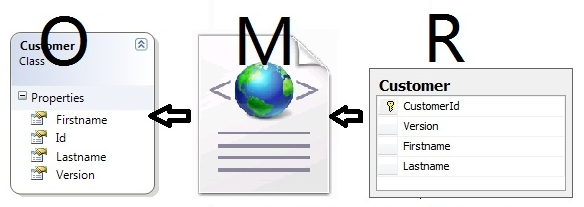
# EF

## 什么是ORM

广义上，ORM（Object Relational Mapping）即对象关系映射，指面向对象的对象模型和关系型数据库的数据结构之间的相互转换

狭义上，ORM指基于关系型数据库的数据存储，实现一个虚拟的面向对象的数据访问接口。理想情况下，通过这种模型持久化对象不需要了解任何关系型数据库存储数据的实现细节

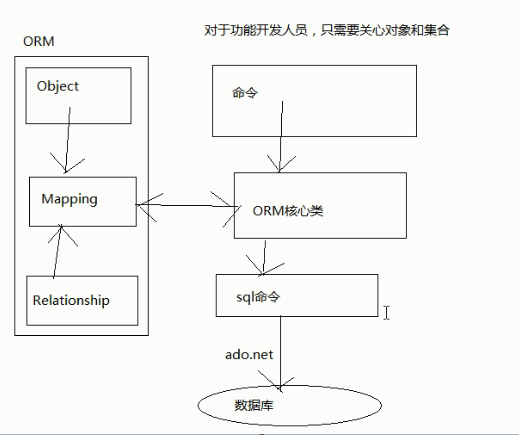


Object:对象部分

Realation：关系型数据库部分

Mapping：映射部分。数据库中表中每一列对应类的每一个属性。

对于功能开发人员，只需要关心对象和集合。开发人员编写的命令交给orm核心类，orm核心类经过mapping后将命令转换为sql命令，进行数据库访问。



Orm效率快，但是速度慢，如果访问量高，不适合用。

## EF

实体框架Entity Framework，简称EF

Ado.Net：微软提供访问数据库的一组API，回顾常用类及功能

EF是微软推出的基于Ado.Net的数据库访问技术，是一套ORM框架

底层访问数据库的实质依然是ado.net

是一套orm框架，即框架中包含了实例模型、数据模型、映射关系三部分

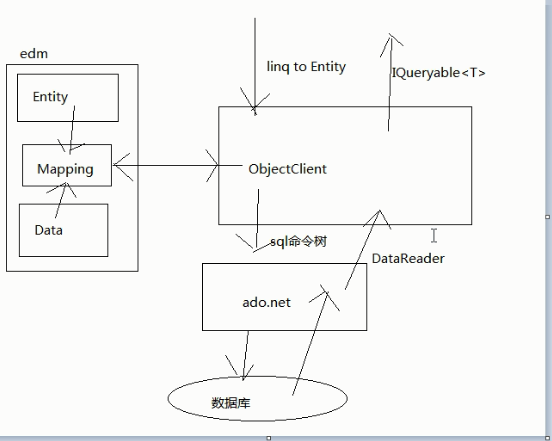
EF中的ORM框架叫做EDM。

Entity：对象部分

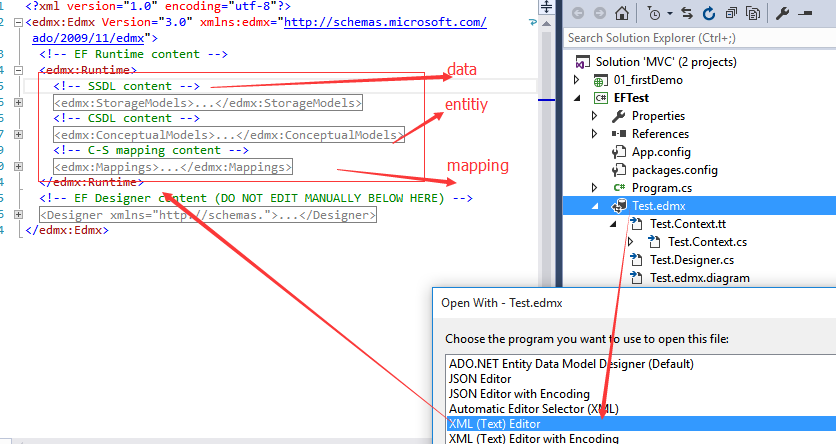
Data：数据库部分

Mapping：映射部分

开发人员通过ling to Entity命令创建命令树，交给ObjectClient核心类，核心类通过EDM将命令转换为sql命令树，交给ado.net去访问数据库。

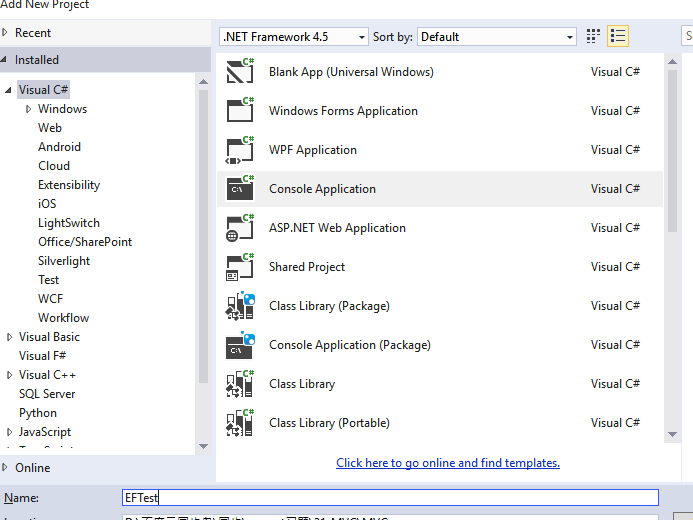


Edm可以用xml去分析

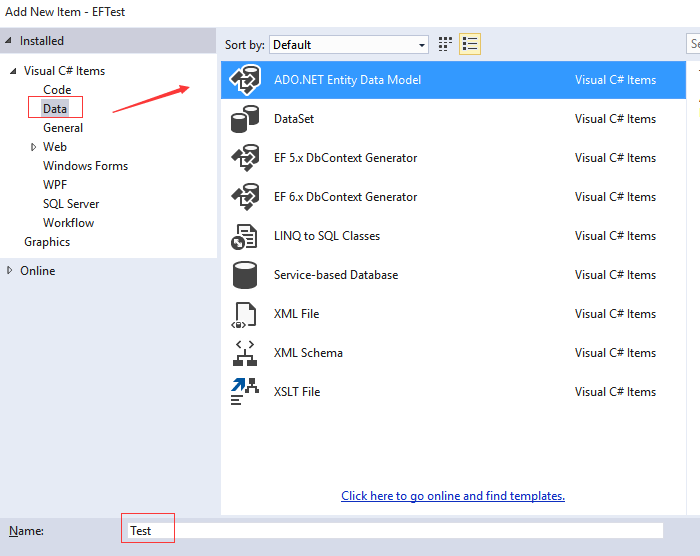


## 案例：使用流程

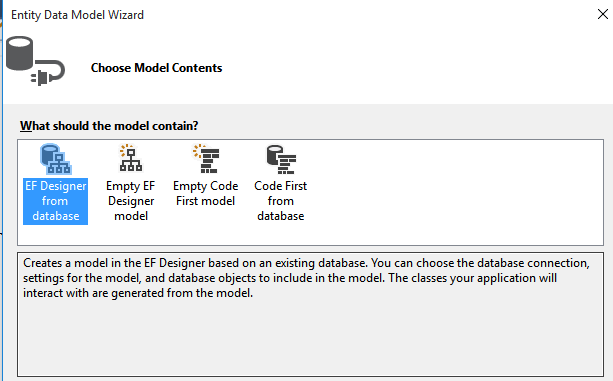
### 创建控制台项目



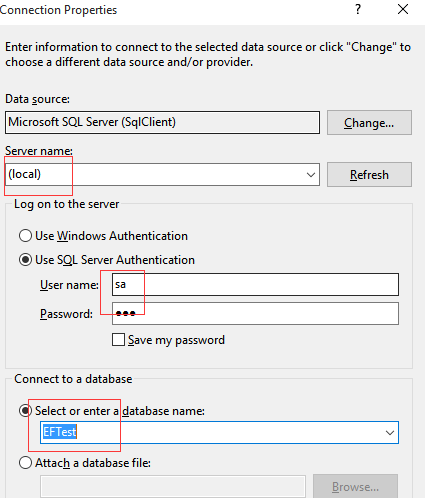
### 添加实体数据模型（EF）



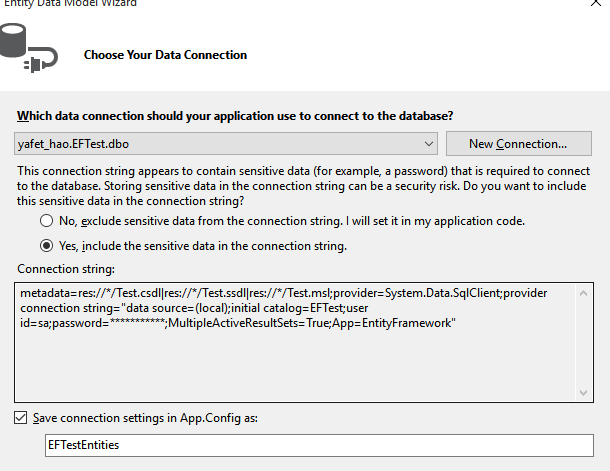
选择从数据库创建



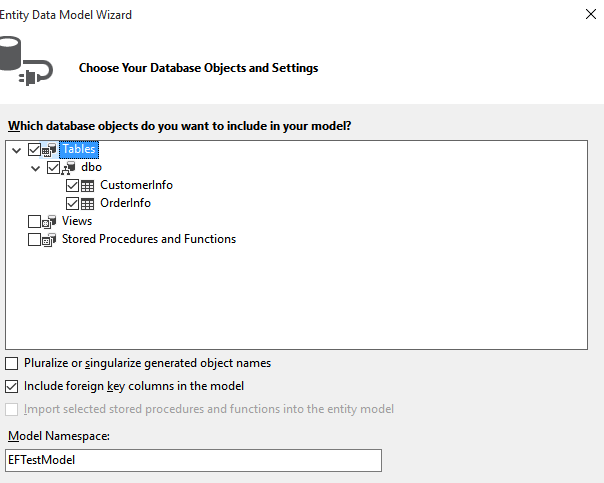
设置连接的属性



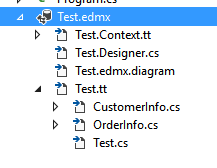
选择要链接的数据库



选择要保护到EF中的表



最后结果

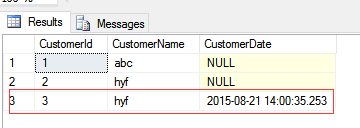


### 集合版CURD

EF中可以像使用集合一样实现CURD。

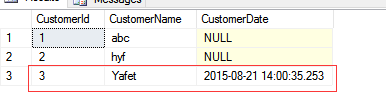
#### 增加

|  |
| --- |
| static void Main(string[] args)  {  //创建EF上下文对象  EFTestEntities context = new EFTestEntities();  //添加。参数为TEntity实体类型。用对象初始化器实现。  context.CustomerInfo.Add(new CustomerInfo() {  CustomerName = "hyf",  CustomerDate = DateTime.Now    });  //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |



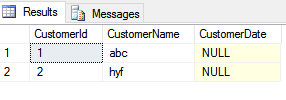
#### 修改

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  //根据id查询数据并赋值给实体  CustomerInfo customerInfo = context.CustomerInfo.Find(3);  customerInfo.CustomerName = "Yafet";    //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |



#### 删除

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  //根据id查询数据并赋值给实体  CustomerInfo customerInfo = context.CustomerInfo.Find(3);  //传入对象进行删除  context.CustomerInfo.Remove(customerInfo);    //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |



#### 查询

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  StringBuilder sb = new StringBuilder();  //遍历  foreach (var item in context.CustomerInfo)  {  sb.AppendFormat("编号为：{0},名称为：{1},日期为：{2}\r\n", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine(sb.ToString());  Console.Read();  } |

### 状态版CURD

##### 增加

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  CustomerInfo cs = new CustomerInfo() {  CustomerName="hao"  };  context.Entry(cs).State = System.Data.Entity.EntityState.Added;  //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |

##### 修改

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  CustomerInfo cs = new CustomerInfo() {  CustomerId = 5,  CustomerName="Yafet"  };  context.Entry(cs).State = System.Data.Entity.EntityState.Modified;  //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |

需要说明的是：没有设置的字段CustomerDate原有的数据会变会空。

##### 修改单列

设置EntityState为Modified，会修改该列的所有数据，没有赋值则设置为空。相应实现修改单列需要用下面的方式

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  CustomerInfo cs = new CustomerInfo() {  CustomerId = 7  };  //把cs添加的上下文中  context.CustomerInfo.Attach(cs);  context.Entry(cs).Property("CustomerName").CurrentValue = "hello";  context.Entry(cs).Property("CustomerName").IsModified = true;  //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |

### 状态跟踪

集合版与状态版实质都是状态追踪。在上下文中有个ObjectStateManager类，来进行状态跟踪。它是通过设置DbSet集合的状态来进行跟踪的。只有在DbSet集合中的对象才能实现状态跟踪。集合版是自动把对象添加到了DbSet中。状态版设置单列时需要手动添加到DbSet。

##### 删除

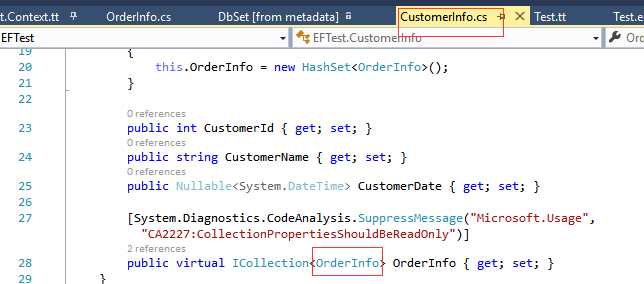
|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  CustomerInfo cs = new CustomerInfo() {  CustomerId = 5,  };  context.Entry(cs).State = System.Data.Entity.EntityState.Deleted;  //把下午文中所有的修改保存到数据库  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |

一把情况下，增加、删除使用集合版、修改使用状态版

### 导航属性实现关联字段的增加

导航属性为在两个实体类型间导航关联提供了一种方式。针对对象参与到其中的每个关系，各对象均可以具有导航属性。使用导航属性，可以在两个方向上导航和管理关系。

在customerInfo类中有一个OrderInfo哈希集合类型的属性。这个属性叫做导航属性。把OrderInfo的实例添加到这个集合中，既可以实现关联。



例如，在往订单OrderInfo中插入数据时，其CustomerId与CustomerInfo表相关联。如何实现数据插入时CustomerId的值设置呢？

第一种方式：通过外键

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  CustomerInfo cs1 = context.CustomerInfo.Find(7);  OrderInfo oi = new OrderInfo()  {  OrderTitle = "apple",  CustomerId = cs1.CustomerId//建立关系的第一种方式，通过外键  };  context.OrderInfo.Add(oi);  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |

第二种方式：通过导航属性

|  |
| --- |
| static void Main(string[] args)  {  //创建EF对象  EFTestEntities context = new EFTestEntities();  CustomerInfo cs1 = context.CustomerInfo.Find(7);  OrderInfo oi = new OrderInfo()  {  OrderTitle = "orange",  };  cs1.OrderInfo.Add(oi);//导航属性  context.OrderInfo.Add(oi);  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |

## 多态与上下文对象

正常方式：

|  |
| --- |
| EFTestEntities context1 = new EFTestEntities();  StringBuilder sb = new StringBuilder();  foreach (var item in context1.CustomerInfo)  {  sb.AppendFormat("编号：{0},姓名：{1},日期：{2}\r\n", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  Console.WriteLine(sb.ToString());  Console.Read(); |

多态方式：

|  |
| --- |
| static void Main(string[] args)  {  //EFTestEntities : DbContext 创建的上下文对象都继承自DbContext。  DbContext context1 = new EFTestEntities();  StringBuilder sb = new StringBuilder();  //set泛型方法返回Dbset集合。与上例中context1.CustomerInfo功能相同  foreach (var item in context1.Set<CustomerInfo>())  {  sb.AppendFormat("编号：{0},姓名：{1},日期：{2}\r\n", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  Console.WriteLine(sb.ToString());  Console.Read();  } |

推荐第二种，依赖倒置跟灵活。

# Ling查询

## 基本查询

|  |
| --- |
| static void Main(string[] args)  {  //EFTestEntities : DbContext 创建的上下文对象都继承自DbContext。  DbContext context1 = new EFTestEntities();  StringBuilder sb = new StringBuilder();  //set泛型方法返回Dbset集合。与上例中context1.CustomerInfo功能相同  foreach (var item in context1.Set<CustomerInfo>())  {  sb.AppendFormat("编号：{0},姓名：{1},日期：{2}\r\n", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  Console.WriteLine(sb.ToString());  Console.Read();  } |

## 条件查询

大于(>)、小于(<)、相等条件（==）

|  |
| --- |
| DbContext context1 = new EFTestEntities();  //ling条件查询  var result = from c in context1.Set<CustomerInfo>() where c.CustomerId < 5 select c;  foreach (var item in result)  {  Console.WriteLine("编号：{0},姓名：{1},日期：{2}", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  Console.Read(); |

首字母匹配

|  |
| --- |
| DbContext context1 = new EFTestEntities();  //ling条件查询  var result = from c in context1.Set<CustomerInfo>() where c.CustomerName.StartsWith("h") select c;  foreach (var item in result)  {  Console.WriteLine("编号：{0},姓名：{1},日期：{2}", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  Console.Read(); |

包含某字

|  |
| --- |
| DbContext context1 = new EFTestEntities();  //ling条件查询  var result = from c in context1.Set<CustomerInfo>() where c.CustomerName.Contains("a") select c;  foreach (var item in result)  {  Console.WriteLine("编号：{0},姓名：{1},日期：{2}", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  Console.Read(); |

多条件查询

|  |
| --- |
| DbContext context1 = new EFTestEntities();  //ling条件查询  var result = from c in context1.Set<CustomerInfo>() where c.CustomerName.Contains("a") && c.CustomerId<5 select c;  foreach (var item in result)  {  Console.WriteLine("编号：{0},姓名：{1},日期：{2}", item.CustomerId, item.CustomerName, item.CustomerDate == null ? "null" : item.CustomerDate.ToString());  }  Console.Read(); |

连接查询

|  |
| --- |
| DbContext context1 = new EFTestEntities();  //ling条件查询  var result = from c in context1.Set<CustomerInfo>() join o in context1.Set<OrderInfo>() on c.CustomerId equals o.CustomerId select o;  foreach (var item in result)  {  Console.WriteLine("客户编号：{0},订单标题：{1}", item.CustomerId, item.OrderTitle);  }  Console.Read(); |

## 多from结合导航属性进行连接查询

|  |
| --- |
| DbContext context1 = new EFTestEntities();  //ling条件查询  var result = from c in context1.Set<CustomerInfo>() from o in c.OrderInfo where o.OrderTitle=="pea" select c;  foreach (var item in result)  {  Console.WriteLine("客户编号：{0},客户名称：{1}", item.CustomerId,item.CustomerName);  }  Console.Read(); |

## 部分列查询

仅想查询某些列而不是所有列时。需要创建一个数据实体类型（最常用的是匿名类型），在查询时进行赋值。

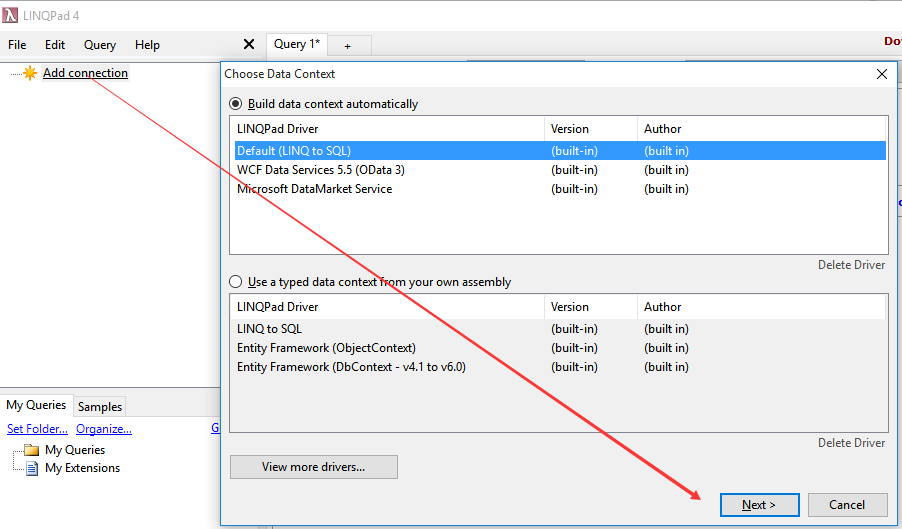
|  |
| --- |
| DbContext context1 = new EFTestEntities();    var result = from c in context1.Set<CustomerInfo>()  select new  {  c.CustomerId, //等同于 CustomerId = c.CustomerID  c.CustomerName//等同于 CustomerName = c.CustomerName  };  foreach (var item in result)  {  Console.WriteLine("客户编号：{0},客户名称：{1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

# Lambda查询

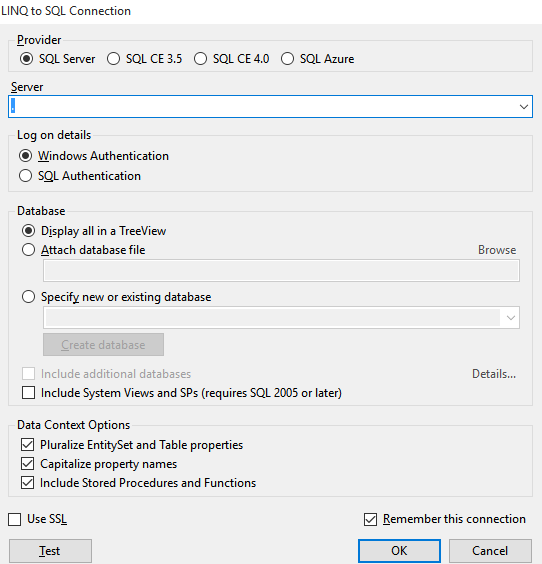
## LINQPAD工具

LINQPad可以把linq语句转换为lambda语句

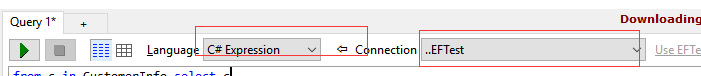
### 创建连接



选择数据库

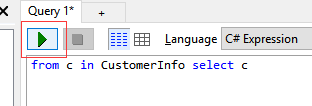


选择语言与连接的数据库

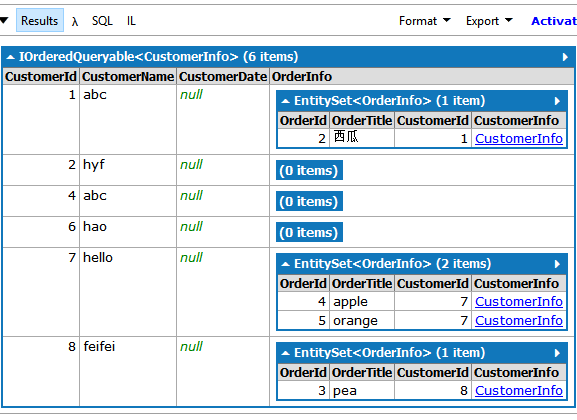


### 编写LINQ语句

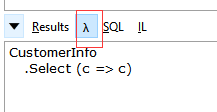
编写并执行



结果



### 查看lambda语句



## 条件查询

|  |
| --- |
| DbContext context1 = new EFTestEntities();  //lambda查询，  var result = context1.Set<CustomerInfo>().Where(c => c.CustomerName.StartsWith("h"));  foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

## 部分列查询

|  |
| --- |
| DbContext context1 = new EFTestEntities();    var result = context1.Set<CustomerInfo>().Select(c => new { c.CustomerId, c.CustomerName });  foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

条件与部分列查询

|  |
| --- |
| DbContext context1 = new EFTestEntities();  var result = context1.Set<CustomerInfo>().Where(c => c.CustomerName.StartsWith("h")).Select(c => new { c.CustomerId, c.CustomerName });  foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

## 排序

|  |
| --- |
| DbContext context1 = new EFTestEntities();  var result = context1.Set<CustomerInfo>().OrderByDescending(c => c.CustomerName);  foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

## 忽略多少条数据

Skip只能在排序后使用。传入参数为忽略的条数

|  |
| --- |
| DbContext context1 = new EFTestEntities();  var result = context1.Set<CustomerInfo>().OrderBy(c=>c.CustomerName).Skip(2);  foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

## 取多少条数据

用法与skip类似

|  |
| --- |
| DbContext context1 = new EFTestEntities();  // var result = context1.Set<CustomerInfo>().Where(c => c.CustomerName.StartsWith("h")).Select(c => new { c.CustomerId, c.CustomerName });  var result = context1.Set<CustomerInfo>().OrderBy(c=>c.CustomerName).Take(1);  foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

## 案例：分页查询

Linq语句不具有，lambda特有的

|  |
| --- |
| DbContext context1 = new EFTestEntities();  int pageSize = 5;  int pageIndex = 1;  var result = context1.Set<CustomerInfo>().OrderBy(c => c.CustomerName).Skip((pageIndex - 1) \* pageSize).Take(pageSize).Select(c => new { c.CustomerId, c.CustomerName });  foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read(); |

### 封装

|  |
| --- |
| private static IQueryable<T> getPaging<T, Tkey>(DbContext context, Expression<Func<T, Tkey>> OrderLambda, int pageSize, int pageIndex) where T : class  {  DbContext context1 = context;  var result = context1.Set<T>().OrderBy(OrderLambda).Skip((pageIndex - 1) \* pageSize).Take(pageSize);  return result;  }  private static IQueryable<T> getPaging<T, Tkey>(DbContext context, Expression<Func<T, bool>> whereLambda, Expression<Func<T, Tkey>> OrderLambda, int pageSize, int pageIndex) where T : class  {  DbContext context1 = new EFTestEntities();  var result = context1.Set<T>().Where(whereLambda).OrderBy(OrderLambda).Skip((pageIndex - 1) \* pageSize).Take(pageSize);  return result;  }  //一般不用这个方法 直接返回所有数据即可  private static IQueryable<dynamic> getPaging<T, Tkey>(DbContext context, Expression<Func<T, Tkey>> OrderLambda, Expression<Func<T, dynamic>> selectLambda, int pageSize, int pageIndex) where T : class  {  DbContext context1 = new EFTestEntities();  var result = context1.Set<T>().OrderBy(OrderLambda).Skip((pageIndex - 1) \* pageSize).Take(pageSize).Select(selectLambda);  return result;  } |

### 调用

|  |
| --- |
| static void Main(string[] args)  {  int pageSize = 5;  int pageIndex = 1;  DbContext dbcontext = new EFTestEntities();  var result1 = getPaging<CustomerInfo, int>(  dbcontext,  c => c.CustomerId,  c => new { c.CustomerId, c.CustomerName },  pageSize,  pageIndex);  foreach (var item in result1)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read();  } |

### 封装思路

类型不同，算法相同用泛型封装，lambda表达式用委托表达式封装。上下文对象用多态封装

# 延迟加载

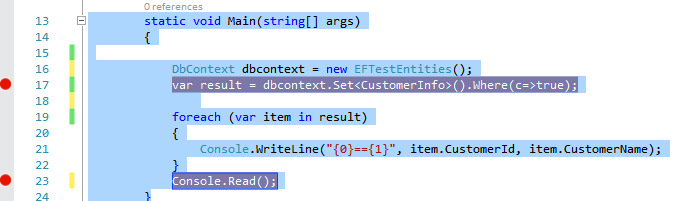
在lambda查询中，数据只有在用到的时候才会加载，这就叫延迟加载。使用where、orderby等返回类型为IQueryable类型集合的方法时，默认会延迟加载，导航属性默认也会延迟。当lambda查询多次使用时，每一次都需要加载。而返回类型为标量值时，会立即加载。

