...

[核心概念 2](#_Toc482695676)

[第1章 asp.net 简介 2](#_Toc482695677)

[第2章 Visual Studio 3](#_Toc482695678)

[第3章 Web窗体 5](#_Toc482695679)

[第4章服务器控件 9](#_Toc482695680)

[第5章 Asp.Net应用程序 14](#_Toc482695681)

[第6章状态管理 18](#_Toc482695682)

[数据访问 23](#_Toc482695683)

[第7章 Ado.net基础 23](#_Toc482695684)

[第8章数据组件和DataSet 27](#_Toc482695685)

[第9章数据绑定 32](#_Toc482695686)

[第10章富数据控件 37](#_Toc482695687)

[第11章缓存和异步界面 43](#_Toc482695688)

[第12章文件和流 49](#_Toc482695689)

[第13章 Linq 55](#_Toc482695690)

[第14章 XML 58](#_Toc482695691)

[构建Asp.net网站 63](#_Toc482695692)

[第15章用户控件 63](#_Toc482695693)

[第16章主题和母版页 65](#_Toc482695694)

[第17章网站导航 70](#_Toc482695695)

[第18章网站部署 76](#_Toc482695696)

[安全 82](#_Toc482695697)

[第19章 Asp.Net安全模型 82](#_Toc482695698)

[第20章表单验证 84](#_Toc482695699)

[第21章成员资格 84](#_Toc482695700)

[第22章 Windows 验证 86](#_Toc482695701)

[第23章授权和角色 87](#_Toc482695702)

[第24章用户配置 88](#_Toc482695703)

[第25章加密 89](#_Toc482695704)

[第26章自定义成员资格提供程序 91](#_Toc482695705)

[高级用户界面 91](#_Toc482695706)

[第27章自定义服务器控件 91](#_Toc482695707)

[第28章设计时支持 100](#_Toc482695708)

[第29章动态图形和GDI+ 104](#_Toc482695709)

[第30章使用Web部件页面的门户 105](#_Toc482695710)

[客户端程序设计 106](#_Toc482695711)

[第31章 JavaScript 和 Ajax 技术 106](#_Toc482695712)

[第32章 Asp.Net Ajax 108](#_Toc482695713)

[第33章 SilverLight 109](#_Toc482695714)

[技术积累 111](#_Toc482695715)

[第34章正则表达式 111](#_Toc482695716)

[.NET扩展知识学习 112](#_Toc482695717)

[35 反射学习 112](#_Toc482695718)

[36 WCF 113](#_Toc482695719)

[37 Silverlight 113](#_Toc482695720)

[38 Linq 113](#_Toc482695721)

[第三方框架学习 113](#_Toc482695722)

[39 AOP 113](#_Toc482695723)

[40 IOC 113](#_Toc482695724)

[41 Spring.Net 113](#_Toc482695725)

[42 Nhibernate 113](#_Toc482695726)

[43 ASP.NET MVC 113](#_Toc482695727)

[44 Castle 113](#_Toc482695728)

[45 SOA 113](#_Toc482695729)

[C#扩展知识学习 113](#_Toc482695730)

[46委托与事件 113](#_Toc482695731)

[47 异步编程 120](#_Toc482695732)

[48 C# DateTime格式化 121](#_Toc482695733)

# 核心概念

必须彻底了解的细节：

1.web页面和web应用程序的生命期、Asp.Net清除处理管道以及Asp.Net配置模型

### 第1章asp.net 简介

html（超文本协议）：实现一些简单的语言规范，几乎可以在任何计算机平台上显示丰富的文本

xml（可扩展标记语言）：制定一套规则，构建平台无关的数据格式，使得不同的应用程序通过xml来彼此交换数据

##### 1.1 Web开发技术的演进

1.1.1 早期的web开发

由服务端调用执行独立的小的应用程序。CGI（Command Gateway Interface，命令网关接口）

有服务端资源解释的脚本Asp和ColdFusion

1.1.2 传统asp的不足

意大利面条似的代码：

降低性能：混合html标签和脚本的编程模式会使应用程序的性能降低很多(ASP ISAPI Internet Server Appliction Programming Interface)扩展处理的时候，脚本引擎必须开关多次来处理相关请求

网页大小容易失控，难于调试

脚本语言：variant数据是一种弱类型，这种弱类型的数据单元需要大量内存，其数据类型只有到了运行时才知道，这会导致比强类型变量性能更差

1.1.3asp.net

1.Asp.net提供一个完全的面向对象的编程模式，包括事件驱动和基于控件的架构，鼓励代码封装和代码复用

2.Asp.net支持多语言开发

3.Asp.net致力于高性能。其页面和组件都是按需编译，不是每次使用时都被解释，还包含精良的数据访问模型和数据缓存

4.Asp.net增强的状态机制、有效的数据绑定、动态绘图、功能更全的安全模型

##### 1.2Asp.Net的7大要点

1.2.1 要点1：Asp.net同.net Framework集成在一起

1.2.2 要点2：Asp.net是预编译执行的，而不是解释执行的

1.2.3 要点3：Asp.net具备多语言支持

1.2.4 要点4：Asp.net运行在公共语言运行库内

自动内存管理和垃圾回收，类型安全，可扩展的元数据、结构化的错误处理、多线程

1.2.5 要点5：Asp.net 是面向对象的

1.2.6 要点6：Asp.net是跨设备和跨浏览器的

1.2.7 要点7：Asp.net 易于部署和易于配置

##### 1.3 Asp.net 3.5:故事在继续

1.3.1 Asp.net 2.0

丰富的控件、母板页、主题、安全和成员资格、数据源控件、web部件、配置文件

1.3.2 Asp.net 3.5

Linq，Asp.net AJAX（异步JS和xml）

1.3.3 Silverlight

让各种操作系统上的各类浏览器运行真正的Asp.net代码(适合创建小程序，显然不适合需要服务代码的程序)

##### 1.4 总结

浅显的讲述了Asp.net以及.net framework中所提供的特性和附加内容

### 第2章Visual Studio

##### 2.1 .net开发模型

2.1.1编译器

将所有代码=>IL

VB=>vbc.exe C#=>csc.exe Jscript=>jsc.exe J#=>vjc.exe

2.1.2 Visual Studio集成开发环境

集成的web开发服务器，多语言开发，更少的代码，直观的编码风格，更短的开发周期，调试

2.1.3 网站和Web项目

基于项目的开发（web application）无项目文件（web site）

2.1.4 创建无项目文件的网站

.net版本，模版，位置，语言，文件系统，本地IIS，ftp站点，远程web服务器

2.1.5多目标

2.1.6设计网页

##### 2.2 Visual Studio 集成开发环境

2.2.1 解决方案资源管理器

可视化的文档管理系统

2.2.2 文档窗口

使用不同的设计器来编辑各种类型的文本

2.2.3 工具箱

提供各种控件

2.2.4 错误列表和任务列表

2.2.5服务器资源管理器

##### 2.3 代码编辑器

2.3.1 添加程序集引用

2.3.2 智能感知和大纲显示

##### 2.4 编码模型

1.内联代码 2.代码隐藏（差别在于后者的页面类不再是隐士的而是被包含所有的页面方法）

2.4.1 代码隐藏文件如何与页面连接

Page指示符指定页面所采用的语言，并且告诉Asp.net在哪里可以找到关联的代码文本（除非你做website，用的是内联代码，所有代码都保存在同一个文件里）

Inherits指示一个已经编译的类的名称(要是使用Inherits不得不每次总要预编译你的类)，该特性指明正在使用的类

CodeFile将你的.aspx文件链接到源代码文件上

2.4.2控件标签如何与页面变量连接

.net类库中的Page类定义基本的功能，允许网页存放其他控件以html行使显现，提供对复杂asp对象的访问，如Request,Response,Session

代码隐藏类从Page类继承以便于获得Asp.net网页基本功能的集合

编译类时，Asp.net会把一些额外的代码合并到你的类中(部分类)

Asp.net编译器创建另外一个类来代表实际的.aspx页面，此类继承自自定义的代码隐藏类（含有一些合并的额外代码）以\_aspx命名（包含初始页面及其控件所需的代码以及一些给出最终所呈现的html的所需代码）

2.4.3 事件如何与事件处理程序连接

手工键入(委托编程)

双击控件

从属性窗口选择事件

连接事件处理程序方法：

<asp:Button id=”cmdOk” OnClick=”cmdOk\_Click runnat=”server”>

cmdOk+=cmdOk.Click

##### 2.5 web项目

比website开发更严格，更灵活的文件管理自定义部署过程，在迁移场景中更适用

2.5.1 创建web项目

编译：在运行他时是由VS编译的，web页面类被编译进和web项目同名的程序集里位于bin目录

代码隐藏：

Page指令：它不是使用CodeFile特性来指定源代码文件，而是使用CodeBehind特性(原因是VS代替Asp.net执行了编译，Asp.net检查CodeFile特性，Vs使用CodeBehind特性)

程序集引用。在website中，所有的引用都记录在web.config里，这样Asp.net在编译时解析引用时能够使用他们。webAppliactaion里是保存在项目文件里，VS编译代码的时候能够引用他们。除了System.Core.dll 和 System.Web.Extensions.dll的引用，他们包含.net 3.5的特性，这些引用被定义在web.config文件里，因为他们包含指定新的配置所需要的类

2.5.2 迁移旧版Visual Studio 创建的网站

注解：website里没有包含控件声明的代码文件在应用程序第一次执行并编译的时候在幕后生成

##### 2.6 Visual Studio调试

2.6.1 单步调试

2.6.2 变量监视

2.6.3 高级断点

##### 2.7 Visual Studio 宏（强大的宏和外接程序框架）

##### 2.8 Web Development Helper

##### 2.9 总结

### 第3章 Web窗体

##### 3.1 页面处理

Web应用程序和传统富客户端程序之间的差别：

Web应用程序在服务器上执行(利用回送的技术)

Web应用程序是无状态的(持久化机制：视图状态)

3.1.1 html表单

所有服务器端的编程框架都是在原始窗体数据上添加了一层抽象，他们分析数据，并一种更为有用的方式显现该数据(允许一个瘦对象层获取窗体控件的值)

string firstName = Request.Form[“FirstName”];

客户端将网页回送到Asp.Net的时候，它抽取并填充Form集合(为了与Asp兼容)，然后配置相应的控件对象 string firstName = txtFirstName .Text;

这样的代码具有类型安全的好处(例如如果是复选框得到是false或者true，而不是html语法里的on)，这样就避免接触到html语法

注解：所有的Asp.net中的控件都是放置在一个单一的<form>标签中，<form>标签的runnat=”Server”时，允许标签运行在服务器。Asp.net不能创建包含多个服务器端的<form>标签的Web窗体，但是跨页发送的技术来创建发送到另一个页面的页面

3.1.2 动态用户界面

书写方便，格式清楚，隐藏底层html的细节的方式，使你的代码不需要紧密耦合到html标准，跨越浏览器

3.1.3 Asp.net 事件模型

1.第一次运行时，Asp.net窗体页面和控件对象，执行初始化代码，页面被呈现为html发送给客户端，

页面对象在服务器内存中被释放

2.客户端事件被触发，引起回送

3.Asp.net截获返回的页面并且重建页面对象，还原对象在页面被发送到客户端时的最后的状态

4.Asp.net响应事件

5.将处理后的页面转换为html发回客户端。页面从内存中释放(假如有新的事件，重复2-4的步骤)

注解：既然html 是完全无状态的，那么Asp.net提供的所有状态都是重建的，事件驱动模型只是一种模拟

3.1.4 自动回送

AutoPostBack为false时可以优化性能，为true时Asp.net使用客户端的JavaScript来连接客户端和服务器的代码。

当AutoPostBack=true时，Asp.net在html页面上添加一个名为\_doPostBack() 的JavaScript方法，当方法被调用时，他触发回送，把窗体所有信息回送到web服务器(Asp.net自动生成\_doPostBack()方法，这段代码将会随着页面中AutoPostBack控件数目的增加而变长，因为必须为每个控件设置事件数据)

任何AutoPostBack=true的控件都可以通过onClick或者onChange事件来连接\_doPostBack()方法(Asp.net自动将客户端的JavaScript变为服务器的Asp.net事件，利用\_doPostBack方法作为媒介)

提示：Asp.net包括两种控制模式，基本的html服务器控件和完整功能的web控件，只有web控件才提供回送

3.1.5 视图状态

优点：每次请求之后服务器资源即被释放，不会造成服务器负担过重而瘫痪

缺点：存储页面中，造成页面过大，客户端不仅要接受一个更大尺寸的页面并且还要将其在下一个请求里发送回服务器，加重了客户端的负担，有GridView大量数据量的控件时，后果更严重。可以将EnableViewState设置为false来假禁用视图状态，但是不得不在每次回送时初始化控件

注解：即使EnableViewState=false，也只是假禁用视图状态，控件也回保持少量的对于维持正常功能所必须的视图状态信息，这种高级权限的视图状态被称为控件状态，不可被禁用

Asp.net不仅对页面和控件属性使用视图状态，对于成员变量或者其他可能用到的数据，它并不会执行相同的动作

注解：web窗体是在每个客户端和服务器端的来回上重建的，它在内存中持久化和保存的时间不会超过它呈现一个简单的请求的时间

3.1.6 Xhtml兼容性

##### 3.2 Web 窗体处理阶段

页面框架初始化=>用户代码初始化=>验证=>事件处理=>清除

3.2.1 页面框架初始化

IsPostBack=false，创建页面 IsPostBack=true 反序列化视图状态信息并应用到控件上

3.2.2 用户代码初始化

Page.Load()事件被触发

视图状态是自动维持的，IsPostBack=false时，必须从动态的数据源抓取数据 IsPostBack= false时，Asp.net从视图状态恢复控件属性

视图状态保存了每个变化了的属性，在Page.Load()初始化控件被记为一次变化，所以任何控件的值的变化被持久化到视图状态里，为了简化视图状态和保持尽量小的页面，要避免在代码里初始化控件，在控件标签里设置属性就不会持久化到视图状态。如果在代码中初始化控件更加简单，考虑设置EnableViewState=false，可禁用控件的视图状态，并在每次Page.Load()事件发生时初始化控件，而不管当前请求是不是一个回送

3.2.3 验证

检查页面的有效性(Page.IsValid)

3.2.4 事件处理

立即反映事件：包括单击按钮，图片，链接，调用JavaScript方法\_doPostBack()来触发一次回发

变化事件：改变控件的选择，文本框变化。当AutoPostBack=true时，事件立即发生，否则将在页面下次返回时发生

事件驱动模型：

优点：提供了更高层次的抽象，让你的代码与维持状态的样本区分开来

缺点：很容易忘记事件模型其实只是一种模拟，让你走出并不能保持正确的假想(如期望在成员变量中保存信息)或者做出可行性不佳的设计决定(如在视图状态中保存大量信息)

3.2.5 自动数据绑定

Page.PreRender

3.2.6 清除

Page.Unload 释放任意非托管的资源

3.2.7 页面流示例

页面处理程序使用设计器代码隐藏区域的代理显示的关联。因为设计器代码被认为是类的一部分(分部类的原因)，它可以关联任何方法，包括私有方法。控件事件处理程序使用不同的机制(控件标签)连接子一起，它们在处理的稍后阶段被绑定，即在.aspx文件中的标记和代码隐藏类合并后。Asp.net通过从代码隐藏类派生新类来创建这个合并的类

3.2.8 Web窗体处理阶段的所有事件发生的顺序

Page.Init=>Page.Load=>TextBox.TextChanged=>Button.Click=>Page.PreRender=>Page.Unload

##### 3.3作为控件容器的页面

Web表单首先呈现自身，然后通知页面中的所有控件呈现自身

第一次创建页面时：检查.aspx文件，对于其中找到的每个控件标签，他创建并配置一个控件对象，并把这

个控件作为子控件加入页面。Page.Controls集合找到所有控件

3.3.1 显示控件树

Asp.net对整个页面使用控件对象模型，包括哪些与服务器端内容并不对应的元素。(如果页面上有一

个服务器控件，Asp.net将会创建两个LiteralControl对象，分别代表这个控件之前和之后的静态内容。可能会出现多个LiteralControl对象，这取决于页面上放置了多少静态内容以及如何把它们和其他控件分隔)

Asp.net分层呈现页面。它只会呈现最顶层的控件。如果这些控件包含其他控件，它们则会提供自己的

Controls属性。这个属性提供对他们子控件的访问(没有对可使用的嵌套控件的层数的限制)

3.3.2 页头

标题：html页面的标题

样式表：IStyleSheet对象的CreateStyleRule()和RegisteStyle()方法动态创建新的样式规则

控件：利用这个集合和HtmlMeta控件类通过编程来添加或者删除元素据标签

3.3.3 动态控件的创建

动态创建的控件总是被呈现该父容器控件的最底层，并且只存在下一次回发之前。虽然视图状态

在Page.Load()之前重建，如果在Pge.Load()中动态创建控件，动态空间视图状态的好处。Asp.net会在Page.Load()事件处理程序结束以后为它应用任意的视图状态信息，这个过程是自动的

动态创建的控件添加事件。使用委托代码附加一个事件处理程序。必须在Page.Load()中执行这一操作.所有特定控件的事件都是在Page.Load()之后触发的。如果之后执行，事件处理程序将在事件发生之后附加到事件上，你将不能再对他作任何响应

newButton.Click+=dynamicButton\_Click();

##### 3.4 Page类

页面本事也是作为控件对象被实例化的

3.4.1 Session、Application和Cache

Session：System.Web.SessionState.HttpSessionState的实例

用于保存需要在网页请求间持久化的任意的类型的用户特定数据键/值形式访问

用于保存用户名、用户ID、购物车之类的信息或者其他用户不再访问网站页面时被丢弃的各种信息

Application：System.Web.HttpApplicationState的实例

键/值对形式访问。这个数据都是整个应用程序的全局数据

Cache：System.Web.Caching.Cache的实例

本质上是键/值对形式，但是可以为集合中每个项目设定特定的过期策略和依赖关系

也保存全局数据，但是它提供更具伸缩性的存储机制，因为Asp.net可以在服务器内存紧张时自

动删除对象

3.4.2 Request：System.Web.HttpRequest的实例。

代表引起页面被加载的http请求的值和属性。它包含所有的URL参数以及其他所有的由客户端发送的信息。可以通过Request发现用户使用的浏览器或者检查其cookie

3.4.3 Response：System.Web.HttpResponse的实例

代表web服务器对客户端请求的响应

Cookie特性和Respon.Redirect(URL)方法：

Server.Transfer(URL):缺点：重定向发生在服务器，所以原始的URL地址还保存在客户端浏览器

里。另外该方法的URL不能是非Asp.net页面、应用程序的页面、其他web服务器上的页面

优点：比Redirect()方法速度快，而且是最快。

提示：转向其他页面的另外一种方法：跨页发送。可以创建一个把自身发送到其他页面的页面，通过 它可以有效的传送所有的视图状态信息和任意的控件内容

HttpRespon类还包括一些不能和Asp.net服务器控件模型一起使用的模型。但在创建自定义http处理程序或返回非html页面类型的内容时可能要用到他们

Redirect(),Write(),OutPutStream()

3.4.4 Server：System.Web.HttpServerUtility的实例。

Transfer(),HtmlDecode(),HtmlEncode(),UrlDecode(),UrlEncode(),MapPath()

3.4.5 User：

User对象代表为web服务器提供请求的用户信息，它允许你检测用户的角色成员资格

3.4.6 Trace：System.Web.TraceContext的实例

启用跟踪：Trace.IsEnabled=true;web.config里启用跟踪

3.4.7 访问其他类中的Http上下文

获取一个不是从Page类派生的其他类的信息时：获取所有的http上下文信息。System.Web.HttpContext类的Current的静态属性,返回一个代表当前请求和响应的所有信息的HttpContext类的实例。它提供相同的作为属性提供的所有内建Asp.net对象

##### 3.5 总结

### 第4章服务器控件

本质上服务器控件就是.net framework中用来表示web窗体中可见元素的类

##### 4.1 服务器控件的类型

Html服务器控件

Web控件

富控件：生成大量的html标记，可以生成客户端的JavaScript来创建用户界面

验证控件：允许你方便的按照几个标准或用户定义的规则去验证关联的输入控件

数据控件：显示大量数据的复杂的网格和列表，支持高级的模板、编辑、排序和分页的特性

导航控件：显示站点地图

登陆控件：支持表单验证，表单验证是一个通过数据库验证用户并跟踪他们状态的Asp.net模型

Web部件控件：支持web部件，web部件是一个用来构建组件化、高可配置性的web门户的Asp.net模型

Asp.net AJAX 控件：让你在web页面里不编写客户端代码就是用AJAX技术

Asp.net 移动控件：支持移动客户端

服务器控件的层次：

所有的服务器控件都继承自：System.Web.UI命名控件中的Control基类。无论是使用html服务器控

件和web控件还是创建自定义控件，都是如此，如所有web窗体都要继承的Page类

ClientID(客户端上控件唯一的标示符)，Controls(页面上子控件的集合)，EnableViewState(视图状态)，ID(服务器上控件标示符)，Page(返回父页面对象的引用)，Parent(控件父类的引用)，Visible(是否被呈现)

DataBind()，FindControl()，HasControls()，Render(基于控件的当前状态编写该控件html的输出)

##### 4.2 Html服务器控件

4.2.1 HtmlControl类

所有的html服务器控件都派生自该类

Attributes(可以在控件标签中访问或者添加特性，使用这个集合来添加没有被指定为公开属性的特

性) Disabled(是否可用) Style(返回应用于控件的Css特性集合) TagName(控件的标签名，如img)

4.2.2 HtmlContainerControl类

包含含开始标签和结束标签的html标签都可以包含其他html内容或者控件

<innerHtml>(返回或设置开始标签和结束标签内的html文本，所有字符保持原样，意味着你可以嵌入

html css样式)<innerText>(返回文本，html css样式无效，原样输出)

4.2.3 HtmlInputControl 类

Html输入控件允许用户交互，包括图形小部件、复选框、文本框、按钮、和下拉框。这些控件和<input>

标签一起被创建，type特性指出输入控件的类型代码(把文本框的当前文本显示在警告消息框中)

Type(获得HtmlInputControl的类型)，Value(获取或者设置与输入空间相关联的值)

4.2.4 Html服务器控件类型

Html服务器控件对应它们底层html标签的特性(htmlHead和htmlTitle)

4.2.5 设置Style 特性和其他属性

要配置样式，可以使用style集合来添加css样式特性。想要设置没有被任何属性公开的其他特性，需要使用Attributes集合。下例使用Attributes 集合把简单的JavaScript与客户端控件的onFocus事件相关联

Protectd void Page\_Load(object sender,System.EventArgs e)

{

//只在页面第一次被请求时执行初始化

//此后，在视图状态中寻找该信息

If(!IsPostBack)

{

//设置Style特性来配置外观

Text1.Style[“font-size”]=”20px”;

Text1.Style[“color”]=”red”;

//使用一个稍微不同但等效的语法来设置Style特性

Text1.Style.Add(“background-color”,”lightyellow”);

//设置默认文本

Text1.Value=”<Enter e-mail address here>”;

//设置其他非标准特性

Text1.Attributes[“onfocus”]=”alert(Text1.value)”;

}

}

<input id=”Text1” type=”text” style=”font-size:20px;color:red;background-color:lightyellow;” value=”<Enter e-mail address here>” onfocus=”alert(Text1.value)” />

4.2.6 编程创建服务器控件

Protected void Page\_Load(object sender,System.EventArgs e)

{

//创建一个新的HtmlTable对象

HtmlTable table1 = new HtmlTable();

//设置表的格式化相关的属性

table1.Border=1;

table1.CellPadding= 3 ;

table1.CellSpacing = 3;

table1.BorderColor=”red”;

//开始向表中添加内容

HtmlTableRow row;

HtmlTableCell cell;

for(int i=0;i<=5;i++)

{

//创建一个新行并设置其背景颜色

row = new HtmlTableRow();

row.BgColor(i%2==0?”lightyellow”:lightcyan);

for(int j=0;j<=4;j++)

{

//创建一个单元格并设置其文本

cell = new HtmlTableCell();

cell.InnerHtml=”Row:”+i.toString()+”<br /Cell:>”+j.toString();

//将该单元格添加到当前行

row.Cells.Add(cell);

}

//将行添加到表

table1.Rows.Add(row);

}

//将表添加到页面

this.Controls.Add(table1);

}

4.2.7 处理服务器端事件

ServerClick:只是简单的单击事件

ServerChange:当改变文本或者改变成员属性时，在回送时，该事件才会发生

所有的变化事件都会在ServerClick事件触发回送之前被触发

##### 4.3 Web控件

4.3.1 WebControl基类

所有的Web控件都从WebControl类继承。WebControl类派生自Control类。因此WebControl的许多特性和方法是和HtmlControl一样的(Controls,FindControl(),Visible) 专属于WebControl的属性：AccessKey(返回和设置键盘快捷键，让用户快速导航到控件)、BackColor(背景色)、BorderColor(边框色)、BorderStyle(边框类型)、BorderWidth(边框厚度)、CssClass(相关控件的Css样式)、Enabled(是否启用)、Font(返回一个包含文本字体样式所有信息的对象)、ForeColor(前景色)、Height(高度)、TabIndex(用来控制控件标号的数字)、ToolTip(将鼠标移动到一个控件上显示的信息)、Width(宽度)

4.3.2 基本的Web控件类

4.3.3 单位

px或者%

//将300转换为一个表示像素的Unit对象，并为其赋值 pnl.Height = Unit.Pixel(300);

//把同一个单位赋给几个控件 Unit myUnit =new Unit(300,UnitType.Pixel); pnl.Heght = my

nit;pnl.Width=myUnit;

4.3.4 枚举

4.3.5 颜色

Using System.Drawing;

1. 使用ARGB(alpha,red,green,blue)
   1. int alpha = 255,red=0,green=25,blue=0;

ctrl.ForeColor = Color.FromArgb(alpha,red,green,blue);

1. 使用预定义的.Net颜色名称
   1. ctrl.ForeColor = Color.Crimson;
2. 使用一个Html颜色名字，可以使用ColorTranslator类描述一个字符型的值

a) ctrl = ForeColor = ColorTranslator.FromHtml(“Blue”);

4.3.6 字体

Font 引用了完整的FontInfo对象，它定义在System.Web.UI.WebControls命名空间里

4.3.7 焦点

每个Web控件都提供一个Focus()方法

4.3.8 默认按钮

<form id=”Form1” DefaultButton=”cmdSubmit” runnat=”server”></form>

4.3.9 可滚动面板

设置ScrollBars=Vertical 、Horizontal、 Both、Auto

4.3.10 处理Web控件事件

使用JavaScript来捕捉一个客户端的事件，并触发一个回发

##### 4.4 List控件

4.4.1 Selectable 列表控件

可以创建列表框、下拉列表和其他重复控件，这些控件可以绑定到一个数据源或编程填充

4.4.2 BulletedList 控件

显示为一个静态的bulleted或数字列表，是相当于一个<ul>无序列表</ul>或者<ol>有序列表</ ol>的集合

都派生自System.Web.UI.WebControls.ListControl类

##### 4.5 输入验证控件

4.5.1 验证控件

<asp:RequiredFieldValidator>不为空验证

<asp:RangeValidator>匹配验证

<asp:CompareValidator>比较验证true 或者false

<asp:RegularExpressionValidator>正则表达式验证

<asp:CustomValidator>普通验证

<asp:ValiationSummy>验证结果汇总

4.5.2 验证流程

4.5.3 BaseValidator类：System.Web.UI.WebControls

检验控件类ControlToValidate(表示需要验证的输入控件)，Display(怎样显示错误)，EnableClientScript(Boolean属性知名客户端校验是否会发生)，Enabled(允许用户启用或者禁用验证器)，ErrorMessage(显示在错误汇总中的错误信息)，Text(验证失败时，显示在验证器上的文本)，IsValid(当所有的验证控件有效时，Page.IsValid=true)，SetFocusOnError(浏览器自动将焦点切换到一个无效的控件上)，VadlidationGroup(验证组)，Validate(以此方法来重新验证控件，并依此更新Page.IsValid的值)

4.5.4 RequiredFieldValidator控件

确保不为空即可

<asp:TextBox id=”Name” runnat=”server” /><asp:RequiredFieldValidator runnat=”server” ControlToValidate=”Name” ErrorMessage=”Name is required” Display = “dynamic”>\*</ asp:RequiredFieldValidator>

4.5.5 RangerValidator 控件

验证输入值是否在预定义的范围内

<asp:TextBox id=”DayOff” /><asp:RangeValidator runnat=”server” display=”dynamic” ControlToVali date=”DayOff” Type=”Date” ErrorMessage=”DayOff is not within the valid interval” MinimumVa lue=”08/05/2008” MaxmumValue=”08/30/2008”></asp:RangeValidator >

4.5.6 CompareValidator控件

将值与值进行比较 ValueToCompare=Value或者 ValueToCompare=”ControlID” Operator(指定要比较的类型，Equal，NotEqual，LessThan，LessThanEqual，GreaterThan，GreaterThanEqual) DataTypeCheck(强制校验控件去检验输入需要的类型(通过Type指定)，而不执行任何额外的比较)

<asp:TextBoxID="txtAge"runat="server"/><asp:CompareValidatorID="ComTxtAge"runat="server"ErrorMessage="You must be at least 18 years old!"ControlToValidate="txtAge"ValueToCompare="18"Type="Integer"Operator="GreaterThanEqual"></asp:CompareValidator>

<asp:TextBoxID="txtPassword"runat="server"/><asp:TextBoxID="txtPassword2"runat="server"/>

<asp:Comparevalidatorrunat="server"errormessage="The passwords don't match"ControlToValidate="txtPassword2"ControlToCompare="txtPassword"Display="Dynamic"><imgsrc="a.gif"alt="The passwords don't match"/></asp:Comparevalidator>

前者用一个\*号指示错误，后者用一个图片

4.5.7 RegularExpressionValidator 控件

校验文本是否和定义在正则表达式中模式匹配

<asp:TextBoxID="txtEmail"runat="server"/>

<asp:RegularExpressionValidatorrunat="server"ControlToValidate="txtEmail"ValidationExpres

sion=".\*.{2,}\..{2,}"ErrorMessage="E-mail is not in a valid format!"Display="Dynamic">\*</asp:RegularExpressionValidator>

4.5.8 CustomValidator 控件

自定义的验证控件，该控件允许执行自定义的客户端和服务端验证子程序，就像其他验证控件一样，他们都接收两个参数，一个对验证器的引用和一个自定义的参数对象，这个对象提供一个Value属性，包括关联输入控件的当前值(你要验证的值)，以及一个IsValid属性，通过他来指定输入的值是否合法

该控件允许执行自定义的客户端和服务端验证子程序，

1. 创建客户端验证子程序

就像其他验证控件一样，他们都接收两个参数，一个对验证器的引用和一个自定义的参数对象，这个对象提供一个Value属性，包括关联输入控件的当前值(你要验证的值)，以及一个IsValid属性，通过他来指定输入的值是否合法

scripttype="text/javascript">

function EmpIDClientValidate(ctl, args) {

args.IsValid = (args.valueOf() % 5 == 0);

}

</script>

2.在后台创建等效验证的服务器端验证子程序

protectedvoid CusValTxtEmpID\_ServerValidate(object source, ServerValidateEventArgs args)

{

try

{

args.IsValid = (int.Parse(args.Value) % 5 == 0);

}

catch

{

args.IsValid=false;

}

}

3.前台定义

<asp:TextBoxID="TxtEmpID"runat="server"></asp:TextBox><asp:CustomValidatorID="CusValTxtEmpID"

runat="server"ErrorMessage="ID must be a multiple of 5"Display="Dynamic"ClientValidationFunction="EmpIDClientValidate"ControlToValidate="txtEmpID"

onservervalidate="CusValTxtEmpID\_ServerValidate"></asp:CustomValidator>

4.5.9 ValidationSummy 控件

不执行验证，在页面上显示错误的总结

<asp:ValidationSummaryID="Summary"ShowSummary="true"DisplayMode="BulletList"HeaderText="<b>Please review the following error:</b>"runat="server"/>

4.5.10 通过编程使用验证器

迭代当前的Validators集合

4.5.11 验证组

ValidationGroup属性设置为相同的就是同一组

使用Page.GetValidators()在给定的验证器组中获取控件，或者是Page.Validate(groupName)

##### 4.6 富控件

4.6.1 AdRotator 控件

4.6.2 Calendar控件

4.7 总结

### 第5章 Asp.Net应用程序

组成包括：文件，页面，处理程序，模块以及可以从web服务器虚拟目录（及其子目录）调用的可执行代码

Asp.net应用程序，用户不会直接运行Asp.net应用程序，他们使用浏览器(客户端)通过http管道请求URL，由Web服务器接收这个请求，web服务器没有单独应用程序的概念，它只是简单的把请求传送给Asp.net工作进程

##### 5.1 剖析Asp.net应用程序

5.1.1 应用程序域（Application domain）

应用程序域是由CLR强制的一个边界，保证一个应用程序不会影响到其他应用程序(或看到内存中的数据)

应用程序域模型：

1. 一个web应用程序内的所有网页和web服务共享同样的内存资源，如全局应用程序数据、每个用户的会话数据和缓存数据。这些信息不可以直接被其他Asp.net或Asp应用程序访问
2. 一个web应用程序内的所有网页和web服务共享同样的核心配置
3. 所有的web应用程序在不同阶段都会引发全局应用程序事件

网页：aspx

Web服务：asmx

代码隐藏文件：

配置文件：

global.ascx

其他组件

5.1.2 应用程序生命期

Asp.net使用延迟初始化(lazy initialization)技术创建应用程序，应用程序域只在应用程序中的页面或者服务第一次获得请求时才被创建

提示：使用静态的HttpRuntime.UnloadAddDomian()方法通过编程来关闭Web应用程序域。（应用程序在下一次请求时自动重启），当你在同一台服务器上运行了多个Web应用程序而其中某些很少用到，这很有用，此时，为保持应用程序域存在所附加的内存也许比加速对后续请求的响应更加重要

5.1.3 应用程序更新

可以随时更新web应用程序而不必重新启动服务器。当你修改虚拟目录里的文件时，Asp.net会执行 你修改web.config 配置文件时的同样动作，切换到一个新的应用程序域(能够随时更新应用程序 的任意部分而不会打断现有的请求，这不是CLR的特性，允许Asp.net无缝的转换到一个心的应

用程序域，CLR在程序及文件执行时总是锁定他们，为了突破这一锁定的限制，Asp.net并没有真正使用虚拟目录里的Asp.net文件，他是使用另一项叫做影子复制的技术，在编译过程中，他在某一目录下为你的文件创建一个副本，Asp.net从这个目录加载程序集，同时这些程序集被锁定)

5.1.4 应用程序目录结构

##### 5.2 global.asax 应用程序文件

允许你为HttpApplication(Application)类编写全局事件的事件处理程序

一个web应用程序的global.ascx文件不可以多于一个，且它必须位于应用程序的根目录而不可以位于任何

子目录内

注解：global.ascx文件使用全局应用程序类应该总是无状态的。这是因为应用程序对象在它们可用的时候会处于不同的请求，如果你在一个请求里设置了一个成员变量，它可能出现在另一个请求里，然而某一办法控制这种现象发生，也不能控制请求会获得哪一个应用程序对象的实例。要解决，除非是静态的，否则不要使用成员变量

5.2.1 应用程序事件

1.对每次请求都发生的事件，包括与请求相关的和响应相关的事件

2.只在某些特定情况下才发生的事件

1.Application\_BeginRequest().这个方法在每次请求开始时被调用

2.Application\_AuthenticateRequest()在执行验证前发生。这是创建你自己的验证逻辑的起点 3.Application\_AuthorizeRequest()发生子用户验证(识别)后，此时应该确定用户的权

限，可以使用此方法给用户赋予特殊的权限

4.Application\_ResolveRequestCache()常和输出缓存一起用，通过输出缓存，Web窗体呈现的Html

被重用，从而不必再次执行你的代码。不过这个事件处理程序仍会运行

5.此时，请求被转交给合适的处理程序，例如对于web窗体的请求，此时应该编译(如有必要)页面并把它实例化

6.Application\_AcquireRequestState()在为用户读取会话特定的信息并把这些信息填充到

Session集合前，该方法会被调用

7.Application\_PreRequestHandlerExecute()会在合适的http处理程序执行请求前被调用

8.此时，适当的处理程序执行请求。例如对窗体的请求，页面的事件处理程序代码就会被执行， 页面呈现为html

9.Application\_PostRequestHandlerExecute()请求被处理后调用

10.Application\_ReleaseRequestState()在会话特定信息即将从Session集合序列化时调用，这样它可

以用于下一个请求

11.Application\_UpdateRequestCache()在信息加入到输出缓存前被调用。例如，你启用了某个网页

的输出缓存，Asp.net在这个时候把它呈现为html插入到页面

12.Application\_EndRequest()请求结束时调用，在对象被释放和重新创建前，适合在这个适合清

理代码

并不是每次都发生的事件

1. Application\_Start()应用程序第一次启动和应用程序域创建时被调用。这个事件处理程序适合提供应用程序范围的初始化代码。例如：你可以加载和缓存在整个应用程序生命期内不会变化的数据，如导航树、静态的产品列表等
2. Session\_Start()会话开始调用，常用于初始化用户特定信息
3. Application\_Error()发生未处理错误
4. Session\_End()会话结束。会话在你的代码显示释放和一般20分钟没有请求后结束。这个方法常用来清理相关的数据。不过你正在用进程中的会话状态存储，可以只调用该方法(在InProc模式而不是State Server 和SqlServer模式)
5. Application\_End()在应用程序结束前被调用，
6. Applciation\_Disposed()通过它验证关键的资源是否被释放

5.2.2 演示应用程序事件

##### 5.3 Asp.net配置

不会被锁定，方便访问和复制，方便编辑和理解

5.3.1 machine.config文件

位置：c:\Windows\Microsoft.NET\Framework\v2.0.50727\Config

<processModel>配置Asp.net工作进程如何回收应用程序和用于执行工作进程的Windows账号

<machineKey>设置用于加密和创建数字签名的服务器特定密钥

5.3.2 web.config 文件

需要锁定特定的设置而不允许他们被覆盖时将会要用到它location节点下的allowOverride=false

对于某些提供web主机服务的公司特别有用，他们不希望某些特定的设置不要被修改，管理员可以修改web服务器上的machine.config文件并使用<location>元素锁定特定的节

提示：有两个办法可以锁定machine.config文件的设置。首先你可以省略<location>标签的path特性来锁定所有应用程序的设置，其次，你可以把path特性设置为相应web应用程序的名字来锁定特定的应用程序

5.3.3 <System.web>设置

在<system.web>不可以更改结构或添加自定义的元素</system.web>

5.3.4 通过编程读写配置节

AppSettings 自定义配置的访问，名称索引的集合

ConnectionStrings 名称索引的集合

OpenWebConfiguration() 返回一个Configuration对象，提供对特定web应用程序的配置信息的访问

OpenMachineConfiguration() 返回configuration，提供对web服务器定义的配置的累积信息的访问

5.3.5 网站管理工具(WAT)：帮助开发人员创建自定义配置的工具

5.3.6 扩展配置文件结构

扩展配置文件的三个步骤：

1. 确定要再配置文件中保存的信息以及如何把它们组织到元素和特性。最理想的是，对于每一个概念 上相关的的设置组都有一个元素，然后通过特性保存与元素相关的片段信息

2. 对于每个新元素，创建一个C#类封装它的信息，从对象抓取信息

3. 在配置文件里注册新节。此时必须使用<configSections>元素，<configSections>元素标识

每一个新元素并把它映射到相关的类

5.3.7 加密配置节

##### 5.4 .net组件

1. 在App\_Code()子目录里创建一个新的.cs文件

2. 在VisualStudio创建一个心的类库项目

虽然二者本质相同，但是对代码的管理却不同。当你希望在多个Web应用程序(甚至不同类型的应用程序)

里重用组件时尤为重要。前者，很容易做一些轻微的修改从而导致共享版本的混乱，后者更适合一个开发团队开发大型应用程序，这种情况下，你希望能够自由地分步完成并编译Web应用程序的不同部分。

提示：App\_Code子目录应该只被应用于和web应用程序紧密耦合的类，可重用的功能单元(如业务库，数据库组件，验证例程，加密应用)应该构建为单独的类库

5.4.1 创建组件

5.4.2 通过App\_Code 目录使用组件

注解：使用App\_Code目录，只能使用一种语言。这个限制来自于Asp.net执行动态编译的方式。从本质

上说，App\_Code目录中的所有类都被编译到同一个目录中，所以你不能混用C# 和VB。

5.4.3 通过Bin目录使用组件

##### 5.5 扩展http管道

应用程序的管道不仅仅限于对.aspx web表单的请求，它还可用于对web服务的请求，甚至可以创建自己的处理程序来处理自定义文本类型

5.5.1 http处理程序和http模块

对Asp.net应用程序的请求都由称为Http处理程序的特殊组件处理。http处理程序是Asp.net请求处理框架的骨架。所有的Http处理程序都在配置文件的<hhtpHandlers>节定义。核心的http处理程序在根web.config文件里。在<httpHandlers>节中<add>元素添加来注册新的处理程序

http模块通过处理应用程序事件参与对请求的处理，在很大程度上和global.ascx类似。一个给定的请求可以在多个Http模块间流动，但是由一个http处理程序结束。在<httpModules>节中使用Add和Remove来添加或者删除模块

Http处理程序和Http模块的一个优势是他们提供了一个可扩展的框架，允许你方便的插入自己的处理模块

5.5.2创建自定义的http处理程序

希望工作于Web表单模型之外的的低层来支持特殊格式的处理，可以实现自己的Http处理程序

创建一个Http处理程序，需要实现IHttpHandler接口。ProcessRequest()，Asp.net获得请求时调用该

方法。Http处理程序在这里处理所有的请求。IsReusable。前者完成任务后，Asp.net检查IsReusable

是否为真，若IsReusable=true，当前处理程序对象可以被另一个和当前请求类型相同的请求重用，否则，当前对象即被释放

5.5.3配置自定义的http处理程序

5.5.4不配置iis的http 注册处理程序

.ashx文件类型已经在IIS里注册，所以发布应用程序的时候不必执行任何IIS配置

.ashx文件通常用于为单个应用程序设计的简单一些的扩展，配置文件更灵活。例如：注册Http处理

程序处理所有给定扩展名的请求，而ashx文件只能服务于特定文件名的请求。

5.5.5创建高级的http处理程序

5.5.6为非html 内容创建http处理程序

<httpHandlers>

<add verb=”\*” path=” \*.gif” type=” ImageGuardHandler” />

<add vern=”\*” path=”\*.png” type=”ImaheGuardHandler”/>

</httpHandlers>

可以使用http处理程序呈现自定义图片，执行特殊的数据库查询或者返回二进制数据。这些实例扩展了Asp.net结构但是略过了Web页面模型，产生了更轻量、更高效的组件。还可以创建异步的处理程序，这在执行耗时但并不大量使用CPU任务提升了可伸缩性。示例：等待读取非常慢的网络资源。

http处理程序不能访问用户特定的会话状态，因为Http处理程序一般用于较低层任务，跳过需要序列化和读取会话状态信息的步骤可以微略提升性能。如果一定要访问会话状态信息，只需要实现IRequireSessionState(修改或者添加)或者IReadOnlySessionState(读取)

5.5.7创建自定义http模块

实现System.Web.IHttpModule接口的类。然后把自己的模块注册到<httpModules>节

当一个http模块被创建时，它注册自己接收特定的全局应用程序事件。模块使用Init()方法中的代码

来注册它的事件

Public class LogUserModule

{

public class Init(httpApplication httpApp)

{

//附加应用程序事件处理程序

httpApp.AuthenticateRequest+=new EventHandler(OnAuthentication);

}

private void OnAuthentication(object sender ,EventArgs e)

{

string name = httpContext.Current.User.Identity.Name;

EventLog log = new EventLog();

log.Source = “Log User Module”;

log.WriteEntry(name+“was authenticated”);

}

public void Disponse(){}

}

<httpModules>

<add name=”LogUserModule” type=”HttpExtensions.LogUserModule,HttpExtensions” />

</ httpModules>

##### 5.6 总结

### 第6章状态管理

Http是一种无状态协议：每次Web请求后，客户端和服务器断开，同时Asp.net引擎释放页面对象。这种架构保证了web应用程序能够同时相应数千个并发访问而不会导致服务器崩溃。缺点是你必须通过其他技术存储Web请求之间的信息并要在需要的时候使用它们。

##### 6.1 Asp.net 状态管理

可以编写自定义的状态管理代码并使用服务器端资源来存储信息，使用服务器端资源的缺点是会降低性能，

并损伤扩展性

##### 6.2 视图状态

视图状态是单个页面中保存信息的第一选择。Asp.net服务器控件也使用视图状态在回发间保存属性值

6.2.1 视图状态示例

publicvoid SaveAllText(ControlCollection controls,bool saveNested)

{

foreach(Control control in controls)

{

if(control isTextBox)

{

ViewState[control.ID] = ((TextBox)control).Text;

}

if((control.Controls!=null)&&saveNested)

{

SaveAllText(control.Controls, true);

}

}

}

publicvoid RestoreAllText(ControlCollection controls,bool saveNested)

{

foreach (Control control in controls)

{

if (control isTextBox)

{

if(ViewState[control.ID]!=null)

((TextBox)control).Text=(string)ViewState[control.ID];

}

if ((control.Controls != null) && saveNested)

{

RestoreAllText(control.Controls, true);

}

}

}

protectedvoid btnSave\_Click(object sender, EventArgs e)

{

SaveAllText(this.table1.Controls,true);

}

protectedvoid btnRestore\_Click(object sender, EventArgs e)

{

RestoreAllText(this.table1.Controls, true);

}

<tableid="table1"runat="server">

<tr><td>Description</td><td>Value</td></tr>

<tr><td><label>ID:</label></td><td><asp:TextBoxID="txtID"runat="server"></asp:TextBox></td></tr>

<tr><td><label>Name</label></td><td><asp:TextBoxID="txtName"runat="server"/></td></tr>

<tr><td><label>Age</label></td><td><asp:TextBoxID="txtAge"runat="server"/></td></tr>

<tr><td><label>Email</label></td><td><asp:TextBoxID="txtEmail"runat="server"/></td></tr>

<tr><td><label>Password</label></td><td><asp:TextBoxID="txtPassword"runat="server"/></td></tr>

<tr><td><asp:ButtonID="btnSave"Text="Save"runat="server"

onclick="btnSave\_Click"/></td><td><asp:ButtonID="btnRestore"

Text="Restore"runat="server"onclick="btnRestore\_Click"/></td></tr>

</table>

6.2.2 在视图状态中保存对象

在视图状态中保存项目时，Asp.net需要将它转换成比特流以便添加到页面的隐藏字段中(序列化)

一个已经序列化的类Customer

//在视图状态中存储一个Customer

Customer cust = new Customer(“Marsala”,”Simons”);

ViewState[“CurrentCustomer”]=cust;

//在视图状态中读取一个Customer

Customer cust;

Cust = (Customer)ViewState[“CurrentCustomer”];

6.2.3 保持成员变量

和控件属性不同，在页面类中添加的成员变量不会保存到视图状态中。但是可以用视图状态来打破这一限制？

1. 创建一个封装视图状态的访问

private Dicationary<string,string> ControlText

{

get

{

if(ViewState[“ControlText”]!=null)

{

Return (Directionary<string,string>)ViewState[“ControlText”];

}

else

{

return new Dictionary<string,string>();

}

}

set{ViewState[“ControlText”]=value;}

}

这个方式的优点是序列化与页面逻辑的其余部分相分离，使得数据存放在其他地方更容易

1. 在Page.PreRender事件中将所有的成员变量保存到视图状态中，并在Page.Load()事件重新获取变量的值。这种方式，其他所有的事件处理程序都可以正常使用成员变量
2. 视图状态信息过大会加大页面的输出，并因此延长页面的传送时间

6.2.4 视图状态评估

放弃视图状态而采用其他的状态管理方式：

1. 需要保存业务核心数据，不允许被用户篡改=》会话状态
2. 保存需要被多个页面使用的信息=》会话状态，cookie，查询字符串
3. 存储的信息量非常大，并且不希望影响页面传送时间=》会话状态或数据库

没有必要使用控件的视图状态

1. 控件从来不会变化，如仅有静态文字的按钮
2. 控件在每次回发中都被填充
3. 该控件时一个输入控件，且它只根据用户活动变化

提示：视图状态作用于所有的变化值而不仅仅是控件显示的文本。比如：要动态改变一个标签的颜色，即使不需要动态改变他的文字，仍然需要启用视图状态

6.2.5 视图状态安全

1．使用散列值保证视图状态信息不被篡改

2． 设置web.config文件或者machine.config中的<pages>元素的enableViewState

##### 6.3 在页面间传送信息

视图状态一个最大的限制是它必须和特定页面紧密绑定，不能跨页面。

6.3.1 查询字符串

优点：它是轻量级的，不会加重服务器负担

缺点：

1. 信息仅限于简单的字符串
2. 信息暴露
3. 服务器不能预知该查询字符串是否被修改
4. 长度受限制

6.3.2 跨页面发送

缺点是很容易导致创建的页面紧密耦合而难于改进和调试

##### 6.4 cookie

//创建cookie对象

HttpCookie cookie = new HttpCookie(“Preferences”);

//在其中设置一个值

Cookie[“LanguagePref”]=”English”;

//添加另一个值

Cookie[“Country”]=”US”;

//将其添加到当前的web响应中

Response.Cookies.Add(cookie);

//该cookie存一年

Cookie.Expires = DateTime.Now.AddYear(1);

//使用Request集合通过cookie的名字来读取cookie

HttpCookie cookie =Request.Cookie[“Preferences”];

//检查是否能找到一个具有该名字的cookie这种情况下，用户禁用cookie，cookie则不存在

string language;

if(cookie!=null)

{

Language = cookie[“language”];

}

//移除一个cookie 的唯一方法是用一个已经过期的cookie来替换它

HttpCookie cookie = new HttpCookie(“LanguagePref”);

Cookie.Expires = DateTime.Now.AddYear(-1);

Response.Cookies.Add(cookie);

Asp.net的会话状态(允许在服务器内存中临时保存特定用户的信息)和表单安全(它强怕用户必须通过登陆页面才能访问网站的一些特定区域)应用了cookie

##### 6.5 会话状态

在一个页面保存信息而在另外一个页面使用信息，它支持包括自定义数据类型的任意对象类型。会话状态是跨页浏览时保存用户购物车内容的理想场所。迫使Web服务器在服务器内存保存额外的信息

