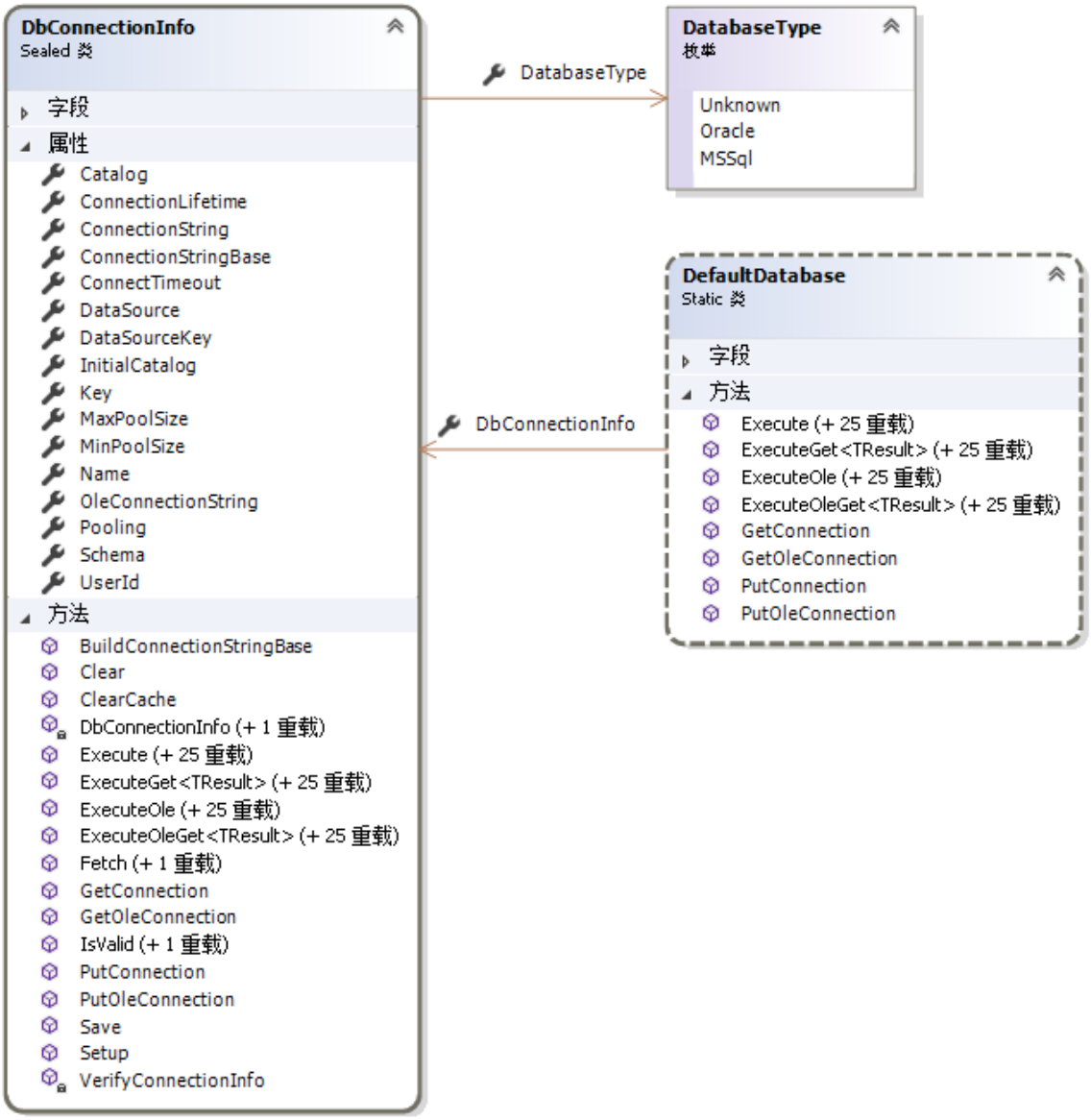
25 持久层开发助手

Phenix丶的持久层引擎与业务框架组合在一起，实现了透明化的持久层操作，使得你的业务逻辑层和数据持久层的代码可以精炼到最少，基本上可以做到免 SQL 编程。不过，Phenix丶并没有关闭直接写持久层代码的开发界面，在 Phenix.Core.Data 命名空间里可以找得到一系列的编程接口。

