1 如何绕过默认的持久层操作

1.1 需求场景

一般业务对象（集合）的 Fetch 方法，都可以通过在业务类上打 ClassAttribute 标签来指定 Fetch 的数据源是在哪个表还是哪个视图，见以下示例：

```
/// <summary>
/// 程序集
/// </summary>
[Phenix.Core.Mapping.ClassAttribute("PH_ASSEMBLY", FriendlyName = "程序集"),
System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("程序集")]
public abstract class AssemblyInfo<T> : Phenix.Business.BusinessBase<T> where T : AssemblyInfo<T>

/// <summary>
/// 程序集类信息
/// </summary>
[Phenix.Core.Mapping.ClassAttribute("PH_ASSEMBLYCLASS", FetchScript = "PH_ASSEMBLYCLASSINFO",
FriendlyName = "程序集类信息"), System.SerializableAttribute(),
System.ComponentModel.DisplayNameAttribute("程序集类信息")]
public abstract class AssemblyClassInfo<T> : Phenix.Business.BusinessBase<T> where T :
AssemblyClassInfo<T>
```

而 Fetch 的条件，可以通过参数方式传入 Criteria 对象、Linq 表达式等来实现（见“使用指南.17.条件检索业务对象”），对于分页检索也有一套机制来满足需求（见“使用指南.16.分页检索业务对象”）。

这些方法，已基本满足了 90%以上的开发需求。剩下的，是那些条件（SQL）语句复杂到必须用手工拼写才能达成目的的开发场景，此时需要干预业务类的持久层数据操作。

1.2 实现方法

1.2.1 持久层

Phenix丶封装了 CSLA 的业务框架，但并没有屏蔽掉其开发接口，比如：

```
protected override void DataPortal_Fetch(object criteria)
protected override void DataPortal_Update()
```

虽然 Phenix丶覆写了它们，接管了 Fetch 和 Save 的持久层数据操作，实现了透明化的持久层引擎。但我们还是可以在继承下来的业务（集合）类里覆写掉它们，绕开默认的持久层操作。

以 Fetch 业务集合类为例：

```
    /// <summary>
    /// 填充对象
    /// </summary>
    protected override void DataPortal_Fetch(object criteria)
    {
      DbConnectionHelper.Execute(DataSourceKey, DoFetchSelf, criteria);
     }

    /// <summary>
    /// 自定义构建业务对象集合
    /// </summary>
    protected void DoFetchSelf(DbConnection connection, object criteria)
    {
      bool oldRaiseListChangedEvents = RaiseListChangedEvents;
      try
      {
        RaiseListChangedEvents = false;
        SelfFetching = true;
        using (DbCommand command = DbCommandHelper.CreateCommand(connection))
        {
          command.CommandText = SQL语句;
          DbCommandHelper.CreateParameter(command, 参数名, 参数值);
          using (DbDataReader reader = DbCommandHelper.ExecuteReader(command,
CommandBehavior.SingleResult))
          {
            while (reader.Read())
              Add(业务类.FetchSelf(reader));
          }
        }
        SelfFetching = false;
      }
      finally
      {
        RaiseListChangedEvents = oldRaiseListChangedEvents;
       }
    }
```

以上代码可拷贝粘贴到自己的业务集合类里，黄底部分的伪代码可根据需求具体实现。

代码里调用到的业务类 FetchSelf()函数，是需要自行编写的。

FetchSelf()函数传入的参数是 reader 对象，可利用它逐个向业务对象字段赋值（提醒：赋值前判断 reader 的 IsDBNull(i)），可参考如下代码：

```
    /// <summary>
    /// 构建自己
```

```
/// </summary>
internal static T FetchSelf(DbDataReader reader)
{
  T result = DynamicCreateInstance();
  用reader填充result的字段;
  result.MarkFetched();
  return result;
}
```

以上代码可拷贝粘贴到自己的业务类里，黄底部分的伪代码可根据需求具体实现。

1.2.2 调用方法

以上述方法实现的持久层，可通过 Phenix 在业务集合基类中提供的 Fetch 函数被调用到：

```
/// <summary>
/// 构建业务对象集合
/// </summary>
/// <param name="criteria">自定义条件</param>
protected static T Fetch(object criteria)
```

但是上述函数是受保护的，业务集合类不能直接暴露给调用者，需编写一个新的 Fetch 公共函数来封装（调用）它：

```
public new static T Fetch(object criteria)
{
  return Phenix.Business.BusinessListBase<T>.Fetch(criteria);
}
```

调用时，可传入自定义查询类的 criteria 对象（提醒：需打上 Serializable 标签才能被序列化传递到服务端）。建议，为了方便调用方传递正确的查询对象，业务集合类应该编写相应的 Fetch 公共函数，以明确传入具体类型的参数：

```
public static T Fetch(自定义查询类 criteria)
{
   return Phenix.Business.BusinessListBase<T>.Fetch(criteria);
}
```

如果没有自定义查询类，可以编写如下的 Fetch 公共函数：

```
public static T Fetch()
{
   return Phenix.Business.BusinessListBase<T>.Fetch(null);
}
```

另外，如果希望能同时兼容 Phenix丶的持久层引擎，请不要覆写业务集合基类的 DataPortal_Fetch 函数，而应该覆写 DoFetchSelf 函数：

```
/// <summary>
/// 自定义构建业务对象集合
/// </summary>
/// <param name="criteria">自定义条件</param>
protected virtual void DoFetchSelf(object criteria)
```

因为业务集合基类的 DataPortal_Fetch 函数，会先判断传入的 criteria 对象是不是自定义条件对象，如果是的话才会调用 DoFetchSelf 函数，否则会执行默认的持久化操作。

1.3 友情提示

以上内容涉及到 CSLA 的核心功能，但本文并未过多介绍，默认读者已经非常熟悉它们了。如需重温，请复读《C#企业应用开发艺术 CSLA.NET 框架开发实战 Expert C#2008 Business Objects》的 4.2.2“取得对象”章节、18.1“数据访问层设计” 章节等内容。

本文提供的伪码，已经完成了自主设计持久层的主要功能，只要拷贝粘贴到自己的业务（集合）类上，将黄底部分的伪码替换掉即可，这些绝大部分是 ADO.NET 代码。