## 延迟加载

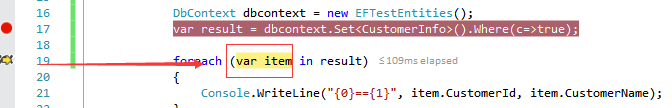
返回类型为IQueryable类型的集合，会延迟加载。

|  |
| --- |
| static void Main(string[] args)  {  DbContext dbcontext = new EFTestEntities();  var result = dbcontext.Set<CustomerInfo>().Where(c=>true);    foreach (var item in result)  {  Console.WriteLine("{0}=={1}", item.CustomerId, item.CustomerName);  }  Console.Read();  } |

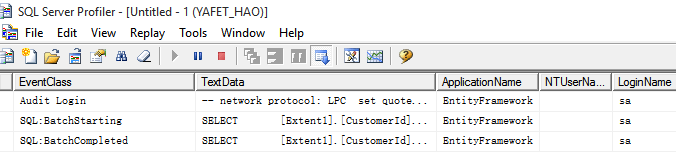
打断点



当执行到第一个断点时，数据库并没有执行任何语句，只是拼接了sql语句，只有用到的时候也就是foreach中执行完 result迭代器后，开始执行拼接的sql语句。



执行的结果：



同样导航属性也是默认延迟加载（EF6好像是立即加载）

|  |
| --- |
| DbContext dbcontext = new EFTestEntities();  var result = dbcontext.Set<CustomerInfo>().First();  foreach (var item in result.OrderInfo)  {  Console.WriteLine("{0}=={1}", item.OrderId, item.OrderTitle);  }  Console.Read(); |

## 立即加载

返回值为标量时。下面的语句当执行完第二条语句后，数据库立即执行。

|  |
| --- |
| DbContext dbcontext = new EFTestEntities();  var result = dbcontext.Set<CustomerInfo>().First();  Console.Read(); |

## 延迟加载转换为立即加载

当语句为延迟加载时，每一次用到都加载，增加数据库压力，可以用ToList方法转换为内存的list集合。以后用到时不需要在去查询数据了。

|  |
| --- |
| DbContext dbcontext = new EFTestEntities();  var result = dbcontext.Set<CustomerInfo>().ToList();//转换为立即加载，把数据库放到内存中的集合中。以后不需再加载  foreach (var item in result)  {  Console.WriteLine(item.CustomerId);  }  foreach (var item in result)  {  Console.WriteLine(item.CustomerName);  }  Console.Read(); |

导航属性立即加载(有待商榷)

|  |
| --- |
| DbContext dbcontext = new EFTestEntities();  var result = dbcontext.Set<CustomerInfo>().Include("OrderInfo");    foreach (var item in result)  {    Console.WriteLine(item.CustomerId);  }  foreach (var item in result)  {  Console.WriteLine(item.CustomerName);  }  Console.Read(); |

## IQueryable和IEnumerable

IQueryable继承自IEnumerable，但是IQueryable增加了3个属性，其中有个重要的Expression类型的属性，可以把lambda多个表达式转换为二叉树的sql语句。

例如，执行下面的语句

|  |
| --- |
| DbContext dbcontext = new EFTestEntities();  var result = dbcontext.Set<CustomerInfo>().Where(c=>c.CustomerName.StartsWith("h")).Where(c=> c.CustomerId<5);  foreach (var item in result)  {  Console.WriteLine(item.CustomerId+"====="+item.CustomerName);  } |

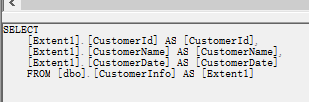
数据库中执行的语句为：

|  |
| --- |
| SELECT  [Extent1].[CustomerId] AS [CustomerId],  [Extent1].[CustomerName] AS [CustomerName],  [Extent1].[CustomerDate] AS [CustomerDate]  FROM [dbo].[CustomerInfo] AS [Extent1]  WHERE ([Extent1].[CustomerName] LIKE N'h%') AND ([Extent1].[CustomerId] < 5) |

多个where拼接为一个sql语句。

而IEnumerable会把所有数据放入内存，之后判断where

|  |
| --- |
| DbContext dbcontext = new EFTestEntities();  var result = dbcontext.Set<CustomerInfo>().AsEnumerable().Where(c=>c.CustomerName.StartsWith("h")).Where(c=> c.CustomerId<5);  foreach (var item in result)  {    Console.WriteLine(item.CustomerId+"====="+item.CustomerName);  } |



# Model First

EF开发方式有三种：

* DB First
* Model First
* Code First

在1.3的案例中，我们使用的就是Db First。也是最常用的。这章我们学习Model First

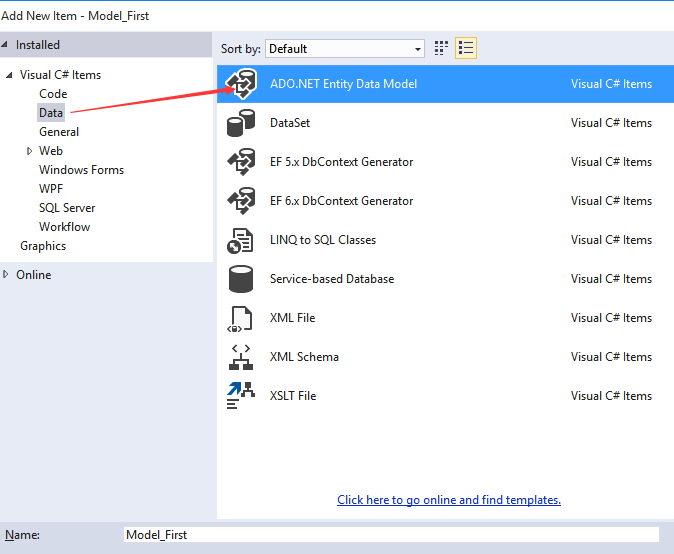
## 什么是Model First

Model First也就是先创建模型，再根据模型创建数据库

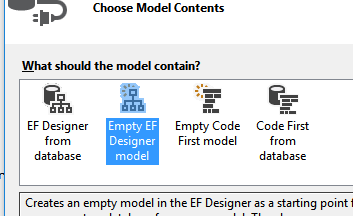
## 流程

### 创建模型文件

创建实体数据模型文件

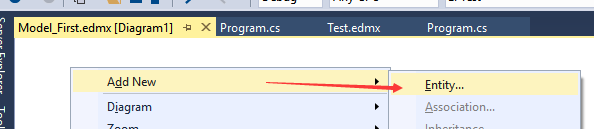


选择空模型

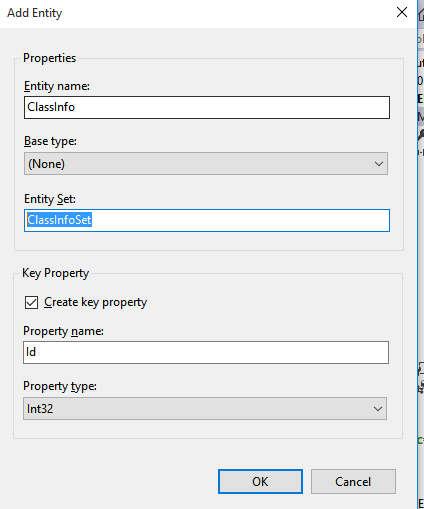


### 添加实体

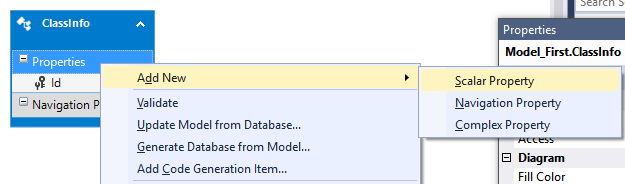
创建实体



添写实体名称，主键属性：



添加标量属性：



修改属性：

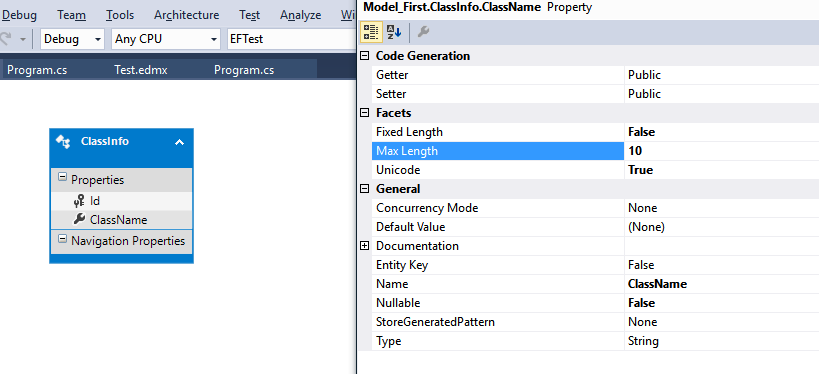
Type：实体类型

Unicode：对应数据库中n，如nchar

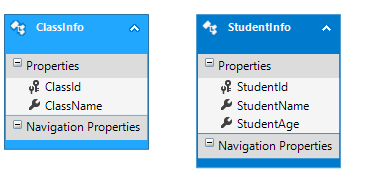
Fixed length：对应数据库中的var，如varchar

这里在facets中的设置对应数据库中的nvarchar(10)

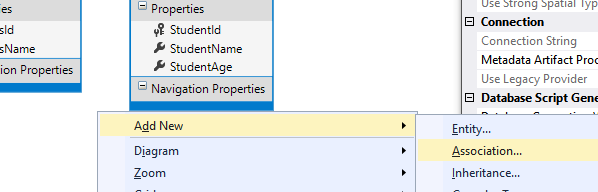
Nullable：是否可为空



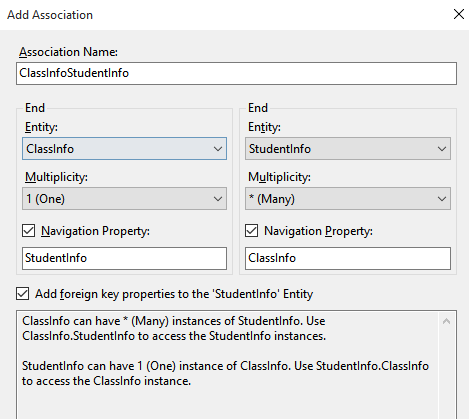
同样的技术创建StudentInfo表



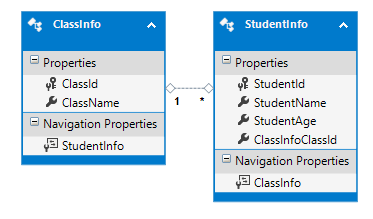
空白位置右击，新增关联



设置对应关系为1对多，添加导航属性，并添加外键信息到SutdentInfo表

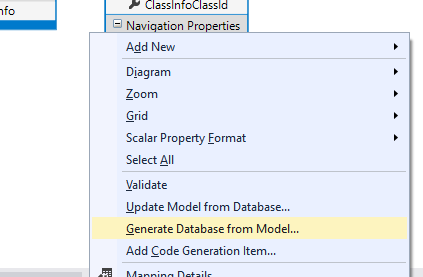


最后的结果

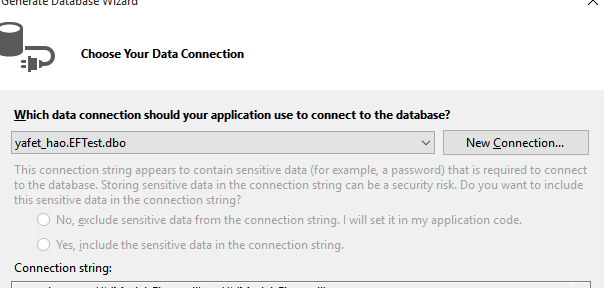


### 根据模型创建数据库

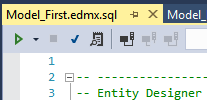
空白右击选择从模型创建数据库



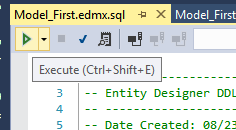
选择数据库



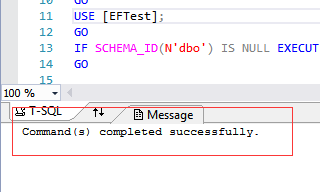
最后生成sql语句



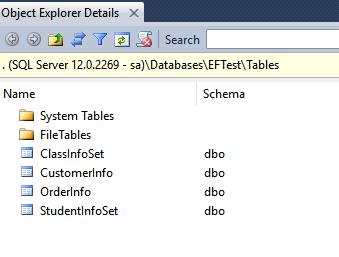
点击执行



执行成功



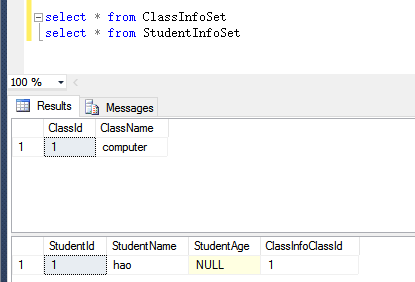
预览数据库



### 向数据库插入数据

|  |
| --- |
| static void Main(string[] args)  {  DbContext context = new Model\_FirstContainer();  ClassInfo ci = new ClassInfo() {  ClassName="computer"  };  ci.StudentInfo.Add(new StudentInfo() {StudentName="hao" });  context.Set<ClassInfo>().Add(ci);  context.SaveChanges();  Console.WriteLine("ok");  Console.Read();  } |

结果：



# Code First

为了在原有项目中引入EF，需要使用Code First。原有项目中，已经有了模型类，所以不再重新生成模型类。

注意：要创建数据库表，需要删除“系统表”下的“[\_\_MigrationHistory]”表

操作步骤

（1）引入程序集EntityFramework，System.Data.Entity（创建EF实体模型文件，删除即可引入程序集）

（2）在配置文件中写连接字符串

（3）创建模型类（如果项目中已经有模型类，则只需要维护关系）

通过导航属性来表示类的关系，注意：导航属性设置成virtual，可以实现延迟加载

特性维护：Table,Key,ForeignKey

（4）创建上下文类，继承自DbContext

调用父类构造方法，传递连接字符串"name=\*\*\*"

