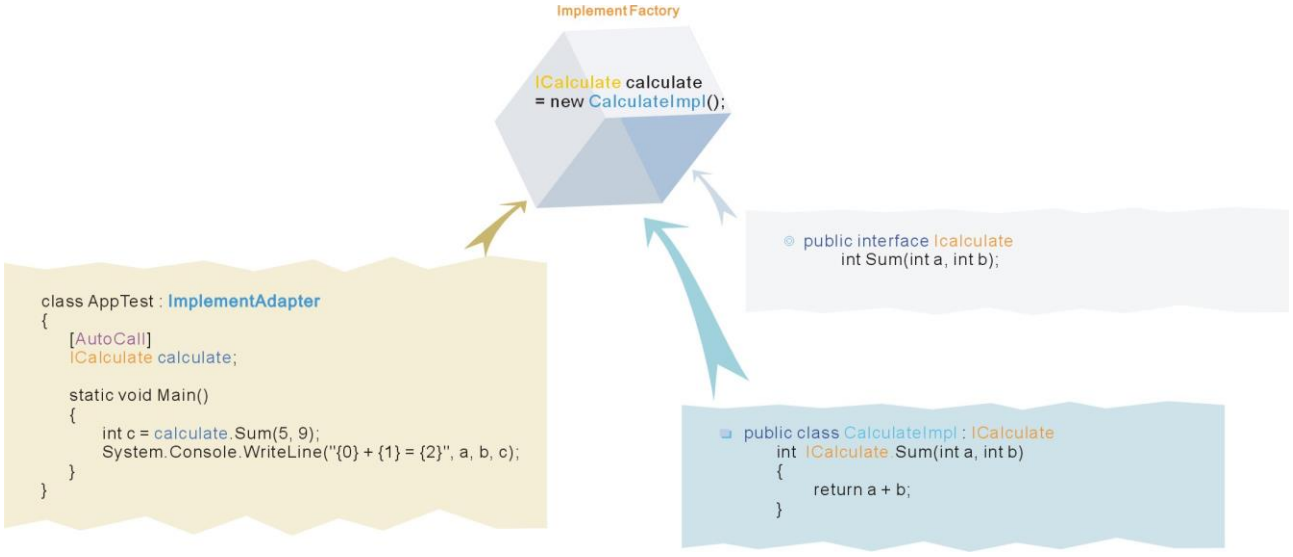
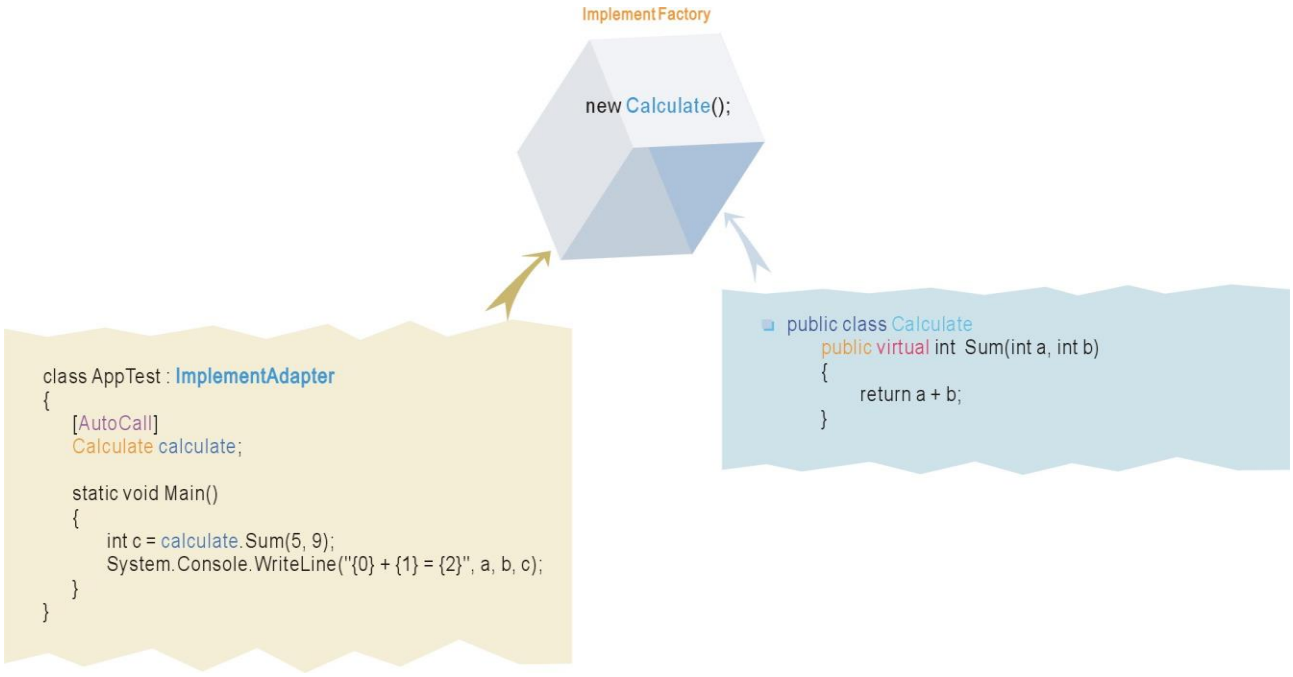
ImplementFactory 技术向导

简介

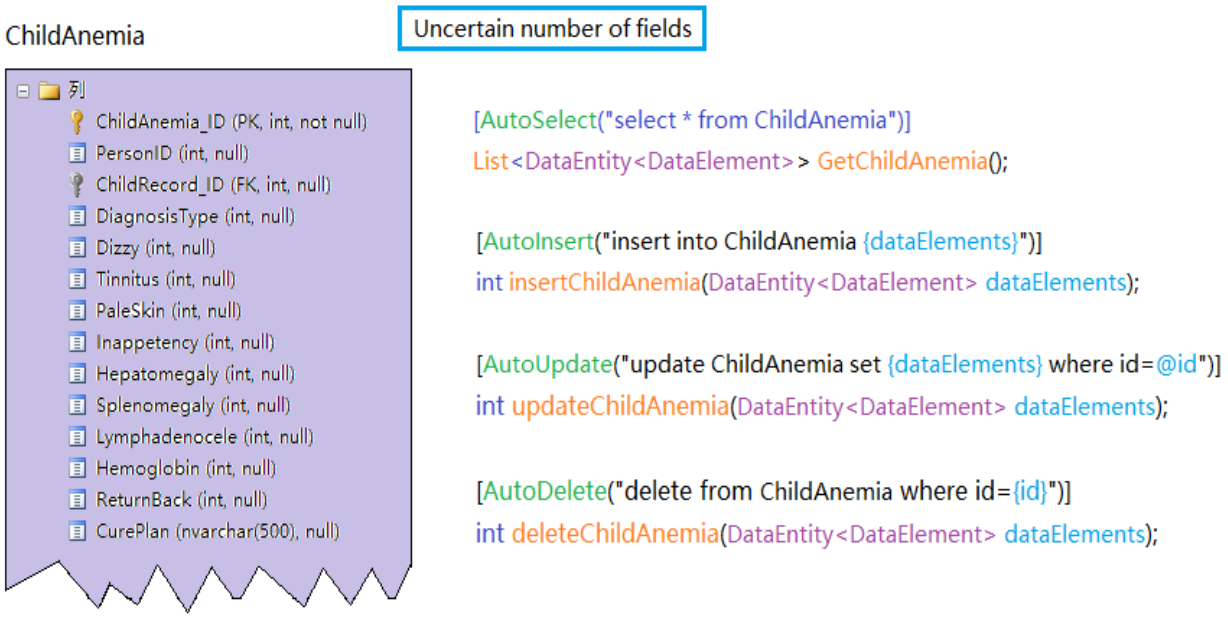
ImplementFactory 组件为业务层级之间的解耦提供了一个很好的解决方案。你只需简单的代码实现就可解除模块之间代码的紧密关联。ImplementFactory 组件强调：每一个业务模块都是一个独立个体，接口实例的加载工作由 ImplementFactory 组件来完成。使用 ImplementFactory 组件可以很方便的实现依赖注入机制(IoC 反转控制)。



