

Draco.DB 类库使用说明

| | |
|-----------------------|-------------------------------------|
| <i>Name:</i> | <i>Draco.DB Library User Guider</i> |
| <i>Version:</i> | <i>0.2</i> |
| <i>Author:</i> | <i>Charles</i> |
| <i>Last Modified:</i> | <i>2013-07-22</i> |

目录

| | |
|--|----|
| 1. 概要 | 3 |
| 1.1 Draco.DB.QuickDataBase.dll | 3 |
| 1.2 Draco.DB.ORM.dll | 3 |
| 1.3 Draco.DB.DbConfiguration.dll | 3 |
| 1.4 Draco.DB.EntityGenerator.exe | 3 |
| 2. QuickDataBase | 4 |
| 2.1 配置 | 4 |
| 2.2 初始化对象 | 5 |
| 2.3 数据库操作 | 6 |
| 3. ORM | 13 |
| 3.1 概述 | 13 |
| 3.2 配置 | 13 |
| 3.3 初始化对象 | 14 |
| 3.4 如何创建实体类 CS 源码 | 14 |
| 3.5 如何操作实体类 | 15 |
| 3.6 序列化、反序列化数据库构架 | 16 |
| 3.7 扩展其它数据库类型支持 | 17 |
| 4. Draco.DB.DbConfiguration | 17 |
| 4.1 配置 | 17 |
| 4.2 初始化对象 | 18 |
| 5. 实体类生成工具 | 18 |
| 5.1 数据库连接 | 18 |
| 5.2 实体类源码生成器输入选项 | 19 |

1. 概要

Draco.DB 是一组用于操作数据库的辅助类库集合。

1.1 Draco.DB.QuickDataBase.dll

此程序集的主要设计目的是提供通用的与数据库类型无关的数据库访问接口，屏蔽数据库类型差异，实现无差别化的数据库操作。本程序集基于DbProvider工作，可以配置方式切换provider组件，具体配置参考第二章。

此程序集默认支持SQLServer, Oracle, Access, Sqlite, SQLServerCE五种数据库类型。

此程序集支持数据库类型扩展，对于未被默认支持的数据库类型，可以通过实现接口并配置的方式实现相应的数据库操作。

1.2 Draco.DB.ORM.dll

此程序集基于Draco.DB.QuickDataBase.dll工作，实现与数据库类型无关的简单的ORM功能。

此程序集支持数据库类型扩展，对于未被默认支持的数据库类型，可以通过实现接口并配置的方式实现相应的数据库操作。

1.3 Draco.DB.DbConfiguration.dll

此程序集仅实现相关配置及初始化操作功能，是可选组件。

1.4 Draco.DB.EntityGenerator.exe

ORM实体类生成器

2. QuickDataBase

2.1 配置

以下表格是使用前的初始配置项，此配置项可以以任何形式存储在任何配置文件中，以供后续初始化对象时使用。

| 名称 | 说明 | 备注 |
|-------------------------|-----------------|----|
| DataServerType | 数据库类型 | 必备 |
| ConnectionString | 数据库连接串 | 必备 |
| ProviderName | 数据库 Provider 名称 | 可选 |

➤ 关于数据库类型 DataServerType：

默认支持的数据库类型包括：

| 类型(大小写不敏感) | 说明 |
|--------------------|---------------------|
| SQLServer | SQLServer 数据库类型 |
| Oracle | Oracle 数据库类型 |
| OleDb | Access 及其它可用OleDb类型 |
| SQLite | SQLite数据库类型 |
| SQLSERVERCE | SQLSERVERCE 数据库类型 |

➤ 关于数据库连接串 ConnectionString：

参考：<http://www.connectionstrings.com>

常用数据库连接串：

| 数据库类型 | 示例 |
|-----------------------------|---|
| SQLServer (.NET) | Data Source=myServerAddress;Initial Catalog=myDataBase;User Id=myUsername;Password=myPassword; |
| SQLServer (.NET) | Data Source=myServerAddress;Initial Catalog=myDataBase;Integrated Security=SSPI; |
| SQLServer (OLE) | Provider=SQLOLEDB;Data Source=myServerAddress;Initial Catalog=myDataBase;User Id=myUsername;Password=myPassword; |
| Oracle (.NET) | Data Source=MyOracleDB;User Id=myUsername;Password=myPassword;Integrated Security=no; |
| Oracle (TNS/ODP.NET) | Data Source=(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=MyHost)(PORT=MyPort))(CONNECT_DATA=(SERVICE_NAME=MyOracleSID)));User Id=myUsername;Password=myPassword; |

| | |
|----------------------|--|
| SQLite (.NET) | Data Source=filename;Version=3;Password=myPassword; |
| Access (OLE) | Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\myFolder\myAccess2007file.accdb;Persist Security Info=False; |

