目录

[一、平台体系 16](#_Toc49253370)

[平台基本概念 16](#_Toc49253371)

[编译过程 16](#_Toc49253372)

[比较好的.net网站 18](#_Toc49253373)

[反编译工具 18](#_Toc49253374)

[AppDomain 21](#_Toc49253375)

[垃圾回收 24](#_Toc49253376)

[regsvr32 24](#_Toc49253377)

[查看环境变量 24](#_Toc49253378)

[GAC 25](#_Toc49253379)

[强名称 25](#_Toc49253380)

[json 26](#_Toc49253381)

[二、WebMVC开发： 27](#_Toc49253382)

[MVC和WebForm之间的区别 27](#_Toc49253383)

[MVP 29](#_Toc49253384)

[MVVM 29](#_Toc49253385)

[路由系统 29](#_Toc49253386)

[Razor语法 31](#_Toc49253387)

[入口，默认的路由： 32](#_Toc49253388)

[css和js引用： 33](#_Toc49253389)

[页面布局 35](#_Toc49253390)

[使用jquery 37](#_Toc49253391)

[AJAX 37](#_Toc49253392)

[一个简单的前后台交互，json 41](#_Toc49253393)

[发布 42](#_Toc49253394)

[三、Web API 43](#_Toc49253395)

[WebApi的路由机制 43](#_Toc49253396)

[数据格式支持json 45](#_Toc49253397)

[MVC WebAPI中的Controllers 45](#_Toc49253398)

[跨域 46](#_Toc49253399)

[过滤器 51](#_Toc49253400)

[一个Controller中多个GET/ POST/ PUT/ DELETE 51](#_Toc49253401)

[WebAPI安全验证的问题(顺便描述了web后台验证) 61](#_Toc49253402)

[服务的测试工具 63](#_Toc49253403)

[WebAPI返回值的问题 65](#_Toc49253404)

[参数传递 72](#_Toc49253405)

[WebAPI异常处理 79](#_Toc49253406)

[WebAPI自宿主 93](#_Toc49253407)

[WebAPI和MVC的区别(需要不断学习和完善) 95](#_Toc49253408)

[四、其他web知识 96](#_Toc49253409)

[超文本/超媒体(HyperTcxt/HyperModia) 96](#_Toc49253410)

[URI/URL: 96](#_Toc49253411)

[HTTP报文 97](#_Toc49253412)

[RPC架构和Restful风格 98](#_Toc49253413)

[Post和Get 101](#_Toc49253414)

[页面状态 102](#_Toc49253415)

[五、基本语法： 103](#_Toc49253416)

[访问权限 103](#_Toc49253417)

[字符串换行 105](#_Toc49253418)

[可选参数 105](#_Toc49253419)

[params 106](#_Toc49253420)

[Type 107](#_Toc49253421)

[浅拷贝与深拷贝 107](#_Toc49253422)

[var 108](#_Toc49253423)

[var 与 DataTable DataRow循环 108](#_Toc49253424)

[可空类型 108](#_Toc49253425)

[去掉0及小数点 109](#_Toc49253426)

[去掉小数点后的零 109](#_Toc49253427)

[反射 109](#_Toc49253428)

[装箱和拆箱 110](#_Toc49253429)

[泛型 110](#_Toc49253430)

[初始化器 111](#_Toc49253431)

[委托 112](#_Toc49253432)

[多播委托 112](#_Toc49253433)

[泛型委托(Func，Action,Predicate) 113](#_Toc49253434)

[泛型约束 115](#_Toc49253435)

[匿名函数 116](#_Toc49253436)

[Lambda表达式 117](#_Toc49253437)

[事件 117](#_Toc49253438)

[事件和委托的区别 118](#_Toc49253439)

[Default关键字 119](#_Toc49253440)

[Attribute 119](#_Toc49253441)

[进程 120](#_Toc49253442)

[线程 120](#_Toc49253443)

[多线程 121](#_Toc49253444)

[死锁 121](#_Toc49253445)

[前台线程和后台线程 122](#_Toc49253446)

[线程池 122](#_Toc49253447)

[异步 136](#_Toc49253448)

[有关线程的一些问题 142](#_Toc49253449)

[IQueryable接口与IEnumberable接口的区别 145](#_Toc49253450)

[延迟加载Lazy<T> 145](#_Toc49253451)

[文件的操作 147](#_Toc49253452)

[LINQ 147](#_Toc49253453)

[数组 153](#_Toc49253454)

[初始化并赋值，无需指定数组长度 153](#_Toc49253455)

[也可以直接作为参数传递 153](#_Toc49253456)

[多个string的if判断 154](#_Toc49253457)

[实用计算 154](#_Toc49253458)

[三角函数 154](#_Toc49253459)

[List 154](#_Toc49253460)

[初始化 154](#_Toc49253461)

[combox绑定List 155](#_Toc49253462)

[list查询和遍历 155](#_Toc49253463)

[使用First代表一定能找到，找不到就抛出异常，必须使用try……catch……进行异常的捕获。 155](#_Toc49253464)

[推荐使用  FirstOrDefault 或Find。 155](#_Toc49253465)

[List增加数据 156](#_Toc49253466)

[list查重 157](#_Toc49253467)

[list去重 157](#_Toc49253468)

[List字符串拼合 157](#_Toc49253469)

[HashTable 158](#_Toc49253470)

[combox绑定HashTable 158](#_Toc49253471)

[HashTable无法还原插入顺序 159](#_Toc49253472)

[DateTime 164](#_Toc49253473)

[中文日期转换 164](#_Toc49253474)

[DataTable 167](#_Toc49253475)

[创建DataTable 167](#_Toc49253476)

[查询和筛选 168](#_Toc49253477)

[去重 169](#_Toc49253478)

[CheckBoxColumn实时获取是否选中 170](#_Toc49253479)

[DataTable转换为List 171](#_Toc49253480)

[AsQueryable()和AsEnumerable()的区别 172](#_Toc49253481)

[DataTable按列查询 172](#_Toc49253482)

[DataTable遍历删除某几行 173](#_Toc49253483)

[DataSet 173](#_Toc49253484)

[DataSet.GetChanges 173](#_Toc49253485)

[Word操作 174](#_Toc49253486)

[基本概念 174](#_Toc49253487)

[直接关闭word 174](#_Toc49253488)

[点击某个按钮，将word置于窗口最前方 174](#_Toc49253489)

[简单操作 175](#_Toc49253490)

[选中某个区域 179](#_Toc49253491)

[表格跨行重复显示表头 179](#_Toc49253492)

[这种方式对合并单元格不适用。 179](#_Toc49253493)

[根据窗口调整表格 179](#_Toc49253494)

[表格宽度100% 179](#_Toc49253495)

[遍历表格 179](#_Toc49253496)

[字体对应表 180](#_Toc49253497)

[修改文字的背景色 180](#_Toc49253498)

[目录操作 180](#_Toc49253499)

[目录跳转 181](#_Toc49253500)

[获取目录位置 182](#_Toc49253501)

[判断当前位置是否是目录 183](#_Toc49253502)

[缩进 184](#_Toc49253503)

[行距 184](#_Toc49253504)

[分词 184](#_Toc49253505)

[将paragraph的\n外的其他文字，加背景色 184](#_Toc49253506)

[上标和下标(可用于科学计数) 186](#_Toc49253507)

[文件夹和文件操作 186](#_Toc49253508)

[打开目录 186](#_Toc49253509)

[查找文件 186](#_Toc49253510)

[下载文件 186](#_Toc49253511)

[基础连接已经关闭: 发送时发生错误 187](#_Toc49253512)

[请求被中止: 未能创建 SSL/TLS 安全通道 188](#_Toc49253513)

[使用WebClient可解决上述问题 188](#_Toc49253514)

[读文本文件 188](#_Toc49253515)

[写文本文件 189](#_Toc49253516)

[Excel操作 189](#_Toc49253517)

[基本概念 189](#_Toc49253518)

[身份证格式的问题 189](#_Toc49253519)

[操作后直接清退进程 189](#_Toc49253520)

[日志操作 190](#_Toc49253521)

[log4net 190](#_Toc49253522)

[File 192](#_Toc49253523)

[六、SignalR： 192](#_Toc49253524)

[七、Socket通信： 192](#_Toc49253525)

[八、Windows服务： 193](#_Toc49253526)

[九、Web Service： 193](#_Toc49253527)

[十、WCF服务： 194](#_Toc49253528)

[基本概念 194](#_Toc49253529)

[WCF 与 Asp.net Web service 比较 195](#_Toc49253530)

[WCF、.Net Remoting和WebService的关系 196](#_Toc49253531)

[DataContractSerializer 和 XmlSerializer的区别： 196](#_Toc49253532)

[基于Asp.net 的应用程序开发与面向服务开发 197](#_Toc49253533)

[创建一个简单的WCF 198](#_Toc49253534)

[WCF在MYSQL升级后获取不到数据，提示调用 SSPI 失败，请参见内部异常 199](#_Toc49253535)

[WCF内置的日志跟踪 200](#_Toc49253536)

[十一、经典桌面应用程序-WinForm： 202](#_Toc49253537)

[Git 202](#_Toc49253538)

[.vs、bin、obj这些控件可以不要上传，也可以忽略，只要.csproj项目文件就可以。 202](#_Toc49253539)

[布局控件 202](#_Toc49253540)

[进程 203](#_Toc49253541)

[获取当前进程并kill。 203](#_Toc49253542)

[带参数调试 203](#_Toc49253543)

[参数如路径有空格的解决方法 204](#_Toc49253544)

[事件 204](#_Toc49253545)

[sender 204](#_Toc49253546)

[调整控件位置 205](#_Toc49253547)

[简单的进度提示 205](#_Toc49253548)

[启动多参数应用程序 206](#_Toc49253549)

[Timer 208](#_Toc49253550)

[Tick事件 208](#_Toc49253551)

[winform嵌入word 209](#_Toc49253552)

[dsoframer 209](#_Toc49253553)

[没有注册类 (异常来自 HRESULT:0x80040154 (REGDB\_E\_CLASSNOTREG)) 210](#_Toc49253554)

[winform实现dll加载时注册 210](#_Toc49253555)

[dsoframer.ocx 报“The associated COM server does not support ActiveX Document emb” 211](#_Toc49253556)

[winform嵌入word 212](#_Toc49253557)

[dsoframer 212](#_Toc49253558)

[winform嵌入word 212](#_Toc49253559)

[dsoframer 212](#_Toc49253560)

[Form通过自定义URL协议在web中启动本地应用程序 212](#_Toc49253561)

[Form 214](#_Toc49253562)

[窗体美化无边框 214](#_Toc49253563)

[窗体始终处于激活状态显示在最前 214](#_Toc49253564)

[单例窗体点开处于激活状态显示在最前 215](#_Toc49253565)

[判断窗体是否打开 215](#_Toc49253566)

[最小化到托盘 215](#_Toc49253567)

[托盘控件notifyIcon运行出现两个或多个托盘图标的解决方法 218](#_Toc49253568)

[TabControl 219](#_Toc49253569)

[鼠标悬停 219](#_Toc49253570)

[鼠标点击 220](#_Toc49253571)

[禁止TabPage的点击事件 221](#_Toc49253572)

[动态添加TabPage 222](#_Toc49253573)

[ListBox 223](#_Toc49253574)

[拖拽 223](#_Toc49253575)

[RichTextBox 226](#_Toc49253576)

[每行文本颜色不通 226](#_Toc49253577)

[GridView 227](#_Toc49253578)

[不显示行头 227](#_Toc49253579)

[列自动填充 227](#_Toc49253580)

[GridView显示tip 227](#_Toc49253581)

[遍历GridView 227](#_Toc49253582)

[GridView显示tip 228](#_Toc49253583)

[GridView按钮 228](#_Toc49253584)

[GridView里的checkbox事件 229](#_Toc49253585)

[GridView里的checkbox事件无法选中 230](#_Toc49253586)

[ 230](#_Toc49253587)

[GridView列头背景 230](#_Toc49253588)

[GridView奇数行背景变化 230](#_Toc49253589)

[OpenFileDialog 231](#_Toc49253590)

[基本属性 231](#_Toc49253591)

[TreeView 231](#_Toc49253592)

[TreeView默认执行afterselect事件 231](#_Toc49253593)

[ToolStrip 231](#_Toc49253594)

[ToolStripButton按钮文字样式 231](#_Toc49253595)

[Panel 232](#_Toc49253596)

[动态往下添加控件 232](#_Toc49253597)

[Splitter 232](#_Toc49253598)

[基本使用 232](#_Toc49253599)

[BackgroudWorker 233](#_Toc49253600)

[基本使用 233](#_Toc49253601)

[管理员身份运行批处理 234](#_Toc49253602)

[配置文件 234](#_Toc49253603)

[读取指定路径配置文件的方法 235](#_Toc49253604)

[RestSharp 235](#_Toc49253605)

[form-data 235](#_Toc49253606)

[常见问题 237](#_Toc49253607)

[winform写的多线程后台处理操作过段时间就会报错 237](#_Toc49253608)

[十二、WPF： 238](#_Toc49253609)

[十三、TypeScript： 239](#_Toc49253610)

[十四、Agular： 239](#_Toc49253611)

[十五、Vue： 239](#_Toc49253612)

[十六、React： 239](#_Toc49253613)

[十七、Entity Framework： 239](#_Toc49253614)

[十八、Core: 241](#_Toc49253615)

[十九、工作流： 242](#_Toc49253616)

[二十、缓存Redis/Memcached： 243](#_Toc49253617)

[Memcached和Redis的区别 243](#_Toc49253618)

[Windows下如何安装 243](#_Toc49253619)

[二十一、消息队列 244](#_Toc49253620)

[MSMQ 245](#_Toc49253621)

[概念 245](#_Toc49253622)

[如何安装 245](#_Toc49253623)

[MSMQ中主要有两个概念 246](#_Toc49253624)

[MSMQ的使用 247](#_Toc49253625)

[RabbitMQ 248](#_Toc49253626)

[安装 248](#_Toc49253627)

[使用 252](#_Toc49253628)

[远程连接RabbitMQ失败 255](#_Toc49253629)

[二十二、NuGet： 255](#_Toc49253630)

[二十三、其他计算机术语和名词 256](#_Toc49253631)

[持久化 256](#_Toc49253632)

[Serverless 256](#_Toc49253633)

[二十四IIS和管道 256](#_Toc49253634)

[WEB服务器解析的一般原理 256](#_Toc49253635)

[Java Web和ASP.NET的区别 257](#_Toc49253636)

[IIS 5.x 260](#_Toc49253637)

[IIS 6.0 264](#_Toc49253638)

[IIS 7.0 265](#_Toc49253639)

[上传文件大小限制 270](#_Toc49253640)

[简化的流程 271](#_Toc49253641)

[管道模型 272](#_Toc49253642)

[IIS服务器网站不支持带加号“+”的目录或文件名的解决方案 273](#_Toc49253643)

[二十五web.config 274](#_Toc49253644)

[二十六 常用的类或必须掌握的类 275](#_Toc49253645)

[RestSharp 275](#_Toc49253646)

[Simple REST and HTTP API Client for .NET 275](#_Toc49253647)

[具体使用： 275](#_Toc49253648)

[HttpContext类 276](#_Toc49253649)

[HttpContet对象之Request 276](#_Toc49253650)

[HttpContent对象之Response 278](#_Toc49253651)

[二十七Ninject 279](#_Toc49253652)

[二十七 应知应会 280](#_Toc49253653)

[编译型和解释型编程语言 280](#_Toc49253654)

[语言的类名和文件名 282](#_Toc49253655)

[编码格式（乱码的缘由） 282](#_Toc49253656)

[URL汉字编码问题以及乱码解决 284](#_Toc49253657)

[IO 285](#_Toc49253658)

[HTTPS 286](#_Toc49253659)

[同源策略(跨域) 290](#_Toc49253660)

[抓包 290](#_Toc49253661)

[Wireshark详细信息的层与对应OSI七层模型 291](#_Toc49253662)

[过滤器 291](#_Toc49253663)

[常见的后台认证机制 292](#_Toc49253664)

[HTTP Basic Auth 292](#_Toc49253665)

[OAuth 293](#_Toc49253666)

[Cookie Auth 293](#_Toc49253667)

[Token Auth 294](#_Toc49253668)

[cookie 295](#_Toc49253669)

[session 296](#_Toc49253670)

[token 296](#_Toc49253671)

[JWT 297](#_Toc49253672)

[json 298](#_Toc49253673)

[复杂json处理 298](#_Toc49253674)

[dll引用 299](#_Toc49253675)

[路径的含义 299](#_Toc49253676)

[AOP 299](#_Toc49253677)

[文件同步 300](#_Toc49253678)

[修改共享文件夹的登录凭据 301](#_Toc49253679)

[计算机端口相关知识 301](#_Toc49253680)

[netstats命令 301](#_Toc49253681)

[端口转发 301](#_Toc49253682)

[端口转发没有作用 302](#_Toc49253683)

[二十九、操作数据库 303](#_Toc49253684)

[连接字符串 303](#_Toc49253685)

[操作二进制文件 303](#_Toc49253686)

# 一、平台体系

平台基本概念

**.NET Framework：**简单点来说是一个框架、而C#应用程序或ASP.NET应用程序都是运行在这个框架之上的、个人理解为相当于java的jdk、其中.NET Framework下包括两个主要的组件：公共语言运行时(CLR)和.NET Framework类库。

**.NET Framework类库：**.NET Framework 类库是一个由类、接口和值类型组成的库，通过该库中的内容可访问系统功能。它是生成 .NET Framework 应用程序、组件和控件的基础。

**托管代码：**通常将在CLR控制下运行的代码称为托管代码(managed code)。

编译过程

**在.NET中,编译分为两个阶段：**

(1)把源代码编译为Microsoft中间语言(IL)。

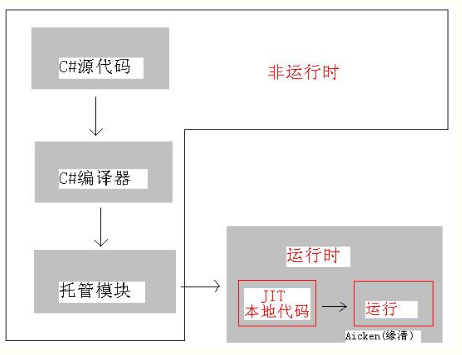
(2)CLR把IL编译为平台专用的代码。

在C#代码运行前，一般会经过两次编译，第一阶段是C#代码向MSIL的编译，第二阶段是IL向本地代码的编译。第一阶段的编译成果是生成托管模块，第二阶段的编译成果是生成本地代码以供运行，第一阶段生成的MSIL是不能直接运行的。

**JIT编译器(Just In Time）：**[实时编译](http://blog.csdn.net/huchunfu/article/details/8064782)，是.Net边运行边编译的一种机制，并不是把整个应用程序一次编译完(这样会有很长的启动时间),而是只编译它调用的那部分代码(这是其名称由来)。代码编译过一次后,得到的本地可执行程序就存储起来,直到退出该应用程序为止,这样在下次运行这部分代码时,就不需要重新编译了。Microsoft认为这个过程要比一开始就编译整个应用程序代码的效率高得多,因为任何应用程序的大部分代码实际上并不是在每次运行期间都执行。使用JIT编译器,从来都不会编译这种代码。

JIT是运行时的一个重要职责模块，它将IL转换为本地CPU指令，从上图可以看出，也许你不敢相信，即时编译这个过程是在运行时发生的，这会不会对性能产生影响呢？事实上答案是虽然是肯定的，但这种开销物有所值。

下图可以表示出JIT的介入时机：



JIT应该是在“(2)CLR把IL编译为平台专用的代码” 过程中介入工作。而生成.exe应该是“(1)把源代码编译为Microsoft中间语言(IL)”这个阶段，且并不是把整个应用程序一次编译完(这样会有很长的启动时间),而是只编译它调用的那部分代码(这是其名称由来)。代码编译过一次后,得到的本地可执行程序就存储起来,直到退出该应用程序为止,这样在下次运行这部分代码时,就不需要重新编译了。所以.NET程序第一次运行的时候很慢。

具体可参见<http://www.cnblogs.com/zjoch/p/5237483.html>

和<http://blog.csdn.net/huchunfu/article/details/8064782>

MSIL 全称为Microsoft Intermediate Language，中文译为“微软中间语言”，它是一种介于高级语言和汇编语言之间的伪汇编语言(姑且这么叫，各位有不同意见的同学不必激动)。当用户编译运行一个.NET程序时，高级语言编译器会将源代码翻译成一组可以独立于CPU的指令。

托管模块(managed module)是一个标准32位或64位Microsoft  Windows可移植可执行体(PE32或PE32+)文件，托管模块需要CLR才能执行，它包含了上面介绍的IL代码，还包含元数据、PE头、CLR头几部分。

比较好的.net网站

博客里有些知识还是比较全的：<https://www.cnblogs.com/lonelyxmas/p/10606881.html>

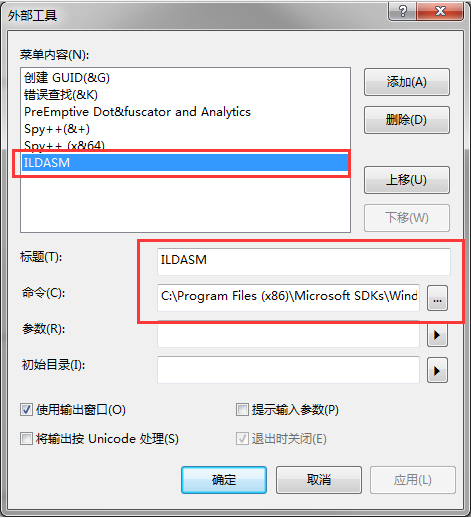
反编译工具

**ILDasm工具：**想要看IL代码需要使用ILDasm工具。默认的路径在(也就是在SDK中的bin下面)：C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.7 Tools\ildasm.exe

a.我们可以win+R.输入：C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.7 Tools\ildasm.exe (window 7 64位 [操作系统](http://www.knowsky.com/system.asp)安装目录) 同样可以打开ildasm。

b.添加到VS中：





c.命令一栏中输入具体的路径：C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ilasm.exe，点击“应用”即可。

**图标含义**



AppDomain

**AppDomain：**AppDomain是CLR的运行单元，它可以加载Assembly、创建对象以及执行程序。AppDomain是CLR实现代码隔离的基本机制。每一个AppDomain可以单独运行、停止；每个AppDomain有自己默认的异常处理；一个AppDomain的运行失败不会影响到其他的AppDomain。

CLR在被CLR Host(windows shell or InternetExplorer or SQL Server)加载后，要创建一个默认的AppDomain，程序的入口点(Main方法)就是在这个默认的AppDomain中执行。

**AppDomain vs 进程:** AppDomain被创建在进程中，一个进程内可以有多个AppDomain。一个AppDomain只能属于一个进程。

**AppDomain vs 线程:** AppDomain是个静态概念，只是限定了对象的边界；线程是个动态概念，它可以运行在不同的AppDomain。一个AppDomain内可以创建多个线程，但是不能限定这些线程只能在本AppDomain内执行代码。CLR中的System.Threading.Thread对象其实是个soft thread，它并不能被操作系统识别；操作系统能识别的是hard thread。

一个soft thread只属于一个AppDomain，穿越AppDomain的是hard thread。当hard thread访问到某个AppDomain时,一个AppDomain就会为之产生一个soft thread。hard thread有thread local storage(TLS)，这个存储区被CLR用来存储这个hard thread当前对应的AppDomain引用以及softthread引用。当一个hard thread穿越到另外一个AppDomain时，TLS中的这些引用也会改变。

**AppDomain vs Assembly:** Assembly是.Net程序的基本部署单元，它可以为CLR提供用于识别类型的元数据等等。Assembly不能单独执行，它必须被加载到AppDomain中，然后由AppDomain创建程序集中的对象。一个Assembly可以被多个AppDomain加载，一个AppDomain可以加载多个Assembly。每个AppDomain引用到某个类型的时候需要把相应的assembly在各自的AppDomain中初始化。因此，每个AppDomain会单独保持一个类的静态变量。

**AppDomain vs 对象:** 任何对象只能属于一个AppDomain。AppDomain用来隔离对象，不同AppDomain之间的对象必须通过Proxy(reference type)或者Clone(value type)通信。

建立和关闭进程

利用 Start 与Kill 方法可以简单建立或者销毁进程。

static void Main(string[] args)

{

System.Diagnostics.Process process = Process.Start("notepad.exe","File.txt");

Thread.Sleep(2000);

process.Kill();

}

Process打开程序需注意

最好写全需打开程序的地址。word等第三方程序除外。

ProcessStartInfo info = new ProcessStartInfo(installversionfilename);

info.Arguments = \_param;

//大于等于6才有必要性以管理员身份运行

if (Environment.OSVersion.Version.Major >= 6)

{

info.Verb = "runas";

}

var process = System.Diagnostics.Process.Start(info);

以管理员身份运行

startInfo.UseShellExecute = true;

startInfo.Verb = "runas";

Process.Start(startInfo);

需要加上这句：startInfo.Verb = "runas";来触发UAC，执行提权；

UAC只能通过操作系统Shell启动进程来触发，所以，设置ProcessStartInfo.UseShellExecute=true；

\*.pdb和\*.dll的文件的区别

dll是程序组成的一部分，我见过的dll可以是一个类，也可以是一个控件，PDB是Program DataBase的缩写，程序数据库 (PDB) 文件保存着调试和项目状态信息，使用这些信息可以对程序的调试配置进行增量链接。详见：<http://blog.csdn.net/wangjianzhongfj/article/details/7339172>。

垃圾回收

**垃圾回收器(GC)：**

regsvr32

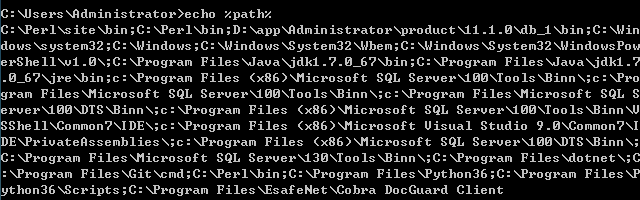
用Regsvr32注册Dll文件起到什么作用: Regsvr32命令用于注册[动态链接库文件](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E9%93%BE%E6%8E%A5%E5%BA%93%E6%96%87%E4%BB%B6" \t "_blank)，是 Windows 系统提供的用来向系统注册控件或者卸载控件的命令，以命令行方式运行。WinXP及以上系统的[regsvr32.exe](https://baike.baidu.com/item/regsvr32.exe" \t "_blank)在windows\system32文件夹下；2000系统的regsvr32.exe在winnt\system32文件夹下。

注册一个组件使用regsvr32 xxx.dll命令。其中DLL（即xxx.dll）输出四个函数。DllGetClassObject，DllRegisterServer，DllUnregisterServer和DllMain。DllRegisterServer由词义看出其用于[注册组件](https://baike.baidu.com/item/%E6%B3%A8%E5%86%8C%E7%BB%84%E4%BB%B6" \t "_blank)，在windows的注册表中给某个dll的登记。注意DLL（xxx.dll）内部包含一个或多个组件。regsvr32 -s xxx.dll即regsvr32调用xxx.dll中的此函数完成组件的注册。多说一句，注册是为了能够索引到xxx.dll然后创建组件。如：

regsvr32.exe /u thumbvw.dll就可以卸载掉图片预览功能。需要恢复时输入regsvr32 thumbvw.dll

查看环境变量

win+R,cmd,echo %path%



GAC

全局程序集缓存(Global Assembly Cache, GAC)是计算机范围内的代码缓存，它存储专门安装的程序集，这些程序集由计算机上的许多应用程序共享。在全局程序集缓存中部署的应用程序必须具有[强名称](https://baike.baidu.com/item/%E5%BC%BA%E5%90%8D%E7%A7%B0" \t "_blank)。

.Net提供的命令行工具gacutil.exe用于支持这一功能。gacutil.exe可以将具有强名称的程序集添至全局程序集缓存。

GAC位置一般在系统盘下Windows/Assembly目录下，如C:\Windows\assembly\。

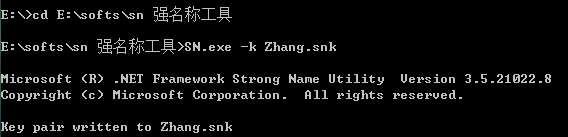
强名称

强名称 (strong name) 由一个[程序集](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E9%9B%86" \t "_blank)的标识组成并通过[公钥](https://baike.baidu.com/item/%E5%85%AC%E9%92%A5)和[数字签名](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D)（针对该程序集生成）加强的名称，其中的标识包括程序集的[简单文本](https://baike.baidu.com/item/%E7%AE%80%E5%8D%95%E6%96%87%E6%9C%AC" \t "_blank)名称、[版本号](https://baike.baidu.com/item/%E7%89%88%E6%9C%AC%E5%8F%B7/7674145)和区域性信息（如果提供的话）。

一个[程序集](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E9%9B%86)通常包括4个自定义属性，用来惟一识别程序集，它们是文件名（无扩展名）、版本号、区域性标识和[公钥](https://baike.baidu.com/item/%E5%85%AC%E9%92%A5" \t "_blank)标记，比如："Hello, version=1.0.0.0, culture=neutral, publicKeyToken=9ebd24cc40ff21fd"，其中publicKeyToken是公钥的64位散列值，可以唯一标识程序集，拥有公钥标记的程序集称为强名称程序集，强名称程序集可以防篡改；可以安装到GAC中，为多个程序集所共享；还可以实施[版本策略](https://baike.baidu.com/item/%E7%89%88%E6%9C%AC%E7%AD%96%E7%95%A5" \t "_blank)（不同版本的强名称程序集被认为是不同的程序集）。而公钥标记为空的[程序集](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E9%9B%86" \t "_blank)只能存在于所引用的程序集的目录以及子目录中，也无法实施版本策略（引用它的程序集会忽略[版本号](https://baike.baidu.com/item/%E7%89%88%E6%9C%AC%E5%8F%B7/7674145" \t "_blank)）。

公钥可以通过.NET SDK提供的SN.exe命令(下载后放在联想电脑E:\softs\sn 强名称工具\)生成，生成方法是：SN.exe -k Zhang.snk，SN.exe可以下载。

这样就得到一个名为Zhang.snk的公/私密钥对文件，将其放至某一目录下，比如G:\，然后就可以创建强名称[程序集](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E9%9B%86" \t "_blank)了，只需要在[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81)中添加下面的[自定义属性](https://baike.baidu.com/item/%E8%87%AA%E5%AE%9A%E4%B9%89%E5%B1%9E%E6%80%A7)即可：[assembly:AssemblyKeyFile(@"G:\Zhang.snk")]。





json

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它使得人们很容易的进行阅读和编写。同时也方便了机器进行解析和生成。

json简单说就是javascript中的对象和数组，所以这两种结构就是对象和数组两种结构，通过这两种结构可以表示各种复杂的结构:

对象：对象在js中表示为“{}”括起来的内容，数据结构为 {key：value,key：value,...}的键值对的结构，在面向对象的语言中，key为对象的属性，value为对应的属性值，所以很容易理解，取值方法为 对象.key 获取属性值，这个属性值的类型可以是 数字、字符串、数组、对象几种。尽量都加上””。

数组：数组在js中是中括号“[]”括起来的内容，数据结构为 ["java","javascript","vb",...]，取值方式和所有语言中一样，使用索引获取，字段值的类型可以是 数字、字符串、数组、对象几种。

查询本机.net版本

dir %WINDIR%\Microsoft.Net\Framework\v\* /B

# 二、WebMVC开发：

MVC和WebForm之间的区别

MVC和WebForm之间的区别可参见“C#”-“应用”-“MVC”中的《[解析ASP.NET WebForm和Mvc开发的区别](http://www.cnblogs.com/yisuowushinian/p/3495610.html)》(http://blog.csdn.net/yisuowushinian/article/details/17646121)。

三层架构分为界面层、业务逻辑层、数据访问层，是从整个应用程序架构的角度来划分的三层。三层是为了解决整个应用程序中各个业务操作过程中不同阶段的代码封装的问题，为了使程序员更加专注的处理某阶段的业务逻辑。很多人就同MVC里的三个核心部件同三层架构等同起来，认为界面层等于View,业务逻辑层等于Controller,数据访问层等于Model，这是完全错误的。

MVC设计模式解决的是页面代码、页面控制逻辑和数据耦合的问题，所以它首先属于界面层(表现层)，主要是为了解决应用程序用户界面的样式替换问题，把展示数据的 HTML 页面尽可能的和业务代码分离。比如：[Asp.net](http://asp.net/) MVC和Structs都是界面层框架。

MVC把纯净的界面展示逻辑（用户界面）独立到一些文件中（Views），把一些和用户交互的程序逻辑（Controller）单独放在一些文件中，在 Views 和 Controller 中传递数据使用一些专门封装数据的实体对象，这些对象，统称为Models。

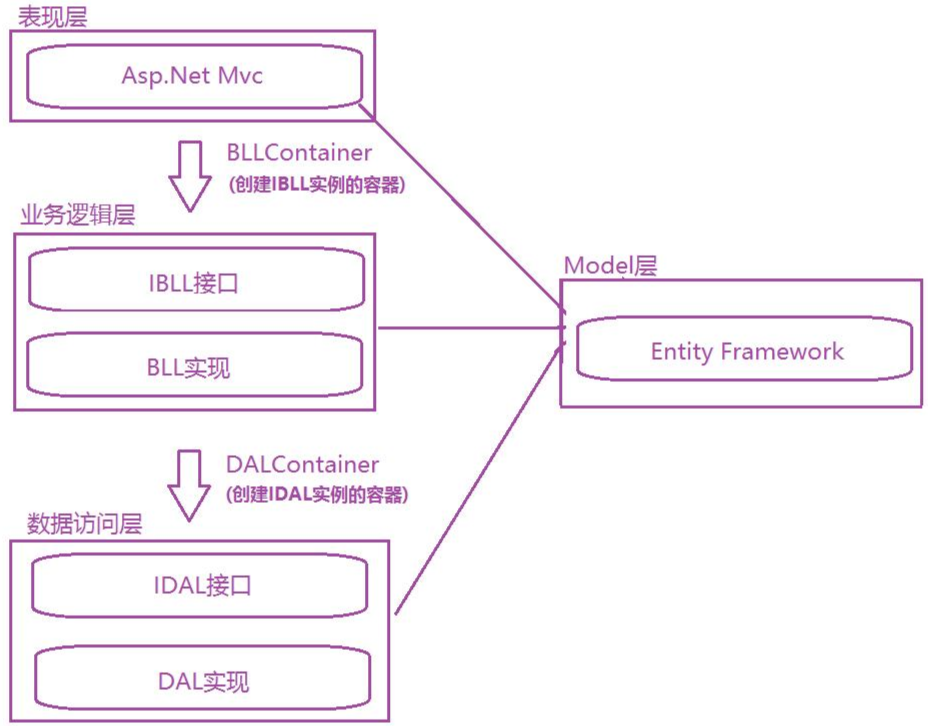
MVC里的Controller是负责对页面进行控制的，比如：页面间跳转，显示逻辑等。三层架构里的业务逻辑层主要是对业务实体数据的加工，把加工后的数据传给页面显示。

MVC里的Model只是数据实体，是数据的容器，不具备什么增删改查的功能，他接收的数据是从业务逻辑层处理好传过来的数据。而三层架构里的数据访问层是具有增删改查功能的，是直接对数据库操作的，为业务逻辑层提供数据支持。也不一定非得用EF，用其他方式能获取数据，ADO.NET也行。

切忌MVC只是三层里的界面层，千万别混淆，这是经典的面试题之一。

所以说MVC和三层毫无关系，是因为它们二者使用范围不同：三层可以应用于任何语言、任何技术的应用程序；而MVC只是为了解决BS应用程序视图层各部分的耦合关系。它们互不冲突，可以同时存在，也可根据情况使用其中一种。

对于多层架构来说（比如我们经常提及的三层架构），MVC是被当成是UI呈现层（Presentation Layer）的设计模式，而Model则更多地体现为访问业务层的入口（Gateway）。如果采用面向服务的设计，将业务功能定义成相应服务并通过接口（契约）的形式暴露出来，这里的Model甚至还可以表示成进行服务调用的代理。



MVP

Windows Form和Web Form,如果都写在aspx.cs中，页面会比较混乱。

MVVM

MVP的进化，广泛应用于WPF和Sliverlight程序。界面主要显示View。相当于逻辑操作都在ViewModel层。

路由系统

ASP.NET通过URL路由系统实现了请求地址与物理文件的分离。具体详见《ASP.NET的路由系统 URL与物理文件的分离》(<http://www.cnblogs.com/artech/archive/2012/03/19/aspnet-routing-01.html>)

ASP.NET MVC的路由系统通过对HTTP请求的解析得到表示Controller、Action和其他相关的数据，并以此为依据激活Controller对象，调用相应的Action方法，并将方法返回的ActionResult（ExecuteResult方法）写入HTTP回复中。

public abstract class ActionResult

{

public abstract void ExecuteResult(ControllerContext context);

}

public class StaticViewResult: ActionResult

{

public override void ExecuteResult(ControllerContext context)

{

context.RequestContext.HttpContext.Response.WriteFile(context.RequestContext.RouteData.Action + ".html");

}

}

至于具体的原理性的东西和模拟的MVC机制，可参见《[通过一个模拟程序让你明白ASP.NET MVC是如何运行的](http://www.cnblogs.com/artech/archive/2011/12/05/asp-mvc-how-to-work.html)》(<http://www.cnblogs.com/artech/archive/2011/12/05/asp-mvc-how-to-work.html>)

App\_Start中的RouteConfig.cs是用来配置路由的。

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

说明默认的路由为：Views/Home/Index.html

打开Views/Home/Index.html：

@usingCompany.Model

@model List<Company.Model.Staff>

@{

Layout = null;

}

@代表是Asp.net代码。

Layout = null; //代表没有使用任何模板。

否则的话：

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

代表本页面的模板是"~/Views/Shared/\_Layout.cshtml"。

Razor语法

定义变量：

@{

var weekday=DateTime.Now.DayOfWeek;

var day=weekday.ToString();

var message="";

}

循环：

<div>

<ul>

@for (int i = 0; i < 10; i++)

{

<li>@i</li>

}

</ul>

</div>

while:

<div>

<ul>

@{

var j = 1;

while (j <= 10)

{

<li>@j</li>

j++;

}

}

</ul>

</div>

入口，默认的路由：

.net framework：App\_Start中的RouteConfig.cs

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

.net core：Startup.cs中的

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

表示默认是Home/Index。

貌似core中的cshtml文件也没有设计、拆分、源视图。

css和js引用：

(1)正常引用可使用：

<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />

<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"></script>

HTML 帮助器用于修改 HTML 输出：

@Url.Content() - URL 内容在此处插入。

@Html.ActionLink() - HTML 链接在此处插入。

(2)当然也可以对css和js文件压缩后再整体引用：

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/bootstrap")

其实是对css和js文件进行压缩处理。通过App\_Start中的BundleConfig.cs文件设置压缩规则，如：

bundles.Add(new ScriptBundle("~/bundles/jquery").Include(

"~/Scripts/jquery-{version}.js"));

bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(

"~/Scripts/bootstrap.js",

"~/Scripts/respond.js"));

@Scripts.Render("~/bundles/jquery")其实就是引用了~/Scripts/jquery-{version}.js下的所有带版本的jquery.js，而"~/Scripts/bootstrap.js"和"~/Scripts/respond.js"则打包在@Scripts.Render("~/bundles/bootstrap")中。

详细的操作请参见技术-C#-应用-MVC中的《ASP.NET MVC 4 RC的JS和CSS打包压缩功能 Scripts.Render和Styles.Render》和《[mvc中Scripts.Render的用法](http://blog.csdn.net/whaxrl/article/details/50953195)》。

页面布局

文件 \_Layout.cshtml 表示应用程序中每个页面的布局。它位于 Views 文件夹中的 Shared 文件夹。

@ViewBag.Title - 在此处插入页面标题。

@RenderBody() - 此处呈现页面内容。

Shared 文件夹（位于 Views 文件夹内）中的 \_ViewStart 文件包含以下内容：

@{Layout = "~/Views/Shared/\_Layout.cshtml";}

这段代码被自动添加到由应用程序显示的所有视图。

如果删除该文件，则必须向所有视图添加这段代码。

能够在 Views 文件夹中找到以下 HTML 文件类型：

|  |  |
| --- | --- |
| **文件类型** | **扩展名** |
| 纯 HTML | .htm or .html |
| 经典 ASP | .asp |
| 经典 ASP.NET | .aspx |
| ASP.NET Razor C# | .cshtml |
| ASP.NET Razor VB | .vbhtml |

文件 Index.cshtml 表示应用程序的首页。它是应用程序的默认文件（首页文件）。

修改路由或新加了Controller或Viewer，如在IIS中查看，还是需要先编译一下，如在VS中则不需要。

Asp.net MVC中提供了RenderSection方法，这样就能够在Layout中定义一些区块，这些区块留给使用Layout的view来实现,比如我们定义的Layout如下， 定义了一个”Footer”的section， 把这个section留给各个view去填充。

<!DOCTYPE html>

<html>

<head>

<title>Sample Layout</head>

<body>

<div>@RenderBody()</div>

<footer>@RenderSection("Footer")</footer>

</body>

</html>

在使用该Layout的view中，可以这样来填充“Footer” section.

@{

Layout = "MyLayout.cshtml";

}

<h1>Main Content!</h1>

@section Footer {

This is the footer.

}

RenderSection方法还有一个参数， 如果使用RenderSection(“Footer”, false) ，则表示，这个section Footer，view中不是必须要实现。

使用jquery

对于MVC来说，尽量少用$(function(){});貌似会有问题，可以如下操作：

var object;

$(document).ready(function () {

$("#btnOK").click(function () {

GetValue();

alert(object.hh1);

alert(object.hh2);

});

function GetValue()

{

object = {

name: "Jack",

sex: "男",

hh1: $("#display1").val(),

hh2: $("#cctext").val(),

getNameFunc: function () {

return function () {

return this.name;

};

}

};

object.site = "http://www.sina.com.cn";

const ddSite = "http://www.youku.com";

}

});

AJAX

Asp.net MVC 抛弃了Asp.net WebForm那种高度封装的控件，让我们跟底层的HTML有了更多的亲近。可以更自由、更灵活的去控制HTML的结构、样式和行为。

以下采用三种方式演示ajax的使用:

第一种：直接写JS代码实现Ajax；

第二种：使用Jquery进行Ajax调用；

<script type="text/javascript">

//execute when the DOM has been loaded

$(document).ready(function () {

//wire up to the form submit event

$("form.hijax").submit(function (event) {

event.preventDefault(); //prevent the actual form post

hijack(this, update\_sessions, "html");

});

});

function hijack(form, callback, format) {

$("#indicator").show();

$.ajax({

url: form.action,

type: form.method,

dataType: format,

data: $(form).serialize(),

completed: $("#indicator").hide(),

success: callback

});

}

function update\_sessions(result) {

//clear the form

$("form.hijax")[0].reset();

$("#comments").append(result);

}

</script>

第三种方法：使用微软自带的Ajax Helper框架来实现;

<% using (Ajax.BeginForm("AddComment", new AjaxOptions

{

HttpMethod = "POST",

UpdateTargetId = "comments",

InsertionMode = InsertionMode.InsertAfter

})) { %>

<%= Html.TextArea("Comment", new{rows=5, cols=50}) %>

<button type="submit">Add Comment</button>

<% } %>

第一、Ajax Helper是微软提供的一种Ajax框架，为了使用Ajax Helper必须使用微软提供的两个Js框架：

<script src="http://www.cnblogs.com/Scripts/MicrosoftAjax.js" type="text/javascript"></script>

<script src="http://www.cnblogs.com/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>

第二、Ajax Helper有几个用法

 Ajax.ActionLink()：它将渲染成一个超链接的标签，类似于Html.ActionLink()。当它被点击之后，将获取新的内容并将它插入到HTML页面中。

Ajax.BeginForm()：它将渲染成一个HTML的Form表单，类似于Html.BeginForm()。当它提交之后，将获取新的内容并将它插入到HTML页面中。

Ajax.RouteLink()：Ajax.RouteLink()类似于Ajax.ActionLink()。不过它可以根据任意的routing参数生成URL,不必包含调用的action。使用最多的场景是自定义的IController，里面没有action。

Ajax.BeginRouteForm()：同样Ajax.BeginRouteForm()类似于Ajax.BeginForm()。这个Ajax等同于Html.RouteLink()。

而每个方法里面的参数会有所不同，具体的用法见：[http://msdn.microsoft.com/zh-cn/library/system.web.mvc.ajaxhelper\_methods(v=VS.98).aspx](http://msdn.microsoft.com/zh-cn/library/system.web.mvc.ajaxhelper_methods%28v=VS.98%29.aspx)

其中一个重要的参数为：AjaxOption，存在有以下几个属性，主要是来规定Ajax的行为的。

一个简单的前后台交互，json

@{

ViewBag.Title = "Index";

Layout = "~/Views/Shared/\_Login.cshtml";

}

@{

if (ViewData["CheckResult"] != null)

{

<div>@ViewData["CheckResult"].ToString()</div>

}

}

<scriptsrc="@Url.Content("~/Scripts/jquery-1.10.2.min.js")"type="text/javascript"></script>

<scripttype="text/javascript">

$(document).ready(function () {

$("#btnOK").click(function(){

var mdata = "";

$.ajax({

async: false,

type: "POST",

dataType: "json",

url: "/Login/GetUsers",

data: mdata,

error: function () {

},

success: function (result) {

alert(result);

var mresult = JSON.parse(result);

alert(mresult[0].Name);

$("#display").val(mresult[0].Name + mresult[0].Message);

}

});

});

});

</script>

<div>

<inputid="btnOK"type="button"value="button"/>

<inputid="display"type="text"/>

</div>

后台LoginController，需要using Newtonsoft.Json;using Newtonsoft.Json.Converters;：

// POST api/values

[method:HttpPost]

publicActionResult Welcome()

{

string mmUserName = Request.Form["UserName"];

string mmUserPassword = Request.Form["UserPassword"];

DataTable mmDT = App.myDataAccess.GetDataTable("select \* from 基金会\_登录帐号 where 登录帐号='"+mmUserName+"' and 登录密码='"+mmUserPassword+"'");

if (mmDT.Rows.Count > 0)

{

ViewData["CheckResult"] = "欢迎 " + mmUserName + " ！";

}

else

{

ViewData["CheckResult"] = "登录帐号或密码错误！";

}

return View();

}

[HttpPost]

publicActionResult GetUsers()

{

System.Data.DataTable mmDT = newSystem.Data.DataTable();

mmDT.Columns.Add("Name");

mmDT.Columns.Add("Message");

System.Data.DataRow mmNewRow = mmDT.NewRow();

mmNewRow["Name"] = "Jimmy";

mmNewRow["Message"] = "Hello";

mmDT.Rows.Add(mmNewRow);

string mmJosn = JsonConvert.SerializeObject(mmDT);

return Json(mmJosn, JsonRequestBehavior.AllowGet);

//return Json("{\"Name\": \"Hello\", \"Message\":\"World\"}", JsonRequestBehavior.AllowGet);

}

发布

右键项目，选择“发布”，发布到文件夹，即可。

比较好的例子请参见：

http://www.cnblogs.com/liqingwen/p/6640861.html

# 三、Web API

Asp.NET Web API包含路由系统在内的核心框架是一个独立于寄宿环境的消息处理管道，这是为什么它能支持多种不同寄宿模式的根源所在。服务寄宿的目的在于提供一个运行环境以监听并接收来自外界的请求，并将请求经过适当“加工”后递交给这个抽象的消息进行处理，最后将管道处理后产生的响应消息利用网络传输返回给客户端。

WebApi的路由机制

WebApi的路由机制和MVC有许多的相似性，所以要想理解Webapi的路由机制，有需要搬出来那些asp.net Rounting里面的对象。这个过程有点复杂，博主就根据自己的理解，提提一些主要的过程：

1、WebApi服务启动之后，会执行全局配置文件Global.asax.cs的 protected void Application\_Start(){GlobalConfiguration.Configure(WebApiConfig.Register);} 方法，通过参数委托执行WebApiConfig.cs里面的 public static void Register(HttpConfiguration config) 这个方法，将所有配置的路由信息添加到 HttpRouteCollection 对象中（MVC里面可能是RouteCollection对象）保存起来。**这里的HttpRoutCollection对象的实例名是Routes，这个很重要，后面要用到**。

2、当我们发送请求到WebApi服务器的时候，比如我们访问http://localhost:21528/api/Order这个url的时候，请求首先还是会被UrlRoutingModule监听组件截获，然后，将截获的请求在Routes路由集合中匹配到对应的路由模板（如果匹配不到对应的路由模板，则返回404），得到对应的IHttpRoute对象。**IHttpRoute对象是Routes集合里面匹配到的一个实体。**

3、将IHttpRoute对象交给当前的请求的上下文对象RequestContext处理，根据IHttpRoute对象里面的url匹配到对应的controller，然后再根据http请求的类型和参数找到对应的action。这样一个请求就能找到对应的方法了。

一个请求过来之后，路由主要需要经历三个阶段

(1)根据请求的url匹配路由模板

(2)找到控制器

(3)找到action

因此，一个Web 应用具有一个全局的路由表,该路由表通过类型System.Web.RouteTable的静态只读属性Routes表示 ,该属性返回一个类型为 system.Web,RoutingRouteCoIlection的集合。我们通过调用 RouteCollection的MapPagcRoute方法将某个物理文件路径映射到—个路由模板上 ,这个过程的本质是:基于指定的路由模板创建一个 Route对象,并将其添加到这个全局路由表中。

启动依然是Startup.cs，不过由于webapi是调用关系，因此该cs文件中没有指定默认路由。

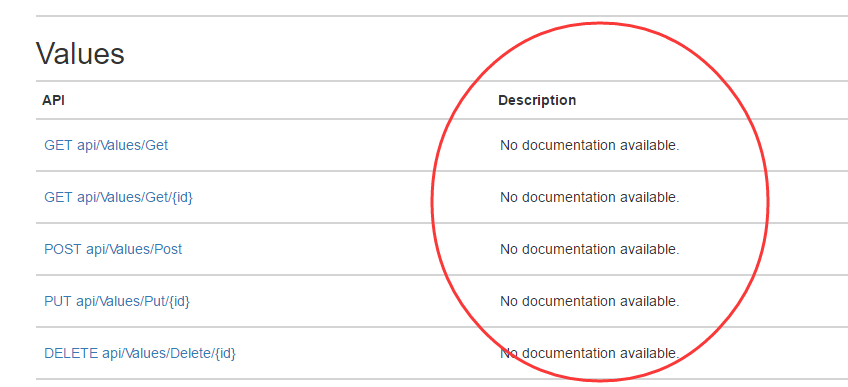
路由文件：App\_Start/WebApiConfig.cs

数据格式支持json

如需要支持json，需要在Global.asax添加：GlobalConfiguration.Configuration.Formatters.XmlFormatter.SupportedMediaTypes.Clear();

启用WebApi里的Api描述信息(Help下的Description)

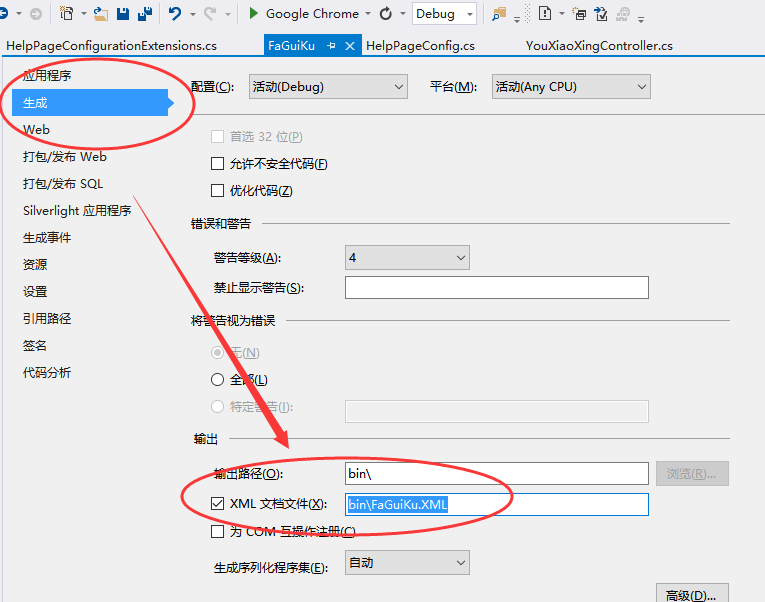
问题:默认情况下新建的Web Api 2项目,自带的Help页下会显示Api的相关信息,但Description那一栏无法获取到数据,如下图所示:



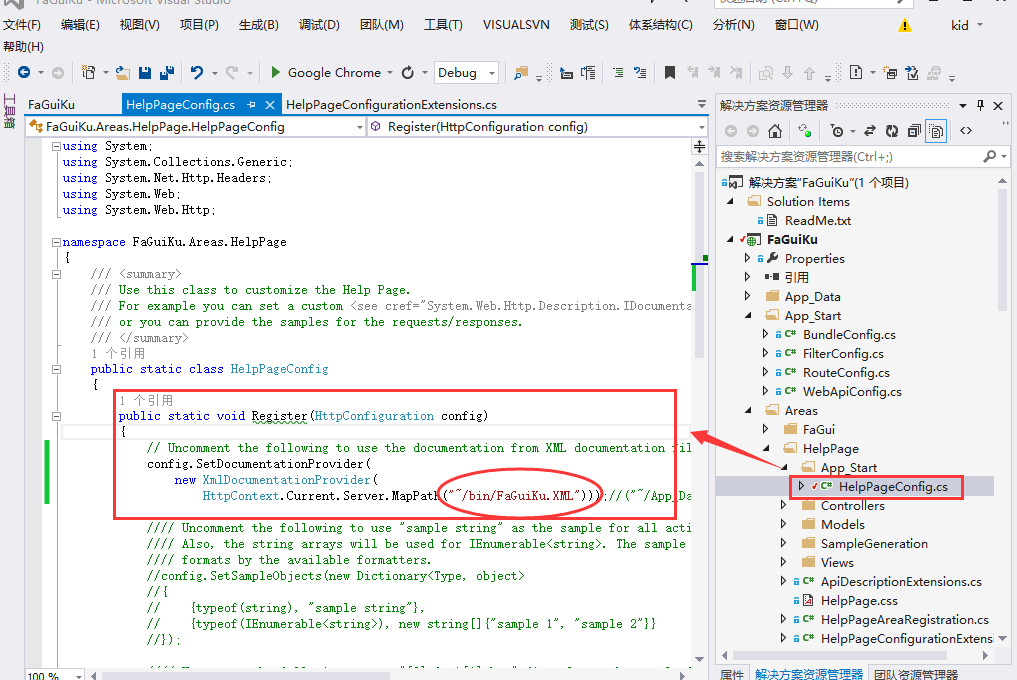
解决:

1.先启用输出的XML文档文件:

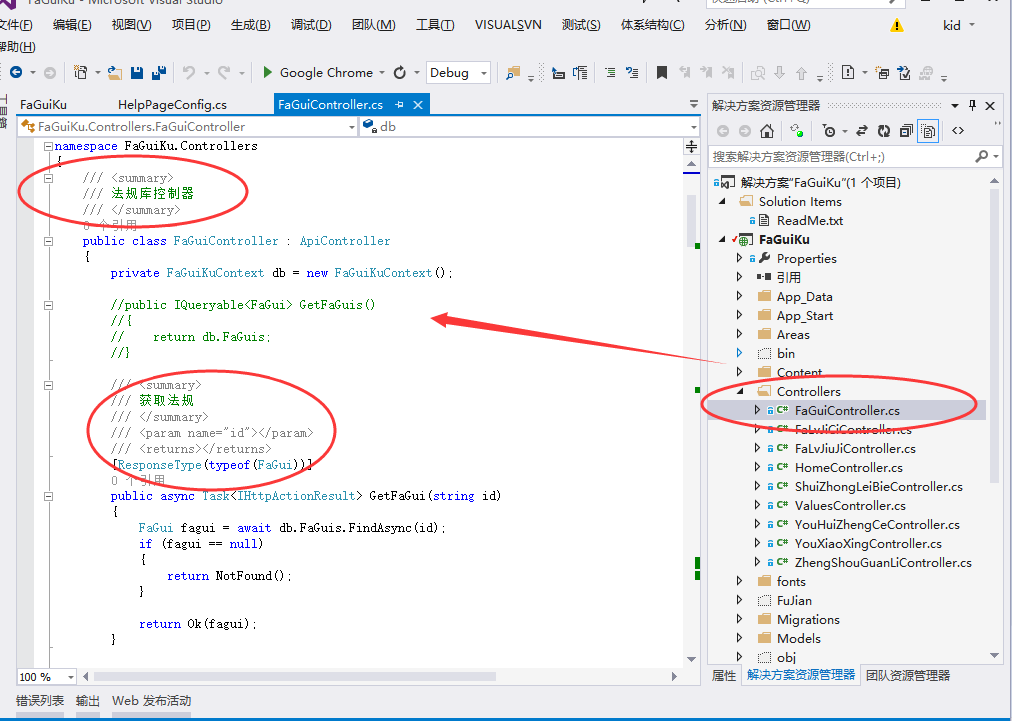
在web api项目上右击->属性,在属性页"生成"标签下,勾选输出下面的"XML 文档文件",如下图所示:



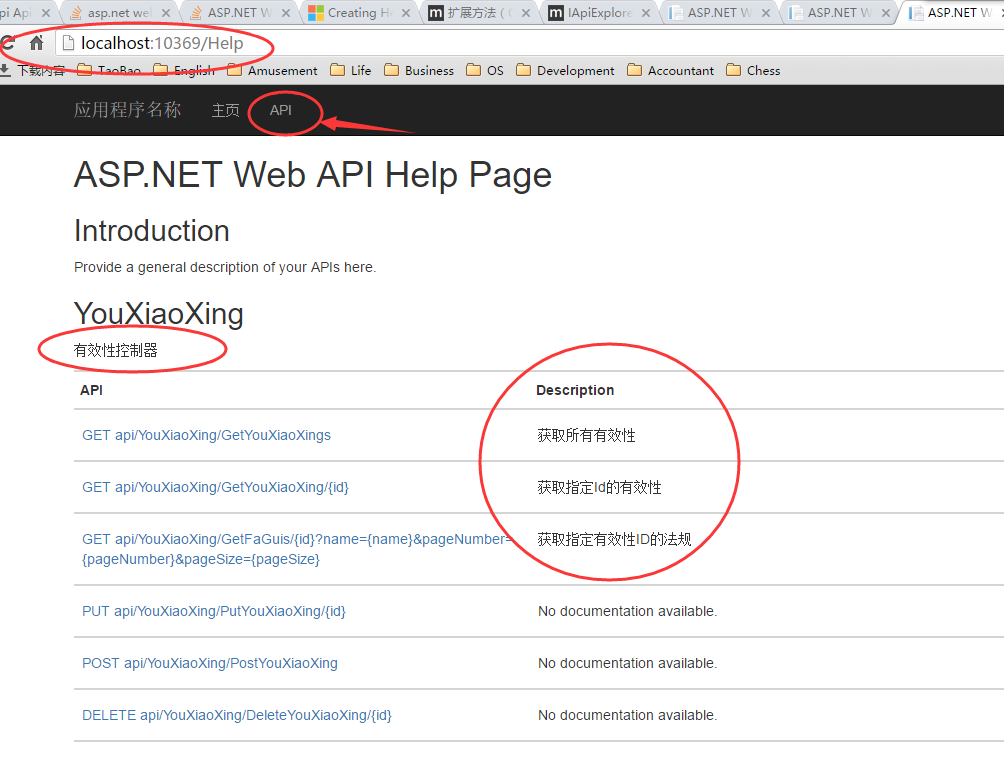
2.在"解决方案管理器"里,打HelpPageConfig.cs,去掉SetDocumentationProvider方法,并把地址改成与1里的对应,如下图所示:



3.在Api上添加注释,如下图所示:



再次运行项目,打开Api Help页面,就可以看到效果了,如下图所示:



webApi 接口帮助调试文档（Swagger UI）

Swagger能成为最受欢迎的REST APIs文档生成工具之一，有以下几个原因：

Swagger 可以生成一个具有互动性的API控制台，开发者可以用来快速学习和尝试API。

Swagger 可以生成客户端SDK代码用于各种不同的平台上的实现。

Swagger 文件可以在许多不同的平台上从代码注释中自动生成。

Swagger 有一个强大的社区，里面有许多强悍的贡献者。

Swagger 文档提供了一个方法，使我们可以用指定的 JSON 或者 YAML 摘要来描述你的 API，包括了比如 names、order 等 API 信息。

你可以通过一个文本编辑器来编辑 Swagger 文件，或者你也可以从你的代码注释中自动生成。各种工具都可以使用 Swagger 文件来生成互动的 API 文档。

创建Swagger组建

1、首先，创建WebApi的项目文件；

2、工具》Nuget包管理器》管理解决方案的Nuget程序包；

3、搜索Swagger，安装Swashbuckle和Swagger-Net、Swagger-Net-Ui包；

4、安装完之后选择项目，右键》属性》生成》输出-XML文档文件 勾选保存

5、运行项目 地址输入swagger\ui\index

我们发现swagger程序集报错

解决方案：

发生上述错误，请在，SwaggerNet类中，注释类上面的两行，就会运行成功

6、注释配置

运行项目之后，我们发现在项目写的注释并没有在swagger页面上面展示，所以我们现在开开启注释；

在SwaggerConfig类中，EnableSwagger的时候添加下面XML解析

using System.Web.Http;

using WebActivatorEx;

using DianshiStandard.Web;

using Swashbuckle.Application;

[assembly: PreApplicationStartMethod(typeof(SwaggerConfig), "Register")]

namespace DianshiStandard.Web

{

public class SwaggerConfig

{

public static void Register()

{

var thisAssembly = typeof(SwaggerConfig).Assembly;

GlobalConfiguration.Configuration

.EnableSwagger(c =>

{

c.SingleApiVersion("v1", "DianshiStandard.Web");

//添加XML解析

c.IncludeXmlComments(GetXmlCommentsPath());

})

.EnableSwaggerUi(c =>

{

});

}

//添加XML解析

private static string GetXmlCommentsPath()

{

return string.Format("{0}/bin/DianshiStandard.Web.XML", System.AppDomain.CurrentDomain.BaseDirectory);

}

}

}

注意，该路径为步骤4时添加的XML文档文件路径。

效果如地址：<http://172.16.168.31:10201/swagger/ui/index>

MVC WebAPI中的Controllers

MVC WebAPI中的Controllers和普通MVC的Controllers类似，不过不再继承于Controller，而改为继承API的ApiController，一个Controller可以包含多个Action，这些Action响应请求的方法与Global中配置的路由规则有关。

使用Attribute声明HTTP方法，如：[HttpGet]，

[NonAction]

publicstring GetPrivateData()

NoAction表示这个方法是不接收请求的，即使以GET开头。还可以使用AcceptVerbs的方式来声明HTTP方法，如：

[AcceptVerbs("MKCOL", "HEAD")]

publicint UpdateByKey(string key, string value)

{

List<TestUseMode> upDataList = allModeList.FindAll((mode) => { if (mode.ModeKey == key) returntrue; returnfalse; });

foreach(var mode in upDataList)

{

mode.ModeValue = value;

}

return upDataList.Count;

}

在WebAPI中使用两外两个类：HttpRequestMessage 和HttpResponseMessage，分别用于封装Requset和Response，获取Request和Response，分别对应MVC中的HttpRequest和HttpResponse两个类。还有一个常见的抽象类：HttpMessageHandler,用于过滤和加工HttpRequestMessage和HttpResponseMessage。

跨域

出于安全考虑，浏览器会限制脚本中发起的跨站请求，浏览器要求JavaScript或Cookie只能访问同域下的内容。

正是由于这个原因，我们不同项目之间的调用就会被浏览器阻止。比如我们最常见的场景：WebApi作为数据服务层，它是一个单独的项目，我们的MVC项目作为Web的显示层，这个时候我们的MVC里面就需要调用WebApi里面的接口取数据展现在页面上。因为我们的WebApi和MVC是两个不同的项目，所以运行起来之后就存在上面说的跨域的问题。

解决方法(已验证)：

在访问api的网站和wep api的web.config中的<system.webServer>节中都增加：

<httpProtocol>

<customHeaders>

<addname="Access-Control-Allow-Origin"value="\*"/>

</customHeaders>

</httpProtocol>

在访问web api时，使用：

$(document).ready(function () {

jQuery.support.cors = true;

$("#btnAPI").click(function () {

$.ajax({

async: false,

type: "GET",

dataType: "json",

url: "http://127.0.0.1:8010/api/values",

error: function (XMLHttpRequest, textStatus, errorThrown) {

alert(XMLHttpRequest.status);

alert(XMLHttpRequest.readyState);

alert(textStatus);

},

success: function (result) {

alert(result);

var mresult = JSON.parse(result);

alert(mresult[0].N);

$("#display").val(mresult[0].N+" " + mresult[0].V);

}

});

});

});

这样跨域的问题就解决了，支持IE8兼容模式，IE8+以及其他主流浏览器。

另外有一篇讲得比较详细的博文，里面的方法没有试过，也可以看看：

<http://www.cnblogs.com/landeanfen/p/5177176.html>

CORS全称Cross-Origin Resource Sharing，中文全称跨域资源共享。它解决跨域问题的原理是通过向http的请求报文和响应报文里面加入相应的标识告诉浏览器它能访问哪些域名的请求。比如我们向响应报文里面增加这个Access-Control-Allow-Origin:http://localhost:8081，就表示支持http://localhost:8081里面的所有请求访问系统资源。其实上面的web.config里的”Access-Control-Allow-Origin”就是这个意思。

下面还有个方法用CORS解决WebApi的跨域问题，没有验证过，先了解一下：

在WebApiCORS项目上面使用Nuget搜索“microsoft.aspnet.webapi.cors”，安装第一个



然后在App\_Start文件夹下面的WebApiConfig.cs文件夹配置跨域

publicstaticclass WebApiConfig

{

publicstaticvoid Register(HttpConfiguration config)

{

//跨域配置

config.EnableCors(new EnableCorsAttribute("\*", "\*", "\*"));

// Web API 路由

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{action}/{id}",

defaults: new { id = RouteParameter.Optional }

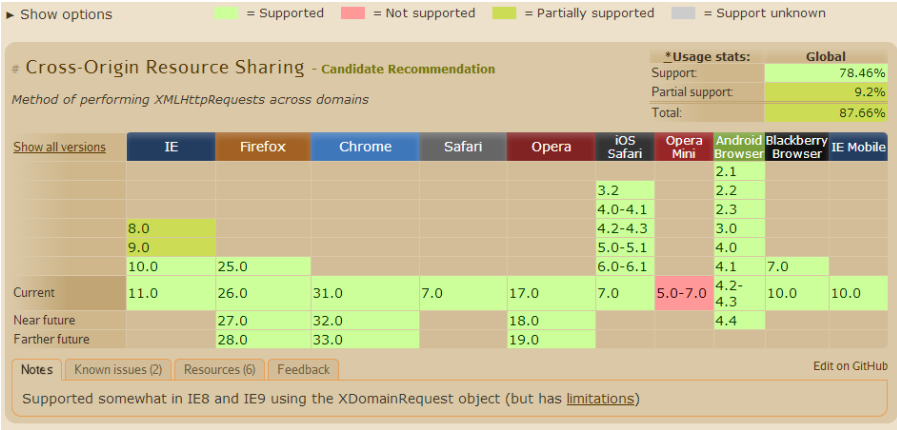
);

}

}

我们暂定三个“\*”号，当然，在项目中使用的时候一般需要指定对哪个域名可以跨域、跨域的操作有哪些等等。

下图显示了这种方法支持的浏览器：



可以看到IE8、9是部分支持的。其实和上面已验证的例子一样，在调用处指定：jQuery.support.cors = true;这一句就能解决IE8、9的问题了。如：

jQuery.support.cors = true;

var ApiUrl = "http://localhost:27221/";

$(function () {

$.ajax({

type: "get",

url: ApiUrl + "api/Charging/GetAllChargingData",

data: {},

success: function (data, status) {

if (status == "success") {

$("#div\_test").html(data);

}

},

error: function (e) {

$("#div\_test").html("Error");

},

complete: function () {

}

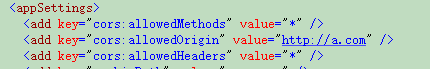
});

});

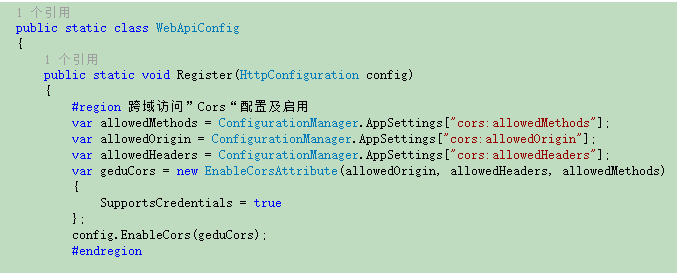
这句话的意思就是指定浏览器支持跨域。原来IE9以上版本的浏览器、谷歌、火狐等都默认支持跨域，而IE8、9却默认不支持跨域，需要我们指定一下。

上文我们使用config.EnableCors(new EnableCorsAttribute("\*", "\*", "\*"));这一句解决了跨域问题，但这种\*号是不安全的。因为它表示只要别人知道了你的请求url，任何请求都可以访问到你的资源。这是相当危险的。所以需要我们做一些配置，限制访问权限。比如我们比较常见的做法如下：

配置方法一、在Web.Config里面



然后在WebApiConfig.cs文件的Register方法里面



配置方法二、如果你只想对某一些api做跨域，可以直接在API的类上面使用特性标注即可。

[EnableCors(origins: "http://localhost:8081/", headers: "\*", methods: "GET,POST,PUT,DELETE")]

publicclass ChargingController : ApiController

{

///<summary>

///得到所有数据

///</summary>

///<returns>返回数据</returns>

[HttpGet]

publicstring GetAllChargingData()

{

return"Success";

}

}

具体可见<http://www.cnblogs.com/landeanfen/p/5177176.html>

过滤器

Web API的简单流程就是从请求到执行到Action并最终作出响应，但是在这个过程有一把“筛子”，那就是过滤器Filter，在从请求到Action这整个流程中使用Filter来进行相应的处理从而作出响应，这对于认证以及授权等是及其重要的，所以说过滤器应用是Web API框架中非常重要的一种实现方式，我们有必要去探讨其原理。

一个Controller中多个GET/ POST/ PUT/ DELETE

**一个Controller中多个GET/ POST/ PUT/ DELETE：**需要修改路由规则。在App\_Start/WebApiConfig.cs中修改：

// Web API 路由

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

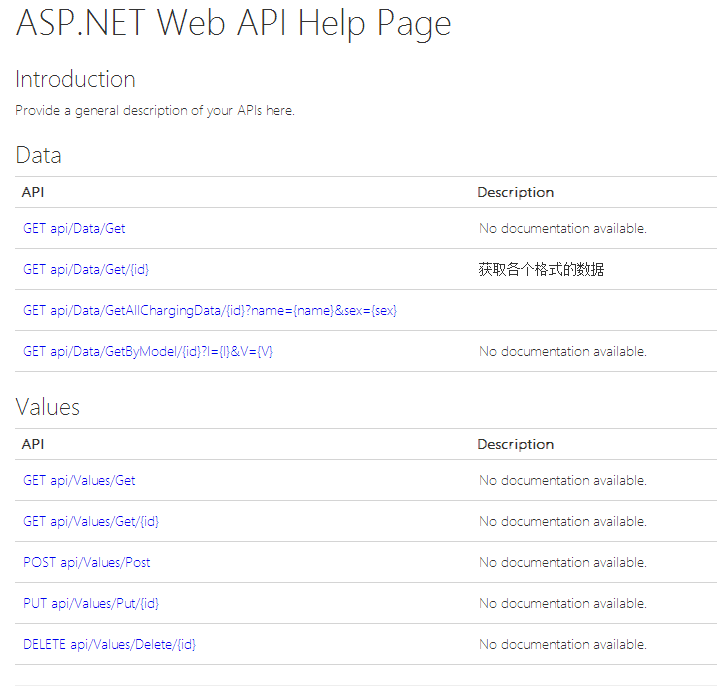
name: "DefaultApi",

routeTemplate: "api/{controller}/{action}/{id}",

defaults: new { id = RouteParameter.Optional }

);

这个时候Help显示如下：



具体的调用如：

默认的GET:<http://localhost:18327/api/data/get>

默认的GET(string id):<http://localhost:18327/api/data/get/5>

上面的例子可以实现，但不一定符合RestFul风格，而从MVC到WebApi，路由机制一直是伴随着这些技术的一个重要组成部分。下面就详细讲一讲WebApi的路由机制。

新建一个WebApi服务的时候，会自动在WebApiConfig.cs文件里面生成一个默认路由：

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

分别来看看各个参数的作用：

* name:"DefaultApi"→表示此路由的名称，这里只需要保证路由名称不重复就OK了。
* routeTemplate: "api/{controller}/{id}"→表示路由的url规则，“api”是固定部分，主要用来标识当前请求的url是一个api服务的接口，区别MVC的路由，当然，这里并不是一定要写成“api”，如果你改成“apiserver”，那么你请求的url里面也需要写成“apiserver”；“{controller}”是控制器的占位符部分，在真实的url里面，该部分对应的是具体的控制器的名称，这个和MVC里面一致；“{id}”是参数的占位符部分，表示参数，一般这个参数都会在default里面设置可选。有了这个路由模板约束请求的url，比如：我们请求的url写成http://localhost:21528/Order，那么肯定是找不到对应的路由的，因为“api”这个参数必选。如果请求的url匹配不到对应的路由，则会向客户端返回一个404的状态码。
* defaults: new { id = RouteParameter.Optional }→表示路由的默认值，比如上面的routeTemplate，{controller}和{id}部分都可以设置默认值，比如：defaults改成new { controller="Order", id = RouteParameter.Optional }，那么我们请求http://localhost:21528/api这个url仍然能访问到GetAll()方法。
* constraints→表示路由约束，一般是一个约束路由模板的正则表达式。比如：我们加入约束条件 constraints: new { id = @"\d+" } ，这就约束必须要匹配一到多个参数id，那么，我们在OrderController里面加入另一个方法

public class OrderController : ApiController

{

[HttpGet]

public object GetAll()

{

return "Success";

}

[HttpGet]

public object GetById(int id)

{

return "Success" + id ;

}

}

我们通过http://localhost:21528/api/Order/2来访问，得到结果，我们再通过<http://localhost:21528/api/Order/a来访问，得到结果404>找不到，而我们访问<http://localhost:21528/api/Order>，竟然连GetAll()方法都找不到了。这是为什么呢？原来就是这个约束在作怪，正则\d+表示匹配一个或多个数字，所以如果请求的url里面没有传数字，则自动匹配不到。所以，如果需要匹配无参的方法，我们把约束改成这样： constraints: new { id = @"\d\*" } ，这个表示匹配0个或多个数字，访问<http://localhost:21528/api/Order>就正常了。

以上是关于默认路由原理的介绍。除了默认路由，我们也可以自定义路由，我们将WebApiConfig.cs里面改成这样：

publicstaticclass WebApiConfig

{

publicstaticvoid Register(HttpConfiguration config)

{

// Web API 路由

config.MapHttpAttributeRoutes();

//1.默认路由

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

//2.自定义路由一：匹配到action

config.Routes.MapHttpRoute(

name: "ActionApi",

routeTemplate: "actionapi/{controller}/{action}/{id}",

defaults: new { id = RouteParameter.Optional }

);

//3.自定义路由二

config.Routes.MapHttpRoute(

name: "TestApi",

routeTemplate: "testapi/{controller}/{ordertype}/{id}",

defaults: new { ordertype="aa", id = RouteParameter.Optional }

);

}

}

除了默认路由，我们加入了另外两个自定义路由规则。

通过action的名称来匹配很好理解，上面的GetAll()是方法名，webApi会默认它就是action的名称，如果你想要方法名和action的名称不一致，你也可以自定义action的名称，这个可以通过特性ActionName来实现，如下：

[ActionName("TestActionName")]

[HttpGet]

publicobject GetById(int id)

{

return"Success" + id ;

}

由于WebApi的路由机制和MVC有许多的相似性，所以要想理解Webapi的路由机制，有需要搬出来那些asp.net Rounting里面的对象。这个过程有点复杂，博主就根据自己的理解，提提一些主要的过程：

1、WebApi服务启动之后，会执行全局配置文件Global.asax.cs的 protected void Application\_Start(){GlobalConfiguration.Configure(WebApiConfig.Register);} 方法，通过参数委托执行WebApiConfig.cs里面的 public static void Register(HttpConfiguration config) 这个方法，将所有配置的路由信息添加到 HttpRouteCollection 对象中（MVC里面可能是RouteCollection对象）保存起来。**这里的HttpRoutCollection对象的实例名是Routes，这个很重要，后面要用到**。

2、当我们发送请求到WebApi服务器的时候，比如我们访问http://localhost:21528/api/Order这个url的时候，请求首先还是会被UrlRoutingModule监听组件截获，然后，将截获的请求在Routes路由集合中匹配到对应的路由模板（如果匹配不到对应的路由模板，则返回404），得到对应的IHttpRoute对象。**IHttpRoute对象是Routes集合里面匹配到的一个实体。**

3、将IHttpRoute对象交给当前的请求的上下文对象RequestContext处理，根据IHttpRoute对象里面的url匹配到对应的controller，然后再根据http请求的类型和参数找到对应的action。这样一个请求就能找到对应的方法了。

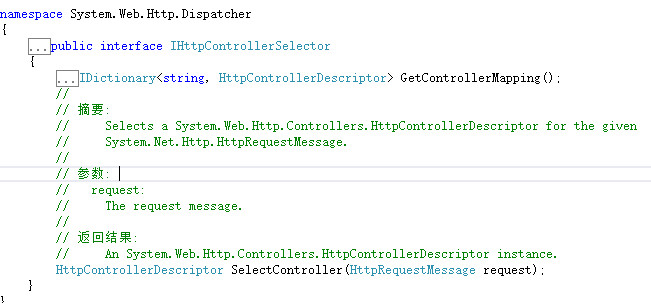
一个请求过来之后，路由主要需要经历三个阶段

(1)根据请求的url匹配路由模板

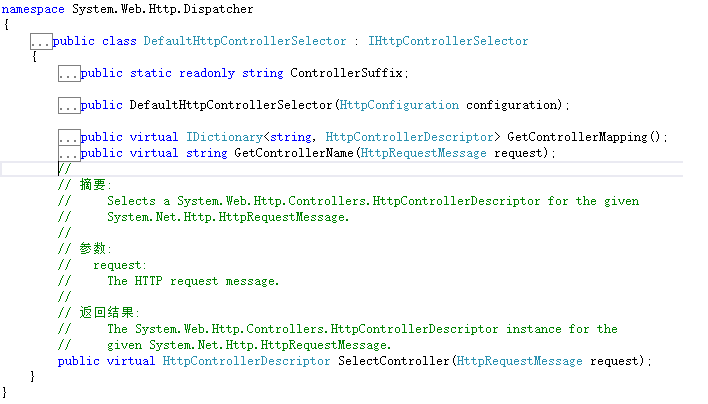
(2)找到控制器

(3)找到action

控制器的选择主要在IHttpControllerSelector这个接口的SelectController()方法里面处理。



该方法将当前的请求以HttpRequestMessage对象作为参数传入，返回HttpControllerDescriptor对象。这个接口默认由DefaultHttpControllerSelector这个类提供实现。



默认实现的方法里面大致的算法机制是：首先在路由字典中找到实际的控制器的名称（比如“Order”），然后在此控制器名称上面加上字符串“Controller”的到请求控制器的全称（比如“OrderController”），最后找到对应的WebApi的Controller，实例化就得到当前请求的控制器对象。

得到了控制器对象之后，Api引擎通过调用IHttpActionSelector这个接口的SelectAction()方法去匹配action。这个过程主要包括：

* 解析当前的http请求，得到请求类型（是get、post、put还是delete）
* 如果路由模板配置了{action}，则直接取到url里面的action名称
* 解析请求的参数

如果路由模板配置了{action}，那么找到对应的action就很简单，如果没有配置action，则会首先匹配请求类型（get/post/put/delete等），然后匹配请求参数，找到对应的action。

WebApi还提供了一个action同时支持多个http方法的请求，使用AcceptVerbs特性去标记。

[AcceptVerbs("GET", "POST")]

public IHttpActionResult GetById(int id)

{

return Ok<string>("Success" + id );

}

这个过程本身是非常复杂的，为了简化，我们只选择了最主要的几个过程。更详细的路由机制可以参考(“技术”-“C#”-“应用”-“WebAPI”中的《ASP.NET Routing路由（深入解析路由系统架构原理）》)：[http://www.cnblogs.com/wangiqngpei557/p/3379095.html](http://www.cnblogs.com/wangiqngpei557/p/3379095.html" \t "_blank)。这文章写得有点深，有兴趣的可以看看。

这时，我们再回到之前路由的第一个问题，如果http请求的方法相同（比如都是post请求），并且请求的参数也相同。这个时候似乎就有点不太好办了，这种情况在实际项目中还是比较多的。比如：

publicclass OrderController : ApiController

{

//订单排产

[HttpPost]

publicvoid OrderProduct([FromBody]string strPostData)

{

}

//订单取消

[HttpPost]

publicvoid OrderCancel([FromBody]string strPostData)

{

}

//订单删除

[HttpPost]

publicvoid OrderDelete([FromBody]string strPostData)

{

}

}

这个时候如果使用我们上面讲的Restful风格的路由是解决不了这个问题的。当然，有园友可能就说了，既然这样，我们在路由模板里面加上“{action}”不就搞定了么！这样确实可行。但还是那句话，不提倡。我们来看看如何使用特性路由解决这个问题。

如果要使用特性路由，首先在WebApiConfig.cs的Register方法里面必须先启用特性路由：

publicstaticvoid Register(HttpConfiguration config)

{

// 启用Web API特性路由

config.MapHttpAttributeRoutes();

//1.默认路由

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

}

一般情况下，当我们新建一个WebApi项目的时候，会自动在Register方法里面加上这句话。

我们在OrderController这个控制器里面加这个action

[Route("Order/SaveData")]

[HttpPost]

public HttpResponseMessage SavaData(ORDER order)

{

return Request.CreateResponse();

}

如例子IndependWebApi中的DataController：

publicstring Get(){……}

publicIHttpActionResult Get(string id){……}

[Route("freeapi/Data/GetAllChargingData")]

[HttpGet]

publicstring GetAllChargingData(int id,string name,string sex)

{……}

[Route("freeapi/Data/GetByModelToDataTable")]

[HttpGet]

publicIHttpActionResult GetByModelToDataTable([FromUri]Item item)

{……}

我们通过<http://localhost:18327/api/data>可以访问Get()方法。

<http://localhost:18327/freeapi/Data/GetAllChargingData?id=1&name=n&sex=s>可以访问GetAllChargingData(int id,string name,string sex)方法。

大小写是不区分的，<http://localhost:18327/freeapi/data/getallchargingdata?id=1&name=n&sex=s>也可以访问。

WebAPI安全验证的问题(顺便描述了web后台验证)

我们知道WEB网站的身份验证一般通过session或者cookie完成的，登录成功后客户端发送的任何请求都带上cookie，服务端根据客户端发送来的cookie来识别用户。

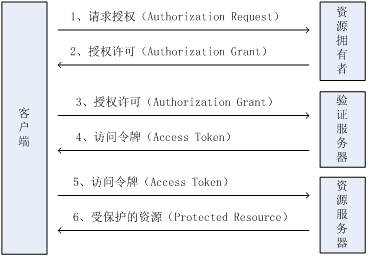
因为WebAPI是个轻量级的数据服务，如果没有加任何验证，任何人只要知道了接口的url，都能够模拟http请求去访问我们的服务接口，从而去增删改查数据库。

对于网站来说，有几种常用的验证方式。

HTTP Basic Auth简单点说明就是每次请求API时都提供用户的username和password，简言之，Basic Auth是配合RESTful API 使用的最简单的认证方式，只需提供用户名密码即可，但由于有把用户名密码暴露给第三方客户端的风险，在生产环境下被使用的越来越少。因此，在开发对外开放的RESTful API时，尽量避免采用HTTP Basic Auth。

OAuth（开放授权）是一个开放的授权标准，允许用户让第三方应用访问该用户在某一web服务上存储的私密的资源（如照片，视频，联系人列表），而无需将用户名和密码提供给第三方应用。

OAuth允许用户提供一个令牌，而不是用户名和密码来访问他们存放在特定服务提供者的数据。每一个令牌授权一个特定的第三方系统（例如，视频编辑网站)在特定的时段（例如，接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。这样，OAuth让用户可以授权第三方网站访问他们存储在另外服务提供者的某些特定信息，而非所有内容。下面是OAuth2.0的流程：



Cookie认证机制由于HTTP协议是无状态的，但对于认证来说，必然要通过一种机制来保存用户状态，而最常用，也最简单的就是Cookie了，它由浏览器自动保存并在发送请求时自动附加到请求头中。尽管在现代Web应用中，Cookie已略显笨重，但它依然是最为重要的用户身份保存方式。Cookie认证机制就是为一次请求认证在服务端创建一个Session对象，同时在客户端的浏览器端创建了一个Cookie对象；通过客户端带上来Cookie对象来与服务器端的session对象匹配来实现状态管理的。默认的，当我们关闭浏览器的时候，cookie会被删除。但可以通过修改cookie 的expire time使cookie在一定时间内有效。

服务的测试工具

soapUI:

由于 Web 服务是被程序调用的， 一般不会提供界面让最终用户或测试人员直接使用，在 soapUI 等工具出现之前，测试人员不得不自己编写程序来测试它， 这就要求测试人员花费很大的精力了解底层的接口，调用关系和详细的协议，导致他们不能把注意力集中到测试中。

soapUI 的出现极大的改变了这一局面。 作为一个开源的工具，soapUI 强大的功能、易用的界面，吸引了很多用户。用户可以在 soapUI 中通过简单的操作完成复杂的测试，不需要了解底层的细节， 极大的减轻了工作量。soapUI 支持多样的测试， 例如功能测试，性能测试，回归测试等。到目前为止 soapUI 的下载量已经超过了 100 万次，成为了事实的 Web 服务测试标准和领先的 Web 服务测试工具。 它不仅仅可以测试基于 SOAP 的 Web 服务，也可以测试 REST 风格的 Web 服务，后者也是本文介绍的重点。

Postman:

Postman 提供功能强大的 Web API 和 HTTP 请求的调试，它能够发送任何类型的HTTP 请求 (GET, POST, PUT, DELETE…)，并且能附带任何数量的参数和 Headers。不仅如此，它还提供测试数据和环境配置数据的导入导出，付费的 Post Cloud 用户还能够创建自己的 Team Library 用来团队协作式的测试，并能够将自己的测试收藏夹和用例数据分享给团队。

[WebApiTestClient](http://www.cnblogs.com/landeanfen/p/5210356.html):

它对于接口的作用、参数作用以及类型、返回值说明等等都能很好展现，并且提供测试调用的功能，输入测试参数发送请求，就能得到相应的返回结果。

WebApiTestClient组件作用主要有以下几个：

（1）将WebApi的接口放到了浏览器里面，以可视化的方式展现出来，比如我们通过http://localhost:8080/Help这个地址就能在浏览器里面看到这个服务里面所有的API接口以及接口的详细说明，省去了我们手写接口文档的麻烦。

（2）能够详细查看API的类说明、方法说明、参数说明、返回值说明。只需要我们在定义方法时候加上 /// <summary> 这种详细注释即可，组件自动读取注释里面的内容展现在界面上面。

（3）可以修改http请求头文件Head和请求体Body里面的参数，指定发送http请求的特性，比如指定我们最常见的contentType指示参数的类型。

（4）组件拥有测试接口的功能，用过Soup UI的朋友应该知道，通过Soup UI能够方便测试WebService参数以及返回值。我们的WebApiTestClient也可以实现类似的功能，直接通过页面上的测试按钮，就能测试接口。

WebApiTestClient是一个开源组件。开源地址：[https://github.com/yaohuang/WebApiTestClient](https://github.com/yaohuang/WebApiTestClient" \t "_blank)

具体的操作可参见（配置过，可以用）：<http://www.cnblogs.com/landeanfen/p/5210356.html>

WebAPI返回值的问题

a.首先，可以返回void类型。

b.也可以返回string类型，如下：

publicstring Get()

{

DataTable mmDT = newDataTable();

mmDT.Columns.Add("N", typeof(string));

mmDT.Columns.Add("V", typeof(decimal));

mmDT.Columns.Add("B", typeof(string));

DataRow mmDR = mmDT.NewRow();

mmDR["N"] = "Tom";

mmDR["V"] = "1.02";

mmDR["B"] = "like books!";

mmDT.Rows.Add(mmDR);

returnJsonConvert.SerializeObject(mmDT,newDataTableConverter());

}

返回的值前后会多出””,如："[{\"N\":\"Tom\",\"V\":1.02,\"B\":\"like books!\"}]"，js端JSON.parse没有问题。但是C#里反序列化就会有问题，提示对象无法转换。

c.这个时候需要IHttpActionResult类型登场了，IHttpActionResult类型是WebApi里面非常重要的一种返回值类型：

a.Json<T>(T content)

使用MVC开发过的朋友一定记得，在MVC里面，请求数据的接口的返回值类型大部分使用的是JsonResult，在MVC里面你一定也写过类似这样的接口：

public JsonResult GetResult()

{

return Json(new { }, JsonRequestBehavior.AllowGet);

}

那么，在WebAPI里面是否也存在类似的用法呢。呵呵，在这点上面，微软总是贴心的。在WebApi的ApiController这个抽象类里面，为我们封装了Json<T>(T content)这个方法，它的用法和MVC里面的JsonResult基本类似。而JsonResult<T>是实现了IHttpActionResult接口的，因此我们可以返回如下的值：

// GET api/data/5

publicIHttpActionResult Get(string id)

{

switch (id)

{

case"entity":

{

Item mmItem = newItem{ I = "1", V = "China" };

return Json<Item>(mmItem);

}

case"list":

{

List<Item> mmList = newList<Item>

{

newItem { I = "1", V = "China" },

newItem { I = "2", V = "America" },

newItem { I = "3", V = "Japan" },

newItem { I = "4", V = "Italy" },

newItem { I = "5", V = "France" }

};

return Json<List<Item>>(mmList);

}

case"table":

{

DataTable mmDT = newDataTable();

mmDT.Columns.Add("N", typeof(string));

mmDT.Columns.Add("V", typeof(decimal));

mmDT.Columns.Add("B", typeof(string));

DataRow mmDR = mmDT.NewRow();

mmDR["N"] = "Tom";

mmDR["V"] = "1.02";

mmDR["B"] = "like books!";

mmDT.Rows.Add(mmDR);

return Json<DataTable>(mmDT);

}

default:

{

//return Json<string>(id.ToString());

return Ok<string>(id.ToString());

}

}

}

这样在客户端，无论是web还是winform下C#中的JsonConvert.DeserializeObject都能正常解析了。

Ok<T>(T content)和Json<T>(T content)比较类似，都可以用，如果你非要问这两者有什么区别，或者说怎么选择两者。那么我的理解是如果是返回实体或者实体集合，建议使用Json<T>(T content)，如果是返回基础类型（如int、string等），使用Ok<T>(T content)。

附(winform下和WebAPI的连接，DataTable无法直接解析，因此没有列出)：

privatevoid btn简单取值\_Click(object sender, EventArgs e)

{

Uri \_baseAddress = newUri("http://localhost:8010/");

Uri \_address = newUri(\_baseAddress, "/api/values/5");

using (WebClient webClient = newWebClient())

{

webClient.Headers["Accept"] = "application/json";

webClient.Encoding = Encoding.UTF8;

webClient.DownloadStringCompleted += WebClient\_DownloadStringCompleted;

webClient.DownloadStringAsync(\_address);

}

}

privatevoid btn简单获取对象\_Click(object sender, EventArgs e)

{

Uri \_baseAddress = newUri("http://localhost:8010/");

Uri \_address = newUri(\_baseAddress, "/api/data/entity");

using (WebClient webClient = newWebClient())

{

webClient.Headers["Accept"] = "application/json";

webClient.Encoding = Encoding.UTF8;

webClient.DownloadStringCompleted += WebClient\_DownloadStringCompleted\_Entity;

webClient.DownloadStringAsync(\_address);

}

}

privatevoid btn简单获取列表\_Click(object sender, EventArgs e)

{

Uri \_baseAddress = newUri("http://localhost:8010/");

Uri \_address = newUri(\_baseAddress, "/api/data/list");

using (WebClient webClient = newWebClient())

{

webClient.Headers["Accept"] = "application/json";

webClient.Encoding = Encoding.UTF8;

webClient.DownloadStringCompleted += WebClient\_DownloadStringCompleted\_List;

webClient.DownloadStringAsync(\_address);

}

}

privatevoid WebClient\_DownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e)

{

ccMsg.Clear();

if (e.Result != null)

{

string mmValue = JsonConvert.DeserializeObject<string>(e.Result);

ccMsg.Text = mmValue;

}

else

{

MessageBox.Show(e.Error.Message);

}

}

privatevoid WebClient\_DownloadStringCompleted\_Entity(object sender, DownloadStringCompletedEventArgs e)

{

ccMsg.Clear();

if (e.Result != null)

{

Item mmItem = JsonConvert.DeserializeObject<Item>(e.Result);

if (mmItem!=null)

{

ccMsg.Text += mmItem.I.ToString() + ":" + mmItem.V.ToString() + Environment.NewLine;

}

}

else

{

MessageBox.Show(e.Error.Message);

}

}

privatevoid WebClient\_DownloadStringCompleted\_List(object sender, DownloadStringCompletedEventArgs e)

{

ccMsg.Clear();

if (e.Result != null)

{

List<Item> mmList = JsonConvert.DeserializeObject<List<Item>>(e.Result);

if (mmList.Count > 0)

{

for (var i = 0; i < mmList.Count; i++)

{

ccMsg.Text += (i + 1) + "、" + mmList[i].I.ToString() + ":" + mmList[i].V.ToString() + Environment.NewLine;

}

}

}

else

{

MessageBox.Show(e.Error.Message);

}

}

当需要向客户端返回找不到记录时，有时需要用到NotFound()方法。它也实现了IHttpActionResult接口。

if (oFind == null)

{

return NotFound();

}

else

{

return Json<ORDER>(oFind);

}

NotFound()方法会返回一个404的错误到客户端。

对于IHttpActionResult接口的返回值来说，还有其他返回的类型：

[HttpGet]

public IHttpActionResult GetContentResult()

{

return Content<string>(HttpStatusCode.OK, "OK");

}

Content<T>向客户端返回值和http状态码。

[HttpGet]

public IHttpActionResult GetBadRequest(ORDER order)

{

if (string.IsNullOrEmpty(order.ID))

return BadRequest();

return Ok();

}

BadRequest()向客户端返回400的http错误。

[HttpGet]

public IHttpActionResult RedirectResult()

{

return Redirect("http://localhost:21528/api/Order/GetContentResult");

}

IHttpActionResult将请求重定向到其他地方。

d.HttpResponseMessage对象。它表示向客户端返回一个http响应的消息对象（包含http状态码和需要返回客户端的消息）。这个对象也有它独特的使用场景：需要向客户端返回HttpResponse时就要用到这个对象。以导出为例，由于需要将导出的Excel文件输出到客户端浏览器，Webapi的服务端需要向Web的客户端输出文件流，这个时候一般的IHttpActionResult对象不方便解决这个问题，于是HttpReponseMessage派上了用场。

将文件流保存在StreamContent对象里面，然后输出到浏览器。在浏览器端即可将Excel输出。

e.以上几种返回值类型能解决我们大部分返回值的问题，当然，你也可以将webapi的接口和普通方法一样，返回任意的类型，**WebApi会自动序列化你自定义任何返回类型，然后将序列化的值写到响应正文里，状态码统一返回200**。比如：

[HttpGet]

publicobject GetOther()

{

var lstRes = new List<ORDER>();

//实际项目中，通过后台取到集合赋值给lstRes变量。这里只是测试。

lstRes.Add(new ORDER() { ID = "aaaa", NO = "111", NAME = "111", DESC = "1111" });

lstRes.Add(new ORDER() { ID = "bbbb", NO = "222", NAME = "222", DESC = "2222" });

return lstRes;

}

和上面的Json、Ok等用法在效果上面没有太大区别。

以上方式不能说哪种最好，因为每种方式都有其特定的使用场景。为了规范WebApi接口，对于一般接口的返回值，尽量使用IHttpActionResult类型作为返回值，毕竟是微软内置的东西，可能为我们考虑了很多我们考虑不到的东西。

详见《[C#进阶系列——WebApi 接口返回值不困惑：返回值类型详解](http://www.cnblogs.com/landeanfen/p/5501487.html)》<http://www.cnblogs.com/landeanfen/p/5501487.html>

这套WebApi进阶教程还是很不错的，记事宝里也下载了一份。

参数传递

[HttpGet]

publicstring GetAllChargingData(int id, string name)

{

return"ChargingData" + id;

}

$.ajax({

type: "get",

url: "http://localhost:27221/api/Charging/GetAllChargingData",

data: { id: 1, name: "Jim", bir: "1988-09-11"},

success: function (data, status) {

if (status == "success") {

$("#div\_test").html(data);

}

}

});

这是get请求最基础的参数传递方式，没什么特别好说的。

如果我们在get请求时想将实体对象做参数直接传递到后台，该如何去做呢？

get请求的时候，默认是将参数全部放到了url里面直接以string的形式传递的，后台自然接不到了。

Get请求的时候可以在参数里面加上[FromUri]即可直接得到对象。

[Route("freeapi/Data/GetByModelToItem")]

[HttpGet]

publicIHttpActionResult GetByModelToItem([FromUri]Item item)

{

return Json<Item> (newItem { I= "这是I的值" + item.I, V= "这是V的值" + item.V });

}

$("#btnGet参数是object").click(function () {

var postdata = { I: "Jim", V: "JV" };

$.ajax({

async: false,

type: "GET",

dataType: "text",

url: "http://127.0.0.1:8010/freeapi/data/GetByModelToItem",

data: postdata,

error: function (XMLHttpRequest, textStatus, errorThrown) {

alert(XMLHttpRequest.status);

alert(XMLHttpRequest.readyState);

alert(textStatus);

},

success: function (result) {

alert(result);

var mresult = JSON.parse(result);

alert(mresult);

$("#display").val(mresult.I + " " + mresult.V);

}

});

});

此处不知为何要用dataType: "text"？

如果你不想使用[FromUri]这些在参数里面加特性的这种“怪异”写法，也可以采用先序列化，再在后台反序列的方式：

[Route("freeapi/Data/GetModelBySerializeObject")]

[HttpGet]

publicIHttpActionResult GetModelBySerializeObject(string stringQuery)

{

Item mmItem = Newtonsoft.Json.JsonConvert.DeserializeObject<Item>(stringQuery);

return Json<Item>(newItem { I = "这是I的值" + mmItem.I, V = "这是V的值" + mmItem.V });

}

$("#btnGet参数序列化").click(function () {

var postdata = { I: "Jim ", V: "备注" };

$.ajax({

async: false,

type: "GET",

dataType: "text",

//contentType: "application/json",

url: "http://127.0.0.1:8010/freeapi/data/GetModelBySerializeObject",

data: { stringQuery: JSON.stringify({ I: "Jim ", V: "备注" }) },

error: function (XMLHttpRequest, textStatus, errorThrown) {

alert(XMLHttpRequest.status);

alert(XMLHttpRequest.readyState);

alert(textStatus);

},

success: function (result) {

alert(result);

var mresult = JSON.parse(result);

alert(mresult);

$("#display").val(mresult.I + " " + mresult.V);

}

});

});

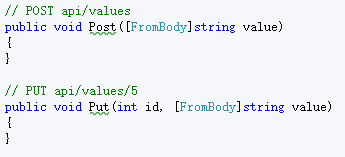
此处不知为何要用dataType: "text"？

中文的值在API端也可以识别。

方法名以Get开头，WebApi会自动默认这个请求就是get请求，而如果你以其他名称开头而又不标注方法的请求方式，那么这个时候服务器虽然找到了这个方法，但是由于请求方式不确定，所以直接返回给你405——方法不被允许的错误。

最后结论：所有的WebApi方法最好是加上请求的方式（[HttpGet]/[HttpPost]/[HttpPut]/[HttpDelete]），不要偷懒，这样既能防止类似的错误，也有利于方法的维护，别人一看就知道这个方法是什么请求。

POST如何传参呢？默认的POST方法：



$.ajax({

type: "post",

url: "http://localhost:27221/api/Charging/SaveData",

data: { "": "Jim" },

success: function (data, status) {}

});

[HttpPost]

publicbool SaveData([FromBody]string NAME)

{

returntrue;

}

我们一般的通过url取参数的机制是键值对，即某一个key等于某一个value，而这里的FromBody和我们一般通过url取参数的机制则不同，它的机制是=value，没有key的概念，并且如果你写了key(比如你的ajax参数写的{NAME:"Jim"})，后台反而得到的NAME等于null。

下面是验证过的写法：

[Route("freeapi/Data/GetPostDataByFromBody")]

[HttpPost]

publicIHttpActionResult GetPostDataByFromBody([FromBody] string name)

{

return Json<Item>(newItem { I = "这是I的值", V = "这是V的值" + name });

}

$("#btnPost参数FromBody").click(function () {

$.ajax({

async: false,

type: "post",

dataType: "text",

url: "http://127.0.0.1:8010/freeapi/data/GetPostDataByFromBody",

contentType: 'application/x-www-form-urlencoded',

data: {"":"孙刘"},

error: function (XMLHttpRequest, textStatus, errorThrown) {

alert(XMLHttpRequest.status);

alert(XMLHttpRequest.readyState);

alert(textStatus);

},

success: function (result) {

var mresult = JSON.parse(result);

alert(mresult.I); //该api返回的是Item对象，不是列表或DataTable，因此alert(mresult[0].I);是会出错的。

$("#display").val(mresult.I + " " + mresult.V);

}

});

});

contentType: 'application/x-www-form-urlencoded'这句一定要加上。

如果你不想使用[FromUri]这些在参数里面加特性的这种“怪异”写法，也可以采用先序列化，再在后台反序列的方式：

待验证。

上面我们通过dynamic类型解决了post请求基础类型数据的传递问题，那么当我们需要传递一个实体作为参数该怎么解决呢？

[Route("freeapi/Data/GetPostDataByItem")]

[HttpPost]

publicIHttpActionResult GetPostDataByItem(Item item)

{

return Json<Item>(newItem {I= "这是I的值" + item.I, V= "这是V的值" + item.V });

}

调用时：

$("#btnPost参数ByItem").click(function () {

$.ajax({

async: false,

type: "post",

dataType: "text",

url: "http://127.0.0.1:8010/freeapi/data/GetPostDataByItem",

//contentType: 'application/json', 必须去掉，否则会出错

data: { I: "Jim", V: "备注" },

error: function (XMLHttpRequest, textStatus, errorThrown) {

alert(XMLHttpRequest.status);

alert(XMLHttpRequest.readyState);

alert(textStatus);

},

success: function (result) {

var mresult = JSON.parse(result);

alert(mresult.I); //该api返回的是Item对象，不是列表或DataTable，因此alert(mresult[0].I);是会出错的。

$("#display").val(mresult.I + " " + mresult.V);

}

});

});

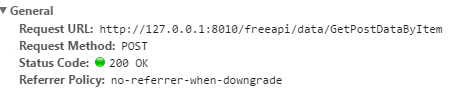
使用实体作为参数的时候，前端直接传递普通json，后台直接使用对应的类型去接收即可，不用FromBody。但是这里需要注意的一点就是，这里不能指定contentType为appplication/json，否则，参数无法传递到后台。

http的Content-Type的类型：

* application/x-www-form-urlencoded ： <form encType=””>中默认的encType，form表单数据被编码为key/value格式发送到服务器（表单默认的提交数据的格式）；
* application/json    ： JSON数据格式

也就是说post请求默认是将表单里面的数据的key/value形式发送到服务，而我们的服务器只需要有对应的key/value属性值的对象就可以接收到。而如果使用application/json，则表示将前端的数据以序列化过的json传递到后端，后端要把它变成实体对象，还需要一个反序列化的过程。按照这个逻辑，那我们如果指定contentType为application/json，然后传递序列化过的对象应该也是可以的。

这时可以看到GET和POST的区别，在chrome中：



这里Post方式没有将参数放在URL后，而是放在了请求体里：



而Get是放在URL中的：





因此，使用HTTP Post的方式时，参数可以有下面几种形式：

    (1)有少量参数时可以将参数写在URL中，如：URL?a=xxxx&b=xxxx

后台：public void Method(string a,string b)

这种方式测试可以，但是调用会提示：{"Message":"请求的资源不支持 http 方法“GET”。"}

   (2)将参数放到body中，后台使用[FromBody]接收，前台：$.ajax(type:POST,url:url,data:JSON字符串);

     后台：public void Method([FromBody]User user)

   (3)ajax传递参数使用JSON字符串，后台直接用类接收，不需要[FromBody]关键字。

WebApi里面put请求一般用于对象的更新。它和用法和post请求基本相同。同样支持[FromBody]，同样可以使用dynamic。

delete请求肯定是用于删除操作的。参数传递机制和post也是基本相同。

WebApi测试工具：WebApiTestClient

详见《[C#进阶系列——WebApi 接口测试工具：WebApiTestClient](http://www.cnblogs.com/landeanfen/p/5210356.html)》

MVC WebApi 在控制器“”上找不到与该请求匹配的操作

使用[FromUri]装饰参数。

[HttpGet]

[Route("api/albums/history/")]

public IHttpActionResult GetHistory([FromUri]string type,[FromUri]int id)

{

//api stuff

}

或者通过路由参数点击api - <http://localhost:50684/api/albums/history/test/1>

[HttpGet]

[Route("api/albums/history/{type}/{id}")]

public IHttpActionResult GetHistory(string type,int id)

{

//api stuff

}

WebAPI异常处理

一般情况下，WebApi作为服务使用，每次客户端发送http请求到我们的WebApi服务里面，服务端得到结果输出response到客户端。这个过程中，一旦服务端发生异常，会统一向客户端返回500的错误。

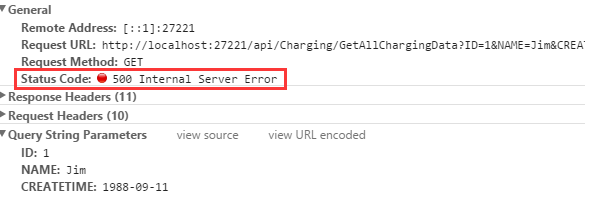
[HttpGet]

publicstring GetAllChargingData([FromUri]TB\_CHARGING obj)

{

thrownew NotImplementedException("方法不被支持");

}



有些时候，我们客户端需要得到更加精确的错误码来判断异常类型，怎么办呢？

MVC里面有IExceptionFilter接口，这个接口用于定义异常筛选器所需的方法，在WebApi里面，也有这么一个异常筛选器，首先在App\_Start里面新建一个类WebApiExceptionFilterAttribute.cs，继承ExceptionFilterAttribute，重写OnException方法。

publicclass WebApiExceptionFilterAttribute : ExceptionFilterAttribute

{

//重写基类的异常处理方法

publicoverridevoid OnException(HttpActionExecutedContext actionExecutedContext)

{

//1.异常日志记录（正式项目里面一般是用log4net记录异常日志）

Console.WriteLine(DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss") + "——" +

actionExecutedContext.Exception.GetType().ToString() + "：" + actionExecutedContext.Exception.Message + "——堆栈信息：" +

actionExecutedContext.Exception.StackTrace);

//2.返回调用方具体的异常信息

if (actionExecutedContext.Exception is NotImplementedException)

{

actionExecutedContext.Response = newHttpResponseMessage(HttpStatusCode.NotImplemented);

}

elseif (actionExecutedContext.Exception is TimeoutException)

{

actionExecutedContext.Response = newHttpResponseMessage(HttpStatusCode.RequestTimeout);

}

//.....这里可以根据项目需要返回到客户端特定的状态码。如果找不到相应的异常，统一返回服务端错误500

else

{

actionExecutedContext.Response = newHttpResponseMessage(HttpStatusCode.InternalServerError);

}

base.OnException(actionExecutedContext);

}

}

**通过判断异常的具体类型，向客户端返回不同的http状态码，示例里面写了两个，可以根据项目的实际情况加一些特定的我们想要捕获的异常，然后将对应的状态码写入http请求的response里面，对于一些我们无法判断类型的异常，统一返回服务端错误500。**关于http的状态码，framework里面定义了一些常见的类型，我们大概看看：

#region程序集 System.dll, v4.0.0.0

// C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.dll

#endregion

using System;

namespace System.Net

{

// 摘要:

// 包含为 HTTP 定义的状态代码的值。

publicenum HttpStatusCode

{

// 摘要:

// 等效于 HTTP 状态 100。 System.Net.HttpStatusCode.Continue 指示客户端可能继续其请求。

Continue = 100,

//

// 摘要:

// 等效于 HTTP 状态 101。 System.Net.HttpStatusCode.SwitchingProtocols 指示正在更改协议版本或协议。

SwitchingProtocols = 101,

//

// 摘要:

// 等效于 HTTP 状态 200。 System.Net.HttpStatusCode.OK 指示请求成功，且请求的信息包含在响应中。这是最常接收的状态代码。