25.1 DbConnectionInfo



DbConnectionInfo 是持久层开发界面的总入口，负责管理数据库的链接，平时用到的 DefaultDatabase 是其特例，提供了默认数据库的操作接口。因为一般情况下我们都是操作的默认数据库，所以我们更多地会接触到 DefaultDatabase，比如以下这套函数代码段的写法（摘自 Phenix.Security.Plugin.TranslationUserNumber 的 Plugin 类）：

```
private static string Translation(DbConnection connection, string userNumber)
{
```

```
      using (DataReader reader = new DataReader(connection,
@"select US_UserNumber
  from PH_User
  where US_Name = :US_Name or US_UserNumber = :US_UserNumber",
        CommandBehavior.SingleRow))
      {
        reader.CreateParameter("US_Name", userNumber);
        reader.CreateParameter("US_UserNumber", userNumber);
        if (reader.Read())
          return reader.GetNullableString(0);
      }
      throw new UserNotFoundException(userNumber);
    }

    /// <summary>
    /// 转译登录工号
    /// </summary>
    public string Translation(string userNumber)
    {
      return DefaultDatabase.ExecuteGet(Translation, userNumber);
     }
```

多数据库的操作，需要用到 DataSourceKey 概念，具体请见“22.数据库集群”章节。我们可以在业务类上申明默认下操作的是哪个数据库（标记 ClassAttribute.DataSourceKey 内容），也可以动态地根据需要指定数据库链接，比如 Fetch 业务对象时就有提供相关的参数（此时静态申明就无效了）：

```
    /// <summary>
    /// 构建业务对象集合
    /// </summary>
    /// <param name="dataSourceKey">数据源键</param>
    /// <param name="criteriaExpression">条件表达式</param>
    /// <param name="orderByInfos">数据排列顺序队列</param>
    public static T Fetch(string dataSourceKey, Expression<Func<TBusiness, bool>> criteriaExpression,
params OrderByInfo[] orderByInfos)
```

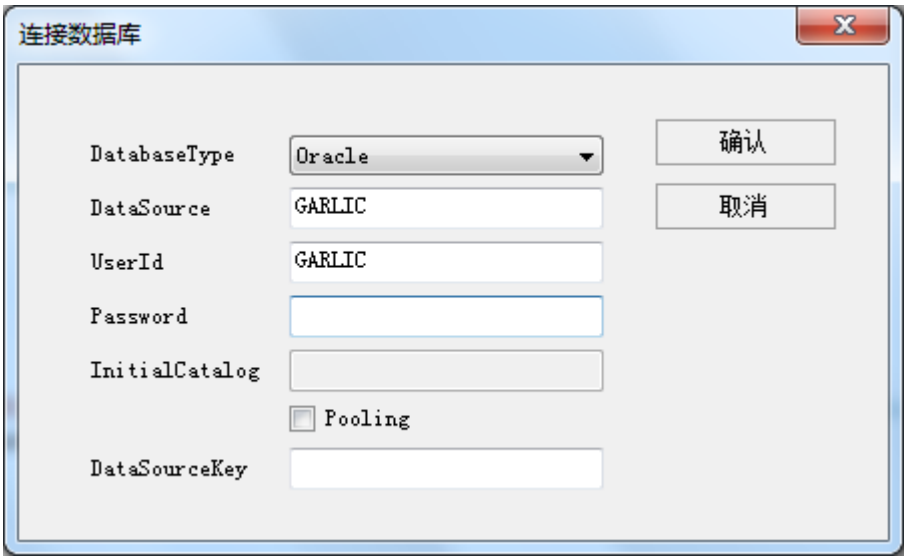
无论如何，执行到持久层的代码，都是通过 DbConnectionInfo 构造出连接串信息：

```
    /// <summary>
    /// 取DbConnection信息
    /// dataSourceKey = String.Empty
    /// </summary>
    /// <returns>DbConnection信息</returns>
```

```
public static DbConnectionInfo Fetch()

/// <summary>
/// 取DbConnection信息
/// </summary>
/// <param name="dataSourceKey">数据源键</param>
/// <returns>DbConnection信息</returns>
public static DbConnectionInfo Fetch(string dataSourceKey)
```