6.5.1 会话架构

注解：除非你确实使用会话状态存储信息，否则每次每次提出新的请求时，Asp.net都会产生一个新的

会话标示符(SessionID)。这样使性能微略提升，不适用SessionID，就不需要保存了。

Asp.net处理Http请求时，会经过一个包含不同模块的管道链，这些模块可以相应应用程序事件。链

上的一个模块是SessionStateModule，该模块负责产生SessionID，从内部提供程序中获取会话数据，并把数据绑定到请求的调用上下文中。当页面处理完毕时，该模块还得存储会话信息，但是不保存会话数据，会话数据持久化到称之为状态提供程序的外部组件中。状态提供程序是可以实现HttpSessionState接口的任意的类，也就是说新建一个.net组件，你就可以自定义如何处理会话状态

如何获得SessionID

1. 使用cookie：SessionID通过一个特殊的cookie(叫做ASP.NET\_SessionId)来传送。使用会话集合时，Asp.net自动产生这个cookie
2. 使用修改过的URL：SessionID通过一个经特殊修改(或嵌入)的URL来传送。这允许你使用会话状态创建应用程序，即使客户端不支持cookie

6.5.2 使用会话状态

//在会话状态中存储DataSet

Session[“ProductsDataSet”]=dsProduct;

//读取DataSet

dsProduct = (DataSet)Session[“ProdctsDataSet];

会话状态在这些情境下会丢失：

1. 用户关闭或者重启浏览器
2. 用户通过另外一个浏览器窗口访问同一页面。不同浏览器处理问题的方式不同
3. 会话超时，默认20分钟
4. 调用Session.Abandon()方法
5. 应用程序域重建时会话也会丢失

前两者会话其实还保存在内存中等待被回收

Count(会话集合中的项目数)，IsCookieless(指示当前会话是在会话集合中还是嵌套在URL中)，IsNewSession(指示会话是否与当前会话一起创建，如果当前会话中没有内容，Asp.net不会追踪或者创建会话cookie。相反每次请求时都会重新创建会话)，Mode(提供一个枚举，指示Asp.net如何保存会话状态信息)，SessionID(会话标识符)，StaticObjects(只是为了向后兼容ASP程序)，Timeout(超时)，Abandon(取消当前会话并释放内存，在退出页面时很有效，能确保服务器中的内存得到快速重用)，Clear(在不改变SessionID的情况下清空所有的会话项目)

6.5.3 配置会话状态

web.config文件中的<sessionState>元素下配置会话状态

1. Mode会话状态模式设置允许你配置用于在请求间保存会话状态信息的会话提供程序
   1. Off 禁用所有的会话状态管理
   2. InProc 传统ASP中保存会话状态的方式。最好的性能但是最差的持久性(对大多数小型网站有效，在web农场中它根本不能工作。)还会产生因为应用程序域不断重启而产生的会话碎片

6.5.4 会话状态安全

会话状态中的信息非常安全，因为他保存在服务器上。但是含有SessionID的cookie很容易被篡改，就是说一个恶意的用户可以偷走cookie然后在另一台计算机上继续使用

1. 使用自定义的会话模块来检查客户端IP地址的变化。唯一真正有效地方法是仅在使用SSL的网站上使用cookie，会话cookie被加密从而在其他计算机上不可用
2. 使用无cookie会话时总将regenerateExpirationId设置为真，这样攻击就不能使用过期的会话ID了，其次在新的用户登录前显示取消当前会话

##### 6.6 应用程序状态

设置一个全局计算器来跟踪某个操作被不同客户端执行的次数

protected void Page\_Load(object sender,EventArgs e)

{

Int count =0;

If(Application[“HitCounterForOrderPage”]!=null)

{

Count = (int)Application[“HitCounterForOrderPage”]

}

Count++;

Application[“HitCounterForOrderPage”]=count;

lblCounter.Text=count.toString();

}

##### 6.7 总结

# 数据访问

### 第7章 Ado.net基础

无Ado.net的数据访问

1. SqlDataSource控件: SqlDataSource控件允许你声明性地定义查询。可以把SqlDataSource链接到GridView之类的富数据控件
2. LINQ TO SQL : 和SqlDataSource一样，LINQ TO SQL不允许执行不能直接映射到查询和更新的数据库命令(如建表)。和SqlDataSource不同的是 LINQ TO SQL 只允许和SQL Server 一起工作和ADO.NET 是完全独立的
3. 用户配置：无需编写任何ADO.NET代码，允许用户特定的数据块保存到数据库里。局限性：不允许你控制保存数据的数据库表的结构
4. 都不能取代ADO.NET，不能达到手工编写得数据库代码那样的灵活性、可自定义性以及性能

##### 7.1 Ado.net架构

7.1.1 Ado.net数据提供程序

数据提供程序是一组用户访问特定数据库，执行SQL命令并获取值的ADO.NET类，本质上是应用程序和数据源之间的一座桥梁

1. Connection：使用它建立和数据源的连接
2. Command：使用它执行SQL 命令和存储过程
3. DataReader：他提供对查询结果的快速的只读和唯进访问方法
4. DataAdapter：可以用它执行两项任务：首先用它将从数据源获得的信息填充到DataSet(一系列和数据库断开的表和关系的集合)其次依照DataSet的修改更新数据库

7.1.2 Ado.net的标准化

7.1.3 Sql Server 2005

1. MARS(多个活动集)，该特性可以在同一时间内执行多个查询

2. 用户定义数据类型：可以使用.NET代码定义自定义类，然后直接将它的实例保存到数据库的

栏位中。可以节省从行中读取若干栏位的时间，应用程序也不再需要创建相应的对象模型

3. 管理存储过程：SQL Server 2005 可以宿主CLR，使得你可以使用纯C#代码在数据库中编写存储过程

4. SQL通知。通知让代码相应数据库变化所产生的事件。在Asp.net中常使用这个特性记录更新时使缓存中失效的数据

5. 快照隔离事务：提升并发访问的性能

7.1.4 基本Ado.net类

1. 基于连接的对象：数据提供对象Connection、Command、DataAdapter、DataReader

2. 基于内容的对象：这些对象其实就是数据的“包“。如DataSet、DataColumn、DataRow、

DataRelation

##### 7.2 Connection类：

任何操作前建立连接

7.2.1 连接字符串

connectionString=WebConfigurationManager.ConnectionStrings[“Northwind”].ConnectionString;

7.2.2 测试连接

7.2.3 连接池

因为建立连接所需要的微小消耗也会显著影响系统性能=》使用连接池。连接池保持已经打开的

数据库连接，这些在连接使用相同数据源的会话间共享，这样就省了不断创建和销毁的时间

7.2.4 连接统计

7.3 Command类和DataReader类：

Command执行所有类型的SQL语句，也可以创建、修改表，索引，约束

7.3.1 Command基础

//执行Sql语句

SqlCommand cmd = new SqlCommand(“Select \* from Employees”,conn);

//执行存储过程

SqlCommand cmd = new SqlCommand(“GetEmployees”,conn);

cmd.CommandType = CommandType.StoredProcedure;

ExcuteNonQuery()执行非select语句或者数据库定义命令(创建、修改、删除)表、索引、约束，返

回受影响的行数

ExecuteScalar()执行select查询，返回命令位于结果记录集第一行第一列的字段，常用来执行使用Count、Sum等计算单个值的聚合select语句

ExecuteReader()执行select语句，返回一个封装了只读，唯进游标的DataReader对象

7.3.2 DataReader类

提供了最快捷且毫无拖沓的数据访问，不过它缺乏非连接的DataSet所具有的排序功能

Read()有记录则返回真，否则为假。将游标前进到流的下一行

GetValue()返回当前行中指定序号的字段值。返回的数据类型是.NET中河数据源类型最相似的哪

一个。dr.GetValue(0)与dr[“First”]等效，后者更易读，但是效率却不高

GetValues()将当前行中的值保存到数组中。保存点栏位数取决于你传递的数组的大小。可以使用dr.FileCount属性确定行中栏位的个数，依此创建合适大小的数组

NextResult()读到下一个行集

Close()关闭dr。如果原命令执行一个带有输出参数的存储过程，该参数仅在dr关闭后才可见

7.3.3 ExecuteReader()方法和DataReader

空值检测：

int ? numberOfHires;

if(reader[“NumberofHires”]==DBNull.Value)

{

numberOfHires = null;

}

if(reader[“NumberOfHires”] .HasValue)

{

numberOfHires=(int?)reader[“NumberOfHires”];

numberOfHires+=1;

}

当CommandBehavior.CloseConnection.时，关闭DataReader的同时，DataReader会关闭关联的连接

当CommandBehavior.SingleRow当仅从一行读取数据时它可以提升查询的性能。用主键(ID)获取单行记录时可以用它来优化性能

当CommandBehavior.SequentialAccess来分批读取二进制字段的值，可以减少大型二进制文件对内存的需求

处理多个结果集

读取大量相关联的数据时特别有用，比如产品和产品类别共同组成的产品目录

String sql=”select \* from A;select \* from B;select \* from C;”

或者含有多个select 的StroedProcedure

开始DataReader通过Read()方法读取A表的全部记录后，就调用NextResult()访问B的记录。

注意：在访问A的记录前不要调用NextResul()

7.3.4 ExecuteScalar()方法

ExecuteScalar()返回一个对象。该对象是查询结果中第一行第一列的值，常用来返回单个字段的查询

7.3.5 ExecuteNonQuery()方法

返回int类型。增、删、改。操作成功则返回>0,否则是-1；

7.3.6 Sql注入式攻击

如何防止：

1. 使用TextBox.MaxLength属性防止用户输入过长的字符串
2. 使用Asp.net验证控件锁定错误的数据(如文本、空格、数值中的特殊字符)
3. 限制错误信息给出的提示。捕捉到异常时，只显示部分通用信息，而不是显示Exception.Message属性中的信息，这样暴露攻击点
4. 对输入文本进行处理，尤其是那些特殊字符
5. 使用参数化的查询命令或者是使用一个存储过程执行以防止SQL注入式攻击

7.3.7使用参数化查询

String sql =”select Orders.CustomerID,Orders.OrderID,Count(UnitPrice) As Items,”+”Sum(UnitPrice\*Quantity) As Total From Orders”+”Inner Join [Order Details].OrderID”+”Where Orders.CustomerID=@CustID”+”Group by Orders.OrderID,Orders.CustomerID”;

SqlCommand cmd = new SqlCommand(sql,conn);

cmd.Parameters.AddWithValue(“@CustID”,txtID.Text);

Post注入式攻击…但是Asp.net EnableEventVadlidation属性

7.3.8调用存储过程

Create Procedure InsertEmployee

@TitleOfCourtesy varchar(25),@LastName varchar(20),@FirstName varchar(10),@EmployeeID int Output As Insert into Employees(TitleOfCourtesy,LastName,FirstName,HireDate) values(@TitleOfCourtesy,@LastName,@FirstName,GetDate())

Set @EmployeeID = @@Identity

SqlCommand cmd = new SqlCommand(“InsertEmployee”,conn);

cmd.CommandType = CommandType.StoredProcedure

将存储过程中的参数加入到Command.Parameters中

cmd.Parameters.Add(new SqlParameter(“@TitleOfCourtesy”,SqlDbType.NVarChar,25));

cmd.Parameters[“@TitleOfCourtesy”].Value = title;

cmd.Parameters.Add(new SqlParameter(“@LastName”,SqlDbType.NVarChar,20));

cmd.Parameters[“@LastName”].Value = lastName;

cmd.Parameters.Add(new SqlParameter(“@FirstName”,SqlDbType.NVarChar,10));

cmd.Parameters[“@FirstName”].Value = firstName

最后一个输出参数

cmd.Parameters.Add(new SqlParameter(“@EmployeeID”,SqlDbType.Int,4));

cmd.Parameters[“@FirstName”].Direction= ParameterDirection.Output;

int num = cmd.ExecutenNonQuery();

int empID = (int)cmd.Parameters[“@EmployeeID”].Value;

如果你想向一个已存储的过程传递null值，你不能使用C#的null引用，它表示未初始化的引用，在字段中表示一个空内容必须将.Net常量DBNull.Value作为参数值来传递

添加参数时隐式指定数据类型

cmd.Parameters.AddWithValue(“@LastName”,lastName);=>根据提供的数据猜测数据类型(对输入参数有效，你不会为输出参数提供值)

##### 7.4 事务

7.4.1事务和Asp.net应用程序(代价由小到大)

1. 存储过程事务：这些事务完全在数据库中处理。存储过程事务提供最佳的性能，因为只需要往返一次数据库。缺点是：需要用SQL语句编写事务处理逻辑

2. 客户端(Ado.Net)的事务：这些事务由Asp.net Web 页面中的代码通过编程来控制。本质上，他们使用和存储过程事务一样的命令，但代码使用封装了这些细节的Ado.Net对象,缺点是：事务在开始和提交时需要额外往返数据库

3. COM+事务：支持分布式事务，需要跨越多个资源管理器时才是很好的选择

使用事务的实践原则

1. 事务尽量短
2. 不要在事务中间使用Select查询往返数据。一般而言，在事务开始前返回数据库。这样可以减少事务，锁定的数据的数目
3. 如果事务中确定要获取记录，仅获取确实需要的记录，可以避免锁定过多资源，从而尽量保证性能
4. 可能的情况下，在存储过程中使用事务而不是使用Ado.Net事务，这样，事务就可以被更快地启动和编译，因为数据库服务器不需要与客户端(Web应用程序)交互
5. 避免使用具有多个独立批处理任务的事务。把各个批处理任务作为单个事务
6. 尽量避免更新或影响大批量记录

7.4.2隔离级别

隔离级别决定了事务对其他事务影响的数据的敏感度。当表、行或某个范围内有共享锁时，其他事务就不可以修改相应的数据，但是多个用户可以使用共享锁并发读取数据。独占锁禁止多个事务同时更新或者读取数据

7.4.3保存点

保存点标记事务回滚的点

##### 7.5 提供程序无关的代码

7.5.1创建工厂

7.5.2用工厂创建对象

7.5.3应用程序无关代码的查询

##### 7.6 总结

### 第8章数据组件和DataSet

1. 如何创建数据组件类

2. 如何要为执行的任务添加独立的方法

3. DataSet与Ado.net特性允许你长时间和数据源断开后继续和数据打交道。DataS在导航、过滤、排序方面给你更多的灵活性

##### 8.1 构建数据访问组件

1. 尽快打开和关闭连接。不要在客户端请求间保持连接，客户端不知道怎么获得连接也不知道什么时候关闭连接

2. 实现错误处理。使用错误处理保证即使SQL命令产生异常连接也能关闭

3. 遵守无状态设计实践。方法通过参数接受所有需要的信息，通过返回值返回所得到的全部数据。多次调用设置属性值远比把所有信息作为参数来调用单个方法所使用的时间长

4. 不要让客户端使用了广泛开放的查询。每个查询应该明智地选择确实需要的列。使用where子句来限制返回的结果

8.1.1 数据包(封装数据的实体类)

将表作为类，将字段作为属性进行封装

8.1.2 存储过程

8.1.3 数据应用类

EmployeeDB类：封装所有数据访问代码以及数据库相关的细节。该类使用实例方法而不是静态方法，尽管该类不保存任何数据库的状态信息，但它确实用私有成员变量保存了数据库连接字符串。因为这是一个实例的类，连接字符串可以在每次创建类的时候而不是在每个方法调用的时候获得，这样可以使代码稍微清晰一些，运行也稍微快一些(避免了多次读取web.config)，不过好处微乎其微，你一样可以在数据库组件中方便的使用静态方法

并发策略：

1. 后到者胜出更新：总是提交更新，适合于数据冲突比较少，当只有一个人负责更新一组特定的数据时，就可以放心的用。Where通过记录的主键与需要更新的记录匹配
2. 完全匹配更新：Update Emp set … where empId=@EmpId and firstName = @FirstName;
3. 基于时间戳的更新：支持时间戳列，由数据源在记录有更新时自动更新的Update Emp set … where empId=@EmpId and TimeStamp = @TimeStamp;
4. 变化的值更新。在Update 中只更新变化了的值
5. 用事务实现封闭并发。封闭并发通过锁定使用中的记录防止并发冲突，此时其他试图访问相同数据的用户将不得不等待

8.1.4 测试数据库组件

##### 8.2非连接的数据

只有数据读取数据才会和数据源断开连接。代码需要追踪用户活动、保存信息并确定何时创建和执行新的命令：缺点并发问题

以下情境下。DataSet比DataReader更加使用：

1. 需要一个方便的包以便在组件间传递数据(如：在组建共享信息或者通过Web服务向客户分发信息)
2. 需要一个方便的文件格式将数据串行化后保存到磁盘上(DataSet包含一个内建的功能可以将自己保存到xml中)
3. 需要在大量的数据前后浏览。DataSet支持分页的列表控件每次只显示部分内容
4. 需要在不同的表间导航。DataSet可以保存所有这些表以及他们之间的关系。因此DataSet可以创建一个主-从页面而不需多次到数据库执行查询
5. 需要通过用户界面绑定数据。DataReader 是唯进的游标，所以不可能同时绑定到多个控件上。它还不支持像DataSet那样对数据进行自定义排序或者按条件过滤
6. 需要以XML方式操作数据
7. 需要通过Web服务提供批量更新。可以让一个客户端下载一个包含大量的DataTable，客户端做了多个修改后再向服务器提交。在这种背景下，Web服务可以通过一次操作就实现所有表的更新

8.2.1 Web 应用程序和DataSet

DataSet 在使用富窗体客户端的分布式应用程序中更有意义。DataSet工作后，仅在需要把更新提交到数据源的时候才重新连接系统。比那些保持基于游标连接的系统响应更多的并非用户，还可以在服务器上缓存数据以及在客户间共享连接池从而共享资源

一般而言：插入、删除、更新记录时不使用DataSet。DataSet可以把一批数据从数据组件带到网页上，支持绑定，可以在GridView之类的高级控件上显示信息。多数Web程序使用DataSet来检索信息，但用直接的命令执行更新

注解：Web服务是唯一可能通过DataSet执行批量更新的真实Web应用场景。这种情况下，富客户端程序以DataSet的形式下载数据，编辑后，再把DataSet回传从而提交更新

8.2.2 xml集成

DataSet提供了XML序列化功能

##### 8.3 DataSet类

注解：DataSet并不代表数据源中的所有信息，也不必和数据源中的表一一对应，只是你的自定义查询结果

DataSet.Tables[]=>DataTable=>DataColumns=>Columns(描述每个字段的名称和数据类型)

=>DataRows=>Rows(包含每天记录的真正数据)

DataRow对象=>DataTable里的每条记录

=>数据源取得的表的一条记录

##### 8.4 DataAdapter类：

提供程序相关的对象，是DataSet中表和数据源间的桥梁，包含查询和更新数据库的全部命令

1. Fill(执行SelectCommand中的查询后向DataSet添加一个DataTab。如果返回多个结果集，则将依次添加多个DataTable对象，还可以向现有的DataTable中添加数据)

2. FillSchema(执行SelectCommand中查询返回架构信息，它向DataSet中添加一个DataTable，该方法并不向DataTable中添加数据，相反，它只利用列名、数据类型、主键和唯一约束等信息配置DataTable)

3. Update(检查DataTable中所有变化并执行适当的InsertCommand、UpdateCommand、DeleteCommand操作为数据源执行批量更新)

8.4.1 填充DataSet

SqlDataAdapter da = new SqlDataAdapter(sql,conn);

DataSet ds = new DataSet();

Da.Fill(ds,”Employee”);

当需要在数据源上执行一系列快速连续的操作时这样很有效

注解：将DataSet和控件绑定时，视图状态中并没有保存数据对象。数据控件只保存足够当前显示的信息。如果需要在多次回发间和DataSet交互，就必须手工方式把DataSet保存到视图状态(这样会显著增加页面大小)或者会话缓存对象中

8.4.2 应用多个表和关系

两个表通用一个DataAdapter合理且很有效，不需要重新创建DataAdapter去更新数据库。不过避免既要用DataAdapter查询数据又要用它提交更新，这样应该为每个表创建对应的DataAdapter为对应的表提供适当的插入、更新、删除命令

1. DataRelation relat = new DataRelation(“p1:relationName”,”p2:PrimaryKeyOfColumnInParentTable”,”p3:ForeignKeyOfColumnInChildTable”);
2. ds.Relation.Add(relat);

调用DataRow.GetChildRows()方法访问相关产品的信息：比独立对象查找相关的信息简单，也远比传统基于连接来执行多次查询要简单

//为父记录(分类)读取子记录(产品)

DataRow[] childRows = row.GetChildRows(relat);

//循环遍历

Foreach(DataRow childRow in childRows)

注解：什么时候使用Join查询，什么时候使用DataRelation？判断标准是：是否要更新获得的数据；如果是，那么使用独立的表和DataRelation更加灵活。如果否，两者都可以用，不过Join查询效率更高，因为它只需在网络间往返一次，而DataRelation则需要往返两次以填充两个独立的表

引用完整性

1. 创建一种没有相关约束的关系

DataRelation relat = new DataRelation(“p1:relationName”,”p2:PrimaryKeyOfColumnInParentTable”,”p3:ForeignKeyOfColumnInChildTable”，false);

1. 在添加关系前将DataSet.EnforceConstraints=false,禁止所有类型的约束检查

8.4.3查找特定行

DataRow[] matcheRows = ds.Tables[“Products”].Select(“Discountined=0”);

缺点：Select不支持参数化查询，有SQL注入书攻击的漏洞，但是很有限，它无法访问真实的数

据源，也不能执行额外的命令

8.4.4在数据访问类里使用DataSet

某些方法比返回一个DataSet更方便一些

DataSet是若类型的，没有编译时的语法检查和智能提示

8.4.5数据绑定

GridView1.DataSource =ds;

//只能绑定单个表，必须显示指定表名：

GridView1.DataMember=”Employees”;或者GridView1.DataSource=ds.Tables[“Employees”];

##### 8.5 DataView类

DataView为DataTable定义一个视图，是DataTable中支持自定义过滤和排序设置的数据的一个表现。

8.5.1 使用DataView排序

1. 使用默认的DataView

Grid1.DataSource = ds.Tables[“Employees”];

2. 按照姓排序并绑定

DataView dv2 = new DataView(ds.Tables[“Employees”]);

Dv2.Sort=”LastName”;

Grid2.DataSource = dv2;

排序只需要将DataView.Sort属性设置为合法的排序表达式就可以

8.5.2使用DataView过滤

DataView dv2 = new DataView(ds.Tables[“Employees”]);

Dv2.RowFilter=”WhereInSQL”;

Grid2.DataSource = dv2;

8.5.3使用关系的高级过滤：基于关系来过滤行

1. 链接两表的表关系

DataRelation relat = new DataRelation(“CatProds”,ds.Tables[“Categories”].Columns[“CategoryID”],ds.Tables[“Products”].Columns[“CategoryID”]);

Ds.Relation.Add(relat);

1. 用一个聚合函数来处理记录中的数据

Max(Child(CatProds).UnitPrice)>50;

1. 作用于DataView

DataView dv = new DataView(ds.Tables[“Categories”]);

Dv.RowFilter =”Max(Child(CatProds).UnitPrice)>50”;

gridView.DataSource = dv;

8.5.4计算列

除了处理直接从数据源获取的字段外，还加入计算列。在读取和更新时计算列被忽略，只表示计算现有值的结果的集合

1. 定义两表关系

DataRelation relat = new DataRelation(“CatProds”,ds.Tables[“Categories”].Columns[“CategoryID”],ds.Tables[“Products”].Columns[“CategoryID”]);

Ds.Relation.Add(relat);

DataColumn count = new DataColumn(“Product(#)”,typeOf(int),”Count(Child(CatProds).CategoryID)”);

DataColumn max = new DataColumn(“Most Expensive Prodcut”,typeOf(decimal),”MAX(Child(CatProds).UnitPrice)”);

1. 添加列

Ds.Tables[“Categories”].Column.Add(count);

Ds.Tables[“Categories”].Column.Add(max);

1. 显示数据

Grid1.DataSource = ds.Tables[“Categories”];

Grid1.DataBind();

##### 8.6强类型的DataSet

强类型的DataSet由DataTable以及DataRow继承的特殊类组成

优点：比起普通DataSet

1. 数据集已经按照你将要获取的数据的表、列、以及数据类型预先初始化了。这样执行查询实际信息的时候会稍微快一些(数据提供程序首先获取最基础的架构信息，然后再填充DataSet)
2. 通过强类型的属性名称而不是基于字段查找的方式访问表和字段的值，不会出现运行时错误

8.6.1 自定义TableAdapter

8.6.2创建强类型的数据集

8.6.3深入强类型的数据集

8.6.4使用强类型的数据集

提示：因为继承，强类型的数据集其实是一个DataSet。可以无缝的把NorthwindDataSet转换为DataSet类并且使用通常的基于字符串的方式查找表和字段。不过这一过程不可逆

##### 8.7 总结

### 第9章数据绑定

##### 9.1 基本的数据绑定

数据绑定的关键特征是：他是声明性的而不是编程性的，数据绑定是在你的代码外随着.aspx页面的控件一起定义的

###### 9.1.1 单值绑定

<asp:TextBox ID=”TextBox1” runnat=”server” Text=’<%#GetFilePath%>’ />

Protected string GetFilePath(){return “après.gif”}

<%#%>:可以是属性的值，成员变量，函数的返回值(只要属性、成员变量或函数具有受保护的或

公有的可见性),可以使运行时的可计算的表达式，如对其他控件属性的引用，使用操作符和文本

的计算或者其他)

数据绑定表示并不直接设置它绑定的属性，只是定义了一个控件属性和其他一段信息之间的连接，

为了让页面计算表达式、执行合适的方法并赋给正确的值，必须显著调用DataBind()方法

###### 9.1.2其他类型的表达式

<asp:Literal runnat=”server” Text=”<% $Appsettings:appName%>”>

区别： 1. 定义不一样

2.<%$%>不需要调用DataBind(),总是在页面呈现时自动计算

3.不可以在页面上随意放置。必须放在空间标签内并由表达式的计算结果设置属性。

如果想像使用文本一样使用表达式，就必须将它包含在Literal标记中(Literal控件把它的文本完全未格式化的HTML输出)

特性：可以创建自己的表达式构造器并把它插入到整个框架中。开发需要嵌入到多个web应用程序的独立应用程序时最为有用

实际上：页面编译，计算代码的表达式也编译。需求：页面编译，表达式不编译，页面被请求时表达式重新计算=》表达式构造器需要重新生成一段完成适当任务的通用代码=》CodeDOM，代码文档对象模型，动态生成代码的模型每一个代码表达式构造器包含一个使用CodeDOM生成的表达式所需代码的GetCodeExpression()方法。也就是说要创建RandomNumberExpressionBuilder,就需要一个GetCodeExpression()方法

1. Public class RandomNumberExpressionBuilder:ExpressionBuilder{…}

Using System.Web.Compilation;Using System.CodeDOM;Using System.ComponentModel;

1. 创建一个执行所需任务的静态方法

Public static string GetRandomNumber(int lowerLimit,int upperLimit)

{

Random ran = new Random();

Int randValue = ran.Next(lowerLimit,upperLimit+1);

Retuen randValue.ToString();

}

1. 重写GetCodeExpression()方法，该方法由Asp.Net发现表达式和表达式构造器匹配的时候调用(编译页面的时候)

Public override CodeExpression GetCodeExpression(BoundPropertyEntry entry,object parsedData,ExpressionBuilderContext context)

{

//entry是不带前缀的数字字符串

If(!entry.Expression.Contains(“,”))

{

Throw new ArgumentException(“Must include two numbers separated by a comma.”);

}

Else

{

String[] numbers = entry.Expression.Split(‘,’);

If(numbers.Length!=2)

{

Throw new ArgumentException(“Only include two numbers”);

}

Else

{

Int lowerLimit,upperLimit;

If(Int32.TryParse(numbers[0],out lowerLimit)&&Int32.TryParse(numbers[1],out upperLimit))

{

//随机数每次都要变化，所以必须使用CodeDOM创建一段动态的代码。做法是创建一个调用GetRandomNumber()静态方法的CodeExpression

}

}

}

}

###### 9.1.3重复值绑定

可以让你一次将一个列表的信息绑定到控件上。可以使自定义集合(ArrayList或者HashTable)，也可以使行的集合(DataReader或者DataSet)

重复值绑定的列表控件

1. <select>标记呈现的控件，包括HtmlSelect、ListBox、DropDownList
2. CheckBoxList、RadioButtonList，他们的子项呈现为独立的复选框或者单选按钮
3. BulletedList，它创建一系列列表或者编号
4. 注解：
   1. 每一个支持重复值数据绑定的控件都有一个DataBind()方法。
   2. 调用Page.DataBind()方法的时候，页面对象会自动调用所含控件的DataBind()方法。
5. 可以作为列表控件数据源的数据类型
   1. 所有内存中的集合类，如：Collection、ArraryList、HashTable、Dictionary
   2. Ado.Net的DataReader对象，它提供基于连接的、唯进的、只读的数据库访问
   3. Ado.Net的DataView，它提供单个非连接的DataTable对象的视图
   4. 其他任意实现ICollection接口的自定义对象
6. DataTextField 只能接收一个字段的名字，不能使用数据绑定来连接多个字段作为数据源
7. 绑定富数据控件
   1. GridView：显示大型表数据的全能网格。支持选择、编辑、排序、分页
   2. DetailView：每次显示一条记录的理想控件，它显示为一个表，每行对应一个字段。支持编辑和可选的排序，允许在一系列记录见浏览
   3. FormView：和DetailView类似，每次显示一条记录，支持编辑和在一系列记录间移动的分页控制。和DetailView的差别是它是基于模板的。允许你在更为灵活的不居中合并字段而不必依赖表格
   4. Menu、TreeView、AdRotator

##### 9.2数据源控件

###### 数据源控件包括那些所有实现IDataSource接口的控件

* 1. SqlDataSource =>可以连接到任意拥有Ado.Net数据提供程序的数据源。SqlServer、Oracle、OleDB、ODBC
  2. ObjectDataSource=>可以连接到自定义的数据访问对象。大型web应用程序倾向使用的数据源控件
  3. AccessDataSource=>Access,最好用于小型网站
  4. XmlDataSource=>连接到xml文件
  5. SiteMapDataSource=>站点导航信息的web.sitemap文件

###### 数据绑定页面的生命期

* 1. 数据源控件可以完成两类任务
     1. 他们可以从数据源中读取数据并为关联的控件提供数据
     2. 在关联的控件编辑数据后它们可以更新数据源
  2. 数据绑定任务顺序
     1. 页面对象被创建(基于.aspx)文件
     2. 页面生命周期开始,Page.Init事件和Page.Load事件发生
     3. 产生所有其他控件
     4. 数据源控件执行所有更新。如果行更新，则触发Updating事件和Updated事件。如果行新增则触发Inserting和Inserted事件，删除则是Deleting 和Deleted事件
     5. Page.PreRender事件发生
     6. 数据源控件执行所有查询并将获得的数据插入到关联的控件，此时触发Selecting事件和Selected事件
     7. 页面被呈现和释放
     8. 注意：该过程在每次页面请求都会重复一遍，这意味着数据源控件在每次页面请求时都会请求数据库

##### 9.3 SqlDataSource

###### 9.3.1 选择记录

* + - 1. 数据绑定探源

DataSourceMode.DataSet=>几乎总会更好一些，支持基于DataSet的高级设置排序、过滤及缓存，这些在DataReader模式下禁用

* + - 1. DataSourceMode.DataReader=>在超大型表格中使用，DataReader只在内存中保存记录一段时间-只要足够将记录信息复制到关联的控件上就可以
      2. 同一个数据源绑定到两个控件上时，查询会执行2次(在页面上进行手工绑定，可以把同一个对象绑定到两个不同的控件上，查询只需执行一次)，显然SqlDataSource带来了额外负荷
         1. 解决办法：

使用缓存，SqlDataSource天生支持缓存

大部分时候不需要将数据源绑定到多个控件，因为富数据控件用用在灵活的不居中呈现多个信息段的能力，这些控件只需要绑定一次

* + - 1. 数据绑定在页面处理结束处理，就是在页面处理之前,也就是说Page.Load=>所有控件的事件=>Page.PreRender=>数据绑定（每次回发时都会发生=>数据绑定完成后重写Page.OnPreRenderComplet()方法，该方法在PreRender状态之后且在视图状态被序列化以及真实的HTML呈现之前发生）

###### 9.3.2 参数化命令

1. 防止Sql注入式攻击
2. 可以通过响应Selecting、Updating、Deleting、Inserting事件为选择操作填充参数，通过SqlDataSourceSelectingEventArgs.Command属性访问将要执行的命令，手工修改参数值
   1. Protected void sourceEmployee\_Selecting(object sender,SqlDataSourceSelectingEventArgs e)

{

//只保留城市名的前三个字符

e.Command.Parameters[“@City”].Value =Request.QueryString[“City”].SubString(0,3);

}

###### 9.3.3 错误处理

1. 底层异常没有得到处理，它会一直向上传递到你的页面，直到破坏整个进程，用户将会看到一些私密的错误信息或出错页面
2. 必须处理错误发生以后的数据源事件(Selected,Inserted,Deleted,Updated)
3. 通过SqlDataSourceStatusEventArgs.ExceptionHandled=true=>避免异常继续传播

Protected void sourceEmployee\_Selecting(object sender,SqlDataSourceStatusEventArgs e)

{

If(e.Exception!=null)

{

//遮盖错误

lblErrorInfo.Text = “An Exception occurred performing the query.”

//考虑被处理的错误

e.ExceptionHandled = true;

}

}

###### 9.3.4 更新记录

1. 支持更新操作的控件：GridView,DetailView,FormView
2. 使用GridView编辑功能=> GridView1.AutoGenerateEditButton = true**;**
3. 更新步骤：
   1. 为执行的操作定义合适的任务，新增(InsertCommand,DeleteCommand,UpdateCommand)

<asp:SqlDataSourceID="SqlDataSource1"runat="server"ConnectionString="<%**$ ConnectionStrings:con1** %>"ProviderName="System.Data.SqlClient"

SelectCommand="SELECT top 15 [userid], [username], [realityName] FROM [T\_Sys\_Public\_Userinfo]"

UpdateCommand="Update T\_Sys\_Public\_Userinfo set username = @username where userid= @origianl\_Username and realityname = @original\_Realityname">

<UpdateParameters>

<asp:ParameterName="username"/>

</UpdateParameters>

</asp:SqlDataSource>

* 1. 限制并发检查：
     1. 更精确的where子句，进行完全匹配.当参数字段和字段名称不一致时的处理方法

SqlDataSource1.ConflictDetection = ConflictOptions.CompareAllValues;

* + 1. 告诉SqldataSource如何命名原始值的参数

SqlDataSource1.OldValuesParameterFormatString = "original\_{0}"**;**

###### 9.3.5 删除记录

1. GridView1.AutoGenerateDeleteButton = true**;**

###### 9.3.6 插入记录

###### 9.3.7 SqlDataSource的不足

* + - 1. 数据访问逻辑嵌套在页面内(需要在网页上硬编码SQL语句)
      2. 在大型应用程序中的维护(每个访问数据库的页面都需要自己的SqlDataSource)
      3. 缺乏灵活性(每个数据访问任务都需要一个单独的SqlDataSource)
      4. 和其他数据任务不兼容(SqlDataSource不能够完成某些类型的任务)

##### 9.4 ObjectDataSource

1. ObjectDataSource在网页上的控件和查询或更新数据的数据访问组件间建立一个连接，ObjectDataSource非常灵活且可以和多种类型的组建一起工作。

2. 遵循的规则

* + 1. 所有逻辑必须包含在单个类中。(如果使用不同的类选择和更新数据，那么必须把它们封装在一个更高层的类中)
    2. 调用方法后它必须提供查询结果
    3. 查询结构必须是记录的集合，可以表现为集合、数组、数据集、数据表以及实现IEnumerable的列表对象
    4. 可以使用实例方法和静态方法。如若使用实例方法，类必须有一个无参构造函数以便ObjectDataSource可以创建所需实例
    5. 必须是无状态的。因为如果ObjectDataSource调用实例方法，创建对象调用实例方法后，ObjectDataSource必须销毁该对象(如果有多个Asp.Net线程调用同一个方法，只需避免出现问题即可)

###### 9.4.1选择记录

1. 第一步：定义ObjectDataSource并指定包含数据访问方法的类的名称，通过TypeName属性提供类的全名来指定(相当于实体类和操作数据库的公共方法)
2. 第二步：将ObjectDataSource指向要绑定的公共方法
3. ObjectDataSource默认情况下显示字段顺序是实体类中属性的声明顺序，而SqlDataSource则是Sql语句中字段的查询顺序
4. 注解：实体类使用错误处理以确保发生错误时连接被正确关闭，但是并没有捕获异常。(使用SqldataSource处理错误方式来处理ObjectDataSource中的错误)
   1. 1.处理Selected,Inserted,Updated,Deleted事件
   2. 2.将异常标志为已处理。
5. ing事件=>在某项任务执行时发生，可以取消或者自制当前正在发生的事件
6. ed事件=>在任务完成后发生，适用于记录活动日志和其他控件同步状态以及处理错误
7. ObjectDisposing=>数据访问对象被释放之前发生(将清理的代码放在Dispose()方法内，可以直接让数据访问类继承System.ComponentModel.Component类并重写Dispose()方法)

###### 9.4.2更新记录

###### 9.4.3使用数据对象执行更新

##### 数据源控件的限制

###### 问题

* + - * 1. 从来都不能显示地处理或者创建用户绑定控件的数据对象(没有机会添加额外的项目)

###### 添加其他项目

1.

###### 9.5.3使用SqlDataSource处理其他选项

###### 9.5.4使用ObjectDataSource 处理其他选项

##### 9.6总结

### 第10章富数据控件

##### 10.1 GridView

###### 10.1.1 GridView简介：

1. GridView是一个用于显示数据的极为灵活的网格控件，它有多易用性特性，包括选择、分页、排序和编辑

###### 10.1.2定义列：

* 1. GridView.AutoGenerateColumns = true =>GridView使用反射来检查数据对象并找到所有字段(记录)或属性(自定义对象)，接着它按发现的次序来为它们逐个创建列
  2. GridView.AutoGenerateColumns = false =>希望(隐藏列、改变它的次序、配置显示)，然后在Columns节自定义列
  3. GridView.AutoGenerateColumns = true =>并且在Columns节自定义列(显示定义的列被加到自动生成的列之前)
  4. 列的类型：
     1. BoundField(用于显示数据源字段的文本)
     2. ButtonField(为列表中的每个项目显示为一个按钮)
     3. CheckBoxField(为列表中的每一个项目显示为一个复选框(对于真、假字段自动使用该类型的列))
     4. CommandField(该列提供选择和编辑按钮)
     5. HyperLinkField(用超链接的形式显示内容(数据源中的一个字段或者静态文本))
     6. ImageField(显示二进制字段的图像数据(通过它显示支持的图片格式))
     7. TemplateField(允许使用自定义模板来绑定多个字段、自定义控件、以及任意格式的html)
  5. BoundField属性
     1. DataField object绑定源字段
     2. DataFormatString string格式化字段DataFormatString = “{0:F2}”=>2位数小数
     3. ApplyFormatInEditMode bool true=>即使在编辑模式时，文本框中的值也会用格式化字符串来格式化

默认为false，即常规形态(显示为1143.02而不是$1,143.02)

* + 1. HeaderText、FootText、HeaderImage string当有标题行(GridView.ShowHeader = true )，有脚注行(GridView.ShowFooter = false )时设置网格标题行和脚注行的文字，标题行常使用描述性的文字，而脚注行常使用动态计算的值。如果在脚注行使用图片而不是文字，则设置HeaderImageUrl属性
    2. ReadOnly bool为true=>在编辑模式不可改变，不会提供任何编辑控件。主键字段通常是只读的
    3. InsertVisible bool为false=>列的值在新增时不可设置
    4. Visible为false=>列不会显示在页面上也不会生成html。
    5. SortExpression object附加在查询之后的表达式。用于执行某一列的排序。和排序一起使用
    6. HtmlEncode =>为真时，所有的文本都用html编码以防止破坏页面。如果想使用超链接就可以禁用它，不过这样不太安全，解决办法：使用HtmlEncode编码所有值，然后响应GridView选择事件提供其他功能可能会更好
    7. NullDisplayText string值为空时要显示的文本，默认是一个空字符串。
    8. ConvertEmptyStringToNull bool为真时，在编辑提交前把所有的空字符串转换成空值
    9. ControlStyle、HeaderStyle、FootStyle、ItemStyle覆盖行的样式，配制本列的外观。

##### 10.2格式化GridView

1. 使用DataFormatString保证日期、时间、货币以及其他数值以合适的方式呈现
2. 希望在网格的各个方面(从标题行到数据)能够应用颜色、字体、边框和对齐的完美组合
3. 截止事件，检查行数据，通过编程来格式化属性
4. 自解释的格式化属性(GridLines：添加或者隐藏表的边框CellPadding，CellSpacing：控制单元格之间的距离Caption，CaptionAlgin：用于在网格的顶部添加标题)

###### 10.2.1 格式化字段

1. 数字格式化字符串

类型 格式化字符串 示例

货币 {0：C} $1,234,50括号表示负值：（$1,234,50），货币符号是与地区相关的

科学计数法(指数) {0：E} 1.23450E+004

百分比 {0：P} 45.6%

固定浮点数 {0：F?} {0：F3}=>112.003 {0:F0}=>112

1. 时间和日期格式化字符串

类型 格式化字符串 示例

短日期 {0：d} M/d/yyyy(10/30/2008)

长日期短时间 {0：f} dddd,MMMM dd,yyyy HH:mm aa(Monday,January 30,2008 10:00 AM)

长日期 {0：D} dddd,MMMM dd,yyyy(Monday,January 30,2008)

长日期长时间 {0：F} dddd,MMMM dd,yyyy HH:mm:ss aa(Monday,January 30,2008 10:00:23 AM)

ISO标准格式 {0：s} yyyy-MM-ddTHH:mm:ss(2008-01-30T10:00:23)

月和日 {0：M} MMMM dd(January 30)

一般 {0：G} M/d/yyyy HH:mm:ss aa(依赖于地区设置)

###### 10.2.2 样式

HeaderStyle、RowStyle、AlternatingRowStyle、SelectedRowStyle、EditRowStyle、EmptyDataRowStyle、FooterStyle、PagerStyle

###### 10.2.3 格式化特定值

响应GridView中的RowDataBound()事件

1. String title = (string)DataBinder.Eval(e.Row.DataItem,”TitleOfCourtesy”);

使用DataBinder.Eval()方法通过反射方式从数据项中读取数据

2．可以吧e.Row.DataItem转换为正确的类型(如ObjectDataSource使用的EmployeeDetail)：DataRowView

##### 10.3 GridView 行选择

1. 加入一个ShowSelectButton为真的CommandField列
2. AutoGenerateSelectButton = true,GridView将自动添加一个选择按钮列

###### 10.3.1使用选择来创建主-从菜单

1. .一个页面添加2个GridView，并用第一个GridView获得的信息在第二个GridView中执行查询
2. SelectIndex返回一个基于0的数字，它反映行网格中位置。
3. SelectDataKey(用逗号分隔一个或者多个关键字段的列表来设置DataKeyNames属性,绑定的每一个名字必须和绑定对象的属性或绑定记录相匹配)

###### 10.3.2 SelectedIndexChanged事件

1. int id = (int)gridView1.selectedDataKey.Values[“EmployeeID”];(从SelectedDataKey属性获取关键字段)

###### 10.3.3将数据字段用作选择按钮

1. 并不需要创建一个新列去支持选择，将现有列变为链接(帮助用户通过唯一标识符选定表中的一行)
2. 添加一个<asp:ButtonField ButtonType = “Button” DataTextField = “EmployeeID” CommandName = “Select”>

指定CommandName属性为Select就可以配置链接产生SelectedIndexChanged事件

##### 10.4对GridView 排序

1. 指定SortExpression=”FirstName”即可实现排序功能

###### 10.4.1使用SqlDataSource排序

1. 使用SqlDataSource时，排序由DataView类内置的排序功能实现(使用DataView排序时，从底层数据库中获得的数据时无序的，排序在内存中进行，这种排序方式不是最快的，且比SqlServer慢，消耗更多资源)

###### 10.4.2使用ObjectDataSource 排序

1. 如果方法返回DataSet或者DataTableObjectDataSource可以和SqlDataSource一样实现自动排序
2. 返回自定义集合，那么必须提供一个接收排序表达式并执行排序的方法(需要创建一个接收一个字符串作参数的方法来使用排序参数，还必须让ObjectDataSource.SoreParameterName的设置值使用和这个参数一模一样的名字)=>

<asp:ObjectDataSource ID=”sourceEmployees” runat=”server” TypeName=”DatabaseComponent.EmployeeDB” SelectMethod=”GetEmployees” SortParameterName=”sortExpression” />

Public EmployeeDetail[] GetEmployees(string sortExpression){}

应用排序：DataView dv = ds.Tables[0].DefaultView; view.Sort = sortExpression

1. 根据排序表达式改变执行的实际查询(在Sql语句中执行排序)
   1. 1.动态创建一个带有Order by子查询的Sql语句。=>Sql注入攻击危险
   2. 2.编写条件逻辑以检查排序表达式并据此执行不同的查询(在查询方法或在存储过程中)。=>代码会显得零碎而且涉及众多的字符串解析

###### 10.4.3排序和选择

1. 问题：选定一行后按任意一列排序，该行的选择仍然会被保留

解决：在GridView.Sorted事件中清空选择(不支持分页，分页会造成混乱)

1. Protected void gridview1\_Sorted(object sender,EventArgs e)

{

ViewState[“SelectValue”] = gridview1.SelectedValue.ToString();//保存所选值

Gridview1.SelectedIndex = -1;//清楚所选索引

}

1. Protected void gridview1\_DataBound(object sender,EventArgs e)

{

If(ViewState[“SelectValue”]!=null)

{

String selectedValue = (string)ViewState[“SelectValue”];

//重新选择最后被选定的行

Foreach(GridViewRow row in gridview1.Rows)

{

String keyValue = gridview1.DataKeys[row.RowIndex].Value.ToString();

If(keyValue ==selectedValue)

{

Gridview1.SelectedIndex = row.RowIndex;

Return;

}

}

}

}

###### 高级排序

1. 处理GridView.Sorting事件(在排序执行前发生，可以修改排序表达式)
   1. Protected void gridview1\_Sorting(object sender,GridViewSortEventArgs e)
      * 1. {
           1. If(e.SortExpression == “FirstName”&&gridview1.SortExpression==”LastName”)
           2. {
        2. SortExpression = “LastName,FirstName”;//在当前排序和所要求的排序基础上复合排序
           1. }
        3. }
2. 通过编程提供一个排序表达式并调用GridView.Sort()方法实现GridView排序(在一个大的数据报表呈现给用户之前需要执行排序，希望用户从预定义的排序选项列表(列在其他控件里)里选择排序)
   1. Protected void listview1\_SelectedIndexChanged(object sender,EventArgs e)

{

Gridview1.Sort(listview1.SelectedValue,SortDirection.Ascending);

}

##### 10.5 GridView 分页

###### 自动分页

1. GridView分页成员

属性 描述

AllowPaging bool 是否启用绑定记录的分页，默认不启用

PageSize int 每页显示数据条数默认是10

PageIndex int分页启用时，获取或者设置基于0的当前显示页码

PagerSetting object是一个为分页控件封装了各种格式化选项的PagerSetting对象

PagerStyle object是分页控件样式

PageIndexChanging 页面选择元素被单击时发生，在导航之前(ing)或之后(ed)

PageIndexChanged

1. 适用范围
   1. 任何继承ICollection接口的数据源都可以使用自动分页(只要使用DataSet，SqlDataSource就支持自动分页，如果使用DataReader就会抛异常)
   2. 自定义数据访问类返回了继承了ICollection的对象(数组，强类型集合，非连接的DataSet)，ObjectDataSource也支持自动分页
   3. 缺点：每次当前PageIndex改变的时候，都需获取和绑定所有数据。分十页就需要执行十次相同的工作(同时以十倍的因子加重数据库的总体负担)使用数据源空间缓存可以减缓，但是当缓存中保存完整的DataSet所需大量的内存时，也不行=>自定义分页

###### ObjectDataSource 的自定义分页

1. 自定义分页优点：
2. 允许最小化宽带需求并且避免在服务器内存中保存大量数据
3. GridView使用的数据量越大，越应该使用自定义缓存
4. 数据库服务器越慢且负载越重，就越趋向于缓存完整的数据对象并且重复利用
5. 缺点：会给数据库带来更多的工作
6. ObjectDataSource是唯一支持自定义分页的数据源
7. 设置ObjectDataSource.EnablePaging = true
8. 设置StartRowIndexParameterName,MaximumRowsParameterName,SelectCountMethod
9. 实现方法:
10. 记录计数(相当于PageSize):GetEmployeeCount()=>该方法获取记录数<asp：ObjectDataSource ID=”sourceEmployee” runat=”server” EnablePaging = “True”>SelectCountMethod = “GetEmployeeCount”…./> SelectCountMethod每次回发时都会执行(可以缓存，但是数目可能不精确)用于获取分页记录的存储过程GetEmployees()方法不再是获取所有记录集合，而是获取将在当前页面显示的部分记录
11. 分页的选择方法
12. 重载GetEmployees(分页开始的行号，每页最大数)方法，通过StartRowIndexParameterName，MaximumRowsParameterName两属性来指定要使用的参数

###### 自定义分页栏

* 1. 改变外观：PagerStyle和PagerSetting
  2. 重要细节：PagerSetting.Mode(numeric,NextPrevious,NumericFirstLast,NextPreviousFirstLast)
  3. 自定义外观：PagerTemplate
  4. 缺点：每次对网格重新排序或移动到另外一个页面时，浏览器需要触发一个回送并呈现全新html页面，这意味着页面将闪动并且回滚到开始，用户体验不和谐
  5. 解决方法：

EnableSortingAndPagingPagingCallBacks = true

##### GridView 模板

1. Eval():System.Web.UI.DataBinder的一个静态方法，自动读取绑定到当前航的数据项、使用反射找到匹配字段。(<%#((EmployeeDetail)Container.DataItem)[“FirstName”]%>)原始写法
2. 使用DataSet模式绑定时，数据项是一个DataRowView、DataReader则是DbDataRecord
3. Eval() 格式化数据字段<%#Eval(“Birthday”,”{0:MM/dd/yy}”)%>

###### 使用多个模板

HeaderTemplate,FooterTemplate,ItemTemplate,AlternatingItemTemplate,EditItemTemplate,InsertItemTemplate

###### 10.6.2在VisualStudio中编辑模板

###### 10.6.3绑定到方法

1. <asp:TemplateField HeaderText=”Status”>

<ItemTemplate>

<img src=’<%#GetStatusPicture(Container.DataItem)%>’ alt=”Status”/>

</ItemTemplate>

</asp:TemplateField>

2.Protected void GetStatusPicture(object dataItem)

{

Int units = Int32.Parse(DataBinder.Eval(dataItem,”UnitsInStock”).ToString());

If(units==0)

{

Return “Cancel.gif”;

}

}

3. 如果使用数据绑定表达式绑定到方法，你将不再可以使用回调来优化GridView的刷新过程

###### 10.6.4处理模板中的事件

###### 10.6.5使用模板编辑

1. 绑定一个编辑值到字段时，必须使用Bind()方法，会创建双向连接，更新后的值将会送回服务器

##### 10.7 ListView

###### 10.7.1分组

###### 10.7.2分页

##### 10.8 DetailsView 和 FormView

###### 10.8.1 DetailView

###### 10.8.2 FormView

.

