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.异常处理机制**

try…catch…finally

异常类：Exception

**2.抽象类**

abstract

特性：不能实例化

可以包含抽象方法和抽象访问器

不能用 seald 修饰符修饰抽象类，因为这两个修饰符的含义是相反的。

采用sealed修饰符的类无法继承，而abstract修饰符要求对类进行继承

从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实际实现

应用场景：一些方法并且想让他们中的一些有默认实现

**1. 进阶**

1.各种方法

**静态方法**

特点：一旦创建直到应用程序结束才会结束（声明周期）；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

问题：创建多了，占用内存大。不是必要不要创建。

调用：类.方法。不用实例化

**构造方法**

用处：初始化对象，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

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

**虚方法 virtual 子类override 重写方法**

作用：允许子类/派生类进行重写，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，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：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**

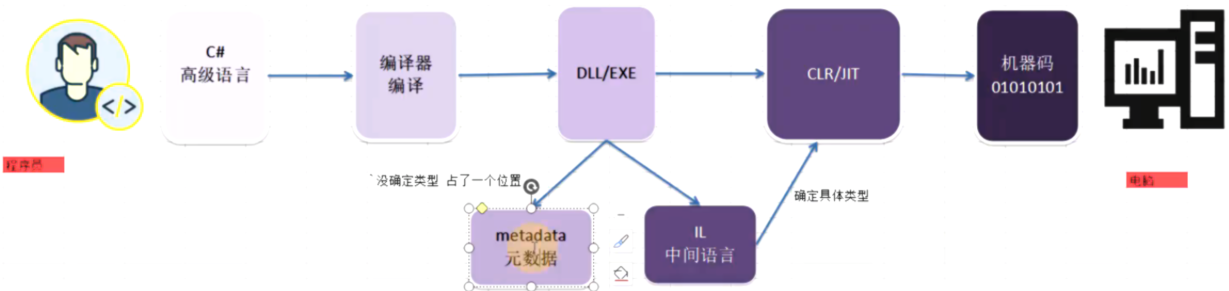
协变逆变是针对接口和委托来说的，离开了他们就没有这种说法。

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

}

}

**6. Lambda**

就是一个匿名方法。

**7. 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。

**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。

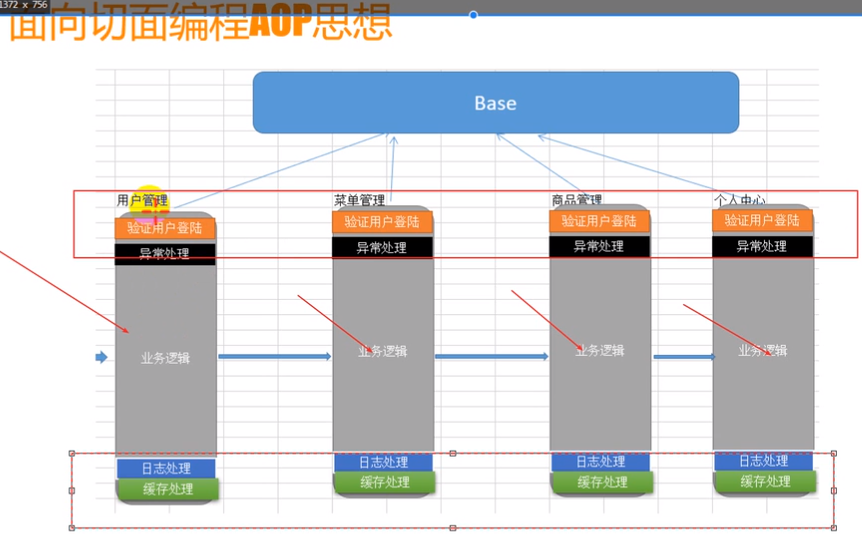
另一种限制线程数量 paralle

**4. 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

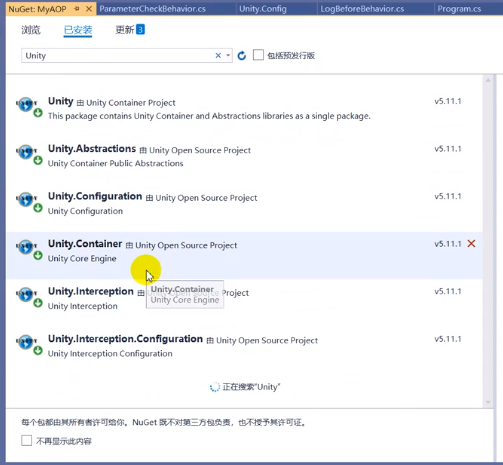
代理模式实现AOP

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

**5. 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。

**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。

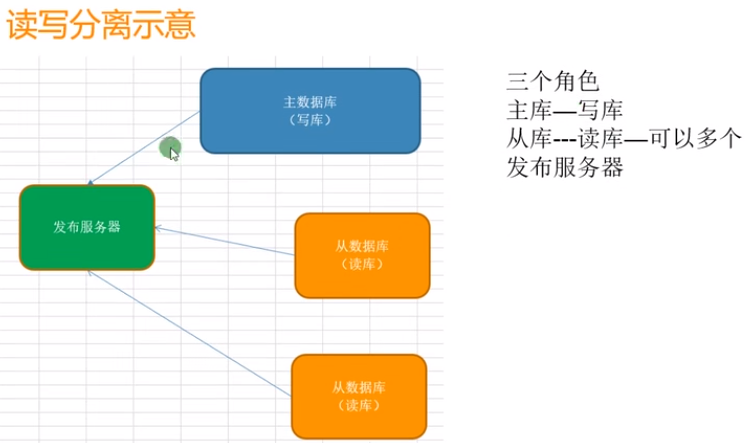
就这样不断的去visit，找到全部的叶节点。而且也找到对应的关系。

