入进来。开发是基于IOC的。把所有的东西都丢到里面，需要的时候上里面去获取。

**内置缓存MemoryCache 的使用**

使用步骤：

①nuget 引入Memory；

②注策服务 service.AddMemoryCache(options=>{ options.Clock=newLocalClock(); })；clock 指定一下过期时间

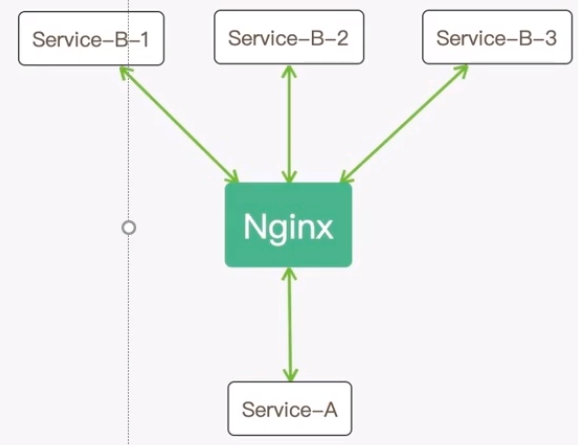
③依赖注入实例，调用就可以常规使用了

将数据放到MemoryCache即进程里面。重启后，进程关闭后就释放了。即为内存（进程）缓存。内存读写块、数据可丢失、体积有限。

问题：空间问题：源于当前进程内存的缓存问题

集群问题：所以会有分布式缓存

**集群**



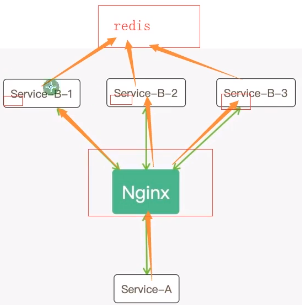
一套代码，实现开了3个端口。三个实例的缓存是在各自的进程(内存)中的，不能共享。

案例：使用nginx 配置三个端口，即创建一个访问集群。启动nginxj进程。刷新页面，三个时间挨个显示。

即ngixn分发请求，默认轮询策略，3个进程自己有缓存。

解决方法：数据共享，使用redis缓存共享

**Redis 数据共享**



下载redis ，启用redis。

①nuget 引入Redis；

②注策服务 services.AddDistributedRedisCache(options=>{

options.Configuration = “127.0.0.1:6379”;

options.InstanceName = “RedisDistributeCache”; //key 的名称

})

③启动redis-配置连接

④依赖注入实例，调用就可以常规使用了

**总结：**不管是redis 还是memory cache 都是存放数据的。key/value

本地缓存：Memory cache 是进程缓存，各个模块进程内的缓存，进程关闭，缓存则消失。

分布式缓存：Redis 缓存是一个单独共用的进程，用以存放多个服务进程的缓存数据

**缓存应用思维风暴**

Redis缓存，项目开发中，能具体解决哪些典型应用

情况一：数据缓存---菜单/权限/类别/数据字典--- 相对稳定即时性要求低的

情况二：缓冲层。文章点赞/阅读量，变化快，数据库压力太大---Redis缓存，读写快，开启持久化---100/1000次才写一次数据库(缓冲层，起到请求拦截，保护数据库)

情况三：密码错误24小时最多5次尝。账号做key+次数做value放到redis里，有效期24小时，验证次数---一天投票

情况四：秒杀，充分利用redis单线程模型。东西少并发高数据库扛不住。思路：先走Redis，(如redis库存10个)有库存才能到数据库，否则直接失败。最起码数据库保住了。 ---没有线程安全问题(因为redis 是单线程)。

