**[Util应用程序框架公共操作类(七):Lambda表达式公共操作类](http://www.cnblogs.com/xiadao521/p/4182266.html)**

　　前一篇扩展了两个常用验证方法，本文将封装两个Lambda表达式操作，用来为下一篇的查询扩展服务。

　　Lambda表达式是一种简洁的匿名函数语法，可以用它将方法作为委托参数传递。在Linq中，大量使用Lambda表达式进行查询，不过这种Lambda表达式被Expression包装成表达式树。表达式树是解释器的一个实现，它的作用是将一种语法转换为另一种语法，比如将Lambda表达式解析为Sql语句。

　　使用Sql列名进行查询的主要问题是，列名是一个字符串，没有智能提示，如果输入错误，也没有编译时检查。使用Lambda表达式查询可以解决这些问题，这是使用强类型的主要好处，另外当列名与属性名不一致时，只需修改映射配置，业务代码不动，从而增强了系统的扩展性。

　　Lambda表达式的强类型在带来诸多好处的同时，也产生了一些问题，比如t => t.Name==”a”这个表达式，如果想把值”a”拿出来进行操作，怎么做到？值”a”对于查询条件来讲，是一个动态传入的参数，如果对表达式树完全不了解，这也不是一件轻松的事。另外还有动态查询的问题，这时候开始怀念弱类型的字符串了，微软提供了一个动态查询的辅助类来解决这个问题，待用到的时候我再介绍。

　　本文介绍的一个方法是GetValue，用来将Lambda谓词表达式中的值取出来，另一个方法是GetCriteriaCount，用来判断Lambda谓词表达式中条件的个数。这两个方法的具体应用将在下一篇介绍。

　　在Util项目中添加一个**Lambda**类，代码如下。

[复制代码](javascript:void(0);)

using System.Linq;

using System.Linq.Expressions;

namespace Util {

/// <summary>

/// Lambda表达式操作

/// </summary>

public class Lambda {

#region GetValue(获取值)

/// <summary>

/// 获取值,范例：t => t.Name == "A",返回 A

/// </summary>

/// <param name="expression">表达式,范例：t => t.Name == "A"</param>

public static object GetValue( LambdaExpression expression ) {

if ( expression == null )

return null;

BinaryExpression binaryExpression = GetBinaryExpression( expression );

if ( binaryExpression != null )

return GetBinaryValue( binaryExpression );

var callExpression = expression.Body as MethodCallExpression;

if ( callExpression != null )

return GetMethodValue( callExpression );

return null;

}

/// <summary>

/// 获取二元表达式

/// </summary>

private static BinaryExpression GetBinaryExpression( LambdaExpression expression ) {

var binaryExpression = expression.Body as BinaryExpression;

if ( binaryExpression != null )

return binaryExpression;

var unaryExpression = expression.Body as UnaryExpression;

if ( unaryExpression == null )

return null;

return unaryExpression.Operand as BinaryExpression;

}

/// <summary>

/// 获取二元表达式的值

/// </summary>

private static object GetBinaryValue( BinaryExpression binaryExpression ) {

var unaryExpression = binaryExpression.Right as UnaryExpression;

if ( unaryExpression != null )

return GetConstantValue( unaryExpression.Operand );

return GetConstantValue( binaryExpression.Right );

}

/// <summary>

/// 获取常量值

/// </summary>

private static object GetConstantValue( Expression expression ) {

var constantExpression = expression as ConstantExpression;

if ( constantExpression == null )

return null;

return constantExpression.Value;

}

/// <summary>

/// 获取方法调用表达式的值

/// </summary>

private static object GetMethodValue( MethodCallExpression callExpression ) {

var argumentExpression = callExpression.Arguments.FirstOrDefault();

return GetConstantValue( argumentExpression );

}

#endregion

#region GetCriteriaCount(获取谓词条件的个数)

/// <summary>

/// 获取谓词条件的个数

/// </summary>

/// <param name="expression">谓词表达式,范例：t => t.Name == "A"</param>

public static int GetCriteriaCount( LambdaExpression expression ) {

if ( expression == null )

return 0;

var result = expression.ToString().Replace( "AndAlso", "|" ).Replace( "OrElse", "|" );

return result.Split( '|' ).Count();

}

#endregion

}

}

[复制代码](javascript:void(0);)

    为了进行测试，需要创建一个测试样例类**Test**，代码如下。

[复制代码](javascript:void(0);)

namespace Util.Tests.Samples {

/// <summary>

/// 测试

/// </summary>

public class Test {

public string Name { get; set; }

public int Age { get; set; }

public int? NullableInt { get; set; }

public decimal? NullableDecimal { get; set; }

public TestA A { get; set; }

public class TestA {

public int Integer { get; set; }

public string Address { get; set; }

public TestB B { get; set; }

public class TestB {

public string Name { get; set; }

}

}

}

}

[复制代码](javascript:void(0);)

    单元测试代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Util.Tests.Samples;

namespace Util.Tests {

/// <summary>

/// Lambda表达式操作测试

/// </summary>

[TestClass]

public class LambdaTest {

#region GetValue(获取成员值)

/// <summary>

/// 获取成员值,委托返回类型为Object

/// </summary>

[TestMethod]

public void TestGetValue\_Object() {

Expression<Func<Test, object>> expression = test => test.Name == "A";

Assert.AreEqual( "A", Lambda.GetValue( expression ) );

}

/// <summary>

/// 获取成员值,委托返回类型为bool

/// </summary>

[TestMethod]

public void TestGetValue\_Boolean() {

//空值返回null

Assert.AreEqual( null, Lambda.GetValue( null ) );

//一级返回值

Expression<Func<Test, bool>> expression = test => test.Name == "A";

Assert.AreEqual( "A", Lambda.GetValue( expression ) );

//二级返回值

Expression<Func<Test, bool>> expression2 = test => test.A.Integer == 1;

Assert.AreEqual( 1, Lambda.GetValue( expression2 ) );

//三级返回值

Expression<Func<Test, bool>> expression3 = test => test.A.B.Name == "B";

Assert.AreEqual( "B", Lambda.GetValue( expression3 ) );

}

/// <summary>

/// 获取可空类型的值

/// </summary>

[TestMethod]

public void TestGetValue\_Nullable() {

//可空整型

Expression<Func<Test, bool>> expression = test => test.NullableInt == 1;

Assert.AreEqual( 1, Lambda.GetValue( expression ) );

//可空decimal

expression = test => test.NullableDecimal == 1.5M;

Assert.AreEqual( 1.5M, Lambda.GetValue( expression ) );

}

/// <summary>

/// 获取成员值，运算符为方法

/// </summary>

[TestMethod]

public void TestGetValue\_Method() {

//1级返回值

Expression<Func<Test, bool>> expression = t => t.Name.Contains( "A" );

Assert.AreEqual( "A", Lambda.GetValue( expression ) );

//二级返回值

expression = t => t.A.Address.Contains( "B" );

Assert.AreEqual( "B", Lambda.GetValue( expression ) );

//三级返回值

expression = t => t.A.B.Name.StartsWith( "C" );

Assert.AreEqual( "C", Lambda.GetValue( expression ) );

}

#endregion

#region GetCriteriaCount(获取谓词条件的个数)

/// <summary>

/// 获取谓词条件的个数

/// </summary>

[TestMethod]

public void TestGetCriteriaCount() {

//0个条件

Assert.AreEqual( 0, Lambda.GetCriteriaCount( null ) );

//1个条件

Expression<Func<Test, bool>> expression = test => test.Name == "A";

Assert.AreEqual( 1, Lambda.GetCriteriaCount( expression ) );

//2个条件，与连接符

expression = test => test.Name == "A" && test.Name == "B";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

//2个条件，或连接符

expression = test => test.Name == "A" || test.Name == "B";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

//3个条件

expression = test => test.Name == "A" && test.Name == "B" || test.Name == "C";

Assert.AreEqual( 3, Lambda.GetCriteriaCount( expression ) );

//3个条件,包括导航属性

expression = test => test.A.Address == "A" && test.Name == "B" || test.Name == "C";

Assert.AreEqual( 3, Lambda.GetCriteriaCount( expression ) );

}

/// <summary>

/// 获取谓词条件的个数，运算符为方法

/// </summary>

[TestMethod]

public void TestGetCriteriaCount\_Method() {

//1个条件

Expression<Func<Test, bool>> expression = t => t.Name.Contains( "A" );

Assert.AreEqual( 1, Lambda.GetCriteriaCount( expression ) );

//2个条件,与连接

expression = t => t.Name.Contains( "A" ) && t.Name == "A";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

//2个条件,或连接,包含导航属性

expression = t => t.Name.Contains( "A" ) || t.A.Address == "A";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

}

#endregion

}

}

[复制代码](javascript:void(0);)

　　 需要注意的是，GetValue方法不仅要能获取t=>t.Name==”a”这样的二元表达式，还要能获取方法调用表达式中的值，比如t=>t.Name.Contains(“a”)。

　　下面再增加一个扩展方法，在Util项目中添加名为**Extensions.Expression**的文件，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

namespace Util {

/// <summary>

/// 表达式扩展

/// </summary>

public static partial class Extensions {

#region Value(获取lambda表达式的值)

/// <summary>

/// 获取lambda表达式的值

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

public static object Value<T>( this Expression<Func<T, bool>> expression ) {

return Lambda.GetValue( expression );

}

#endregion

}

}

[复制代码](javascript:void(0);)

 　　Lambda表达式不仅在查询上大展身手，而且在表现层，比如Mvc上也有大量的应用。本文只介绍下一篇基础查询扩展需要用到的两个方法，其它方法我会在需要用到的时候补充进来。

[**应用程序框架实战二十三:基础查询扩展**](http://www.cnblogs.com/xiadao521/p/4184439.html)

　　上面两篇已经作好准备，本文将进行基础查询扩展。当使用了Entity Framework这样的ORM框架以后，我们查询的核心被集中在IQueryable的Where方法上。

　　如果UI需要通过姓名查询一个客户，会在UI上放置一个输入框作为客户姓名的查询条件。服务端接收以后通过Where方法进行过滤，如下所示，entities表示DbContext的子类。

var queryable = entities.Customers.Where( t => t.Name == name );

　　当然，也可以使用Linq语句来完成。

var queryable = from c in entities.Customers

where c.Name == name

select c;

　　这些代码看上去很不错，但不论是上面的扩展方法还是Linq语句，其结果都是错的。如果操作人员正好在查询条件的框中输入了一个“张三”，确实会把名称为“张三”的客户全部找出来，但是如果操作人员什么也不输入，直接点击查询按钮，结果会怎样？

　　上面的代码会强制引入查询条件，哪怕输入值是空的，这与我们的预期不符，所以大家的办法是添加一个判断，像下面这样。

IQueryable<Customer> queryable = entities.Customers;

if( name != "" )

queryable = queryable.Where( t => t.Name == name );

　　将输入值与""进行比较并不健壮，如果操作人员在某个查询条件输入框中不小心打了个空格，依然会引入错误查询条件，所以你把代码改造为下面这样。

IQueryable<Customer> queryable = entities.Customers;

if(!string.IsNullOrWhiteSpace( name ) )

queryable = queryable.Where( t => t.Name == name );

　　但是string.IsNullOrWhiteSpace只能针对字符串，对于其它类型需要先调用ToString，代码继续修改。

IQueryable<Customer> queryable = entities.Customers;

if( value != null && !string.IsNullOrWhiteSpace(value.ToString() ) )

queryable = queryable.Where( t => t.XXX == value );

　　对于非字符串类型的查询条件，为了保障ToString的安全，需要在之前判断是否为null，否则可能抛出null异常。上面的代码比较健壮了，但是非常丑陋，如果只有一个查询条件，这不是大问题，但有10个条件呢？

[复制代码](javascript:void(0);)

IQueryable<Customer> queryable = entities.Customers;

if( value1 != null && !string.IsNullOrWhiteSpace(value1.ToString() ) )

queryable = queryable.Where( t => t.F1 == value1 );

if( value2 != null && !string.IsNullOrWhiteSpace(value2.ToString() ) )

queryable = queryable.Where( t => t.F2 == value2 );

if( value3 != null && !string.IsNullOrWhiteSpace(value3.ToString() ) )

queryable = queryable.Where( t => t.F3 == value3 );

......

[复制代码](javascript:void(0);)

　　打开你自己的项目来检查一下，应该和上面代码类似，这些杂乱无章的判断把查询的主题冲淡了。

　　我上面讨论的是相等（==）运算符，对于像Contains这样的Like查询，它不害怕空字符串“”，但是如果字符串中带了空格“   ”，查询结果也是错的。可见，Where这个核心查询方法，并不适合直接在应用程序中使用，除非你的查询条件是必填项。对于从界面传过来的查询条件基本都是可选的，所以我们有必要进行查询扩展。

　　以上介绍了扩展Where方法的动机，下面开始进行扩展。

　　通过上面的示例代码可以看出，每当需要调用where时，都需要进行一个判断，我们的目标就是把这个判断隐藏到框架背后。

　　首先考虑过滤方法的名称，我命名为**Filter**，表示这是一个过滤器方法。

　　再考虑Filter的方法签名，很显然返回类型是泛型的IQueryable<>，那么参数呢？

　　我最初的做法是提供两个参数，第一个参数是Lambda表达式，第二个参数是查询条件的输入值。之所以需要第二个参数，是因为我当时不清楚怎么从Lambda表达式中把输入值提取出来，方法如下所示。

[复制代码](javascript:void(0);)

/// <summary>

/// 过滤

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TMember">实体属性类型</typeparam>

/// <param name=" queryable">查询对象</param>

/// <param name="predicate">过滤条件</param>

/// <param name="value">属性值</param>

public static IQueryable<TEntity> Filter<TEntity, TMember>( this IQueryable<TEntity> queryable, Expression<Func<TEntity, bool>> predicate, TMember value ){

　　　　 if (value == null)

return queryable;

if (string.IsNullOrWhiteSpace(value.ToString()))

return queryable;

return queryable.Where( predicate );

}

[复制代码](javascript:void(0);)

　　调用代码如下。

IQueryable<Customer> queryable = entities.Customers;

queryable = queryable.Filter( t => t.F1 == value1, value1 ).Filter( t => t.F2 == value2, value2 ).Filter( t => t.F3 == value3, value3 );

　　可以看到，调用代码比直接使用Where已经清爽多了，不过这个Filter不是完美的，对于值类型的输入条件，结果是错的。比如value1是一个int类型，它的默认值为0，它将逃过string.IsNullOrWhiteSpace的检测。那么我们添加一个条件来检测默认值好不好呢，比如if(value == default(TMember)) return; 。这是不行的，如果你要搜索某字段为0的记录就会失效。

　　导致这个问题的原因是值类型无法为空，对引用类型没有影响，我的解决方案是强制使用可空值类型。对于查询来讲，一般不会直接传递一个条件参数，因为大部分UI都要求分页，传递多个参数是不方便的。我通过创建一个**查询实体**来强制实施上面的原则，查询实体拥有一些查询属性，且每个属性都是可空的，并且会帮我过滤掉字符串参数中的空格，待我介绍到应用层的时候再详细说明。

