[kQL.orm 使用手册（v1.0）](#header-n3)  
 [简介](#header-n5)  
 [kQL.orm须知](#header-n8)  
 [kQL.orm概述](#header-n14)  
 [kQL.orm下载](#header-n46)  
 [kQL.orm价格](#header-n49)  
 [目录](#header-n95)  
 [零、典型案例](#header-n96)  
 [一、快速入门](#header-n126)  
 [二、基础篇](#header-n172)  
 [1、实体生成（kQL.orm.cmdTool.exe）](#header-n173)  
 [2、必须知道的三大类型（DyQuery、Dy、DyResult）](#header-n179)  
 [3、弄明白Lambda推导形参（表别名），【很重要】](#header-n189)  
 [4、IDyQuery接口，定义查询接口](#header-n198)  
 [5、IProc接口，执行存储过程](#header-n207)  
 [6、IInsertInst接口，执行插入操作](#header-n218)  
 [7、IDelete、IDeleteInst接口，执行删除操作](#header-n230)  
 [8、IUpdate、ISet、IUpdateInst接口，执行更新操作](#header-n239)  
 [9、ISelect接口，执行查询操作](#header-n249)  
 [10、IOrder接口，指定排序](#header-n254)  
 [11、IJoin接口，连接查询](#header-n258)  
 [12、IWhere接口、条件过滤](#header-n262)  
 [13、IGroup、IHaving、分组过滤](#header-n271)  
 [14、强大的DyResult，继承自IDyResult接口](#header-n276)  
 [三、进阶篇](#header-n292)  
 [1、开发过程中调试](#header-n293)  
 [2、版本控制](#header-n301)  
 [3、表、视图、存储过程约定规则](#header-n325)  
 [4、存储过程允许返回多张表](#header-n333)  
 [5、In子查询，exists子查询](#header-n340)  
 [6、Update..From..方式更新数据](#header-n348)  
 [7、当结果集返回多个同名字段时、 默认填充第一个字段](#header-n358)  
 [8、Where条件，bool类型，null类型的处理](#header-n369)  
 [9、动态查询示例](#header-n376)  
 [10、DyResult中的 AsT...方法](#header-n383)  
 [11、通过truncate命令清空表](#header-n392)  
 [12、额外的配置项](#header-n396)  
 [四、扩展函数速查表](#header-n404)  
 [字符串函数](#header-n408)  
 [日期时间函数](#header-n506)  
 [数学函数](#header-n553)  
 [聚合函数](#header-n643)  
 [转型函数](#header-n673)  
 [系统函数](#header-n687)  
 [自定义扩展函数](#header-n709)

# kQL.orm 使用手册（v1.0）

## 简介

**kQL.orm**是基于.Net 4.0平台的，针对MS-SQL数据库开发的一套轻量级数据访问层框架。小巧优雅，更重要的是功能强大到让你**脑洞大开**。几乎无需配置，就能快速上手！

#### kQL.orm须知

1. kQL.orm支持的数据库只有一种SQL Server(2005+)，暂没有计划对其他数据库做扩展支持。  
2. kQL.orm更加注重的是实用性，所以无需与其他ORM产品去做性能的比较（但不等于说kQL.orm的性能就一塌糊涂）。  
3. BUG或改进，可提交： chwmqq@126.com  
4. kQL.orm不免费，按并发数收费，可下载体验版。

**变迁里程：**

1. kQL.orm原名(k.dbtool.engine数据访问中间件)  
2. 2013年6月发布了（k.dbtool.engine v1.0.0.0），基于存储过程的配置调用及代码生成。  
3. ......  
4. 2015年6月发布了（k.dbtool.engine v1.0.0.5），基于Expression。  
5. ......  
6. 2017年6月6日(k.dbtool.engine v1.0.0.9)正式更名为：kQL.orm v1.0，并面向市场开放。  
 正式版version: v1.0.0.0  
 试用版version: v1.0.0.1   
7. 2017年6月9日，添加CopyToRemote功能，重新支持数据库自增列（纠结，本人比较倾向于设计表的时候不用自增列），  
 正式版version: v1.0.0.2  
 试用版version: v1.0.0.3

#### kQL.orm概述

1. kQL.orm产品采用**(db first)**数据库优先模式进行处理，即您需要先建好数据库。
2. 配套模型层生成工具（kQL.orm.cmdTool.exe）。通过该工具可以自动生成实体层（或者叫Entity层或Model层或...），即：表>>实体，视图>>实体，存储过程>> 实体。当数据库表结构修改后，只需重新执行一遍该工具，就自动同步了映射的实体。使用方便，可以使开发者更加专注于数据库本身的设计，更加集中精力关注于多个实体模型之间的逻辑处理。
3. **简化数据访问操作，提升开发效率。支持扩展函数的嵌套调用，支持存储过程、视图、表的操作，支持复杂的嵌套查询（多表join、In子查询、Exists子查询），支持大批量数据插入（bulk insert），支持复杂的更新操作（update..from..）。**
4. 使项目结构和代码更加精炼，减少30%+的数据访问层代码量。
5. 支持实例对象的混合操作（即A类型实例a，B类型实例b，...，可以在同一批次中提交增、删、改），支持实例类型条件的操作（如：更新表无需定义主键，只根据条件）
6. **自动版本控制**（基于数据库TIMESTAMP类型）或（框架内定义的v字段，支持两种类型int,guid。int用于自增长的版本、guid用于无需自增的版本控制）
7. **对数据库内置函数做了扩展，扩展函数可多层嵌套调用，字段支持四则运算**，几乎支持所有的数据库内置函数，包括：字符串函数, 时间日期函数, 数学函数, 转型函数, 聚合函数, 系统函数, 扩展函数
8. **支持上下文环境中调用方法、属性等动态计算的变量。**（如：Where(user=>user.name == GetUserName() )，其中GetUserName()是上下文中的方法。
9. 实例对象的增、删、改，内置事务处理。对于所有的查询，使用的事务隔离级别READ UNCOMMITTED.
10. **支持不同SQL Server数据库之间的数据同步，通过CopyToRemote方法实现数据库之间的同步。**强大之处在于通过定义DyQuery表达式，就可以实现大数据量的复制。很多应用场景下很有用。

#### kQL.orm下载

[kQL.orm v1.0 下载](http://pan.baidu.com/s/1pKF5gmR)

#### kQL.orm价格

支付宝账号：[chwmqq@126.com](mailto:chwmqq@126.com)

|  |  |  |  |
| --- | --- | --- | --- |
| 实例并发数 | 价格 | 适用对象 | license有效期 |
| 无限制并发 | ￥9,999 | 互联网或大型应用项目 | 永久 |
| 500个实例并发 | ￥6,999 | 大型应用项目 | 永久 |
| 100个实例并发 | ￥5,999 | 中大型应用项目 | 永久 |
| 50个实例并发 | ￥2,999 | 适合中小型项目 | 永久 |
| 5个实例并发 | ￥299 | 适合个人项目 | 永久 |
| 1个实例并发 | 免费 | 适合开发测试项目 | 下载体验版 |

\*\*一个license文件只能用于一台机器

## 目录

#### 零、典型案例

批插、和数据同步时注意需要设置超时时间，因为默认只有30秒。

1. 批量插入

//本地库数据初始化 插入10万条用户数据   
 var localDy = new Dy();//本地库执行者   
 int insertCount = 100000;//插入数据的数量  
 Console.WriteLine("开始测试\_导数据,本地库插入{0}条用户数据", insertCount);  
 Console.WriteLine("开始生成数据");  
 List<tb\_user> userlist = new List<tb\_user>();  
 for (int i = 200; i < insertCount + 200; i++)  
 {  
 userlist.Add(  
 new tb\_user  
 {  
 自增NO = i, //自增列 框架不会去插入  
 账号 = string.Format("U{0:D4}", i),  
 密码 = "12345678",  
 用户名 = string.Format("Tester{0:D3}", i),  
 性别 = i % 2 == 0,  
 年龄 = RNG.Next(20, 60),  
 会员等级 = (byte)(RNG.Next(1, 255)),  
 积分 = RNG.Next(1000, 10000),  
 消费能力 = Math.Abs((short)RNG.Next(1, 100)),  
 头像 = 获取头像(i),  
 注册日期 = DateTime.Now  
 });  
 }  
 Console.WriteLine("完成生成数据，开始执行插入本地库");  
 long ms = localDy.BulkInsert(userlist); //本地库插入10万条数据  
 Console.WriteLine("插入本地库{1}条数据执行:{0}毫秒", ms, insertCount);  
 Console.WriteLine();