情况五：分布式锁---基于Redis。多个集群问题，加锁，不用了，删掉。超时机制，守护线程。

情况六：分布式Session。集群中session不一致，创建一个分布式session

以上只是基于key-value 的使用

并发高：同一时间请求多。

**5. 缓存分类**

基于Http协议缓存：客户端缓存，cdn 缓存

服务端响应缓存：反向代理，中间件，

数据获取缓存：本地缓存，分布式缓存

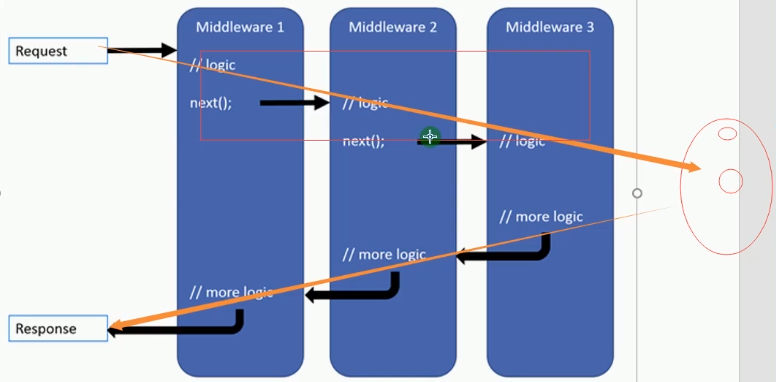
**基于http协议缓存**

方式一：特性 ResponseCache(Duration=60) 会在响应头中加上 Cache-Contro:public,max-age=60; (.net core 内置特性)

方式二：base.HttpContext.Response[“Cache-Control”]=”public,max-age=60”;

方式三：扩展一个Filter特性

方式四：中间件中，请求开始前或之后，会穿过一系列的中间件(管道)，也可以在其中添加缓存设置



**响应式缓存之反向代理缓存**

反向代理基于nginx的 启用 start nginx

把生成的页面缓存起来，分布式请求时，只要url不变，一直访问一个端口，叫响应式缓存。

**响应式缓存之中间件缓存**

第一步：service.AddResponseCaching(); 注入服务

第二部：app.UseResponseCaching(); 使用中间件缓存

第三步：配合特性 ResponseCache(Duration=60)

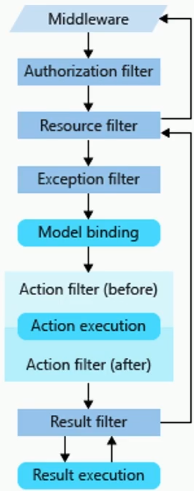
中间件响应缓存，服务器缓存，asp.net core 内部的，可跨浏览器，用户的缓存。

Ngxin 是程序外部的缓存，中间件缓存是程序内部的缓存。缓存的是html。

**响应式缓存之资源filter**

进入mvc后，资源filter缓存。缓存的是IActionResult

Filter是针对Action 层面的，要不前，要不后



**服务器缓存**

1.本地缓存—Redis 分布式缓存

2.Aotufac-AOP 模式提供

Aotufac-AOP使用

第一步：引入Autofac.Extensions.DependencyInjection ,asp.net core 默认ioc容器没有办法做扩展，需要引入这个包。

第二部：引入 Autofac.Extras.DynamicProxy 这个是做ioc的。

第三步：在CreateHostBuilder 中 .UseServiceProviderFactory(new AutofacServiceProviderFactory()) 将IOc换成引入的

第四部： 配置AOP缓存容器



优点：不改变代码。可控

**缓存两件事**

客户端缓存、CDN缓存、反向代理、本地/分布式缓存 都是为了保护数据库(降低，减少数据库压力)。大型项目的思路。

缩短网络路径，加快响应速度，减少请求，降低服务器压力

缓存用的好，1万次请求，可能只有1次到数据库。反过来，我们的数据库的处理能力，只有真实请求的万份之一。如果没有缓存的保护，高并发时，数据库是秒蹦。

**缓存采坑 之缓存预热**

项目刚上线，数据库里面挂掉？

因为没有缓存保护，请求全部到数据库，抗不住---所以挂了。

解决：缓存预热，写个项目把数据导入缓存。准备好了，再发布出去。

案例：携程被删库---12小时内就恢复了数据库，系统起不来，一开放又挂了，72小时恢复。

**缓存采坑 之缓存穿透**

友商测试，id=-1

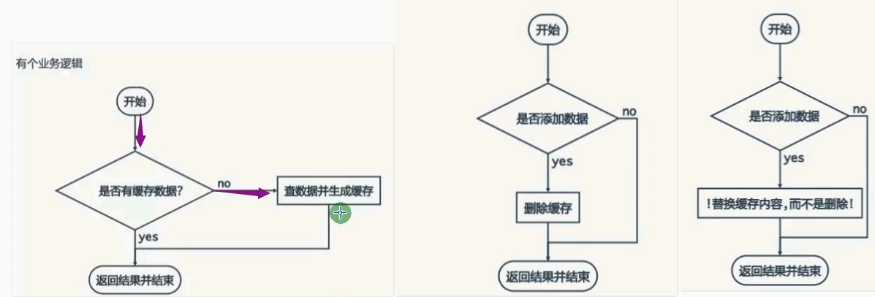
请求不存在的数据，永远没有缓存，失去保护的数据库也会抗不住(绕过缓存)