OK = 200,

//

// 摘要:

// 等效于 HTTP 状态 201。 System.Net.HttpStatusCode.Created 指示请求导致在响应被发送前创建新资源。

Created = 201,

//

// 摘要:

// 等效于 HTTP 状态 202。 System.Net.HttpStatusCode.Accepted 指示请求已被接受做进一步处理。

Accepted = 202,

//

// 摘要:

// 等效于 HTTP 状态 203。 System.Net.HttpStatusCode.NonAuthoritativeInformation 指示返回的元信息来自缓存副本而不是原始服务器，因此可能不正确。

NonAuthoritativeInformation = 203,

//

// 摘要:

// 等效于 HTTP 状态 204。 System.Net.HttpStatusCode.NoContent 指示已成功处理请求并且响应已被设定为无内容。

NoContent = 204,

//

// 摘要:

// 等效于 HTTP 状态 205。 System.Net.HttpStatusCode.ResetContent 指示客户端应重置（或重新加载）当前资源。

ResetContent = 205,

//

// 摘要:

// 等效于 HTTP 状态 206。 System.Net.HttpStatusCode.PartialContent 指示响应是包括字节范围的 GET

// 请求所请求的部分响应。

PartialContent = 206,

//

// 摘要:

// 等效于 HTTP 状态 300。 System.Net.HttpStatusCode.MultipleChoices 指示请求的信息有多种表示形式。

// 默认操作是将此状态视为重定向，并遵循与此响应关联的 Location 标头的内容。

MultipleChoices = 300,

//

// 摘要:

// 等效于 HTTP 状态 300。 System.Net.HttpStatusCode.Ambiguous 指示请求的信息有多种表示形式。默认操作是将此状态视为重定向，并遵循与此响应关联的

// Location 标头的内容。

Ambiguous = 300,

//

// 摘要:

// 等效于 HTTP 状态 301。 System.Net.HttpStatusCode.MovedPermanently 指示请求的信息已移到 Location

// 头中指定的 URI 处。接收到此状态时的默认操作为遵循与响应关联的 Location 头。

MovedPermanently = 301,

//

// 摘要:

// 等效于 HTTP 状态 301。 System.Net.HttpStatusCode.Moved 指示请求的信息已移到 Location 头中指定的

// URI 处。接收到此状态时的默认操作为遵循与响应关联的 Location 头。原始请求方法为 POST 时，重定向的请求将使用 GET 方法。

Moved = 301,

//

// 摘要:

// 等效于 HTTP 状态 302。 System.Net.HttpStatusCode.Found 指示请求的信息位于 Location 头中指定的

// URI 处。接收到此状态时的默认操作为遵循与响应关联的 Location 头。原始请求方法为 POST 时，重定向的请求将使用 GET 方法。

Found = 302,

//

// 摘要:

// 等效于 HTTP 状态 302。 System.Net.HttpStatusCode.Redirect 指示请求的信息位于 Location 头中指定的

// URI 处。接收到此状态时的默认操作为遵循与响应关联的 Location 头。原始请求方法为 POST 时，重定向的请求将使用 GET 方法。

Redirect = 302,

//

// 摘要:

// 等效于 HTTP 状态 303。作为 POST 的结果，System.Net.HttpStatusCode.SeeOther 将客户端自动重定向到

// Location 头中指定的 URI。用 GET 生成对 Location 标头所指定的资源的请求。

SeeOther = 303,

//

// 摘要:

// 等效于 HTTP 状态 303。作为 POST 的结果，System.Net.HttpStatusCode.RedirectMethod 将客户端自动重定向到

// Location 头中指定的 URI。用 GET 生成对 Location 标头所指定的资源的请求。

RedirectMethod = 303,

//

// 摘要:

// 等效于 HTTP 状态 304。 System.Net.HttpStatusCode.NotModified 指示客户端的缓存副本是最新的。未传输此资源的内容。

NotModified = 304,

//

// 摘要:

// 等效于 HTTP 状态 305。 System.Net.HttpStatusCode.UseProxy 指示请求应使用位于 Location 头中指定的

// URI 的代理服务器。

UseProxy = 305,

//

// 摘要:

// 等效于 HTTP 状态 306。 System.Net.HttpStatusCode.Unused 是未完全指定的 HTTP/1.1 规范的建议扩展。

Unused = 306,

//

// 摘要:

// 等效于 HTTP 状态 307。 System.Net.HttpStatusCode.RedirectKeepVerb 指示请求信息位于 Location

// 头中指定的 URI 处。接收到此状态时的默认操作为遵循与响应关联的 Location 头。原始请求方法为 POST 时，重定向的请求还将使用

// POST 方法。

RedirectKeepVerb = 307,

//

// 摘要:

// 等效于 HTTP 状态 307。 System.Net.HttpStatusCode.TemporaryRedirect 指示请求信息位于 Location

// 头中指定的 URI 处。接收到此状态时的默认操作为遵循与响应关联的 Location 头。原始请求方法为 POST 时，重定向的请求还将使用

// POST 方法。

TemporaryRedirect = 307,

//

// 摘要:

// 等效于 HTTP 状态 400。 System.Net.HttpStatusCode.BadRequest 指示服务器未能识别请求。如果没有其他适用的错误，或者不知道准确的错误或错误没有自己的错误代码，则发送

// System.Net.HttpStatusCode.BadRequest。

BadRequest = 400,

//

// 摘要:

// 等效于 HTTP 状态 401。 System.Net.HttpStatusCode.Unauthorized 指示请求的资源要求身份验证。 WWW-Authenticate

// 头包含如何执行身份验证的详细信息。

Unauthorized = 401,

//

// 摘要:

// 等效于 HTTP 状态 402。保留 System.Net.HttpStatusCode.PaymentRequired 以供将来使用。

PaymentRequired = 402,

//

// 摘要:

// 等效于 HTTP 状态 403。 System.Net.HttpStatusCode.Forbidden 指示服务器拒绝满足请求。

Forbidden = 403,

//

// 摘要:

// 等效于 HTTP 状态 404。 System.Net.HttpStatusCode.NotFound 指示请求的资源不在服务器上。

NotFound = 404,

//

// 摘要:

// 等效于 HTTP 状态 405。 System.Net.HttpStatusCode.MethodNotAllowed 指示请求的资源上不允许请求方法（POST

// 或 GET）。

MethodNotAllowed = 405,

//

// 摘要:

// 等效于 HTTP 状态 406。 System.Net.HttpStatusCode.NotAcceptable 指示客户端已用 Accept 头指示将不接受资源的任何可用表示形式。

NotAcceptable = 406,

//

// 摘要:

// 等效于 HTTP 状态 407。 System.Net.HttpStatusCode.ProxyAuthenticationRequired 指示请求的代理要求身份验证。

// Proxy-authenticate 头包含如何执行身份验证的详细信息。

ProxyAuthenticationRequired = 407,

//

// 摘要:

// 等效于 HTTP 状态 408。 System.Net.HttpStatusCode.RequestTimeout 指示客户端没有在服务器期望请求的时间内发送请求。

RequestTimeout = 408,

//

// 摘要:

// 等效于 HTTP 状态 409。 System.Net.HttpStatusCode.Conflict 指示由于服务器上的冲突而未能执行请求。

Conflict = 409,

//

// 摘要:

// 等效于 HTTP 状态 410。 System.Net.HttpStatusCode.Gone 指示请求的资源不再可用。

Gone = 410,

//

// 摘要:

// 等效于 HTTP 状态 411。 System.Net.HttpStatusCode.LengthRequired 指示缺少必需的 Content-length

// 头。

LengthRequired = 411,

//

// 摘要:

// 等效于 HTTP 状态 412。 System.Net.HttpStatusCode.PreconditionFailed 指示为此请求设置的条件失败，且无法执行此请求。

// 条件是用条件请求标头（如 If-Match、If-None-Match 或 If-Unmodified-Since）设置的。

PreconditionFailed = 412,

//

// 摘要:

// 等效于 HTTP 状态 413。 System.Net.HttpStatusCode.RequestEntityTooLarge 指示请求太大，服务器无法处理。

RequestEntityTooLarge = 413,

//

// 摘要:

// 等效于 HTTP 状态 414。 System.Net.HttpStatusCode.RequestUriTooLong 指示 URI 太长。

RequestUriTooLong = 414,

//

// 摘要:

// 等效于 HTTP 状态 415。 System.Net.HttpStatusCode.UnsupportedMediaType 指示请求是不支持的类型。

UnsupportedMediaType = 415,

//

// 摘要:

// 等效于 HTTP 状态 416。 System.Net.HttpStatusCode.RequestedRangeNotSatisfiable 指示无法返回从资源请求的数据范围，因为范围的开头在资源的开头之前，或因为范围的结尾在资源的结尾之后。

RequestedRangeNotSatisfiable = 416,

//

// 摘要:

// 等效于 HTTP 状态 417。 System.Net.HttpStatusCode.ExpectationFailed 指示服务器未能符合 Expect

// 头中给定的预期值。

ExpectationFailed = 417,

//

UpgradeRequired = 426,

//

// 摘要:

// 等效于 HTTP 状态 500。 System.Net.HttpStatusCode.InternalServerError 指示服务器上发生了一般错误。

InternalServerError = 500,

//

// 摘要:

// 等效于 HTTP 状态 501。 System.Net.HttpStatusCode.NotImplemented 指示服务器不支持请求的函数。

NotImplemented = 501,

//

// 摘要:

// 等效于 HTTP 状态 502。 System.Net.HttpStatusCode.BadGateway 指示中间代理服务器从另一代理或原始服务器接收到错误响应。

BadGateway = 502,

//

// 摘要:

// 等效于 HTTP 状态 503。 System.Net.HttpStatusCode.ServiceUnavailable 指示服务器暂时不可用，通常是由于过多加载或维护。

ServiceUnavailable = 503,

//

// 摘要:

// 等效于 HTTP 状态 504。 System.Net.HttpStatusCode.GatewayTimeout 指示中间代理服务器在等待来自另一个代理或原始服务器的响应时已超时。

GatewayTimeout = 504,

//

// 摘要:

// 等效于 HTTP 状态 505。 System.Net.HttpStatusCode.HttpVersionNotSupported 指示服务器不支持请求的

// HTTP 版本。

HttpVersionNotSupported = 505,

}

}

定义好了异常处理方法，剩下的就是如何使用了。可以根据实际情况，在不同级别使用统一的异常处理机制。

1、接口级别

[WebApiExceptionFilter]

[HttpGet]

publicstring GetAllChargingData([FromUri]TB\_CHARGING obj)

{

thrownew NotImplementedException("方法不被支持");

}

执行到异常后，会先进到OnException方法：



**如果需要，甚至可以向Status Code里面写入自定义的描述信息，并且还可以向我们的Response的Content里面写入我们想要的信息**。我们稍微改下OnException方法：

if (actionExecutedContext.Exception is NotImplementedException)

{

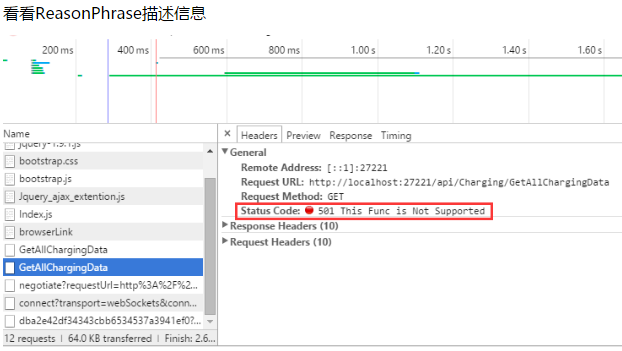
var oResponse = new HttpResponseMessage(HttpStatusCode.NotImplemented);

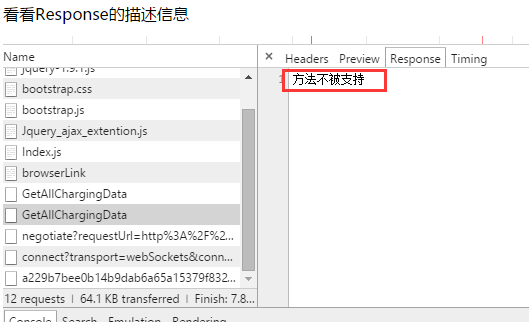
oResponse.Content = new StringContent("方法不被支持");

oResponse.ReasonPhrase = "This Func is Not Supported";

actionExecutedContext.Response = oResponse;

}





2、控制器级别

如果想要某一个或者多个控制器里面的所有接口都使用异常过滤，直接在控制器上面标注特性即可。

* 某一个控制器上面启用异常过滤

[WebApiExceptionFilter]

publicclass ChargingController : BaseApiController

{

#region Get

[HttpGet]

publicstring GetAllChargingData([FromUri]TB\_CHARGING obj)

{

thrownew NotImplementedException("方法不被支持");

}

}

* 多个控制器上面同时启用异常过滤

[WebApiExceptionFilter]

publicclass BaseApiController : ApiController

{

}

然后控制器继承BaseApiController：

publicclass ChargingController : BaseApiController

{

#region Get

[HttpGet]

publicstring GetAllChargingData([FromUri]TB\_CHARGING obj)

{

thrownew NotImplementedException("方法不被支持");

}

}

这样，所有继承BaseApiController的子类都会启用异常过滤。

3、全局配置

如果需要对整个应用程序都启用异常过滤，则需要做如下两步：

(1)在Global.asax全局配置里面添加 GlobalConfiguration.Configuration.Filters.Add(new WebApiExceptionFilterAttribute()); 这一句，如下：

void Application\_Start(object sender, EventArgs e)

{

// 在应用程序启动时运行的代码

AreaRegistration.RegisterAllAreas();

GlobalConfiguration.Configure(WebApiConfig.Register);

RouteConfig.RegisterRoutes(RouteTable.Routes);

GlobalConfiguration.Configuration.Filters.Add(new WebApiExceptionFilterAttribute());

}

在WebApiConfig.cs文件的Register方法里面添加  config.Filters.Add(new WebApiExceptionFilterAttribute()); 这一句，如下：

publicstaticvoid Register(HttpConfiguration config)

{

//跨域配置

config.EnableCors(new EnableCorsAttribute("\*", "\*", "\*"));

// Web API 路由

config.MapHttpAttributeRoutes();

RouteTable.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{action}/{id}",

defaults: new { id = RouteParameter.Optional }

).RouteHandler = newSessionControllerRouteHandler();

config.Filters.Add(new WebApiExceptionFilterAttribute());

}

上面说的是全局的异常捕获以及处理方式，在某些情况下，我们希望以异常的方式向客户端发送相关信息，可能就需要用到我们的HttpResponseException。比如：

[HttpGet]

public TB\_CHARGING GetById(string id)

{

//从后台查询实体

var oModel = server.Find(id);

if (oModel == null)

{

var resp = new HttpResponseMessage(HttpStatusCode.NotFound)

{

Content = new StringContent(string.Format("没有找到id={0}的对象", id)),

ReasonPhrase = "object is not found"

};

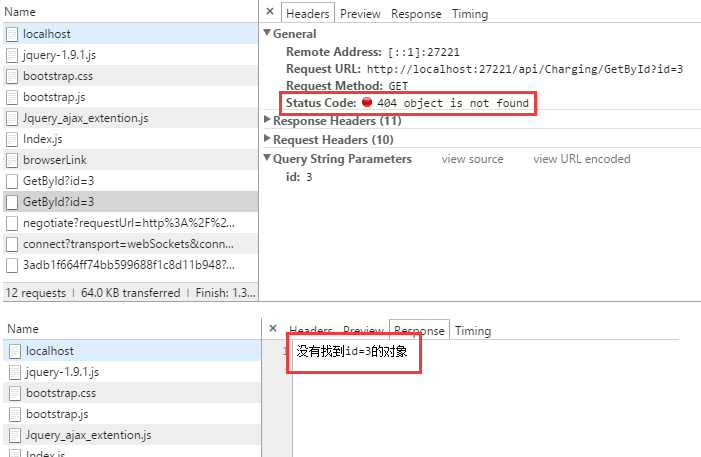
thrownew HttpResponseException(resp);

}

return oModel;

}

执行之后浏览器里面查看结果：



这里既使用了HttpResponseMessage，又使用了HttpResponseException，那么，像这种可控的异常，我们是否可以直接以HttpResponseMessage的形式直接返回到客户端而不用抛出异常呢？这里就要谈谈这两个对象的区别了，个人的理解是HttpResonseMessage对象用来响应讯息并包含状态码及数据内容，HttpResponseException对象用来向客户端返回包含错误讯息的异常。

当呼叫 Web API 服务时发生了与预期上不同的错误时，理当应该中止程序返回错误讯息，这时对于错误的返回就该使用 HttpResponseException，而使用 HttpResponseMessage 则是代表着当客户端发送了一个工作请求而 Web API 正确的完成了这个工作，就能够使用 HttpResponseMessage 返回一个 201 的讯息，所以 HttpResponseMessage 与 HttpResponseException 在使用上根本的目标就是不同的，用 HttpResponseMessage 去返回一个例外错误也会让程序结构难以辨别且不够清晰。

HttpError对象提供一致的方法来响应正文中返回错误的信息。准确来说，HttpError并不是一个异常，只是用来包装错误信息的一个对象。其实在某一定的程度上，HttpError和HttpResponseMessage使用比较相似，二者都可以向客户端返回http状态码和错误讯息，并且都可以包含在HttpResponseException对象中发回到客户端。但是，一般情况下，HttpError只有在向客户端返回错误讯息的时候才会使用，而HttpResponseMessage对象既可以返回错误讯息，也可返回请求正确的消息。其实关于HttpError没什么特别好讲的，我们来看一个例子就能明白：

public HttpResponseMessage Update(dynamic obj)

{

TB\_Product oModel = null;

try

{

var id = Convert.ToString(obj.id);

oModel = Newtonsoft.Json.JsonConvert.DeserializeObject<TB\_Product>(Convert.ToString(obj.dataModel));

//...复杂的业务逻辑

}

catch(Exception ex)

{

return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex.Message);

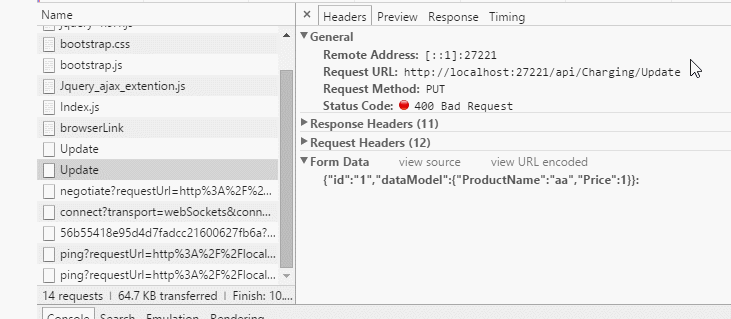
}

return Request.CreateResponse<TB\_Product>(HttpStatusCode.OK, oModel);

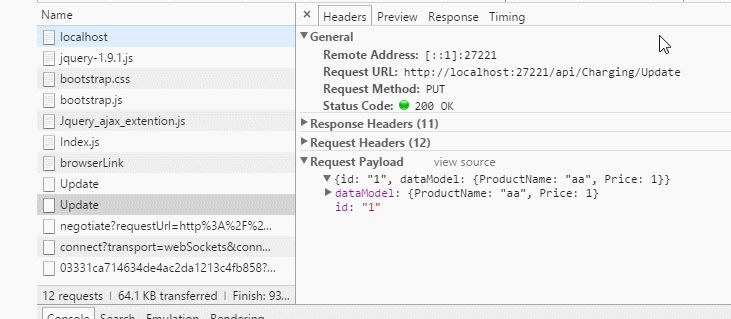
}

假如现在在执行try里面复杂业务逻辑的时候发生了异常，我们捕获到了异常然后向客户端返回HttpError对象，这个对象里面包含我们自定义的错误讯息，如果正常则返回HttpResponseMessage对象。

如果请求异常：



如果请求正常：



详细可见<http://www.cnblogs.com/landeanfen/p/5363846.html>

WebAPI自宿主

Asp.Net WebAPI生成的是一个程序集，并不是独立的进程，因此，要运行的时候必须将其承载在相应的宿主上，一般比较常见的是IIS承载。很多时候，我们为了简化部署或者功能集成，需要将其承载到独立的进程上，这种方式一般称之为Self-Host。

这里的API编写位置有区分，包括写在宿主程序中，和写在自己的单独程序集中，网上的例子大多是写在宿主程序中的。

默认ASP.NET Web API在Self Host寄宿模式下用于解析程序集的AssembliesResolver是一个DefaultAssembliesResolver对象，它只会提供当前应用程序域已经加载的程序集。如果我们将HttpController定义在非寄宿程序所在的程序集中（实际上在采用Self Host寄宿模式下，我们基本上都会选择在独立的项目定义HttpController类型），即使我们将它们部署在宿主程序运行的目录中，宿主程序启动的时候也不会主动去加载这些程序集。由于当前应用程序域中并不曾加载这些程序集，HttpController类型解析将会失败，HttpController的激活自然就无法实现。

通过Nuget获取WebAPI SeltHost库，添加引用：



using System.Web.Http;

using System.Web.Http.SelfHost;

using System.Web.Mvc;

WebApi自宿主：

private void btn启动IndependWebApi\_Click(object sender, EventArgs e)

{

Assembly.Load("IndependWebApi"); //webapi不在宿主程序中一定要包含这句话，引用API生成的dll

var config = new HttpSelfHostConfiguration("http://localhost:8100");

config.Routes.MapHttpRoute("default", "api/{controller}/{id}", new { id = RouteParameter.Optional });

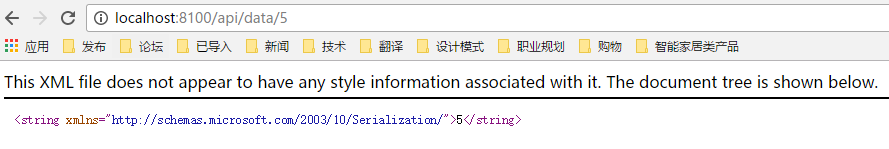
var server = new HttpSelfHostServer(config);

server.OpenAsync().Wait();

ccMsg.Text = "Server is opened";

}

网址输入：<http://localhost:8100/api/data/5>，即可以看到执行结果：



不过返回的是xml，且原本[Route("freeapi/Data/GetEntity")]方式并不支持。

WebAPI和MVC的区别(需要不断学习和完善)

1、Asp.net MVC 是用来创建返回视图（Views）与数据的Web应用，而Asp.net Web API是一种简单轻松地成熟的HTTP服务，它只返回数据，不返回视图（Views）。

2、Asp.net Web API可以通过.Net Framework来帮助我们构建REST-ful服务，而且他支持内容协商（根据客户端能接受的格式要求，返回相应的JSON，XML，ATOM），同时Asp.net Web API支持自我宿主（self-hosting），而MVC并不支持（只能宿主在IIS中）。

3、Asp.net Web API可以返回特定的数据类型，比如JSON，XML，或者其他在请求头中定义的数据格式。而MVC只能利用Json Result返回JSON数据类型。

4、Asp.net Web API 根据HTTP 谓语动词来映射Action，但MVC只是映射 Action 名称。

5、Asp.net Web API 一种全新的框架，它是Asp.net Framework 核心库的一部分。在Asp.net Web API 一些存在MVC中的特征（model binding、filters、路由）是存在System.Web.Http程序集中，而MVC是存在System.Web.Mvc中。因此，Web API 可以和Asp.net一起使用，也可以做独立的服务层。

7、总之，WebApi在提供数据方面，是比MVC更加轻量的架构。

Web API 和 MVC可以说是两个不同的东西。Web API更倾向于基于HTTP协议的服务，直接返回用户的数据请求。MVC是建站的一种框架，倾向于响应用户的页面请求，处理数据并在view中呈现。

# 四、其他web知识

超文本/超媒体(HyperTcxt/HyperModia)

超文本是一份呈现文本内容的电子文档,其核心在于可以利用内嵌的“超链接(Hyperlink直接访问引用的另—份文档。超媒体对超文本做了简单的扩展使之可以呈现多媒体内容(比如图片、音频和视频等〉。HTML文档是我们常见的最为典型的超文本/超媒体文件。

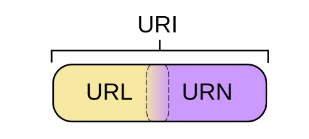
URI/URL:

A [Uniform Resource Name](https://en.wikipedia.org/wiki/Uniform_Resource_Name) (URN) may be compared to a person's name, while a [Uniform Resource Locator](https://en.wikipedia.org/wiki/Uniform_Resource_Locator) (URL) may be compared to their street address.

URI是统一资源标识符，是一个用于标识某一互联网资源名称的字符串。 该种标识允许用户对任何（包括本地和互联网）的资源通过特定的协议进行交互操作。URI由包括确定语法和相关协议的方案所定义。由是三个组成部分：访问资源的命名机制、存放资源的主机名、资源自身的名称，由路径表示。

URL是统一资源定位，是对可以从互联网上得到的资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

.net中的URI我理解是不是一个类，里面的组成部分是这个对象的属性。URL其实就是一个字符串,比如百度URL即是[http://www.baidu.com](http://www.baidu.com/" \t "_blank)。



URI具有URL和URN两种主要的表现形式,只有前者具有可寻址。

HTTP

超文本传输协议(HyperText Transfer Protocol -- HTTP)是一个设计来使客户端和服务器顺利进行通讯的协议。

HTTP/1.1 协议规定的 HTTP 请求方法有 OPTIONS、GET、HEAD、POST、PUT、DELETE、TRACE、CONNECT 这几种。

GET 请求响应

GET请求会显示请求指定的资源。一般来说GET方法应该只用于数据的读取。

GET会方法请求指定的页面信息，并返回响应主体，GET被认为是不安全的方法，因为GET方法会被网络蜘蛛等任意的访问。

使用GET方法时，查询字符串（键值对）被附加在URL地址后面一起发送到服务器：

/test/demoform.html?name1=value1&name2=value2

特点：

GET请求能够被缓存

GET请求会保存在浏览器的浏览记录中

以GET请求的URL能够保存为浏览器书签

GET请求有长度限制

GET请求主要用以获取数据

POST 发送数据给服务器处理

POST 发送数据给服务器处理，数据包含在HTTP信息正文中

POST请求会向指定资源提交数据，请求服务器进行处理，如：表单数据提交、文件上传等，请求数据会被包含在请求体中。

POST方法可能会创建新的资源或/和修改现有资源。

使用POST方法时，查询字符串在POST信息中单独存在，和HTTP请求一起发送到服务器：

POST /test/demoform.html HTTP/1.1

Host: w3schools.com

name1=value1&name2=value2

特点：

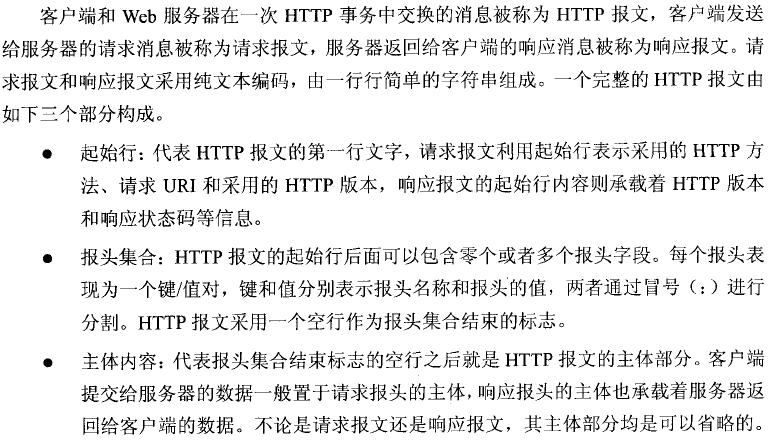
POST请求不能被缓存下来

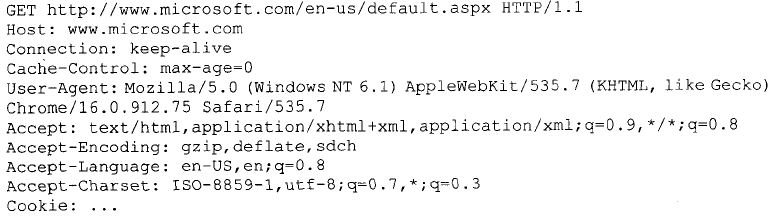
POST请求不会保存在浏览器浏览记录中

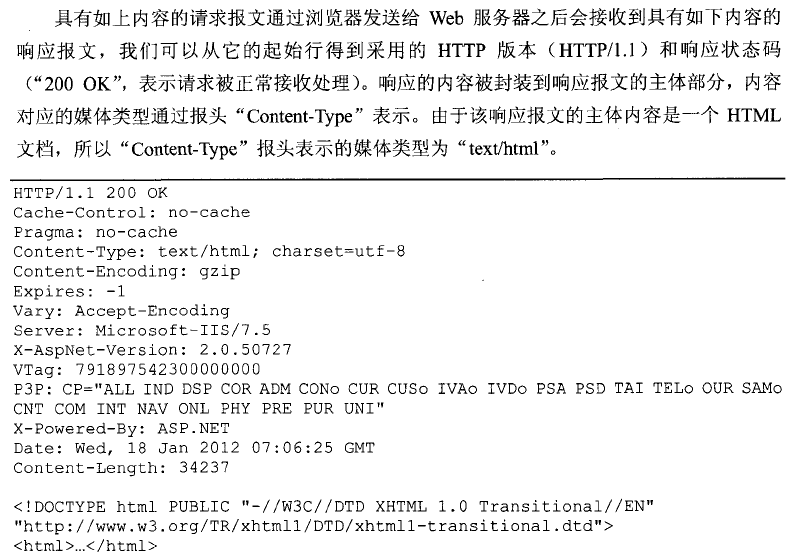
以POST请求的URL无法保存为浏览器书签

POST请求没有长度限制

HTTP报文







RPC架构和Restful风格

RPC（Remote Promote Call） 一种进程间通信方式。允许像调用本地服务一样调用远程服务。

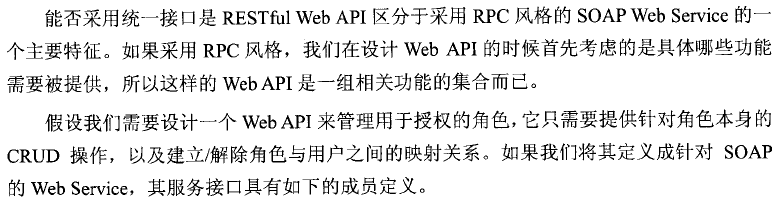
RPC框架的主要目标就是让远程服务调用更简单、透明。RPC框架负责屏蔽底层的传输方式（TCP或者UDP）、序列化方式（XML/JSON/二进制）和通信细节。开发人员在使用的时候只需要了解谁在什么位置提供了什么样的远程服务接口即可，并不需要关心底层通信细节和调用过程。

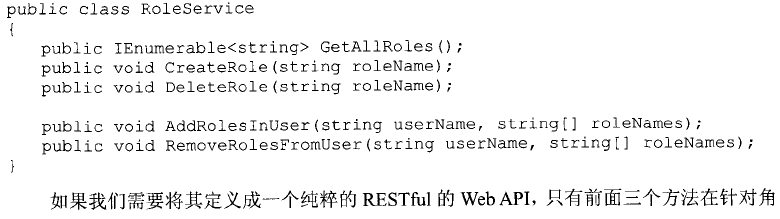
而REST则不然，它的URL主体是资源，是个名词。而且也仅支持HTTP协议，规定了使用HTTP Method表达本次要做的动作，类型一般也不超过那四五种。这些动作表达了对资源仅有的几种转化方式。

两者没有高下之分，无非是一种约定俗成的标准。习惯用RPC就用RPC，能理解REST就用REST。

我们现在拥有2个服务框架，一个基于RPC(远程过程调用(Remote Procedure Call) )机制的WCF和一个基于HTTP的ASP.NET Web Api。

由于REST是面向资源的,所以一个Web API旨在提供针对单一资源的操作。针对资源的基本操作唯CRUD而已,



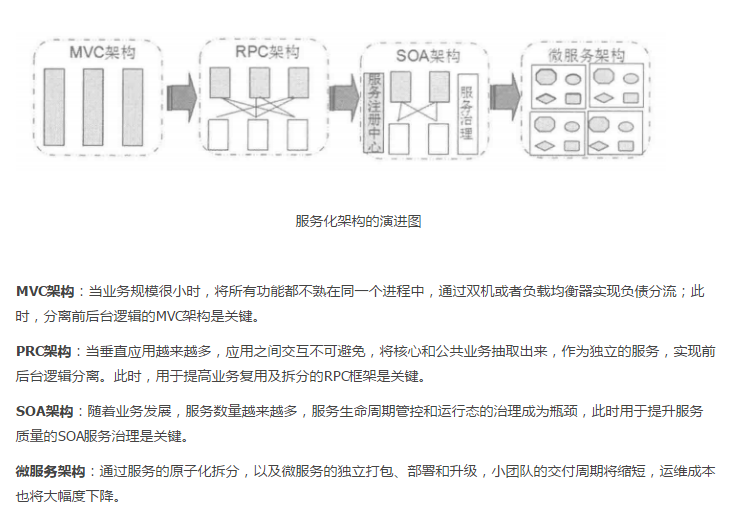




RESTful只需要维护资源的状态,但不需要维护客户端的状态。对于它来说,每次请求都是全新的,它只需要针对本次请求进行相应的操作,而不需要将本次请求的相关信息记录下来以便用于后续来自相同客户端请求的处理

WCF可以不依赖于IIS。

WCF可以配置成BasicHttpBinding来兼容（或者说变身成）WS。  
WCF可以基于TCP或者MessegeQueue来传输数据。  
WCF的可配置性比WS强，比如安全性。  
WCF可以是有状态的，并支持事务。  
WCF 支持多种通信协议 Http/Https 、TCP/UDP、MSMQ、命名管道、对等网、消息可达性、事务流等。



幂等性(Idempotent):是一个数学上的概念,在这里表示发送—次和多次请求引起的边界效应是—致的。在网速不够快的情况下,客户端发送一个请求后不能立即得到响应,由于不能确定请求是否被成功提交,所以它有可能会再次发送另—个相同的请求。幂等性决定了第二个请求是有效。

Post和Get

假如使用“GET”或省略其属性，浏览器将该值绑定在页面所有控件上，成为一个查询字符串，且附在被请求页面的URL上。当这个请求到达Web服务器时，其值由ASP的Request.QueryString集合提供。然而，假如设置METHOD属性为“POST”，浏览器将值包装进发送服务器的HTTP报头中，通过Request.Form集合提供给ASP。

通常来说，可以在所有的HTML窗体中使用GET方法。然而，浏览器或服务器的URL字符串长度存在一定的限制。因此，附有长的字符串可能会引起溢出和某些字符串的字符被截掉。同时，查询字符串出现在浏览器的地址栏和所有的保存的链接和收藏夹中。不仅如此，还显露了通过Web服务器时在HTTP请求中不想显示的值，它也可能出现你的服务器和其他路由服务器的日志文件中。在HTTP请求报头中的值很少是可见的，并且不出现在日志文件中。

在Chrome中，查看HTTP报文能看出差异，Get在报文头的URL中，而Post则在报文正文的body中。

页面状态

**第一部分：页面状态**

Http协议————“无状态协议”  
Web服务器每分钟对上千个用户进行管理的一种方式就是执行所谓的“无状态”链接。只要有一个希望浏览器返回一个页面、图像或其他资源的请求，就发生以下事情：  
1、连接到服务器  
2、告诉服务器想要的页面、图像或者其他项  
3、服务器发送请求的资源  
4、服务器切断连接，把用户忘的干干净净。  
也就是页面之间在Http协议下是没有任何关系的，这样就需要有状态管理来传输页面之间的数据。

**第二部分：基于客户端的状态**

状态信息放在客户端上。  
一、视图状态（ASP.NET特有）  
二、隐藏的窗体域  
三、Cookie  
四、查询字符串

**第三部分 基于服务器的状态**

信息存储在服务器上，尽管其安全型较高，但会占用较多的web服务器资源。服务器端通常用以下方式实现状态管理：  
**一、Application对象  
二、Session对象**

简单的说，Application是应用程序级别的状态存储，Session是会话级别的状态存储。  
另外作用域不同，Application对象针对所有用户都生效，Session对象则相反，每个用户都有自己的Session对象，它的生命周期起始于服务器产生对用户请求页面的响应，终止于用户断开与服务器的连接。

举例来说，当手机开机时，相当于一个Applicaion开始，然后当一个朋友打电话过来，此时一个Session开始，挂电话时相当于这个Session结束，然后又有一个朋友打电话过来，此时另一个Session开始。手机未关机说明Application还未结束。

由于Application和Session状态都存储在内存中，但是当服务器重新启动时，保留的状态就会消失了，为了保留其状态，就必须将状态保存到数据库。如网站计数器。

# 五、基本语法：

语法糖

Dictionary<int, string> dict = new Dictionary<int, string>() { { 1, "1" }, { 2, "2" } };

对象初始化器：

RequestEntity request = new RequestEntity() {

BaseUrl = url,

Router= "operation/templateManage/list",

MethodType= RestSharp.Method.GET,

Headers=new Dictionary<string, object>()

{

{"appId", CpdatCloudClient.Interop.Utility.Token.appId },

{"appKey", CpdatCloudClient.Interop.Utility.Token.appKey },

{"agentId", systemCode },

}

};

访问权限

a. 在namespace中的类、接口默认是internal类型的，也可以显式地定义为public类型，不允许是其他访问类型。

b. 在一个类里面，属性和方法默认是private的，可以显式地定义为public、private、protected、internal或protected internal等访问类型。

c. 接口中不能定义成员变量，接口中的方法默认为public的访问权限，但是不能显式地定义任何访问类型。

d. 抽象类中必须有一个以上的抽象方法，抽象方法可以是public、internal、protected,不能是private的访问类型。

public可以被任意存取；

  protected只可以被本类和其继承子类存取；

  internal只可以被本组合体（Assembly）内所有的类存取，组合体是C#语言中类被组合后的逻辑单位和物理单位，其编译后的文件扩展名往往是“.DLL”或“.EXE”。

protected internal 唯一的一种组合限制修饰符，它只可以被本组合体内所有的类和这些类的继承子类所存取。

private只可以被本类所存取。

abstract用来修饰抽象类，表示该类只能作为父类被用于继承，而不能进行对象实例化。  
    抽象类可以包含抽象的成员，但这并非必须。  
    abstract不能和new同时用。  
    下面是抽象类用法的伪码：      
    abstract class A  {     public abstract void F();  }    
    abstract class B: A  {     public void G() {}  }   
    class C: B  {     public override void F()      {   //方法F的实现     }  }      
    抽象类A内含一个抽象方法F()，它不能被实例化。类B继承自类A，其内包含了一个实例方法G(),但并没有实现抽象方法F()，所以仍然必须声明为抽象类。  
    类C继承自类B，实现类抽象方法F()，于是可以进行对象实例化。       
sealed用来修饰类为密封类，阻止该类被继承。同时对一个类作abstract和sealed的修饰是没有意义的，也是被禁止的。

字符串换行

string mmString = "select \* from A" +

" where ID='a'";

ccMsg.Text += mmString + Environment.NewLine;

mmString = @"select \* from A

where ID='a'";

ccMsg.Text += mmString + Environment.NewLine;

可选参数

public void Test(int id,string name="ronaldo",int age=20)

{

}

//可选参数必须位于所有必需参数之后，调用如下：

Test(1);         //同时使用了两个参数的默认值

Test(1,age:12);     //name使用了默认值,age使用命名参数的方式设置为12

params

params 后跟一维数组，表示不确定个数的参数。

关于params关键字的用法，需要注意以下几点：

1． params只能用于一维数组，不能用于多维数组和诸如ArrayList、List等任何类似于数组的集合类型。

2． 被加上params关键字的形参，必须是形参列表中最后一个形参，并且方法声明中只允许一个 params 关键字。

3． 用params关键字的方法，有三种调用形式：

第一种，列举出数组各元素：Sum(1,2,3)，这也是最常用的形式；

第二种，像没有加params关键字的数组形参那样，用数组名做实参：Sum(new int[]{1,2,3})或int n=new int[]{1,2,3};Sum(n);；

第三种，加params关键字的参数在调用时可以省略：Sum();//返回0；这种方式有时可以少定义一个方法重载，但当明确定义了重载int Sum()时，编译器会优先调用int Sum()，而不是Sum(params int[] values)。而且省略params型参数，方法内部仍会new一个元素个数为0的数组，效率略查。

第四种，不省略params型参数，用null代替，效率比第三种略高，因为其内部不会new这个数组。

privatevoid btn基础\_Click(object sender, EventArgs e)

{

ccMsg.Clear();

var i = myMethod("myMethod(\"\",\"Jack\")", "Jack");

ccMsg.Text += "共：" + i + "个params参数！" +Environment.NewLine;

i = myMethod("myMethod(\"\",\"Jack\", \"Lucy\", \"Tom\")", "Jack", "Lucy", "Tom");

ccMsg.Text += "共：" + i + "个params参数！" +Environment.NewLine;

i = myMethod("myMethod(\"\",\"Jack\", \"Lucy\", \"Tom\", \"Json\", \"Albert\")", "Jack", "Lucy", "Tom", "Json", "Albert");

ccMsg.Text += "共：" + i + "个params参数！" +Environment.NewLine;

//直接传递数组

i = myMethod("myMethod(\"\",数组)", newstring[5] { "Jack", "Lucy", "Tom", "Json", "Albert" });

ccMsg.Text += "共：" + i + "个params参数！" +Environment.NewLine;

}

///<summary>

/// params 后跟一维数组，表示不确定个数的参数。

///</summary>

///<param name="mmMsg"></param>

///<param name="names"></param>

///<returns></returns>

int myMethod(string mmMsg,paramsstring[] names)

{

ccMsg.Text += Environment.NewLine+"==========================" + Environment.NewLine;

ccMsg.Text += mmMsg + ":" + Environment.NewLine;

for (var i = 0; i < names.Length; i++)

{

ccMsg.Text += (i+1) + "、" + names[i]+Environment.NewLine;

}

return names.Length;

}

Type

dtSource.Columns[colIndex].DataType == Type.GetType("System.DateTime")

浅拷贝与深拷贝

浅拷贝(MemberwiseClone方法)：如果字段是值类型的，则对该字段执行逐位复制，如果字段是引用类型，则复制引用但不复制引用的对象；因此，原始对象及其副本引用同一对象。也就是对象的引用依然指向原来的对象。

深拷贝：等待具体研究。

**浅拷贝：指的是拷贝一个对象时，仅仅拷贝对象的引用进行拷贝，但是拷贝对象和源对象还是引用同一份实体。此时，其中一个对象的改变都会影响到另一个对象。**

**深拷贝：**指的是拷贝一个对象时，不仅仅把对象的引用进行复制，还把该对象引用的值也一起拷贝。这样进行深拷贝后的拷贝对象就和源对象互相独立，其中任何一个对象的改动都不会对另外一个对象造成影响。

.NET中值类型默认是深拷贝的，而对于引用类型，默认实现的是浅拷贝。

var

1、var 仅能声明方法内的局部变量

2、var 声明的变量在被赋值后类型即确定下了，后续程序中不能在赋其他类型的值

3、var x = new object() 没有意义，不要写这样的代码...............

var 与 DataTable DataRow循环

循环DataTable.Rows的时候请记住别用var,因为DataRowCollection是array，遍历出来是object，这时使用var无法识别对象类型。

可空类型

可空类型仅针对于值类型，不能针对引用类型去创建。System.Nullable<T>简写为T ?

int? num = null;

如果num.HasValue为false，那么在使用value值的时候会抛出异常。把一个Nullable的变量x赋值给一个非Nullable的变量y可以这么写：int y = x ?? -1;

去掉0及小数点

my最优价方法.去掉最高报价百分比.ToString().TrimEnd(new char[] { '0', '.' })

执行后，10.0000000变成了1。

去掉小数点后的零

Convert.ToDecimal(my最优价方法.去掉最高报价百分比).ToString("0.##")

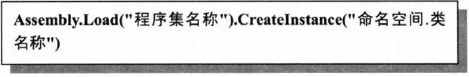
执行后，10.0000000变成了10。

反射

反射是一个程序集发现及运行的过程，通过反射可以得到\*.exe或\*.dll等程序集内部的信息。使用反射可以看到一个程序集内部的接口、类、方法、字段、属性、特性等等信息。在System.Reflection命名空间内包含多个反射常用的类。

可以找出对象所包含成员变量，属性，方法等，然后进行赋值，调用等。但这些成员变量，属性，方法的名称在编程的时候都是不知道的。反射可以使用未知或动态的dll。这也是投标分析系统中的webservice为什么没有引用而使用反射的原因。还有就是在简单工厂和工厂中可以用到，引用了DLL,由于对象是个接口或抽象类如动物，用户也无法知道一个接口有多少个可以使用的实现类，具体是实例化成狗还是猫得等到getInstance时才知道，如:Fruit f = Factory.getInstance("Reflect.Apple");

具体的格式如下：



装箱和拆箱

object是所有类的父类。Int等值类型和object的转换通过装箱和拆箱操作，就像外面套了个箱子，可能会有性能损耗，而且object无法在运行的时候检查类型，可能会报错。

泛型

泛型是[程序设计语言](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1%E8%AF%AD%E8%A8%80" \t "_blank)的提供的一种特性，.net是2.0版本开始的。将类型参数化以达到代码复用提高软件开发工作效率，[泛型类](https://baike.baidu.com/item/%E6%B3%9B%E5%9E%8B%E7%B1%BB)是引用类型，是堆对象，主要是引入了类型参数这个概念。本质是占位符。泛型是基于JIT的，由CLR运行时支持的。代码经过第一次C#编译器编译后，产生的IL是并没有为T指定一个特定的类型，泛型这种开放类型，不能被实例化，真正的泛型实例化，发生在JIT编译时，产生的机器码取决于给T传入的类型。也就是运行时注入类型。

CLR中有专门的IL指令支持泛型操作。

→初次编译时，生成IL代码和元数据，T只是类型占位符，在编译时不进行实例化  
→JIT编译时，以实际类型替换元数据中的T占位符  
→将元数据转换为本地代码

string showType<T>(T ppInput)

{

return ppInput.ToString(); //T是占位符

}

泛型类：class MyCollection<T> { T variable1; private void Add(T param){ } }

使用：

ccMsg.Text += new Fans<double>().TypeTransferToString(100.65);

因此，相较于object，泛型并没有装箱和拆箱的操作，性能和使用普通方法(参数直接指定为int)完全一致，而且它是指定类型的参数，保证了类型安全。而且泛型约束也使得泛型的使用和控制比object更安全、灵活，类型无法满足泛型约束的要求，编译就无法通过。

泛型的协变和逆变：协变(Convariant)和逆变(Contravariant)的出现，使数组、委托、泛型类型的隐式转换变得可能。 子类转换成基类，称之为协变；基类转换成子类，称之为逆变。.NET4.0以来，支持了泛型接口的协变和逆变。

具体详见：<http://www.cnblogs.com/darrenji/p/3851827.html>

初始化器

对象初始化器将创建对象和赋值合为一行，其中赋值就是在后面大括号里做的。这里的赋值可以给所有属性赋值，也可以给部分属性赋值。

Person person = new Person { Name = "Slark", Age = 100, Address = "Xi'an" };

集合初始化器的原理也很简单，就是它默默的为我们调用了List的Add方法依次为我们添加了这3个元素。

List<Person> personList = new List<Person>

{

new Person { Name = "Slark1", Age = 101, Address = "Xi'an1" },

new Person { Name = "Slark2", Age = 102, Address = "Xi'an2" },

new Person { Name = "Slark3", Age = 103, Address = "Xi'an3" }

};

委托

简单说它就是一个能把方法当参数传递的对象，而且还知道怎么调用这个方法，同时也是粒度更小的“接口”（约束了指向方法的签名）。

delegateint Calculator (int x);此委托适用于任何有着int返回类型和一个int类型参数的方法，如：staticint Double (int x) { return x \* 2; }

创建一个委托实例，将该此方法赋值给该委托实例：Calculator c = new Calculator(Double); 也可以简写成：Calculator c = Double;这个方法可以通过委托调用：int result = c(2);也可以：int result = c.Invoke(2);也就是Invoke可以省略。

如果Double方法是临时的，只调用一次，若在整个程序中不会有第二次调用，我们就可以使用匿名函数或Lambda表达式。

多播委托

一个委托实例不仅可以指向一个方法，还可以指向多个方法。例如：MyDelegate d = MyMethod1;d += MyMethod2; 调用时，按照方法被添加的顺序依次执行。注意，对于委托，+= 和 -= 对null是不会报错的，如：MyDelegate d;

d += MyMethod1;// 相当于MyDelegate d = MyMethod1;

当一个类的实例的方法被赋给一个委托对象时，在上下文中不仅要维护这个方法，还要维护这个方法所在的实例。System.Delegate 类的Target属性指向的就是这个实例。

X x = newX();

ProgressReporter p = x.InstanceProgress; //X类的InstanceProgress()方法，非静态

p(1);

Console.WriteLine(p.Target == x); // True

对于静态方法，System.Delegate 类的Target属性是Null，所以将静态方法赋值给委托时性能更优。

假如你的委托是返回某一个类型的，那么，在调用多播委托的过程中，返回值会被依次赋值，最终的返回值就是最后一次委托调用的返回值。所以说多播委托一般都没有返回类型（也就是说返回类型为void），因为给多播委托设返回类型一般是没有什么实际意义的。

泛型委托(Func，Action,Predicate)

含有泛型参数的委托，例如：delegateTCalculator<T>(T argOne,T argTwo) whereT : struct;

或：delegatevoidCalculator2<T>(T argOne, T argTwo) whereT : struct;

如下例：

delegateTCalculator<T>(T argOne,T argTwo) whereT : struct;

delegatevoidCalculator2<T>(T argOne, T argTwo) whereT : struct;

privatevoid btn泛型委托\_Click(object sender, EventArgs e)

{

Calculator2<int> mmDoCal2 = DoSum2;

mmDoCal2(9, 16);

Calculator<int> mmDoCal = DoSum;

ccMsg.Text += mmDoCal(5, 6).ToString() + Environment.NewLine;

}

int DoSum(int x, int y)

{

return (x + y);

}

void DoSum2(int x, int y)

{

ccMsg.Text=(x + y).ToString()+ Environment.NewLine;

}

Func和Action都是.NET里面的内置委托，它有很多重载。区别是Func最后一个参数是返回值。详见“C#”—“语法”中的《[Func与Action](http://www.cnblogs.com/Gyoung/archive/2013/04/04/2997050.html)》(<http://www.cnblogs.com/Gyoung/archive/2013/04/04/2997050.html>)

Action：没有参数没有返回值

Action<T>:有参数没有返回值

Func<T>: 有返回值

Predicate<T>:有一个bool类型的返回值，多用在比较的方法中，Predicate只有一个参数，且返回一个bool类型的值。具体使用如下：

List<TT最优价方法> pp最优价方法列表=new List<TT最优价方法>();

pp最优价方法列表.Find((item) => item.最优价方法ID==pp最优价方法ID);

List<TT最优价方法> mm数量大于列表 = pp最优价方法列表.FindAll((item) => item.有效报价限制数量 > my最优价方法.有效报价限制数量);

还可以这样，降序排列：

mm数量大于列表.Sort(new Comparison<TT最优价方法>((x, y) =>

{

if (x.有效报价限制数量 > y.有效报价限制数量)

return -1;

else if (x.有效报价限制数量 == y.有效报价限制数量)

return 0;

else

return 1;

}));

当不开启编译器优化，List<int>.ForEach比一个for-loop还要快！而且List<T>.ForEach比标准的foreach-loop来的快。虽然ForEach方法在迭代List<T>是非常快，但是碰到数组时就不一样了。一维数组没有ForEach方法，而且用ForEach比用foreach来的慢很多。原因是编译器没有为foreach 迭代数组生成IEnumerator<T>代码。用foreach来迭代数组，不会有方法调用，但是Array.ForEach还是会为每次迭代调用一次委托（一个callvirt调用？）。

除此之外，List<T>还有如下使用技巧：

List<T>.Sort() → 排序T

List<T>.Find() → 找出一個T

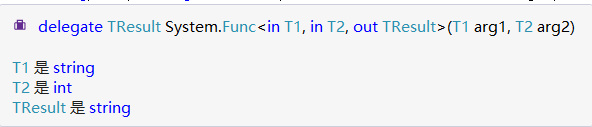
List<T>.FindAll() →找出多個T

List<T>.Exist() →判斷T是否存在

使用时请看具体的提示：

Func<string, int, string> mmF5 = (a, b) => a+"18-1="+(b-1).ToString();

ccMsg.Text += "\n" + mmF5("大家请看：", 18);

****

三个类型分别是T1，T2，都是in输入,TResult是out，即输出。

泛型约束

可以给泛型的类型参数上加约束，可以要求这些类型参数满足一定的条件。

where T: struct类型参数需是值类型

where T : class类型参数需是引用类型

where T : new()类型参数要有一个public的无参构造函数

where T : <base class name>类型参数要派生自某个基类

where T : <interface name>类型参数要实现了某个接口

where T : U这里T和U都是类型参数，T必须是或者派生自U

这些约束，可以同时一起使用：class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new() { // ... }

public static int 数字排序<T>(T x, T y, bool 是否升序) where T : IComparable

{

if (x.CompareTo(y) > 0)

return 是否升序 ? -1 : 1;

else if (x.CompareTo(y) == 0)

return 0;

else

return 是否升序 ? 1 : -1;

}

匿名函数

我们用委托调用方法时，是根据方法名调用的，当需要调用的方法很多时，我们创建很多的方法，但是很多方法又不常用，现在我们可以用匿名方法代替。

ReturnString mmReturn = newReturnString(GetName);

mmReturn += delegate(string s)

{

ccMsg.Text += "匿名函数" + s + "\n";

};

ReturnString2 mmReturn2 = delegate (string s)

{

return ("带返回值的匿名函数：" + s + "\n");

};

mmReturn("Jack");

ccMsg.Text+=mmReturn2("Simmy");

Lambda表达式

简化了匿名委托的使用，让你让代码更加简洁，优雅。

ReturnString mmReturnString = (x) =>{ ccMsg.Text += "Lambda=" + x + "\n"; };

mmReturnString("Jack");

GetSum mmSum = (x, y) => x + y;

ccMsg.Text += "5+1=" + mmSum(5, 1);

事件

当我们使用委托场景时，我们很希望有这样两个角色出现：广播者和订阅者。我们需要这两个角色来实现订阅和广播这种很常见的场景。

广播者这个角色应该有这样的功能：包括一个委托字段，通过调用委托来发出广播。而订阅者应该有这样的功能：可以通过调用 += 和 -= 来决定何时开始或停止订阅。

事件就是描述这种场景模式的一个词。事件是委托的一个子集，为了满足“广播/订阅”模式的需求而生。

声明一个事件很简单，只需在声明一个委托对象时加上event关键字就行。如下：

public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);

publicclass IPhone6

{

public **event** PriceChangedHandler PriceChanged;