➤ 关于提供器 DbProvider Name

一般在安装 DbProvider 时，安装程序会自动配置对应的 provider 名称，.NET 的 Provider 默认配置储存在配置文件：

C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config\machine.config

C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG\machine.config

中的 **DbProviderFactories** 小节中。

常见的 DbProvider 名称：

| 名称(大小写敏感) | 说明 | |
|------------------------------------|-------------------------------|--|
| System.Data.SqlClient | SQLServer (Microsoft) | |
| System.Data.OracleClient | Microsoft 提供的 Oracle Provider | |
| Oracle.DataAccess.Client | Oracle 官方提供的 Oracle Provider | |
| System.Data.OleDb | OleDb (Microsoft) | |
| System.Data.Odbc | Odbc (Microsoft) | |
| System.Data.SQLite | SQLite(开源社区) | |
| System.Data.SqlServerCe.4.0 | SQLServerCE 4.0 (Microsoft) | |
| System.Data.SqlServerCe.3.5 | SQLServerCE 3.5 (Microsoft) | |

如何自定义配置 DbProvider

示例：在 App.config 或者 web.config 文件的 configuration 节中配置以下内容

```
<system.data>
  <DbProviderFactories>
    <remove invariant="System.Data.SQLite" />
    <add name="SQLite Factory"
invariant="System.Data.SQLite" description="SQLite
factory"
type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite,
Version=1.0.66.0, Culture=neutral,
PublicKeyToken=db937bc2d44ff139"/>
  </DbProviderFactories>
```

2.2 初始化对象

相关类及对象：

| 类 | 说明 | |
|---|----|--|
|---|----|--|

| | | |
|---|-----------|--|
| Draco.DB.QuickDataBase.Configuration.ConnectionInfo | 数据库连接信息对象 | |
| Draco.DB.QuickDataBase.IDataBaseContext | 数据库上下文对象 | |
| Draco.DB.QuickDataBase.IDataBaseHandler | 数据库操作 | |

初始化过程：

读取必要配置信息→创建数据库连接信息对象→创建数据库上下文对象→获取数据库操作对象→开始操作数据库。

初始化示例：

```
String connectionString="Data Source=.;Initial Catalog = MyDB; User
Id=sa;Password=sa;"; String dataBaseType = "SQLServer";
Draco.DB.QuickDataBase.Configuration.ConnectionInfo connInfo = new
    ConnectionInfo(connectionString,
dataBaseType);
Draco.DB.QuickDataBase.IDataBaseContext ctx
= new
    Draco.DB.QuickDataBase.DataBaseContext(c
onnInfo);
Draco.DB.QuickDataBase.IDataBaseHandler handler = ctx.Handler;
//开始数据库操作
//.....
```

采用 Configuration 构架配置数据库连接信息：

示例：

在 App.config 或 web.config 中配置以下节：

```
<configSections>
  <sectionGroup name="draco_DB">
    <section name="connection"
type="Draco.DB.QuickDataBase.Configuration.ConnectionInfoSectionHandler,Draco.DB.
QuickDataBase"/>
  </sectionGroup>
</configSections>
<draco_DB>
  <connection>
    <add key="dataBaseType" value="SQLSERVER">
    <add key="connectionString" value="Data Source=.;Initial
Catalog=Nunit;User Id=sasa;Password=sasa;">
    <add key="providerName" value="System.Data.SqlClient">
  </connection>
</draco_DB>
```

代码调用方法：

```
Draco.DB.QuickDataBase.Configuration.ConnectionInfo connInfo =
(ConnectionInfo)ConfigurationSettings.GetConfig("draco_DB/connection"
);
```

2.3 数据库操作

当获取到 *Draco.DB.QuickDataBase.IDataBaseHandler* 的实例对象以后，就可以使用此对象执行数据库操作。此对象提供的数据库操作包括：

➤ 基本数据库操作

| 方法 | 说明 | 备注 |
|---------------------------------|----------------|----|
| ExecuteNonQuery | 执行非查询 SQL | |
| ExecuteQuery | 执行查询 SQL | |
| ExecuteReader | 执行 Reader 查询 | |
| ExecuteScalar | 执行 SQL，返回单值 | |
| ExecuteProcedureNonQuery | 执行非查询存储过程 | |
| ExecuteProcedureQuery | 执行查询存储过程 | |
| ExecuteProcedureReader | 执行 Reader 存储过程 | |
| ExecuteProcedureScalar | 执行存储过程，回单值 | |