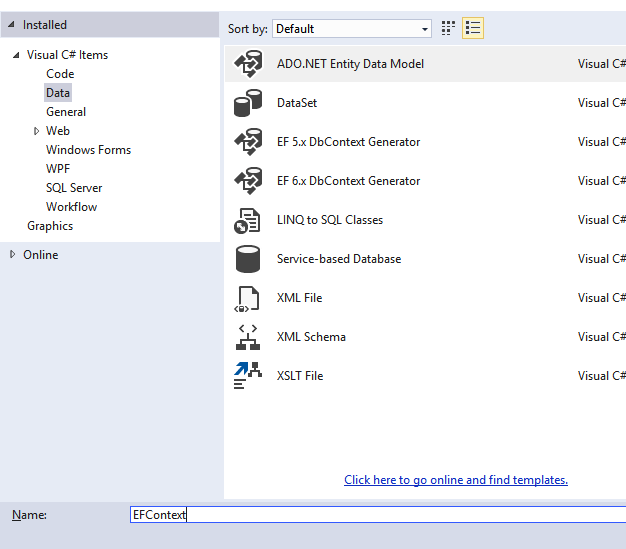
（5）根据类型创建数据库表

使用context.Database.CreateIfNotExists()完成数据库中表的创建

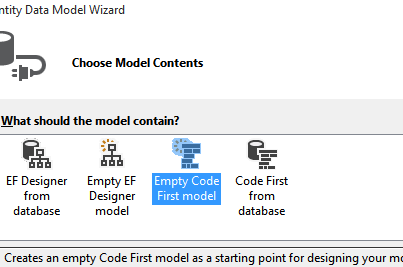
调用context.SaveChanges()方法完成保存

在vs2015中可以创建空的EF模型文件来引入程序集，并且自动创建了上下文类。

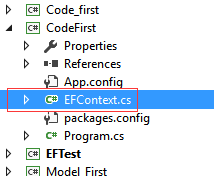
## 创建空的代码优先的模型文件



选择空的代码优先模型



自动创建了上下文类：



## 修改配置文件中自动创建的连接字符串

|  |
| --- |
| <connectionStrings>  <add name="EFContext" connectionString="data source=.;initial catalog=EFTest;user id=sa;password=123;MultipleActiveResultSets=True;App=EntityFramework" providerName="System.Data.SqlClient" />  </connectionStrings> |

## 创建与维护模型类

以model的模式创建两个有关联的表：

|  |  |
| --- | --- |
| ContactType | ContactInfo |
| public class ContactType  {  public int TypeId { get; set; }  public string TypeName { get; set; }  } | public class ContactInfo  {  public int ContactId { get; set; }  public string ContactName { get; set; }  public ContactType ContactType { get; set; }  } |

维护表需要用到特性。

维护ContactInfo类：

|  |
| --- |
| [Table("ContactInfo")]//表面类对应的数据库表的名称  public class ContactInfo  {  [Key]//表面字段对应的是主键  public int ContactId { get; set; }  public string ContactName { get; set; }  [ForeignKey("ContactType")]//参数为导航属性的名称。对应的是该类型的主键  public int TypeId { get; set; }//添加该字段，用来设置外键关系  public virtual ContactType ContactType { get; set; }//通过导航属性来表示类的关系。导航属性设置成virtual，可以实现延迟加载  } |

维护ContactType类：

|  |
| --- |
| [Table("ContactType")]  public class ContactType  {  public ContactType()  {  ContactInfo = new HashSet<ContactInfo>();//构造函数中创建导航属性的集合。  }  [Key]  public int TypeId { get; set; }  public string TypeName { get; set; }  public virtual ICollection<ContactInfo> ContactInfo { get; set; }//导航属性，1：N所以用集合  } |

## 关联上下文类

|  |
| --- |
| public class EFContext : DbContext  {  //第一步构造函数中。设置连接字符串的名称。  public EFContext()  : base("name=EFContext")  {  }  //为每一个类创建dbset集合类型的属性。  public DbSet<ContactInfo> ContactInfo { get; set; }  public DbSet<ContactType> ContactType { get; set; }  } |

向数据库中插入数据

|  |
| --- |
| DbContext context = new EFContext();  context.Database.CreateIfNotExists();//如果数据库不存在则创建表  ContactType ct = new ContactType()  {  TypeName = "friends"  };  ct.ContactInfo.Add(new ContactInfo()  {  ContactName = "haoyufeng"  });  context.Set<ContactType>().Add(ct);  context.SaveChanges();  Console.WriteLine("ok");  Console.Read(); |

# 数据漕管理上下文

在一次操作过程中，需要对数据库中数据进行多次操作时，一般采用上下文实例唯一的方式。主要有两个作用:

* 事务作用：当一部分操作有误时，进行回滚不继续执行。例如向客户表中插入数据失败时，不会继续向订单表中添加数据。
* 减少和服务器数据库的交互次数。

如何实现上下文实例唯一呢？

可以用数据漕[[1]](#footnote-1)技术来保证实例唯一。

## 数据漕使用

创建数据库上下文数据漕管理工厂

|  |
| --- |
| public static class ContextFacyory  {  public static DbContext GetContext()  {  DbContext dbContext = CallContext.GetData("Context") as DbContext;//数据漕中拿数据  if (dbContext == null)  {  dbContext = new Model\_FirstContainer();  CallContext.SetData("Context", dbContext);//往数据漕中放入数据  }  return dbContext;  }  } |

在多个方法中使用同一个上下文

|  |
| --- |
| static void Main(string[] args)  {  AddClassInfo();  AddStudentInfo();  Save();  }  private static void Save()  {  DbContext context = ContextFacyory.GetContext();  context.SaveChanges();  }  private static void AddStudentInfo()  {  DbContext context = ContextFacyory.GetContext();  ClassInfo ci = context.Set<ClassInfo>().Local.First();//获取数据漕中ClassInfo  ci.StudentInfo.Add(new StudentInfo()  {  StudentName = "hao"  });  }  private static void AddClassInfo()  {  ClassInfo ci = new ClassInfo()  {  ClassName = "net"  };  DbContext context = ContextFacyory.GetContext();//获取数据漕中的上下文  context.Set<ClassInfo>().Add(ci);  } |

# MVC简介

## 什么是MVC

MVC模式(Model-View-Controller)是软件工程中的一种软件架构模式[[2]](#footnote-2)，把软件系统分为三个基本部分：模型(Model)、视图(View)和控制器(Controller)。最早由Trygve Reenskaug在1978年提出，并应用在Smalltalk系统中。这时距离微软在1985年推出Windows1.0还有7年之久，即所有的输入、输出、逻辑控制，都要由软件开发者完全实现。

因为输入、输出、处理都是要自己编写程序，所以需要三类人员：

* 精通电路的专业人员，接收电路板传过来的信息，转化为二进制的信息。
* 精通显示设备的专业人员，能够让屏幕中显示出相应的信息。
* 精通各种算法、各种逻辑的专业人员，能够将信息进行加工。

为了让这三类人能更好的发挥自己的长处，Trygve Reenskaug提出了一个解决方案：MVC模式。按照职责的不同，分成了以下三种：

* 控制器(Controller）- 接收用户输入，并完成模型、视图的调用。
* 视图 (View） - 为用户展示结果。
* 模型 (Model）- 业务逻辑处理。

MVC是为了不同的人进行分工开发而提出的。

[](http://zh.wikipedia.org/wiki/File:ModelViewControllerDiagramZh.png)

## 三层模式

三层模式：是MVC设计思想[[3]](#footnote-3)的一种实现，但是分工的内容与MVC完全不一样，三层只是对处理阶段进行的进一步的具体的划分，输入、输出不需要专门的人员进行处理。三层还有一个更重要的思想：解耦；可以进行代码重用。 三层比MVC意义更大，MVC是软件架构，而三层是软件工程当中的一个工程模块[[4]](#footnote-4)。

三层模式包括：

* UI层
* 业务逻辑层
* 数据访问层
* 模型层

如果要严格区分，UI层指View与Controller，业务逻辑层、数据访问层、模型层都被包括在Model中

## Asp.net MVC

Asp.net MVC是微软在遵循MVC思想下提供的一款产品，它与webForm是同一级别，只是一个UI层的框架。它完成了前端页面与后台控制的最大限度解耦。

它将web应用程序分成三个主要组件[[5]](#footnote-5)即：视图（View）、控制器（Controller）、模型（Model）。

* View 是用户接口层组件。主要是将Model中的数据展示给用户。ASPX和ASCX文件被用来处理视图的职责。
* Controller 处理用户交互，从model中获取数据并将数据传给指定的view。
* Model是存储或者是处理数据的组件，具有两个方面的含义：DomainModel，ViewModel。

域模型DomainModel是实现业务逻辑层对实体类相应操作，包括逻辑操作与数据库操作，如验证规则、数据访问和业务逻辑等应用程序信息；

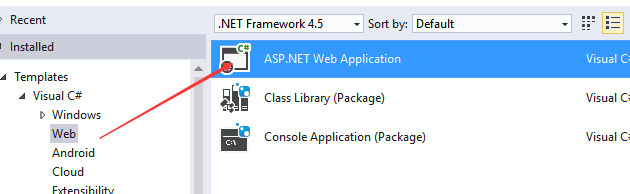
视图模型ViewModel：是与显示页面强关联的模型对象，用于实现页面强类型，如做了连接查询得到的结果要显示到前台，就没有相应的对象存在，这时，就可以新建一个类来包含结果集中的行

三层中的model是数据模型，每一个类对应数据库中的一张表，每一个属性对应表中的一个字段。而Asp.net MVC中的Model是显示模型，是与显示页面强关联的模型对象，不一定与数据库中的表相对应，而且它有额外的域模型。

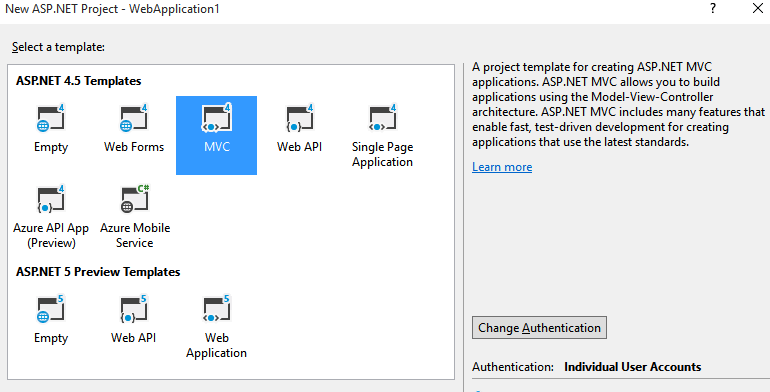
# 案例：MVC创建流程

## 创建MVC项目

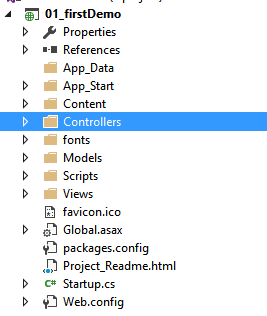
选择asp.net web application



选择MVC



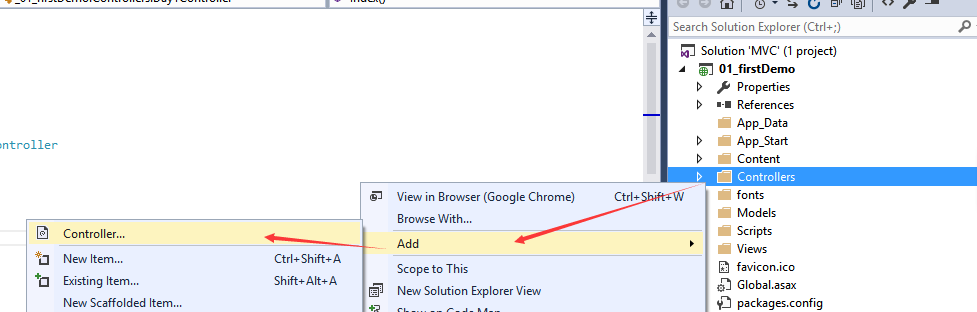
项目结构如下：



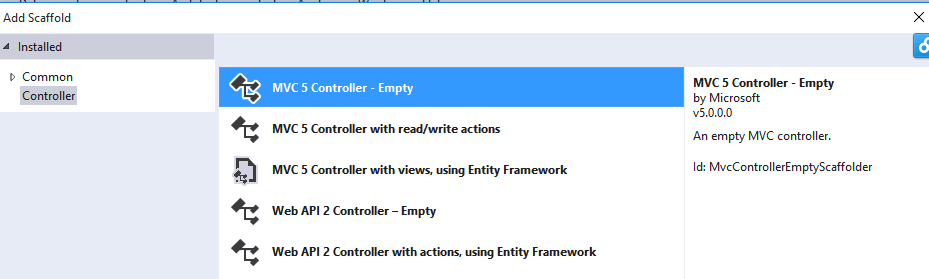
我们的任务就是在Controllers、Models以及Views中添加内容。

## 创建控制器

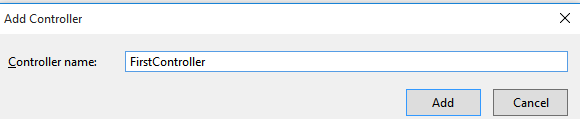
在Controllers文件夹下，创建控制器



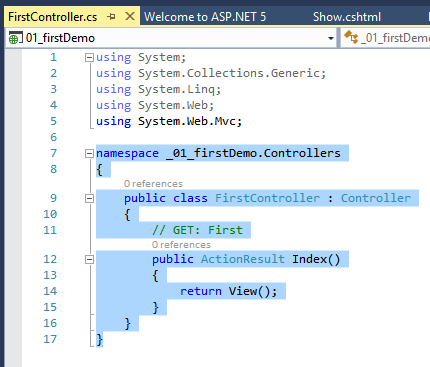
在弹出的对话框中选择空的控制器



填写控制器的名称，后缀需要为Controller



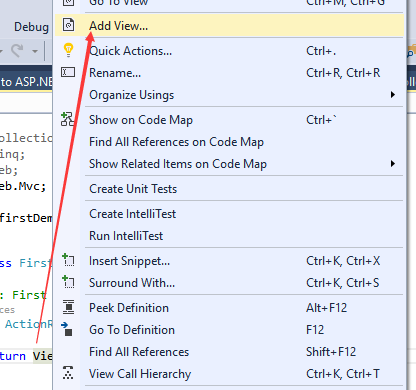
会创建一个名为FirstController的cs文件



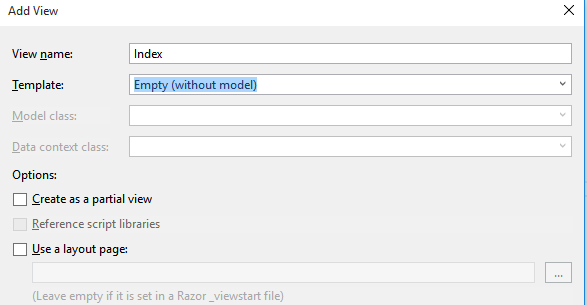
|  |
| --- |
| **控制器与行为** |
| 从上面创建的cs文件中，可以看到控制器实质上就是继承自Controller的类。返回类型为ActionResult的方法叫做行为。可以说控制器由各种行为构成 |