缓存key-null—短时间。或过滤key，但比较麻烦。

**缓存采坑 之缓存击穿**

某个热点数据过期，数据库瞬间垮掉？

缓存丢失保护的一瞬间，大量请求涌入，数据库挂掉



避免缓存失效(增删改数据会更新缓存而不是删除缓存)

热点数据不过期，或有效期长，避免缓存被击穿

**缓存采坑 之缓存雪崩**

数据批量过期，预热时缓存的有效期都设置的同一时间，缓存全没了数据库蹦了。

解决方案：过期时间分开点。

**终极解决方案？**

有没有办法一站式解决上面全部问题？

不管是那种坑，最终都是数据库抗不住，是因为并发高，终极解决办法就是避免并发----分布式锁，互斥访问数据库。

只是一个锁，实时很困难，真的太过于限制并发，不好。

分布式缓存中没有，申请锁，不一定是在redis进程中，进行请求数据库，那数据库就是安全的。

**6.源码解读与进阶之路**

理解流程：5类Filter顺序，异常处理生效范围

扩展定时：不仅是面向收缩引擎编程，还知其所以然

掌握心法：观摩他人智慧，提升自身技术



技巧：

自上而下，先整体再细节

望文生义，该猜猜该过过

理解程序设计，抓住核心套路

**Asp.net core 全貌**

Vs2022 Preview2 + .NET6 Preview6

Program

2.Startup

3.M-V-C

4.Appsettings

命令行启动流程---请求来了，运行流程

**源码编译**

启动startv.cmd

②查看启动流程：服务器启动，Kestrel服务器开始监听服务器的请求

③查看响应流程：http请求的处理过程

**main 方法中三步**

创建builder(创造者模式)并配置，包括环境参数、contentroot、配置IOC、配置loggerfactory、IOC容器/IIS、Kestrel等(只是配置)。只在做一系列的基础配置。

②构建builder.Build()，build()之前，检测哪个类上面有某个特性，然后实例化它，然后调用它的Configure方法。这里就是留了一个扩展点! 即无侵入式扩展。

扩展之HostingStartup

项目启动时，额外处理逻辑，支持无侵入---SkyAPM 1 项目内 2 项目外需要配置

扩展点：配置中间件。

希望有点额外的逻辑，请求层，链路追踪。请来龙去迈

无侵入式扩展，只需要在launchSetting.json 配置一下，可以把你的全部源代码给送出去(可怕)，开个后门。

构建host = builder.Build(); 构建Host：先HostStartup，实例化IOC容器，各种默认注册，注册了前面的服务。然后把StartUp类找出来。支持IOC容器扩展。实例化了一个Host(有个IOC容器)，就是完成IOC注册。

③把host run 起来：去Startup.ConfiguService 完成IOC容器注册

再完成Configure组装中间件(先StartupFilter) 。

得到管道模型HttpContextFactory---

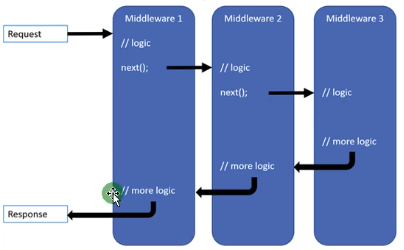
把三者打包给Kestrel，启动监听绑定动作。请求来了，循环监听服务器，收到请求就丢给线程池，HttpContextFactory去解析成HttpContext,交给管道模型，就各种中间件处理---MVC---执行过去了。