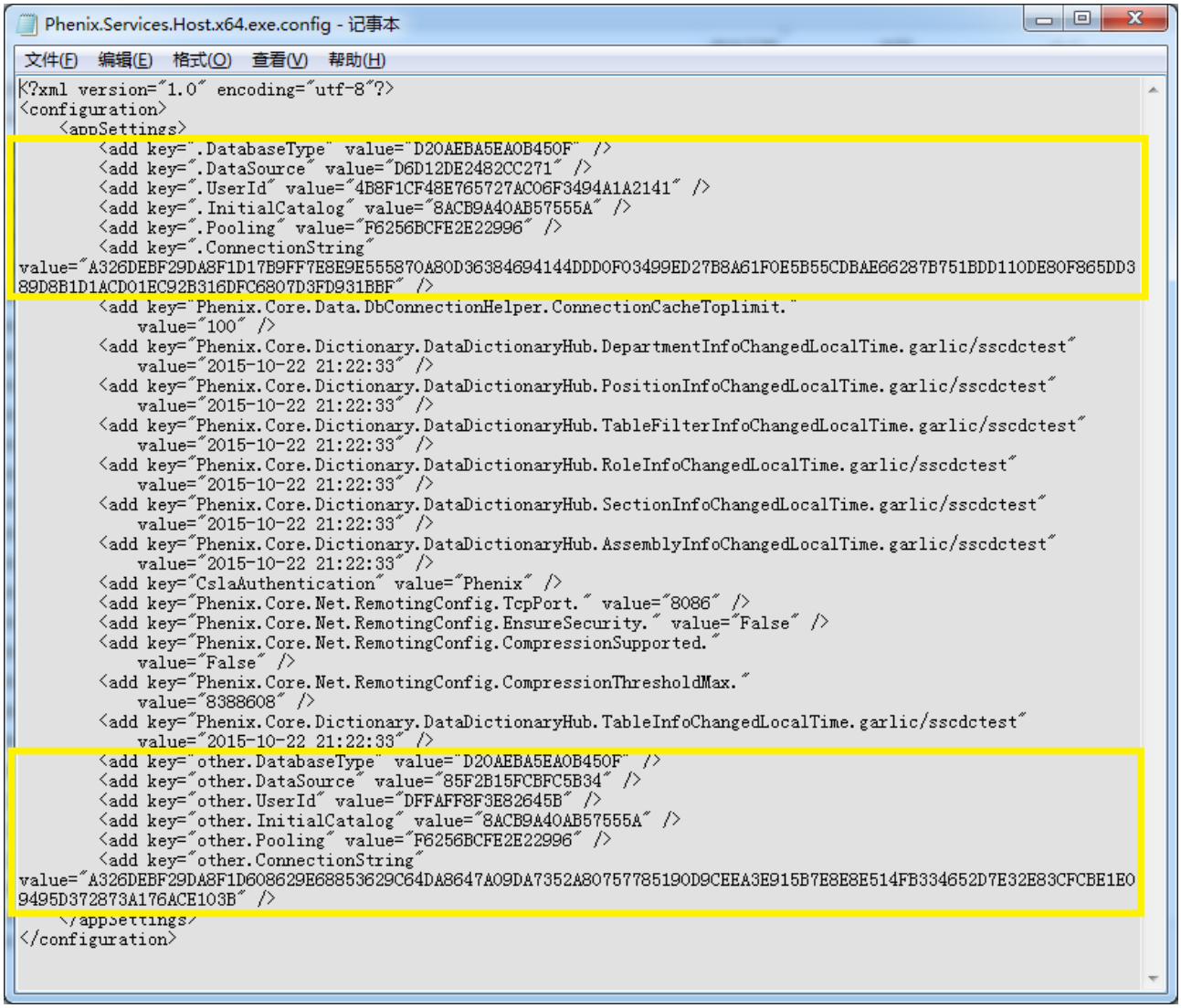
连接串的信息保存在了应用程序的 Config 文件里（IIS 环境下是在 webconfig 里），一般我们是通过配置界面来保存的：



其实也可以用编写代码的方法配置：

```
new DbConnectionInfo("other", DatabaseType.Oracle, "mydb", "mydbuser", "mydbpassword", "", true).Save();
```