也可以是一个无接口的类（需要把 public 方法标识为 virtual 方法）

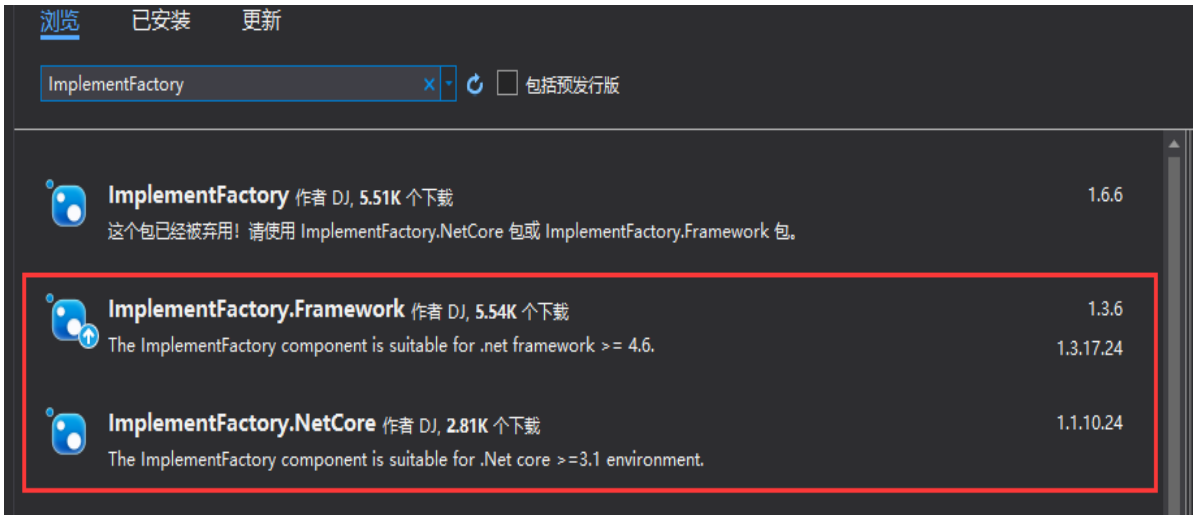


对于动态创建的表，动态创建的字段，不确定字段数量或名称的情况，可以采用 DataEntity<DataElement> 数据集合来操作数据。

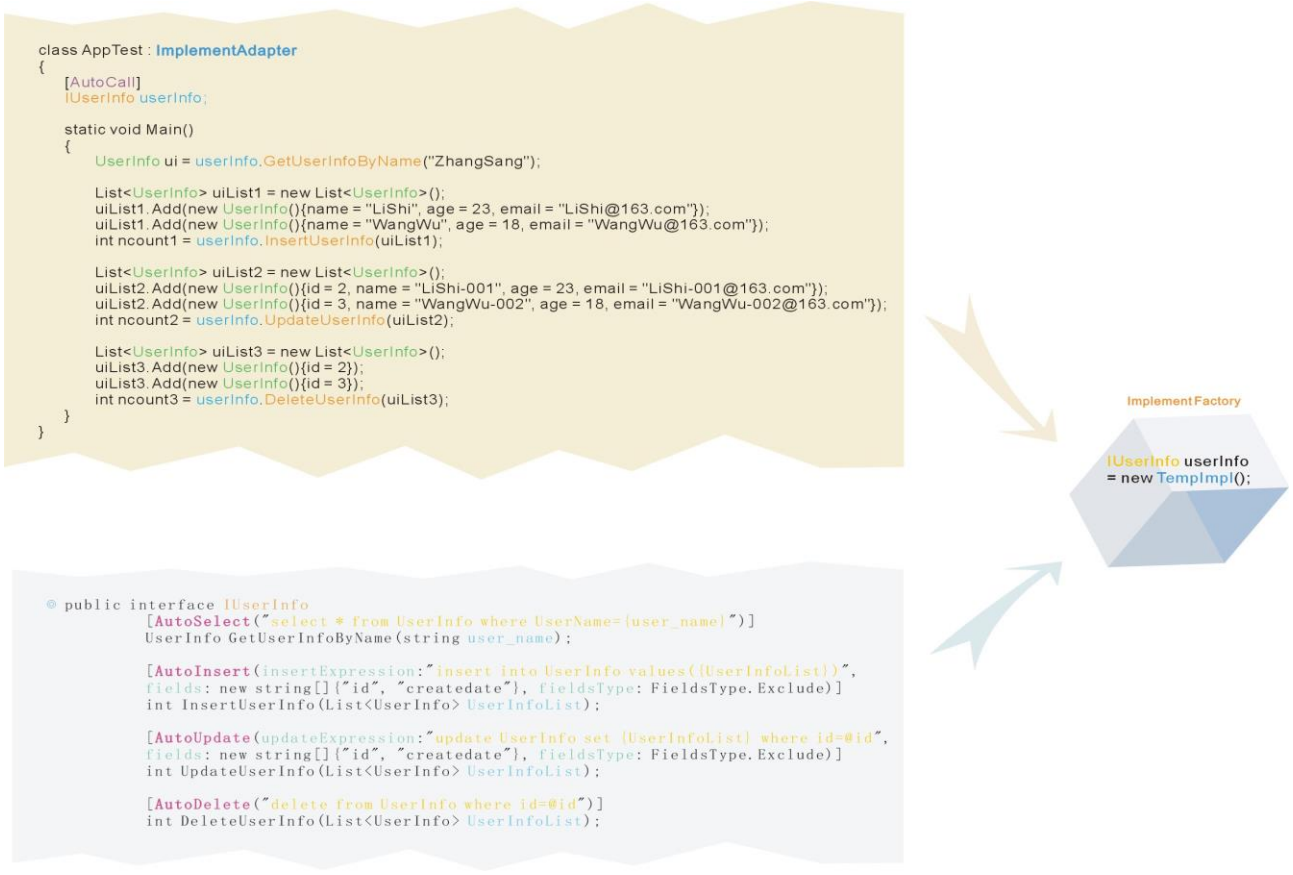


使用 ImplementFactory 组件可以轻松实现接口实例自动装配工作，及面向接口对数据源进行查询、新增、修改、删除的操作。在 dot Net Framework 环境下使用，版本必须大于或等于 v4.6。dot Net Core 环境下，版本必须大于或等于 3.1。

在 Visual Studio 中使用 Nuget 搜索 ImplementFactory 引用该组件，如果你的项目是 .net framework 请选择 ImplementFactory.Framework 包；如果你的项目是 .net core 请选择 ImplementFactory.NetCore 包



ImplementFactory 组件实现面向接口对数据源进行查询(AutoSelect)、新增(AutoInsert)、修改(AutoUpdate)、删除(AutoDelete)、统计记录数量(AutoCount)、执行存储过程(AutoProcedure)的操作：



在使用 AutoSelect 属性时，你还可以采用对象属性动态生成 Where 条件。只需要在 class 或对象属性加入 [Condition] 属性标识，即可动态生成 where 条件，可配合 [FieldMapping] 属性标识一起使用；当在 class 加入 [Condition] 时，该对象内所有满足条件的属性将会被例为 where 条件，单独标识 [Condition]的属性将采用独立指定的相关值，未标识[Condition]的属性满足条件后将采用默认的 WhereIgrone 策略，且默认采用 and 关系连接符。如何在 sql 表达式中使用动态 where 匹配，可查看 <sql 表达式中标识符匹配方式>

例:

数据实体属性

```
[Condition("like", Condition.WhereIgrons.igroneEmptyNull)]
[FieldMapping("telphone")]
public string telphoneNumber { get; set; }
```

数据接口类方法:

```
[AutoSelect("select * from EmployeeInfo where {employeeInfo}")]
List<EmployeeInfo> query(EmployeeInfo employeeInfo);
```

如果 sql 表达式里包含计算公式，你可以采用 **$calculate**(10 * (2 + 1)) 或 **$cal**(10 * (2 + 1)) 运算标签例 select * from UserInfo where 1=1 order by cdatetime asc limit **$cal(10*(1-1))**, 10 运算后结果:
select * from UserInfo where 1=1 order by cdatetime asc limit **0**, 10

接口操作数据还可使用泛型接口或泛型方法参数，泛型的名称可以做为 sql 表达式中的表名，最终采用实际传入的对象类型名称来替换泛型名称。

```
[AutoSelect("select * from {T} order by id desc")]
T query<T>();
```

你也可以把接口声明为泛型接口。当声明为泛型接口时，泛型接口的参数名称必须为大写的 T，如果多个泛型参数侧为多个大写的 T

```
public interface IBaseDataOperate<T, TT, TTT>
{
        //method
       [AutoSelect("select * from {TT} where id={id}")]
       TT getData(int id);

       [AutoInsert("insert into {T} {data}",
           fields: new string[] { "id" },   fieldsType: FieldsType.Exclude)]
       int insert(T data);
}
```

如果所传入的实体对象类型包含 TableAttribute 类型，
即：System.ComponentModel.DataAnnotations.Schema.TableAttribute
那么将采用 Table 属性的 Name 值来替换 sql 表达式中的表名
例：

```
[Table("UserInfo")]
public class UserData
{
    //将采用 UserInfo 字符串来替换 sql 表达式中的 {T}
}
```

在使用 ImplementFactory 组件过程中，创建一个继承 AutoCall 类的子类，可以方便的对每一个接口及接口方法进行有效的控制(AOP)和异常拦截处理。

```
public class myAutoCall: AutoCall
{
    //执行接口实例方法前被调用
    public override bool ExecuteBeforeFilter(Type interfaceType, object implement, string methodName, PList<Para> paras)
    {
        //如果返回 false,则接口方法不被执行
        return true;
    }

    //执行接口实例方法后被调用
    public override bool ExecuteAfterFilter(Type interfaceType, object implement, string methodName, PList<Para> paras, object result)
    {
        //如果返回 false，则执行接口方法后的结果不会返回给调用者
        return true;
    }

    //拦截所有接口实例所发生的异常
    public override void ExecuteException(Type interfaceType, object implement, string methodName, PList<Para> paras, Exception ex)
    {
        base.ExecuteException(interfaceType, implement, methodName, paras, ex);
    }
```

```
}
```

如何使用 myAutoCall

```
class TestUnit : ImplementAdapter
{
    [myAutoCall]
    IMixedJson mixedJson;

    public DJsonItem GetStudentInfoByName(string name)
    {
        return mixedJson.StudentInfoJsonItem(name);
    }
}
```

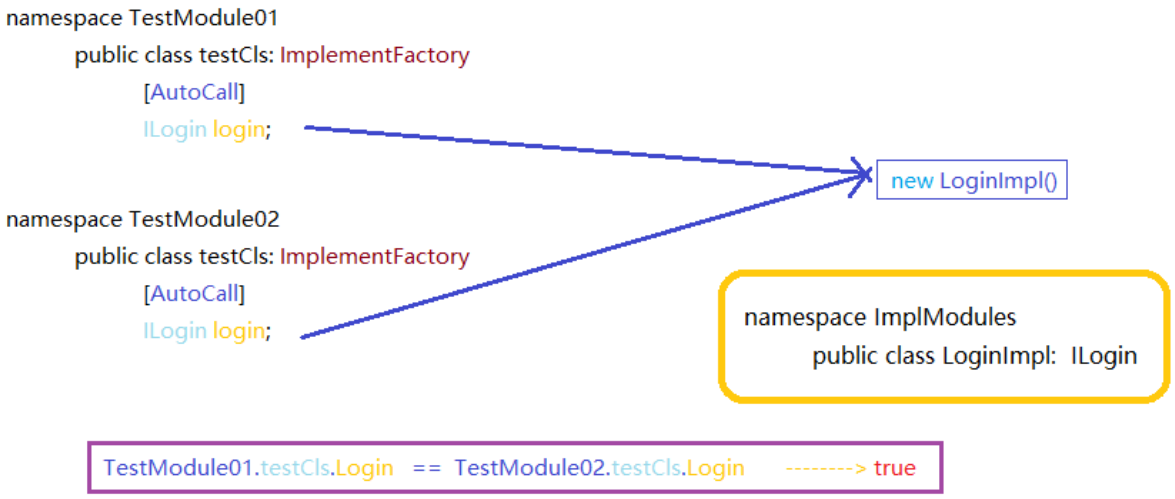
通过继承 AutoCall 类，很方便的实现 AOP 机制，利用 AOP 机制拦截任何一个与之相关的接口实例和接口方法。

以此同时，进行数据操作情况，你可以选择 ImplementFactory 所提供的数据适配器，如果你的业务有特殊需求（**使用数据库类型为非 Microsoft Sql Server 的情况下**），你可以选择自己提供一个有效的数据源适配器，而你只需要实现 IDataServerProvider 接口就可以为 ImplementFactory 提供一个有效的数据适配器。

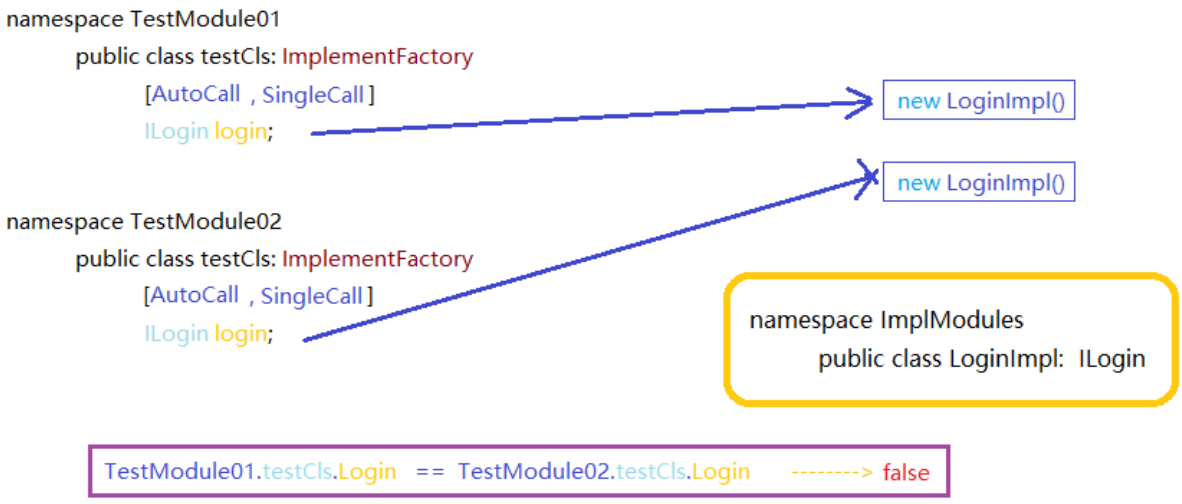
```
using MySql.Data.MySqlClient;

public class DataServerProvider : IDataServerProvider
{
    DataAdapter IDataServerProvider.CreateDataAdapter(DbCommand dbCommand)
    {
        return new MySqlDataAdapter((MySqlCommand)dbCommand);
    }

    DbCommand IDataServerProvider.CreateDbCommand(string sql, DbConnection connection)
    {
        return new MySqlCommand(sql, (MySqlConnection)connection);
    }

    DbConnection IDataServerProvider.CreateDbConnection(string connectString)
    {
        return new MySqlConnection (connectString);
    }

    DbParameter IDataServerProvider.CreateDbParameter(string fieldName, object fieldValue)
    {
        return new MySqlParameter(fieldName, fieldValue);
    }
}
```

