目录

ASP.Net MVC 核心基础 2
一、ASP.Net MVC 简介 2
二、ASP.net MVC 起步 3
三、Razor 语法： 5
四、知识点补充和复习（看着老师反编译即可） 7
五、Controller 给 View 传递数据的方式： 7
六、关于 Action 的参数： 8
七、View 的查找 8
八、其他类型的 ActionResult 9
九、杂项 Misc 11
十、过滤器（Filter） 16
Nuget 18
一、NuGet 简介 18
二、Nuget.org 寻宝 19
三、图形界面 19
四、命令行的使用 21
Entity Framework 22
一、 相关知识复习 22
二、 高级集合扩展方法 23
三、 linq 25
四、 C#6.0 语法 27
五、 Entity Framework 简介 28
六、 EF 的安装 28
七、 EF 简单 DataAnnotations 实体配置 28
八、 EF 模型的两种配置方式 29
九、 FluentAPI 配置 T_Persons 的方式 30
十、 EF 的基本增删改查 31
十一、 EF 原理及 SQL 监控 32
十二、 执行原始 SQL 32
十三、不是所有 lambda 写法都能被支持 33
十四、EF 对象的状态 34
十五、Fluent API 更多配置 36
十六、一对多关系映射 37
十七、 配置一对多关系： 38
十八、多对多关系配置 40
十九、延迟加载（LazyLoad） 43
二十、不延迟加载，怎么样一次性加载？ 44
二十一、延迟加载的一些坑 45
二十二、实体类的继承 46
二十三、其他 48

ASP.Net MVC+Entity Framework 的架构 48
一、 了解一些不推荐的做法 48
二、 EO、DTO、ViewModel 49
三、 多层架构 49

ASP.Net MVC 核心基础

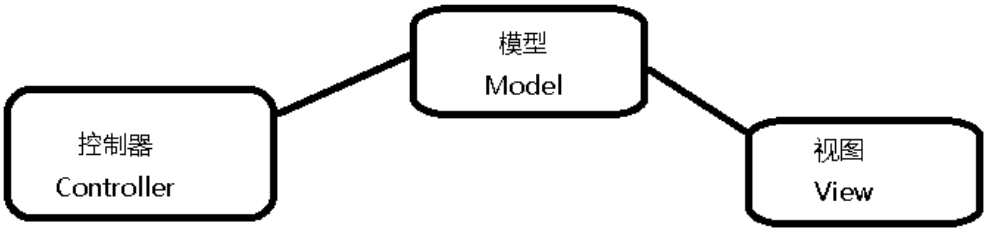
一、ASP.Net MVC 简介

(一) 什么是 ASP.Net MVC？HttpHandler 是 ASP.net 的底层机制，如果直接使用 HttpHandler 进行开发难度比较大、工作量大。因此提供了 ASP.Net MVC、ASP.Net WebForm 等高级封装的框架，简化开发，他们的底层仍然是 HttpHandler、HttpRequest 等这些东西。比如 ASP.Net MVC 的核心类仍然是实现了 IHttpHandler 接口的 MVCHandler。

(二) ASP.net WebForm、和 ASP.net MVC 的关系？都是对 HttpHandler 的封装框架，ASP.net WebForm 是微软为了让开发 ASP.Net 像开发 WinForm 一样傻瓜化发明的框架，有很多缺点；ASP.net MVC 采用了 MVC 的思想，更适合现代项目的开发，因此 ASP.net MVC 在逐步取代 ASP.Net WebForm。

(三) 为什么 ASP.Net MVC 更好？程序员有更强的掌控力，不会产生垃圾代码；程序员能够更清晰的控制运行过程，因此安全、性能、架构等更清晰。WebForm 和 ASP.netMVC 在“入门”和“深入”两个要素之间正好相反。

(四) 什么是 MVC 模式？模型（Model）、视图（View）、控制器（Controller）



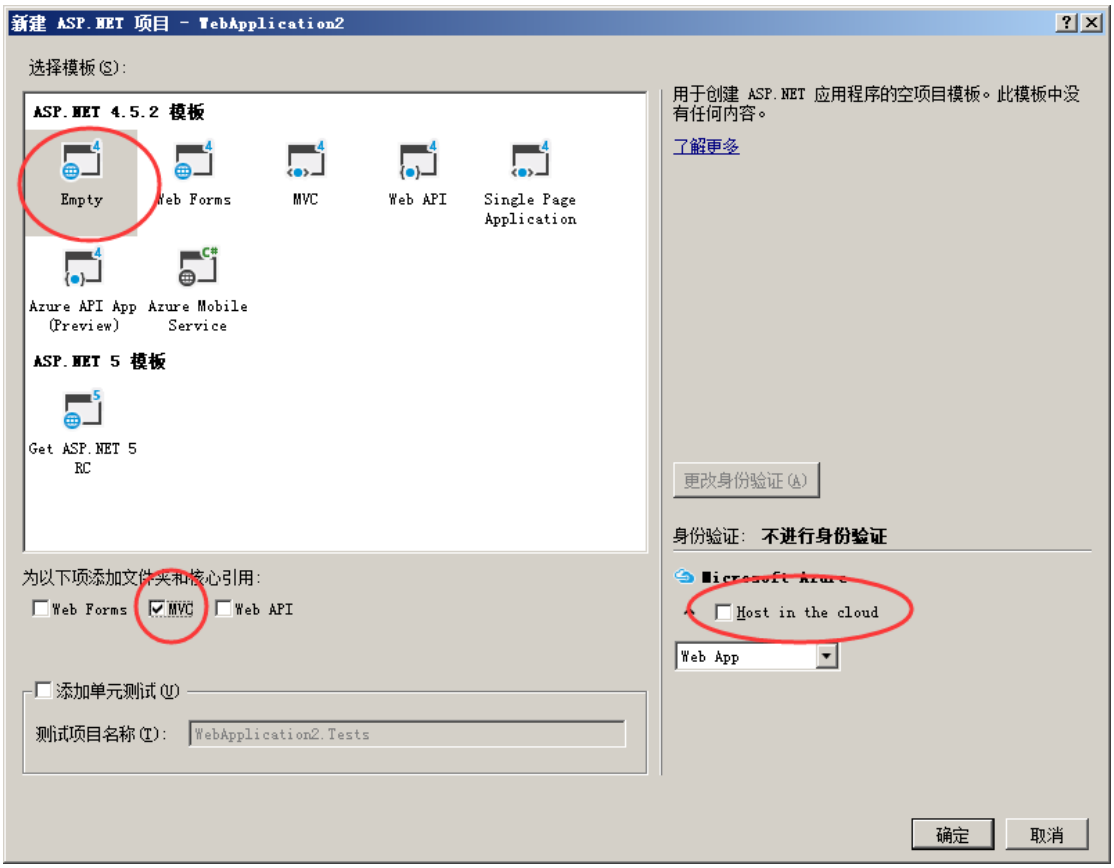
Model 负责在 View 和控制器之间进行数据的传递：用户输入的内容封装为 Model 对象，发给 Controller；要显示的数据由 Controller 放到 Model 中，然后扔给 View 去显示。Controller 不直接和 View 进行交互。

(五) ASP.net MVC 与“三层架构”没有任何的关系。三层架构中的 UI 层可以用 ASP.Net MVC 来实现。

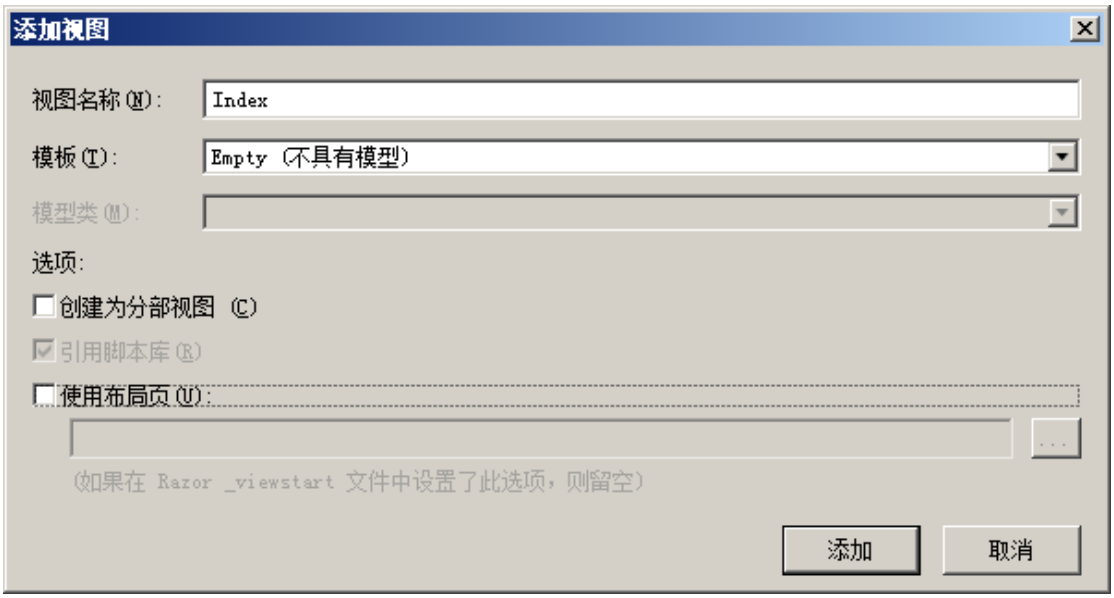
(六) “约定大于配置”：恶心的“配置文件地狱”，基础阶段按照默认配置来，先不管复杂、难懂的“路由”等。

二、ASP.net MVC 起步

1) 项目的创建：讲课使用 VS2015，用 VS2013 也可以，但是建议大家和老师的版本一致，特别是后续项目会用到 VS2015 的新特性（VS2015 中没有“右键”→“解析”，要用“Ctrl+.”）。新建项目→Visual C#→Web→【ASP.Net Web 应用程序】，不要勾选【将 Application Insights 添加到项目】，然后【确定】。在下一步的界面中选中“Empty”（初学者不要用 MVC 的模板项目，会太乱），勾选【MVC】，不要勾选【Host in the cloud】。一定注意：上面图标选“empty”，不要选“MVC”；下面勾选 MVC，否则会生成很多没用的代码。



2) 控制器的建立、视图的建立：在 Controllers 文件夹下点右键→【添加】→【控制器】→选择【MVC5 控制器-空】，类的名字以 Controller 结尾，比如“TestController”，会自动在 Views 文件夹下创建一个 Test 文件夹（如果不新建就手动建，这个文件夹的名字必须是 TestController 去掉 Controller），在 Views/Test 下新建视图 Index（和 TestController 的 Index 方法一致）：添加→视图



3) 新建一个用来收集用户参数的类：IndexReqModel（类名无所谓）包含 Num1、Num2 两个属性（只要不重名，大小写都可以）。然后声明一个 IndexRespModel 类用来给 view 传递数据显示，有 Num1、Num2、Result。也可以同一个类实现，但是这样写看起来比较清晰。

```
public class TestController : Controller
{
    0 个引用
    public ActionResult Index(IndexReqModel model)
    {
        IndexRespModel resp = new IndexRespModel();
        resp.num1 = model.Num1;
        resp.num2 = model.Num2;
        resp.result = model.Num1 + model.Num2;
        return View(resp);
    }
}
```

4) 在 Index.cshtml 如下编写

```
@model WebApplication2.Models.IndexRespModel

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <input type="text" value="@Model.num1" />+
    <input type="text" value="@Model.num2" />=@Model.result
</body>
</html>
```

5) 调试启动后，浏览器访问：http://ip 地址/Test/Index?num1=1&num2=5

画图分析执行过程、数据流动过程：当用户访问“/Test/Index?num1=1&num2=5”的时

候，会找到 Controllers 下的 TestController 的 Index 方法去执行，把请求参数按照名字填充到 Index 方法的参数对象中（MVC 引擎负责创建对象，给数据复制，并且进行类型的转换），reutrn View(resp)就会找到 Views 下的和自己的“类名、方法名”相对应的 Index.cshtml，然后把数据 resp 给到 Index.cshtml 去显示。@model (要小写) WebApplication2.Models.IndexRespModel 表示传递过来的数据是 IndexRespModel 类型的，@Model（大写开头）指向传递过来的对象。

cshtml 模板就是简化 HTML 的拼接的模板，最终还是生成 html 给浏览器显示，不能直接访问 cshtml 文件。

三、Razor 语法：

1. Razor 语法非常简单，@启动的区域为标准的 C#代码，其他部分是普通的 html 代码。
2. 用法：
   a) @{string a="abc";}@a @{C#代码块}。有标签就是 html 代码
   b) @Model
   c) @Model.dog.Name
   d) @if(),@foreach()等 C#语句
3. 下面的代码是不行的，因为纯文字被视为 C#代码：

```
if(Model.IsOK)
{
        启用
}
```

要使用“@:”前缀（不推荐），如下：

```
if(Model.IsOK)
{
        @:启用
}
```

如果要在代码区块中输出大量文字，只要在代码前后加上 Html 标签即可

```
if(Model.IsOK)
{
        <span>启用</span>
}
```

razor 会智能识别哪块是 C#，哪块是 HTML，HTML 中想运行 C#代码就用@，想在 C#中代码中输入 HTML 就写“HTML 标签”。

但是如果由于样式等原因不详加上额外的标签，那么可以用<text></text>标记，特殊的<text>不会输出到 Html 中。

4. 不要习惯性在@item 后写分号
5. Razor 理解 HTML 标记语言的结构，当<li>标签关闭的时候他也可以自动转回代码：

```
@foreach(var item in strs)
{
    <li>yes @item</li>
}
```

6. Razor 语法 ：razor 会自动识别哪块是普通字符，哪块是表达式，主要就是根据特殊符号来分辨（“识别到这里是否能被当成一个合法的 C#语句”）。不能这样写<a

href="Course@CourseId.ashx">否则 razor 会认为 ashx 是 CourseId 的一个属性，应该加上()强制让引擎把 CourseId 识别成一个单独的语法<a href="Course(@CourseId).ashx">，不确定的地方就加个括号。通过 VS 中编辑器的代码着色也可以分辨出来。