➤ 类型适配及参数化 SQL

使用 `Draco.DB.QuickDataBase.IDataBaseHandler` 的实例对象获取适配器对象并创建单一参数化对象或参数化对象数组：

```
//以IDataBaseHandler对象获取Draco.DB.QuickDataBase.IDataBaseAdapter对象
Draco.DB.QuickDataBase.Adapter.IDataBaseAdapter adp = handler.DbAdapter;
//创建单一参数对象
System.Data.IDataParameter para = adp.CreateParameter("@Para", "abc");
//快速创建参数化对象数组
Draco.DB.QuickDataBase.IDataParameters parameters = new
Draco.DB.QuickDataBase.Common.DataParameters(adp);
parameters.AddParameterValue("FiledNameX", 1);
parameters.AddParameterValue("FiledNameY", "abc");
parameters.AddParameterValue("FiledNameZ", DateTime.Now);
System.Data.IDataParameter[] _Parameters = parameters.Parameters;
```

➤ 匿名参数适配

手动创建参数化对象数组依然十分繁琐，大部分应用中可以采用匿名参数适配的方式处理包含匿名参数的 SQL。

```
String SQL = "select * from tab where field1=? and (field2=? or field3>?)";
Draco.DB.QuickDataBase.ParameterizedSQL param =
handler.DbAdapter.AdaptSQLAnonymousParams(SQL, 100, "ABC", DateTime.Now);
DataSet ds =handler.ExecuteQuery(param.SQL, param.Parameters);
```

➤ SQL 生成器

手动创建常用的 select, insert, update, delete 等 SQL 语句依然是繁琐枯燥的操作步骤，可以使用 SQL 生成器来创建常用 SQL 语句及其参数化对象列表。

```
//以IDataBaseHandler对象获取Draco.DB.QuickDataBase.IDataBaseAdapter对象
Draco.DB.QuickDataBase.Adapter.IDataBaseAdapter adp = handler.DbAdapter;
Draco.DB.QuickDataBase.ISQLGenerator SQLGen = adp.GetSQLGenerator();
```

SQL 生成器提供的方法：

| 方法 | 说明 | |
|------------------------|---------------|--|
| CreateSelectSQL | 创建查询 SQL | |
| CreateInsertSQL | 创建 Insert SQL | |

| | | |
|---------------------------------|--------------------|--|
| CreateUpdateSQL | 创建 Update SQL | |
| CreateDeleteSQL | 创建 Delete SQL | |
| CreatePagedSQL | 创建分页 SQL | |
| GetDataBaseTimeSQL | 获取查询数据库时间的 SQL | |
| ConvertDateTimeToSQL | 把时间转换为数据库可识别的时间字符串 | |
| CreateDateTimeSQLSegment | 创建查询时间的 SQL 片段 | |

➤ 数据库构架操作

有时需要对数据库或数据表的定义执行查询或编辑操作，此时可使用

Draco.DB.QuickDataBase.IDataBaseSchemaHandler 接口。

```
//以IDataBaseHandler对象获取Draco.DB.QuickDataBase.IDataBaseAdapter对象
Draco.DB.QuickDataBase.Adapter.IDataBaseAdapter adp = handler.DbAdapter;
Draco.DB.QuickDataBase.IDataBaseSchemaHandler schemaHandler =
    adp.GetSchemaHandler(handler);
```

SchemaHandler 提供的功能方法：

| 方法 | 说明 |
|-----------------------------------|-------------------------------|
| GetTableNames | 获取当前数据库中的所有的表名称 |
| ReadColumns | 获取数据表中的所有的数据列的定义 |
| AddFieldToTable | 为数据表添加新的数据列 |
| GetTableComment | 获取数据表的注释信息 |
| GetColumnComment | 获取数据列的注释信息 |
| GetDataTypeByColumnName | 获取数据列的数据类型 |
| ConvertToDbType | 把字符数据类型转换为 System.Data.DbType |
| IsTableHasSequence | 判断数据表是否存在序列值 |
| GetColumnDefaultValue | 获取数据列的默认值 |
| IsAutoAddColumn | 判断数据列是否是自增列 |
| SynchSequence | 同步数据表的序列值 |
| MapAdoType | 把简要数据类型映射为 ADO 数据类型 |
| GetDbServerScript | 根据参数获取 create 数据表的 SQL |
| GetSqlOfTableInfo | 获取查询数据表信息的 SQL |
| GetAlterSqlOfColumnInfo | 获取修改数据列的 SQL |
| GetAlterAddSqlOfColumnInfo | 获取添加数据列的 SQL |

➤ 处理数据库类型相关的差异化 SQL

根据具体的业务逻辑，难免遇到一些 SQL 必须包含数据库特性或特有函数，对于此类与数据库类型相关的差异化 SQL，可以采用命名 SQL 的方式予以处理。其原理是分别编写与各种数据库类型相关的 SQL，然后在为这些 SQL 命名后分别存储到多个 XML 文件中，在执行时根据命名名称和数据库类型自动查找对应的 SQL 语句并执行。