##### 10.9高级网格

###### 10.9.1 GridView中的汇总

1. 设置GridView.ShowFooter=true
2. 在GridView.FooterRow中加入内容.

###### 10.9.2单个表中的父/子关系

###### 10.9.3使用检查表编辑字段

###### 10.9.4处理来自数据库的图片

###### 10.9.5探测并发冲突

##### 10.10总结

### 第11章缓存和异步界面

1. 定义：
   1. 缓存是在内存中保存创建代价高的信息副本的一种技术(可以同时提高性能和可扩展性)
2. 自我约束：
3. 服务器内存资源有限，如果保存信息过多，这些信息就会保存到页面的硬盘文件上，导致整个系统减慢

##### 理解Asp.Net缓存

1. 缓存类型：
   1. 输出缓存：

最简单的缓存类型，只保存最终生成的Html的一个副本。下次该页面再次被请求时，页面没有真正运行，显示的则是html副本中的内容

* 1. 数据缓存：

由代码手工导入的将重建耗时(如:从数据库获得的DataSet)的重要对象保存其中的一种缓存对象。其他页面可以检查这一信息是否存在而重用它。数据缓存从概念上来说和使用应用程序状态相同，但是它对服务器更友好，因为缓存太大或者性能受影响时项目可以自动从缓存移除，项目还可以自动过期

* 1. 部分页缓存：

输出缓存的特殊类型-它不像输出缓存那样缓存整个页面的html，而是只缓存部分html，部分页缓存保存页面上用户控件的输出。

* 1. 数据源缓存：

这种缓存建立在数据源控件内，技术而因，数据源缓存使用数据缓存，差别是：你不必显示处理这一过程，你只是要配置适当的属性，数据源控件就会自行管理缓存的保存和读取

##### 输出缓存

1. 输出缓存特征：
   1. 使用输出缓存时，页面最终产生的html被缓存，页面再次被请求时：不会创建控件对象，不会开始新的页面生命周期，代码也不会被执行
2. 备注：

a) Asp.Net页面可能会使用不经过Asp.Net处理的静态资源。不用担心这些项目的缓存，IIS使用比Asp.Net缓存更有效的方式来缓存文件

###### 11.2.1 声明性的输出缓存

1. <%@OutputCacheDuration="20"VaryByParam="None"%>
2. 提示：重新编译缓存页面时，Asp.Net会自动移除缓存
3. Duration表示缓存时间
4. VaryByParm表示缓存有效环境

###### 11.2.2 缓存和查询字符串

1. VaryByParm设置概述：
2. None：表示任何场所都适应
3. \*：页面使用查询字符串时，告诉Asp.Net按不同的QueryString缓存页面的独立副本

###### 11.2.3 使用特定查询字符串参数的缓存

1. <%@OutputCacheDuration="20"VaryByParam="ProductID;CurrencyType"%>

(使用ProductID参数和CurrencyType参数的请求会被缓存，ProductID=1，ProductID=2,ProductID=3的数据从第一次请求后都会被缓存20秒)

1. 注解：

输出缓存可以随服务器端数据、查询字符串中的数据的变化而很好的工作，但是，输出缓存不可以随依赖特定用户的信息(如Cookie)产生的页面输出一起工作。输出缓存也不可以和使用表单的事件驱动的页面一起工作。(此时事件被忽略，每次请求只会重送静态页面)=>应该使用部分缓存或者数据缓存来缓存特定信息，但不能缓存整个页面

###### 11.2.4 自定义缓存控制

1. Asp.Net允许你创建自己的过程来确定是保存一个新的页面版本还是使用现有的缓存。这样的代码检查任意合适的信息并且返回一个字符串，Asp.Net使用这个字符串来使用缓存，如果你的代码为不同的请求使用了相同的字符串，那么页面将会使用缓存,否则，Asp.Net会产生一个新的缓存版本并单独保存它
2. <%@OutputCacheDuration="20"VaryByParam="MyCache"%>
3. 在Golbal.asax应用程序中创建用于产生自定义缓存字符串的过程，重载GetVaryByCustomString()方法
4. VaryByHeader:允许你从请求的Http标题获得的值来保存单独版本的页面

<%@OutputCacheDuration="25"VaryByParam="None"VaryByHeader="Accp Tests"%>

###### 11.2.5 使用HttpCachePolicy类进行缓存

1. 内置属性Response.Cache=>提供一个System.Web.HttpCachePlicy的实例，允许你通过编程控件缓存

//在服务器上缓存该页面

**Response.Cache.SetCacheability(**HttpCacheability.Public);

//在进入该页面60秒后启动缓存

**Response.Cache.SetExpires(**DateTime.Now.AddSeconds(60));

1. Response.Cache适用于输出缓存，Page.Cache适用于数据缓存

###### 11.2.6 缓存后替换和部分页缓存

1. 不能缓存整个页面（解决办法）：
2. 部分页缓存：找出要缓存内容，然后封装到专用的用户控件中，然后缓存该控件的输出
3. 实现办法：创建一个用户控件，然后把OutputCache指令加入用户控件即可(页面不会缓存，控件会被缓存)
4. 缺点：如果页面获得是用户控件的一个缓存版本，它就不能通过代码和控件交互(如：你的控件提供一些属性，web页面代码就不能修改或访问这些属性)。使用缓存版本的用户控件的时候，只是把一段html插入到页面上，响应的控件对象不可用
5. 缓存后替换：找出动态内容，用某些使用Substitution控件的东西替代这部分内容
6. 实现办法：HttpResponse.WriteSubstitution()接收一个参数：一个指向页面类中回调方法的代理。返回页面要替换的部分
7. 原理：Asp.Net框架页面接收一个缓存页面=>自动获得回调方法获得动态内容=>将获得动态内容自动插入到该缓存页面(创建一个生成动态内容的方法，Asp.Net确保这个方法每次都会被调用而且它的内容从来不会被缓存)
8. 注解：生成动态内容的方法必须是静态方法(因为页面内容从缓存提供，页面对象不会被创建)
9. 缺点：页面布局无法控制
10. 解决办法：把动态的内容封装在某个控件中，控件自身呈现时可以使用Response.WriteSubstitution()
11. Substitution控件：该控件自己的内容全部是动态的
12. 注解：自定义控件可以自由使用Response.WriteSubstitution()方法来设置自身的缓存行为。例如：即使页面由输出缓存提供，AdRatator使用这项功能保证页面的广告内容总是变动的

###### 缓存用户配置

1. 缓存用户配置：Asp.Net允许用户对一组页面应用相同的缓存设置
2. 实现步骤：
   1. 在配置文件中配置：

<system.web>

<caching>

<outputCacheSettings>

<outputCacheProfiles>

<addname="MyCachesSetting"duration="60"varyByParam="None"/>

</outputCacheProfiles>

</outputCacheSettings>

</caching>

</system.web>

* 1. 在需要应用的页面添加：

<%@OutputCacheCacheProfile="MyCachesSetting"%>

###### 11.2.8缓存配置

##### 11.3数据缓存

1. 原理：把创建代价高的项加入到一个特殊的内置集合对象中(Cache)
2. 和Application的却别：
3. Cache是线程安全的：
   1. 意味着添加或者移除项目的时候不必显示锁定或者解锁Cache集合
   2. Cache集合中对象线程必须是安全的
   3. 如果创建一个自定义对象，多个客户使用，就会产生非法数据(在web页面里创建多个副本)
4. 缓存中的项目是自动移除的
   1. 项目过期时、项目所依赖的对象或者文件发生变化时、服务器内存变低时，Asp.Net会自动移除Cache里的缓存项目(所以每次使用时都要检查该缓存对象是否存在)
5. 缓存内的项目支持依赖性
   1. 可以把缓存的对象链接到文件、数据库表或其他资源。(如果该资源发生变化，缓存的对象会被自动标识为非法并被移除)

###### 11.3.1 向缓存添加项目

1. 添加缓存事项：
2. 过期策略：
   1. absoluteExpiration:TimeSpan.Zero=>绝对过期：明确知道一个指定项目的合法时间，如：股票图、天气预报Cache.Insert(“MyItem”,2,null,DateTime.Now.AddMinutes(60),TimeSpan.Zero);
   2. slidingExpiration:DateTime.Max=>可调过期：Asp.Net等待一段不活动的时间后释放那些不活动的缓存对象Cache.Insert(“MyItem”,1,null,DateTime.MaxValue,TimeSpan.FromMinutes(10));
3. 清空缓存

foreach**(**DictionaryEntry **de** in **Cache)**

**{**

**Cache.Remove(de.Key.ToString());**

**}**

###### 11.3.2 简单的缓存测试

###### 11.3.3 缓存优先级

1. 适用范围：优先级只在Asp.Net需要执行缓存清理时有效，这是一个因为内存紧张而提前移除缓存项目的过程
2. NotRemovable=>High=>AboveNormal=>Normal=>BelowNormal=>Low

###### 11.3.4 使用数据源控件的缓存

1. 如果三个控件绑定到同一个数据源，在页面呈现前需要执行三次独立的数据库查询(使用缓存)
2. 数据源控件的相关缓存属性：

属性 描述

EnableCaching ture使用缓存

CacheExpirationPolicy DataSourceCacheExpiry (Absolute,Sliding)过期策略

CacheDuration 缓存对象保存时间

CacheKeyDependency 允许让缓存的项目依赖数据缓存中的其他项目

SqlCacheDependency 允许让缓存的项目依赖数据库中表

1. SqlDataSource缓存尽在DataSourceMode = DataSet有效
2. ObjectDataSource它只返回DataSet或者DataTable，如果返回对象类型NotSupportException异常

##### 缓存依赖

1. 缓存依赖：缓存依赖允许你让被缓存的项目依赖其他资源，这样当该资源发生变化时缓存项目就会被自动移除
2. 缓存依赖类型：1.其他缓存项目；2文件或者文件夹；3.数据库查询

###### 11.4.1 文件和缓存项目依赖

1. 创建缓存依赖：CacheDependency **cdd =** newCacheDependency**(Server.MapPath(**"aa.xml"**));**
2. 添加一个依赖于此文件的缓存：Cache.Insert("TestItem"**, testItem,cdd);**
3. 对文件夹中的所有文件执行删，改，重命名，则会引发缓存依赖，新增文件夹不会

###### 11.4.2 聚合依赖

1. 聚合依赖：一个项目依赖于多个资源
2. CacheDependency **cd1 =** newCacheDependency**(Server.MapPath(**"a1.xml"**));**

CacheDependency **cd2 =** newCacheDependency**(Server.MapPath(**"a2.xml"**));**

CacheDependency**[] dependencys =** newCacheDependency**[] { cd1, cd2 };**

AggregateCacheDependency **acd =** newAggregateCacheDependency**();**  **acd.Add(dependencys);**

3. AggregateCacheDependency的真正价值体现在能够同时包含多个从CacheDepenpency继承的任意对象

###### 11.4.3 移除项目回调

1. 回调的基本用法，只是在回调方法处理前后不能和页面进行交互
2. 对不存在的项目调用Cache.Remove()也是安全的
3. CacheItemRemoveReason枚举，项目移除原因

###### 11.4.4 理解Sql缓存通知

1. 当数据库中的相关数据被修改时自动使缓存的数据对象失效的技术

###### 11.4.5 SqlServer2000 和Sqlserver2007 中的缓存通知

###### 11.4.6 Sqlserver 2005 和 Sqlserver 2008 中的缓存通知

##### 11.5自定义缓存依赖

###### 11.5.1 基本的自定义缓存依赖

###### 11.5.2 使用消息队列的自定义缓存依赖

##### 11.6异步界面

1. 异步web页面优点：

可以大大提高网站的可伸缩性，尤其适用于那些包含消耗大量时间查询数据库的代码的页面

1. 原理：

把需要长时间等待的代码移到非Asp.Net线程里

1. 503错误：

.Net维持一个能够处理页面请求的线程池。获得一个新的请求时，Asp.Net从中取出一个可用的线程并用它处理整个页面。这个线程同时实例化页面、运行事件处理代码以及返回呈现的html。如果请求频率很高(比它处理请求还要快)，那么未处理的请求将被保存在队列里，如果队列满了，Asp.Net将会产生503错误”服务器不可用”来拒绝额外的请求

1. 如果页面代码需要长时间等待(如：读取远程文件、调用远端计算机上的对象或web服务，从一个较慢的数据库查询大量数据)，即便没有实际工作在执行，请求线程也会被挂起。Web服务器具有处理更多请求的资源(因为你的线程没有占CPU),但他没有任何可用的线程。根据等待时间的长短以及网站的请求量，这可能会大幅影响网站的吞吐量，阻止网站处理它本应该能够处理的请求数，如果页面涉及频繁的等待，就可用使用异步页面来释放Asp.Net请求线程。
2. 异步页面并不比普通的同步页面快，实际上，来回切换线程带来的额外负载可能使它更慢一些，好处在于其他请求(不涉及长时间操作的那些请求)可用更快的被处理
3. 异步web页面和异步客户端编程技术(Ajax)。服务器端异步Web页面处理的潜在优势是它允许你更有效的处理耗时的请求，因此在请求繁忙的时候其他用户不需要等待。客户端异步编程潜在的优势是页面对终端用户显得更具响应性

###### 11.6.1 创建异步界面

1. <%@PageAsync="true"%> 告诉Asp.Net它要生成的页面类应该实现IHttpAsyncHandler而不是IHttpHandler

###### 11.6.2 在异步界面中查询数据

###### 11.6.3错误处理

###### 11.6.4在异步任务中使用缓存

###### 11.6.5多异步任务和超时

##### 11.7总结

### 第12章文件和流

1. RDBMS(数据库)：多用户处理能力、实时处理并发访问、支持缓存、支持低层次的磁盘优化

访问其他地方(文件系统)的数据问题：

1. 读取其他程序产生的信息、为测试而编写临时日志、创建管理文件以便管理员可以上传文件以及查看服务器当前情况

##### 12.1 使用文件系统

1. 最基本的文件访问：获取现有文件和目录的信息、执行典型的文件系统操作(复制文件、创建目录)
2. System.IO:
   1. Directory、File：提供一组静态方法，可以通过它们获取任意服务器上可见文件和目录的信息
   2. DriveInfo、DirectoryInfo、FileInfo：使用相似的实例方法和属性获取同上的信息
   3. 差别：DriveInfo、DirectoryInfo、FileInfo调用前必须实例化，Directory、File则不必
   4. Directory、File：更适合处理一次性任务,每次调用的时候都要执行安全检查
   5. DriveInfo、DirectoryInfo、FileInfo：获取若干信息更好，不必再每次调用方法时都提供文件或目录的名称，而且还更快一些，因为他们只在你创建实例对象的时候执行一次安全检查

###### 12.1.1 Directory 类和File类

1. Directory方法：

方法 描述

CreateDirectory() 创建一个新目录，如果指定一个不存在的目录创建目录，Asp.Net也会创建所有需要的目录

Delete() 删除对应的空目录，如果要删除目录及其内容，则要设置第二个可选参数为真

Exists() 返回真或假表明指定目录是否存在

GetCreationTime() 返回目录创建、访问、写入时间。还有个Set方法

GetLastAccessTime()

GetLastWriteTime()

GetDirectories() 返回一个字符串数组，每个字符串代表在指定目录中的一个子目录或者文件，每个

GetFiles() 方法可以接受指定搜索表达式(如ASP\*.\*)的第二个参数

GetLogicalDrives() 返回一个字符串数组，返回类似如c:\

GetParent() 解析提供的目录字符串并告诉你它的父目录。你可以自行查找”\”字符或者，更准确一些，Path.DirectorySeperatorChar)实现这一功能

GetCurrentDirectory() 设置和获取当前目录，当使用相对路径而不是完整路径时这样特别有用

SetCurrentDirectory()

Move() 接收两个参数：原路径和目标路径。目录及其内容可以移动到任意路径，只要在同一磁盘即可

GetAccessControl() 返回一个System.Security.AccessControl.DirectorySecurity对象，你可以通过编程使

SetAccessControl() 用这个对象查看设置改变应用到目录的windows访问控制列表(ACL)

1. File方法

方法 描述

Copy() 接收2个参数，源文件和目标文件的完整文件名。第3个参数表示是否允许覆写

Delete() 删除指定文件，找不到文件不会抛异常

Exists() 指定文件是否存在

GetAttributes 获取或者设置一个可以包含FileAttributes枚举中的值的任意组合的枚举值

SetAttributes

GetCreationTime() 同Directory

GetLastAccessTime()

GetLastWriteTime()

Move() 同Directory，但是可以跨磁盘移动文件甚至在移动时给文件改名(或者不移动直接改名)

Create()、CreateText() 创建一个指定的文件，并返回一个可用于写的FileStream对象。CreateText()执行同样的任务，但是返回一个封装了流的StreamWriter对象

Open()、OpenText() OpenText()、OpenRead()以只读方式打开文件,返回一个FileStream或StreamReader

OpenRead()、OpenWrite() OpenWriter() 以只写方式打开文件、返回FileStream()

ReadAllText() 读取整个文件、并以单一字符串、字符串数组(每个字符串代表一行)、字节数组的

ReadAllLine() 形式返回内容。仅在文件较小时才使用这些方法。对于大型文件，使用流每次读取

ReadAllBytes() 一块以节省内存使用

WriteAllText() 使用提供的字符串、字符串数组(每个字符串代表一行)、字节数组一次写入完整文

WriteAllLine() 件的如果文件已经存在，会覆盖原文件

WriteAllBytes()

GetAccessControl() 返回一个System.Security.AccessControl.DirectorySecurity对象，你可以通过编程使

SetAccessControl() 用这个对象查看设置改变应用到目录的windows访问控制列表(ACL)

1. 注意C#中文件路径的写法：@”C\Program Files\”和”C\\Program Files\\”是等效的。

###### 12.1.2 DirectoryInfo 类和 FileInfo类

1. DirectoryInfo和FileInfo成员

1. DirectoryInfo和FileInfo类是Directory和File类功能的镜像
2. Directory和File只提供了方法，而前者也提供了属性
3. DirectoryInfo和FileInfo成员

成员 描述

Attributes 允许你使用FileAttribute枚举的值组合来读取或设置特性

GetCreationTime() 创建时间、最近访问时间、最后写入时间

GetLastAccessTime()

GetLastWriteTime()

Exists() 可以创建与当前物理目录并不对应的FileInfo对象或者DirectoryInfo对象。但不能使用CreationTime之类的属性或MoveTo()之类的方法

FullName、Name 完整名称、目录或文件名(带有扩展名)、扩展名

Extension

Delete() 如果或目录存在，则移除。删除目录时，目录必须为空；否则必须指定一个为真的可选参数

Refresh() 更新对象使其与文件系统发生的任意变化同步

Create() 创建指定的文件或目录

MoveTo() 复制目录及其内容或者文件。对于DirectoryInfo对象，你需要指定新路径：对于FileInfo对象，你需要指定路径和文件名

1. DirectoryInfo独有特性

Parent和Root 返回一个代表父或根目录的DirectoryInfo对象

CreateSubdirectory() 在DirectoryInfo对象代表的目录中按指定名称创建一个目录。它返回一个代表创建的子目录的DirectoryInfo对象

GetDirectories() 返回代表当前目录的所有子目录的DirectoryInfo对象数组

GetFiles() 返回代表当前目录中所有文件的FileInfo对象的数组

1. FileInfo独有特性

Directory 返回一个代表父目录的DirectoryInfo对象

DirectoryName 返回父目录的字符串名称

Length 返回长整型(64位整数)的文件字节数

CopyTo() 把文件复制到参数所指定的新路径和名称。它返回一个代表新(复制的)文件的FileInfo对象。可以加入一个为“真”的参数以允许覆写

Create() 创建指定的文件夹、并返回一个可用来写文件FileStream对象。

CreateText() CreateText()执行同样的任务但返回封装了流的StreamWriter对象

Open()、OpenRead() 打开一个文件(假设存在)、OpenRead()、OpenText()以只读方式打开文件返

OpenText()、 FileStream或StreamReader。OpenWriter()以只写方式打开文件返回一个

OpenWriter() FileStream

2. void **CreateDirectoryAndFiles()**

**{**

string **path =** DateTime.Now.ToString("yyyyMMddHHmmss"**);**

//定义新的目录

DirectoryInfo **di =** newDirectoryInfo**(**@"C:\Temp\" **+ path);**

//定义新的文件 (还不是创建)

FileInfo **fi =** newFileInfo**(**"C:\\Temp\\" **+ path +** "\\" **+ path +** ".txt"**);**

//创建，注意顺序，不能在一个不存在的目录里面创建文件

**di.Create();**

FileStream **fs = fi.Create();**

**fs.Close();** //释放流以便操作

**}**

**3.** long **CalculateDirectorySize(**DirectoryInfo **di,**bool **includeSubdirectories)**

**{**

long **totalsize = 0;**

//增长每一个文件

FileInfo**[] files = di.GetFiles();**

foreach**(**FileInfo **fi** in **files)**

**{**

**totalsize += fi.Length;**

**}**

//增长每一个子目录，则执行本操作

if**(includeSubdirectories)**

**{**

DirectoryInfo**[] dti = di.GetDirectories();**

foreach**(**DirectoryInfo **dif** in **dti)**

**{**

**totalsize += CalculateDirectorySize(dif,** true**);**

**}**

**}**

return **totalsize;**

**}**

###### 12.1.3 DriveInfo类

###### 12.1.4使用Attributes

###### 12.1.5使用通配符过滤文件

1. 得到所有.txt格式文件

void **GetAllTxtFiles()**

**{**

DirectoryInfo **di =** newDirectoryInfo**(**@"C:\Temp\"**);**

FileInfo**[] fis = di.GetFiles(**"\*.txt"**);**

**}**

###### 12.1.6获取文件的版本信息

###### 12.1.7 Path类

1. 用处，使用文件路径或者目录路径首先(因为用普通字符串保存不安全，规范化错误)

2. 规范化错误：特殊类型的应用程序错误，它会在你的代码假定用户提供的值总是符合标准时发生。是一项低层的技术，但是错误一旦发生通常会导致用户可以执行某个本应该收到限制的动作(Sql注入式攻击)

3. Path类可以从文件和目录方面来处理Sql注入式攻击

4. Path方法

方法 描述

Combine() 合并文件和目录的路径

ChangeExtension() 修改字符串中文件的扩展名，如果未指定扩展名，当前扩展名会被删除

GetDirectoryName() 返回目录信息，它是第一个和最后一个目录分隔符(\)之间的文本

GetFileName() 返回路径的文件名部分

GetFileNameWithoutExtension() 和上面类似，但是忽略了扩展名

GetFullPath() 该方法对绝对路径无效，按当前目录把相对路径转换为绝对路径

GetPathRoot() 获取指定路径的根目录(例如：C:\)字符串，对于相对目录，返回Null

HasExtension() 确定路径是否包含扩展名

IsPathRooted() 如果是绝对路径则返回真，相对路径则返回假

###### 12.1.8文件浏览器

##### 12.2 使用流读写文件

流：允许你用相似的方式(作为顺序节流)对待不用数据源的一种抽象，所有的.Net流类继承自System.IO.Stream

创建一个流FileStream **fs =** newFileStream**(**Path**,**FileMode**,**FileAccess**,**FileShare**)**

FileMode枚举值 :

Append 如果文件存在就打开并找到文件尾，否则创建一个新文件

Create 创建新文件，已存在就覆盖

CreateNew 创建新文件，已存在就报错

Open 打开现存文件

OpenOrCreate 文件存在就打开，否则创建新文件

Truncate 打开现存文件，打开后，文件被截断至0字节(清空文件)

###### 文本文件

1. 创建一个文件流

FileStream fs = new FileStream(@”C:\myfile.txt”,FileMode.Created);

1. 将文件流作为参数创建一个写入流

StreamWriter sw = new StreamWriter(fs);(StreamWriter sw = File.CreateText(@”C:\myfile.txt”));

1. 使用写入流进行写入操作

Sw.write(“This is a string”);(sw.writeline(“This is a string”));

1. 写入操作结束后，关闭写入流

Sw.Flush();sw.Close();

1. 将文件流作为参数创建一个读入流

StreamReader sr = new StreamReader(fs);(StreamReader sr = File.OpenText(@”C:\myfile.txt”));

1. 使用读入流进行进行读取操作

Sr.Read();(Sr.ReadLine();)

1. 其他操作

Sr.ReadToEnd()=>读取整个文件内容

File =>ReadAllText()、ReadAllBytes()=>适用于小型文件。大型文件应该使用FileStream一次读取一部分内容来减轻内存负载

###### 二进制文件

1. BinaryWriter bw = new BinaryWriter(File.OpenWrite(@“C:\Binaryfile.bin”))
2. 写入流的时候可以不关注流的类型，但是读取的时候就一定要指定流的类型
3. Int=>ReadInt32,String=>ReadString
4. Seek()=>底层方法，偏移

###### 上传文件

上传控件：

HtmlInputFile =>html服务器控件 =>必须指定form的enctype = mullipart/form

FileUpload=>Asp.Net服务器控件 =>Asp.Net已经封装好，不存指定

###### 12.2.4使文件对多用户安全

###### 12.2.5压缩

##### 12.3 序列化

##### 12.4 总结

### 第13章 Linq

##### 13.1 Linq基础

1. Linq:语言继承查询(Language Integerated Query)
2. Linq to SQL与Linq to XML(两种能够访问外部数据的linq)
3. Linq to DataSet与Linq to Objects(呈现定制化数据视图提供了灵活性)
4. Linq to Objects的本质是：使用声明性的Linq表达式替代迭代逻辑

###### 13.1.1 延迟执行

1. foreach与linq的区别：

* + - 1. 最主要区别：处理匹配集合类型的方式
      2. Foreach：匹配的集合被创建为一个特定类型的集合(强类型的List<T>)
      3. Linq：匹配的集合仅通过linq实现的IEnumerable<T>的集合

2. 嵌入在System.Linq.Enumerable类的一个私有类

* + - 1. 不含Where条件的SelectIterator<T>实例
      2. 复合了多个集合的UnionIterator<T>实例
      3. 包含Where条件的WhereIterator<T>实例

1. 通过IEnumerable<T>接口和结果交互，所以不必知道代码所使用的具体迭代类
2. 当在Linq迭代对象如WhereIterator<T>上进行迭代时，它自动获取所需要的数据，当调用控件的DataBind()方法时，Asp.Net就会对匹配结果集进行迭代以便获得它需要的数据，并把它传递给控件
3. 如果从数据库获取结果集或在结果集上应用了排序指令，linq就会使用一个不同的策略并在循环的开始获得全部结果

###### 13.1.2 Linq 使如何工作的

* 1. 使用linq时需要创建一个linq表达式
  2. Linq表达式的返回值是一个实现了IEnumerable<T>的迭代器对象
  3. 对迭代器对象进行枚举时，linq执行它的工作
  4. Linq to objects用户获取数据的代码几乎总是比用foreach块进行比较执行得要慢，原因是，在工作时，linq需要更多的代理和及方法调用，使内存对象操作性能更好。相反，连接数据库、访问web服务、从文件系统获取信息之类的操作会更慢一些。但是使用索引查找包含大量已经排序的信息的集合要远比linq查询效率高，linq会从头到尾遍历所有数据

###### 13.1.3 Linq 表达式

1. 投影：将查询的数据转换为各种结构的能力
2. 分组：var matches = from employee in employees group employee by employee.TitleOfCourtesy into g select g;

Var categories = from p in products group p by p.category into g select new {Category = g.Key,MaximumPrice = g.Max(p=>p.UnitPrice),MinimumPrice = g.Min(p=>p.UnitPrice),AveragePrice = g.Average(p=>p.UnitPrice)};

###### 13.1.4 揭秘Linq表达式

1. 匿名方法

Var matches = employees.Select(delegate(EmployeeDetails employee){return new {First = employee.FirstName,Last = employee.LastName}};);

简化后：matches = employees.Select(employee=>new{First = employee.FirstName,Last = employee.LastName});

##### 13.2 Linq to DataSet

1. Field<T>:扩展所有的DataRow对象，并让你能够以强类型的方式访问字段

String value = dataRow.Field<string>(“FirstName”);

2. AsEnumable():将弱类型的DataTable转化成Linq对象

###### 13.2.1强类型的DataSet

1. 强类型的DataSet不必依赖于Field<T>和AsEnumable<T>扩展方法

###### 13.2.2空值

1. Field<T>将空值(DBNull.Value)转换成为一个真正的空引用

Var matches = from p in ds.Tables[“products”].AsEnumable()

Where p.Field<DateTime>(“DiscountinedTime”)!=null

Select p;

##### 13.3 Linq to Sql

###### 13.3.1数据实体类

###### 13.3.2 DataContext

1. DataContext dc = new DataContext(connectString);

Table<EmployeeDetails> table = dc.GetTable<EmployeeDetails>();

###### 13.3.3揭秘Linq to Sql 查询

1. 在Table<T>对象上执行linq表达式时，linq表达式返回一个System.Data.Linq.DataQuery对象。DataQuery对象封装获取结果所需的细节，即Sql查询(DataQuery保存了获取数据的命令并通过一个枚举器暴露数据)

2. IEnumerable<EmployeeDetails> matches = from employee in table where TestName(employee.LastName) select employee报错。不支持

IEnumerable<EmployeeDetails> matches = from employee in table.AsEnumerable(EmployeeDetails) where TestName(employee.LastName) select employe就可以了

本质上：使用Linq to sql从数据库获取数据，然后利用linq to objects过滤内存中的对象。必须从IQueryable<T>接口(由Table<T>对象实现)降级到更基本的IEnumerable<T>接口

1. 注解：

IEnumerable<T>为Linq to objects提供内存过滤

IQueryable<T>为Linq to sql提供动态查询的能力

1. 注意：
   1. 由于延迟执行的模型，直接对结果进行迭代，linq to sql才会打开对数据库的连接。如果此时发生连接错误，会被抛出
   2. 当对数据跌打完毕时，linq to sql自动关闭连接。迭代、或者DataReader对象循环的时候尽量不要执行非常耗时的操作，实在不行的话，就调用ToList<T>或者ToArray<T>方法，他们立即查询数据并返回结果的强类型或者数组
   3. 每次跌打结果都会执行linq to sql查询，如果要把相同的结果集绑定到多个不同的控件，就会号不必要的增大数据库要执行的工作(并增大不一致数据的可能性)，实在不行的话，可以调用 ToList<T>或者ToArray<T>获得唯一的集合然后进行绑定
   4. 每次对结果进行迭代的时候都会执行linq to sql查询，但所有被更新的数据都会被忽略(出于让linq to sql变更追踪机制能够正确工作的考虑)。但是后续的查询会得到被添加到数据库的新纪录里面

###### Linq to Sql 和数据库组件

* 1. 返回IEnumerable<T>所有的数据将从数据库获取然后再过滤
  2. 返回IQueryable<T>将会使用一个直接的Sql查询过滤数据
  3. IQueryable<T>扩展了IEnumerable<T>，IEnumerable<T>等效于AsEnumerable<T>方法。如果web页面基于返回的对象使用Linq表达式，所有数据豆浆从数据库获取然后再过滤。但是对于IQueryable<T>，将会使用一个直接的Sql查询过滤数据

###### 选取单个记录和值

Single() 或者SingleOrDefault()。前者为空会报错，后者为空引用类型则为null，值类型则为0.多于一条数据的会报错

###### 13.3.6自动生成数据类

###### 13.3.7关系

* 1. Strorage：让Linq to sql忽略属性设置代码而直接访问对象的私有字段，可以利用这个避开代码验证
  2. OtherKey：外键
  3. ThisKey：主键
  4. DataContext.DeferredLoadingEnabled设置来禁用延迟加载
  5. 让linq to sql立即加载相关的数据:在dataContext执行查询前设置DataLoadOptions属性

DataContext = new DataContext(connectionString);

DataLoadOptions loadOptions = new DataLoadOptions();

loadOptions.LoadWith<Customer>(customer=>customer.Orders);

dataContext.LoadOptions = loadOptions;

###### 13.3.8为存储过程生成方法

* 1. ExecuteMethodCall()调用存储过程方法
  2. 返回一个IExecuteResult<T>对象。其有2个属性：ReturnValue(存储过程返回值)，GetParameterValue(存储过程输出参数值)
  3. ExecuteCommand()、ExecuteQuery<T>执行Sql语句，差别是或者会返回记录
  4. IExecuteResult可以转换为ISingleResult，同样提供ReturnValue属性，但是没有GetParameterValue()方法，但是它扩展了IEnumerable<T>，而不是IQueryable<T>，就是说可以对结果进行枚举，操作也不是以Linq的方式进行的
  5. 调用更新、插入、删除的存储过程和上面类似。生成的方法和存储过程严格匹配，具有相同的参数和返回值类型

###### 13.3.9提交变更

* 1. 实例管理
     1. DataContext获取记录时，在一个内存表中用主键记录该记录，如果再次读取同一个记录，DataContext会使用已经存在的那个实例
     2. DataContext对任意指定的主键只允许一个内存对象。
     3. 负面效应：Datacontext对同一数据进行多次迭代而Linq to sql执行了查询，Datacontext还是忽略了新的信息
     4. 解决办法：1.创建一个新的Datacontext，它会有自己的变更跟踪2.对重要查询信息调用DataContext.Refresh()
  2. 更新：

1. DataContext.SubmitChange():如果数据不需更新，把DataContext.ObjectTrackingEnabled = false来减小Datacontext负载
   1. 插入和删除

InsertOnSubmit(),DeleteOnSubmit(),DeleteAllSubmit()

##### 13.4 LinqDataSource

###### 13.4.1显示数据

###### 13.4.2获取关联数据

###### 13.4.3编辑数据

###### 13.4.4验证

##### 13.5 总结

### 第14章 XML

##### 14.1什么时候使用xml

1. 需要处理已经保存在xml中的数据（必须和现存一个使用了特定xml应用程序交换数据时）
2. 希望用xml保存数据并为未来的整合做好准备，使用xml存数据，以后将设计第三方应用程序来读取数据
3. 使用依赖于xml的技术。(web服务使用各种建立在xml上的标准)
4. 存储数据时必须决定的两件事情：
   1. 确定数据结构化的方式（逻辑格式）
   2. 确定数据保存的方式（数据的物理存储）
   3. Xml是格式的选择而不是存储的选择，就是说，你决定使用xml格式保存数据，你还要决定xml是要保存到数据字段里还是要插入到一个文件里，或者只是加载到内存中

##### 14 2 xml简介

###### 14.2.1 xml的优点

###### 14.2.2 格式良好的xml

###### 14.2.3 xml命名空间

1. 命名空间：通过明确元素使用的标记语言来消除元素的歧义

###### 14.2.4 xml架构

1. 架构：一个定义你的标记语言规则的格式文档称之为架构

##### 14.3基于流的xml处理

1 System.Xml.

2. 读写方法：XmlTextReader，XmlTextWriter

3. 如果xml文件非常大，一次把整个文档加载到内存里都不实际时，这些类就没有必要了

###### 写xml文件

1. 将XML数据写入文件的两种方式：

1. 可以使用XmlDocument和XDocument类创建文档，结束时把它写入文件（在创建xml文档后还要进行其他的操作，如：搜索、转换、验证，那么内存中构建xml文档就是很好的选择，它还是用线性方式写xml文档的唯一方式，因为它允许你在任意地方插入新的节点）
2. 用XmlTextWriter直接把一个文档写入流，它在你写的时候逐个节点输出数据（为直接写入文件提供了更为简单、性能更好的模型，因为它不是立刻就将整个xml文档保存到内存中）

2. 可以使用XmlDocument、XDocument、XmlTextWriter类创建不保存在文件内的xml数据，这三个类都允许你把信息写入任意的流

3. 当用XmlTextWriter创建xml文件时，将会有和在web应用程序编写任何文件代码一样的限制。（要采取措施（如生成唯一的文件名）确保两个不同的客户端不会同时运行同一段代码并且试图同时编写同一个文件）

4. Formatting和Indentation，允许你设定xml是否按典型的层次进行自动缩进和缩进的空格数

###### 14.3.2读xml文件

1. 读取并导航XML文件内容的方法
   1. 用XmlDocument、XPathNavigator（只读）、XDocument类一次性将文档加载内存中，减少文件被打开的次数
   2. 使用XmlTextReader（基于流的读取器），每次读取文档的一个节点（文件以顺序读取，不能自由的移动到父、子、兄弟节点，要循环已深入xml文档中的元素来读取直到要找的内容）
   3. XmlTextReader减少了内存负担，但是只提供了最少的灵活性，文件以顺序读取，不能够像处理内存中的Xml那样自由的移动到父、子、兄弟节点。每次从流中只能读取一个节点，通常需要多个递归才能深入xml文档找到所需内容
   4. XmlTextReader，必须尽可能快地结束任务并关闭阅读器，它在文件上有一个锁，在知道文档结构时，XmlTextReader为更快更方便地读取xml内容提供了辅助方法

2.

##### 14.4 内存中的xml处理

* 1. 基于流的xml处理，提供最小的负载同时灵活性也最小，而且忽略空格和注释
  2. XmlDocument，是xml的标准化接口，但对事件要求比较多
  3. XPathNavigator，和XmlDocument类似，XPathNavigator在内存中保存整个xml文档，在内存中保存整个xml文档，提供比XmlDocument稍快，更有效的模型，并且还提供了增强的搜索功能，和XmlDocument相比，没又提供修改和保存的功能
  4. XDocument，不能和XmlDocument一起使用

###### 14.4.1 XmlDocument

1. XmlDocument把信息保存为树的结点，节点可以是：一个元素，特性，注释或者元素的一个值，每一个单独的XmlNode对象代表一个节点，XmlDocument将同一层次的XmlNode对象放在XmlNodeList集合中。XmlDocument.ChildNodes获取第一层节点
2. 注意：XmlDocument和用户并发，多个用户共用一个文件系统的时候，如果该文件系统为非共享属性，其他用户访问就会报错
   1. XmlUrlResolver检查提供的文件路径并将其传给XmlDownloadResolver（提供url则执行web请求，提供文件路径则执行打开文件流）
   2. 如果可以使用文件流，则以共享的方式打开文件，多用户调用文件的时候就不会发生并发冲突
   3. 最好的解决办法是缓存得到的xml文件或XmlDocument对象来减少竞争
3. 如果创建一个XmlDocument并用从其他数据源获得的信息填充它，不能使用Load(),应该是LoadXml()

###### 14.4.2 XPathNavigator

1. XPathNavigator是将所有信息加载到内存中并且使用基于游标的方式允许使用MoveToNext()之类的方法在xml数据间移动，且每次只能定位到一个节点
2. XPathNavigator使用一个较小的、更为流线型的节点集合，不支持XmlDeclaration类型的节点

###### 14.4.3 XDocument

1. XDocument是管理内存中XML所有功能的模型，它较前者，更擅长构建xml内容（XmlDocument比较复杂，XmlPathNavigator不支持构建）
2. 如果以非线性的方式生成XML，如：将一系列的元素写进根元素里，然后又在这些元素里添加更多的信息，这就需要XDocument这样的类

##### 14.5搜索xml 内容

XmlDocument：使用GetElementByTagName；XmlNavigator使用XPath；XDocument使用Element()

###### 14.5.1 搜索XmlDocument

1. XmlDocument.GetElementByTagName:特定名称的元素搜索整个文档并返回一个包含所有匹配的XmlNode对象的XmlNodeList
2. XmlElement.GetElementByTagName()：在某个特定元素上搜索xml文档的部分信息

###### 14.5.2 使用XPath搜索XmlDocument

1. 基于条件的过滤筛选数据
2. /子节点搜索/DvdList/DVD选择根元素<DvdList>的所有子元素<DVD>
3. //节点的所有嵌套层,为子节点进行递归搜索 //DVD/Title选择<DVD>元素的所有子元素<Title>
4. @节点的某个特性/DvdList/DVD/@ID从<DVD>元素选取叫做ID的特性
5. \*路径中的任意元素/DvdList/DVD/\* 选择<DVD>的所有子节点
6. |组合多个路径/DvdList/DVD/Title| /DvdList/DVD/Director选择<DVD>元素中的<Title>和<Director>
7. .当前默认节点
8. ..表示父节点
9. []定义一个选择条件,可以测试包含的节点或特性的值。/DvdList/DVD[Title = ‘Forrest Gump’]
10. Starts-with /DvdList/DVD[starts-with(Title,’p’)]
11. Position基于位置获取元素/DvdList/DVD[position()=2]或者/DvdList/DVD[2]
12. Count匹配元素名称的元素个数
13. 子节点：父节点的下一层节点后代节点：在任意层上被包含的节点

###### 14.5.3使用Linq搜索XmlDocument

ElementAfterSelf()、ElementBeforeSelf() 找到兄弟元素

Ancestors()、Descendants()遍历xml层次

##### 14.6 验证xml 内容

读取一段非法的xml文档就会报错

###### 14.6.1 基本架构

1. 用架构验证文档：XmlValidatingReader

###### 14.6.2 验证XmlDocument

###### 14.6.3 使用XmlDocument进行验证

##### 14.7 转换xml 内容

XSL(Extensible Stylesheet Language,可扩充样式表语言)，一种用于创建样式表的基于xml的语言。样式表（也叫做转换）是一种特殊的文档，可以将(在XSLT处理器的帮助下)把你的xml文档转化为其他文档

###### 14.7.1 基本的样式表

###### 14.7.2使用XslCompiledTransform

使用样式表以及XslCompiledTransform可以将xml转换为格式化的html

###### 14.7.3使用xml控件

Xml控件是在页面独立的部分显示xsl转换后的结果

###### 14.7.4使用Linq to Xml 转换 Xml

##### 14.8 xml 数据绑定

a) 是从xml文件获取数据信息，它为其他控件的数据源提供一个XmlDocument对象（非XDocument对象）

1. Xml可以分为无限层，而SqlDataSource返回扁平表的数据

###### 14.8.1 非层次化绑定

XmlDocument实现了IEnumerable接口

缺陷：XmlDocument.Nodes集合只包含第一层结点

###### 14.8.2使用XPath

* 1. 通常绑定XmlNode时，只会显示它的特性值。可以使用XPath数据绑定表达式获得嵌套元素的文本
  2. 最灵活的办法是使用一个定义了XPath数据表达式的模板<%# XPath(“Title”)%>

###### 14.8.3嵌套的网格

使用主从网格形式来显示数据

###### 14.8.4使用TreeView的层次化绑定

TreeView支持内建智能

###### 14.8.5使用XSLT

XmlDataSource为XSL转换提供了类似的内建支持，不是用样式表转换为HTML，而是一种源XML文档转换为另一种便于数据绑定使用的XML结构

###### 14.8.6绑定到来自其他源的Xml内容

通过XmlDataSource.Data属性提供一个XML字符串

###### 14.8.7通过XmlDataSource更新Xml

* 1. 用户提交更新时，代码调用XmlDataSource.GetXmlDocument()方法以XmlDocument对象的形式获得当前Xml内容
  2. 代码找到相应的节点，并用XmlDocument的功能实现变更
  3. 代码调用XmlDataSource.Save()方法提交变更

##### 14.9 xml 和 Ado.net数据集

Ado.net通过非连接的DataSet和DataTable对象支持XML

1. GetXml() 得到以xml格式表现的DataSet中的数据
2. WriteXml() 把DataSet中的内容写入文件或者TextWriter、XmlWriter、或者Stream对象
3. ReadXml() 从文件或者TextWriter、XmlWriter、或者Stream对象读取xml数据
4. GetXmlSchema() 以字符形式得到DataSet XML的XML架构，不返回数据
5. WriteXmlSchema() 只把描述DataSet结构的XML架构写入文件或者TextWriter、XmlWriter、或者Stream对象
6. ReadXmlSchema() 从文件或TextReader、XmlWriter、或者Stream对象读取XML架构，并用它配置DataSet结构
7. InferXmlSchema() 从文件或TextReader、XmlWriter、或者Stream对象读取DataSet内容的XML文档，并用它推断DataSet应用的架构。可以替换ReadXmlSchema()方法，但它并不保证保存所有的数据类型的信息

###### 14.9.1把DataSet转化为Xml

###### 14.9.2把DataSet 作为Xml访问

##### 14.10总结

# 构建Asp.net网站

### 第15章用户控件

1. 用户控件：是一小段页面，可以包含静态的html代码和web服务器控件（一旦创建就可以在多个页面重用，还可以加入自己的属性、事件、方法）
2. 自定义服务器控件：是被编译的类，通过编程产生自己的html

##### 15.1用户控件基础

用户控件和web页面之间的区别：

* + 1. 控件指令：UserControl:ontrol，WebPage:age
    2. 扩展名：UserControl：.ascx，继承自System.Web.UI.UserControl；WebPage：aspx，继承自UI.WebPage
    3. 请求状态：UserControl不能被客户端直接请求

###### 15.1.1创建简单的用户控件

在父页面使用UserControl：<%@Register src=”....”%>

###### 15.1.2把页面转换成用户控件

* 1. 删除所有的<html><body><head><form>标签
  2. 将Page指令改为Control指令，并且删除Control指令不支持的特性
  3. 如果使用代码隐藏模型，要在Control指令里包含ClassName特性
  4. 改扩展名

##### 15.2为用户控件添加代码

###### 15.2.1处理事件

UserControl方法必须为public的，载体页面才能触发该UserControl页面的事件

###### 15.2.2添加属性

载体页面设置属性：

在代码里面通过用户控件来调用属性设置

###### 15.2.3使用自定义对象

就是将对象作为UserControl自己的内置对象，参数来处理

###### 15.2.4添加事件

* 1. 方法和属性：用户控件响应网页代码带来的变化
  2. 事件：用户控件通知网页发生了某个活动，然后网页代码做出响应
  3. protectedvoid **gridLinkList\_RowCommand(**object **sender,** GridViewCommandEventArgs **e)**sender是触发事件源对象，e是事件源对象具体触发的事件对象

###### 15.2.5暴露内部Web控件

* 1. 用户控件包含的控件只能被用户控件自身访问，载体页面不能接收包含控件的事件，不能够设置它的属性，也不能调用这些控件的方法
  2. 需要暴露很少属性的时候：

定义公开属性，利用变量改变UC内的属性

Public Color ForeColor

{

Get{return inkTime.ForeColor;}

Set{inkTime.ForeColor = value;}

}

* 1. 很多属性的时候：

定义公开控件，将控件作为变量进行传递

Public LinkButton InnerLink

{

Get{retun lnkTime;}

}

##### 15.3动态加载用户控件

1. 在Page.Load() 事件中加入用户控件
2. 使用容器控件或PlaceHolder控件来确定用户控件出现的位置
3. 设置ID属性来给用户控件一个唯一的名称。然后通过Page.FindControl()方法获取对控件的引用

##### 15.4缓存部分页面

* 1. 缓存用户控件之后，它本质上变成了一段静态的Html，这样网页代码将不可以访问用户控件对象。Asp.Net只是根据用户控件创建一个或多个通用的对象类型。
  2. 声明的方式（在网页上添加一个用户标签）：就会添加一个StaticPartialCachingControl
  3. 编程创建（LoadControl）：PartialCachingControl

###### 15.4.1 VaryByControl

<uc1:VaryByControlID="VaryByControl1"runat="server"/>

###### 15.4.2共享缓存控件

<uc1:VaryByControlID="VaryByControl1"runat="server"Shared=”True”/>

##### 15.5 总结

### 第16章主题和母版页

##### 16.1 层叠样式表

###### 16.1.1创建Css

###### 16.1.2应用样式表规则

<link href = “StyleSheet.css” rel = “stylesheet” type = “text/css”

1. Web控件：CssClass
2. Html元素：class
3. 一段HTML内容（div,span）：class

##### 16.2 主题

主题和Css样式表区别：

* 1. 主题基于控件而不是HTML
  2. 主题应用在服务器上
  3. 可以通过配置文件来应用主题
  4. 主题不能像CSS那样级联

###### 16.2.1 主题文件夹和外观

###### 16.2.2应用简单主题

###### 16.2.3 处理主题冲突

* 1. 如果控件属性和主题定义的属性产生冲突，则优先使用主题定义的属性
  2. 优先使用控件属性，则覆盖某些特定的细节从而优化主题=》SheetSheetTheme取代Theme，对整个页面控件有效
  3. 配置特定控件从而忽略主题=》EnableTheming = false

###### 16.2.4 为控件创建多个外观

<asp:ImageButtonrunat="server"ID="ibtnOK"SkinID="OKButton"/>

为每一个控件选择你要应用的ShinID

###### 16.2.5 含有模板和图片的外观

<asp:ImageButtonrunat="server"SkinID="CancelButton"ImageUrl="buttonCancel.jpg"/>

保证图片的url相同于主题所在文件夹，而不是页面所在文件夹

###### 16.2.6在主题里使用Css

* 1. 希望格式化那些不和服务器控件对应的html元素
  2. 倾向于使用样式表
  3. 已经有很完善的样式表了，没有必要去创建主题来实现同样的格式化
  4. 在主题中使用样式表：
     1. 把样式表放在主题文件夹
     2. 将head标签变成服务器控件

###### 16.2.7通过配置文件来应用主题

<configuration>

<system.web>

<pages theme(styleSheetTheme)= “Funky”>

</system.web>

</configuration>

###### 16.2.8动态应用主题

根据用户角色来配置页面外观

Page.Theme = Session["Theme"**].ToString();**

##### 16.3 标准化网站布局

* 1. 用户控件：允许你定义一个小页面（不能解决页面布局问题）
  2. HTML框架：在一个浏览器窗口中同时显示多个页面的HTML的基本工具（它里面的每个页面不得不单独请求服务器，一个框架中的页面不能和其他框架中的页面相互通信）
  3. 母版页：用于标准化web页面布局