1. **数据同步**、字段的四则运算

//场景一：本地库与远程库 表结构一致>>单表->同步到远程服务器  
 //kQL.orm.demo命名空间下为本地库的实体类  
 //kQL.orm.remotemodels命名空间下为远程库的实体类  
 var localDy = new Dy();//本地库执行者  
 var remoteDy = new Dy("remoteServer");//远程库执行者  
 Console.WriteLine("开始执行导入远程服务器");   
 var query1 = new DyQuery<kQL.orm.demo.tb\_user>().AsQuery();  
 var result = localDy  
 .CopyToRemote<kQL.orm.demo.remotemodels.tb\_user>(query1, remoteDy);  
 Console.WriteLine("完成导入远程服务器，本次执行毫秒:{0}", result.Item1);  
 Console.WriteLine();  
   
 //场景二：本地库与远程库 表结构不一致>>多表的组合->同步到远程服务器  
 //\*\*\*\*灵活定义、快速同步  
 var query2 = new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(JoinWay.InnerJoin, t3 => t3)  
 .On<tb\_order, tb\_order\_detail>((t2, t3) => t2.订单ID == t3.订单ID)   
 .Group(t2 => t2.订单ID).Group(t2 => t2.账号)  
 .Having<tb\_order\_detail>(WhereWay.And,   
 t3 => t3.订单ID.Dy\_Count() > 5  
 ).Select<tb\_order, tb\_order\_detail>(  
 (t2, t3) => new  
 {  
 t2.订单ID,  
 t2.账号,  
 明细数量 = t3.订单ID.Dy\_Count(),  
 总金额Max = t3.支付价.Dy\_Max(),  
 总金额Min = t3.支付价.Dy\_Min(),  
 总金额Sum = t3.支付价.Dy\_Sum(),  
 总金额Avg = t3.支付价.Dy\_Avg(),  
 R金额 = ((t3.支付价.Dy\_Max() + t3.支付价.Dy\_Min() - t3.支付价.Dy\_Sum() \* t3.支付价.Dy\_Avg()) / t3.支付价.Dy\_Min()).Dy\_Convert<decimal, decimal>("decimal(18,2)") //四则运算及转型  
 }  
 ).AsQuery();  
 var result = localDy.CopyToRemote<kQL.orm.demo.remotemodels.tb\_order\_info>(query2, remoteDy);  
 Console.WriteLine("本地多表的组合->同步到远程服务器->本次执行毫秒:{0}", result.Item1);

1. Exists子查询

//Exists 子查询   
 var query5 = new DyQuery<tb\_user>()  
 .Exists(WhereWay.And  
 , new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(JoinWay.InnerJoin, t3 => t3)  
 .On<tb\_order, tb\_order\_detail>((t2, t3) => t2.订单ID == t3.订单ID)  
 //与下面等价  
 //.Join<tb\_order\_detail>(JoinWay.InnerJoin, (t2, t3) => t2.订单ID == t3.订单ID)  
 .Group(t2 => t2.订单ID).Group(t2 => t2.账号)  
 .Having<tb\_order\_detail>(WhereWay.And, t3 => t3.订单ID.Dy\_Count() > 5)  
 //.Select<tb\_order, tb\_order\_detail>((t2, t3) => new { t2.订单ID, t2.账号, 明细数量 = t3.订单ID.Dy\_Count() })  
 .Where<tb\_order, tb\_user>((t2, t1) => t2.账号 == t1.账号)  
 .Select(t2 => 1).AsQuery()  
 )  
 .Select(t1 => new { t1.账号, t1.用户名 })  
 .AsQuery();  
 Console.WriteLine(localDy.Done(query5).AsJson());

1. 嵌套In子查询，扩展函数的嵌套调用

//IN 查询  
 var query4 = new DyQuery<tb\_user>()  
 .Where(t1 => t1.用户名.Dy\_Substring(1, 6).Dy\_Right(1).Dy\_In(new List<string> { "1", "2", "3" }))  
 .Where(t1 => t1.账号.Dy\_In(new DyQuery<tb\_order>(t2 => t2).Select(t2 => t2.账号)))  
 .Select(t1 => t1.账号).AsQuery();  
 Console.WriteLine(localDy.Done(query4).AsJson());

1. 组合的Where条件

var query8 = new DyQuery<tb\_user>()  
 .Where(t1 =>   
 (t1.用户名.Dy\_EndsWith("1") || t1.账号.Dy\_Right(1) == "2")   
 &&   
 (t1.用户名.Dy\_Contains("3") || t1.账号.Dy\_Contains("3"))  
 ).Select(t1 => t1.账号).AsQuery();

#### 一、快速入门

1. 创建数据库 db-demo,执行演示程序中的建表脚本db-demo.sql、其中db-demo-remote.sql用于模拟本地到远程库的数据同步
2. 打开演示项目，这里使用控制台程序，其中依赖组件【kQL.orm.dll】，app.config配置文件，配置连接字符串

<appSettings>   
 <add key="kQL.orm.connstr" value="server=local;database=db\_demo;uid=sa;pwd=sa;"/>   
 </appSettings>

1. 找到模型生成工具中kQL.orm.cmdTool.exe.config文件，配置如下节点

<appSettings>   
 <add key="kQL.orm.connstr" value="server=local;database=db\_demo;uid=sa;pwd=sa;"/>  
 <add key="Namespace" value="k.dbtool.test.demo.models"/>  
 <add key="ModelsOutputs" value="D:\demo\models"/>  
</appSettings>

kQL.orm.connstr：连接字符串（连到刚才那个db-demo库）  
Namespace：您项目中实体层的命名空间名称  
ModelsOutputs：实体文件输出目录

1. 打开kQL.orm.cmdTool.exe，运行modelmake命令，实体层就生成了（切换到demo项目，显示所有文件，将生成的实体类型文件引用进来即可）
2. 在demo项目Program.cs > Main函数中，导入命名空间 kQL.orm.expr、kQL.orm 使用kQL进行数据访问，分为两个步骤： 定义查询，即 var dyQuery = new kQL.orm.expr.DyQuery().AsQuery(); 执行查询，即 var result = new kQL.orm.Dy().Done(dyQuery);
3. 增加演示

//单条插入、添加一个用户  
 var user1 = new tb\_user  
 {  
 账号 = string.Format("U{0:D4}", 1),  
 密码 = "12345678",  
 用户名 = string.Format("Tester{0:D3}", 1),  
 性别 = true,  
 年龄 = new Random().Next(20, 60),  
 会员等级 = (byte)(new Random().Next(1, 255)),  
 积分 = new Random().Next(1000, 10000),  
 消费能力 = Math.Abs((short)new Random().Next(1, 100)),  
 头像 = 获取头像(1),  
 注册日期 = DateTime.Now  
 };  
 var query1 = new DyQuery<tb\_user>().Insert(user1).AsQuery();  
 var result = new Dy().Done(query1);  
 Console.WriteLine(result.AsJson());

1. 查询演示

//根据账号查找用户  
 var query1 = new DyQuery<tb\_user>().Where(t1 => t1.账号 == "U0001").AsQuery();  
 var result1 = new Dy().Done(query1);  
 Console.WriteLine(result1.AsJson());

1. 修改演示

/\*方法一：通过实体修改\*/  
 //根据账号查找用户  
 var query1 = new DyQuery<tb\_user>().Where(t1 => t1.账号 == "U0001").AsQuery();  
 var result1 = new Dy().Done(query1);  
 Console.WriteLine(result1.AsJson());  
 var user = result1.AsT<tb\_user>();  
 //修改用户密码  
 user.密码 = "00000000";  
 query1 = new DyQuery<tb\_user>().Update(user).AsQuery();  
 result1 = new Dy().Done(query1);  
 Console.WriteLine(result1.AsJson());  
  
 /\*方法二：通过条件修改\*/  
 query1 = new DyQuery<tb\_user>().Update(t1=>t1.密码 == "00000000").Where(t1 => t1.账号 == "U0001").AsQuery();  
 result1 = new Dy().Done(query1);  
 Console.WriteLine(result1.AsJson());

1. 删除演示

/\*方法一：通过实体删除\*/  
 //根据账号查找用户  
 var query1 = new DyQuery<tb\_user>().Where(t1 => t1.账号 == "U0001").AsQuery();  
 var result1 = new Dy().Done(query1);  
 Console.WriteLine(result1.AsJson());  
 var user = result1.AsT<tb\_user>();   
 query1 = new DyQuery<tb\_user>().Delete(user).AsQuery();  
 result1 = new Dy().Done(query1);  
 Console.WriteLine(result1.AsJson());  
  
 /\*方法二：通过条件删除\*/  
 query1 = new DyQuery<tb\_user>().Delete().Where(t1 => t1.账号 == "U0001").AsQuery();  
 result1 = new Dy().Done(query1);  
 Console.WriteLine(result1.AsJson());