7. 如果不能自动提示，把页面关掉再打开就行了。如果还不能自动提示，只要运行没问题就行。cshtml 文件中如果有警告甚至错误，只要运行没问题就没关系。
8. 333@qq.com，razor 会自动识别出来是邮箱，所以 razor 不会把 @qq.com 当成 qq 对象的 com 属性。但是对于特殊的邮箱或者就是要显示@，那么可以使用@转义@，也就是“@@”
9. item_@item.Length 会把 @item.Length 识别成邮箱，因此用上()成为：<li>item_@(item.Length)
10. 易错，要区分 C#代码和 html 代码，下面是对的：style='display: @(message.IsHide ? "none" : "block")' 下面是错误的： style="display: (@message.IsHide) ? none : block" 为了避免 C#中的字符串的""和 html 的属性值的""冲突，建议如果 html 属性中嵌入了 C#代码，那么 html 的属性的值用单引号。
11. Razor 的@会自动把内容进行 htmlencode 输出，避免了 XSS 攻击，如果不想编码输出，那么用@Html.Raw()
12. razor 注释使用@*注释内容*@，不过谁会在 cshtml 中写注释？？？
13. （*）razor 中调用泛型方法的时候，由于<>会被认为是 html 转回标记模式，因此要用圆括号括起来，比如@(Html.Test<String>)，()永远是很强大的。不过 View 中一般不会调用复杂的方法。
14. 如果 cshtml 中任何 html 标签的属性中以"~/"开头，则会自动进行虚拟路径的处理，当然一般是给<script>的 src 属性、<link>的 href 属性、<a>标签的 href 属性、<img>的 src 属性用的。
15. html 标签的任何属性的值如果是 C#的值，那么如果是 bool 类型的值，那么如果值是 false，则不会渲染这个属性，如果值是 true，则会渲染成“属性名=属性名”，比如：
16.
```
@{
    bool b1 = true;
    bool b2 = false;
}
```
17. <aaa aa="/1.html" checked="@b1" ac="@b2">aaa</aaa>
18. 这个特性是为<input type="radio/checkbox">的 checked 属性和<select>的<option>的 selected 属性使用的，这样避免了进行三元运算符判断。
19. （看着老师反编译即可）cshtml 是编译生成一个动态的程序集；在 cshtml 中写@this.GetType().Assembly.Location 可以拿到编译生成的程序集的 dll 文件的路径，反编译可以看到 cshtml 最终生成一个类，类中就是在拼接 html；类是继承自 WebViewPage，后续用的@Model、@Html 等都是 WebViewPage 类的成员。
20. 尽可能维持 View 的简单，不要在 View 中写业务逻辑以及过去复杂的代码；

总结：

1、@就是 C#，<aaa></aaa>就是 html

2、如果想让被识别成 html 的当成 C#那就用@()

3、如果想让被识别成 C#的当成 html，用等标签，如果不想生成额外的标签，就用<text></text>

4、如果不想对内容 htmlencode 显示就用@Html.Raw

5、属性的值如果以"~/"开头会进行虚拟路径处理

6、属性值如果是 bool 类型，如果是 false 就不输出这个属性，如果 true 就输出“属性名=属性名”<input type="checkbox" checked="@b1"/>

四、知识点补充和复习（看着老师反编译即可）

1、dynamic 是 C#中提供的一个语法，可以实现像 JavaScript 一样的动态语言，可以到运行的时候再去发现属性的值或者调用方法。

```
dynamic p = new Person();
p.Name = "rupeng.com";
p.Hello();
```

这样即使没有成员：p.Age=3;编译也不报错，只有运行的时候才报错。

dynamic 的好处是灵活，坏处是不容易在开发时候发现错误、而且性能低（反射看看）。

如果 dynamic 指向 System.Dynamic.ExpandoObject()对象，这样可以给对象动态赋值属性（不能指向方法）：

```
dynamic p = new System.Dynamic.ExpandoObject();
p.Name = "rupeng.com";
p.Age = 10;
Console.WriteLine(p.Name+","+p.Age);
```

2、var 类型推断

```
var i=3;
var s="abc";
```

编译器会根据右边的类型推断出来 var 是什么类型，反编译一下。

var 和 dynamic 的区别：var 是在编译的时候确定，dynamic 是运行的时候动态确定的；var 变量不能指向其他类型的，dynamic 可以。

3、匿名类型

匿名类型是 C#中提供的一个新语法：var p = new{Age=15,Name="rupeng.com"};这是创建一个匿名类的对象，这个类没有名字。反编译看一下（IL 模式下看）编译器生成了一个类，这个类是 internal（IL 中是 private）、属性是只读的、初始值是通过构造函数传递。

因为匿名类型的属性是只读的，所以匿名类型的属性是无法赋值；因为匿名类型是 internal，所以无法跨程序集访问其成员。

五、Controller 给 View 传递数据的方式：

1、ViewData：以 ViewData["name"]="rupeng";string s=(string)ViewData["name"]这样键值对的方式进行数据传送。

2、ViewBag：ViewBag 是 dynamic 类型的参数，是对 ViewData 一个动态类型封装，用起来更方便，和 ViewData 共同操作一个数据。ViewBag.name=""; @ViewBag.name。用 ViewBag 传递数据非常方便，但是因为 ASP.Net MVC 中的“Html 辅助类”等对于 ViewBag 有一些特殊约定，一不小心就跳坑了（http://www.cnblogs.com/rupeng/p/5138575.html），所以尽量不要用 ViewBag，而是使用 Model，虽然会麻烦“越麻烦工资越高”。

3、Model：可以在 Controller 中通过 return View(model)赋值，然后在 cshtml 中通过 Model 属性来访问这个对象；

如果在 cshtml 中通过“@model 类型”（注意 model 小写）指定类型，则 cshtml 中的 Model 就是指定的强类型的，这样的 cshtml 叫“强类型视图”；如果没有指定“@model 类型”， 则 cshtml 中的 Model 就是 dynamic。

六、关于 Action 的参数：

ASP.Net MVC 会自动对参数做类型转换；

对于 boolean 类型的参数（或者 Model 的属性），如果使用 checkbox，则 value 必须是 "true"，否则值永远是 false。对于 double、int 等类型会自动进行类型转换。

1、一个 Controller 可以有多个方法，这些方法叫 Action。通过“Controller 名字/方法名”访问的时候就会执行对应的方法。

2、Action 的三种类型的参数：普通参数、Model 类、FormCollection。

1）普通参数：Index(string name,int age)。框架会自动把用户请求的 QueryString 或者 Post 表单中的值根据参数名字映射对应参数的值，适用于查询参数比较少的情况。int 的可空问题。

2）Model 类。这种类叫 ViewModel。

3）FormCollection，采用 fc["name"]这种方法访问，类似于 HttpHandler 中用 ctx ["name"]。适用于表单元素不确定、动态的情况。

3、Action 的方法不能重载，所以如果一个 Controller 中不能同时存在这两个 Action：public ActionResult T1(string name)；public ActionResult T1(int age)

特殊情况：给 Action 方法上标注[HttpGet]、[HttpPost]，这样当发出 Get 请求的时候执行标注[HttpGet]的 Action 方法，当发出 Post 请求的时候执行标注[HttpPost]的 Action 方法。没有标注的，常见的应用：标注[HttpGet]的 Action 展示初始页面，标注[HttpPost]的 Action 处理表单提交请教。案例：做一个报名页面。

4、Action 参数可以一部分普通参数，一部分 Model；

5、Action 参数如果在请求中没有对应的值，则 Model 类的形式则取默认值：数字是 0、boolean 是 false、引用类型是 null。如果是普通参数的形式：会报错，如果允许为空，要使用 int?。可以使用 C#的可选参数语法来设定默认值。Index(string name="tom");

6、上传文件的参数用 HttpPostedFileBase 类型，注意不是 HttpPostedFile 类型；

七、View 的查找

1、return View()会查找 Views 的 Controller 名字的 Action 的名字的 cshtml；

2、return View("Action1")，查找 Views 的 Controller 名字下的“Action1.cshtml”，如果找不到则到特殊的 Shared 文件夹下找“Action1.cshtml”。

3、return View("Action1")中如何传递 model？return View("Action1",model)。陷阱：如果 model 传递的是 string 类型，则需要 return View("Action1",(object)str)为什么？看一下重载！

注意 return View("Action1")不是重定向，浏览器和服务器之间只发生了一次交互，地址

栏还是旧的 Action 的地址。这和重定向 return Redirct("/Index/Action1");不一样

应用：执行报错，return View("Error",(object)msg) 通用的报错页面。为了防止忘了控制重载，封装成一个通用方法。

做一个增删改查

八、其他类型的 ActionResult

1、View()是一个方法，它的返回值是 ViewResult 类型，ViewResult 继承自 ActionResult，如果你确认返回的是 View()，返回值写成 ViewResult 也行，但是一般没这个必要，因为那样就不灵活了。因为 ViewResult 还有其他子类。

2、RedirectResult，重定向，最终就是调用 response.Redirect()。用法：return Redirect("http://www.rupeng.com"); return Redirect("~/1.html");

3、ContentResult（用得少）

返回程序中直接拼接生成的文本内容。ContentResult Content(string content, string contentType)

4、文件

1）FileContentResult File(byte[] fileContents, string contentType) 返回 byte[]格式的数据

2) FileContentResult File(byte[] fileContents, string contentType, string fileDownloadName)：fileDownloadName 设定浏览器端弹出的保存建议的文件名。

3）FileStreamResult File(Stream fileStream, string contentType) 返回 Stream 类型的数据（框架会帮着 Dispose，不用也不能 Dispose）

4）FileStreamResult File(Stream fileStream, string contentType, string fileDownloadName)

5）FilePathResult File(string fileName, string contentType) 返回文件名指定的文件，内部还是流方式读取文件；

6）FilePathResult File(string fileName, string contentType, string fileDownloadName)

如果是返回动态生成的图片（比如验证码），则不用设置 fileDownloadName；如果是“导出学生名单”、“下载文档”等操作则要设定 fileDownloadName。

注意：如果在 Controller 中要使用 System.IO 下的 File 类，因为和 File 方法重名了，所以要用命名空间来引用了。

5、404

HttpNotFoundResult HttpNotFound()

6、JavaScriptResult JavaScript(string script)（用得少）

返回 JavaScript 代码字符串，和 return Content("alert('Hello World!');", "application/x-javascript");效果是一样的。因为违反分层的原则，尽量不要用。

7、Json

JsonResult Json(object data) 把 data 对象序列化成 json 字符串返回给客户端，并且设置 contentType 为"application/json"。

Json 方法默认是禁止 Get 请求的（主要为了防止 CSRF 攻击，举例：在 A 网站中嵌入一个请求银行网站给其他账号转账的 Url 的 img），只能 Post 请求。所以如果以 Get 方式访问是会报错的。如果确实需要以 Get 方式方式，需要调用 return Json(data, JsonRequestBehavior.AllowGet)

ASP.net MVC 默认的 Json 方法实现由如下的缺点：

1） 日期类型的属性格式化成的字符串是"\/Date(1487305054403)\/"这样的格式，在客户端要用 js 代码格式化处理，很麻烦。

2）json 字符串中属性的名字和 C#中的大小写一样，不符合 js 中“小写开头、驼峰命名”的习惯。在 js 中也要用大写去处理。

3）无法处理循环引用的问题（尽管应该避免循环引用），会报错“序列化类型为***的对象时检测到循环引用”

```
public class Parent
{
    public string Name { get; set; }
    public Child Child { get; set; }
}
public class Child
{
public DateTime BirthDay { get; set; }
public string Name { get; set; }
public Parent Father { get; set; }
}
public ActionResult T1(string name)
{
Child c = new Child();
c.BirthDay = DateTime.Now;
c.Name = "tom";

Parent father = new Parent();
father.Name = "tidy";

father.Child = c;
c.Father = father;

return Json(c,JsonRequestBehavior.AllowGet);
}
```

后面的项目阶段讲解决方案。Json.Net

8、重定向：

1）Redirect(string url)

2）RedirectToAction(string actionName, string controllerName)：其实就帮助拼接生成 url，最终还是调用 Redirect()，个人不喜欢 RedirectToAction，“少即是多”。注意别调错了重载。

3）RedirectToAction 和 return View 的区别：RedirectToAction 是让客户端重定向，是一个新的 Http 请求，所以无法读取 ViewBag 中的内容；return View()是一次服务器一次处理中的转移。

Redirect 和 return View 的区别：

1、 Redirect 是让浏览器重定向到新的地址；return View 是让服务器把指定的 cshtml 的内容运行渲染后给到浏览器；

2、 Redirect 浏览器和服务器之间发生了两次交互；return View 浏览器和服务器之间发生了 1 次交互

3、 Redirect 由于是两次请求，所以第一次设置的 ViewBag 等这些信息，在第二次是取不到；而 View 则是在同一个请求中，所以 ViewBag 信息可以取到。

4、 如果用 Redirect，则由于是新的对 Controller/Action 的请求，所以对应的 Action 会被执行到。如果用 View，则是直接拿某个 View 去显示，对应的 Action 是不执行的。

什么情况用 View？服务器端产生数据，想让一个 View 去显示的；

什么情况用 Redirect?让浏览器去访问另外一个页面的时候。

九、杂项 Misc

1、TempData

在 SendRedirect 客户端重定向或者验证码等场景下，由于要跨请求的存取数据，是不能放到 ViewBag、Model 等中，需要“暂时存到 Session 中，用完了删除”的需求：实现起来也比较简单：

存入：

```
Session["verifyCode"] = new Random().Next().ToString();
```

读取：

```
String code = (string) Session["verifyCode"];
Session["verifyCode"] = null;
if(code==model.Code)
{
    //...
}
```

ASP.Net MVC 中提供了一个 TempData 让这一切更简单。

在一个 Action 存入 TempData，在后续的 Action 一旦被读取一次，数据自动销毁。TempData 默认就是依赖于 Session 实现的，所以 Session 过期以后，即使没有读取也会销毁。