ImplementAdapter 首次加载时会加载该 IDataServerProvider 实例。

默认情况下，AutoCall 是采用实例引用的机制（或接口实例继承 IUnSingleInstance 接口）

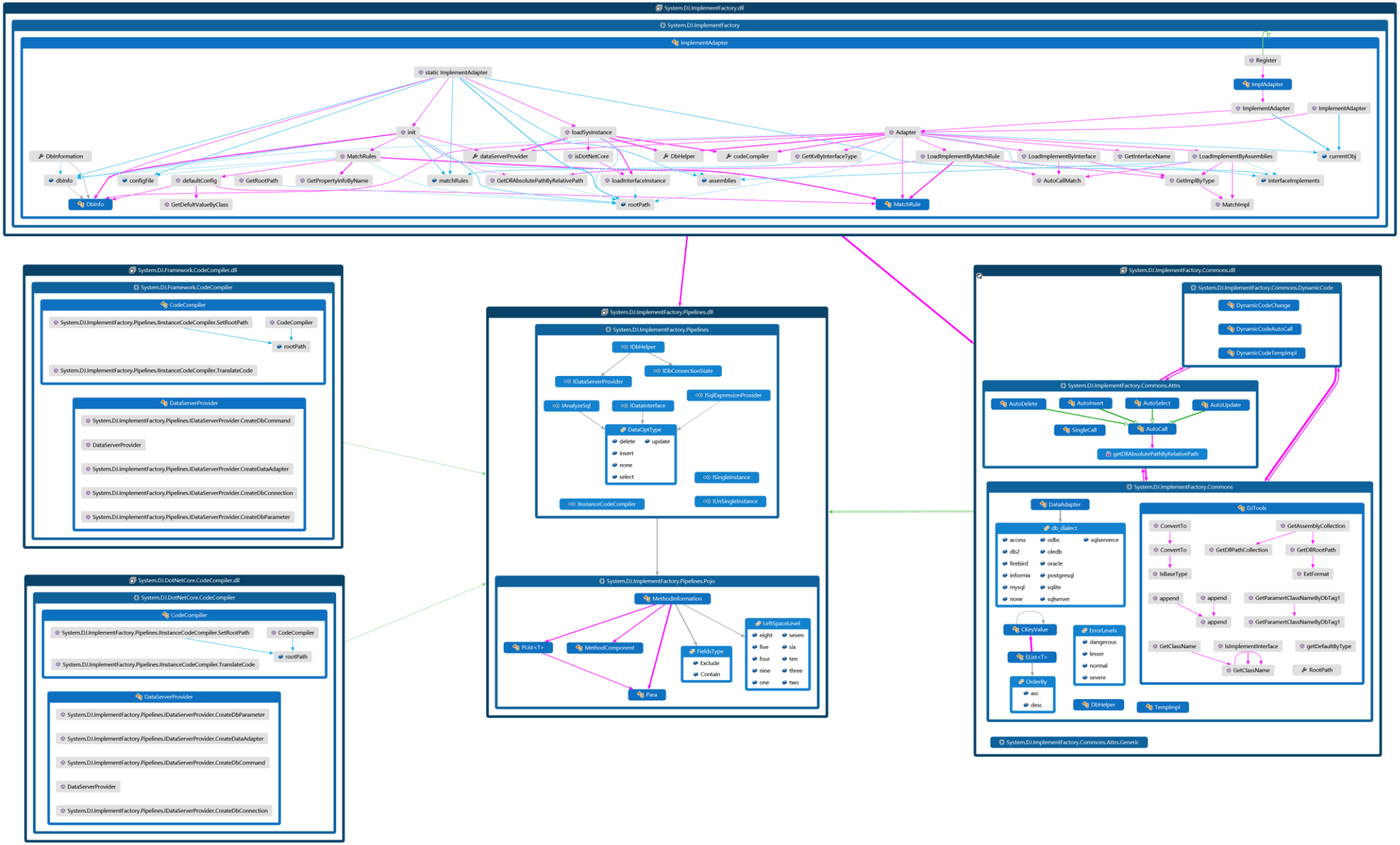


根据实际需求，你也可以采用实例非引用机制，而这仅需在 AutoCall 的基础上增加 SingleCall 即可(也可让 ILogin 接口的实例实现 ISingleInstance 接口):



ImplementFactory 组件为自动扫描接口类型成员，自动为接口类型成员装配接口实例、面向接口操作数据源、实现接口实例加载拦截、接口方法调用拦截、以及对应的异常拦截提供了一个有效的解决方案。

作者 Allan QQ: 564343162 Emial: 564343162@qq.com



ImplementFactory 组件结构介绍

```
namespace System.DJ.ImplementFactory
    public abstract class ImplementAdapter
        public ImplementAdapter();

        public static IDbHelper DbHelper { get; set; }
        public static IDataServerProvider dataServerProvider { get; set; }

        public static string GetDllAbsolutePathByRelativePath(string dllRelativePath);
        public static string GetRootPath();
        public static void Register(object currentObj);
        public static string dbConnectionString{ get; set; }
```

ImplementAdapter 抽象类用于获取当前类实例，扫描该实例包含的有效接口成员，及后继加载操作。

方法介绍

方法名称	描述	类型	参数	返回值
ImplementAdapter	构造方法	构造方法	无	无
GetDllAbsolutePathByRelativePath	根据 dll 文件相对路径获取其绝对路径	静态方法	dllRelativePath: dll 文件相对路径	返回 dll 文件绝对路径
GetRootPath	获取当前运用的根路径	静态方法	无	返回当前运用的根路径
Register	注册当前类实例，如果当前类已继承了其它基类，那么需要使用此方法注册当前类。	静态方法	Object 当前类实例	无

属性介绍

属性名称	描述	属性域	值类型
DbHelper	如果不采用 ImplementFactory 自带的数据访问功能，可为 ImplementFactory 提供一个有自定义数据源访问实例	静态属性	IDbHelper 接口实例
dataServerProvider	为 ImplementFactory 提供一数据访问相关对象提供实例	静态属性	IDataServerProvider 接口实例
dbConnectionString	为 ImplementFactory 提供一个有效的数据库连接字符串(例: Data Source=(local);Initial Catalog=DatabaseName;User Id=sa;Password=sa;)，该字符串将取代组件自取的数据库连接字符串。	静态属性	string 字符串类型

```
namespace System.DJ.ImplementFactory.Pipelines
    public interface IDbHelper
        IDbConnectionState dbConnectionState { get; set; }
        IDataServerProvider dataServerProvider { get; set; }

        int delete(string sql, List<DbParameter> parameters, ref string err);
        int insert(string sql, List<DbParameter> parameters, ref string err);
        DataTable query(string sql, List<DbParameter> parameters, ref string err);
```

```
DataTable query(string sql, ref string err);
int update(string sql, List<DbParameter> parameters, ref string err);
```

IDbHelper 该接口为 ImplementFactory 提供一个自定义数据源访问类实例。

方法介绍

方法名称	描述	参数名称	参数说明	返回值
delete	执行删除操作时调用此方法	sql	Sql 语句表达式	返回所删除记录数量
		parameters	带参数的 Sql 语句的参数对象集合	
		err	如果发生异常，返回异常信息	
insert	执行新增操作时调用该方法	sql	Sql 语句表达式	返回所新增记录数量
		parameters	带参数的 Sql 语句的参数对象集合	
		err	如果发生异常，返回异常信息	
query	执行查询操作时调用该方法	sql	Sql 语句表达式	返回查询结果 DataTable 数据集
		parameters	带参数的 Sql 语句的参数对象集合	
		err	如果发生异常，返回异常信息	
query	执行查询操作时调用该方法	sql	Sql 语句表达式	返回查询结果 DataTable 数据集
		err	如果发生异常，返回异常信息	
update	执行查询操作时调用该方法	sql	Sql 语句表达式	返回所更新记录数量
		parameters	带参数的 Sql 语句的参数对象集合	
		err	如果发生异常，返回异常信息	

属性说明

属性名称	描述	属性域	值类型
dbConnectionState	数据库连接状态	静态属性	IDbConnectionState 接口实例
dataServerProvider	为 IDbHelper 提供一数据访问相关对象提供实例	静态属性	IDataServerProvider 接口实例

```
namespace System.DJ.ImplementFactory.Pipelines
    public interface IDataServerProvider
        DbConnection CreateDbConnection(string connectString);
        DbParameter CreateDbParameter(string fieldName, object fieldValue);
        DbCommand CreateDbCommand(string sql, DbConnection connection);
        DataAdapter CreateDataAdapter(DbCommand dbCommand);
```

IDataServerProvider: 该接口为 IDbHelper 提供数据库访问相关对象，**如果数据库为非 Microsoft SQL Server 类型的数据库，则必需重新实现该接口的实例，提供与当前数据库匹配的对象，且该接口实例作用域必须为 public 类型，所提供的实例将被组件自动加载。**

方法介绍

方法名称	描述	类型	参数	返回值
CreateDbConnection	创建一个 DbConnection 对象	接口方法	connectString：数据库连接字符串	返回一个 DbConnection 对象
CreateDbParameter	创建一个 DbParameter 对象	接口方法	fieldName：字段名称 fieldValue：字段值	返回一个 DbParameter 对象

CreateDbCommand	创建一个 DbCommand 对象	接口方法	sql：sql 语句表达式 connection：数据库连接对象	返回一个 DbCommand 对象
CreateDataAdapter	创建一个 DataAdapter 对象	接口方法	dbCommand：DbCommand 对象	返回一个 DataAdapter 对象

```
namespace System.DJ.ImplementFactory.Pipelines
    public interface IDbConnectionState
        void DbConnection_Created(DbConnection sender);
        void DbConnection_CreatedFail(Exception ex);
        void DbConnection_Disposed(DbConnection sender, EventArgs e);
        void DbConnection_StateChange(DbConnection sender, StateChangeEventArgs e);
```

IDbConnectionState 当数据连接对象状态发生变更时，触发该接口方法。

方法介绍

方法名称	描述	类型	参数	返回值
DbConnection_Created	当数据库连接创建成功后触发			
DbConnection_CreatedFail	当数据库连接创建失败后触发			
DbConnection_Disposed	当数据库连接资源释放后触发			
DbConnection_StateChange	当数据库连接状态发生改变时触发			

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public class AutoCall : Attribute
        public AutoCall();

        public AutoCall(string MatchRuleOrClassName)

        public AutoCall(string MatchRuleOrClassName, bool IgnoreCase);

        public string MatchRuleOrClassName { get; set; }

        public bool IgnoreCase { get; set; }

        public static void SetDataProviderAssemble(string dllRelativePath);

        public virtual void CreateInstanceByInterface(Type interfaceType, object implement, AutoCall autoCall, int tempImplementCount);