#### 二、基础篇

##### 1、实体生成（kQL.orm.cmdTool.exe）

通过命令行工具，可以生成实体模型层代码。几乎支持数据库全部数据类型。

public class tb\_product  
 {  
 [Description]  
 public Guid 产品ID { get; set; }  
 public String 产品名称 { get; set; }  
 public String 分类编号 { get; set; }  
 public Single 进价 { get; set; }  
 public Decimal 运费 { get; set; }  
 public Decimal 税率 { get; set; }  
 public Decimal 零售价 { get; set; }  
 public Double 会员折扣 { get; set; }  
 public String 产品描述\_中文 { get; set; }  
 public String 产品描述\_英文 { get; set; }  
 [Category("XML")]  
 public String RSS { get; set; }  
 public Byte[] 附件1 { get; set; }  
 public Byte[] 附件2 { get; set; }  
 [Category("TIMESTAMP")]  
 public String 版本 { get; set; }   
 }

1. [Description]标记为主键，如果表有联合主键，则实体类型中将会有多个属性被[Description]特性标签标记。  
2. [Category]标记描述二级类别，目前仅三种 XML、TIMESTAMP、IDENTITY  
3. 数据库XML类型，c#中映射为string类型，当增删改时需要c#类型去映射到数据库类型。  
4. 数据库TIMESTAMP类型，严格来讲应该映射为byte[8],但为了更方便的进行值比较，框架内做了处理，将返回16进制字符串。  
5. IDENTITY自增列

##### 2、必须知道的三大类型（DyQuery、Dy、DyResult）

DyQuery<T>类型：定义各种查询的入口，一般形式：var query = new DyQuery<T>() ; //T为具体的类型  
  
public class DyQuery<TLeft> : AbsDyQuery , ICriteria<TLeft>  
  
public interface ICriteria<TLeft>  
 : IDyQuery  
 , IJoin<TLeft>,IOn<TLeft>  
 , IWhere<TLeft>, IGroup<TLeft>, IHaving<TLeft>, IOrder<TLeft>, ISelect<TLeft>  
 , IInsertInst<TLeft>  
 , IDelete<TLeft> , IDeleteInst<TLeft>  
 , IUpdate<TLeft>, IUpdateInst<TLeft>,ISet<TLeft>  
 , ITruncate<TLeft>  
 , IProc<TLeft>

Dy类型：执行DyQuery的操作类，真正执行数据库访问操作，一般形式：var result = new Dy().Done(query); //返回DyResult类型  
  
public class Dy  
 {  
 public DyResult Query(IDyQuery dyQuery);  
 public void BulkInsert<T>(List<T> models, int perCommitRowCount = 102400, bool tableLock = true); //批量插入  
 /// <summary>  
 /// 返回 Item1:执行毫秒数,Item2:执行的消息  
 /// </summary>   
 public Tuple<long,string> CopyToRemote<T>(IDyQuery sourceDyQuery, Dy destinationDy) where T : class;  
 }

DyResult类型：查询的结果，执行dy操作后，将结果集DataSet保存至DyResult内部，DyResult提供了很多将结果集转成特定结构的函数，一般形式：  
 1. result.AsJson();//将结果转为json字符串，byte[]将进行Base64编码，除了byte[8]将返回16进制字符串（TIMESTAMP）  
 2. result.TList<T>();//将结果转为对象列表  
 3. 更多.....  
   
 public interface IDyResult  
 {  
 object AsRC(int tableIndex = 0);  
 string AsJson();  
 T AsT<T>(int tableIndex = 0) where T : new();  
 List<T> AsTList<T>(int tableIndex = 0) where T : new();  
 PagedList<T> AsTPageList<T>(int tableIndex = 0) where T : new();  
 //TreeNode<T> AsTree<T>(Expression<Func<T, T, bool>> mapping = null, int tableIndex = 0) where T : new();  
 TreeNode<T> AsTree<T>(Expression<Func<T, T, bool>> mapping = null, string dictClassName = "", object dictClassVal = null, int tableIndex = 0) where T : new();  
 Dictionary<string, ArrayList> AsKeyValues(int tableIndex = 0);  
 dynamic AsDyT(int tableIndex = 0);  
 List<dynamic> AsDyTList(int tableIndex = 0);  
  
 Tuple<T0, T1> AsT\_OneOne<T0, T1>(int tableIndex = 0) where T0 : new() where T1 : new();  
 Tuple<T0, T1, T2> AsT\_OneOne<T0, T1, T2>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new();  
 Tuple<T0, T1, T2, T3> AsT\_OneOne<T0, T1, T2, T3>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new();  
 Tuple<T0, T1, T2, T3, T4> AsT\_OneOne<T0, T1, T2, T3, T4>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new();  
 Tuple<T0, T1, T2, T3, T4, T5> AsT\_OneOne<T0, T1, T2, T3, T4, T5>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new();  
 Tuple<T0, T1, T2, T3, T4, T5, T6> AsT\_OneOne<T0, T1, T2, T3, T4, T5, T6>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new();  
 Tuple<T0, T1, T2, T3, T4, T5, T6, T7> AsT\_OneOne<T0, T1, T2, T3, T4, T5, T6, T7>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new() where T7 : new();  
  
 List<Tuple<T0, T1>> AsTList\_OneOne<T0, T1>(int tableIndex = 0) where T0 : new() where T1 : new();  
 List<Tuple<T0, T1, T2>> AsTList\_OneOne<T0, T1, T2>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new();  
 List<Tuple<T0, T1, T2, T3>> AsTList\_OneOne<T0, T1, T2, T3>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new();  
 List<Tuple<T0, T1, T2, T3, T4>> AsTList\_OneOne<T0, T1, T2, T3, T4>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new();  
 List<Tuple<T0, T1, T2, T3, T4, T5>> AsTList\_OneOne<T0, T1, T2, T3, T4, T5>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new();  
 List<Tuple<T0, T1, T2, T3, T4, T5, T6>> AsTList\_OneOne<T0, T1, T2, T3, T4, T5, T6>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new();  
 List<Tuple<T0, T1, T2, T3, T4, T5, T6, T7>> AsTList\_OneOne<T0, T1, T2, T3, T4, T5, T6, T7>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new() where T7 : new();  
  
 Tuple<One, List<Many>> AsT\_OneMany<One, Many>(Expression<Func<One, One>> oneGroupExpr = null, int tableIndex = 0) where One : new() where Many : new();  
 List<Tuple<One, List<Many>>> AsTList\_OneMany<One, Many>(Expression<Func<One, One>> oneGroupExpr = null, int tableIndex = 0) where One : new() where Many : new();  
   
 }

综上，不管怎样，当你需要访问数据库时就3个步骤

1. var query = new DyQuery<T>().AsQuery();//定义DyQuery  
2. var result = new Dy().Done(query);//执行数据库访问  
3. var jsonResult = result.AsJson();//将结果集，转成json字符串

##### 3、弄明白Lambda推导形参（表别名），【很重要】

直接上代码说明：

//DyQuery<tb\_user>未指定别名，默认为t1,即指定tb\_user表的别名为t1  
 var query = new DyQuery<tb\_user>()   
 .Where(t1 => t1.用户名=="Test" )  
 .Select(t1 => t1.账号)  
 .AsQuery();  
   
 //DyQuery<tb\_user>显式指定别名t1,即指定tb\_user表的别名为t1  
 var query = new DyQuery<tb\_user>(t1=>t1)  
 .Where(t1 => t1.用户名=="Test" )  
 .Select(t1 => t1.账号)  
 .AsQuery();  
   
 //DyQuery<tb\_user>显式指定别名t2,即指定tb\_user表的别名为t2  
 var query = new DyQuery<tb\_user>(t2=>t2)  
 .Where(t2 => t2.用户名=="Test" )  
 .Select(t2 => t2.账号)  
 .AsQuery();   
   
 //DyQuery<tb\_user>显式指定别名t2,Join<tb\_user>表的别名为t3  
 //自连接的操作，其中第一个tb\_user别名为t2，第二个tb\_user别名为t3  
 var query = new DyQuery<tb\_user>(t2=>t2)   
 .Join<tb\_user>(JoinWay.InnerJoin,   
 (t2, t3) => t2.用户名 == t3.用户名  
 )  
 .Where(t2 => t2.用户名=="Test" )  
 .Select(t2 => t2.账号)  
 .AsQuery();   
   
