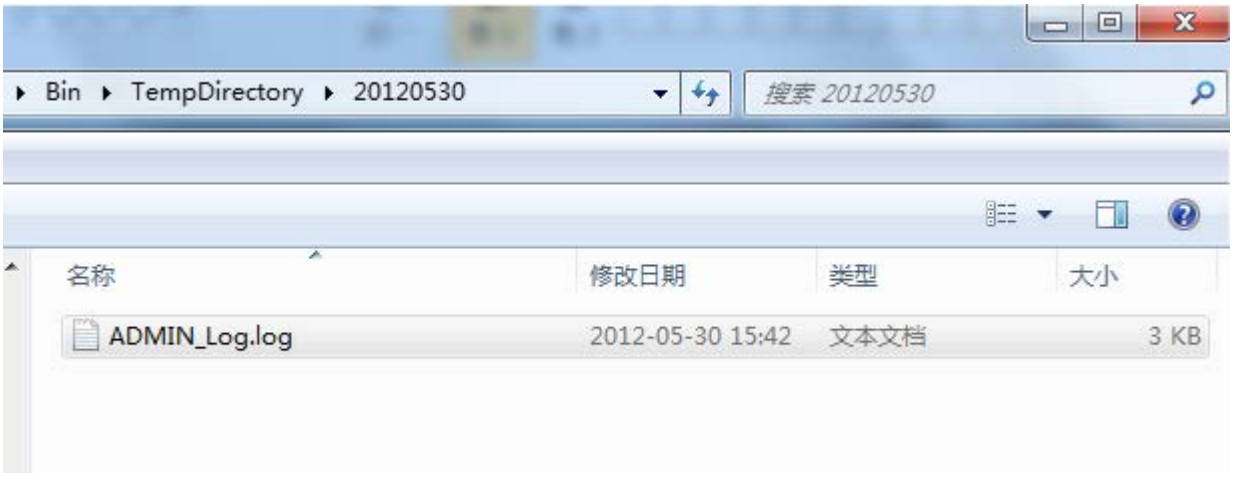
12 业务结构对象模型

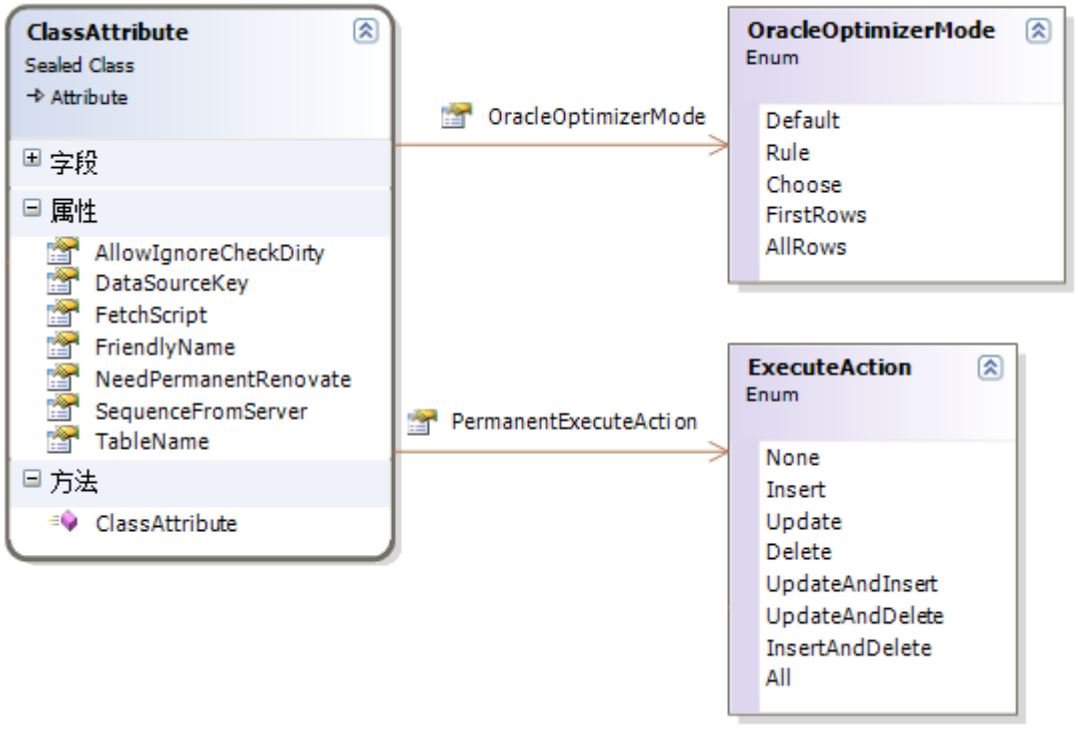
12.6 业务数据的提交及过程

12.6.1Mapping 标签

一般情况下，业务数据的提交都可以通过 Phenix丶持久层引擎自动持久化到数据库中。持久层引擎按照业务类上的 Mapping 标签动态拼装 CRUD 语句。在调试状态（Phenix.Core.Debugging = true）或当前用户为ADMIN权限(Phenix.Business.Security.User.IsAdminRole = true)下，可以在TempDirectory子目录的日志文件中查看到这些 SQL 语句的日志，方便跟踪、查错。