　　无独有偶，我在园子里看到一篇文章和我上面的查询扩展非常类似，只是他的第二个参数用了bool类型。使用bool类型的好处是更加灵活，当然代价是需要写更多代码。调用代码如下所示。

IQueryable<Customer> queryable = entities.Customers;

queryable = queryable.Filter( t => t.F1 == value1, !string.IsNullOrWhiteSpace(value1)).Filter( t => t.F2 == value2, value2 != 0 );

　　在长时间使用了两个参数的方案后，我感觉非常别扭，我为什么要传入第二个值？直接从Lambda参数中提取出输入值不是更好？下面我们说干就干。

[复制代码](javascript:void(0);)

public static IQueryable<T> Filter<T>( this IQueryable<T> queryable, Expression<Func<T, bool>> predicate ) {

if ( predicate.**Value()** == null )

return queryable;

if ( string.IsNullOrWhiteSpace( predicate.**Value()**.ToString() ) )

return queryable;

return queryable.Where( predicate );

　　　　}

[复制代码](javascript:void(0);)

　　这里的关键方法是**Value**，这个自定义方法是上一篇扩展的，它能够从Lambda谓词表达式中把输入值提取出来。

　　这个方案与我之前使用的方案类似，只是省下一个参数，它同样需要使用可空值类型。

　　目前的代码还有一个问题，如果程序员一次传入多个条件，会导致什么结果？

IQueryable<Customer> queryable = entities.Customers;

queryable = queryable.Filter( t => t.F1 == value1 && t.F2 == value2 && t.F3 == value3 )

　　如果value1=”a”,value2和value3是空值，我得把t.F1 == value1拆出来，再传到where中去。当然是可以做到，但太费力，所以我想了个偷懒的方法，一次只允许传递一个条件，一次传入多个条件将抛出异常。

[复制代码](javascript:void(0);)

public static IQueryable<T> Filter<T>( this IQueryable<T> queryable, Expression<Func<T, bool>> predicate ) {

**if ( Lambda.GetCriteriaCount( predicate ) > 1 )**

throw new InvalidOperationException( String.Format( "仅允许添加一个条件,条件：{0}", predicate ) );

if ( predicate.Value() == null )

return queryable;

if ( string.IsNullOrWhiteSpace( predicate.Value().ToString() ) )

return queryable;

return queryable.Where( predicate );

}

[复制代码](javascript:void(0);)

**GetCriteriaCount**是我在上一篇创建的第二个方法，用来获取Lambda谓词表达式中的条件个数，只要大于1个，就会抛出InvalidOperationException异常。

　　为了保证程序员不会把null传进来，添加一个null检测。

[复制代码](javascript:void(0);)

public static IQueryable<T> Filter<T>( this IQueryable<T> queryable, Expression<Func<T, bool>> predicate ) {

**predicate.CheckNull( "predicate" );**

if ( Lambda.GetCriteriaCount( predicate ) > 1 )

throw new InvalidOperationException( String.Format( "仅允许添加一个条件,条件：{0}", predicate ) );

if ( predicate.Value() == null )

return queryable;

if ( string.IsNullOrWhiteSpace( predicate.Value().ToString() ) )

return queryable;

return queryable.Where( predicate );

}

[复制代码](javascript:void(0);)

**CheckNull**用于检测对象是否空值，如果为null将抛出异常。

　　上面介绍了Filter方法的封装过程，现在开始扩展Util应用程序框架。

　　创建一个名为**Util.Datas**的类库，并添加相关依赖，这个项目用于放置数据相关公共操作。创建**Extensions.Query**.cs文件，它用来对查询进行扩展，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq;

using System.Linq.Expressions;

using Util.Datas.Queries;

namespace Util.Datas {

/// <summary>

/// 查询扩展

/// </summary>

public static class Extensions {

/// <summary>

/// 过滤

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="queryable">查询对象</param>

/// <param name="predicate">谓词</param>

public static IQueryable<T> Filter<T>( this IQueryable<T> queryable, Expression<Func<T, bool>> predicate ) {

predicate = QueryHelper.ValidatePredicate( predicate );

if ( predicate == null )

return queryable;

return queryable.Where( predicate );

}

}

}

[复制代码](javascript:void(0);)

　　检测代码移到一个名为**QueryHelper**的internal类中，因为我后面还需要用到这段逻辑，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

namespace Util.Datas.Queries {

/// <summary>

/// 查询操作

/// </summary>

internal class QueryHelper {

/// <summary>

/// 验证谓词，无效返回null

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="predicate">谓词</param>

public static Expression<Func<T, bool>> ValidatePredicate<T>( Expression<Func<T, bool>> predicate ) {

predicate.CheckNull( "predicate" );

if ( Lambda.GetCriteriaCount( predicate ) > 1 )

throw new InvalidOperationException( String.Format( "仅允许添加一个条件,条件：{0}", predicate ) );

if ( predicate.Value() == null )

return null;

if ( string.IsNullOrWhiteSpace( predicate.Value().ToString() ) )

return null;

return predicate;

}

}

}

[复制代码](javascript:void(0);)

　　为了让大家可以把Demo运行起来，我还创建了**Util.Datas.Ef.Tests**测试项目，SqlScripts目录中的Test.sql用来建库，数据库名为UnitTest，之所以不使用Test，是害怕把你本地的Test数据库给删掉了，这个数据库安装在你的**D:\Data**目录中，如果不合适请自行修改。

　　Samples目录中的Employee类是测试的实体，它非常简单，只有一个Name属性。

　　Repositories目录中的EmployeeRepository是测试仓储，为了简单，没有创建仓储的接口，因为这里没什么用。

　　本文的集成测试FilterTest位于QueryTests目录，代码如下。

[复制代码](javascript:void(0);)

using System.Linq;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Util.Datas.Ef.Tests.Repositories;

using Util.Datas.Ef.Tests.Samples;

namespace Util.Datas.Ef.Tests.QueryTests {

/// <summary>

/// 过滤测试

/// </summary>

[TestClass]

public class FilterTest {

/// <summary>

/// 测试初始化

/// </summary>

[TestInitialize]

public void TestInit() {

EmployeeRepository repository = GetEmployeeRepository();

repository.Clear();

repository.Add( Employee.GetEmployee() );

repository.Add( Employee.GetEmployee2() );

}

/// <summary>

/// 获取员工仓储

/// </summary>

private EmployeeRepository GetEmployeeRepository() {

return new EmployeeRepository( new TestUnitOfWork() );

}

/// <summary>

/// 测试Filter过滤

/// </summary>

[TestMethod]

public void TestFilter() {

EmployeeRepository repository = GetEmployeeRepository();

//用where查询

var result = repository.Find().Where( t => t.Name == "" );

Assert.AreEqual( 0, result.Count() );

//用Fileter查询

result = repository.Find().Filter( t => t.Name == "" );

Assert.AreEqual( 2, result.Count() );

Assert.AreEqual( Employee.GetEmployee().Name, result.ToList()[0].Name );

Assert.AreEqual( Employee.GetEmployee2().Name, result.ToList()[1].Name );

}

}

}

[复制代码](javascript:void(0);)

　　我在测试中比较了Where与Filter的不同，你可以自己运行一下，如果还不知道如何运行测试，请参考[Util应用程序框架公共操作类(二):数据类型转换公共操作类（源码篇）](http://www.cnblogs.com/xiadao521/p/4092846.html)。

　　当然使用Where查询比较死板，你需要在编译时期固定查询字段和操作符，这对于某些需要更灵活的场景并不合适，不过一般的系统对查询灵活性要求都不高。

　　本文虽然是针对IQueryable进行扩展，但思路上对于更原始的Ado.Net直接操作Sql同样适用。可以看出，.Net Framework给你提供的API比较原始，如果需要满足自己的需求，就需要扩展你的应用程序框架。另外不要轻视这个小小的扩展和封装，因为你的大多业务都需要查询，如果你有100个模块，每个模块有5个查询条件，能帮你省下500个判断。判断语句不仅枯燥而且容易喧宾夺主，扰乱你的查询主题。

[**应用程序框架实战二十四:基础查询扩展 - 分页与排序**](http://www.cnblogs.com/xiadao521/p/4199953.html)

　　上一篇介绍了IQueryable的Where方法存在的问题，并扩展了一个名为Filter的过滤方法，它是Where方法的增强版。本篇将介绍查询的另一个重要主题——分页与排序。

　　对于任何一个信息系统，查询都需要分页，因为不可能直接返回表中的所有数据。

　　如果直接使用原始的Ado.Net，我们可以编写一个通用分页存储过程来进行分页查询，然后通过一个DataTable返回给业务层。不过进入Entity Framework时代，分页变得异常简单，通过Skip和Take两个方法配合就可以完成任务。

　　为了让分页查询变得更加简单，我们需要进一步扩展和封装。

　　先考虑输入参数，表现层需要将一些分页参数传递到应用层，为此我们可以定义一个分页对象来承载和计算分页相关的数据。

　　在Util.Domains项目的Repositories目录中，创建IPager接口和它的实现类Pager。

**IPager**接口代码如下。

[复制代码](javascript:void(0);)

namespace Util.Domains.Repositories {

/// <summary>

/// 分页

/// </summary>

public interface IPager {

/// <summary>

/// 页数，即第几页，从1开始

/// </summary>

int Page { get; set; }

/// <summary>

/// 每页显示行数

/// </summary>

int PageSize { get; set; }

/// <summary>

/// 总行数

/// </summary>

int TotalCount { get; set; }

/// <summary>

/// 总页数

/// </summary>

int PageCount { get; }

/// <summary>

/// 跳过的行数

/// </summary>

int SkipCount { get; }

/// <summary>

/// 排序条件

/// </summary>

string Order { get; set; }

}

}

[复制代码](javascript:void(0);)

**Pager**类代码如下。

[复制代码](javascript:void(0);)

namespace Util.Domains.Repositories {

/// <summary>

/// 分页

/// </summary>

public class Pager : IPager {

/// <summary>

/// 初始化分页

/// </summary>

public Pager()

: this( 1 ) {

}

/// <summary>

/// 初始化分页

/// </summary>

/// <param name="page">页索引</param>

/// <param name="pageSize">每页显示行数,默认20</param>

/// <param name="totalCount">总行数</param>

public Pager( int page, int pageSize = 20, int totalCount = 0 ) {

Page = page;

PageSize = pageSize;

TotalCount = totalCount;

}

private int \_pageIndex;

/// <summary>

/// 页索引，即第几页，从1开始

/// </summary>

public int Page {

get {

if ( \_pageIndex <= 0 )

\_pageIndex = 1;

return \_pageIndex;

}

set { \_pageIndex = value; }

}

/// <summary>

/// 每页显示行数

/// </summary>

public int PageSize { get; set; }

/// <summary>

/// 总行数

/// </summary>

public int TotalCount { get; set; }

/// <summary>

/// 总页数

/// </summary>

public int PageCount {

get {

if ( TotalCount == 0 )

return 0;

if ( ( TotalCount % PageSize ) == 0 )

return TotalCount / PageSize;

return ( TotalCount / PageSize ) + 1;

}

}

/// <summary>

/// 跳过的行数

/// </summary>

public int SkipCount {

get {

if ( Page > PageCount )

Page = PageCount;

return PageSize \* ( Page - 1 );

}

}

/// <summary>

/// 排序条件

/// </summary>

public string Order { get; set; }

}

}

[复制代码](javascript:void(0);)

　　我将排序条件Order也打包到IPager接口中，这是因为排序与分页密切相关，甚至在调用Skip方法之前，.Net强制要求设置排序条件。

　　在调用Skip方法时需要计算出跳过的行数，SkipCount提供了这个功能。

　　由于客户端可能传递错误的分页参数，所以需要在Pager中进行修正。

**PagerTest**单元测试代码如下。

[复制代码](javascript:void(0);)

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Util.Domains.Repositories;

namespace Util.Domains.Tests.Repositories {

/// <summary>

/// 分页测试

/// </summary>

[TestClass]

public class PagerTest {

#region 测试初始化

/// <summary>

/// 分页

/// </summary>

private Pager \_pager;

/// <summary>

/// 测试初始化

/// </summary>

[TestInitialize]

public void TestInit() {

\_pager = new Pager();

}

#endregion

#region 默认值

/// <summary>

/// 分页默认值

/// </summary>

[TestMethod]

public void Test\_Default() {

Assert.AreEqual( 1, \_pager.Page );

Assert.AreEqual( 20, \_pager.PageSize );

Assert.AreEqual( 0, \_pager.TotalCount );

Assert.AreEqual( 0, \_pager.PageCount );

}

#endregion

#region PageCount(总页数)

/// <summary>

/// 总行数为0，每页20行，页数为0

/// </summary>

[TestMethod]

public void TestPageCount\_TotalCountIs0() {

\_pager.TotalCount = 0;

Assert.AreEqual( 0, \_pager.PageCount );

}

/// <summary>

/// 总行数为100，每页20行，页数为5

/// </summary>

[TestMethod]

public void TestPageCount\_TotalCountIs100() {

\_pager.TotalCount = 100;

Assert.AreEqual( 5, \_pager.PageCount );

}

/// <summary>

/// 总行数为1，每页20行，页数为1

/// </summary>

[TestMethod]

public void TestPageCount\_TotalCountIs1() {

\_pager.TotalCount = 1;

Assert.AreEqual( 1, \_pager.PageCount );

}

/// <summary>

/// 总行数为100，每页10行，页数为10

/// </summary>

[TestMethod]

public void TestPageCount\_PageSizeIs10\_TotalCountIs100() {

\_pager.PageSize = 10;

\_pager.TotalCount = 100;

Assert.AreEqual( 10, \_pager.PageCount );

}

#endregion

#region Page(页索引)

/// <summary>

/// 页索引小于1，则修正为1

/// </summary>

[TestMethod]

public void TestPage\_Less1() {

\_pager.Page = 0;

Assert.AreEqual( 1, \_pager.Page );

\_pager.Page = -1;

Assert.AreEqual( 1, \_pager.Page );

}

#endregion

#region SkipCount(跳过的行数)

/// <summary>

/// 跳过的行数

/// </summary>

[TestMethod]

public void TestSkipCount() {

\_pager.TotalCount = 100;

\_pager.Page = 0;

Assert.AreEqual( 0, \_pager.SkipCount );

\_pager.Page = 1;

Assert.AreEqual( 0, \_pager.SkipCount );

\_pager.Page = 2;

Assert.AreEqual( 20, \_pager.SkipCount );

\_pager.Page = 3;

Assert.AreEqual( 40, \_pager.SkipCount );

\_pager.Page = 4;

Assert.AreEqual( 60, \_pager.SkipCount );

\_pager.Page = 5;

Assert.AreEqual( 80, \_pager.SkipCount );

\_pager.Page = 6;

Assert.AreEqual( 80, \_pager.SkipCount );

}

/// <summary>

/// 跳过的行数

/// </summary>

[TestMethod]

public void TestSkipCount\_2() {

\_pager.TotalCount = 99;

\_pager.Page = 0;

Assert.AreEqual( 0, \_pager.SkipCount );

\_pager.Page = 1;

Assert.AreEqual( 0, \_pager.SkipCount );

\_pager.Page = 2;