}

使用：

if (PriceChanged != null)

PriceChanged(oldPrice, price);

事件和委托的区别

把上面的event关键字拿掉，不就和委托一样，结果不是一样的吗，到底有何不同？

可以用事件的地方就一定可以用委托代替。

但事件有一系列规则和约束用以保证程序的安全可控，事件只有 += 和 -= 操作，这样订阅者只能有订阅或取消订阅操作，没有权限执行其它操作。如果是委托，那么订阅者就可以使用 = 来对委托对象重新赋值（其它订阅者全部被取消订阅），甚至将其设置为null，甚至订阅者还可以直接调用委托，这些都是很危险的操作，广播者就失去了独享控制权。事件保证了程序的安全性和健壮性。

对照字段和属性，委托和事件关系似乎也是如此，它是对委托实例的一层封装，使得客户端不能随意更改或者重置内部的委托列表，只能够允许往列表中增加或者移除委托，使得代码具有更好的封装性。这样来看，事件其实只不过类似声明了一个进行了封装的委托类型变量而已。也有人说事件是委托的一种实例，待验证。

.NET 框架为事件编程定义了一个标准模式。设定这个标准是为了让.NET框架和用户代码保持一致。System.EventArgs是标准模式的核心，它是一个没有任何成员，用于传递事件参数的基类。

Default关键字

default关键字可在switch语句或泛型代码中使用。switch中的用法无需多说。在泛型代码中，用来指定类型参数的默认值。对于引用类型为空，对于值类型为零。

public T Function(arg)

    {

        T temp = default(T);

        if (arg != null) temp = arg.Favs;

        return temp;

    }

因为对于泛型T来说，有可能是值类型，也有可能是引用类型。此关键字对于引用类型会返回空，对于数值类型会返回零。对于结构，此关键字将返回初始化为零或空的每个结构成员，具体取决于这些结构是值类型还是引用类型。

Attribute

定制特性attribute，本质上是一个类，其为目标元素提供关联附加信息，并在运行期以反射的方式来获取附加信息。

http://www.cnblogs.com/atomplus/archive/2009/04/21/1440371.html

进程

进程代表了操作系统上运行着的一个应用程序。进程拥有自己的程序块，拥有独占的资源和数据，并且可以被操作系统调度。But，即使是同一个应用程序，当被强制启动多次时，也会被安放到不同的进程之中单独运行。

线程

线程有时候也被称为轻量级进程，它的概念和进程十分相似，是一个可以被调度的单元，并且维护自己的堆栈和上下文环境。线程是附属于进程的，一个进程可以包含1个或多个线程，并且同一进程内的多个线程共享一块内存块和资源。

进程和线程的区别：一个完整的进程拥有自己独立的内存空间和数据,但是同一个进程内的线程是共享内存空间和数据的。一个进程对应着一段程序,它是由一些在同一个程序里面独立的同时的运行的线程组成的。线程有时也被称为并行运行在程序里的轻量级进程,线程被称为是轻量级进程是因为它的运行依赖与进程提供的上下文环境,并且使用的是进程的资源。

多线程程序作为一种多任务、并发的工作方式，当然有以下的优点：  
　　1) 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。

2) 使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

　　3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

多线程

主要注意两点：1.一个是共享资源的锁定问题；2.一个是异步的显示问题，如果线程最终执行的结果需要在主线程中显示，那就要考虑延迟的问题。

for (int i = 1; i <= 10; i++)

{

ParameterizedThreadStart mmThreadStart = newParameterizedThreadStart(mmLog.ShowInText);

newThread(mmThreadStart).Start(newObjItem { Item1 = i, Item2 = ccMsg });

}

Thread.Sleep(20000);

ccMsg.Text += mmLog.ReturnText(); //主线程写入到控件中，因为各个子线程相当于异步，所以控件显示没有数据，加上Thread.Sleep(20000);延迟一下就可以

Thread的写法：不带参数(ThreadStart)和带参数(ParameterizedThreadStart)。

ParameterizedThreadStart mmThreadStart = newParameterizedThreadStart(mmLog.ShowInText\_lock);

newThread(mmThreadStart).Start(newObjItem { Item1 = i, Item2 = ccMsg });

也可以简写为：

newThread(mmLog.ShowInText\_lock).Start(newObjItem { Item1 = i, Item2 = ccMsg });

Thread的join方法：MSDN：Blocks the calling thread until a thread terminates. 当 a thread 调用Join方法的时候，MainThread 就被停止执行，直到 a thread 线程执行完毕。MainThread 在 NewThread.Join() 被调用后被阻塞，直到 NewThread 执行完毕才继续执行。

死锁

是指两个或两个以上的线程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

产生死锁的原因主要是：  
（1） 因为系统资源不足。  
（2） 进程运行推进的顺序不合适。  
（3） 资源分配不当等。

前台线程和后台线程

前台线程：只有所有的前台线程都结束，应用程序才能结束。默认情况下创建的线程都是前台线程。在VS2010中体现在系统在编译运行时点击结束，在无前台线程的情况下，系统停止运行。在有前台线程的情况下，系统UI消失，但是显示还是编译运行状态，需要手动点击停止状态才会关闭。

后台线程：只要所有的前台线程结束，后台线程自动结束。通过Thread.IsBackground设置后台线程。必须在调用Start方法之前设置线程的类型，否则一旦线程运行，将无法改变其类型。

通过BeginXXX方法运行的线程都是后台线程。

线程池

线程的创建和销毁需要很大的性能开销，在Windows NT内核的操作系统中，每个进程都会包含一个线程池。而在.NET中呢，也有自己的线程池，它是由CLR负责管理的。

线程池相当于一个缓存的概念，在该池中已经存在了一些没有被销毁的线程，而当应用程序需要一个新的线程时，就可以从线程池中直接获取一个已经存在的线程。相对应的，当一个线程被使用完毕后并不会立刻被销毁，而是放入线程池中等待下一次使用。

通过CLR线程池所建立的线程总是默认为后台线程，优先级数为ThreadPriority.Normal。线程池初始化时是没有线程的在线程池里的。线程的初始化与其他线程一样，但是在完成任务以后，该线程不会自行销毁，而是以挂起的状态返回到线程池。直到应用程序再次向线程池发出请求时，线程池里挂起的线程就会再度激活执行任务。这样既节省了建立线程所造成的性能损耗，也可以让多个任务反复重用同一线程，从而在应用程序生存期内节约大量开销。

CLR线程池分为工作者线程(workerThreads)与I/O线程(completionPortThreads)两种:

* 工作者线程是主要用作管理CLR内部对象的运作，**通常用于计算密集的任务。**
* I/O(Input/Output)线程主要用于与外部系统交互信息，如输入输出，CPU仅需在任务开始的时候，将任务的参数传递给设备，然后启动硬件设备即可。等任务完成的时候，CPU收到一个通知，一般来说是一个硬件的中断信号，此时CPU继续后继的处理工作。在处理过程中，CPU是不必完全参与处理过程的，如果正在运行的线程不交出CPU的控制权，那么线程也只能处于等待状态，即使操作系统将当前的CPU调度给其他线程，此时线程所占用的空间还是被占用，而并没有CPU处理这个线程，可能出现线程资源浪费的问题。如果这是一个网络服务程序，每一个网络连接都使用一个线程管理，可能出现大量线程都在等待网络通信，随着网络连接的不断增加，处于等待状态的线程将会很消耗尽所有的内存资源。可以考虑使用线程池解决这个问题。

　　线程池的最大值一般默认为1000、2000。当大于此数目的请求时，将保持排队状态，直到线程池里有线程可用。

使用CLR线程池的工作者线程一般有两种方式：

* 通过ThreadPool.QueueUserWorkItem()方法；
* 通过委托；

不论是通过ThreadPool.QueueUserWorkItem()还是委托，调用的都是线程池里的线程。

通过以下两个方法可以读取和设置CLR线程池中工作者线程与I/O线程的最大线程数。

1. ThreadPool.GetMax(out in workerThreads,out int completionPortThreads)；
2. ThreadPool.SetMax(int workerThreads,int completionPortThreads)；

ThreadPool线程池中有两个重载的静态方法可以直接启动工作者线程：

* ThreadPool.QueueUserWorkItem(waitCallback)；
* ThreadPool.QueueUserWorkItem(waitCallback,Object)；

使用ThreadPool.QueueUserWorkItem：

for (int i = 1; i <= 10; i++)

{

bool mmResult = ThreadPool.QueueUserWorkItem(mmLog.ShowInText\_lock, newObjItem { Item1 = i, Item2 = ccMsg });

if (!mmResult)

{

mmErrorMsg += "线程" + i.ToString() + "分配失败！\n";

}

}

BeginInvoke与EndInvoke委托异步调用线程：

异步调用委托的步骤如下：

1. 建立一个委托对象，通过**IAsyncResult** BeginInvoke(string name,AsyncCallback callback,object state)异步调用委托方法，BeginInvoke方法除最后的两个参数外，其他参数都是与方法参数相对应的。
2. 利用EndInvoke(IAsyncResult--上一步BeginInvoke返回的对象)方法就可以结束异步操作，获取委托的运行结果。

void Done()

{

Thread.Sleep(5000);

}

privatevoid btn简单BeginInvoke\_Click(object sender, EventArgs e)

{

Control.CheckForIllegalCrossThreadCalls = true;

ccMsg.Clear();

MethodInvoker myMethodInvoker = newMethodInvoker(Done);

ccMsg.Text += "开始异步执行……" + Environment.NewLine;

IAsyncResult mmResult = myMethodInvoker.BeginInvoke(null, null); //没有回调函数和参数

for (var i = 0; i < 50; i++)

{

ccMsg.Text += i + 1;

}

ccMsg.Text += Environment.NewLine;

//此处造成阻塞，UI窗体假死

myMethodInvoker.EndInvoke(mmResult);

ccMsg.Text += "异步执行完成！" +Environment.NewLine;

}

如果你将上述代码（BeginInvoke和EndInvoke）运行在一个WinForm程序中，会发现窗体依然陷入”假死”。对于这种情况，你可能会陷入疑惑：之前说异步函数都执行在线程池中，因此可以肯定异步函数的执行不会引起UI线程的忙碌，但为什么窗体依然陷入了”假死”？问题就在于EndInvoke。EndInvoke此时扮演的角色就是”线程锁”，它充当了一个调用线程与异步线程之间的调度器，有时调用线程需要使用异步函数的执行结果，那么调度线程就需要在异步执行完之前一直等待，直到得到结果方可继续运行。EndInvoke一方面负责监听异步函数的执行状况，一方面将调用线程挂起。因此在Win Form环境下，UI线程的”假死”并不是因为线程忙碌造成，而是被EndInvoke”善意的”暂时封锁，它只是为了等待异步函数的完成。

我们可以对EndInvoke总结如下：

　　●  在执行EndInvoke时，调用线程会进入挂起状态，一直到异步函数执行完成。

●  使用EndInvoke可以使应用程序得知异步函数何时执行完毕。

如果将上述写法称为”异步”，你一定觉得这种”异步”徒具其名，虽然知道异步函数何时执行完毕，也得到了异步函数的传值，但我们的调用线程仍然会等待函数执行完毕，在等待过程中线程阻塞，实际上与同步调用无异。

现在我们把问题稍微复杂化，考虑异步函数抛出异常的一种情形。我们需要了解在何处捕捉到异常，是BeginInvoke，还是EndInvoke？甚至是有没有可能无法捕捉异常？答案是EndInvoke。BeginInvoke的工作只是开始线程池对于异步函数的执行工作，EndInvoke则需要处理函数执行完成的所有信息，包括其中产生的异常。

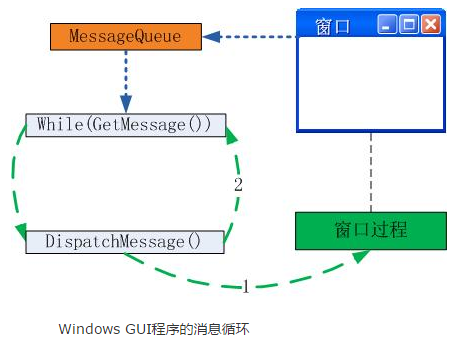
try  
{  
// 应用程序在此处会造成阻塞，直到SleepOneSecond执行完成  
invoker.EndInvoke(tag);  
}  
catch (Exception ex)  
{  
// 此处可以捕捉异常  
MessageBox.Show(ex.Message);  
}

执行以上代码后，你将发现只有在使用EndInvoke时，才会捕捉到异常，否则异常将丢失。需要注意的是，直接在编译器中运行程序是无法产生捕获异常的，只有在Debug、Release环境下运行，异常才会以对话框的形式直接弹出。

临时插一句， 在什么时候我们需要使用Invoke，使用invoke和直接运行有何区别啊？Invoke以同步方式开始执行委托，其参数必须与要执行的方法完全一致。BeginInvoke就是以异步方式开始执行委托，它与您需要执行的方法具有相同的参数，另外还有两个可选参数。简单说，如果一个工作需求大量时间，会阻塞UI线程，而使用INVOKE方式，可以解决这个问题。但是他们仍然在UI线程上，只是处理方式不同。这是在网上找到的答案，需要验证。

BeginInvoke() 调用时，当前线程会启用线程池中的某个线程来执行此方法，当前线程不被阻塞，继续运行后面的代码， Invoke() 调用时，会阻塞当前线程，等到 Invoke() 方法返回才继续执行后面的代码，而这两个方法中执行的是一个委托。且不能使用匿名函数或Lambda表达式。不过可以只用Action或Func,如：this.Invoke(newAction<string>((x) => { ccMsg.Text = x+sum.ToString(); }),"结果=");

Windows GUI程序是基于消息机制的，有个主线程维护着一个消息泵。这个消息泵让windows程序生生不息。



Windows程序有个消息队列，窗体上的所有消息是这个队列里面消息的最主要来源。这里的while循环使用了GetMessage（）这个方法，这是个阻塞方法，也就是队列为空时方法就会被阻塞，从而这个while循环停止运动，这避免了一个程序把cpu无缘无故地耗尽，让其它程序难以得到响应。

这个主线程维护着整个窗体以及上面的子控件。当它得到一个消息，就会调用DispatchMessage方法派遣消息，这会引起对窗体上的窗口过程的调用。窗口过程里面当然是程序员提供的窗体数据更新代码和其它代码。

Dotnet窗体程序封装了上述的while循环，这个循环就是通过Application.Run方法启动的。

如果从另外一个线程操作windows窗体上的控件，就会和主线程产生竞争，造成不可预料的结果，甚至死锁。因此windows GUI编程有一个规则，就是只能通过创建控件的线程来操作控件的数据，否则就可能产生不可预料的结果。

因此，dotnet里面，为了方便地解决这些问题，Control类实现了ISynchronizeInvoke接口，提供了Invoke和BeginInvoke方法来提供让其它线程更新GUI界面控件的机制。

如果从线程外操作windows窗体控件，那么就需要使用Invoke或者BeginInvoke方法，通过一个委托把调用封送到控件所属的线程上执行。

Invoke或者BeginInvoke方法都需要一个委托对象作为参数。委托类似于回调函数的地址，因此调用者通过这两个方法就可以把需要调用的函数地址封送给界面线程。这些方法里面如果包含了更改控件状态的代码，那么由于最终执行这个方法的是界面线程，从而避免了竞争条件，避免了不可预料的问题。如果其它线程直接操作界面线程所属的控件，那么将会产生竞争条件，造成不可预料的结果。

使用Invoke完成一个委托方法的封送，就类似于使用SendMessage方法来给界面线程发送消息，是一个同步方法。也就是说在Invoke封送的方法被执行完毕前，Invoke方法不会返回，从而调用者线程将被阻塞。

使用BeginInvoke方法封送一个委托方法，类似于使用PostMessage进行通信，这是一个异步方法。也就是该方法封送完毕后马上返回，不会等待委托方法的执行结束，调用者线程将不会被阻塞。但是调用者也可以使用EndInvoke方法或者其它类似WaitHandle机制等待异步操作的完成。

但是在内部实现上，Invoke和BeginInvoke都是用了PostMessage方法，从而避免了SendMessage带来的问题。而Invoke方法的同步阻塞是靠WaitHandle机制来完成的。

Control的Invoke和BeginInvoke的参数为delegate，委托的方法是在Control的线程上执行的，也就是我们平时所说的UI线程。这种方法也可以解决我们出现的在另外一个线程中控制UI线程中的Control输出的问题。

Delegate.BeginInvoke方法是从ThreadPool取出一个线程来执行这个方法，以获得异步执行效果的。也就是说，如果采用这种方式提交多个异步委托，那么这些调用的顺序无法得到保证。而且由于是使用线程池里面的线程来完成任务，使用频繁，会对系统的性能造成影响。

这里需要纠正一个误区，那就是Control类上的异步调用BeginInvoke并没有开辟新的线程完成委托任务，而是让界面控件的所属线程完成委托任务的。看来异步操作就是开辟新线程的说法不一定准确。

可以看下面一个例子，可以理解得更深刻：

privatevoid btn跨UI线程调用控件\_Click(object sender, EventArgs e)

{

Control.CheckForIllegalCrossThreadCalls = true;

ccMsg.Clear();

//Done("Mally"); //主线程上执行，肯定要阻塞。

//不是从创建控件的线程调用它

Thread mmThread = newThread(newParameterizedThreadStart(Done));

mmThread.Start("Jimmy");

//Control.CheckForIllegalCrossThreadCalls = false;

}

void Done(object text)

{

Thread.Sleep(5000);

//不是从创建控件的线程调用它

//ccMsg.Text = "新线程"+Thread.CurrentThread.Name+"中调用，参数："+text.ToString()+"！" + Environment.NewLine;

//解决办法：

this.Invoke(newWriteInvoke(WriteInText), "Jimmy");

}

void WriteInText(string text)

{

ccMsg.Text += "Begin……"; //Begin之前是没有问题的，UI界面正常操作

Application.DoEvents();

Thread.Sleep(5000); //这时UI界面无法正常操作，也印证了执行ccMsg.Text += text;已经变成了UI线程

ccMsg.Text += text;

}

delegatevoidWriteInvoke(string text);

特别说明下，调用control和Invoke方法里用到的control只要线程相同就行，如某窗体中(this代表该Form对象)：

this.Invoke(newAction<string>((x) => { ccMsg.Text = x+sum.ToString(); }),"结果=");

ccMsg.Invoke(newAction<string>((x) => { ccMsg.Text = x + sum.ToString(); }), "结果=");

btnTask线程池.Invoke(newAction<string>((x) => { ccMsg.Text = x + sum.ToString(); }), "结果=");

执行结果是一样的。

原理是Control.BeginInvoke and Control.Invoke通过FindMarshalingControl方法，通过一个循环向上回溯，从当前控件开始回溯父控件，直到找到最顶级的父控件，用它作为封送对象。例如，我们调用窗体上一个进度条的Invoke方法封送委托，但是实际上会回溯到主窗体，通过这个控件对象来封送委托。因为主窗体是主线程消息队列相关的，发送给主窗体的消息才能发送到界面主线程消息队列。

InvokeRequired属性是每个Control对象都具有的属性，它会返回true和false，当是true的时候，表示它在另外一个线程上面，这是必须通过Invoke,BeginInvoke这些方法来调用更新UI对象的方法，当是false的时候，有两种情况，1：位于当前线程上面，可以通过直接去调用修改UI对象的方法，2：位于不同的线程上，不过控件或窗体的句柄不存在。对于句柄是否存在的判断，可以通过IsHandleCreated来获取，如果句柄不存在，是不能调用Invoke...这些方法的，这时候你必须等待句柄的创建。

上面的例子中，我们觉得这种”异步”徒具其名，如果想知道异步操作什么时候执行完，并执行后续的操作，我们可以利用IAsyncResult对象的回调函数来实现。

delegateList<string>VoidInvoke(string[] name, refstring msg);

List<string>GetList(string[] name,refstring msg)

{

Thread.Sleep(10000);

msg = "操作成功";

List<string> mmlist = newList<string> { };

for (var i = 0; i < name.Length; i++)

mmlist.Add(name[i]);

return mmlist;

}

privatevoid btn带回调的BeginInvoke\_Click(object sender, EventArgs e)

{

Control.CheckForIllegalCrossThreadCalls = true;

ccMsg.Clear();

VoidInvoke myVoidInvoke = newVoidInvoke(GetList);

string mmMsg = string.Empty;

myVoidInvoke.BeginInvoke(newstring[] { "Tom","Lu","Pie","Jimmy","Oliver"},ref mmMsg,newAsyncCallback(LogTableCallBack),null);

ccMsg.Text += "主程序结束调用！";

}

privatevoid LogTableCallBack(IAsyncResult tag)

{

AsyncResult result = (AsyncResult)tag;

VoidInvoke myVoidInvoke = (VoidInvoke)result.AsyncDelegate;

string msg = string.Empty;

List<string> mmList = myVoidInvoke.EndInvoke(ref msg, tag);

if (ccMsg.InvokeRequired)

{

ccMsg.Invoke(newAction(()=> {

ccMsg.Clear();

ccMsg.Text = msg + Environment.NewLine;

for (var i = 0; i < mmList.Count; i++)

{

ccMsg.Text += i.ToString() + Environment.NewLine;

}

}));

}

}

因此，InvokeRequired属性用来判断windows窗体线程和当前的调用者线程是否是同一个，如果是同一个就没有必要封送了，直接访问这个GUI控件吧。否则，就不要那么直接表白了，就需要Invoke或者BeginInvoke做媒了。

我们可以认为回调函数实际上是异步函数的一种延续。

详情请参见《[C#异步调用的实现机制及调用方法](http://www.cnblogs.com/longle/archive/2011/12/03/2274922.html)》(<http://www.cnblogs.com/longle/archive/2011/12/03/2274922.html>)

### Task

使用Task：任务(Task)是一个管理并行工作单元的轻量级对象。它通过使用CLR的线程池来避免启动专用线程，可以更有效率的利用线程池。

var mmTasks = newTask[10];

for (int i = 1; i <= 10; i++)

{

mmTasks[i - 1] = newTask(mmLog.ShowInText\_lock, newObjItem { Item1 = i, Item2 = ccMsg });

mmTasks[i - 1].Start();

}

//Task.WaitAny(mmTasks); //Task.WaitAny()静态方法等待任何一个任务完成。

Task.WaitAll(mmTasks); //Task.WaitAll()静态方法等待所有任务完成。

或者：

var mmTasks = newTask[10];

for (int i = 1; i <= 10; i++)

{

mmTasks[i - 1] = Task.Run(()=> {

mmLog.ShowInText\_lock(newObjItem { Item1 = i, Item2 = ccMsg });

});

}

Task.WaitAll(mmTasks); //Task.WaitAll()静态方法等待所有任务完成。

ccMsg.Text += mmLog.ReturnText() + mmErrorMsg;

带返回值的Task：

var mmTasks = newTask<int>[10];

for (int i = 1; i <= 10; i++)

{

mmTasks[i-1] = Task.Run(() => {

mmLog.ShowInText\_lock(newObjItem { Item1 = i, Item2 = ccMsg });

Thread.Sleep(1000);

return i-1;

});

int j = mmTasks[i - 1].Result;

ccMsg.Text += "执行Task[" + j.ToString() + "]"+Environment.NewLine;

}

Task.WaitAll(mmTasks);

ccMsg.Text += mmLog.ReturnText() + mmErrorMsg;

线程池使得线程可以充分有效地被使用，减少了任务启动的延迟。但是不是所有的情况都适合使用线程池中的线程。

任务(Task)是一个管理并行工作单元的轻量级对象。它通过使用CLR的线程池来避免启动专用线程，可以更有效率的利用线程池。

对于Task.Run来说传入方法不能有参数，可以有返回值。要获得结果，要在Run()（返回Task<T>类型）之后调用Task<T>类型的Result属性获取。可以看出，获取结果时，Task是会阻塞当前进程的，等待线程执行完毕才继续。也就是说，Result的关系，因此线程将等待result得到后再往下进行。

Task同步运行RunSynchronously：

1. Task<**int**>  task = CreateTask("Task 2");
2. task.RunSynchronously(); //运行在主线程中，等同于直接运行： TaskMethod("Task 2");

这里没有使用Start()而是使用RunSynchronously()，直接同步运行！等同于直接运行 Result = TaskMethod("Task 2");因此，该Task不是运行在线程池中，而是运行在主线程中。

异步

详见《走进异步编程的世界 - 开始接触 async await(已验证)》(http://www.cnblogs.com/liqingwen/p/5831951.html)。

privateasyncTask<int> CountCharactersAsync(int id, string address)

{

var wc = new System.Net.Http.HttpClient();

ccMsg.Text += "开始调用id=" + id.ToString() + ":" + watch.ElapsedMilliseconds.ToString() + "ms\n";

var result = wc.GetStringAsync(address);

var mmReturn=(await result).Length; //关键是await

ccMsg.Text += "调用完成id=" + id.ToString() + ":" + watch.ElapsedMilliseconds.ToString() + "ms\n";

return mmReturn;

}

privatevoid btn简单异步\_Click(object sender, EventArgs e)

{

watch.Start();

conststring url1 = "http://www.cnblogs.com/";

conststring url2 = "http://www.cnblogs.com/liqingwen/";

var result1 = CountCharactersAsync(1, url1);

var result2 = CountCharactersAsync(2, url2);

for (var i = 0; i < 3; i++)

{

ExtraOperation(i + 1);

}

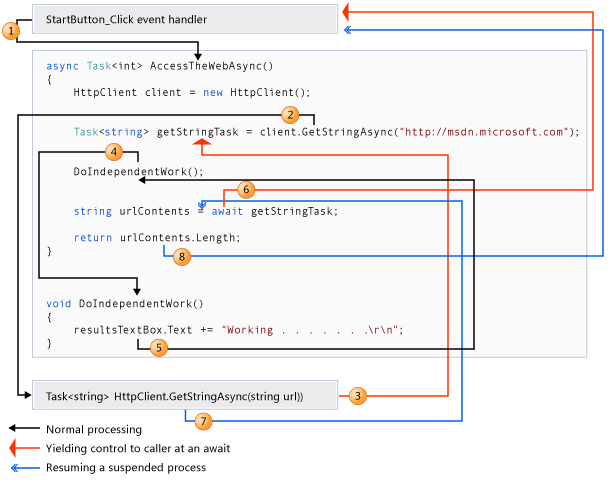
ccMsg.Text += "[" + url1 + "]的字符个数：" + result1.Result + "\n";

ccMsg.Text += "[" + url2 + "]的字符个数：" + result2.Result + "\n";

}

如上，Task<T>：调用方法要从调用中获取一个 T 类型的值，异步方法的返回类型就必须是Task<T>。调用方法从 Task 的 Result 属性(result1.Result)获取的就是 T 类型的值。

异步编程中最需弄清的是控制流是如何从方法移动到方法的。 下图可引导你完成该过程。



关系图中的数值对应于以下步骤。

1. 事件处理程序调用并等待 AccessTheWebAsync 异步方法。
2. AccessTheWebAsync 可创建 [HttpClient](https://msdn.microsoft.com/zh-cn/library/system.net.http.httpclient(v=vs.120).aspx) 实例并调用 [GetStringAsync](https://msdn.microsoft.com/zh-cn/library/hh551746(v=vs.120).aspx) 异步方法以下载网站内容作为字符串。
3. **GetStringAsync** 中发生了某种情况，该情况挂起了它的进程。 可能必须等待网站下载或一些其他阻止活动。 为避免阻止资源，**GetStringAsync** 会将控制权出让给其调用方 AccessTheWebAsync。

**GetStringAsync** 返回 [Task](https://msdn.microsoft.com/zh-cn/library/dd321424(v=vs.120).aspx)，其中 TResult 为字符串，并且 AccessTheWebAsync 将任务分配给 getStringTask 变量。 该任务表示调用 **GetStringAsync** 的正在进行的进程，其中承诺当工作完成时产生实际字符串值。

1. 由于尚未等待 getStringTask，因此，AccessTheWebAsync 可以继续执行不依赖于 **GetStringAsync** 得出的最终结果的其他工作。 该任务由对同步方法 DoIndependentWork 的调用表示。
2. DoIndependentWork 是完成其工作并返回其调用方的同步方法。
3. AccessTheWebAsync 已用完工作，可以不受 getStringTask 的结果影响。 接下来，AccessTheWebAsync 需要计算并返回该下载字符串的长度，但该方法仅在具有字符串时才能计算该值。

因此，AccessTheWebAsync 使用一个 await 运算符来挂起其进度，并把控制权交给调用 AccessTheWebAsync 的方法。AccessTheWebAsync 将 **Task(Of Integer)** 或 **Task<int>** 返回至调用方。 该任务表示对产生下载字符串长度的整数结果的一个承诺。

|  |
| --- |
| **说明 说明** |
| 如果 GetStringAsync（因此 getStringTask）在 AccessTheWebAsync 等待前完成，则控件会保留在 AccessTheWebAsync 中。 如果异步调用过程 (getStringTask) 已完成，并且 AccessTheWebSync 不必等待最终结果，则挂起然后返回到 AccessTheWebAsync 将造成成本浪费。 |

在调用方内部（此示例中的事件处理程序），处理模式将继续。 在等待结果前，调用方可以开展不依赖于 AccessTheWebAsync 结果的其他工作，否则就需等待片刻。事件处理程序等待 AccessTheWebAsync，而 AccessTheWebAsync 等待 **GetStringAsync**。

1. **GetStringAsync** 完成并生成一个字符串结果。 字符串结果不是通过按你预期的方式调用 **GetStringAsync** 所返回的。（请记住，此方法已在步骤 3 中返回一个任务。）相反，字符串结果存储在表示完成方法 getStringTask 的任务中。 await 运算符从 getStringTask 中检索结果。 赋值语句将检索到的结果赋给 urlContents。
2. 当 AccessTheWebAsync 具有字符串结果时，该方法可以计算字符串长度。 然后，AccessTheWebAsync 工作也将完成，并且等待事件处理程序可继续使用。 在此主题结尾处的完整示例中，可确认事件处理程序检索并打印长度结果的值。

如果你不熟悉异步编程，请花 1 分钟时间考虑同步行为和异步行为之间的差异。 当其工作完成时（第 5 步）会返回一个同步方法，但当其工作挂起时（第 3 步和第 6 步），异步方法会返回一个任务值。 在异步方法最终完成其工作时，任务会标记为已完成，而结果（如果有）将存储在任务中。

<https://msdn.microsoft.com/zh-cn/library/hh191443(v=vs.120).aspx>

async 和 await 关键字不会导致创建其他线程。 因为异步方法不会在其自身线程上运行，因此它不需要多线程。 只有当方法处于活动状态时，该方法将在当前同步上下文中运行并使用线程上的时间。 可以使用 [Task.Run](https://msdn.microsoft.com/zh-cn/library/hh195051(v=vs.120).aspx) 将占用大量 CPU 的工作移到后台线程，但是后台线程不会帮助正在等待结果的进程变为可用状态。

async 和 await 关键字的使用也可以参见《async & await 的前世今生》(<http://www.cnblogs.com/jesse2013/p/async-and-await.html>)，里面有关线程和异步也是讲得挺好的。

个人认为，控制台的入口方法不支持async,所有我们在入口方法里面不能 用 await，因此我们可以这么来：

static async Task Test(){

    // 方法打上async关键字，就可以用await调用同样打上async的方法

    // await 后面的方法将在另外一个线程中执行

    await GetName();

}

static void Main(string[] args){

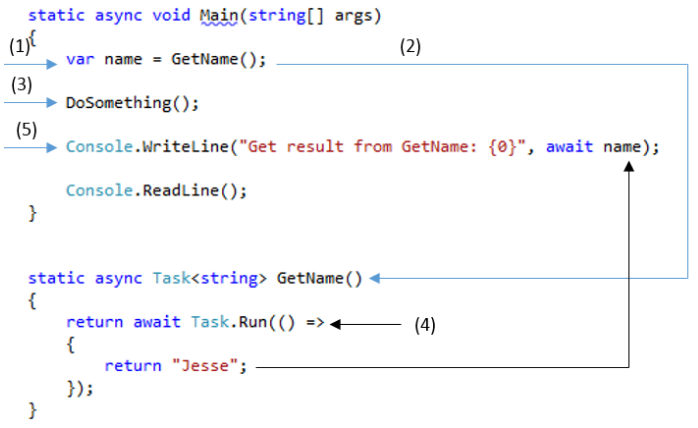
    Test(); // 这个方法其实是多余的, 本来可以直接写下面的方法

    // await GetName()

    // 但是由于控制台的入口方法不支持async,所有我们在入口方法里面不能 用 await

    Console.WriteLine("Current Thread Id :{0}", Thread.CurrentThread.ManagedThreadId);

}



1. 进入主线程开始执行
2. 调用async方法，返回一个Task，注意这个时候另外一个线程已经开始运行，也就是**GetName里面的 Task** 已经开始工作了
3. 主线程继续往下走
4. 第3步和第4步是同时进行的，主线程并没有挂起等待
5. 如果另一个线程已经执行完毕，name.IsCompleted=true，主线程仍然不用挂起，直接拿结果就可以了。如果另一个线程还同有执行完毕, name.IsCompleted=false，那么主线程会挂起等待，直到返回结果为止。

而且，await并不是针对于async的方法，而是针对async方法所返回给我们的Task，这也是为什么所有的async方法都必须返回给我们Task。

Task.GetAwaiter()方法会给我们返回一个awaitable的对象，通过调用这个对象的GetResult方法就会挂起主线程，当然也不是所有的情况都会挂起。还记得我们Task的特性么？ 在一开始的时候就启动了另一个线程去执行这个Task，当我们调用它的结果的时候如果这个Task已经执行完毕，主线程是不用等待可以直接拿其结果的，如果没有执行完毕那主线程就得挂起等待了。

await 实质是在调用awaitable对象的GetResult方法。

### Parallel

这个函数的功能和Task有些相似，就是并发执行一系列任务，然后等待所有完成。和Task比起来，省略了Task.WaitAll这一步，自然也缺少了Task的相关管理功能。它有两种形式:

　　Parallel.Invoke( params Action[] actions);

　　Parallel.Invoke(Action[] actions,TaskManager manager,TaskCreationOptions options);

private static bool IsPrimeNumber(int number)

{

if (number < 1)

{

return false;

}

if (number == 1 && number == 2)

{

return true;

}

for (int i = 2; i < number; i++)

{

if (number % i == 0)

{

return false;

}

}

return true;

}

private void btnParallel\_Click(object sender, EventArgs e)

{

DateTime start = DateTime.Now;

for (int i = 0; i <= 100; i++)

{

bool b = IsPrimeNumber(i);

Thread.Sleep(100);

}

DateTime end = DateTime.Now;

TimeSpan timespan = end - start; //10秒左右

start = DateTime.Now;

Parallel.For(1, 100, (i) =>

{

bool b = IsPrimeNumber(i);

Thread.Sleep(100);

});

end = DateTime.Now;

timespan = end - start; //2秒左右

}

}

上例可以看出并行编程的效果。

<https://www.cnblogs.com/scmail81/p/9521096.html>

有关线程的一些问题

多线程有什么用？

（1）发挥多核CPU的优势

随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的，4核、8核甚至16核的也都不少见，如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU上就浪费了75%。 单核CPU上所谓的"多线程"那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程"同时"运行罢了 。多核CPU上的多线程才是真正的多线程，它能让你的多段逻辑同时工作，多线程，可以真正发挥出多核CPU的优势来，达到充分利用CPU的目的。

（2）防止阻塞

从程序运行效率的角度来看，单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核CPU我们还是要应用多线程，就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间(比如远程连接web service服务)，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

（3）便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

迭代器(iterator)：迭代器模式使得你能够获取到序列中的所有元素而不用关心是其类型是array，list，linked list或者是其他什么序列结构。这一点使得能够非常高效的构建数据处理通道(data pipeline)--即数据能够进入处理通道，进行一系列的变换，或者过滤，然后得到结果。

在.NET中，迭代器模式被IEnumerator和IEnumerable及其对应的泛型接口所封装。如果一个类实现了IEnumerable接口，那么就能够被迭代；调用GetEnumerator方法将返回IEnumerator接口的实现，它就是迭代器本身。迭代器类似[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)中的游标，他是数据序列中的一个位置记录。迭代器只能向前移动，同一数据序列中可以有多个迭代器同时对数据进行操作。

在[C#](http://lib.csdn.net/base/csharp" \t "_blank" \o "C#知识库)1中已经内建了对迭代器的支持，那就是foreach语句。使得能够进行比for循环语句更直接和简单的对集合的迭代，编译器会将foreach编译来调用GetEnumerator和MoveNext方法以及Current属性，如果对象实现了IDisposable接口，在迭代完成之后会释放迭代器。

可以在不实现IEnumerable就能使用foreach语句，在编译器碰到yield return时，它会自动生成IEnumerable 接口的方法。在实现迭代器的方法或属性中，返回类型必须是IEnumerable, IEnumerator, IEnumerable<T>，或 IEnumerator<T>。迭代器使得遍历一些零碎数据的时候很方便，不用去实现Current, MoveNext 这些方法。

例子：

//如果不实现接口IDisposable，会提示 using 语句中使用的类型必须可隐式转换为"System.IDisposable"

classMusicTitle :IDisposable

{

publicstring[] names = { "Tubular Bells", "Hergest Ridge", "Ommadawn", "Platinum", "Lemon Tree" };

publicIEnumerator<string> GetEnumerator()

{

for (var i = 0; i < names.Length; i++)

{

yieldreturn names[i];

}

}

publicIEnumerable<string> Reverse()

{

for (var i = names.Length - 1; i >= 0; i--)

{

yieldreturn names[i];

}

}

///<summary>

///

///</summary>

///<param name="index">起始位置，从0开始</param>

///<param name="length"></param>

///<returns></returns>

publicIEnumerable<string> Subset(int index, int length)

{

for (var i = index; i < index + length; i++)

{

yieldreturn"姓名：" + names[i];

}

}

privateIntPtr handle;

privateBook component = newBook(); //这里实例化了一个对象

///<summary>

///解决：MusicTitle不实现接口成员IDisposable.Dispose

///</summary>

publicvoid Dispose()

{

Dispose(true);

//GC.SuppressFinalize(this); 表示gc回收对象的时候不执行析构函数，避免dispose 释放非托管资源后，gc 回收对象的时候再执行析构函数回收非托管资源。

GC.SuppressFinalize(this);

}

protectedvirtualvoid Dispose(bool disposing)

{

if (disposing)

{

// 销毁托管资源

//component.Dispose(); 某些类可以在此处手动释放。

}

// 这里销毁非托管资源

CloseHandle(handle);

handle = IntPtr.Zero;

}

[System.Runtime.InteropServices.DllImport("Kernel32")]

privateexternstaticBoolean CloseHandle(IntPtr handle);

}

IQueryable接口与IEnumberable接口的区别

IEnumerable<T> 泛型类在调用自己的SKip 和 Take 等扩展方法之前数据就已经加载在本地内存里了，而IQueryable<T> 是将Skip ,take 这些方法表达式翻译成T-SQL语句之后再向SQL服务器发送命令，它并不是把所有数据都加载到内存里来才进行条件过滤。

延迟加载Lazy<T>

在项目开发中，经常会遇到特定的对象使用的加载问题，有的实例对象我们创建之后并非需要使用，只是根据业务场景来调用，所以可能会导致很多无效的实例加载, 延迟初始化出现于.NET 4.0，主要用于提高性能，避免浪费计算，并减少程序内存要求。也可以称为，按需加载。

private void btn延迟加载\_Click(object sender, EventArgs e)

{

ccMsg.Clear();

Lazy<UserTemplate> mmUserTemplate = new Lazy<UserTemplate>();

//ccMsg.Text += mmUserTemplate.Value.Text();

ccMsg.Text += "开始给属性赋值：" + Environment.NewLine;

mmUserTemplate.Value.Id = 1; //只有在此时才执行构造函数，也就是用到才实例化

ccMsg.Text += mmUserTemplate.Value.Text() + Environment.NewLine;

mmUserTemplate.Value.Name = "Jimmy";

ccMsg.Text += "结束给属性赋值：" + Environment.NewLine;

}

private void btn非延迟加载\_Click(object sender, EventArgs e)

{

ccMsg.Clear();

UserTemplate mmUserTemplate = new UserTemplate();

//ccMsg.Text += mmUserTemplate.Text();

ccMsg.Text += "开始给属性赋值：" + Environment.NewLine;

mmUserTemplate.Id = 1;

ccMsg.Text += mmUserTemplate.Text() + Environment.NewLine;

mmUserTemplate.Name = "Jimmy";

ccMsg.Text += "结束给属性赋值：" + Environment.NewLine;

}

class UserTemplate

{

/// <summary>

/// 如果想用初始化器，并且重写了UserTemplate构造函数，如没有UserTemplate()构造函数，则需要是实现一个。

/// </summary>

public UserTemplate()

{

}

public UserTemplate(System.Windows.Forms.TextBoxBase ppText)

{

ppText.Text += "UserTemplate执行初始化！";

}

public string Name { get; set; }

public int Id { get; set; }

public string Tip { get; set; }

internal string Text()

{

return Tip;

}

}

文件的操作

修改文件时间：

File.SetCreationTime(mmFileName, DateTime.Parse(ccData.Text));

File.SetLastWriteTime(mmFileName, DateTime.Parse(ccData.Text));

File.SetLastAccessTime(mmFileName, DateTime.Parse(ccData.Text));

LINQ

语言集成查询, 通过LINQ，我们可以使用相同API操作不同的数据源。这种API有点类似SQL的形式。

必须实现IEnumerable接口。

LINQ关联的技术：

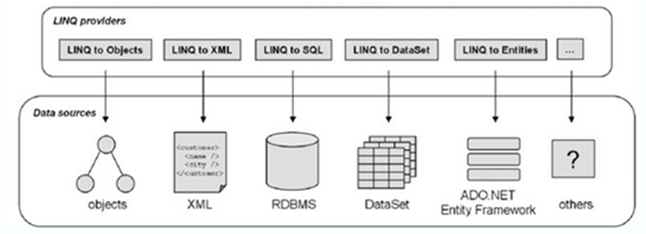
(1)隐式类型、匿名类型、对象初始化器;

(2)Lambda表达式

(3)扩展方法

(4)Yield迭代器，延迟计算

(5)表达式树: 表达式树允许在运行期间建立对数据源的查询，因为表达式树存储在程序集中。



从这幅图中，我们可以知道LINQ包括五个部分：LINQ to Objects、LINQ to XML、LINQ to SQL、LINQ to DataSet、LINQ to Entities。

目前，还可以下载其他第三方提供程序，例如[LINQ to JSON](http://wenku.baidu.com/link?url=hJ-1uqu9qyWG8sEbNXpexAQ1ZQt6HVMdXMjGlidgWYytrHHpmQ_m22x6ZPVlVlMsvdKeMSR-McihLOisZD_bfev8olwp027KWYr2ejtjQTK)、LINQ to MySQL、LINQ to Amazon、LINQ to Flickr和LINQ to SharePoint。无论使用什么数据源，都可以通过LINQ使用相同的API进行操作。

LINQ查询时有两种语法可供选择：查询表达式（Query Expression）和方法语法（Fluent Syntax）。

.NET公共语言运行库（CLR）并不具有查询表达式的概念。所以，编译器会在程序编译时把查询表达式转换为方法语法，即对扩展方法的调用。所以使用方法语法会让我们更加接近和了解LINQ的实现和本质，并且一些查询只能表示为方法调用。但另一方面，查询表达式通常会比较简单和易读。不管怎样，这两种语法是互相补充和兼容的，我们可以在一个查询中混合使用查询表达式和方法语法。

**LINQ to Objects：**

int[] arr = new int[] { 8, 5, 89, 41, 1, 2, 3, 65, 1 }

var m = from t in arr where t < 5 orderby t descending select t;

int mmCount = 0;

foreach (var j in m)

{

ccMsg.Text += j;

ccMsg.Text += mmCount == m.Count<int>() - 1 ? "" : ",";

mmCount++;

}

List<Item> mmList = new List<Item>

{

new Item {I= "刘云",V= "女"},

new Item {I= "张伟",V= "男"},

new Item {I= "韩伟",V= "女"},

new Item {I= "王兰",V= "女"},

new Item {I= "赵楠",V= "男"}

};

var m = from t in mmList where t.V == "女" orderby t.I select t.I;

m.Count<string>();

var n = from t in mmList where t.V == "男" orderby t.I select t;

n.Count<Item>() – 1;

n.Count<Item>()，也可以使用n.Count ()

**删除重复项：**

var n = (from t in mmList where t.V == "男" orderby t.I select t.I).Distinct().ToList();

**Select：**Select 将序列的每个元素经过lambda表达式处理后投影到一个新类型元素上。

**SelectMany：**

c#编译器会把“复合from子句”的查询表达式转换为SelectMany()扩展方法。

将序列的每个元素经过lambda表达式处理后投影到一个 IEnumerable<TResult>，再将多个IEnumerable<TResult>序列合并为一个返回序列IEnumerable<TResult>。

List<Bouquet> bouquets = new List<Bouquet> {

new Bouquet{ flowers=new List<string>{ "sunflower", "daisy", "daffodil", "larkspur" } },

new Bouquet{ flowers=new List<string>{ "tulip", "rose", "orchid" } },

new Bouquet{ flowers=new List<string>{ "gladiolis", "lily", "snapdragon", "aster", "protea" } },

new Bouquet{ flowers=new List<string>{ "larkspur", "lilac", "iris", "dahlia" } }

};

IEnumerable<List<string>> lists = bouquets.Select(bq => bq.flowers);

ccMsg.Text += "下面是Select的用法：" + Environment.NewLine;

int i = 0;

int j = 0;

foreach (List<string> flowers in lists)

{

foreach (string flower in flowers)

{

ccMsg.Text += flower;

ccMsg.Text += i == lists.Count()-1 && j == flowers.Count-1 ? "" : ",";

j++;

}

i++;

}

ccMsg.Text += Environment.NewLine;

ccMsg.Text += "下面是SelectMany的用法：" + Environment.NewLine;

IEnumerable<string> flowerlist = bouquets.SelectMany(bq => bq.flowers);

i = 0;

foreach (string flower in flowerlist)

{

ccMsg.Text += flower;

ccMsg.Text += i == flowerlist.Count() - 1 ? "" : ",";

i++;

}

**into:**可以使用 into 上下文关键字创建一个临时标识符，以便将 group、join 或 select 子句的结果存储到新的标识符中。

var items = from trans in transports

join vehs in vehicles on trans.How equals vehs.vehicleHow

into lst

select new {

Name=trans.Name,

list=lst

};

**Group:** 按某字段归类。

var productlist = from i in

(

from p in products

join pc in productcategorys

on p.ProductCategoryID equals pc.ProductCategoryID

select new

{

Name = p.Name,

Price = p.Price,

ProductCategoryName = pc.ProductCategoryName,

ProductCategoryID = pc.ProductCategoryID

}

)

group i by i.ProductCategoryName into groupChild

select new

{

ProductCategoryName = groupChild.Key,

MaxPrice = groupChild.Max(x => x.Price),

MinPrice = groupChild.Min(x => x.Price),

AveragePrice = groupChild.Average(x => x.Price),

SumPrice = groupChild.Sum(x => x.Price)

};

如果需要按多个字段归类：

var wflist = from u in db.TWelFare

where u.fy\_no == fy\_no

orderby u.we\_no

group u by new { weno = u.we\_no, wename = u.we\_name } into g

select new { g.Key.weno, g.Key.wename };

字符串分割Split

<https://www.cnblogs.com/yu6688/p/13447774.html>

多字符分隔

string s="abcdeabcdeabcde";

string[] sArray1=s.Split(new char[3]{'c','d','e'}) ;

foreach(string i in sArray1)

Console.WriteLine(i.ToString());

使用正则表达式

using System.Text.RegularExpressions;  
string str="aaajsbbbjsccc";  
string[] sArray=Regex.Split(str,"js",RegexOptions.IgnoreCase);  
foreach (string i in sArray) Response.Write(i.ToString());

数组

初始化并赋值，无需指定数组长度

string[] fa = new string[] { "1","2"};

也可以直接作为参数传递

var a = myMethod("", new string[] { "1", "2" });

ccMsg.Text += "共：" + a + "个params参数！" + Environment.NewLine;

作为方法myMethod，它的声明如下：

int myMethod(string mmMsg,params string[] names){……}

或

int myMethod(string mmMsg, string[] names){……}

多个string的if判断

if (!new string[] { "子模块", "自定义变量", "系统变量" }.Contains(e.Node.Text))

实用计算

三角函数

int a = 3, b = 4, c = 5;

double xielenth=Math.Sqrt(a \* a + b \* b);//勾股定理求更号

double ca = Math.Atan2(b, a) \* 180 / Math.PI; //求线段间的角度

类

属性添加注释

Form里的属性：

[Description("容器分辨率"), Category("自定义")]

public Size PanelSize

{

get { return \_PanelSize; }

set { \_PanelSize = value; }

}

XML注释，一般的类也可以用：

/// <summary>

/// 容器分辨率

/// </summary>

[Description("容器分辨率"), Category("自定义")]

public Size PanelSize

{

get { return \_PanelSize; }

set { \_PanelSize = value; }

}

List

初始化

public static List<Color> colorlist = new List<Color> {Color.Red,Color.Green,Color.Yellow,Color.Blue,Color.Orange };

对象初始化：

var books = new List<Book>

{

new Book(){Name="C# learning guide",Year=2015}, //两种写法都行

new Book{Name="C# step by step",Year=2014}

};

combox绑定List

Common.list\_性别.Insert(0, new Item { Key = "", Value = "" });

cb性别.DataSource = Common.list\_性别;

cb性别.ValueMember = "Key";

cb性别.DisplayMember = "Value";

list查询和遍历

使用First代表一定能找到，找不到就抛出异常，必须使用try……catch……进行异常的捕获。

推荐使用  FirstOrDefault 或Find。

List<Item> mmList=Common.list\_地区.Where(item=>item.Key.EndsWith("0000")).ToList<Item>();

mmList.ForEach(item =>

{

ccTree.Nodes.Add(new TreeNode { Name = item.Key, Text = item.Value });

});

foreach (TreeNode node in ccTree.Nodes)

{

mmList = Common.list\_地区.Where(item => !item.Key.EndsWith("0000") && item.Key.StartsWith(node.Name.Substring(0, 2))).ToList<Item>();

mmList.ForEach(item =>

{

node.Nodes.Add(new TreeNode { Name = item.Key, Text = item.Value });

});

}

List增加数据

public static List<Item> list\_地区 = new List<Item> {

new Item{Key="130000",Value="河北省"},

new Item{Key="130100",Value="石家庄市"},

new Item{Key="130200",Value="唐山市"},

new Item{Key="130300",Value="秦皇岛市"},

new Item{Key="130400",Value="邯郸市"},

new Item{Key="130500",Value="邢台市"},

new Item{Key="320000",Value="江苏省"},

new Item{Key="320100",Value="南京市"},

new Item{Key="320200",Value="无锡市"},

new Item{Key="320300",Value="徐州市"},

new Item{Key="320400",Value="常州市"},

new Item{Key="320500",Value="苏州市"}

};

list查重

var tags = listTag.GroupBy(i => i).Where(g => g.Count() > 1).ToList();

            foreach (var r in tags)

            {

                int index = listTag.IndexOf(r.Key);

                //errorList.Add(new ErrorItem { Column = 0, Row = index, ErrorMsg = "编号列中不允许包含重复的值:" + r.Key + "！" });

            }

list去重

List<string> list1 = dtSource.AsEnumerable().Select(item => "'" + item.Field<string>("ticket\_fullnum") + "'").ToList();

List<string> list2 = dtSource.AsEnumerable().Select(item => "('" + item.Field<string>("ticket\_fullnum") + "','" + item.Field<string>("ticket\_num") + "')").Distinct().ToList();

还有下面的方法：

List<User> nonDuplicateList1 = users.Distinct().ToList();//通过User类的Equals实现去重

            List<User> nonDuplicateList2 = users.Where((x,i)=>users.FindIndex(z=>z.name == x.name) == i).ToList();//Lambda表达式去重

            List<User> nonDuplicateList3 = new List<User>();//通过循环方式去重

            foreach(User user in users)

            {

                if(nonDuplicateList3.Exists(x=>x.name==user.name) == false)

                {

                    nonDuplicateList3.Add(user);

                }

            }

List字符串拼合

return string.Join(",", list.ToArray());

根据DataTable拼合

var listsql = string.Join(",", dt.AsEnumerable().Select(d => d.Field<string>("project\_id")).ToArray());

如果需要增加字符，如拼合成：’xx’,’xxxxx’

List<string> list = dt.AsEnumerable().Select(item => "'" + item.Field<string>("PROJECT\_ID") + "'").ToList();

var condition = string.Join(",", list);

不能通过var condition = string.Join(",", "'" +list + "'");直接拼合。

多个List合并

List<int> listA = new List<int> {1,2,3,5,7,9};

List<int> listB = new List<int> {13,4,17,29,2};

List<int> Result = listA.Union(listB).ToList<int>(); //剔除重复项

List<int> Result = listA.Concat(listB).ToList<int>(); //保留重复项

HashTable

foreach (DictionaryEntry item in \_hashtable)

combox绑定HashTable

ArrayList list = new ArrayList();

            foreach (DictionaryEntry entry in \_hashtable)

            {

                list.Add(entry);

            }

            list.Insert(0, new DictionaryEntry("--请选择--", "--请选择--"));

            cbTitle.DataSource = list.Clone();

            cbTitle.DisplayMember = "Value";

            cbTitle.ValueMember = "Key";

            cbLongitude.DataSource = list.Clone();

            cbLongitude.DisplayMember = "Value";

            cbLongitude.ValueMember = "Key";

            cbLatitude.DataSource = list.Clone();

            cbLatitude.DisplayMember = "Value";

            cbLatitude.ValueMember = "Key";

            cbTitle.SelectedValue = \_hashtable.Contains("标题") ? "标题":"";

            cbLongitude.SelectedValue = \_hashtable.Contains("经度") ? "经度" : "";

            cbLatitude.SelectedValue = \_hashtable.Contains("纬度") ? "纬度" : "";

### HashTable无法还原插入顺序

HashTable在Add的时候是根据hashcode排序的，无法还原插入的顺序。

可还原插入顺序的键/值对

自己写HashList。

namespace CommonFunction

{

public class HashList : Hashtable

{

private ArrayList alKey = new ArrayList();

private ArrayList alValue = new ArrayList();

public override void Add(object key, object value)

{

alKey.Add(key);

alValue.Add(value);

base.Add(key, value);

}

public override void Clear()

{

alKey.Clear();

alValue.Clear();

base.Clear();

}

public override void Remove(object key)

{

alKey.Remove(key);

alValue.Remove(base[key]);

base.Remove(key);

}

public override ICollection Keys

{

get

{

return alKey;