示例：

Step1：

创建 xxx.SQLServer.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <SQL name="Test.TimeSQL">
    <![CDATA[
      select getdate() as d
    ]]>
  </SQL>
</configuration>
```

创建 xxx.Oracle.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <SQL name="Test.TimeSQL">
    <![CDATA[
      select sysdate from dual
    ]]>
  </SQL>
</configuration>
```

这里的命名规则是

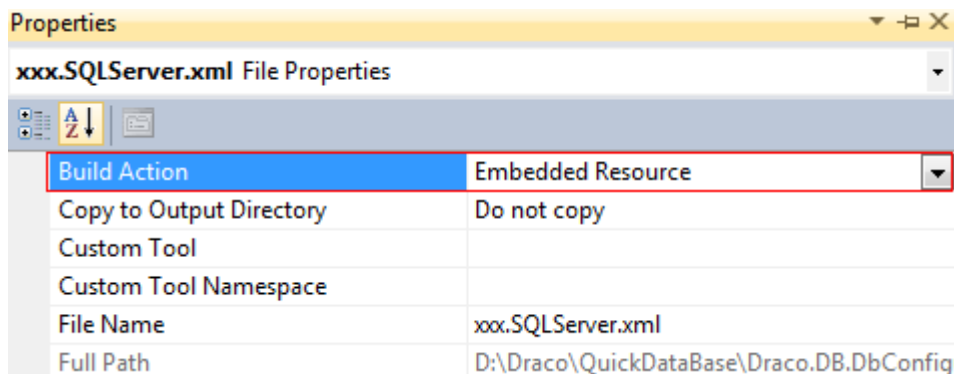
<xxx>.<DataBaseType>.xml

这里的 xxx 是自定义的任意名称，DataBaseType 是有效的数据库类型（同数据库连接信息的 DataServerType，大小写不敏感）。

Step2:

把以上配置文件添加到开发项目中，并设置文件编译选项为“嵌入的资源”或“Embedded Resource”。

在 xml 文件上点击右键，选择属性，设置 Build Action 为 Embedded Resource。



这样把 xml 文件作为资源编译到 dll 程序集或 exe 程序中。

Step3：在开发项目的AssemblyInfo.cs文件中添加

Draco.DB.QuickDataBase.Common.NamedSQLAttribute属性，以标识当前程序集包含命名SQL资源文件。

```
//标识当前程序集包含命名SQL文件
[assembly: Draco.DB.QuickDataBase.Common.NamedSQL]
```

Step4 开始使用命名 SQL

```
//以IDataBaseHandler对象获取Draco.DB.QuickDataBase.IDataBaseAdapter对象
Draco.DB.QuickDataBase.Adapter.IDataBaseAdapter adp = handler.DbAdapter;
String timeSQL = adp.NamedSQLs["Test.TimeSQL"];
Object obj = handler.ExecuteScalar(timeSQL);
```

以上代码会根据当前配置的数据库类型加载对应的命名 SQL 文件，然后读取文件中名称为“Test.TimeSQL”的 SQL 语句，这样就可以根据配置的数据库类型执行与之对应的 SQL 语句了。

➤ 处理数据库事务

Draco.DB.QuickDataBase.IDataBaseHandler 接口对象包含事务相关方法：

| 方法 | 说明 | |
|----------------------------|-------------------|--|
| StartTransaction | 在当前 Handler 中开启事务 | |
| CommitTransaction | 提交当前 Handler 中的事务 | |
| RollbackTransaction | 回滚当前 Handler 中的事务 | |

注意：事务对象保存在线程上下文中，因此事务与 Handler 是分离的，在同一个线程上可以使用多个 handler 对象操作同一个事务对象。此外，这里的事务方法可以多次开启或多次关闭或嵌套调用，但是开启与关闭必须成对调用。提示：一般每次从

Draco.DB.QuickDataBase.IDataBaseContext 实例的 Handler 属性中获取的 IDataBaseHandler 对象都是新的实例。每一个 Handler 对象都可从 __HandlerID 属性中获取唯一的 HandlerID，可以以 HandlerID 区分多个不同的 Handler 对象实例。

事务的简化处理 通常采用 Draco.DB.QuickDataBase.Common.DbTransactionScope 来简化事务的管理，避免事务执行体混淆不清或忘记提交或回滚事务。

```
using (Draco.DB.QuickDataBase.Common.DbTransactionScope scope = new
    DbTransactionScope(handler))
{
    try{
        //执行事务体

        //..... scope.Complete(); //标识事务成功
    }
    catch { }
}
```