这三个过程就是完整的过程。容器也是可以替换的。

Builder(建造者)模式定义：指将一个复杂对象的构造与它的表示分离，使同样的构造过程可以创建不同的表示。

它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

**7.中间件**



管道模型：启动环节，各种初始化，跟服务器打交道。

响应环节，走mvc流程，处理请求给出响应。

链接点？管道模型，何谓管道模型？就是个委托。

管道在Configure 中配置。

最原生Use中间件，委托嵌套，一共有三层。

最开始只是执行Use，然后是next，最后是context。

Asp.net core 流程走一遍：梳理事件点：IOC注册，准备builder，configure, build, rund

委托里面是委托，请求来了，就能一层层的调用，就像管道一样。

管道就是委托！动态组装---随意指定环节轻松扩展

Use、Run、UseWhen、Map、MapWhen 这些都是对基础Use方法的封装，终结就是不调用next, 后3个都是开了独立的Branch，然后执行。

**UseMiddleware<T>普通类**

源码看看调用过程

1.没有借口约束，所以各种检查校验

2.对象初始化靠反射，但是构造参数支持依赖注入

3.构造函数支持传入参数，第一个参数是next

4.方法名字Invoke---InvokeAsync

普通中间件，最终组装了一个委托嵌套，委托包含2个元素(method实例方法，target对象)---常驻内存的

普通类是启动时实例化，嵌套委托常驻内存，初始化支持传参数。

接口实现类是响应请求的时候才实例化，用完支持主动释放，默认实例化不能传参数。

网站开发中，Middleware适合干什么？

1.全部请求都要经过--- 日志/性能监控/跨域/压缩/前端缓存/加密解密

2.包裹在外层做请求拦截---黑白名单/鉴权授权/反爬虫/限流/链路追踪

3.应对特殊条件的请求---robot/rss/防盗链

**总结：**启动环节，各种初始化，跟Kestrel服务器打交道

响应环节，走MVC流程，处理请求给出响应

链接点：Http Pipeline，委托嵌套，全部请求都经过

目标：扩展一套静态化高并发架构。

任意请求都要经历中间件的处理，而且合一随意扩展，那就能够对任意请求做扩展处理。

**8.高并发**

提到高并发，都会说到集群负载均衡

快速搭建Nginx + .NET6集群负载均衡

实例集群

Nginx配置

启动Nginx

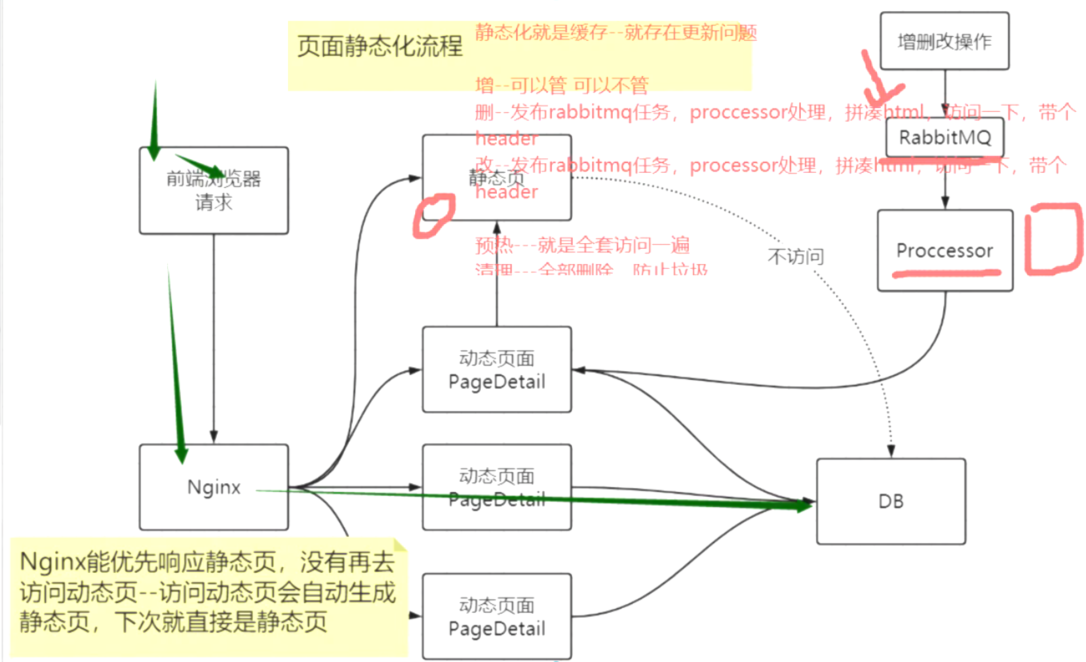
支持多种负载均衡策略：轮询、ip、least\_conn、fair、url\_hash