}

}

public override ICollection Values

{

get

{

return alValue;

}

}

public override System.Collections.IDictionaryEnumerator GetEnumerator()

{

return base.GetEnumerator();

}

public void Sort()

{

alKey.Sort();

alValue.Sort();

}

}

}

private void btnHashList和HashTable对比\_Click(object sender, EventArgs e)

{

Hashtable hashtable = new Hashtable();

CommonFunction.HashList hashlist = new CommonFunction.HashList();

ccMsg.Clear();

hashtable.Add("运营商", "运营商");

hashtable.Add("站点", "站点");

hashtable.Add("经度", "经度");

hashtable.Add("纬度", "纬度");

hashlist.Add("运营商", "运营商");

hashlist.Add("站点", "站点");

hashlist.Add("经度", "经度");

hashlist.Add("纬度", "纬度");

ccMsg.Text += "HashTable排序：" + Environment.NewLine;

foreach (DictionaryEntry item in hashtable)

{

ccMsg.Text += item.Value.ToString() + Environment.NewLine;

}

ccMsg.Text += "HashList排序,不按录入顺序：" + Environment.NewLine;

foreach (DictionaryEntry item in hashtable)

{

ccMsg.Text += item.Value.ToString() + Environment.NewLine;

}

ccMsg.Text += "HashList排序，按照录入顺序：" + Environment.NewLine;

foreach (object item in hashlist.Values)

{

ccMsg.Text += item.ToString() + Environment.NewLine;

}

foreach (object item in hashlist.Keys)

{

ccMsg.Text += hashlist[item].ToString() + Environment.NewLine;

}

}

如果觉得不方便，想实现foreach (DictionaryEntry item in hashtable)这样的效果，HashList可以这么写：

foreach (object item in hashitem.Keys)

{

string id = item != null ? item.ToString() : "";

ResourceItem resourceitem = hashitem[item] != null ? (ResourceItem)hashitem[item] : null;

}

DateTime

中文日期转换

class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("请输入一个日期：");

            string strDate = Console.ReadLine();

            string dc = Baodate2Chinese(strDate);

            Console.WriteLine(dc);

        }

        private static string Baodate2Chinese(string strDate)

        {

            char[] strChinese= new char[] {

                 '〇','一','二','三','四','五','六','七','八','九','十'

             };

            StringBuilder result = new StringBuilder();

            //// 依据正则表达式判断参数是否正确

            //Regex theReg = new Regex(@"(d{2}|d{4})(/|-)(d{1,2})(/|-)(d{1,2})");

            if (!string.IsNullOrEmpty(strDate))

            {

                // 将数字日期的年月日存到字符数组str中

                string[] str = null;

                if (strDate.Contains("-"))

                {

                    str = strDate.Split('-');

                }

                else if (strDate.Contains("/"))

                {

                    str = strDate.Split('/');

                }

                // str[0]中为年，将其各个字符转换为相应的汉字

                for (int i = 0; i < str[0].Length; i++)

                {

                    result.Append(strChinese[int.Parse(str[0][i].ToString())]);

                }

                result.Append("年");

                // 转换月

                int month = int.Parse(str[1]);

                int MN1 = month / 10;

                int MN2 = month % 10;

                if (MN1 > 1)

                {

                    result.Append(strChinese[MN1]);

                }

                if (MN1 > 0)

                {

                    result.Append(strChinese[10]);

                }

                if (MN2 != 0)

                {

                    result.Append(strChinese[MN2]);

                }

                result.Append("月");

                // 转换日

                int day = int.Parse(str[2]);

                int DN1 = day / 10;

                int DN2 = day % 10;

                if (DN1 > 1)

                {

                    result.Append(strChinese[DN1]);

                }

                if (DN1 > 0)

                {

                    result.Append(strChinese[10]);

                }

                if (DN2 != 0)

                {

                    result.Append(strChinese[DN2]);

                }

                result.Append("日");

            }

            else

            {

                throw new ArgumentException();

            }

            return result.ToString();

        }

    }

DataTable

### 创建DataTable

private DataTable CreateData()

{

DataTable dt = new DataTable();

dt.Columns.Add(new DataColumn("类型"));

dt.Columns.Add(new DataColumn("2005-1月", typeof(decimal)));

dt.Columns.Add(new DataColumn("2005-2月", typeof(decimal)));

dt.Columns.Add(new DataColumn("2005-3月", typeof(decimal)));

dt.Columns.Add(new DataColumn("2005-4月", typeof(decimal)));

dt.Columns.Add(new DataColumn("2005-5月", typeof(decimal)));

dt.Columns.Add(new DataColumn("2005-6月", typeof(decimal)));

dt.Rows.Add(new object[] { "员工人数", 437, 437, 414, 397, 387, 378 });

dt.Rows.Add(new object[] { "人均月薪", 3964, 3961, 3979, 3974, 3967, 3972 });

dt.Rows.Add(new object[] { "成本TEU", 3104, 1339, 3595.8, 3154.5, 2499.8, 3026 });

dt.Rows.Add(new object[] { "人均生产率", 7.1, 3.06, 8.69, 7.95, 6.46, 8.01 });

dt.Rows.Add(new object[] { "占2005年3月人数比例", 1.06, 1.06, 1, 0.96, 0.93, 0.91 });

return dt;

}

### 查询和筛选

mmDT\_人员.DefaultView.RowFilter = mmSearchString;

mmDT\_人员.DefaultView.Sort = "地区代码 asc,出生年月 desc,姓名 asc";

dgvDataList.DataSource = mmDT\_人员.DefaultView.ToTable();

### 去重

/// datatable去重

/// </summary>

/// <param name="dtSource">需要去重的datatable</param>

/// <param name="columnNames">依据哪些列去重</param>

/// <returns></returns>

public static DataTable GetDistinctTable(DataTable dtSource, params string[] columnNames)

{

DataTable distinctTable = dtSource.Clone();

try

{

if (dtSource != null && dtSource.Rows.Count > 0)

{

DataView dv = new DataView(dtSource);

distinctTable = dv.ToTable(true, columnNames);

}

}

catch (Exception ee)

{

MessageBox.Show(ee.ToString());

}

return distinctTable;

}

<https://blog.csdn.net/jiankunking/article/details/48684239>

//去掉重复行

DataView dv = table.DefaultView;

table = dv.ToTable(true, new string[] { "name", "code" });

此时table 就只有name、code无重复的两行了，如果还需要id值则

table = dv.ToTable(true, new string[] { "id","name", "code" });//第一个参数true 启用去重复，类似distinct

去掉空行

DataView dv = dt.DefaultView;  
dv.RowFilter = "F1<>'' or F1 is not null";//F1为列名  
System.Data.DataTable dt = dv.ToTable();

CheckBoxColumn实时获取是否选中

private void Dgv\_CellContentClick(object sender, DataGridViewCellEventArgs e)

{

if (sender!=null && tabMainControl.SelectedTab != null)

{

DataGridView dgv=(DataGridView)sender;

if (e.ColumnIndex >= 0 && e.RowIndex >= 0)

{

//遍历datagridview中的每一行，判断是否选中，若为选中，则选中

for (int i = 0; i < dgv.Rows.Count; i++)

{

var m = dgv.Rows[i].Cells[0].Value!=null ? dgv.Rows[i].Cells[0].Value.ToString():"";

var m1 = dgv.Rows[i].Cells[0].EditedFormattedValue != null ? dgv.Rows[i].Cells[0].EditedFormattedValue.ToString() : "";

if ((Convert.ToBoolean(dgv.Rows[i].Cells[0].EditedFormattedValue) == true))

{

//dgvDataList.Rows[i].Cells[0].Value = true;

var code = dgv.Rows[i].Cells[tabMainControl.SelectedTab.Text + "\_" + keyItemName].EditedFormattedValue.ToString();

if (!ht.Contains(code))

{

addToSelect(dgv.Rows[i], i);

}

}

else

{

removeFromSelect(dgv.Rows[i], i);

}

}

}

}

一定要CellContentClick事件，并且一定要用dgv.Rows[i].Cells[0].EditedFormattedValue

var m = dgv.Rows[i].Cells[0].Value!=null ? dgv.Rows[i].Cells[0].Value.ToString():"";

var m1 = dgv.Rows[i].Cells[0].EditedFormattedValue != null ? dgv.Rows[i].Cells[0].EditedFormattedValue.ToString() : "";

DataTable转换为List

//DataTable的NAEM列转换为list

var list = dt.AsEnumerable().Select(item => item.Field<string>("NAME")).ToList();

这是LINQ里的写法：

List<string> s = (from a in dt.AsEnumerable()//每次用Linq来操作集合的时候都会用到AsQueryable()和AsEnumerable(),下面我们讲区别

select a.Field<string>("F2")).ToList<string>();//这里的“F2”就是列名

foreach (var item in s)

{

　　MessageBox.Show("Test:" + item.ToString());//读取结果显示

}

AsQueryable()和AsEnumerable()的区别

区别就大了 ，AsEnumerable() 是 LINQ TO OBJECT，AsQueryable 是 LINQ TO SQL。

DataTable按列查询

DataTable dtSource = (DataTable)dgvTask.DataSource;

if (!string.IsNullOrEmpty(tbKey.Text.Trim()))

{

filter += dtSource.Columns.Count > 0 ? " AND (" : "";

for (int colIndex = 0; colIndex < dtSource.Columns.Count; colIndex++)

{

//判断时间的列

var columnName = dtSource.Columns[colIndex].DataType == Type.GetType("System.DateTime") ? "to\_char("+ dtSource.Columns[colIndex].ColumnName + ", 'yyyy-MM-dd')" : dtSource.Columns[colIndex].ColumnName;

filter += (colIndex == 0?"":" or ") + columnName + " like '%" + tbKey.Text.Trim() + "%'";

}

filter += dtSource.Columns.Count > 0 ? ") " : "";

}

DataTable遍历删除某几行

DataRow[] drs = ht\_selected.Select("id='" + dgvDataList.Rows[e.RowIndex].Cells[uniqueItem].EditedFormattedValue.ToString() + "'");

foreach (DataRow dr in drs)

{

ht\_selected.Rows.Remove(dr);

}

DataSet

### DataSet.GetChanges

获取 [DataSet](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.dataset?view=netframework-4.7.2) 的副本，该副本包含自上次加载以来或自调用 [AcceptChanges()](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.dataset.acceptchanges?view=netframework-4.7.2" \l "System_Data_DataSet_AcceptChanges) 以来的所有更改。我们可以通过：

var change = drProjectInfo.Table.DataSet.GetChanges();

//更新项目信息

Dics.GetDbClient().UpdateToDb(Dics.connStr, "SELECT \* FROM t\_project", change);

来完成更新。

不过使用的时候需要注意，该副本包含自上次加载以来或自调用 [AcceptChanges()](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.dataset.acceptchanges?view=netframework-4.7.2" \l "System_Data_DataSet_AcceptChanges) 以来的所有更改。多线程环境下会有问题。因此在结束其他线程操作时需要通过[AcceptChanges()](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.dataset.acceptchanges?view=netframework-4.7.2#System_Data_DataSet_AcceptChanges)来更新DataSet。

this.Invoke(new MethodInvoker(() =>

{

for (var i = 0; i < countNum; i++)

{

var m = dgvProject.Rows[i].Cells["PROJECT\_ID"].Value;

DataRow[] drs = dt\_num.Select("BELONG\_PROJECT='" + dgvProject.Rows[i].Cells["PROJECT\_ID"].Value + "'");

dgvProject.Rows[i].Cells["PJ\_TASK\_NUM"].Value = drs.Length > 0 ? drs[0]["num"] : 0;

}

DataTable dtProject = (DataTable)dgvProject.DataSource;

dtProject.AcceptChanges();

}));

Word操作

基本概念

引用Microsoft.Office.Interop.Word。

好用的帮助文档：

<http://blog.sina.com.cn/s/blog_6a3c68cf0100ydk3.html>

直接关闭word

Document对象：acDoc.Close(false);只是关闭了文档对象，Application对象：wordApp.Quit();才会关闭通过程序打开的word，也就变相实现了没有word文档打开时，关闭word。

点击某个按钮，将word置于窗口最前方

range.Select();

acDoc.Activate();

acDoc.Application.Activate();

简单操作

关于内容倒着加，这里有一段解释：

在Windows中：

'\r' 回车，回到当前行的行首，而不会换到下一行，如果接着输出的话，本行以前的内容会被逐一覆盖；

'\n' 换行，换到当前位置的下一行，而不会回到行首；

包含光标移动和插入文字、图片等操作。如果不移动光标，内容是倒着加的。

System.IO.FileInfo fileInfo1 = new System.IO.FileInfo(FileDirectory + "1.jpg");

System.IO.FileInfo fileInfo2 = new System.IO.FileInfo(FileDirectory + "2.jpg");

object oMissing = Type.Missing;

object count = 0;

object wdLineUnit = Microsoft.Office.Interop.Word.WdUnits.wdLine;//换一行;

object wdWordUnit = Microsoft.Office.Interop.Word.WdUnits.wdWord;

object wdCharacterUnit = Microsoft.Office.Interop.Word.WdUnits.wdCharacter;

object moveCount = 1;

object moveExtend = Microsoft.Office.Interop.Word.WdMovementType.wdExtend;

#region 换行，可用

acDoc.Application.Selection.MoveDown(ref wdLineUnit, ref count, ref oMissing);//移动焦点

acDoc.Application.Selection.TypeParagraph();//插入段落

#endregion

acDoc.Application.Selection.Text = "移动一下光标的位置";

moveCount = 1;

//wdCharacterUnit = Microsoft.Office.Interop.Word.WdUnits.wdCharacter;

acDoc.Application.Selection.MoveRight(ref wdCharacterUnit, ref moveCount);//光标移动到当前结尾

acDoc.Application.Selection.TypeParagraph();//插入段落

acDoc.Application.Selection.Text = "移动一下光标的位置";

moveCount = 1;

acDoc.Application.Selection.MoveRight(ref wdCharacterUnit, ref moveCount);//光标移动到当前结尾

acDoc.Application.Selection.TypeParagraph();//插入段落

Microsoft.Office.Interop.Word.Paragraph p1;

p1 = acDoc.Paragraphs.Add(acDoc.Application.Selection.Range);

p1.Alignment = WdParagraphAlignment.wdAlignParagraphCenter;

var shape = acDoc.InlineShapes.AddPicture(fileInfo1.FullName, Type.Missing, Type.Missing, p1.Range);

shape.Width = 400; //设置图片宽度

shape.Height = 200; //设置图片高度

//p1.Range.InsertParagraphAfter();

moveCount = 1;

acDoc.Application.Selection.MoveRight(ref wdCharacterUnit, ref moveCount);//光标移动到当前结尾

acDoc.Application.Selection.TypeParagraph();//插入段落

p1 = acDoc.Paragraphs.Add(acDoc.Application.Selection.Range);

p1.Alignment = WdParagraphAlignment.wdAlignParagraphCenter;

shape = acDoc.InlineShapes.AddPicture(fileInfo2.FullName, Type.Missing, Type.Missing, p1.Range);

shape.Width = 400; //设置图片宽度

shape.Height = 200; //设置图片高度

//p1.Range.InsertParagraphAfter();

moveCount = 1;

acDoc.Application.Selection.MoveRight(ref wdCharacterUnit, ref moveCount);//光标移动到当前结尾

acDoc.Application.Selection.TypeParagraph();//插入段落

acDoc.Application.Selection.Text = "移动";

moveCount = 1;

acDoc.Application.Selection.MoveRight(ref wdCharacterUnit, ref moveCount);//光标移动到当前结尾

acDoc.Application.Selection.TypeParagraph();//插入段落

shape = acDoc.InlineShapes.AddPicture(fileInfo1.FullName, Type.Missing, Type.Missing, acDoc.Application.Selection.Range);

shape.Width = 400; //设置图片宽度

shape.Height = 200; //设置图片高度

moveCount = 1;

acDoc.Application.Selection.MoveRight(ref wdCharacterUnit, ref moveCount);//光标移动到当前结尾

目前的疑问：

moveCount = 1;

acDoc.Application.Selection.MoveRight(ref wdCharacterUnit, ref moveCount);//光标移动到当前结尾

acDoc.Application.Selection.TypeParagraph();//插入段落

好像对图片有效，其他的文字仍然可以使用：

p.Range.InsertParagraphAfter();

acDoc.Application.Selection.MoveDown();

选中某个区域

paragraph.Range.Select();

表格跨行重复显示表头

tables[1]. Rows[1].HeadingFormat = (int)WdConstants.wdToggle

这种方式对合并单元格不适用。

根据窗口调整表格

tables[1].AutoFitBehavior(WdAutoFitBehavior.wdAutoFitWindow);

表格宽度100%

tables[1].PreferredWidthType == WdPreferredWidthType.wdPreferredWidthPercent && tables[1].PreferredWidth != 100

遍历表格

for (var r = 1; r < tb.Rows.Count+1; r++)

{

var columnIndex = 0;

foreach (Cell cell in tables[i].Rows[r].Cells)

{

var text= cell.Range.Text;

}

}

字体对应表

\_FontFormatHash = new HashList();

\_FontFormatHash.Add("一号", 26);

\_FontFormatHash.Add("小一", 24);

\_FontFormatHash.Add("二号", 22);

\_FontFormatHash.Add("小二", 18);

\_FontFormatHash.Add("三号", 16);

\_FontFormatHash.Add("小三", 15);

\_FontFormatHash.Add("四号", 14);

\_FontFormatHash.Add("小四", 12);

\_FontFormatHash.Add("五号", 10.5);

\_FontFormatHash.Add("小五", 9);

\_FontFormatHash.Add("六号", 7.5);

\_FontFormatHash.Add("小六", 6.5);

修改文字的背景色

range.Shading.Texture = 0;

range.Shading.BackgroundPatternColor = WdColor.wdColorLightGreen;

目录操作

更新操作：

//更新目录

if (acDoc.TablesOfContents.Count > 0)

{

acDoc.TablesOfContents[1].Update(); //全部更新，包含页码

//acDoc.TablesOfContents[1].UpdatePageNumbers(); //更新页码

Range range\_Content = acDoc.TablesOfContents[1].Range;

for (var i = 1; i <= range\_Content.Paragraphs.Count; i++)

{

Paragraph p = range\_Content.Paragraphs[i];

var t = p.Range.Text;

}

}

else

{

txtMsg.Text += "没有找到目录，请先添加目录！" + Environment.NewLine;

}

添加目录可参见：<http://www.cnblogs.com/sumu/archive/2011/01/19/1939509.htm>

目录跳转

if (acDoc.TablesOfContents.Count > 0)

{

acDoc.TablesOfContents[1].Update();

Range range\_Content = acDoc.TablesOfContents[1].Range;

int index = 0;

for (var i = 1; i <= range\_Content.Paragraphs.Count; i++)

{

Paragraph p = range\_Content.Paragraphs[i];

var text = p.Range.Text;

range\_Content.Hyperlinks[1].Follow(ref NewWindow,ref AddHistory,ref Missing,ref Missing); //跳转到对应的目录

}

}

获取目录位置

public class Position

{

int start=0;

int end=0;

public int Start { get => start; set => start = value; }

public int End { get => end; set => end = value; }

}

Position GetPos\_TableContent()

{

Position pos = new Position();

//更新目录

if (acDoc.TablesOfContents.Count > 0)

{

acDoc.TablesOfContents[1].Update();

Range range\_Content = acDoc.TablesOfContents[1].Range;

pos.Start = range\_Content.Start; //Range的Start

pos.End = range\_Content.End; //Range的End

}

return pos;

}

判断当前位置是否是目录

/// <summary>

/// 判断当前区域是否是目录

/// </summary>

/// <param name="pos\_Range">当前区域的位置</param>

/// <param name="pos\_TableContent">目录的位置</param>

bool isTableContent(Position pos\_Range, Position pos\_TableContent)

{

return (pos\_Range.Start >= pos\_TableContent.Start && pos\_Range.Start <= pos\_TableContent.End) || (pos\_Range.End >= pos\_TableContent.Start && pos\_Range.End <= pos\_TableContent.End);

}

缩进

首行缩进：paragraph.CharacterUnitFirstLineIndent=2；

paragraph.FirstLineIndent=2；

左侧缩进：paragraph.CharacterUnitLeftIndent=2；

或（不推荐）：paragraph.LeftIndent = acDoc.Application.CentimetersToPoints((float)0.35);

行距

固定行距：paragraph.LineSpacing=30；

1.5倍行距：paragraph.LineSpacingRule = Microsoft.Office.Interop.Word.WdLineSpacing.wdLineSpace1pt5;

分词

words通过索引获取，可以强制转换为Range。

Words w = range.Words;

foreach (var wo in range.Words)

{

Range m1 = (Range)wo;

var m2 = m1.Text;

}

将paragraph的\n外的其他文字，加背景色

public void SetUnPassItemBackColor(dynamic range)

{

if (range.Text.Contains("\n"))

{

WordOperation.SetBackColor(range, 13434828); //WdColor.wdColorLightGreen

}

else

{

dynamic w = range.Words;

int start = range.Start;

int end = range.Start;

foreach (var wo in range.Words)

{

Microsoft.Office.Interop.Word.Range m1 = (Microsoft.Office.Interop.Word.Range)wo;

if (m1.Text == "\n" || m1.Text == "\r")

{

end = m1.Start;

break;

}

}

if (end > start)

{

Microsoft.Office.Interop.Word.Range m2 = acDoc.Range(start, end - 1);

WordOperation.SetBackColor(m2, 13434828);

}

}

}

上标和下标(可用于科学计数)

https://www.cnblogs.com/fzj16888/p/5515661.html

mmTable.Cell(bookmarkrowindex + i + 1, x + 1).Range.Text = "SO\x2084\x00B2\x207B(mg/L)";

Aspose.Words操作

客户机上不用安装word，使用上比原生的要简单。

<https://www.evget.com/article/2019/11/5/33183.html>

操作书签

Document doc = new Document(targetPath);

DocumentBuilder builder = new DocumentBuilder(doc);

foreach (Bookmark mark in doc.Range.Bookmarks)

{

if (mark != null)

{

if (mark.Name.Equals("照片"))

{

var images = allfiles.Where(the => the.StartsWith(imgFolder + dr["身份号码"].ToString())).ToList(); ;

var imgPath = "";

foreach (string img in images)

{

if (!img.ToLower().EndsWith(".docx") && !img.ToLower().EndsWith(".doc") && !img.ToLower().EndsWith(".pdf"))

{

imgPath = img;

break;

}

}

mark.Text = "";

if (File.Exists(imgPath))

{

builder.MoveToBookmark(mark.Name);

builder.InsertImage(imgPath, RelativeHorizontalPosition.Margin, 0, RelativeVerticalPosition.Margin, 0, 80, 116, WrapType.Inline);

}

else

{

builder.MoveToBookmark(mark.Name);

builder.Write("照\n\n片\n");

}

}

else

{

mark.Text = "";

builder.MoveToBookmark(mark.Name);

builder.Write(dr[mark.Name].ToString());

}

}

}

doc.Save(targetPath);

一定要mark.Text = "";否则写入的内容是添加在后面。

操作字符串

通过文本替换的方式实现，需要引用using Aspose.Words.Drawing;和using Aspose.Words.Replacing;

Document doc = new Document(targetPath);

DocumentBuilder builder = new DocumentBuilder(doc);

foreach (DataColumn column in dt.Columns)

{

doc.Range.Replace("{" + column.ColumnName + "}", dr[column.ColumnName].ToString(), new FindReplaceOptions(FindReplaceDirection.Forward));

}

if(!string.IsNullOrEmpty(imgPath) || File.Exists(imgPath))

doc.Range.Replace(new Regex(@"\{照片\}"), new ReplaceAndInsertImage(imgPath, 80, 116), false);

else

doc.Range.Replace("{照片}", "照\n\n片\n", new FindReplaceOptions(FindReplaceDirection.Forward));

doc.Save(targetPath);

具体的ReplaceAndInsertImage类需要去实现：

namespace GenDocuments

{

class ReplaceAndInsertImage: IReplacingCallback

{

/// <summary>

/// 需要插入的图片路径

/// </summary>

public string url { get; set; }

public double width { get; set; }

public double height { get; set; }

public ReplaceAndInsertImage(string url, double width, double height)

{

this.url = url;

this.width = width;

this.height = height;

}

public ReplaceAction Replacing(ReplacingArgs e)

{

//获取当前节点

var node = e.MatchNode;

//获取当前文档

Document doc = node.Document as Document;

DocumentBuilder builder = new DocumentBuilder(doc);

//将光标移动到指定节点

builder.MoveTo(node);

//插入图片

builder.InsertImage(url,width,height);

return ReplaceAction.Replace;

}

}

}

多个word合并

Document docMerge = new Document();

docMerge.RemoveAllChildren();

if (radMerge.Checked)

{

backgroundWorker1.ReportProgress(procent, "正在合并申报表……");

for (var i = 0; i < paths.Count; i++)

{

Document srcDoc = new Document(paths[i]);

docMerge.AppendDocument(srcDoc, ImportFormatMode.UseDestinationStyles);

if (i > 1)

docMerge.Sections[i].HeadersFooters.LinkToPrevious(false);

}

docMerge.Save(targetfolder + "国家职业技能鉴定申报表-" + DateTime.Now.ToString("yyyy年M月d日 H时m分") + ".docx"); //s秒

for (var i = 0; i < paths.Count; i++)

{

try

{

if (File.Exists(paths[i])) File.Delete(paths[i]);

}

catch

{

continue;

}

}

}

Aspose.Cells操作

客户机上不用安装excel，使用上比原生的要简单。

将excel的内容转换为datatable

方法一：

Workbook workbook = new Workbook(path);

if (workbook.Worksheets.Count > 0)

{

var name = workbook.Worksheets[0].Name;

Cells cells = workbook.Worksheets[0].Cells;

if (cells == null)

{

return;

}

for (int i = 0; i < cells.MaxDataRow + 1; i++)

{

for (int j = 0; j < cells.MaxDataColumn + 1; j++)

{

string s = cells[i, j].StringValue.Trim();

//一行行的读取数据，插入数据库的代码也可以在这里写

}

}

}

方法二：

Workbook workbook = new Workbook(path);

if (workbook.Worksheets.Count > 0)

{

var name = workbook.Worksheets[0].Name;

Cells cells = workbook.Worksheets[0].Cells;

if (cells == null)

{

return;

}

DataTable dt = new DataTable();

if (cells.MaxRow > 0)

{

for (int j = 0; j < cells.MaxDataColumn + 1; j++)

{

string s = cells[0, j].StringValue.Trim();

if (!dt.Columns.Contains(s))

{

dt.Columns.Add(s);

}

}

cells.ExportDataTable(dt,0, 0, cells.MaxDataRow,true);//noneTitle

}

}

文件夹和文件操作

打开目录

Process.Start(@"explorer.exe", "/select,\"" + fullName + "\"");

Process.Start(fullName);

查找文件

List<FileInfo> files = dicInfo.GetFiles("\*.\*").Where(file => file.Name.ToLower().EndsWith(".xls") || file.Name.ToLower().EndsWith(".xlsx")).ToList();

下载文件

float percent = 0;

try

{

System.Net.ServicePointManager.SecurityProtocol = (System.Net.SecurityProtocolType)3072 | System.Net.SecurityProtocolType.Tls | System.Net.SecurityProtocolType.Ssl3 | (System.Net.SecurityProtocolType)768;//(System.Net.SecurityProtocolType)3072;

System.Net.HttpWebRequest Myrq = (System.Net.HttpWebRequest)System.Net.HttpWebRequest.Create(URL);

System.Net.HttpWebResponse myrp = (System.Net.HttpWebResponse)Myrq.GetResponse();

FileStream writeStream; // 写入本地文件流对象

long totalBytes = myrp.ContentLength;

System.IO.Stream st = myrp.GetResponseStream();

System.IO.Stream so = new System.IO.FileStream(filename, System.IO.FileMode.Create);

long totalDownloadedByte = 0;

byte[] by = new byte[1024];

int osize = st.Read(by, 0, (int)by.Length);

while (osize > 0)

{

totalDownloadedByte = osize + totalDownloadedByte;

so.Write(by, 0, osize);

osize = st.Read(by, 0, (int)by.Length);

}

so.Close();

st.Close();

//System.Net.WebClient client = new System.Net.WebClient();

//System.Net.ServicePointManager.SecurityProtocol = (System.Net.SecurityProtocolType)3072;

//byte[] data = client.DownloadData(URL); //https://c.cicdi.com/zkzs/fileserver/filePath/20190808/1565228819017\_电磁环境勘察平台.zip http://172.16.241.5:23588/fileserver/filePath/20190808/1565228819017\_电磁环境勘察平台.zip "http://172.16.12.2:8080/filePath/20190808/1565228819017\_电磁环境勘察平台.zip"

//FileStream fs = new FileStream(filename, FileMode.Create);

////将byte数组写入文件中

//fs.Write(data, 0, data.Length);

//fs.Close();

return true;

}

catch (System.Exception ex)

{

msg = ex.Message;

return false;

}

这里有个坑，System.Net.ServicePointManager.SecurityProtocol = (System.Net.SecurityProtocolType)3072 | System.Net.SecurityProtocolType.Tls | System.Net.SecurityProtocolType.Ssl3 | (System.Net.SecurityProtocolType)768;

## 基础连接已经关闭: 发送时发生错误

这里有个坑，WebClient 请求的时候出现 “基础连接已经关闭: 发送时发生错误”问题。

System.Net.ServicePointManager.SecurityProtocol = (System.Net.SecurityProtocolType)3072 | System.Net.SecurityProtocolType.Tls | System.Net.SecurityProtocolType.Ssl3 | (System.Net.SecurityProtocolType)768;

<https://bbs.csdn.net/topics/390934154>

如果还是下载错误，需要升级framework到4.5

## 请求被中止: 未能创建 SSL/TLS 安全通道

同上。

## 使用WebClient可解决上述问题

使用WebClient即可解决上述问题。

public static string DownloadFile(string url, string filename)

{

try

{

WebClient client = new WebClient();

client.DownloadFile(url, filename);

return filename;

}

catch

{

return "";

}

}

## 查看网站的SSL/TLS

Chrome:



## TLS 1.3



System.Net.ServicePointManager.SecurityProtocol = (System.Net.SecurityProtocolType)3072 | System.Net.SecurityProtocolType.Tls | System.Net.SecurityProtocolType.Ssl3 | (System.Net.SecurityProtocolType)768 | (System.Net.SecurityProtocolType)12288 | (System.Net.SecurityProtocolType)0;

<https://docs.microsoft.com/zh-cn/dotnet/api/system.net.securityprotocoltype?view=net-5.0#System_Net_SecurityProtocolType_SystemDefault>

## 文本

读文本文件

a. File.ReadAllText

public static string getContent(string fileName)

{

string tmp = File.ReadAllText(fileName, Encoding.UTF8);

return tmp;

}

或者：

b. StreamReader

//存在

StreamReader stream = new StreamReader(fileName, Encoding.UTF8);

string str = stream.ReadLine();

stream.Close();

return str;

写文本文件

a. StreamWriter

using (StreamWriter sw = new StreamWriter(fileName, false))

sw.WriteLine(arg);

逐行替换某段文字

string[] lines = System.IO.File.ReadAllLines(strFilePath);

List<string> list = new List<string>();

var isNeedDelete = false;

var isHasCicdi = false;

for (int x = 0; x < lines.Length; x++)

{

if (lines[x].Contains(";(cicdi-start)"))

{

isNeedDelete= isHasCicdi = true;

}

if (!isNeedDelete) list.Add(lines[x]);

if (lines[x].Contains(";(cicdi-end)"))

{

isNeedDelete = false;

}

if (lines[x].Contains(";(cicdi-start)"))

{

//如果原来就有，原位置替换

list.Add(";(cicdi-start)");

list.Add("(COMMAND \"FILEDIA\" 0)");

list.Add("(COMMAND \"NETLOAD\" \"" + dir.Replace("\\", "\\\\") + "DesignModel.dll\" nil)");

list.Add("(COMMAND \"MENUUNLOAD\" \"JSPTPD\_XL\" nil)");

list.Add("(COMMAND \"MENUUNLOAD\" \"CICDI\_MENU\" nil)");

list.Add("(COMMAND \"MENULOAD\" \"" + dir.Replace("\\", "\\\\") + "DesignModel.mns\" nil)");

list.Add("(COMMAND \"FILEDIA\" 1)");

list.Add(";(cicdi-end)");

}

}

if (!isHasCicdi)

{

//如果从来没有添加lsp，新增

list.Add(";(cicdi-start)");

list.Add("(COMMAND \"FILEDIA\" 0)");

list.Add("(COMMAND \"NETLOAD\" \"" + dir.Replace("\\", "\\\\") + "DesignModel.dll\" nil)");

list.Add("(COMMAND \"MENUUNLOAD\" \"JSPTPD\_XL\" nil)");

list.Add("(COMMAND \"MENUUNLOAD\" \"CICDI\_MENU\" nil)");

list.Add("(COMMAND \"MENULOAD\" \"" + dir.Replace("\\", "\\\\") + "DesignModel.mns\" nil)");

list.Add("(COMMAND \"FILEDIA\" 1)");

list.Add(";(cicdi-end)");

}

lines = list.ToArray();

File.WriteAllLines(strFilePath, lines);

Excel操作

基本概念

引用Microsoft.Office.Interop.Excel。

身份证格式的问题

对于列头：

worksheet.Cells[1, i + 1] = columnName;

if (columnName.Equals("身份证号"))

{

worksheet.Cells[1, i + 1].NumberFormat = "@";

mSF = i;

}

对于数据：

if (i == mSF)

{

worksheet.Cells[r + 2, i + 1].NumberFormat = "@";

worksheet.Cells[r + 2, i + 1] = "'" + myDGV.Rows[r][i];

mSF = i;

}

else

{

worksheet.Cells[r + 2, i + 1] = myDGV.Rows[r][i];

}

在数据前面加上'，拷贝到word，包括excel显示都没问题，只有显示单元格详情的时候，会显示该符号。

工作薄打印

foreach (Microsoft.Office.Interop.Excel.Workbook wb in excelApp.Workbooks)

{

wb.ExportAsFixedFormat(Microsoft.Office.Interop.Excel.XlFixedFormatType.xlTypePDF, targetPath);

}

sheet打印

int x = 0;

for (int i = 1; i <= acWorkBook.Worksheets.Count; i++)

{

var sheet = acWorkBook.Worksheets[i];

if (sheet.Name.StartsWith("表1") || sheet.Name.StartsWith("表2") || sheet.Name.StartsWith("表一") || sheet.Name.StartsWith("表二") || sheet.Name.StartsWith("表三") || sheet.Name.StartsWith("表四") || sheet.Name.StartsWith("表五"))

{

var tp = targetPath.Replace(".pdf", "-" + (++x).ToString() + ".pdf");

sheet.ExportAsFixedFormat(Microsoft.Office.Interop.Excel.XlFixedFormatType.xlTypePDF, tp);

pdfs.Add(tp);

}

}

PDFHelper.PdfMerge(pdfs, targetPath);

操作后直接清退进程

Process.Start(fullName);的情况下，直接关闭就可以。

如果是调用的COM，则必须：

xlApp.Quit();

GC.Collect();//强行销毁

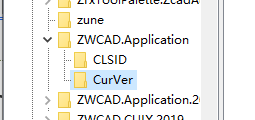
System.Runtime.InteropServices.Marshal.ReleaseComObject(xlApp);

其中xlApp 是Microsoft.Office.Interop.Excel.Application。

关于progID

注册表。





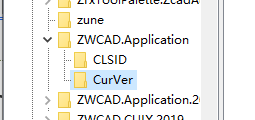


CAD操作

多个版本CAD

也是上图的注册表。





这里只会有一个版本的CLSID和CurVer，应该能判断默认的CAD版本。

默认加载菜单

找到CAD安装目录下的Support目录，C:\Program Files\ZWSOFT\ZWCAD 2019\Support，里面有lsp文件。



在里面添加如下内容：

(COMMAND "FILEDIA" 0)

(COMMAND "NETLOAD" "E:\\gitsmart\\SmartDesignTool\\SmartDesignTool\\bin\\Debug\\DesignModel.dll" nil)

(COMMAND "MENUUNLOAD" "JSPTPD\_XL" nil)

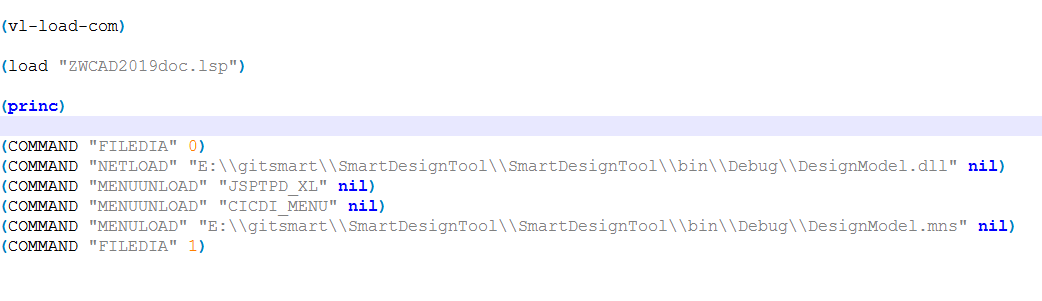
(COMMAND "MENUUNLOAD" "JSPTPD\_WX" nil)

(COMMAND "MENUUNLOAD" "CICDI\_MENU" nil)

(COMMAND "MENULOAD" "E:\\gitsmart\\SmartDesignTool\\SmartDesignTool\\bin\\Debug\\DesignModel.mns" nil)

(COMMAND "FILEDIA" 1)

如图：



mns是菜单文件，里面指定menugroup, MENUUNLOAD的作用是先卸载对应的menugroup，然后再加载，从而实现菜单跟随mns文件更新的目的。

日志操作

log4net

log4net是.Net下一个非常优秀的开源日志记录组件。log4net记录日志的功能非常强大。它可以将日志分不同的等级，以不同的格式，输出到不同的媒介。

下载地址：<http://logging.apache.org/log4net/download_log4net.cgi>

下载：[log4net-2.0.8-bin-newkey.zip](http://mirror.bit.edu.cn/apache/logging/log4net/binaries/log4net-2.0.8-bin-newkey.zip)

自己写的类

public void WriteLog(string msg)

{

string filePath = AppDomain.CurrentDomain.BaseDirectory + "Log";

if (!Directory.Exists(filePath))

{

Directory.CreateDirectory(filePath);

}

string logPath = AppDomain.CurrentDomain.BaseDirectory + "Log\\" + DateTime.Now.ToString("yyyy-MM-dd") + ".txt";

try

{

using (StreamWriter sw = File.AppendText(logPath))

{

sw.WriteLine("消息：" + msg);

sw.WriteLine("时间：" + DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"));

sw.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

sw.WriteLine();

sw.Flush();

sw.Close();

sw.Dispose();

}

}

catch (IOException e)

{

using (StreamWriter sw = File.AppendText(logPath))

{

sw.WriteLine("异常：" + e.Message);

sw.WriteLine("时间：" + DateTime.Now.ToString("yyy-MM-dd HH:mm:ss"));

sw.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

sw.WriteLine();

sw.Flush();

sw.Close();

sw.Dispose();

}

}

}

winform下使用日志：<https://www.cnblogs.com/xuxuzhaozhao/p/6640623.html>

基础里也有例子。

web api略有不同，参见：

E:\gitsmart\SmartDocService\SmartDocService

File

File.AppendAllText方法

此方法在 .NET Framework 2.0 版中是新增的。打开一个文件，向其中追加指定的字符串，然后关闭该文件。如果文件不存在，此方法创建一个文件，将指定的字符串写入文件，然后关闭该文件。

File.AppendText方法

创建一个 StreamWriter，它将 UTF-8 编码文本追加到现有文件。

# 六、SignalR：

**ASP.NET SignalR**是为 ASP.NET 开发人员提供的一个库，可以简化开发人员将实时 Web 功能添加到应用程序的过程。实时Web功能是指这样一种功能：当所连接的客户端变得可用时服务器代码可以立即向其推送内容，而不是让服务器等待客户端请求新的数据。

# 七、Socket通信：

# 八、Windows服务：

# 九、Web Service：

WSDL（Web Service Description Language）

你会怎样向别人介绍你的Web service有什么功能，以及每个函数调用时的参数呢？你可能会自己写一套文档，你甚至可能会口头上告诉需要使用你的Web service的人。

这些非正式的方法至少都有一个严重的问题：当程序员坐到电脑前，想要使用你的Web service的时候，他们的工具(如Visual Studio)无法给他们提供任何帮助，因为这些工具根本就不了解你的Web service。解决方法是：用机器能阅读的方式提供一个正式的描述文档。Web service描述语言(WSDL)就是这样一个基于XML的语言，用于描述Web service及其函数、参数和返回值。因为是基于XML的，所以WSDL既是机器可阅读的，又是人可阅读的，这将是一个很大的好处。一些最新的开发工具既能根据你的Web service生成WSDL文档，又能导入WSDL文档，生成调用相应Web service的代码。

Web服务器描述语言是用XML文档来描述Web服务的标准，是Web服务的接口定义语言，由Ariba、Intel、IBM、MS等共同提出，通过WSDL，可描述Web服务的三个基本属性：

服务做些什么——服务所提供的操作（方法）；

如何访问服务——和服务交互的数据格式以及必要协议；

服务位于何处——协议相关的地址，如URL。

# 十、WCF服务：

基本概念

**WCF**(Windows Communication Foundation)是微软公司推出的面向服务技术的集大成者(windows通信接口)，涵盖继承了其之前发布的所有的分布式应用程序的编程模型。

Windows Communication Foundation (WCF) 是用于构建面向服务的应用程序的框架。借助 WCF，可以将数据作为异步消息从一个服务终结点发送至另一个服务终结点。服务终结点可以是由 IIS 承载的持续可用的服务的一部分，也可以是应用程序中承载的服务。终结点可以是从服务终结点请求数据的服务客户端。消息可以是从以 XML 格式发送的单个字符或单词，到复杂的二进制数据流。虽然在WCF 出现之前也可以开发此类应用，但是 WCF 的出现使此类应用的开发变得更加简单。

WCF合并了Web服务、.net Remoting、消息队列和Enterprise Services的功能并集成在Visual Studio中。

基本介绍可见“老老实实学WCF”系列。

我们在平时开发的过程中常用的项目类型有“WCF 服务应用程序”和“WCF服务库”。

　　WCF服务应用程序，是一个可以执行的程序，它有独立的进程，WCF服务类契约的定义，可以直接看到运行的效果。此项目模板基于IIS托管的程序，如本系列的第一节所示。在开发基于IIS托管的WCF服务程序时，比较多见，自学的时候也可以使用这种类型，简单易懂。

　　WCF服务库，可以认为是一个包含WCF服务以及契约定义的类库。不能直接运行，你可以在其他项目里引用，在宿主里启用托管这个库，有点类似于我们在Web项目中应用的类库。考虑WCF服务设计的时候，服务类的定义为单独的库，可以为其它项目使用。提高代码的复用性。

　　当然你也可以修改这些代码，比如把WCF服务程序里的类，移到一个单独的类库里，或是把类库里的类移到WCF服务程序中。√╳

[WCF 与 Asp.net Web service 比较](http://www.cnblogs.com/forgetu/archive/2010/05/22/wcf-vs-aspnet-web-services.html)

Asp.net Web service 通常依赖于 [XmlSerializer](http://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlserializer(v=VS.100).aspx) 将 .NET Framework 的数据类型转换为 XML 格式在服务中传送，或将从服务中接收的 XML 格式的数据转换为 .NET Framework 中的对象。

WCF 使用 [DataContractAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datacontractattribute(v=VS.100).aspx) 和 [DataMemberAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datamemberattribute(v=VS.100).aspx) 来标记可以被序列化的 .NET Framework 类型。

[DataContract]

public class Item

{

[DataMember]

public string ID;

[DataMember]

public decimal Qty;

[DataMember]

public decimal Price;

}

DataContractAttribute 可以应用于类或结构体上，DataMemberAttribute可以应用于标记为 public 或 private 的字段和属性上。

WCF、.Net Remoting和WebService的关系

.Net Remoting是普通的TCP通讯，适合于局域网，效率高； WebService是基于Http协议，适合于广域网，效率低；WCF是对.Net Remoting、 WebService等的简化、统一，可以通过配置来切换不同的底层实现，代码几乎不用动。

DataContractSerializer 和 XmlSerializer的区别：

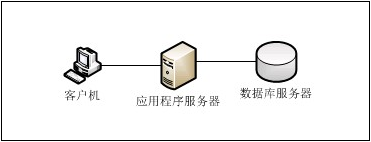
1、DataContractSerializer 不对 .NET Framework 中的数据类型在 XML 中的表示方式作控制，因此性能要比 XmlSerializer 高。

2、XmlSerializer 不能确定要序列化的字段或属性，而 DataContractSerializer 使用 DataMemberAttribute 可以明确的确定要序列化的字段或属性。

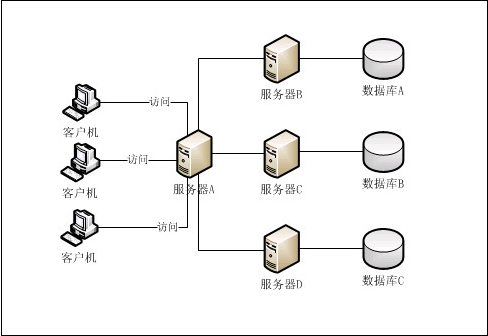
3、DataContractSerializer 可以序列化实现了 IDictionary 接口的类。

4、由于 DataContractSerializer 可以访问对象的非 public 成员，因此在反序列化时需要运行在完全信任模式下，而 XmlSerializer 不需要。

基于Asp.net 的应用程序开发与面向服务开发



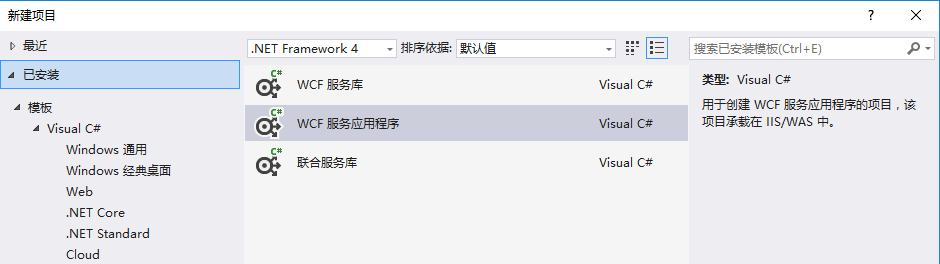
在基于Asp.net 的应用程序开发中，我们由客户机的浏览器访问应用程序服务器，然后通过应用程序服务器中的数据库连接去连接数据库服务器，读取或是操作数据，有时候可能会多一个文件服务器。大家可以观察到，基本上所有的应用都放在了一台服务器上，但对于一个，由于业务上的需要（如：与外部系统交互），一台服务器很难支持所有的应用。我们再看下面的图：



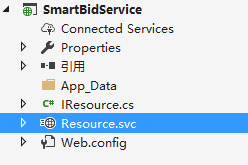
客户机使用浏览器访问服务器A，服务器A为了业务需要与其他各种应用部署在服务器B、C、D....再通过WCF技术互相通信，相互访问...然而面向服务的好处不仅仅在此，他还提供了不同语言不同操作系统的可交互性。

创建一个简单的WCF

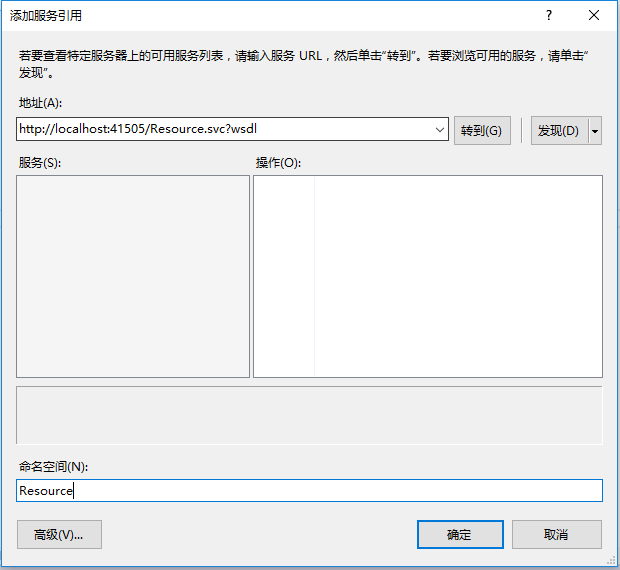
创建WCF应用程序：



F2重命名IService和ServiceReference1。



在客户端如winform中添加服务引用。



此处命名空间可以任意起。

Resource.ResourceClient client = new Resource.ResourceClient();

var m=client.GetData(2);

客户端就可以使用了。

## WCF在MYSQL升级后获取不到数据，提示调用 SSPI 失败，请参见内部异常

MYSQL默认SSL连接是打开的，因为现在还不了解SSL相关设置，所以干脆就直接关闭了，我是通过config连接字符串上加上了 “ SslMode=none; ” 关闭的，目前设置后错误还未发生，顺便说下编码问题的话记得加上 “ Charset=utf8; ”

例子：<add name="smart\_bid" providerName="mysql" connectionString="Database='smart\_bid';Data Source=172.16.7.16;User ID=cpdat;Password=cicdi@123;CharSet=utf8;SslMode=none;Persist Security Info=True" />

<https://www.pianshen.com/article/9289755960/>

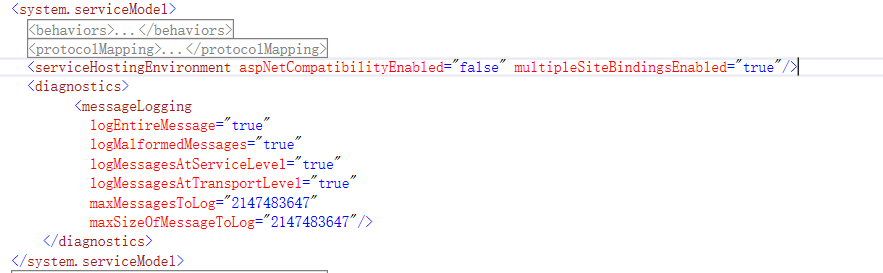
<https://blog.csdn.net/venrar/article/details/80102784>

WCF内置的日志跟踪

<https://www.cnblogs.com/daryl/p/7160971.html>

<https://www.cnblogs.com/jfzhu/p/4030008.html>

在web.config中：



<diagnostics>

<messageLogging

logEntireMessage="true"

logMalformedMessages="true"

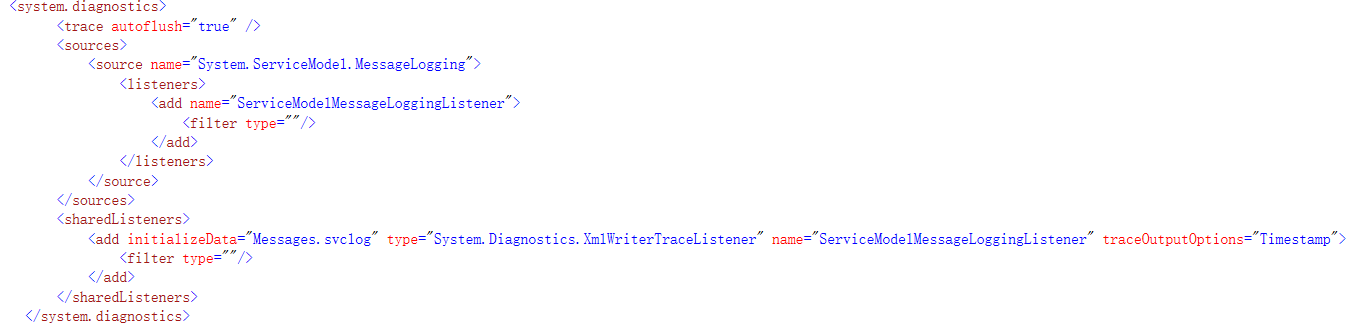
logMessagesAtServiceLevel="true"

logMessagesAtTransportLevel="true"

maxMessagesToLog="2147483647"

maxSizeOfMessageToLog="2147483647"/>

</diagnostics>



以下配置和<system.serviceModel平行：

<system.diagnostics>

<trace autoflush="true" />

<sources>

<source name="System.ServiceModel.MessageLogging">

<listeners>

<add name="ServiceModelMessageLoggingListener">

<filter type=""/>

</add>

</listeners>

</source>

</sources>

<sharedListeners>

<add initializeData="Messages.svclog" type="System.Diagnostics.XmlWriterTraceListener" name="ServiceModelMessageLoggingListener" traceOutputOptions="Timestamp">

<filter type=""/>

</add>

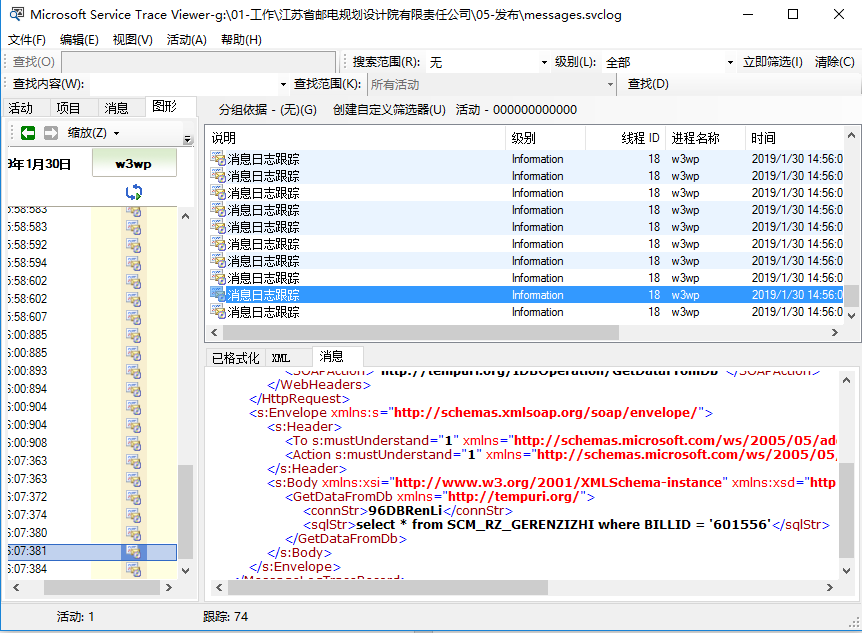
</sharedListeners>

</system.diagnostics>

会生成一个Messages.svclog。

使用SvcTraceViewer.exe打开，一般在C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools目录。

打开svclog文件，找到对应时间的请求，就可以查看请求的参数了：



# 十一、经典桌面应用程序-WinForm：

Git

.vs、bin、obj这些控件可以不要上传，也可以忽略，只要.csproj项目文件就可以。

团队管理器：<https://www.cnblogs.com/mqxs/p/9497096.htm>

有冲突，点合并按钮后，会有上一冲突下一冲突，点击处理即可。

窗体

窗体显示在最前面。

方法一：设置窗体的TopMost属性为true.