退出 scope 时，当事务标识为成功，scope 会自动提交事务，否则 scope 会自动回滚事务。

➤ 扩展其它数据库类型支持 对于默认不被支持的数据库类型，可以以扩展接口的方式予以支持。实现新的数据库类型支持，可能需要实现的接口包括：

| 接口 | 说明 | 备注 |
|--|-----------|--|
| Draco.DB.QuickDataBase.IDataBaseContext | 数据库上下文接口 | 必需 |
| Draco.DB.QuickDataBase.IDataBaseHandler | 数据库操作接口 | 可选，默认可采用 <i>Draco.DB.QuickDataBase.Common</i> <i>. DataBaseHandler</i> 类 |
| Draco.DB.QuickDataBase.Adapter.IDataBaseAdapter | 数据库适配接口 | 必需，可从 <i>Draco.DB.QuickDataBase.Common</i> <i>. DataBaseAdapter</i> 派生 |
| Draco.DB.QuickDataBase.IDataBaseSchemaHandler | 数据库构架操作接口 | 必需 |
| Draco.DB.QuickDataBase.ISQLGenerator | SQL 生成器接口 | 必需，可从 <i>Draco.DB.QuickDataBase.Common</i> <i>. SQLGenerator</i> 派生 |

当实现以上接口之后，采用配置方式添加新的数据库类型支持。
创建配置文件**DbAdapter.config**：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <DbProviders>
    <DbProvider>
      <Name>MYSQL</Name>
    <DbAdapter>Draco.DB.QuickDataBase.MySQLClient.MySQLAdapter,
    Draco.DB.DAL.MYSQL</DbAdapter>
    </DbProvider>
  </DbProviders>
</configuration>
```

Name属性标识新的数据库类型”MySQL”,DbAdapter配置新的
*Draco.DB.QuickDataBase.Adapter.IDataBaseAdapter*的接口实现类。

然后把 DbAdapter.config 拷贝到程序运行目录下。

注意：运行目录下的 DbAdapter.config 文件会屏蔽默认的 DbAdapter 配置，如果仍然需要默认的配置，则需要把默认 DbAdapter 配置添加到 DbAdapter.config 文件。

➤ 其它特性

1. 适配表名称与字段名称 有时数据表名称或者字段名称会无意中使用到数据库的关键字，部分数据库无法解析处理包含表名称或字段名称为数据库关键字的 SQL 语句，此时就需要用到表名称适配和 字段名称适配。

Draco.DB.QuickDataBase.Adapter.IDataBaseAdapter.AdaptColumnName() 方法用来适配表名称与字段名称。示例：

| 数据库类型 | 适配前 | 适配后 |
|------------------|-------|---------|
| SQLServer | table | [table] |
| Oracle | table | “table” |

注意，Oracle 的字段或表名称添加引号以后，是大小写敏感的。

2. 执行包含多条 SQL 语句的 SQL 文件

`Draco.DB.QuickDataBase.Common.ExecuteSqlFile.ExecuteFileSql()` 方法执行 SQL 文件，其中文件第一行为多个 SQL 语句之间的分隔符，如果第一行为空，则默认采用 "--#" 为语句分隔符。`Draco.DB.QuickDataBase.Common.ExecuteSqlFile.ExecuteSqlScript()` 方法执行包含多个 SQL 语句的 SQL 文本，并需要指定语句分隔符。

3. 导出导入表结构

`Draco.DB.QuickDataBase.Common.TableSchemaInOut` 类包含表结构导入导出的方法。

4. 导出导入表数据

`Draco.DB.QuickDataBase.Common.TableDataInOut` 类包含表数据导入导出的方法。

5. 汉字串转换为拼音字符串

`Draco.DB.QuickDataBase.Utility.CharacterHelper.ConvertToLetter` 方法

6. 时间戳生成器

`Draco.DB.QuickDataBase.Utility.TimestampGenerator`

3. ORM

3.1 概述

`Draco.DB.ORM.dll` 此程序集基于 `Draco.DB.QuickDataBase` 构建，主要设计目的是实现简单的 ORM 模型，实现根据数据库自动生成数据库构架描述对象，数据库构架序列化反序列化、自动生成数据访问实体类型 C# 代码、自动生成简单 SQL，基于对象的数据库访问等功能。

本程序集同时支持数据库类型扩展，对于未被默认支持的数据库类型，可以通过实现接口扩展的方式实现相应的数据库操作及 ORM 操作

3.2 配置

本程序集支持的数据库类型及其配置与 `Draco.DB.QuickDataBase` 相同。

3.3 初始化对象

涉及到的接口与对象

| 名称 | 说明 | |
|--|-----------|--|
| Draco.DB.QuickDataBase.Configuration.ConnectionInfo | 数据库连接信息对象 | |
| Draco.DB.ORM.IORMContext | ORM 上下文 | |
| Draco.DB.ORM.IEntityHandler<T> | 实体类操作 | |

初始化步骤：

读取必要配置信息→创建数据库连接信息对象→创建ORM上下文对象→获取实体类操作对象→开始操作实体类。

初始化示例：