Assert.AreEqual( 20, \_pager.SkipCount );

\_pager.Page = 3;

Assert.AreEqual( 40, \_pager.SkipCount );

\_pager.Page = 4;

Assert.AreEqual( 60, \_pager.SkipCount );

\_pager.Page = 5;

Assert.AreEqual( 80, \_pager.SkipCount );

\_pager.Page = 6;

Assert.AreEqual( 80, \_pager.SkipCount );

}

/// <summary>

/// 跳过的行数

/// </summary>

[TestMethod]

public void TestSkipCount\_3() {

\_pager.TotalCount = 0;

\_pager.Page = 1;

Assert.AreEqual( 0, \_pager.SkipCount );

}

#endregion

}

}

[复制代码](javascript:void(0);)

　　现在有了Pager来传递分页参数，但分页结果采用什么类型返回呢？一种办法是通过List<T>返回对象集合，再定义几个out参数来返回分页参数，但这种做法比较丑陋，out只应该在必要时才使用。

　　一个更好的办法是创建派生自List<T>的自定义集合，只需要添加几个分页属性即可。

　　在Util.Domains项目的Repositories目录中，创建**PagerList**分页列表，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.Linq;

namespace Util.Domains.Repositories {

/// <summary>

/// 分页集合

/// </summary>

/// <typeparam name="T">元素类型</typeparam>

public class PagerList<T> : List<T> {

/// <summary>

/// 分页集合

/// </summary>

/// <param name="pager">查询对象</param>

public PagerList( IPager pager )

: this( pager.Page, pager.PageSize, pager.TotalCount, pager.Order ) {

}

/// <summary>

/// 分页集合

/// </summary>

/// <param name="totalCount">总行数</param>

public PagerList( int totalCount )

: this( 1, 20, totalCount ) {

}

/// <summary>

/// 分页集合

/// </summary>

/// <param name="page">页索引</param>

/// <param name="pageSize">每页显示行数</param>

/// <param name="totalCount">总行数</param>

public PagerList( int page, int pageSize, int totalCount )

: this( page, pageSize, totalCount, "" ) {

}

/// <summary>

/// 分页集合

/// </summary>

/// <param name="page">页索引</param>

/// <param name="pageSize">每页显示行数</param>

/// <param name="totalCount">总行数</param>

/// <param name="order">排序条件</param>

public PagerList( int page, int pageSize, int totalCount, string order ) {

var pager = new Pager( page, pageSize, totalCount );

TotalCount = pager.TotalCount;

PageCount = pager.PageCount;

Page = pager.Page;

PageSize = pager.PageSize;

Order = order;

}

/// <summary>

/// 页索引，即第几页，从1开始

/// </summary>

public int Page { get; private set; }

/// <summary>

/// 每页显示行数

/// </summary>

public int PageSize { get; private set; }

/// <summary>

/// 总行数

/// </summary>

public int TotalCount { get; private set; }

/// <summary>

/// 总页数

/// </summary>

public int PageCount { get; private set; }

/// <summary>

/// 排序条件

/// </summary>

public string Order { get; private set; }

/// <summary>

/// 转换分页集合的元素类型

/// </summary>

/// <typeparam name="TResult">目标元素类型</typeparam>

/// <param name="converter">转换方法</param>

public PagerList<TResult> Convert<TResult>( Func<T, TResult> converter ) {

var result = new PagerList<TResult>( Page, PageSize, TotalCount, Order );

result.AddRange( this.Select( converter ) );

return result;

}

}

}

[复制代码](javascript:void(0);)

　　PagerList可以接收一个IPager的参数，这样可以快速设置分页参数。

　　当你从仓储中获取到PagerList<T>，T类型参数是一个领域层的聚合，如果你的应用层操作的是Dto，这个PagerList就无法使用，将一个PagerList<TEntity>完整转换为PagerList<TDto>需要好几行乏味的赋值代码。为了解决这个问题，提供了一个**Convert**方法，该方法接收一个Func<T, TResult>参数，Func是.Net内置的一个标准委托，我们可以传递一个方法完成Entity到Dto的转换,其它分页参数的赋值操作会在Convert中完成。

**PagerListTest**单元测试代码如下。

[复制代码](javascript:void(0);)

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Util.Domains.Repositories;

using Util.Domains.Tests.Samples;

namespace Util.Domains.Tests.Repositories {

/// <summary>

/// 分页集合测试

/// </summary>

[TestClass]

public class PagerListTest {

/// <summary>

/// 分页集合

/// </summary>

private PagerList<Employee> \_list;

/// <summary>

/// 测试初始化

/// </summary>

[TestInitialize]

public void TestInit() {

\_list = new PagerList<Employee>( 1, 2, 3 );

\_list.Add( new Employee() );

\_list.Add( new Employee(){Name = "B"} );

}

/// <summary>

/// 元素个数

/// </summary>

[TestMethod]

public void TestCount() {

Assert.AreEqual( 2, \_list.Count );

}

/// <summary>

/// 用索引获取元素

/// </summary>

[TestMethod]

public void TestIndex() {

Assert.AreEqual( "B", \_list[1].Name );

}

/// <summary>

/// 转换类型

/// </summary>

[TestMethod]

public void TestConvert() {

var result = \_list.Convert( t => new EmployeeDto() );

Assert.AreEqual( 2, result.Count );

Assert.AreEqual( 1, result.Page );

Assert.AreEqual( 2, result.PageSize );

Assert.AreEqual( 3, result.TotalCount );

Assert.AreEqual( 2, result.PageCount );

}

}

}

[复制代码](javascript:void(0);)

　　准备工作已经就绪，现在开始扩展IQueryable的分页和排序功能。

　　注意观察IPager接口中的排序条件Order，它是一个字符串类型，使用弱类型的字符串是有原因的。要在IQueryable上进行排序，第一次升序调用OrderBy，降序调用OrderByDescending，如果要继续添加第二个排序条件，升序调用ThenBy，降序调用ThenByDescending。可以看到，排序API并不易用，如果要设置多个排序条件相当麻烦。更重要的一点是这些方法的参数是强类型的Func或Expression，而表现层传过来的参数一般都是字符串，这些字符串无法直接传递给上述方法，更不要谈排序方向和多个排序字段。

　　从上面可以看出，弱类型也不是一无是处，它可以提供强大的灵活性。为了弥补Linq强类型查询的不足，微软提供了一组动态查询帮助类，其中**DynamicQueryable**为IQueryable扩展了几个常用方法，它可以接收字符串参数，并解析为相应的Expression。

　　由于这一组帮助类内容很少，所以我不想为此引用一个额外的程序集。我将这些帮助类放到了Util项目的Lambdas目录的Dynamics子目录中，并修改它们的命名空间为Util.Lambdas.Dynamics，这样Resharper就不会显示警告了。

　　这几个动态查询帮助类的代码就不贴了，有兴趣可下载本文的示例代码文件。

　　在Util.Datas项目中找到Extensions.Query.cs文件，添加下面的扩展代码。

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

using Util.Datas.Queries;

using Util.Domains.Repositories;

using Util.Lambdas.Dynamics;

namespace Util.Datas {

/// <summary>

/// 查询扩展

/// </summary>

public static class Extensions {

/// <summary>

/// 过滤

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="predicate">谓词</param>

public static IQueryable<T> Filter<T>( this IQueryable<T> source, Expression<Func<T, bool>> predicate ) {

predicate = QueryHelper.ValidatePredicate( predicate );

if ( predicate == null )

return source;

return source.Where( predicate );

}

/// <summary>

/// 排序

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="propertyName">排序属性名，多个属性用逗号分隔，降序用desc字符串，范例：Name,Age desc</param>

public static IQueryable<T> OrderBy<T>( this IQueryable<T> source, string propertyName ) {

return source.OrderByDynamic( propertyName );

}

/// <summary>

/// 创建分页列表

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="page">页索引，表示第几页，从1开始</param>

/// <param name="pageSize">每页显示行数，默认20</param>

public static PagerList<T> PagerResult<T>( this IQueryable<T> source, int page, int pageSize = 20 ) {

return PagerResult( source, new Pager( page, pageSize ) );

}

/// <summary>

/// 创建分页列表

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="pager">分页对象</param>

public static PagerList<T> PagerResult<T>( this IQueryable<T> source, IPager pager ) {

source = OrderBy( source, pager );

source = Pager( source, pager );

return CreatePageList( source, pager );

}

/// <summary>

/// 排序

/// </summary>

private static IQueryable<T> OrderBy<T>( IQueryable<T> source, IPager pager ) {

if ( pager.Order.IsEmpty() )

return source;

return source.OrderBy( pager.Order );

}

/// <summary>

/// 分页

/// </summary>

private static IQueryable<T> Pager<T>( IQueryable<T> source, IPager pager ) {

if ( pager.TotalCount <= 0 )

pager.TotalCount = source.Count();

return source.Skip( pager.SkipCount ).Take( pager.PageSize );

}

/// <summary>

/// 创建分页列表

/// </summary>

private static PagerList<T> CreatePageList<T>( IEnumerable<T> source, IPager pager ) {

var result = new PagerList<T>( pager );

result.AddRange( source.ToList() );

return result;

}

}

}

[复制代码](javascript:void(0);)

　　这里扩展了**OrderBy**方法，在方法内部委托给**OrderByDynamic**执行，OrderByDynamic方法由DynamicQueryable提供。

　　PagerResult方法用来获取分页结果，有两个重载，第一个重载方法 **PagerList<T> PagerResult<T>( this IQueryable<T> source, int page, int pageSize = 20 )** 接收两个分页参数，在使用这个重载之前假定排序已经完成。另一个重载方法 **PagerList<T> PagerResult<T>( this IQueryable<T> source, IPager pager )**接收一个分页对象，它会同时完成分页和排序操作。

　　我在实际应用中，几乎总是使用第二个重载，因为我在应用层使用了**查询实体**,查询实体是从Pager派生的查询参数对象，待介绍到应用层再详述。

　　还有一点需要注意，Pager对象的**TotalCount**是允许设置的，我在获取总行数的时候作了一个判断，如果TotalCount已经被设置，就不会调用Count方法。这样设计的原因是调用Count方法的开销很高，可能导致表扫描或索引扫描，如果在执行 PagerResult之前已经执行过Count，就不需要再重复执行。

　　本篇介绍的方法，应用层可以这样调用。

var dtos = Repository.Find().Filter( t => t.Name.Contains( "a" ) ).OrderBy( t => t.CreateTime ).PagerResult( 1 ).Convert( t => t.ToDto() );

或

var dtos = Repository.Find().Filter( t => t.Name.Contains( testQuery.Name ) ).PagerResult( testQuery ).Convert( t => t.ToDto() );

　　上面的代码已经比较简单，不过我将查询功能单独提取出来，使用**查询对象**模式进行封装，进一步简化操作。

　　下一篇将介绍查询条件，它是规约模式的一种实现。

[**Util应用程序框架公共操作类(八):Lambda表达式公共操作类(二)**](http://www.cnblogs.com/xiadao521/p/4206696.html)

　　前面介绍了查询的基础扩展，下面准备给大家介绍一些有用的查询封装手法，比如对日期范围查询，数值范围查询的封装等，为了支持这些功能，需要增强公共操作类。

　　Lambda表达式公共操作类，我在前面已经简单介绍过，要么从Lambda表达式中获取信息，要么动态创建Lambda表达式，本篇直接贴代码，如果发现有些代码与以前不同，原因有二，一是之前还未用到，只发了部分代码，二是可能代码已重构。需要说明的一点是，我不会考虑代码兼容性，API命名随时可能修改，如果你直接将本系列的代码用到你的项目上要格外注意。

　　修改Util项目的**Lambda**类，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

using System.Reflection;

using DynamicExpression = Util.Lambdas.Dynamics.DynamicExpression;

namespace Util {

/// <summary>

/// Lambda表达式操作

/// </summary>

public class Lambda {

#region GetName(获取成员名称)

/// <summary>

/// 获取成员名称，范例：t => t.Name,返回 Name

/// </summary>

/// <param name="expression">表达式,范例：t => t.Name</param>

public static string GetName( LambdaExpression expression ) {

var memberExpression = GetMemberExpression( expression );

if ( memberExpression == null )

return string.Empty;

string result = memberExpression.ToString();

return result.Substring( result.IndexOf( ".", StringComparison.Ordinal ) + 1 );

}

/// <summary>

/// 获取成员表达式

/// </summary>

private static MemberExpression GetMemberExpression( LambdaExpression expression ) {

if ( expression == null )

return null;

var unaryExpression = expression.Body as UnaryExpression;

if ( unaryExpression == null )

return expression.Body as MemberExpression;

return unaryExpression.Operand as MemberExpression;

}

#endregion

#region GetMember(获取成员)

/// <summary>

/// 获取成员

/// </summary>

/// <param name="expression">表达式,范例：t => t.Name</param>

public static MemberInfo GetMember( LambdaExpression expression ) {

var memberExpression = GetMemberExpression( expression );

if ( memberExpression == null )

return null;

return memberExpression.Member;

}

#endregion

#region GetValue(获取值)

/// <summary>

/// 获取值,范例：t => t.Name == "A",返回 A

/// </summary>

/// <param name="expression">表达式,范例：t => t.Name == "A"</param>

public static object GetValue( LambdaExpression expression ) {

if ( expression == null )

return null;

var memberExpression = expression.Body as MemberExpression;

if ( memberExpression != null )

return GetMemberValue( memberExpression );

BinaryExpression binaryExpression = GetBinaryExpression( expression );

if ( binaryExpression != null )

return GetBinaryValue( binaryExpression );

var callExpression = expression.Body as MethodCallExpression;

if ( callExpression != null )

return GetMethodValue( callExpression );

return null;

}

/// <summary>

/// 获取二元表达式

/// </summary>

private static BinaryExpression GetBinaryExpression( LambdaExpression expression ) {

var binaryExpression = expression.Body as BinaryExpression;

if ( binaryExpression != null )

return binaryExpression;

var unaryExpression = expression.Body as UnaryExpression;

if ( unaryExpression == null )

return null;

return unaryExpression.Operand as BinaryExpression;

}

/// <summary>

/// 获取二元表达式的值

/// </summary>

private static object GetBinaryValue( BinaryExpression binaryExpression ) {

var unaryExpression = binaryExpression.Right as UnaryExpression;

if ( unaryExpression != null )

return GetConstantValue( unaryExpression.Operand );

var memberExpression = binaryExpression.Right as MemberExpression;

if ( memberExpression != null )

return GetMemberValue( memberExpression );

return GetConstantValue( binaryExpression.Right );

}

/// <summary>

/// 获取属性表达式的值

/// </summary>

private static object GetMemberValue( MemberExpression expression ) {

if ( expression == null )

return null;

var field = expression.Member as FieldInfo;

if ( field != null ) {

var constValue = GetConstantValue( expression.Expression );

return field.GetValue( constValue );

}

var property = expression.Member as PropertyInfo;

if ( property == null )

return null;

var value = GetMemberValue( expression.Expression as MemberExpression );

return property.GetValue( value );

}

/// <summary>

/// 获取常量值

/// </summary>

private static object GetConstantValue( Expression expression ) {

var constantExpression = expression as ConstantExpression;

if ( constantExpression == null )

return null;

return constantExpression.Value;