        public virtual bool ExecuteAfterFilter(Type interfaceType, object implement, string methodName, PList<Para> paras, object result);
```

public virtual bool ExecuteBeforeFilter(Type interfaceType, object implement, string methodName, PList<Para> paras);

public virtual void ExecuteException(Type interfaceType, object implement, string methodName, PList<Para> paras, Exception ex);

public IDataProvider GetDataProvider(string dataProviderNamespace, string dataProviderClassName, AutoCall autoCall);

public void GetParaByBaseType(Type dataType, string dbTag, object data, string fieldName, List<DbParameter> dbParas);

public void GetParaListByEntity(object entity, List<DbParameter> dbParas, EList<CKeyValue> paraNameList);

public void GetSqlByDataProvider(IDataProvider dataProvider, PList<Para> paraList, List<DbParameter> dbParameters, AutoCall autoCall, DataOptType dataOptType, ref string sql);

public virtual string ExecuteInterfaceMethodCodeString(MethodInformation method);

public virtual bool LoadAfterFilter<T>(T impl);

public virtual bool LoadBeforeFilter(Type interfaceType);

protected void ExecInterfaceMethodOfCodeStr_DataOpt(MethodInformation method, DataOptType dataOptType, string sql, ref string code);

AutoCall 可继承该类实例实现 AOP 机制，对接口实例及接口方法的调用拦截，和异常拦截。并且可以利用 ExecuteInterfaceMethodCodeString 方法进行特殊方法属性功能的扩展。

方法说明

方法名称	描述	类型	参数	返回值
AutoCall	构建一个实例	构造方法	无	无
AutoCall	构建一个实例	构造方法	MatchRuleOrClassName：匹配一个实例类名，可以是一个有效的正则表达式；	
AutoCall	构建一个实例	构造方法	MatchRuleOrClassName：匹配一个实例类名，可以是一个有效的正则表达式； IgnoreCase：匹配实例类名称时是否忽视大小写，true 为忽视大小写，false 为匹配大小写，默认值为 false；	
SetDataProviderAssemble	设置数据提供者程	静态方	dllRelativePath：包含	无

	序集相对路径(dll 文件相对路径),默认为当前程序集	法	数据提供者的 dll 文件相对路径	
LoadBeforeFilter	AOP 机制，加载接口实例前被调用	虚方法	interfaceType：接口类型	Bool 类型，返回 true 时允许继续加载接口实例，返回 false 时阻止加载当前接口实例。
LoadAfterFilter	AOP 机制，加载接口实例后被调用	泛型虚方法	泛型：接口实例类型	Bool 类型，返回 true 时允许把已加载的接口实例赋给类成员接口变量，反之异然。
ExecuteBeforeFilter	AOP 机制，执行接口方法前被调用	虚方法	interfaceType:接口类型; implement：接口实例; methodName：接口方法名称; paras：方法参数集合	Bool 类型，返回 true 时允许继续执行接口方法，反之异然。
ExecuteAfterFilter	AOP 机制，执行接口方法后被调用	虚方法	interfaceType:接口类型; implement：接口实例; methodName：接口方法名称; paras：方法参数集合	Bool 类型，返回 true 时允许把执行接口方法后的结果返回给调用者，反之异然
ExecuteException	AOP 机制，执行接口方法发生异常时调用，拦截异常信息	虚方法	interfaceType:接口类型; implement：接口实例; methodName：接口方法名称; paras：方法参数集合; ex：接口方法异常信息	无
ExecuteInterfaceMethodCodeString	在接口代理方法内自定义执行实例接口方法及数据处理代码	虚方法	method：接口方法信息对象	string 类型，返回自定义执行接口方法及数据处理代码字符串。
CreateInstanceByInterface	每次根据接口创建待装配的实例对象时调用	虚方法	interfaceType：接口类型; implement：接口实例; autoCall：AutoCall 对象; tempImplementCount：临时实例数量	无
ExecInterfaceMethodOf CodeStrDataOpt	执行 select、insert、update、delete 数据操作时被调用	保护类型方法，可被子类调用	method：接口方法信息对象; dataOptType：数据操作类型(select、insert、update、delete); sql：方法属性(AutoSelect、AutoInsert、AutoUpdate、AutoDelete)传入的 sql 语句表达式; ref code：返回生成的代码字符串;	无(由 ref 参数返回)

GetSqlByDataProvider	当执行接口方法时,根据数据提供者获取 sql 语句表达式(动态创建 sql 语句)	公共方法	dataProvider: sql 语句数据提供者; paraList：接口方法参数集合; dbParameters：sql 语句参数集合; autoCall：AutoCall 对象 dataOptType：数据操作类型 select、insert、update、delete ref sql：返回根据数据动态生成的 sql 语句字符串	
GetDataProvider	根据名称空间和类名获取 sql 语句提供者接口实例	公共方法	dataProviderNamespace：sql 语句提供者接口实例所属名称空间; dataProviderClassName：sql 语句提供者接口实例类名称; autoCall：AutoCall 对象	返回一个实现 IDataProvider 接口的动态 sql 语句提供者实例
GetDbParaListByEntity	根据数据实体(接口方法参数中包含)和带参数的 sql 语句中包含的参数集合来创建 DbParameter 集合	公共方法	entity：数据实体; dbParas: DbParameter 参数集合; paraNameList：接口方法参数集合	无(由 dbParas 参数对象携带返回)
GetDbParaByBaseType	用数据实体中属于基本类型(int,string,bool 等)的属性及其值来创建 DbParameter 集合	公共方法	dataType：数据类型(数据实体属性域); dbTag：数据源参数类型(如:sqlserver 参数类型为@，mysql 参数类型为?，oracle 为:); data：数据源属性值; fieldName：sql 语句中字段名称(同时也是数据实体属性名称); dbParas：DbParameter 集合	无(由 dbParas 参数对象携带返回)

例:

```
public class Login : ImplementAdapter
    [AutoCall]
    IAuthenticate authenticate;
```

```
namespace System.DJ.ImplementFactory.Commons
    public class EList<T> : IEnumerable<T> where T : CKeyValue
```

EList：以 CKeyValue 对象为元素的一个 List 集合，在键值对的情况方便使用。

```
namespace System.DJ.ImplementFactory.Commons
    public class CKeyValue : IComparable<CKeyValue>
```

CKeyValue：自定义了一个键值对对象

```
namespace System.DJ.ImplementFactory.Commons.Attrs
```

```
    public class SingleCall : Attribute
```

SingleCall：指标当前接口类型的成员变量采用非引用机制装配（对应独立的内存地址）

```
namespace System.DJ.ImplementFactory.Pipelines
    public enum DataOptType
        select = 0,
        insert,
        update,
        delete,
        count,
        none
```

DataOptType：数据操作枚举

```
namespace System.DJ.ImplementFactory.Pipelines.Pojo
    public class MethodInformation
        public MethodInformation();

        public MethodInfo methodInfo { get; set; }
        public string dataProviderClassName { get; set; }
        public string dataProviderNamespace { get; set; }
        public string[] ResultExecMethod { get; set; }
        public string ParaListVarName { get; set; }
        public string AutoCallVarName { get; set; }
        public string StartSpace { get; set; }
        public MethodComponent methodComponent { get; }
        public string[] Fields { get; set; }
        public FieldsType fieldsType { get; set; } = FieldsType.Exclude;
        public Type ofInstanceType { get; set; }
        public Type ofInterfaceType { get; set; }
        public PList<Para> paraList { get; set; }

        public void append(ref string code, string s);
        public void append(ref string code, LeftSpaceLevel level, string s);
        public void append(ref string code, LeftSpaceLevel level, string s, params string[] arr);
        public string getDefaultByType(Type type);
        public string getSpace(int tabNum);
```

MethodInformation：承载接口方法信息及与接口方法相关基本方法

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
getSpace	根据指定的tab数量获取左边代码空格空字符串	公共方法	tabNum：左边缩进 tab 数量	空字符串
getDefaultByType	根据类型获取该数据对应的类型默认值	公共方法	type：数据类型	返回默认值字符串
append	把需格式化的字符串附加到 code 字符串后，附加前在 code 字符串后加回车换行	公共方法	ref code：返回附加后的结果； level：左边 tab 层级量(空字符串)； s：要附加的字符串； arr：格式化{0}、{1}..字符串待的替换字符系列	无(参数 ref code 返回)

append	把普通字符串附加到 code 字符串后，附加前在 code 字符串后加回车换行	公共方法	ref code：返回附加后的结果； level：左边 tab 层级量(空字符串)； s：要附加的字符串；	无(参数 ref code 返回)
append	左边默认按照 0 个 tab 的缩进把普通字符串附加到 code 字符串后，附加前在 code 字符串后加回车换行	公共方法	ref code：返回附加后的结果； s：要附加的字符串；	无(参数 ref code 返回)

属性介绍

属性名称	描述	属性域	值类型
ofInterfaceType	设置\获取接口类型	普通属性	Type
ofInstanceType	设置\获取实例类型	普通属性	Type
methodInfo	设置\获取 System.Reflection.MethodInfo 接口的系统方法信息对象	普通属性	MethodInfo
methodComponent	设置\获取接口方法组件信息对象	普通属性	MethodComponent 对象
fields	字段名称集合，需要排除或包含的字段名称	普通属性	new string[]
fieldsType	指示字段名称集合类型：排除或包含。 FieldsType: Exclude, Contain	普通属性	枚举类型 enum
StartSpace	设置\获取默认左缩进空间(空字符串)	普通属性	空字符串
AutoCallVarName	设置\获取 AutoCall 对象变量名称	普通属性	string
ParaListVarName	设置\获取接口方法参数集合变量名称	普通属性	string
dataProviderNamespace	设置\获取动态 sql 提供者实例所在名称空间	普通属性	string
dataProviderClassName	设置\获取动态 sql 提供者实例类名称	普通属性	string
ResultExecMethod	执行 select\count 查询后，结果执行方法，参数值类型：new string[] { nameSpace, className, methodName }，该参数所指向的方法参数类型及返回值类型必须与数据接口方法返回值类型一至		
paraList	设置\获取接口方法参数集合	普通属性	PList<Para>

```
namespace System.DJ.ImplementFactory.Pipelines.Pojo
    public class MethodComponent
        public string ResultVariantName { get; set; }
        public string ResultTypeName { get; set; }
        public string InstanceVariantName { get; set; }
        public string InterfaceMethodName { get; set; }
        public string MethodParas { get; set; }
```

MethodComponent：方法信息组件

属性介绍

属性名称	描述	属性域	值类型
ResultVariantName	获取\设置接口方法返回结果变量名称	公共	string
ResultTypeName	获取\设置接口方法返回值类型全名	公共	String

InstanceVariantName	获取\设置接口实例变量名称	公共	String
InterfaceMethodName	获取\设置接口方法名称	公共	String
MethodParas	获取\设置接口方法参数变量名称字符串系列,多个用英文状态逗号相隔	公共	string

```
namespace System.DJ.ImplementFactory.Pipelines.Pojo
public enum FieldsType
        // 排除
        Exclude,
        // 包含
        Contain
```

FieldsType: 指示字段名称集合类型是 Exclude(排除) 或 Contain(包含), 默认为 Exclude

```
namespace System.DJ.ImplementFactory.Pipelines.Pojo
    public class PList<T> : List<T> where T : Para
        public PList();

        public Para this[string paraName] { get; }
        public Para this[string paraName, bool ignoreNull] { get; }
```

PList: 定义了一个元素为 Para 对象的集合

属性介绍

属性名称	描述	属性域	值类型
this[string paraName]	根据参数名称获取参数对象的检索器	公共	Para 对象
this[string paraName, bool ignoreNull]	根据参数名称获取参数对象的检索器。ignoreNull: 是否忽视参数值为 null 的情况, 为 true 时 null 转换为空字符串。	公共	Para 对象

```
namespace System.DJ.ImplementFactory.Pipelines.Pojo
    public class Para
        public Para(Guid guid);

        public Type ParaType { get; set; }
        public string ParaTypeName { get; set; }
        public string ParaName { get; set; }
        public object ParaValue { get; set; }
        public Guid ID { get; }
```

Para: 接口方法参数实体

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
Para	创建Guid类型ID的构造方法	构造方法	guid: 创建对象时传入一个 Guid 类型的值	无

属性介绍

属性名称	描述	属性域	值类型
ParaType	获取\设置接口方法参数类型	公共	Type
ParaTypeName	获取\设置接口方法参数类型全名	公共	string
ParaName	获取\设置接口方法参数名称	公共	string
ParaValue	获取\设置接口方法参数值	公共	object
ID	获取 Para 对象 ID	公共	Guid

```
namespace System.DJ.ImplementFactory.Pipelines
    public interface ISingleInstance
        object Instance { get; set; }
```

ISingleInstance：标识对应的接口采用非引用机制加载实例，每个实例都对应着不同的内存地址。

属性介绍

属性名称	描述	属性域	值类型
Instance	获取\设置接口当前实例，如果该属性值为 null，当接口类型的成员变量被装配时由组件自动为该属性赋值	公共	object

```
namespace System.DJ.ImplementFactory.Pipelines
    public interface IUnSingleInstance
```

IUnSingleInstance：指定的接口实例全部强制采用引用机制，所有同类型的接口实例全部采用同一个内存地址，IUnSingleInstance 接口的优先级高于 ISingleInstance 接口