应用场景：验证码；

TempData、ViewBag、ViewData、Model

2、HttpContext 与 HttpContextBase、HttpRequest 与 HttpRequestBase、HttpPostedFile 与 HttpPostedFileBase。进行 asp.net mvc 开发的时候尽量使用****Base 这些类，不要用 asp.net 内核原生的类。HttpContext.Current(X)

1）在 Controller 中 HttpContext 是一个 HttpContextBase 类型的属性（真正是 HttpContextWrapper 类型，是对 System.Web.HttpContext 的封装），System.Web.HttpContext 是一个类型。这两个类之间没有继承关系。

System.Web.HttpContext 类型是原始 ASP.Net 核心中的类，在 ASP.Net MVC 中不推荐使用这个类（也可以用）。

2）HttpContextBase 能“单元测试”，System.Web.HttpContext 不能。

3）怎么样 HttpContextBase.Current？其实是不推荐用 Current，而是随用随传递。

4）HttpContextBase 的 Request、Response 属性都是 HttpRequestBase、HttpResponseBase 类型。Session 等也如此。

5）如果真要使用 HttpContext 类的话，就要 System.Web.HttpContext

3、Views 的 web.config 中的 system.web.webPages.razor 的 pages/namespaces 节点下配置 add 命名空间，这样 cshtml 中就不用 using 了。

```
<system.web.webPages.razor>
  <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, S
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Routing" />
      <add namespace="WebApplication2" />
    </namespaces>
  </pages>
</system.web.webPages.razor>
```

4、Layout 布局文件：

@RenderBody()渲染正文部分；cshtml 的 Layout 属性设定 Layout 页面地址；@RenderSection("Footer")用于渲染具体页面中用@section Footer{}包裹的内容，如果 Footer 是可选的，那么使用@RenderSection("Footer",false)，可以用 IsSectionDefined("Footer")实现“如果没定义则显示***”的效果。

5、可以在 Views 文件夹下建一个_ViewStart.cshtml 文件，在这个文件中定义 Layout，这样不用每个页面中都设定 Layout，当然具体页面也可以通过设定 Layout 属性来覆盖默认的实现；

6、@Html.DropDownList

如果在页面中输出一个下拉列表或者列表框，就要自己写 foreach 拼接 html，还要写 if 判断哪项应该处于选中状态

```
<select>
    @foreach(var p in (IEnumerable<Person>)ViewBag.list)
    {
        <option selected="@(p.Id==3)">@p.Name</option>
    }
</select>
```

asp.net mvc 中提供了一些“Html 辅助方法”（其实就是 Controller 的 Html 属性中的若干方法，其实是扩展方法）用来简化 html 代码的生成。

DropDownList 是生成下拉列表的。

1）DropDownList(this HtmlHelper htmlHelper, string name, IEnumerable<SelectListItem> selectList)

string name 参数用来设定 <select>标签的 name 属性的值，id 属性的值默认和 name 一致。

下拉列表中的项（<option>）以 SelectListItem 集合的形式提供，SelectListItem 的属性：

bool Selected：是否选中状态，也就是是否生成 selected="selected"属性；

string Text：显示的值，也就是<option>的 innerText 部分；

string Value：生成的 value 属性，注意是 string 类型；

案例：

```
public ActionResult DDL1()
{
    List<Person> list = new List<Person>();
    list.Add(new Person { Id=77,Name="lily",IsMale=false});
    list.Add(new Person { Id = 222, Name = "tom", IsMale = true });
    list.Add(new Person { Id = 36, Name = "lucy", IsMale = false });

    List<SelectListItem> siList = new List<SelectListItem>();
```

```
    foreach(var person in list)
    {
        SelectListItem slItem = new SelectListItem();
        slItem.Text = person.Name+"("+(person.IsMale?"男":"女")+")";
        slItem.Value = person.Id.ToString();
        if(person.Id==222)
        {
            slItem.Selected = true;
        }
        siList.Add(slItem);
    }
    ViewBag.list = list;
    return View(siList);
}

<div>
    @Html.DropDownList("pid",Model)
</div>
```

2）DropDownList(this HtmlHelper htmlHelper, string name, IEnumerable<SelectListItem> selectList, object htmlAttributes)

htmlAttributes 属性用来生成 select 标签的其他属性，通常以匿名类对象的形式提供，比如 new { onchange = "javascript:alert('ok')", style = "color:red", aaa = "rupeng", id = "yzk",@class="warn error" }

会生成如下的样子:

```
<select aaa="rupeng" class="warn error" id="yzk" name="pid" onchange="javascript:alert(&#39;ok&#39;)" style="color:red">
```

支持自定义属性，给你原样输出，具体什么含义自己定；由于 class 是关键字，所以不能直接用 class=""，要加上一个@前缀，这其实是 C#中给变量名取名为关键字的一种语法；id 默认和 name 一致，如果设定了 id 则覆盖默认的实现。

3）构造一个特殊的集合类 SelectList，他会自动帮着做现成集合的遍历

```
public ActionResult DDL2()
{
    List<Person> list = new List<Person>();
    list.Add(new Person { Id = 666, Name = "lily", IsMale = false });
    list.Add(new Person { Id = 222, Name = "tom", IsMale = true });
    list.Add(new Person { Id = 333, Name = "lucy", IsMale = false });

    SelectList selectList = new SelectList(list, "Id", "Name");
    return View(selectList);
}
    @Html.DropDownList("name",(SelectList)Model)
```

IEnumerable items 参数是用来显示的原始对象数据，string dataValueField 为“对象的哪个属性用做生成 value 属性”， string dataTextField 为“对象的哪个属性用做生成显示的文本属性”。

用 SelectList 的好处是简单，但是如果说要同时显示多个属性的时候，就只能用非 SelectList 的方

式了。

SelectList 还可以设定第四个参数：哪个值被选中。SelectList selectList = new SelectList(list, "Id", "Name",222);

一个坑：不能让 cshtml 中 DropDownList 的第一个 name 参数和 ViewBag 中任何一个属性重名 http://www.cnblogs.com/rupeng/p/5138575.html。

建议尽量不要通过 ViewBag 传递，都通过 Model 传递

7、@Html.ListBox 用法和@Html.DropDownList 一模一样

8、为什么不推荐使用“Html 辅助方法”，因为不符合复杂项目的开发流程，前端程序员看不懂这些。好处是可以把表单验证、绑定等充分利用起来，开发效率高，但是在互联网项目中开发效率并不是唯一关注因素。在 asp.net mvc6 中已经不推荐用 html 辅助方法的表单。

9、通过 Request.IsAjaxRequest()这个方法可以判断是否来自于 Ajax 请求，这样可以让 ajax 请求和非 ajax 请求响应不同的内容。原理：Ajax 请求的报文头中有 x-requested-with: XMLHttpRequest。如果使用 System.Web.HttpContext，那么是没有这个方法的，那么自己就从报文头中取数据判断。

10、数据验证：

1） asp.net mvc 会自动根据属性的类型进行基本的校验，比如如果属性是 int 类型的，那么在提交非整数类型的数据的时候就会报错。

注意 ASP.net MVC 并不是在请求验证失败的时候抛异常，而是把决定权交给程序员，程序员需要决定如何处理数据校验失败。在 Action 中根据 ModelState.IsValid **判断是否验证通过，如果没有通过下面的方法拿到报错信息：**

```
string errorMsg = WebMVCHelper.GetValidMsg(this.ModelState);

public static string GetValidMsg(ModelStateDictionary modelState)// 有两个 ModelStateDictionary 类，别弄混乱了。要使用 System.Web.Mvc 下的
{
    StringBuilder sb = new StringBuilder();
    foreach(var key in modelState.Keys)
    {
        if(modelState[key].Errors.Count<=0)
        {
            continue;
        }
        sb.Append("属性【").Append(key).Append("】错误：");
        foreach (var modelError in modelState[key].Errors)
        {
            sb.AppendLine(modelError.ErrorMessage);
        }
    }
    return sb.ToString();
}
```

2） ASP.Net MVC 提供了在服务器端验证请求数据的能力。**要把对应的 Attribute 标记到 Model 的属性上（标记到方法参数上很多地方不起作用）。**

常用验证 Attribute：

a) [Required] 这个属性是必须的
b) [StringLength(100)]，字符串最大长度 100；[StringLength(100,MinimumLength=10)] 长度要介于 10 到 100 之间
c) [RegularExpression(@"aa(\d)+bb")] 正则表达式
d) [Range(35,88)] 数值范围。字符串长度范围的话请使用 [StringLength(100,MinimumLength=10)]
e) [Compare("Email")]这个属性必须和 Email 属性值一样。
f) [EmailAddress] 要是邮箱地址
g) [Phone]电话号码，规则有限

3）验证 Attribute 上都有 ErrorMessage 属性，用来自定义报错信息。ErrorMessage 中可以用{0}占位符作为属性名的占位。

4）数据验证+Html 辅助类高级控件可以实现很多简化的开发，连客户端+服务器端校验都自动实现了，但是有点太“WebForm”了，因此这里先学习核心原理，避免晕菜。

11、自定义验证规则 ValidationAttribute

直接或者继承自 ValidationAttribute。

1）如果能用正则表达式校验的直接从 RegularExpressionAttribute 继承。

```
public class QQNumberAttribute : RegularExpressionAttribute
{
    public QQNumberAttribute() : base(@"^\d{5,10}$")//不要忘了^$
    {
        this.ErrorMessage = "{0}属性不是合法的 QQ 号，QQ 号需要 5-10 位数字";
        //设定 ErrorMessage 的默认值。使用的人也可以覆盖这个值
    }
}
```

手机号的正则表达式：@"^1(3[0-9]|4[57]|5[0-35-9]|7[01678]|8[0-9])\d{8}$"

2）直接继承自 ValidationAttribute，重写 IsValid 方法

比如校验中国电话号码合法性

```
public class CNPhoneNumAttribute:ValidationAttribute
{
    public CNPhoneNumAttribute()
    {
        this.ErrorMessage = "电话号码必须是固话或者手机，固话要是3-4位区号开头，手机必须以13、15、18、17开头";
    }

    //注意，不要override  ValidationResult IsValid(object value, ValidationContext validationContext)
    public override bool IsValid(object value)
    {
        if(value is string)
        {
            string s = (string)value;
```

```
                if(s.Length==13)//手机号
                {
                    if(s.StartsWith("13")|| s.StartsWith("15") || s.StartsWith("17") || s.StartsWith("18"))
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
                else if(s.Contains("-"))//固话
                {
                    //010,021    0755    0531
                    string[] strs = s.Split('-');
                    if(strs[0].Length==3|| strs[0].Length==4)
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
                else
                {
                    return false;
                }
            }
            else
            {
                return false;
            }
        }
    }
```

3）还可以让 Model 类实现 IValidatableObject 接口，用的比较少，不讲了。

十、过滤器（Filter）

AOP（面向切面编程）是一种架构思想，用于把公共的逻辑放到一个单独的地方，这样就不用每个地方都写重复的代码了。比如程序中发生异常，不用每个地方都 try...catch...只要在（Global 的 Application_Error）中统一进行异常处理。不用每个 Action 中都检查当前用户是否有执行权限，ASP.net MVC 中提供了一个机制，每个 Action 执行之前都会执行我们

的代码，这样统一检查即可。

一夫当关万夫莫开！

1、四种 Filter

在 ASP.Net MVC 中提供了四个 Filter（过滤器）接口实现了这种 AOP 机制：IAuthorizationFilter、IActionFilter、IResultFilter、IExceptionFilter。

1）IAuthorizationFilter 一般用来检查当前用户是否有 Action 的执行权限，在每个 Action 被执行前执行 OnAuthorization 方法；

2）IActionFilter 也是在每个 Action 被执行前执行 OnActionExecuting 方法，每个 Action 执行完成后执行 OnActionExecuted 方法。和 IAuthorizationFilter 的区别是 IAuthorizationFilter 在 IActionFilter 之前执行，检查权限一般写到 IAuthorizationFilter 中；

3）IResultFilter，在每个 ActionResult 的前后执行 IResultFilter。用的很少，后面有一个应用。

4）IExceptionFilter，当 Action 执行发生未处理异常的时候执行 OnException 方法。在 ASP.net MVC 中仍然可以使用“Global 的 Application_Error”，但是建议用 IExceptionFilter。

定义的类可以在 Global 中 GlobalFilters.Filters.Add(new XXXFilter());的方式添加为全局的过滤器。

2、IAuthorizationFilter 案例：只有登录后才能访问除了 LoginController 之外的 Controller。

1） 编写一个类 CheckAuthorFilter，实现 IAuthorizationFilter 接口

2） 在 Global 中注册这个 Filter：GlobalFilters.Filters.Add(new CheckAuthorFilter());

3） CheckAuthorFilter 中实现 OnAuthorization 方法。filterContext.ActionDescriptor 可以获得 Action 的信息：filterContext.ActionDescriptor.ActionName 获得要执行的 Action 的名字；filterContext.ActionDescriptor.ControllerDescriptor.ControllerName 为要执行的 Controller 的名字；filterContext.ActionDescriptor.ControllerDescriptor.ControllerType 为要执行的 Controller 的 Type；filterContext.HttpContext 获得当前请求的 HttpContext；如果给“filterContext.Result”赋值了，那么就不会再执行要执行的 Action，而是以“filterContext.Result”的值作为执行结果（注意如果是执行的 filterContext.HttpContext.Response.Redirect()，那么目标 Action 还会执行的）。

4） 检查当前用户是否登录，如果没有登录则 filterContext.Result = new ContentResult() { Content = "没有权限" }; 或者 filterContext.Result = new RedirectResult("/Login/Index");（最好不要 filterContext.HttpContext.Response.Redirect("/Login/Index");）

5） A 用户有一些 Action 执行权限，B 用户有另外一些 Action 的执行权限；

3、IActionFilter 案例：日志记录，记录登录用户执行的 Action 的记录，方便跟踪责任。

4、IExceptionFilter 案例：记录未捕获异常。

```
public class ExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext filterContext)
    {
        File.AppendAllText("d:/error.log", filterContext.Exception.ToString());
        filterContext.ExceptionHandled = true;//如果有其他的 IExceptionFilter 不再执行
        filterContext.Result = new ContentResult() { Content= "error" };
    }
}
```