}

/// <summary>

/// 获取方法调用表达式的值

/// </summary>

private static object GetMethodValue( MethodCallExpression callExpression ) {

var argumentExpression = callExpression.Arguments.FirstOrDefault();

var memberExpression = argumentExpression as MemberExpression;

if ( memberExpression != null )

return GetMemberValue( memberExpression );

return GetConstantValue( argumentExpression );

}

#endregion

#region GetParameter(获取参数)

/// <summary>

/// 获取参数，范例：t.Name,返回 t

/// </summary>

/// <param name="expression">表达式，范例：t.Name</param>

public static ParameterExpression GetParameter( LambdaExpression expression ) {

if ( expression == null )

return null;

BinaryExpression binaryExpression = GetBinaryExpression( expression );

if ( binaryExpression == null )

return null;

return GetParameterByMember( binaryExpression.Left );

}

/// <summary>

/// 递归获取参数

/// </summary>

private static ParameterExpression GetParameterByMember( Expression expression ) {

if ( expression == null )

return null;

ParameterExpression result = expression as ParameterExpression;

if ( result != null )

return result;

MemberExpression memberExpression = expression as MemberExpression;

if ( memberExpression == null )

return null;

return GetParameterByMember( memberExpression.Expression );

}

#endregion

#region GetAttribute(获取特性)

/// <summary>

/// 获取特性

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <typeparam name="TAttribute">特性类型</typeparam>

/// <param name="propertyExpression">属性表达式</param>

public static TAttribute GetAttribute<TEntity, TProperty, TAttribute>( Expression<Func<TEntity, TProperty>> propertyExpression )

where TAttribute : Attribute {

var memberInfo = GetMember( propertyExpression );

return memberInfo.GetCustomAttribute<TAttribute>();

}

#endregion

#region GetAttributes(获取特性列表)

/// <summary>

/// 获取特性列表

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <typeparam name="TAttribute">特性类型</typeparam>

/// <param name="propertyExpression">属性表达式</param>

public static IEnumerable<TAttribute> GetAttributes<TEntity, TProperty, TAttribute>( Expression<Func<TEntity, TProperty>> propertyExpression ) where TAttribute : Attribute {

var memberInfo = GetMember( propertyExpression );

return memberInfo.GetCustomAttributes<TAttribute>();

}

#endregion

#region Constant(获取常量)

/// <summary>

/// 获取常量表达式，自动转换值的类型

/// </summary>

/// <param name="expression">表达式</param>

/// <param name="value">值</param>

public static ConstantExpression Constant( Expression expression, object value ) {

var memberExpression = expression as MemberExpression;

if ( memberExpression == null )

return Expression.Constant( value );

return Expression.Constant( value, memberExpression.Type );

}

#endregion

#region GetCriteriaCount(获取谓词条件的个数)

/// <summary>

/// 获取谓词条件的个数

/// </summary>

/// <param name="expression">谓词表达式,范例：t => t.Name == "A"</param>

public static int GetCriteriaCount( LambdaExpression expression ) {

if ( expression == null )

return 0;

var result = expression.ToString().Replace( "AndAlso", "|" ).Replace( "OrElse", "|" );

return result.Split( '|' ).Count();

}

#endregion

#region Equal(等于表达式)

/// <summary>

/// 创建等于运算lambda表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> Equal<T>( string propertyName, object value ) {

var parameter = CreateParameter<T>();

return parameter.Property( propertyName )

.Equal( value )

.ToLambda<Func<T, bool>>( parameter );

}

/// <summary>

/// 创建参数

/// </summary>

private static ParameterExpression CreateParameter<T>() {

return Expression.Parameter( typeof( T ), "t" );

}

#endregion

#region NotEqual(不等于表达式)

/// <summary>

/// 创建不等于运算lambda表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> NotEqual<T>( string propertyName, object value ) {

var parameter = CreateParameter<T>();

return parameter.Property( propertyName )

.NotEqual( value )

.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region Greater(大于表达式)

/// <summary>

/// 创建大于运算lambda表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> Greater<T>( string propertyName, object value ) {

var parameter = CreateParameter<T>();

return parameter.Property( propertyName )

.Greater( value )

.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region Less(小于表达式)

/// <summary>

/// 创建小于运算lambda表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> Less<T>( string propertyName, object value ) {

var parameter = CreateParameter<T>();

return parameter.Property( propertyName )

.Less( value )

.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region GreaterEqual(大于等于表达式)

/// <summary>

/// 创建大于等于运算lambda表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> GreaterEqual<T>( string propertyName, object value ) {

var parameter = CreateParameter<T>();

return parameter.Property( propertyName )

.GreaterEqual( value )

.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region LessEqual(小于等于表达式)

/// <summary>

/// 创建小于等于运算lambda表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> LessEqual<T>( string propertyName, object value ) {

var parameter = CreateParameter<T>();

return parameter.Property( propertyName )

.LessEqual( value )

.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region Contains(调用Contains方法)

/// <summary>

/// 调用Contains方法

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> Contains<T>( string propertyName, object value ) {

return Call<T>( propertyName, "Contains", value );

}

/// <summary>

/// 调用方法

/// </summary>

private static Expression<Func<T, bool>> Call<T>( string propertyName, string methodName, object value ) {

var parameter = CreateParameter<T>();

return parameter.Property( propertyName )

.Call( methodName, value )

.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region Starts(调用StartsWith方法)

/// <summary>

/// 调用StartsWith方法

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> Starts<T>( string propertyName, string value ) {

var parameter = CreateParameter<T>();

var property = parameter.Property( propertyName );

var call = Expression.Call( property, property.Type.GetMethod( "StartsWith", new Type[] { typeof( string ) } ),

Expression.Constant( value ) );

return call.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region Ends(调用EndsWith方法)

/// <summary>

/// 调用EndsWith方法

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

public static Expression<Func<T, bool>> Ends<T>( string propertyName, string value ) {

var parameter = CreateParameter<T>();

var property = parameter.Property( propertyName );

var call = Expression.Call( property, property.Type.GetMethod( "EndsWith", new Type[] { typeof( string ) } ),

Expression.Constant( value ) );

return call.ToLambda<Func<T, bool>>( parameter );

}

#endregion

#region ParsePredicate(解析为谓词表达式)

/// <summary>

/// 解析为谓词表达式

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

/// <param name="operator">运算符</param>

public static Expression<Func<T, bool>> ParsePredicate<T>( string propertyName, object value, Operator @operator ) {

var parameter = Expression.Parameter( typeof( T ), "t" );

return parameter.Property( propertyName ).Operation( @operator, value ).ToLambda<Func<T, bool>>( parameter );

}

/// <summary>

/// 解析为谓词表达式

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="predicateExpression">谓词表达式字符串,参数占位符为@0,@1,@2 ...</param>

/// <param name="values">值</param>

public static Expression<Func<T, bool>> ParsePredicate<T>( string predicateExpression, params object[] values ) {

return DynamicExpression.ParseLambda( typeof( T ), typeof( bool ), predicateExpression, values ) as Expression<Func<T, bool>>;

}

#endregion

}

}

[复制代码](javascript:void(0);)

　　在Util.Tests单元测试项目新建两个测试样例，代码如下。

[复制代码](javascript:void(0);)

using System.ComponentModel.DataAnnotations;

namespace Util.Tests.Samples {

/// <summary>

/// 测试1

/// </summary>

public class Test1 {

[StringLength( 20, ErrorMessage = "长度不能超过20" )]

[Required( ErrorMessage = "名称不能为空" )]

[Display( Name = "名称" )]

public string Name { get; set; }

public int Age { get; set; }

public int? NullableInt { get; set; }

public decimal? NullableDecimal { get; set; }

public TestA A { get; set; }

public class TestA {

public int Integer { get; set; }

public string Address { get; set; }

public TestB B { get; set; }

public class TestB {

public string Name { get; set; }

}

}

}

}

using System;

using System.ComponentModel.DataAnnotations;

namespace Util.Tests.Samples {

/// <summary>

/// 测试2

/// </summary>

[Serializable]

public class Test2 {

public Test2() {

}

public Test2( string name ) {

Name = name;

}

[Required(ErrorMessage = "名称不能为空")]

public string Name { get; set; }

public int Int { get; set; }

public int? NullableInt { get; set; }

public decimal? NullableDecimal { get; set; }

public decimal Decimal { get; set; }

public TestA A { get; set; }

[Serializable]

public class TestA {

public int Integer { get; set; }

public string Address { get; set; }

public TestB B { get; set; }

public class TestB {

public string Name { get; set; }

}

}

}

}

[复制代码](javascript:void(0);)

　　修改Util.Tests单元测试项目**LambdaTest**单元测试，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Linq.Expressions;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Util.Tests.Samples;

namespace Util.Tests {

/// <summary>

/// 表达式测试

/// </summary>

[TestClass]

public class LambdaTest {

#region GetName(获取成员名称)

/// <summary>

/// 获取成员名称

/// </summary>

[TestMethod]

public void TestGetName() {

//空值返回空字符串

Assert.AreEqual( "", Lambda.GetName( null ) );

//返回一级属性名

Expression<Func<Test1, string>> expression = test => test.Name;

Assert.AreEqual( "Name", Lambda.GetName( expression ) );

//返回二级属性名

Expression<Func<Test1, string>> expression2 = test => test.A.Address;

Assert.AreEqual( "A.Address", Lambda.GetName( expression2 ) );

//返回三级属性名

Expression<Func<Test1, string>> expression3 = test => test.A.B.Name;

Assert.AreEqual( "A.B.Name", Lambda.GetName( expression3 ) );

//测试可空整型

Expression<Func<Test1, int?>> expression4 = test => test.NullableInt;

Assert.AreEqual( "NullableInt", Lambda.GetName( expression4 ) );

//测试类型转换

Expression<Func<Test1, int?>> expression5 = test => test.Age;

Assert.AreEqual( "Age", Lambda.GetName( expression5 ) );

}

#endregion

#region GetValue(获取成员值)

/// <summary>

/// 获取成员值,委托返回类型为Object

/// </summary>

[TestMethod]

public void TestGetValue\_Object() {

Expression<Func<Test1, object>> expression = test => test.Name == "A";

Assert.AreEqual( "A", Lambda.GetValue( expression ) );

}

/// <summary>

/// 获取成员值,委托返回类型为bool

/// </summary>

[TestMethod]

public void TestGetValue\_Boolean() {

//空值返回null

Assert.AreEqual( null, Lambda.GetValue( null ) );

//一级返回值

Expression<Func<Test1, bool>> expression = test => test.Name == "A";

Assert.AreEqual( "A", Lambda.GetValue( expression ) );

//二级返回值

Expression<Func<Test1, bool>> expression2 = test => test.A.Integer == 1;

Assert.AreEqual( 1, Lambda.GetValue( expression2 ) );

//三级返回值

Expression<Func<Test1, bool>> expression3 = test => test.A.B.Name == "B";

Assert.AreEqual( "B", Lambda.GetValue( expression3 ) );

}

/// <summary>

/// 获取可空类型的值

/// </summary>

[TestMethod]

public void TestGetValue\_Nullable() {

//可空整型

Expression<Func<Test1, bool>> expression = test => test.NullableInt == 1;

Assert.AreEqual( 1, Lambda.GetValue( expression ) );

//可空decimal

expression = test => test.NullableDecimal == 1.5M;

Assert.AreEqual( 1.5M, Lambda.GetValue( expression ) );

}

/// <summary>

/// 获取成员值，运算符为方法

/// </summary>

[TestMethod]

public void TestGetValue\_Method() {

//1级返回值

Expression<Func<Test1, bool>> expression = t => t.Name.Contains( "A" );

Assert.AreEqual( "A", Lambda.GetValue( expression ) );

//二级返回值

expression = t => t.A.Address.Contains( "B" );

Assert.AreEqual( "B", Lambda.GetValue( expression ) );

//三级返回值

expression = t => t.A.B.Name.StartsWith( "C" );

Assert.AreEqual( "C", Lambda.GetValue( expression ) );

}

/// <summary>

/// 从实例中获取值

/// </summary>

[TestMethod]

public void TestGetValue\_Instance() {

var test = new Test1() { Name = "a", A = new Test1.TestA() { Address = "b", B = new Test1.TestA.TestB() { Name = "c" } } };

//一级属性

Expression<Func<string>> expression = () => test.Name;

Assert.AreEqual( "a", Lambda.GetValue( expression ) );

//二级属性

Expression<Func<string>> expression2 = () => test.A.Address;

Assert.AreEqual( "b", Lambda.GetValue( expression2 ) );

//三级属性

Expression<Func<string>> expression3 = () => test.A.B.Name;

Assert.AreEqual( "c", Lambda.GetValue( expression3 ) );

}

/// <summary>

/// 测试值为复杂类型

/// </summary>

[TestMethod]

public void TestGetValue\_Complex() {

var test = new Test1() { Name = "a", A = new Test1.TestA() { Address = "b" } };

//获取表达式的值

Expression<Func<Test1, bool>> expression = t => t.Name == test.Name;

Assert.AreEqual( "a", Lambda.GetValue( expression ), "==test.Name" );

Expression<Func<Test1, bool>> expression2 = t => t.Name == test.A.Address;

Assert.AreEqual( "b", Lambda.GetValue( expression2 ), "==test.A.Address" );

//获取方法的值

Expression<Func<Test1, bool>> expression3 = t => t.Name.Contains( test.Name );

Assert.AreEqual( "a", Lambda.GetValue( expression3 ), "Contains test.Name" );

Expression<Func<Test1, bool>> expression4 = t => t.Name.Contains( test.A.Address );

Assert.AreEqual( "b", Lambda.GetValue( expression4 ), "==test.A.Address" );

}

#endregion

#region GetParameter(获取参数)

/// <summary>

/// 获取参数

/// </summary>

[TestMethod]

public void TestGetParameter() {

//空值返回null

Assert.AreEqual( null, Lambda.GetParameter( null ) );

//一级返回值

Expression<Func<Test1, object>> expression = test => test.Name == "A";

Assert.AreEqual( "test", Lambda.GetParameter( expression ).ToString() );

//二级返回值

Expression<Func<Test1, object>> expression2 = test => test.A.Integer == 1;

Assert.AreEqual( "test", Lambda.GetParameter( expression2 ).ToString() );

//三级返回值

Expression<Func<Test1, object>> expression3 = test => test.A.B.Name == "B";

Assert.AreEqual( "test", Lambda.GetParameter( expression3 ).ToString() );

}

#endregion

#region GetCriteriaCount(获取谓词条件的个数)

/// <summary>

/// 获取谓词条件的个数

/// </summary>

[TestMethod]

public void TestGetCriteriaCount() {

//0个条件

Assert.AreEqual( 0, Lambda.GetCriteriaCount( null ) );

//1个条件

Expression<Func<Test1, bool>> expression = test => test.Name == "A";

Assert.AreEqual( 1, Lambda.GetCriteriaCount( expression ) );

//2个条件，与连接符

expression = test => test.Name == "A" && test.Name == "B";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

//2个条件，或连接符

expression = test => test.Name == "A" || test.Name == "B";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