 //更复杂点的例子，DyQuery<tb\_user>未指定tb\_user别名，默认为t1;Exists子查询中，DyQuery<tb\_order>显式指定别名为t2，Join<tb\_order\_detail>别名为t3; 通过Where<tb\_order, tb\_user>((t2, t1) => t2.账号 == t1.账号) 显式指定表别名，实现跨表条件组合查询。  
 var query5 = new DyQuery<tb\_user>()  
 .Exists(WhereWay.And  
 , new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(JoinWay.InnerJoin, t3 => t3).On<tb\_order, tb\_order\_detail>((t2, t3) => t2.订单ID == t3.订单ID)  
 //与下面等价  
 //.Join<tb\_order\_detail>(JoinWay.InnerJoin, (t2, t3) => t2.订单ID == t3.订单ID)  
 .Group(t2 => t2.订单ID).Group(t2 => t2.账号)  
 .Having<tb\_order\_detail>(WhereWay.And, t3 => t3.订单ID.Dy\_Count() > 5)  
 //.Select<tb\_order, tb\_order\_detail>((t2, t3) => new { t2.订单ID, t2.账号, 明细数量 = t3.订单ID.Dy\_Count() })  
 .Where<tb\_order, tb\_user>((t2, t1) => t2.账号 == t1.账号)  
 .Select(t2 => 1).AsQuery()  
 )  
 .Select(t1 => new { t1.账号, t1.用户名 })  
 .AsQuery();

当指定了表别名后，之后的Join,Where,Group,Select等方法中，表别名都要保持一致

##### 4、IDyQuery接口，定义查询接口

/// <summary>  
 /// 标准DyQuery  
 /// </summary>   
 IDyQuery AsQuery();  
 /// <summary>  
 /// 分页DyQuery pageIndex:从1开始  
 /// </summary>   
 IDyQuery AsQueryPaged(int pageIndex, int pageSize);  
 /// <summary>  
 /// TopN DyQuery  
 /// </summary>   
 IDyQuery AsQueryTopN(int N);

var query = new DyQuery()....AsQuery();//通用标准接口，【增、删、改、查】 var query = new DyQuery()....AsQueryPaged();//分页接口，【查】 var query = new DyQuery()....AsQueryTopN();//前N条数据接口，【查】

##### 5、IProc接口，执行存储过程

kQL.orm使用存储过程很方便，因为通过kQL.orm.cmdTool工具，将存储过程生成了实体类。

//数据库中存储过程名：sp\_add\_order 将生成以下实体类  
 public class sp\_add\_order   
 {  
 public Guid 订单号 { get; set; }  
 public String 订单名称 { get; set; }  
 public String 账号 { get; set; }  
 public Decimal 金额 { get; set; }   
 }

//调用无参数存储过程  
 var query1 = new DyQuery<sp\_get\_order\_all>().Proc(new sp\_get\_order\_all { });  
 var result1 = dy.Query(query1);  
 Console.WriteLine(result1.AsJson());  
   
 //调用有参数存储过程  
 var query2 = new DyQuery<sp\_add\_order>().Proc(new sp\_add\_order  
 {  
 订单号 = Guid.NewGuid(),//存储过程的参数  
 订单名称 = "O" + DateTime.Now.ToString("yyyyMMddHHmm"),//存储过程的参数  
 账号 = "U0005", //存储过程的参数  
 金额 = 1200.00m //存储过程的参数  
 });  
 var result2 = dy.Query(query2);  
 Console.WriteLine(result2.AsJson());

必须遵守的约定，自定义的存储过程不能使用 return value 作为返回值，因为这样框架无法接收到返回结果。如果存储过程有值返回，请在存储过程最后一句中使用 select value 的形式，在通过获取result.AsRC()方法获取首行首列，或result.AsT()，或result.AsTList()...等形式

##### 6、IInsertInst接口，执行插入操作

public interface IInsertInst<TLeft> : IDyQuery  
 {  
 IInsertInst<TLeft> Insert(TLeft inst);  
 IInsertInst<TLeft> Insert(List<TLeft> insts);  
 IInsertInst<TLeft> Insert<TRight>(TRight inst);  
 IInsertInst<TLeft> Insert<TRight>(List<TRight> insts);  
 }

Insert支持实体对象或实体对象列表的混合插入，混合插入即可以使用不同类型的实体对象进行插入 Insert插入永远在同一个事务中，框架内自动加了事务。其他的Orm框架可能要要显示定义Transaction。 如大批量插入可以通过，Dy类的BlukInsert方法

//单个实体对象插入  
 var user1 = new tb\_user  
 {  
 账号 = string.Format("U{0:D4}", 1),  
 密码 = "12345678",  
 用户名 = string.Format("Tester{0:D3}", 1),  
 性别 = true,  
 年龄 = new Random().Next(20, 60),  
 会员等级 = (byte)(new Random().Next(1, 255)),  
 积分 = new Random().Next(1000, 10000),  
 消费能力 = Math.Abs((short)new Random().Next(1, 100)),  
 头像 = 获取头像(1),  
 注册日期 = DateTime.Now  
 };  
 var query1 = new DyQuery<tb\_user>().Insert(user1).AsQuery();  
 var result = new Dy().Done(query1);  
 //result.RowCount;//影响的行数

//混合不同类型实体对象插入  
 var multi\_category = new List<tb\_categories> {  
 new tb\_categories {  
 子ID = "100",  
 父ID = "",  
 分类名称="数码",  
 顺序=1  
 }  
 , new tb\_categories {  
 子ID = "100100",  
 父ID = "100",  
 分类名称="电脑",  
 顺序=1  
 }  
 , new tb\_categories {  
 子ID = "100101",  
 父ID = "100",  
 分类名称="手机",  
 顺序=1  
 }  
 , new tb\_categories {  
 子ID = "100100100",  
 父ID = "100100",  
 分类名称="笔记本",  
 顺序=1  
 }  
 , new tb\_categories {  
 子ID = "100100101",  
 父ID = "100100",  
 分类名称="台式机",  
 顺序=1  
 }  
 , new tb\_categories {  
 子ID = "100101100",  
 父ID = "100101",  
 分类名称="苹果",  
 顺序=1  
 }  
 , new tb\_categories {  
 子ID = "100101101",  
 父ID = "100101",  
 分类名称="小米",  
 顺序=1  
 }  
 };  
   
 var multi\_products = new List<int> { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 }.Select(i =>  
 {  
 Thread.Sleep(50);  
 return new tb\_product  
 {  
 产品ID = Guid.NewGuid(),  
 产品名称 = string.Format("Product{0:D6}", i),  
 分类编号 = multi\_category[i % 7].子ID,  
 进价 = (float)new Random().NextDouble(),  
 运费 = (decimal)new Random().NextDouble(),  
 税率 = (decimal)new Random().NextDouble(),  
 零售价 = (decimal)new Random().NextDouble() \* 100,  
 会员折扣 = new Random().NextDouble(),  
 产品描述\_中文 = "中文",  
 产品描述\_英文 = "English",  
 RSS = "<root><product></product></root>",  
 //附件1 = "", //参考[头像]  
 //附件2 = "",  
 //版本 timestamp //无须指定  
 };  
 }).ToList();  
 var dyQuery = new DyQuery<tb\_user>()  
 .Insert(user1).Insert(multi\_user).Insert(multi\_category)  
 .Insert(multi\_products).AsQuery();  
 var result = dy.Query(dyQuery);  
 if(result.RowCount == 1 /\*user1\*/ + multi\_user.Count + multi\_category.Count + multi\_products.Count){  
 //插入成功   
 }else{  
 //插入失败   
 }

//大批量数据插入   
 var multi\_user\_batch = new List<int> { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 }.Select(i =>  
 {  
 Thread.Sleep(50);  
 return new tb\_user  
 {  
 账号 = string.Format("U{0:D4}", i),  
 密码 = "12345678",  
 用户名 = string.Format("Tester{0:D3}", i),  
 性别 = i % 2 == 0,  
 年龄 = new Random().Next(20, 60),  
 会员等级 = (byte)(new Random().Next(1, 255)),  
 积分 = new Random().Next(1000, 10000),  
 消费能力 = Math.Abs((short)new Random().Next(1, 100)),  
 头像 = 获取头像(i),  
 注册日期 = DateTime.Now,   
 };  
 }).ToList();   
 dy.BulkInsert(multi\_user\_batch);

##### 7、IDelete、IDeleteInst接口，执行删除操作

public interface IDelete<TLeft>  
 {  
 ICriteria<TLeft> Delete();  
 }  
 public interface IDeleteInst<TLeft> : IDyQuery  
 {  
 IDeleteInst<TLeft> Delete(TLeft inst);  
 IDeleteInst<TLeft> Delete(List<TLeft> insts);  
 IDeleteInst<TLeft> Delete<TRight>(TRight inst);  
 IDeleteInst<TLeft> Delete<TRight>(List<TRight> insts);  
 }

根据实体对象删除，实体必须定义主键，框架会检测实体T类型的[Description]主键标记，根据主键进行删除 根据条件删除，无须实体定义主键

//根据主键删除  
 var query = new DyQuery<tb\_user>().Delete(new tb\_user { 账号 = "U0001" }).AsQuery();  
 var rowCount = dy.Query(query).RowCount;  
 if (rowCount == 1)  
 {  
 //成功  
 }  
 else {  
 //失败  
 }  
   