**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

**7. 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

**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. 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。

**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：

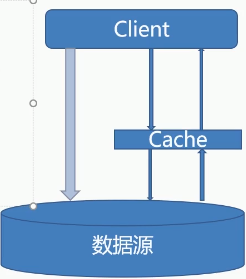
form 表单导入

js 导入，可以做到无刷新导入，可以使用组件

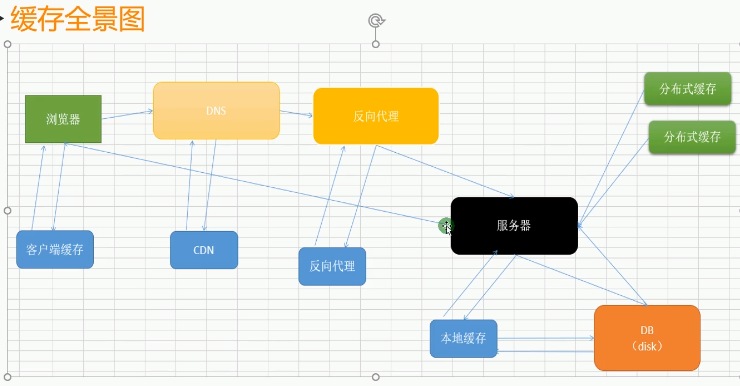
**54. 多级缓存**

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**

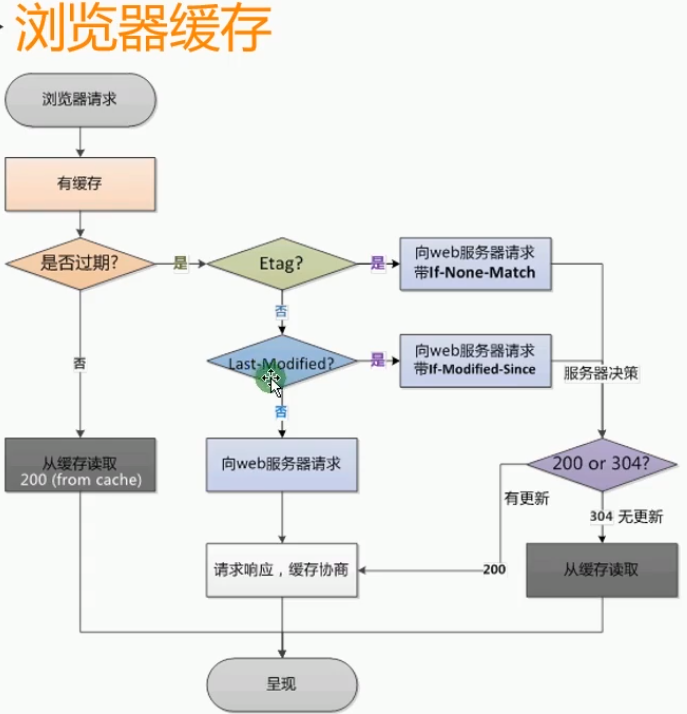


缓存—存时

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

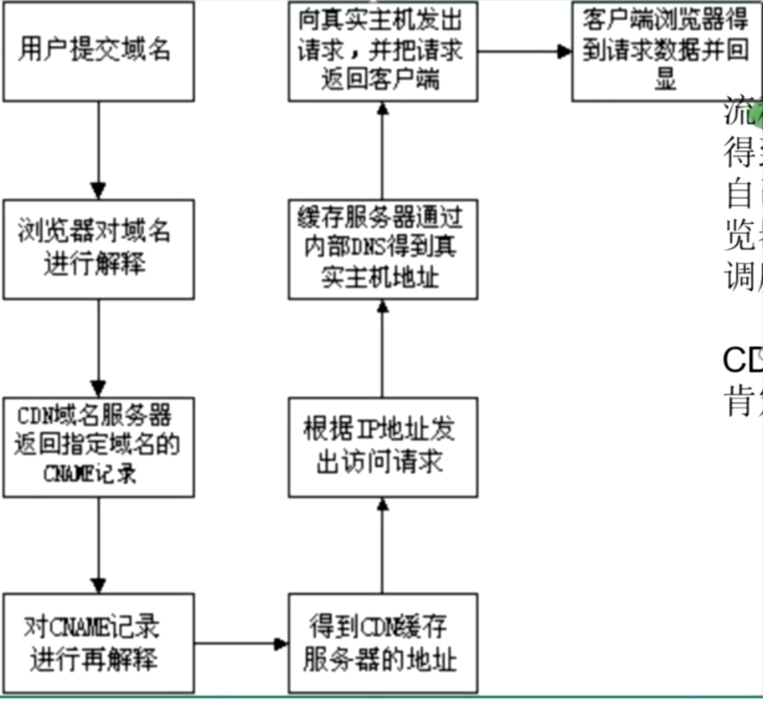
好处：快速获取，加快响应速度；减少请求，降低服务器压力

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---

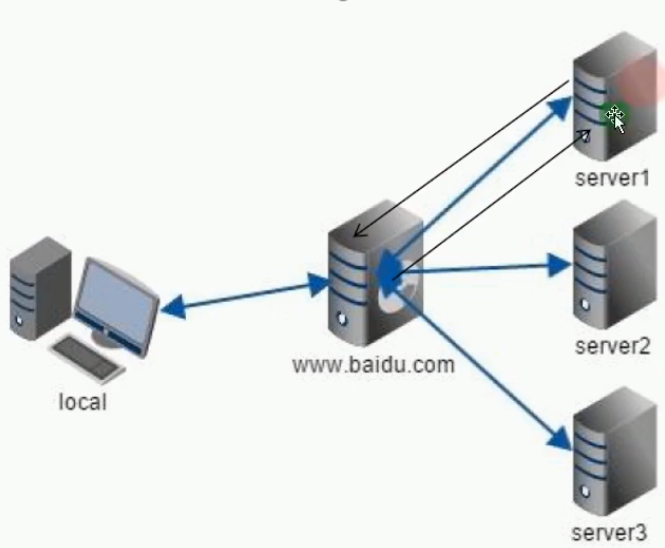
得到CND缓存服务器的地址---根据IP地址发出访问请求---缓存服务器通过内部DNS得到真实主机地址---向真实主机发出请求，并把请求返回客户端---客户端浏览器得到请求数据并回显