方法二：在打开窗体的时候，在窗体的  Form.Show()方法中，把父窗体 this 作为参数放进去：

1. formTreatRecord = new FormTreatRecord();
2. formTreatRecord.Show(this);
3. formTreatRecord.Focus();

布局控件

布局控件"WeifenLuo.WinFormsUI.Docking"是一个非常棒的开源控件，用过的人都深有体会，该控件之强大、美观、不亚于商业控件。而且控件使用也是比较简单的。

<https://www.cnblogs.com/wuhuacong/archive/2009/07/09/1520082.html>

去掉边框

窗体的boderstyle属性设为None

去掉边框增加阴影

<https://www.cnblogs.com/big-lll/p/6792744.html>

窗体阴影

1、添加命名空间：  
using System.Runtime.InteropServices;

2、定义常量值及函数：  
private const int CS\_DropSHADOW = 0x20000;  
private const int GCL\_STYLE = (-26);  
[DllImport("user32.dll", CharSet = CharSet.Auto)]  
public static extern int SetClassLong(IntPtr hwnd, int nIndex, int dwNewLong);  
[DllImport("user32.dll", CharSet = CharSet.Auto)]  
public static extern int GetClassLong(IntPtr hwnd, int nIndex);

3、构造方法下引用：  
SetClassLong(this.Handle, GCL\_STYLE, GetClassLong(this.Handle, GCL\_STYLE) | CS\_DropSHADOW);

进程

获取当前进程并kill。

var currentprocess = Process.GetCurrentProcess();

var thisid = currentprocess.Id;

var thisname = currentprocess.ProcessName;

var list = Process.GetProcessesByName(thisname);

if (list.Length > 1)

{

foreach (var each in list)

{

if (each.Id != thisid)

{

each.Kill();

}

}

}

带参数调试

有时main需要带参数，如string[] arg，可以在调试-属性中，输入命令行参数，参数之间用空格隔开，也可以运行的时候输入：xx.exe 参数1 参数2，也可以：

ProcessStartInfo info = new ProcessStartInfo(installversionfilename);

info.Arguments = \_param;

var process = System.Diagnostics.Process.Start(info);

但是注意一下，有的时候参数中需要传递路径，但是路径中会有空格，这是参数无法识别的，会将路径拆成多个参数，怎么解决这个问题？

参数如路径有空格的解决方法

加上引号，如：

"C:\Program Files (x86)\SmartClient\plugs\10910010\_线路CAD工具集\JSPTPD.mns" "BNewCableRouteAction.dll"

或

"C:\Program Files (x86)\SmartClient\plugs\10910010\_线路CAD工具集\JSPTPD.mns" BNewCableRouteAction.dll

info.FileName =Dics.installPath + "ZWCAD菜单安装程序\\CADMenuInstall.exe";

info.Arguments ="\""+Dics.plugPath + pluginfo.plugId + "\_" + pluginfo.plugName + "\\" + pluginfo.plugEntry + "\"" + " " + pluginfo.plugParam;

事件

说的通俗一些，就是：有一个叫做EventHandler 的家伙，他会告诉你（主程序），有一些事情发生了：这个事情是谁导致的呢？是某个object类型对象导致的，它用Source或Sender来表示。这个事情是什么事呢？e的内容就是事情的内容了。比方说你点了一个按钮，程序怎么知道应该用哪个函数来处理这个动作呢？那么EventHandler 这个家伙会告诉程序："button1(sender)被点击(e)了，请调用对应的处理函数"。当然这个函数是谁，这个函数要做什么，是由你自己写的。

sender

sender指的是触发事件的控件。

private void button2\_Click(object sender, RoutedEventArgs e)

{

Button btn = (Button)sender;//方法一

btn.Content = "kick ME!@";

//button2.Content = "kick ME";//方法二

}

如果是按钮button的话，那sender就是那个button。

DataGridView dgvList = sender as DataGridView;

调整控件位置

lableLoginInfo.Location = new Point(this.Location.X + this.Width - lableLoginInfo.Width, lableLoginInfo.Location.Y);

简单的进度提示

后台执行，前台显示滚动动画。

CpdatCloundCommon.Ing showLoad = new CpdatCloundCommon.Ing();

showLoad.SetInfo("生成中，请稍后！");

var msg = "";

DataTable ds = (DataTable)dgvCreateTable.DataSource;

ThreadStart ts = new ThreadStart(() => {

bool isOK=ExcelOperation.ExportExcels(saveFileName, ds,ref msg);

this.Invoke(new MethodInvoker(() =>

{

MessageBox.Show(showLoad, msg);

showLoad.DialogResult = System.Windows.Forms.DialogResult.OK;

if (isOK)

{

Process.Start(saveFileName);

}

}));

});

Thread gen=new Thread(ts);

gen.IsBackground = true;

gen.Start();

showLoad.ShowDialog();

启动多参数应用程序

<https://www.cnblogs.com/thirst/p/4311288.html>

对于一个应用程序，Program.cs：

namespace 多参数启动的程序

{

static class Program

{

/// <summary>

/// 应用程序的主入口点。

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

Form1:

private void Form1\_Load(object sender, EventArgs e)

{

String[] CmdArgs = System.Environment.GetCommandLineArgs();

if (CmdArgs.Length > 1)

{

//参数0是它本身的路径

String arg0 = CmdArgs[0].ToString();

String arg1 = CmdArgs[1].ToString();

String arg2 = CmdArgs[2].ToString();

MessageBox.Show(arg0);//显示这个程序本身路径

MessageBox.Show(arg1);//显示得到的第一个参数

MessageBox.Show(arg2);//显示得到的第二个参数

}

}

其他应用程序在调用的时候：

private void Menu启动多参数应用程序\_Click(object sender, EventArgs e)

{

string arg1 = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaa";

string arg2 = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbb";

System.Diagnostics.Process p = new System.Diagnostics.Process();

p.StartInfo.WorkingDirectory = @"E:\正在开发的系统\练手的项目\多参数启动的程序\多参数启动的程序\bin\Debug"; //要启动程序路径

p.StartInfo.FileName = "多参数启动的程序.exe";//需要启动的程序名

p.StartInfo.Arguments = arg1 + " " + arg2;//传递的参数

p.Start();//启动

}

也可以从Main中获取，Program.cs：

namespace 多参数启动的程序

{

static class Program

{

/// <summary>

/// 应用程序的主入口点。

/// </summary>

[STAThread]

static void Main(string[] args)

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1(args));

}

}

}

Form1:

public partial class Form1 : Form

{

public Form1(string[] args)

{

InitializeComponent();

if (args.Length > 1)

{

foreach (string arg in args)

{

MessageBox.Show("从Main(string[] args)获取：" + arg);

}

}

}

private void Form1\_Load(object sender, EventArgs e)

{

String[] CmdArgs = System.Environment.GetCommandLineArgs();

if (CmdArgs.Length > 1)

{

//参数0是它本身的路径

foreach (string arg in CmdArgs)

{

MessageBox.Show("从CommandLineArgs获取：" + arg);

}

}

}

}

Timer

Tick事件

当指定的计时器间隔已过去而且计时器处于启用状态时发生

① 直接拖控件Timer，设置属性 Enabled=true（计时器运行）；Interval=1000（计时器触发时间，以毫秒计算），然后写触发事件就可以了;

private void timer1\_Tick(object sender, EventArgs e)

{

　　　this.label1.Text = DateTime.Now.ToString();

}

② 与第一种一样，只是不拖控件而已

private void Form1\_Load(object sender, EventArgs e)  
      {  
           Timer timer1 = new Timer();  
           timer1.Interval = 1000;  
           timer1.Enabled = true;  
           timer1.Tick += new EventHandler(timer1EventProcessor);//添加事件  
      }

　public void timer1EventProcessor(object source, EventArgs e)  
        {   
            this.label1.Text = DateTime.Now.ToString();  
        }

winform嵌入word

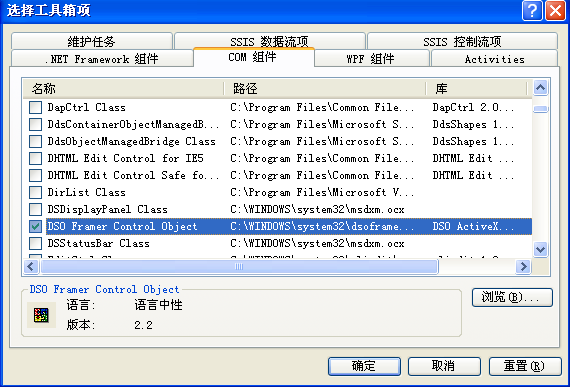
dsoframer

<https://blog.csdn.net/liu1324457514/article/details/78647902>

<https://blog.csdn.net/lishan_liang/article/details/7789730>

把dsoframer.ocx文件复制到c:\windows\system32目录下

         开始->运行->regsvr32 dsoframer.ocx , 系统会提示DSOFramer.ocx中的DllRegisterServer成功。



var fileName = @"C:\Users\Dell\Desktop\abcssaa.docx";

ccAxFramerControl.Open(fileName); //打开Word文档并显示

acDoc = WordOperation.GetDocByFullName(fileName);

if (acDoc != null)

{

acDoc.Application.Selection.Text = "打开并插入文字成功！";

}

没有注册类 (异常来自 HRESULT:0x80040154 (REGDB\_E\_CLASSNOTREG))

出现这个错误，先试试在配置管理器中将Any CPU 改为 X86的。是由于系统的差异造成的。

如果还不行，是因为第三方组件使用的COM组件未注册。在开发机上注册了，但是在客户机上还需要注册，而且不是仅仅将dll拷贝过去就可以。dsoframer就是这种情况。

winform实现dll加载时注册

为了解决上一个问题，没有注册类，我们需要dll加载时注册，有两种方法解决：

方法一：调用Regsvr32法

既然可以在运行栏中输入“Regsvr32.exe 路径”的方法来注册，那么，一定可以在C#程序中采用同样的方法来调用Regsvr32，以实现注册：

Process p = new Process();

p.StartInfo.FileName = "Regsvr32.exe";

p.StartInfo.Arguments = "/s C:\\DllTest.dll";//路径中不能有空格

p.Start();

用这种方法，注意要添加对命名空间System.Diagnostics的引用：System.Diagnostics;

这种方法有一个不足之处，那就是注册工作是在本程序之外由Regsvr32.exe程序来完成的，系统内不方便知道注册的结果，也不方便对注册过程弹出的对话框进行自定义和控制。

**方法二：调用DllRegisterServer函数法，第二种方法推荐使用**

既然方法一不大实用，那么我们就来寻找一种真正实用的方法来达到我们的目的吧。研究Regsvr32.exe和Dll文件，我们会发现，其实每个需要注册的文件都包括一个DllRegisterServer()方法，Regsvr32.exe就是通过调用该方法来完成Dll的注册的。呵呵，知道了这个，我们就可以自己调用DllRegisterServer()来完成注册过程啦。

注意要添加对命名空间Microsoft.Win32的引用：

using Microsoft.Win32;

using System.Runtime.InteropServices;

[DllImport("dsoframer.ocx")]

public static extern int DllRegisterServer();//注册时用

[DllImport("dsoframer.ocx")]

public static extern int DllUnregisterServer();//取消注册时用

对于dsoframer控件，必须要.ocx，其他的也可以换成.dll

接下来就不难啦：

int i = DllRegisterServer(); if (i >= 0) { //注册成功! } else { //注册失败} }

dsoframer.ocx 报“The associated COM server does not support ActiveX Document emb”

用的dsoframer版本太低，要用2.3版本的，网上说URL问题，是因为低版本不支持中文字符，导致URL中含有中文字符或URL不正确。新的DSO支持中文字符！

winform嵌入word

dsoframer

winform嵌入word

dsoframer

Form通过自定义URL协议在web中启动本地应用程序

<https://www.cnblogs.com/glepooek/p/9006501.html>

<https://blog.csdn.net/jackychen_king/article/details/7743811>

你必须添加一个新的*key*以及相关的*value*到*HKEY\_CLASSES\_ROOT*中，来使应用程序可以处理特殊的*URL*协议。

在*shell key*之下，一个*key*使用一个动词（就像*open*）将被添加。一个*command*（命令）*key*和一个*DDEEXEC*（动态数据交换执行）*key*都是使用动词来添加的。这*command*和*DDEEXEC keys*之后的*values*都是用来调用（或者启动）处理新协议的应用程序。

当一个用户点击一个注册了你的自定义*URL*协议的链接后，*Windows Internet Explorer*（[*IE*](https://www.baidu.com/s?wd=IE&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)）启动注册的*URL*协议的处理器。如果指定*shellopen*命令在注册表中包含一个*%1*参数的话，*Internet Explorer*传递这个*URI*给注册协议的处理器。这最后的统一资源标识符（*URI*）被编码（*%1*）；即*16*进制换码符被转换为等价的*UTF-16*字符。例如，用*%20*字符串取代空格。

对于NSIS打包程序脚本.nsi来说，如果需要添加上述的注册表：

Windows Registry Editor Version 5.00

[HKEY\_CLASSES\_ROOT\PCTV]

@="PCTVProtocol"

"URL Protocol"="\"C:\\Program Files (x86)\\CpdatCloudClient\\CpdatCloudClient.exe\""

[HKEY\_CLASSES\_ROOT\PCTV\DefaultIcon]

@="\"C:\\Program Files (x86)\\CpdatCloudClient\\CpdatCloudClient.exe,1\""

[HKEY\_CLASSES\_ROOT\PCTV\shell]

[HKEY\_CLASSES\_ROOT\PCTV\shell\open]

[HKEY\_CLASSES\_ROOT\PCTV\shell\open\command]

@="\"C:\\Program Files (x86)\\CpdatCloudClient\\CpdatCloudClient.exe\" \"%1\""

需要在.nsi脚本中添加：

WriteRegStr HKCR "CPDATCLOUD" "" "CPDATCLOUDProtocol"

WriteRegStr HKCR "CPDATCLOUD" "URL Protocol" "$INSTDIR\CpdatCloudClient.exe"

WriteRegStr HKCR "CPDATCLOUD\DefaultIcon" "" "$INSTDIR\CpdatCloudClient.exe,0"

WriteRegStr HKCR "CPDATCLOUD\shell" "" ""

WriteRegStr HKCR "CPDATCLOUD\shell\open" "" ""

WriteRegStr HKCR "CPDATCLOUD\shell\open\command" "" '"$INSTDIR\CpdatCloudClient.exe" -open "%1"'

即可实现上述的注册表操作。

Form

窗体美化无边框

无边框窗体拖动边框，会和Panel的Dock属性冲突，只需要设置Form的padding属性就可以解决了。详见基础里的Monitor项目。

无边框的winform窗体得最小值设置

在Win7系统流行时代，没有无边框窗口用的非常多，不过在没有边框的form窗口设置大小后运行（非常小时），就会出现窗口变大的问题。表面上看窗口没法调的更小了（WinForm窗口在没有边框模式下宽度最小值为123）。这对这个问题本人有两种解决方案。

一是可以在load事件中将它的大小更改为任意大小。实测此方法有效。

二种就是在窗体内，添加一个panel，然后将所有东东放在panel中，然后将form的背景设置为透明色。

窗体始终处于激活状态显示在最前

窗体的TopMost属性设为True。

窗体最大化遮住任务栏

this.MaximumSize = new Size(Screen.PrimaryScreen.WorkingArea.Width, Screen.PrimaryScreen.WorkingArea.Height);

单例窗体点开处于激活状态显示在最前

ImageViewer.SetWaterMark frmSetWaterMark = ImageViewer.SetWaterMark.GetInstance("temp", modelId, file);

frmSetWaterMark.Show();

frmSetWaterMark.Activate();

判断窗体是否打开

Form1 F1 ;

if(F1 == null || F1.IsDisposed)

{

   F1 = new Form1();

   F1.Show();//未打开，直接打开。

}

Else

{

   F1.Activate();//已打开，获得焦点，置顶。

}

最小化到托盘

使用NotifyIcon控件。



设置一个图标。

private void Mainfrm\_FormClosing(object sender, FormClosingEventArgs e)

{

//最小化到托盘notifyIcon1

DialogResult result = MessageBox.Show("是否退出？选否,最小化到托盘", "操作提示", MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question);

if (result == DialogResult.Yes)

{

this.Dispose();

Application.Exit();

}

else

{

e.Cancel = true;

this.WindowState = FormWindowState.Minimized;

this.Visible = false;

this.notifyIcon1.Visible = true;

}

}

private void Mainfrm\_Load(object sender, EventArgs e)

{

notifyIcon1.Visible = false;

notifyIcon1.DoubleClick += NotifyIcon1\_DoubleClick; ;

this.SizeChanged += Mainfrm\_SizeChanged; ;

}

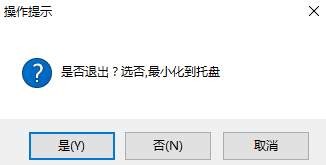
private void NotifyIcon1\_DoubleClick(object sender, EventArgs e)

{

this.WindowState = FormWindowState.Maximized;

this.Visible = true;

}





托盘控件notifyIcon运行出现两个或多个托盘图标的解决方法

有form1,form2,form3.................. 用了一个notifyIcon1.设置论坛ICON图像，结果在打开form2,form3时就出现了多个托盘图标，关掉form2,form图标还在，鼠标移动上面就消失，这是资源没有释放干净导致的。

首先在属性里，notifyIcon1.Visible = false，然后在form1 load里面，notifyIcon1.Visible = true;//默认图标不可见，托盘图标可见,以防出现多个托盘图标，一劳永逸，不管来多少个form都不怕，只有一个托盘图标。

private void ConsoleForm\_Load(object sender, EventArgs e)

{

//默认最小化到托盘

SwitchWindowState();

}

public void SwitchWindowState()

{

if (WindowState == FormWindowState.Minimized)

{

//还原窗体显示

WindowState = FormWindowState.Normal;

//激活窗体并给予它焦点

this.Activate();

//任务栏区显示图标

this.ShowInTaskbar = true;

//托盘区图标隐藏

notifyIcon1.Visible = false;

this.Show();

}

else

{

//还原窗体显示

WindowState = FormWindowState.Minimized;

//任务栏区显示图标

this.ShowInTaskbar = false;

//托盘区图标隐藏

notifyIcon1.Visible = true;

}

}

<http://blog.sina.com.cn/s/blog_ad7fd0f4010180mz.html>

加右键菜单：





WinForm 菜单项的大小、高宽的手动控制

控制菜单项的第一级的下级菜单项的大小：

直接通过(ContextMenuStrip对象).AutoSize = false、(ContextMenuStrip对象).Size = new Size(50,50)就可以控制大小。

控制菜单项的第二级或多级的下级菜单项的大小：

通过(ToolStripMenuItem对象).DropDown.AutoSize = false、(ToolStripMenuItem对象).DropDown.Size = new Size(50,50)才可以控制大小。

控制菜单项的缩略图的大小：

通过第一级菜单的(ContextMenuStrip对象).ImageScalingSize = new Size(50,50)来控制大小。

# 在winform里面怎么获取相对于某个控件的鼠标坐标，类似Location

Point pt = button1.PointToClient(Control.MousePosition)

pt就是相对于控件左上角的坐标，具体数值可以由 pt.X,pt.Y 得到  
 Point pt = plSubModule.PointToClient(Control.MousePosition);

if (pt.X >= 410 && !picRobot.Visible)

{

picRobot.Visible = true;

this.picRobot.Location = new System.Drawing.Point(394, pt.Y); //528

}

else picRobot.Visible = false;

TabControl

鼠标悬停

private void ChatTab\_MouseMove(object sender, MouseEventArgs e)

{

bool isUnLinked = IsFileLinked();

for (int i = 0; i < ChatTab.TabPages.Count; i++)

{

if (ChatTab.GetTabRect(i).Contains(e.Location))

{

ChatTab.TabPages[i].ImageIndex = i + 16;

}

else

{

ChatTab.TabPages[i].ImageIndex = i == 0 ? i + 10 : (isUnLinked ? i + 13 : i + 10);

}

}

}

private void ChatTab\_MouseLeave(object sender, EventArgs e)

{

bool isUnLinked = IsFileLinked();

for (int i = 0; i < ChatTab.TabPages.Count; i++)

{

ChatTab.TabPages[i].ImageIndex = i == 0 ? i + 10 : (isUnLinked ? i + 13 : i + 10);

}

}

鼠标点击

private void ChatTab\_Selecting(object sender, TabControlCancelEventArgs e)

{

bool isUnLinked = IsFileLinked();

switch (e.TabPageIndex)

{

case 1:

{

if (isUnLinked)

{

ChatTab.SelectedIndex = 0;

}

break;

}

case 2:

{

if (isUnLinked)

{

ChatTab.SelectedIndex = 0;

}

break;

}

}

}

禁止TabPage的点击事件

private void ChatTab\_Selecting(object sender, TabControlCancelEventArgs e)

{

bool isUnLinked = IsFileLinked();

switch (e.TabPageIndex)

{

case 1:

{

if (isUnLinked)

{

e.Cancel = true;

}

break;

}

case 2:

{

if (isUnLinked)

{

e.Cancel = true;

}

break;

}

}

}

动态添加TabPage

private void cbSelect\_SelectedIndexChanged(object sender, EventArgs e)

{

var text = cbSelect.Text;

if (!string.IsNullOrEmpty(text))

{

if (isTabExist(text))

{

//存在标签

}

else

{

TabPage tabpage = new TabPage(text);

tabMainControl.TabPages.Add(tabpage);

}

}

}

bool isTabExist(string text)

{

foreach (TabPage tab in tabMainControl.TabPages)

{

if (tab.Text == text)

{

return true;

}

}

return false;

}

ListBox

拖拽

首页控件的AllowDrop需要设置为True

private void ZYSelectedList\_MouseDown(object sender, MouseEventArgs e)

{

indexofsource = ((ListBox)sender).IndexFromPoint(e.X, e.Y);

if (indexofsource != ListBox.NoMatches)

{

((ListBox)sender).DoDragDrop(((ListBox)sender).Items[indexofsource].ToString(), DragDropEffects.Move);

}

}

private void ZYSelectedList\_DragOver(object sender, DragEventArgs e)

{

//拖动源和放置的目的地一定是一个ListBox

if (e.Data.GetDataPresent(typeof(System.String)) && ((ListBox)sender).Equals(ZYSelectedList))

{

e.Effect = DragDropEffects.Move;

}

else

e.Effect = DragDropEffects.None;

}

int indexoftarget, indexofsource;

private void ZYSelectedList\_DragDrop(object sender, DragEventArgs e)

{

ListBox listbox = (ListBox)sender;

indexoftarget = listbox.IndexFromPoint(listbox.PointToClient(new Point(e.X, e.Y)));

if (indexoftarget != ListBox.NoMatches)

{

Object obj = listbox.Items[indexofsource];

//List<string> lstMenus = (List<string>)listbox.DataSource;

//AppGIMenusMod modMenu = new AppGIMenusMod();

//modMenu = lstMenus[indexofsource];

ZYSelectedList.Items.RemoveAt(indexofsource);//从数据源中移除当前拖动项

ZYSelectedList.Items.Insert(indexoftarget, obj);//拖动项重新插入到数据源指定位置

//ZYSelectedList.DataSource = null;//这句一定要写，我不知道是什么原因，我一直以为winform中数据源回自动更新，可是如果我不释放一下数据源就算排好序，Listbox中也不会显示

//ZYSelectedList.DisplayMember = "MenuName";

//ZYSelectedList.ValueMember = "ID";

//ZYSelectedList.DataSource = lstMenus;//重新设置数据源

ZYSelectedList.SelectedIndex = indexoftarget;

}

}

注：拖拽事件使用的listBox1\_MouseDown(s, e)会‘屏蔽掉’双击事件的使用的listBox1\_DoubleClick。使用if (e.Clicks > 1)即可判断：  
private void ZYSelectedList\_MouseDoubleClick(object sender, MouseEventArgs e)

{

if (e.Clicks > 1)

{

int index = ZYSelectedList.IndexFromPoint(e.Location);

if (index != System.Windows.Forms.ListBox.NoMatches)

{

ZYSelectedList.Items.Remove(ZYSelectedList.SelectedItem); //执行双击事件

}

else

{

ZYSelectedList.SelectedIndex = -1;//不做任何操作，将ListBox的选中项取消

}

}

}

private void ZYSelectedList\_MouseDown(object sender, MouseEventArgs e)

{

if (e.Clicks <= 1)

{

indexofsource = ((ListBox)sender).IndexFromPoint(e.X, e.Y);

if (indexofsource != ListBox.NoMatches)

{

((ListBox)sender).DoDragDrop(((ListBox)sender).Items[indexofsource].ToString(), DragDropEffects.Move);

}

}

}

RichTextBox

每行文本颜色不通

TextBox无法实现这样的功能。

private void setColor(RichTextBox richTextBox1)

{

int start = 0;

int count = 1;

int index = richTextBox1.Text.IndexOf('\n', start);

while (index > -1)

{

richTextBox1.Select(start, index - start);

if (richTextBox1.SelectedText.Contains("已校正"))

{

richTextBox1.SelectionColor = Color.Green;

}

else

{

richTextBox1.SelectionColor = Color.Red;

}

count++;

start = index + 1;

index = richTextBox1.Text.IndexOf('\n', start);

}

richTextBox1.Select(start, richTextBox1.Text.Length - start);

}

GridView

不显示行头



GridView的RowHeadersVisible 属性设为False。

GridView的RowHeadersWidth 是设置宽度的。

列自动填充

GridView的AutoSizeColumnsMode 属性设为Fill。

GridView显示tip

dgvModel.ShowCellToolTips = true;

dgvModel.Rows[e.RowIndex].Cells[e.ColumnIndex].ToolTipText = hasModel ? "已上传模版，点击后可重新上传" : "尚未上传模版，点击后可开始定义";

遍历GridView

foreach (DataGridViewRow item in dgvCreateTable.Rows)

            {

                string code = item.Cells[0].Value.ToString();

                if (code == titleCode)

                {

                    item.Cells[0].Value = "标题";

                }

               }

GridView显示tip

dgvTemplateTable.CellFormatting += DgvTemplateTable\_CellFormatting;

GridView按钮

private void DgvTemplateTable\_CellFormatting(object sender, DataGridViewCellFormattingEventArgs e)

{

var columnIndex = e.ColumnIndex;

var rowIndex = e.RowIndex;

DataRowView drv = dgvTemplateTable.Rows[e.RowIndex].DataBoundItem as DataRowView;

var dr = drv.Row;

string isHasTemplate = dr["server\_file\_name"].ToString();

string colHeader = dgvTemplateTable.Columns[e.ColumnIndex].HeaderText;

switch (colHeader)

{

case Header\_TemplateOpt:

e.Value = string.IsNullOrEmpty(isHasTemplate) ? global::DocDesignManager.Properties.Resources.up\_16 : global::DocDesignManager.Properties.Resources.edit16;

break;

}

}

GridView里的checkbox事件

private void dataGridView1\_CellContentClick(object sender, DataGridViewCellEventArgs e)

{

if (e.ColumnIndex == 0 && e.RowIndex != -1)

{

//获取控件的值

var m=this.dataGridView1.Rows[e.RowIndex].Cells[0].EditedFormattedValue.ToString();

//或者可以做其他事件处理程序

}

}

或者

foreach (DataGridViewRow dgvr in dgvPublishList.Rows)

{

var selected = dgvr.Cells["choose"].EditedFormattedValue.ToString();

if (selected.Equals("True"))

{

var pTask = new PublishTask();

pTask.formId = dgvr.Cells["formId"].Value.ToString();

pTask.drawer = dgvr.Cells["drawer"].Value.ToString();

pTask.publishOrder =(int) dgvr.Cells["publishOrder"].Value;

publishTaskList.Add(pTask);

}

}

GridView里的checkbox事件无法选中



GridView列头背景

dgvTemplateTable.EnableHeadersVisualStyles = false;

dgvTemplateTable.ColumnHeadersDefaultCellStyle.BackColor =Color.FromArgb(234, 250, 254)

GridView奇数行背景变化

//奇数行的单元格的背景色为浅蓝色

dgvGroupTable.AlternatingRowsDefaultCellStyle.BackColor = Color.FromArgb(234, 250, 254);

具体的属性请参见：

<https://blog.csdn.net/wangzhen209/article/details/51744518>

GridView增加行并设置颜色变化

var index = dgvResult.Rows.Add();

dgvResult.Rows[index].Cells["检查项"].Value = state.State.Name;

dgvResult.Rows[index].Cells["检查结果"].Value = state.State.Msg;

dgvResult.Rows[index].Cells["检查结果"].Style.ForeColor = state.State.Msg.Contains("格式已校正")? Color.Green : Color.Red;

dgvResult.Rows[index].Cells["ID"].Value = state.State.Id;

DataGridView 清空时出现System.ArgumentException:“不能清除此列表。”

DataGridView 清空时出现System.ArgumentException:“不能清除此列表。”

原因是DataGridView 绑定了数据源，如果说数据源不需要了，就把数据源取消即可。如下代码：

dataGridView4.DataSource = null;

OpenFileDialog

基本属性

openFileDialog.Filter = "Word document(\*.doc;\*.docx)|\*.dox;\*.docx|All Files(\*.\*)|\*.\*";

// 设置允许选择多个文件，该属性默认为false的，即只允许选择一个文件

openFileDialog.Multiselect = true;

TreeView

TreeView默认执行afterselect事件

因为TreeView的TabIndex,最小，所以焦点在TreeView上。

ToolStrip

ToolStripButton按钮文字样式

按钮本身属性找不到，可在ToolStrip属性中查找。

PictureBox和ToolStripButton属性

其他控件都有toolTipText属性来设置其提示文本.但是PictureBox没有这个属性.该如何解决?

<https://bbs.csdn.net/topics/90207797>

ToolTip toolTip1 = new ToolTip();

toolTip1.AutoPopDelay = 5000;

toolTip1.InitialDelay = 1000;

toolTip1.ReshowDelay = 500;

toolTip1.ShowAlways = true;

toolTip1.SetToolTip(cpQualification, "公司资质");

Panel

动态往下添加控件

//添加查询条件

var index = GetUcNum() + 1;

SearchControl ucControl = new SearchControl("条件"+ index.ToString(), text, cbAttributesdt);

ucControl.Dock = DockStyle.Top;

ucControl.AfterClosed += UcControl\_AfterClosed;

plConditions.Controls.Add(ucControl);

plConditions.Controls.SetChildIndex(ucControl, 0); //控件index索引，索引一样，且设置为Top就会自动往下添加了。

//关闭控件

private void UcControl\_AfterClosed(object sender, EventArgs e)

{

if (sender == null)

{

return;

}

SearchControl ucControl = (SearchControl)sender;

ucControl.Dispose();

}

Splitter

基本使用

如左右结构，则在父容器左边添加一个panel，dock为left，紧接着在父容器里添加splitter，dock为left，在父容器右边添加一个panel，dock为right，

BackgroudWorker

基本使用

主要属性：

WorkerReportsProgress:True

WorkerSupportsCancancellation:True

主方法：

//后台异步下载文件

backgroundWorker1.RunWorkerAsync();

//主线程(UI)需要执行的其它代码

……

//等待辅助线程(backgroundWorker1)执行完成

while (this.backgroundWorker1.IsBusy)

{

Application.DoEvents();

}

//主线程(UI)继续执行其它代码

backgroundWorker1\_DoWork：辅助线程执行的工作，尽量不要操作UI。

backgroundWorker1\_ProgressChanged：辅助线程报告进度，这个方法里可以直接操作UI控件。

int i = e.ProgressPercentage;

string state = e.UserState.ToString();

if (i < dgvSelect.Rows.Count)

{

dgvSelect.Rows[i].Cells["下载状态"].Value = state;

dgvSelect.Rows[i].Cells["下载状态"].Style.ForeColor = Color.Green;

}

backgroundWorker1\_RunWorkerCompleted：辅助线程完成之后的工作。

CefSharp

CefSharp让您在.NET应用程序中嵌入Web程序。它是基于Google引擎而的开源框架。

运行环境： .net framework \*\*环境和 vc++ 2013 runtime (x86/x64)\*\*

基本使用

browser = new ChromiumWebBrowser(getUrl());

browser.DownloadHandler = new DownLoadFile();

browser.MenuHandler = new MenuHandler();

browser.Left = 0;

browser.Top = 0;

this.Controls.Add(browser);

browser.Dock = DockStyle.Fill;

jsBridge();

重新加载页面

base.browser.Load(Utils.Settings.GetAppSetting("WebHostAddress") + Utils.Settings.GetAppSetting("SmartDesignFrontPath") + "index.html#/BasicLayout/MyWorkSpace");

管理员身份运行批处理

using (Process proc = new Process())

{

string command = @"c:\users\danvy\Desktop\script\test.bat";

proc.StartInfo.FileName = command;

proc.StartInfo.WorkingDirectory = Path.GetDirectoryName(command);

//run as admin

proc.StartInfo.Verb = "runas";

proc.Start();

while (!proc.HasExited)

{

proc.WaitForExit(1000);

}

}

配置文件

类库调用自己的配置文件

## 读取指定路径配置文件的方法

如果没有ConfigurationManager，需要重新引用一下system.configuration.

public static string GetAppSetting(string key)

{

Assembly assembly = Assembly.GetCallingAssembly();

string path = string.Format("{0}.config", assembly.Location);

ExeConfigurationFileMap configFile = new ExeConfigurationFileMap();

configFile.ExeConfigFilename = path;

Configuration config = ConfigurationManager.OpenMappedExeConfiguration(configFile, ConfigurationUserLevel.None);

return config.AppSettings.Settings[key].Value;

}

其中：

Assembly assembly = Assembly.GetCallingAssembly();

string path = string.Format("{0}.config", assembly.Location);

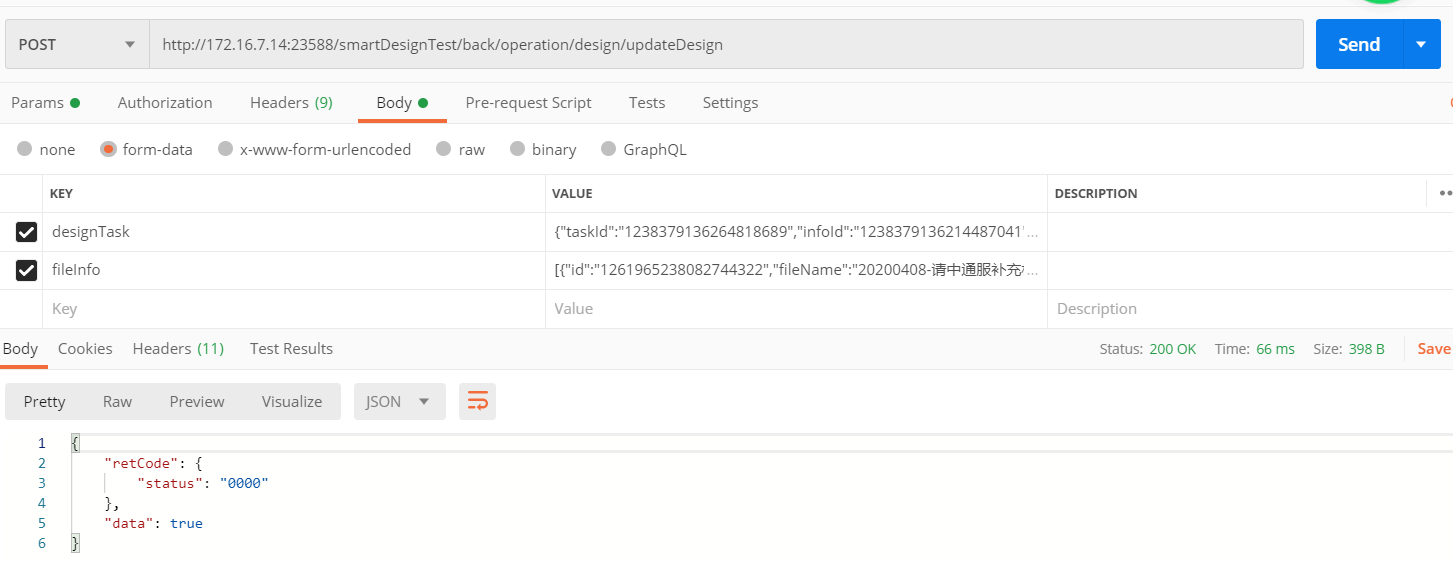
读出的是引用该类库的程序目录下的config文件，如：

E:\gitsmart\SmartDesignTool\SmartDesignTool\bin\Debug\DesignModel.dll.config

RestSharp

<https://www.cppentry.com/bencandy.php?fid=106&id=163897>

form-data



var request = new RestRequest("operation/design/updateDesign", Method.POST);

request.RequestFormat = DataFormat.Json;

request.AlwaysMultipartFormData = true;

request.AddHeader("Content-Type", "multipart/form-data");

request.AddParameter("designTask", "{\"designTask\":{\"taskId\":\"1238379136264818689\",\"infoId\":\"1238379136214487041\",\"designName\":\"南京移动2018年室外站主设备工程（第二批）\",\"designNo\":\"1002463428ZK00001\",\"city\":\"3201\",\"major\":110001,\"unitCode\":\"GC0001\",\"currentNode\":\"1000030001\",\"fileFolder\":\"106001\",\"workFlowId\":\"118001\",\"submitter\":\"889151\",\"createTime\":\"2020-03-13 16:19:29\",\"updateTime\":\"2020-03-13 16:19:29\",\"status\":101001,\"userNodeInfoDto\":[{\"userName\":\"田军\",\"nodeName\":\"提交设计\",\"nodeId\":\"1000030001\",\"nodeOrder\":1},{\"userName\":\"葛贺文\",\"nodeName\":\"审核\",\"nodeId\":\"1000030002\",\"nodeOrder\":2},{\"userName\":\"汪海波\",\"nodeName\":\"审定\",\"nodeId\":\"1000030003\",\"nodeOrder\":3},{\"userName\":\"刘露\",\"nodeName\":\"标检\",\"nodeId\":\"1000030004\",\"nodeOrder\":4}],\"errorInfoList\":[],\"currentNodeName\":\"提交设计\",\"userName\":\"田军\",\"userName2\":\"葛贺文\",\"userName3\":\"汪海波\",\"userName4\":\"刘露\"}}");

request.AddParameter("fileInfo", "[{\"id\":\"1261965238082744322\",\"fileName\":\"20200408-请中通服补充材料(1).docx\",\"remoteFileUrl\":\"filePath/20200517/1589710931904\_20200408-请中通服补充材料(1).docx\",\"size\":1573354,\"uploadTime\":1589710933595,\"status\":101001,\"operateTime\":1589710933597,\"filetype\":\"138001\"},{\"id\":\"1261965259058458626\",\"fileName\":\"【图纸表格】广州开发区禾丰路中DC-EFH.xlsx\",\"remoteFileUrl\":\"filePath/20200517/1589710936965\_【图纸表格】广州开发区禾丰路中DC-EFH.xlsx\",\"size\":150669,\"uploadTime\":1589710938596,\"status\":101001,\"operateTime\":1589710938598,\"filetype\":\"138002\"},{\"id\":\"1261967106477404162\",\"fileName\":\"中国通信服务苏设计〔2016〕53号关于印发“执业资格（资质）管理办法”的通知(1).pdf\",\"remoteFileUrl\":\"filePath/20200517/1589711377401\_中国通信服务苏设计〔2016〕53号关于印发“执业资格（资质）管理办法”的通知(1).pdf\",\"size\":671377,\"uploadTime\":1589711379054,\"status\":101001,\"operateTime\":1589711379057,\"filetype\":\"138003\"}]");

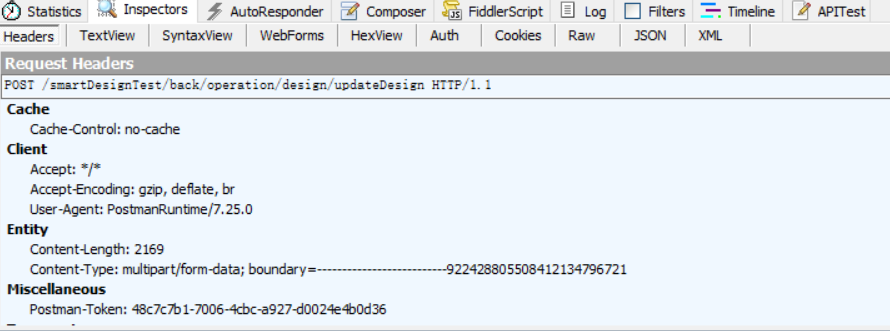
IRestResponse response = client.Execute(request);

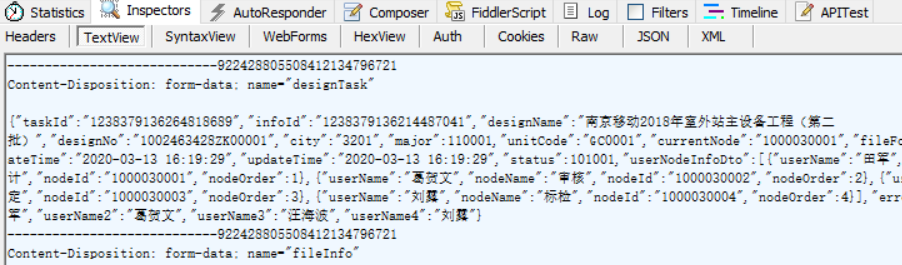
var content=response.Content;

记住:

request.AddParameter,不需要ParameterType.RequestBody

可使用Fidder抓包：

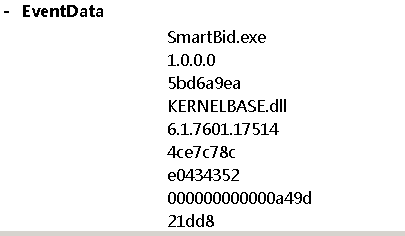


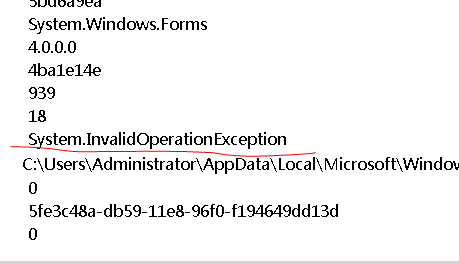


常见问题

winform写的多线程后台处理操作过段时间就会报错

错误信息，在windows日志里可见：





多个错误信息，结合起来看，根据红线的提示，判断是UI线程的问题，加上CheckForIllegalCrossThreadCalls = false;问题不再出，考虑可能是多线程时，对主线程操作的控制没做好。

# 十二、WPF：

# 十三、TypeScript：

# 十四、Agular：

# 十五、Vue：

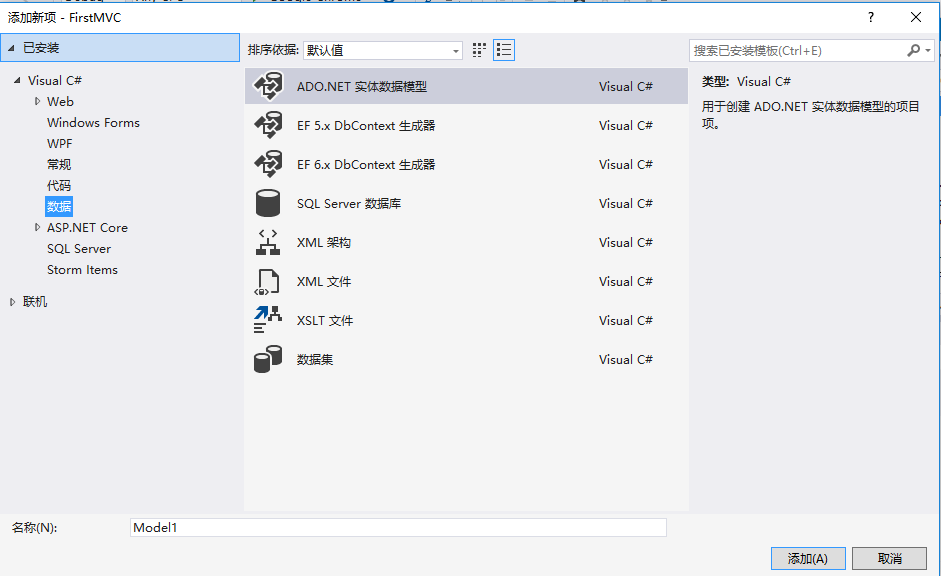
# 十六、React：

# 十七、Entity Framework：

EF ，全称Entity FrameWork。就是微软以ADO.NET为基础发展的所谓ORM（对象关系映射框架，或者说是数据持久化框架）。

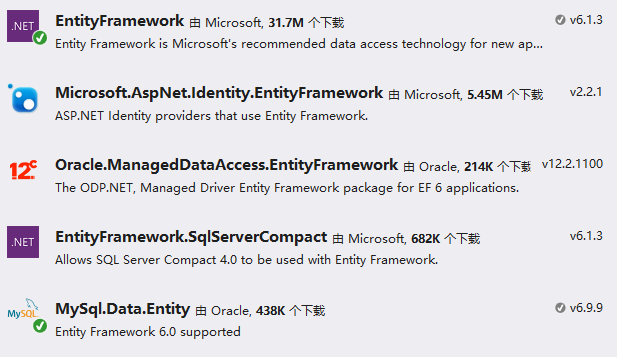
详情可查看“C#”-“应用”-“MVC”中的《解析ASP.NET Mvc开发之查询数据实例》(<http://blog.csdn.net/yisuowushinian/article/details/17743407>)。

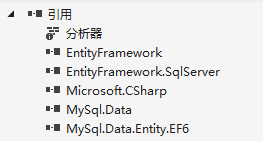
“Model”文件夹右键，选择“ADO.NET实体数据模型”：



如何添加oracle支持，在NuGet管理器中点击“浏览”，输入“Oracle.ManagedDataAccess”，会查找出来两个包，分别是Oracle.ManagedDataAccess.dll和Oracle.ManagedDataAccess.EntityFrameWork.dll，安装即可。

MySQL:





DbContext: using System.Data.Entity;

# 十八、Core:

得装个vs2015 并且保证已经升级至 update3及以上，官网上下载的最新版都是包含了update3的，vs2015 visual C#->.Net Core：如果可以大家看到的应该是3个可选项：类库，控制台程序，net core web application;记得当时第一想的怎么没有winform的呢，后来觉得winform里面需要一些资源皮肤等现有资源这些在其他系统没有，估计也是这里看不到的原因吧。

VS2017：

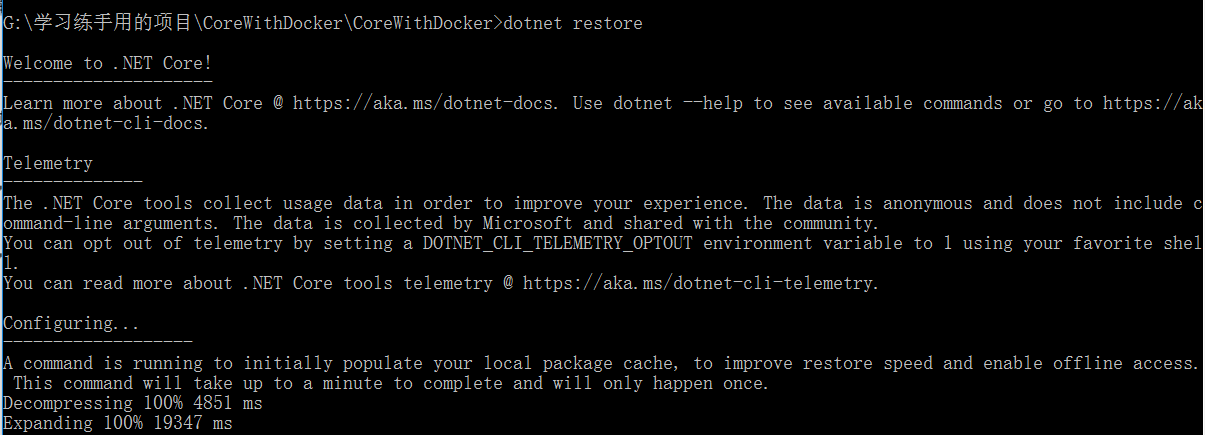
1.1版本去掉了project.json后，如何生成跨平台包？netcore小组去掉project.json重要文件后，其任务都放在了项目的.csproj文件中。们要生成跨平台运行包，就需要操作这个项目文件的配置信息，这里只需要添加如下代码：

<PropertyGroup>

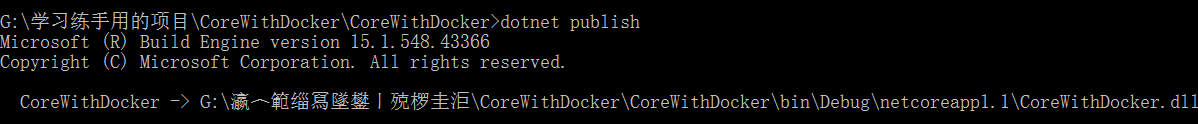
<RuntimeIdentifiers>win7-x64;ubuntu.16.04-x64</RuntimeIdentifiers>

</PropertyGroup>

然后再通过发布命令就能够生成跨平台包了，是不是挺简单的，所以上面弄了一个Session小节做干货呢呵呵；咋们来测试下吧，首先在项目根目录这样以此执行命令：**dontnet restore**



再执行你简短的发布命令(这里我直接采用默认参数的方式执行，发布包会生成再bin下面)：**dotnet publish**



这个时候我们能够在目录：**bin\Debug\netcoreapp1.1**下面看到**publish**文件夹，里面就是咋们程序执行文件了，为了测试下效果，这里我通过命令：**dontnet WebApp01.dll** (由于我这里安装了sdk所以可以直接这样命令运行)来运行我的测试用例项目：



# 十九、工作流：

# 二十、缓存Redis/Memcached：

Memcached和Redis的区别

高性能键值缓存服务器memcached也经常被拿来与Redis进行比较：这两者都可用于存储键值映射，彼此的性能也相差无几，但是Redis能够自动以两种不同的方式将数据写入硬盘，并且Redis除了能存储普通的字符串键之外，还可以存储其他4种数据结构，而memcached只能存储普通的字符串键。这些不同之处使得Redis可以用于解决更为广泛的问题，并且既可以用作主数据库（primary database）使用，又可以作为其他存储系统的辅助数据库（auxiliary database）使用。

Windows下如何安装

下载如：Redis-x64-3.2.100.msi的安装包。

安装完成后，运行命令提示符，进入目录(如：C:\Program Files\Redis)，运行：**redis-server.exe redis.windows.conf**，启动redis 服务。

窗口可以关闭，因为是在本机运行的，这里要注意端口号，同时要保持端口不要关闭。当然您也可以将[Redis作为Windows服务](http://stackoverflow.com/questions/6476945/how-do-i-run-redis-on-windows)在后台一直开启。

再开一个控制台应用程序连接之前启动的Redis，如redis-cli.exe -h 127.0.0.1 -p 6379

其中 –h后面是本机的ip地址(如下方的172.16.147.121)，后面的是端口。然后就可以执行set 给key为city赋值:

redis 172.16.147.121:6379> set city Shanghai

通过get可以获取指定key为city的值了。

redis 172.16.147.121:6379> get city

# 二十一、消息队列

在企业应用系统领域，会面对不同系统之间的通信、集成与整合，尤其当面临异构系统时，这种分布式的调用与通信变得越发重要。其次，系统中一般会有很多对实时性要求不高的但是执行起来比较较耗时的地方，比如发送短信，邮件提醒，更新文章阅读计数，记录用户操作日志等等，如果实时处理的话，在用户访问量比较大的情况下，对系统压力比较大。

因此，使用消息队列的场景一般如下：

1.存在异构系统间的整合；

2.对操作的实时性要求不高，而需要执行的任务极为耗时；

消息队列是一个公用存储空间，它可以存在于内存中或物理文件中，因此，消息以两种方式发送，即快递方式和可恢复模式。它们的区别是消息存储位置的不同，快递方式，为了消息的快速传递，所以把消息放置在内存中，而不放在物理磁盘上，以获得较高的处理能力；而可恢复模式在传送过程的每一步骤中，都把消息写入物理磁盘上，这样当保存消息队列的机器发生故障而重新启动后，可以把发送的消息恢复到故障发送之前的状态，以获得更好的消息恢复能力。消息队列可以放在发送方、接收方所在的机器上，也可以单独放置在另外一台机器上。另外，采用消息队列机制，发送方不必要担心接收方是否启动，是否发生故障等因素，只要消息成功发送出去，就可以认为处理完成，而实际上对方可能甚至未开机，或者实际消息传递到对方可能在第二天。

面对这些问题，我们一般会将这些请求，放在消息队列中处理；异构系统之间使用消息进行通讯。

MSMQ

### 概念

MSMQ全称MicroSoft Message Queue，微软消息队列,是在多个不同的应用之间实现相互通信的一种异步传输模式，相互通信的应用可以分布于同一台机器上，也可以分布于相连的网络空间中的任一位置。它的实现原理是：消息的发送者把自己想要发送的信息放入一个容器中（我们称之为Message），然后把它保存至一个系统公用空间的消息队列(Message Queue)中；本地或者是异地的消息接收程序再从该队列中取出发给它的消息进行处理。

优点:稳定、消息优先级、脱机能力以及安全性,有保障的消息传递和执行许多业务处理的可靠的防故障机制。

缺点:MSMQ不适合于Client需要Server端实时交互情况.大量请求时候,响应延迟.