 //多实体删除 根据主键  
 List<tb\_user> userList = new List<tb\_user>()  
 {  
 new tb\_user { 账号 = "U0002" },  
 new tb\_user { 账号 = "U0003" }  
 };   
 var query = new DyQuery<tb\_user>().Delete(new tb\_user { 账号 = "U0001" }).Delete(userList).AsQuery();  
 var rowCount = dy.Query(query).RowCount;  
 if (rowCount == 1 /\*new tb\_user { 账号 = "U0001" }\*/+ userList.Count)  
 {  
 //成功  
 }  
 else {  
 //失败  
 }  
   
 //根据条件删除  
 var query = new DyQuery<tb\_user>().Delete().Where(t1 => t1.用户名.Dy\_EndsWith("3") && t1.年龄 > 20).AsQuery();  
 var rowCount = dy.Query(query).RowCount;

##### 8、IUpdate、ISet、IUpdateInst接口，执行更新操作

public interface IUpdate<TLeft>  
 {  
 ISet<TLeft> Update();  
 ICriteria<TLeft> Update(Expression<Func<TLeft, bool>> setExpr);   
 }  
 public interface ISet<TLeft>  
 {  
 ICriteria<TLeft> Set(Expression<Func<TLeft, bool>> setExpr);  
 ICriteria<TLeft> Set<TRight>(Expression<Func<TLeft, TRight, bool>> setExpr);  
 ICriteria<TLeft> Set<TRight1, TRight2>(Expression<Func<TLeft, TRight1, TRight2, bool>> setExpr);  
 ICriteria<TLeft> Set<TRight1, TRight2, TRight3>(Expression<Func<TLeft, TRight1, TRight2, TRight3, bool>> setExpr);  
 ICriteria<TLeft> Set<TRight1, TRight2, TRight3, TRight4>(Expression<Func<TLeft, TRight1, TRight2, TRight3, TRight4, bool>> setExpr);  
 ICriteria<TLeft> Set<TRight1, TRight2, TRight3, TRight4, TRight5>(Expression<Func<TLeft, TRight1, TRight2, TRight3, TRight4, TRight5, bool>> setExpr);  
 }   
 public interface IUpdateInst<TLeft> : IDyQuery  
 {  
 IUpdateInst<TLeft> Update(TLeft inst);  
 IUpdateInst<TLeft> Update(List<TLeft> insts);  
 IUpdateInst<TLeft> Update<TRight>(TRight inst);  
 IUpdateInst<TLeft> Update<TRight>(List<TRight> insts);  
 }

根据实体对象更新，实体必须定义主键，框架会检测实体T类型的[Description]主键标记，根据主键进行更新 根据条件更新，无须实体定义主键 更新字段时使用等号赋值；更新多个字段，使用 | 分割；

//实体更新 通过主键  
 var query = new DyQuery<tb\_user>().AsQuery();  
 var userList = dy.Query(query).AsTList<tb\_user>();  
 foreach (var user in userList)  
 {  
 user.年龄 = user.年龄 + 2;  
 user.用户名 = user.用户名 + "N";  
 }   
 var query0 = new DyQuery<tb\_user>().Update(userList).AsQuery();  
 var rowCount = dy.Query(query0).RowCount;   
   
 //根据条件更新  
 var query = new DyQuery<tb\_user>()  
 .Update(t1 =>   
 t1.账号 == "xxx" | t1.年龄 == 0  
 ).Where(t1 =>   
 t1.账号.Dy\_Contains("00")  
 ).AsQuery();  
 var rowCount = dy.Query(query).RowCount;  
   
 //update from 更新 [产品名称]变更需要将订单明细中[产品名称]同时变更  
 var query = new DyQuery<tb\_order\_detail>().Update()  
 .Set<tb\_product>(  
 (t1, t2) =>   
 t1.产品名称 == t2.产品名称.Dy\_Substring(1, 4)  
 ).Join<tb\_product>(JoinWay.InnerJoin,   
 (t1, t2) => t1.产品ID == t2.产品ID  
 ).Where(t1 => t1.支付价 > 30).AsQuery();  
 var rowCount = dy.Query(query).RowCount;

##### 9、ISelect接口，执行查询操作

public interface ISelect<TLeft>  
 {  
 ICriteria<TLeft> Select();  
 ICriteria<TLeft> Select(Expression<Func<TLeft, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight>(Expression<Func<TRight, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight1, TRight2>(Expression<Func<TRight1, TRight2, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight1, TRight2, TRight3>(Expression<Func<TRight1, TRight2, TRight3, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight1, TRight2, TRight3, TRight4>(Expression<Func<TRight1, TRight2, TRight3, TRight4, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight1, TRight2, TRight3, TRight4, TRight5>(Expression<Func<TRight1, TRight2, TRight3, TRight4, TRight5, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight1, TRight2, TRight3, TRight4, TRight5, TRight6>(Expression<Func<TRight1, TRight2, TRight3, TRight4, TRight5, TRight6, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight1, TRight2, TRight3, TRight4, TRight5, TRight6, TRight7>(Expression<Func<TRight1, TRight2, TRight3, TRight4, TRight5, TRight6, TRight7, dynamic>> selectExpr);  
 ICriteria<TLeft> Select<TRight1, TRight2, TRight3, TRight4, TRight5, TRight6, TRight7, TRight8>(Expression<Func<TRight1, TRight2, TRight3, TRight4, TRight5, TRight6, TRight7, TRight8, dynamic>> selectExpr);  
 }

> Select<tb\_user>(t1=>t1) //查询tb\_user表的所有字段 => select t1.\*  
> Select<tb\_user>(t1=>new tb\_user{ 用户名 = t1.用户名,账号 = t1.账号}) //查询tb\_user表的指定字段 => select t1.用户名,t1.账号  
> Select<tb\_user>(t1=>1) //查询=> select 1  
> Select<tb\_user>(t1=> new { 总计 = DyExtFn.Dy\_CountN(1) ) //查询=> select count(1) as 总计  
> Select<tb\_user>(t1=> new { 总计 = t1.账号.Dy\_CountN(1) ) //查询=> select count(t1.账号) as 总计  
> Select<tb\_order, tb\_order\_detail>((t1, t2) => new { t1.订单ID, t1.账号, 明细数量 = t2.订单ID.Dy\_Count() }) //=> select t1.订单ID, t1.账号,count(t2.订单ID) as 明细数量

##### 10、IOrder接口，指定排序

public interface IOrder<TLeft>  
 {  
 ICriteria<TLeft> Order<TField>(OrderWay orderWay, Expression<Func<TLeft, TField>> orderExpr);  
 ICriteria<TLeft> Order<TRight, TRightField>(OrderWay orderWay, Expression<Func<TRight, TRightField>> orderExpr);  
 }  
   
 //指定排序  
 var query = new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(  
 JoinWay.InnerJoin,   
 (t2, t3) => t2.订单ID == t3.订单ID  
 )  
 .Order<DateTime>(OrderWay.Desc, t2 => t2.订单时间)  
 .Order<Decimal>(OrderWay.Asc, t3 => t3.支付价)

##### 11、IJoin接口，连接查询

public interface IJoin<TLeft>  
 {  
 IOn<TLeft> Join<TRight>(JoinWay joinWay, Expression<Func<TRight, TRight>> aliasExpr);  
 ICriteria<TLeft> Join<TRight>(JoinWay joinWay, Expression<Func<TLeft, TRight, bool>> onExpr);  
 ICriteria<TLeft> Join<TRight1, TRight2>(JoinWay joinWay, Expression<Func<TRight1, TRight2, bool>> onExpr);  
 }  
 public interface IOn<TLeft>  
 {  
 ICriteria<TLeft> On<TRight1, TRight2>(Expression<Func<TRight1, TRight2, bool>> onExpr);  
 }  
   
 var query = new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(JoinWay.InnerJoin, t3 => t3)  
 .On<tb\_order, tb\_order\_detail>((t2, t3) => t2.订单ID == t3.订单ID)

##### 12、IWhere接口、条件过滤

public interface IWhere<TLeft>  
 {  
 #region 基本条件  
 ICriteria<TLeft> Where(Expression<Func<TLeft, bool>> whereExpr);  
 ICriteria<TLeft> Where<TRight>(Expression<Func<TRight, bool>> whereExpr);  
 ICriteria<TLeft> Where<TRight1, TRight2>(Expression<Func<TRight1, TRight2, bool>> whereExpr);  
 //==============================================================>>>>>>>>>>>>>>>>>>>>>>>>>>>  
 ICriteria<TLeft> Where(WhereWay whereWay, Expression<Func<TLeft, bool>> whereExpr);  
 ICriteria<TLeft> Where<TRight>(WhereWay whereWay, Expression<Func<TRight, bool>> whereExpr);  
 ICriteria<TLeft> Where<TRight1, TRight2>(WhereWay whereWay, Expression<Func<TRight1, TRight2, bool>> whereExpr);  
 #endregion  
   