##### 16.4 母版页基础

###### 16.4.1简单的母版页

只有在母版页中才能使用ContentPlaceHolder

###### 16.4.2简单的内容页

带有母版页的事件发生顺序

* 1. 创建母版页控件
  2. 添加内容页的子控件
  3. 触发母版页的Page.Init事件
  4. 内容页的Page.Init事件

###### 16.4.3默认内容

###### 16.4.4具有表格和Css布局的母版页

1. Html是基于流的布局，随着内容的增加，页面会被重新组织，其他内容会被挤到一边
2. 解决办法：使用HTML表格或者CSS定位来控制布局
3. <Http://www.csszengarden>和<http://www.bluerobot.com/web/layouts>基于css的布局
4. 使用表格：
   1. 将不同部分分解到行和列里，然后将ContentPlaceHolder加入到某个单元格里
   2. 使用css定位，就是将内容放入<div>标签，然后使用绝对坐标控制<div>位置或者把它们浮动到页面的某一边，然后ContentPlaceHolder放入<div>中
   3. Position、left、right、top、bottom

###### 16.4.5母版页和相对路径

* 1. 母版页不像主题那样，会自动将主题文件路径解析正确
  2. 解决办法：将各种涉及到路径的标签转变为web控件，并用”~”作为URL的开头
  3. <img src=”~/MasterPages/banner.jpg runat=”server”>

###### 16.4.6通过配置文件应用母版页

<configuration>

<system.web>

<pages masterPageFile = “SiteTemplate.master”>

</system.web>

</configuration>

页面MasterPageFile优先于web.config

##### 16.5 高级母版页

###### 16.5.1和母版页类交互

* 1. 母版页解决问题：是在所有运用母版页页面精确复制一些东西还是在每个页面改变这些东西
  2. 允许内容页和母版页进行交互：
     1. 在母版页类里添加公有的属性和方法
     2. 在内容页中加入MasterType指令，在指令里只需指定.master的虚拟路径即可

###### 16.5.2动态设置母版页

设置Page.MasterPageFile。必须在Page.Init阶段完成，只有所有需要动态设置的母版页里包含相同的占位符

###### 16.5.3嵌套母版页

##### 16.6 总结

###### 16.6.1 在内容页面中编写代码访问母版页中的控件

在内容页面中有个**Master**对象，它是MasterPage类型，它代表当前内容页面的母版页。通过这个对象的**FindControl**方法，我们可以找到母版面中的控件，这样就可以在内容页面中操作母版页中的控件了。

TextBox txt = (TextBox)((MasterPage)Master).FindControl("txtMaster");  
        txt.Text = this.txtContent1.Text;

###### 16.6.2 在内容页面中编写代码访问母版页中的属性和方法

* 1. 可能通过Master对象进行访问，只不过在这里要把Master对象转换成具体的母版页类型，然后再调用母版页中的属性和方不法
  2. 说明:**母版页中要被内容页面调用的属性和方法必须是Public修改的。否则无法调到**
  3. **代码：**

**public string TextValue**        {  
            get  
            {  
                return this.txtMaster.Text;  
            }  
            set  
            {  
                this.txtMaster.Text = value;  
            }  
        }  
         **public void show(string str)**        {  
            txtMaster.Text = str;  
        }

* 1. **((MasterPage\_MP)Master).**show(this.txtContent1.Text);

**((MasterPage\_MP)Master).TextValue** = this.txtContent1.Text;

###### 16.6.3在母版页中访问内容页面的控件：

1. 在母版页中可以通过在**ContentPlaceHolder**控件中调用**FindControl**方法来取得控件，然后对控件进行操作。
2. ((TextBox)this.ContentPlaceHolder1.FindControl("txtContent1")).Text = this.txtMaster.Text;

###### 16.6.4在母版页中访问内容页面中的方法和属性：

* + 1. 在母版页中调用子页面中的属性和方法有点难度，因为我们无法像上一步中那样通过**FindControl**来找到方法和属性。
    2. C#是的“反射”，它可以使我们动态获取页面对象，并且可以调用它的属性和方法。
    3. Type t = this.ContentPlaceHolder1.Page.GetType();  
        PropertyInfo pi = t.GetProperty("ContentValue"); //获取ContentValue属性  
        pi.SetValue(this.ContentPlaceHolder1.Page,this.txtMaster.Text,null);    //给属性赋值  
        MethodInfo mi = t.GetMethod("SetValue"); //获取SetValue()方法  
        object[] os = new object[1];   //建造输入参数  
        os[0] = txtMaster.Text;  
        mi.Invoke(this.ContentPlaceHolder1.Page, os);    //调用SetValue方法
    4. **在有多个内容页面使用母版面的情况下，在母版页中根据不同的内容页面实现不同的操作**
    5. string s = this.ContentPlaceHolder1.Page.GetType().ToString();   //取出内容页面的类型名称  
       if (s == "ASP.default17\_aspx")   //根据不同的内容页面类型执行不同的操作  
       {  
              ((TextBox)this.ContentPlaceHolder1.FindControl("TextBox2")).Text = "MastPage";  
       }  
       else if (s == "ASP.default18\_aspx")  
       {  
              ((TextBox)this.ContentPlaceHolder1.FindControl("TextBox2")).Text = "Hello MastPage";  
       }

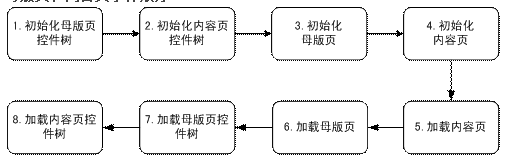
###### 16.6.5母版页运行机制

1. 母版页仅仅是一个页面模板，单独的母版页是不能被用户所访问的。单独的内容页也不能够使用。母版页和内容页有着严格对应关系。母版页中包含多少个ContentPlaceHolder控件，那么内容页中也必须设置与其相对应的Content控件。当客户端浏览器向服务器发出请求，要求浏览某个内容页面时，ASP.NET引擎将同时执行内容页和母版页的代码，并将最终结果发送给客户端浏览器。

###### 16.6.6母版页和内容页的运行过程可以概括为以下5个步骤。

1. 用户通过键入内容页的URL来请求某页。
2. 获取内容页后，读取@ Page指令。如果该指令引用一个母版页，则也读取该母版页。如果是第一次请求这两个页，则两个页都要进行编译。
3. 母版页合并到内容页的控件树中
4. 各个Content控件的内容合并到母版页中相应的ContentPlaceHolder控件中
5. 呈现得到结果页。

###### 16.6.7母版页和内容页事件顺序



（1）母版页中控件Init事件；  
（2）内容页中Content控件Init事件；  
（3）母版页Init事件；  
（4）内容页Init事件；  
（5）内容页Load事件；  
（6）母版页Load事件；  
（7）内容页中Content控件Load事件；  
（8）内容页PreRender事件；  
（9）母版页PreRender事件；  
（10）母版页控件PreRender事件。  
（11）内容页中Content控件PreRender事件。

###### 16.6.8嵌套母版页

###### 16.6.9嵌套母版页

### 第17章网站导航

1. MultiView控件和Wizard控件：将若干个步骤浓缩到一个页面里，简化导航
2. 站点地图模型：定义网站的导航结构并直接绑定到富控件
3. 富导航控件：包括TreeView和Menu

##### 17.1多视图页面

将原本几个不同页面里的代码放到同一个页面里

###### 17.1.1 MultiView控件

* 1. 提示：可以通过编程添加视图（和其他所有控件一样），实例化一个新的试图对象并通过Views集合的Add或者AddAt()把它加入到MultiView
  2. MultiView可识别的命名名称：
     1. Button CommandName MultiView Attribute CommandArgument
     2. PrevView PreviousViewCommandName 移动到前一视图
     3. NextView NextViewCommandName 下一视图
     4. SwitchViewByID SwitchViewByIDCommandName 移动到视图ID的视图
     5. SwitchViewByIndex SwitchViewByIndexCommandName 移动到视图Index的视图
  3. MultiView创建的页面比普通页面负载大，所有视图上的控件都在每次回发时创建并持久化到视图状态中
  4. MultiView使用的理想场所：处理一组扩张的输入控件，就不需要频繁使用滚动条了

###### 17.1.2 Wizard控件

1. 和MultiView类似，同样支持显示几个视图中的一个，但是它还包含一系列可定制的内建行为。包括导航按钮、一个带有分步链接的侧栏、样式和模板
2. 支持非线性导航，即可以省略某一步骤
3. 控件总会在视图状态中被实例化并持久化，如果要简化复杂的向导，可以将步骤分解到几个单独的页面，使用Server.Transfer()跳转即可
4. 向导事件：
   1. ActiveStepChanged 控件切换到一个新的步骤时发生，无论是页面还是代码（逐步提交）
   2. CancelButtonClick Cancel按钮被触发时发生
   3. FinishButtonClick （最后提交）
   4. NextButtonClick和PreviousButtonClick
   5. SideBarButtonClick 侧栏区域里的按钮被单击时发生
5. 向导样式：
   1. ControlStyle 作用于Wizard控件的所有区域
   2. HeaderStyle 标题区域，只在你设置了HeaderText属性后看见
   3. SideBarStyle 侧栏区域
   4. SideBarButton 侧栏区域中的按钮
   5. StepStyle 定义步骤内容的区域
   6. NavigationButtonStyle 导航区域的按钮
   7. StartNextButtonStyle 第一个步骤里的“下一步”导航按钮（StepType为Start时）
   8. StepNextButtonStyle StepType = step
   9. StepPreviousButtonStyle 前一步导航按钮 StepType= step
   10. FinishPreviousButtonStyle 最后一个步骤的“前一步”导航按钮（StepType = Finish）
   11. CancelButtonStyle Wizard.DisplayCancelButton设为真，它作用于“取消”按钮
6. 向导模板
   1. HeaderTemplate 标题区域
   2. SideBarTemplate 侧栏区域，通常包含每个步骤的连接
   3. StartNavigationTemplate 第一个步骤的导航按钮
   4. StepNavigationTemplate 中间步骤的导航按钮
   5. FinishNavigationTemplate 最后一个步骤的导航按钮

##### 17.2 站点地图

一个网站有很多页面，可能就需要某个导航系统来帮助用户从一个页面跳转到另一个页面

###### 17.2.1 定义站点地图

* 1. 使用asp.net的站点提供程序，.sitemap文件必须放在根目录下
  2. 如果是一个根目录的节点，可以不设置url
  3. url路径：url="~/第15章：用户控件/LinkTable.aspx">

###### 17.2.2 绑定站点地图

<asp:SiteMapDataSourceID="SiteMapDataSource1"runat="server"/>

<asp:TreeViewID="TreeView1"runat="server"DataSourceID="SiteMapDataSource1">

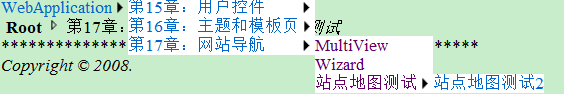
</asp:TreeView>

###### 17.2.3 导航路径

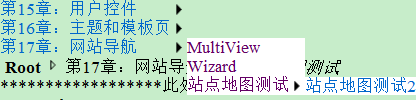
* 1. Asp.Net导航控件：TreeView、Menu、SiteMapPath
  2. 只有SiteMapPath提供导航路径：显示用户当前位置，并允许用户链接回到更高的层级
  3. TreeView和Menu从SiteMapDataSource获取数据源，SiteMapPath则不是
  4. SiteMapPath在告知用户当前位置以及提供用户在层次结构间向上跳转上很有用，不过必须和其他导航控件一起使用
  5. SiteMapPath与外观相关属性：
     1. ShowToolTips bool
     2. ParentLevelDisplayed int
     3. RenderCurrentNodeLink bool
     4. PathDirection string
     5. PathSeparator string
  6. SiteMapPath样式和模板
     1. NodeStyle NodeTemplate 路径出于根节点和当前节点的所有部分
     2. CurrentNodeStyle CurrentNodeTemplate 代表当前页面的节点
     3. RootNodeStyle RootNodeTemplate 代表根的节点
     4. PathSeparatorStyle PathSeparatorTemplate 每个节点间的分隔符

###### 17.2.4 显示站点地图的一部分

* 1. 1,2,3,4,5（当前节点是站点地图测试）



* 1. 跳过根节点 ：从2开始ShowStartingNode="false"



* 1. 从当前节点开始：从此时节点开始，现在是1StartFromCurrentNode="true"



* 1. 从指定节点开始：
     1. StartingNodeUrl="~/第17章：网站导航/站点地图测试.aspx"



* + 1. StartingNodeOffset="1"



###### 17.2.5 站点地图对象

* 1. 使用编程导航：
     1. 修改页面的显示：可以获取当前节点的信息并用它配置页头和标题之类的细节
     2. 实现不同的导航逻辑：只显示当前页面的部分子节点，或者希望创建”前一页/下一页导航”按钮
  2. 入口：
     1. SiteMap类：CurrentNode，RootNode属性=》返回SiteMapNode对象，通过该对象可以获取来自站点地图的信息，包括标题，描述，URL值
     2. SiteMapNode属性：
        1. ParentNode 包含当前节点的导航结构中的上一层节点。如果当前节点是根节点，则返回空引用
        2. ChildNodes 返回所有子节点集合。HasChildNodes属性是否有子节点
        3. PreviousSibling 返回之前的兄弟节点，没有返回空引用
        4. NextSibling 返回之后的兄弟节点，没有返回空引用

###### 17.2.6 加入自定义站点地图信息

在站点地图中添加除了模板定义之外的节点信息

###### 17.2.7 创建自定义的SiteMapProvider

1. 创建自定义的SiteMapProvider:
   * 1. 需要将站点地图信息保存在不同的数据源(如关系数据库)中
     2. 按不同的架构保存站点地图信息，和Asp.Net预期的xml格式不同，这种情况最可能出现在你恰好有一个系统用来保存站点地图
     3. 需要一个高度灵活性的站点地图，随时可以进行创建，可能希望按当前登陆用户角色来显示不同的站点地图信息
     4. 需要改变一个XmlSiteMapProvider实现中的一个限制，可能希望具有重复URL的节点

b） StaticMapProvider:使用不同数据源而保持相同的逻辑结构

站点地图提供程序必须实现抽象类SiteMapProvider

c） 将站点地图保存到数据库

d） 创建站点地图提供程序

* 1. 第一步：覆盖Initialize（）方法从web.config文件读取所需的全部信息
  2. 为站点提供程序配置元素：
     1. 数据库（站点地图数据存储在这里）的连接字符串
     2. 返回站点地图的存储过程的名字
     3. 数据库的提供程序，它将允许你使用提供程序无关的代码

###### 17.2.8 URL映射

<urlMappings>

<addurl="~/Category.aspx"mappedUrl="~/Default.aspx?category=default"/>

<addurl="~/Software.aspx"mappedUrl="~/Default.aspx?category=software"/>

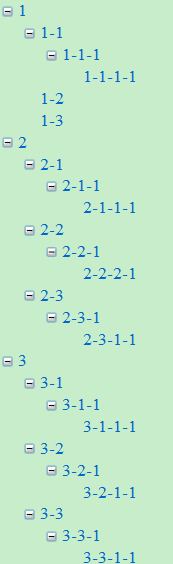
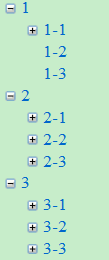
</urlMappings>

* 1. 重定向的实现和Server.Transfer一样
  2. Request.Path和Request.Querystring属性反映新的（mappedUrl）URL
  3. Request.RawUrl属性返回被请求的原始的友好URL(url)
  4. 站点地图先尝试匹配请求的url（由request.）

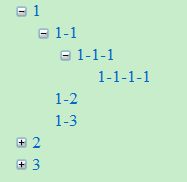
###### 17.2.9安全调整

使用安全调整，所有用户不允许访问的页面（基于授权规则）都不会出现在站点地图中

##### 17.3 TreeView 控件

* 1. 通过编程添加TreeNode:
     1. TreeNode **tn =** newTreeNode**(**"4"**,** "04"**);**
     2. tv.Nodes[0].ChildNodes.Add(tn);
  2. 原始3\*3的TreeView
     1. 
  3. 控制显示层数：
     1. tv.ExpandDepth =1;只有前2层会被显示，其余会折叠起来
     2. 
  4. 控制总共包含多少层：
  5. 打开和展开节点：
     1. tv.Nodes[1].Expanded = false;

tv.Nodes[2].Expanded = false;

* + 1. 

###### 17.3.1 TreeNode

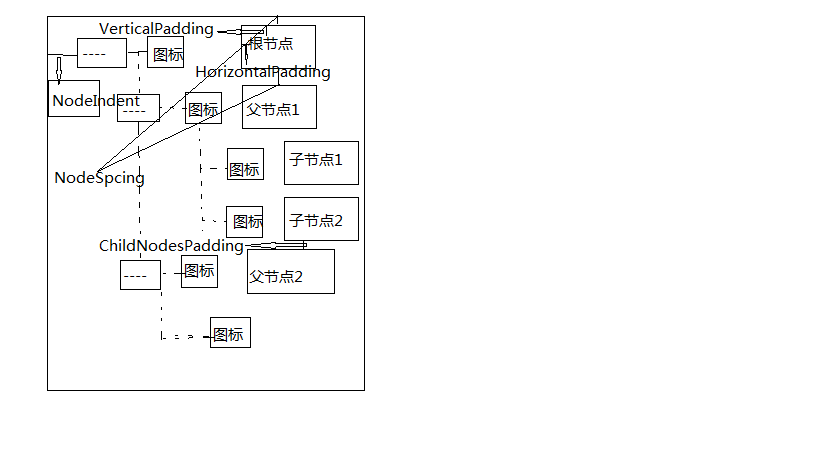
* 1. 两种模式：
     1. 选择模式：会回发并引发TreeView.SelectedNodeChanged事件（默认模式）
     2. 导航模式：会导航到一个新的页面。只要把TreeNode的NavigateUrl设为非空字符串（sitemap）即是该模式。不会引发选择事件

###### 17.3.2按需填充站点

* 1. 操作方式：只需要将最后时刻想要填入的内容的TreeNode的PopulateOnDemand =true即可
  2. 展开分支：引发TreeNodePopulate事件，在该事件中可以加入下一层节点
  3. 当TreeView.PopulateNodesFromClient=true(默认)，TreeView执行一个客户端的回调来从事件中获取它所需要的节点
  4. 当客户端浏览器不支持回调，无论PopulateNodesFromClient为何，TreeView会触发一次正常的回发并获得相同的结果，浏览器会刷新

###### 17.3.3 TreeView样式

* 1. 类：TreeNodeStyle
  2. 属性：
     1. ImageUrl 节点旁显示的图片的URL
     2. NodeSpacing 当前节点和相邻节点的垂直间距(像素为单位
     3. VerticalPadding 节点文字和文字周围边界的上下间距（像素）
     4. horizontalPadding 节点文字和文字周围边界的左右间距（像素）
     5. ChildNodesPadding展开的父节点的最后一个子节点和其下一个兄弟节点的间距(像素)
     6. NodeIndent 设置树结构里各个子层级间缩进（从左起）的像素
     7. ShowExpandCollapse关闭书中节点列
     8. CollapseImageUrl TreeView折叠和展开的指示器（通常用+号和-号）
     9. ExpandImageUrl
     10. NoExpandImageUrl设置没有子节点的节点旁边显示的内容
     11. ShowCheckBox 是否显示复选框



* 1. TreeView样式属性
     1. NodeStyle 应用到所有子节点
     2. RootNodeStyle 仅应用到第一层（根）节点
     3. ParentNodeStyle 包含其他节点的节点，不包含根节点
     4. LeafNodeStyle 包含其他节点且不是根节点的节点
     5. SelectedNodeStyle 当前选中的节点
     6. HoverNodeStyle 鼠标停留
  2. 如果不希望某个节点被选中，TreeNode.SelectAction = None
  3. 将所有的TreeNode放到<LevelStyles>中，它会根据元素位置判断层级
  4. 设置图片
     1. TreeViewNode.ImageUrl为单个节点设置图片
     2. CollapseImageUrl收合的节点
     3. ExpandImageUrl展开的节点
     4. NoExpandImageUrl没有子节点而不能展开的节点
     5. TreeViewImageSet，TreeView自带图片

##### 17.4 Menu控件

###### 17.4.1 Menu样式

1. MenuItem属性：
   1. Text 显示的文字
   2. ToolTip 鼠标停留时显示的文字
   3. Value 值
   4. NavigatorUrl 如果设置，自动链接到该Url，否则需要响应Menu.MenuItemClick事件确定要执行的活动
   5. Target 默认当前窗体打开
   6. Selectable 是否可选
   7. ImageUrl 显示在菜单项旁边的图片（在文字的右边）
   8. PopOutImageUrl 菜单项包含子项时显示在菜单旁边的图片（在右边），默认是一个实心箭头
   9. SeparatorImageUrl 菜单项下面显示的图片，用于分隔下一个菜单项
   10. Orientation menu是水平还是垂直布局
2. Menu的Items属性有一个MenuItem对象的集合，每一个MenuItem有一个包含嵌套项的ChildItem集合

###### 17.4.2 Menu模板

StaticMenuItemTemplate

DynamicMenuItemTemplate

##### 17.5 总结

### 第18章网站部署

##### 18.1 IIS

* 1. IIS本质：是一个负责处理特定端口请求的windows服务
  2. IIS5.0服务自身监听端口以接收基于HTTP/HTTPS的请求，服务亲自或者通过被称为ISAPI扩展的DLL请求
  3. IIS6.0:通过修改web服务器的处理模型提升稳定性和web服务器的安全性：请求处理呗分割到几个组件中，然后把这些请求传送给万维网发布服务管理的工作进程。每个工作进程都负责处理从各个端口接收到的请求并特别处理他们配置的那一部分，优点：每个工作进程都相互独立运行，可以再web应用程序间实现某种程度上的隔离
  4. IIS7.0：扩展为允许Web服务器接收任何端口上除了HTTP和TCP/IP之外的任何协议的请求：
     1. 使得IIS称为所有需要使服务通过一个或者多个端口和协议在网络上可用的组件的第一流宿主
     2. Web服务器模块化，允许在请求处理的整体生命周期上进行更好的扩展并能够更好地和ASP.NET结合

18.1.1 IIS站点和虚拟目录

* + - * 1. 在所有版本的IIS里，都是万维网发布服务负责管理和运行站点以及它们的web应用程序
        2. 配置站点注册端口上所有进入的请求都通过Windows网络堆栈传送给IIS（其实是万维网发布服务），它处理请求并再次通过Windows网络堆栈把响应传送给客户端
        3. 虚拟目录的创建只是为了暴露web服务器站点子应用程序的内容
        4. 两种类型的虚拟目录：

通过从父站点继承配置、行为(安全、执行等)和状态向外部暴露的简单目录

具有自己的配置、行为和状态且配置为独立应用程序的虚拟目录

18.1.2 IIS管理控制台和IIS配置

* 1. IIS5.0、6.0里：全部的配置保存在本地系统一个叫做metabase的文件里，5.0为二进制的数据库，6.0则是xml数据存储
  2. IIS7.0：Web服务器配置设置的默认值保存在一个基于XML的中央配置里，而特定站点和虚拟目录的设置则保存在站点或虚拟目录的设置则保存在站点或虚拟目录中web.config文件单独的配置节<system.webServer>里，这里也是ASP.NET自身保存配置的配置文件

18.1.3将网址、虚拟目录和文件映射到URL

* 1. 传统URL格式：<http://192.168.1.107/第17>章：网站导航/站点地图测试.aspx
  2. 回环地址：127.0.0.1（localhost）

18.1.4深入IIS架构

额请求处理架构

* + 1. IIS根据请求URL里传入的文件扩展名决定如何处理请求
    2. IIS决定自己如何处理一个请求后，然后开始真正处理请求，运行于操作系统中一个由IIS管理的进程里
    3. IIS5.0,6.0里的请求处理：
       1. 所有文件名都是通过ISAPI的扩展在IIS配置里注册(ISAPI是负责处理URL中具有特定文件扩展名的资源请求的DLL)
       2. 没有和IIS关联的文件名由Web服务器直接处理并在没有任何另外处理的情况下返回给客户端
       3. ISAPI并不处理任何请求，它只是将请求交给ASP.NET运行时，ISAPI完全基于.NET平台并且由托管代码编写
          1. ISAPI处理步骤：

请求经过特定阶段(验证、授权)

请求通过ASP.NET，ISAPI扩展将请求从web服务器(可以是任意的应用程序，可以是VS或者服务器)转送到ASP.NET运行时(ASP.NET处理完全和web服务器独立的请求)

ASP.NET运行时处理结束之后将自己的结果返回给ISAPI扩展

ISAPI将处理结果返回给IIS

IIS将结果返回给客户端

* + - 1. CGI是最早能够实现动态web应用程序的一种方式，很多语言(Perl、PHP)使用这种请求模型
         1. 缺点：它为每个进入的请求请求一个独立的进程，它至少在Windows上是一个数量巨大、资源集中且非常慢的操作
      2. ASP.NET自身的运行和IIS并不是独立的
         1. 缺点：

ASP.NET不能参与整体请求的最初阶段：某些模块（验证模块）需要被执行2次，第一次为IIS实现然后为ASP.NET（为什么ASP.NET提供自己的验证模块，为了从之前的请求处理步骤中抓取已经由IIS验证的Windows身份），不能通过ASP.NET实现某些事情，因为它并没有参与请求的所有阶段

请求处理的类型是一体的：CGI是直接整合到IIS的，无论你是否使用它们，它们都是IIS的一部分，都会占用一定的资源

* + 1. IIS7.0的请求处理
       1. Web服务器的模块化
       2. 统一、更透明和灵活的配置系统
       3. ASP.NET和IIS的更好整合
       4. 对于大多数Web宿主场景的整合支持
       5. IIS天生基于HttpModules和HttpHandles的概念构建。
          1. 模块提供通用目的功能(验证、授权、日志、请求压缩)
          2. 而处理程序和文件扩展名关联并因此负责处理请求的URl中它们所配置的文件扩展名的请求
          3. 每个模块和处理程序在一个单独的dll里实现，该DLL宿主于负责处理请求的进程中，每个文件名与一个负责处理特定文件名的处理程序相关联
          4. 对于需要在所有请求执行的通用功能，IIS7.0支持Http模块
          5. ISAPI在7.0里正常工作，它们通过脚本和出路程序映射配置
          6. 优势：

让你能够运行确实需要的模块和处理程序（其他的根本没有加载），7.0细化安装选项，既减少了工作进程需要的资源，也减少了web服务器被攻击的可能性

允许你在请求处理生命期的各个阶段扩展web服务器，只要编写自定义的模块或处理程序并通过IIS管理控制台注册他即可

请求模型同时包含了由C/C++实现的原生处理模块和处理程序，由.NET语言实现的托管模块和处理程序，该类实现一个IHttpModule，IHttpHandle，可以自定义Web服务器的请求处理管道

ASP.NET整合模式中，可以随时把基于ASP.NET的模块整合到宿主于IIS7.0和其他web应用程序

处理请求的处理模型

* + - * 1. IIS5.0：

请求通过ISAPI扩展处理，ISAPI扩展直接加载到万维网发布服务，它负责处理IIS5.0的全部请求(aspnet\_isapi.exe)

ASP.NET ISAPI只是一个分发者，它并不处理请求，它只是接受请求，将请求转发给ASP.NET运行时，ASP.NET运行时完全运行在CLR的上下文中(aspnet\_wp.exe)

在工作进程内，ASP.NET应用程序由AppDomains隔离，这样一个web应用程序不会再无意间破坏另一个web应用程序的内存。如果aspnet\_wp.exe没有运行，aspnet\_isapi.exe会自动加载它

ISAPI隔离级别设置为“高”，IIS就在dllhost.exe进程中执行ISAPI应用程序

应用程序域：是CLR为了安全和可靠性而而隔离运行在同一个宿主进程中的.NET组件的一种机制

dllhost.exe进程提供了一个完全可配置的进程模型在machine.config文件的<processModel>节点中

缺点：

如果直接运行在web服务器进程的应用程序崩溃，会导致整个应用程序崩溃（IIS加入了隔离级别配置并在单独的京城里执行web应用程序），这一行为默认没有应用到ASP.NET应用程序，它们运行于外部aspnet\_wp.exe进程。只影响传统的ASP应用程序以及所有其他类型的ISAPI扩展（PHP、Perl或者自定义的C++ ISAPI扩展）

核心模式的网络栈与用户模式的web服务器之间的切换、Web服务器进程和外部进程的dllhost.exe或asp.net\_wp.exe工作进程之间的切换，很消耗性能

* + - * 1. IIS6.0:

一个叫做HTTP.SYS的核心模式驱动，它负责从客户端接收HTTP请求

核心模式(HTTPS.SYS)把请求转发给所有将自身注册到特定URL的进程（所有注册核心模式驱动的应用程序都可以接收HTTP请求而不必运行整个WEB服务器）

IIS采用核心模式驱动的架构运行所谓的W3C工作进程（w3wp.exe），每个工作进程运行一个或者多个应用程序

隔离机制：应用程序运行于独立进程中，一个程序的崩溃完全不会影响到其他应用程序的运行

WAS：（Web服务进程管理服务）=>Windows进程激活服务，不仅仅用于基于HTTP的请求，它监视工作进程之间的活动。WAS会自动重启失败的进程

Web应用程序（虚拟目录）被分配到应用程序池（取代IIS5.0的隔离级别）

在Machine.config<processModel>元素配置的配置传统的ASP.NET进程模式是禁止的。都在IIS6.0的工作进程中进行配置

缺点：单独的IIS工作进程实例需要加载单独的CLR实例，会额外消耗内存

* + - * 1. IIS7.0

IIS7.0中，ISAPI并不是主要负责执行请求

IIS的每一个工作进程（应用程序池）运行于2中模式：

新的ASP.NET整合模式：基于模块和处理程序的请求处理，允许托管和原生的模块和处理程序能够参与请求处理的生命期

传统模式：和IIS6.0的处理方式一样

18.1.5安装IIS

##### 18.2管理网站

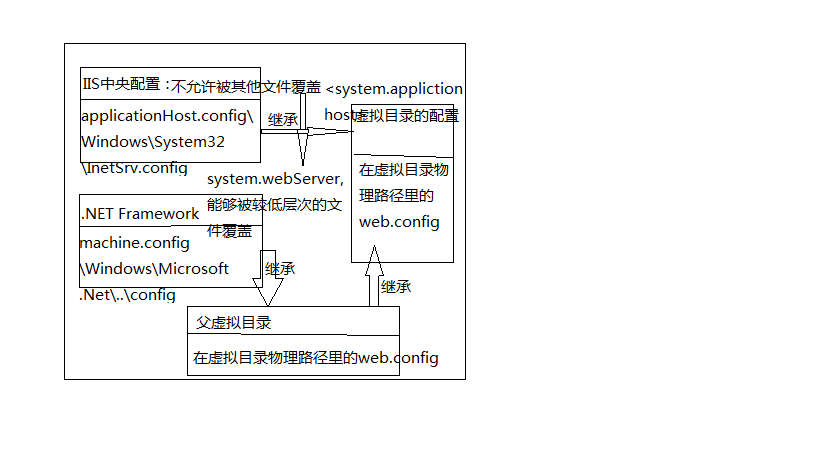
18.2.1 管理虚拟目录以及IIS 5.x 和IIS 6.0 的网站

1. 权限：
   1. 读取。最基本的权限，不能禁止
   2. 运行脚本。禁止则只能访问静态页面类型（有风险）
   3. 执行。允许用户运行普通的可执行文件或CGI应用程序。（安全风险，应该禁止）
   4. 写入。允许用户在WEB服务器上添加、修改、删除文件。（应该总被禁止，用户可以执行上传文危险的可执行文件，或者占尽可用的磁盘空间）
   5. 浏览。允许用户获取虚拟目录中文件的完整列表，包括受限制的内容（通常会禁用浏览）
2. 将自己的文件扩展名映射到ASP.NET运行时
   1. 使用IIS控制台将文件扩展名映射到合适版本的ASP.NET ISAPI
   2. 实现ProcessRequest，在此方法里加入代码处理前面指定的文件扩展名的请求
   3. 在应用程序的web.config文件配置里HTTP处理程序，这样ASP.NET运行时才能够知道这个文件扩展名必须由之前创建的HTTP处理程序处理

18.2.2管理IIS6.0中的应用程序池

* 1. DefaultAppPool：该应用程序池作为网络服务运行，所有在默认网站里的web应用程序都被配置为运行在这个应用程序池里
  2. 创建自己的应用程序池：
     1. 稳定性问题：在单独的应用程序池里运行那些具有稳定性问题的较老的应用程序，这样他们的问题不会影响到其他应用程序了
     2. 内存漏洞：对资源密集的应用程序以及具有内存漏洞的老应用程序应该定期回收资源，就可以创建单独的池配置进程回收
     3. 安全性：web程序有和其他不同程序不同的权限
     4. 管理：在WEB宿主场景下，为不同的（组）用户隔离管理性的应用程序和普通的应用程序
  3. 应用程序池(工作进程)提供了一个根据不同条件（如安全性或可靠性）隔离应用程序的机制
  4. 回收一个应用程序池（工作进程通常意味着停止一个旧进程并未应用程序池创建一个工作进程的新实例
  5. 设计可回收的应用程序池涉及设计Web农场应用程序的某些步骤，使用外部会话状态是设置可回收应用程序的关键，因为会话状态是保存在工作进程的进程空间里
  6. Web应用程序是运行在应用程序池中的（应用程序池实现了web应用程序的隔离，取代了IIS5.0里的隔离模式）
  7. 应用程序池是使用Windows验证方式连接SqlServer的绝佳场所，这比Sql验证安全，不必在web.config文件里保存用户名和密码

18.2.3在IIS 7.0 管理虚拟目录和站点

* 1. 从默认站点继承行为和配置的虚拟目录。右键点击虚拟站点，并从上下文菜单中选择”增加虚拟目录”
  2. 具有自己行为、配置、状态的应用程序，选择“增加应用程序“
  3. 区别：如果虚拟目录时应用程序，则需指定一个应用程序池
  4. 基于IIS的设置也是直接保存在web应用程序虚拟目录的web.config里
  5. 配置：
     1. 
     2. IIS7.0使用相同的管道处理原生以及托管(ASP.NET)的模块和处理程序
     3. 整合模式：IIS7.0使用标准的ASP.NET创建实现ASP.NET创建实现ASP.NET IHttpModule接口的类来扩展IIS
     4. 可以将一个模块配置为站点根级别的模块，那么运行于Web服务器上的所有Web应用程序重用该模块
     5. 当IIS运行于ASP.NET整合模式时，模块能配置为为所有web应用程序执行，甚至那些fie基于ASP.NET的web应用程序也是如此，只需确保该模块对全局可用
        1. 给自定义模块一个强名字
        2. 然后使用gacutil.exe应用程序把它追加到全局程序集缓存(GAC,Global Assembly Cache)里，gacutil.exe随.Net Framework一起发布
        3. 通过IIS配置该模块，”添加模块”对话框里的”类型”属性里使用它的强名字即可

18.2.4在IIS 7.0 管理应用程序池

1. 新建站点：
   * + 1. 指定逻辑名称
       2. 制定本地物理路径
       3. 指定应用程序池
       4. 进行配置绑定(协议、端口、ip地址)

##### 18.3部署你的Asp.net 应用程序

* + 1. 部署ASP.NET WEB应用程序其实只是将应用程序的目录结构拷贝到目标机器并配置环境
    2. 需要额外配置：
       1. 将所有需要的应用程序文件复制到目标机器：（如果使用了通过GAC访问的全局程序集），就必须检查这些程序集是否放入其中。如果没有，必须使用.NET Framework的命令行设施gacutil.exe来安装它们
       2. 为应用程序创建并配置数据库：
       3. 按需为应用程序配置IIS
       4. 为工作进程用户建立Windows账户权限
       5. 添加IIS文件映射：需要处理任何和默认ASP.NET安装注册的扩展名不同的文件扩展名URL，在IIS添加相应的文件映射
       6. 通过web.config文件为生产环境配置ASP.NET：正确添加(或修改)所有的连接字符串，应用程序设置以及安全和验证设置、会话状态设置、全球化设置
       7. 有时候还需要修改machine.config：在一个web宿主环境里，为了负载平衡，应用程序运行在多台服务器上，必须在所有这些机器间同步用于加密表单验证以及社图状态的密钥，这些密钥保存在machine.config里且需要在web农场的每台机器上都相同，这样一台及其才能够解密之前处理请求的另一台计算机所加密的信息

18.3.1 在 IIS 5.x 和 IIS 6.0上验证Asp.net安装

列举当前计算机安装的ASO.NET版本以及匹配的ISAPI扩展：aspnet\_regiis –lv

18.3.2在 IIS 5.x 和 IIS 6.0 上Asp.Net 并行执行

18.3.3在 IIS 7.0 里 Asp.net并行执行

18.3.4在 IIS 5.x 上部署时配置 Http 运行时设置

关于CLR的设置（ASP.NET运行时）<system.web>下方的<httpRuntime>节

18.3.5在 Asp.net 中编译模型

* + - * 1. 动态编译：

优点：使得随时修改标签文件甚至源代码文件成为可能，而应用程序在代码修改后自动编译

缺点：动态编译是在第一次请求时发生，修改代码后的第一次请求时间会比较长

* + - * 1. 站点预编译：

将整个网站编译到二进制文件中，这样就不会有代码文件甚至标记文件，如ASPX或ASMX文件，也被编译并以二进制文件的形式部署到目标计算机上

编译工具：aspnet\_compiler.exe(编译器必须在部署整个网站前在测试、开发计算机上启动)

Aspnet\_complizer –m metabase path

-v virtual directory path

-p physical path

Target directory

-u(表示可更新页面)：默认是移除所有页面和用户控件中的标记代码并把它们编译到一个位于bin目录的程序集中

-fixednames开关，让编译器生成固定的程序集名

* + - * 1. 让应用程序暂时离线：在web应用程序的根目录放置一个叫做App\_Offline.htm的文件即可（该文件大小）512kb，否则显示IE自带友好错误页面）

18.3.6使用 VisualStudio 进行部署

18.3.7 Visual Studio 2005 Web 部署项目

* 1. Aspnet\_merge.exe
     1. 将aspnet\_compiler.exe生成的程序集合并为一个预编译的程序集
     2. 只组合WEB UI内容程序集，把页面和用户控件生成的程序集合并为一个程序集
     3. 为网站的每个文件夹创建一个web UI内容（页面和用户控件）的程序集

##### 18.4 Asp.Net中的VirtualPathProvider

* 1. VirtualPathProvider类：允许你实现一种可从服务器访问的”虚拟URL”，就可以动态产生对某个URL的响应而不需要在硬盘上保存ASPX或HTML文件
  2. 意义：
     1. VirtualPathProvider提供了部署Web应用程序（或者，至少是Web应用程序的一部分）的有一种可能。
     2. 可以将Web应用程序的页面保存到了除问价系统以外的其他地方，而并未编写自己的接触页面框架，该框架使用从数据可动态获得的信息来创建控件并把它们添加到页面上，可以直接从数据库获取整个页面
     3. 可以用VirtualPathProvider类访问文件系统以外的其他特性，如主题和外观
     4. Windows SharePoint Services使用相同的方式从文件系统外获取内容
     5. VirtualPathProvider只能和ISAPI关联的资源一起工作

##### 18.5 Asp.Net中的健康监视

监测的机制：向应用程序添加事件、性能计数器、跟踪的技术

18.5.1 理解基本结构

在一组事件类里实现的事件类型，负责处理不同类型事件的提供程序

<healthMonitoring>

<providers><eventMappings><profiles><rules>

18.5.2 事件和提供程序

* + 1. 事件定义了应用程序的真实状态，提供程序指定事件如何被处理
    2. ASP.NET内置的事件提供程序：System.Web.Management
       1. 心跳：HeartBeats，定期引发的事件，内存消耗，CPU负载
       2. 应用程序生命期引发的事件：开始、关闭、会话开始、会话结束
       3. 安全稽核事件：WebAuditEvent，失败的登陆者视图访问没有必要的权限的资源
       4. 基于请求和响应的事件：请求打开、结束、有关产生的响应信息
       5. 错误：应用程序错误、基于请求的错误

##### 18.6总结

# 安全

### 第19章 Asp.Net安全模型

1. 安全本质：免受侵权
2. 采用机制：用户验证、授权、禁止对敏感资源的访问以及保护服务器存储的数据和网络上传输的数据

##### 19.1何为创建安全软件

###### 19.1.1理解潜在的威胁

MichelleHoward、SteveLipner 《TheSecurityDevelopmentLifecycle》

###### 19.1.2安全编程原则

安全编程原则：

绝对不要相信用户输入

对用户的输入信息总要进行强验证

验证代码：只根据合法值而不是非法值去验证用户的输入

绝不要使用字符串连接符来创建SQL语句

使用参数化的语句来确保你的程序不会被SQL注入

###### 19.1.3理解GateKeeper

##### 19.2 理解安全级别

###### 19.2.1验证

###### 19.2.2授权

###### 19.2.3机密性和完整性

###### 19.2.4互相协作

##### 19.3 IIS 安全

###### 19.3.1 IIS 5.x 和 IIS 6.0 中验证和授权

###### 19.3.2 IIS 7.0 的安全配置

###### 19.3.3理解SSl

##### 19.4 Asp.Net安全架构

###### 19.4.1验证

###### 19.4.2授权

###### 19.4.3安全上下文

###### 19.4.4成员资格和Roles API

##### 19.5总结

### 第20章表单验证

##### 20.1 表单验证介绍

###### 20.1.1 为什么使用表单验证

###### 20.1.2 为什么不适用表单验证

###### 20.1.3 为什么不自己实现Cookie 验证

###### 20.1.4表单验证类

##### 20.2实现表单验证

###### 20.2.1配置表单验证

###### 20.2.2禁止匿名用户访问

###### 20.2.3创建自定义登录页面

###### 20.2.4自定义凭证存储

###### 20.2.5表单验证中的持久cookie

##### 20.3 IIS 7.0 和表单验证

##### 20.4 总结

### 第21章成员资格

##### 21.1介绍 Asp.net 的成员资格API

##### 21.2使用成员资格API

###### 21.2.1 配置表单验证

###### 21.2.2 创建数据存储

###### 21.2.3 配置连接字符串和成员资格提供程序

###### 21.2.4 创建并验证用户

##### 21.3使用安全控件

###### 21.3.1 登陆控件

###### 21.3.2 登陆状态控件

###### 21.3.3 登陆视图控件

###### 21.3.4 恢复密码控件

###### 21.3.5 修改密码控件

###### 21.3.6 创建用户(CreateUserWizard)控件

##### 21.4在IIS 7.0里面配置成员资格

###### 21.4.1 配置提供程序和用户

###### 21.4.2 在其他应用程序里使用成员资格API

##### 21.5使用成员资格类

###### 21.5.1 从存储过程中获得用户

###### 21.5.2 更新存储中的用户

###### 21.5.3 创建和删除用户

###### 21.5.4 检验用户

###### 21.5.5 在windows 窗体中使用成员资格

##### 21.6总结

### 第22章 Windows 验证

##### 22.1介绍windows 验证

###### 22.1.1为什么是使用 windows 验证

###### 22.1.2为什么不适用windows 验证

###### 22.1.3 windows 验证机制

##### 22.2实现windows 验证

###### 22.2.1配置IIS 5.x 和 IIS 6.0

###### 22.2.2配置IIS7.0

###### 22.2.3配置 ASP.NET

###### 22.2.4禁止匿名访问

###### 22.2.5访问Windows用户信息

##### 22.3身份模拟

###### 22.3.1 在windows 2008 中执行身份模拟

###### 22.3.2 在 windows xp 中执行身份模拟

###### 22.3.3 在 windows server 2003 中执行身份模拟和委托

###### 22.3.4在 windows vista 上的身份模拟

###### 22.3.5在 windows server 2008 上的身份模拟和代理

###### 22.3.6经过配置的身份模拟

###### 22.3.7可编程的身份模拟

##### 22.4总结

### 第23章授权和角色

##### 23.1 URL授权

##### 23.2文件授权

##### 23.3在代码中检查授权

###### 23.3.1使用IsInRole()方法

###### 23.3.2使用PrincipalPermission类

##### 23.4为基于角色的授权使用角色服务

###### 23.4.1和角色一起使用LoginView 控件

###### 23.4.2通过程序访问角色

###### 23.4.3在windows 验证中使用角色服务

##### 23.5在 IIS 5 和 IIS 6 中保护非网页资源

###### 23.5.1添加一个文件类型的映射

###### 23.5.2写一个自定义的Http 处理程序

##### 23.6 IIS 7.0 里授权和角色

###### 23.6.1在IIS7.0 里使用Asp.net 角色授权

###### 23.6.2在IIS 7.0里管理Asp.net角色

##### 23.7总结

### 第24章用户配置

##### 24.1理解用户配置

###### 24.1.1用户配置的性能

###### 24.1.2用户配置如何存储数据

###### 24.1.3用户配置和验证

###### 24.1.4用户配置和自定义的数据组件

##### 24.2使用 SqlProfileProvider

###### 24.2.1创建用户配置表

###### 24.2.2配置提供程序

###### 24.2.3定义用户配置属性

###### 24.2.4使用用户配置属性

###### 24.2.5用户配置序列化

###### 24.2.6用户配置组

###### 24.2.7用户配置和自定义数据类型

###### 24.2.8用户配置API

###### 24.2.9匿名用户配置

##### 24.3自定义的用户配置提供程序

###### 24.3.1自定义的用户配置程序类

###### 24.3.2设计FactoredProfileProvider

###### 24.3.3编码实现 FactoredProfileProvider

###### 24.3.4测试 FactoredProfileProvider

##### 24.4总结

### 第25章加密

##### 25.1加密数据：机密性问题

##### 25.2 .NET加密命名空间

##### 25.3 .NET 加密类

###### 25.3.1对称加密算法

###### 25.3.2非对称加密

###### 25.3.3抽象加密类

###### 25.3.4 ICryptoTransform 接口

###### 25.3.5 CryptoStrim

##### 25.4加密敏感数据

###### 25.4.1管理秘密信息

###### 25.4.2使用对称算法

###### 25.4.3使用非对称算法

###### 25.4.4加密数据库中的敏感数据

##### 25.5加密查询字符串

###### 25.5.1包装查询字符串

###### 25.5.2创建一个测试页面

##### 25.6总结

### 第26章自定义成员资格提供程序

##### 26.1自定义提供程序的架构

##### 26.2创建自定义程序的基本步骤

###### 26.2.1自定义提供程序的整体设计

###### 26.2.2设计并实现自定义存储机制

###### 26.2.3实现提供程序类

###### 26.2.4使用自定义提供程序类

##### 26.3总结

# 高级用户界面

### 第27章自定义服务器控件

* 1. 服务器控件比较用户控件：
     1. 服务器控件允许完全地控制所生成的HTML
     2. 服务器控件提供更好的设计时支持（类似于VS自带服务器控件）

##### 27.1 自定义服务器控件入门

* 1. 服务器控件：直接或者间接派生自System.Web.UI.Control类的控件
     1. Control类：提供对所有的服务器控件都通用的属性和方法(ID、ViewState、Controls)
     2. System.Web.UI.WebControl：添加功能实现标准的样式(Font、ForeColor、BackColor)

###### 27.1.1创建简单的自定义控件

* 1. 直接生成HTML：
     1. HtmlTextWrite.Write()=>Write()方法输出的内容是浏览器最终显示的内容
  2. 管理样式特性和标签：
     1. RenderBeginTag：锚标签的开始标签
     2. RenderEndTag：锚标签的结束标签
  3. 样式特性、标签枚举：
     1. HtmlTextWriterTag:定义HTML标签,如：<a><p><font>
     2. HtmlTextWriterAttribute:定义HTML标签特性，如：onClick,href,align,alt
     3. HtmlTextWriterStyle:定义了14个样式特性,BackgroundColor,BackgroundImage,BorderStyle,BorderWidth,
        1. Color,FontFamily,FontSize,FontStyle,FontWeight,Height,Width，以分号分割，形成CSS样式信息列表
  4. Render()原理：
     1. 定义所有要添加到后续的标签上的特性
     2. 创建开始标签（RenderBeginTag），这些特性都被置于该标签内部
  5. HtmlTextWriter关键方法：
     1. AddAttribute() 添加任意的HTML特性及其值到一个HtmlTextWriter的输出流。这个特性自动地

被应用到你通过调用RenderBeginTag()创建的下一个标签。从HtmlTextWriterAttribute枚举

* + 1. AddStyleAttribute() 添加一个HTML样式特性及其值到一个HtmlTextWriter的输出流。这个特性被自动地应用到你通过调用RenderBeginTag()方法创建的下一个标签。从HtmlTextWriterStyle枚举，并且能根据所用浏览器是上层客户端还是下层客户端恰当的实现
    2. RenderBeginTag() 输出HTML元素的标签。如写一个锚标签，将输出一个<a>,从HtmltextWriterTag枚举
    3. RenderEndTag() 输出HTML元素的结束标签。输出一个</a>结束标签，不需要指定标签名称
    4. WriteBeginTag() 与RenderBeginTag()方法类似，但它不给开始标签输出结束字符”>”。这意味着能调用WriteAttribute()方法给这个标签添加更多的特性。关闭开始标签，可以使用Write(HtmlTextWriter.TagRightChar),它编写结束字符”>”
    5. WriteEndTag() 给当前的HTML标签输出一个元素的结束符号”>”，即通过使用WriteBeginTag()方法打开的最后一个HTML元素
    6. WriteAttribute() 输出一个HTML特性到输出流。这个方法的调用必须跟在WriteBeginTag()方法调用之后

###### 27.1.2使用自定义控件

* 1. App\_Code目录下：<%@Register TagPrefix=”apress” Namespace=”CustomServerControlsLibrary” %>
  2. 非App\_Code目录下：<%@Register TagPrefix=”apress” Namespace=”CustomServerControlsLibrary” Assembly=”CustomServerControlsLibrary”%>
  3. 可以重用标签前缀：把两个不同的命名空间或者两个完全不同的程序集映射到同一个标签前缀完全有效
  4. 多个页面使用同一个控件，Web.config配置：

<system.web>

<pages>

<controls>

<addtagPrefix="apress"namespace="CustomServerControlsLibrary"assembly="CustomServerControlsLibrary"/>

</controls>

</pages>

</system.web>

###### 27.1.3工具箱中的自定义控件

* 1. 丢失解决方案：File=>Add=>Existing Project找回控件项目
  2. 引用控件：
     1. Add Reference,选择Project，选择自定义控件项目
     2. 当在自定义控件项目被加载到设计环境中，自定义控件会自动出现在工具箱里
     3. 想让控件在任何web应用程序中都可用，且不想让web应用程序开发人员更改自定义控件的代码，部署编译后的自定义控件的程序集，右击工具箱=》Choose Items=》单击 .Net Framework Components =>浏览=》选择自定义程序集

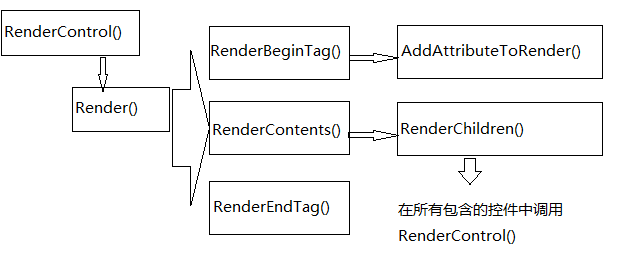
###### 27.1.4创建一个支持样式属性的web控件

* 1. WebControl不仅包含基本的样式属性，如Font、ForeColor、BackColor，而且它还自动的将这些属性呈现在控件的标签里。
  2. 工作原理：WebControl假设它应该这些特性到一个HTML标签，称作基本标签(base tag)。如果你输出多个元素，这些特性将被添加到包含其他元素的最外围元素。在构造函数为Web控件指定基本标签
  3. ASP.NET包含Render()方法实现的方法：
     1. RenderBeginTag() :输出控件的开始标签，以及你指定的特性
     2. RenderContents():输出开始和结束标签之间的所有内容，包括文本内容或其他HTML标签。这个方法是经常覆盖来输出自定义控件内容的方法
     3. RenderEndTag():输出控件的结束标签
  4. 调用WebControl的构造函数来指定基本控件标签(public **LinkControlThird():** base**(**HtmlTextWriterTag.A)**{}**).如果调用无参构造函数，一个<span>标签会自动产生
  5. 如果需要，可以覆盖Render()方法来改变ASP.NET的输出行为
  6. 无论何时自定义控件覆盖一个方法的时=时候，它都应该通过使用base这个关键字来调用基类的实现（确保你不会意外地抑制住其他需要运行的代码，通常基类方法所做的只是激发一个事件，但不总是）
  7. 手工创建的控件与VisualStudio生成控件的区别（VS生成控件包含了一系列自动生成的样板代码）：
     1. 文件的开始有一组using语句，它们导入有用的ASP.NET命名空间
     2. 控件类添加了Text属性，它被保存到视图状态里
     3. 控件类覆盖了RenderContents()方法以输出Text属性的内容
     4. 控件类声明和Text属性被配置了设计时支持的特性修饰
  8. 如果控件需要创建任何类型的用户界面，需要从WebControl类中派生

如果只需要知道UI特征的一个子集，或者组合多个控件，它们每个都拥有自己特定的样式属性，也许想要从Control类派生

.NET类库遵循的基本原则：总是从WebControl类派生，即使一些属性并不相关

###### 27.1.5呈现过程

* 1. 呈现起点（RenderControl()方法）：是ASP.NET用来把网页上的每个控件呈现成HTML的公共呈现方法。不能覆盖RenderControl()方法，但是可以覆盖Render()方法，但是如果只覆盖，却不实现其基类的实现，其他的呈现方法都将不会被激发
  2. Render()基类的实现调用了RenderBeginTag(),RenderContonts(),RenderEndTag()方法。RenderContents()的基类实现调用了另一个呈现方法：RenderChildren().这个方法调用Controls集合中的子控件集合，并且调用了每个控件的RenderChildren()=>可以很容易的从其他控件构建自己的控件
  3. 如何选择覆盖呈现的方法：
     1. 如果用新的内容替换整个呈现过程，或者想在基本标签之前添加HTML内容（JS代码），就可以覆盖Render()方法
     2. 利用自动的样式特性，定义一个基本标签(在调用基类构造函数时指定标签名参数,HtmlTextWriterTag.A)，并覆盖RenderContents()方法
     3. 如果想阻止子控件被显示或者要指定它们的呈现方式(例如让他们以相反的顺序呈现)，可以覆盖RenderChildren()方法
  4. 呈现过程图：
     1. 