然后 GlobalFilters.Filters.Add(new ExceptionFilter());

5、总结好处：一次编写，其他地方默认就执行了。可以添加多个同一个类型的全局 Filter，按照添加的顺序执行。

6、(*)非全局 Filter：只要让实现类继承自 FilterAttribute 类，然后该实现哪个 Filter 接口就实现哪个（四个都支持）。不添加到 GlobalFilters 中，而是把这个自定义 Attribute 添加到 Controller 类上这样就只有这个 Controller 中操作会用到这个 Filter。如果添加到 Action 方法上，则只有这个 Action 执行的时候才会用到这个 Filter。

Nuget

一、NuGet 简介

我们以前在 Windows 平台下安装软件的时候，要自己去网上搜索、下载、安装，但是有如下问题：下载的位置不好找；容易下载到带病毒的软件；可能会遇到和当前操作系统不兼容的问题；除非软件内置更新功能，否则很难完成更新。而使用 360 软件管家之类的软件：可以自动的完成软件的下载，360 帮着防止带病毒的软件；如果下载的软件和当前操作系统不兼容，下载之前就会提示；会自动提示更新。

手机上的应用市场也是有类似的好处。

我们进行软件开发的时候，经常会用到第三方的开发包（俗称 dll），比如 NPOI、MYSQL ADO.net 驱动等。如果自己去网上下载的问题：不好搜，不好找；容易下载到错误的；要自己进行安装配置；要选择和当前环境一致的版本（比如有的开发包在.net 2.0 和.net 4.5 中要用不同的版本）；这个开发包可能还要使用其他的安装包。

微软提供了 NuGet 这个软件，自动帮我们进行开发包的下载、安装，并且会根据当前的环境找到合适的版本，下载相关的依赖的开发包，还可以自动更新最新版本。

NuGet 在 VS2013 以上都提供了。有图形界面和命令行两种使用方式。

二、Nuget.org 寻宝

https://www.nuget.org

搜索；页面每块信息是什么意思；

三、图形界面

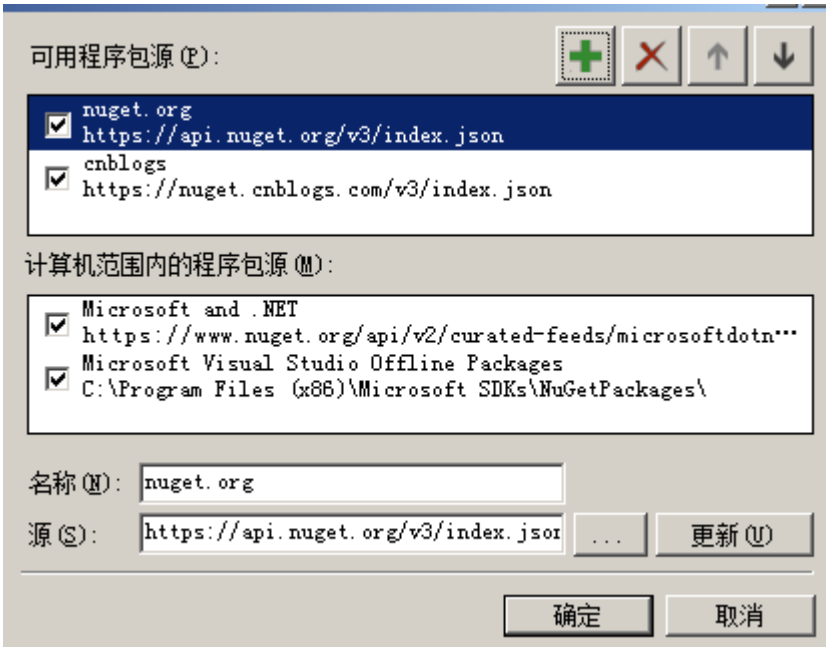
在项目的“引用”中点击右键【管理 Nuget 程序包】，在【浏览】中输入程序包的名字（比如 NPOI），然后搜索。

由于 nuget 的服务器在国外，可能你的网络连不上或者速度慢，可以从其他镜像站点下

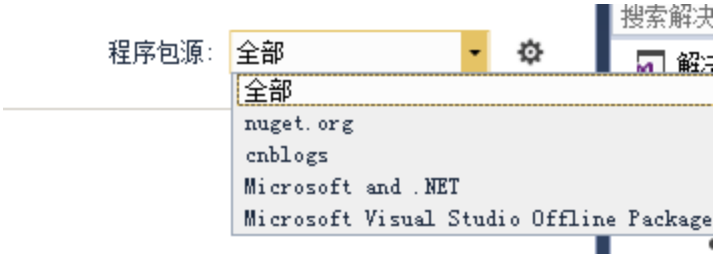
载。如果连的好好的，不用操作下面的：

1、 找一个可用的镜像站点，目前可用的是博客园的镜像，地址 **https://nuget.cnblogs.com/v3/index.json** 也许在你学习的时候这个已经不能用了，在参考资料中找其他镜像地址。

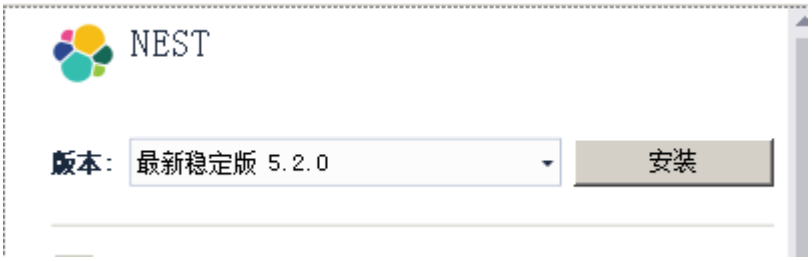
2、 在 VS 的【工具】→【选项】→【NuGet 包管理器】→【程序包源】，点击【添加】



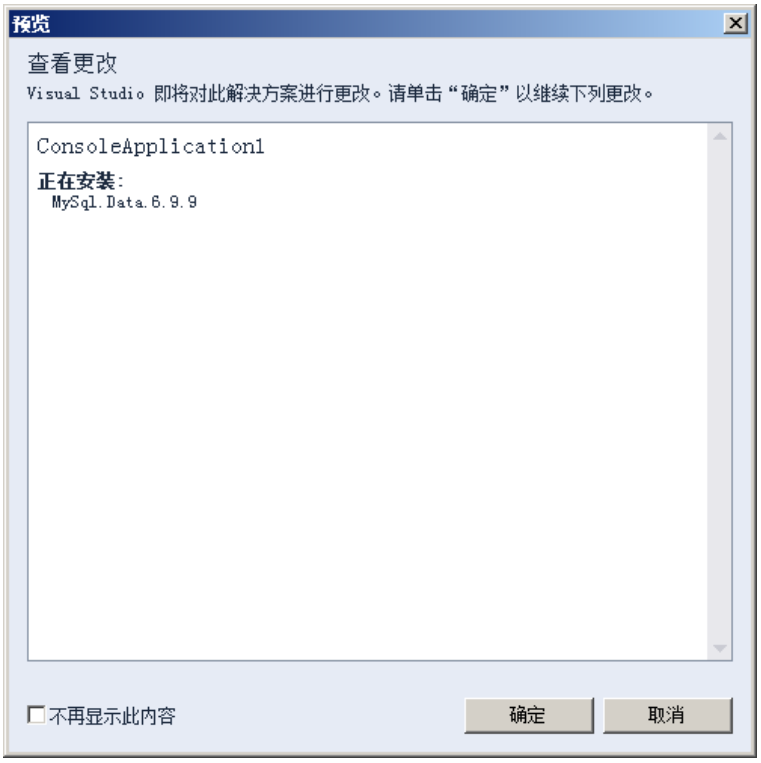
然后在搜索安装的时候在【程序包源】中选择我们添加的镜像



3、 在搜索结果中找到合适的结果，点击后在右侧选择合适的版本（建议和老师上课的版本一致，避免麻烦）



点击【安装】以后可能会弹出要求【同意】协议的对话框，点击【同意】即可。



在【输出】的【程序包管理器】中出现“========== 已完成 ==========”的时候说明安装完成，有可能会报错。

4、会自动添加引用。

5、有的安装包会自动修改 App.config 等配置文件。

6、Nuget 安装包信息在 packages.config 中，对应的安装包在解决方案的 packages 文件夹下，把项目拷给别人的时候没必要拷 packages 文件夹，别人拿到以后会自动下载恢复这些安装包。

7、如果要删除某个安装包，不能只删除引用，否则还会自动恢复、添加，还要手动删除 packages.config 中的内容。最好使用图形界面的“卸载”功能。

四、命令行的使用

命令行方式安装更方便、灵活。

1、在【程序包管理器控制台】视图中（如果没显示出来，则主菜单【工具】→【NuGet 包管理器】→【程序包管理器控制台】）。

输入：Install-Package 程序包的名字

安装完成的标志

```
执行 nuget 操作花费时间 3.91 sec
已用时间: 00:00:08.3942905
PM>
```

2、 可以在【程序包源】中指定镜像站点，【默认项目】指的是安装到哪个项目中

3、指定版本：:Install-Package 安装包 -Version 版本号，比如

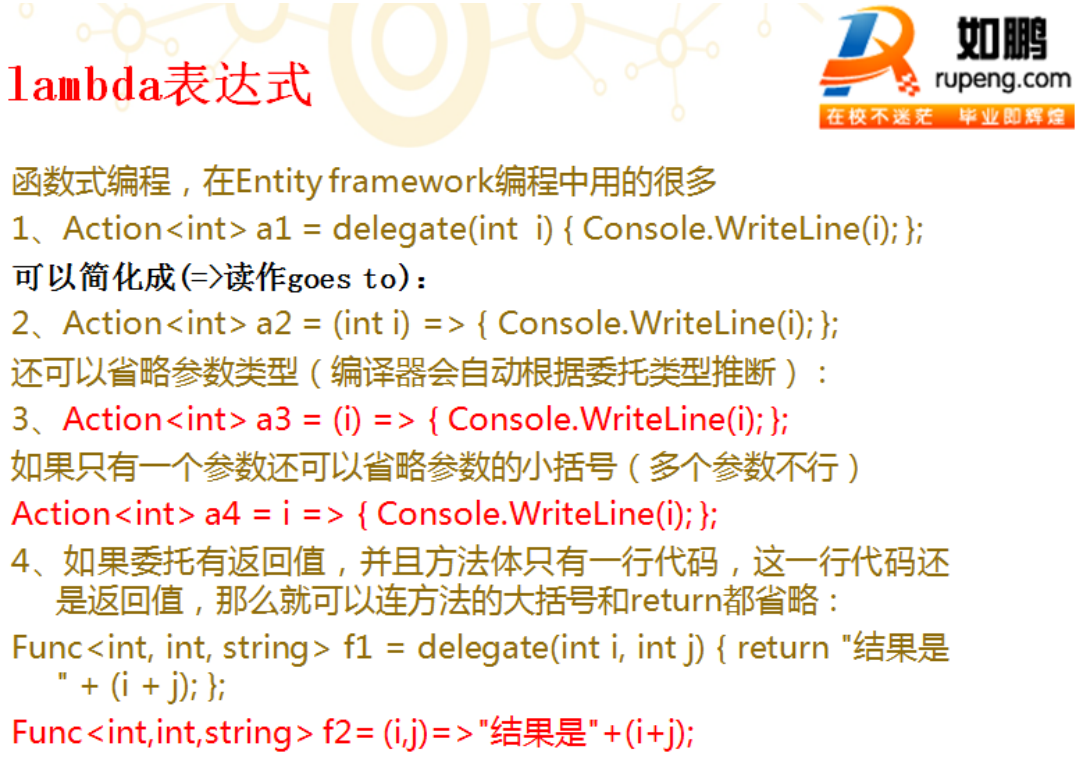
Install-Package MySql.Data -Version 6.8.8

3、 卸载 UnInstall-Package MySql.Data

Entity Framework

一、 相关知识复习

1. **var 类型推断：var p =new Person();**
2. **匿 名 类 型 。 var a =new {p.Name,Age=5,Gender=p.Gender,Name1=a.Name};//{p.Name}=={Name=p.Name}**
3. **给新创建对象的属性赋值的简化方法：Person p = new Person{Name="tom",Age=5}; 等价于 Person p = new Person();p.Name="tom";p.Age=5;**
4. **lambda 表达式：**



4、 集合常用扩展方法

- 集合常用扩展方法：

  Where（支持委托）、Select（支持委托）、Max、Min、OrderBy

  First（获取第一个，如果一个都没有则异常）

  FirstOrDefault（获取第一个，如果一个都没有则返回默认值）

  Single （获取唯一一个，如果没有或者有多个则异常）

  SingleOrDefault （获取唯一一个，如果没有则返回默认值，如果有多个则异常）

注意 lambda 中照样要避免变量重名的问题：var p =persons.Where(p => p.Name == "rupeng.com").First();

二、 高级集合扩展方法

//学生

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public bool Gender { get; set; }
    public int Salary { get; set; }

    public override string ToString()
    {
        return string.Format("Name={0},Age={1},Gender={2},Salary={3}",
            Name, Age, Gender, Salary);
    }
}
//老师
public class Teacher
{
    public Teacher()
    {
      this.Students=new List<Person>();
    }
    public string Name { get; set; }
    public List<Person> Students { get; set; }
}
var s0 =new Person { Name="tom",Age=3,Gender=true,Salary=6000};
var s1 = new Person { Name = "jerry", Age = 8, Gender = true, Salary = 5000 };
var s2 = new Person { Name = "jim", Age = 3, Gender = true, Salary = 3000 };
var s3 = new Person { Name = "lily", Age = 5, Gender = false, Salary = 9000 };
var s4 = new Person { Name = "lucy", Age = 6, Gender = false, Salary = 2000 };
var s5 = new Person { Name = "kimi", Age = 5, Gender = true, Salary = 1000 };
```