```
namespace System.DJ.ImplementFactory.Pipelines
    public interface ISqlExpressionProvider
        string provideSql(DbList<DbParameter> dbParameters, AutoCall.DataOptType dataOptType, PList<Para> paraList, object[] methodParameters);
```

ISqlExpressionProvider： 根据数据动态生成 sql 语句表达式接口，在数据属性(AutoSelect、AutoInsert、AutoUpdate、AutoDelete)中使用

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
provideSql	根据数据动态生成 sql 语句表达式	接口方法	dbParameters：数据参数集体，如果所创建的 sql 中包含参数，必须在 dbParameters 中添加对应的参数; dataOptType：数据操作类型(select、insert、update、delete); paraList：具备参数名、数据类型、参数值等信息的参数集合; methodParameters：接口方法参数值集合，顺序与接口方法参数位置序号对应	返回所创建的 sql 语句表达式

```
namespace System.DJ.ImplementFactory.Pipelines.Pojo
    public class DbList<T> : List<T> where T : DbParameter
        public DbList();

        public T this[string name] { get; }
```

public static IDataServerProvider dataServerProvider { get; set; }

public void Add(string name, object value);

DbList<T>：自定义 System.Data.DbParameter 元素集合，支持 foreach 遍历元素。

属性介绍

属性名称	描述	属性域	值类型

this[name]	DbParameter 对象检索器。根据字段名称检索参数对象。
	例：
	DbList<DbParameter> paras = new DbList<DbParameter>();
	paras.Add("name", "Lucy");
	paras.Add("age", 23);
	
	DbParameter para = paras["age"];
		公共	DbParameter

dataServerProvider	设置或返回一个数据对象提供者		

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
Add	新增一个数据参数	公共方法	name：字段名称; value：字段值	无

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public abstract class AbsDataInterface : AutoCall, IDataInterface
        public AbsDataInterface();
        public override string ExecuteInterfaceMethodCodeString(MethodInformation method);
```

AbsDataInterface：实现数据操作属性的公共抽象。自动识别数据操作类型。

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public sealed class AutoSelect：AbsDataInterface
        public AutoSelect(string selectExpression);
        public AutoSelect(string dataProviderNamespace, string dataProviderClassName);
        public AutoSelect(string selectExpression, string[] ResultExecMethod);
public AutoSelect(string dataProviderNamespace, string dataProviderClassName, string[] ResultExecMethod);

        public override string ExecuteInterfaceMethodCodeString(MethodInformation method);
```

AutoSelect： 数据查询属性，依附于接口方法上。

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
AutoSelect	携带 select 查询语句的构造方法	构造方法	selectExpression：select 查询语句表达式	无
AutoSelect	指定由提供者根据数据动态生成 sql 语句	构造方法	dataProviderNamespace：数据提供者类所在名称空间;	无

			dataProviderClassName: 数据提供者类名(需继承 ISqlExpressionProvider 接口)	
AutoSelect	携带 select 查询语句, 并提供查询结果执行方法	构造方法	selectExpression: select 查询语句表达式 ResultExecMethod: 指定结果执行方法, 参数值格式: new string[] { nameSpace, className, methodName }, 该参数所指向的方法参数类型及返回值类型必须与数据接口方法返回值类型一至	
AutoSelect	指定由提供者根据数据动态生成 sql 语句, 并提供查询结果执行方法		dataProviderNamespace : 数据提供者类所在名称空间; dataProviderClassName: 数据提供者类名(需继承 ISqlExpressionProvider 接口) ResultExecMethod: 指定结果执行方法, 参数值格式: new string[] { nameSpace, className, methodName }, 该参数所指向的方法参数类型及返回值类型必须与数据接口方法返回值类型一至	
ExecuteInterfaceMethod CodeString	在接口代理方法内自定义执行实例接口方法及数据处理代码	重写 AutoCall 类的虚方法	method: 接口方法信息对象	string 类型, 返回自定义执行接口方法及数据处理代码字符串

参数 selectExpression: sql 语句与接口方法参数匹配规则

sql-selectExpression	描述	案例
{MethodParameterName}	匹配接口方法参数名称	[AutoSelect("select * from UserInfo where name={**userName**}")] UserInfo IUserInfo.GetUserInfoByName(string **userName**);
{EntityPropertyName}	匹配接口方法参数实体对象中的属性名称	[AutoSelect("select * from UserInfo where name={**userName**}")] UserInfo IUserInfo.GetUserInfoByName(UserInfo userInfo); 其中{**userName**}匹配该方法参数实体对象 userInfo. **userName**, 组件会自动解析
@userName	表示带参数的 sql 语句, 组件会自动生成与 sql 语句中参数对应的 DbParameter 集合	[AutoSelect("select * from UserInfo where name=@userName")] **UserInfo** IUserInfo.GetUserInfoByName(**UserInfo** userInfo); 组件会自动生成与 sql 语句中参数对应的 Db 参数集合: List<DbParameter> dbParas = List<DbParameter>(); dbParas.Add(new SqlParameter("userName", userInfo. userName)); 如果接口方法参数非数据实体对象, 也可直接与@userName 直接匹配, 同样自动成生 Db 参数集合。

		[AutoSelect("select * from UserInfo where name=@userName")] UserInfo IUserInfo.GetUserInfoByName(string userName);

```
namespace System.DJ.ImplementFactory.Pipelines
    public interface IDataInterface
```

IDataInterface：AutoSelect| AutoInsert | AutoUpdate | AutoDelete 定义为数据操作属性

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public class AutoCount : AbsDataInterface
        public AutoCount(string countExpression);
        public AutoCount(string dataProviderNamespace, string dataProviderClassName);
        public AutoCount(string countExpression, string[] ResultExecMethod);
        public AutoCount(string dataProviderNamespace, string dataProviderClassName, string[]
    ResultExecMethod);
```

AutoCount：统计记录数量，返回值类型为：int 或 bool，当为 int 类型时返回统计的数量；当为 bool 类型时返回 true(存在记录)/false(零条记录)

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
AutoCount	根据 sql 表达式获取记录数量或返回是否存在记录	构造方法	countExpression：统计记录数量的 sql 表达式	无
AutoCount	根据 sql 表达式获取记录数量或返回是否存在记录	构造方法	dataProviderNamespace：数据提供者类所在名称空间； dataProviderClassName：数据提供者类名(需继承 ISqlExpressionProvider 接口)	
AutoCount	根据 sql 表达式获取记录数量或返回是否存在记录	构造方法	countExpression：统计记录数量的 sql 表达式 ResultExecMethod：指定结果执行方法，参数值格式：new string[] { nameSpace, className, methodName }，该参数所指向的方法参数类型及返回值类型必须与数据接口方法返回值类型一至	
AutoCount	根据 sql 表达式获取记录数量或返回是否存在记录	构造方法	dataProviderNamespace：数据提供者类所在名称空间； dataProviderClassName：数据提供者类名(需继承 ISqlExpressionProvider 接口) ResultExecMethod：指定结果执行方法，参数值格式：new string[] { nameSpace, className, methodName }，该参数所指向的方法参数类型及返回值类型必须与数据接口方法返回值类型一至	

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public class AutoProcedure : AbsDataInterface
        public AutoProcedure(string sqlExpression);
```

```
public AutoProcedure(string sqlExpression, string[] fields);
public AutoProcedure(string dataProviderNamespace, string dataProviderClassName);
public AutoProcedure(string sqlExpression, string[] fields, FieldsType fieldsType);
```

AutoProcedure：执行存储过程操作属性，依附于接口方法。
方法介绍：

方法名称	方法描述	方法类型	参数说明	返回值
AutoProcedure	执行存储过程的 sql 表达式	构造方法	sqlExpression：执行存储过程的 sql 表达式	无
AutoProcedure	执行存储过程的 sql 表达式	构造方法	sqlExpression：执行存储过程的 sql 表达式； fields：需要排除的字段列表	
AutoProcedure	执行存储过程的 sql 表达式	构造方法	sqlExpression：执行存储过程的 sql 表达式； fields：需要排除的字段列表 fieldsType：指定 fields 所列字段是排除或包含，默认-排除	
AutoProcedure	执行存储过程的 sql 表达式	构造方法	dataProviderNamespace：数据提供者类所在名称空间； dataProviderClassName：数据提供者类名(需继承 ISqlExpressionProvider 接口)	

```
namespace System.DJ.ImplementFactory.Commons
    public class DataEntity<T> : IEnumerable<T>, IEnumerable where T : DataElement
        public DataEntity();

        public T this[string name] { get; }

        public int Count { get; }

        public bool Add(string name, object value);
        public void Clear();
        public void Foreach(Func<DataElement, bool> func);
        public void Foreach(Action<DataElement> action);
        public void Remove(string name);
        public void RemoveAt(int index);
```

DataEntity<T>：在不明确字段数量或字段名称的情况下使用。支持 foreach(DataElement de in dataEntities) 遍历 DataElement 元素

属性介绍

属性名称	属性描述	返回值

this[name]	字段数据检索器。
	例：
	DataEntity<DataElement> datas = new DataEntity<DataElement>()
	datas.Add("name", "Lucy");
	datas.Add("age", 23);
		如果存在，返回一个 DataElement 对象

	DataElement dataElement = datas["name"];	
Count	获取 DataEntity<DataElement>集合所包含的字段数量	返回 DataEntity<DataElement> 集合的字段数量

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
Add	为集合新增一个数据元素(DataElement:字段)	公共方法	name：字段名称 value：字段值	无
Clear	删除集合所包含的所有数据元素	公共方法		
Foreach	Lambda Func 类型的数据元素遍历器	公共方法	Func<DataElement, bool> func：用于遍历字段数据元素的 lambda 表达式，在循环数据元素的过程中，如果在 lambda 体内返回 false 则停止遍历数据元素。	
Foreach	Lambda Action 类型的数据元素遍历器	公共方法	d Action<DataElement> action：用于遍历字段数据元素的 lambda 表达式	
Remove	根据字段名称删除指定字段数据元素(DataElement)	公共方法	name：一个有效的字段名称	
RemoveAt	根据集合内元素序号删除指定字段数据元素(DataElement)	公共方法	Index：一个有效的字段数据元素序号	

```
namespace System.DJ.ImplementFactory.Commons
    public class DataElement
        public DataElement(string name, object value);

        public string name { get; set; }
        public object value { get; set; }

        public override string ToString();
```

DataElement：字段数据元素对象

属性介绍

属性名称	属性描述	返回值
name	字段名称	字符串类型的数据：字段名称
value	字段值	object 类型的字段值

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
DataElement	创建一个字段数据元素对象	构造方法	name：字段名称 value：字段值	无
ToString	返回字段值数据，如果字段值 value 为 null，则返回空字符串	公共方法		

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public class FieldMapping : Attribute
        public FieldMapping(string fieldName);
        public string FieldName { get; }
        public static string GetFieldMapping(PropertyInfo propertyInfo);
```