//3个条件

expression = test => test.Name == "A" && test.Name == "B" || test.Name == "C";

Assert.AreEqual( 3, Lambda.GetCriteriaCount( expression ) );

//3个条件,包括导航属性

expression = test => test.A.Address == "A" && test.Name == "B" || test.Name == "C";

Assert.AreEqual( 3, Lambda.GetCriteriaCount( expression ) );

}

/// <summary>

/// 获取谓词条件的个数，运算符为方法

/// </summary>

[TestMethod]

public void TestGetCriteriaCount\_Method() {

//1个条件

Expression<Func<Test1, bool>> expression = t => t.Name.Contains( "A" );

Assert.AreEqual( 1, Lambda.GetCriteriaCount( expression ) );

//2个条件,与连接

expression = t => t.Name.Contains( "A" ) && t.Name == "A";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

//2个条件,或连接,包含导航属性

expression = t => t.Name.Contains( "A" ) || t.A.Address == "A";

Assert.AreEqual( 2, Lambda.GetCriteriaCount( expression ) );

}

#endregion

#region Equal(创建等于表达式)

/// <summary>

/// 创建等于表达式

/// </summary>

[TestMethod]

public void TestEqual() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Age == 1;

Assert.AreEqual( expected.ToString(), Lambda.Equal<Test1>( "Age", 1 ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Integer == 1;

Assert.AreEqual( expected2.ToString(), Lambda.Equal<Test1>( "A.Integer", 1 ).ToString() );

}

#endregion

#region NotEqual(创建不等于表达式)

/// <summary>

/// 创建不等于表达式

/// </summary>

[TestMethod]

public void TestNotEqual() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Age != 1;

Assert.AreEqual( expected.ToString(), Lambda.NotEqual<Test1>( "Age", 1 ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Integer != 1;

Assert.AreEqual( expected2.ToString(), Lambda.NotEqual<Test1>( "A.Integer", 1 ).ToString() );

}

#endregion

#region Greater(创建大于表达式)

/// <summary>

/// 创建大于表达式

/// </summary>

[TestMethod]

public void TestGreater() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Age > 1;

Assert.AreEqual( expected.ToString(), Lambda.Greater<Test1>( "Age", 1 ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Integer > 1;

Assert.AreEqual( expected2.ToString(), Lambda.Greater<Test1>( "A.Integer", 1 ).ToString() );

}

#endregion

#region Less(创建小于表达式)

/// <summary>

/// 创建小于表达式

/// </summary>

[TestMethod]

public void TestLess() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Age < 1;

Assert.AreEqual( expected.ToString(), Lambda.Less<Test1>( "Age", 1 ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Integer < 1;

Assert.AreEqual( expected2.ToString(), Lambda.Less<Test1>( "A.Integer", 1 ).ToString() );

}

#endregion

#region GreaterEqual(创建大于等于表达式)

/// <summary>

/// 创建大于等于表达式

/// </summary>

[TestMethod]

public void TestGreaterEqual() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Age >= 1;

Assert.AreEqual( expected.ToString(), Lambda.GreaterEqual<Test1>( "Age", 1 ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Integer >= 1;

Assert.AreEqual( expected2.ToString(), Lambda.GreaterEqual<Test1>( "A.Integer", 1 ).ToString() );

}

#endregion

#region LessEqual(创建小于等于表达式)

/// <summary>

/// 创建小于等于表达式

/// </summary>

[TestMethod]

public void TestLessEqual() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Age <= 1;

Assert.AreEqual( expected.ToString(), Lambda.LessEqual<Test1>( "Age", 1 ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Integer <= 1;

Assert.AreEqual( expected2.ToString(), Lambda.LessEqual<Test1>( "A.Integer", 1 ).ToString() );

}

#endregion

#region Contains(调用Contains方法)

/// <summary>

/// 调用Contains方法

/// </summary>

[TestMethod]

public void TestContains() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Name.Contains( "A" );

Assert.AreEqual( expected.ToString(), Lambda.Contains<Test1>( "Name", "A" ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Address.Contains( "A" );

Assert.AreEqual( expected2.ToString(), Lambda.Contains<Test1>( "A.Address", "A" ).ToString() );

//三级属性

Expression<Func<Test1, bool>> expected3 = t => t.A.B.Name.Contains( "A" );

Assert.AreEqual( expected3.ToString(), Lambda.Contains<Test1>( "A.B.Name", "A" ).ToString() );

}

#endregion

#region Starts(调用StartsWith方法)

/// <summary>

/// 调用StartsWith方法

/// </summary>

[TestMethod]

public void TestStarts() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Name.StartsWith( "A" );

Assert.AreEqual( expected.ToString(), Lambda.Starts<Test1>( "Name", "A" ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Address.StartsWith( "A" );

Assert.AreEqual( expected2.ToString(), Lambda.Starts<Test1>( "A.Address", "A" ).ToString() );

//三级属性

Expression<Func<Test1, bool>> expected3 = t => t.A.B.Name.StartsWith( "A" );

Assert.AreEqual( expected3.ToString(), Lambda.Starts<Test1>( "A.B.Name", "A" ).ToString() );

}

#endregion

#region Ends(调用EndsWith方法)

/// <summary>

/// 调用EndsWith方法

/// </summary>

[TestMethod]

public void TestEnds() {

//一级属性

Expression<Func<Test1, bool>> expected = t => t.Name.EndsWith( "A" );

Assert.AreEqual( expected.ToString(), Lambda.Ends<Test1>( "Name", "A" ).ToString() );

//二级属性

Expression<Func<Test1, bool>> expected2 = t => t.A.Address.EndsWith( "A" );

Assert.AreEqual( expected2.ToString(), Lambda.Ends<Test1>( "A.Address", "A" ).ToString() );

//三级属性

Expression<Func<Test1, bool>> expected3 = t => t.A.B.Name.EndsWith( "A" );

Assert.AreEqual( expected3.ToString(), Lambda.Ends<Test1>( "A.B.Name", "A" ).ToString() );

}

#endregion

#region GetConst(获取常量表达式)

/// <summary>

/// 获取常量表达式

/// </summary>

[TestMethod]

public void TestGetConst() {

Expression<Func<Test1, int?>> property = t => t.NullableInt;

ConstantExpression constantExpression = Lambda.Constant( property, 1 );

Assert.AreEqual( typeof( int ), constantExpression.Type );

}

#endregion

#region GetAttribute(获取特性)

/// <summary>

/// 测试获取特性

/// </summary>

[TestMethod]

public void TestGetAttribute() {

DisplayAttribute attribute = Lambda.GetAttribute<Test1, string, DisplayAttribute>( t => t.Name );

Assert.AreEqual( "名称", attribute.Name );

}

#endregion

#region GetAttributes(获取特性列表)

/// <summary>

/// 测试获取特性列表

/// </summary>

[TestMethod]

public void TestGetAttributes() {

IEnumerable<ValidationAttribute> attributes = Lambda.GetAttributes<Test1, string, ValidationAttribute>( t => t.Name );

Assert.AreEqual( 2, attributes.Count() );

}

#endregion

}

}

[复制代码](javascript:void(0);)

[**Util应用程序框架公共操作类(九):Lambda表达式扩展**](http://www.cnblogs.com/xiadao521/p/4206717.html)

　　上一篇对Lambda表达式公共操作类进行了一些增强，本篇使用扩展方法对Lambda表达式进行扩展。

　　修改Util项目的Extensions.Expression.cs文件，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq;

using System.Linq.Expressions;

using System.Reflection;

using Util.Lambdas;

namespace Util {

/// <summary>

/// 表达式扩展

/// </summary>

public static partial class Extensions {

#region Property(属性表达式)

/// <summary>

/// 创建属性表达式

/// </summary>

/// <param name="expression">表达式</param>

/// <param name="propertyName">属性名,支持多级属性名，与句点分隔，范例：Customer.Name</param>

public static Expression Property( this Expression expression, string propertyName ) {

if ( propertyName.All( t => t != '.' ) )

return Expression.Property( expression, propertyName );

var propertyNameList = propertyName.Split( '.' );

Expression result = null;

for ( int i = 0; i < propertyNameList.Length; i++ ) {

if ( i == 0 ) {

result = Expression.Property( expression, propertyNameList[0] );

continue;

}

result = result.Property( propertyNameList[i] );

}

return result;

}

/// <summary>

/// 创建属性表达式

/// </summary>

/// <param name="expression">表达式</param>

/// <param name="member">属性</param>

public static Expression Property( this Expression expression, MemberInfo member ) {

return Expression.MakeMemberAccess( expression, member );

}

#endregion

#region Operation(操作)

/// <summary>

/// 操作

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="operator">运算符</param>

/// <param name="value">值</param>

public static Expression Operation( this Expression left, Operator @operator, object value ) {

switch ( @operator ) {

case Operator.Equal:

return left.Equal( value );

case Operator.NotEqual:

return left.NotEqual( value );

case Operator.Greater:

return left.Greater( value );

case Operator.Less:

return left.Less( value );

case Operator.GreaterEqual:

return left.GreaterEqual( value );

case Operator.LessEqual:

return left.LessEqual( value );

case Operator.Contains:

return left.Call( "Contains", value );

case Operator.Starts:

return left.StartsWith( value );

case Operator.Ends:

return left.EndsWith( value );

}

throw new NotImplementedException();

}

#endregion

#region StartsWith(头匹配)

/// <summary>

/// 头匹配

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression StartsWith( this Expression left, object value ) {

return left.Call( "StartsWith", new[] { typeof( string ) }, value );

}

#endregion

#region EndsWith(尾匹配)

/// <summary>

/// 尾匹配

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression EndsWith( this Expression left, object value ) {

return left.Call( "EndsWith", new[] { typeof( string ) }, value );

}

#endregion

#region Call(调用方法表达式)

/// <summary>

/// 创建调用方法表达式

/// </summary>

/// <param name="instance">调用的实例</param>

/// <param name="methodName">方法名</param>

/// <param name="values">参数值列表</param>

public static Expression Call( this Expression instance, string methodName, params Expression[] values ) {

return Expression.Call( instance, instance.Type.GetMethod( methodName ), values );

}

/// <summary>

/// 创建调用方法表达式

/// </summary>

/// <param name="instance">调用的实例</param>

/// <param name="methodName">方法名</param>

/// <param name="values">参数值列表</param>

public static Expression Call( this Expression instance, string methodName, params object[] values ) {

if ( values == null || values.Length == 0 )

return Expression.Call( instance, instance.Type.GetMethod( methodName ) );

return Expression.Call( instance, instance.Type.GetMethod( methodName ), values.Select( Expression.Constant ) );

}

/// <summary>

/// 创建调用方法表达式

/// </summary>

/// <param name="instance">调用的实例</param>

/// <param name="methodName">方法名</param>

/// <param name="paramTypes">参数类型列表</param>

/// <param name="values">参数值列表</param>

public static Expression Call( this Expression instance, string methodName, Type[] paramTypes, params object[] values ) {

if ( values == null || values.Length == 0 )

return Expression.Call( instance, instance.Type.GetMethod( methodName, paramTypes ) );

return Expression.Call( instance, instance.Type.GetMethod( methodName, paramTypes ), values.Select( Expression.Constant ) );

}

#endregion

#region Equal(等于表达式)

/// <summary>

/// 创建等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression Equal( this Expression left, Expression right ) {

return Expression.Equal( left, right );

}

/// <summary>

/// 创建等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression Equal( this Expression left, object value ) {

return left.Equal( Lambda.Constant( left, value ) );

}

#endregion

#region NotEqual(不等于表达式)

/// <summary>

/// 创建不等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression NotEqual( this Expression left, Expression right ) {

return Expression.NotEqual( left, right );

}

/// <summary>

/// 创建不等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression NotEqual( this Expression left, object value ) {

return left.NotEqual( Lambda.Constant( left, value ) );

}

#endregion

#region Greater(大于表达式)

/// <summary>

/// 创建大于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression Greater( this Expression left, Expression right ) {

return Expression.GreaterThan( left, right );

}

/// <summary>

/// 创建大于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression Greater( this Expression left, object value ) {

return left.Greater( Lambda.Constant( left, value ) );

}

#endregion

#region Less(小于表达式)

/// <summary>

/// 创建小于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression Less( this Expression left, Expression right ) {

return Expression.LessThan( left, right );

}

/// <summary>

/// 创建小于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression Less( this Expression left, object value ) {

return left.Less( Lambda.Constant( left, value ) );

}

#endregion

#region GreaterEqual(大于等于表达式)

/// <summary>

/// 创建大于等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression GreaterEqual( this Expression left, Expression right ) {

return Expression.GreaterThanOrEqual( left, right );

}

/// <summary>

/// 创建大于等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression GreaterEqual( this Expression left, object value ) {

return left.GreaterEqual( Lambda.Constant( left, value ) );

}

#endregion

#region LessEqual(小于等于表达式)

/// <summary>

/// 创建小于等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression LessEqual( this Expression left, Expression right ) {

return Expression.LessThanOrEqual( left, right );

}

/// <summary>

/// 创建小于等于运算表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="value">值</param>

public static Expression LessEqual( this Expression left, object value ) {

return left.LessEqual( Lambda.Constant( left, value ) );

}

#endregion

#region Compose(组合表达式)

/// <summary>

/// 组合表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="first">左操作数</param>

/// <param name="second">右操作数</param>

/// <param name="merge">合并操作</param>

internal static Expression<T> Compose<T>( this Expression<T> first, Expression<T> second,

Func<Expression, Expression, Expression> merge ) {

var map = first.Parameters.Select( ( f, i ) => new { f, s = second.Parameters[i] } ).ToDictionary( p => p.s, p => p.f );

var secondBody = ParameterRebinder.ReplaceParameters( map, second.Body );

return Expression.Lambda<T>( merge( first.Body, secondBody ), first.Parameters );

}

#endregion

#region And(与表达式)

/// <summary>

/// 与操作表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression And( this Expression left, Expression right ) {

if ( left == null )

return right;

if ( right == null )

return left;

return Expression.AndAlso( left, right );

}

/// <summary>

/// 与操作表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression<Func<T, bool>> And<T>( this Expression<Func<T, bool>> left, Expression<Func<T, bool>> right ) {

if ( left == null )

return right;

if ( right == null )

return left;

return left.Compose( right, Expression.AndAlso );

}

#endregion

#region Or(或表达式)

/// <summary>

/// 或操作表达式

/// </summary>

/// <param name="left">左操作数</param>

/// <param name="right">右操作数</param>

public static Expression Or( this Expression left, Expression right ) {

return Expression.OrElse( left, right );

}

/// <summary>

/// 或操作表达式

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

/// <param name="first">左操作数</param>

/// <param name="second">右操作数</param>

/// <returns></returns>

public static Expression<Func<T, bool>> Or<T>( this Expression<Func<T, bool>> first, Expression<Func<T, bool>> second ) {

return first.Compose( second, Expression.OrElse );

}

#endregion

#region Value(获取lambda表达式的值)

/// <summary>

/// 获取lambda表达式的值

/// </summary>

/// <typeparam name="T">对象类型</typeparam>

public static object Value<T>( this Expression<Func<T, bool>> expression ) {

return Lambda.GetValue( expression );