```
List<Person> list = new List<Person>();
list.Add(s0);
list.Add(s1);
list.Add(s2);
list.Add(s3);
list.Add(s4);
list.Add(s5);
Teacher t1 = new Teacher { Name="如鹏网张老师"};
t1.Students.Add(s1);
t1.Students.Add(s2);

Teacher t2 = new Teacher { Name = "如鹏网刘老师" };
t2.Students.Add(s2);
t2.Students.Add(s3);
t2.Students.Add(s5);

Teacher[] teachers = { t1,t2};
```

1. **Any()，判断集合是否包含元素，返回值是 bool，一般比 Coun()>0 效率高。Any 还可以指定条件表达式。** `bool b = list.Any(p => p.Age > 50);` 等价于 `bool b = list.Where(p=>p.Age>50).Any();`
2. **Distinct()，剔除完全重复数据。(*)注意自定义对象的 Equals 问题：需要重写 Equals 和 GetHashCode 方法来进行内容比较。**
3. **排序：升序** list.OrderBy(p=>p.Age)；降序 list.OrderByDescending(p=>p.Age)。指定多个排序规则，而不是多个 OrderBy，而是：list.OrderByDescending(p=>p.Age).ThenBy(p=>p.Salary)，也支持 ThenByDescending()。注意这些操作不会影响原始的集合数据。
4. Skip(n)跳过前 n 条数据；Take(n)获取最多 n 条数据，如果不足 n 条也不会报错。常用来分页获取数据。list.Skip(3).Take(2)跳过前 3 条数据获取 2 条数据。
5. **Except(items1)排除当前集合中在 items1 中存在的元素。用 int 数组举例。**
6. **Union(items1)把当前集合和 items1 中组合。用 int 数组举例。**
7. Intersect(**items1**) **把当前集合和 items1 中取交集。用 int 数组举例。**
8. **分组：**

```
foreach(var g in list.GroupBy(p => p.Age))
{
    Console.WriteLine(g.Key+":"+g.Average(p=>p.Salary));
}
```

9. **SelectMany：把集合中每个对象的另外集合属性的值重新拼接为一个新的集合**

```
foreach(var s in teachers.SelectMany(t => t.Students))
{
    Console.WriteLine(s);//每个元素都是 Person
}
```

注意不会去重，如果需要去重要自己再次调用 Distinct()

10. **Join**

```
class Master
```

```
{
    public long Id { get; set; }
    public string Name { get; set; }
}
class Dog
{
    public long Id { get; set; }
    public long MasterId { get; set; }
    public string Name { get; set; }
}
Master m1 = new Master { Id = 1, Name = "杨中科" };
Master m2 = new Master { Id = 2, Name = "比尔盖茨" };
Master m3 = new Master { Id = 3, Name = "周星驰" };
Master[] masters = { m1,m2,m3};

Dog d1 = new Dog { Id = 1, MasterId = 3, Name = "旺财" };
Dog d2 = new Dog { Id = 2, MasterId = 3, Name = "汪汪" };
Dog d3 = new Dog { Id = 3, MasterId = 1, Name = "京巴" };
Dog d4 = new Dog { Id = 4, MasterId = 2, Name = "泰迪" };
Dog d5 = new Dog { Id = 5, MasterId = 1, Name = "中华田园" };
Dog[] dogs = { d1, d2, d3, d4, d5 };
```

Join 可以实现和数据库一样的 Join 效果，对有关联关系的数据进行联合查询

下面的语句查询所有 Id=1 的狗，并且查询狗的主人的姓名。

```
var result = dogs.Where(d => d.Id > 1).Join(masters, d => d.MasterId, m => m.Id,
    (d,m)=>new {DogName=d.Name,MasterName=m.Name});
foreach(var item in result)
{
    Console.WriteLine(item.DogName+","+item.MasterName);
}
```

三、 linq

1、 简介

查询 Id>1 的狗有如下两种写法：

```
1）var r1 = dogs.Where(d => d.Id > 1);
2）var r2 = from d in dogs
      where d.Id>1
      select d;
```

第一种写法是使用 lambda 的方式写的，官方没有正式的叫法，我们就叫“lambda 写法”；

第二种是使用一种叫 Linq（读作：link）的写法，是微软发明的一种类似 SQL 的语法，给我们一个新选择。两种方法是可以互相替代的，没有哪个好、哪个坏，看个人习惯。我的经验：需要 join 等复杂用法的时候 Linq 更易懂，一般的时候“lambda 写法”更清晰，更

紧凑。

反编译得知，这两种写法最终编译成同样的东西，所以本质上一样的。

2、 辟谣

“Linq 被淘汰了”是错误的说法，应该是“Linq2SQL 被淘汰了”。linq 就是微软发明的这个语法，可以用这种语法操作很多数据，操作 SQL 数据就是 Linq2SQL，linq 操作后面学的 EntityFramework 就是 Linq2Entity，linq 操作普通.Net 对象就是 Linq2Object、Linq 操作 XML 文档就是 Linq2XML。

3、 linq 基本语法

以 from item in items 开始，items 为待处理的集合，item 为每一项的变量名；

最后要加上 select，表示结果的数据；记得 select 一定要最后。这是刚用比较别扭的地方。

看各种用法，不用解析：

1）

```
var r= from d in dogs
           select d.Id;
```

2）

```
var r = from d in dogs
           select new{d.Id,d.Name,Desc="一条狗"};
```

3） 排序

```
var items = from d in dogs
                //orderby d.Age
                //orderby d.Age descending
                orderby d.Age,d.MasterId descending
            select d;
```

4） join

```
var r9 = from d in dogs
      join m in masters on d.MasterId equals m1.Id
      select new { DogName=d.Name,MasterName=m.Name};
```

注意 join 中相等不要用==，要用 equals。

写 join 的时候 linq 比“lambda” 漂亮

5） group by

```
var r1 = from p in list
         group p by p.Age into g
         select new { Age = g.Key, MaxSalary = g.Max(p=>p.Salary), Count = g.Count() };
```

4、混用

只有 Where,Select,OrderBy,GroupBy,Join 等这些能用 linq 写法，如果要用下面的“Max,Min,Count,Average,Sum,Any,First,FirstOrDefault,Single,SingleOrDefault,Distinct,Skip,Take 等”则还要用 lambda 的写法（因为编译后是同一个东西，所以当然可以混用）。

```
var r1 = from p in list
      group p by p.Age into g
      select new { Age = g.Key, MaxSalary = g.Max(p=>p.Salary), Count = g.Count() };
int c = r1.Count();
var item = r1.SingleOrDefault();
```

```
var c = (from p in list
         where p.Age>3
         select p
         ).Count();
```

lambda 对 linq 说：论漂亮我不行，论强大你不行！

四、 C#6.0 语法

1. 属性的初始化“public int Age{get;set;}=6”。低版本.Net 中怎么办？
2. nameof：可以直接获得变量、属性、方法等的名字的字符串表现形式。获取的是最后一段的名称。如果在低版本中怎么办？

```
class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person();
        string s1 = nameof(p1);
        string s2 = nameof(Person);
        string s3 = nameof(p1.Age);
        string s4 = nameof(Person.Age);
        string s5 = nameof(p1.F1);
        Console.ReadKey();
    }
}

    public class Person
    {
    public int Age { get; set; }
    public string Name { get; set; }
    public void Hello()
    {

    }

    public static void F1()
    {

    }
}
```

好处：避免写错了，可以利用编译时检查。

应用案例：ASP.Net MVC 中的[Compare("BirthDay")]改成[Compare(nameof(BirthDay))]

4、 ??语法：int j = i ?? 3; 如果 i 为 null 则表达式的值为 3，否则表达式的值就是 i 的值。

如果在低版本中怎么办？int j = (i == null)?3:(int)i;

应用案例：string name = null;Console.WriteLine(name??"未知");

5、 ?.语法：string s8 = null;string s9 = s8?.Trim(); 如果 s8 为 null，则不执行 Trim()，让表达

式的结果为 null。在低版本中怎么办？string s9=null;if(s8!=null){s9=s8.Trim();};

五、 Entity Framework 简介

1、 ORM：Object Relation Mapping ，通俗说：用操作对象的方式来操作数据库。

2、 插入数据库不再是执行 Insert，而是类似于 Person p = new Person();p.Age=3;p.Name="如鹏网";db.Save(p);这样的做法。

3、 ORM 工具有很多 Dapper、PetaPoco、NHibernate，最首推的还是微软官方的 Entity Framework，简称 EF。

4、 EF 底层仍然是对 ADO.Net 的封装。EF 支持 SQLServer、MYSQL、Oracle、Sqlite 等所有主流数据库。

5、 使用 EF 进行数据库开发的时候有两个东西建：建数据库(T_Persons)，建模型类(Person)。根据这两种创建的先后顺序有 EF 的三种创建方法：

a) DataBase First（数据库优先）：先创建数据库表，然后自动生成 EDM 文件，EDM 文件生成模型类。简单展示一下 DataBase First 的使用。

b) Model First（模型优先）：先创建 Edm 文件，Edm 文件自动生成模型类和数据库；

c) Code First（代码优先）：程序员自己写模型类，然后自动生成数据库。没有 Edm。

DataBase First 简单、方便，但是当项目大了之后会非常痛苦；Code First 入门门槛高，但是适合于大项目。Model First……

无论哪种 First，一旦创建好了数据库、模型类之后，后面的用法都是一样的。业界都是推荐使用 Code First，新版的 EF 中只支持 Code First，因此我们这里只讲 Code First。

6、 Code First 的微软的推荐用法是程序员只写模型类，数据库由 EF 帮我们生成，当修改模型类之后，EF 使用“DB Migration”自动帮我们更改数据库。但是这种做法太激进，不适合很多大项目的开发流程和优化，只适合于项目的初始开发阶段。Java 的 Hibernate 中也有类似的 DDL2SQL 技术，但是也是用的较少。“DB Migration”也不利于理解 EF，因此在初学阶段，我们将会禁用“DB Migration”，采用更实际的“手动建数据库和模型类”的方式。

7、 如果大家用过 NHibernate 等 ORM 工具的话，会发现开发过程特别麻烦，需要在配置文件中指定模型类属性和数据库字段的对应关系，哪怕名字完全也一样也要手动配置。使用过 Java 中 Struts、Spring 等技术的同学也有过类似“配置文件地狱”的感觉。

像 ASP.Net MVC 一样，EF 也是采用“约定大于配置”这样的框架设计原则，省去了很多配置，能用约定就不要自己配置。

六、 EF 的安装

1、 基础阶段用控制台项目。使用 NuGet 安装 EntityFramework。会自动在 App.config 中中增加两个 entityFramework 相关配置段；

2、 在 web.config 中配置连接字符串

<add name="conn1" connectionString="Data Source=.;Initial Catalog=test1;User ID=sa;Password=msn@qq888" providerName="System.Data.SqlClient" />

易错点：不能忘了写 providerName="System.Data.SqlClient"

七、 EF 简单 DataAnnotations 实体配置

1、 数据库中建表 T_Perons，有 Id（主键，自动增长）、Name、CreateDateTime 字段。

2、 创建 Person 类

[Table("T_Persons")]//因为类名和表名不一样，所以要使用 Table 标注

```
public class Person
{
    public long Id { set; get; }
    public string Name { get; set; }
    public DateTime CreateDateTime { get; set; }
}
```

因为 EF 约定主键字段名是 Id，所以不用再特殊指定 Id 是主键，如果非要指定就指定[Key]。因为字段名字和属性名字一致，所以不用再特殊指定属性和字段名的对应关系，如果需要特殊指定，则要用[Column("Name")]

（*）必填字段标注[Required]、字段长度[MaxLength(5)]、可空字段用 int?、如果字段在数据库有默认值，则要在属性上标注[DatabaseGenerated]

注意实体类都要写成 public，否则后面可能会有麻烦。

3、 创建 DbContext 类（模型类、实体类）

```
public class MyDbContext:DbContext
{
    public MyDbContext():base("name=conn1")
        //name=conn1 表示使用连接字符串中名字为 conn1 的去连接数据库
    {
    }
    public DbSet<Person> Persons { get; set; }//通过对 Persons 集合的操作就可以完成对 T_Persons 表的操作
}
```

4、 测试

```
MyDbContext ctx = new MyDbContext();
Person p = new Person();
p.CreateDateTime = DateTime.Now;
p.Name = "rupeng";
ctx.Persons.Add(p);
ctx.SaveChanges();
```

注意：MyDbContext 对象是否需要 using 有争议，不 using 也没事。每次用的时候 new MyDbContext 就行，不用共享同一个实例，共享反而会有问题。SaveChanges()才会把修改更新到数据库中。

EF 的开发团队都说要 using DbContext，很多人不 using，只是想利用 LazyLoad 而已，但是那样做是违反分层原则的。我的习惯还是 using。

异常的处理：如果数据有错误可能在 SaveChanges()的时候出现异常，一般仔细查看异常信息或者一直深入一层层的钻 InnerException 就能发现错误信息。举例：创建一个 Person 对象，不给 Name、CreateDateTime 赋值就保存。

八、 EF 模型的两种配置方式

EF 中的模型类的配置有 DataAnnotations、FluentAPI 两种。上面这种在模型类上[Table("T_Persons")]、[Column("Name")]这种方式就叫 DataAnnotations

这种方式比较方便，但是耦合度太高，一般的类最好是 POCO（Plain Old C# Object，没有继承什么特殊的父类，没有标注什么特殊的 Attribute，没有定义什么特殊的方法，就是一堆普通的属性）；不符合大项目开发的要求。微软推荐使用 FluentAPI 的使用方式，因此后面主要用 FluentAPI 的使用方式。

九、 FluentAPI 配置 T_Persons 的方式

1. 数据库中建表 T_Perons，有 Id（主键，自动增长）、Name、CreateDateTime 字段。
2. 创建 Person 类。模型类就是普通 C#类