## 创建与编辑视图

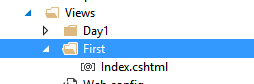
在控制器中的view上右击选择添加视图



填写视图名称。默认名称与方法的名称相同



在Views文件夹下会多出一个与控制器名称相同的文件夹，其包含了一个cshtml类型的视图文件。

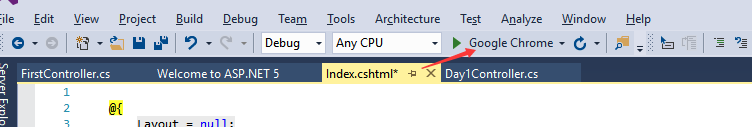


在视图中就可以进行html代码的编写了

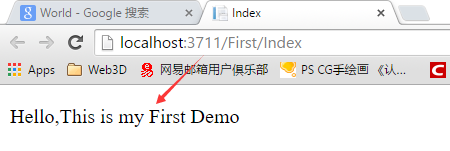


## 预览网页

在视图被选中的情况下，点击浏览器进行调试。



结果如下：

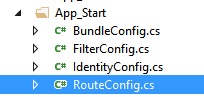


在地址栏中显示的结果为：/控制器名称/行为名称。这个结构是由路由控制的。

我们请求的不再是页面而是控制器下的一个行为

## 设置路由规则

在App\_Start下打开RouteConfig。



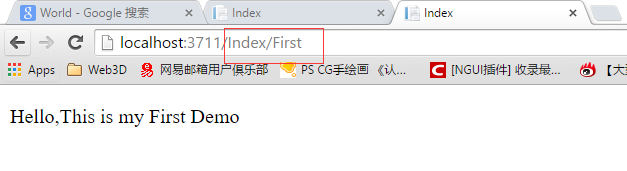
其内容如下：



修改url为：

|  |
| --- |
| {action}/{controller}/{id} |

输入以下地址才能正确访问到页面

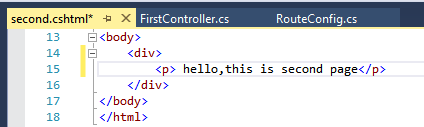


## 行为与视图

在MVC中，访问时，实际访问的是某个类的某个方法 。例如上面的案例中，实际访问的是FirstController类的Index方法。而在asp.net中，访问时，实际访问的是某个类。

在行为（某个方法）中不指定显示页面时，会采用与行为同名的页面进行显示。当指定显示页面时，会显示该指定的页面。行为可以指定不同的页面，实现了控制器与视图的解耦。

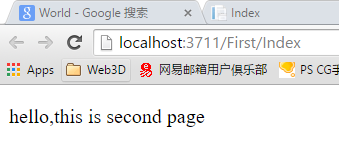
例如，新建名为second的视图，编写内容如下：



指定视图为second

|  |
| --- |
| public ActionResult Index()  {  return View("second");  } |

访问页面时返回的是second视图中的内容



# MVC+EF快速增删改查

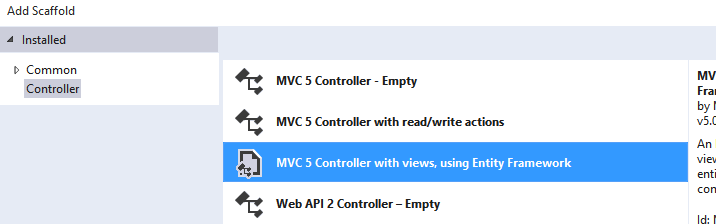
## 添加EF实体模型

在models文件夹下添加实体模型。

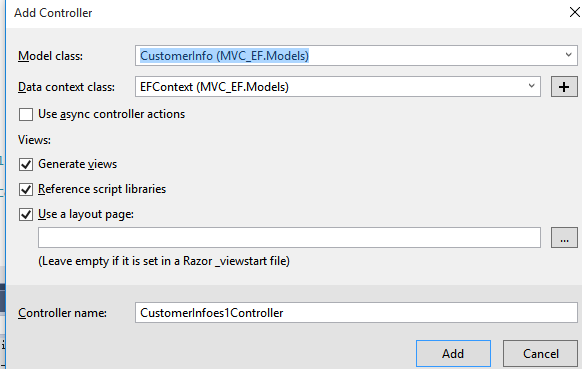


## 添加控制器

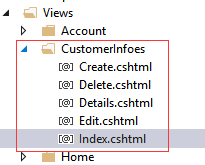
需要生成下项目，才能添加层控制器



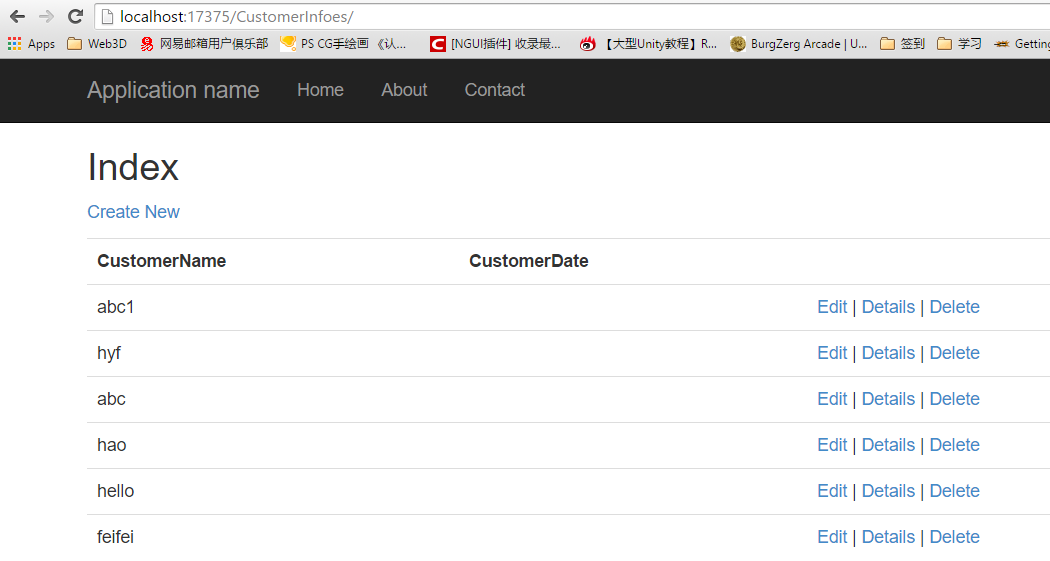
Model class为EF中要绑定的类。



自动生成了以下视图：



## 访问网站



# 路由

路由用来控制请求规则

# 数据传递

通过ViewData可实现从控制器向视图中传递数据。

## 控制器中设置数据

|  |
| --- |
| public class FirstController : Controller  {  public ActionResult Index()  {  ViewData["Name"] = "hao";//后台中设置数据  return View("Index");  }  } |

## 视图中展示数据

|  |
| --- |
| <body>  <div>  <p>@ViewData["Name"]</p>  </div>  </body> |

## 传递强类型数据

强类型数据在编译阶段可以对源代码进行有效性检查。

### 流程

#### 创建类

在Models文件夹中创建Person.cs

|  |
| --- |
| public class Person  {  public int Id { get; set; }  public string Name { get; set; }  } |

#### 控制器设置

|  |
| --- |
| public ActionResult Index()  {  ViewData.Model = new Person() { Id = 1, Name = "hao" };  return View("Index");  } |

#### 视图中展示数据

首先引入类，才能有智能提示

|  |
| --- |
| @model \_01\_firstDemo.Models.Person |

使用属性

|  |
| --- |
| <body>  <div>  <p>@ViewData.Model.Name</p>  </div>  </body> |

## 简写形式（ViewBag）

|  |
| --- |
| ViewBag.a = "hao"; // ViewData["a"] = "hao"; |

# HtmlHelper

## 超链接

使用a标签的形式：

|  |
| --- |
| <body>  <div>  <a href="/CustomerInfoes/Index">首页</a> @\*请求controllers文件夹下CustomerInfoes控制器中的Index行为\*@  </div>  </body> |

存在的一个问题是，当路由规则改变时，所有href都需要改变。改进方法使用：

|  |
| --- |
| <a href="@Url.Action("Index","CustomerInfoes")">首页</a>@\*首页</a> 第一个参数是行为，第二个参数是控制器\*@ |

会根据路由规则生成链接，更进一步的方法：

|  |
| --- |
|  |

## Encode

|  |
| --- |
| @Html.Encode("<p>hello</p") |

结果：

|  |
| --- |
| &lt;p&gt;hello&lt;/p |

## 包含其他页面

|  |
| --- |
| @Html.Action("Index", "CustomerInfoes") @\*第一个参数行为，第二个参数控制器\*@ |

## 表单布局

### 表单行为

|  |
| --- |
| @using (Html.BeginForm("Index", "CustomerInfoes"))  {  <input type="text" />  } |

生成的前台代码：

|  |
| --- |
| <div>  <form action="/CustomerInfoes" method="post">  <input type="text" />  </form>  </div> |

### 自动装配

根据参数生成控件的name、id属性，value会根据ViewData中同名键，如果没有同名键则去ViewData的model中查找同名的属性，获取其数据赋值给value属性；如果没有同名，则value为空

例如：在控制器中有个viewdata

|  |
| --- |
| ViewData["txtbox"] = "hello"; |

在视图中：

|  |
| --- |
| @using (Html.BeginForm("Index", "CustomerInfoes"))  {  @Html.TextBox("txtbox") @\*根据参数txtbox为控件的id、name赋值，value为ViewData中同名键txtbox的值\*@  } |

生成的html代码：

|  |
| --- |
| <form action="/CustomerInfoes" method="post">  <input id="txtbox" name="txtbox" type="text" value="hello" />  </form> |

当ViewState中的值为类型时，如下表

|  |
| --- |
| ViewData.Model = new CustomerInfos() { Id = 1, Name = "Yafet" }; |

参数为属性名称即可绑定其值：

|  |
| --- |
| @Html.TextBox("Name") |

生成的Html代码：

|  |
| --- |
| <input id="Name" name="Name" type="text" value="Yafet" /> |

需要说明的的是label不会自动装配：

|  |
| --- |
| @Html.Label("txtbox") |

生成的html代码：

|  |
| --- |
| <label for="txtbox">txtbox</label> |

#### 下拉列表自动装配

ViewData的值为SelectListItem集合时，使用DropDownList方法可以自动装配下拉列表

在控制器中：

|  |
| --- |
| List<SelectListItem> list = new List<SelectListItem>();  //遍历CustomerInfo表中数据  foreach (var item in db.Set<CustomerInfo>())  {  list.Add(  new SelectListItem()  {  Text = item.CustomerName,  Value = item.CustomerId.ToString()  });  }  ViewData["list"] = list;// List<SelectListItem>类型值 |

视图中完成装配：

|  |
| --- |
| @Html.DropDownList("list") |

最终的结果：

|  |
| --- |
| <select id="list" name="list">  <option value="1">abc1</option>  <option value="2">hyf</option>  <option value="4">abc</option>  <option value="6">hao</option>  <option value="7">hello</option>  <option value="8">feifei</option>  </select> |

#### 强类型结合自动装配

在上面的自动装配中，参数都是我们手动输入的，即使与ViewData中的键不匹配，编译时也不报错。自动装配不成功，很难找到错误。

用强类型结合自动装配（\*For）在编译时会进行错误检查，可以有效防止装配失败。但是需要注意的是它只能装配model。

下面代码为使用方法：

在控制器中设置model

|  |
| --- |
| ViewData.Model = new CustomerInfo() { CustomerId = 1, CustomerName = "hao" }; |

在视图中，首先引入类型

|  |
| --- |
| @model MVC\_EF.Models.CustomerInfo |

进行装配：

|  |
| --- |
| @Html.TextBoxFor(c => c.CustomerName); |

# Controller详解

## 行为重载

行为本质上就是方法。当方法目的是作为一个资源来请求时，叫做行为[[6]](#footnote-6)。行为可以重载。针对行为的重载要满足两个条件：

* 参数不同
* 必须指定请求方式。行为默认的请求方式为HttpGet。

下面的代码演示了行为的重载：

|  |
| --- |
| // 默认请求为get  public ActionResult Index()  {  ViewBag.a = 0;  return View();  }  [HttpPost] // 必须指定请求  public ActionResult Index(int id)// 方法参数不同  {  ViewBag.a = id;//传递过来的参数的值是：name为id的标签元素的值。  return View();  } |

在视图中：

|  |
| --- |
| @ViewData["a"]  @using (Html.BeginForm("Index", "Test"))  {  <input type="text" id="id" name="id" /> @\*name为id与行为中参数的名称相同\*@  <input type="submit" value="提交" />  } |

第一次过来时：



执行的是Index()行为。

然后文本框中输入：123，提交：



执行的是Index(int id)行为，id的值为文本框中输入的值。

|  |  |
| --- | --- |
| 注意 | 如果请求方式为HtppGet，Index(int id)的参数以下面的方式获得值 |
| http://127.0.0.1/Test/Index?id=1 | |

## 数据传递

有三种方式可以实现从客服端向服务器端传递数据：

* Get方式
* Post方式
* 路由规则

控制器中代码：

|  |
| --- |
| public ActionResult Index(int id)  {  ViewBag.a = id;  return View();  } |