```
String connectionString = "Data Source=.;Initial Catalog=MyDB;User
Id=sa;Password=sa;";
String dataBaseType = "SQLServer";
Draco.DB.QuickDataBase.Configuration.ConnectionInfo connInfo = new
ConnectionInfo(connectionString, dataBaseType);
Draco.DB.ORM.IORMContext octx = new ORMContext(connInfo);
MyEntity MyEntityObj = new MyEntity();
Draco.DB.ORM.Common.EntityHandle<MyEntity> Handler =
octx.CreateHandler<MyEntity>(MyEntityObj);
```

3.4 如何创建实体类 CS 源码

1. 以代码方式创建

```
Draco.DB.ORM.IORMContext octx = new ORMContext(connInfo);
Draco.DB.ORM.Schema.Vendor.ISchemaLoader Loader = octx.SchemaLoader;
Draco.DB.ORM.Schema.Dbml.Database dataBase = Loader.Load("Name",
"Draco.DataTblCtrlLayer", null); //加载数据库构架
Draco.DB.ORM.Generator.EntityCodeGenerator Generator = new
EntityCodeGenerator(dataBase); //创建代码生成器
string Code = Generator.GenerateCode("MyEntity", "GUID", true); //创建c#源代码
System.IO.File.WriteAllText("D:\\MyEntity.cs", Code, System.Text.Encoding.UTF8); //输出c#源码到文件
```

2. 以工具方式创建

参考实体类生成工具说明。

3. 默认支持的数据库主键生成器

| 名称 | | |
|---------------------|--|--|
| GUID | 全球唯一标识，例： B5EB53AD-F865-4b8e-AEC6-B1D11437383E | |
| SIMPLEGUID | 简单 Guid 数据库主键,在保证唯一的前提下，比 Guid 更短，例：1e003ff4ceda1f0e77115197 | |
| TIMESTAMP | 时间戳数据库主键，例：201010151456311550062788 | |
| LOCAL | 数据库本地支持的数据库主键策略，SQLServer 则是自增，Oracle 则是表序列 | |
| ORM_SEQUENCE | ORM 实现的自定义序列主键，整形主键，由 ORM_SEQUENCE 表维护各个表的主键序列值 | |

4.如何扩展自定义主键生成器

实现Draco.DB.ORM.PKGenerator.IPKGenerator接口，如果需要数据库操作支持，则实现Draco.DB.ORM.PKGenerator.IDBReferencePKGenerator接口。

在创建数据库实体类源码时，把自定义的主键生成器类及其程序集完整名称作为 Generator.GenerateCode() 方法的第2个参数传入即可。

3.5 如何操作实体类

示例：

```
MyEntity MyEntityObj = new MyEntity();
MyEntityObj.Name = "name";
MyEntityObj.Tag = "tag";
Draco.DB.ORM.Common.EntityHandler<MyEntity> Handle =
octx.CreateHandler<MyEntity>(MyEntityObj);
Handle.Save(); //保存数据

System.Data.DataSet ds = Handle.GetDataSet(); //获取DataSet
System.Object obj = Handle.GetOneField("Name");
System.Collections.Generic.List<MyEntity> list =
Handle.AddExpression(Draco.DB.ORM.Common.AutoSQL.Expression.Eq("type",
2)).GetArrayList(); //附加查询条件
int i = Handle.Delete(); //删除
```

可以使用实体类中自动生成的 DAO 类简化实体类操作。
DAO 类型定义类似于以下代码：

```

/// <summary>
/// MyEntity的DAO
/// </summary>
public partial class DAO4MyEntity : EntityHandler<MyEntity>
{
    ///<summary>
    ///构造DAO
    ///</summary>
    /// <param name='entity'>实体类</param>
    /// <param name='handler'>操作接口</param>
    public DAO4MyEntity(MyEntity entity,
        Draco.DB.QuickDataBase.IDataBaseHandler handler)
        : base(entity, handler)
    {
    }
}
}

```

简化的 DAO 操作：

```

Draco.DB.QuickDataBase.IDataBaseHandler handler = octx.Handler;
MyEntity entity = new MyEntity();
entity.Name = "name123";
DAO4MyEntity DAO = new DAO4MyEntity (entity,handler);
DAO.Save();

```

3.6 序列化、反序列化数据库构架

序列化数据库构架到文件