```
public class Person
{
public long Id { set; get; }
public string Name { get; set; }
public DateTime CreateDateTime { get; set; }
}
```

3. 创建一个 PersonConfig 类，放到 ModelConfig 文件夹下（PersonConfig、EntityConfig 这样的名字都不是必须的）

```
class PersonConfig: EntityTypeConfiguration<Person>
{
    public PersonConfig()
    {
        this.ToTable("T_Persons");//等价于[Table("T_Persons")]
    }
}
```

4. 创建 DbContext 类

```
public class MyDbContext:DbContext
{
    public MyDbContext():base("name=conn1")
    {
    }
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
      base.OnModelCreating(modelBuilder);
      modelBuilder.Configurations.AddFromAssembly(
                      Assembly.GetExecutingAssembly());
  }
    public DbSet<Person> Persons { get; set; }
}
```

下面这句话：

```
modelBuilder.Configurations.AddFromAssembly(Assembly.GetExecutingAssembly()
);
```

代表从这句话所在的程序集加载所有的继承自 EntityTypeConfiguration 为模型配置类。

还有很多加载配置文件的做法（把配置写到 OnModelCreating 中或者把加载的代码写死到 OnModelCreating 中），但是这种做法是最符合大项目规范的做法。

和以前唯一的不同就是：模型不需要标注 Attribute；编写一个 XXXConfig 类配置映射关系；

DbContext 中 override **OnModelCreating**；

5. 以后的用法和以前一样。
6. 多个表怎么办？创建多个表的实体类、Config 类，并且在 DbContext 中增加多个 **DbSet** 类型的属性即可。

十、 EF 的基本增删改查

获取 DbSet 除了可以 ctx.Persons 之外，还可以 ctx.Set<Person>()。

1） 增加，讲过了。一个点：如果 Id 是自动增长的，创建的对象显然不用指定 Id 的值，并且在 SaveChanges ()后会自动给对象的 Id 属性赋值为新增行的 Id 字段的值。

2） 删除。先查询出来要删除的数据，然后 Remove。这种方式问题最少，虽然性能略低，但是删除操作一般不频繁，不用考虑性能。后续在“状态管理”中会讲其他实现方法。

```
MyDbContext ctx = new MyDbContext();
var p1= ctx.Persons.Where(p => p.Id == 3).SingleOrDefault();
if(p1==null)
{
    Console.WriteLine("没有 id=3 的人");
}
else
{
    ctx.Persons.Remove(p1);
}
ctx.SaveChanges();
```

怎么批量删除，比如删除 Id>3 的？查询出来一个个 Remove。性能坑爹。如果操作不频繁或者数据量不大不用考虑性能，如果需要考虑性能就直接执行 sql 语句（后面讲）

3） 修改：先查询出来要修改的数据，然后修改，然后 SaveChanges()

```
MyDbContext ctx = new MyDbContext();
var ps = ctx.Persons.Where(p => p.Id > 3);
foreach(var p in ps)
{
    p.CreateDateTime = p.CreateDateTime.AddDays(3);
    p.Name = "haha";
}
ctx.SaveChanges();
```

性能问题？同上。

4） 查。因为 DbSet 实现了 IQueryable 接口，而 IQueryable 接口继承了 IEnumerable 接口，所以**可以使用所有的 linq、lambda 操作。给表增加一个 Age 字段，然后举例 orderby、groupby、where 操作、分页等。一样一样的。**

5） 查询 order by 的一个细节

EF 调用 Skip 之前必须调用 OrderBy：如下调用 var items = ctx.Persons.Skip(3).Take(5); 会报错“The method 'OrderBy' must be called before the method 'Skip'.)”，要改成：var items = ctx.Persons.OrderBy(p=>p.CreateDateTime).Skip(3).Take(5);

这也是一个好习惯，因为以前就发生过（写原始 sql）：分页查询的时候没有指定排序规则，以为默认是按照 Id 排序，其实有的时候不是，就造成数据混乱。写原始 SQL 的时候也要注意一定要指定排序规则。

十一、 EF 原理及 SQL 监控

EF 会自动把 Where()、OrderBy()、Select()等这些编译成“表达式树(Expression Tree)”，然后会把表达式树翻译成 SQL 语句去执行。（编译原理，AST）因此不是“把数据都取到内存中，然后使用集合的方法进行数据过滤”，因此性能不会低。但是如果这个操作不能被翻译成 SQL 语句，则或者报错，或者被放到内存中操作，性能就会非常低。

怎么查看真正执行的 SQL 是什么样呢？

DbContext 有一个 Database 属性，其中的 Log 属性，是 Action<String>委托类型，也就是可以指向一个 void A(string s)方法，其中的参数就是执行的 SQL 语句，每次 EF 执行 SQL 语句的时候都会执行 Log。因此就可以知道执行了什么 SQL。

EF 的查询是“延迟执行”的，只有遍历结果集的时候才执行 select 查询，ToList()内部也是遍历结果集形成 List。

查看 Update 操作，会发现只更新了修改的字段。

观察一下前面学学习时候执行的 SQL 是什么样的。Skip().Take()被翻译成了？Count()被翻译成了？

```
var result = ctx.Persons.Where(p => p.Name.StartsWith("rupeng"));看看翻译成了什么？
var result = ctx.Persons.Where(p => p.Name.Contains("com"));呢？
var result = ctx.Persons.Where(p => p.Name.Length>5); 呢？
var result = ctx.Persons.Where(p => p.CreateDateTime>DateTime.Now);  呢？
```

再看看（好牛）：

```
long[] ids = { 2,5,6};//不要写成int[]
var result = ctx.Persons.Where(p => ids.Contains(p.Id));
```

EF 中**还可以多次指定 where 来实现动态的复合检索：**

```
//必须写成 IQueryable<Person>，如果写成 IEnumerable 就会在内存中取后续数据
IQueryable<Person> items = ctx.Persons;//为什么把 IQueryable<Person>换成 var 会编译出错
items = items.Where(p=>p.Name=="rupeng");
items = items.Where(p=>p.Id>5);
```

查看一下生成的 SQL 语句。

(*)EF 是跨数据库的，如果迁移到 MYSQL 上，就会翻译成 MYSQL 的语法。要配置对应数据库的 Entity Framework Provider。

细节：

每次开始执行的__MigrationHistory 等这些 SQL 语句是什么？是 DBMigration 用的，也就是由 EF 帮我们建数据库，现在我们用不到，用下面的代码禁用：

```
Database.SetInitializer<XXXDbContext>(null);
```

XXXDbContext 就是项目 DbContext 的类名。一般建议放到 XXXDbContext 构造函数中。注意这里的 Database 是 System.Data.Entity 下的类，不是 DbContext 的 Database 属性。如果写到 DbContext 中，最好用上全名，防止出错。

十二、 执行原始 SQL

不要“手里有锤子，到处都是钉子”！

在一些特殊场合，需要执行原生 SQL。

执行非查询语句，调用 DbContext 的 Database 属性的 ExecuteSqlCommand 方法，可以通过占位符的方式传递参数：

```
ctx.Database.ExecuteSqlCommand("update T_Persons set Name={0},CreateDateTime=GetDate()", "rupeng.com");
```

占位符的方式不是字符串拼接，经过观察生成的 SQL 语句，发现仍然是参数化查询，因此不会有 SQL 注入漏洞。

执行查询：

```
var q1 = ctx.Database.SqlQuery<Item1>("select Name,Count(*) Count from T_Persons where Id>{0} and CreateDateTime<={1} group by Name",
    2, DateTime.Now); //返回值是 DbRawSqlQuery<T> 类型，也是实现了 IEnumerable 接口
foreach(var item in q1)
{
    Console.WriteLine(item.Name+":"+item.Count);
}
class Item1
{
    public string Name { get; set; }
    public int Count { get; set; }
}
```

类似于 ExecuteScalar 的操作比较麻烦：

```
int c = ctx.Database.SqlQuery<int>("select count(*) from T_Persons").SingleOrDefault();
```

十三、不是所有 lambda 写法都能被支持

下面想把 Id 转换为字符串比较一下是否为"3"（别管为什么）：

```
var result = ctx.Persons.Where(p => Convert.ToString(p.Id)=="3");
```

运行会报错（也许高版本支持了就不报错了），这是一个语法、逻辑上合法的写法，但是 EF 目前无法把他解析为一个 SQL 语句。

出现“System.NotSupportedException”异常一般就说明你的写法无法翻译成 SQL 语句。

想获取创建日期早于当前时间一小时以上的数据：

```
var result = ctx.Persons.Where(p => (DateTime.Now - p.CreateDateTime).TotalHours>1);
```

同样也可能会报错。

怎么解决？

尝试其他替代方案（没有依据，只能乱试）：

```
var result = ctx.Persons.Where(p => p.Id==3);
```

EF 中提供了一个 SQLServer 专用的类 SqlFunctions，对于 EF 不支持的函数提供了支持，比如：

```
var result = ctx.Persons.Where(p =>
                SqlFunctions.DateDiff("hour",p.CreateDateTime,DateTime.Now)>1);
```

十四、EF 对象的状态

1、简介

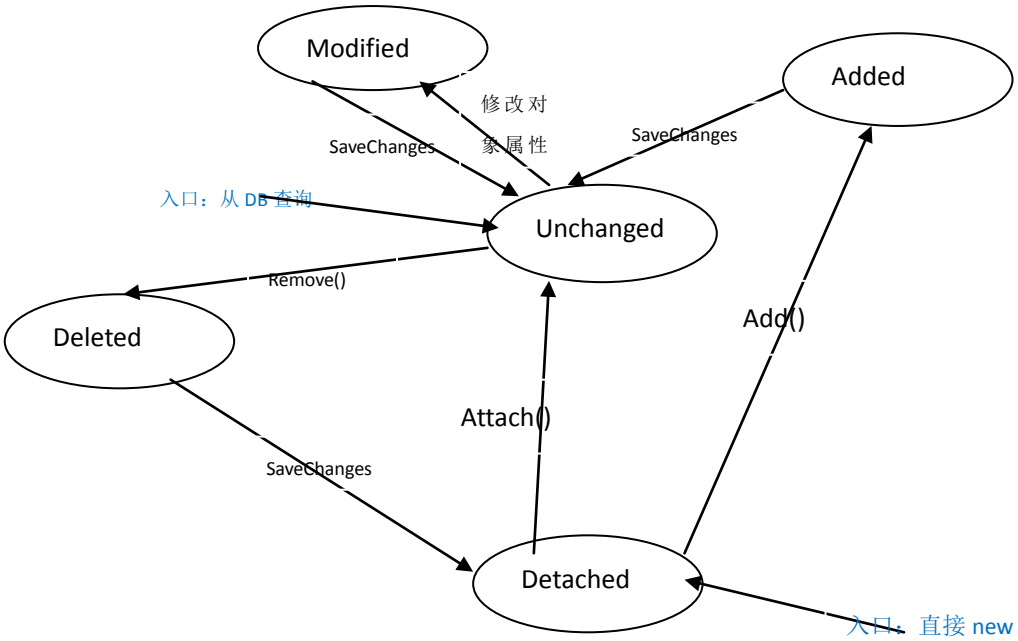
为什么查询出来的对象 Remove()、再 SaveChanges()就会把数据删除。而自己 new 一个 Person()对象，然后 Remove()不行？

为什么查询出来的对象修改属性值后、再 SaveChanges()就会把数据库中的数据修改。

因为 EF 会跟踪对象状态的改变。

EF 中中对象有五个状态：Detached（游离态，脱离态）、Unchanged（未改变）、Added（新增）、Deleted（删除）、Modified（被修改）。

3、 状态转换



Add()、Remove()修改对象的状态。所有状态之间几乎都可以通过：Entry(p).State=xxx 的方式 进行强制状态转换。

通过代码来演示一下。这个状态转换图没必要记住，了解即可。

状态改变都是依赖于 Id 的（Added 除外）

4、 应用（*）

当 SavaChanged()方法执行期间，会查看当前对象的 EntityState 的值，决定是去新增（Added）、修改（Modified）、删除（Deleted）或者什么也不做（UnChanged）。下面的做法不推荐，在旧版本中一些写法不被支持，到新版 EF 中可能也会不支持。

ObjectStateManager

1） 不先查询再修改再保存，而是直接更新部分字段的方法：

```
var p = new Person();
p.Id = 2;
ctx.Entry(p).State = System.Data.Entity.EntityState.Unchanged;
p.Name = "adfad";
ctx.SaveChanges();
```

也可以：

```
var p = new Person();
p.Id = 5;
p.Name = "yzk";
ctx.Persons.Attach(p);// 等 价 于 ctx.Entry(p).State = System.Data.Entity.EntityState.Unchanged;
ctx.Entry(p).Property(a => a.Name).IsModified = true;
ctx.SaveChanges();
```

2）不先查询再 Remove 再保存，而是直接根据 Id 删除的方法：

```
var p = new Person();
p.Id = 2;
ctx.Entry(p).State = System.Data.Entity.EntityState.Deleted;
ctx.SaveChanges();
```

注意下面的做法并不会删除所有 Name="rupeng.com" 的，因为更新、删除等都是根据 Id 进行的：

```
var p = new Person();
p.Name = "rupeng.com";
ctx.Entry(p).State = System.Data.Entity.EntityState.Deleted;
ctx.SaveChanges();
```

上面其实是在：delete * from t_persons where Id=0

5、 EF 优化的一个技巧

如果查询出来的对象**只是供显示使用，不会修改、删除**后保存，那么可以使用 AsNoTracking()来使得查询出来的对象是 Detached 状态，这样对对象的修改也还是 Detached 状态，EF 不再跟踪这个对象状态的改变，能够提升性能。

```
var p1 = ctx.Persons.Where(p => p.Name == "rupeng.com").FirstOrDefault();
Console.WriteLine(ctx.Entry(p1).State);
```

改成：