Start nginx

Nginx -s reload

**如何降低数据库压力？** 80% 压力都在数据库

索引、读写分离、分库分表分区、sql写好点。 治标不治本，最好不用数据库。理解静态页



**理解静态化**

一言以蔽之，一入一出一更新

一入：生成静态页

一出：静态页提供响应

一更新：数据变化了

好处：降低服务器压力---降低数据库压力---加快响应速度---应对高并发

**中间件拦截**

怎么生成静态页，如京东商品详情页，格式一样就是数据不同。

以前，自己实现的模板引擎+发布系统---类似于Vue—html里面去写标签和变量，发布系统就是一条数据替换，生成一个html

现在：在管道模型中扩展个中间件，就是读取响应，把html存起来。

访问动态页面时，顺便生成了一波静态页，保存在硬盘---靠的就是middleware扩展，读取响应的stream。

**Nginx 动静结合流程(响应 静态页)**

如果静态文件已生成，优先使用静态文件

如果没有静态文件，则访问动态网页并生成

Nginx 当web服务器---直接访问静态文件。

**Nginx 动静分离配置**

静态—响应

动态--- 响应 + 生成，下次就静态

与Nginx 反向代理缓存配置比较：

更新问题：反向代理缓存，是由配置文件管理的。一旦生成，除非手动删除或过期

静态化：这个是可控的，直接访问动态页就可以更新。

**9.Options**

1.新建Option 实体

2.ConfigureService绑定

3.IOC注入到使用的地方

What

options就是读取配置文件信息的，就是为了把配置文件信息读取统一下，不要到处折腾

why

集中配置数据获取，避免依赖配置文件格式

配置信息来源多样化---appsettings.json xml memory 远程Apollo

格式层次可变

缓存数据的要求，数据更新的要求

When

封装组件，需要初始化信息都可以用

比如StaticPageMiddleware

How

配置花样繁多

使用花样繁多

读取源码

Config 完成iOC注入，多种注册方式其实没啥区别

IOptions---IOptionsMonitor---IOptionsSnapshot存储应该有区别

理解泛型问题：项目里会配置N多个Option,IOC咋注册？实际上看到的是Configure---IOption<T>，T类型不一样，实例会是不同的类。

只支持一个Value访问，所以没有name。缓存的典型应用，注册的是单例，所以IOtions<EmailOption>全局只有一个，所以这个\_value 就是缓存

直接单例，读取的就是缓存，不支持数据变化，性能高，只能读默认名字

Monitory 只读一次，写入缓存，但是支持数据修改，靠的是监听文件更新（onchange）数据，实时变更

作用域注册，一次请求内数据是缓存不变的，但是不同请求是每次都会重新第一次数据

**55. Nginx配置**

Nginx是lgor Sysoev为俄罗斯访问量第二的rambler.ru站点设计开发的。从2004年发布至今，凭借开源的力量，已经接近成熟与完善。

Nginx功能丰富，可作为HTTP服务器，也可作为反向代理服务器，邮件服务器。支持FastCGI、SSL、Virtual Host、URL Rewrite、Gzip等功能。并且支持很多第三方的模块扩展。

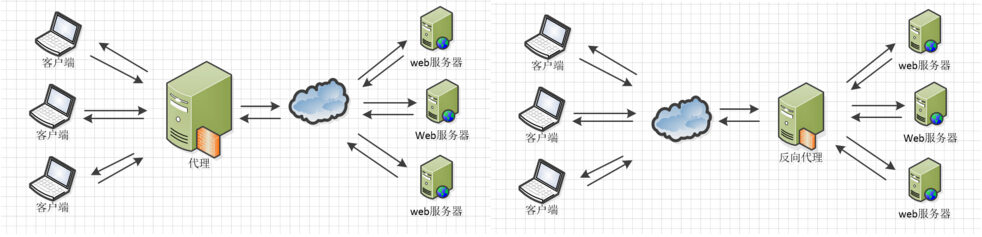
Nginx的稳定性、功能集、示例配置文件和低系统资源的消耗让他后来居上，在全球活跃的网站中有12.18%的使用比率，大约为2220万个网站。