```

//获取加载器
Draco.DB.ORM.Schema.Vendor.ISchemaLoader Loader = octx.SchemaLoader;
Draco.DB.ORM.Schema.Dbml.Database dataBase = Loader.Load("Name",
    "Draco.DataTblCtrlLayer");//加载数据库构架
using (Stream dbmlFile = File.OpenWrite("D://dbml.xml"))
{
    Draco.DB.ORM.Schema.Dbml.DbmlSerializer.Write(dbmlFile, dataBase);
}

```

从文件反序列化构架到内存对象


```

Draco.DB.ORM.Schema.Dbml.Database dataBase = null;
using (Stream dbmlFile = File.OpenRead("D://dbml.xml"))
{
    dataBase = Draco.DB.ORM.Schema.Dbml.DbmlSerializer.Read(dbmlFile);
}

```

3.7 扩展其它数据库类型支持

需要实现的接口

| 名称 | 说明 |
|--|-------------------|
| Draco.DB.ORM.Adapter.IORMAdapter | 匹配新的数据库类型工厂 |
| Draco.DB.ORM.Schema.Vendor.ISchemaLoader | 实现新的数据库构架加载器 |
| Draco.DB.ORM.AutoSQL.Common.SQLBuilder<T> | 实现新的数据库类型 SQL 自动化 |
| Draco.DB.ORM.PKGenerator.IPKGenerator | 实现新的数据库主键生成策略 |

配置 DbAdapter.config 文件，与 QuickDataBase 相同。

4. Draco.DB.DbConfiguration

Draco.DB.DbConfiguration.dll 是提供缺省配置加载和提供初始化对象的辅助组件。

4.1 配置

组件缺省从 App.config 或 web.Config 文件的 AppSettings 小节加载数据库配置信息。

| 名称 | 说明 |
|-------------------------|------------------|
| ConnectionString | 数据库连接串 |
| DataServerType | 数据库类型 |
| ProviderName | 数据库 ProviderName |

以上配置选项的意义参考 QuickDataBase 的配置章节。

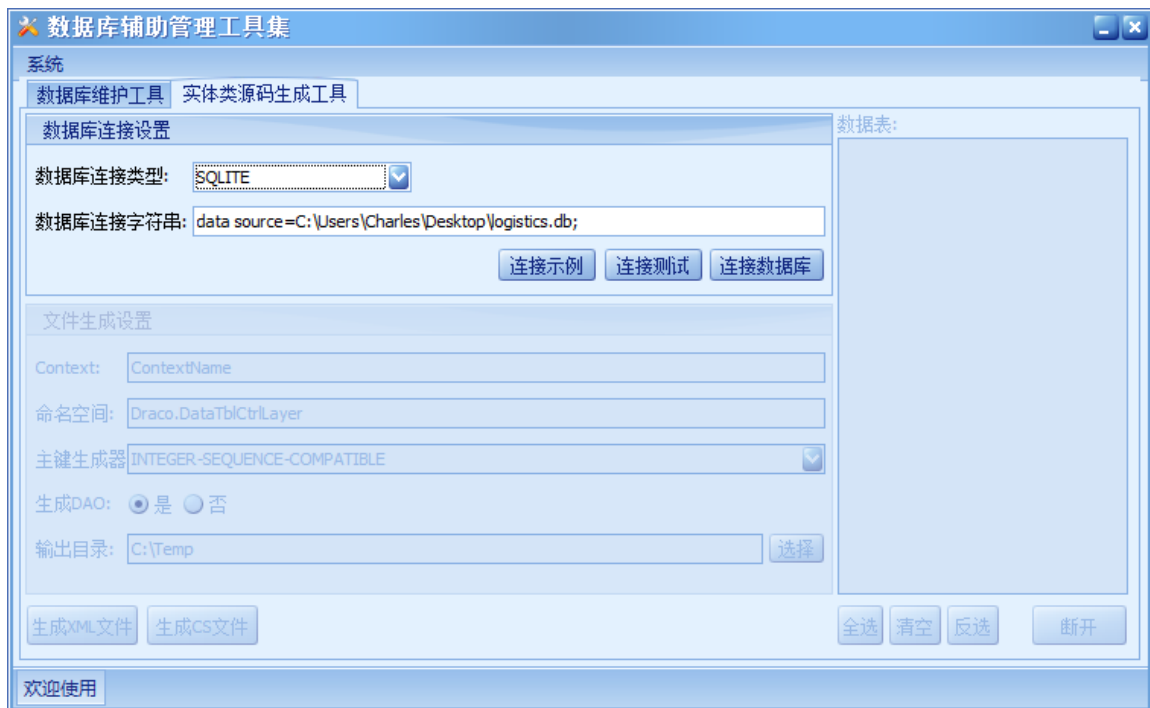
4.2 初始化对象

从 `Draco.DB.DbConfiguration.DbConfigInfo.Instance` 中获取 `Draco.DB.DbConfiguration.IDbConfigInfo` 的静态实例对象，然后从 `IDbConfigInfo` 中获取 `Draco.DB.QuickDataBase.IDataBaseContext` 和 `Draco.DB.ORM.IORMContext octx` 的实例对象。