//创建内存对象来捕获呈现输出

StringWriter **writer =** newStringWriter**();**

HtmlTextWriter **output =** newHtmlTextWriter**(writer);**

//将控件呈现到内存字符串

**lcThird.RenderControl(output);**

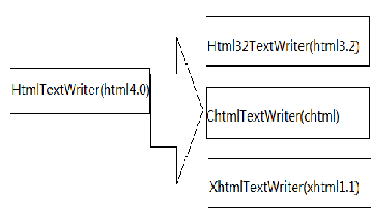
//显示HTML（编码它的内容、使其能以文本格式显示在浏览器中）

**lblHTML.Text =** "The HTML for lcThird is <br /><blockquote>"+Server.HtmlEncode(writer.ToString())+"</blockquote>"**;**

* + 1. 这段代码可以简化呈现代码，例如创建和配置HtmlTable是比较容易的，然后调用它的RenderControl()方法，而不是直接输出<table><tr><td>标签到输出流.

##### 27.2 处理不同的浏览器

###### 27.2.1 HtmlTextWriter

* 1. 
  2. 避免使用HtmlTextWriter.Write()方法输出输出原始的HTML文本(如果是只支持html3.2版本的浏览器，输出的CSS将没有效果)
  3. 在任何可能的地方使用高级方法如：RenderBeginTag()、RenderEndTag(),这样控件会有更强的灵活性,ASP.NET将会创建和传入正确的HtmlTextWriter，基于请求页面的浏览器的兼容性，HTML标记能自适应

###### 27.2.2浏览器检测

* 1. ASP.NET尝试将D:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG\Browsers目录下的字符串与发起请求时提供代理的用户代理字符串进行比对
  2. 该目录下<browser>元素包含了parentID特性，该特性引用另外一个<browser>定义，并从其继承所有设置

###### 27.2.3浏览器属性

* 1. HttpRequest对象的Browser属性来监测当前浏览器的配置，该属性返回一个HttpBrowserCapacities对象的引用(UserAgent获取用户代理字符串)
  2. 发出Http请求=>HttpBrowserCapacities对象被创建(基于相对应的.browser文件以浏览器能力信息填充)
  3. HttpBrowserCapacities属性
     1. Browser 用户代理首部中与请求一起发送的浏览器字符串
     2. MajorVersion 浏览器的主版本号(4.5中的4)
     3. MinorVersion 浏览器的次版本号(4.5中的5)
     4. Type 浏览器的名称和主版本号
     5. Version 浏览器的版本号
     6. Beta 是否是beta版本
     7. AOL 是否是AOL(美国在线)浏览器
     8. Platform 用户使用操作系统的平台名称
     9. Win16、win32 是否是win16、win32位的操作系统
     10. ClrVersion 客户端计算机上.NET CLR最高版本号。GetClrVersion()得到所有安装的CLR版本信息。当在网页中嵌入.NET windows form（窗体）控件的时候才有意义(客户端计算机不需要CLR运行普通的ASP.NET网页)
     11. ActiveXControls（ActiveX控件）、BackgroundSounds（背景声音）、Cookies（cookie）、Frames(支持框架)、Tables(HTML表格)、JavaScript（javascript，已经不用了，推荐使用EcmaScriptVersion属性）、VBScript（vbscript）、JavaApplets（java小程序）
     12. EcmaScriptVersion 浏览器支持的ECMA脚本的版本号
     13. MSDomVersion 客户端支持的微软HTML DOM的版本号
     14. Crawler 是否是web爬虫(crawler)搜索引擎
  4. HttpBrowserCapacities不能判断浏览器功能的当前状态(IE6支持JS，返回true，禁用JS后，仍然返回true)，ASP.NET致死读取在发起请求的时候从浏览器端传入服务器端的用户代理信息，并将此字符串与在.browser文件中预先定义的用户代理信息比较

###### 27.2.4覆盖浏览器类型侦测

* 1. ASP.NET让你能够完全决定页面如何呈现而不必依赖于自动的浏览器侦测（识别浏览器）=>在Page.PreInit阶段或者声明性(通过Page指令)地设置Page.ClientTarget属性
  2. 使用Page.ClientTarget后，自动侦测被禁用,ASP.NET在之后的请求中使用你指定的浏览器设置
  3. 指定定义假名(alias指定的ie5)，每个假名被映射到一个特定的用户代理字符串(这个用户代理的浏览器设置在对应的.browser文件中声明)

服务器的根web.config下

<clientTarget>

<add alias=”ie5” userAgent=”Mozilla/4.0(compatible;MSIE 5.5;Windows NT 4.0)”/>

</clientTarget>

<%@ Page ClientTarget=”ie5” %>具有该指令的页面总是强制ASP.NET按照IE5的行为来处理该页面

###### 27.2.5自适应呈现

* 1. ASP.NET 1.x中：每个控件都需要拥有内置的智能来自我调节以适应不同的浏览器
  2. ASP.NET 2.x中：全新的基于控件适配器的自适应性呈现模型(3.5中没有变化)，控件与适配器分离
  3. 通过.browser文件把任意控件与一个适配器连接起来
  4. 控件适配器被插入呈现过程而起作用。适配器在web控件生命周期中的每一个状态都被ASP.NET调用，这就使得适配器能够调整呈现过程并且处理其他细节（设备专属的视图状态逻辑）
  5. System.Web.UI.Adapters.ControlAdapter派生(.Control或者.WebControl)
  6. ControlAdapter.Control属性访问控件适配器连接的控件
  7. http://www.asp.net/CSSAdapters

###### 27.3 控件状态和事件

* 1. ASP.NET使用web控件在HTML和HTTP的底层细节之上创建了一个面向对象的抽象层，有两个基础：视图状态(View State)和回传(postback)
     1. 视图状态：一种在多次请求之间保存信息的机制
     2. 回传：一种使一个页面把表单数据集合回传到相同的URl的技术

###### 27.3.1视图状态

* 1. 回传之后需要使用视图状态集合恢复私有信息
  2. Page.RegisterRequiresViewStateEncryption()方法请求页面对视图状态信息进行加密
  3. 控件的ViewState属性和页面的ViewState属性是完全分开的：在控件的ViewState里了一项，在页面上是不能访问的，反之亦然
  4. 当页面被呈现为HTML的时候，ASP.NET提取页面和所有组合控件的视图状态，并将它们融合成一个特别的树形结构
  5. 注意：
     1. 不要在视图状态中保存大型对象，会减少页面的传输时间
     2. 视图状态任意受到包容它的页面的限制，将一个页面的EnableViewState属性设置为false，则控件的视图状态在每次回传之后会丢失（如果有确保控件能正常工作的重要信息，应该将这些信息保存在控件状态信息里）
     3. 在ViewState中读取一个不存在的项，报错
  6. WebControl类提供了ViewState属性，并未提供例如：Cache、Application、Session属性，可以通过静态的HttpContext.Current属性来访问这些属性
  7. 更大的灵活性来定制怎么存储视图状态信息：
     1. 覆盖LoadViewState()和SavaViewState()方法
     2. SaveViewState() 总是在一个控件被呈现为HTML之前被调用，可以使用该方法返回单一的可序列化对象
     3. LoadViewState() 控件在后续的回传中被重新创建的时候，该方法会被调用。你作为一个参数接收保存的对象，然后就可以用它来配置控件属性了
     4. 开发一个在视图状态中使用单一对象保存多项信息的更紧凑的方式时、从一个现存的控件派生新控件，想阻止其保存它的状态的时候、管理如何实现保存一个复杂控件的内嵌子控件的状态的时候

###### 27.3.2控件状态

* 1. ASP.NET包含功能的控件状态用来存储控件正在使用的数据（技术上工作方式和视图状态相同）
  2. ASP.NET将试图状态和控件状态信息放在同一个隐藏字段里，区别是：控件状态不受EnableViewState属性设置的影响
  3. 注意控件状态存储的信息，通常仅限于关键字段，如页面的索引或者数据的键值
  4. 要使用控件状态，必须以覆盖OnInit()为开端，然后调用Page.RegisterRequiresControlState()方法来标明你的控件需要访问控件状态
  5. 与视图状态不同：你不能直接通过集合的形式来访问控件状态，必须覆盖两个方法：SaveControlState()和LoadControlState()
  6. 工作模式：取到被基类序列化的任意控件状态，并且将这些来自基类的控件状态与一个包含你的新的可序列化的对象的一个对象合并。可以通过System.Web.Pair类来实现
  7. 保存单一对象

protected override object SaveControlState()

{

//从基类获得状态

object baseState = base.SaveControlState();

//结合你想要存储的状态对象，然后返回最终对象

return new Pair(baseState, someData);

}

* 1. 保存几部分信息

protected override object SaveControlState()

{

//从基类获得状态

object baseState = base.SaveControlState();

//结合你想要存储的状态对象，然后返回最终对象

//return new Pair(baseState, someData);

Pair p1 = new Pair(stringData, intData);

Pair p2 = new Pair(baseState,p1);

return p2;

}

###### 27.3.3回传数据和change事件

* 1. 自定义控件处理被传送到页面的数据，需要实现IPostBackDataHandler接口，实现该接口，向ASP.NET标明，当一个回传发生时，控件需要一个时机来检查回传数据
  2. IPostBackDataHandler方法：
     1. LoadPostData():页面被回传时，ASP.NET在任何控件事件被激发之前调用这个方法，允许你检查被回传回来的数据，并且相应地更新控件状态（不该在此时激发change事件，因为其他控件没有更新）
     2. RaisePostDataChangedEvent():在页面上所有输入控件都被初始化了之后，必要的话，ASP.NET调用该方法来给你机会激发change事件
  3. 必须使用name特性来为该控件添加UniqueID。ASP.NET将这个字符串与回传数据作比较，不添加该字段，LoadPostData()方法将不会被调用，而且将不能访问回传数据
  4. 另一种做法：在自定义OnInit()方法中调用Page.RegisterRequiresPostBack()方法，此时，如果没有显示呈现它，ASP.NET将添加唯一标识ID，以确保你仍然能够接收到回传
  5. 实现IPostBackDataHandler的方法以给控件响应用户修改的能力

###### 27.3.4触发回传

* 1. ASP.NET回传机制：JavaScript函数\_doPostBack()
  2. \_doPostBack()函数接收两个参数：触发回传控件的名字和一个代表额外回传的数据的字符串
  3. ASP.NET提供Page.ClientScript.GetPostBackEventReference()方法简化对\_doPostBack()函数的访问，该方法创建一个对客户端的\_doPostBack()函数的引用，通常将该函数引用放在控件的一个html元素的onclick特性中
  4. 当页面被回传的时候，ASP.NET判断哪一个控件触发了回传(通过查看每一个框架的UniqueID属性)，如果那个控件实现了IPostBackDataHandler事件处理程序，ASP.NET就以时间数据调用RaisePostDataChangedEvent()方法

##### 27.4 扩展现有的Web控件

1. 两种方式：组合和继承

###### 27.4.1组合控件

1. 继承System.Web.UI.WebControls.CompositeControl(自身派生自webControl)
2. 必须覆盖CreateChildControls()方法来添加子控件（可以创建一个或者多个子控件对象，设置它的属性和事件处理程序）
3. 将这些子控件加入到当前控件的Controls集合中
4. 优点：不需要定制呈现代码，呈现工作呗委托给了构成的服务器控件
5. CompositeControl控件实现了INamingContainer接口，该接口没有任何实现方法，仅仅是告诉ASP.NET保证所有的子控件拥有唯一的ID值，ASP.NET是通过在控件的ID之前附加了服务器控件的ID来实现这点的
6. 简单起见：应该使用成员变量跟踪构成控件,这使得控件里的任何方法都能访问成员控件,这时还不应该创建这些控件，创建控件的实例是CreateChildControls()方法的功能
7. 在网页中无法访问这个组合控件中的任何一个子控件，如果允许访问某个属性，必须在自定义控件类中添加属性过程（这些属性只是在试图状态里存储信息，而不是直接访问子控件，这是因为子控件也许还不存在，这些属性将会在CreateChildControls()方法里被应用到子控件上,所有的控件都在一个<span>标签里被呈现）
8. 处理回传，事件，以及创建复杂控件(如一个扩展了的GridView，或者一个导航助手)，那么使用组合控件能极大地简化工作

###### 27.4.2派生控件

* 1. 从现存的控件类派生一个跟家专用的控件,通过覆盖或者直接增加你需要的功能,而不需要重新创建整个控件（并不是总是可行的，因为有的控件在私有方法里编写了一些有关基础设施的关键的代码,使得你无法覆盖）
  2. 可以创建一个派生控件以便能用某些样式或格式化属性预先初始化一个现存的控件（如：在OnInit()方法里设置样式的自定义Calendar或GridView，这样当添加控件的时候，该实例就按照你所期望的样式格式化了，也可以添加新方法或者属性来增加全新的功能）
  3. 使用派生控件的原因：
     1. 为特定数据创建标签
     2. 如：一个Label显示xml内容，普通情况下label可能会输出一些乱码
     3. 定义一个转化规则的方法
        1. Public static string ConvertXmlToHtmlText(string inputText)
        2. {
           1. //替换所有的开始标签和结束标签
           2. String startPatten = @“<([^>])>”;
           3. Regex regEx = new Regex(startPatten);
           4. String outputText = regEx.Replace(inputText,”&lt;<b>$1&gt;</b>”);
           5. outputText = outputText.Replace(“”,”&nbsp;”);
           6. outputText = outputText.Replace(“\r\n”,”<br />”);
        3. }
     4. 覆盖呈现方法：RenderContents()

Public override void RenderContents(HtmlTextWriter output)

{

String xmlText = XmlLabel.ConvertXmlToHtmlText(Text)

Output.Write(xmlText);

}

##### 27.5 模板化控件

1. 模板化控件和样式允许创建控件和功能，同时又不将用户锁定于固定布局

###### 27.5.1.创建模板化控件

* 1. 创建模板化控件步骤：
     1. 创建一个组合控件。这个控件要从WebControl类派生并且实现INamingContainer接口,以确保每个子控件的名字都唯一
     2. 创建一个或者多个模板容器。模板容器允许用户在网页的.aspx文件中声明式地制定模板，为了支持模板，只需要接受一个ITemplate对象的控件属性
     3. Private ITemplate itemTemplate;
     4. Public ITemplate ItemTemplate{get{return itemTemplate;}set{itemTemplate = value;}}
     5. 模板并不存储在视图状态中，模板总是从.aspx文件读取，而且不会通过程序改变，这意味着可以即那个它存储在一个私有变量里，并且在每次回传的时候重建
     6. ITemplate接口定义了唯一一个方法：InstantiateIn()，该方法在现存控件的内部创建一个模板实例，本质上：当该方法被调用时，ASP.NET解析模板并且基于模板内部的标签和代码重建控件。这些控件之后被添加到传入该方法的控件容器中
     7. 在CreateChildControls()方法里，你使用InstantiateIn()方法创建了模板并将模板加入到了Controls集合中
     8. Protected override void CreateChildControls()

{

//在开始之前清空控件集合

Controls.Clear();

If(RepeatCount>0)

{

For()

{

Panel p = new Panel();

itemTemplate.InstantiateIn(p);

Controls.Add(p);

}

}

}

* 1. 要使用这个控件，需要为ItemTemplate属性提供一个模板，可以在<ItemTemplate>标签中声明式地添加HTML和控件标签
  2. 如果将PersistChildren特性的参数设置为false，则ASP.NET解析器将认为在任何内嵌的标签都是子控件

###### 27.5.2使用定制的模板

* 1. 创建一个自定义控件类来用作模板容器，需要实现INamingContainer接口
  2. 调整SuperSimpleRepeater的CreateChildControls()方法，让它创建SimpleRepeaterItem控件的实例
  3. 每一个模板属性都是用TemplateContrainer特性来表明当控件实例化模板的时候，他将使用什么类型的模板容器

###### 27.5.3样式

1. System.Web.UI.WebControls.Style类代表完整的样式信息集合
2. CopyFrom()
   1. 从一个对象复制样式到调用的样式对象，现存的值将被覆盖
3. MergeWith()
   1. 合并比两个样式对象的属性，就是第一个样式对象中存在的样式特性将不会被来自第二个样式对象的样式值覆盖

##### 27.6 总结

### 第28章设计时支持

* 1. 自定义控件需求：
     1. 运行时：与网页和代码交互
     2. 设计时：与VS交互

##### 主要成员

* + 1. 一系列不同的元素参与提供设计时支持：
    2. 特性：
       1. 使用原因：
          1. 为剪裁在“属性”窗口显示的内容提供信息并提供控件在工具箱的图标信息
          2. 特性配置了属性石如何序列化的
          3. 特性可以把其他设计时组件附加到控件（类型转换器、类型编辑器、控件设计器）
    3. Web资源：
       1. Web资源是自定义控件使用的相关文件，可以将这些文件嵌入到自定义控件程序集并通过一个特殊格式的URL在需要的时候取得它们
    4. 类型转换器：
       1. 可以将复杂或不常见的数据类型转换为更常见的数据类型或把更常见的数据类型转换为复杂类型
    5. 类型编辑器：为设置复杂类型的值提供一个图形界面
    6. 控件设计器：每个控件都有一个空间设计器，它管理控件设计时的外观和行为

##### 28.2设计时特性

* 1. 特性：即被编译到自定义控件的程序集的元数据里面的声明性的标志（特性是类）
     1. 特性可以被不同的代理读取和解释，可以添加一些特性来给CLR编译器或自定义工具传递信息

###### 28.2.1 Properties 窗口

Properties窗口显示一个简短的描述。（要将这个信息添加到自己的控件，需要用Category和Description特性修饰控件的每一个属性）

控件属性的特性：

特性 控件属性的描述

Browsable（true|false） 在properties属性窗口中是否会被显示

Category(string) 将在properties属性窗口的哪个类别下出现的字符串，不指定则是default类别里

Description（string） 该属性被选中时所显示的描述的字符串

DefaultValue() 在窗口中显示的属性的默认值（默认值通常是初始值，这时你不必使用DefaultValue

特性，该特性有时能让代码生成器优化其所生成的标签，如果默认值匹配的话，就

跳过默认属性）

Themeable（true|false） 是否作为外观的一部分被配置

Localizable（true|false） 是否可以本地化=》VS允许他的值被存储到一个卫星程序集里面

ReadOnly（bool） 在窗口中设置是否只读

DesignOny(bool) 为true时，该属性仅在设计时可以用

ImmutableObject（bool） 确保这个对象的子属性的只读和父属性对应

MergableProperty(bool) 配置当某个控件的多个实例被一起选中时，Properties窗口如何动作，所有选中控件

的属性可以一起设置

ParenthesizePropertyName(bool) VS是否在Properties窗口里该属性的两侧显示园括号

Bindable（bool） Vs将在DataBindings对话框中显示该属性，并且允许该属性被绑定到数据源里的

一个字段上

RefreshProperties() 指定当所修饰的属性变化时，Properties窗口中其余内容是否必须更新

应用于类和程序集的特性

特性 描述

DefaultEvent(string) 表明默认事件的名称。当在设计环境双击时，VS自动给该默认事件添加事件处理

程序，如没有默认事件，双击只会选中它

DefaultProperty(string) 表明默认属性的名称。该控件第一次被选择的时候，DefaultProperty是在Properties

窗口中默认高亮显示的属性（如没默认属性，第一次选择该控件时没有属性被选中）

ControlValuePropertyAttribute(string) 指定把ControlParameter作为数据源绑定到这个控件时默认使用的属性名

称

NonVisualControl（） 表明控件没用运行时外观

TagPrefix（string，string） 将一个前缀与一个命名空间关联，前缀是在向.aspx页面上添加空间标签时使用

ToolboxBitmap(type.string) 指定控件添加到工具箱后要显示的位图。默认vs使用一个通用的齿轮图标

ToolboxData(string) 指定把控件从工具箱拖放到页面上时控件在.aspx文件的标记。默认是只包含ID和

Runat特性的空标记

例子：

TagPrefix

###### 28.2.2特性与继承

* 1. 从一个具有设计时特性的基类派生一个控件时，该控件继承其父类设计时功能

[Browsable(false)]

Public override Unit Height

{

Get{return base.Height;}

Set{base.Height = value;}

}

###### 28.2.3工具箱图标

1. 位图必须和自定义控件类具有相同的名字，其扩展名为.bmp CustomTextBox CustomTextBox.bmp
2. 必须是16\*16像素，否则VS尝试压缩他的时候容易变形
3. 必须是16色且仅16色
4. 一旦添加位图文件，使用属性串口设置Build Action为 Embedded Resource即可
5. VS在工具箱上自动添加的控件项目，不会显示自定义图标。只有手工配置工具箱并把控件添加到某个永久页签上时，才会看到自定义图标
6. [ToolboxBitmap(typeof(CustomTextBox),@"Images\CustomTextBox.bmp")]
7. [ToolboxBitmap(typeof(CustomTextBox)," CustomTextBox1.bmp")]
8. [ToolboxBitmap(typeof(System.Web.UI.WebControls.TextBox)]

##### web资源

自定义控件会有其他相关联的非代码资源，有脚本文件、样式表、和需要与控件一起使用图像

###### 28.3.1创建资源

* 1. 创建web资源步骤：
     1. 将资源文件添加到项目中
     2. 在Properties窗口中，将Build Action 属性设置为Embedded Resource，而不是Content
     3. 这样，这些资源文件就被嵌入到编译后的程序集里
     4. 使这些资源能够通过程序访问到：
        1. [assembly:webResource(“CuctomServerControls.button1.jpg”,”image/jpeg”)]
        2. [assembly:webResource(内嵌资源全名,内容类型)]

###### 28.3.2获取资源

* 1. 告诉WebResource.axd处理程序需要什么资源和它在哪个程序集里面：Page.ClientScript.GetWebResoureUrl()
  2. WebResource,axd?a=命名空间&r=文件名.jpg&t=相当于一个时间戳

###### 28.3.3文本替换

* 1. 第三个bool类型参数，告诉他自动执行替换,使得可以创建指向其它嵌入资源的基本文本资源
  2. 假设一个web控件使用数个HTML文件提供帮助信息，可以使用超链接在HTML文件间相互连接，但是要把它们嵌入为Web资源
  3. 如果没有自动替换，就不能创建指向其他资源的资源，因为我们并不知道ASP.NET将要生成的资源名称

##### 代码序列化

1. 代码序列化：在properties窗口中配置控件属性时,VS需要能在.aspx文件创建和修改相应的控件标签（使用复合类型,模板化控件,支持子控件的时候，可能会遇到麻烦）
2. 当使用复合类型的时候作为属性时,或者当创建模板化控件时,或者一个支持子控件的时候,代码序列化可能会遇到麻烦

###### 28.4.1类型转换器

* 1. 类型转换器:是将某个特定的数据类型转换为字符串表示以及对应相反的过程（绝大多数核心的.NET数据类型都拥有默认的很棒的类型转换器,在System.ComponentModel命名空间里）
  2. 使用对象属性的控件
  3. 自定义的类型控制器
     1. 派生自TypeConverter
     2. CanConvertFrom()：检查数据类型,如类型转换器能够从某数据类型转换到自定义数据类型则返回true
     3. ConvertFrom()：Convert提供数据类型To自定义类型
     4. CanConvertTo()：类型转换器从自定义对象转换到数据类型则返回true
     5. ConvertTo()：Convert自定义数据类型To请求的数据类型
  4. Format属性：该属性公开一个自定义的RichLabelFormattingOptions类的实例
  5. 面向对象很好的实践：总是给其派生的基类机会来处理不准备支持的消息
  6. ExpandableObjectConverter:
     1. Asp.net 支持许多对象属性。最好的例子（Font属性，该属性引用一个FontInfo对象,他有Bold、Italic、Name等其他属性）
     2. 给自定义对象属性以+号的编辑能力只需创建一个从ExpandableObjectConverter类派生出来的自定义类型转换器来代替TypeConverter基类即可
     3. 修改类型转换器的时候，控件设计器和控件构造器的时候，修改不会在重新编译之后立即出现到设计环境里面，需要关闭带测试页面的解决方案重新打开它
     4. 问题:修改某个子属性，在Format框中显示的字符串没有立即更新（同步编译）

解决:应用NotifyParentProperty和RefreshProperties特性到RichLabelFormattingOptions类的属性上

###### 28.4.2序列化特性

* 1. 使用特性控制控件的属性如何被序列化到.aspx文件中（DesignerSerializationVisibility和PersistenceMode）
  2. DesignerSerializationVisibility：决定是否序列化一个属性
     1. Visible：默认值，指定了某个属性应该被序列化，而且作用于简单的数据类型(如字符串、日期和枚举类型)以及数值数据类型
     2. Content：序列化一个对象的完整内容，可以序列化具有多个属性的复杂类型,如集合
     3. Hidden：属性不可被序列化
  3. PersistentMode：指定一个属性是如何被序列化的
     1. Attribute：默认值，属性将被作为控件的HTML特性来序列化
     2. InnerProperty：作为控件的内嵌标签持久化。这是用来生成复合内嵌的对象层级的优先设置。如：Calendar和GridView
     3. InnerDefaultProperty：属性将被持久化存储在控件的标签内，将是控件标签的唯一内容。如：Label的Text属性，当使用默认属性时，属性名不会出现在内嵌的内容里
     4. EncodedInnerDefaultProperty：除了在持久化之前的内容将被HTML编码处理外，其他与InnerDefaultProperty相同

###### 28.4.3类型编辑器

##### 28.5控件设计器

###### 28.5.1设计时Html

###### 28.5.2智能标签

##### 28.6总结

### 第29章动态图形和GDI+

##### 29.1 ImageMap控件

###### 29.1.1创建热点

###### 29.1.2处理热点单击

###### 29.1.3自定义热点

##### 29.2使用GDI+ 画面

###### 29.2.1简单绘制

###### 29.2.2图像格式和质量

###### 29.2.3 Graphics 类

###### 29.2.4 GraphicPath 类

###### 29.2.5画笔

###### 29.2.6画刷

##### 29.3在网页上嵌入动态图形

###### 29.3.1使用png格式

###### 29.3.2传递信息给动态图像

###### 29.3.3使用GDI+ 的自定义控件

###### 29.3.4用GDI+ 绘制图表

###### 29.3.5总结

### 第30章使用Web部件页面的门户

##### 30.1典型门户页面

##### 30.2初级web部件页面

###### 30.2.1创建页面设计

###### 30.2.2 WebPathManager 和 WebPartZero

###### 30.2.3向页面添加web 部件

###### 30.2.4定制页面

##### 30.3创建web 部件

###### 30.3.1简单的web部件任务

###### 30.3.2开发高级web 部件

###### 30.3.3 web 部件编辑器

###### 30.3.4 连接web 部件

###### 30.3.5 自定义动词和web 部件

###### 30.3.6用户控件和高级web部件

###### 30.3.7动态上传web 部件

###### 30.3.8针对web 部件的授权

###### 30.3.9个性化的最后任务

##### 30.4总结

# 客户端程序设计

### 第31章 JavaScript 和 Ajax 技术

##### 31.1 JavaScript本质

###### 31.1.1 Html文档对象模型

###### 31.1.2客户端事件

###### 31.1.3 脚本块

###### 31.1.4 操纵Html元素

###### 31.1.5调试Javascript

##### 31.2基本javascript 示例

###### 32.2.1创建javascript页面处理器

###### 32.2.2使用javascript 异步下载图片

###### 32.2.3呈现脚本块

##### 31.3脚本注入攻击

###### 31.3.1请求验证

###### 31.3.2禁用请求验证

##### 31.4带有javascript 的自定义控件

###### 31.4.1弹出窗口

###### 31.4.2滚动按钮

##### 31.5框架

###### 31.5.1框架导航

###### 31.5.2内联框架

##### 31.6理解ajax

###### 31.6.1 xmlhttprequest 对象

###### 31.6.2 ajax示例

##### 31.7在客户端回调中使用ajax

###### 31.7.1创建客户端回调

###### 31.7.2深入了解客户端回调

###### 31.7.3自定义空间的客户端回调

##### 31.8 总结

### 第32章 Asp.Net Ajax

##### 32.1介绍Asp.net Ajax

###### 32.1.1 客户端的 Asp.net ajax：脚本库

###### 32.1.2服务端的 Asp.net ajax： ScriptManager

##### 32.2服务器回调

###### 32.2.1 Asp.net ajax 中的web 服务

###### 32.2.2 web 服务代理

###### 32.2.3 在页面放置web 方法

###### 32.2.4 Asp.net ajax 应用程序服务

##### 32.3 Asp.Net ajax 服务端控件

###### 32.3.1使用UpdatePanel 的局部呈现

###### 32.3.2使用Timer 的定时刷新

###### 32.3.3使用UpdateProcess 的长时间更新

##### 32.4深入客户端库

###### 32.4.1 理解客户端模型

###### 32.4.2 JavaScript 的面向对象编程

###### 32.4.3 web-page 框架

##### 32.5控件扩展器

###### 32.5.1 安装Asp.net ajax 控件工具包

###### 32.5.2 AutoCompleteExtender

###### 32.5.3 Asp.net ajax 控件工具包

##### 32.6总结

### 第33章 SilverLight

##### 33.1理解SilverLight

###### 33.1.1 SilverLight 和 Flash

###### 33.1.2应用SilverLight

###### 33.1.3SilverLight 和 WPF

###### 33.1.4安装 Silverlight 和 Visual Studio 扩展

##### 33.2创建Silverlight 项目

###### 33.2.Html 入口页面

###### 33.2.1 Silverlight 初始化脚本

###### 33.2.2 Xaml 页面

###### 33.2.3 Xaml 代码隐藏页

###### 33.2.4属性和事件

###### 33.2.5编译Silverlight

##### 33.3 Silverlight 本质

###### 33.3.1 Silverlight 中的 .net framework 类

###### 33.3.2 Canvas 类

###### 33.3.3文本

###### 33.3.4和 HTML 交互

###### 33.3.5隔离存储

##### 33.4 SilverLight 和 Asp.net

###### 33.4.1 Asp.net Futures

###### 33.4.2 Silverlight 和 Asp.net 之间的通信

##### 33.5绘制二维图形

###### 33.5.1简单形状

###### 33.5.2路径和几何图形

###### 33.5.3刷子

###### 33.5.4透明度

##### 33.6动画

###### 33.6.1动画基础

###### 33.6.2定义动画

###### 33.6.3交互性动画演示

###### 33.6.4变换

##### 33.7总结

2011-6-28 15:39

# 技术积累

### 第34章正则表达式

##### 34.1 正则表达式基本：

###### 34.1.1定义：

正则表达式是用来检验和操作字符串的强大工具。简单的理解正则表达式可以认为是一种特殊的验证字符串。

###### 34.1.2常见语法：

1. 字符匹配、重复匹配、字符定位、转义匹配和其他高级语法（字符分组、字符替换和字符决策）；
   1. 字符匹配

|  |  |  |
| --- | --- | --- |
| **字符语法** | **语法解释** | **语法例子** |
| \d | 匹配数字（0~9） | ‘\d’匹配8,不匹配12； |
| \D | 匹配非数字 | ‘\D’匹配c,不匹配3； |
| \w | 匹配任意单字符 | ‘\w\w’ 匹配A3,不匹配@3； |
| \W | 匹配非单字符 | ‘\W’匹配@,不匹配c； |
| \s | 匹配空白字符 | ‘\d\s\d’匹配3 d,不匹配abc； |
| \S | 匹配非空字符 | ‘\S\S\S’匹配A#4,不匹配3 d； |
| . | 匹配任意字符 | ‘....’匹配A$ 5,不匹配换行； |
| […] | 匹配括号中任意字符 | [b-d]匹配b、c、d, 不匹配e； |
| [^…] | 匹配非括号字符 | [^b-z]匹配a,不匹配b-z的字符； |

* 1. 重复匹配
  2. 字符定位
  3. 转义匹配

# .NET扩展知识学习

### 35 反射学习

##### 35.1 反射概念

##### 35.1.1 定义:

反射是一种机制,通过反射机制我们可以知道未知类型对象的类型信息

##### 35.1.2 使用场景

* + - 1. 有一个对象a,这个对象不是我们定义的，也许是通过网络捕捉到的，也许是使用泛型定义的，但我们想知道这个对象的类型信息，想知道这个对象有哪些方法或者属性什么的．甚至我们想进一步调用这个对象的方法．关键是现在我们只知道它是一个对象，不知道它的类型，自然不会知道它有哪些方法等信息
      2. 我们有一个dll文件，我们想调用里面的类．现在假设这个dll文件的类的定义，数量等不是固定的，是经常变化的．也许某一天你要在这个dll里面增加一个类定义．也许你觉得这没什么问题，现在关键是我们在另一个程序集里面要调用这个dll，这是我们的程序必须能够适应这个dll的变化，也就是说即使改变了dll文件的定义也不需要改变我们的程序集．这时候我们就会使用一个未知dll．我们该怎么办？

##### 35.2

### 36 WCF

### 37 Silverlight

### 38 Linq

# 第三方框架学习

### 39 AOP

### 40 IOC

### 41 Spring.Net

### 42 Nhibernate

### 43 ASP.NET MVC

### 44 Castle

### 45 SOA

# C#扩展知识学习

### 46委托与事件

##### 46.1委托

###### 46.1.1定义:

委托是一个类，它定义了方法的类型，使得可以将方法当作另一个方法的参数来进行传递，这种将方法动态地赋给参数的做法，可以避免在程序中大量使用If-Else(Switch)语句，同时使得程序具有更好的可扩展性。

###### 46.1.2使用场景:

将方法作为参数传递，委托类型，委托是一个类

###### 46.1.3定义代码:

public delegate void GreetingDelegate(string name);

public static void EnglishGreeting(string name)

{

Console.WriteLine("Morning,{0}", name);

}

public static void ChineseGreeting(string name)

{

Console.WriteLine("早上好,{0}", name);

}

public static void GreetPeople(string name, GreetingDelegate makingGreet)

{

makingGreet(name);

}

static void Main(string[] args)

{

GreetPeople("Jimmy Zhang", EnglishGreeting);

GreetPeople("张子阳", ChineseGreeting);

Console.ReadLine();

}

###### 46.1.4备注:

1. 委托是一个类，使用之前必须实例化或者是进行赋值
2. 注册到委托的方法，每一次执行委托的时候，其注册过且未注销的方法都会执行一遍，注销过的不再执行
3. 使用委托可以将多个方法绑定到同一个委托变量，当调用此变量时(这里用“调用”这个词，是因为此变量代表一个方法)，可以依次调用所有绑定的方法。

##### 46.2事件

###### 46.2.1定义:

* 1. Event封装了委托类型的变量，使得：在类的内部，不管你声明它是public还是protected，它总是private的。在类的外部，注册“+=”和注销“-=”的访问限定符与你在声明事件时使用的访问符相同。
  2. 声明一个事件不过类似于声明一个进行了封装的委托类型的变量而已

###### 46.2.2使用场景:

为什么在类型中使用事件向外部提供方法注册，而不是直接使用委托变量的原因

* + 1. 封装性和易用性
    2. 事件应该由事件发布者触发，而不应该由客户端（客户程序）来触发

###### 46.2.3定义代码:

public class GreetingManager

{  
//这一次我们在这里声明一个事件  
public event GreetingDelegate MakeGreet;  
  
 public void GreetPeople(string name)

{  
        MakeGreet(name);  
    }

}

###### 46.2.4备注:

对事件的声明实际是声明一个私有的委托变量

##### 46.3Observer设计模式

###### 46.3.1定义:

Observer设计模式是为了定义对象间的一种一对多的依赖关系，以便于当一个对象的状态改变时，其他依赖于它的对象会被自动告知并更新。Observer模式是一种松耦合的设计模式。

###### 46.3.2使用场景:

* 1. 假设热水器由三部分组成：热水器、警报器、显示器，它们来自于不同厂商并进行了组装。那么，应该是**热水器**仅仅负责烧水，它不能发出警报也不能显示水温；在水烧开时由**警报器**发出警报、**显示器**显示提示和水温。
  2. 问题：如何在水烧开的时候通知报警器和显示器？Observer设计模式中主要包括如下两类对象：

1. Subject：监视对象，它往往包含着其他对象所感兴趣的内容。在本范例中，热水器就是一个监视对象，它包含的其他对象所感兴趣的内容，就是temprature字段，当这个字段的值快到100时，会不断把数据发给监视它的对象。
2. Observer：监视者，它监视Subject，当Subject中的某件事发生的时候，会告知Observer，而Observer则会采取相应的行动。在本范例中，Observer有警报器和显示器，它们采取的行动分别是发出警报和显示水温。
   1. 在本例中，事情发生的顺序应该是这样的：
3. 警报器和显示器告诉热水器，它对它的温度比较感兴趣(注册)。
4. 热水器知道后保留对警报器和显示器的引用。
5. 热水器进行烧水这一动作，当水温超过95度时，通过对警报器和显示器的引用，自动调用警报器的MakeAlert()方法、显示器的ShowMsg()方法。

###### 46.3.3定义代码:

###### 46.3.4 备注:

###### 46.3.5 .Net Framework中的委托与事件

46.3.5.1 .Net Framework的编码规范

1. 委托类型的名称都应该以EventHandler结束。
2. 委托的原型定义：有一个void返回值，并接受两个输入参数：一个Object 类型，一个 EventArgs类型(或继承自EventArgs)。
3. 事件的命名为 委托去掉 EventHandler之后剩余的部分。
4. 继承自EventArgs的类型应该以EventArgs结尾。

46.3.5.2 说明

* 1. 委托声明原型中的Object类型的参数代表了Subject，也就是监视对象，在本例中是 Heater(热水器)。回调函数(比如Alarm的MakeAlert)可以通过它访问触发事件的对象(Heater)。
  2. EventArgs 对象包含了Observer所感兴趣的数据，在本例中是temperature。
  3. 上面这些其实不仅仅是为了编码规范而已，这样也使得程序有更大的灵活性。比如说，如果我们不光想获得热水器的温度，还想在Observer端(警报器或者显示器)方法中获得它的生产日期、型号、价格，那么委托和方法的声明都会变得很麻烦，而如果我们将热水器的引用传给警报器的方法，就可以在方法中直接访问热水器了。

##### 46.4 总结

###### 46.4.1事件与Observer

1. 单独谈论事件，我们说发布者(publisher)、订阅者(subscriber)、客户端(client)
2. 讨论Observer模式，我们说主题(subject)和观察者(observer)。客户端通常是包含Main()方法的Program类

###### 46.4.2事件与委托的使用场合

当使用委托变量时，客户端可以直接通过委托变量触发事件，也就是直接调用pub.NumberChanged(100)，这将会影响到所有注册了该委托的订阅者。而事件的本意应该为在事件发布者在其本身的某个行为中触发，比如说在方法DoSomething()中满足某个条件后触发。通过添加event关键字来发布事件，事件发布者的封装性会更好，事件仅仅是供其他类型订阅，而客户端不能直接触发事件（语句pub.NumberChanged(100)无法通过编译），事件只能在事件发布者Publisher类的内部触发（比如在方法pub.DoSomething()中），换言之，就是NumberChanged(100)语句只能在Publisher内部被调用。

###### 46.4.3为什么委托定义的返回值通常都为void？

委托变量可以供多个订阅者注册，如果定义了返回值，那么多个订阅者的方法都会向发布者返回数值，结果就是后面一个返回的方法值将前面的返回值覆盖掉了，因此，实际上只能获得最后一个方法调用的返回值

###### 46.4.4如何让事件只允许一个客户订阅？

* 1. 将事件声明为private的，然后提供两个方法来进行注册和取消注册

代码如下：

// 定义事件发布者  
public class Publishser {  
    private event GeneralEventHandler NumberChanged;    // 声明一个私有事件  
    // 注册事件  
    public void Register(GeneralEventHandler method) {  
        NumberChanged = method;  
    }  
    // 取消注册  
    public void UnRegister(GeneralEventHandler method) {  
        NumberChanged -= method;  
    }  
  
    public void DoSomething() {  
        // 做某些其余的事情  
        if (NumberChanged != null) {    // 触发事件  
            string rtn = NumberChanged();  
            Console.WriteLine("Return: {0}", rtn);      // 打印返回的字符串，输出为Subscriber3  
        }  
    }  
}

* 1. 事件访问器（Event Accessor）的东西，它用来封装委托变量

代码如下：

class Program {  
    static void Main(string[] args) {  
        Publishser pub = new Publishser();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
  
        pub.NumberChanged -= sub1.OnNumberChanged;  // 不会有任何反应  
        pub.NumberChanged += sub2.OnNumberChanged;  // 注册了sub2  
        pub.NumberChanged += sub1.OnNumberChanged;  // sub1将sub2的覆盖掉了  
          
        pub.DoSomething();          // 触发事件  
    }  
}  
  
// 定义委托  
public delegate string GeneralEventHandler();  
  
// 定义事件发布者  
public class Publishser {  
    // 声明一个委托变量  
    private GeneralEventHandler numberChanged;  
    // 事件访问器的定义  
    public event GeneralEventHandler NumberChanged {  
        add {  
            numberChanged = value;  
        }  
        remove {  
            numberChanged -= value;  
        }  
    }  
      
    public void DoSomething() {  
        // 做某些其他的事情  
        if (numberChanged != null) {    // 通过委托变量触发事件  
            string rtn = numberChanged();  
            Console.WriteLine("Return: {0}", rtn);      // 打印返回的字符串  
        }  
    }  
}  
  
// 定义事件订阅者  
public class Subscriber1 {  
    public string OnNumberChanged() {  
        Console.WriteLine("Subscriber1 Invoked!");  
        return "Subscriber1";  
    }  
}

###### 46.4.5获得多个返回值与异常处理

* + 1. 获得多个订阅者的返回值:
       1. 委托定义在编译时会生成一个继承自MulticastDelegate的类，而这个MulticastDelegate又继承自Delegate，在Delegate内部，维护了一个委托链表，链表上的每一个元素，为一个只包含一个目标方法的委托对象。而通过Delegate基类的GetInvocationList()静态方法，可以获得这个委托链表。
    2. 异常处理:
       1. 在异常处理中, 很有可能在触发事件时，订阅者的方法会抛出异常，而这一异常会直接影响到发布者，使得发布者程序中止，而后面订阅者的方法将不会被执行。
       2. 代码：

// 触发某个事件，以列表形式返回所有方法的返回值  
public static object[] FireEvent(Delegate del, params object[] args){  
  
    List<object> objList = new List<object>();  
  
    if (del != null) {  
        Delegate[] delArray = del.GetInvocationList();  
        foreach (Delegate method in delArray) {  
            try {  
                // 使用DynamicInvoke方法触发事件  
                object obj = method.DynamicInvoke(args);  
                if (obj != null)  
                    objList.Add(obj);  
            } catch { }  
        }  
    }  
    return objList.ToArray();  
}

public void DoSomething() {  
    // 做某些其他的事情  
    Program5.FireEvent(MyEvent, this, EventArgs.Empty);  
}

###### 46.4.6委托中订阅者方法超时的处理

* 1. 超时：方法运行所需要的时间超过一定的时间，导致服务器端与客户端之间的连接断开
  2. 超时和异常的区别：就是超时并不会影响事件的正确触发和程序的正常运行，却会导致事件触发后需要很长才能够结束。
  3. 委托的定义会生成继承自MulticastDelegate的完整的类，其中包含Invoke()、BeginInvoke()和EndInvoke()方法
  4. 直接调用委托时，实际上是调用了Invoke()方法，它会中断调用它的客户端，然后在客户端线程上执行所有订阅者的方法（客户端无法继续执行后面代码），最后将控制权返回客户端。用于方法的异步执行，即是在调用BeginInvoke()之后，客户端从线程池中抓取一个闲置线程，然后交由这个线程去执行订阅者的方法，而客户端线程则可以继续执行下面的代码。
  5. BeginInvoke()接受“动态”的参数个数和类型，其中前面参数的个数和类型与委托定义中接受的参数个数和类型相同，最后两个参数分别是AsyncCallback和Object类型
  6. 注意：
     1. 在委托类型上调用BeginInvoke()时，此委托对象只能包含一个目标方法，所以对于多个订阅者注册的情况，必须使用GetInvocationList()获得所有委托对象，然后遍历它们，分别在其上调用BeginInvoke()方法。如果直接在委托上调用BeginInvoke()，会抛出异常，提示“委托只能包含一个目标方法”。
     2. 如果订阅者的方法抛出异常，.NET会捕捉到它，但是只有在调用EndInvoke()的时候，才会将异常重新抛出。而在本例中，我们不使用EndInvoke()（因为我们不关心订阅者的执行情况），所以我们无需处理异常，因为即使抛出异常，也是在另一个线程上，不会影响到客户端线程（客户端甚至不知道订阅者发生了异常，这有时是好事有时是坏事）。
     3. BeginInvoke()方法属于委托定义所生成的类，它既不属于MulticastDelegate也不属于Delegate基类，所以无法继续使用可重用的FireEvent()方法，我们需要进行一个向下转换，来获取到实际的委托类型。

###### 46.4.7委托和方法的异步调用

对于每一个异步调用都通过创建线程来进行操作显然会对性能产生一定的影响，同时操作也相对繁琐一些

.Net中可以通过委托进行方法的异步调用，就是说客户端在异步调用方法时，本身并不会因为方法的调用而中断，而是从线程池中抓取一个线程去执行该方法，自身线程（主线程）在完成抓取线程这一过程之后，继续执行下面的代码，这样就实现了代码的并行执行。

使用线程池的好处就是避免了频繁进行异步调用时创建、销毁线程的开销。

当客户端调用EndInvoke()时，如果异步调用的方法没有执行完毕，则会中断当前线程而去等待该方法，只有当异步方法执行完毕后才会继续执行后面的代码。所以在调用完BeginInvoke()后立即执行EndInvoke()是没有任何意义的。我们通常在尽可能早的时候调用BeginInvoke()，然后在需要方法的返回值的时候再去调用EndInvoke()，或者是根据情况在晚些时候调用。