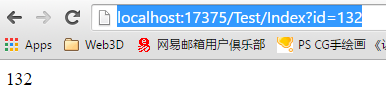
视图中代码：

|  |
| --- |
| @ViewData["a"] |

### Get方式

|  |
| --- |
| http://localhost:17375/Test/Index?id=132 |

结果：



### Post方式

提交的表单中元素的name属性需要与行为的参数的名称相同。

|  |
| --- |
| @ViewData["a"]  @using (Html.BeginForm(Url))  {  @Html.TextBox("id"); //textboax的name需要与行为的参数的名称相同  <input type="submit" value="提交" />  } |

点击按钮提交时，行为会把name为id的textbox的值赋值给其参数。

### 路由方式

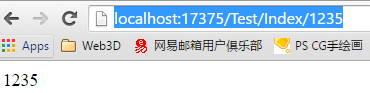
在路由规则设置中修改id与参数的名称一致。在这里名称就是id

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

在地址栏：

|  |
| --- |
| http://localhost:17375/Test/Index/1235 |

结果：



## 自动装配多个标签的值到参数

当参数中获取多个提交过来的值时，如果定义多个参数效率比较低。最常用的是定义一个类型。类型的属性与标签的属性同名或者get过来的键同名。

使用方法：

1. 定义类型：

|  |
| --- |
| public class Student  {  public int Id { get; set; }  public string Name { get; set; }  } |

1. 控制器中代码：

|  |
| --- |
| public ActionResult Index()  {  return View();  }  /// <summary>  /// post过来时，用一个Student类型参数来接收提交的值  /// </summary>  /// <param name="st"></param>  /// <returns></returns>  [HttpPost]  public ActionResult Index(Student st)  {  string s = st.Id + ":" + st.Name;  return Content(s);  } |

1. 表单中标签的元素的属性与Student类型的属性同名

|  |
| --- |
| @using (Html.BeginForm("Index","Test"))  {  <input type="text" name="Id" />  <br />  <input type="text" name="Name" />  <input type="submit" value="提交" />  } |

1. 当按”提交”按钮时，控制器中会自动把标签为Id的标签的值赋值给Student参数的Id属性，把标签为Name的标签的值赋值给Student参数的Name属性。

## 案例：实现修改操作

1. 创建行为

|  |
| --- |
| private EFContext db = new EFContext();  public ActionResult Modify(int Customerid)  {  ViewData.Model = db.Set<CustomerInfo>().Find(Customerid);  return View();  } |

1. 修改路由规则

|  |
| --- |
| url: "{controller}/{action}/{Customerid}", |

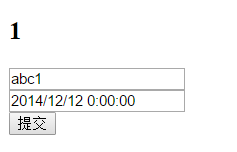
1. 视图：

|  |
| --- |
| <div>  <h2>@ViewData.Model.CustomerId</h2>  @using (Html.BeginForm("Modify", "CustomerInfoes"))  {  @Html.TextBoxFor(c => c.CustomerName)  <br />  @Html.TextBoxFor(c => c.CustomerDate)  <br />  <input type="submit" value="提交" />  }  </div> |

1. 通过路由规则访问网址

|  |
| --- |
| http://localhost:17375/CustomerInfoes/Modify/1 |

结果：



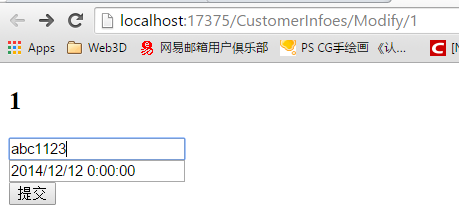
1. 查看源代码

|  |
| --- |
| <h2>1</h2>  <form action="/CustomerInfoes/Modify/1" method="post">  <input id="CustomerName" name="CustomerName" type="text" value="abc1" />  <br />  <input data-val="true" data-val-date="The field CustomerDate must be a date." id="CustomerDate" name="CustomerDate" type="text" value="2014/12/12 0:00:00" />  <br />  <input type="submit" value="提交" />  </form> |

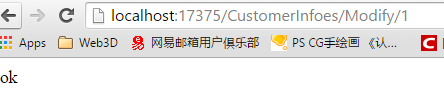
1. 编写post提交的行为。参数为CustomerInfo类型。

|  |
| --- |
| [HttpPost]  public ActionResult Modify(CustomerInfo customerInfo)  {  db.Entry(customerInfo).State = EntityState.Modified;  db.SaveChanges();  return Content("ok");  } |

1. 修改值：



1. 成功后返回如下结果：



## 行为返回结果

行为返回类型为ActionResult，是一个抽象类，需要返回具体类型的结果对象。下面为常见的结果对象，它们都直接或间接继承自ActionResult

* ViewResult：使用View()可以指定一个页面，也可以指定传递的模型对象，如果没有指定参数则表示返回与Action同名的页面
* ContentResult：使用Content(string content)返回一个原始字符串
* RedirectResult：使用Redirect(string url)将结果转到其它的Action
* JsonResult：使用Json(object data)将data序列化为json数据并返回，推荐加上JsonRequestBehavior.AllowGet可以处理Get请求，一般结合客户端的ajax请求进行返回。用法见第 15 章。

### RedirectResult

|  |
| --- |
| public ActionResult Index()  {  return Redirect("/CustomerInfoes/Create");//参数为网址  } |

使用Redirect的缺陷是路由规则改变时，需要重新调整URL。而RedirectToAction根据路由规则生成网址

|  |
| --- |
| public ActionResult Index()  {  return RedirectToAction("Create", "CustomerInfoes");  } |

# 路由Route

## 与路由相关类型

路由规则Route：制定路由的名称Name、url规则、默认值Default、约束constraints

路由数据RouteData：当前上下文匹配到某个路由规则后，得到的一个对象

路由集合RouteCollection：存放路由规则对象的集合

路由表RouteTable：包含一个静态的RouteCollection类型的属性

在Global.asax中，RouteTable中存储的是当前应用程序的所有路由规则。路由规则存在所静态属性Routes中。Routes是一个RouteCollection类型。

|  |
| --- |
| outeConfig.RegisterRoutes(RouteTable.Routes); |

在RouteConfig类中。MapRoute返回的是一个Route对象，该对象会被添加到RouteCollection集合中。

|  |
| --- |
| public class RouteConfig  {  public static void RegisterRoutes(RouteCollection routes)  {  routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");  routes.MapRoute(  name: "Default",  url: "{controller}/{action}/{id}",  defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  );  }  } |

## 自定义路由规则

自定义路由规则的要求是：越具体越靠前，越抽象越靠后

|  |
| --- |
| routes.MapRoute(  name: "NewsShow",  url: "News/{year}-{month}-{day}-{id}",  defaults: new { Controller = "News", Action = "Show" },  constraints: new { year = @"\d{4}$", month = @"\d{1,2}$" }  );  routes.MapRoute(  "NewsList",  "News/{type}-{pageSize}-{pageIndex}",  new { controller = "News", action = "List" }  ); |

当进行匹配时，它从最上面定义的规则开始匹配，如果没有匹配上，则开始匹配第二个，还没有匹配上，就顺序往下进行，直到匹配上为止。如果没有匹配成功，返回404错误。  
需要注意：

名称不能重复

范围小的规则要写在前面

自定义路由规则时，为了seo优化，不要写得太深，即/不要太过，最好不要超过2个

不能使用/开头

不要改变关键字controller、action

### 使用自定义路由的方法

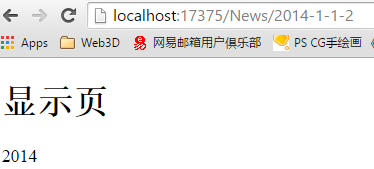
创建一个News控制器。Show为请求的URL符合NewsShow规则时请求的行为。请求中News后面的类似{year}-{month}-{day}-{id}这样的值自动装配为Show行为的参数。

|  |
| --- |
| public class NewsController : Controller  {    public ActionResult Index()  {  return View();  }  public ActionResult Show(int year, int month, int day, int id)  {  ViewBag.year = year;  return View();  }  public ActionResult List(int type, int pageSize, int pageIndex)  {  ViewBag.type = type;  return View();  }  } |

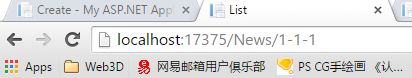
Show对应的视图为：

|  |
| --- |
| <div>  <h1>显示页</h1>  @ViewBag.year  </div> |

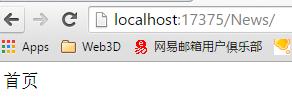
当请求的网页如下时,将调用Show行为



类似的，当请求的网页为下面的网页时，将调用List行为。



当请求如下时，将调用Index行为



# 异步分页加载（jquery）

## 行为重载以实现首页加载列表

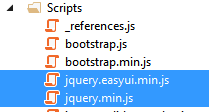
为了方便，我们让首页实现加载列表，也就是返回一个加载列表的视图

|  |
| --- |
| public ActionResult Index()  {  return View("CustomerList");  } |

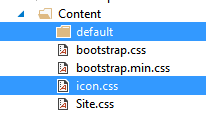
## 引入easyUI

在CustomerList视图中，引入easyUI

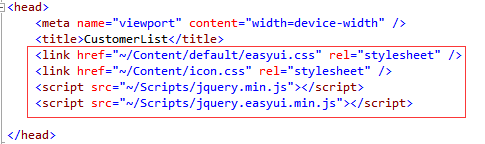
首先把脚本放入scripts文件夹



把主题、样式放入Content文件夹



在head中引用



## 添加展示数据表格的table元素

|  |
| --- |
| <table id="dg" title="Custom DataGrid Pager" style="width:700px;height:250px"  data-options="rownumbers:true,singleSelect:true,pagination:true,pageList:[5,10,15],url:'@Url.Action("GetPageList","CustomerInfoes")',method:'post'">  <thead>  <tr>  <th data-options="field:'CustomerId',width:80">Customer Id</th>  <th data-options="field:'CustomerName',width:80">Customer Name</th  </tr>  </thead>  </table> |

## 页面加载完成调用datagrid

调用dataGrid初始化表格以及获取并展示数据

|  |
| --- |
| <script type="text/javascript">  $(function () {  var pager = $('#dg').datagrid().datagrid('getPager'); // get the pager of datagrid  pager.pagination({  buttons: [{  iconCls: 'icon-search',  handler: function () {  alert('search');  }  }, {  iconCls: 'icon-add',  handler: function () {  alert('add');  }  }, {  iconCls: 'icon-edit',  handler: function () {  alert('edit');  }  }]  });  })  </script> |

## 返回数据的行为

|  |
| --- |
| public ActionResult GetPageList()  {  int pageSize = 5;  int pageIndex = 1;  int.TryParse(this.Request["page"], out pageIndex);  int.TryParse(this.Request["rows"], out pageSize);  pageSize = pageSize <= 0 ? 5 : pageSize;  pageIndex = pageIndex < 1 ? 1 : pageIndex;  int total = db.CustomerInfo.Count();  var rows = db.CustomerInfo  .OrderByDescending(c => c.CustomerId)  .Skip((pageIndex - 1) \* pageSize)  .Take(pageSize)  .Select(c => c);  return Json(new { total, rows }, JsonRequestBehavior.AllowGet);  } |

## 第二种方式

|  |
| --- |
| JQuery EasyUI的datagrid的使用方式总结  第一步：添加样式和js脚本在前台添加展示数据表格的table元素  例如：<table id="tt"></table>    第二步：在doucment.ready中初始化表格的属性以及数据获取的方式。  例如：  $("#tt").datagrid({  url: '/Home/GetPageList',  title: '顾客列表',  width: 700,  height: 400,  fitColumns: true,  idField: 'CustomerID',  loadMsg: '正在加载顾客的信息...',  pagination: true,  singleSelect: false,  pageSize: 10,  pageNumber: 1,  pageList: [10, 20, 30],  queryParams: {},  columns: [[  { field: 'ck', checkbox: true, align: 'left', width: 50 },  { field: 'CustomerID', title: '主键', width: 80 },  { field: 'CustomerName', title: '用户名', width: 120 },  { field: 'CustomerDate', title: '注册时间', width: 80, align: 'right',  formatter:function(value,row,index){  return (eval(value.replace(/\/Date\((\d+)\)\//gi, "new Date($1)"))).pattern("yyyy-M-d h:m:s");  }  }  ]],  toolbar: [{  id: 'toolbarAdd',  text: '添加',  iconCls: 'icon-add',  handler: function () {  $.messager.alert("添加", "添加正文");  }  }]  });  第三步：后台设置加载的数据：  注意：表格Post或者get回来的请求中  page：3 代表page为key，然后选择的当前页码为3  rows：10 代表一页的大小为10  后台返回的数据的格式为：{total：''，rows：[{},{}]}  只要包含了总数tatol字段，rows是具体的行数  例如：  Asp.Net MVC 例子：  public JsonResult GetAllUserInfos()  {  int pageSize = 5;  int pageIndex = 1;  int.TryParse(this.Request["page"], out pageIndex);  int.TryParse(this.Request["rows"], out pageSize);  pageSize = pageSize <= 0 ? 5 : pageSize;  pageIndex = pageIndex < 1 ? 1 : pageIndex;  var temp = db.UserInfo  .OrderBy(u=>u.Sort)  .Skip<UserInfo>((pageIndex-1)\*pageSize)  .Take<UserInfo>(pageSize)  .ToList<UserInfo>();  Hashtable ht = new Hashtable();  ht["total"] = db.UserInfo.Count();  ht["rows"] = temp;  return Json(ht);  }    Asp.Net WebForm 例子：  public void ProcessRequest(HttpContext context)  {  context.Response.ContentType = "text/plain";  var strWebName = context.Request["WebName"] ?? string.Empty;  var GoodsNo = context.Request["GoodsNo"] ?? string.Empty;  int categoryId = 0;  int pageIndex = 1;  int pageSize = 10;  int.TryParse(context.Request["rows"], out pageSize);  int.TryParse(context.Request["page"], out pageIndex);  decimal priceLeft = 0;  decimal priceRight = 1000000;  int goodsStatus = 0;  decimal.TryParse(context.Request["PriceLeft"], out priceLeft);  decimal.TryParse(context.Request["PriceRight"], out priceRight);  int.TryParse(context.Request["CategoryId"], out categoryId);  int.TryParse(context.Request["GoodsStatus"], out goodsStatus);  var goodsQueryParamter = new GoodsQueryParamter();      goodsQueryParamter.GoodsStatus = (Model.GoodsModel.GoodsStatusEnum)goodsStatus;  var ds = goodsService.GetGoodsList(goodsQueryParamter);  string json = string.Empty;  if (ds != null && ds.Tables.Count > 0)  {  System.Text.StringBuilder rowJson = new System.Text.StringBuilder();  int colLen = ds.Tables[0].Columns.Count;  DataColumnCollection col = ds.Tables[0].Columns;  foreach (DataRow row in ds.Tables[0].Rows)  {  System.Text.StringBuilder colJson = new System.Text.StringBuilder();  rowJson.Append("{");  for (int i = 0; i < colLen; i++)  {  colJson.Append("\"" + col[i].ColumnName + "\":\"" + row[i].ToString() + "\",");  }  rowJson.Append(colJson.ToString().TrimEnd(','));  rowJson.Append("},");  }  json = "{\"total\":" + ds.Tables[0].Rows[0]["sumGoods"] + ",\"rows\":[" + rowJson.ToString().TrimEnd(',') + "]}";  }  context.Response.Write(json);  }  关于请求参数与响应：  请求参数：page表示当前页码  rows表示每页显示的数据条数  返回json格式：total表示总条数  rows表示具体要显示的数据 |