}

#endregion

#region ToLambda(创建Lambda表达式)

/// <summary>

/// 创建Lambda表达式

/// </summary>

/// <typeparam name="TDelegate">委托类型</typeparam>

/// <param name="body">表达式</param>

/// <param name="parameters">参数列表</param>

public static Expression<TDelegate> ToLambda<TDelegate>( this Expression body, params ParameterExpression[] parameters ) {

return Expression.Lambda<TDelegate>( body, parameters );

}

#endregion

}

}

[复制代码](javascript:void(0);)

　　在Util项目中添加Operator枚举，代码如下。

[复制代码](javascript:void(0);)

using System.ComponentModel;

namespace Util {

/// <summary>

/// 操作符

/// </summary>

public enum Operator {

/// <summary>

/// 等于

/// </summary>

[Description( "等于" )]

Equal,

/// <summary>

/// 不等于

/// </summary>

[Description( "不等于" )]

NotEqual,

/// <summary>

/// 大于

/// </summary>

[Description( "大于" )]

Greater,

/// <summary>

/// 小于

/// </summary>

[Description( "小于" )]

Less,

/// <summary>

/// 大于等于

/// </summary>

[Description( "大于等于" )]

GreaterEqual,

/// <summary>

/// 小于等于

/// </summary>

[Description( "小于等于" )]

LessEqual,

/// <summary>

/// 头尾匹配

/// </summary>

[Description( "头尾匹配" )]

Contains,

/// <summary>

/// 头匹配

/// </summary>

[Description( "头匹配" )]

Starts,

/// <summary>

/// 尾匹配

/// </summary>

[Description( "尾匹配" )]

Ends

}

}

[复制代码](javascript:void(0);)

　　找到Util.Tests测试项目，修改Extensions目录下的ExpressionExtensionTest，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Util.Tests.Extensions {

/// <summary>

/// 表达式扩展测试

/// </summary>

[TestClass]

public class ExpressionExtensionTest {

#region 测试初始化

/// <summary>

/// 参数表达式

/// </summary>

private ParameterExpression \_parameterExpression;

/// <summary>

/// 表达式1

/// </summary>

private Expression \_expression1;

/// <summary>

/// 表达式2

/// </summary>

private Expression \_expression2;

/// <summary>

/// 测试初始化

/// </summary>

[TestInitialize]

public void TestInit() {

\_parameterExpression = Expression.Parameter( typeof( Person ), "t" );

\_expression1 = \_parameterExpression.Property( "Name" ).Call( "Contains", Expression.Constant( "A" ) );

\_expression2 = \_parameterExpression.Property( "Birthday" )

.Property( "Value" )

.Property( "Year" )

.Greater( Expression.Constant( 2000 ) );

}

#endregion

#region And(与操作)

/// <summary>

/// 测试And方法，连接两个表达式

/// </summary>

[TestMethod]

public void TestAnd() {

var andExpression = \_expression1.And( \_expression2 ).ToLambda<Func<Person, bool>>( \_parameterExpression );

Expression<Func<Person, bool>> expected = t => t.Name.Contains( "A" ) && t.Birthday.Value.Year > 2000;

Assert.AreEqual( expected.ToString(), andExpression.ToString() );

}

/// <summary>

/// 测试And方法，连接两个表达式

/// </summary>

[TestMethod]

public void TestAnd\_2() {

Expression<Func<Person, bool>> left = t => t.Name == "A";

Expression<Func<Person, bool>> right = t => t.Name == "B";

Expression<Func<Person, bool>> expected = t => t.Name == "A" && t.Name == "B";

Assert.AreEqual( expected.ToString(), left.And( right ).ToString() );

}

#endregion

#region Or(或操作)

/// <summary>

/// 测试Or方法，连接两个表达式

/// </summary>

[TestMethod]

public void TestOr() {

var andExpression = \_expression1.Or( \_expression2 ).ToLambda<Func<Person, bool>>( \_parameterExpression );

Expression<Func<Person, bool>> expected = t => t.Name.Contains( "A" ) || t.Birthday.Value.Year > 2000;

Assert.AreEqual( expected.ToString(), andExpression.ToString() );

}

/// <summary>

/// 测试Or方法，连接两个表达式

/// </summary>

[TestMethod]

public void TestOr\_2() {

Expression<Func<Person, bool>> left = t => t.Name == "A";

Expression<Func<Person, bool>> right = t => t.Name == "B";

Expression<Func<Person, bool>> expected = t => t.Name == "A" || t.Name == "B";

Assert.AreEqual( expected.ToString(), left.Or( right ).ToString() );

}

#endregion

#region Value(获取成员值)

/// <summary>

/// 获取成员值

/// </summary>

[TestMethod]

public void TestValue() {

Expression<Func<Person, bool>> expression = test => test.Name == "A";

Assert.AreEqual( "A", expression.Value() );

}

#endregion

#region 运算符操作

/// <summary>

/// 测试相等

/// </summary>

[TestMethod]

public void TestEqual\_Nullable() {

\_expression1 = \_parameterExpression.Property( "Age" ).Equal( 1 );

Assert.AreEqual( "t => (t.Age == 1)",

\_expression1.ToLambda<Func<Person, bool>>( \_parameterExpression ).ToString() );

}

/// <summary>

/// 测试不相等

/// </summary>

[TestMethod]

public void TestNotEqual\_Nullable() {

\_expression1 = \_parameterExpression.Property( "Age" ).NotEqual( 1 );

Assert.AreEqual( "t => (t.Age != 1)",

\_expression1.ToLambda<Func<Person, bool>>( \_parameterExpression ).ToString() );

}

/// <summary>

/// 测试大于

/// </summary>

[TestMethod]

public void TestGreater\_Nullable() {

\_expression1 = \_parameterExpression.Property( "Age" ).Greater( 1 );

Assert.AreEqual( "t => (t.Age > 1)",

\_expression1.ToLambda<Func<Person, bool>>( \_parameterExpression ).ToString() );

}

/// <summary>

/// 测试大于等于

/// </summary>

[TestMethod]

public void TestGreaterEqual\_Nullable() {

\_expression1 = \_parameterExpression.Property( "Age" ).GreaterEqual( 1 );

Assert.AreEqual( "t => (t.Age >= 1)",

\_expression1.ToLambda<Func<Person, bool>>( \_parameterExpression ).ToString() );

}

/// <summary>

/// 测试小于

/// </summary>

[TestMethod]

public void TestLess\_Nullable() {

\_expression1 = \_parameterExpression.Property( "Age" ).Less( 1 );

Assert.AreEqual( "t => (t.Age < 1)",

\_expression1.ToLambda<Func<Person, bool>>( \_parameterExpression ).ToString() );

}

/// <summary>

/// 测试小于等于

/// </summary>

[TestMethod]

public void TestLessEqual\_Nullable() {

\_expression1 = \_parameterExpression.Property( "Age" ).LessEqual( 1 );

Assert.AreEqual( "t => (t.Age <= 1)",

\_expression1.ToLambda<Func<Person, bool>>( \_parameterExpression ).ToString() );

}

#endregion

#region Person（测试类）

/// <summary>

/// 测试

/// </summary>

public class Person {

public string Name { get; set; }

public int? Age { get; set; }

public DateTime? Birthday { get; set; }

}

#endregion

}

}

[复制代码](javascript:void(0);)

　　需要注意的是，如果需要合并表达式，比如And或Or操作，需要用到一个ParameterRebinder类，它从ExpressionVisitor派生，这个类我是从国外一个网站上直接Copy过来的，代码如下。

[复制代码](javascript:void(0);)

using System.Collections.Generic;

using System.Linq.Expressions;

namespace Util.Lambdas {

/// <summary>

/// 参数重绑定操作

/// </summary>

public class ParameterRebinder : ExpressionVisitor {

/// <summary>

/// 参数字典

/// </summary>

private readonly Dictionary<ParameterExpression, ParameterExpression> \_map;

/// <summary>

/// 初始化参数重绑定操作

/// </summary>

/// <param name="map">参数字典</param>

public ParameterRebinder( Dictionary<ParameterExpression, ParameterExpression> map ) {

\_map = map ?? new Dictionary<ParameterExpression, ParameterExpression>();

}

/// <summary>

/// 替换参数

/// </summary>

/// <param name="map">参数字典</param>

/// <param name="exp">表达式</param>

public static Expression ReplaceParameters( Dictionary<ParameterExpression, ParameterExpression> map, Expression exp ) {

return new ParameterRebinder( map ).Visit( exp );

}

/// <summary>

/// 访问参数

/// </summary>

/// <param name="parameterExpression">参数</param>

protected override Expression VisitParameter( ParameterExpression parameterExpression ) {

ParameterExpression replacement;

if ( \_map.TryGetValue( parameterExpression, out replacement ) )

parameterExpression = replacement;

return base.VisitParameter( parameterExpression );

}

}

}

[复制代码](javascript:void(0);)

[**Util应用程序框架公共操作类(十一):表达式生成器**](http://www.cnblogs.com/xiadao521/p/4206780.html)

　　本篇介绍的表达式生成器，用于动态创建表达式。

　　在Util项目Lambdas目录中，添加**ExpressionBuilder**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

namespace Util.Lambdas {

/// <summary>

/// 表达式生成器

/// </summary>

public class ExpressionBuilder<TEntity> {

/// <summary>

/// 初始化表达式生成器

/// </summary>

public ExpressionBuilder() {

Parameter = Expression.Parameter( typeof( TEntity ), "t" );

}

/// <summary>

/// 参数

/// </summary>

private ParameterExpression Parameter { get; set; }

/// <summary>

/// 获取参数

/// </summary>

public ParameterExpression GetParameter() {

return Parameter;

}

/// <summary>

/// 创建表达式

/// </summary>

/// <param name="property">属性表达式</param>

/// <param name="operator">运算符</param>

/// <param name="value">值</param>

public Expression Create<T>( Expression<Func<TEntity, T>> property, Operator @operator, object value ) {

return Parameter.Property( Lambda.GetMember( property ) ).Operation( @operator, value );

}

/// <summary>

/// 转换为Lambda表达式

/// </summary>

/// <param name="expression">表达式</param>

public Expression<Func<TEntity, bool>> ToLambda( Expression expression ) {

if ( expression == null )

return null;

return expression.ToLambda<Func<TEntity, bool>>( Parameter );

}

}

}

[复制代码](javascript:void(0);)

　　在Util.Tests测试项目中，添加**ExpressionBuilderTest**单元测试，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Util.Lambdas;

using Util.Tests.Samples;

namespace Util.Tests {

/// <summary>

/// 测试表达式生成器

/// </summary>

[TestClass]

public class ExpressionBuilderTest {

/// <summary>

/// 表达式生成器

/// </summary>

ExpressionBuilder<Test2> \_builder;

/// <summary>

/// 测试初始化

/// </summary>

[TestInitialize]

public void TestInit() {

\_builder = new ExpressionBuilder<Test2>();

}

/// <summary>

/// 创建表达式

/// </summary>

[TestMethod]

public void TestCreate\_Int() {

Expression<Func<Test2, int>> property = t => t.Int;

var expression = \_builder.Create( property, Operator.Equal, 1 );

Expression<Func<Test2, bool>> expected = t => t.Int == 1;

Assert.AreEqual( expected.ToString(), \_builder.ToLambda( expression ).ToString() );

}

/// <summary>

/// 创建表达式

/// </summary>

[TestMethod]

public void TestCreate\_Int\_Nullable() {

Expression<Func<Test2, int?>> property = t => t.NullableInt;

var expression = \_builder.Create( property, Operator.Equal, 1 );

Assert.AreEqual( "t => (t.NullableInt == 1)", \_builder.ToLambda( expression ).ToString() );

}

}

}

[复制代码](javascript:void(0);)

[**应用程序框架实战二十五:查询条件(规约模式应用)**](http://www.cnblogs.com/xiadao521/p/4235120.html)

　　前面已经做了一些准备工作，本篇将介绍查询条件的封装，它是规约模式的一个应用。

　　规约使用一个对象来封装谓词，我之前已经介绍过它在验证方面的应用，本篇是规约模式在查询方面的应用。

　　规约的强大之处在于，能够将一堆杂乱无章的条件判断或查询条件封装起来，以一个清晰的概念来表达，并使得这些谓词具备了可复用的能力。

　　首先在Util.Domains项目的Repositories目录中创建**ICriteria**接口，这个接口表示一个查询条件，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

namespace Util.Domains.Repositories {

/// <summary>

/// 查询条件

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

public interface ICriteria<TEntity> where TEntity : class,IAggregateRoot {

/// <summary>

/// 获取谓词

/// </summary>

Expression<Func<TEntity, bool>> GetPredicate();

}

}

[复制代码](javascript:void(0);)

　　由于我们使用了EF这种ORM框架，查询条件的结果是一个**Expression<Func<TEntity, bool>>**的谓词表达式。

　　在Util.Datas项目中，打开Extensions.Query.cs文件，增加以下代码。

[复制代码](javascript:void(0);)

/// <summary>

/// 过滤

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="criteria">查询条件</param>

public static IQueryable<T> Filter<T>( this IQueryable<T> source, ICriteria<T> criteria ) where T : class,IAggregateRoot {

if ( criteria == null )

return source;

var predicate = criteria.GetPredicate();

if ( predicate == null )

return source;

return source.Where( predicate );

}

[复制代码](javascript:void(0);)

　　我们在IQueryable对象上扩展了一个**Filter**方法，该方法接收一个查询条件，如果查询条件有效，就使用Where方法添加过滤条件。

　　基础工作就这么多，下面来看几个范例。

　　在信息系统中，经常会进行范围查询，比如一个日期段的查询。这看起来是一个简单的需求，初学者一般这样写，**t => t.Date >= BeginDate && t.Date <= EndDate**，其结果可能是错误的，这是由于从表现层传入的查询条件是可选的，如果客户没有进行输入，结果就是错的。

　　对于范围查询来讲，还有更多的细节需要思考，比如，起始日期和结束日期都没有输入，或只输入了起始日期或结束日期，也可能客户输入的起始日期比结束日期还大。为了获得健壮性，我们会对查询条件进行各种判断，从而导致杂乱无章的代码。更要命的是，这些代码无法复用，在另一个范围查询的位置，我们必须把之前的代码复制过去进行修改。

　　一个更好的办法是把范围查询逻辑封装到查询条件对象中，以后需要进行范围查询时，即可随手拈来。

　　在Util.Datas项目Queries目录中，新建Criterias目录，创建一个查询条件基类**CriteriaBase**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Util.Domains;

using Util.Domains.Repositories;

namespace Util.Datas.Queries.Criterias {

/// <summary>

/// 查询条件

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

public abstract class CriteriaBase<TEntity> : ICriteria<TEntity> where TEntity : class, IAggregateRoot {

/// <summary>

/// 谓词

/// </summary>

protected Expression<Func<TEntity, bool>> Predicate { get; set; }

/// <summary>

/// 获取谓词

/// </summary>

public virtual Expression<Func<TEntity, bool>> GetPredicate() {

return Predicate;

}

}

}

[复制代码](javascript:void(0);)