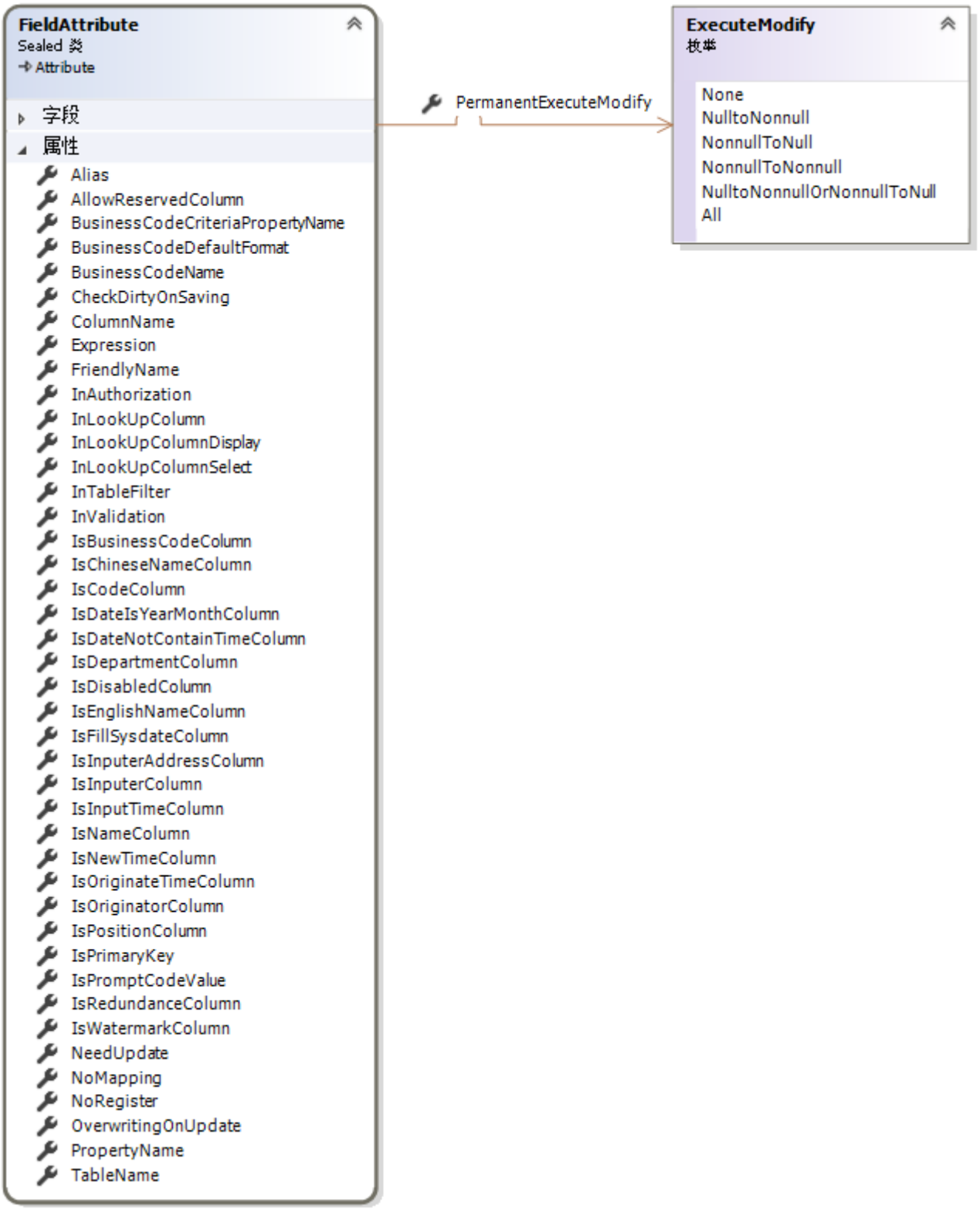
12.6.1.1 ClassAttribute



以下是 ClassAttribute 标签中属性的说明：

属性	说明	备注
DataSourceKey	数据源键	用于分数据库；
TableName	指示该类是映射哪张(主)表	如果关联了多个表，那么本属性指示的是主表，提交对象的增删改结果时将会针对主表进行持久化；
FetchScript	为自动构建业务集合对象提供 select 语句，语句中字段别名在类字段上用 FieldAttribute.Alias 标注	也可以直接提供视图名，视图语句中字段别名在类字段上用 FieldAttribute.Alias 标注，而实际映射的表字段名用 FieldAttribute.ColumnName 标注；
FriendlyName	指示该类的友好名	用于提示信息中；
PermanentExecuteAction	指示当处于哪种执行动作时本字段需要记录新旧值	缺省为 Phenix.Core.Mapping.ExecuteAction.None；当包含 Phenix.Core.Mapping.ExecuteAction.Update 时需结合 FieldAttribute.PermanentExecuteModify 起作用；
NeedPermanentRenovate	指示需要持久化动态刷新	缺省为 false；
SequenceFromServer	是否从服务器获取序号	在 New 业务对象时为 PrimaryKey 字段赋服务器上的唯一值，这在本地处于多进程环境里是更加稳妥的； 缺省为 false，也就是从本地(一个进程有一个)序号生成器获取，性能高；
AllowIgnoreCheckDirty	允许忽略校验数据库数据在下载到提交期间是否被更改过	缺省为 false，一旦发现将抛出异常：CheckSaveException，即禁止关闭乐观锁验证机制；
OralceOptimizerMode	Oracle 优化模式	缺省为 OracleOptimizerMode.Default；

12.6.1.2 FieldAttribute



以下是 FieldAttribute 标签中属性的说明：

属性	说明	备注

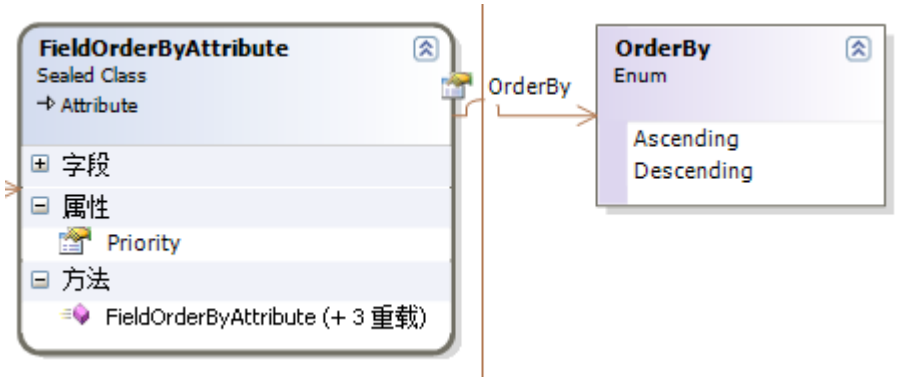