这行示例代码，初始化了 DataSourceKey="other"的数据库连接串，然后调用 Save()函数将配置信息保存在了 config 文件里：



25.2 Execute/ExecuteOle、ExecuteGet/ExecuteOleGet

只要拿到 DbConnectionInfo 对象，就可以操作数据库了。

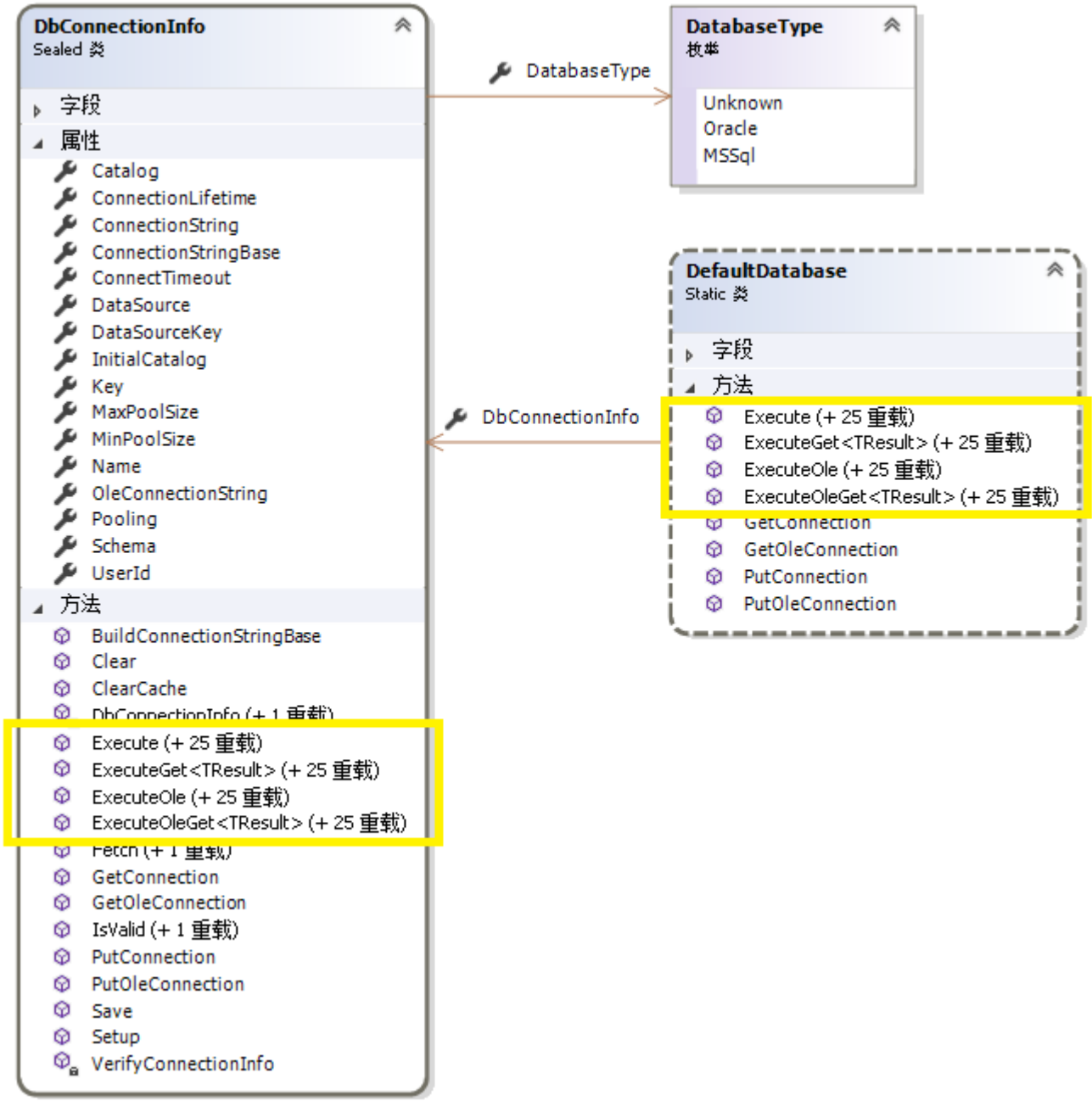
Phenix 提供了 ADO 和 OLEDB 两套数据库访问方式，在执行数据库操作时可以自行选择，区别在于执行函数的不同，名称各自为 Execute/ExecuteOle、ExecuteGet/ExecuteOleGet。比如：

```
/// <summary>
/// 转译登录工号
/// </summary>
public string Translation(string userNumber)
{
  return DefaultDatabase.ExecuteOleGet(Translation, userNumber);
 }
```

也可以这样写，效果是一样的：

```
/// <summary>
/// 转译登录工号
/// </summary>
public string Translation(string userNumber)
{
  return DbConnectionInfo.Fetch().ExecuteOleGet(Translation, userNumber);
 }
```

Execute 系列函数有很多的重载，可以为其回调函数传递至少一打的参数量：



向 Execute()函数传递回调函数，这种做法是因为要在 Execute()函数里，将数据库的连接资源、事务的控制交给 Phenix丫处理，来减轻你的编码量、避免失误。

这样，因为事务是由 Execute()函数发起的，所以在你的回调函数里千万不要操作事务，应该交给 Execute()函数来“打扫战场”。

```
private void CompanyRegister(System.Data.Common.DbTransaction transaction)
{
    try
    {
        var company = Company.New(_companyUserView);
        company.Disabled = false;
        company.Code = RandomStrHelper.GetChineseSpell(company.Chnname);

        var companyUser = CompanyUser.New(_companyUserView);
        companyUser.SetDefaultValue(company.SCP_ID);


        var user = User.New(_companyUserView.LoginName.ToUpper(), _companyUse

        var userRole = UserRole.New(user.US_ID, GetAdminRoleId());


        company.Save(transaction,companyUser, user, userRole);

        Phenix.Business.Security.UserPrincipal.ChangePassword(transaction, _c
        RegisterSuccess();
    }
    catch (Exception er)
    {
        transaction.Rollback();
        RegisterFail(er.Message);
    }
    finally
    {
        //_companyUserView = null;
    }
}
```

以上举了一个反例，在拦截异常的陷阱里把异常信息给吃掉了，还将事务做了回滚（Rollback）。这样做，会使得 Execute()函数在执行完这个回调函数后，没有异常可拦截，也就认为它是正常执行完的，就接着继续提交（Commit）事务了，结果抛出一个不希望出现的异常 InvalidOperationException（The transaction has already been committed or rolled back)。