```
var p1 = ctx.Persons.AsNoTracking().Where(p => p.Name == "rupeng.com").FirstOrDefault();
Console.WriteLine(ctx.Entry(p1).State);
```

因为 AsNoTracking()是 DbQuery 类（DbSet 的父类）的方法，所以要先在 DbSet 后调用 AsNoTracking()。

十五、Fluent API 更多配置

基本 EF 配置只要配置实体类和表、字段的对应关系、表间关联关系即可。如果利用 EF 的高级配置，可以达到更多效果：如果数据错误（比如字段不能为空、字符串超长等），会在 EF 层就会报错，而不会被提交给数据库服务器再报错；如果使用自动生成数据库，也能帮助 EF 生成更完美的数据库表。

这些配置方法无论是 DataAnnotations、FluentAPI 都支持，下面讲 FluentAPI 的用法，DataAnnotations 感兴趣的自己查（http://blog.csdn.net/beglorious/article/details/39637475）。

尽量用约定，EF 配置越少越好。Simple is best 参考资料：http://www.cnblogs.com/nianming/archive/2012/11/07/2757997.html

1、 HasMaxLength 设定字段的最大长度

```
public PersonConfig()
{
    this.ToTable("T_Persons");
    this.Property(p => p.Name).HasMaxLength(50);//长度为 50
}
```

依赖于数据库的“字段长度、是否为空”等的约束是在数据提交到数据库服务器的时候才会检查；EF 的配置，则是由 EF 来检查的，如果检查出错，根本不会被提交给服务器。

如果插入一个 Person 对象，Name 属性的值非常长，保存的时候就会报 DbEntityValidationException 异常，这个异常的 Message 中看不到详细的报错消息，要看 EntityValidationErrors 属性的值。

```
var p = new Person();
p.Name = "非常长的字符串";
ctx.Persons.Add(p);
try
{
    ctx.SaveChanges();
}
catch(DbEntityValidationException ex)
{
    StringBuilder sb = new StringBuilder();
    foreach(var ve in ex.EntityValidationErrors.SelectMany(eve=>eve.ValidationErrors))
    {
        sb.AppendLine(ve.PropertyName+":"+ve.ErrorMessage);
    }
    Console.WriteLine(sb);
}
```

2、（有用）字段是否可空：

this.Property(p => p.Name).IsRequired() 属性不能为空；

this.Property(p => p.Name).IsOptional() 属性可以为空；（没用的鸡肋！）

EF 默认规则是“主键属性不允许为空，引用类型允许为空，可空的值类型 long?等允许为空，值类型不允许为空。”基于“尽量少配置”的原则：如果属性是值类型并且允许

为 null，就声明成 long?等，否则声明成 long 等；如果属性属性值是引用类型，只有不允许为空的时候设置 IsRequired()。

3、 其他一般不用设置的（了解即可）

a) 主键：this.HasKey(p => p.pId);

b) 某个字段不参与映射数据库：this.Ignore(p => p.Name1);

c) this.Property(p => p.Name).IsFixedLength(); 是否对应固定长度

d) this.Property(p => p.Name).IsUnicode(false) 对应的数据库类型是 varchar 类型，而不是 nvarchar

e) this.Property(p => p.Id).HasColumnName("Id1"); Id 列对应数据库中名字为 Id 的字段

f) this.Property(p => p.Id).HasDatabaseGeneratedOption(System.ComponentModel.DataAnnotations.Schema.DatabaseGeneratedOption.Identity) 指定字段是自动增长类型。

4、 流动起来

因为 ToTable()、Property()、IsRequired()等方法的还是配置对象本身，因此可以实现类似于 StringBuilder 的链式编程，这就是“Fluent”一词的含义；

因此下面的写法：

```
public PersonConfig()
{
    this.ToTable("T_Persons");
    this.HasKey(p => p.Id);
    this.Ignore(p => p.Name2);

    this.Property(p => p.Name).HasMaxLength(50);
    this.Property(p => p.Name).IsRequired();

    this.Property(p => p.CreateDateTime).HasColumnName("CreateDateTime");
    this.Property(p => p.Name).IsRequired();
}
```

可以简化成：

```
public PersonConfig()
{
    this.ToTable("T_Persons").HasKey(p => p.Id).Ignore(p => p.Name2);
    this.Property(p => p.Name).HasMaxLength(50).IsRequired();
    this.Property(p => p.CreateDateTime).HasColumnName("CreateDateTime").IsRequired();
}
```

后面用的时候都 Database.SetInitializer<XXXDbContext>(null);

十六、一对多关系映射

EF 最有魅力的地方在于对于多表间关系的映射，可以简化工作。

复习一下表间关系：

1) 一对多（多对一）：一个班级对应着多个学生，一个学生对着一个班级。一方是另外一方的唯一。在多端有一个指向一端的外键。举例：班级表：T_Classes(Id,Name) 学生表 T_Students(Id,Name,Age,ClassId)

2) 多对多：一个老师对应多个学生，一个学生对于多个老师。任何一方都不是对方的唯一。需要一个中间关系表。具体：学生表 T_Students(Id,Name,Age,ClassId)，老师表 T_Teachers(Id,Name,PhoneNum)，关系表 T_StudentsTeachers(Id,StudentId,TeacherId)

和关系映射相关的方法：

1） 基本套路 this.Has****(p=>p.A).With***() 当前这个表和 A 属性的表的关系是 Has 定义，With 定义的是 A 对应的表和这个表的关系。Optional/Required/Many

2） HasOptional() 有一个可选的（可以为空的）

3） HasRequired() 有一个必须的（不能为空的）

4） HasMany() 有很多的

5） WithOptional() 可选的

6） WithRequired() 必须的

7） WithMany() 很多的

举例：

在 AAA 实体中配置 this. HasRequired(p=>p.BBB).WithMany();是什么意思？

在 AAA 实体中配置 this. HasRequired(p=>p.BBB). WithRequired ();是什么意思？

十七、 配置一对多关系：

1、 先按照正常的单表配置把 Student、Class 配置起来，T_Students 的 ClassId 字段就对应 Student 类的 ClassId 属性。WithOptional()

```
using (MyDbContext ctx = new MyDbContext())
{
    Class c1 = new Class { Name = "三年二班" };
    ctx.Classes.Add(c1);
    ctx.SaveChanges();
    Student s1 = new Student { Age = 11, Name = "张三", ClassId = c1.Id };
    Student s2 = new Student { Name = "李四", ClassId = c1.Id };
    ctx.Students.Add(s1);
    ctx.Students.Add(s2);
    ctx.SaveChanges();
}
```

2、 给 Student 类增加一个 Class 类型、名字为 Class（不一定非叫这个，但是习惯是：外键名去掉 Id）的属性，要声明成 virtual（后面讲原因）。

3、 然后就可以实现各种对象间操作了：

a) Console.WriteLine(ctx.Students.First().Class.Name)

b) 然后数据插入也变得简单了，不用再考虑“先保存 Class，生成 Id，再保存 Student”了。这样就是纯正的“面向对象模型”，ClassId 属性可以删掉。

```
Class c1 = new Class { Name = "五年三班" };
ctx.Classes.Add(c1);
Student s1 = new Student { Age = 11, Name = "皮皮虾"};
Student s2 = new Student { Name = "巴斯"};
```

```
s1.Class = c1;
s2.Class = c1;
ctx.Students.Add(s1);
ctx.Students.Add(s2);
ctx.Classes.Add(c1);
ctx.SaveChanges();
```

4、 如果 ClassId 字段可空怎么办？直接把 ClassId 属性设置为 long?

5、 还可以在 Class 中配置一个 public virtual ICollection<Student> Students { get; set; } = new List<Student>(); 属性。最好给这个属性初始化一个对象。注意是 virtual。这样就可以获得所有指向了当前对象的 Stuent 集合，也就是这个班级的所有学生。我个人不喜欢这个属性，业界的大佬也是建议“尽量不要设计双向关系”，因为可以通过 Class clz = ctx.Classes.First(); var students = ctx.Students.Where(s => s.ClassId == clz.Id);来查询获取到，思路更清晰。

不过有了这样的集合属性之后一个方便的地方：

```
Class c1 = new Class { Name = "五年三班" };
ctx.Classes.Add(c1);
Student s1 = new Student { Age = 11, Name = "皮皮虾" };
Student s2 = new Student { Name = "巴斯" };

c1.Students.Add(s1);//注意要在Students属性声明的时候= new List<Student>();或者在之前赋值
c1.Students.Add(s2);

ctx.Classes.Add(c1);
ctx.SaveChanges();
```

EF会自动追踪对象的关联关系，给那些有关联的对象也自动进行处理。

在进行数据遍历的时候可能会报错“已有打开的与此 Command 相关联的 DataReader，必须首先将它关闭。”

```
foreach(var s in ctx.Students)
{
    Console.WriteLine(s.Name);
    Console.WriteLine(s.Class.Name);
}
```

以后会讲解决方案。

一对多深入：

1、 默认约定配置即可，如果非要配置，可以在 StudentConfig 中如下配置：this.HasRequired(s => s.Class).WithMany().HasForeignKey(s => s.ClassId);；表示“我需要(Require)一个 Class，Class 有很多(Many)的 Student；ClassId 是这样一个外键”。如果 ClassId 可空，那么就要写成：this.HasOptional (s => s.Class).WithMany().HasForeignKey(s => s.ClassId);

2、

如果这样Class clz = ctx.Classes.First();

foreach (Student s in clz.Students)访问，也就是从一端发起 对多端的方法，那么就会报错“找不到 Class_Id 字段”

需要在 ClassConfig 中再反向配置一遍

HasMany(e => e.Students).WithRequired().HasForeignKey(e=>e.ClassId);

因为如果在 Class 中引入 Students 属性，还要再在 ClassConfig 再配置一遍反向关系，很麻烦。因此再次验证“不要设计双向关系”。

3、 如果一张表中有两个指向另外一个表的外键怎么办？比如学生有“正常班级 Class”（不能空）和“小灶班级 XZClass”（可以空）两个班。

在 StudentConfig 中：

this.HasRequired(s => s.Class).WithMany().HasForeignKey(s => s.ClassId);

this. HasOptional (s => s.XZClass).WithMany().HasForeignKey(s => s.XZClassId);

十八、多对多关系配置

老师和学生：

```
class Student
{
    public long Id { set; get; }
    public string Name { get; set; }
    public virtual ICollection<Teacher> Teachers { get; set; }=new List<Teacher>();
}

class Teacher
{
    public long Id { set; get; }
    public string Name { get; set; }

    public virtual ICollection<Student> Students { get; set; }=new List< Student >();
}
class StudentConfig : EntityTypeConfiguration<Student>
{
    public StudentConfig()
    {
        ToTable("T_Students");
    }
}
class TeacherConfig : EntityTypeConfiguration<Teacher>
{
    public TeacherConfig()
    {
        ToTable("T_Teachers");
        this.HasMany(e => e.Students).WithMany(e => e.Teachers)//易错，容易丢了
```

WithMany 的参数

```
                .Map(m => m.ToTable("T_TeacherStudentRelations").MapLeftKey("TeacherId").MapRightKey("StudentId"));
        }
    }
```

关系配置到任何一方都可以

这样不用中间表建实体（也可以为中间表建立一个实体，其实思路更清晰），就可以完成多对多映射。当然如果中间关系表还想有其他字段，则要必须为中间表建立实体类。。

测试：

```
Teacher t1 = new Teacher();
t1.Name = "张老师";
t1.Students = new List<Student>();
Teacher t2 = new Teacher();
t2.Name = "王老师";
t2.Students = new List<Student>();

Student s1 = new Student();
s1.Name = "tom";
s1.Teachers = new List<Teacher>();

Student s2 = new Student();
s2.Name = "jerry";
s2.Teachers = new List<Teacher>();

t1.Students.Add(s1);
```

附录：

1、 关于 WithMany()的参数

1）在一对多关系中，如果只配置多端关系并且没有给 WithMany()指定参数的话，在进行反向关系操作的时候就会报错。要么在一端也配置一次，最好的方法就是还是只配置多端，只不过给 WithMany()指定参数：

```
class StudentConfig:EntityTypeConfiguration<Student>
{
    public StudentConfig()
    {
        ToTable("T_Students");
        this.HasRequired(e => e.Class).WithMany(e=>e.Students)
        .HasForeignKey(e=>e.ClassId);
    }
}
```

当然还是不建议用反向的集合属性，如果 Class 没有 Students 这个集合属性的话，就不用（也不能）WithMany 的参数了。

2） 关于多对多关系配置的 WithMany()问题

上次讲课配置多对多的关系没有给 WithMany 设定参数，这样反向操作的时候就会出错，应该改成：this.HasMany(e => e.Students).WithMany(e=>e.Teachers)

总结：一对多的中不建议配置一端的集合属性，因此配置的时候不用给 WithMany()参数，如果配置了集合属性，则必须给 WithMany 参数；多对多关系必须要给 WithMany()参数。

总结一对多、多对多的“最佳实践”

一对多最佳方法（不配置一端的集合属性）：

1、 多端

```
public class Student
{
    public long Id { get; set; }
    public string Name { get; set; }
    public long ClassId { get; set; }
    public virtual Class Class { get; set; }
}
```

2、 一端

```
public class Class
{
    public long Id { get; set; }
    public string Name { get; set; }
}
```

3、 在多端的模型配置(StudentConfig)中：

```
this.HasRequired(e => e.Class).WithMany() .HasForeignKey(e=>e.ClassId);
```

一对多的配置（在一端配置一个集合属性，极端不推荐）

1、多端

```
public class Student
{
    public long Id { get; set; }
    public string Name { get; set; }
    public long ClassId { get; set; }
    public virtual Class Class { get; set; }
}
```

2、 一端

```
public class Class
{
    public long Id { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Student> Students { get; set; } = new List<Student>();
}
```

3、 多端的配置(StudentConfig)中

```
this.HasRequired(e => e.Class).WithMany(e=>e.Students)//WithMany()的参数不能丢
                .HasForeignKey(e=>e.ClassId);
```

多对多最佳配置

1、 两端模型

```
public class Student
{
    public long Id { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Teacher> Teachers { get; set; } = new List<Teacher>();
}
public class Teacher
{
public long Id { get; set; }
public string Name { get; set; }
public virtual ICollection<Student> Students { get; set; } = new List<Student>();
}
```

2、 在其中一端配置(StudentConfig)