NoMapping	指示该字段不参与映射关系	缺省为 false；主要用于区分在名称相同的情况下的非数据库映射字段；
NoRegister	指示该字段不参与注册	缺省为 false；在注册程序集的时候，配置表 PH_AssemblyClassProperty 中不会被添加上本字段信息；
TableName	指示该字段对应的表名	如果为空，则认为从类的 ClassAttribute.TableName 获取；
ColumnName	指示该字段对应的表列名	如果为空，则认为从字段名获取，获取方法为：字段名第一个字符如果为'_'将被裁剪掉；
Alias	指示该字段对应的别名	如果为空，则认为从 PropertyName 获取；当 ClassAttribute.FetchScript 中存在字段别名时，应该与之对应；
Expression	指示该字段对应的表达式	如果为空，则认为从 Alias 获取；
PropertyName	指示该字段对应的属性名	如果为空，则认为从字段名获取，获取方法为：字段名第一个字符如果为'_'将被裁剪掉，如果是 IsLower 字符则被 ToUpper；
FriendlyName	指示该字段的友好名	用于提示信息中；
IsPrimaryKey	指示该字段是主键	缺省为 false；
IsNewTimeColumn	指示该字段是"新增时间"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultNewTimeColumnName 时必定是新增时间，除非 AllowReservedColumn = false
IsWatermarkColumn	是否水印字段，即仅在 insert 时被提交更新(当所属类表名和字段表名一致时有效)	缺省为 false；IsPrimaryKey 、IsNewTimeColumn 的字段必定是水印字段；
NeedUpdate	指示该字段需要提交更新(当所属类表名和字段表名一致时有效)	缺省为 false；
OverwritingOnUpdate	指示该字段在Update时做覆盖重写(当非水印字段且所属类表名和字段表名一致时有效)	缺省为 false，仅当(对象内)新旧值存在差异时才更新；IsInputerInfoColumn 、IsDisabledColumn 的字段必定做覆盖重写(除