AsyncResult可以唯一的获取到与它相关的调用了的方法（或者也可以理解成委托对象）

我们在调用BeginInvoke()后不再需要保存IAysncResult了，因为AysncCallback委托将该对象定义在了回调方法的参数列表中；

### 47 异步编程

##### 47.1 异步基础

###### 47.1.1 适用于异步应用:

* + 1. 响应灵敏的界面
    2. 可伸缩良好的服务应用

###### 47.1.2 实现异步的方式:

* + 1. 创建一个后台线程，将任务委托给这个线程执行，但这种类型的异步并不是所有的都是高效的
    2. 计算密集型，CPU是主要资源。解方程，排序，压缩，图形图像处理等。对于这类应用无论怎样你都必须占用CPU时间，所以你可以通过创建后台线程的方式来实现异步。
    3. IO密集型，IO是主要瓶颈。IO不仅仅是我们常说的读取磁盘文件，像数据库操作、FTP、HTTP等一切网络通信，磁盘访问等都属于IO。当我们进行IO操作时，我们实际上是不需要占用CPU的，比如磁盘访问，一般现代的磁盘驱动器都有自己的控制系统，它自己能够控制寻道等，就相当于磁盘自己有一个小CPU在工作，而我们机器的CPU这个时候就可以解放出来（不用等待，可以干其他的活儿），是不占用的。那么这个时候如果我们能使用异步IO（发起异步IO然后立即返回，当异步IO执行完毕后会通知你）将会对我们的应用的效率带来革命性的影响，因为IO相对于CPU来说是非常非常非常慢速的设备，我们甚至只需要很少的线程就可以处理很多的任务。

###### 47.1.3

###### 47.1.4

##### 47.2

###### 47.2.1

###### 47.2.2

###### 47.2.3

##### 47.3

###### 47.1.1

###### 47.1.2

##### 47.4

###### 47.1.1

###### 47.1.2

### 48 C#DateTime格式化

##### 48.1 GridView中Dataformatstring 格式化日期应注意的问题

1. HtmlEncode="False"

2. DataFormatString="{0:d}"

##### 48.2 C#格式化日期时间

DateTime dt = DateTime.Now;  
Label1.Text = dt.ToString();//2005-11-5 13:21:25  
Label2.Text = dt.ToFileTime().ToString();//127756416859912816  
Label3.Text = dt.ToFileTimeUtc().ToString();//127756704859912816  
Label4.Text = dt.ToLocalTime().ToString();//2005-11-5 21:21:25  
Label5.Text = dt.ToLongDateString().ToString();//2005年11月5日  
Label6.Text = dt.ToLongTimeString().ToString();//13:21:25  
Label7.Text = dt.ToOADate().ToString();//38661.5565508218  
Label8.Text = dt.ToShortDateString().ToString();//2005-11-5  
Label9.Text = dt.ToShortTimeString().ToString();//13:21  
Label10.Text = dt.ToUniversalTime().ToString();//2005-11-5 5:21:25  
?2005-11-5 13:30:28.4412864  
Label1.Text = dt.Year.ToString();//2005  
Label2.Text = dt.Date.ToString();//2005-11-5 0:00:00  
Label3.Text = dt.DayOfWeek.ToString();//Saturday  
Label4.Text = dt.DayOfYear.ToString();//309  
Label5.Text = dt.Hour.ToString();//13  
Label6.Text = dt.Millisecond.ToString();//441  
Label7.Text = dt.Minute.ToString();//30  
Label8.Text = dt.Month.ToString();//11  
Label9.Text = dt.Second.ToString();//28  
Label10.Text = dt.Ticks.ToString();//632667942284412864  
Label11.Text = dt.TimeOfDay.ToString();//13:30:28.4412864  
Label1.Text = dt.ToString();//2005-11-5 13:47:04  
Label2.Text = dt.AddYears(1).ToString();//2006-11-5 13:47:04  
Label3.Text = dt.AddDays(1.1).ToString();//2005-11-6 16:11:04  
Label4.Text = dt.AddHours(1.1).ToString();//2005-11-5 14:53:04  
Label5.Text = dt.AddMilliseconds(1.1).ToString();//2005-11-5 13:47:04  
Label6.Text = dt.AddMonths(1).ToString();//2005-12-5 13:47:04  
Label7.Text = dt.AddSeconds(1.1).ToString();//2005-11-5 13:47:05  
Label8.Text = dt.AddMinutes(1.1).ToString();//2005-11-5 13:48:10  
Label9.Text = dt.AddTicks(1000).ToString();//2005-11-5 13:47:04  
Label10.Text = dt.CompareTo(dt).ToString();//0  
//Label11.Text = dt.Add(?).ToString();//问号为一个时间段  
Label1.Text = dt.Equals("2005-11-6 16:11:04").ToString();//False  
Label2.Text = dt.Equals(dt).ToString();//True  
Label3.Text = dt.GetHashCode().ToString();//1474088234  
Label4.Text = dt.GetType().ToString();//System.DateTime  
Label5.Text = dt.GetTypeCode().ToString();//DateTime  
Label1.Text = dt.GetDateTimeFormats('s')[0].ToString();//2005-11-05T14:06:25  
Label2.Text = dt.GetDateTimeFormats('t')[0].ToString();//14:06  
Label3.Text = dt.GetDateTimeFormats('y')[0].ToString();//2005年11月  
Label4.Text = dt.GetDateTimeFormats('D')[0].ToString();//2005年11月5日  
Label5.Text = dt.GetDateTimeFormats('D')[1].ToString();//2005 11 05  
Label6.Text = dt.GetDateTimeFormats('D')[2].ToString();//星期六 2005 11 05  
Label7.Text = dt.GetDateTimeFormats('D')[3].ToString();//星期六 2005年11月5日  
Label8.Text = dt.GetDateTimeFormats('M')[0].ToString();//11月5日  
Label9.Text = dt.GetDateTimeFormats('f')[0].ToString();//2005年11月5日 14:06  
Label10.Text = dt.GetDateTimeFormats('g')[0].ToString();//2005-11-5 14:06  
Label11.Text = dt.GetDateTimeFormats('r')[0].ToString();//Sat, 05 Nov 2005 14:06:25 GMT  
Label1.Text =string.Format("{0:d}",dt);//2005-11-5  
Label2.Text =string.Format("{0:D}",dt);//2005年11月5日  
Label3.Text =string.Format("{0:f}",dt);//2005年11月5日 14:23  
Label4.Text =string.Format("{0:F}",dt);//2005年11月5日 14:23:23  
Label5.Text =string.Format("{0:g}",dt);//2005-11-5 14:23  
Label6.Text =string.Format("{0:G}",dt);//2005-11-5 14:23:23  
Label7.Text =string.Format("{0:M}",dt);//11月5日  
Label8.Text =string.Format("{0:R}",dt);//Sat, 05 Nov 2005 14:23:23 GMT  
Label9.Text =string.Format("{0:s}",dt);//2005-11-05T14:23:23  
Label10.Text = string.Format("{0:t}",dt);//14:23  
Label11.Text = string.Format("{0:T}",dt);//14:23:23  
Label12.Text = string.Format("{0:u}",dt);//2005-11-05 14:23:23Z  
Label13.Text = string.Format("{0:U}",dt);//2005年11月5日 6:23:23  
Label14.Text = string.Format("{0:Y}",dt);//2005年11月  
Label15.Text = string.Format("{0}",dt);//2005-11-5 14:23:23?  
Label16.Text = string.Format("{0:yyyyMMddHHmmssffff}",dt);   //yyyymm等可以设置,比如Label16.Text = string.Format("{0:yyyyMMdd}",dt);http://imgcache.qq.com/ac/b.gif

常用绑定日期格式化：<%# Eval("DateTime","{0:yyyy-MM-dd}")%>

### 49 微信公众平台开发

##### 49.1 验证服务器地址

###### 49.1.1 开发者提交信息后，微信服务器将发送GET请求到填写的服务器地址URL上，GET请求携带四个参数

1. signature微信加密签名，signature结合了开发者填写的token参数和请求中的timestamp参数、nonce参数。
2. timestamp时间戳
3. nonce随机数
4. echostr随机字符串

###### 49.1.2 开发者通过检验signature对请求进行校验（下面有校验方式）。若确认此次GET请求来自微信服务器，请原样返回echostr参数内容，则接入生效，成为开发者成功，否则接入失败。

1. 将token、timestamp、nonce三个参数进行**字典序**排序
2. 将三个参数字符串拼接成一个字符串进行sha1加密
3. 开发者获得加密后的字符串可与signature对比，标识该请求来源于微信

###### 49.1.3 获取access\_token

access\_token是公众号的全局唯一票据，公众号调用各接口时都需使用access\_token。开发者需要进行妥善保存。access\_token的存储至少要保留512个字符空间。access\_token的有效期目前为2个小时，需定时刷新，重复获取将导致上次获取的access\_token失效。

1. 为了保密appsecrect，第三方需要一个access\_token获取和刷新的中控服务器。而其他业务逻辑服务器所使用的access\_token均来自于该中控服务器，不应该各自去刷新，否则会造成access\_token覆盖而影响业务；
2. 目前access\_token的有效期通过返回的expire\_in来传达，目前是7200秒之内的值。中控服务器需要根据这个有效时间提前去刷新新access\_token。在刷新过程中，中控服务器对外输出的依然是老access\_token，此时公众平台后台会保证在刷新短时间内，新老access\_token都可用，这保证了第三方业务的平滑过渡；
3. access\_token的有效时间可能会在未来有调整，所以中控服务器不仅需要内部定时主动刷新，还需要提供被动刷新access\_token的接口，这样便于业务服务器在API调用获知access\_token已超时的情况下，可以触发access\_token的刷新流程。
4. <https://api.weixin.qq.com/cgi-bin/token?grant_type=client_credential&appid=APPID&secret=APPSECRET>
5. grant\_type获取access\_token填写client\_credential
6. appid第三方用户唯一凭证
7. secret第三方用户唯一凭证密钥，即appsecret

##### 49.2 自定义菜单

###### 49.2.1自定义菜单创建接口

1. 自定义菜单创建说明

1、自定义菜单最多包括3个一级菜单，每个一级菜单最多包含5个二级菜单。

2、一级菜单最多4个汉字，二级菜单最多7个汉字，多出来的部分将会以“...”代替。

3、创建自定义菜单后，菜单的刷新策略是，在用户进入公众号会话页或公众号profile页时，如果发现上一次拉取菜单的请求在5分钟以前，就会拉取一下菜单，如果菜单有更新，就会刷新客户端的菜单。测试时可以尝试取消关注公众账号后再次关注，则可以看到创建后的效果。

1. 自定义菜单按钮类型
   1. Click:点击推事件

用户点击click类型按钮后，微信服务器会通过消息接口推送消息类型为event 的结构给开发者（参考消息接口指南），并且带上按钮中开发者填写的key值，开发者可以通过自定义的key值与用户进行交互；

* 1. View:跳转URL

用户点击view类型按钮后，微信客户端将会打开开发者在按钮中填写的网页URL，可与网页授权获取用户基本信息接口结合，获得用户基本信息。

* 1. Scancode\_push:扫码推事件

用户点击按钮后，微信客户端将调起扫一扫工具，完成扫码操作后显示扫描结果（如果是URL，将进入URL），且会将扫码的结果传给开发者，开发者可以下发消息。

* 1. Scancode\_waitmsg:扫码推事件且弹出“消息接收中”提示框

用户点击按钮后，微信客户端将调起扫一扫工具，完成扫码操作后，将扫码的结果传给开发者，同时收起扫一扫工具，然后弹出“消息接收中”提示框，随后可能会收到开发者下发的消息。

* 1. pic\_sysphoto：弹出系统拍照发图

用户点击按钮后，微信客户端将调起系统相机，完成拍照操作后，会将拍摄的相片发送给开发者，并推送事件给开发者，同时收起系统相机，随后可能会收到开发者下发的消息。

* 1. pic\_photo\_or\_album：弹出拍照或者相册发图

用户点击按钮后，微信客户端将弹出选择器供用户选择“拍照”或者“从手机相册选择”。用户选择后即走其他两种流程。

* 1. pic\_weixin：弹出微信相册发图器

用户点击按钮后，微信客户端将调起微信相册，完成选择操作后，将选择的相片发送给开发者的服务器，并推送事件给开发者，同时收起相册，随后可能会收到开发者下发的消息。

* 1. location\_select：弹出地理位置选择器

用户点击按钮后，微信客户端将调起地理位置选择工具，完成选择操作后，将选择的地理位置发送给开发者的服务器，同时收起位置选择工具，随后可能会收到开发者下发的消息。

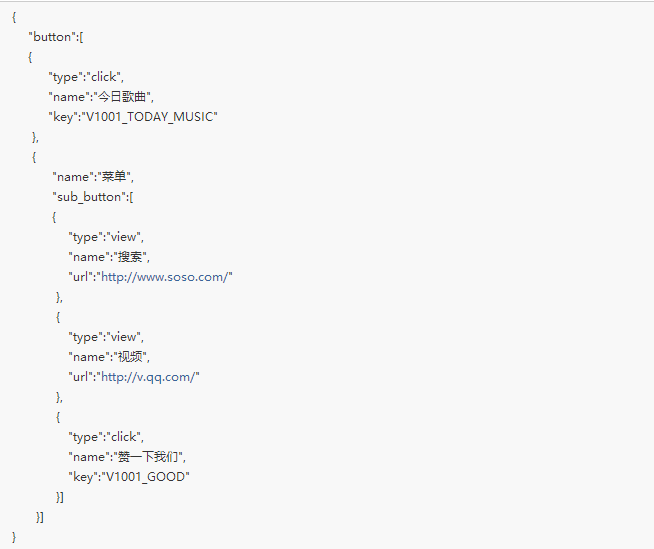
* 1. media\_id：下发消息（除文本消息）

用户点击media\_id类型按钮后，微信服务器会将开发者填写的永久素材id对应的素材下发给用户，永久素材类型可以是图片、音频、视频、图文消息。请注意：永久素材id必须是在“素材管理/新增永久素材”接口上传后获得的合法id。

* 1. view\_limited：跳转图文消息URL

用户点击view\_limited类型按钮后，微信客户端将打开开发者在按钮中填写的永久素材id对应的图文消息URL，永久素材类型只支持图文消息。请注意：永久素材id必须是在“素材管理/新增永久素材”接口上传后获得的合法id。

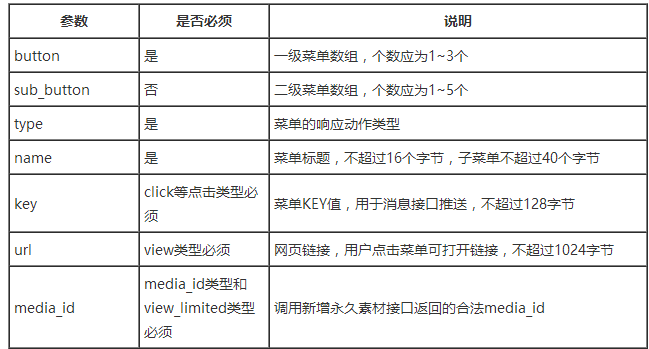
1. click和view的请求示例



1. http请求方式

<https://api.weixin.qq.com/cgi-bin/menu/create?access_token=ACCESS_TOKEN>

1. 参数说明：



1. 返回结果：

###### 49.2.2 自定义菜单查询接口

1. 使用接口创建自定义菜单后，开发者还可使用接口查询自定义菜单的结构。另外请注意，在设置了个性化菜单后，使用本自定义菜单查询接口可以获取默认菜单和全部个性化菜单信息。
2. http请求方式：GET

<https://api.weixin.qq.com/cgi-bin/menu/get?access_token=ACCESS_TOKEN>

1. 注：menu为默认菜单，conditionalmenu为个性化菜单列表。字段说明请见个性化菜单接口页的说明。

###### 49.2.3 自定义菜单删除接口

1. 使用接口创建自定义菜单后，开发者还可使用接口删除当前使用的自定义菜单。另请注意，在个性化菜单时，调用此接口会删除默认菜单及全部个性化菜单。
2. 请求说明

<https://api.weixin.qq.com/cgi-bin/menu/delete?access_token=ACCESS_TOKEN>

###### 49.2.4 自定义菜单事件推送

1. 用户点击自定义菜单后，微信会把点击事件推送给开发者，请注意，点击菜单弹出子菜单，不会产生上报。**请注意，第3个到第8个的所有事件，仅支持微信iPhone5.4.1以上版本，和Android5.4以上版本的微信用户，旧版本微信用户点击后将没有回应，开发者也不能正常接收到事件推送。**

[1 点击菜单拉取消息时的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#.E7.82.B9.E5.87.BB.E8.8F.9C.E5.8D.95.E6.8B.89.E5.8F.96.E6.B6.88.E6.81.AF.E6.97.B6.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

[2 点击菜单跳转链接时的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#.E7.82.B9.E5.87.BB.E8.8F.9C.E5.8D.95.E8.B7.B3.E8.BD.AC.E9.93.BE.E6.8E.A5.E6.97.B6.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

[3 scancode\_push：扫码推事件的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#scancode_push.EF.BC.9A.E6.89.AB.E7.A0.81.E6.8E.A8.E4.BA.8B.E4.BB.B6.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

[4 scancode\_waitmsg：扫码推事件且弹出“消息接收中”提示框的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#scancode_waitmsg.EF.BC.9A.E6.89.AB.E7.A0.81.E6.8E.A8.E4.BA.8B.E4.BB.B6.E4.B8.94.E5.BC.B9.E5.87.BA.E2.80.9C.E6.B6.88.E6.81.AF.E6.8E.A5.E6.94.B6.E4.B8.AD.E2.80.9D.E6.8F.90.E7.A4.BA.E6.A1.86.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

[5 pic\_sysphoto：弹出系统拍照发图的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#pic_sysphoto.EF.BC.9A.E5.BC.B9.E5.87.BA.E7.B3.BB.E7.BB.9F.E6.8B.8D.E7.85.A7.E5.8F.91.E5.9B.BE.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

[6 pic\_photo\_or\_album：弹出拍照或者相册发图的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#pic_photo_or_album.EF.BC.9A.E5.BC.B9.E5.87.BA.E6.8B.8D.E7.85.A7.E6.88.96.E8.80.85.E7.9B.B8.E5.86.8C.E5.8F.91.E5.9B.BE.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

[7 pic\_weixin：弹出微信相册发图器的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#pic_weixin.EF.BC.9A.E5.BC.B9.E5.87.BA.E5.BE.AE.E4.BF.A1.E7.9B.B8.E5.86.8C.E5.8F.91.E5.9B.BE.E5.99.A8.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

[8 location\_select：弹出地理位置选择器的事件推送](https://mp.weixin.qq.com/wiki/19/a037750e2df0261ab0a84899d16abd33.html#location_select.EF.BC.9A.E5.BC.B9.E5.87.BA.E5.9C.B0.E7.90.86.E4.BD.8D.E7.BD.AE.E9.80.89.E6.8B.A9.E5.99.A8.E7.9A.84.E4.BA.8B.E4.BB.B6.E6.8E.A8.E9.80.81)

1. 点击菜单拉取消息时的事件推送
   1. 推送XML数据包示例：

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[FromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[CLICK]]></Event>

<EventKey><![CDATA[EVENTKEY]]></EventKey>

</xml>

* 1. 参数说明



1. 点击菜单跳转链接时的事件推送
   1. 推送XML数据包示例：

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[FromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[VIEW]]></Event>

<EventKey><![CDATA[www.qq.com]]></EventKey>

<MenuId>MENUID</MenuId>

</xml>

* 1. 参数说明



1. scancode\_push：扫码推事件的事件推送
   1. 推送XML数据包示例：

<xml><ToUserName><![CDATA[gh\_e136c6e50636]]></ToUserName>

<FromUserName><![CDATA[oMgHVjngRipVsoxg6TuX3vz6glDg]]></FromUserName>

<CreateTime>1408090502</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[scancode\_push]]></Event>

<EventKey><![CDATA[6]]></EventKey>

<ScanCodeInfo><ScanType><![CDATA[qrcode]]></ScanType>

<ScanResult><![CDATA[[1]](http://weixin.qq.com/r/5HXdxQ-EKFJXrUum9yD2)]></ScanResult>

</ScanCodeInfo>

</xml>

* 1. 参数说明：



1. scancode\_waitmsg：扫码推事件且弹出“消息接收中”提示框的事件推送
   1. 推送XML数据包示例：

<xml><ToUserName><![CDATA[gh\_e136c6e50636]]></ToUserName>

<FromUserName><![CDATA[oMgHVjngRipVsoxg6TuX3vz6glDg]]></FromUserName>

<CreateTime>1408090606</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[scancode\_waitmsg]]></Event>

<EventKey><![CDATA[6]]></EventKey>

<ScanCodeInfo><ScanType><![CDATA[qrcode]]></ScanType>

<ScanResult><![CDATA[[2]](http://weixin.qq.com/r/5HXdxQ-EKFJXrUum9yD2)]></ScanResult>

</ScanCodeInfo>

</xml>

* 1. 参数说明



1. pic\_sysphoto：弹出系统拍照发图的事件推送
   1. 推送XML数据包示例：

<xml><ToUserName><![CDATA[gh\_e136c6e50636]]></ToUserName>

<FromUserName><![CDATA[oMgHVjngRipVsoxg6TuX3vz6glDg]]></FromUserName>

<CreateTime>1408090651</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[pic\_sysphoto]]></Event>

<EventKey><![CDATA[6]]></EventKey>

<SendPicsInfo><Count>1</Count>

<PicList><item><PicMd5Sum><![CDATA[1b5f7c23b5bf75682a53e7b6d163e185]]></PicMd5Sum>

</item>

</PicList>

</SendPicsInfo>

</xml>

* 1. 参数说明



1. pic\_photo\_or\_album：弹出拍照或者相册发图的事件推送
   1. 推送XML数据包示例：

<xml><ToUserName><![CDATA[gh\_e136c6e50636]]></ToUserName>

<FromUserName><![CDATA[oMgHVjngRipVsoxg6TuX3vz6glDg]]></FromUserName>

<CreateTime>1408090816</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[pic\_photo\_or\_album]]></Event>

<EventKey><![CDATA[6]]></EventKey>

<SendPicsInfo><Count>1</Count>

<PicList><item><PicMd5Sum><![CDATA[5a75aaca956d97be686719218f275c6b]]></PicMd5Sum>

</item>

</PicList>

</SendPicsInfo>

</xml>

* 1. 参数说明



1. pic\_weixin：弹出微信相册发图器的事件推送
   1. 推送XML数据包示例：

<xml><ToUserName><![CDATA[gh\_e136c6e50636]]></ToUserName>

<FromUserName><![CDATA[oMgHVjngRipVsoxg6TuX3vz6glDg]]></FromUserName>

<CreateTime>1408090816</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[pic\_weixin]]></Event>

<EventKey><![CDATA[6]]></EventKey>

<SendPicsInfo><Count>1</Count>

<PicList><item><PicMd5Sum><![CDATA[5a75aaca956d97be686719218f275c6b]]></PicMd5Sum>

</item>

</PicList>

</SendPicsInfo>

</xml>

* 1. 参数说明：



1. location\_select：弹出地理位置选择器的事件推送
   1. 推送XML数据包示例：

<xml><ToUserName><![CDATA[gh\_e136c6e50636]]></ToUserName>

<FromUserName><![CDATA[oMgHVjngRipVsoxg6TuX3vz6glDg]]></FromUserName>

<CreateTime>1408091189</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[location\_select]]></Event>

<EventKey><![CDATA[6]]></EventKey>

<SendLocationInfo><Location\_X><![CDATA[23]]></Location\_X>

<Location\_Y><![CDATA[113]]></Location\_Y>

<Scale><![CDATA[15]]></Scale>

<Label><![CDATA[ 广州市海珠区客村艺苑路 106号]]></Label>

<Poiname><![CDATA[]]></Poiname>

</SendLocationInfo>

</xml>

* 1. 参数说明：

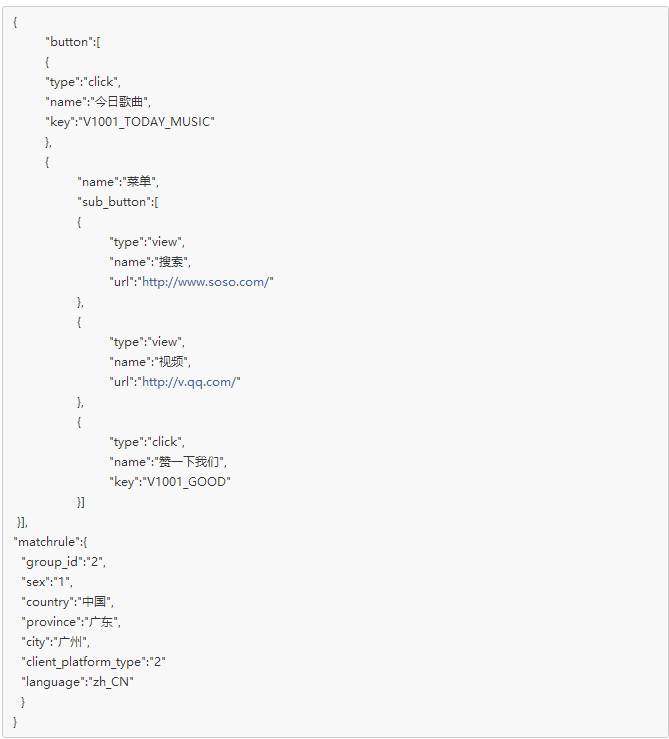


###### 49.2.5 个性化菜单接口

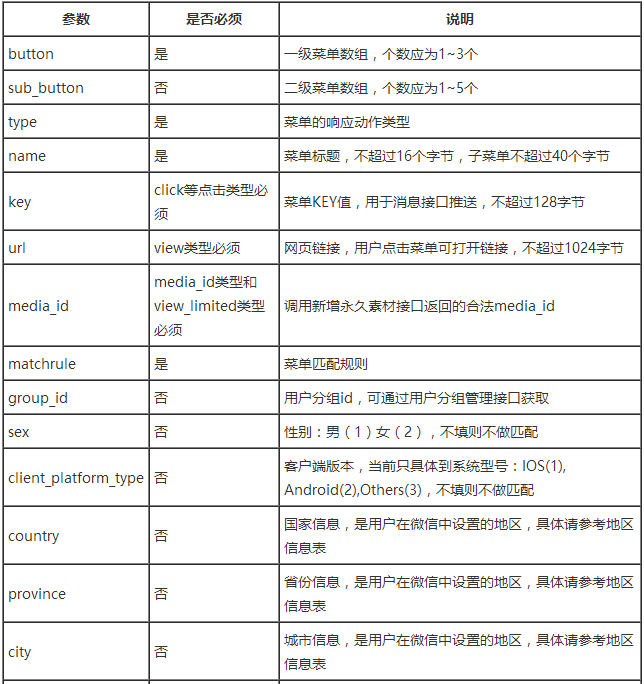
1. 为了帮助公众号实现灵活的业务运营，微信公众平台新增了个性化菜单接口，开发者可以通过该接口，让公众号的不同用户群体看到不一样的自定义菜单。该接口开放给已认证订阅号和已认证服务号。
2. 开发者可以通过以下条件来设置用户看到的菜单：
3. 用户分组（开发者的业务需求可以借助用户分组来完成）
4. 性别
5. 手机操作系统
6. 地区（用户在微信客户端设置的地区）
7. 语言（用户在微信客户端设置的语言）
8. 个性化菜单接口说明：
9. 个性化菜单要求用户的微信客户端版本在iPhone6.2.2，Android 6.2.4以上。
10. 菜单的刷新策略是，在用户进入公众号会话页或公众号profile页时，如果发现上一次拉取菜单的请求在5分钟以前，就会拉取一下菜单，如果菜单有更新，就会刷新客户端的菜单。测试时可以尝试取消关注公众账号后再次关注，则可以看到创建后的效果。
11. 普通公众号的个性化菜单的新增接口每日限制次数为2000次，删除接口也是2000次，测试个性化菜单匹配结果接口为20000次
12. 出于安全考虑，一个公众号的所有个性化菜单，最多只能设置为跳转到3个域名下的链接
13. 创建个性化菜单之前必须先创建默认菜单（默认菜单是指使用普通自定义菜单创建接口创建的菜单）。如果删除默认菜单，个性化菜单也会全部删除
14. 个性化菜单匹配规则说明：
15. 当公众号创建多个个性化菜单时，将按照发布顺序，由新到旧逐一匹配，直到用户信息与matchrule相符合。如果全部个性化菜单都没有匹配成功，则返回默认菜单。
16. 例如公众号先后发布了默认菜单，个性化菜单1，个性化菜单2，个性化菜单3。那么当用户进入公众号页面时，将从个性化菜单3开始匹配，如果个性化菜单3匹配成功，则直接返回个性化菜单3，否则继续尝试匹配个性化菜单2，直到成功匹配到一个菜单。
17. 根据上述匹配规则，为了避免菜单生效时间的混淆，决定不予提供个性化菜单编辑API，开发者需要更新菜单时，需将完整配置重新发布一轮。
18. 创建个性化菜单
    1. http请求方式：POST（请使用https协议）

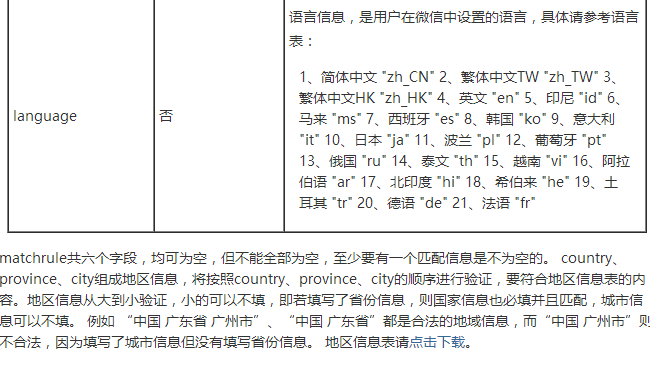
<https://api.weixin.qq.com/cgi-bin/menu/addconditional?access_token=ACCESS_TOKEN>

* 1. 请求事例



* 1. 参数说明





1. 删除个性化菜单
   1. http请求方式：POST

<https://api.weixin.qq.com/cgi-bin/menu/delconditional?access_token=ACCESS_TOKEN>

1. 测试个性化菜单匹配结果
   1. http请求方式：POST

<https://api.weixin.qq.com/cgi-bin/menu/trymatch?access_token=ACCESS_TOKEN>

* 1. 请求示例

{

"user\_id":"weixin"

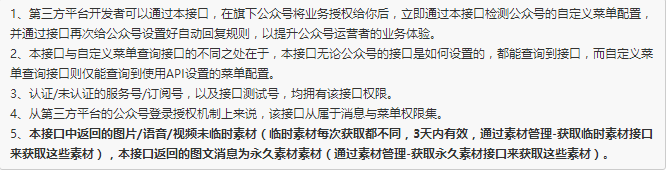
}

user\_id可以是粉丝的OpenID，也可以是粉丝的微信号。

* 1. 返回结果 该接口将返回菜单配置

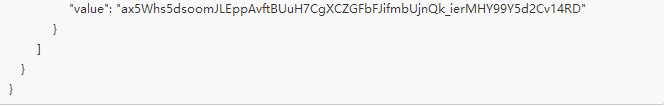


###### 49.2.6 获取公众号的菜单配置

1. 本接口将会提供公众号当前使用的自定义菜单的配置，如果公众号是通过API调用设置的菜单，则返回菜单的开发配置，而如果公众号是在公众平台官网通过网站功能发布菜单，则本接口返回运营者设置的菜单配置。
2. 
3. **接口调用请求说明**
   1. http请求方式 GET

<https://api.weixin.qq.com/cgi-bin/get_current_selfmenu_info?access_token=ACCESS_TOKEN>

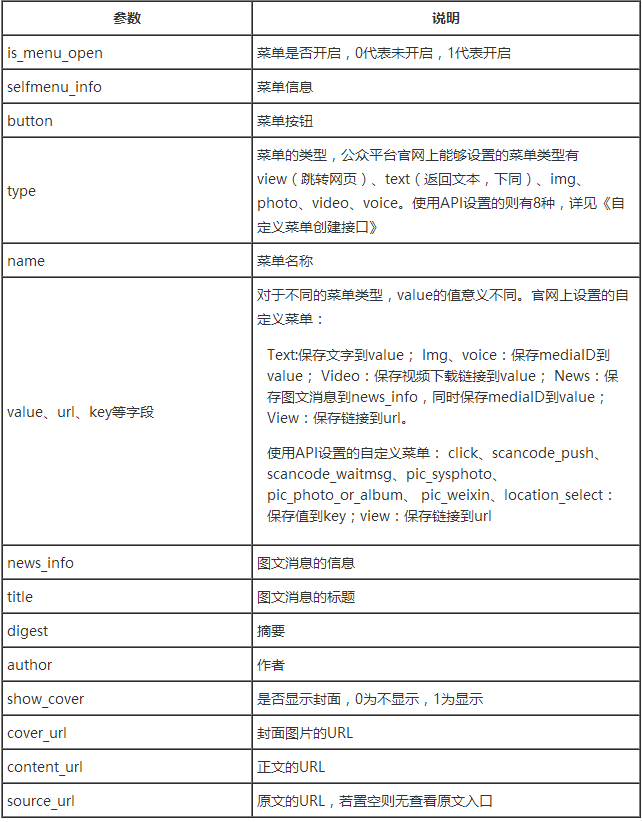
1. **返回结果说明**
   1. 如果公众号是在公众平台官网通过网站功能发布菜单，则本接口返回的自定义菜单配置样例如下：



* 1. 如果公众号是通过API调用设置的菜单，自定义菜单配置样例如下：



* 1. **参数说明**



##### 49.3 消息管理

###### 49.3.1接收消息-接收普通消息

1. 当普通微信用户向公众账号发消息时，微信服务器将POST消息的XML数据包到开发者填写的URL上。
2. 请注意：
3. 关于重试的消息排重，推荐使用msgid排重。
4. 微信服务器在五秒内收不到响应会断掉连接，并且重新发起请求，总共重试三次。假如服务器无法保证在五秒内处理并回复，可以直接回复空串，微信服务器不会对此作任何处理，并且不会发起重试。详情请见“[发送消息-被动回复消息](https://mp.weixin.qq.com/wiki/1/6239b44c206cab9145b1d52c67e6c551.html)”。
5. 为了保证更高的安全保障，开发者可以在公众平台官网的开发者中心处设置消息加密。开启加密后，用户发来的消息会被加密，公众号被动回复用户的消息也需要加密（但开发者通过客服接口等API调用形式向用户发送消息，则不受影响）。关于消息加解密的详细说明，请见“[消息加解密说明](https://mp.weixin.qq.com/wiki/11/2d2d1df945b75605e7fea9ea2573b667.html)”。
6. 各消息类型的推送XML数据包结构如下：
   1. 文本消息
   2. 图片消息
   3. 语音消息
   4. 视频消息
   5. 小视频消息
   6. 地理位置消息
   7. 链接消息
7. 文本消息

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1348831860</CreateTime>

<MsgType><![CDATA[text]]></MsgType>

<Content><![CDATA[this is a test]]></Content>

<MsgId>1234567890123456</MsgId>

</xml>



1. 图片消息

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1348831860</CreateTime>

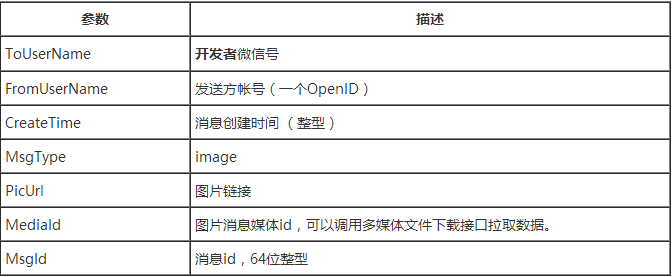
<MsgType><![CDATA[image]]></MsgType>

<PicUrl><![CDATA[this is a url]]></PicUrl>

<MediaId><![CDATA[media\_id]]></MediaId>

<MsgId>1234567890123456</MsgId>

</xml>



1. 语音消息

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1357290913</CreateTime>

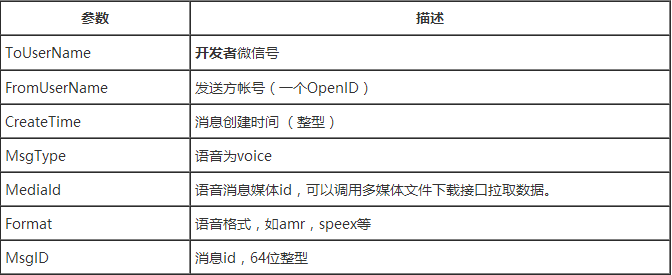
<MsgType><![CDATA[voice]]></MsgType>

<MediaId><![CDATA[media\_id]]></MediaId>

<Format><![CDATA[Format]]></Format>

<MsgId>1234567890123456</MsgId>

</xml>



请注意，开通语音识别后，用户每次发送语音给公众号时，微信会在推送的语音消息XML数据包中，增加一个Recongnition字段（注：由于客户端缓存，开发者开启或者关闭语音识别功能，对新关注者立刻生效，对已关注用户需要24小时生效。开发者可以重新关注此帐号进行测试）。开启语音识别后的语音XML数据包如下：

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1357290913</CreateTime>

<MsgType><![CDATA[voice]]></MsgType>

<MediaId><![CDATA[media\_id]]></MediaId>

<Format><![CDATA[Format]]></Format>

<Recognition><![CDATA[腾讯微信团队]]></Recognition>

<MsgId>1234567890123456</MsgId>

</xml>

多出的字段中，Format为语音格式，一般为amr，Recognition为语音识别结果，使用UTF8编码。

1. 视频消息

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1357290913</CreateTime>

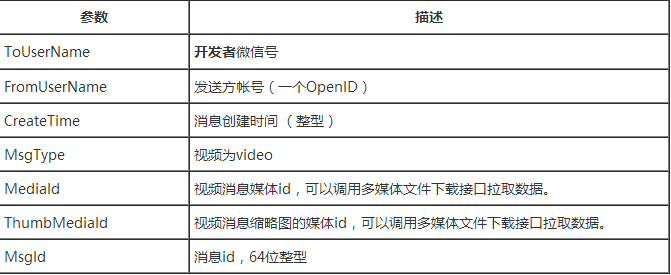
<MsgType><![CDATA[video]]></MsgType>

<MediaId><![CDATA[media\_id]]></MediaId>

<ThumbMediaId><![CDATA[thumb\_media\_id]]></ThumbMediaId>

<MsgId>1234567890123456</MsgId>

</xml>



1. 小视频消息

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1357290913</CreateTime>

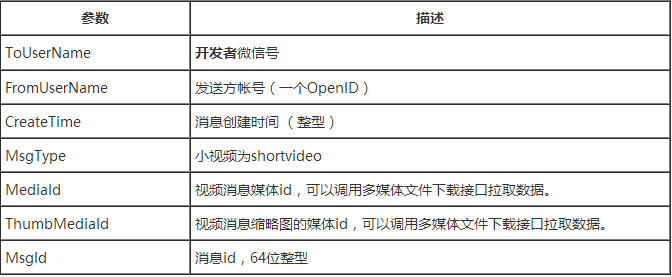
<MsgType><![CDATA[shortvideo]]></MsgType>

<MediaId><![CDATA[media\_id]]></MediaId>

<ThumbMediaId><![CDATA[thumb\_media\_id]]></ThumbMediaId>

<MsgId>1234567890123456</MsgId>

</xml>



1. 地理位置消息

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1351776360</CreateTime>

<MsgType><![CDATA[location]]></MsgType>

<Location\_X>23.134521</Location\_X>

<Location\_Y>113.358803</Location\_Y>

<Scale>20</Scale>

<Label><![CDATA[位置信息]]></Label>

<MsgId>1234567890123456</MsgId>

</xml>



1. 链接消息

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>1351776360</CreateTime>

<MsgType><![CDATA[link]]></MsgType>

<Title><![CDATA[公众平台官网链接]]></Title>

<Description><![CDATA[公众平台官网链接]]></Description>

<Url><![CDATA[url]]></Url>

<MsgId>1234567890123456</MsgId>

</xml>



###### 49.3.2 接收消息-接收事件推送

1. 在微信用户和公众号产生交互的过程中，用户的某些操作会使得微信服务器通过事件推送的形式通知到开发者在开发者中心处设置的服务器地址，从而开发者可以获取到该信息。其中，某些事件推送在发生后，是允许开发者回复用户的，某些则不允许，详细说明请见本页末尾的微信推送消息与事件说明。
2. 关注/取消关注事件
   1. 用户在关注与取消关注公众号时，微信会把这个事件推送到开发者填写的URL。方便开发者给用户下发欢迎消息或者做帐号的解绑。
   2. **微信服务器在五秒内收不到响应会断掉连接，并且重新发起请求，总共重试三次**
   3. **假如服务器无法保证在五秒内处理并回复，可以直接回复空串，微信服务器不会对此作任何处理，并且不会发起重试。**
   4. 推送XML数据包示例：

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[FromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[subscribe]]></Event>

</xml>

* 1. 参数说明



1. 扫描带参数二维码事件
   1. 用户扫描带场景值二维码时，可能推送以下两种事件：
      1. 如果用户还未关注公众号，则用户可以关注公众号，关注后微信会将带场景值关注事件推送给开发者
      2. 如果用户已经关注公众号，则微信会将带场景值扫描事件推送给开发者。
   2. **用户未关注时，进行关注后的事件推送**
      1. 推送XML数据包示例：

<xml><ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[FromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[subscribe]]></Event>

<EventKey><![CDATA[qrscene\_123123]]></EventKey>

<Ticket><![CDATA[TICKET]]></Ticket>

</xml>

* + 1. 参数说明



* 1. **用户已关注时的事件推送**
     1. 推送XML数据包示例：

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[FromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

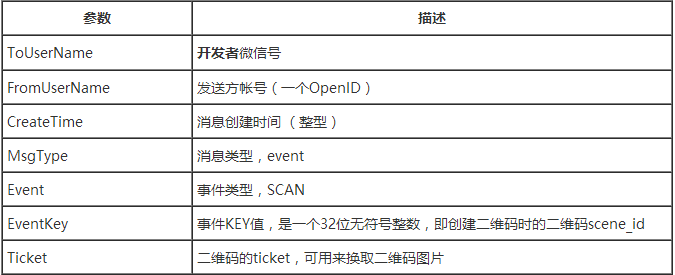
<Event><![CDATA[SCAN]]></Event>

<EventKey><![CDATA[SCENE\_VALUE]]></EventKey>

<Ticket><![CDATA[TICKET]]></Ticket>

</xml>

* + 1. 参数说明



1. 上报地理位置事件
   1. 用户同意上报地理位置后，每次进入公众号会话时，都会在进入时上报地理位置，或在进入会话后每5秒上报一次地理位置，公众号可以在公众平台网站中修改以上设置。上报地理位置时，微信会将上报地理位置事件推送到开发者填写的URL。
   2. 推送XML数据包示例

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[fromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[LOCATION]]></Event>

<Latitude>23.137466</Latitude>

<Longitude>113.352425</Longitude>

<Precision>119.385040</Precision>

</xml>

* 1. 参数说明



1. 自定义菜单事件
   1. 用户点击自定义菜单后，微信会把点击事件推送给开发者，请注意，点击菜单弹出子菜单，不会产生上报。
2. 点击菜单拉取消息时的事件推送
   1. 推送XML数据包示例：

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[FromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[CLICK]]></Event>

<EventKey><![CDATA[EVENTKEY]]></EventKey>

</xml>

* 1. 参数说明：



1. 点击菜单跳转链接时的事件推送
   1. 推送XML数据包示例

<xml>

<ToUserName><![CDATA[toUser]]></ToUserName>

<FromUserName><![CDATA[FromUser]]></FromUserName>

<CreateTime>123456789</CreateTime>

<MsgType><![CDATA[event]]></MsgType>

<Event><![CDATA[VIEW]]></Event>

<EventKey><![CDATA[www.qq.com]]></EventKey>

</xml>

* 1. 参数说明



###### 49.3.3 发送消息-被动回复消息

###### 49.3.4 发送消息-被动回复时的加解密

###### 49.3.5 发送消息-客服消息

###### 49.3.6 发送消息-群发接口

###### 49.3.7 发送消息-模板消息接口

###### 49.3.8 发送消息-模板消息运营规范

###### 49.3.9 获取公众号自动回复配置

### 50

##### 50.1

###### 50.1.1

1. 1.