```
Draco.DB.DbConfiguration.IDbConfigInfo DbInfo =
    Draco.DB.DbConfiguration.DbConfigInfo.Instance;
Draco.DB.QuickDataBase.IDataBaseContext ctx = DbInfo.DataBaseContext;
Draco.DB.ORM.IORMContext octx = DbInfo.ORMContext;
```

5. 实体类生成工具

实体类生成工具是为 ORM 生成实体操作类的辅助工具。

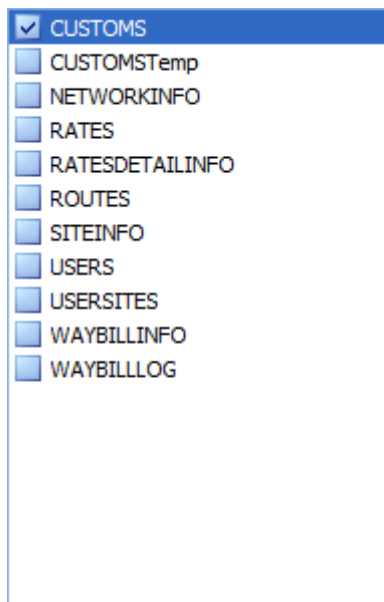


5.1 数据库连接

默认支持 SQLServer, Oracle, SQLite，配置选项参考 QuickDataBase 配置章节。

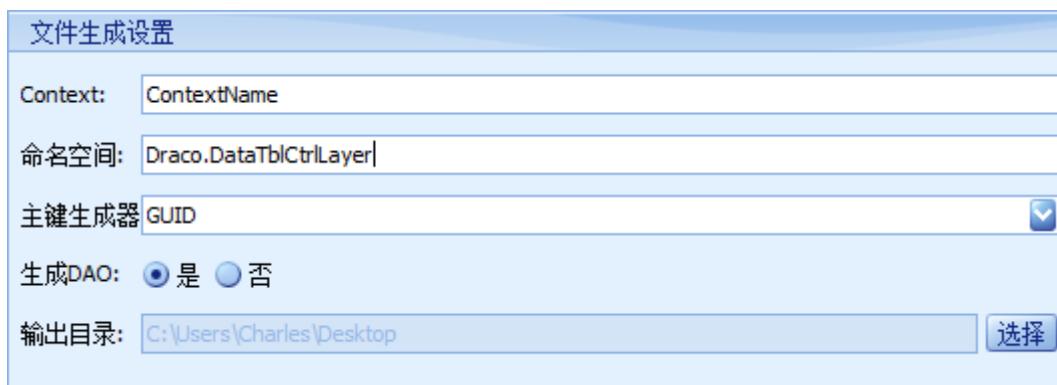
5.2 实体类源码生成器输入选项

连接数据库后，从右侧选择需要生成实体类源码的表：



| | |
|-------------------------------------|-----------------|
| <input checked="" type="checkbox"/> | CUSTOMS |
| <input type="checkbox"/> | CUSTOMSTemp |
| <input type="checkbox"/> | NETWORKINFO |
| <input type="checkbox"/> | RATES |
| <input type="checkbox"/> | RATESDETAILINFO |
| <input type="checkbox"/> | ROUTES |
| <input type="checkbox"/> | SITEINFO |
| <input type="checkbox"/> | USERS |
| <input type="checkbox"/> | USERSITES |
| <input type="checkbox"/> | WAYBILLINFO |
| <input type="checkbox"/> | WAYBILLOG |


设置实体类 CS 源码的命名空间和主键生成器，选择输出目录



文件生成设置

Context:

命名空间:

主键生成器: 

生成DAO: ☒ 是 ☐ 否

输出目录:

各配置选项

| 选项名称 | 说明 | |
|---------|-----------------|------------------|
| Context | 用于生成表构架 xml 文件 | (可选,xml 文件也可选) |
| 命名空间 | 用于实体类命名空间 | |
| 主键生成器 | 选择可用的主键生成器 | |
| 生成 DAO | 是否生成辅助 DAO 类型 | |
| 输出目录 | 输出目录，如果目录不存在则错误 | |



生成 XML 文件：生成表结构 XM 文件（运行时不需要此文件）

生成 CS 文件：生成实体类 CS 源码