FieldMapping: 对象属性与字段映射属性标识

属性介绍

属性名称	属性描述	返回值
FieldName	映射的字段名称	字符串类型的数据: 字段名称

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
FieldMapping	对象属性与字段映射属性标识	构造方法	fieldName: 映射的字段名称	无
GetFieldMapping	传入对象 PropertyInfo 属性对象, 获取映射的字段名称	公共静态方法	propertyInfo : PropertyInfo 属性对象	

```
namespace System.DJ.ImplementFactory.NetCore.Commons.Attrs
    public class Condition : Attribute
        public Condition();
        public Condition(LogicSign logicSign);
        public Condition(string field_compare);
        public Condition(WhereIgrons whereIgrons);
        public Condition(string field_compare, WhereIgrons whereIgrons);
        public Condition(LogicSign logicSign, WhereIgrons whereIgrons);
        public Condition(string field_compare, LogicSign logicSign);
        public Condition(string fieldMapping, string compareSign);
        public Condition(string fieldMapping, string compareSign, WhereIgrons whereIgrons);
        public Condition(string fieldMapping, string compareSign, LogicSign logicSign);
        public Condition(string field_compare, LogicSign logicSign, WhereIgrons whereIgrons);
        public Condition(string fieldMapping, string compareSign, LogicSign logicSign, WhereIgrons
whereIgrons);

        public string CompareSign { get; set; }
        public WhereIgrons where_igrones { get; set; }
        public string TableSign { get; set; }
        public string FieldMapping { get; set; }
        public LogicSign logic_sign { get; set; }

        public string Unit(PropertyInfoExt propertyInfo);

        //忽视属性值
        public enum WhereIgrons
        {
            none = 0, //无忽视
            igroneEmpty = 2, //忽视空字符串
            igroneNull = 4, //忽视 null
            igroneEmptyNull = 8, //忽视空字符串或null
            igroneFalse = 16, //忽视布尔值为 false
            igroneMinMaxDate = 32, //忽视最小或最大日期
            igroneMinDate = 64, //忽视最小日期
            igroneMaxDate = 128, //忽视最大日期
            igroneZero = 256 //忽视为零的值
```

```
    }
    public enum LogicSign
    {
        and = 0,
        or = 1
    }

    public class PropertyInfoExt
    {
        public PropertyInfoExt();

        public string Name { get; set; }
        public object Value { get; set; }
        public Type PropertyType { get; set; }
        public Type DeclaringType { get; set; }

        public object GetValue(object entity);
        public object GetValue();
    }
```

Condition: 使用 AutoSelect 查询数据时，创建动态 where 条件标识。只需要在 class 或对象属性加入 [Condition] 属性标识，即可动态生成 where 条件，可配合 [FieldMapping] 属性标识一起使用；当在 class 加入 [Condition] 时，该对象内所有满足条件的属性将会被例为 where 条件，单独标识[Condition]的属性将采用独立指定的相关值，未标识[Condition]的属性满足条件后将采用默认的 WhereIgrone 策略，且默认采用 and 关系连接符。如何在 sql 表达式中使用动态 where 匹配，可查看<sql 表达式中标识符匹配形式>

属性介绍

属性名称	属性描述	返回值
FieldMapping	字符类型，映射的数据表字段名称,为空时等同于属性名称;如果同时存在 [FieldMapping] 属性，在创建 where 条件时 [FieldMapping] 属性里的 fieldName 优先级高于 [Condition] 属性里的 FieldMapping 值	
CompareSign	字符类型，比较符号,可为: like, left_like(llike), right_like(rlike), >=(大于等于), <=(小于等于), <>(不等于), >(大于), <(小于), =(等于)	
logic_sign	枚举类型，逻辑连接符：and / or 详细参见 LogicSign	
where_igrones	枚举类型，设置忽视属性值，详细参见 WhereIgrons	

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
Condition	动态 where 条件属性标识	构造方法	参见属性说明	无

默认 WhereIgrons 策略(default-strategy):

属性值类型	WhereIgrons 策略
string / Guid	WhereIgrons.igroneEmptyNull
bool（Boolean）	WhereIgrons.igroneFalse
DateTime	WhereIgrons.igroneMinMaxDate
int/Int16/Int64/double/float/decimal	WhereIgrons.igroneZero
其它	WhereIgrons.igroneNull

采用动态创建 where 条件时，sql 表达式中标识符匹配形式(SqlMatch):

sql 表达式	where 语句(满足条件)	where 语句(不满足条件)
select * from UserInfo {ParamName}	where 1=1 and name='zl'	空字符

select * from UserInfo where {ParamName}	1=1 and name='zl'	1=1
select * from UserInfo where age=21 and {ParamName}	1=1 and name='zl'	1=1
select * from UserInfo where age=21 and ({ParamName} or key='A232')	1=1 and name='zl'	1=1
select * from UserInfo where age=21 {ParamName}	and name='zl'	空字符

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public class AutoInsert: AbsDataInterface
        public AutoInsert(string insertExpression);
        public AutoInsert(string insertExpression, DataOptType sqlExecType);
        public AutoInsert(string insertExpression, string[] fields);
        public AutoInsert(string insertExpression, string[] fields, DataOptType sqlExecType)
        public AutoInsert(string dataProviderNamespace, string dataProviderClassName);
        public AutoInsert(string dataProviderNamespace, string dataProviderClassName, DataOptType
sqlExecType)
        public AutoInsert(string insertExpression, string[] fields, FieldsType fieldsType);
        public AutoInsert(string insertExpression, string[] fields, FieldsType fieldsType, DataOptType
sqlExecType)
        public AutoInsert(string dataProviderNamespace, string dataProviderClassName, bool
EnabledBuffer);
        public AutoInsert(string insertExpression, string[] fields, FieldsType fieldsType, bool
EnabledBuffer);
```

AutoInsert：新增数据操作属性，依附于接口方法。

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
AutoInsert	携带 insert 数据新增语句的构造方法	构造方法	insertExpression：insert 数据新增语句表达式	无
AutoInsert	携带 insert 数据新增语句的构造方法，支持复杂对象（Dictionary<string, baseType> | List<DataEntity> | DataEntity）构建字段赋值项	构造方法	insertExpression：执行 insert 操作的 sql 语句 fields：字段名称数组,采用复杂对象构建字段赋值项时，执行 insert 操作时必须排除\包含的字段名称，默认采用 Exclude sqlExecType：sql 表达式执行类型	无
AutoInsert	同上	构造方法	insertExpression：执行 insert 操作的 sql 语句 fields：字段名称数组,采用复杂对象构建字段赋值项时，执行 insert 操作时必须排除\包含的字段名称。	

			fieldsType：指示字段名称集合类型是 Exclude(排除)或 Contain(包含)，默认为 Exclude	
AutoInsert	同上	构造方法	insertExpression：执行 insert 操作的 sql 语句 fields：字段名称数组,采用复杂对象构建字段赋值项时，执行 insert 操作时必须排除\包含的字段名称。 fieldsType：指示字段名称集合类型是 Exclude(排除)或 Contain(包含)，默认为 Exclude EnabledBuffer：是否启用缓存机制，默认为 false(不启用)	
AutoInsert	指定由提供者根据数据动态生成 insert 语句	构造方法	dataProviderNamespace：数据提供者类所在名称空间; dataProviderClassName：数据提供者类名(需继承 ISqlExpressionProvider 接口)	无
AutoInsert	同上	构造方法	dataProviderNamespace：数据提供者类所在名称空间; dataProviderClassName：数据提供者类名(需继承 ISqlExpressionProvider 接口) EnabledBuffer：是否启用缓存机制，默认为 false(不启用)	

参数 insertExpression：sql 语句与接口方法参数匹配规则

sql-insertExpression	说明	案例
继承 selectExpression	与 selectExpression 相同	参考 selectExpression
{EntityListParaName}	匹配接口方法参数元素类型为数据实体的 List 类型参数名称	[AutoInsert(insertExpression: “insert into UserInfo values({userInfos})”, excludeFields: new string[]{ “id” })] int insertUserInfo(List<UserInfo> userInfos); 组件自动解析 userInfos 实体集合对象，重新生成 insert 语句为： Insert into UserInfo(userName,age,address) values(@userName,@age,@address)

		并且生成与之对应的 Db 参数集合 List<DbParameter> 因为接口方法参数是 List 集合的数据实体，所生成的 insert 语句和 Db 集合将会被循环执行。
{EntityParaName}	匹配接口方法参数元素类型为数据实体参数名称	参考{EntityListParaName}，区别在于无需循环执行。
@ParaName @EntityPropertyName	sql 语句中仅包含 Db 参数标识时，接口方法参数可以为 DataTable 或 Dictionary 类型的参数。	因为 DataTable 和 Dictionary 类型的参数，接口方法未执行时，是无法得知该参数所包含字段名称，导致无法重新构建 insert 或 update 语句。所以只有在 sql 语句使用 Db 参数的情况下，明确 sql 语句中的所有字段，无需重构 sql 语句的情况下，才能使用 DataTable 和 Dictionary 类型的参数来进行 insert 或 update 数据操作。

Dictionary<string, baseType>:
string 字段名称
baseType 字段值，类型必须是 System 下的基础数据类型: string、int、long、int16、float、Single、double、bool、decimal、Guid、Datetime 等

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public sealed class AutoUpdate: AbsDataInterface
        public AutoUpdate(string updateExpression);
        public AutoUpdate(string updateExpression, DataOptType sqlExecType)
        public AutoUpdate(string updateExpression, string[] fields);
        public AutoUpdate(string updateExpression, string[] fields, DataOptType sqlExecType)
        public AutoUpdate(string dataProviderNamespace, string dataProviderClassName);
        public AutoUpdate(string dataProviderNamespace, string dataProviderClassName, DataOptType
sqlExecType)
        public AutoUpdate(string updateExpression, string[] fields, FieldsType fieldsType);
        public AutoUpdate(string updateExpression, string[] fields, FieldsType fieldsType, DataOptType
sqlExecType)
        public AutoUpdate(string dataProviderNamespace, string dataProviderClassName, bool
EnabledBuffer);
        public AutoUpdate(string updateExpression, string[] fields, FieldsType fieldsType, bool
EnabledBuffer);
```

AutoUpdate: 更改数据操作属性，依附于接口方法。

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
AutoUpdate	携带 update 数据更新语句的构造方法	构造方法	updateExpression: update 数据更改语句表达式	无
AutoUpdate	携带 update 数据更新语句的构造方法，支持复杂对象（Dictionary<string, baseType> | List<DataEntity> | DataEntity）构建字段赋值项	构造方法	updateExpression: 执行 update 操作的 sql 语句 fields: 采用复杂对象构建字段赋值项时，执行 update 操作时必须排除\包含的字段名称 sqlExecType: sql 表达式执行	无

			类型	
AutoUpdate	同上	构造方法	updateExpression：执行 update 操作的 sql 语句 fields: 采用复杂对象构建字段赋值项时，执行 update 操作时必须排除\包含的字段名称 fieldsType：指示字段名称集合类型是 Exclude(排除)或 Contain(包含)，默认为 Exclude	
AutoUpdate	同上	构造方法	updateExpression：执行 update 操作的 sql 语句 fields: 采用复杂对象构建字段赋值项时，执行 update 操作时必须排除\包含的字段名称 fieldsType：指示字段名称集合类型是 Exclude(排除)或 Contain(包含)，默认为 Exclude EnabledBuffer: 是否启用缓存机制，默认为 false(不启用)	
AutoUpdate	指定由提供者根据数据动态生成 update 语句	构造方法	dataProviderNamespace：数据提供者类所在名称空间; dataProviderClassName: 数据提供者类名(需继承 ISqlExpressionProvider 接口)	无
AutoUpdate	同上	构造方法	dataProviderNamespace：数据提供者类所在名称空间; dataProviderClassName: 数据提供者类名(需继承 ISqlExpressionProvider 接口) EnabledBuffer: 是否启用缓存机制，默认为 false(不启用)	
ExecuteInterfaceMethod CodeString	在接口代理方法内自定义执行实例接口方法及数据处理代码	重写 AutoCall 类的虚方	method：接口方法信息对象	string 类型，返回自定义执行接口方法及数