牛逼吹的差不多啦，如果你还不过瘾，你可以百度百科或者一些书上找到这样的夸耀，比比皆是。

Nginx常用功能

1、Http代理，反向代理：作为web服务器最常用的功能之一，尤其是反向代理。

这里我给来2张图，对正向代理与反响代理做个诠释，具体细节，大家可以翻阅下资料。



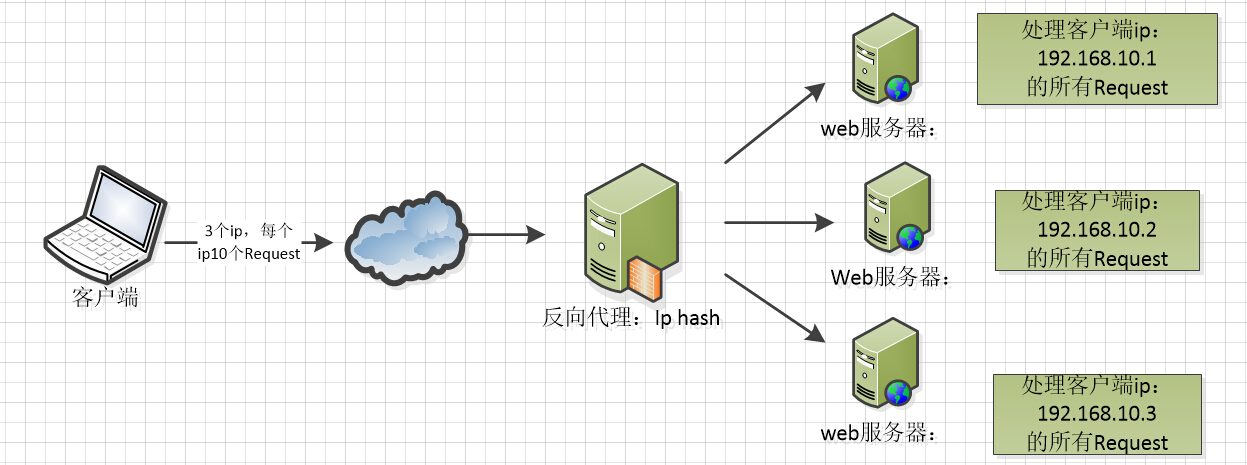
Nginx在做反向代理时，提供性能稳定，并且能够提供配置灵活的转发功能。Nginx可以根据不同的正则匹配，采取不同的转发策略，比如图片文件结尾的走文件服务器，动态页面走web服务器，只要你正则写的没问题，又有相对应的服务器解决方案，你就可以随心所欲的玩。并且Nginx对返回结果进行错误页跳转，异常判断等。如果被分发的服务器存在异常，他可以将请求重新转发给另外一台服务器，然后自动去除异常服务器。

2、负载均衡

Nginx提供的负载均衡策略有2种：内置策略和扩展策略。内置策略为轮询，加权轮询，Ip hash。扩展策略，就天马行空，只有你想不到的没有他做不到的啦，你可以参照所有的负载均衡算法，给他一一找出来做下实现。

上3个图，理解这三种负载均衡算法的实现



Ip hash算法，对客户端请求的ip进行hash操作，然后根据hash结果将同一个客户端ip的请求分发给同一台服务器进行处理，可以解决session不共享的问题。 

3、web缓存

Nginx可以对不同的文件做不同的缓存处理，配置灵活，并且支持FastCGI\_Cache，主要用于对FastCGI的动态程序进行缓存。配合着第三方的ngx\_cache\_purge，对制定的URL缓存内容可以的进行增删管理。

4、Nginx相关地址

源码：https://trac.nginx.org/nginx/browser

官网：http://www.nginx.org/

Nginx配置文件结构

如果你下载好啦，你的安装文件，不妨打开conf文件夹的nginx.conf文件，Nginx服务器的基础配置，默认的配置也存放在此。

在nginx.conf的注释符号位#

nginx文件的结构，这个对刚入门的同学，可以多看两眼。

**nginx文件结构**

... #全局块

events { #events块

...