# 异步表单实现添加（ajaxHelper）

## 跳转到添加页

在第 15 章中，点击添加按钮后，触发Add行为。

|  |
| --- |
| iconCls: 'icon-add',  handler: function () {  window.location.href = "@Url.Action("Add","CustomerInfoes")";  } |

在Add行为中返回添加页视图

|  |
| --- |
| public ActionResult Add()  {  return View();  } |

## 添加页使用异步表单

### 添加引用

|  |
| --- |
| <script src="~/Scripts/jquery.min.js"></script>  <script src="~/Scripts/jquery.unobtrusive-ajax.min.js"></script> @\*异步表单需要引入的脚本可以用nuget下载：Install-Package Microsoft.jQuery.Unobtrusive.Ajax \*@ |

### 创建异步表单

|  |
| --- |
| <body>  <div>  @using (Ajax.BeginForm("Add", "CustomerInfoes", new AjaxOptions()  {  OnSuccess = "AfterOK" /\*提交成功后触发事件\*/  }))  {  //创建表单元素  @Html.TextBoxFor(c => c.CustomerName)  <br />  <input type="submit" value="提交" />  }  </div>  </body> |

### 行为接收提交信息

创建行为，并接受信息

|  |
| --- |
| [HttpPost]  public ActionResult Add(CustomerInfo customerInfo)  {  string result = "0";  db.CustomerInfo.Add(customerInfo);  if (db.SaveChanges() > 0)  {  result = "1";  }  return Content(result);  } |

### 判断返回信息

|  |
| --- |
| <script type="text/javascript">  function AfterOK(msg) {  if (msg == "1") {  location.href = "@Url.Action("Index","CustomerInfoes")";  } else {  alert("添加失败！");  }  }  </script> |

# 校验

例如需要校验客户端上名为CustomerName的文本框。

|  |
| --- |
| <input id="CustomerName" name="CustomerName" type="text" value="" /> |

可以进行如下操作

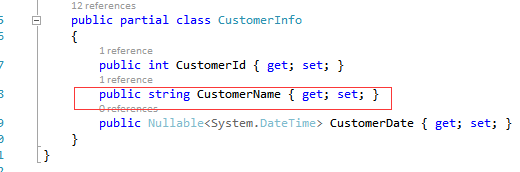
## 引入脚本

|  |
| --- |
| <script src="~/Scripts/jquery.validate.min.js"></script>  <script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script> |

## 通过伙伴类添加特性

为其同名的属性CustomerName属性添加特性。

CustomerName



可以创建一个新的cs文件，定义两个类，其中一个为CustomerInfo部分类：

|  |
| --- |
| public partial class CustomerInfo  {  }  /// <summary>  /// 定义伙伴类  /// </summary>  public class CustomerInfoFrieds  {  //属性名称需要与CustomerInfo中的名称一致  public string CustomerName { get; set; }  } |

为CustomerName属性添加特性

|  |
| --- |
| [Required(ErrorMessage = "用户名不能为空")] //ErrorMessage为自定义错误消息  public string CustomerName { get; set; } |

CustomerInfo类共享CustomerInfoFrieds类的元信息

|  |
| --- |
| [MetadataType(typeof(CustomerInfoFrieds))]  public partial class CustomerInfo  {  } |

## 添加提示信息

|  |
| --- |
| //创建表单元素  @Html.TextBoxFor(c => c.CustomerName)  //校验提示信息  @Html.ValidationMessageFor(c=>c.CustomerName) |

## 客户端查看源代码

|  |
| --- |
| <input data-val="true" data-val-required="用户名不能为空" id="CustomerName" name="CustomerName" type="text" value="" /> |

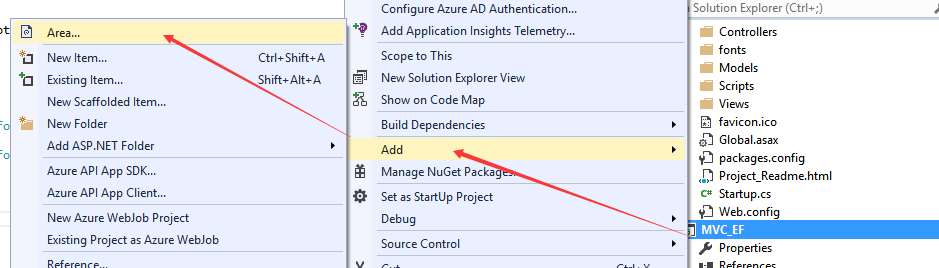
综合：主要的校验特性

|  |
| --- |
| [Required]  [StringLength]  [Range]  [RegularExpression]  属性ErrorMessage：指定错误提示信息 |

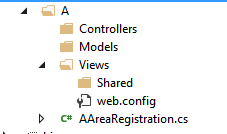
# 区域

当项目中控制器非常多，想要按功能对其进行分类时，可以把它们划分到区域中，这个区域相当于一个子项目。管理更加方便

## 添加区域



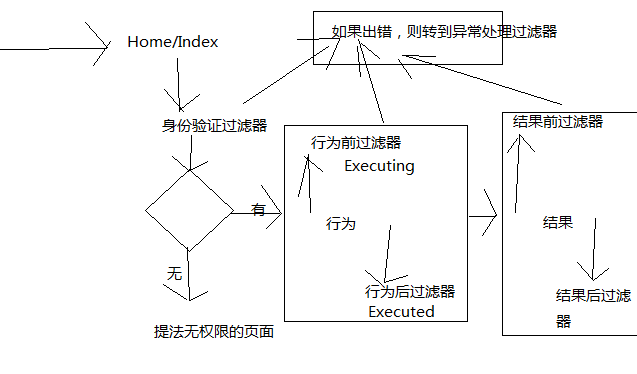
区域的结构：



## 重写路由规则

可以在AreaRegistration.cs中重写区域中请求的路由规则

# 过滤器



过滤器思想是AOP（面向切片编程），也就少无侵入（不修改源代码）的插入新的功能。

## 身份验证过滤器

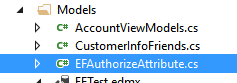
身份验证过滤器在行为之前执行。验证失败可以自定义跳转页面。

有两种实现方式：

* 使用特性
* 控制器中重写OnAuthorization方法

### 使用特性

新建类



继承AuthorizeAttribute类，并重写OnAuthorization方法

|  |
| --- |
| public override void OnAuthorization(AuthorizationContext filterContext)  {  string name = filterContext.HttpContext.Request["name"];  string pwd = filterContext.HttpContext.Request["pwd"];  if (name == "admin" && pwd == "123")  {  filterContext.HttpContext.Response.Write("身份验证成功！");  }  else    {  //服务器跳转，后续的行为不会再执行。使用filterContext.HttpContext.Response.Redirect，客户端会跳转，但是服务器端还会执行后续的行为。浪费服务器资源。  filterContext.Result = new RedirectResult("/NoAccess.html");  }  } |

编写错误提示页



为行为添加特性

|  |
| --- |
| [EFAuthorize]  public ActionResult Index()  {  return View("CustomerList");  } |

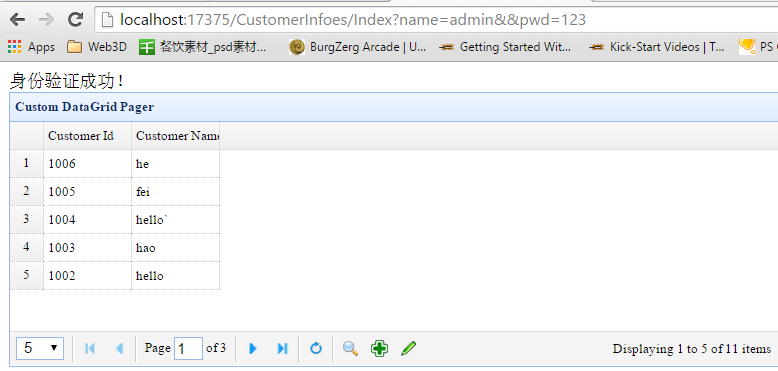
请求该行为：

|  |
| --- |
| http://localhost:17375/CustomerInfoes/Index |

结果：



如果传入正确的参数则访问成功



#### 全局特性

在行为上添加的验证特性，只会在请求该行为时进行验证。如果想要所有行为都进行验证，则可以把特性添加到控制器类上。

|  |
| --- |
| namespace MVC\_EF.Controllers  {  [EFAuthorize]  public class CustomerInfoesController : Controller  { |

### 控制器中重写OnAuthorization方法

|  |
| --- |
| public class CustomerInfoesController : Controller  {  protected override void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext)  {  string name = filterContext.HttpContext.Request["name"];  string pwd = filterContext.HttpContext.Request["pwd"];  if (name == "admin" && pwd == "123")  {  filterContext.HttpContext.Response.Write("身份验证成功！");  }  else  {  //服务器跳转，后续的行为不会再执行。使用filterContext.HttpContext.Response.Redirect，客户端会跳转，但是服务器端还会执行后续的行为。浪费服务器资源。  filterContext.Result = new RedirectResult("/NoAccess.html");  }  } |

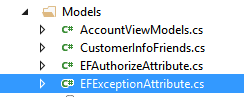
## 异常处理过滤器

当代码发生异常处理的时候，会自动转到异常处理过滤器来，进行异常处理。  
与身份验证过滤器使用方法一样，也有两种使用方式

* 使用特性
* 重写OnException

### 使用特性

新建类，后缀名称为Attribute，

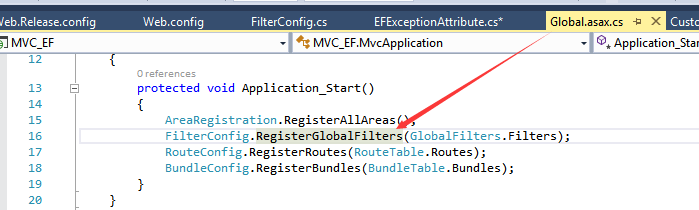


继承自HandleErrorAttribute，重写OnException()方法

|  |
| --- |
| public class EFExceptionAttribute : HandleErrorAttribute  {  public override void OnException(ExceptionContext filterContext)  {  base.OnException(filterContext);//建议不删除  //异常处理:一把就是记录错误日志  string Error = filterContext.Exception.Message;  File.WriteAllText(@"D:\error.txt", Error);  //转到异常处理页  filterContext.Result = new RedirectResult("/Error.html");  }  } |

异常处理特性与身份验证特性一样，也可以在行为、控制器类上添加，但是异常处理有个额外的全局设置，可以对所有的控制器类起作用。使用方法如下：

(1) 在global.asax中，在RegisterGlobalFilters上右击，点击转到定义。



(2) 添加自动异常处理类到GlobalFilterCollection集合中

|  |
| --- |
| public static void RegisterGlobalFilters(GlobalFilterCollection filters)  {  //filters.Add(new HandleErrorAttribute());  filters.Add(new MVC\_EF.Models.EFExceptionAttribute());  } |

### 异常处理的两种规范

当ASP.NET MVC程序出现了异常，怎么处理更加规范？

1. 最简单的方法是设置<customErrors/>节点

<customErrors>节点用于定义一些自定义错误信息的信息。此节点有Mode和defaultRedirect两个属性，其中defaultRedirect属性是一个可选属性，表示应用程序发生错误时重定向到的默认URL，如果没有指定该属性则显示一般性错误。Mode属性是一个必选属性，它有三个可能值，它们所代表的意义分别如下：  
On 表示在本地和远程用户都会看到自定义错误信息。  
Off 禁用自定义错误信息，本地和远程用户都会看到详细的错误信息。  
RemoteOnly 表示本地用户将看到详细错误信息，而远程用户将会看到自定义错误信息。

这里有必要说明一下本地用户和远程用户的概念。当我们访问asp.net应用程时所使用的机器和发布asp.net应用程序所使用的机器为同一台机器时成为本地用户，反之则称之为远程用户。在开发调试阶段为了便于查找错误Mode属性建议设置为Off，而在部署阶段应将Mode属性设置为On或者RemoteOnly，以避免这些详细的错误信息暴露了程序代码细节从而引来黑客的入侵。

<error>子节点

在<customErrors>节点下还包含有<error>子节点，这个节点主要是根据服务器的HTTP错误状态代码而重定向到我们自定义的错误页面，注意要使<error>子节点下的配置生效，必须将<customErrors>节点节点的Mode属性设置为“On”。下面是一个例子：

**[html]** [view plaincopy](http://blog.csdn.net/sundacheng1989/article/details/9000596)

1. **<customErrors** mode="On" defaultRedirect="GenericErrorPage.htm"**>**
2. **<error** statusCode="403" redirect="403.htm" **/>**
3. **<error** statusCode="404" redirect="404.htm" **/>**
4. **</customErrors>**

在上面的配置中如果用户访问的页面不存在就会跳转到404.htm页面，如果用户没有权限访问请求的页面则会跳转到403.htm页面，403.htm和404.htm页面都是我们自己添加的页面，我们可以在页面中给出友好的错误提示。

2. 使用过滤器HandleErrorAttribute

随着ASP.NET MVC版本的更新，出现了HandleErrorAttribute，使用Filter以AOP的思想实现了针对于Action的异常处理。使用此Filter后，当程序中出现异常的时候，会去封装这些异常信息，然后路由自动转到该Controller对应的Error.cshtml中，如果此路径下没有改文件，则会到shared目录中寻找此文件。另外一个相关的是在Global.asax中的protected void Application\_Error(object sender, EventArgs e)方法，是捕捉异常的最后一道防线，也就是说，这是最高层次的异常捕获处理逻辑。使用HandleErrorAttribute后，找到了Error.cshtml，则此时异常已经被捕获处理，所以不会再次被Application\_Error捕获处理。此外，可以通过@model HandleErrorInfo 在Error.cshtml中定制显示异常信息。有一点需要注意的是，HandleErrorAttribute是在customErrors基础之上的，如果想使用HandleErrorAttribute，customErrors的Mode必须要设置为On或RemoteOnly. 否则，HandleErrorAttribute将不起作用。

3. 自定义显示路径

如果遇到异常后不单单是要自定义异常显示，而且还需要进行日志记录。此时，可以通过继承扩展HandleErrorAttribute来实现新的Filter,然后在lobal.ascx中对所有的Action进行注册，filters.Add(new MyHandleErrorAttribute()).在这里详细说明一下另一种方式。我们可以写一个专门的Controller和Action来进行异常发生时的路由设置.