 #region Exists 子查询  
 ICriteria<TLeft> Exists(WhereWay whereWay, IDyQuery existsDyQuery);  
 ICriteria<TLeft> ExistsNot(WhereWay whereWay, IDyQuery existsDyQuery);  
 #endregion   
 }

//条件过滤  
 private static string GetUserId() //上下文计算函数  
 {  
 return "U0005";  
 }  
 var query =  
 //new DyQuery<tb\_order\_detail>()  
 //or  
 new DyQuery<tb\_order\_detail>(t1 => t1) //指定表别名  
 .Join<tb\_order>(JoinWay.InnerJoin, (t1, t2) => t1.订单ID == t2.订单ID)  
 .Join<tb\_product>(JoinWay.InnerJoin, (t1, t3) => t1.产品ID == t3.产品ID)  
 .Join<tb\_order, tb\_user>(JoinWay.InnerJoin, (t2, t4) => t2.账号 == t4.账号)  
 .Where<tb\_user>(t4 => t4.账号 == GetUserId()) //=> GetUserId() /\*上下文计算函数\*/  
 .Where<tb\_order>(t2 =>  
 t2.订单时间.Dy\_DateDiff(DateTime.Now, "dd") >= 0  
 &&  
 t2.订单时间.Dy\_DateAdd(-5, "mm").Dy\_DateDiff(DateTime.Now, "dd") > 0  
 )  
 .Select<tb\_order\_detail, tb\_order, tb\_product, tb\_user>  
 (  
 (t1, t2, t3, t4) =>  
 new  
 {  
 t4.账号,  
 产品简称 = t3.产品名称.Dy\_Substring(1, 2),  
 t2.订单名称,  
 支付价 = t1.支付价.Dy\_M\_Floor(),  
 }  
 )  
 .AsQuery();

Where推导类型如：t1，t2...类型字段，如需通过直接数据库进行字段处理，必须使用扩展函数【Dy\_】开头的扩展函数。 Where中支持上下文环境的计算函数、变量、属性等 如上例：GetUserId()

##### 13、IGroup、IHaving、分组过滤

public interface IGroup<TLeft>  
 {  
 ICriteria<TLeft> Group<TField>(Expression<Func<TLeft, TField>> groupExpr);  
 ICriteria<TLeft> Group<TModel, TField>(Expression<Func<TModel, TField>> groupExpr);  
 }  
   
 public interface IHaving<TLeft>  
 {  
 ICriteria<TLeft> Having(WhereWay whereWay, Expression<Func<TLeft, bool>> havingExpr);  
 ICriteria<TLeft> Having<TModel>(WhereWay whereWay, Expression<Func<TModel, bool>> havingExpr);  
 }

//group,having  
 var query = new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(JoinWay.InnerJoin, t3 => t3) //定义tb\_order\_detail表别名 t3  
 .On<tb\_order, tb\_order\_detail>((t2, t3) => t2.订单ID == t3.订单ID)  
 //与下面等价  
 //.Join<tb\_order\_detail>(JoinWay.InnerJoin, (t2, t3) => t2.订单ID == t3.订单ID)   
 .Group(t2 => t2.订单ID).Group(t2 => t2.账号)  
 .Having<tb\_order\_detail>(WhereWay.And, t3 => t3.订单ID.Dy\_Count() > 5)  
 .Select<tb\_order, tb\_order\_detail>(  
 (t2, t3) => new  
 {  
 t2.订单ID,  
 t2.账号,  
 明细数量 = t3.订单ID.Dy\_Count(),  
 总金额Max = t3.支付价.Dy\_Max(),  
 总金额Min = t3.支付价.Dy\_Min(),  
 总金额Sum = t3.支付价.Dy\_Sum(),  
 总金额Avg = t3.支付价.Dy\_Avg(),  
 R金额 = ((t3.支付价.Dy\_Max() + t3.支付价.Dy\_Min() - t3.支付价.Dy\_Sum() \* t3.支付价.Dy\_Avg()) / t3.支付价.Dy\_Min()).Dy\_Convert<decimal, decimal>("decimal(18,2)")  
 }  
 )  
 .AsQuery();

##### 14、强大的DyResult，继承自IDyResult接口

public interface IDyResult  
 {  
 object AsRC(int tableIndex = 0);//首行首列  
 string AsJson();//转json字符串  
 T AsT<T>(int tableIndex = 0) where T : new();//转对象，通常与 AsQueryTopN(1)一起使用  
 List<T> AsTList<T>(int tableIndex = 0) where T : new();//转对象列表  
 PagedList<T> AsTPageList<T>(int tableIndex = 0) where T : new();//转分页对象列表，通常与AsQueryPaged()一起使用  
 TreeNode<T> AsTree<T>(Expression<Func<T, T, bool>> mapping = null, string dictClassName = "", object dictClassVal = null, int tableIndex = 0) where T : new();//将结果转成树结构，分类数据处理的时候很方便  
 Dictionary<string, ArrayList> AsKeyValues(int tableIndex = 0);//将数据结果转置，画图和统计时比较有用  
 dynamic AsDyT(int tableIndex = 0);//转动态对象  
 List<dynamic> AsDyTList(int tableIndex = 0);//转动态对象列表  
   
 //以下是一对一关系的数据处理  
 Tuple<T0, T1> AsT\_OneOne<T0, T1>(int tableIndex = 0) where T0 : new() where T1 : new();  
 Tuple<T0, T1, T2> AsT\_OneOne<T0, T1, T2>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new();  
 Tuple<T0, T1, T2, T3> AsT\_OneOne<T0, T1, T2, T3>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new();  
 Tuple<T0, T1, T2, T3, T4> AsT\_OneOne<T0, T1, T2, T3, T4>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new();  
 Tuple<T0, T1, T2, T3, T4, T5> AsT\_OneOne<T0, T1, T2, T3, T4, T5>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new();  
 Tuple<T0, T1, T2, T3, T4, T5, T6> AsT\_OneOne<T0, T1, T2, T3, T4, T5, T6>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new();  
 Tuple<T0, T1, T2, T3, T4, T5, T6, T7> AsT\_OneOne<T0, T1, T2, T3, T4, T5, T6, T7>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new() where T7 : new();  
   
 List<Tuple<T0, T1>> AsTList\_OneOne<T0, T1>(int tableIndex = 0) where T0 : new() where T1 : new();  
 List<Tuple<T0, T1, T2>> AsTList\_OneOne<T0, T1, T2>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new();  
 List<Tuple<T0, T1, T2, T3>> AsTList\_OneOne<T0, T1, T2, T3>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new();  
 List<Tuple<T0, T1, T2, T3, T4>> AsTList\_OneOne<T0, T1, T2, T3, T4>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new();  
 List<Tuple<T0, T1, T2, T3, T4, T5>> AsTList\_OneOne<T0, T1, T2, T3, T4, T5>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new();  
 List<Tuple<T0, T1, T2, T3, T4, T5, T6>> AsTList\_OneOne<T0, T1, T2, T3, T4, T5, T6>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new();  
 List<Tuple<T0, T1, T2, T3, T4, T5, T6, T7>> AsTList\_OneOne<T0, T1, T2, T3, T4, T5, T6, T7>(int tableIndex = 0) where T0 : new() where T1 : new() where T2 : new() where T3 : new() where T4 : new() where T5 : new() where T6 : new() where T7 : new();  
   
 //以下是一对多关系的数据处理，oneGroupExpr指定主表的分组字段  
 Tuple<One, List<Many>> AsT\_OneMany<One, Many>(Expression<Func<One, One>> oneGroupExpr = null, int tableIndex = 0) where One : new() where Many : new();  
 List<Tuple<One, List<Many>>> AsTList\_OneMany<One, Many>(Expression<Func<One, One>> oneGroupExpr = null, int tableIndex = 0) where One : new() where Many : new();  
   
 }

DyResult对象有两个属性，

/// <summary>   
 /// 记录影响的行数, 只对 增、删、改 有效，查询返回结果永远是0  
 /// </summary>  
 public int RowCount { get; set; }  
 /// <summary>  
 /// 结果集  
 /// </summary>  
 public DataSet Records { get; set; }