为感性理解，请看一下 Execute()函数操作事务的代码：

```
using (DbTransaction transaction = CreateTransaction(connection))
  try
  {
    result = doExecute(transaction, in1, in2, in3, in4, in5, in6, in7, in8, in9, in10, in11, in12);
    transaction.Commit();
  }
```

```
        catch
        {
          transaction.Rollback();
          throw;
        }
```

所以，你的回调函数（doExecute）中如发生有任何非正常事件，都请以抛出异常的方式逐层传递出来：

```
    private static string Translation(DbConnection connection, string userNumber)
    {
      using (DataReader reader = new DataReader(connection,
@"select US_UserNumber
  from PH_User
  where US_Name = :US_Name or US_UserNumber = :US_UserNumber",
        CommandBehavior.SingleRow))
      {
        reader.CreateParameter("US_Name", userNumber);
        reader.CreateParameter("US_UserNumber", userNumber);
        if (reader.Read())
          return reader.GetNullableString(0);
      }
      throw new UserNotFoundException(userNumber);
    }
```

Execute()函数在拦截到你的回调函数（doExecute）异常时会先回滚事务（即使以上案例中未用到事务时也请抛出异常），然后继续将异常抛给调用方，你可以在调用方的代码里拦截异常并处理它们：

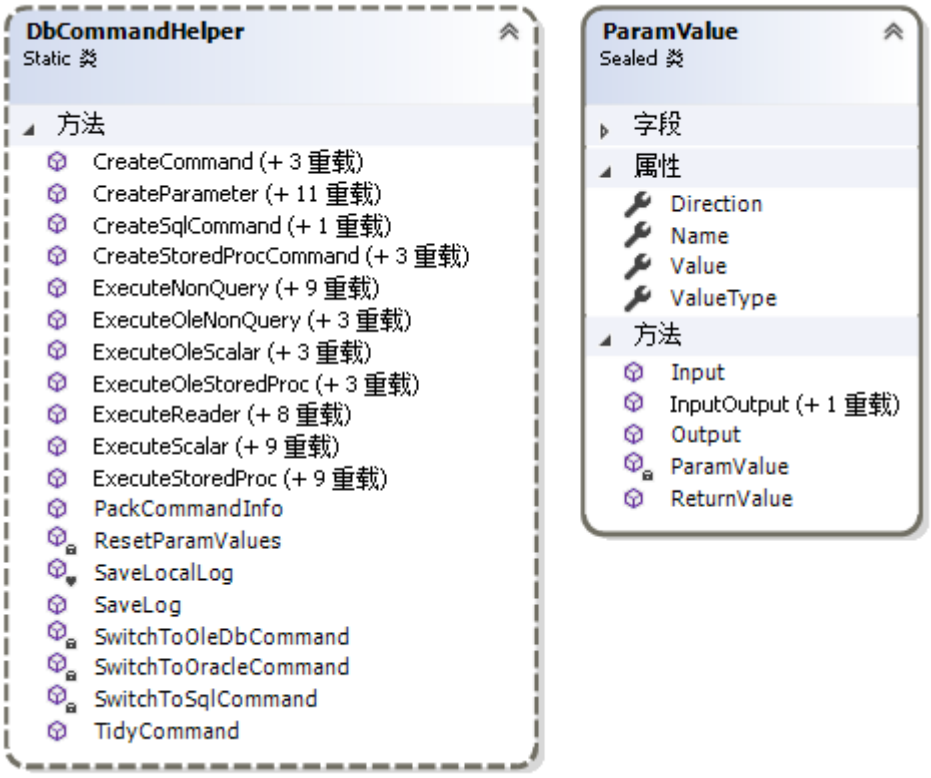
```
    /// <summary>
    /// 转译登录工号
    /// </summary>
    public string Translation(string userNumber)
    {
      try
      {
        return DefaultDatabase.ExecuteOleGet(Translation, userNumber);
      }
      catch (UserNotFoundException e)
      {
        return null;
      }
    }
```

至于处理完后是否继续向外抛，还是吃掉它，就交给你自行判断了。

在此提醒一下，就是“单一职责”在函数编码中的具体落实。遵守这一原则到位的话，是可以降低代码的复杂度，为维护工作减轻理解代码的负担，避免差错的。请不要在核心代码（本处是指你写的回调函数（doExecute））里吃掉异常，也不能代劳他人做提交事务、回滚事务的动作，更不能释放事务。也就是 Execute()函数 new 出来的 transaction 应该由它自己释放，Execute()函数发起的事务也应该由它自己全权负责，做到各尽其职。

Phenix丶在这里有区别其他框架的做法，是显式地传递 connection、transaction 给到回调函数，这可以让你在代码里就明确是否有被事务控制。

25.3 DBCommandHelper



DBCommandHelper 负责具体的数据库操作，提供了 DbCommand、DbDataReader、DbParameter 等生成器。在此，希望你不要自己 new 出来这些对象，务必交给 DBCommandHelper 构建出来，这可以让你的系统兼容到各类的数据库，至少现在能做到在 Oracle 和 MSSql 之间的自由切换。