```
this.HasMany(e => e.Teachers).WithMany(e=>e.Students).Map(m =>//不要忘了WithMany的参数
m.ToTable("T_StudentTeachers").MapLeftKey("StudentId").MapRightKey("TeacherId"));
```

4、 多对多中 移除关系：t.Students.Remove(t.Students.First()); 添加关系

5、 (*)多对多中还可以为中间表建立一个实体方式映射。当然如果中间关系表还想有其他字段，则要必须为中间表建立实体类（中间表和两个表之间就是两个一对多的关系了）。

6、 数据库创建策略（*）：

如果数据库创建好了再修改模型或者配置，运行就会报错，那么就要手动删除数据库或者：

```
Database.SetInitializer(new DropCreateDatabaseIfModelChanges<XXXContext>());
```

如果报错“数据库正在使用”，可能是因为开着 Mangement Studio，先关掉就行了。

知道就行了，只适合学习时候使用。

CodeFirst Migration 参考(*)： http://www.cnblogs.com/libingql/p/3330880.html 太复杂，不符合 Simple is Best 的原则，这是为什么有一些开发者不用 EF，而使用 Dapper 的原因。

做项目的时候建议初期先把主要的类使用 EF 自动生成表，然后干掉 Migration 表，然后就 Database.SetInitializer<XXXContext>(null);以后对数据库表的修改都手动完成，也就是手动改实体类、手动改数据库表。

十九、延迟加载（LazyLoad）

如果 public virtual Class Class { get; set; }（实体之间的关联属性又叫做“导航属性(Navigation Property)”）把 virtual 去掉，那么下面的代码就会报空引用异常

```
var s = ctx.Students.First();
Console.WriteLine(s.Class.Name);
```

联想为什么？凭什么！！！

改成 virtual 观察 SQL 的执行。执行了两个 SQL，先查询 T_Students，再到 T_Classes 中查到对应的行。

这叫“延迟加载”（LazyLoad），只有用到关联的对象的数据，才会再去执行 select 查询。注意延迟加载只在关联对象属性上，普通属性没这个东西。

注意：启用延迟加载需要配置如下两个属性（默认就是 true，因此不需要去配置，只要别手贱设置为 false 即可）

```
context.Configuration.ProxyCreationEnabled = true;
context.Configuration.LazyLoadingEnabled = true;
```

分析延迟加载的原理：打印一下拿到的对象的 GetType()，再打印一下 GetType().BaseType;我们发现拿到的对象其实是 Student 子类的对象。（如果和我这里结果不一致的话，说明：类不是 public，没有关联的 virtual 属性）

因此 EF 其实是动态生成了实体类对象的子类，然后 override 了这些 virtual 属性，类似于这样的实现：

```
public class StudentProxy:Student
{
    private Class clz;
    public override Class    Class
    {
        get
        {
            if(this.clz==null)
            {
                this.clz=    ....//这里是从数据库中加载 Class 对象的代码
            }
            return this.clz;
        }
    }
}
```

再次强调：如果要使用延迟加载，类必须是 public，关联属性必须是 virtual。

延迟加载（LazyLoad）的优点：用到的时候才加载，没用到的时候才加载，因此避免了一次性加载所有数据，提高了加载的速度。缺点：如果不用延迟加载，就可以一次数据库查询就可以把所有数据都取出来（使用 join 实现），用了延迟加载就要多次执行数据库操作，提高了数据库服务器的压力。

因此：如果关联的属性几乎都要读取到，那么就不要用延迟加载；如果关联的属性只有较小的概率（比如年龄大于 7 岁的学生显示班级名字，否则就不显示）则可以启用延迟加载。这个概率到底是多少是没有一个固定的值，和数据、业务、技术架构的特点都有关系，这是需要经验和直觉，也需要测试和平衡的。

注意：启用延迟加载的时候拿到的对象是动态生成类的对象，是不可序列化的，因此不能直接放到进程外 Session、Redis 等中，解决方法？

二十、不延迟加载，怎么样一次性加载？

用 EF 永远都要把导航属性设置为 virtual。又想方便（必须是 virtual）又想效率高！

使用 Include()方法：var s = ctx.Students.Include("Class").First();//

观察生成的 SQL 语句，会发现只执行一个使用 join 的 SQL 就把所有用到的数据取出来了。当然拿到的对象还是 Student 的子类对象，但是不会延迟加载。（不用研究“怎么让他返回 Student 对象”）

Include("Class")的意思是直接加载 Student 的 Class 属性的数据。注意只有关联的对象属性才可以用 Include，普通字段不可以

直接写"Class"可能拼写错误，如果用 C#6.0，可以使用 nameof 语法解决问这个问题：

var s = ctx.Students.Include(nameof(Student.Class)).First();

也可以 using System.Data.Entity;然后 var s = ctx.Students.Include(e=>e.Class).First(); 推荐这种做法。

如果有多个属性需要一次性加载，也可以写多个 Include：

var s = ctx.Students.Include(e=>e.Class) .Include(e=>e.Teacher).First();

如果 Class 对象还有一个 School 属性，也想把 School 对象的属性也加载，就要：

var s = ctx.Students.Include("Class").Include("Class. School").First(); 或者更好的

```
var s = ctx.Students.Include(nameof(Student.Class))
        .Include(nameof(Student.Class)+"."+nameof(Class.School)).First();
```

二十一、延迟加载的一些坑

1、 DbContext 销毁后就不能再延迟加载了，因为数据库连接已经断开

下面的代码最后一行会报错：

```
Student s;
using (MyDbContext ctx = new MyDbContext())
{
    s = ctx.Students.First();
}
Console.WriteLine(s.Class.Name);
```

两种解决方法：

1）用 Include，不延迟加载（推荐）

```
Student s;
using (MyDbContext ctx = new MyDbContext())
{
    s = ctx.Students.Include(t=>t.Class).First();
}
Console.WriteLine(s.Class.Name);
```

3） 关闭前把要用到的数据取出来

```
Class c;
using (MyDbContext ctx = new MyDbContext())
{
    Student s = ctx.Students.Include(t=>t.Class).First();
```

```
    c = s.Class;
}
Console.WriteLine(c.Name);
```

2、两个取数据一起使用

下面的程序会报错：

已有打开的与此 Command 相关联的 DataReader，必须首先将它关闭。

```
foreach(var s in ctx.Students)
{
    Console.WriteLine(s.Name);
    Console.WriteLine(s.Class.Name);
}
```

因为 EF 的查询是“延迟执行”的，只有遍历结果集的时候才执行 select 查询，而由于延迟加载的存在到 s.Class.Name 也会再次执行查询。ADO.Net 中默认是不能同时遍历两个 DataReader。因此就报错。

解决方法有如下

三种解决方式：

1）允许多个 DataReader 一起执行：在连接字符串上加上 MultipleActiveResultSets=true,但只适用于 SQL 2005 以后的版本。其他数据库不支持。

2）执行一下 ToList()，因为 ToList()就遍历然后生成 List：

```
foreach(var s in ctx.Students.ToList())
{
    Console.WriteLine(s.Name);
    Console.WriteLine(s.Class.Name);
}
```

3) 推荐做法：用 Include 预先加载：

```
foreach(var s in ctx.Students.Include(e=>e.Class))
{
    Console.WriteLine(s.Name);
    Console.WriteLine(s.Class.Name);
}
```

二十二、实体类的继承

所有实体类都会有一些公共属性，可以把这些属性定义到一个父类中。比如：

```
public abstract class BaseEntity
{
    public long Id { get; set; }   //主键
    public bool IsDeleted { get; set; } = false; //软删除
    public DateTime CreateDateTime { get; set; } = DateTime.Now;//创建时间
    public DateTime DeleteDateTime { get; set; }   //删除时间
}
```

使用公共父类的好处不仅是写实体类简单了，而且可以提供一个公共的 Entity 操作类：

```
class BaseDAO<T> where T:BaseEntity
{
    private MyDbContext ctx;//不自己维护 MyDbContext 而是由调用者传递，因为调用者可以要执行很多操作，由调用者决定什么时候销毁。
    public BaseDAO (MyDbContext ctx)
    {
        this.ctx = ctx;
    }
    public IQueryable<T> GetAll()//获得所有数据（不要软删除的）
    {
        return ctx.Set<T>().Where(t=>t.IsDeleted==false);//这样自动处理软删除，避免了忘了过滤软删除的数据
    }

    public IQueryable<T> GetAll(int start,int count) //分页获得所有数据（不要软删除的）
    {
        return GetAll().Skip(start).Take(count);
    }

    public long GetTotalCount()//获取所有数据的条数
    {
        return GetAll().LongCount();
    }

    public T GetById(long id)//根据 id 获取
    {
        return GetAll().Where(t=>t.Id==id).SingleOrDefault();
    }

    public void MarkDeleted(long id)//软删除
    {
        T en = GetById(id);
        if(en!=null)
        {
        en.IsDeleted = true;
        en.DeleteDateTime = DateTime.Now;
        ctx.SaveChanges();
        }
    }
}
```

DAL 同层内返回 IQueryable 比 IEnumerable 更好

下面的代码会报错：

```
using (MyDbContext ctx = new MyDbContext())
{
    BaseDAO<Student> dao = new BaseDAO<Student>(ctx);
    foreach(var s in dao.GetAll())
    {
        Console.WriteLine(s.Name);
        Console.WriteLine(s.Class.Name);
    }
}
```

原因是什么？

怎么 Include？需要 using System.Data.Entity;

```
using (MyDbContext ctx = new MyDbContext())
{
    BaseDAO<Student> dao = new BaseDAO<Student>(ctx);
    foreach(var s in dao.GetAll().Include(t=>t.Class))
    {
        Console.WriteLine(s.Name);
        Console.WriteLine(s.Class.Name);
    }
}
```

有两个版本的 Include、AsNoTracking：

1) DbQuery 中的：DbQuery<TResult> AsNoTracking()、DbQuery<TResult> Include(string path)
2) QueryableExtensions 中的扩展方法：AsNoTracking<T>(this IQueryable<T> source)、Include<T>(this IQueryable<T> source, string path)、Include<T, TProperty>(this IQueryable<T> source, Expression<Func<T, TProperty>> path)

DbSet 继承自 DbQuery；Where()、Order、Skip()等这些方法返回的是 IQueryable 接口。因此如果在 IQueryable 接口类型的对象上调用 Include、AsNoTracking 就要 using System.Data.Entity

二十三、其他

还有其他优秀的 ORM 框架：NHibernate、Dapper、PetaPoco、IBatis.Net；

ASP.Net MVC+Entity Framework 的架构

一、了解一些不推荐的做法

有的项目里是直接把 EF 代码写到 ASP.Net MVC 的 Controller 中，这样做其实不符合分层的原则。ASP.Net MVC 是 UI 层的框架，EF 是数据访问的逻辑。

如果就要这么做怎么做的呢？

如果在 Controller 中 using DbContext，把查询的结果的对象放到 cshtml 中显示，那么一旦在 cshtml 中访问关联属性，那么就会报错。因为关联属性可以一直关联下去，很诱惑人，

include 也来不及。

如果不 using 也没问题，因为会自动回收。但是这是打开了“潘多拉魔盒”，甚至可以在 UI 层更新数据。相当于把数据逻辑写到了 UI 层。

有的三层架构中用实体类做 Model，这样也是不好的，因为实体类属于 DAL 层的逻辑。

没有最好的架构，只有最合适的架构！
架构不是设计出来的，而是演化出来的！

二、EO、DTO、ViewModel

EO（Entity Object，实体对象）就是 EF 中的实体类，对 EO 的操作会对数据库产生影响。EO 不应该传递到其他层。

DTO（Data Transfer Object，数据传输对象），用于在各个层之间传递数据的普通类。DTO 有哪些属性取决于其他层要什么数据。DTO 一般是“扁平类”，也就是没有关联属性，都是普通类型属性。一些复杂项目中，数据访问层（DAL）和业务逻辑层（BLL）直接传递用一个 DTO 类，UI 层和 BLL 层之间用一个新的 DTO 类。简单的项目共用同一个 DTO。DTO 类似于三层架构中的 Model。

EO 相当于 DataTable，不能传输到 DAL 之外；
DTO 就是三层 Model，在各个层中间传输数据用的

ViewModel（视图模型），用来组合来自其他层的数据显示到 UI 层。简单的数据可能可以直接把 DTO 交给界面显示，一些复杂的数据可以要从新转换为 ViewModel 对象。

三、多层架构

搭建一个 ASP.Net MVC 三层架构项目：DAL、BLL、DTO、UI(asp.net mvc)。UI、DAL、BLL 都引用 DTO；BLL 引用 DAL；EF 中的所有代码都定义到 DAL 中，BLL 中只访问 DAL、BLL 中不要引用 DAL 中的 EF 相关的类、不要在 BLL 中执行 Include 等操作、所有数据的准备工作都在 DAL 中完成。

四、架构退化

因为很多项目中的逻辑都写到 DAL 中，因为主要就是对数据库的操作，BLL 就变成了对 DAL 的单纯的转发，没有必要的麻烦。因此对于这种情况，可以把 UI+BLL+DAL 的架构退化成 UI+Service 的架构。可以理解把 BLL+DAL 都写到一个 Service 层中。

没有“正确的架构”、“错误的架构”，只有最合适的架构。

CRUD 例子，带关联关系。班级管理、学生管理、民族。

注意：.Net 中配置文件都是加载 UI 项目(ASP.net MVC)的，而不是加载 DAL 中的配置文件，因此 EF 的配置、连接字符串应该挪到 UI 项目中。

“合适的架构”：能够满足当前项目的要求，并且适度的考虑以后项目的发展，不要想得“太远”，不要“过度架构”；让新手能够非常快的上手（金蝶、赞同）。

UI 项目虽然不直接访问 EF 中的类，但是仍然需要在 UI 项目的 App.config（Web.config）中对 EF 做配置，也要在项目中通过 Nuget 安装 EF，然后并且要把连接字符串也配置到 UI 项目的 App.config（Web.config）中。