		非 AllowReservedColumn = false）；
AllowReservedColumn	是否允许作为保留字段使用	缺省为 true；，要禁用保留字段必须将 AllowReservedColumn = false
IsBusinessCodeColumn	指示该字段是"业务码"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultBusinessCodeColumnName 时必定是业务码（除非 AllowReservedColumn = false）；
BusinessCodeName	业务码名称	缺省为所属类名.字段名
BusinessCodeDefaultFormat	业务码缺省格式	
BusinessCodeCriteriaPropertyName	业务码条件属性名称	
IsRedundanceColumn	指示该字段是冗余字段	缺省为 false；
IsOriginatorColumn	指示该字段是"制单人"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultOriginatorColumnName 时必定是制单人（除非 AllowReservedColumn = false）；
IsOriginateTimeColumn	指示该字段是"制单时间"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultOriginateTimeColumnName 时必定是制单时间(除非 AllowReservedColumn = false)；
IsInputerColumn	指示该字段是"输入人"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultInputerColumnName 时必定是输入人（除非 AllowReservedColumn = false）；
IsDepartmentColumn	指示该字段是"部门"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultDepartmentColumnName 时必定是部门（除非 AllowReservedColumn = false）；
IsPositionColumn	指示该字段是"岗位"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultPositionColumnName 时必定是岗位（除

		非 AllowReservedColumn = false）；
IsInputTimeColumn	指示该字段是"输入时间"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultInputTimeColumnName 时必定是入时间（除非 AllowReservedColumn = false）；
IsInputerAddressColumn	指示该字段是"输入人地址"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultInputerAddressColumnName 时必定是输入人地址（除非 AllowReservedColumn = false）；
IsDisabledColumn	指示该字段是"禁用"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultDisabledColumnName 时必定是禁用字段（除非 AllowReservedColumn = false）；删除对象时，仅将禁用字段置为 Phenix.Core.Mapping.CodingStandards.DefaultDisabledTrueValue；
IsCodeColumn	指示该字段是"代码"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultCodeColumnName 时必定是代码（除非 AllowReservedColumn = false）；
IsNameColumn	指示该字段是保留字段："名称"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultNameColumnName 时必定是名称（除非 AllowReservedColumn = false）；
IsChineseNameColumn	指示该字段是"中文名"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultChineseNameColumnName 时必定是中文名（除非 AllowReservedColumn = false）；
IsEnglishNameColumn	指示该字段是"英文名"字段	缺省为 false；当 ColumnName 包含