所有操作数据库的函数都重载了有提供 DbConnectionInfo 参数，如不传值的话就等于是在操作默认的数据库。

25.3.1DbCommand 生成器

以下例子改编自 Phenix.Core.Data.DataHub 的服务端代码。

```
    public object FetchList(ICriterions criterions)
    {
      return DbConnectionInfo.Fetch(criterions.DataSourceKey).ExecuteGet((Func<DbConnection,
ICriterions, object>)FetchList);
    }

    public object FetchList(DbConnection connection, ICriterions criterions)
    {
      using (DbCommand command = DbCommandHelper.CreateSqlCommand(connection, criterions.Sql))
      {
        return FetchList(command, criterions.ResultCoreType);
      }
    }

    private static IList<object> FetchList(DbCommand command, Type entityType)
    {
      List<object> result = new List<object>();
      using (DbDataReader reader = DbCommandHelper.ExecuteReader(command,
CommandBehavior.SingleResult))
      {
        IList<FieldMapInfo> fieldMapInfos = ClassMemberHelper.GetFieldMapInfos(entityType, reader);
        while (reader.Read())
        {
          object obj = Activator.CreateInstance(entityType, true);
          if (EntityHelper.Fetch(reader, obj, fieldMapInfos))
            result.Add(obj);
        }
      }
      return result;
     }
```

注意及时释放从生成器获得的 DbCommand 对象。生成器是帮你生成对象，它的职责仅此而已，你要对获得的对象负责到底。

25.3.2DbDataReader 生成器

```
    private static IList<object> FetchList(DbCommand command , Type entityType)
    {
      List<object> result = new List<object>();
      using (DbDataReader reader = DbCommandHelper.ExecuteReader(command,
CommandBehavior.SingleResult))
      {
        IList<FieldMapInfo> fieldMapInfos = ClassMemberHelper.GetFieldMapInfos(entityType, reader);
        while (reader.Read())
```

```
        {
          object obj = Activator.CreateInstance(entityType, true);
          if (EntityHelper.Fetch(reader, obj, fieldMapInfos))
            result.Add(obj);
        }
      }
      return result;
    }
```

注意及时释放从生成器获得的 DbDataReader 对象，也应尽量避免在游标遍历时耗费太多时间。

25.3.3DbParameter 生成器

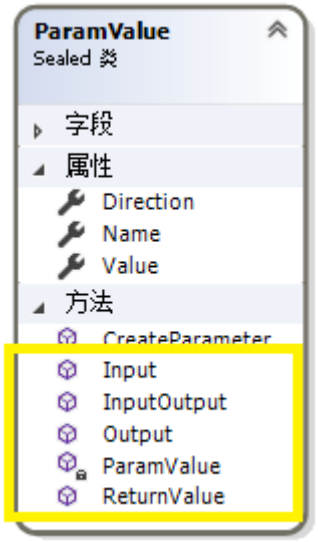
DbCommand 有 Parameters 属性，用惯了 ADO.NET 会直接在代码里向这个属性赋值。但是，当你在用 Phenix丶时，请务必避免这种写法，原因还是 Phenix丶提供的 DBCommandHelper 生成器考虑到了数据库的兼容性问题。

```
        using (DbCommand assemblyInsertCommand = DbCommandHelper.CreateCommand(transaction,
@"insert into PH_Assembly
  (AS_ID, AS_Name, AS_Caption)
  values
  (:AS_ID, :AS_Name, :AS_Caption)"))
        {
          DbCommandHelper.CreateParameter(assemblyInsertCommand, "AS_ID", assemblyId);
          DbCommandHelper.CreateParameter(assemblyInsertCommand, "AS_Name", assemblyName);
          DbCommandHelper.CreateParameter(assemblyInsertCommand, "AS_Caption", assemblyCaption);
          DbCommandHelper.ExecuteNonQuery(assemblyInsertCommand);
        }
```

以上代码还是略显繁琐，所以 DBCommandHelper 提供了更精简的写法：

```
        DbCommandHelper.ExecuteNonQuery(transaction,
@" insert into PH_Assembly
  (AS_ID, AS_Name, AS_Caption)
  values
  (:AS_ID, :AS_Name, :AS_Caption)",
          ParamValue.Input("AS_ID", assemblyId),
          ParamValue.Input("AS_Name", assemblyName),
          ParamValue.Input("AS_Caption", assemblyCaption));
```

注意 ParamValue 有提供了 Input、OutPut、ReturnValue 工厂函数，请按需使用：



25.3.4ExecuteNonQuery 方法

```
        DbCommandHelper.ExecuteNonQuery(transaction,
@" insert into PH_Assembly
  (AS_ID, AS_Name, AS_Caption)
  values
  (:AS_ID, :AS_Name, :AS_Caption)",
          ParamValue.Input("AS_ID", assemblyId),
          ParamValue.Input("AS_Name", assemblyName),
          ParamValue.Input("AS_Caption", assemblyCaption));
```