　　根据数据类型不同，范围查询有很多种类，比如日期范围查询、日期时间范围查询、整数范围查询、浮点数范围查询等。我们需要为范围查询条件创建一个基类**SegmentCriteria**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Util.Domains;

using Util.Lambdas;

namespace Util.Datas.Queries.Criterias {

/// <summary>

/// 段过滤条件

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <typeparam name="TValue">值类型</typeparam>

public abstract class SegmentCriteria<TEntity, TProperty, TValue> : CriteriaBase<TEntity>

where TEntity : class, IAggregateRoot

where TValue : struct {

/// <summary>

/// 初始化段过滤条件

/// </summary>

/// <param name="propertyExpression">属性表达式</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

protected SegmentCriteria( Expression<Func<TEntity, TProperty>> propertyExpression, TValue? min, TValue? max ) {

Builder = new ExpressionBuilder<TEntity>();

PropertyExpression = propertyExpression;

Min = min;

Max = max;

if ( IsMinGreaterMax( min, max ) ) {

Min = max;

Max = min;

}

}

/// <summary>

/// 最小值是否大于最大值

/// </summary>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

protected abstract bool IsMinGreaterMax( TValue? min, TValue? max );

/// <summary>

/// 属性表达式

/// </summary>

public Expression<Func<TEntity, TProperty>> PropertyExpression { get; set; }

/// <summary>

/// 表达式生成器

/// </summary>

private ExpressionBuilder<TEntity> Builder { get; set; }

/// <summary>

/// 最小值

/// </summary>

public TValue? Min { get; set; }

/// <summary>

/// 最大值

/// </summary>

public TValue? Max { get; set; }

/// <summary>

/// 获取谓词

/// </summary>

public override Expression<Func<TEntity, bool>> GetPredicate() {

var first = CreateLeftExpression();

var second = CreateRightExpression();

return Builder.ToLambda( first.And( second ) );

}

/// <summary>

/// 创建左操作数，即 t => t.Property >= Min

/// </summary>

private Expression CreateLeftExpression() {

if ( Min == null )

return null;

return Builder.Create( PropertyExpression, Operator.GreaterEqual, GetMinValue() );

}

/// <summary>

/// 获取最小值

/// </summary>

protected virtual TValue? GetMinValue() {

return Min;

}

/// <summary>

/// 创建右操作数，即 t => t.Property &lt;= Max

/// </summary>

private Expression CreateRightExpression() {

if ( Max == null )

return null;

return Builder.Create( PropertyExpression, GetMaxOperator(), GetMaxValue() );

}

/// <summary>

/// 获取最大值相关的运算符

/// </summary>

protected virtual Operator GetMaxOperator() {

return Operator.LessEqual;

}

/// <summary>

/// 获取最大值

/// </summary>

protected virtual TValue? GetMaxValue() {

return Max;

}

}

}

[复制代码](javascript:void(0);)

　　对于日期范围查询，日期是否包含时间非常重要，它们在行为上是不同的。如果日期不包含时间，那么需要为结束日期加一天，并修改运算符为小于。

　　日期时间范围查询条件**DateTimeSegmentCriteria**,代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Util.Domains;

namespace Util.Datas.Queries.Criterias {

/// <summary>

/// 日期时间段过滤条件 - 包含时间

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

public class DateTimeSegmentCriteria<TEntity, TProperty> : SegmentCriteria<TEntity, TProperty, DateTime> where TEntity : class,IAggregateRoot {

/// <summary>

/// 初始化日期时间段过滤条件

/// </summary>

/// <param name="propertyExpression">属性表达式</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public DateTimeSegmentCriteria( Expression<Func<TEntity, TProperty>> propertyExpression, DateTime? min, DateTime? max )

: base( propertyExpression, min, max ) {

}

/// <summary>

/// 最小值是否大于最大值

/// </summary>

protected override bool IsMinGreaterMax( DateTime? min, DateTime? max ) {

return min > max;

}

}

}

[复制代码](javascript:void(0);)

　　日期范围查询条件**DateSegmentCriteria**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Util.Domains;

namespace Util.Datas.Queries.Criterias {

/// <summary>

/// 日期段过滤条件 - 不包含时间

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

public class DateSegmentCriteria<TEntity, TProperty> : SegmentCriteria<TEntity, TProperty, DateTime> where TEntity : class,IAggregateRoot {

/// <summary>

/// 初始化日期段过滤条件

/// </summary>

/// <param name="propertyExpression">属性表达式</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public DateSegmentCriteria( Expression<Func<TEntity, TProperty>> propertyExpression, DateTime? min, DateTime? max )

: base( propertyExpression, min, max ) {

}

/// <summary>

/// 最小值是否大于最大值

/// </summary>

protected override bool IsMinGreaterMax( DateTime? min, DateTime? max ) {

return min > max;

}

/// <summary>

/// 获取最小值

/// </summary>

protected override DateTime? GetMinValue() {

return base.GetMinValue().SafeValue().Date;

}

/// <summary>

/// 获取最大值

/// </summary>

protected override DateTime? GetMaxValue() {

return base.GetMaxValue().SafeValue().Date.AddDays( 1 );

}

/// <summary>

/// 获取最大值相关的运算符

/// </summary>

protected override Operator GetMaxOperator() {

return Operator.Less;

}

}

}

[复制代码](javascript:void(0);)

　　整数范围查询条件**IntSegmentCriteria**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Util.Domains;

namespace Util.Datas.Queries.Criterias {

/// <summary>

/// 整数段过滤条件

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

public class IntSegmentCriteria<TEntity, TProperty> : SegmentCriteria<TEntity, TProperty,int> where TEntity : class,IAggregateRoot {

/// <summary>

/// 初始化整数段过滤条件

/// </summary>

/// <param name="propertyExpression">属性表达式</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public IntSegmentCriteria( Expression<Func<TEntity, TProperty>> propertyExpression, int? min, int? max )

: base( propertyExpression,min,max){

}

/// <summary>

/// 最小值是否大于最大值

/// </summary>

protected override bool IsMinGreaterMax( int? min, int? max ) {

return min > max;

}

}

}

[复制代码](javascript:void(0);)

　　double范围查询条件**DoubleSegmentCriteria**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Util.Domains;

namespace Util.Datas.Queries.Criterias {

/// <summary>

/// double数值段过滤条件

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

public class DoubleSegmentCriteria<TEntity, TProperty> : SegmentCriteria<TEntity, TProperty, double> where TEntity : class,IAggregateRoot {

/// <summary>

/// 初始化double数值段过滤条件

/// </summary>

/// <param name="propertyExpression">属性表达式</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public DoubleSegmentCriteria( Expression<Func<TEntity, TProperty>> propertyExpression, double? min, double? max )

: base( propertyExpression, min, max ) {

}

/// <summary>

/// 最小值是否大于最大值

/// </summary>

protected override bool IsMinGreaterMax( double? min, double? max ) {

return min > max;

}

}

}

[复制代码](javascript:void(0);)

　　decimal范围查询条件**DecimalSegmentCriteria**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

using Util.Domains;

namespace Util.Datas.Queries.Criterias {

/// <summary>

/// decimal数值段过滤条件

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

public class DecimalSegmentCriteria<TEntity, TProperty> : SegmentCriteria<TEntity, TProperty, decimal> where TEntity : class,IAggregateRoot {

/// <summary>

/// 初始化decimal数值段过滤条件

/// </summary>

/// <param name="propertyExpression">属性表达式</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public DecimalSegmentCriteria( Expression<Func<TEntity, TProperty>> propertyExpression, decimal? min, decimal? max )

: base( propertyExpression, min, max ) {

}

/// <summary>

/// 最小值是否大于最大值

/// </summary>

protected override bool IsMinGreaterMax( decimal? min, decimal? max ) {

return min > max;

}

}

}

[复制代码](javascript:void(0);)

　　我们现在进行日期范围查询，就比较简单了，代码如下。

queryable.Filter( new DateSegmentCriteria<Test, DateTime>( t => t.Date, BeginDate,EndDate ) );

　　不过上面的代码用起来还不是太顺手，可以将范围查询扩展到IQueryable，代码如下。

[复制代码](javascript:void(0);)

/// <summary>

/// 过滤整数段

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public static IQueryable<T> FilterInt<T, TProperty>( this IQueryable<T> source,

Expression<Func<T, TProperty>> propertyExpression, int? min, int? max ) where T : class,IAggregateRoot {

return source.Filter( new IntSegmentCriteria<T, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤double数值段

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public static IQueryable<T> FilterDouble<T, TProperty>( this IQueryable<T> source,

Expression<Func<T, TProperty>> propertyExpression, double? min, double? max ) where T : class,IAggregateRoot {

return source.Filter( new DoubleSegmentCriteria<T, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤decimal数值段

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public static IQueryable<T> FilterDecimal<T, TProperty>( this IQueryable<T> source,

Expression<Func<T, TProperty>> propertyExpression, decimal? min, decimal? max ) where T : class,IAggregateRoot {

return source.Filter( new DecimalSegmentCriteria<T, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤日期段，不包含时间

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public static IQueryable<T> FilterDate<T, TProperty>( this IQueryable<T> source,

Expression<Func<T, TProperty>> propertyExpression, DateTime? min, DateTime? max ) where T : class,IAggregateRoot {

return source.Filter( new DateSegmentCriteria<T, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤日期时间段，包含时间

/// </summary>

/// <typeparam name="T">实体类型</typeparam>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="source">数据源</param>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public static IQueryable<T> FilterDateTime<T, TProperty>( this IQueryable<T> source,

Expression<Func<T, TProperty>> propertyExpression, DateTime? min, DateTime? max ) where T : class,IAggregateRoot {

return source.Filter( new DateTimeSegmentCriteria<T, TProperty>( propertyExpression, min, max ) );

}

[复制代码](javascript:void(0);)

　　日期范围查询的调用代码简化为如下代码。

queryable.FilterDate( t => t.Date, BeginDate, EndDate)

　　本文介绍了如何使用查询条件对象封装范围查询，当然你可以用类似的方法将业务中的查询条件封装起来。

　　规约模式还有其它用法，更强大的用法，请参考陈晴阳老兄的这篇http://www.cnblogs.com/daxnet/p/3925426.html

[**应用程序框架实战二十六:查询对象**](http://www.cnblogs.com/xiadao521/p/4237606.html)

　　信息系统的查询需求千变万化，在仓储中为每个查询需求创建一个特殊方法，将导致大量乏味而臃肿的接口。

　　一种更加可行的办法是，在应用层服务中描述查询需求，并通过仓储执行查询。

　　为了能够更好的描述查询需求，可以将查询功能从仓储中抽取出来，专门创建一个查询对象。

　　查询最复杂的部分是条件过滤，这也是查询对象的主要职责。查询对象可以认为是规约模式的一个变种，允许查询对象动态创建查询条件。

　　在Util.Domains项目Repositories目录中，创建查询对象基接口**IQueryBase**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Linq.Expressions;

namespace Util.Domains.Repositories {

/// <summary>

/// 查询对象

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

public interface IQueryBase<TEntity> : IPager where TEntity : class, IAggregateRoot {

/// <summary>

/// 获取谓词

/// </summary>

Expression<Func<TEntity, bool>> GetPredicate();

/// <summary>

/// 获取排序

/// </summary>

string GetOrderBy();

}

}

[复制代码](javascript:void(0);)

　　IQueryBase接口主要用来支持将查询对象传入仓储中。

　　在仓储接口**IRepository**中增加两个方法，代码如下。

[复制代码](javascript:void(0);)

/// <summary>

/// 查询

/// </summary>

/// <param name="query">查询对象</param>

IQueryable<TEntity> Query( IQueryBase<TEntity> query );

/// <summary>

/// 分页查询

/// </summary>

/// <param name="query">查询对象</param>

PagerList<TEntity> PagerQuery( IQueryBase<TEntity> query );

[复制代码](javascript:void(0);)

　　在仓储类**Repository**中实现这两个方法，代码如下。

[复制代码](javascript:void(0);)

/// <summary>

/// 查询

/// </summary>

/// <param name="query">查询对象</param>

public IQueryable<TEntity> Query( IQueryBase<TEntity> query ) {

return FilterBy( Find(), query );

}

/// <summary>

/// 过滤

/// </summary>

protected IQueryable<TEntity> FilterBy( IQueryable<TEntity> queryable, IQueryBase<TEntity> query ) {

var predicate = query.GetPredicate();

if ( predicate == null )

return queryable;

return queryable.Where( predicate );

}

/// <summary>

/// 分页查询

/// </summary>

/// <param name="query">查询对象</param>

public virtual PagerList<TEntity> PagerQuery( IQueryBase<TEntity> query ) {

return Query( query ).PagerResult( query );

}

[复制代码](javascript:void(0);)

　　以上代码完成了将查询对象传入仓储并获取结果，下面开始实现查询对象。

　　在Util.Datas项目Queries目录中，创建**IQuery**接口，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

using Util.Domains;

using Util.Domains.Repositories;

namespace Util.Datas.Queries {

/// <summary>

/// 查询对象

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TKey">实体标识类型</typeparam>

public interface IQuery<TEntity, TKey> : IQueryBase<TEntity> where TEntity : class,IAggregateRoot<TKey> {

/// <summary>

/// 添加谓词,仅能添加一个条件,如果参数值为空，则忽略该条件

/// </summary>

/// <param name="predicate">谓词</param>

/// <param name="isOr">是否使用Or连接</param>

IQuery<TEntity, TKey> Filter( Expression<Func<TEntity, bool>> predicate, bool isOr = false );

/// <summary>

/// 过滤条件

/// </summary>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

/// <param name="operator">运算符</param>

IQuery<TEntity,TKey> Filter( string propertyName, object value, Operator @operator = Operator.Equal );

/// <summary>

/// 添加查询条件

/// </summary>

/// <param name="criteria">查询条件</param>

IQuery<TEntity,TKey> Filter( ICriteria<TEntity> criteria );

/// <summary>

/// 过滤int数值段

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

IQuery<TEntity,TKey> FilterInt<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, int? min,int? max );

/// <summary>

/// 过滤double数值段

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

IQuery<TEntity,TKey> FilterDouble<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, double? min,

double? max );

/// <summary>

/// 过滤日期段，不包含时间

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

IQuery<TEntity,TKey> FilterDate<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, DateTime? min,

DateTime? max );

/// <summary>

/// 过滤日期时间段，包含时间

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

IQuery<TEntity,TKey> FilterDateTime<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression,

DateTime? min, DateTime? max );

/// <summary>

/// 过滤decimal数值段

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

IQuery<TEntity,TKey> FilterDecimal<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, decimal? min,

decimal? max );

/// <summary>

/// 与连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="query">查询对象</param>

IQuery<TEntity,TKey> And( IQuery<TEntity,TKey> query );

/// <summary>

/// 与连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="predicate">谓词</param>

IQuery<TEntity,TKey> And( Expression<Func<TEntity, bool>> predicate );

/// <summary>

/// 或连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="query">查询对象</param>

IQuery<TEntity,TKey> Or( IQuery<TEntity,TKey> query );

/// <summary>

/// 或连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="predicate">谓词</param>

IQuery<TEntity,TKey> Or( Expression<Func<TEntity, bool>> predicate );