**[csharp]** [view plaincopy](http://blog.csdn.net/sundacheng1989/article/details/9000596)

1. **public** **class** UtilityController : Controller
2. {
3. **public** ActionResult Error()
4. {
5. **return** View();
6. }
8. **public** ActionResult PageNotFound()
9. {
10. **return** View();
11. }
12. }

当出现异常的时候，把异常抛到最顶端，由Application\_Error统一处理。这里的统一处理就包括，记录日志，重新进行页面定向等。

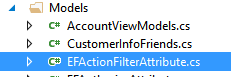
**[csharp]** [view plaincopy](http://blog.csdn.net/sundacheng1989/article/details/9000596)

1. **protected** **void** Application\_Error(**object** sender, EventArgs e)
2. {
3. var ex = Server.GetLastError();
4. Log.Error(ex); //记录日志信息
5. var httpStatusCode = (ex **is** HttpException) ? (ex **as** HttpException).GetHttpCode() : 500; //这里仅仅区分两种错误
6. var httpContext = ((MvcApplication)sender).Context;
7. httpContext.ClearError();
8. httpContext.Response.Clear();
9. httpContext.Response.StatusCode = httpStatusCode;
10. var shouldHandleException = **true**;
11. HandleErrorInfo errorModel;
13. var routeData = **new** RouteData();
14. routeData.Values["controller"] = "Utility";
16. **switch** (httpStatusCode)
17. {
18. **case** 404:
19. routeData.Values["action"] = "PageNotFound";
20. errorModel = **new** HandleErrorInfo(**new** Exception(**string**.Format("No page Found", httpContext.Request.UrlReferrer), ex), "Utility", "PageNotFound");
21. **break**;
23. **default**:
24. routeData.Values["action"] = "Error";
25. Exception exceptionToReplace = **null**; //这里使用了EntLib的异常处理模块的一些功能
26. shouldHandleException = ExceptionPolicy.HandleException(ex, "LogAndReplace", **out** exceptionToReplace);
27. errorModel = **new** HandleErrorInfo(exceptionToReplace, "Utility", "Error");
28. **break**;
29. }
31. **if** (shouldHandleException)
32. {
33. var controller = **new** UtilityController();
34. controller.ViewData.Model = errorModel; //通过代码路由到指定的路径
35. ((IController)controller).Execute(**new** RequestContext(**new** HttpContextWrapper(httpContext), routeData));
36. }
37. }
38. }

## 行为过滤器和结果过滤器

行为即逻辑处理的过程；结果即构造输出字符串的过程

### 创建类



### 继承并重写方法

|  |
| --- |
| public class EFActionFilterAttribute:ActionFilterAttribute  {  //行为执行前  public override void OnActionExecuting(ActionExecutingContext filterContext)  {  base.OnActionExecuting(filterContext);  }  //行为执行前  public override void OnActionExecuted(ActionExecutedContext filterContext)  {  base.OnActionExecuted(filterContext);  }  //结果执行前，既输出前  public override void OnResultExecuting(ResultExecutingContext filterContext)  {  base.OnResultExecuting(filterContext);  }  //结果输出后，即输出字符串后执行。  public override void OnResultExecuted(ResultExecutedContext filterContext)  {  base.OnResultExecuted(filterContext);  }  } |

### 使用特性

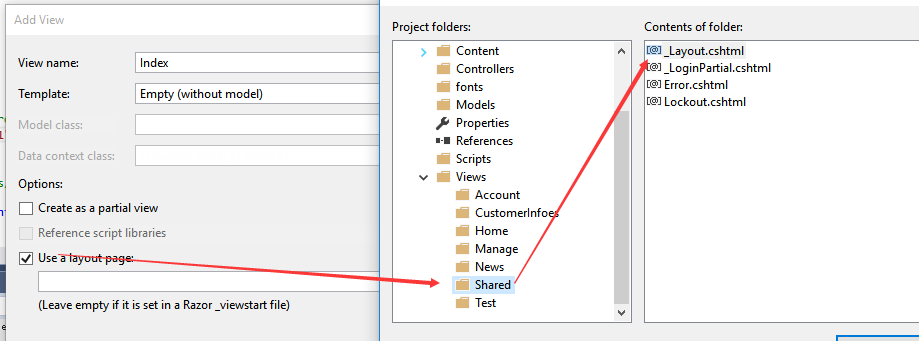
在行为前或控制器前引入特性即可。

# 布局

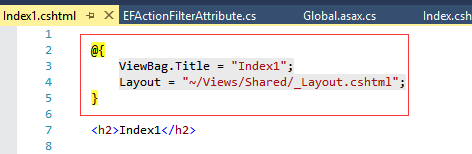
模板页在MVC中叫布局。使用模板，就是做好一个页面，在里面空出来区域，以便在子页面中填充，快速完成页面开发。

一把不用，一般是创建css文件,使用时引入该文件css，实现布局

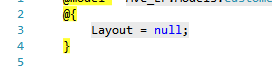
## 创建带布局的视图



视图原型如下



如果以后不想使用模板，则可进行如下设置：



## 查看模板

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>@ViewBag.Title - My ASP.NET Application</title>  @Styles.Render("~/Content/css")  @Scripts.Render("~/bundles/modernizr")  </head>  <body>  <div class="navbar navbar-inverse navbar-fixed-top">  <div class="container">  <div class="navbar-header">  <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">  <span class="icon-bar"></span>  <span class="icon-bar"></span>  <span class="icon-bar"></span>  </button>  @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })  </div>  <div class="navbar-collapse collapse">  <ul class="nav navbar-nav">  <li>@Html.ActionLink("Home", "Index", "Home")</li>  <li>@Html.ActionLink("About", "About", "Home")</li>  <li>@Html.ActionLink("Contact", "Contact", "Home")</li>  </ul>  @Html.Partial("\_LoginPartial")  </div>  </div>  </div>  <div class="container body-content">  @RenderBody()  <hr />  <footer>  <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>  </footer>  </div>  @Scripts.Render("~/bundles/jquery")  @Scripts.Render("~/bundles/bootstrap")  @RenderSection("scripts", required: false)  </body>  </html> |

在视图中输入的代码将默认代替@RenderBody()。

### 自定义替换位置

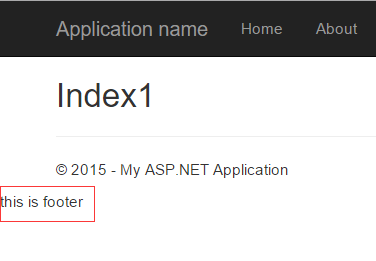
例如，我们在模板的最下面添加如下代码：

|  |
| --- |
| @RenderSection("Footer")  </body> |

在视图中：

|  |
| --- |
| @section Footer  {  <p>this is footer</p>  } |

查看网页



# MVC原理

1. 客户端发一个请求过来。
2. 服务器内核模块的HTTP.SYS接收请求，并放到消息队列中。
3. HTTP.SYS通知IIS中的inetInfo，inetInfo查询IIS配置信息，从连接池中到可用工作进程，最后通知工作进程去消息队列中拿消息。
4. 工作进程中的非托管的asp.net\_isapi从消息队列中拿到请求报文，并对请求报文进行封装得到一个ecb句柄。
5. asp.net\_isapi另一个任务是调用托管模块ISPI\_Runtime的pr方法。pr方法将ecb句柄作为参数传递过来做进一步封装得到HttpWorkerRequest对象。
6. 调用HttpRuntime的PR方法，一方面将HttpWorkerRequest对象封装为HttpContext对象，另一方面通过对象池来技术来获取Application对象，如果池中有对象则返回，没有则创建。在创建时: 1.确保init方法执行，2.确保Start方法被调用，3. 创建对象，初始化内部模块。
7. 获得Application对象后，开始管道执行过程。

## Application请求管道中的19个事件

(1)BeginRequest: 开始处理请求

(2)AuthenticateRequest授权验证请求，获取用户授权信息

(3):PostAuthenticateRequest获取成功

(4): AunthorizeRequest 授权，一般来检查用户是否获得权限

(5):PostAuthorizeRequest:获得授权

(6):ResolveRequestCache:获取页面缓存结果

(7):PostResolveRequestCache 已获取缓存

(8):PostMapRequestHandler 创建页面对象:创建 最终处理当前http请求的 Handler 实例： 第一从HttpContext中获取当前的PR Handler ，Create

(9):PostAcquireRequestState 获取Session

(10)PostAcquireRequestState 获得Session

(11)PreRequestHandlerExecute:准备执行页面对象

执行页面对象的ProcessRequest方法

(12)PostRequestHandlerExecute 执行完页面对象了

(13)ReleaseRequestState 释放请求状态

(14)PostReleaseRequestState 已释放请求状态

(15)UpdateRequestCache 更新缓存

(16)PostUpdateRequestCache 已更新缓存

(17)LogRequest 日志记录

(18)PostLogRequest 已完成日志

(19)EndRequest 完成

## 第7、8个事件之间

获取HttpHandler，根据请求的不同执行不同的方法。如果是aspx则指向page类的PR方法；如果是ashx则指向ashx类的pr方法。

## 第11、12个事件之间

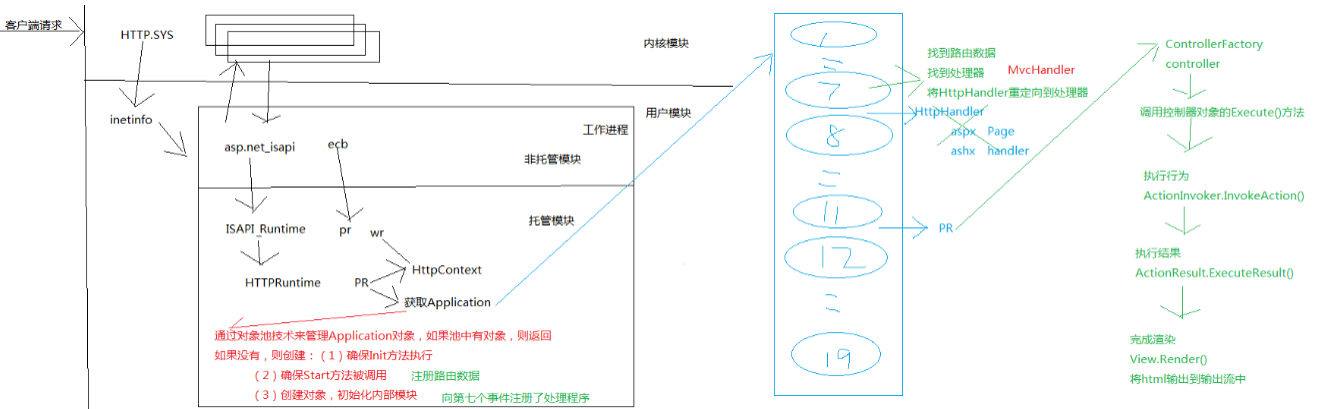
执行PR方法。如果是page类，走页面生命周期。对于MVC，没有页面生命周期

## MVC执行过程

在创建Application对象时，Start方法中会注册路由数据。在初始化模块时有一个模块叫URLRoutingModule，它的Init方法向第七个事件注册了处理程序。

在第7个事件时，执行处理程序。1.找到路由数据，2.找到处理器，3.将HttpHandler重定向到处理器。第7、8事件之间获取HttpHandler的程序不再执行。

在第11、12个事件之间时，通过ControllerFacotry获取控制器对象，之后调用对象的Execute方法，在方法内部执行行为（ActionInvoker.InvokeAction()），执行结果（ActionResult.ExecuteResult()），得到html标签完成渲染（View.Render()）将html输出到输出流中。



1. 本质上数据漕是键值对集合，它包含的对象是唯一的、本地的（内存中的），独享的（其他线程无法访问）。定义:线程内独占的实例唯一的键值对集合 [↑](#footnote-ref-1)
2. 软件架构模式描述软件系统里的基本的结构组织或纲要。架构模式提供一些事先定义好的子系统，指定它们的责任，并给出把它们组织在一起的法则和指南。称之为系统模式 [↑](#footnote-ref-2)
3. 更好的组织项目开发过程中各种人员的分工 [↑](#footnote-ref-3)
4. 模块是指整个系统中一些相对对独立的程序单元，每个程序单元完成和实现一个相对独立的软件功能。通俗点就是一些独立的程序段。 [↑](#footnote-ref-4)
5. 组件化程序设计强调真正的软件可重用性和高度的互操作性。它侧重于组件的产生和装配，这两方面一起构成了组件化程序设计的核心；组件化的目的是为了模块化程序，使得各个模块可以单独开发、单独测试。翻译成设计语言，可以理解为，设计模块化，控件可复用。 [↑](#footnote-ref-5)
6. 方法与行为的不同在于职责（职能）不同：对于运行用户作为资源请求的的方法，我们称为行为。对于进行代码封装的方法，我们称为方法。 [↑](#footnote-ref-6)