### 如何安装

安装消息队列，在控制面板-程序和功能里。

VS中添加引用“System.Messaging”。

命名空间 using System.Messaging;

默认存储路径 C:\WINDOWS\system32\msmq\storage。

缺省情况下，消息序列化xml格式。

### MSMQ中主要有两个概念

一个是消息Message：Message是通信双方需要传递的消息，它可以是文本、图片、视频等。消息包含发送和接收者的标识，只有指定的用户才能取得消息。

一个是队列Queue：用来保存消息的存储空间，MSMQ中主要包括以下几种队列类型：

公共队列：在整个消息队列网络中复制，有可能由网络连接的所有站点访问。路径格式为：机器名称\队列名称

专用队列（或叫私有队列）：不在整个网络中发布，它们仅在所驻留的本地计算机上可用，专用队列只能由知道队列的完整路径名称或标签的应用程序访问。路径格式为：机器名称\Private$\队列名称

日志队列：包含确认在给定“消息队列中发送的消息回执消息”。路径格式为：机器名称\队列名称\Journal$

响应队列：包含目标应用程序接收到消息时返回给发送应用程序的响应消息，包括机器日志队列、机器死信队列和机器事务死信队列。

机器日志队列对应的格式为：机器名称\Journal$；

机器死信队列对应的格式为：机器名称\Deadletter$；

机器信道死信队列对应的格式为：机器名称\XactDeadletter$。

### MSMQ的使用

MessageQueue mq = null;

if (!MessageQueue.Exists(@".\Private$\LeeMQ"))

{

mq = MessageQueue.Create(@".\Private$\LeeMQ");

}

else

{

mq = new MessageQueue(@".\Private$\LeeMQ");

}

mq.Send("13951601173@139.com", "Email");

mq.Send("360610043@qq.com", "Email");

MessageQueue mq = new MessageQueue(@".\Private$\LeeMQ");

mq.Formatter = new XmlMessageFormatter(new string[] { "System.String" });

System.Messaging.Message msg = mq.Receive();//引用的队列中可用的第一条消息

ccMsg.Text += string.Format("{0}:{1}", msg.Label.ToString(), msg.Body.ToString()) + Environment.NewLine;

服务器端代码需要注意的是，公共队列只能在域环境中创建，如果个人电脑没有加入域环境，所以不能创建公共队列。

## RabbitMQ

### 安装

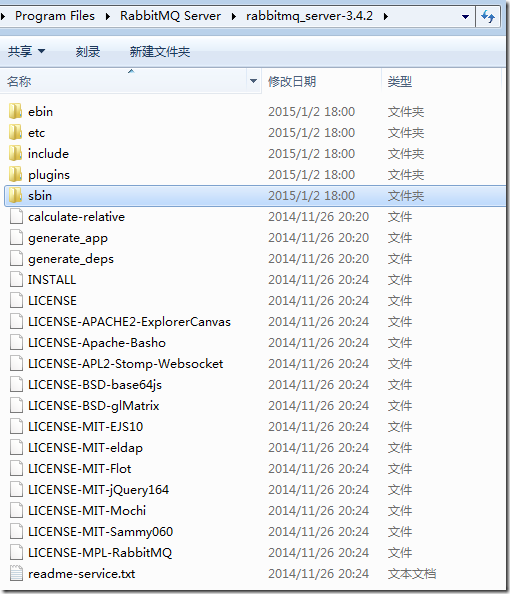
RabbitMQ使用Erlang编写，需要运行在Erlang运行时环境上，所以在安装RabbitMQ Server之前需要安装Erlang 运行时环境，可以到[Erlang官网(http://www.erlang.org/downloads)下载](file:///F:\\绿色版软件\\记事宝V1.0\\记事宝\\UserFile\\文档笔记\\技术\\知识体系\\Erlang官网(http:\\www.erlang.org\\downloads)下载)对应平台的安装文件。如果没有安装运行时环境，安装RabbitMQ Server的时候，会提示需要先安装Erlang环境。安装完成之后，确保已经将Erlang的安装路径注册到系统的环境变量中。安装完Erlang之后，这个环境会自动设置，如果没有，在administrator环境下在控制台下面输入，也可以设置：

Setx ERLANG\_HOME “D:\Program Files (x86)\erl6.3″

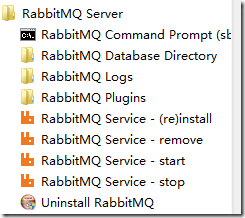
然后，去RabbitMQ官网(<http://www.rabbitmq.com/download.html>)下载[RabbitMQ Server服务端程序](http://www.rabbitmq.com/download.html" \t "_blank)，选择合适的平台版本下载。安装完成之后，就可以开始使用了。

现在就可以对RabbitMQ Server进行配置了。

首先，切换到RabbitMQ Server的安装目录：



在sbin下面有很多batch文件，用来控制RabbitMQ Server，当然您也可以直接在安装开始菜单中来执行相应的操作：

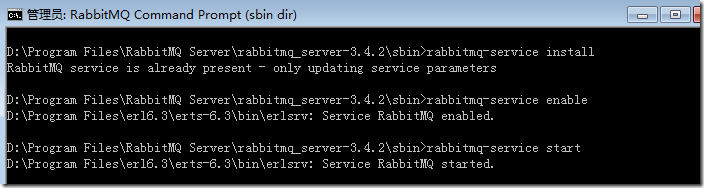


最简单的方式是使RabbitMQ以Windows Service的方式在后台运行，所以我们需要以管理员权限打开cmd，然后切换到sbin目录下，依次执行这三条命令即可：

rabbitmq-service install

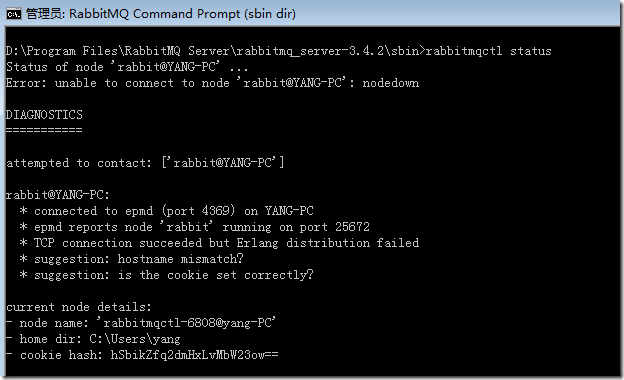
rabbitmq-service enable

rabbitmq-service start

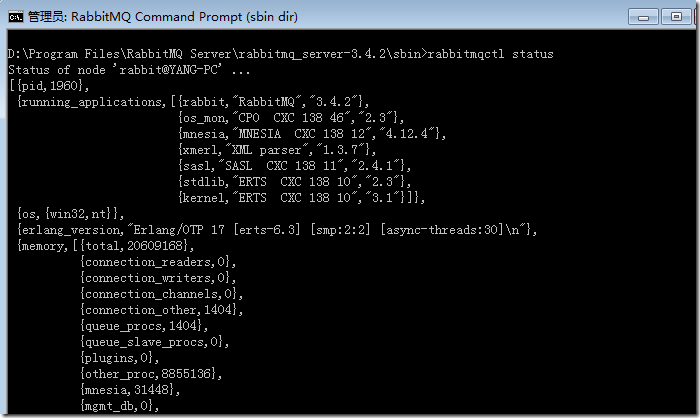


现在RabbitMQ的服务端已经启动起来了。

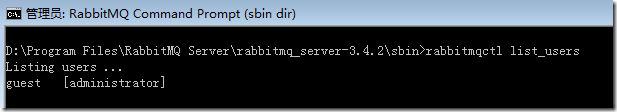
下面可以使用sbin目录下面的rabbitmqctl.bat这个脚本来查看和控制服务端状态的，在cmd中直接运行rabbitmqctl status。如果看到以下结果：



显示node没有连接上，需要到C:\Windows目录下，将.erlang.cookie文件，拷贝到用户目录下 C:\Users\{用户名}，该目录下会有这个文件，但还是要复制，这是Erlang的Cookie文件，允许与Erlang进行交互，现在重复运行刚才的命令就会得到如下信息：



RabbitMQ Server上面也有用户概念，安装好之后，使用rabbitmqctl list\_users命令，可以看到上面目前的用户：

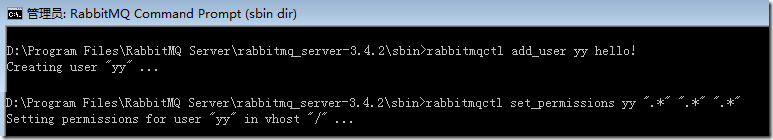


可以看到，现在只有一个角色为administrator的名为guest的用户，这个是RabbitMQ默认为我们创建的，他有RabbitMQ的所有权限，一般的，我们需要新建一个我们自己的用户，设置密码，并授予权限，并将其设置为管理员，可以使用下面的命令来执行这一操作：

rabbitmqctl add\_user yy hello!

rabbitmqctl set\_permissions yy ".\*" ".\*" ".\*"

rabbitmqctl set\_user\_tags yy administrator



上面的一条命令添加了一个名为yy的用户，并设置了密码hello！，下面的命令为用户yy分别授予对所有消息队列的配置、读和写的权限。

现在我们可以将默认的guest用户删掉，使用下面的命令即可：

rabbitmqctl delete\_user guest

如果要修改密码，可以使用下面的命令：

rabbitmqctl change\_password {username} {newpassowrd}

### 使用

在.NET中使用RabbitMQ需要下载RabbitMQ的客户端程序集，可以到[官网(http://www.rabbitmq.com/releases/rabbitmq-dotnet-client/v3.4.2/rabbitmq-dotnet-client-3.4.2-dotnet-3.5.zip)下载](http://www.rabbitmq.com/releases/rabbitmq-dotnet-client/v3.4.2/rabbitmq-dotnet-client-3.4.2-dotnet-3.5.zip)，下载解压后就可以得到RabbitMQ.Client.dll，这就是RabbitMQ的客户端。

RabbitMQ是一个消息代理。他从消息生产者(producers)那里接收消息，然后把消息送给消息消费者（consumer）在发送和接受之间，他能够根据设置的规则进行路由，缓存和持久化。

通常，消息生产者，消息消费者和消息代理不在同一台机器上。

消息生产者:

var factory = new ConnectionFactory();

factory.HostName = "127.0.0.1";

factory.UserName = "tj"; //RabbitMQ登录的用户名和密码，需要命令行分配好才能用，默认是Guest，但是只能在本机访问。

factory.Password = "123456";

using (var connection = factory.CreateConnection())

{

using (var channel = connection.CreateModel())

{

channel.QueueDeclare("hello", false, false, false, null);

Book[] message = {

new Book { Name = "C#深入浅出", Year = 2010 },

new Book { Name = "JAVA深入浅出", Year = 2009 }

};

var body = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(message));

channel.BasicPublish("", "hello",null, body);

ccMsg.Text += "操作成功！";

}

}

消息消费者:

var factory = new ConnectionFactory();

factory.HostName = "127.0.0.1";

factory.UserName = "tj";

factory.Password = "123456";

using (var connection = factory.CreateConnection())

{

using (var channel = connection.CreateModel())

{

channel.QueueDeclare("hello", false, false, false, null);

var consumer = new QueueingBasicConsumer(channel);

channel.BasicConsume("hello", true, consumer);

while (true)

{

var ea = (BasicDeliverEventArgs)consumer.Queue.Dequeue();

Book[] messages = JsonConvert.DeserializeObject<Book[]>(Encoding.UTF8.GetString(ea.Body));

for (var i = 0; i < messages.Length; i++)

{

Book book = messages[i];

ccMsg.Text += i == 0 ? "" : Environment.NewLine;

ccMsg.Text += "Name:"+book.Name+",Year:"+book.Year.ToString();

}

ccMsg.Text += Environment.NewLine+"操作成功！";

}

}

}

### 远程连接RabbitMQ失败

官网上面给的例子都是在本地使用系统默认的guest用户连接的。没有给出远程连接的例子，于是阅读文档发现, 刚刚安装好rabbitmq-server,系统会自动创建一个名为“/”的virtual host，同时也会创建一个用户名和密码都是guest的用户，并且应用"/ virtual host"的所有访问权限。因此在rabbitmq安装的机器上使用官网给出的例子运行是没问题的。如果要切换到远程机器访问的话，单纯的修改factory.setHost("localhost");是不行的。因为guest用户只是被容许从localhost访问。

# 二十二、NuGet：

Nuget是一个[.NET](http://baike.baidu.com/item/.NET" \t "_blank)平台下的开源的项目，它是[Visual Studio](http://baike.baidu.com/item/Visual%20Studio" \t "_blank)的扩展。在使用Visual Studio开发基于.NET Framework的应用时，Nuget能把在项目中添加、移除和更新引用的工作变得更加快捷方便。

Nuget是 ASP .NET Gallery 的一员。NuGet 是免费、开源的包管理开发工具，专注于在 .NET 应用开发过程中，简单地合并第三方的组件库。

当需要分享开发的工具或是库，需要建立一个Nuget package，然后把这个package放到 Nuget的站点。如果想要使用别人已经开发好的工具或是库，只需要从站点获得这个package，并且安装到自己的Visual Studio项目或是解决方案里。

NuGet能更方便地把一些dll和文件(如jquery)添加到项目中,而不需要从文件中复制拷贝。

# 二十三、其他计算机术语和名词

持久化

持久化是将程序数据在[持久](http://baike.baidu.com/item/%E6%8C%81%E4%B9%85)状态和瞬时状态间转换的机制。通俗的讲，就是瞬时数据（比如内存中的数据，是不能永久保存的）持久化为持久数据（比如持久化至数据库中，能够长久保存）。

也就是把数据找个不是内存的地方存起来，就算你的程序关掉了重新打开，还能读回来，目标不限于XML、二进制文件、数据库、写到服务器上等等。

Serverless

不意味着不再涉及服务器。只是意味着开发人员不再需要考虑太多了。计算资源以服务的方式被使用，我们不必管理物理容量或限制。服务提供商承担了大部分管理服务器，数据存储和其他基础架构资源的责任。让开发人员将重点从服务器级别转移到任务级别。无服务器解决方案让开发人员专注于他们的应用程序或系统需要做什么，消除了后端基础架构的复杂性。

# 二十四IIS和管道

WEB服务器解析的一般原理

动态WEB中，程序依然使用客户端和服务端，客户端依然使用浏览器（IE、FireFox等），通过网络(Network)连接到服务器上，使用HTTP协议发起请求（Request），现在的所有请求都先经过一个WEB Server Plugin（服务器插件）来处理，此插件用于区分是请求的是静态资源(\*.htm或者是\*.htm)还是动态资源。

　　如果WEB Server Plugin发现客户端请求的是静态资源(\*.htm或者是\*.htm)，则将请求直接转交给WEB服务器，之后WEB服务器从文件系统中取出内容，发送回客户端浏览器进行解析执行。

如果WEB Server Plugin发现客户端请求的是动态资源（\*.jsp、\*.asp/\*.aspx、\*.php），则先将请求转交给WEB Container(WEB容器)，在WEB Container中连接数据库，从数据库中取出数据等一系列操作后动态拼凑页面的展示内容，拼凑页面的展示内容后，把所有的展示内容交给WEB服务器，之后通过WEB服务器将内容发送回客户端浏览器进行解析执行。

Java Web和ASP.NET的区别

https://www.cnblogs.com/5ishare/p/6534407.html

不管是什么语言开发的web应用程序，都是在解决一个问题，那就是用户输入url怎么把对应的页面响应出来。

肤浅理解：

javabean作为model，称为"组件"，其实就是一个类或者实体类，但不一定是对应数据库表的实体类。

servlet作为controller，这里的controller并不是mvc中的controller，只是负责业务处理包括数据库访问等。写一个servlet类都要写一个对应的xml配置节点. jsp可以调用servlet中的类来进行数据处理，看上去有点像 webForm调用 handler.ashx。

具体分析：

以IIS 6.0为例，在工作进程w3wp.exe中，利用Aspnet\_ispai.dll加载.NET运行时（如果.NET运行时尚未加载）。IIS 6引入了应用程序池的概念，一个工作进程对应着一个应用程序池。一个应用程序池可以承载一个或者多个Web应用，每个Web应用映射到一个IIS虚拟目录。与IIS 5.x一样，每一个Web应用运行在各自的应用程序域中。

如果HTTP.SYS接收到的HTTP请求是对该Web应用的第一次访问，当成功加载了运行时后，会通过AppDomainFactory为该Web应用创建一个应用程序域（AppDomain）。随后，一个特殊的运行时IsapiRuntime被加载。IsapiRuntime定义在程序集System.Web中，对应的命名空间为System.Web.Hosting。IsapiRuntime会接管该HTTP请求。

IsapiRuntime会首先创建一个IsapiWorkerRequest对象，用于封装当前的HTTP请求，并将该IsapiWorkerRequest对象传递给ASP.NET运行时：HttpRuntime，从此时起，HTTP请求正式进入了ASP.NET管道。根据IsapiWorkerRequest对象，HttpRuntime会创建用于表示当前HTTP请求的上下文（Context）对象：HttpContext。

IsapiWorkerRequest是比较底层的对象，HttpRuntime接到请求之后会将其分析拆解，创建HttpRequest、HttpResponse对象，一次Http请求需要好几个对象，还有HttpServerUtility类型的对象处理网站虚拟路径和服务器文件系统之间的映射关系，为了管理这些对象，定义了HttpContext来统一处理参数的表示问题。

随着HttpContext被成功创建，HttpRuntime会利用HttpApplicationFactory创建新的或者获取现有的HttpApplication对象。实际上，ASP.NET维护着一个HttpApplication对象池，HttpApplicationFactory从池中选取可用的HttpApplication用户处理HTTP请求，处理完毕后将其释放到对象池中。HttpApplicationFactory负责处理当前的HTTP请求。HttpRuntime创建HttpContext成功之后，会创建HttpApplication对象，需要注意的是HttpApplication对象创建的来源，一是通过HttpApplicationFactory创建一种是HttpApplication对象池获取，它和Java Web中可不太一样。JavaWeb中的Servlet是单例多线程，通过Servlet对象只创建一个，通过线程池来响应请求。

在HttpApplication初始化过程中，会根据配置文件加载并初始化相应的HttpModule对象。对于HttpApplication来说，在它处理HTTP请求的不同的阶段会触发不同的事件（Event），而HttpModule的意义在于通过注册HttpApplication的相应的事件，将所需的操作注入整个HTTP请求的处理流程。ASP.NET的很多功能，比如身份验证、授权、缓存等，都是通过相应的HttpModule实现的。

当请求转入ASP.NET管道后，最终负责处理该请求的是与请求资源类型相匹配的HttpHandler对象，但是在Handler正式工作之前，ASP.NET会先加载并初始化所有配置的HttpModule对象。HttpModule在初始化的过程中，会将一些功能注册到HttpApplication相应的事件中，那么在HttpApplication整个请求处理生命周期中的某个阶段，相应的事件会被触发，通过HttpModule注册的事件处理程序也得以执行。所有的HttpModule都实现了IHttpModule接口。

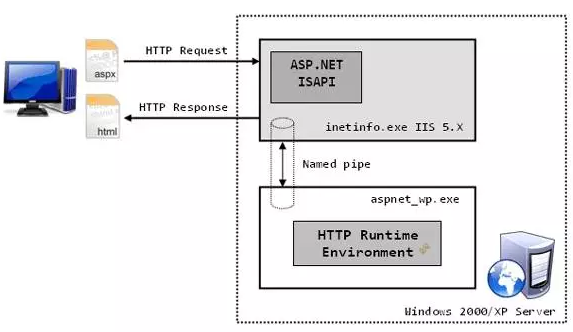
这里的HttpModule起到了过滤的功能，这就和JavaWeb的Filter有点类型，利用它们可以做一些例如权限等一些处理。这种设计的好处也很明显，扩展性很强，用户可以自己选择使用。

而最终完成对HTTP请求的处理实现在另一个重要的对象中：HttpHandler。对于不同的资源类型，具有不同的HttpHandler。比如.aspx页对应的HttpHandler为System.Web.UI.Page，WCF的.svc文件对应的HttpHandler为System.ServiceModel.Activation.HttpHandler。

HttpHander有点类似JavaWeb的Servlet，最终处理还是需要它们，而且都是web的基础，JavaWeb中jsp最终还是会以Servlet对象来运行，asp.net中也是一样。

IIS 5.x

IIS 5.x是如何处理基于ASP.NET资源（比如.aspx,.asmx等）请求的：

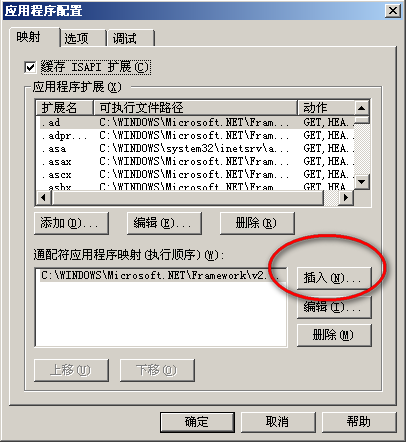


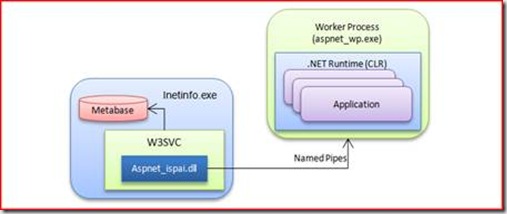
IIS 5.x一个显著的特征就是 Web Server 和真正的 ASP.NET Application 的分离。作为 Web Server 的IIS运行在一个名为 InetInfo.exe 的进程上，InetInfo.exe 是一个Native Executive，并不是一个托管的程序，而我们真正的 ASP.NET Application 则是运行在一个叫做 aspnet\_wp 的 Worker Process 上面，在该进程初始化的时候会加载CLR，所以这是一个托管的环境。

ISAPI： 指能够处理各种后缀名的应用程序。 ISAPI 是下面单词的简写 ：Internet Server Application Programe Interface，互联网服务器应用程序接口。

IIS 5.x运行在进程InetInfo.exe中，在该进程中一个最重要的服务就是名为World Wide Web Publishing Service（简称W3SVC）的Windows Service。W3SVC的主要功能包括HTTP请求的监听、工作进程的管理以及配置管理（通过从Metabase中加载相关配置信息）等。

当检测到某个HTTP Request后，先根据扩展名判断请求的是否是静态资源（比如.html,.img,.txt,.xml等），如果是则直接将文件内容以HTTP Response的形式返回。如果是动态资源（比如.aspx,asp,php等等），则通过扩展名从IIS的脚本影射（Script Map）找到相应的ISAPI Dll。这也是IIS5.x和IIS6为何有“映射”的原因：





IIS5与ASP.NET

ISAPI是Internet服务器API（Internet Server Application Programming Interface）的缩写，是一套本地的（Native）Win32 API，具有较高的执行性能,是IIS和其他动态Web应用或者平台之间的纽带。比如ASP ISAPI桥接IIS与ASP，而ASP.NET ISAPI则连接着IIS与ASP.NET。ISPAI定义在一个Dll中，ASP.NET ISAPI对应的Dll为Aspnet\_isapi.dll,你可以在目录“%windir%\Microsoft.NET\Framework\{version no}\”中找到该Dll。

如果我们请求的是一个基于ASP.NET的资源类型，比如：.aspx Web Page、 .asmx Web Service或者.svc WCF Service等，Aspnet\_isapi.dll会被加载，ASP.NET ISAPI扩展会创建ASP.NET的工作进程（如果该进程尚未启动），对于IIS 5.x来说，该工作进程为aspnet.exe。IIS进程与工作进程之间通过命名管道（Named Pipes）进程通信，以获得最好的性能。

在工作进程初始化过程中，.NET 运行时（CLR）被加载，从而构建了一个托管的环境。对于某个Web应用的初次请求，CLR会为其创建一个AppDomain。在此AppDomain中，HTTP运行时（HTTP Runtime）被加载并用以创建相应的应用。对于寄宿于IIS 5.x的所有Web 应用都运行在同一个进程（工作进程Aspnet\_wp.exe）的不同AppDomain中。

IIS 5.x至少存在着如下两个方面的不足：

a. ISAPI Dll被加载到InetInfo.exe进程中，它和工作进程之间是一种典型的跨进程通信方式，尽管采用性能最好的命名管道，但是仍然会带来性能的瓶颈；

b. 所有的ASP.NET应用，运行在相同的进程（aspnet\_wp.exe）中的不同的应用程序域（AppDomain）中，基于应用程序域的隔离级别不能从根本上解决一个应用程序对另一个程序的影响，在更多的时候，我们需要不同的Web应用运行在不同的进程中。

IIS 5模式的特点：

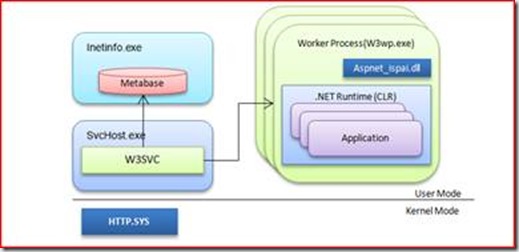
首先，同一台主机上在同一时间只能运行一个aspnet\_wp进程，每个基于虚拟目录的 ASP.NET Application 对应一个Application Domain，也就是说每个 Application 都运行在同一个 Worker Process 中，Application之间的隔离是基于Application Domain的，而不是基于Process的。

其次，ASP.NET  ISAPI不但负责创建 aspnet\_wp Worker Process，而且负责监控该进程，如果检测到 aspnet\_wp 的 Performance 降低到某个设定的下限，ASP.NET ISAPI会负责结束掉该进程。当 aspnet\_wp 结束掉之后，后续的 Request 会导致ASP.NET ISAPI重新创建新的 aspnet\_wp Worker Process。

最后，由于 IIS 和 Application 运行在他们各自的进程中，他们之间的通信必须采用特定的通信机制。本质上 IIS 所在的 InetInfo 进程和 Worker Process 之间的通信是同一台机器不同进程的通信（local interprocess communications），处于 Performance 的考虑，他们之间采用基于 Named pipe 的通信机制。ASP.NET ISAPI 和 Worker Process 之间的通信通过他们之间的一组 Pipe 实现。同样处于 Performance 的原因，ASP.NET ISAPI 通过异步的方式将 Request 传到 Worker Process 并获得 Response，但是 Worker Process 则是通过同步的方式向 ASP.NET ISAPI 获得一些基于 Server 的变量。

IIS 6.0

在IIS 6.0中，为了解决第一个问题，ISAPI.dll被直接加载到工作进程中。为了解决第2个问题，引入了应用程序池（Application Pool）的机制。我们可以为一个或者多个Web应用创建应用程序池，每一个应用程序池对应一个独立的工作进程，从而为运行在不同应用程序池中的Web应用提供基于进程的隔离级别。IIS 6.0的工作进程名称为w3wp.exe。



IIS6与ASP.NET

对于IIS 6.0来说，W3SVC主要承载着三大功能：

HTTP请求接收：接收HTTP.SYS监听到的HTTP请求；

配置管理：从元数据库（Metabase）中加载配置信息对相关组件进行配置；

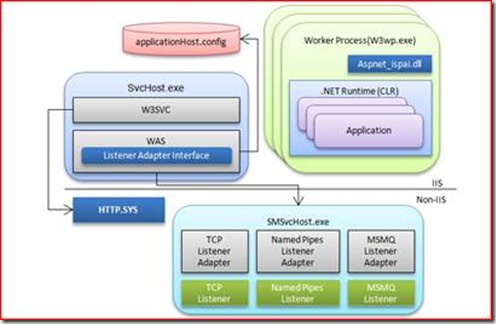
进程管理：创建、回收、监控工作进程。

IIS 7.0

在IIS 7.0，后两组功能被移入WAS（Windows进程激活服务）中，接收HTTP请求的任务依然落在W3SVC头上。WAS的引入为IIS 7.0一项前所未有的特性：同时处理HTTP和非HTTP请求（以前只可用于 HTTP 应用程序的 IIS 的所有功能现在都可用于运行 Windows Communication Foundation (WCF) 服务的应用程序）。

“管道模式”针对ASP.NET应用程序生命周期而言，区分为 经典模式和集成模式。IIS7.0集成管道是一种统一的请求处理管道，它同时支持本机代码和托管代码模块。实现IHttpModule接口的托管代码模块可访问该请求管道中的所有事件。例如，托管代码模块可用于 ASP.NET 网页（.aspx 文件）和 HTML 页（.htm 或 .html 文件）的 ASP.NET Forms 身份验证。在IIS6.0中，有两个请求处理管道(和上面介绍IIS6.0的一致)：1.用于本机代码ISAPI筛选器和扩展组件；2.用于托管代码应用程序组件。在IIS7.0 中，ASP.NET运行时与Web服务器集成，有了一个针对所有请求的统一请求处理管道。

IIS 7.0的整体构架以及整个请求处理流程:无论是从W3SVC接收到的HTTP请求，还是通过WCF提供的监听适配器接收到的请求，最终都会传递到WAS。如果相应的工作进程（或者应用程序池）尚未创建，其创建之；否则将请求分发给对应的工作进程进行后续的处理。WAS在进行请求处理过程中，通过内置的配置管理模块加载相关的配置信息对相关的组建进行配置，与IIS 5.x和IIS 6.0基于Metabase的配置信息存储不同的是，IIS 7.0大都将配置信息存放于XML形式的配置文件中。基本的配置存放在applicationHost.cofig中。

[](http://images.cnblogs.com/cnblogs_com/artech/WindowsLiveWriter/IISASP.NET_457/clip_image008_2.jpg)

IIS 7与ASP.NET

从上面对IIS 5.x和IIS 6.0的介绍中，我们不难发现这一点，IIS与ASP.NET是两个相互独立的管道（Pipeline），在各自管辖范围内，它们各自具有自己的一套机制对HTTP请求进行处理。两个管道通过ISAPI实现“联通”：IIS是第一道屏障，当对HTTP请求进行必要的前期处理（比如身份验证等）后，通过ISAPI将请求分发给ASP.NET管道。当ASP.NET在自身管道范围内完成对HTTP请求的处理后，处理后的结果再返回到IIS，IIS对其进行后期处理（比如日志记录、压缩等），最终生成HTTP响应（HTTP Response）。从另一个角度讲，IIS运行在非托管的环境中，而ASP.NET管道则是托管的，从这个意义上讲，ISAPI还是连接非托管环境和托管环境的纽带。图5反映了IIS 6.0与ASP.NET之间的桥接关系。

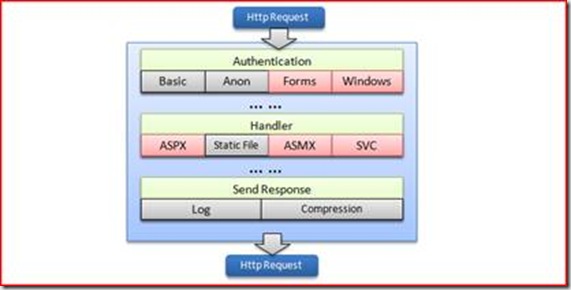
[](http://images.cnblogs.com/cnblogs_com/artech/WindowsLiveWriter/IISASP.NET_457/clip_image012_2.jpg)

基于IIS 6.0与ASP.NET双管道设计

对于Windows平台下的IIS来讲，ASP.NET无疑是一等公民，它们之间不应该是“井水不犯河水”的关系，而应该是“你中有我，我中有你”的关系。为此，在IIS 7.0中，实现了两者的集成。对于集成模式下的IIS 7.0，我们获得如下的好处。

允许我们通过本地代码（Native Code）和托管代码（Managed Code）两种方式定义IIS Module，这些IIS Module注册到IIS中形成一个通用的请求处理管道。由这些IIS Module组成的这个管道能够处理所有的请求，不论请求基于怎样的资源类型。比如，可以将FormsAuthenticationModule提供的Forms认证应用到基于.aspx，CGI和静态文件的请求。

将ASP.NET提供的一些强大的功能应用到原来难以企及的地方，比如将ASP.NET的URL重写功能置于身份验证之前；采用相同的方式去实现、配置、检测和支持一些服务器特性（Feature），比如Module、Handler映射、错误定制配置（Custom Error Configuration）等。



基于IIS 7.0与ASP.NET集成管道设计

以IIS 6.0为例，在工作进程w3wp.exe中，利用Aspnet\_ispai.dll加载.NET运行时（如果.NET运行时尚未加载）。IIS 6引入了应用程序池的概念，一个工作进程对应着一个应用程序池。一个应用程序池可以承载一个或者多个Web应用，每个Web应用映射到一个IIS虚拟目录。与IIS 5.x一样，每一个Web应用运行在各自的应用程序域中。

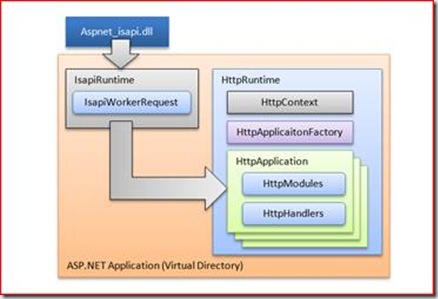
如果HTTP.SYS接收到的HTTP请求是对该Web应用的第一次访问，当成功加载了运行时后，会通过AppDomainFactory为该Web应用创建一个应用程序域（AppDomain）。随后，一个特殊的运行时IsapiRuntime被加载。IsapiRuntime定义在程序集System.Web中，对应的命名空间为System.Web.Hosting。IsapiRuntime会接管该HTTP请求。

IsapiRuntime会首先创建一个IsapiWorkerRequest对象，用于封装当前的HTTP请求，并将该IsapiWorkerRequest对象传递给ASP.NET运行时：HttpRuntime，从此时起，HTTP请求正式进入了ASP.NET管道。根据IsapiWorkerRequest对象，HttpRuntime会创建用于表示当前HTTP请求的上下文（Context）对象：HttpContext。

随着HttpContext被成功创建，HttpRuntime会利用HttpApplicationFactory创建新的或者获取现有的HttpApplication对象。实际上，ASP.NET维护着一个HttpApplication对象池，HttpApplicationFactory从池中选取可用的HttpApplication用户处理HTTP请求，处理完毕后将其释放到对象池中。HttpApplicationFactory负责处理当前的HTTP请求。

在HttpApplication初始化过程中，会根据配置文件加载并初始化相应的HttpModule对象。对于HttpApplication来说，在它处理HTTP请求的不同的阶段会触发不同的事件（Event），而HttpModule的意义在于通过注册HttpApplication的相应的事件，将所需的操作注入整个HTTP请求的处理流程。ASP.NET的很多功能，比如身份验证、授权、缓存等，都是通过相应的HttpModule实现的。

而最终完成对HTTP请求的处理实现在另一个重要的对象中：HttpHandler。对于不同的资源类型，具有不同的HttpHandler。比如.aspx页对应的HttpHandler为System.Web.UI.Page，WCF的.svc文件对应的HttpHandler为System.ServiceModel.Activation.HttpHandler。上面整个处理流程下图所示。

[](http://images.cnblogs.com/cnblogs_com/artech/WindowsLiveWriter/IISASP.NET_457/clip_image016_2.jpg)

详细可查阅《**[WCF技术剖析之二：再谈IIS与ASP.NET管道](http://www.cnblogs.com/artech/archive/2009/06/20/1507165.html)**》(<http://www.cnblogs.com/artech/archive/2009/06/20/1507165.html>)。

上传文件大小限制

默认情况下，IIS7的上传限制为200K。当上传文件小于30M时，可以通过如下方法设置：

在iis7中找到asp设置，在“asp”的“限制属性”中最后一行“最大请求主体限制”，修改该值为你所想要的，如2G（2000000000，单位为B）。当上传文件要求大于30M时，继续如下修改：

1、停止IIS7

2、找到“C:\Windows\System32\inetsrv\config\schema\IIS\_schema.xml”文件。

这个文件是只读的，即使用管理员权限也不能修改。要先修改文件的权限，然后去掉只读属性才可以。

1.右键文件->属性->安全，选中目标用户，点击高级，修改文件所有者；

2.确定后点击编辑，就可以修改当前用户的权限了，添加“写入”权限。至此，权限设置OK了。

3.将文件的只读属性去掉。

用记事本打开该文件，找到“<attribute name="maxAllowedContentLength" type="uint" defaultValue="30000000" />”，将“30000000”修改为你想要的值（如2000000000）保存。

将“C:\Windows\System32\inetsrv\config\schema\IIS\_schema.xml”文件加上只读属性。

但是，win2008最大只能上传小于2G的文件。这个要注意。亲测有效的，文件所有者不用改回来。

简化的流程

当请求一个\*.aspx文件的时候，这个请求会被inetinfo.exe进程截获，它判断文件的后缀（aspx）之后，将这个请求转交给 ASPNET\_ISAPI.dll，ASPNET\_ISAPI.dll会通过http管道（Http PipeLine）将请求发送给ASPNET\_WP.exe进程，在ASPNET\_WP.exe进程中通过HttpRuntime来处理这个请求，处理完毕将结果返回客户端。

ASPNET\_ISAPI.DLL：是处理.aspx文件的win32组件。其实IIS服务器是只能识别.html文件的，当IIS服务器发现被请求的文件是.aspx文件时，IIS服务器将其交给aspnet\_isapi.dll来处理。

aspnet\_wp.exe进程：ASP.NET框架进程，提供.net运行的托管环境，.net的CLR(公共语言运行时)就是寄存在此进程中。

管道模型

ASP.NET 请求处理过程是基于管道模型的，管道模型其实就是一个按照流程对请求和响应依次处理的过程。比如ASP.NET管道模型是由多个HttpModule和HttpHandler组成，ASP.NET把http请求依次传递给管道中 各个HttpModule，最终被HttpHandler处理，处理完成后，再次经过管道中的HTTP模块，把结果返回给客户端。我们可以在每个 HttpModule中都可以干预请求的处理过程。

HttpApplication对象是ASP.NET中处理请求的重要对象，但是，这种类型的对象实例不是由程序员来创建的，而是由ASP.NET帮助我们创建的。

为了便于扩展处理工作，HttpApplication采用处理管道的方法进行处理，将处理的过程分为多个步骤，每个步骤通过事件的形式暴露给程序员，这些事件按照固定的处理顺序依次触发，程序员通过编写事件处理方法就可以自定义每一个请求的扩展处理过程。

对于ASP.NET WebForm来说，后台代码类是前台页面类的父类，前台页面类则是后台代码类的子类。这一点，可以通过查看每个aspx文件中的头部，我们都会看到以下的一句代码：

<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="FirstPage.aspx.cs" Inherits="WebFormDemo.FirstPage" %>

具体详见<http://www.cnblogs.com/edisonchou/p/4216337.html>

对于ASP.NET MVC来说，

ASP.NET MVC 4的源码：[http://pan.baidu.com/s/1bnF8ZPt](http://pan.baidu.com/s/1bnF8ZPt" \t "_blank" \o "mvc 4 source code)

IIS服务器网站不支持带加号“+”的目录或文件名的解决方案

网站文件目录如/js/jquery-1.6.2+fix-9521.js在某些IIS服务器中是不支持的，解决方法如下：

打开IIS，点击对应网站（或者当前服务器名，这样是设置整个服务器的）；

找到 IIS >> 请求筛选（双击）；

看右侧，找到“编辑功能设置”，点击进入；

勾选“允许双重转义”；完成。

“#”的文件名无法下载的解决方案

<http://120.79.95.62:8002/resource/temp/多项四#-TJ-001.docx>是下载不下来的，需要改成%23：

[http://120.79.95.62:8002/resource/temp/多项四%23-TJ-001.docx](http://120.79.95.62:8002/resource/temp/%E5%A4%9A%E9%A1%B9%E5%9B%9B%23-TJ-001.docx)

这样就可以把多项四#-TJ-001.docx文件下载下来了。

# 二十五web.config

web.config是asp.net中保存配置信息（比如数据库连接字符串等）的重要文件。它是基于xml的文本文件方式放在Web应用程序的任何目录中，并且默认不随源文件编译到Dll中，而运行环境随时监视着它是否有改变，一但有变动，系统会自动重新加载里面的最新内容。

IIS在Asp.net网站启动时，会加载配置文件中的配置信息，然后缓存这些信息，不会每次要用都去读取配置文件，只是IIS会随时监视着这些文件的变化，一量有变化，它会重新去读取并缓存配置信息。

Asp.net网站运行时会按照以下方式加载配置文件中的节点信息：

1) 如果在当前运行页面所在的目录下有web.config文件，则查找是否存在所需要的节点，如果存在则返回结果，并停止下一步地查找。

2) 如里所在目录不存在web.config配置或者配置文件里没有所需要的节点，则查找它所在的上一级目录的配置文件中的节点，直到网站根目录。

3) 如果网站根目录中都不存在web.config或者所需要的配置节点，转而到“windows目录\Microsoft.NET\Framework\对应.net版本\config\web.config”中去查找。

4) 如果第3条中还没找到，继续到“windows目录\Microsoft.NET\Framework\对应.net版本\config\machine.config”中去查找。

5) 如果还没找到，那就报错吧。

在配置文件的表示字段上,尽量不要有重复的。

具体参见《[asp.net web.config的学习笔记](http://www.cnblogs.com/Bulid-For-NET/archive/2013/01/11/2856632.html)》(<http://www.cnblogs.com/Bulid-For-NET/archive/2013/01/11/2856632.html>)

# 二十六 常用的类或必须掌握的类

RestSharp

Simple REST and HTTP API Client for .NET

具体使用：

var inputPara = new PublishInputParam();

inputPara.taskId = Guid.NewGuid().ToString();

inputPara.publishTaskList = publishTaskList;

inputPara.needDrawingCode = cbNeedDrawingCode.Checked;

var client = new RestClient("https://58.240.86.44:23589/srfa/");

var request = new RestRequest("publish/start", Method.POST);

request.AddHeader("Authorization", Utility.getToken());

request.AddJsonBody(inputPara);

IRestResponse response = client.Execute(request);

var content = response.Content;

HttpContext类

HttpContext类它对Request、Respose、Server等等都进行了封装,并保证在整个请求周期内都可以随时随地的调用。该对象提供对请求的内部Request、Response 和 Server 属性的访问。HttpContext的命名空间：System.Web（在 system.web.dll 中）;除了对几个对象进行封装外它还有个HttpContext.Item，通过它你可以在HttpContext的生存周期内提前存储一些临时的数据，方便随时使用。

生存周期：从客户端用户点击并产生了一个向服务器发送请求开始---服务器处理完请求并生成返回到客户端为止。针对每个不同用户的请求，服务器都会创建一个新的HttpContext实例直到请求结束,服务器销毁这个实例。

HttpContet对象之Request

ASP.NET Request 封装了客户端请求信息，是从客户端得到数据(从浏览器获取数据);常用的三种取得数据的方法是：Request.Form、Request.QueryString、Request其第三种是前两种的一个缩写，可以取代前两种情况。而前两种主要对应的Form提交时的两种不同的提交方法：分别是Post方法和Get方法。

public partial class RequestPage : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

StringBuilder strBuider = new StringBuilder();

strBuider.Append("客户端主机的IP地址：" + HttpContext.Current.Request.UserHostAddress + "</br>");

strBuider.Append("客户端浏览器版本:" + Request.UserAgent + "</br>");

strBuider.Append("当前由哪个页面URL跳转过来:" + Request.UrlReferrer+ "</br>");

strBuider.Append("当前要求的URL:" + Request.Url + "</br>");

strBuider.Append("当前要求的URL绝对地址:" + Request.Url.AbsolutePath + "</br>");

strBuider.Append("当前要求的URL绝对URI:" + Request.Url.AbsoluteUri + "</br>");

strBuider.Append("当前要求的URL主机名跟端口:" + Request.Url.Authority + "</br>");

strBuider.Append("当前要求的URL实例主机的一部分:" + Request.Url.Host + "</br>");

strBuider.Append("当前要求的URL端口:" + Request.Url.Port + "</br>");

strBuider.Append("当前要求的URL的内容第一个段:" + Request.Url.Segments[0] + "</br>");

strBuider.Append("当前要求的URL的内容第二个段:" + Request.Url.Segments[1] + "</br>");

strBuider.Append("当前要求的URL的内容第三个段:" + Request.Url.Segments[2] + "</br>");

strBuider.Append("浏览器地址栏后的参数"+Request.QueryString + "</br>");

strBuider.Append("当前网页在服务器端的实际路径:" + Request.PhysicalPath + "</br>");

strBuider.Append("当前文件的物理地址:" + Request.PhysicalApplicationPath + "</br>");

strBuider.Append("当前网页的相对地址:" + Request.Path + "<br/>");

strBuider.Append("当前页面的URL:" + Request.RawUrl + "<br/>");

strBuider.Append("客户端上传的文件（个数）:" + Request.Files.Count + "<br/>");

strBuider.Append("当前执行网页的相对地址:" + Request.FilePath + "<br/>");

strBuider.Append("客户端浏览器的信息:" + Request.Browser + "<br/>");

strBuider.Append("当前运行程序的服务器端虚拟目录:" + Request.ApplicationPath + "<br/>");

strBuider.Append("客户端浏览器的字符设置:" + Request.ContentEncoding + "<br/>");

Response.Write(strBuider.ToString());

}

}

}

HttpContent对象之Response

Response代表了服务器响应对象。每次客户端发出一个请求的时候，服务器就会用一个响应对象来处理这个请求，处理完这个请求之后，服务器就会销毁这个相应对象，以便继续接受其它客服端请求。

详情请参见《[ASP.NET基础之HttpContext学习](http://www.cnblogs.com/wujy/p/3264475.html)》(<http://www.cnblogs.com/wujy/p/3264475.html>)

# 二十七Ninject

Ninject是一个IOC容器用来解决程序中组件的耦合问题，它的目的在于做到最少配置。其他的的IOC工具过于依赖配置文件，需要使用assembly-qualified名称来进行定义，庸长且复杂常常因为打错字而破坏程序。这些是他的优点，也是为什么要选择它。Ninject同时不能进行热插拔。

使用NuGet下载，

using Ninject;

注册：

IKernel ninjectKernel = newStandardKernel();

ninjectKernel.Bind<DI和IOC容器.IValueCalculater>().To<DI和IOC容器.LinqValueCalculater>(); //想使用的类型和他的接口进行绑定

// LinqValueCalculater继承实现了接口IValueCalculater

DI和IOC容器.IValueCalculater v = ninjectKernel.Get<DI和IOC容器.IValueCalculater>();

等同于：

DI和IOC容器.IValueCalculater v = new DI和IOC容器.LinqValueCalculater();

也等同于：

DI和IOC容器.IValueCalculater v = (DI和IOC容器.IValueCalculater)ninjectKernel.Get(typeof(DI和IOC容器.IValueCalculater));

参见：<http://www.cnblogs.com/tylerdonet/p/3297915.html>

# 二十七 应知应会

编译型和解释型编程语言

计算机高级语言按程序的执行方式可以分为编译型和解释型两种。

编译型语言是指使用专门的编译器，针对特定平台（操作系统）将某种高级语言源代码一次性“翻译”成可被该平台硬件执行的机器码（包括机器指令和操作数），并包装成该平台所能识别的可执行性程序的格式，这个转换过程称为编译（Compile）。编译生成的可执行性程序可以脱离开发环境，在特定的平台上独立运行。

有些程序编译结束后，还可能需要对其他编译好的目标代码进行链接，即组装两个以上的目标代码模块生成最终的可执行性程序，通过这种方式实现低层次的代码复用。

因为编译型语言是一次性地编译成机器码，所以可以脱离开发环境独立运行，而且通常运行效率较高；但因为编译型语言的程序被编译成特定平台上的机器码，因此编译生成的可执行性程序通常无法移植到其他平台上运行；如果需要移植，则必须将源代码复制到特定平台上，针对特定平台进行修改，至少也需要采用特定平台上的编译器重新编译。

现有的C、C++、FORTRAN、Pascal等高级语言都属于编译型语言。

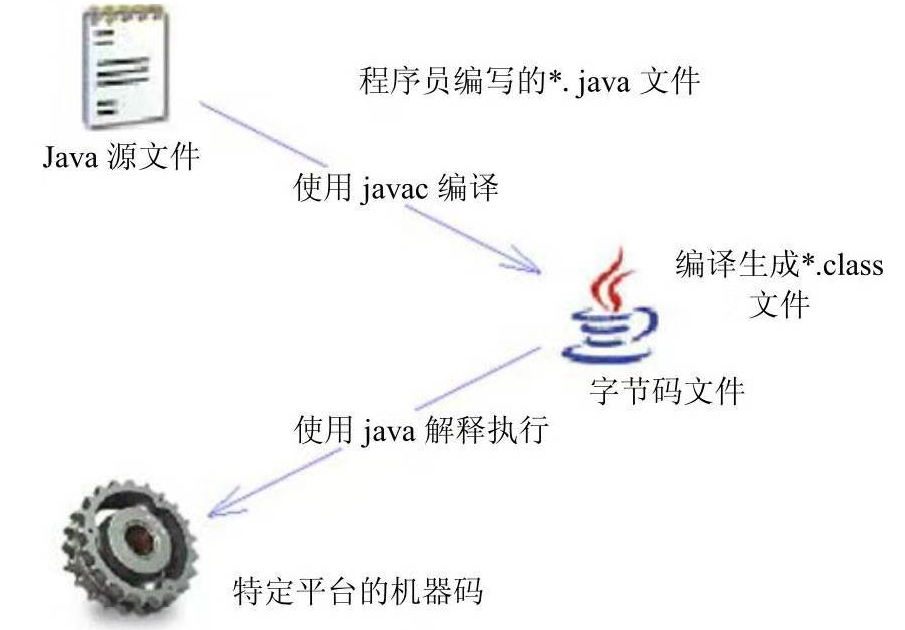
解释型语言是指使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行的语言。解释型语言通常不会进行整体性的编译和链接处理，解释型语言相当于把编译型语言中的编译和解释过程混合到一起同时完成。

可以认为：每次执行解释型语言的程序都需要进行一次编译，因此解释型语言的程序运行效率通常较低，而且不能脱离解释器独立运行。但解释型语言有一个优势：跨平台比较容易，只需提供特定平台的解释器即可，每个特定平台上的解释器负责将源程序解释成特定平台的机器指令即可。解释型语言可以方便地实现源程序级的移植，但这是以牺牲程序执行效率为代价的。

现有的Ruby、Python等语言都属于解释型语言。

除此之外，还有一种伪编译型语言，如Visual Basic，它属于半编译型语言，并不是真正的编译型语言。它首先被编译成P-代码，并将解释引擎封装在可执行性程序内，当运行程序时，P-代码会被解析成真正的二进制代码。P-代码其实就是微软中间语言。表面上看起来，Visual Basic可以编译生成可执行性的EXE文件，而且这个EXE文件也可以脱离开发环境，在特定平台上运行，非常像编译型语言。实际上，在这个EXE文件中，既有程序的启动代码，也有链接解释程序的代码，而这部分代码负责启动Visual Basic解释程序，再对Visual Basic代码进行解释并执行（微软中间语言的作用）。

Java语言比较特殊，由Java语言编写的程序需要经过编译步骤，但这个编译步骤并不会生成特定平台的机器码，而是生成一种与平台无关的字节码（也就是\*.class文件，类似.net中的exe微软中间语言）。当然，这种字节码不是可执行性的，必须使用Java解释器来解释执行。因此，我们可以认为：Java语言既是编译型语言，也是解释型语言。或者说，Java语言既不是纯粹的编译型语言，也不是纯粹的解释型语言。Java程序的执行过程必须经过先编译，后解释两个步骤，如下图所示。



当使用Java编译器编译Java程序时，生成的是与平台无关的字节码，这些字节码不面向任何具体平台，只面向JVM。不同平台上的JVM都是不同的，但它们都提供了相同的接口。JVM是Java程序跨平台的关键部分，只要为不同平台实现了相应的虚拟机，编译后的Java字节码就可以在该平台上运行。

Java程序源文件的后缀必须是.java，不能是其他文件后缀名。

语言的类名和文件名

[插图] 在通常情况下，Java程序源文件的主文件名可以是任意的。但有一种情况例外：如果Java程序源代码里定义了一个public类，则该源文件的主文件名必须与该public类（也就是该类定义使用了public关键字修饰）的类名相同。

由于Java程序源文件的文件名必须与public类的类名相同，因此，一个Java源文件里最多只能定义一个public类。

注意：

一个Java源文件可以包含多个类定义，但最多只能包含一个public类定义；如果Java源文件里包含public类定义，则该源文件的文件名必须与这个public类的类名相同。

.NET不存在这样的限制。

编码格式（乱码的缘由）

不管是磁盘还是网络传输，最小的存储单元都是字节，而不是字符，所以I/O操作的都是字节而不是字符。

数据持久化或网络传输都是以字节进行的，所以必须要有字符到字节或字节到字符的转化，在这个转化过程中，如未指定具体的编码格式，系统会使用默认的编码格式，由于各终端默认编码的差异，有可能产生乱码。

但是各编程语言为什么有操作字符的I/O接口呢？这是因为我们的程序中通常操作的数据都是字符形式的，为了操作方便当然要提供一个直接写字符的I/O接口，如此而已。

对于java来说，InputStreamReader类是字节到字符的转化桥梁，InputStream到Reader的过程要指定编码字符集，否则将采用操作系统默认字符集，很可能会出现乱码问题。StreamDecoder正是完成字节到字符的解码的实现类。也就是当你用如下方式读取一个文件时：

try {

StringBuffer str = new StringBuffer();

char[] buf = new char[1024];

FileReader f = new FileReader("file");

while(f.read(buf)>0){

str.append(buf);

}

str.toString();

} catch (IOException e) {

}

FileReader类就是按照上面的工作方式读取文件的，FileReader继承了InputStreamReader类，实际上是读取文件流，然后通过StreamDecoder解码成char，只不过这里的解码字符集是默认字符集。

## URL汉字编码问题以及乱码解决

URL就是网址，只要上网，就一定会用到。注：URL复制到浏览器再复制回来是可以的，但是如果复制到浏览器，回车了，再复制到微信或文本框，中文会变成%。

一般来说，URL只能使用英文字母、阿拉伯数字和某些标点符号，不能使用其他文字和符号。比如，世界上有英文字母的网址 “http://www.abc.com”，但是没有希腊字母的网址“http://www.aβγ.com”（读作阿尔法-贝塔-伽玛.com）。这是因为网络标准[RFC 1738](http://www.ietf.org/rfc/rfc1738.txt" \t "_blank) 做了硬性规定. 这意味着，如果URL中有汉字，就必须编码后使用。但是麻烦的是，RFC 1738没有规定具体的编码方法，而是交给应用程序（浏览器）自己决定。这导致“URL编码”成为了一个混乱的领域。

下面就让我们看看，“URL编码”到底有多混乱。我会依次分析四种不同的情况，在每一种情况中，浏览器的URL编码方法都不一样。把它们的差异解释清楚之后，我再说如何用Javascript找到一个统一的编码方法。

<https://blog.csdn.net/shabijiayong/article/details/79094438>

无论网页使用什么字符集，IE传送给服务器的总是“q=%B4%BA%BD%DA”，而Firefox传送给服务器的总是“q=%E6%98 %A5%E8%8A%82”。也就是说，**在Ajax调用中，IE总是采用GB2312编码（操作系统的默认编码），而Firefox总是采用utf-8编码。这就是我们的结论4。**

假定前面你都看懂了，那么此时你应该会感到很头痛。因为，实在太混乱了。不同的操作系统、不同的浏览器、不同的网页字符集，将导致完全不同的编码结果。如果程序员要把每一种结果都考虑进去，是不是太恐怖了？有没有办法，能够保证客户端只用一种编码方法向服务器发出请求？

回答是有的，就是使用Javascript先对URL编码，然后再向服务器提交，不要给浏览器插手的机会。因为Javascript的输出总是一致的，所以就保证了服务器得到的数据是格式统一的。

Javascript语言用于编码的函数，一共有三个，最古老的一个就是escape()。虽然这个函数现在已经不提倡使用了，但是由于历史原因，很多地方还在使用它，所以有必要先从它讲起。

实际上，escape()不能直接用于URL编码，它的真正作用是返回一个字符的Unicode编码值。比如“春节”的返回结果是%u6625%u8282，也就是说在Unicode字符集中，“春”是第6625个（十六进制）字符，“节”是第8282个（十六进制）字符。它的具体规则是，除了ASCII字母、数字、标点符号“@ \* \_ + - . /”以外，对其他所有字符进行编码。在\u0000到\u00ff之间的符号被转成%xx的形式，其余符号被转成%uxxxx的形式。对应的解码函数是 unescape()。