/// <summary>

/// 添加排序，支持多次调用OrderBy创建多级排序

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="expression">属性表达式</param>

/// <param name="desc">是否降序</param>

IQuery<TEntity,TKey> OrderBy<TProperty>( Expression<Func<TEntity, TProperty>> expression, bool desc = false );

/// <summary>

/// 添加排序，支持多次调用OrderBy创建多级排序

/// </summary>

/// <param name="propertyName">排序属性</param>

/// <param name="desc">是否降序</param>

IQuery<TEntity,TKey> OrderBy( string propertyName, bool desc = false );

/// <summary>

/// 获取列表

/// </summary>

/// <param name="queryable">数据源</param>

List<TEntity> GetList( IQueryable<TEntity> queryable );

/// <summary>

/// 获取分页列表

/// </summary>

/// <param name="queryable">数据源</param>

PagerList<TEntity> GetPagerList( IQueryable<TEntity> queryable );

}

}

[复制代码](javascript:void(0);)

　　查询对象中定义的过滤方法默认都是以And方式连接，所以需要增加专门的Or连接方法。当查询需求比较复杂时，可以创建多个查询对象进行合并，从而创建更为复杂的查询条件。

　　为了使用方便，查询对象本身还提供了获取数据的方法，需要传入IQueryable 对象，以便执行实际的查询操作。

　　在Util.Datas项目Queries目录中，创建查询对象实现类**Query**，代码如下。

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

using Util.Datas.Queries.Criterias;

using Util.Datas.Queries.OrderBys;

using Util.Domains;

using Util.Domains.Repositories;

namespace Util.Datas.Queries {

/// <summary>

/// 查询对象

/// </summary>

/// <typeparam name="TEntity">实体类型</typeparam>

/// <typeparam name="TKey">实体标识类型</typeparam>

public class Query<TEntity, TKey> : Pager, IQuery<TEntity, TKey> where TEntity : class ,IAggregateRoot<TKey> {

#region 构造方法

/// <summary>

/// 初始化查询对象

/// </summary>

public Query() {

OrderBuilder = new OrderByBuilder();

}

/// <summary>

/// 初始化查询对象

/// </summary>

/// <param name="pager">分页对象</param>

public Query( IPager pager ) : this() {

Page = pager.Page;

PageSize = pager.PageSize;

TotalCount = pager.TotalCount;

OrderBy( pager.Order );

}

#endregion

#region 属性

/// <summary>

/// 查询条件

/// </summary>

private ICriteria<TEntity> Criteria { get; set; }

/// <summary>

/// 排序生成器

/// </summary>

private OrderByBuilder OrderBuilder { get; set; }

#endregion

#region GetPredicate(获取谓词)

/// <summary>

/// 获取谓词

/// </summary>

public Expression<Func<TEntity, bool>> GetPredicate() {

if ( Criteria == null )

return null;

return Criteria.GetPredicate();

}

#endregion

#region GetOrderBy(获取排序)

/// <summary>

/// 获取排序

/// </summary>

public string GetOrderBy() {

Order = OrderBuilder.Generate();

if ( string.IsNullOrWhiteSpace( Order ) )

Order = "Id desc";

return Order;

}

#endregion

#region 过滤条件

/// <summary>

/// 添加谓词,仅能添加一个条件,如果参数值为空，则忽略该条件

/// </summary>

/// <param name="predicate">谓词</param>

/// <param name="isOr">是否使用Or连接</param>

public IQuery<TEntity, TKey> Filter( Expression<Func<TEntity, bool>> predicate,bool isOr = false ) {

predicate = QueryHelper.ValidatePredicate( predicate );

if ( predicate == null )

return this;

if ( isOr )

Or( predicate );

else

And( predicate );

return this;

}

/// <summary>

/// 过滤条件

/// </summary>

/// <param name="propertyName">属性名</param>

/// <param name="value">值</param>

/// <param name="operator">运算符</param>

public IQuery<TEntity, TKey> Filter( string propertyName, object value, Operator @operator = Operator.Equal ) {

return Filter( Lambda.ParsePredicate<TEntity>( propertyName, value, @operator ) );

}

/// <summary>

/// 添加查询条件

/// </summary>

/// <param name="criteria">查询条件</param>

public IQuery<TEntity, TKey> Filter( ICriteria<TEntity> criteria ) {

And( criteria.GetPredicate() );

return this;

}

/// <summary>

/// 过滤int数值段

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public IQuery<TEntity, TKey> FilterInt<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, int? min, int? max ) {

return Filter( new IntSegmentCriteria<TEntity, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤double数值段

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public IQuery<TEntity, TKey> FilterDouble<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, double? min, double? max ) {

return Filter( new DoubleSegmentCriteria<TEntity, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤日期段，不包含时间

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public IQuery<TEntity, TKey> FilterDate<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, DateTime? min, DateTime? max ) {

return Filter( new DateSegmentCriteria<TEntity, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤日期时间段，包含时间

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public IQuery<TEntity, TKey> FilterDateTime<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, DateTime? min, DateTime? max ) {

return Filter( new DateTimeSegmentCriteria<TEntity, TProperty>( propertyExpression, min, max ) );

}

/// <summary>

/// 过滤decimal数值段

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="propertyExpression">属性表达式，范例：t => t.Age</param>

/// <param name="min">最小值</param>

/// <param name="max">最大值</param>

public IQuery<TEntity, TKey> FilterDecimal<TProperty>( Expression<Func<TEntity, TProperty>> propertyExpression, decimal? min, decimal? max ) {

return Filter( new DecimalSegmentCriteria<TEntity, TProperty>( propertyExpression, min, max ) );

}

#endregion

#region 连接

/// <summary>

/// 与连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="query">查询对象</param>

public IQuery<TEntity, TKey> And( IQuery<TEntity, TKey> query ) {

return And( query.GetPredicate() );

}

/// <summary>

/// 与连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="predicate">谓词</param>

public IQuery<TEntity, TKey> And( Expression<Func<TEntity, bool>> predicate ) {

if ( Criteria == null ) {

Criteria = new Criteria<TEntity>( predicate );

return this;

}

Criteria = new AndCriteria<TEntity>( Criteria.GetPredicate(), predicate );

return this;

}

/// <summary>

/// 或连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="query">查询对象</param>

public IQuery<TEntity, TKey> Or( IQuery<TEntity, TKey> query ) {

return Or( query.GetPredicate() );

}

/// <summary>

/// 或连接,将传入的查询条件合并到当前对象

/// </summary>

/// <param name="predicate">谓词</param>

public IQuery<TEntity, TKey> Or( Expression<Func<TEntity, bool>> predicate ) {

if ( Criteria == null ) {

Criteria = new Criteria<TEntity>( predicate );

return this;

}

Criteria = new OrCriteria<TEntity>( Criteria.GetPredicate(), predicate );

return this;

}

#endregion

#region OrderBy(排序)

/// <summary>

/// 添加排序，支持多次调用OrderBy创建多级排序

/// </summary>

/// <typeparam name="TProperty">属性类型</typeparam>

/// <param name="expression">属性表达式</param>

/// <param name="desc">是否降序</param>

public IQuery<TEntity, TKey> OrderBy<TProperty>( Expression<Func<TEntity, TProperty>> expression, bool desc = false ) {

return OrderBy( Lambda.GetName( expression ), desc );

}

/// <summary>

/// 添加排序，支持多次调用OrderBy创建多级排序

/// </summary>

/// <param name="propertyName">排序属性</param>

/// <param name="desc">是否降序</param>

public IQuery<TEntity, TKey> OrderBy( string propertyName, bool desc = false ) {

OrderBuilder.Add( propertyName, desc );

GetOrderBy();

return this;

}

#endregion

#region Clear(清理)

/// <summary>

/// 清理

/// </summary>

public void Clear() {

Criteria = null;

OrderBuilder = new OrderByBuilder();

}

#endregion

#region GetList(获取列表)

/// <summary>

/// 获取列表

/// </summary>

/// <param name="queryable">数据源</param>

public List<TEntity> GetList( IQueryable<TEntity> queryable ) {

return Execute( queryable ).OrderBy( Order ).ToList();

}

/// <summary>

/// 执行过滤和分页

/// </summary>

private IQueryable<TEntity> Execute( IQueryable<TEntity> queryable ) {

queryable.CheckNull( "queryable" );

queryable = FilterBy( queryable );

GetOrderBy();

return queryable;

}

/// <summary>

/// 过滤

/// </summary>

private IQueryable<TEntity> FilterBy( IQueryable<TEntity> queryable ) {

if ( Criteria == null )

return queryable;

return queryable.Where( Criteria.GetPredicate() );

}

#endregion

#region GetPagerList(获取分页列表)

/// <summary>

/// 获取分页列表

/// </summary>

/// <param name="queryable">数据源</param>

public PagerList<TEntity> GetPagerList( IQueryable<TEntity> queryable ) {

return Execute( queryable ).PagerResult( this );

}

#endregion

}

}

[复制代码](javascript:void(0);)

　　应用层服务可以这样查询。

[复制代码](javascript:void(0);)

public PagerList<ApplicationDto> Query( ApplicationQuery query ) {

return new Query<Application>(query)

.Filter( t => t.Code == query.Code )

.Filter( t => t.Name.Contains( query.Name ) )

.Filter( t => t.Enabled == query.Enabled )

.FilterDate( t => t.CreateTime, query.BeginCreateTime, query.EndCreateTime )

.GetPagerList( Repository.Find() ).Convert( t => t.ToDto() );

}

[复制代码](javascript:void(0);)

或

[复制代码](javascript:void(0);)

public PagerList<ApplicationDto> Query2( ApplicationQuery query ) {

var queryObject = new Query<Application>( query )

.Filter( t => t.Code == query.Code )

.Filter( t => t.Name.Contains( query.Name ) )

.Filter( t => t.Enabled == query.Enabled )

.FilterDate( t => t.CreateTime, query.BeginCreateTime, query.EndCreateTime );

return Repository.PagerQuery( queryObject ).Convert( t => t.ToDto() );

}

[复制代码](javascript:void(0);)

[**Util应用程序框架公共操作类(十二):Lambda表达式公共操作类(三)**](http://www.cnblogs.com/xiadao521/p/4245361.html)

　　今天在开发一个简单查询时，发现我的Lambda操作类的GetValue方法无法正确获取枚举类型值，以至查询结果错误。

　　我增加了几个单元测试来捕获错误，代码如下。

[复制代码](javascript:void(0);)

　　　　 /// <summary>

/// 测试值为枚举

/// </summary>

[TestMethod]

public void TestGetValue\_Enum() {

var test1 = new Test1();

test1.NullableEnumValue = LogType.Error;

//属性为枚举,值为枚举

Expression<Func<Test1, bool>> expression = test => test.EnumValue == LogType.Debug;

Assert.AreEqual( LogType.Debug.Value(), Lambda.GetValue( expression ) );

//属性为枚举,值为可空枚举

expression = test => test.EnumValue == test1.NullableEnumValue;

Assert.AreEqual( LogType.Error, Lambda.GetValue( expression ) );

//属性为可空枚举,值为枚举

expression = test => test.NullableEnumValue == LogType.Debug;

Assert.AreEqual( LogType.Debug, Lambda.GetValue( expression ) );

//属性为可空枚举,值为可空枚举

expression = test => test.NullableEnumValue == test1.NullableEnumValue;

Assert.AreEqual( LogType.Error, Lambda.GetValue( expression ) );

//属性为可空枚举,值为null

test1.NullableEnumValue = null;

expression = test => test.NullableEnumValue == test1.NullableEnumValue;

Assert.AreEqual( null, Lambda.GetValue( expression ) );

}

[复制代码](javascript:void(0);)

　　单元测试成功捕获了Bug，我打开Lambda操作类，准备修改GetValue方法，代码见[Util应用程序框架公共操作类(八):Lambda表达式公共操作类(二)](http://www.cnblogs.com/xiadao521/p/4206696.html)。

　　面对GetValue杂乱无章的代码，我顿时感觉无法下手，必须彻底重构它。

　　我之前也看过一些Lambda表达式解析的代码和文章，基本都是使用NodeType来进行判断。我一直没有使用这种方式，是因为NodeType数量庞大，并且多种NodeType可能转换为同一种Expression类型。我当时认为用switch判断NodeType工作量太大，所以直接采用As转换为特定表达式，再判断是否空值。

　　我把这种山寨方法称为瞎猫碰到死耗子，主要依靠单元测试来捕获需求，通过断点调试，我可以知道转换为哪种特定表达式。这种方法在前期看上去貌似很有效，比判断NodeType的代码要少，但由于使用表达式的方式千差万别，负担越来越重，以至无法维护了。

　　为了彻底重构GetValue方法，我需要补充一点表达式解析的知识，我打开开源框架linq2db，仔细观察他是如何解析的。终于看出点眉目，依靠NodeType进行**递归**判断。

　　我以前只知道使用NodeType进行判断，但不知道应该采用递归的方式，真是知其然不知其所以然。

　　我对GetValue进行了重构，代码如下。

[复制代码](javascript:void(0);)

　　　　 /// <summary>

/// 获取值,范例：t => t.Name == "A",返回 A

/// </summary>

/// <param name="expression">表达式,范例：t => t.Name == "A"</param>

public static object GetValue( Expression expression ) {

if ( expression == null )

return null;

switch ( expression.NodeType ) {

case ExpressionType.Lambda:

return GetValue( ( (LambdaExpression)expression ).Body );

case ExpressionType.Convert:

return GetValue( ( (UnaryExpression)expression ).Operand );

case ExpressionType.Equal:

case ExpressionType.NotEqual:

case ExpressionType.GreaterThan:

case ExpressionType.LessThan:

case ExpressionType.GreaterThanOrEqual:

case ExpressionType.LessThanOrEqual:

return GetValue( ( (BinaryExpression)expression ).Right );

case ExpressionType.Call:

return GetValue( ( (MethodCallExpression)expression ).Arguments.FirstOrDefault() );

case ExpressionType.MemberAccess:

return GetMemberValue( (MemberExpression)expression );

case ExpressionType.Constant:

return GetConstantExpressionValue( expression );

}

return null;

}

/// <summary>

/// 获取属性表达式的值

/// </summary>

private static object GetMemberValue( MemberExpression expression ) {

if ( expression == null )

return null;

var field = expression.Member as FieldInfo;

if ( field != null ) {

var constValue = GetConstantExpressionValue( expression.Expression );

return field.GetValue( constValue );

}

var property = expression.Member as PropertyInfo;

if ( property == null )

return null;

var value = GetMemberValue( expression.Expression as MemberExpression );

return property.GetValue( value );

}

/// <summary>

/// 获取常量表达式的值

/// </summary>

private static object GetConstantExpressionValue( Expression expression ) {

var constantExpression = (ConstantExpression)expression;

return constantExpression.Value;

}

[复制代码](javascript:void(0);)

　　运行了全部测试，全部通过，说明没有影响之前的功能。这正是自动化回归测试的威力，如果没有单元测试，我哪里敢重构这些代码呢。另外，修改Bug采用TDD的方式，能够一次修复，永绝后患，值得你拥有。

　　同时，我还重构了其它类似的代码，就不再贴出，下次我发放源码时，有兴趣可以看看。