25.3.5ExecuteReader 方法

```
      using (DataReader reader = new DataReader(DbCommandHelper.ExecuteReader(connection,
@"select AS_ID
  from PH_Assembly
  where upper(AS_Name) = :AS_Name ",
        CommandBehavior.SingleRow,
        ParamValue.Input("AS_Name", assemblyName .ToUpper())))
      {
        return reader.Read() ? reader.GetNullableInt64(0) : -1;
       }
```

25.3.6ExecuteScalar 方法

```
      return (long)(DbCommandHelper.ExecuteScalar(connection,
@"select AS_ID
```

```
from PH_Assembly
where upper(AS_Name) = :AS_Name",
          ParamValue.Input("AS_Name", assemblyName .ToUpper())) ?? -1);
```

25.3.7ExecuteStoredProc 方法

```
ParamValue[] paramValues = DbCommandHelper.ExecuteStoredProc(transaction, "PRODUCT_CARD_NO_P",
  ParamValue.Input("iID", iID),
  ParamValue.Input("iType", iType),
  ParamValue.Output("oCardNo", typeof(string)));
return (string)paramValues[2].Value;
```

25.4 EasyEntity 的持久层方法

Phenix 不鼓励在你的代码中混合 SQL 编程，应该由 ORM 框架解耦数据库底层结构。虽说 Phenix.Business（封装 CSLA）是功能全面的业务框架，但在纯粹服务端（+持久层）的编程场景下却显得过于笨重，所以 Phenix 又提供了一套轻量级的 EasyEntity 框架（在 Phenix.Core.Data 命名空间）以适应面向服务的编程场景。

EasyEntity 框架的开发界面与 Phenix.Business 尽可能保持一致，采用 Phenix 同一套 ORM 框架，能将面向数据库编程转变为面向对象的编程，但又能低耗地处理数据、高效地操作数据库表记录。

25.4.1新增记录

```
int insertCount = 10;
long?[] userIds = null;
userIds = Phenix.Core.Data.DefaultDatabase.ExecuteGet(TestInsert, insertCount);

private static long?[] TestInsert(DbTransaction transaction, int count)
{
  List<long?> result = new List<long?>(count);
  for (int i = 0; i < count; i++)
  {
    string s = Phenix.Core.Data.Sequence.Value.ToString().Substring(5);
    UserEasy user = UserEasy.New(s, s, s, null, null, null, 0, null, null, null, 0, null);
    if (user.Save(transaction))
      result.Add(user.US_ID);
  }
  return result.ToArray();
}
```

本例代码见“Phenix.Test.使用指南.25.4”工程。

EasyEntity 类的代码可由 Addin 工具生成（见“03.Addin 工具使用方法”的“生成轻量级的 Entity 类”章节），包含了一个可为所有字段赋值的 New()函数：

```
public static UserEasy New(string usernumber, string password, string name, DateTime? login, DateTime? logout, DateTime? loginfailure, int? loginfailurecount, string loginaddress, long? US_DP_ID, long? US_PT_ID, int? locked, DateTime? passwordchangedtime)
    {
      UserEasy result = NewPure();
      result._usernumber = usernumber;
      result._password = password;
      result._name = name;
      result._login = login;
      result._logout = logout;
      result._loginfailure = loginfailure;
      result._loginfailurecount = loginfailurecount;
      result._loginaddress = loginaddress;
      result._US_DP_ID = US_DP_ID;
      result._US_PT_ID = US_PT_ID;
      result._locked = locked;
      result._passwordchangedtime = passwordchangedtime;
      return result;
    }
```

通过 New()函数可快速新增并初始化对象，然后调用 Save()函数提交到数据库。你可以根据需要传入数据库连接还是事务对象：

```
/// <summary>
/// 保存
/// 调用时运行在持久层的程序域里
/// </summary>
public bool Save(DbConnection connection);

/// <summary>
/// 保存
/// 调用时运行在持久层的程序域里
/// </summary>
public bool Save(DbTransaction transaction);
```

所以这些代码只适合写在服务端代码里，比如继承自 Phenix.Core.Data.ServiceBase<T>的 Service 对象（相当于 Phenix.Business 的 Command 对象）的 DoExecute()重载函数里。

25.4.2更新记录

如果照搬 Phenix.Business 的写法，也是可以将数据 Fetch()到本地 EasyEntity 对象，然后修改它的属性再 Save()到数据库。但这样做，对于执行在服务端的代码，不需要和客户端界面交互，就显得即繁琐也很低效了。此时，我们只需一行代码即可实现同样的功能：

```
UserEasyList.UpdateRecord(transaction, UserEasy.US_IDProperty == userId,
  PropertyValue.Set(UserEasy.LockedProperty, 1),
  PropertyValue.Set(UserEasy.LogoutProperty, DateTime.Now));
```