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

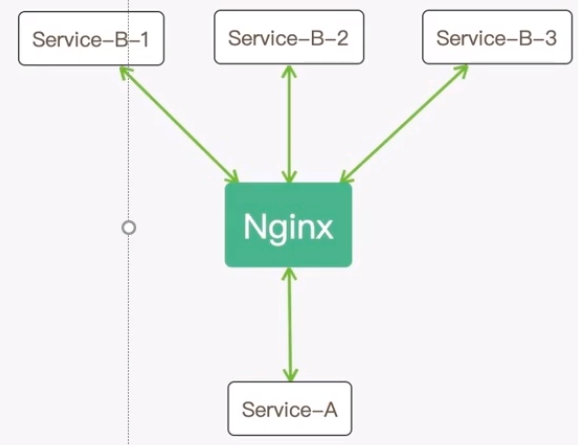
③依赖注入实例，调用就可以常规使用了

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**



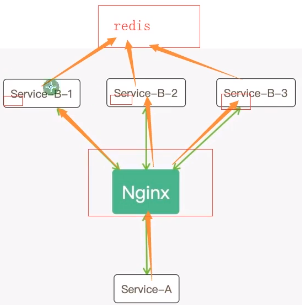
一套代码，实现开了3个端口。三个实例的缓存是在各自的进程(内存)中的，不能共享。

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

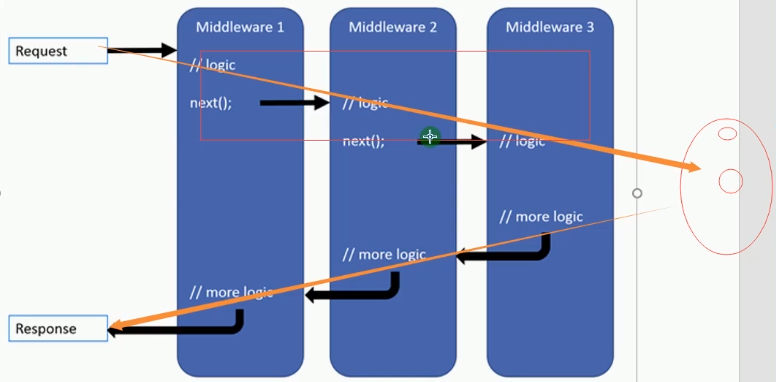
**基于http协议缓存**

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)

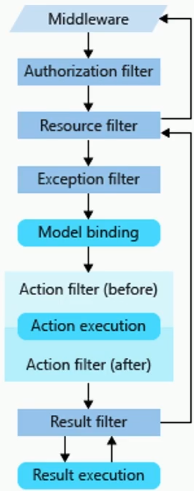
中间件响应缓存，服务器缓存，asp.net core 内部的，可跨浏览器，用户的缓存。

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

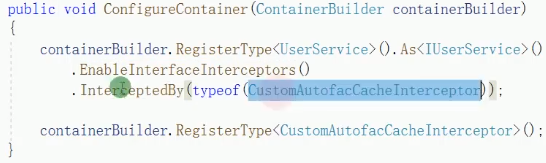
Aotufac-AOP使用

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

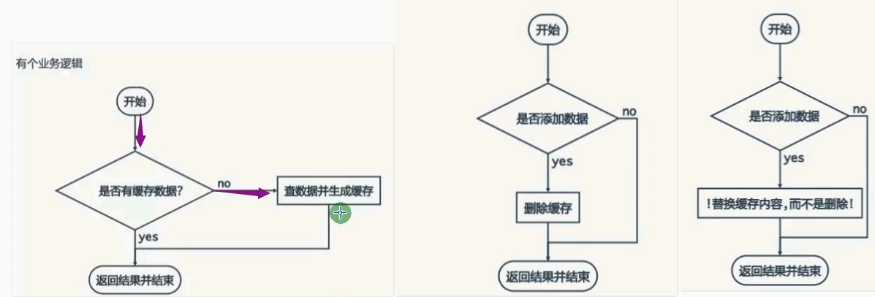
请求不存在的数据，永远没有缓存，失去保护的数据库也会抗不住(绕过缓存)