		法		据处理代码字符串

参数 updateExpression：sql 语句与接口方法参数匹配规则

sql-updateExpression	说明	案例
继承 selectExpression	与 selectExpression 相同	参考 selectExpression
{EntityListParaName}	匹配接口方法参数元素类型为数据实体的 List 类型参数名称	[AutoUpdate(insertExpression: "update UserInfo set {userInfos} where id=@id", excludeFields: new string[]{ "id" })] int updateUserInfo(List<UserInfo> userInfos); 组件自动解析 userInfos 实体集合对象，重新生成 update 语句为： update UserInfo set username=@ userName, age=@ age, address=@ address where id=@id 并且生成与之对应的 Db 参数集合 List<DbParameter> 因为接口方法参数是 List 集合的数据实体，所生成的 update 语句和 Db 集合将会被循环执行。
{EntityParaName}	匹配接口方法参数元素类型为数据实体参数名称	参考{EntityListParaName}，区别在于无需循环执行。
@ParaName @EntityPropertyName	sql 语句中仅包含 Db 参数标识时，接口方法参数可以为 DataTable 或 Dictionary 类型的参数。	因为 DataTable 和 Dictionary 类型的参数，接口方法未执行时，是无法得知该参数所包含字段名称，导致无法重新构建 insert 或 update 语句。所以只有在 sql 语句使用 Db 参数的情况下，明确 sql 语句中的所有字段，无需重构 sql 语句的情况下，才能使用 DataTable 和 Dictionary 类型的参数来进行 insert 或 update 数据操作。

```
namespace System.DJ.ImplementFactory.Commons.Attrs
    public sealed class AutoDelete：AbsDataInterface
        public AutoDelete(string deleteExpression);
        public AutoDelete(string deleteExpression, DataOptType sqlExecType)
        public AutoDelete(string deleteExpression, bool EnabledBuffer);
        public AutoDelete(string dataProviderNamespace, string dataProviderClassName);
        public AutoDelete(string dataProviderNamespace, string dataProviderClassName, DataOptType sqlExecType)
        public AutoDelete(string dataProviderNamespace, string dataProviderClassName, bool EnabledBuffer);
```

AutoDelete：删除数据操作属性，必须依附于接口方法。

方法介绍

方法名称	方法描述	方法类型	参数说明	返回值
AutoDelete	携带 delete 数据删除语句的构造方法	构造方法	deleteExpression：update 数据更改语句表达式 sqlExecType：sql 表达式执行类型	无

AutoDelete	同上	构造方法	deleteExpression：update 数据更改语句表达式 EnabledBuffer：是否启用缓存机制，默认为 false(不启用)	
AutoDelete	指定由提供者根据数据动态生成 delete 语句	构造方法	dataProviderNamespace：数据提供者类所在名称空间; dataProviderClassName：数据提供者类名(需继承 ISqlExpressionProvider 接口)	无
AutoDelete	同上	构造方法	dataProviderNamespace：数据提供者类所在名称空间; dataProviderClassName：数据提供者类名(需继承 ISqlExpressionProvider 接口) EnabledBuffer：是否启用缓存机制，默认为 false(不启用)	
ExecuteInterfaceMethod CodeString	在接口代理方法内自定义执行实例接口方法及数据处理代码	重写 AutoCall 类的虚方法	method：接口方法信息对象	string 类型，返回自定义执行接口方法及数据处理代码字符串

参数 deleteExpression：sql 语句与接口方法参数匹配规则

sql-deleteExpression	说明	案例
继承 selectExpression	与 selectExpression 相同	参考 selectExpression
{EntityListParaName}	匹配接口方法参数元素类型为数据实体的 List 类型参数名称	[AutoDelete(“delete from UserInfo where id=@id”)] int deleteUserInfo(List<UserInfo> userInfos); 组件自动解析 userInfos 实体集合对象，生成与之对应的 Db 参数集合 List<DbParameter> 因为接口方法参数是 List 集体的数据实体，所生成的 delete 语句和 Db 集合将会被循环执行。
{EntityParaName}	匹配接口方法参数元素类型为数据实体参数名称	参考{EntityListParaName}，区别在于无需循环执行。

配置文件：implementFactory.config
数据源相关配置（仅一项）
{ConnectionString="Data Source=(local);Initial Catalog=DbTest;User Id=sa;Password=sa;",DatabaseType="sqlserver",SqlProviderRelativePathOfDll="DataProvider",IsShowCode=true}

接口与实例关系配置（可根据项目需求配置多项）
{DllRelativePathOfImpl="BLL",ImplementNameSpace="BLL.Unit.impl",MatchImplExpression="^Com[a-z0-9_]*",InterFaceName="Pipeline",IgnoreCase=True,IsShowCode=true}

相关参数介绍

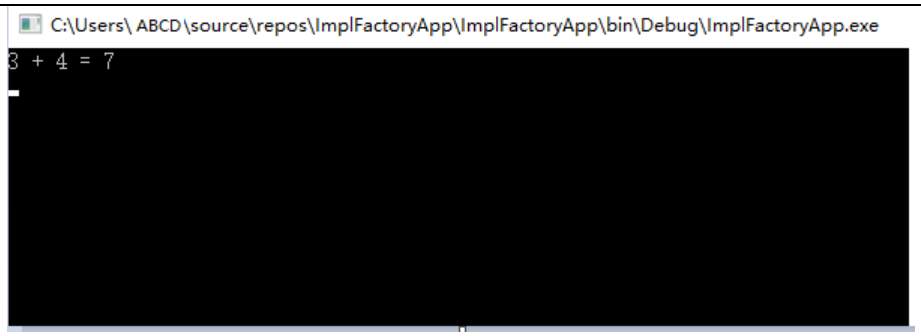
类别	参数名称	参数说明	范例
数据源配置	ConnectionString	数据库接连字符串	Data Source=(local);Initial Catalog=DbTest;User Id=sa;Password=sa;
	DatabaseType	数据库类型(sqlserver、oracle、mysql)	sqlserver
	SqlProviderRelativePathOfDll	[可选]动态 sql 提供者所在的 dll 文件程序集的相对路径,注：该提供者必须继承 System.DJ.ImplementFactory.IDataProvider 接口	扩展名.dll 可省略
	IsShowCode	是否显示临时程序集对应的代码, 设置为 true 显示，反之异然	false
接口与实例关系	DllRelativePathOfImpl	[可选] - 实例类所在 dll 文件的相对路径，如果为空,表示实例类和 exe 文件属同一 dll 文件	BLL.dll (扩展名.dll 可省略, 可写为 BLL)
	ImplementNameSpace	[可选] - 指定实现 interface 类的实例所在的 namespace	BLL.Unit.impl
	MatchImplExpression	[*必选*] - 匹配实现接口(interface)类的实例名称,可以是一个完整的类名称,但不包含 namespace. 也可以是一个正则表达式	CalculateImpl 或 ^Cal[a-z0-9_]*
	InterFaceName	[*必选*] - 接口名称，可以是一个 namespace.interfaceClassName 完整的接口名称，或部分名称空间，也可是 interfaceClassName	ImplFactory.AppTest.DataInterface.SaleOrder.ICalculate 或 ImplFactory.AppTest.DataInterface 或 ICalculate
	IgnoreCase	如果 MatchImplExpression 是正则表达式，匹配时是否忽略大小写。true 为忽略，反之异然	false – 区分大小写
	IsShowCode	是否显示对应临时程序集的代码，默认 false[不显示], true[显示]	false - 不显示

简单案例

业务模块解耦

1> 创建一个项目名称为 ImplFactoryApp 的控制台 c#项目，选择 framework v4.0，在项目中添加引用: System.DJ.ImplementFactory.dll

运行效果:




```
namespace ImplFactoryApp
{
    class Program
    {

        static void Main(string[] args)
        {
            int a = 3;
            int b = 4;
            TestUnit testUnit = new TestUnit();

            ICalculate calculate = testUnit.GetCalculate();
            int c = calculate.Sum(a, b);

            System.Console.WriteLine(string.Format("{0} + {1} = {2}", a, b, c));
            System.Console.Read();
        }
    }
}
```

创建 TestUnit.cs 文件

```
using System.DJ.ImplementFactory.Attrs;

namespace ImplFactoryApp
{
    class TestUnit: ImplementAdapter
    {
        [AutoCall]
        ICalculate calculate;   //此成员变量由 ImplementFactory 来完成接口实例装配工作

        public ICalculate GetCalculate()
        {
            return calculate;
        }
    }
}
```

创建 ICalculate 接口的实例文件 CalculateImpl.cs

```
namespace ImplFactoryApp
{
```

```
    public class CalculateImpl : ICalculate
    {
        int ICalculate.Sum(int a, int b)
        {
            return a + b;
        }
    }
}
```

创建接口文件 ICalculate.cs

```
namespace ImplFactoryApp
{
    public interface ICalculate
    {
        int Sum(int a, int b);
    }
}
```

注：由于接口 ICalculate 和接口实例 CalculateImpl 处于同一程序集，所以不需要配置 ImplementFactory.config，反之，如果接口和接口实例处在不同的程序集，则必须在配置文件中指定与该接口对应的 DllRelativePathOfImpl 属性值。当然组件提供了自动扫描机制，无需配置也可以为接口装配实例，只是这样性能会差一点。

2> 数据操作自动装配案例

如果是 dotNetCore 项目，必须引入依赖 Microsoft.CodeAnalysis.CSharp

创建一个项目名称为 ImplFactoryForDb 的 Windows 窗体运用 项目，选择 framework v4.0，并且在项目中添加引用：System.DJ.ImplementFactory.dll, System.DJ.Framework.CodeCompiler.dll, System.DJ.ImplementFactory.Commons.dll, System.DJ.ImplementFactory.Pipelines.dll

数据库类型： SQL Server

创建数据库： DbTest

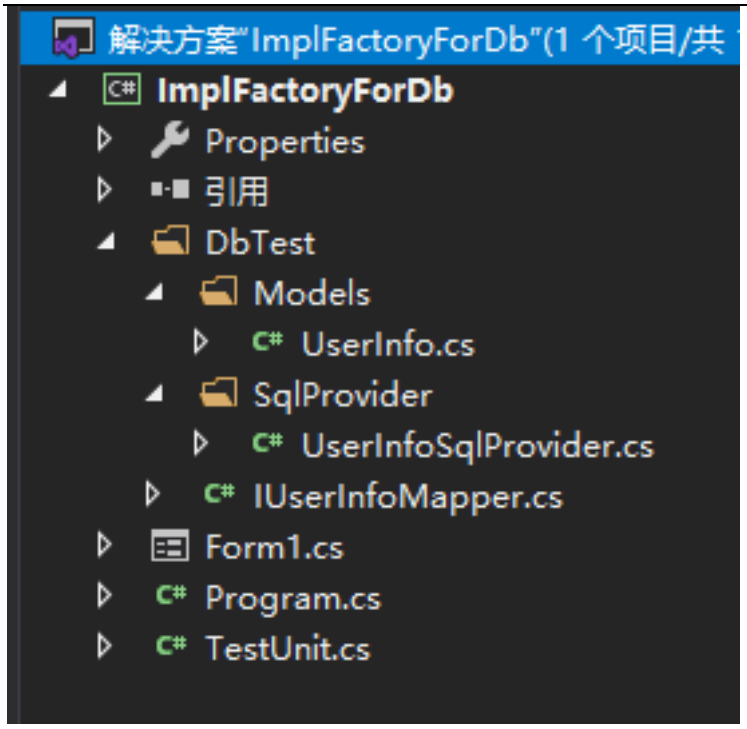
创建数据表：UserInfo

```
CREATE TABLE [dbo].[UserInfo](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [name] [varchar](50) NULL,
    [age] [int] NULL,
    [address] [varchar](500) NULL
)
```

ImplementFactory.config 数据相关配置：

```
{ConnectionString="Data Source=(local);Initial Catalog=DbTest;User Id=sa;Password=sa;",DatabaseType="sqlserver",SqlProviderRelativePath="",IsShowCode=false}
```

项目解决方案目录结构：



运行效果:

控件属性

类型	控件文本(Text)	控件名称(Name)
button	查询所有	bttnSearch
	新增	bttnInsert
	新增 DataTable	bttnInsertFromDataTable
	更改	bttnUpdate
	删除	bttnDelete
	根据名字模糊查询	bttnSearchByName
	根据 sql 提供者查询	bttnSearchBySqlProv

	根据名字和年龄查询	bttnSearchByNameAge
TextBox	textBox1	textBox1
DataGridView	dataGridView1	dataGridView1