# Asp.Net Core

### .NET CORE

ASP.NET Core 是一个跨平台的高性能开源框架，用于生成基于云且连接 Internet 的新式应用程序。使用ASP.NET Core，您可以：

* 创建 Web 应用程序和服务、IoT 应用和移动后端。
* 在 Windows、macOS 和 Linux 上使用喜爱的开发工具。
* 部署到云或本地。
* 在 .NET Core 或 .NET Framework 上运行。

##### 为什么使用 .NET CORE

###### 优点

1. 生成 Web UI 和 Web API 的统一场景
2. 针对可测试性进行构建。
3. Razor Pages 可以使基于页面的编码方式更简单高效。
4. 开放源代码和以社区为中心、
5. 集成新式客户端框架和开发工作流
6. 基于环境的云就绪配置系统。
7. 内置依赖项注入。
8. 轻型的高性能模块化HTTP 请求管道。
9. 能够在IIS、Nginx、Apache、Docker 上进行托管或在自己的进程中进行自托管。
10. 定目标到.NET Core 时，可以使用并行应用版本控制。
11. 简化新式Web 开发的工具。

###### 使用ASP.NET Core MVC 生成 Web API 和 Web UI

1. Model-View-Controller (MVC) 模式使Web API 和Web 应用可测试。
2. Razor Pages 是基于页面的编程模型，它让Web UI 的生成更加简单高效
3. Razor 标记提供了适用于Razor 页面和MVC 视图的高效语
4. 标记帮助程序使服务器端代码可以在Razor 文件中参与创建和呈现HTML 元素。
5. 内置的多数据格式和内容协商支持使Web API 可访问多种客户端，包括浏览器和移动设备。
6. 模型绑定自动将HTTP 请求中的数据映射到操作方法参数。
7. 模型验证自动执行客户端和服务器端验证。

###### 客户端开发

1. ASP.NET Core 与常用客户端框架和库（包括Blazor、Angular、React 和Bootstrap）无缝集成。

###### 面向 .NET Framework 的 ASP.NET Core

1. ASP.NET Core 2.x 可以面向 .NET Core 或 .NET Framework。
2. 面向.NET Framework 的ASP.NET Core 应用无法跨平台，它们仅在Windows 上运行。
3. 使用.NET Standard 2.0编写的库在实现.NET Standard 2.0 的任何.NET 平台上运行。
4. ASP.NET Core 3.0 以及更高版本只能在.NET Core 中运行
5. 与.NET Framework 相比，.NET Core 的部分优势包括：
   1. 跨平台。在macOS、Linux 和Windows 上运行。
   2. 增强的性能
   3. 并行版本控制
   4. 新API
   5. 开源

##### ASP.NET Core 2.2 新增的功能

###### 开发 API 分析器和约定

###### 错误详细信息支持

###### 终结点路由

###### 运行状况检查

###### Kestrel 对 HTTP/2 的支持

###### Kestrel 配置

###### IIS 进程内承载

###### SignalR Java 客户端

###### CORS 改进

###### 响应压缩

###### 项目模版

###### 验证性能

###### HTTP客户端性能

### RAZOR

##### WHAT

###### Razor关键字:

1. page
2. namespace
3. functions
4. inherits
5. model
6. section
7. helper

###### C# Razor 关键字

1. case
2. do
3. default
4. for
5. foreach
6. if
7. else
8. lock
9. switch
10. try
11. catch
12. finally
13. using
14. while

##### 语法参考

Razor 是一种标记语言，用于将基于服务器的代码嵌入到网页中。Razor 语法由 Razor 标记、C# 和 HTML 组成。包含 Razor 的文件通常具有 .cshtml 文件扩展名

###### 呈现HTML

1. 默认 Razor 语言为 HTML。从Razor 标记呈现 HTML 与从HTML 文件呈现为 HTML 并没有什么不同。服务器会按原样呈现 .cshtml Razor 文件中的 HTML 标记。

###### Razor 语法

1. Razor 支持 C# ,并使用 @ 符号从 HTML 转换为 C# 。 Razor 计算 C# 表达式，并将他们呈现在 HTML 输出中。
2. 当 @ 符号后跟 Razor 保留关键字时，它会转换为 Razor 特定标记。否则会转换为纯 C#
3. 若要对 Razor 标记中的 @ 符号进行转义，请使用另一个 @ 符号如：
   1. <p>@@UserName</p>
4. 包含电子邮件地址的 HTML 属性和内容不将 @ 符号视为转换字符。 Razor 分析不会处理以下示例中的电子邮件。
   1. <a href=<mailto:support@contoso.com>> Support@contoso.com</a>

###### 隐式Razor 表达式

1. 隐式 Razor 表达式以 @ 开头，后跟 C# 代码
   1. <p>@DateTime.Now</p>
   2. <p>@(new Random().Next())</p>
   3. <p>@HttpContext.User.Identity.Name</p>
2. 隐式表达式不能包含空格，但 C# await 关键字除外。如果该 C# 语句具有明确的结束标记。则可混用空格。
   1. <p>@await DoSomething("hello","world");</p>
   2. 隐式表达式不能包含 C# 泛型,因为括号（<>）内的字符会被解释为 HTML 标记。
   3. 泛型方法调用必须包装在显示 Razor 表达式或者 Razor 代码块中

###### 显示 Razor 表达式

* + - 1. 显示 Razor 表达式由 @ 符号和（）组成。若要呈现上一周的时间，如：

<p>Last week this time:@(DateTime.Now – TimeSpan.FromDays(7))</p>

* + - 1. [使用表达式输出\*\*\*@\*\*\*.com](mailto:使用表达式输出***@***.com)

@{

string emailName = "havenlau";

string emailType = "hotmail";

}

<p>@emailName@@@(emailType).com</p>

###### 表达式编码

计算结果为字符串的 C# 表达式采用 HTML 编码。计算结果为 IHtmlContent 的 C# 表达式直接通过 IHtmlContent.WriteTo 呈现。计算结果不为 IHtmlContent 的 C# 表达式通过 ToString() 转换为字符串，并在呈现前进行编码。

* + - 1. 计算结果为字符串的 C# 表达式

@("<b>Hello world!</b>")

输出：<b>Hello world!</b>

* + - 1. 计算结果为IHtmlContent 的 C# 表达式

@Html.Raw("<b>Hello world!</b>")

输出： **Hello world!**

* + - 1. 计算结果不为 IHtmlContent 的表达式

@Html.Raw("12345")

输出：12345

* + - 1. Html.Raw XSS 攻击

@Html.Raw("<script>window.alert('你被攻击了')</script>")

###### Razor 代码块

Razor 代码块以 @ 开头，并括在｛｝中。代码块内的 C# 代码不会被呈现。这点与表达式不一样，一个视图中的代码块和表达式共享相同的作用域并按顺序进行定义

* + - 1. 输出内容

@{

var quote = "this is a quote.";

}

<p>@quote</p>

@{

quote = "this is a a a a quote";

}

<p>@quote</p>

* + - 1. 本地函数

@{

string PrintMyString(string mystring)

{

return string.Format($"<b>your string is {mystring}</b>");

}

Html.Raw(PrintMyString("aaaaaaa"));

Write(PrintMyString("heeeeee"));

}

@Html.Raw(PrintMyString("yyyyyyy"))

输出：<b>your string is heeeeee</b> **your string is yyyyyyy**

* + - 1. 隐式转换。代码块中的默认语言为 C# ，不过， Razor 页面可以转回 HTML

@{

var inCSharp = true;

<p>Now in HTML, was in C# @inCSharp</p>

}

* + - 1. 待分隔符的显示转换

若要定义呈现HTML的代码子节，请使用Razor <text>标记将要呈现的字符括起来：

@{

var list = new List<string> { "a", "b", "c", "d" };

for (int i = 0; i < list.Count; i++)

{

var str = list[i];

<text><p>我要在表达式里输出数据了：@str</p></text>

@:<p>@@:输出：@str</p>

}

}

输出：

我要在表达式里输出数据了：a

@:输出：a

我要在表达式里输出数据了：b

@:输出：b

我要在表达式里输出数据了：c

@:输出：c

我要在表达式里输出数据了：d

@:输出：d

###### 控制结构

控制结构是对代码块的扩展。代码块的各个方面（转换为标记、内联C#）同样适用于以下结构：

* + - 1. 条件语句 @if 、else if 、else 和 @switch

else 和 else if 之间不需要 @

* + - 1. 循环语句 @for、@foreach、@while、@do while
      2. 复合语句 @using
      3. 异常处理 @try、catch、finally
      4. @lock 语句

###### 指令

Razor 指令由隐式表达式表示：@ 符号后跟保留关键字。指令通常用于更改视图分析方式或启用不同的功能。

1. @using 指令用于向生成的视图中添加 C# Using指令：
2. @model 指令指定传递到视图的模型类型
3. @inherits 指令对视图继承的类提供完全控制

@inherits 和 @model可以在同一视图中使用。

@inherits 可位于视图导入的 \_ViewImports.cshtml文件中

1. @inject 指令允许Razor 页面将服务从服务容器注入到视图
2. @function 指令允许 Razor 页面将C# 代码块添加到视图中
3. @section 指令与布局结合使用，允许视图将内容呈现在 HTML 页面的不同部分。

###### 模版化 Razor 委托

1. 定义一个执行模版化引擎的委托
2. 传入执行数据和模版

###### 标记帮助程序

1. 相关指令
   1. @addTagHelper向视图提供标记帮助程序
   2. @removeTagHelper从视图中删除以前添加的标记帮助程序
   3. @tagHelperPrefix指定标记前缀，以启用标记帮助程序支持并阐明标记帮助程序的用法。

###### 检查为视图生成的 Razor C# 类

###### 视图查找和区分大小写

1. Razor 视图引擎为视图执行区分大小写的查找。但是，实际查找取决于基础文件系统：
   1. 基于文件的源：
      1. 在使用不区分大小写的文件系统的操作系统上（windows），物理文件提供程序查找不区分大小写。例如，return View(“Test”) 可匹配 /Views/Home/Test.cshtml、/Views/home/test.cshtml 以及任何其他大小写变体
      2. 在区分大小写的文件系统的操作系统上（Linux、OSX）以及使用 EmbeddedFileProvider构建的文件系统)上，查找区分大小写。例如 return View(“Test”) 专门匹配 /View/Home/Test.cshtml
   2. 预编译视图。在 ASP.NET Core 2.0 及更高版本中，预编译视图查找在所有操作系统上均不区分大小写。该行为与Windows上物理文件提供程序的行为相同。如果两个预编译视图仅大小写不同，则查找的结果具有不确定性。
2. 建议开发人员将文件和目录名称的大小写与以下项的大小写匹配：
   1. 区域、控制器和操作名称
   2. Razor 页面
3. 匹配大小写可确保无论使用哪种基础文件系统，部署都能找到其视图。

##### 视图组件

视图组件与分部视图类似，但他们的功能更加强大。视图组件不适用模型绑定,并且仅依赖调用时提供的数据。视图组件也适用于Razor Pages

###### 视图组件

1. 呈现一个区块，而不是整个响应
2. 包括控制器和视图间发现的相同关注点分离和可测试性优势
3. 可以由参数和业务逻辑
4. 通常从布局页调用

###### 视图组件可以用于具有可重用呈现逻辑（对分部视图来说过于复杂）的任何位置

1. 动态导航菜单
2. 标记云（查询数据库的位置）
3. 登录面板
4. 购物车
5. 最近发布的文章
6. 典型博客上的边栏内容
7. 一个登录面板，呈现在每页上并显示注销或登录链接，具体决定于用户的登录状态

###### 视图组件由两部分组成：

1. 类（通常派生自ViewComponent）
2. 及其返回的结果（通常为视图）
3. 与控制器一样，视图组件也可以时POCO，但大多数开发人员都希望利用派生自ViewComponent的可用方法和属性

###### 创建视图组件

1. 视图组件类可通过以下任一方法创建视图组件类
   1. 从ViewComponent派生
   2. 使用 [ViewComponent]属性修饰类，或者从具有[ViewComponent]属性的类派生
   3. 创建名称以ViewComponent后缀结尾的类
2. 与控制器一样，视图组件必须时公共，非嵌套和非抽象的类。视图组件名称是删除了”ViewComponent” 后缀的类名。也可以使用ViewComponentAttribute.Name属性显示指定它。
3. 视图组件类：
   1. 完全支持构造函数依赖关系注入
   2. 不参与控制器生命周期，这意味着不能在视图组件中使用筛选器

###### 视图组件方法

视图组件以返回Task<IViewComponentResult>的InvokeAsync方法。或是以返回 IViewComponentResult的同步 Invoke方法定义其逻辑。参数直接来自视图组件的调用，而不是来自模型绑定。视图组件从不直接处理请求。通常，视图组件通过调用View方法来初始化模型并将其传递到视图

1. 定义返回Task<IViewComponentResult>的InvokeAsync方法，或是返回IViewComponentResult的同步Invoke方法
2. 一般通过调用ViewComponent View 方法来初始化模型并将其传递到视图。
3. 参数来自调用方法，而不是HTTP。没有模型绑定
4. 不可直接作为HTTP终结点进行访问。通过代码调用他们（通常在视图中）。视图组件从不处理请求。
5. 在签名上重载，而不是当前HTTP请求的任何详细信息

###### 视图搜索路径

1. 运行时搜索路径：
   1. /Views/{Controller Name}/Components/{View Component Name}/{View Name}
   2. /Views/Shared/Components/{View Component Name}/{View Name}
   3. /Pages/Shared/Components/{View Component Name}/{View Name}
2. 搜索路径适用于使用控制器+视图和Razor Pages的项目
   1. 视图组件的默认视图名称为默认，这意味着文件通常名为“Default.cshtml”.可以在创建视图组件结果或调用View方法时指定不同的视图名称
   2. 建议将视图文件命名为Default.cshtml并使用 Views/Shared/Components/{View Component Name}/{View Name}路径。此示例中使用的 PriorityList 视图组件对视图组件视图使用。Views/Shared/Components/PriorityList/Default.cshtml

###### 调用视图组件

1. 使用视图组件：
   1. 要使用视图组件，请在视图中调用以下内容：

@await Component.InvokeAsync(“Name of view component”,{Anonymous Type Containing Parameters})

* 1. 参数传递给InvokeAsync()方法，本文中开发的 PriorityList 视图组件调用子 Views/ToDo/Index.cshmlt视图文件。使用两个参数调用 InvokeAsync()方法

@await Component.InvokeAsync(“PriorityList”,new {maxPriority = 4,isDone = true})

###### 调用视图组件作为标记帮助程序

1. 调用视图组件作为标记帮助程序：
   * + - 1. <vc:priority-list max-priority=”2” is-done=”false”></ vc:priority-list>
         2. 标记帮助程序采用Pascal大小写格式的类和方法参数将转换为各自相应的短横线格式。要调用视图组件的标记帮助程序使用<vc></vc>元素。

<vc:[view-component-name] parameter1=”parameter1 value” parameter2=”parameter2 value”>

</vc:[view-component-name]>

* + - * 1. 要将视图组件用作标记帮助程序，请使用@addTagHelper指令注册包含视图组件的程序集。如果视图组件位于”MyWebApp”的程序集中，请将以下指令添加到\_ViewImports.cshtml文件

@addTagHelper \*， MyWebApp

* + - * 1. 可将视图组件作为标记帮助程序注册到任何引用视图组件的文件。

@await Component.InvokeAsync(“PriorityList”,new {maxPriority = 4,isDone = true})

<vc:priority-list max-priority=”2” is-done=”false”></ vc:priority-list>

###### 从控制器直接调用视图组件

1. 视图组件通常从视图调用，但时可以直接从控制器方法调用。尽管视图组件不定义控制器等终结点，但可以轻松实现返回 ViewComponentResult内容的控制器操作
   1. public IActionResult IndexVC()

{

return ViewComponent(“PriorityList”,new {maxPriority = 4,isDone = true});

}

###### 演练

1. 创建视图组件类
   1. 添加ViewComponent类
   2. public class PriorityListViewComponent:ViewComponent{}
2. 创建视图组件Razor视图
   1. 创建Views/Shared/Components文件夹。此文件夹必须命名为Components
   2. 创建Views/Shared/Components/PriorityList 文件夹。此文件夹名称必须与视图组件类的名称或者类名去掉后缀（如果为AAAViewComponent 未使用[ViewComponent(“BBB”)]则名字为AAA 否则必须为BBB）
   3. 创建Views/Shared/Components/PriorityList/Default.cshtml Razor视图。Razor视图获取并显示TodoItem列表。如果视图组件InvokeAsync()不传递视图名称，则按照约定使用Default作为视图名称
   4. 要替代特定控制器的默认样式，请将视图添加到控制器特定的视图文件夹（Views/ToDo/Components/PriorityList/Default.cshtml）
   5. 如果视图组件是控制器特定的，则可将其添加到控制器特定的文件夹（Views/ToDo/Components/PriorityList/Default.cshtml）
   6. 从视图调用视图组件

<div>

@await Component.InvokeAsync(“PriorityList”,new {maxPriori = 2, isDone = false}

</div>

* 1. 从控制器调用视图组件

public IActionResult IndexVC()

{

return ViewComponent(viewName,new { maxPriority = 3,isDone = false})

}

* 1. 使用标记帮助程序调用视图组件

@addTagHelper \*, RazorPagesMovie

<**vc:priority-list** **max-price**="65" **is-on**="false"></**vc:priority-list**>

1. 说明：
   1. 视图组件类可以包含在项目的任意文件夹内
   2. 类名以AAAViewComponent ViewComponent的后缀结尾，运行时将在从视图引用类组件时使用字符串AAA。
   3. 使用[ViewComponet(Name=”PriorityList”)]属性可以更改用于引用视图组件的名称
   4. [ViewComponent]属性通知视图组件选择器在查找与组件相关联的视图时使用名称AAA，以及在从视图引用组件类型时使用字符串“AAA”。
   5. 组件使用依赖关系注入以使数据上下文可用
   6. InvokeAsync()公开可以从视图调用的方法，且可以采用任意数量的参数
   7. InvokeAsync()方法返回满足isDone和maxPriority参数的ToDo参数
2. 视图组件类

##### Razor 文件编译

###### Razor编译

1. 调用相关的MVC视图 和 Razor页时，Razor文件在运行时编译。不支持在生成时发布Razor文件。可以使用预编译工具，在生成和发布时使用Razor SDK 编译Razor文件并将其与应用一起部署
2. Razor SDK默认启用Razor 文件的生成时和发布时编译。Razor 文件更新后，支持在生成时编辑这些文件。默认情况下，只有编译Razor文件所需的编译的Views.dll（而非.cshtml）文件或者引用程序集随应用一起部署

###### 运行时编译

1. 通过Razor文件的运行时编译补充生成时编译。RazorViewEngineOptions的AllowRecompilingViewsOnFileChange 获取或者设置一个值，该值确定当磁盘上的文件发生更改时是否重新编译和更新Razor文件（Razor视图和Razor Pages）

##### 应用程序模型

Asp.Net Core MVC 会定义一个应用程序模型，用于表示MVC应用的各个组件。通过读取和处理此模型可修改MVC元素的各个行为。默认情况下，MVC 遵循特定的约定，以确保哪些视作控制器，类上的哪些方法是操作，以及参数和路由的行为方式。

###### 模型和提供程序

ASP.NET Core MVC 应用程序模型包括用于描述 MVC 应用程序的抽象接口和具体实现类。 此模型是 MVC 根据默认约定发现应用的控制器、操作、操作参数、路由和筛选器的结果。 通过使用应用程序模型，可以修改应用以遵循与默认 MVC 行为不同的约定。 参数、名称、路由和筛选器都用作操作和控制器的配置数据。

1. ASP.NET Core MVC 应用程序模型具有以下结构：

ApplicationModel >>>

控制器（ControllerModel）>>>

操作（ActionModel）>>>

参数（ParameterModel）

1. 该模型的每个级别都有权访问公用Properties集合，层次结构中的较低级别可以访问和覆盖由较高级别设置的属性值。创建操作时，属性保存到ActionDescriptor.Properties.
2. 当处理请求时，可通过ActionContext.ActionDescriptor.Properties 访问某个约定添加或修改的任何属性。若要基于每项操作对筛选器、模型绑定等进行配置，使用属性不失为一个好方法。

###### IApplicationModelProvider

1. Asp.Net Core MVC 使用提供程序模式（由IApplicationModelProvider接口定义）加载应用程序模型。
   1. IApplicationModelProvider接口实现相互包装，每个实现都基于其Order属性以升序调用OnProvidersExecuting.然后，按相反的的顺序调用 OnProvidersExecuted方法
   2. 首先（Order=-1000）DefaultApplicationModelProvider
   3. 然后（Order=-990）AuthorizationApplicationModelProvider

CorsApplicationModelProvider

1. DefaultApplicationModelProvider
   1. 将全局筛选器添加到上下文
   2. 将控制器添加到上下文
   3. 将公共控制器方法作为操作添加
   4. 将操作方法参数添加到上下文
   5. 应用路由和其他属性
   6. 某些内置行为由 DefaultApplicationModelProvider 实现。此提供程序负责构造ControllerModel，后者又引用ActionModel、PropertyModel和ParameterModel实例。DefaultApplicationModelProvider类是一个内部框架实现细节，未来将对其进行更改
2. AuthorizationApplicationModelProvider
   1. 负责应用与AuthorizeFilter和AllowAnonymousFilter属性关联的行为
3. CorsApplicationModelProvider
   1. 可实现与 IEnableCorsAttribute 、IDisableCorsAttribute 和DisableCorsAuthorizationFilter 关联的行为

###### 约定

1. 应用程序模型定义了抽象对象，通过约定抽象对象来自定义模型行为比重写整个模型或提供程序更简单。建议使用这些抽象来修改应用的行为
2. 通过使用约定可以编写能动态应用自定义项的代码。使用筛选器可以修改框架的行为，而利用自定义项可以控制整个应用连接在一起的方式
3. 可用约定如下：
   1. IApplicationModelConvention
   2. IControllerModelConvention
   3. IActionModelConvention
   4. IParameterModelConvention
4. 应用约定：
   1. 将他们添加到MVC选项。或实现Attribute 并将他们应用于控制器，操作或者操作参数（类似于Filters）。与筛选器不同的时，约定尽在应用启动时执行，而不作为每个请求的一部分执行
5. 向应用程序添加属性
   1. 第一步：

public class ApplictionDescription : IApplicationModelConvention

{

private readonly string \_description;

public ApplictionDescription(string description)

{

\_description = description;

}

public void Apply(ApplicationModel application)

{

application.Properties["description"] = \_description;

}

}

* 1. 第二步：

services.AddMvc(options =>

{

options.Conventions.Add(new ApplictionDescription("HavenLau's Application Description"));

options.Conventions.Add(new NamespaceRoutingConvention());

});

* 1. 第一步：

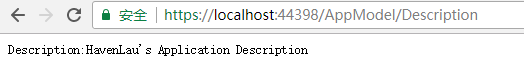
public string Description()

{

return "Description:" + ControllerContext.ActionDescriptor.Properties["description"];

}

* 1. 第四步：



1. 修改控制器模型，以包含自定义属性。这些属性将覆盖应用程序模型中指定的具有相同名称的现有属性。以下约定属性可以在控制器级别添加说明
   1. 第一步：

public class ControllerDescriptionAttribute: Attribute, IControllerModelConvention

{

private readonly string \_description;

public ControllerDescriptionAttribute (string description)

{

\_description = description;

}

public void Apply(ControllerModel controller)

{

controller.Properties["description"] = \_description;

}

}

* 1. 第二步：

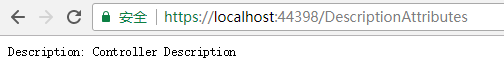
public string Index()

{

return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];

}

* 1. 第三步：



1. 修改ActionModel说明，可向各项操作应用不同的属性约定，并覆盖已在应用程序或控制器级别应用的行为
   1. 第一步：

public class ActionDescriptionAttribute: Attribute, IControllerModelConvention

{

private readonly string \_description;

public ControllerDescriptionAttribute (string description)

{

\_description = description;

}

public void Apply(ControllerModel controller)

{

controller.Properties["description"] = \_description;

}

}

* 1. 第二步：

[ActionDescription("Action Description")]

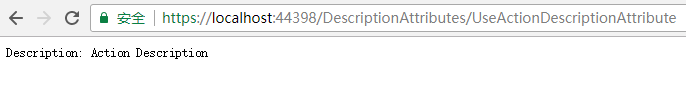
public string UseActionDescriptionAttribute()

{

return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];

}

* 1. 第三步：



1. 修改ParameterModel。可以将以下约定应用于操作参数，以修改其BindingInfo。以下约定要求参数为路由参数；忽略其他可能的绑定源（比如查询字符串）
   1. 第一步：

public class MustBeInRouteParameterModelConvention: Attribute, IParameterModelConvention

{

public void Apply(ParameterModel parameter)

{

if (parameter.BindingInfo == null)

{

parameter.BindingInfo = new BindingInfo();

}

parameter.BindingInfo.BindingSource = BindingSource.Path;

}

}

* 1. 第二步：

public class ParameterModelController : Controller

{

public string GetById([MustBeInRouteParameterModelConvention]int id)

{

return $"Bound to id:{id}";

}

}

* 1. 第三步：

/ParameterModel/GetById/123 正确

/ParameterModel/GetById/?id=123 错误

##### 应用程序模型在WebApiCompatShim中的使用

ASP.NET Core MVC 使用一组不同于 ASP.NET Web API 2 的约定。 使用自定义约定，可以修改 ASP.NET CoreMVC 应用的行为，使其与 Web API 应用保持一致。 Microsoft 附带了专用于此的 WebApiCompatShim

###### 使用

1. 若要使用 Web API Compatibility Shim，需将该包添加到项目中，然后通过调用 Startup 中的AddWebApiConventions ，将约定添加到 MVC

services.AddMvc().AddWebApiConventions();

1. 该填充程序提供的约定仅适用于应用中已应用特定属性的部分。 以下四个属性用于控制哪些控制器应使用该填充程序的约定来修改自己的约定：
   1. UseWebApiActionConventionsAttribute
   2. UseWebApiOverloadingAttribute
   3. UseWebApiParameterConventionsAttribute
   4. UseWebApiRoutesAttribute
2. 操作约定

UseWebApiActionConventionsAttribute 用于根据名称将 HTTP 方法映射到操作（例如， Get 将映射到 HttpGet ）。 它仅适用于不使用属性路由的操作。

1. 重载

UseWebApiOverloadingAttribute 用于应用 WebApiOverloadingApplicationModelConvention 约定。 此约定可向操作选择过程添加 OverloadActionConstraint ，以将候选操作限制为其请求满足所有非可选参数的操作。

1. 参数约定

UseWebApiParameterConventionsAttribute 用于应用 WebApiParameterConventionsApplicationModelConvention 操作约定。 此约定指定用作操作参数的简单类型默认来自 URI，而复杂类型来自请求正文。

1. 路由

UseWebApiRoutesAttribute 控制是否应用 WebApiApplicationModelConvention 控制器约定。 启用后，此约定用于向路由添加对区域的支持。

1. System.Web.Http.ApiController 基类，用于替换 Web API 提供的等效项。

这允许针对 Web API 编写并且继承自 ApiController 的控制器在 ASP.NET Core MVC 上运行时，能够按照设计的方式运行。 此控制器基类使用上面列出的所有 UseWebApi\* 属性进行修饰。 ApiController 公开了与在 Web API 中找到的属性、方法和结果类型兼容的属性、方法和结果类型。

##### 使用 ApiExplorer 记录应用

###### 应用程序模型在每个级别公开了 ApiExplorer 属性，该属性可用于遍历应用的结构。

1. 这可用于使用 Swagger 等工具为 Web API 生成帮助页。 ApiExplorer 属性公开了 IsVisible 属性，后者可设置为指定应公开的应用模型部分
2. 使用此方法（和附加约定，如有需要），可以在应用中的任何级别启用或禁用 API 可见性。

public class EnableApiExplorerApplicationConvention : IApplicationModelConvention

{

public void Apply(ApplicationModel application)

{

application.ApiExplorer.IsVisible = true;

}

}

##### 筛选器

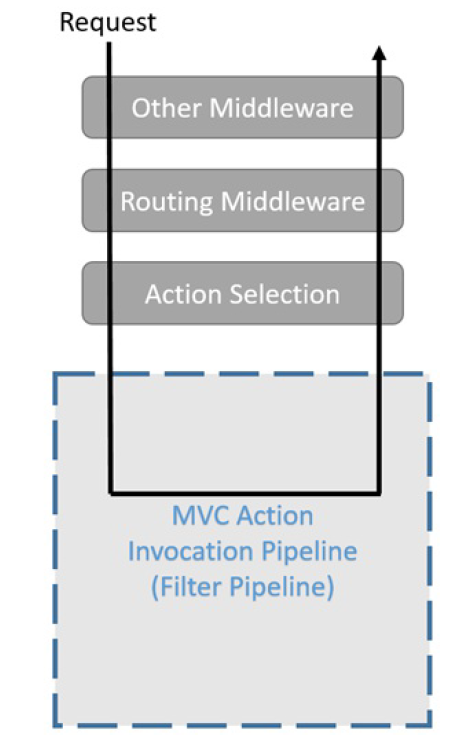
通过使用Asp.Net Core MVC 中的筛选器，可以在请求管道中的特定阶段之前或之后运行代码。可以创建自定义筛选器，用于处理横切关注点。筛选器可以避免跨操作复制代码。

###### 内置筛选器处理任务

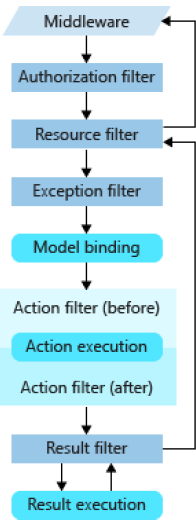
1. 授权（防止用户访问未获授权的资源）
2. 确保所有请求都使用HTTPS
3. 响应缓存（对请求管道进行短路出路，以便返回缓存的响应）

###### 工作原理

1. 筛选器在MVC操作调用管道（筛选器管道）内运行。筛选器管道在MVC选择了要执行的操作之后运行



1. 筛选器类型。每种筛选器都在筛选管道中的不同阶段执行。
   1. 授权筛选器最先执行，用于确定是否已经针对当前用户授权。如果请求未获授权，他们可以让管道短路
   2. 资源筛选器是授权后最先处理请求的筛选器。 它们可以在筛选器管道的其余阶段运行之前以及管道的其余阶段完成之后运行代码。 出于性能方面的考虑，可以使用它们来实现缓存或以其他方式让筛选器管道短路。 它们在模型绑定之前运行，所以可以影响模型绑定。
   3. 操作筛选器可以在调用单个操作方法之前和之后立即运行代码。 它们可用于处理传入某个操作的参数以及从该操作返回的结果。 不可在 Razor Pages 中使用操作筛选器。
   4. 异常筛选器用于在向响应正文写入任何内容之前，对未经处理的异常应用全局策略。
   5. 结果筛选器可以在执行单个操作结果之前和之后立即运行代码。 仅当操作方法成功执行时，它们才会运行。 对于必须围绕视图或格式化程序的执行的逻辑，它们很有用。
   6. 执行顺序



###### 实现

1. 通过不同的接口定义，筛选器同时支持同步和异步实现
2. 可在其管道阶段之前和之后运行代码的同步筛选器定义OnStageExecuting方法和OnStageExecuted方法
3. 异步筛选器定义单一的OnStageExecutingAsync方法。此方法采用 FilterTypeExecutionDelegate 委托来执行筛选器的管道阶段。
4. 可以在多个类中为多个筛选器阶段实现接口。例如：ActionFilterAttribute类实现IActionFilter、IResultFilter及其异步等接口

###### IFilterFactory

1. IFilterFactory 实现IFilterMetadata。因此，IFilterFactory 实例可在筛选器管道中的任意位置用作IFilterMetadata实例。
2. 当该框架准备调用筛选器时，它会尝试将其转换为IFilterFactory。如果强制转换成功，则调用CreateInstance 方法来创建将调用的IFilterMetadata实例。这提供了一种很灵活的设计，因为无需在应用启动时显示设置精确的筛选器管道
3. 筛选器属性
   1. ActionFilterAttribute
   2. ExceptionFilterAttribute
   3. ResultFilterAttribute
   4. FormatFilterAttribute
   5. ServicesFilterAttribute
   6. TypeFilterAttribute

###### 筛选器作用域和执行顺序

1. 使用筛选器
   1. 可以将筛选器添加到管道中的三个作用域之一
   2. 可以使用属性将筛选器添加到特定的操作方法或控制器类
   3. 可以注册所有控制器和操作的全局筛选器。（通过将筛选器添加到ConfigureServices中的MvcOptions.Filters）集合，可以将其添加为全局筛选器
2. 默认执行顺序

当管道的某个特定阶段由多个筛选器时，作用域可确定筛选器执行的默认顺序。

全局筛选器>>>类筛选器>>>方法筛选器

1. 全局筛选器（Global） OnActionExecuting

2. 控制器 OnActionExecuting

3. 方法 OnActionExecuting

4. 方法 OnActionExecuted

5. 控制器 OnActionExecuted

6. 全局筛选器（Global） OnActionExecuted

1. 重写默认顺序
   1. 通过重写筛选器的Order属性。值越小 Executing越先执行

###### 取消和设置短路

1. 通过设置提供给筛选器方法的context参数上的Result属性。可以在筛选器管道的任意位置设置短路。

public void OnResourceExecuting(ResourceExecutingContext context)

{

context.Result = new ContentResult()

{

Content = “Resource unavailable – header should not be set”

}

}

###### 添加筛选器

1. 可按类型或实例添加筛选器。
   1. 添加实例，该实例将用于每个请求
   2. 添加类型，则将激活类型，这意味着将为每个请求创建一个实例。并且依赖关系注入将填充所有构造函数的依赖项。按类型添加器等效于 filters.Add(new TypeFilterAttribute(typeof(MyFilter)))
   3. 如果将筛选器作为属性实现并直接添加到控制器类或者操作方法中。则不能使用DI提供构造函数依赖项。因为属性在应用时必须提供自己的构造函数参数
2. 如果筛选器具有一些需要从DI访问的依赖项，有以下方法可以用。
   1. 可以实现以下接口之一，将筛选器应用于类或操作方法

ServiceFilterAttribute

TypeFilterAttribute

在属性上实现的 IFilterFactory

* 1. ServiceFilterAttribute

options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader", "Result filter added to MvcOptions.Filters")); //added by an instance

options.Filters.Add(typeof(SampleActionFilter)); //added by type

services.AddScoped<AddHeaderWithFactoryAttribute>(); //added by DI

1. 使用ServiceFilterAttribute时，IsReusable 设置会提示，筛选器实例可能在其创建的请求范围之外被重用。
2. 该框架不保证在稍后的某个时刻将创建筛选器的单个实例，或不会从DI容器重新请求筛选器。如果使用的筛选器依赖于具有除单一实例以外的生命周期的服务，请避免使用IsReusable
3. 使用ServiceFilterAttribute时，不注册筛选器类型会引发异常：

No service for type ‘’ has been regiested。

1. ServiceFilterAttribute 可实现 IFilterFactory。IFilterFactory公开用于创建IFilterMetadata实例的CreateInstance方法。CreateInstance方法从服务容器（DI）中加载指定的类型
   1. TypeFilterAttribute
2. TypeFilterAttribute 与 ServiceFilterAttribute 类似，但不会直接从 DI 容器解析其类型。 它使用Microsoft.Extensions.DependencyInjection.ObjectFactory 对类型进行实例化。
3. 使用 TypeFilterAttribute 引用的类型不需要先注册在容器中。 它们具备由容器实现的依赖项。
4. TypeFilterAttribute 可以选择为类型接受构造函数参数。
5. 使用 TypeFilterAttribute 时， IsReusable 设置会提示：筛选器实例可能在其创建的请求范围之外被重用。 该框架不保证将创建筛选器的单一实例。 如果使用的筛选器依赖于具有除单一实例以外的生命周期的服务，请避免使用 IsReusable 。

###### 授权筛选器

1. 授权筛选器：
   1. 控制对操作方法的访问。
   2. 是筛选器管道中要执行的第一个筛选器。
   3. 具有在它之前的执行的方法，但没有之后执行的方法。
2. 用户只有在编写自己的授权框架时，才应编写自定义授权筛选器。 建议配置授权策略或编写自定义授权策略，而不是编写自定义筛选器。 内置筛选器实现只负责调用授权系统。
3. 切勿在授权筛选器内引发异常，因为没有任何能处理该异常的组件（异常筛选器不会进行处理）。 在出现异常时请小心应对。

###### 资源筛选器

1. 说明
   1. 实现 IResourceFilter 或 IAsyncResourceFilter 接口，
   2. 它们的执行会覆盖筛选器管道的绝大部分。
   3. 只有授权筛选器在资源筛选器之前运行。
2. 如果需要使某个请求正在执行的大部分工作短路，资源筛选器会很有用。 例如，如果响应在缓存中，则缓存筛选器可以绕开管道的其余阶段。
3. 前面所示的短路资源筛选器便是一种资源筛选器。 另一个示例是 DisableFormValueModelBindingAttribute：
   1. 可以防止模型绑定访问表单数据。
   2. 如果要上传大型文件，同时想防止表单被读入内存，那么此筛选器会很有用。

###### 操作筛选器

1. 说明
   1. 实现 IActionFilter 或 IAsyncActionFilter 接口，
   2. 它们的执行围绕着操作方法的执行。
2. ActionExecutingContext 提供以下属性：
   1. ActionArguments ：用于处理对操作的输入。
   2. Controller ：用于处理控制器实例。
   3. Result ：设置此属性会使操作方法和后续操作筛选器的执行短路。 引发异常也会阻止操作方法和后续筛选器的执行，但会被视为失败，而不是一个成功的结果。
3. ActionExecutedContext 提供 Controller 和 Result 以及以下属性：
   1. Canceled ：如果操作执行已被另一个筛选器设置短路，则为 true。
   2. Exception ：如果操作或后续操作筛选器引发了异常，则为非 NULL 值。 将此属性设置为 NULL 可有效地“处理”异常，并且将执行 Result ，就像它是从操作方法正常返回的一样。
4. 对于 IAsyncActionFilter ，一个向 ActionExecutionDelegate 的调用可以达到以下目的
   1. 执行所有后续操作筛选器和操作方法。
   2. 返回 ActionExecutedContext 。
5. 若要设置短路，可将 ActionExecutingContext.Result 分配到某个结果实例，并且不调用ActionExecutionDelegate
6. 操作筛选器可用于验证模型状态，并在状态为无效时返回任何错误：
7. OnActionExecuted 方法在操作方法之后运行，可通过 ActionExecutedContext.Result 属性查看和处理操作结果。
8. 如果操作执行已被另一个筛选器设置短路，则 ActionExecutedContext.Canceled 设置为 true。 如果操作或后续操作筛选器引发了异常，则 ActionExecutedContext.Exception 设置为非 NULL 值。 将ActionExecutedContext.Exception 设置为 null：
   1. 有效地“处理”异常。
   2. 执行 ActionExecutedContext.Result ，从操作方法中将它正常返回。

###### 异常筛选器

1. 说明
   1. 异常筛选器可实现 IExceptionFilter 或 IAsyncExceptionFilter 接口。 它们可用于为应用实现常见的错误处理策略。
   2. 代码

public override void OnException(ExceptionContext context)

{

if (!\_hostingEnvironment.IsDevelopment())

{

return;

}

var result = new ViewResult { ViewName = "CustomError" };

result.ViewData = new Microsoft.AspNetCore.Mvc.ViewFeatures.ViewDataDictionary(\_modelMetadataProvider, context.ModelState);

result.ViewData.Add("Exception", context.Exception);

context.Result = result;

}

1. 异常筛选器：
   1. 没有之前和之后的事件。
   2. 实现 OnException 或 OnExceptionAsync 。
   3. 处理控制器创建、模型绑定、操作筛选器或操作方法中发生的未经处理的异常。
   4. 请不要捕获资源筛选器、结果筛选器或 MVC 结果执行中发生的异常。
   5. 若要处理异常，请将 ExceptionContext.ExceptionHandled 属性设置为 true，或编写响应。 这将停止传播异常。异常筛选器无法将异常转变为“成功”。 只有操作筛选器才能执行该转变。
   6. 非常适合捕获发生在 MVC 操作中的异常。
   7. 并不像错误处理中间件那么灵活。
   8. 建议使用中间件处理异常。 仅在需要根据所选 MVC 操作以不同方式执行错误处理时，才使用异常筛选器。例如，应用可能具有用于 API 终结点和视图/HTML 的操作方法。 API 终结点可能返回 JSON 形式的错误信息，而基于视图的操作可能返回 HTML 形式的错误页。
   9. ExceptionFilterAttribute 可以子类化

###### 结果筛选器

1. 说明
   * + - 1. 实现接口：

IResultFilter 或 IAsyncResultFilter 。

IAlwaysRunResultFilter 或 IAsyncAlwaysRunResultFilter

* + - * 1. 它们的执行围绕着操作结果的执行。
        2. 代码：

public class AddHeaderFilterWithDi : IResultFilter

{

private ILogger \_logger;

public AddHeaderFilterWithDi(ILoggerFactory loggerFactory)

{

\_logger = loggerFactory.CreateLogger<AddHeaderFilterWithDi>();

}

public void OnResultExecuted(ResultExecutedContext context)

{

// Can't add to headers here because response has already begun.

}

public void OnResultExecuting(ResultExecutingContext context)

{

var headerName = "OnResultExecuting";

context.HttpContext.Response.Headers.Add(

headerName, new string[] { "ResultExecutingSuccessfully" });

\_logger.LogInformation($"Header added:{headerName}");

}

}

* + - * 1. 要执行的结果类型取决于所执行的操作。 返回视图的 MVC 操作会将所有 Razor 处理作为要执行的ViewResult 的一部分。 API 方法可能会将某些序列化操作作为结果执行的一部分。
        2. 当操作或操作筛选器生成操作结果时，仅针对成功的结果执行结果筛选器。 当异常筛选器处理异常时，不执行结果筛选器。
        3. OnResultExecuting 方法可以将 ResultExecutingContext.Cancel 设置为 true，使操作结果和后续结果筛选器的执行短路。 设置短路时，通常应写入响应对象，以免生成空响应。 如果引发异常，则会导致：

阻止操作结果和后续筛选器的执行。

结果被视为失败而不是成功。

* + - * 1. 当 OnResultExecuted 方法运行时，响应可能已发送到客户端，而且不能再更改（除非引发了异常）。 如果操作结果执行已被另一个筛选器设置短路，则 ResultExecutedContext.Canceled 设置为 true。
        2. 如果操作结果或后续结果筛选器引发了异常，则 ResultExecutedContext.Exception 设置为非 NULL 值。 将Exception 设置为 NULL 可有效地“处理”异常，并防止 MVC 在管道的后续阶段重新引发该异常。 在处理结果筛选器中的异常时，可能无法向响应写入任何数据。 如果操作结果在其执行过程中引发异常，并且标头已刷新到客户端，则没有任何可靠的机制可用于发送失败代码。
        3. 对于 IAsyncResultFilter ，通过调用 ResultExecutionDelegate 上的 await next 可执行所有后续结果筛选器和操作结果。 若要设置短路，可将 ResultExecutingContext.Cancel 设置为 true，并且不调用ResultExecutionDelegate 。

1. IResultFilter和IAsyncResultFilter
   * + - 1. IAlwaysRunResultFilter 和 IAsyncAlwaysRunResultFilter 接口声明了一个针对操作结果运行的 IResultFilter实现。 除非应用 IExceptionFilter 或 IAuthorizationFilter 并使响应短路，否则会将筛选器应用于操作结果。
         2. 换句话说，这些“始终运行”的筛选器始终运行，除非异常或授权筛选器使它们短路。 除 IExceptionFilter 和IAuthorizationFilter 之外的筛选器不会使它们短路。
         3. 代码

public class UnprocessableResultFilter : Attribute, IAlwaysRunResultFilter

{

public void OnResultExecuting(ResultExecutingContext context)

{

if (context.Result is StatusCodeResult statusCodeResult &&

statusCodeResult.StatusCode == 415)

{

context.Result = new ObjectResult("Can't process this!")

{

StatusCode = 422,

};

}

}

public void OnResultExecuted(ResultExecutedContext context)

{

}

}

###### 在筛选器管道中使用中间件

资源筛选器的工作方式与中间件类似，即涵盖管道中的所有后续执行。 但筛选器又不同于中间件，它们是MVC 的一部分，这意味着它们有权访问 MVC 上下文和构造。

1. 可以在筛选器管道中使用中间件。 如果有一个中间件组件，该组件需要访问 MVC路由数据，或者只能针对特定控制器或操作运行，则可能需要这样做。
2. 若要将中间件用作筛选器，可创建一个具有 Configure 方法的类型，该方法可指定要注入到筛选器管道的中间件。
3. 代码：

public class LocalizationPipeline

{

public void Configure(IApplicationBuilder applicationBuilder)

{

var supportedCultures = new[]

{

new CultureInfo("en-US"),

new CultureInfo("fr")

};

var options = new RequestLocalizationOptions

{

DefaultRequestCulture = new RequestCulture(culture: "en-US", uiCulture: "en-US"),

SupportedCultures = supportedCultures,

SupportedUICultures = supportedCultures

};

options.RequestCultureProviders = new[]

{ new RouteDataRequestCultureProvider() { Options = options } };

applicationBuilder.UseRequestLocalization(options);

}

}

1. 然后，可以使用 MiddlewareFilterAttribute 为所选控制器或操作或者在全局范围内运行中间件：

[Route("{culture}/[controller]/[action]")]

[MiddlewareFilter(typeof(LocalizationPipeline))]

public IActionResult CultureFromRouteData()

{

return Content($"CurrentCulture:{CultureInfo.CurrentCulture.Name},"

+ $"CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");

}

##### 应用程序部件

应用程序部件是应用程序资源的一种抽象，可通过它发现控制器、视图组件或标记帮助程序等 MVC 功能。

AssemblyPart 就是一种应用程序部件，用于封装程序集引用以及公开类型和编译引用。

功能提供程序使用应用程序部件填充 ASP.NET Core MVC 应用的功能。 应用程序部件的主要用例是允许将应用配置为从程序集中发现（或避免加载）MVC 功能。

###### 简介：

1. MVC 应用从应用程序部件中加载其功能。 具体而言，AssemblyPart 类表示受程序集支持的应用程序部件。 可以使用这些类发现和加载 MVC 功能，比如控制器、视图组件、标记帮助程序和 Razor 编译源。 ApplicationPartManager负责跟踪可用于 MVC 应用的应用程序部件和功能提供程序。 配置 MVC 时，可以与 Startup 中的ApplicationPartManager 交互：

// create an assembly part from a class's assembly

var assembly = typeof(Startup).GetTypeInfo().Assembly;

services.AddMvc()

.AddApplicationPart(assembly);

// OR

var assembly = typeof(Startup).GetTypeInfo().Assembly;

var part = new AssemblyPart(assembly);

services.AddMvc()

.ConfigureApplicationPartManager(apm => apm.ApplicationParts.Add(part));

1. 默认情况下，MVC 将搜索依赖项树并查找控制器（甚至在其他程序集中）。 若要加载任意程序集（例如，从在编译时未引用的插件），可以使用应用程序部件。

可以使用应用程序部件避免查找特定程序集或位置中的控制器。 可以通过修改 ApplicationPartManager 的ApplicationParts 集合，控制应用可用的部件（或程序集）。 ApplicationParts 集合中条目的顺序并不重要。 重要的是在使用 ApplicationPartManager 配置容器中的服务之前，对该类进行完全配置。 例如，应在调用AddControllersAsServices 之前完全配置 ApplicationPartManager 。 如果未进行完全配置，就意味着，在方法调用之后添加的应用程序部件中的控制器不受影响（不注册为服务），这可能导致不正确的应用程序行为。

1. 如果某个程序集包含你不想使用的控制器，则将该程序集从 ApplicationPartManager 中删除：

services.AddMvc().ConfigureApplicationPartManager(apm =>

{

var dependentLibrary = apm.ApplicationParts

.FirstOrDefault(part => part.Name == "DependentLibrary");

if (dependentLibrary != null)

{

apm.ApplicationParts.Remove(dependentLibrary);

}

})

1. 除了项目的程序集及其从属程序集， ApplicationPartManager 还默认包含 Microsoft.AspNetCore.Mvc.TagHelpers 和Microsoft.AspNetCore.Mvc.Razor 的部件。

##### 应用程序功能提供程序

###### 应用程序功能提供程序用于检查应用程序部件，并为这些部件提供功能。

1. 以下 MVC 功能有内置功能提供程序：
   1. 控制器
   2. 元数据引用
   3. 标记帮助程序
   4. 视图组件
2. 功能提供程序从 IApplicationFeatureProvider<T> 继承，其中 T 是功能的类型。 你可以为上面列出的任意 MVC 功能类型实现自己的功能提供程序。
3. ApplicationPartManager.FeatureProviders 集合中功能提供程序的顺序可能很重要，因为靠后的提供程序可以对前面的提供程序所执行的操作作出反应。
4. 默认情况下，ASP.NET Core MVC 会忽略泛型控制器（例如， SomeController<T> ）。
5. 提供程序在默认提供程序后面运行并为指定的类型列表（在 EntityTypes.Types 中定义）添加泛型控制器实例：

public class GenericControllerFeatureProvider : IApplicationFeatureProvider<ControllerFeature>

{

public void PopulateFeature(IEnumerable<ApplicationPart> parts, ControllerFeature feature)

{

// This is designed to run after the default ControllerTypeProvider,

// so the list of 'real' controllers has already been populated.

foreach (var entityType in EntityTypes.Types)

{

var typeName = entityType.Name + "Controller";

if (!feature.Controllers.Any(t => t.Name == typeName))

{

// There's no 'real' controller for this entity, so add the generic version.

var controllerType = typeof(GenericController<>)

.MakeGenericType(entityType.AsType()).GetTypeInfo();

feature.Controllers.Add(controllerType);

}

}

}

}

实体类型：  
public static class EntityTypes

{

public static IReadOnlyList<TypeInfo> Types => new List<TypeInfo>()

{

typeof(Sprocket).GetTypeInfo(),

typeof(Widget).GetTypeInfo(),

};

public class Sprocket { }

public class Widget { }

}

将该功能提供程序添加到 Startup 中：

services.AddMvc()

.ConfigureApplicationPartManager(apm =>

apm.FeatureProviders.Add(new GenericControllerFeatureProvider()));

##### 自定义模型绑定

###### 默认模型绑定器限制

1. 默认模型绑定器支持大多数常见的 .NET Core 数据类型，能够满足大部分开发人员的需求。
2. 他们希望将基于文本的输入从请求直接绑定到模型类型。 绑定输入之前，可能需要对其进行转换。 例如，当拥有某个可以用来查找模型数据的键时。 基于该键，用户可以使用自定义模型绑定器来获取数据。

###### 模型绑定查看

1. 模型绑定为其操作对象的类型使用特定定义。
2. 简单类型转换自输入中的单个字符串。 复杂类型转换自多个输入值。框架基于是否存在 TypeConverter 来确定差异。 如果简单 string -> SomeType 映射不需要外部资源，建议创建类型转换器。
3. 创建自己的自定义模型绑定器之前，有必要查看现有模型绑定器的实现方式。 考虑使用 ByteArrayModelBinder，它可将 base64 编码的字符串转换为字节数组。 字节数组通常存储为文件或数据库 BLOB 字段。
4. 使用ByteArrayModelBinder.Base64编码的字符串可以表示二进制数据。可以采用 base64 编码的字符串，并使用 ByteArrayModelBinder 将其转换为字节数组。 实现IModelBinderProvider 的 ByteArrayModelBinderProvider 将 byte[] 参数映射到 ByteArrayModelBinder
5. 创建自己的自定义模型绑定器时，可实现自己的 IModelBinderProvider 类型，或使用 ModelBinderAttribute。
6. 若要使用自定义模型绑定器提供程序，请将其添加到 ConfigureServices 中：

options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());

###### 建议和最佳做法

1. 不应尝试设置状态代码或返回结果（例如 404 Not Found）。 如果模型绑定失败，那么该操作方法本身的操作筛选器或逻辑会处理失败。
2. 对于消除操作方法中的重复代码和跨领域问题最为有用。
3. 通常不应用其将字符串转换为自定义类型，而应选择用 TypeConverter 来完成此操作。

##### Note

1. C# Razor 关键字必须用 @（@C# Razor Keyword）进行双转移（例如，@（@case））。第一个@对 Razor 分析器转义。第二个@对 C# 分析器转义

### TagHelper

标记帮助程序使服务器端代码可以在Razor 文件中参与创建和呈现HTML元素。例如内置的 ImageTagHelper 可以将版本号追加到图像名称。每当图像发生变化时，服务器都会为图像生成一个新的唯一版本，因此客户端总能获得当前图像。标记帮助程序使用C# 创建，基于元素名称、属性名称或父标记以HTML元素为目标。

##### 标记帮助程序的功能

###### 友好的体验

1. HTML 友好开发体验在多数情况下，使用标记帮助程序的 Razor 标记看起来像是标准HTML。熟悉 HTML/CSS/JavaScript 的前端设计师，无需学习 C# Razor 语法即可编辑Razor。

###### IntelliSense 环境

1. 用于创建 HTML 和 Razor 标记的丰富 IntelliSense 环境这与 HTML 帮助程序形成鲜明对比，HTML 帮助程序是 Razor 视图中标记的曾用服务器端创建方法。

###### 使用仅在服务器上可用的信息，可提高生产力，并能生成更稳定、可靠和可维护的代码

1. 大多数内置标记帮助程序以标准 HTML 元素为目标，为该元素提供服务器端属性。

##### 管理标记帮助程序作用域

###### TagHelper作用域由 @addTagHelper、@removeTagHelper和“！”选择退出字符联合控制

###### 使用@addTagHelper 添加标记帮助程序

1. @addTagHelper 指令让视图可以使用标记帮助程序
   1. 如果创建名为AuthoringTagHelpers 的新ASP.NET Core Web 应用，将向项目添加以下Views/\_ViewImports.cshtml 文件：
   2. @addTagHelper \*，Microsoft.AspNetCore.Mvc.TagHelpers
   3. @addTagHelper \*, AuthoringTagHelpers
   4. @addTagHelper 指令让视图可以使用标记帮助程序。
   5. 在此示例中，视图文件是Pages/\_ViewImports.cshtml，“Pages”文件夹及其子目录中的所有视图文件都会默认继承它，使得标记帮助程序可用。
   6. 上面的代码使用通配符语法（“\*”），指定程序集(Microsoft.AspNetCore.Mvc.TagHelpers) 中的所有标记帮助程序对于 Views 目录或子目录中的所有视图文件均可用。
   7. @addTagHelper第一个参数指定要加载的标记帮助程序（\*代表加载所有的程序），第二个参数指定包含标记帮助程序的程序集
2. 公开标记帮助程序
   1. @usingAuthorizingTagHelpers

@addTagHelper\*,Microsoft.AspNetCore.Mvc.TagHelpers

@addTagHelper \*,AuthorizingTagHelpers

* 1. 如果项目包含具有默认命名空间的TagHelper，则可以使用完全项目名添加到

@usingAuthorizingTagHelpers

@addTagHelper\*,Microsoft.AspNetCore.Mvc.TagHelpers

@addTagHelper AuthorizingTagHelpers.TagHelpers.EmailTagHelper,AuthorizingTagHelpers

* 1. 使用通配符添加

@addTagHelperAuthorizingTagHelpers.TagHelpers.E\*,AuthorizingTagHelpers

@addTagHelperAuthorizingTagHelpers.TagHelpers.Email\*,AuthorizingTagHelpers

* 1. 将 @addTagHelper 指令添加到 Views/\_ViewImports.cshtml 文件，将使标记帮助程序对于 Views 目录及子目录中的所有视图文件可用。如果想选择仅对特定视图公开标记帮助程序，可在这些视图文件中使用 @addTagHelper 指令。

###### 删除标记帮助程序

1. 使用@removeTagHelper删除标记帮助程序
   1. @removeTagHelper 与 @addTagHelper 具有相同的两个参数，它会删除之前添加的标记帮助程序。例如，应用于特定视图的 @removeTagHelper 会删除该视图中的指定标记帮助程序。在 Views/Folder/\_ViewImports.cshtml 文件中使用 @removeTagHelper ，将从 Folder中的所有视图删除指定的标记帮助程序。
2. 使用 \_ViewImports.cshtml 文件控制标记帮助程序作用域
   1. 可将 \_ViewImports.cshtml 添加到任何视图文件夹，视图引擎将同时应用该文件和Views/\_ViewImports.cshtml 文件中的指令。如果为 Home 视图添加空的Views/Home/\_ViewImports.cshtml 文件，则不会发生任何更改，因为\_ViewImports.cshtml 文件是附加的。添加到 Views/Home/\_ViewImports.cshtml 文件（不在默认 Views/\_ViewImports.cshtml 文件中）的任何 @addTagHelper 指令，都只会将这些标记帮助程序公开给 Home 文件夹中的视图。
3. 选择退出元素
   1. 使用标记帮助程序选择退出字符（“!”），可在元素级别禁用标记帮助程序
   2. 例如：选择退出字符在<span>中禁用Email验证