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---

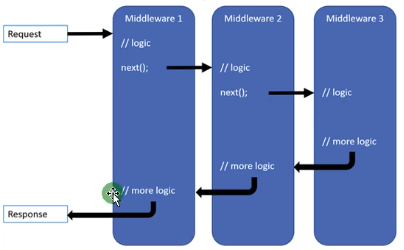
把三者打包给Kestrel，启动监听绑定动作。请求来了，循环监听服务器，收到请求就丢给线程池，HttpContextFactory去解析成HttpContext,交给管道模型，就各种中间件处理---MVC---执行过去了。

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

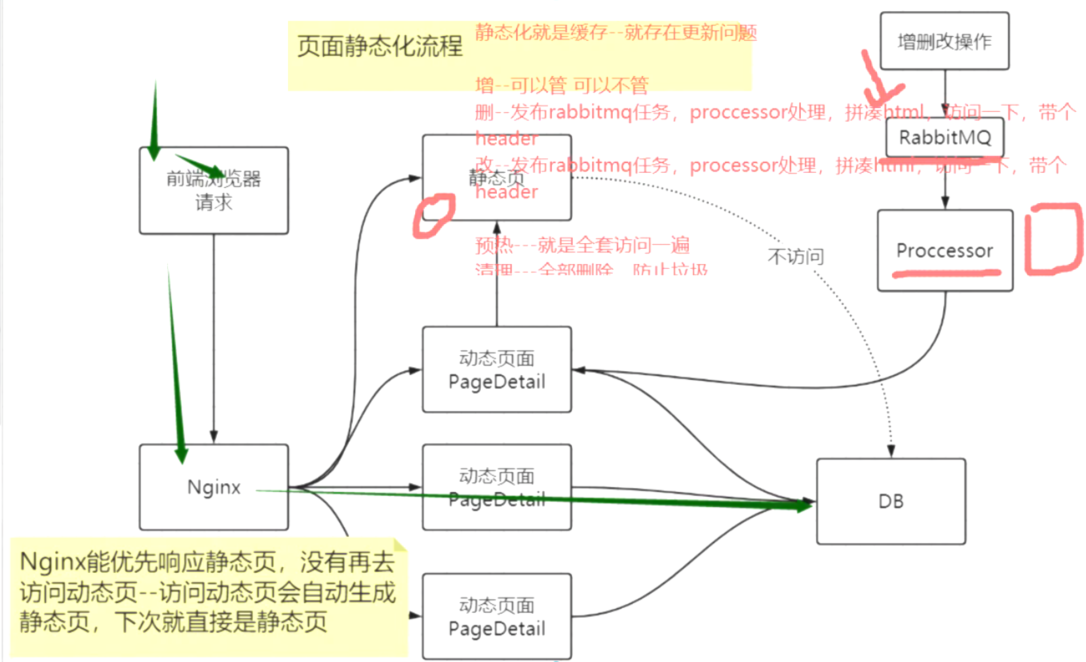
支持多种负载均衡策略：轮询、ip、least\_conn、fair、url\_hash

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。

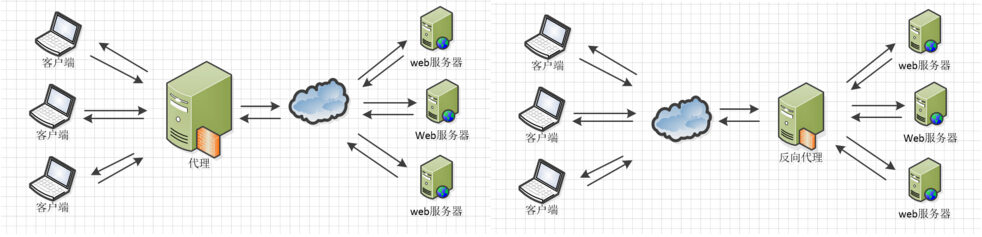
Nginx的稳定性、功能集、示例配置文件和低系统资源的消耗让他后来居上，在全球活跃的网站中有12.18%的使用比率，大约为2220万个网站。