通过实现IDyResult接口，将结果转化为各种常用的数据结构

PagedList<T> AsTPageList<T>();  
PagedList<T>.PageIndex ==>> 分页索引  
PagedList<T>.PageSize ==>> 页大小  
PagedList<T>.TotalCount ==>> 总记录数  
PagedList<T>.TotalPages ==>> 总行数  
PagedList<T>.HasPreviousPage ==>> 是否有前一页  
PagedList<T>.HasNextPage ==>> 是否有后一页

TreeNode<T> AsTree<T>();  
TreeNode<T>.Parent ==>> 父节点T类型对象  
TreeNode<T>.Current ==>> 当前节点对象  
TreeNode<T>.Level ==>> 当前层次  
TreeNode<T>.Flag ==>> 标记，默认-1；剪枝时有用  
TreeNode<T>.IsLeaf ==>> 是否叶子节点  
TreeNode<T>.ParentNode ==>> 父节点TreeNode类型  
TreeNode<T>.Children ==>> 子节点TreeNode类型

具体看demo程序中SampleCode->测试\_结果集

#### 三、进阶篇

##### 1、开发过程中调试

有时我们在开发的过程中查看生成的SQL语句，可以通过如下方式： var query = new DyQuery().AsQuery(); var SQL = query.ToString();//生成的SQL语句

##### 2、版本控制

版本控制通过两种方式进行控制

1. 方法一：设计数据表的时，将字段定义为timestamp类型，实体文件中生成 [Category(TIMESTAMP)] string 类型、将转为16进制字符串
2. 方法二：设计数据表的时，添加v字段，设置为int类型或uniqueidentifier类型；v为其他类型时框架不会截取处理

* int类型，可以作为增长的版本控制，从1开始一直往上累加  
   uniqueidentifier类型，可以作为变化控制，版本不需要累计的时候可以使用

1. 当你使用了版本控制，增、改 数据的时候，改版本字段不需要你去手动维护。
2. 其中TIMESTAMP类型的版本由数据库自动维护；其中v字段为int或uniqueidentifier类型时，框架会截取处理。

使用方式：

public class tb\_product  
 {  
 [Description]  
 public Guid 产品ID { get; set; }  
 public String 产品名称 { get; set; }  
 public String 分类编号 { get; set; }  
 public Single 进价 { get; set; }  
 public Decimal 运费 { get; set; }  
 public Decimal 税率 { get; set; }  
 public Decimal 零售价 { get; set; }  
 public Double 会员折扣 { get; set; }  
 public String 产品描述\_中文 { get; set; }  
 public String 产品描述\_英文 { get; set; }  
 [Category("XML")]  
 public String RSS { get; set; }  
 public Byte[] 附件1 { get; set; }  
 public Byte[] 附件2 { get; set; }  
 [Category("TIMESTAMP")]  
 public String 版本 { get; set; }   
 }  
   
 //TIMESTAMP演示 （v字段类似 ）  
 var query = new DyQuery<tb\_product>()  
 .Where(t1 => t1.产品ID == "E4EB08C5-0CBC-432A-B410-01C40BD9D1E9" ).AsQuery();  
 var product = dy.Query(query).AsT<tb\_product>();  
 //现在需要修改 进价  
 product.进价 = 100;  
 //\*\*并发环境下，另一个线程已经修改了这条数据  
 query = new DyQuery<tb\_product>().Update(product).AsQuery();  
 var rowCount = dy.Query(query).RowCount;//rowCount == 0 修改失败

##### 3、表、视图、存储过程约定规则

//存储过程  
 create proc sp\_get\_order\_all  
 as  
 begin  
 select top 3 \* from tb\_order(nolock)  
 end  
  
 go  
==>>生成实体  
 public class sp\_get\_order\_all  
 {  
   
 }  
  
//视图 => 具体操作跟表实体对象一致   
create view v\_user\_order  
as   
 select t1.订单名称,t1.订单ID,t2.购买数量,t1.账号 from tb\_order t1  
 join tb\_order\_detail t2  
 on t1.订单ID = t2.订单ID  
==>>生成实体  
 public class v\_user\_order  
 {  
 public String 订单名称 { get; set; }  
 public Guid 订单ID { get; set; }  
 public Int32 购买数量 { get; set; }  
 public String 账号 { get; set; }   
 }

\*\*约定，下划线*开头的存储过程，或视图，将不会生成实体文件。有时为了维护方便，会在数据库中建立一些日常维护的脚本，不需要给程序调用，这种场景下，可以使用下划线*开头的存储过程或视图 通常表以tb开头、实体以v开头、存储过程以sp开头

##### 4、存储过程允许返回多张表

IDyResult接口中的大多数方法，都有一个默认形参tableIndex=0，默认取结果集的第一张表。大多数情况都可以得到满足。但有些场景下需要返回多张表，kQL.orm处理的方式，通过存储过程。

//存储过程  
create proc sp\_multi\_tb  
as  
begin  
 select \* from tb\_user  
 select \* from tb\_categories  
 select \* from tb\_product  
end  
==>>生成实体  
 public class sp\_multi\_tb  
 {  
   
 }  
   
 var query = new DyQuery<sp\_multi\_tb>().Proc(new sp\_multi\_tb { }).AsQuery();  
 var result = dy.Query(query);  
 var userList = result4.AsTList<tb\_user>(0);//第一张表的数据  
 var categoryList = result4.AsTList<tb\_categories>(1);//第二张表的数据  
 var productList = result4.AsTList<tb\_product>(2);//第三张表的数据  
 Console.WriteLine("tb\_user count:{0},tb\_categories count:{1},tb\_product count:{2}"  
 , userList.Count, categoryList.Count, productList.Count);

##### 5、In子查询，exists子查询

//IN 查询  
 var query = new DyQuery<tb\_user>().Where(t1 => t1.用户名.Dy\_Right(1).Dy\_In(new List<string> { "1", "2", "3" }))  
 .Select(t1 => t1.账号).AsQuery();   
   
 //Exists 子查询   
 var query = new DyQuery<tb\_user>()  
 .Exists(WhereWay.And  
 , new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(JoinWay.InnerJoin, t3 => t3)  
 .On<tb\_order, tb\_order\_detail>((t2, t3) => t2.订单ID == t3.订单ID)  
 //与下面等价  
 //.Join<tb\_order\_detail>(JoinWay.InnerJoin, (t2, t3) => t2.订单ID == t3.订单ID)  
 .Group(t2 => t2.订单ID).Group(t2 => t2.账号)  
 .Having<tb\_order\_detail>(WhereWay.And, t3 => t3.订单ID.Dy\_Count() > 5)  
 //.Select<tb\_order, tb\_order\_detail>((t2, t3) => new { t2.订单ID, t2.账号, 明细数量 = t3.订单ID.Dy\_Count() })  
 .Where<tb\_order, tb\_user>((t2, t1) => t2.账号 == t1.账号)  
 .Select(t2 => 1).AsQuery()  
 )  
 .Select(t1 => new { t1.账号, t1.用户名 })  
 .AsQuery();

\*\*如果子查询中定义了与父查询相同的表别名会引起别名冲突， .Exists(WhereWay.And, new DyQuery(t2 => t2) ..//定义与父查询不同的别名

##### 6、Update..From..方式更新数据

通过ISet接口实现

var query = new DyQuery<tb\_order\_detail>().Update()  
 .Set<tb\_product>((t1, t2)   
 => t1.产品名称 == t2.产品名称.Dy\_Substring(1, 4)  
 ).Join<tb\_product>(JoinWay.InnerJoin,   
 (t1, t2) => t1.产品ID == t2.产品ID  
 ).Where(t1 => t1.支付价 > 30)  
 .AsQuery();

t1.产品名称 == t2.产品名称.Dy\_Substring(1, 4) //使用t2表进行更新t1表的字段

##### 7、当结果集返回多个同名字段时、 默认填充第一个字段

如果一个存储过程返回多个同名字段

create proc sp\_user\_selfjoin  
as  
begin   
 select \* from tb\_user t1 join tb\_user t2  
 on t1.账号 = t2.账号  
end  
  
==>>结果集中， 将会有多个同名字段，如：账号,账号1,用户名,用户名1  
默认情况下，使用第一个字段填充，即：账号、用户名  
var query = new DyQuery<tb\_user>().Proc(new sp\_user\_selfjoin{} ).AsQuery();  
var user = dy.Query(query).AsT<tb\_user>();//默认使用第一个字段填充 账号、用户名.....  
/\*如果想要取得所有字段，包括 账号1、用户名1 \*/  
使用 AsT\_OneOne方法,返回Tuple元组  
var tupleUser = dy.Query(query).AsT\_OneOne<tb\_user,tb\_user>();  
var user1 = tupleUser.Item1 => 账号、用户名.....  
var user2 = tupleUser.Item2 => 账号1、用户名1.....