<!span asp-validation-for="Email" class="text-danger"></!span>

###### 使用 @tagHelperPrefix 阐明标记帮助程序用途

1. 使用@tagHelperPrefix指定标记前缀字符串
   1. 可以将 @tagHelperPrefix th:标记添加到 Views/\_ViewImports.cshtml 文件
   2. <th:label asp-for:”Password” class = “col-md-2”></th:label>
2. 适用于 @addTagHelper 的层次结构规则也适用于 @tagHelperPrefix

###### 自结束标记帮助程序

1. 许多标记帮助程序都不能用作自结束标记。某些标记帮助程序被设计为自结束标记。使

用未被设计为自结束的标记帮助程序会抑制呈现的输出。自结束标记帮助程序会在呈现的输出中生成自结束标记。

###### 标记帮助程序的Intellisense支持

1. Nuget添加 Microsoft.AspNetCore.Razor.Tools

##### TagHelper 和 HtmlHelper的比较

标记帮助程序附加到Razor视图中的HTML元素，HTML帮助程序是作为与Razor 视图中HTML交织的方法被调用的。

###### 比较呈现<label class="caption" for="Title">LabelText</label>;

1. HtmlHelper 写法
   1. @Html.Label("Title", "LabelText",new { @class = "caption" });



1. TagHelper写法
   1. <**labelasp-for**="Movie[0].Title"class="caption"></**label**>



###### TagHelper 优点

1. 与 HTML 帮助程序方法相比，此标记更清晰，更容易阅读、编辑和维护。 C# 代码会被减少至服务器需要知道的最小值。 Visual Studio 编辑器以独特的字体显示标记帮助程序的目标标记。

##### TagHelper 和Web服务器控件的比较

###### TagHelper 特点

1. 标记帮助程序不拥有与其相关的元素：它们只是参与元素和内容的呈现。ASP.NET Web 服务器控件在页面上进行声明和调用。
2. 通常不能撰写 Web 服务器控件时，多个标记帮助程序可作用于同一元素
3. 标记帮助程序可以修改其作用域内 HTML 元素的标记和内容，但不会直接修改页面上的其他内容。
4. 使用标记帮助程序时，本身就用 C# 语言工作，因此无需进行类型转换。
5. Web 服务器控件使用 System.ComponentModel 实现组件和控件的运行时和设计时行为。 System.ComponentModel 包括用于属性和类型转换器的实现、数据源绑定和组件授权的基类和接口。与通常派生自 TagHelper 的标记帮助程序相比，TagHelper 基类仅公开两个方法，即 Process 和ProcessAsync 。

###### Web服务器控件特点

1. Web 服务器控件具有可观的生命周期，因而难以进行开发和调试。
2. 通过 Web 服务器控件，可使用客户端控件向客户端文档对象模型 (DOM) 元素添加功能。标记帮助程序没有 DOM。
3. Web 服务器控件包括自动浏览器检测。标记帮助程序不了解浏览器。
4. Web 服务器控件的作用域较广，并且可以执行影响页面其他部分的操作，从而可能造成意想不到的副作用。
5. Web 服务器控件使用类型转换器将字符串转换为对象。

##### 定位点标记帮助程序（<a></a>）

定位点标记帮助程序可通过添加新属性来增强标准的 HTML 定位点 ( <a ... ></a> ) 标记。按照约定，属

性名称将使用前缀 asp- 。 asp- 属性的值决定呈现的定位点元素的 href 属性值。

###### asp-controller

1. asp-controller属性可分配用于生成URL的控制器
   1. <a asp-controller=”Speaker” asp-action=”Index”>All Speakers</a>
   2. <a href=”/Speaker”>All Speaker</a>
2. 如果指定了asp-controller 属性而未指定 asp-action ,则默认的asp-action 值为与当前正在执行的的视图关联的控制器操作。

###### asp-action

1. <a asp-controller="Speaker" asp-action="Evaluations">Speaker Evaluations</a>

###### asp-all-route-data

1. <a asp-route="speakerevalscurrent" asp-all-route-data="parms">Speaker Evaluations</a>

###### asp-area

1. <a asp-area="Blogs" asp-controller="Home" asp-action="AboutBlog">About Blog</a>
2. <a asp-area="Sessions" asp-page="/Index">View Sessions</a>

###### asp-fragment

1. <a asp-controller="Speaker" asp-action="Evaluations" asp-fragment="SpeakerEvaluations">Speaker Evaluations</a>

###### asp-host

1. <a asp-protocol="https" asp-host="microsoft.com" asp-controller="Home" asp-action="About">About</a>

###### asp-page RP（RP Supported in Razor Pages only）

1. <a asp-page="/Attendee">All Attendees</a>

###### asp-page-handler RP（RP Supported in Razor Pages only）

1. <a asp-page="/Attendee" asp-page-handler="Profile" asp-route-attendeeid="12">Attendee Profile</a>

###### asp-protocol

1. <a asp-protocol="https" asp-controller="Home" asp-action="About">About</a>

###### asp-route

1. <a asp-route="speakerevals">Speaker Evaluations</a>

###### asp-route-*{value}*

1. <a asp-page="/Attendee" asp-route-attendeeid="10">View Attendee</a>

##### 缓存标记帮助程序（<cache></cache>）

缓存标记帮助程序通过将其内容缓存到内部 ASP.NET Core 缓存提供程序中，极大地提高了 ASP.NET Core 应用的性能。

###### 缓存帮助程序属性

1. enabled
   1. Boolean 默认true
   2. 确定是否缓存了缓存标记帮助程序所包含的内容。默认值为 true 。如果设置为 false ，则不会缓存呈现的输出。
2. expires-on
   1. DateTimeOffset如：new DateTIme(2019,6,12,15,30,0)
   2. 为缓存项设置的绝对到期时间
3. expires-after
   1. TimeSpan如 TimeSpan.FromMinutes(10)
   2. 为缓存项设置的相对到期时间
4. expires-sliding
   1. TimeSpan 如 TimeSpan.FromMinutes(10)
   2. 设置某个缓存项的值未被访问时，该缓存项应被逐出的时间
5. vary-by-header
   1. String 如 User-Agent、User-Agent,content-encoding
   2. 在标头值发生变更时触发缓存刷新。
6. vary-by-query
   1. String 如Make、Make,Model
   2. 接受查询字符串（Query）中逗号分割的Keys列表，他们在任何列出的键值发生变更时触发缓存刷新。
7. vary-by-route
   1. String 如 Make、Make,Model
   2. 接受路由参数名称的逗号分割列表，用于在路由数据参数值发生变更时触发缓存刷新。
8. vary-by-cookie
   1. String 如a.cookiename、a.cookie.name,b.cookie.name
   2. 在Cookie值发生变更时触发缓存刷新。
9. vary-by-user
   1. Boolean如true、false默认 true
   2. 指定当已登录用户（或上下文主体）发生变更时是否重置。[可以通过@User.Identity.Name在Razor](mailto:可以通过@User.Identity.Name在Razor)视图中查看
10. vary-by
    1. 自定义缓存数据。当属性的字符串发生更改时重置
    2. <cache vary-by=”@Model”></cache>
11. priority
    1. CacheItemPriority 如 High,Low,NeverRemove,Normal 默认Normal
    2. 为内置缓存提供程序提供缓存逐出指导。在内存压力下，服务器按照Low,Normal,High,NeverRemove 依次逐出缓存项
    3. priority 属性并不能保证特定级别的缓存保留。 CacheItemPriority 仅供参考。将此属性设置为 NeverRemove并不能保证缓存项将始终保留。

##### 分布式缓存帮助程序（< distributed-cache ></ distributed-cache>）

###### 分布式缓存帮助程序属性

1. 分布式缓存标记帮助程序继承自与缓存标记帮助程序相同的类
2. name
   1. name 是必须的 string 例如 my-distributed-cache-unique-key-101

###### 分布式缓存标记帮助程序 IDistributedCache实现

1. Asp.Net Core 中内置了 IDistributedCache的两个实现。一个是基于SqlServer，另一个是基于Redis中

##### 环境标记帮助程序（< environment></ environment >）

###### Include 基于已包括的宿主环境名称，控制已包含内容的呈现

1. include 属性表现出与names属性相似的行为。include属性值中列出的环境必须与应用程序的托管环境（IHostingEnvironment.EnvironmentName）匹配才能呈现<environment>标记的内容
   1. <environment include=”Staging,Production”>

<strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>

</environment>

###### exclude 与include属性相反

1. 当托管环境与exclude属性值中列出的环境不匹配时，将呈现标记的内容
   1. <environment include=”Development”>

<strong>HostingEnvironment.EnvironmentName is not Development</strong>

</environment>

##### 表单标记帮助程序

1. 为MVC 控制器操作或命名路由生成HTML <FORM> action 属性值

2. 生成隐藏的请求验证令牌，防止XSS （在HTTP Post 操作方法中与[ValidateAntiForgeryToken]）属性配合时使用

3. 提供 asp-route-<Parameter Name> 属性，其中 <Parameter Name> 添加到路由值。Html.BeginForm 和 Html.EndForm 的routeValues 参数提供类似的功能

4. 具有HTML 帮助程序替代项 Html.BeginForm 和 Html.BeginRouteForm

###### 使用命名路由

1. asp-route 标记帮助程序属性还可为HTML action 属性生成标记。具有名为register 的路由的应用可将以下标记用于注册页
   1. <form asp-route=”register” method=”post”></form>
2. Views/Account 文件夹中的许多视图（在新建使用个人用户账户身份验证的Web应用时生成）包含asp-route-returnurl属性
   1. <form asp-controller=”Account” asp-action=”Login”

asp-route-returnurl=”@ViewData[“ReturnUrl”]” method=”post” class=”form-horizontal”

role=”form”>

</form>

##### 窗体标记帮助程序

窗体标记帮助程序在生成的<button …>或<input type=”image” …>标记上生成formaction属性。formaction属性控制窗体在何处提交数据。它绑定到image 类型的 <input>元素以及<button>元素。窗体标记帮助程序允许多个AnchorTagHelper asp-属性 来控制为响应元素生成的formaction链接

###### AnchorTagHelper属性

1. asp-controller 控制器的名称
2. asp-action 操作方法的名称
3. asp-area 区域名称
4. asp-page Razor Page 的名称
5. asp-page-handler Razor Page 处理程序的名称
6. asp-route 路由的名称
7. asp-route-{value} 单个URL 路由值。例如 asp-route-id=”1234”
8. asp-all-route-data 所有路由值
9. asp-fragment URL 片段

###### 提交到控制器示例

1. 选中输入或按钮时，下面的标记将窗体提交到HomeController的Index 操作：

<form method=”post”>

<button asp-controller=”Home” asp-action=”Index”>Click Me</button>

<input type=”image” src=”..” alt=”Or Click Me” asp-controller=”Home” asp-action=”Index”>

</form>

###### 提交到路由

1. public class HomeController:Controller

{

[Route(“/Home/Test”,Name=”Custom”)]

public string Test(){ return “”;}

}

1. 以下标记将窗体提交到/Home/Test终结点

<form method=”post”>

<button asp-route=”Custom”>Click Me</button>

<input type=”image” src=”” Alt =”Or Click Me” asp-route=”Custom” />

</form>

1. <form method=”post”>

<button formaction=”/Home/Test”>Click Me</button>

<input type=”image” src=”” Alt =”Or Click Me” formaction=”/Home/Test” />

</form>

###### 输入标记帮助程序

输入标记帮助程序将HTML <input> 元素绑定到 Razor 视图中的表达式

语法：<input asp-for=”Expression Name”>

1. 为 asp-for 属性中指定的表达式名称生成id和name的HTML属性。asp-for =”Property1.Property2”与m=>m.Property1.Property2相等。表达式的名称用于 asp-for 属性值。
2. 根据模型类型和应用于模型属性的数据注释特性设置HTML type 特性值
3. 如果已经指定，不会覆盖 HTML type 属性值
4. 通用应用于模型属性的数据注释特性生成HTML5验证特性
5. 具有与Html.TextBoxFor 和 Html.EditorFor 重叠的HTML 帮助程序功能。
6. 提供类型强化。如果属性的名称更改，但未更新标记帮助程序，则会收到属性报错
7. .NET 类型 和 HTML type属性对应表
   1. Bool type=”checkbox”
   2. String type=”text”
   3. DateTime type=”datetime-local”
   4. Byte type=”number”
   5. Int type=” number”
   6. Single、Double type=” number”
8. .NET 特性和HTML type属性对应表
   1. [EmailAddress] type=”email”
   2. [Url] type=”url”
   3. [HiddentInput] type=”hidden”
   4. [Phone] type=”tel”
   5. [DataType(DataType.Password)] type=”password”
   6. [DataType(DataType.Date)] type=”date”
   7. [DataType(DataType.Time)] type=”time”

###### 输入标记帮助程序的HTML帮助程序替代项

1. Html.TextBox 、Html.TextBoxFor、Html.Editor 和 Html.EditorFor 与输入标记帮助程序的功能存在重叠。
2. 输入标记帮助程序会自动设置type属性;而Html.TextBox 和Html.TextBoxFor不会。
3. Html.Editor和Html.EditorFor处理集合、复杂对象和模版;而输入标记帮助程序不会。
4. Html.EditorFor 和 Html.TextBoxFor是强类型(使用Lambda表达式);而Html.TextBox和Html.Editor 不是（使用表达式名称）

###### HtmlAttributes

1. @Html.Editor()[和@Html.EditorFor()在执行其默认模版的时使用名为](mailto:和@Html.EditorFor()在执行其默认模版的时使用名为) HtmlAttributes的特殊ViewDataDictionary条目。此行为可选择使用 addtionalViewData参数增强。
2. 键”htmlattributes”区分大小写。键“htmlattributes”的处理方式与传递到输入帮助程序的htmlattributes对象（例如@Html.TextBox()）的处理方式类似
3. @Html.EditorFor(model=>model.YourProperty,
4. new {htmlAttributes = new {@class=”myClass”,style=”Wdith:100px”}})

###### 表达式名称

1. asp-for 属性值时ModelExpression，并且时lambda表达式的右侧。因此，asp-for=”Property1”,在生成的代码中变成 m=>m.Property1,这也是无需使用Model 前缀的原因。“@”字符可用作内联表达式的开头,并移到m.前面：
   1. @{ var joe=”Joe”;}

<input asp-for=”@joe”>

* 1. 生成以下HTML：

<input type=”text” id=”joe” name=”joe” value=”joe”>

1. 使用集合时，asp-for=”CollectionProperty[23].Member” 在i具有23时生成与asp-for=”CollectionProperty[i].Member” 相同的名称
2. 在ASP.NET Core MVC 计算ModelExpression的值时，它会检查多个源，包括ModelState。以<input type=”text” asp-for=”@Name”> 为例。计算出的value属性时第一个非null 值，属于：
   1. 带有”Name” 键的ModelState 条目
   2. Model.Name 表达式的结果

###### 导航子属性

1. 还可以使用视图模型的属性路径导航到子属性。设想一个包含子Address属性的更复杂的模型类
   1. @obj.ObjectPropertyName.Property

###### 表达式名称和集合

1. 应尽量在 asp-for 或 Html.DisplayFor 等效上下文中使用值 foreach。一般情况下，for 优于 foreach（如果情况允许使用的话），因为它不需要分配枚举器，但在LINQ表达式中评估索引器的成本高昂，应最大限度地减少使用它。

##### 文本标记帮助程序

Textarea TagHelper 标记帮助程序类似于输入标记帮助程序

1. 通过模型为 <textarea>元素生成id和name 属性以及数据验证属性

2. 提供强类型化

3. HTML 帮助程序替代项：Html.TextAreaFor

##### 标签标记帮助程序

1. 在 <label>元素中为表达式名称生成标签描述和for属性

2. HTML 帮助程序替代项：Html.LabelFor

3. Label Tag Helper 通过纯HTML 标签元素提供如下优势：

可自动从Display属性中获取描述性标签值。预期的显示名称可能会随时间变化,Display属性和标签标记帮助程序的组合会在其被使用的所有位置应用Display

源代码中的标记更少

模型属性的强类型化

##### 验证标记帮助程序

有两个验证标记帮助程序。Validation Message Tag Helper （为模型中的单个属性显示验证消息）和 Validation Summary Tag Helper（显示验证错误的摘要）。 Input Tag Helper 根据模型类的注释属性将HTML5客户端验证属性添加到元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

###### 验证消息标记帮助程序

1. 将HTML5 data-valmsg-for=”Property” 属性添加到span 元素中，该元素会附加到指定模型属性的输入字段中的验证错误消息。JQuery会在发生客户端验证错误时在<span>元素中显示错误消息。
2. 还会在服务器上执行验证。客户端可能已禁用JavaScript，一些验证仅可在服务器端执行
3. HTML 帮助程序替代项：
   1. Html.ValidationMessageFor
4. Validation Message Tag Helper 与HTML span元素中的 asp-validation-for 属性配合使用

<span asp-validation-for=”Email”></span>

1. 对于同一属性，通常在Input 标记帮助程序后使用 Validation Message Tag Helper。这样做可在导致错误的输入附近显示所有验证错误信息

###### 验证摘要标记帮助程序

1. 针对具有 asp-validation-summary 属性的<div>元素
2. HTML 帮助程序替代项：@Html.ValidationSummary
3. Validation Summary Tag Helper 用于显示验证消息的摘要。asp-validation-summary属性值：
   1. ValidationSummary.All 属性和模型级别
   2. ValidationSummary.ModelOnly 模型
   3. ValidationSummary.None None

##### 选择标记帮助程序

1. 为模型属性生成 select 元素和关联的option项

2. 具有HTML 帮助程序替代项 Html.DropDownListFor 和 Html.ListBoxFor

3. Select Tag Helper asp-for 为select 元素指定模型属性名称,asp-items 指定 option项。如：

<select asp-for=”Country” asp-items = “Model.Countries”></select>

4. asp-for 属性值是特殊情况，它不要求提供 Model 前缀，但其他标记帮助程序属性需要该前缀(例如 asp-items)

###### 枚举绑定

1. 通常可以方便地将<select>与enum属性配合使用并通过enum值生成 SelectListItem 元素

public class CountryEnumViewModel

{

public CountryEnum EnumCountry{get;set;}

}

<select asp-for=”EnumCountry” asp-items=”Html.GetEnumSelectList<CountryEnum>()”></select>

1. 可使用Display 属性修饰枚举器列表，以获取更丰富的UI

public enum CountryEnum

{

[Display(Name=”United Mexican States”)]

Mexio

}

###### 选项组

1. 如果视图模型包含一个或多个SelectListGroup对象，则会生成 HTML <optgroup>

public class CountryViewGroup

{

public CountryViewGroup()

{

var NorthAmericaGroup = new SelectListGroup{Name=”North American”};

var EuropeGroup = new SelectListGroup{Name=”Europe”};

Countries = new List<SelectListItem>

{

new SelectListItem{ Value=”Mex”,Text=”Mexico”,Group=NorthAmericaGroup},

new SelectListItem{ Value=”CAN”,Text=”Canada”,Group=NorthAmericaGroup},

new SelectListItem{ Value=”US”,Text=”USA”,Group=NorthAmericaGroup},

new SelectListItem{ Value=”ES”,Text=”Spain”,Group= EuropeGroup },

new SelectListItem{ Value=”DE”,Text=”Germany”,Group= EuropeGroup },

new SelectListItem{ Value=”FR”,Text=”France”,Group= EuropeGroup },

}

}

}

1. 多重选择
   1. 如果asp-for 属性中指定的属性为 IEnumabe，选择标记帮助程序会自动生成 multiple=”multipe”
2. 无选定内容
   1. 如果发现自己在多个页面中使用“未指定”选项，可创建模版用于消除重复的HTML

### BLAZOR

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### EF CORE

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### Authentication && Authorization

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### DAPPER

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### Debug

##### VSCode单元测试

###### 插件

1. C#语言扩展

<https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp>

1. C#XML注释

<https://marketplace.visualstudio.com/items?itemName=k--kato.docomment>

1. C#辅助

<https://marketplace.visualstudio.com/items?itemName=jchannon.csharpextensions>

EditorConfig,Guildes,One Dark Theme,Project Manager ,Setting Sync

###### 创建项目

1. 创建解决方案

dotnet new sln -o vscode\_debug\_prj

1. 进入解决方案内部

cd /d vcode\_debug\_prj

1. 添加项目

dotnet new classlib -o vscode\_debug\_prj.common

dotnet new console -o vscode\_debug\_prj.consoleapp

dotnet new xunit -o vscode\_debug\_prj.unittest

1. 将项目添加到解决方案中

dotnet sln add vscode\_debug\_prj.common/vscode\_debug\_prj.common.csproj

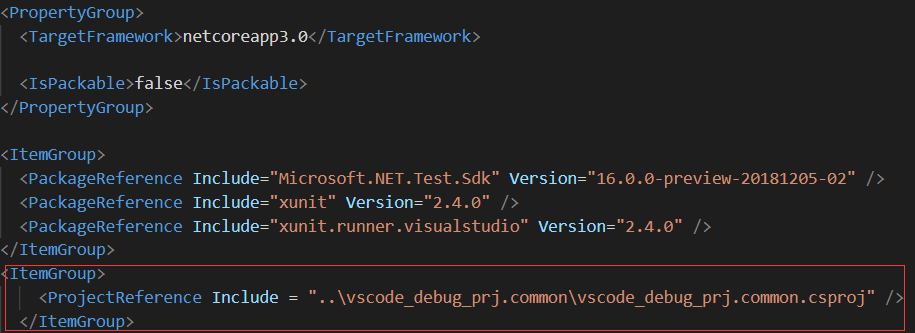
dotnet sln add vscode\_debug\_prj.consoleapp /vscode\_debug\_prj.consoleapp.csproj

dotnet sln add vscode\_debug\_prj.unittest /vscode\_debug\_prj.unittest.csproj

###### 使用VSCode

* 1. 使用VSCode打开vscode\_debug\_prj.sln所在到目录
  2. 添加项目间的依赖关系

编辑csproj文件



* 1. 还原项目依赖

dotnet restore

* 1. 编译项目

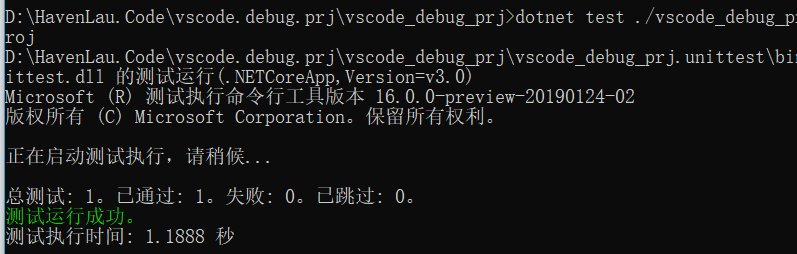
dotnet build

* 1. 编写方法，并且调用



* 1. 使用命令行运行单元测试

dotnet test ./vscode\_debug\_prj.unittest/vscode\_debug\_prj.unittest.csproj



* 1. 使用VSCode运行单元测试

添加task.json



第一个任务build\_console 运行时会编译VSCodeTutorial.ConsoleApp项目及其依赖的项目

第二个任务UnitTest则是单元测试项目，运行dotnet test命令

特殊的设置"isTestCommand": true 标识为测试项目后可以通过快捷方式运行该命令

添加launch.json

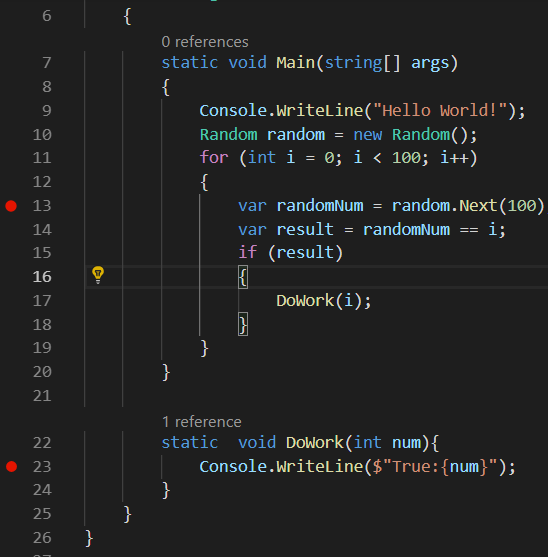
* 1. 添加 task.json

##### VSCode代码调试

###### 调试技巧

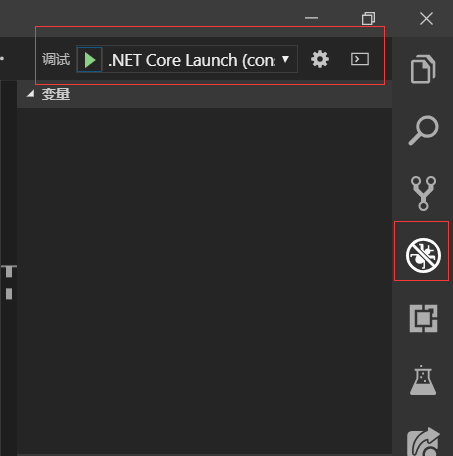
* 1. 设置断点

在代码编辑区域，点击左侧代码行行号的左边，即可在该行设置断点



* 1. 启动调试

使用菜单：调试->启动调试，或者使用快捷键F5启动调试



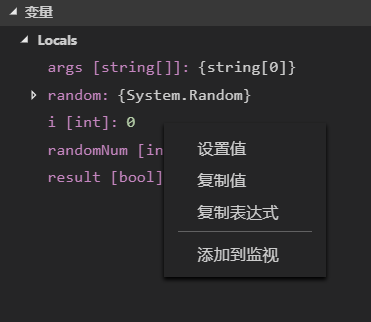
###### 界面介绍

1. 顶部Debug工具栏

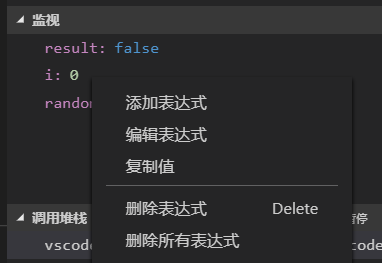


| **按钮&快捷键** | **说明** |
| --- | --- |
| 绿色向右箭头（F5） | 继续执行程序，如果遇到断点则会停留到该断点 |
| 蓝色向右箭头（F10） | 单步跳过：执行一条语句，但是遇到方法调用时不进入，直到方法执行完成后直接继续。 |
| 蓝色向下箭头（F11） | 单步调试：执行一条语句，遇到方法调用时会进入方法进行调试 |
| 蓝色向上箭头（Shift+F11） | 单步跳出：执行当前当前方法并到下一步骤，如果当前方法有断点则会到下一个断点 |
| 绿色环形箭头（Ctrl+Shift+F5） | 重新启动调试 |
| 红色方块（Shift+F5） | 停止调试 |

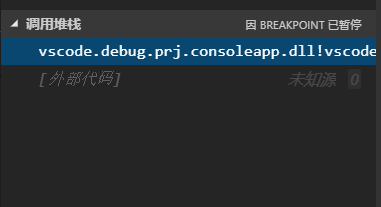
1. 变量区域



1. 监视

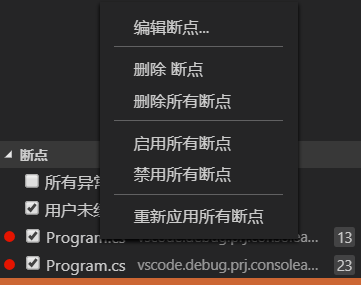


1. 调用堆栈



| **项** | **说明** |
| --- | --- |
| heelodotnet.dll | 调用的动态链接库 |
| hellodotnet | 命名空间 |
| Program.Main(string[] args) | 调用的类&方法 |
| Line | 当前调用方法的行号 |

1. 断点



###### 调试快捷点

| **快捷键** | **说明** |
| --- | --- |
| 快捷键：F5 | 继续执行程序，如果遇到断点则会停留到该断点 |
| 快捷键：F10 | 单步跳过：执行一条语句，但是遇到方法调用时不进入，直到方法执行完成后直接继续。 |
| 快捷键：F11 | 单步调试：执行一条语句，遇到方法调用时会进入方法进行调试 |
| 快捷键：Shift+F11 | 单步跳出：执行当前当前方法并到下一步骤，如果当前方法有断点则会到下一个断点 |
| 快捷键：Ctrl+Shift+F5 | 重新启动调试 |
| 快捷键：Shift+F5 | 停止调试 |
| 快捷键：F9 | 切换断点:跳到下一个断点 |
| 快捷键：Shift+F9 | 列断点：在当前光标的下一行增加一个断点 |

### 会话和应用状态

HTTP 是无状态的协议。不采取其他步骤的情况下，HTTP 请求是不保留用户值或应用状态的独立消息。

| **存储方法** | **存储机制** |
| --- | --- |
| Cookie | HTTPCookie（可能包括使用服务端应用代码存储的数据） |
| 会话状态 | HTTPCookie和服务端应用代码 |
| TempData | HTTP Cookie 或会话状态 |
| 查询字符串 | HTTP查询字符串 |
| 隐藏字段 | HTTP窗体字段 |
| HttpContext.Items | 服务器端应用代码 |
| 缓存 | 服务器端应用代码 |
| 依赖关系注入 | 服务器端应用代码 |

##### Cookie

###### Cookie 存储所有请求的数据。因为 Cookie 是随每个请求发送的，所以它们的大小应该保持在最低限度。

1. 理想情况下，仅标识符应存储在 Cookie 中，而数据则由应用存储。
2. 大多数浏览器 Cookie 大小限制为 4096 个字节。每个域仅有有限数量的 Cookie 可用。

###### 由于 Cookie 易被篡改，因此它们必须由服务器进行验证。

* + - 1. 客户端上的 Cookie 可能被用户删除或者过期。但是，Cookie 通常是客户端上最持久的数据暂留形式。

###### Cookie 通常用于个性化设置，其中的内容是为已知用户定制的。

* + - 1. 大多数情况下，仅标识用户，但不对其进行身份验证。
      2. Cookie 可以存储用户名、帐户名或唯一的用户 ID（例如 GUID）。然后，可以使用 Cookie 访问用户的个性化设置，例如首选的网站背景色。

##### 会话状态

会话状态是在用户浏览 Web 应用时用来存储用户数据的 ASP.NET Core 方案。

###### 会话状态机制

* + - 1. 会话状态使用应用维护的存储来配置会话状态保存客户端所有请求的数据。
      2. 会话数据由缓存支持并被视为临时数据 - 站点应在没有会话数据的情况下继续运行。
      3. 关键应用程序数据应存储在用户数据库中，并仅作为性能优化缓存在会话中。
      4. ASP.NET Core 通过向客户端提供包含会话 ID 的 Cookie 来维护会话状态，该会话 ID 与每个请求一起发送到应用。应用使用会话 ID 来获取会话数据。

###### 会话状态行为：

* + - 1. 由于会话 Cookie 是特定于浏览器的，因此不能跨浏览器共享会话。
      2. 浏览器会话结束时删除会话 Cookie。
      3. 如果收到过期的会话 Cookie，则创建使用相同会话 Cookie 的新会话。
      4. 不会保留空会话 - 会话中必须设置了至少一个值以保存所有请求的会话。会话未保留时，为每个新的请求生成新会话 ID。
      5. 应用在上次请求后保留会话的时间有限。应用设置会话超时，或者使用 20 分钟的默认值。会话状态适用于存储特定于特定会话的用户数据，但该数据无需永久的会话存储。
      6. 调用 ISession.Clear 实现或者会话过期时，会删除会话数据。
      7. 没有默认机制告知客户端浏览器已关闭或者客户端上的会话 Cookie 被删除或过期的应用代码。
      8. ASP.NET Core MVC 和 Razor Pages 模板包括对一般数据保护条例 (GDPR) 的支持。

###### 在内存中缓存会话状态数据（服务器方案）

1. 使用粘性会话将每个会话加入到单独服务器上的特定应用实例。默认情况下 Azune应用服务使用应用程序请求路由（ARR）强制实施粘性会话。
   * + - 1. 影响程序伸缩性，使web应用更新变得复杂
         2. 推荐使用Redis或 SqlServer 分布式缓存。他们不需要粘性会话。
2. 通过IDataProtector加密会话 Cookie。
3. 必须正确配置数据保护，以确定在每台计算机上读取会话 Cookie

###### 配置会话状态

1. 关系类库：
2. Microsoft.AspNetCore.App metapackage 中包含的 Microsoft.AspNetCore.Session 包提供中间件来管理会话状态。
3. 启用会话状态中间件
   1. 任一 IDistributedCache内存缓存。 IDistributedCache实现会话后备存储
   2. 对 ConfigureServices中 AddSession的调用
   3. 对 Configure中 UseSession 的调用



1. 中间件的顺序很重要。配置会话状态后，HttpContext.Session可用。调用UseSession以前无法访问HttpContext.Session
2. 在应用已经开始写入到响应流之后，不能创建有新会话Cookie的新会话。

###### 以异步方式加载会话状态

1. 只有在TryGetValue、Set或者 Remove方法显示调用 ISeesion.LoadAsync方法，Asp.Net Core 默认会话提供程序才会从基础的 IDistrubutedCache后备存储中以异步形式加载会话记录。
2. 如果未先调用 LoadAsync，则会同步加载基础会话记录，可能对性能产生大规模影响。

###### 会话选项SessionOptions

1. Cookie用于创建Cookie的设置
   1. 名称默认为 SessionDefaults.CookieName(.AspNetCore.Session)
   2. 路径默认为SessionDefaults.CookiePath(/)
   3. SameSite 默认为SameSiteMode.Lax(1)
   4. HttpOnly默认为 true
   5. IsEsstensial默认为 false
2. IdelTimeout超时时间
   1. 显示放弃内容前，内容可以空闲多长时间。每个会话访问都会重置超时。
   2. 此设置仅适用于会话内容，不适用于Cookie。默认为20分钟。
3. IOTimeout从存储加载会话或者提交回存储的超时时间
   1. 此设置仅适用于异步操作，可使用 InfiniteTimeSpan禁用超时，默认1分钟。

###### 设置或获会话状态

1. 使用HttpContext.Session获取会话状态。此属性是ISession的实现。
2. 相关包：
   1. Microsoft.AspNetCore.App
   2. Microsoft.AspNetCore.Http
   3. Microsoft.AspNetCore.Http.Extensions
3. 相关扩展方法
   1. Get(ISession,String)
   2. GetInt32(Session,String)
   3. GetString(ISession,string)
   4. SetInt32(ISession,String,Int32)
   5. SetString(ISession,String,String)

###### 注意

1. 请勿将敏感数据存储在会话状态中。用户可能不会关闭浏览器并清除会话 Cookie。某些浏览器会保留所有浏览器窗口中的有效会话 Cookie。会话可能不限于单个用户 - 下一个用户可能继续使用同一会话 Cookie 浏览应用。
2. 必须对所有的会话数据进行序列化以启用分布式缓存方案，即使是在使用内存中缓存的时候。提供最小的字符串和数字序列化程序

##### TempData

此属性存储未读取的数据。 Keep 和 Peek 方法可用于检查数据，而不执行删除。多个请求需要数据时，TempData 非常有助于进行重定向。使用 Cookie 或会话状态通过 TempData 提供程序实现 TempData。

###### TempData 提供程序

1. 基于 cookie 的 TempData 提供程序默认用于存储 cookie 中的 TempData。
   1. 使用由 Base64UrlTextEncoder 编码的 IDataProtector 对 Cookie 数据进行加密，然后进行分块。
   2. 未压缩 Cookie 数据，因为压缩加密的数据会导致安全问题，如 CRIME 和 BREACH 攻击。
   3. 提供程序CookieTempDataProvider

###### 选择 TempData 提供程序

1. 应用是否已使用会话状态？如果是，使用会话状态 TempData 提供程序对应用没有额外的成本（除了数据的大小）。
2. 应用是否只对相对较小的数据量（最多 500 个字节）使用 TempData？如果是，Cookie TempData 提供程序将为每个携带 TempData 的请求增加较小的成本。如果不是，会话状态 TempData 提供程序有助于在使用TempData 前，避免在每个请求中来回切换大量数据。
3. 应用是否在多个服务器上的服务器场中运行？如果是，无需其他任何配置，即可在数据保护外使用 CookieTempData 提供程序
4. 大多数 Web 客户端（如 Web 浏览器）针对每个 Cookie 的最大大小和/或 Cookie 总数强制实施限制。使用 Cookie TempData提供程序时，请验证应用未超过这些限制。考虑数据的总大小。解释加密和分块导致的 Cookie 大小增加。

###### 配置 TempData 提供程序

# Design &&Pattern

### InverseofControl（IoC）

##### WHAT

“控制反转”或者“控制倒置”, 它体现的意思是控制权的转移，即原来控制权在A手中，现在需要B来接管。

###### Control

1. 222
2. 2222
3. 2222

##### WHERE

##### WHY

##### HOW

### DI

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### AOP

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### DDD

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### Git用法

##### 客户端操作

###### 设置账户

需要和github账户设置一致

* + - * 1. git config –global user.name xxx
        2. git config –global user.email xxx@foxmail.com

###### 查看设置

1. git config --list

###### 创建git本地库

* + - 1. git init (在当前目录下创建本地仓库)

###### 查看git状态

* + - * 1. git status(显示需要提交的文件和未追踪的文件)

uncommited已有的，修改过但未提交的文件

untracked原先没有的，新建的

###### 添加git文件到暂存区(需要和版本库区分)

* + - 1. git add <name>

###### 提交文件

1. git commit -m “add a commit remark”
2. -m表示注释

###### 删除文件（夹），并没有提交到服务器

* + - 1. git rm test.txt // 删除文件
      2. git rm -r filebook //删除文件夹

###### 删除文件（夹），并提交到服务器

* + - 1. git rm test.txt => git commit -m “delete a file”
      2. rm test.txt => git commit -am “delete a file”
      3. 命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

###### 操作日志

* + - 1. **git log --decorate --graph --oneline --all** #显示当前及之前的版本号
      2. **git log --pretty=oneline** #将版本历史显示为一行，历史版本号全部显示
      3. **git log --pretty=oneline --abbrev-commit** #将版本历史显示为一行，历史版本号部分显示
      4. **git log --graph**  #查看分支合并图

###### 版本回退

执行版本退回后，本地工作区的内容会自动和回退到的版本库版本的内容保持同步

* + - * 1. **git reset --hard HEAD^**回退到上一个版本
        2. **git reset --hard HEAD^^**回退到上上个版本，以此类推，一次提交即为一个版本
        3. **git reset --hard *e9efa77***回退到***e9efa77***版本

###### 还原操作

丢弃工作区的操作，但不会丢失暂存区的操作(add操作能将更改添加到暂存区)，实际上就是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”

* + - * 1. **git checkout -- readme.txt**

###### 暂存区撤销操作

工作区修改了文件，而且执行了add，没执行commit操作，暂存区还是可以撤销的

* + - * 1. **git reset HEAD readme.txt**
        2. 备注：git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

##### 操作服务器

###### 配置远程仓库免密登陆

1. 在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有**id\_rsa**和**id\_rsa.pub**这两个文件，如果已经有了，可直接跳到下一步。
2. 如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：ssh-keygen -t rsa -C "xxx@foxmail.com" 一路回车，执行生成 id\_rsa 私钥和 id\_rsa.pub 公钥，Windows用户在git bash中输入上述指令
3. 获得key的内容，复制下来，添加到gitHub的SSH key中
   1. **windows位置：C:\Users\用户名\.ssh\id\_rsa.pub**‬‬
   2. **Linux位置：cat ~/.ssh/id\_rsa.pub**
4. **ssh -T git@github.com** #验证key，根据提示输入yes，添加为信任主机

###### 添加远程仓库

1. **git remote add origin https://github.com/xxx/LearnGit.git（https方式）**
2. 此处可以为https地址也可以是ssh地址，orign为设置的远程仓库的别名，强烈建议使用ssh方式，因为https方式每次都要输入用户名和密码
3. 修改传输协议
   1. **git remote rm <远程主机名>**（删除远程仓库）
   2. **设置传输方式和目标远程仓库**
   3. **git push -u <远程主机名><本地分支名>**
4. 因为git本身是分布式版本控制系统，可以同步到另外一个远程库，当然也可以同步到另外两个远程库，所以**一个本地库可以既关联GitHub，又关联码云**！
   1. **git remote rm origin**
   2. 先关联GitHub的远程库：

**git remote add github** [**git@github.com:xxx/LearnGit.git**](mailto:git@github.com:xxx/LearnGit.git)

远程库的名称叫github，不叫origin了

* 1. 再关联码云的远程库

**git remote add gitee git@gitee.com:xxx/LearnGit.git**

* 1. 推送到GitHub，使用命令

**git push github master**

* 1. 推送到码云，使用命令

**git push gitee master**

###### 查看远程仓库及传输协议

1. **git remote**
2. **git remote -v 查看名称和详细地址**

###### 删除远程仓库

1. **git remote remove <远程主机名>**

###### 推送本地分支到远程仓库

1. **git push <远程主机名><本地分支名>:<远程分支名>(**如果省略远程分支名，则表示将本地分支推送与之存在“追踪关系”的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。**)**
2. **git push origin <本地分支名>**
3. **git push origin master**
4. 如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样以后就可以不加任何参数使用git push。
5. **git push -u <远程主机名><本地分支名> 例如：git push -u origin master**

###### 将远程仓库克隆为本地仓库

1. **git clone** [**git@github.com:xxx/LearnGit.git**](mailto:git@github.com:xxx/LearnGit.git)
   1. 不能使用别名
   2. 默认情况下，从远程clone到本地的库只能看到master分支
2. 将远程的分支同步到本地

**git checkout -b <本地分支名><远程主机名>/<远程分支名>**

前提是远程<远程主机名>必须存在名为<远程分支名>的分支，而且<本地分支名>和<远程分支名>最好一致。

###### 本地仓库更新

将远程存储库中的更改合并到当前分支中。在默认模式下，git pull是git fetch后跟git merge FETCH\_HEAD的缩写。更准确地说，git pull使用给定的参数运行git fetch，并调用git merge将检索到的分支头合并到当前分支中。使用--rebase，它运行git rebase而不是git merge。

1. 拉取分支
   1. **git pull <远程主机名><远程分支名>:<本地分支名>**
   2. 比如，要取回origin主机的next分支，与本地的master分支合并

**git pull origin next:master**

* 1. 如果远程分支(next)要与当前分支合并，则冒号后面的部分可以省略

**git pull origin next**

* 1. 表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再执行git merge。

**git fetch origin** =>**git merge origin/next**

1. 手动建立追踪关系

在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系(tracking)。比如，在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是说，本地的master分支自动“追踪”origin/master分支。

* 1. **git branch --set-upstream-to=远程主机名/<远程分支名>  <本地分支名>**
  2. 比如git branch --set-upstream-to=origin/next master，指定master分支追踪origin/next分支

1. 分支合并
   1. 本地当前分支自动与对应的origin主机”追踪分支”(remote-tracking branch)进行合并

**git pull origin**

* 1. 如果当前分支只有一个追踪分支，连远程主机名都可以省略

**git pull**

表示，当前分支自动与唯一一个追踪分支进行合并。

1. 采用rebase模式合并
   1. **git pull --rebase <远程主机名><远程分支名>:<本地分支名>**
2. **git fetch和git pull的区别**
   1. git fetch：相当于是从远程获取最新版本到本地，不会自动合并。
      1. **git fetch origin mastergit log -p master..origin/mastergit merge origin/master**
      2. 以上命令的含义：

首先从远程的origin的master主分支下载最新的版本到origin/master分支上

然后比较本地的master分支和origin/master分支的差别

最后进行合并

* + 1. 上述过程其实可以用以下更清晰的方式来进行：

**git fetch origin master:tmpgit diff tmp git merge tmp**

* 1. git pull：相当于是从远程获取最新版本并merge到本地
     1. **git pull origin master**
     2. 上述命令其实相当于git fetch 和 git merge
     3. **在实际使用中，git fetch更安全一些，因为在merge前，可以查看更新情况，然后再决定是否合并。**

###### 查看分支

1. git branch

###### 创建分支

1. **git branch <name>**

###### 创建并切换到分支

1. **git checkout -b <name>**
2. git checkout命令加上-b参数表示创建并切换，相当于以下两条命令
   1. git branch <name>git checkout <name>

###### 切换分支

1. **git checkout <name>**

切换分支后，在git bash中显示为绿色

###### 删除分支

1. **git branch -d <name>**
   1. 如果分支没有合并，删除分支就表示会丢失修改，此时git无法使用-d删除，可使用-D强行删除
   2. **git branch -D <name>**

###### 合并分支

git合并默认使用Fast forward模式，一旦删除分支，会丢掉分支信息，也就看不出来曾经做过合并

1. **git merge <name>** #基于当前分支，合并另外一个分支，前提需要保证分支之间不冲突
2. 如果强制禁用Fast forward模式，即普通模式，Git就会在merge时生成一个新的commit
   1. **git merge --no-ff -m "there is a comment" <name>**

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

* 1. **工作中，肯定需要不管有没有分支被删除，都要从分支历史上就查看所有的历史分支信息，所以要使用普通模式合并。**

###### 创建tag

1. **git tag <tagname>**#默认在HEAD版本
2. 对指定的commit版本创建tag
   1. 需要先找到历史commit的id

**git log --pretty=oneline --abbrev-commit**

* 1. 然后对指定的commit创建tag：

**git tag <tagname><commitid>**

1. 创建带有说明的tag，用-a指定标签名，-m指定说明文字

**git tag -a <tagname> -m "there is a tag description" [<commitid>]**

1. 通过-s用私钥签名一个标签，签名采用PGP签名

**git tag -s <tagname> -m "there is a tag description" [<commitid>]**

必须首先安装gpg（GnuPG），如果没有找到gpg，或者没有gpg密钥对，就会报错，参考GnuPG帮助文档配置Key。

###### 查看tag

1. **查看tag**

**git tag** #显示的tag不是按时间顺序排列，而是按字母顺序排列

1. 如果想查看tag和commit的对应关系

**git log --pretty=oneline --abbrev-commit**

1. 如果想查看tag的的详细情况

**git show <tagname>**

###### 删除tag

创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除：

1. 删除本地tag

**git tag -d <tagname>**

1. 删除服务器 tag
   1. 先本地删除：**git tag -d <tagname>**
   2. 再远程删除：**git push origin :refs/tags/<tagname>**

###### 推送标签至远程

1. 推送标签至远程

**git push origin <tagname>**

1. 一次性推送全部尚未推送到远程的本地标签

**git push origin --tags**

###### 现场的保存与恢复

1. 将目前的工作现场保存

**git stash**

1. 查看所有保存的工作现场

**git stash list**

1. 工作现场存在，恢复stash内容
   1. **git stash apply stash@{0}**

恢复后，stash内容并不删除，你需要用**git stash drop stash@{0}**来删除；

* 1. **git stash pop**

恢复的同时把stash内容也删了，这种方式省时省力

1. **注意点：**
   1. 如果在分支下新建文件，而尚未执行add操作，stash无法将新文件纳入保存的现场，因为stash只对被修改的被追踪的文件和暂存的变更有效，对于新文件必须先执行add。
   2. 如果修改分支下的已被追踪的文件，不管有没有对修改的文件进行add操作，如果执行stash，所有修改会被纳入保存的现场，而文件会恢复成修改前的状态。恢复现场后，文件又呈现被修改后的状态。特别的是，如果修改的文件在stash前已经被add了，恢复现场后，暂存区的内容就会清空，相当于这个文件从未被add一样。

###### 设置Git UI颜色

1. 让Git显示颜色，会让命令输出看起来更醒目

**git config --global color.ui true**

###### 忽略特殊文件

1. 在Git工作区的根目录下创建一个特殊的**.gitignore**文件，然后把要忽略的文件名填进去
2. <https://github.com/github/gitignore>
3. 忽略文件的原则是
   1. 忽略操作系统自动生成的文件，比如缩略图等；
   2. 忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
   3. 忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。
4. 把.gitignore也提交到Git
   1. **git add .gitignore**
   2. **git commit -m "there is a description"**
   3. 检验.gitignore的标准是git status命令是不是显示working tree clean
   4. **使用Windows的注意**：如果在资源管理器里新建一个.gitignore文件，系统会非常弱智地提示必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。
5. 如果确实想要添加已经被.gitignore忽略的文件，可以用-f强制添加到Git

**git add -f test.class**

1. 怀疑.gitignore写的有问题，需要查找哪个规则写错了，可以用git check-ignore命令检查

**git check-ignore -v App.class**.gitignore:3:\*.class App.class

表示.gitignore的第3行规则忽略了App.class这个文件，于是我们就可以知道应该修订哪个规则。

###### 为命令配置别名

1. 命令可以简写，用git st表示git status，再比如用co表示checkout、ci表示commit、br表示branch：
   1. **git config --global alias.co checkoutgit config --global alias.ci commitgit config --global alias.br branch**
   2. 以后提交就可以简写成：**git ci -m "there is a description"**
   3. --global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。
2. 命令git reset HEAD <filename>可以撤销暂存区的修改（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：
   1. **git config --global alias.unstage 'reset HEAD'**可以简化命令
   2. **git unstage test.py** = **git reset HEAD test.py**
3. 配置一个git last，让其显示最后一次提交信息：
   1. **git config --global alias.last 'log -1'**
   2. 用git last就能显示最近一次的提交
   3. git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

###### 修改配置文件

1. 配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。
2. 每个仓库的Git配置文件都放在.git/config文件中
3. 而当前用户的Git配置文件放在**用户主目录下的一个隐藏文件.gitconfig**中

##### 多人协作

###### 首先将远程仓库克隆为本地仓库

* + - * 1. **git clone git@github.com:xxx/LearnGit.git**

###### 在本地创建和远程分支对应的分支

* + - * 1. **git checkout -b <本地分支名> origin/<远程分支名>**

###### 本地和远程分支的名称最好一致；

1. 在本地分支完成任务后，可以试图用**git push <远程主机名><本地分支名>**推送自己的修改；
2. 如果推送失败，则表明远程分支比本地更新，需要先用**git pull**试图合并；
3. 如果pull失败并提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令**git branch --set-upstream-to=<远程主机名>/<远程分支名>  <本地分支名>**创建链接；
4. 如果合并有冲突，则解决冲突，并在本地提交（add => commit）；
5. 没有冲突或者解决掉冲突后，再用**git push <远程主机名><本地分支名>**推送就能成功。

### Git用法

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

# Client

### Angular

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### Redis

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### 负载均衡

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### 容灾备份

##### 委托

###### 定义

1. 222
2. 2222
3. 2222

### 读写分离

##### 委托

###### 定义

1. 222
2. 2222
3. 2222