注意，是在 EntityListBase<T, TEntity>里提供的接口方法：

```
/// <summary>
/// 更新记录
/// 条件类的字段映射关系请用Phenix.Core.Mapping.CriteriaFieldAttribute标注
/// </summary>
/// <param name="connection">数据库连接</param>
/// <param name="criteria">条件对象</param>
/// <param name="propertyValues">属性值队列</param>
    public static int UpdateRecord(DbConnection connection, ICriteria criteria, params PropertyValue[] propertyValues);

/// <summary>
/// 更新记录
/// </summary>
/// <param name="connection">数据库连接</param>
/// <param name="criteriaExpression">条件表达式</param>
/// <param name="propertyValues">属性值队列</param>
public static int UpdateRecord(DbConnection connection, Expression<Func<TEntity, bool>> criteriaExpression, params PropertyValue[] propertyValues);

/// <summary>
/// 更新记录
/// </summary>
/// <param name="connection">数据库连接</param>
/// <param name="criteriaExpression">条件表达式</param>
/// <param name="propertyValues">属性值队列</param>
public static int UpdateRecord(DbConnection connection, CriteriaExpression criteriaExpression, params PropertyValue[] propertyValues);

/// <summary>
/// 更新记录
/// 条件类的字段映射关系请用Phenix.Core.Mapping.CriteriaFieldAttribute标注
/// </summary>
/// <param name="transaction">数据库事务</param>
```

```
    /// <param name="criteria">条件对象</param>
    /// <param name="propertyValues">属性值队列</param>
    public static int UpdateRecord(DbTransaction transaction, ICriteria criteria, params PropertyValue[] propertyValues)

    /// <summary>
    /// 更新记录
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    /// <param name="criteriaExpression">条件表达式</param>
    /// <param name="propertyValues">属性值队列</param>
    public static int UpdateRecord(DbTransaction transaction, Expression<Func<TEntity, bool>> criteriaExpression, params PropertyValue[] propertyValues)

    /// <summary>
    /// 更新记录
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    /// <param name="criteriaExpression">条件表达式</param>
    /// <param name="propertyValues">属性值队列</param>
    public static int UpdateRecord(DbTransaction transaction, CriteriaExpression criteriaExpression, params PropertyValue[] propertyValues)
```

这些函数的最后一个参数是数目可变的属性值对象数组，为 Update 数据表提供需要更新的字段值。

```
UserEasyList.UpdateRecord(transaction, UserEasy.US_IDProperty == userId,
  PropertyValue.Set(UserEasy.LockedProperty, 1),
  PropertyValue.Set(UserEasy.LogoutProperty, DateTime.Now));
```

25.4.3删除记录

同样你可以 Fetch()得到一个 EasyEntity 对象，然后调用它的 Delete()函数再 Save()它到数据库，也可以参考以下代码，简单地直接通知数据库删除表记录。

```
UserEasyList.DeleteRecord(transaction, UserEasy.US_IDProperty == userId);
```

接口方法也是在 EntityListBase<T, TEntity>里：

```
/// <summary>
/// 删除记录
/// 条件类的字段映射关系请用Phenix.Core.Mapping.CriteriaFieldAttribute标注
/// </summary>
/// <param name="connection">数据库连接</param>
/// <param name="criteria">条件对象</param>
 public static int DeleteRecord(DbConnection connection, ICriteria criteria)

/// <summary>
/// 删除记录
/// </summary>
/// <param name="connection">数据库连接</param>
/// <param name="criteriaExpression">条件表达式</param>
public static int DeleteRecord(DbConnection connection, Expression<Func<TEntity, bool>>
criteriaExpression)

/// <summary>
/// 删除记录
/// </summary>
/// <param name="connection">数据库连接</param>
/// <param name="criteriaExpression">条件表达式</param>
public static int DeleteRecord(DbConnection connection, CriteriaExpression criteriaExpression)

/// <summary>
/// 删除记录
/// 条件类的字段映射关系请用Phenix.Core.Mapping.CriteriaFieldAttribute标注
/// </summary>
/// <param name="transaction">数据库事务</param>
/// <param name="criteria">条件对象</param>
```

```
    public static int DeleteRecord(DbTransaction transaction, ICriteria criteria)

    /// <summary>
    /// 删除记录
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    /// <param name="criteriaExpression">条件表达式</param>
    public static int DeleteRecord(DbTransaction transaction, Expression<Func<TEntity, bool>>
criteriaExpression)

    /// <summary>
    /// 删除记录
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    /// <param name="criteriaExpression">条件表达式</param>
    public static int DeleteRecord(DbTransaction transaction, CriteriaExpression criteriaExpression)
```