另一种方式，使用动态对象

var query = new DyQuery<tb\_user>().Proc(new sp\_user\_selfjoin{} ).AsQuery();  
var user = dy.Query(query).AsDyT<tb\_user>();  
//user.账号  
//user.账号1  
//.......

##### 8、Where条件，bool类型，null类型的处理

\*\*Where条件中bool类型必须使用二元表达式

var query = new DyQuery<tb\_user>()   
 .Where(t1 =>   
 t1.性别 == true //必须使用二元表达式  
 && t1.注册日期 == null // is null  
 && t1.上次登录日期 !=null //is not null  
 ).AsQuery()

##### 9、动态查询示例

业务中页面查询条件动态组合，简单演示动态查询

var criteria = new DyQuery<tb\_user>();  
if(!string.IsNullOrEmpty( Request["user\_name"])){   
 criteria.Where(t1=>t1.用户名 == Request["user\_name"] )  
}  
if(!string.IsNullOrEmpty( Request["user\_age"])){   
 criteria.Where(t1=>t1.年龄 == int.Parse(Request["user\_age"]) )  
}  
...  
var top1Query = criteria.Select(t1=>t1.账号).AsQueryTopN(1);  
var result = dy.Query(top1Query);

##### 10、DyResult中的 AsT...方法

以上所有演示，AsT...相关方法都使用的是kQL.orm.cmdTool生成的实体对象 \*\*您可以自动定义类型进行映射，不一定使用生成的实体对象 \*\*映射的规则，通过结果集字段查找T类型中同名的属性

//如 在项目中定义了一个Order\_DTO,用来接收tb\_order,tb\_order\_detail的结果  
public class Order\_DTO {  
 public string 订单ID{get;set;}  
 public string 订单名称{get;set;}  
 public string 订单明细ID{get;set;}  
 public string 产品名称{get;set;}  
 public decimal 支付价{get;set;}   
}  
  
var query = new DyQuery<tb\_order>(t2 => t2)  
 .Join<tb\_order\_detail>(JoinWay.InnerJoin,   
 (t2, t3) => t2.订单ID == t3.订单ID  
 ).Order<DateTime>(OrderWay.Asc, t2 => t2.订单时间)  
 .Select<tb\_order>(t2 => new { t2.订单ID, t2.账号 ,t2.订单名称})  
 .Select<tb\_order\_detail>(t3 =>   
 new { t3.订单明细ID, t3.产品名称, t3.支付价 }  
 ).AsQuery();  
var dtoList = dy.Query(query).AsTList<Order\_DTO>();

##### 11、通过truncate命令清空表

public interface ITruncate<TLeft> : IDyQuery  
 {  
 IDyQuery Truncate();  
 }  
   
 //清空操作  
 dy.Query(new DyQuery<tb\_user>().Truncate().AsQuery());  
 dy.Query(new DyQuery<tb\_categories>().Truncate());  
 dy.Query(new DyQuery<tb\_order>().Truncate());  
 dy.Query(new DyQuery<tb\_order\_detail>().Truncate());  
 dy.Query(new DyQuery<tb\_product>().Truncate());

##### 12、额外的配置项

kQL.orm.command.timeout：sql command命令执行超时时间，默认30秒 kQL.orm.bulk.timeout：批量插入超时时间，默认0不超时

<appSettings>   
 <add key="kQL.orm.command.timeout" value="30"/>  
 <add key="kQL.orm.bulk.timeout" value="0"/>   
 <add key="kQL.orm.connstr" value="server=local;database=db\_demo;uid=sa;pwd=sa;"/>  
 <--key可以使用你喜欢的名字 new Dy("remoteServer")-->  
 <add key="remoteServer" value="server=local;database=db\_demo\_remote;uid=sa;pwd=sa;"/>  
</appSettings>

#### 四、扩展函数速查表

public static class DyExtFn：所有扩展函数都定义在该类中

##### 字符串函数

|  |  |  |
| --- | --- | --- |
| 扩展函数名 | DB函数名 | 说明 |
| Dy\_Ascii | ascii |  |
| Dy\_Char | char |  |
| Dy\_NChar | nchar |  |
| Dy\_Unicode | unicode |  |
| Dy\_Quotename | quotename |  |
| Dy\_Soundex | soundex |  |
| Dy\_PatIndex | patindex |  |
| Dy\_CharIndex | charindex |  |
| Dy\_Difference | difference |  |
| Dy\_Left | left |  |
| Dy\_Right | right |  |
| Dy\_Len | len |  |
| Dy\_Lower | lower |  |
| Dy\_Upper | upper |  |
| Dy\_LTrim | ltrim |  |
| Dy\_RTrim | rtrim |  |
| Dy\_Reverse | reverse |  |
| Dy\_Space | space | 静态方法， DyExtFn.Dy\_Space(N) |
| Dy\_Str | str |  |
| Dy\_Stuff | stuff |  |
| Dy\_Substring | substring |  |
| Dy\_Replace | replace |  |
| Dy\_Replicate | replicate |  |

##### 日期时间函数

|  |  |  |
| --- | --- | --- |
| 扩展函数名 | DB函数名 | 说明 |
| Dy\_DateAdd | dateadd |  |
| Dy\_DateDiff | datediff |  |
| Dy\_DateName | datename |  |
| Dy\_DatePart | datepart |  |
| Dy\_GetDate | getdate | 静态方法，DyExtFn.Dy\_GetDate() |
| Dy\_Day | day |  |
| Dy\_Month | month |  |
| Dy\_Year | year |  |
| Dy\_GetUtcDate | getutcdate | 静态方法，DyExtFn.Dy\_GetUtcDate() |
| Dy\_TimeFrt | 无 | 上下文中DateTime对象的扩展方法 |

##### 数学函数

|  |  |  |
| --- | --- | --- |
| 扩展函数名 | DB函数名 | 说明 |
| Dy\_M\_Abs | abs |  |
| Dy\_M\_PI | pi | 静态方法，DyExtFn.Dy\_M\_PI() |
| Dy\_M\_Cos | cos |  |
| Dy\_M\_Sin | sin |  |
| Dy\_M\_Cot | cot |  |
| Dy\_M\_Tan | tan |  |
| Dy\_M\_ACos | acos |  |
| Dy\_M\_ASin | asin |  |
| Dy\_M\_ATan | atan |  |
| Dy\_M\_Degrees | degrees |  |
| Dy\_M\_Radians | radians |  |
| Dy\_M\_Exp | exp |  |
| Dy\_M\_Log | log |  |
| Dy\_M\_Log10 | log10 |  |
| Dy\_M\_Ceiling | ceiling |  |
| Dy\_M\_Floor | floor |  |
| Dy\_M\_Power | power |  |
| Dy\_M\_Sqrt | sqrt |  |
| Dy\_M\_Sign | sign |  |
| Dy\_M\_Rand | rand | 静态方法，DyExtFn.Dy\_M\_Rand() |
| Dy\_M\_Round | round |  |

##### 聚合函数

|  |  |  |
| --- | --- | --- |
| 扩展函数名 | DB函数名 | 说明 |
| Dy\_Avg | avg |  |
| Dy\_Count | count |  |
| Dy\_CountN | count | 静态方法，DyExtFn.Dy\_CountN(1) |
| Dy\_Max | max |  |
| Dy\_Min | min |  |
| Dy\_Sum | sum |  |

##### 转型函数

|  |  |  |
| --- | --- | --- |
| 扩展函数名 | DB函数名 | 说明 |
| Dy\_Convert | convert |  |
| Dy\_Cast | cast |  |

##### 系统函数

|  |  |  |
| --- | --- | --- |
| 扩展函数名 | DB函数名 | 说明 |
| Dy\_NewId | newid |  |
| Dy\_IsNumeric | isnumeric |  |
| Dy\_IsNull | isnull |  |
| Dy\_IsDate | isdate |  |

##### 自定义扩展函数

|  |  |  |
| --- | --- | --- |
| 扩展函数名 | DB函数名 | 说明 |
| Dy\_StartsWith | like '[参数]%' |  |
| Dy\_EndsWith | like '%[参数]' |  |
| Dy\_Contains | like '%[参数]%' |  |
| Dy\_StartsWithNot | not like '[参数]%' |  |
| Dy\_EndsWithNot | not like '%[参数]' |  |
| Dy\_ContainsNot | not like '%[参数]%' |  |
| Dy\_StrGT | [字段名] > '[参数]' |  |
| Dy\_StrGE | [字段名] >= '[参数]' |  |
| Dy\_StrLT | [字段名] < '[参数]' |  |
| Dy\_StrLE | [字段名] <= '[参数]' |  |
| Dy\_In | in ([参数1],[参数2],[参数3],...) | 支持子查询 |
| Dy\_InNot | not in ([参数1],[参数2],[参数3],...) | 支持子查询 |