}

http #http块

{

... #http全局块

server #server块

{

... #server全局块

location [PATTERN] #location块

{

...

}

location [PATTERN]

{

...

}

}

server

{

...

}

... #http全局块

}

1、全局块：配置影响nginx全局的指令。一般有运行nginx服务器的用户组，nginx进程pid存放路径，日志存放路径，配置文件引入，允许生成worker process数等。

2、events块：配置影响nginx服务器或与用户的网络连接。有每个进程的最大连接数，选取哪种事件驱动模型处理连接请求，是否允许同时接受多个网路连接，开启多个网络连接序列化等。

3、http块：可以嵌套多个server，配置代理，缓存，日志定义等绝大多数功能和第三方模块的配置。如文件引入，mime-type定义，日志自定义，是否使用sendfile传输文件，连接超时时间，单连接请求数等。

4、server块：配置虚拟主机的相关参数，一个http中可以有多个server。

5、location块：配置请求的路由，以及各种页面的处理情况。

下面给大家上一个配置文件，作为理解，同时也配入我搭建的一台测试机中，给大家示例。

########### 每个指令必须有分号结束。#################

#user administrator administrators; #配置用户或者组，默认为nobody nobody。

#worker\_processes 2; #允许生成的进程数，默认为1

#pid /nginx/pid/nginx.pid; #指定nginx进程运行文件存放地址

error\_log log/error.log debug; #制定日志路径，级别。这个设置可以放入全局块，http块，server块，级别以此为：debug|info|notice|warn|error|crit|alert|emerg

events {

accept\_mutex on; #设置网路连接序列化，防止惊群现象发生，默认为on

multi\_accept on; #设置一个进程是否同时接受多个网络连接，默认为off

#use epoll; #事件驱动模型，select|poll|kqueue|epoll|resig|/dev/poll|eventport

worker\_connections 1024; #最大连接数，默认为512

}

http {

include mime.types; #文件扩展名与文件类型映射表

default\_type application/octet-stream; #默认文件类型，默认为text/plain

#access\_log off; #取消服务日志

log\_format myFormat '$remote\_addr–$remote\_user [$time\_local] $request $status $body\_bytes\_sent $http\_referer $http\_user\_agent $http\_x\_forwarded\_for'; #自定义格式

access\_log log/access.log myFormat; #combined为日志格式的默认值

sendfile on; #允许sendfile方式传输文件，默认为off，可以在http块，server块，location块。

sendfile\_max\_chunk 100k; #每个进程每次调用传输数量不能大于设定的值，默认为0，即不设上限。

keepalive\_timeout 65; #连接超时时间，默认为75s，可以在http，server，location块。

upstream mysvr {

server 127.0.0.1:7878;

server 192.168.10.121:3333 backup; #热备

}

error\_page 404 https://www.baidu.com; #错误页

server {

keepalive\_requests 120; #单连接请求上限次数。

listen 4545; #监听端口

server\_name 127.0.0.1; #监听地址

location ~\*^.+$ { #请求的url过滤，正则匹配，~为区分大小写，~\*为不区分大小写。

#root path; #根目录

#index vv.txt; #设置默认页

proxy\_pass http://mysvr; #请求转向mysvr 定义的服务器列表

deny 127.0.0.1; #拒绝的ip

allow 172.18.5.54; #允许的ip

}

}

}

[复制代码](javascript:void(0);)

上面是nginx的基本配置，需要注意的有以下几点：

1、1.$remote\_addr 与$http\_x\_forwarded\_for 用以记录客户端的ip地址； 2.$remote\_user ：用来记录客户端用户名称； 3.$time\_local ： 用来记录访问时间与时区；4.$request ： 用来记录请求的url与http协议；

  5.$status ： 用来记录请求状态；成功是200， 6.$body\_bytes\_s ent ：记录发送给客户端文件主体内容大小；7.$http\_referer ：用来记录从那个页面链接访问过来的； 8.$http\_user\_agent ：记录客户端浏览器的相关信息；

2、惊群现象：一个网路连接到来，多个睡眠的进程被同事叫醒，但只有一个进程能获得链接，这样会影响系统性能。

3、每个指令必须有分号结束。