窗体对应代码：Form1.cs

```
using Models;
using Pipeline;
using System;
using System.Collections.Generic;
using System.DJ.DJson;
using System.DJ.DJson.Commons;
using System.DJ.ImplementFactory;
using System.DJ.ImplementFactory.Commons.Attrs;
using System.Windows.Forms;

namespace ImplFactoryForDb
{
    public partial class Form1 : Form
    {
        TestUnit testUnit = new TestUnit();

        public Form1()
        {
            InitializeComponent();

            bttnSearchBySqlProv.Click += BttnSearchBySqlProv_Click;
            bttnSearchByNameAge.Click += BttnSearchByNameAge_Click;
            bttnSearchByName.Click += bttnSearchByName_Click;
            bttnSearch.Click += BttnSearch_Click;
            bttnUpdate.Click += BttnUpdate_Click;
            bttnDelete.Click += BttnDelete_Click;
            bttnInsert.Click += BttnInsert_Click;
            bttnInsertFromDataTable.Click += BttnInsertDt_Click;
        }

        void testJsonToEntity()
        {
            string json = "[\r\n{\"id\": 1, \"name\": \"张三\", \"age\": 21, \"address\": \"广东少\r\n 深圳市
南山区\"}, ";
            json += "{\"id\": 2, \"name\": \"张少涛\", \"age\": 21, \"address\": \"广东少深圳市龙华区\"},
\r\n";
            json += "{\"id\": 5, \"name\": \"张红生\", \"age\": 18, \"address\": \"广东少深圳市宝安区
\"}\r\n]";
            //json += "";

            List<UserInfo> userInfos = ExtTool.FromJson<UserInfo>(json);
        }

        private void BttnInsertDt_Click(object sender, EventArgs e)
        {
```

```
        int num = testUnit.InsertUserInfo_DataTable();
        textBox1.Text = num.ToString();
        ShowData();
    }

    private void BttnInsert_Click(object sender, EventArgs e)
    {
        int num = testUnit.InsertUserInfo();
        textBox1.Text = num.ToString();
        ShowData();
    }

    private void BttnSearchBySqlProv_Click(object sender, EventArgs e)
    {
        List<UserInfo> userInfos = testUnit.GetUserInfosBySqlProvider();
        showDataByList(userInfos);
    }

    private void BttnSearchByNameAge_Click(object sender, EventArgs e)
    {
        List<UserInfo> userInfos = new List<UserInfo>();
        UserInfo userInfo= testUnit.GetUserInfoByNameAndAge();
        if (null == userInfo)
        {
            MessageBox.Show("你要查找的数据不存在");
            return;
        }
        userInfos.Add(userInfo);
        showDataByList(userInfos);
    }

    private void bttnSearchByName_Click(object sender, EventArgs e)
    {
        List<UserInfo> userInfos = testUnit.GetUserInfosByName();
        showDataByList(userInfos);
    }

    void showDataByList(List<UserInfo> userInfos)
    {
        textBox1.Text = "";
        if (null == userInfos) return;
        string txt = "";
        foreach (var item in userInfos)
        {
            txt += item.ToJsonUnit() + "\r\n";
        }
        textBox1.Text = txt;
    }

    private void BttnSearch_Click(object sender, EventArgs e)
```

```
        {
            ShowData();
        }

        private void BttnUpdate_Click(object sender, EventArgs e)
        {
            int num = testUnit.UpdateUserInfo();
            textBox1.Text = num.ToString();
            ShowData();
        }

        private void BttnDelete_Click(object sender, EventArgs e)
        {
            int num = testUnit.DeleteUserInfo();
            textBox1.Text = num.ToString();
            ShowData();
        }

        void ShowData()
        {
            dataGridView1.DataSource = testUnit.GetUserInfos();
        }
    }
}
```

在项目根目录下创建 TestUnit.cs 文件

```
using Commons;
using Models;
using Pipeline;
using System;
using System.Collections.Generic;
using System.Data;
using System.DJ.ImplementFactory;

namespace ImplFactoryForDb
{
    class TestUnit: ImplementAdapter
    {
        [myAutoCall]
        IUserInfoMapper UserInfoMapper; //由 ImplementFactory 自动完成数据接口实例装载工作

        public List<UserInfo> GetUserInfosByName()
        {
            List<UserInfo> userInfos = UserInfoMapper.GetUserInfosByName("张");
            return userInfos;
        }

        public UserInfo GetUserInfoByNameAndAge()
        {
            UserInfo userInfo = new UserInfo();
```

```
            userInfo.name = "王五";
            userInfo.age = 20;

            UserInfo userInfo1 = UserInfoMapper.GetUserInfoByNameAndAge(userInfo);
            return userInfo1;
        }

        public List<UserInfo> GetUserInfosBySqlProvider()
        {
            UserInfo userInfo = new UserInfo();
            userInfo.age = 21;
            userInfo.address = "深圳";

            List<UserInfo> userInfos = UserInfoMapper.GetUserInfosBySqlProvider(userInfo);
            return userInfos;
        }

        public DataTable GetUserInfos()
        {
            return UserInfoMapper.GetUserInfos();
        }

        public int InsertUserInfo()
        {
            List<UserInfo> userInfos = new List<UserInfo>();
            UserInfo userInfo = new UserInfo();
            userInfo.name = "张三";
            userInfo.age = 21;
            userInfo.address = "广东少深圳市南山区";
            userInfos.Add(userInfo);

            userInfo = new UserInfo();
            userInfo.name = "张少涛";
            userInfo.age = 21;
            userInfo.address = "广东少深圳市龙华区";
            userInfos.Add(userInfo);

            userInfo = new UserInfo();
            userInfo.name = "李意";
            userInfo.age = 23;
            userInfo.address = "广东少深圳市南山区";
            userInfos.Add(userInfo);

            userInfo = new UserInfo();
            userInfo.name = "王五";
            userInfo.age = 20;
            userInfo.address = "广东少深圳市宝安区";
            userInfos.Add(userInfo);

            userInfo = new UserInfo();
```

```
        userInfo.name = "张红生";
        userInfo.age = 18;
        userInfo.address = "广东少深圳市宝安区";
        userInfos.Add(userInfo);

        userInfo = new UserInfo();
        userInfo.name = "王五";
        userInfo.age = 36;
        userInfo.address = "广西省漓江市";
        userInfos.Add(userInfo);

        int num = UserInfoMapper.InsertUserInfo(userInfos);
        return num;
    }

    public int InsertUserInfo_DataTable()
    {
        int num = 0;
        DataTable dt = new DataTable();
        dt.Columns.Add("name", typeof(string));
        dt.Columns.Add("age", typeof(int));
        dt.Columns.Add("address", typeof(string));

        DataRow row = dt.NewRow();
        row["name"] = "万君杰";
        row["age"] = 33;
        row["address"] = "湖南省长沙市";
        dt.Rows.Add(row);

        row = dt.NewRow();
        row["name"] = "史芸香";
        row["age"] = 28;
        row["address"] = "天津市";
        dt.Rows.Add(row);

        row = dt.NewRow();
        row["name"] = "李广";
        row["age"] = 26;
        row["address"] = "山东省济南市";
        dt.Rows.Add(row);

        row = dt.NewRow();
        row["name"] = "秦蕊";
        row["age"] = 22;
        row["address"] = "江西省九江市";
        dt.Rows.Add(row);

        num = UserInfoMapper.InsertUserInfo(dt);

        return num;
```

```
        }

        Random rnd = new Random();
        public int UpdateUserInfo()
        {
            List<UserInfo> userInfos = new List<UserInfo>();
            UserInfo userInfo = new UserInfo();
            userInfo.id = 1;
            userInfo.name = "张三";
            userInfo.age = 21;
            userInfo.address = "广东少深圳市南山区 - " + rnd.Next(1, 10).ToString("D2");
            userInfos.Add(userInfo);

            userInfo = new UserInfo();
            userInfo.id = 2;
            userInfo.name = "张少涛";
            userInfo.age = 21;
            userInfo.address = "广东少深圳市龙华区 - " + rnd.Next(1, 10).ToString("D2");
            userInfos.Add(userInfo);

            int num = UserInfoMapper.UpdateUserInfo(userInfos);
            return num;
        }

        public int DeleteUserInfo()
        {
            List<UserInfo> userInfos = new List<UserInfo>();
            UserInfo userInfo = new UserInfo();
            userInfo.id = 2;
            userInfos.Add(userInfo);

            userInfo = new UserInfo();
            userInfo.id = 3;
            userInfos.Add(userInfo);

            int num = UserInfoMapper.DeleteUserInfo(userInfos);
            return num;
        }
    }
}
```

在 DbTest 目录下创建接口文件 IUserInfoMapper.cs

```
using ImplFactoryForDb.DbTest.Models;
using System.Collections.Generic;
using System.Data;
using System.DJ.ImplementFactory.Commons.Attrs;

namespace ImplFactoryForDb.DbTest
{
    public interface IUserInfoMapper
```

```
    {
        [AutoSelect("select * from UserInfo where name like '%{name}%'")]
        List<UserInfo> GetUserInfosByName(string name);

        [AutoSelect("select * from UserInfo where name=@name and age=@age")]
        UserInfo GetUserInfoByNameAndAge(UserInfo userInfo);

        [AutoSelect(dataProviderNamespace: "ImplFactoryForDb.DbTest.SqlProvider",
            dataProviderClassName: "UserInfoSqlProvider")]
        List<UserInfo> GetUserInfosBySqlProvider(UserInfo userInfo);

        [AutoSelect("select * from UserInfo")]
        DataTable GetUserInfos();

        [AutoInsert(insertExpression: "if not exists(select * from UserInfo where name=@name and age=@age) begin insert into UserInfo values({userInfos}) end",
            fields: new string[] { "id" })]
        int InsertUserInfo(List<UserInfo> userInfos);

        [AutoInsert("if not exists(select * from UserInfo where name=@name and age=@age) begin insert into UserInfo(name,age,address) values(@name,@age,@address) end")]
        int InsertUserInfo(DataTable dataTable);

        [AutoUpdate(updateExpression: "update UserInfo set {userInfos} where id=@id",
            fields: new string[] { " address" } , fieldsType: FieldsType.Contain)]
        int UpdateUserInfo(List<UserInfo> userInfos);

        [AutoDelete("delete from UserInfo where id=@id")]
        int DeleteUserInfo(List<UserInfo> userInfos);
    }
}
```

在 DbTest/ Models 目录下创建数据实体文件 UserInfo.cs

```
using System.DJ.ImplementFactory;
namespace ImplFactoryForDb.DbTest.Models
{
    public class UserInfo
    {
        public int id { get; set; }
        public string name { get; set; }
        public int age { get; set; }
        public string address { get; set; }
    }
}
```

在 DbTest/ SqlProvider 目录下创建数据实体文件 UserInfoSqlProvider.cs

```
using ImplFactoryForDb.DbTest.Models;
using System.Collections.Generic;
using System.Data.Common;
using System.Data.SqlClient;
```

```
using System.DJ.ImplementFactory.Pipelines;
using System.DJ.ImplementFactory.Pipelines.Pojo;

namespace ImplFactoryForDb.DbTest.SqlProvider
{
    public class UserInfoSqlProvider : ISqlExpressionProvider
    {
        string ISqlExpressionProvider.provideSql(List<DbParameter> dbParameters,
AutoCall.DataOptType dataOptType, params object[] methodParameters)
        {
            string sql = "select * from UserInfo";
            if (null == methodParameters) return sql;
            if (0 == methodParameters.Length) return sql;

            UserInfo userInfo = methodParameters[0] as UserInfo;

            string whereStr = "";
            if (!string.IsNullOrEmpty(userInfo.name))
            {
                whereStr += " and name=@name";
                dbParameters.Add(new SqlParameter("name", userInfo.name));
            }

            if (0 < userInfo.age)
            {
                whereStr += " and age=@age";
                dbParameters.Add(new SqlParameter("age", userInfo.age));
            }

            if (!string.IsNullOrEmpty(userInfo.address))
            {
                whereStr += " and address like '%" + userInfo.address + "%'";
            }

            if (!string.IsNullOrEmpty(whereStr))
            {
                whereStr = whereStr.Substring(" and ".Length);
                sql += " where " + whereStr;
            }

            return sql;
        }
    }
}
```