还有两个地方需要注意:首先，无论网页的原始编码是什么，一旦被Javascript编码，就都变为unicode字符。也就是说，Javascipt函数的输入和输出，默认都是Unicode字符。这一点对下面两个函数也适用。其次，escape()不对“+”编码。但是我们知道，网页在提交表单的时候，如果有空格，则会被转化为+字符。服务器处理数据的时候，会把+号处理成空格。所以，使用的时候要小心。

encodeURI()是Javascript中真正用来对URL编码的函数。它着眼于对整个URL进行编码，因此除了常见的符号以外，对其他一些在网址中有特殊含义的符号“; / ? : @ & = + $ , #”，也不进行编码。编码后，它输出符号的utf-8形式，并且在每个字节前加上%。它对应的解码函数是decodeURI()。需要注意的是，它不对单引号'编码。

最后一个Javascript编码函数是encodeURIComponent()。与encodeURI()的区别是，它用于对URL的组成部分进行个别编码，而不用于对整个URL进行编码。

因此，“; / ? : @ & = + $ , #”，这些在encodeURI()中不被编码的符号，在encodeURIComponent()中统统会被编码。至于具体的编码方法，两者是一样。

它对应的解码函数是decodeURIComponent()。

IO

I/O问题可以说是当今Web应用中所面临的主要问题之一，因为在当前这个海量数据时代，数据在网络中随处流动。这个流动的过程中都涉及I/O问题，可以说大部分Web应用系统的瓶颈都是I/O瓶颈。

不管是磁盘还是网络传输，最小的存储单元都是字节，而不是字符，所以I/O操作的都是字节而不是字符。

## HTTPS

HTTP是明文传输的，也就意味着，介于发送端、接收端中间的任意节点都可以知道你们传输的内容是什么。这些节点可能是路由器、代理等。

密码学中的“密码”术语与网站登录时用的密码（password）是不一样的概念，password 翻译过来其实是“口令”，它是用于认证用途的一组文本字符串。而密码学中的密码（cipher）是一套算法(algorithm)，这套算法用于对消息进行加密和解密，从明文到密文的过程称之为加密，密文反过来生成明文称之为解密，加密算法与解密算法合在一起称为**密码算法**。有关密码和认证机构等相关知识，请参见密码和加密-《HTTPS 为什么更安全，先看这些》一文。

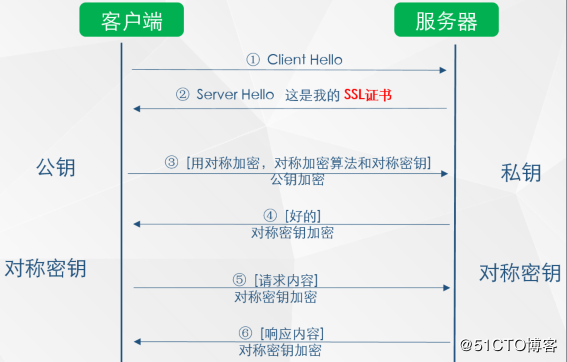
HTTPS其实就是**secure http**的意思，也就是HTTP的安全升级版。HTTP是应用层协议，位于HTTP协议之下是传输协议TCP。TCP负责传输，HTTP则定义了数据如何进行包装。HTTPS相对于HTTP其实就是在HTTP跟TCP中间加多了一层加密层**TLS/SSL**。

https=http+ssl，顾名思义，https是在http的基础上加上了SSL保护壳，信息的加密过程就是在SSL中完成的。

SSL是个加密套件，负责对HTTP的数据进行加密。TLS是SSL的升级版。现在提到HTTPS，加密套件基本指的是TLS。

具体的内容可以参见密码和加密-《HTTPS 科普扫盲帖》。

在获取公钥的过程中，我们又引出了一个新的话题：**如何安全的获取公钥，并确保公钥的获取是安全的**， 那就需要用到终极武器了：SSL 证书（需要购买）和CA机构。



如上图所示，在第 ② 步时服务器发送了一个SSL证书给客户端，SSL 证书中包含的具体内容有证书的颁发机构、有效期、公钥、证书持有者、签名，通过第三方的校验保证了身份的合法，解决了公钥获取的安全性

以浏览器为例说明如下整个的校验过程：

（1）首先浏览器读取证书中的证书所有者、有效期等信息进行一一校验

（2）浏览器开始查找操作系统中已内置的受信任的证书发布机构CA，与服务器发来的证书中的颁发者CA比对，用于校验证书是否为合法机构颁发

（3）如果找不到，浏览器就会报错，说明服务器发来的证书是不可信任的。

（4）如果找到，那么浏览器就会从操作系统中取出  颁发者CA  的公钥，然后对服务器发来的证书里面的签名进行解密

（5）浏览器使用相同的hash算法计算出服务器发来的证书的hash值，将这个计算的hash值与证书中签名做对比

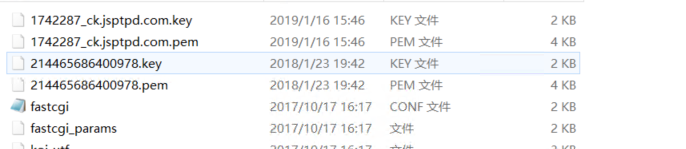
（6）对比结果一致，则证明服务器发来的证书合法，没有被冒充

（7）此时浏览器就可以读取证书中的公钥，用于后续加密了

由于HTTPS涉及到中间机构的校验，且这个校验的过程不是无偿的，需要收费，因为需要像第三方机构申请CA证书。

申请对应的证书，如公司阿里云申请的证书：





下载对应web服务器的版本，如nginx/tomcat/IIS。

笔者是采用的nginx+tomcat的架构方式，因此配置的方式以nginx+tomcat的方式进行讲解

在nginx的配置文件中，新增如下配置项，在这个地方有一个参数：ssl on，如果这个参数开启，http和https将不能共存。里面对应的信息都可以通过CA机构获取到

listen 443 ssl;

ssl\_certificate /iyunwen/server/ssl/20180731.cer;

ssl\_certificate\_key /iyunwen/server/ssl/20180731.key;

ssl\_prefer\_server\_ciphers on;

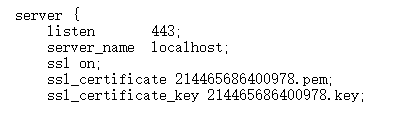
ssl\_session\_timeout 10m;

ssl\_session\_cache shared:SSL:10m;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers "EECDH+ECDSA+AESGCM EECDH+aRSA+AESGCM EECDH+ECDSA+SHA384 EECDH+ECDSA+SHA256 EECDH+aRSA+SHA384 EECDH+aRSA+SHA256 EECDH+aRSA+RC4 EECDH EDH+aRSA !aNULL !eNULL !LOW !3DES !MD5 !EXP !PSK !SRP !DSS !RC4";

我们公司：



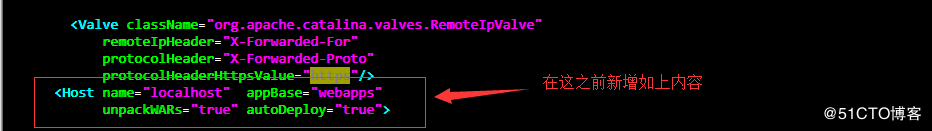
但配置nginx之后，对于tomcat需要在配置文件conf/server.xml文件中新增如下内容

<Valve className="org.apache.catalina.valves.RemoteIpValve"

remoteIpHeader="X-Forwarded-For"

protocolHeader="X-Forwarded-Proto"

protocolHeaderHttpsValue="https"/>



至此关于HTTPS的所有内容已经结束。

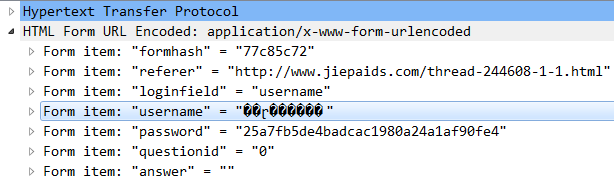
## 同源策略(跨域)

同源策略/SOP（Same origin policy）是一种约定，由Netscape公司1995年引入浏览器，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSFR等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源(只要三者不相同，其实都是跨域)。

## 抓包

有时，有的网站并不是简单地就可以抓取，而是需要登录后才能操作，如何获取登录时需要的参数呢，这时我们就需要使用抓包工具来进行了。

抓包工具有很多，小到最常用的web调试工具firebug，还有fiddler，大到通用的强大的抓包工具wireshark。我们使用的是wireshark：



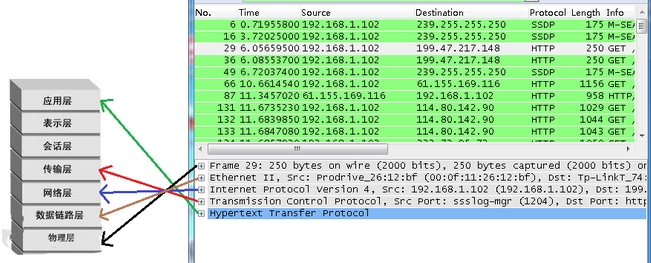
应该可以看出参数了。

我用基金的试了一下：



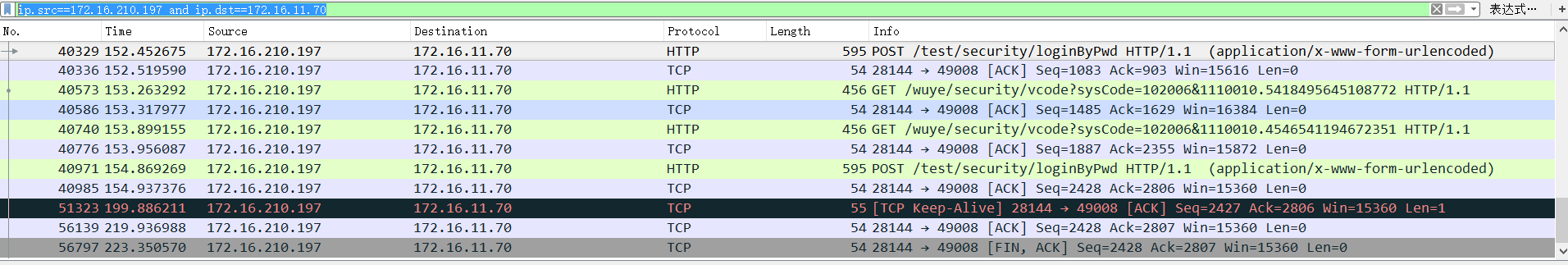
“帮助”-“wiki”：可以打开Wireshark wiki。

### Wireshark详细信息的层与对应OSI七层模型

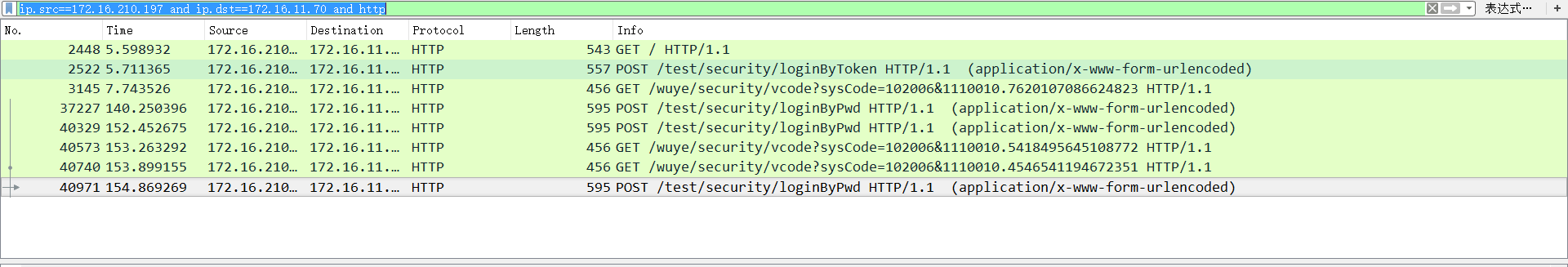


### 过滤器

ip.src==172.16.210.197 and ip.dst==172.16.11.70



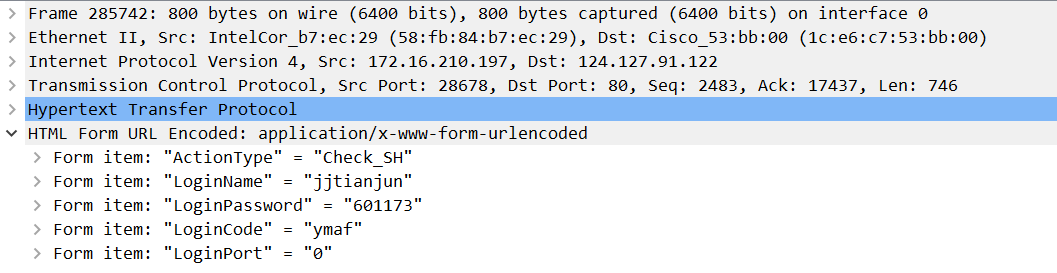
ip.src==172.16.210.197 and ip.dst==172.16.11.70 and http



每次更换过滤规则要按一下右侧的箭头或回车。

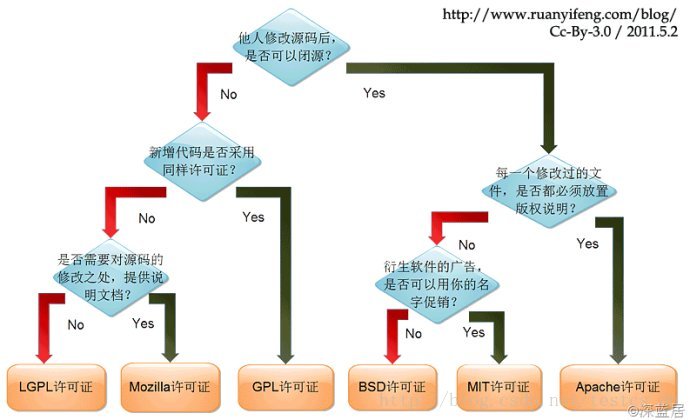
http.host=="jj.chinapostdoctor.org.cn" and http

访问基金网。



开源协议

# 一张图看懂开源许可协议



https://blog.csdn.net/testcs\_dn/article/details/38496107

常见加密算法

AES 可逆

加密AES的key和iv。

AES中的Key的取值首先取决于KeySize设定的值（128、192、256 ---- 单位为：位，即1个字节等于8位的位）：  
所以如果KeySize设定的值为128，则Key为一个16字节的byte[]  
如果KeySize设定的值为192，则Key为一个24字节的byte[]  
如果KeySize设定的值为256，则Key为一个32字节的byte[]

AES中的IV只能为一个16字节的byte[]

/// <summary>

/// AES 加密

/// </summary>

/// <param name="str">明文（待加密）</param>

/// <param name="key">密文</param>

/// <returns></returns>

public static string AesEncrypt(string text, string password, string iv)

{

RijndaelManaged rm = new RijndaelManaged();

//rm.Key = Encoding.UTF8.GetBytes("cRnd7cdPWPv5HrjB4Q-C");

rm.Mode = CipherMode.CBC;

rm.Padding = PaddingMode.PKCS7;

rm.KeySize = 128;

rm.BlockSize = 128;

byte[] pwdBytes = System.Text.Encoding.UTF8.GetBytes(password);

byte[] keyBytes = new byte[16];

int len = pwdBytes.Length;

if (len > keyBytes.Length)

{

len = keyBytes.Length;

}

System.Array.Copy(pwdBytes, keyBytes, len);

rm.Key = keyBytes;

byte[] ivBytes = System.Text.Encoding.UTF8.GetBytes(iv);

rm.IV = ivBytes;

ICryptoTransform iCryptoTransform = rm.CreateEncryptor();

byte[] textBytes = Encoding.UTF8.GetBytes(text);

byte[] targetBytes = iCryptoTransform.TransformFinalBlock(textBytes, 0, textBytes.Length);

return Convert.ToBase64String(targetBytes);

}

/// <summary>

/// AES解密

/// </summary>

/// <param name="text">密文</param>

/// <param name="password">密钥</param>

/// <param name="iv">向量</param>

/// <returns></returns>

public static string AESDecrypt(string text, string password, string iv)

{

RijndaelManaged rijndaelManaged = new RijndaelManaged();

rijndaelManaged.Mode = CipherMode.CBC;

rijndaelManaged.Padding = PaddingMode.PKCS7;

rijndaelManaged.KeySize = 128;

rijndaelManaged.BlockSize = 128;

byte[] encryptedData = Convert.FromBase64String(text);

byte[] pwdBytes = System.Text.Encoding.UTF8.GetBytes(password);

byte[] keyBytes = new byte[16];

int len = pwdBytes.Length;

if (len > keyBytes.Length)

{

len = keyBytes.Length;

}

System.Array.Copy(pwdBytes, keyBytes, len);

rijndaelManaged.Key = keyBytes;

byte[] ivBytes = System.Text.Encoding.UTF8.GetBytes(iv);

rijndaelManaged.IV = ivBytes;

ICryptoTransform iCryptoTransform = rijndaelManaged.CreateDecryptor();

byte[] targetBytes = iCryptoTransform.TransformFinalBlock(encryptedData, 0, encryptedData.Length);

return Encoding.UTF8.GetString(targetBytes);

}

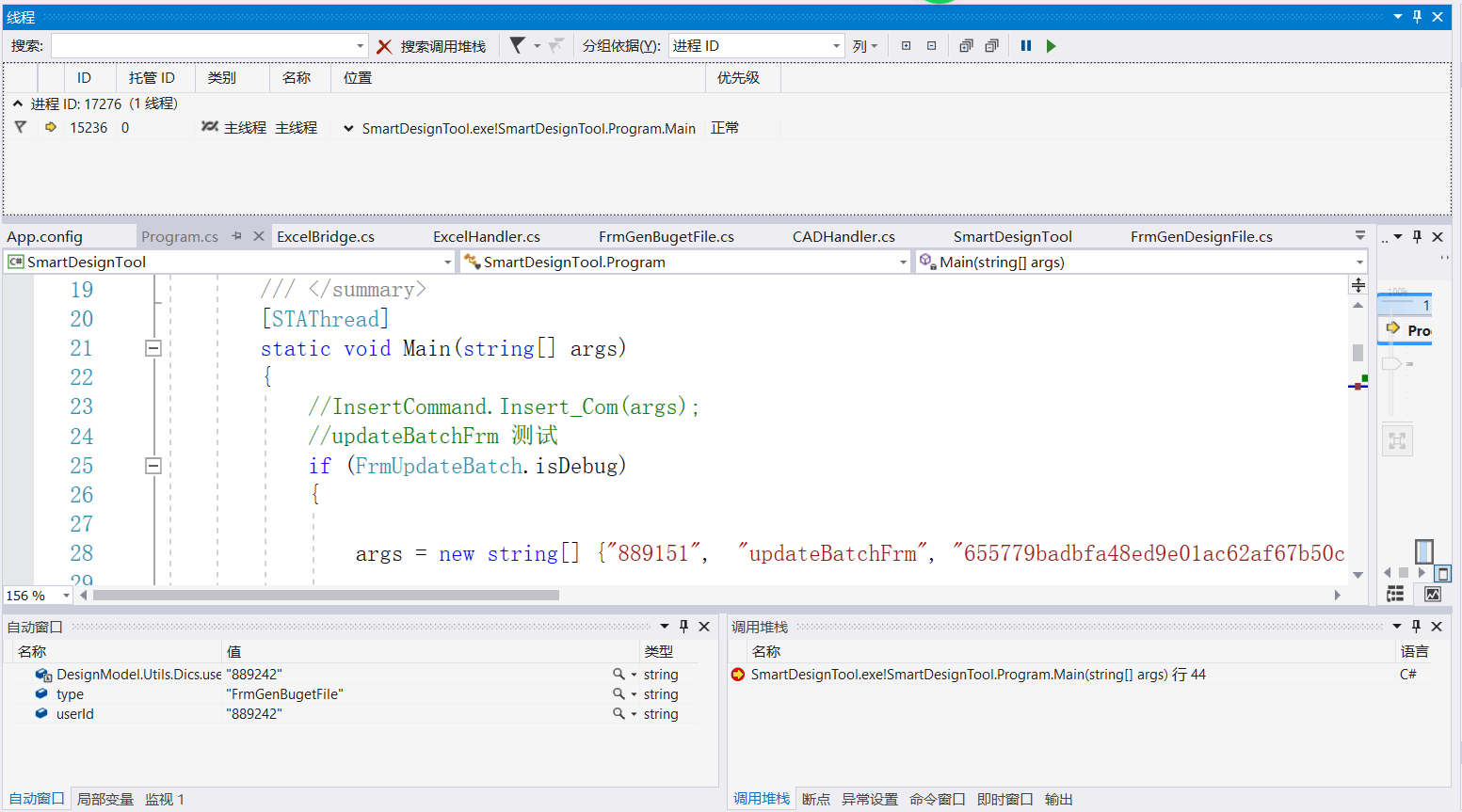
多线程Debug

VS 2017

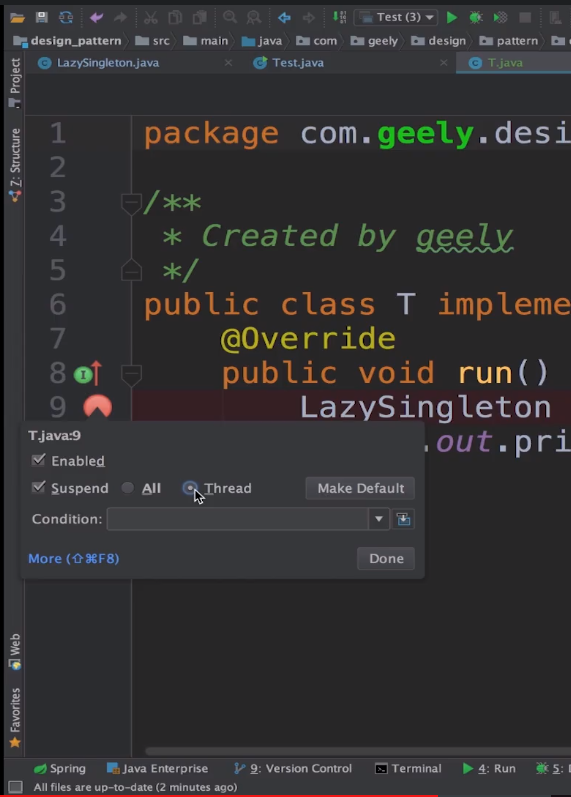
调试多线程应用程序时，“并行堆栈”窗口非常有用。 它有多个视图：

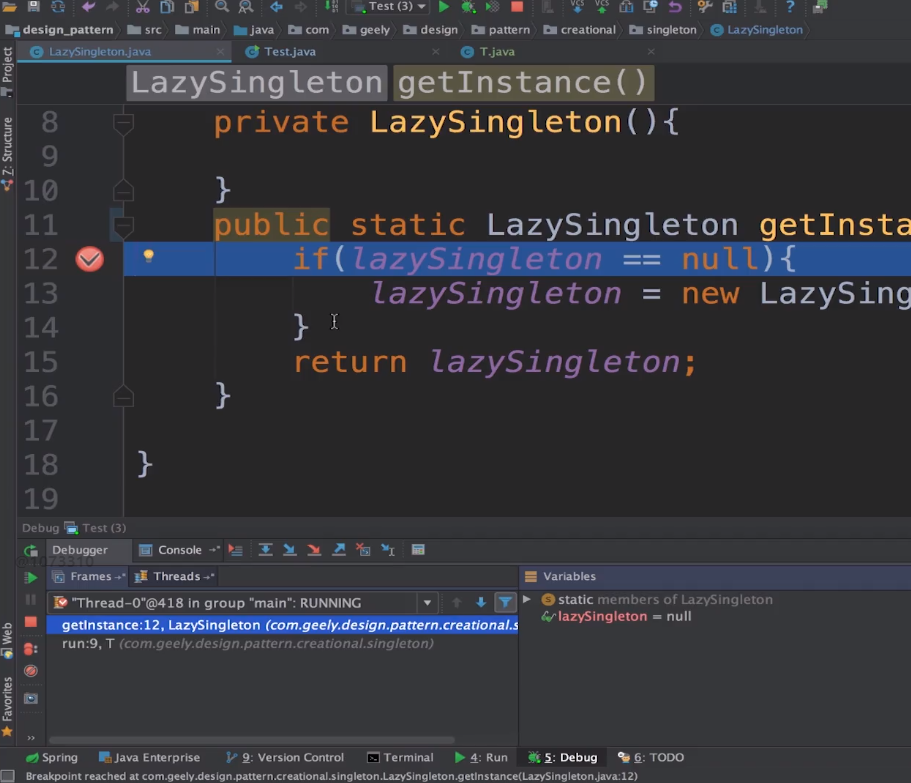
* [“线程”视图](https://docs.microsoft.com/zh-cn/visualstudio/debugger/using-the-parallel-stacks-window?view=vs-2017#threads-view)显示应用中所有线程的调用堆栈信息。 可以在线程和这些线程上的堆栈帧之间进行导航。
* [任务视图](https://docs.microsoft.com/zh-cn/visualstudio/debugger/using-the-parallel-stacks-window?view=vs-2017#tasks-view)显示以任务为中心的调用堆栈信息。
  + 在托管代码中，“任务”视图示 [System.Threading.Tasks.Task](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task) 对象的调用堆栈。
  + 在本机代码中，“任务”视图显示[任务组](https://docs.microsoft.com/zh-cn/cpp/parallel/concrt/task-parallelism-concurrency-runtime)、[并行算法](https://docs.microsoft.com/zh-cn/cpp/parallel/concrt/parallel-algorithms)、[异步代理](https://docs.microsoft.com/zh-cn/cpp/parallel/concrt/asynchronous-agents)和[轻量级任务](https://docs.microsoft.com/zh-cn/cpp/parallel/concrt/task-scheduler-concurrency-runtime)的调用堆栈。
* [方法视图](https://docs.microsoft.com/zh-cn/visualstudio/debugger/using-the-parallel-stacks-window?view=vs-2017#method-view)以选定方法为中心显示调用堆栈。

只能在调试会话中打开“并行堆栈”窗口。 选择“调试” > “窗口” > “并行堆栈” 。



IDEA





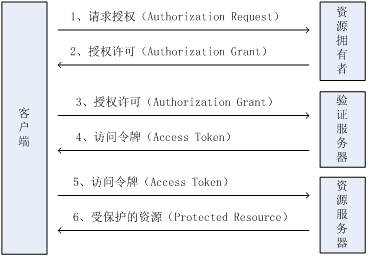
常见的后台认证机制

HTTP Basic Auth

HTTP Basic Auth简单点说明就是每次请求API时都提供用户的username和password，简言之，Basic Auth是配合RESTful API 使用的最简单的认证方式，只需提供用户名密码即可，但由于有把用户名密码暴露给第三方客户端的风险，在生产环境下被使用的越来越少。因此，在开发对外开放的RESTful API时，尽量避免采用HTTP Basic Auth。

### OAuth

OAuth（开放授权）是一个开放的授权标准，允许用户让第三方应用访问该用户在某一web服务上存储的私密的资源（如照片，视频，联系人列表），而无需将用户名和密码提供给第三方应用。

OAuth允许用户提供一个令牌，而不是用户名和密码来访问他们存放在特定服务提供者的数据。每一个令牌授权一个特定的第三方系统（例如，视频编辑网站)在特定的时段（例如，接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。这样，OAuth让用户可以授权第三方网站访问他们存储在另外服务提供者的某些特定信息，而非所有内容  
 下面是OAuth2.0的流程：  


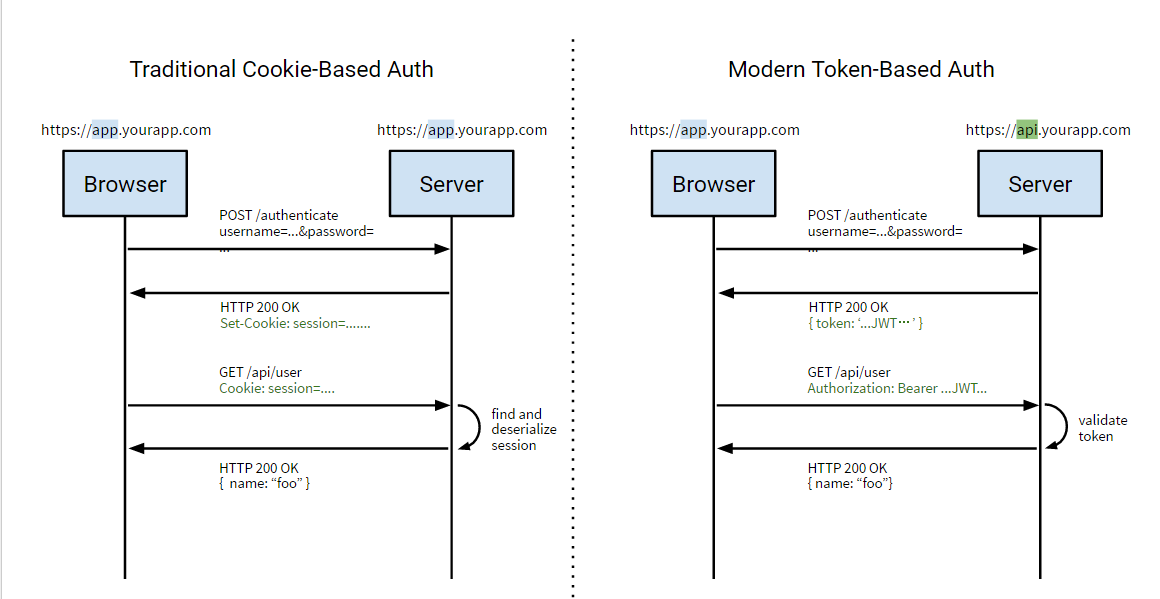
这种基于OAuth的认证机制适用于个人消费者类的互联网产品，如社交类APP等应用，但是不太适合拥有自有认证权限管理的企业应用；

Cookie Auth

Cookie认证机制就是为一次请求认证在服务端创建一个Session对象，同时在客户端的浏览器端创建了一个Cookie对象；通过客户端带上来Cookie对象来与服务器端的session对象匹配来实现状态管理的。默认的，当我们关闭浏览器的时候，cookie会被删除。但可以通过修改cookie 的expire time使cookie在一定时间内有效；

就是我们之前网站常用的session模式。下图中的传统模式也是这种模式。

Token Auth



Token机制相对于Cookie机制又有什么好处呢？

支持跨域访问: Cookie是不允许垮域访问的，这一点对Token机制是不存在的，前提是传输的用户认证信息通过HTTP头传输.

无状态(也称：服务端可扩展行):Token机制在服务端不需要存储session信息，因为Token 自身包含了所有登录用户的信息，只需要在客户端的cookie或本地介质存储状态信息.

更适用CDN: 可以通过内容分发网络请求你服务端的所有资料（如：javascript，HTML,图片等），而你的服务端只要提供API即可.

去耦: 不需要绑定到一个特定的身份验证方案。Token可以在任何地方生成，只要在你的API被调用的时候，你可以进行Token生成调用即可.

更适用于移动应用: 当你的客户端是一个原生平台（iOS, Android，Windows 8等）时，Cookie是不被支持的（你需要通过Cookie容器进行处理），这时采用Token认证机制就会简单得多。

CSRF:因为不再依赖于Cookie，所以你就不需要考虑对CSRF（跨站请求伪造）的防范。

性能: 一次网络往返时间（通过数据库查询session信息）总比做一次HMACSHA256计算 的Token验证和解析要费时得多.

不需要为登录页面做特殊处理: 如果你使用Protractor 做功能测试的时候，不再需要为登录页面做特殊处理.

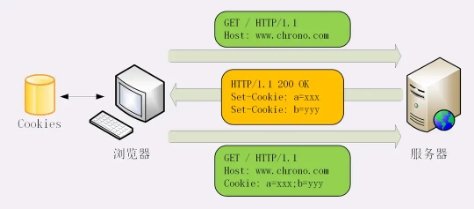
基于标准化:你的API可以采用标准化的 JSON Web Token (JWT). 这个标准已经存在多个后端库（.NET, Ruby, Java,Python, PHP）和多家公司的支持（如：Firebase,Google, Microsoft）.

cookie

浏览器第一次访问服务端时，服务器此时肯定不知道他的身份，所以创建一个独特的身份标识数据，格式为key=value，放入到Set-Cookie字段里，随着响应报文发给浏览器。

浏览器看到有Set-Cookie字段以后就知道这是服务器给的身份标识，于是就保存起来，下次请求时会自动将此key=value值放入到Cookie字段中发给服务端。

服务端收到请求报文后，发现Cookie字段中有值，就能根据此值识别用户的身份然后提供个性化的服务。



session

Session是存储在Tomcat的容器中，所以如果后端机器是多台的话，因此多个机器间是无法共享Session的，此时可以使用Spring提供的分布式Session的解决方案，是将Session放在了Redis中。

token

HTTP 是一种没有状态的协议，也就是它并不知道是谁是访问应用。这里我们把用户看成是客户端，客户端使用用户名还有密码通过了身份验证，不过下回这个客户端再发送请求时候，还得再验证一下。

解决的方法就是，当用户请求登录的时候，如果没有问题，我们在服务端生成一条记录，这个记录里可以说明一下登录的用户是谁，然后把这条记录的 ID 号发送给客户端，客户端收到以后把这个 ID 号存储在 Cookie 里，下次这个用户再向服务端发送请求的时候，可以带着这个 Cookie ，这样服务端会验证一个这个 Cookie 里的信息，看看能不能在服务端这里找到对应的记录，如果可以，说明用户已经通过了身份验证，就把用户请求的数据返回给客户端。

上面说的就是 Session，我们需要在服务端存储为登录的用户生成的 Session ，这些 Session 可能会存储在内存，磁盘，或者数据库里。我们可能需要在服务端定期的去清理过期的 Session 。

比起传统的身份验证方法，Token 扩展性更强，也更安全点，非常适合用在 Web 应用或者移动应用上。Token 的中文有人翻译成 “令牌”，我觉得挺好，意思就是，你拿着这个令牌，才能过一些关卡。

使用基于 Token 的身份验证方法，在服务端不需要存储用户的登录记录。大概的流程是这样的：

客户端使用用户名跟密码请求登录

服务端收到请求，去验证用户名与密码

验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端

客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里

客户端每次向服务端请求资源的时候需要带着服务端签发的 Token

服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据

JWT

实施 Token 验证的方法挺多的，还有一些标准方法，比如 JWT，读作：*jot* ，表示：JSON Web Tokens 。JWT 标准的 Token 有三个部分：

header（头部）

payload（数据，又称载荷）

signature（签名）

中间用点分隔开，并且都会使用 Base64 编码，所以真正的 Token 看起来像这样：

eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJuaW5naGFvLm5ldCIsImV4cCI6IjE0Mzg5NTU0NDUiLCJuYW1lIjoid2FuZ2hhbyIsImFkbWluIjp0cnVlfQ.SwyHTEx\_RQppr97g4J5lKXtabJecpejuef8AqKYMAJc

小知识：Base64是一种基于64个可打印字符来表示二进制数据的表示方法。由于2的6次方等于64，所以每6个比特为一个单元，对应某个可打印字符。三个字节有24个比特，对应于4个Base64单元，即3个字节需要用4个可打印字符来表示。JDK 中提供了非常方便的 **BASE64Encoder** 和 **BASE64Decoder**，用它们可以非常方便的完成基于 BASE64 的编码和解码

json

### 复杂json处理

JObject bodyContent = JObject.Parse(list[0].CONTENT); //JObject.Parse("{\"srfa\": [{\"id\": \"查勘日期\",\"name\": \"查勘日期\",}]}");

JToken record = bodyContent["srfa"];

List<Model.VarModel> record = JsonConvert.DeserializeObject<List<Model.VarModel>>(bodyContent["srfa"].ToString());

<https://www.cnblogs.com/mdzz-1992/p/7116549.html>

if (bodyContent.Property("纬度") != null)

if (bodyContent.Property("data") != null)

{

JToken datas = bodyContent["data"];

foreach (JObject data in datas)

{

foreach (JProperty jp in data.Properties())

{

string jp\_name = jp.Name;

JToken jp\_value = jp.Value;

var m = jp.Value.ToString();

//绑定公共变量

if (jp\_name.Equals("planDesignList"))

{

list\_design = JsonConvert.DeserializeObject<List<PlanDesignModel>>(jp.Value.ToString());

}

}

}

通过上面的var m = jp.Value.ToString();就可以解决带{的问题。

dll引用

如类库中System.Web中没有HttpContext类，可能是dll版本的问题，原来是在framework 2.0的项目版本下添加的，现在重新条件一次System.Web引用，发现就有了。

路径的含义

"./"：代表目前所在的目录。

" . ./"代表上一层目录。

"/"：代表根目录。

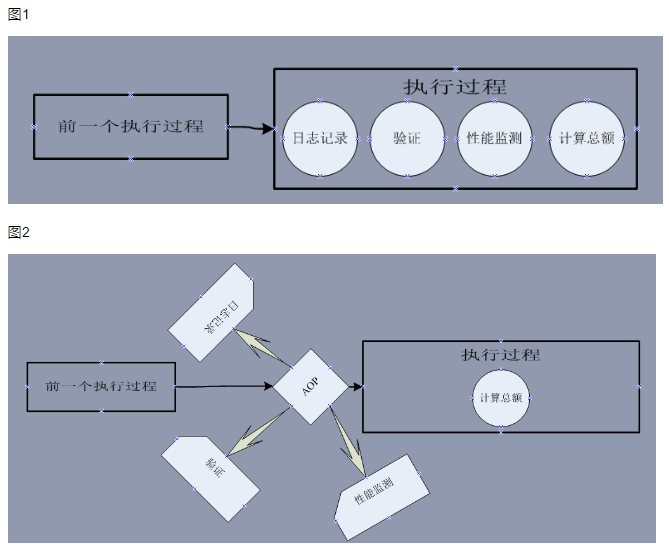
AOP

AOP的作用就是横切关注点，然后将分离后的关注点已面的形式来呈现，这是概念性的说法，举个列子来说明吧：

比如说有个处理过程是计算提交订单中的货品总额，然后想在这个过程中执行之前记录一下数据或者是执行一些必要的操作。

比如说记录日志，然后是选择记录日志的方式，是选择存本地文件还是存库，又或者是选择了存本地文件后选择数据的存储介质（XML格式、文本格式、加密格式、序列化格式等等）。

这只是其中的一个点，比如说还有验证等等其它一些方面的关注点。



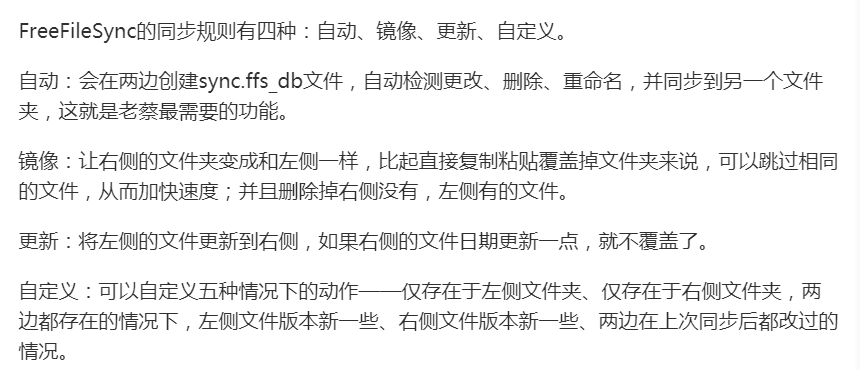
从图1、图2中我们就可以看出AOP的目的，就是将日志记录、验证、性能监测这些关注点从一个执行过程中分离了出来，让彼此不再有关系以及和计算总额的关系。在此可以把分离出来的关注点封装，已“面”的形式展现出来，这样的情况下也使得这些“面”可以在其它地方复用。

文件同步

可以用同步软件FreeFileSync：

<https://jingyan.baidu.com/article/eae07827aeaeb91fec54852b.html>

<http://www.360doc.com/content/16/0415/07/7863900_550736102.shtml>



修改共享文件夹的登录凭据

<https://www.cnblogs.com/xiaodai0/p/9513754.html>

计算机端口相关知识

### netstats命令

使用netstats命令，查看是否我们绑定端口（listenport）已经在侦听中，运行：

**NETSTAT.EXE -antp tcp|findstr LISTENING|findstr 22**

如果没有看到有输出条目，证明端口不侦听，自然也就无法代理了。

### 端口转发

有时对方的数据库只提供一台IP作为白名单。这时就需要端口转发。将对方数据库的端口转发为白名单的端口，如172.16.241.145可以访问172.16.0.72上的oracle，只需要在172.16.241.145上做端口转发。本地只需要访问172.16.241.145上的1521，就可以连接172.16.0.72上的1521。

<https://www.freebuf.com/articles/system/176889.html>

端口转发命令，在cmd里执行：

netsh interface portproxy add v4tov4 listenport=15210 listenaddress=0.0.0.0 connectport=15210 connectaddress=172.16.0.43

这时访问：

<http://172.16.0.43:15210/X5DataService/hr/getDepts.do>

就可以通过：

<http://172.16.241.145:15210/X5DataService/hr/getDepts.do> 的方式。

netsh interface portproxy add v4tov4 listenport=8081 listenaddress=0.0.0.0 connectport=8081 connectaddress=172.16.0.42

netsh interface portproxy add v4tov4 listenport=4018 listenaddress=0.0.0.0 connectport=4018 connectaddress=172.16.7.10

查看已经配置的端口转发：

netsh interface portproxy show v4tov4

删除端口映射：

netsh interface portproxy delete v4tov4 listenaddress=10.139.1.30 listenport=13016

### 端口转发没有作用

1、使用netstats命令，查看是否我们绑定端口（listenport）已经在侦听中，运行：NETSTAT.EXE -antp tcp|findstr LISTENING|findstr 22

22是转发的端口号。

2、如果没有看到有输出条目，证明端口不侦听，自然也就无法代理了，这时需要检查我们的IP HELPE服务是否开启。在控制面板->管理工具->服务，打开后，查找IP Help是否开启。该服务主要用于IPV6转换技术提供隧道连接。IP HELPE服务开启后就可以端口转发了。

.NET转JAVA没那么难<https://www.cnblogs.com/rabbityi/p/7079554.html>

.NET C#到Java没那么难，都是面向对象的语言，而且语法还是相似的，先对比一下开发环境，再到Servlet，再到MVC，都是一样一样的，只是JAVA的配制项比较多而已，只要配好一个，后面都是copy，简直so easy，而且用myeclipse可以省一些配制

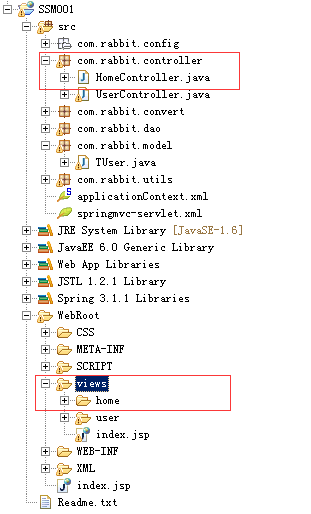
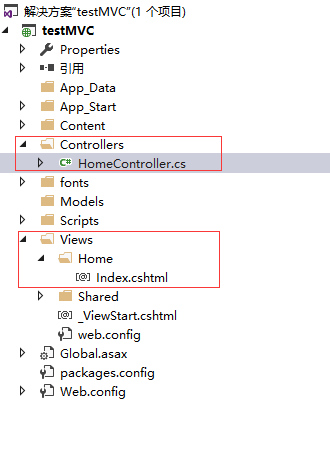
Java与.NET的区别不在它们的本身，最大的区别在于他们背后的力量，相信大家都能感觉得到，我就不多说。

.NET转JAVA没那么难-MVC篇

（1）项目结构

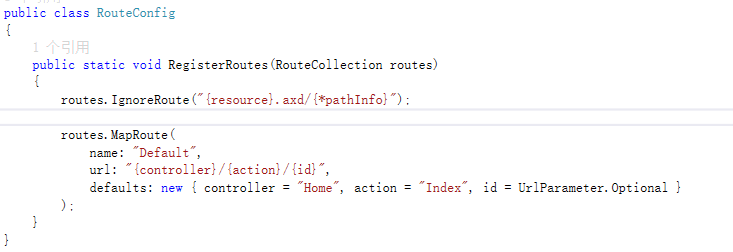
左边是用VS2015新建的MVC项目，右边是通过MyEclipse2016新建的SpringMVC项目，结构基本一致。

Java中的源码都放在src目录，jsp文件都在views目录里，也可以放在WEB-INF目录里。



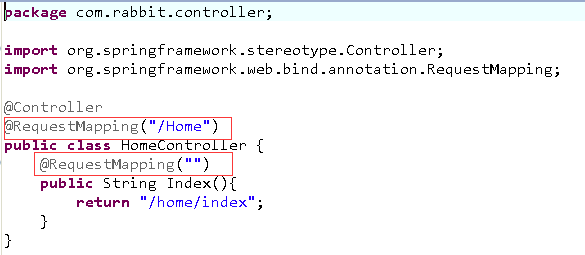
（2）路由比较

NET的MVC里面，有一个专门注册路由的地方，博客园都是MVC的高手，这里就不再啰嗦了。所有的路由规则都统一放在一起管理，直接了当，一目了然。



Java Spring的Route规则是通过注解（NET的特性）的方式，写在Controller中的，也可以用配制文件来配制。（SpringMVC也可以把所有的路由规则写在配制文件里统一管理，也比较清楚）

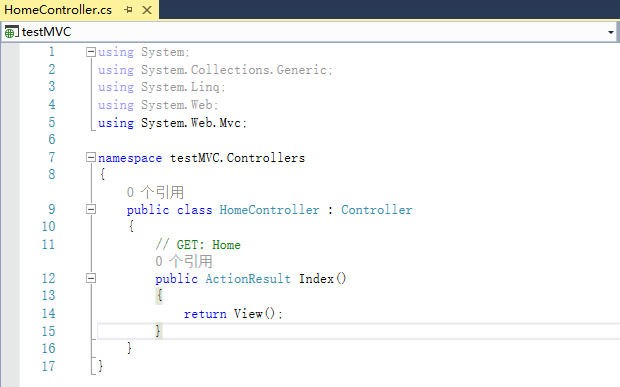
注意：RequestMapping就是Spring的路由，Spring在初始化的时候，会扫描指定位置下面所有的类，如果有@Controller的注解，就将其记录为Controller,然后会解析@RequestMapping，构建类似于RouteTable这样的东西。



（3）Controller比较

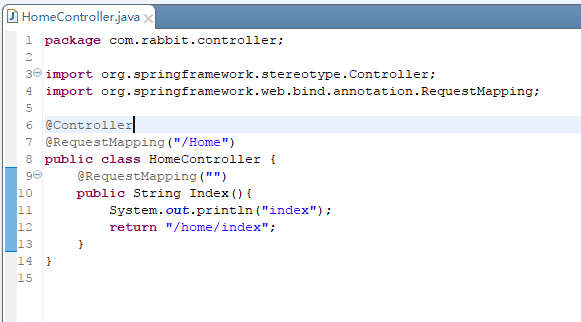
上边是.NET的Controller的代码，下边是Java的Controller的代码，基本一致。

.NET用的是约定大于配制（也可以用.NET的特性），而NET的话，约定了所有的Controller都放在指定的目录下面(经“冲杀”同学提醒只要继承Controller就可以了，放在哪个目录都可以，不过建议还是按约定放在指定的位置)，都继承于Controller父类/接口。



@Controller就是Java Spring的Controller，Spring在初始化的时候，会扫描指定位置下面所有的类，如果有@Controller的注解，就将其记录为Controller

当然，编码习惯好的Java开发者，也会把所有Controller放在同一个目录/Package的，例如我把Controller都放在com.rabbit.controller包。



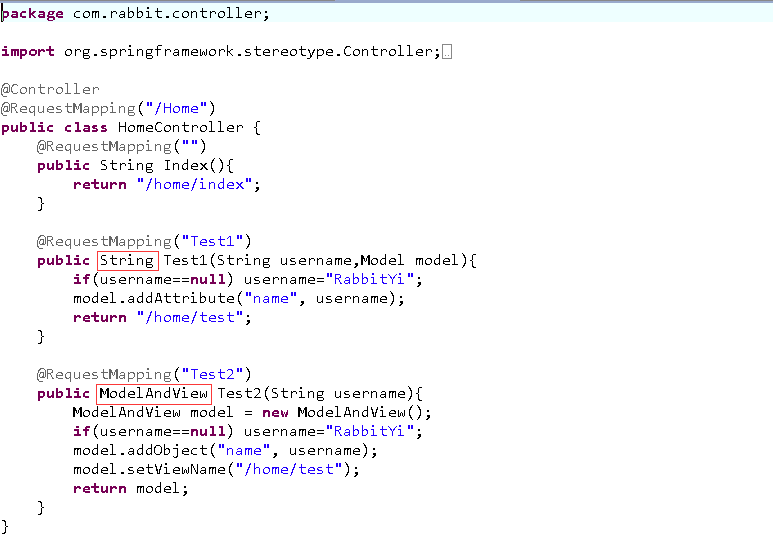
（4）view比较

1).NET MVC的View用的是cshtml文件，用的Razor引擎表达式 ，还有aspx视图引擎；Java的View用的是jsp文件，用的是 JSTL标签，EL表达式

2).NET MVC的view一般就两种，一种是使用原生的aspx写法，一种是razor写法，但这仅仅是写法上的区别，还是一套的实现。而在Spring MVC中就不同了，view的实现却有很多的选择，我们可以使用jsp，可以使用velocity，也可以使用更为强大的freemarker，即便是使用jsp做为view的实现，也可以有很多的写法，比如使用jstl、el或直接写jsp等等。

3).NET MVC与Spring MVC各有所长，比如在view的layout布局上，.NET已经很好的集成了，但Spring MVC做这个还要import或include，如果想做到更好，还需要第三方的layout jar。但是Spring MVC因为它出生就是Spring，所以与生俱来就可以对IOC、AOP的无缝整合，.NET需要用Autofac这种第三方框架实现IOC，AOP .Net MVC已经整合进去（就是各种Filter）

看看Spring MVC的返回视图是不是也和.NET MVC一样一样的呢

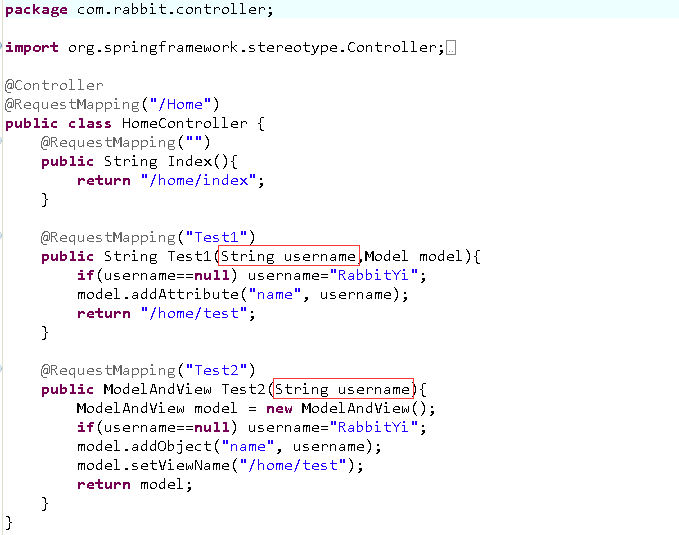


（5）Action参数绑定和返回JSON

参数绑定这块，两者的处理都差不多，如果一个数据类型可以和字符进行相互转换，那么系统就会自动帮你做绑定。如果不行的话，自己写一个类型和字符的转换类即可。

1).如下参数username可以通过get,post传递过来，Action都可以正常接收，只要名字一致即可，名字不一致时可以用注解实现。

2).如下参数传Model也是可以的，只要名字一致就可以自动绑定。



返回JSON这块两者的处理都差不多，都是自动把对象序列化后返回

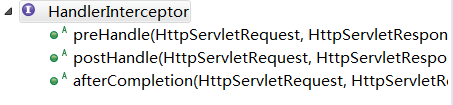
1).如下使用ResponseBody注解，直接返回POJO对象即可自动序列化成JSON，这里需要引入Jackson的包。



（6）Filter过滤器和Interceptor拦截器

1).Action过滤器[HandlerInterceptor](http://www.cnblogs.com/bigdataZJ/p/springmvc6.html" \t "_blank)和 .NET IActionFilter

那什么是拦截，又为什么要拦截。对于Spring MVC来说，拦截器主要的工作对象就是用户的请求，拦截下来之后，我们可以在拦截的各个阶段进行自定义处理。常见的比如可以做权限验证，登录系统后，系统获取到你的请求然后分析下你的用户权限，是Administrator还是User还是Guest。





2).全局错误过滤器 Spring [ControllerAdvice(ExceptionHandler)](http://blog.csdn.net/chenaschen/article/details/51291566" \t "_blank) 和 .NET IExceptionFilter。

.NET转JAVA没那么难-MVC篇

（1）项目结构

# 二十九、操作数据库

连接字符串

mySQL:

Database=kjgb\_data;Data Source=127.0.0.1;Port=3356;User Id=root;Password=kjgb2017;charset=utf8;

操作二进制文件

mySQL:

private void btn存储文件\_Click(object sender, EventArgs e)

{

if (openFileDialog1.ShowDialog() == DialogResult.Cancel)

{

return;

}

string mmFileName = openFileDialog1.FileName;

byte[] bytes = null;

bytes = File.ReadAllBytes(mmFileName);

using (MySqlConnection conn = new MySqlConnection(ConfigurationSettings.AppSettings["kjgbConnectionString"].ToString()))

{

using (MySqlCommand cmd = new MySqlCommand())

{

cmd.CommandText = "insert into Files (ID,File) value(@ID,@ParaBlob)";

cmd.Parameters.Add("@ID",MySqlDbType.VarChar);

cmd.Parameters.Add("@ParaBlob", MySqlDbType.LongBlob);

cmd.Parameters[0].Value = Guid.NewGuid().ToString();

cmd.Parameters[1].Value = bytes;

cmd.Connection = conn;

conn.Open();

int i = cmd.ExecuteNonQuery();

conn.Close();

}

}

}

private void btn读取文件\_Click(object sender, EventArgs e)

{

DataTable mmDT = \_DA.GetDataTable("select \* from Files");

if (mmDT.Rows.Count > 0)

{

byte[] bytes = (byte[])mmDT.Rows[0]["File"];

FileStream fs=new FileStream(AppDomain.CurrentDomain.BaseDirectory+mmDT.Rows[0]["ID"].ToString()+".mdb",FileMode.CreateNew);

BinaryWriter bw = new BinaryWriter(fs);

bw.Write(bytes, 0, bytes.Length);

bw.Close();

fs.Close();

}

}