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。

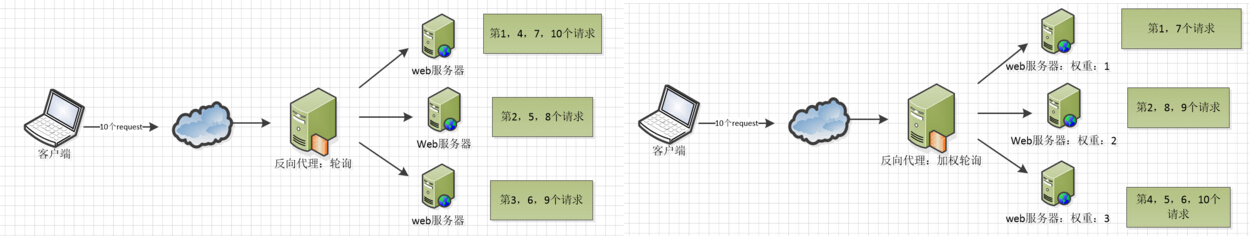


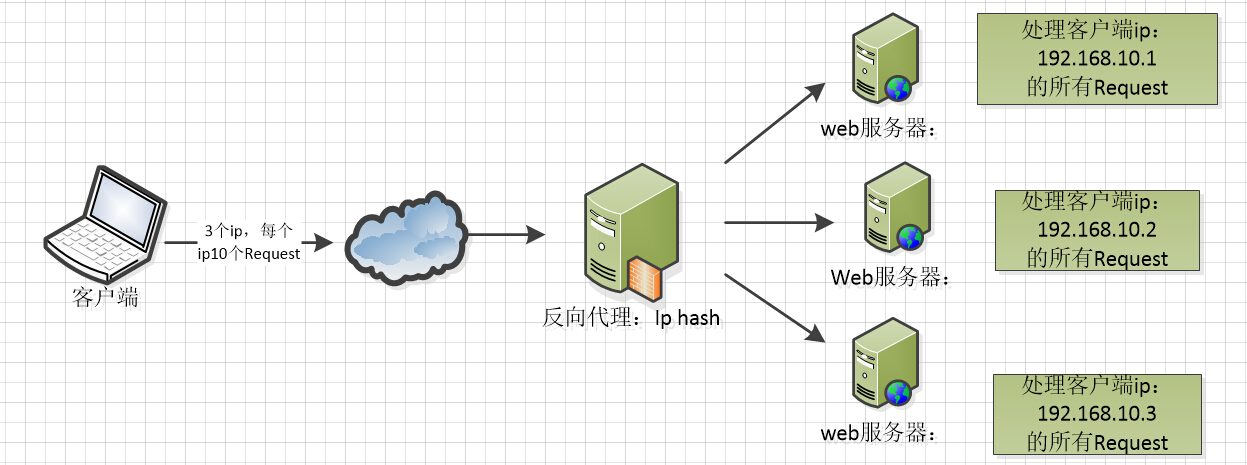
Nginx在做反向代理时，提供性能稳定，并且能够提供配置灵活的转发功能。Nginx可以根据不同的正则匹配，采取不同的转发策略，比如图片文件结尾的走文件服务器，动态页面走web服务器，只要你正则写的没问题，又有相对应的服务器解决方案，你就可以随心所欲的玩。并且Nginx对返回结果进行错误页跳转，异常判断等。如果被分发的服务器存在异常，他可以将请求重新转发给另外一台服务器，然后自动去除异常服务器。

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。