		Phenix.Core.Mapping.CodingStandards.DefaultEnglishNameColumnName 时必定是英文名（除非 AllowReservedColumn = false）；
IsFillSysdateColumn	指示该字段内容应自动填充自系统时间	缺省为 false
IsDateNotContainTimeColumn	指示该字段是"日期(不包含时间部分)"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultDateNotContainTimeColumnName 时必定是日期（不包含时间部分）（除非 AllowReservedColumn = false）；
IsDateIsYearMonthColumn	指示该字段是"年月(当月的头天零时)"字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultDateIsYearMonthColumnName 时必定是年月（当月的头天零时）（除非 AllowReservedColumn = false）；
IsPromptCodeValue	指示该字段是提示码值	缺省为 false；配合 UI 设计期构建下拉框及其配置；
CheckDirtyOnSaving	指示该字段在提交保存时校验数据库数据在下载到保存期间是否被其他进程更改过，一旦发现将报错：CheckDirtyException	缺省为 false；当为 true 时用于乐观锁验证机制；
PermanentExecuteModify	指示当处于哪种执行变更时本字段需要记录新旧值	缺省为 ExecuteModify.All；当 ClassAttribute.PermanentExecuteAction = ExecuteAction.Update 时起作用；
InAuthorization	用于授权	缺省为 true；当为 false 时，关联属性的读写不受权限控制，不受界面 BarManager、ReadWriteAuthorization 组件控制；
InValidation	用于校验	缺省为 true；
InTableFilter	用于表过滤器	缺省为 true；
InLookUpColumn	用于 LookUp 列	缺省为 false；

12.6.1.3 FieldOrderByAttribute



用于 Fetch()业务集合对象时，按照所标识的字段和排序条件拼接 order by 语句，返回队列被排序过的业务集合对象。

示例如下：

```
    /// <summary>
    /// US_USERNUMBER
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<string> UsernumberProperty =
RegisterProperty<string>(c => c.Usernumber);
    //* 指定本字段用于排序
    [Phenix.Core.Mapping.FieldOrderBy(OrderBy.Descending)]
    [Phenix.Core.Mapping.Field(FriendlyName = "US_USERNUMBER", Alias = "US_USERNUMBER", TableName =
"PH_USER", ColumnName = "US_USERNUMBER", NeedUpdate = true)]
    private string _usernumber;
    /// <summary>
    /// US_USERNUMBER
    /// </summary>
    [System.ComponentModel.DisplayName("US_USERNUMBER")]
    public string Usernumber
    {
      get { return GetProperty(UsernumberProperty, _usernumber); }
      set { SetProperty(UsernumberProperty, ref _usernumber, value); }
     }
```

当 Fetch()业务集合对象时，Phenix 的持久层引擎提交到数据库的 select 语句会自动添加如下的排序条件：

```
2016-05-30 14:56:34 <Command Text="select US_ID,US_USERNUMBER,US_NAME from PH_USER order by US_USERNUMBER
DESC " Timeout="30" />
```

如果 Fetch()时带入了动态排序条件的参数，比如以下语句：

```
users = UserList.Fetch(new[] {OrderByInfo.Ascending(User.NameProperty)});
```

则 FieldOrderByAttribute 标签所提供的排序条件的优先级将被置后，结果如下：

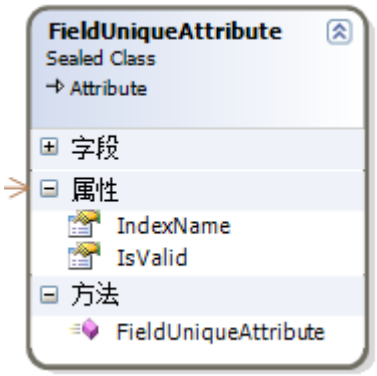
```
2016-05-30 14:56:35 <Command Text="select US_ID,US_USERNUMBER,US_NAME from PH_USER order by US_NAME ASC ,US_USERNUMBER DESC " Timeout="30" />
```

以下是 FieldOrderByAttribute 标签中属性的说明：

属性	说明	备注
OrderBy	排序方式	指示该字段参与排序的方式；
Priority	优先级	指示该字段参与排序的次序；

Fetch()获得的业务集合对象，如有 Add()新的业务对象，Phenix丶会自动按照上述排序条件将 item 置放到适当的 Index 位置，无需在代码里指定，即使指定也无效。但是，集合里业务对象的那些作为排序条件被纳入到排序管理的属性，它们的值如有被修改的话，Phenix丶是不会自动调整所属业务对象的 Index 位置的，除非将这个业务对象 Remove()后再重新 Add()进来。

12.6.1.4 FieldUniqueAttribute



用于提交更新时，判断是否存在重复数据，提升应用系统的友好性。

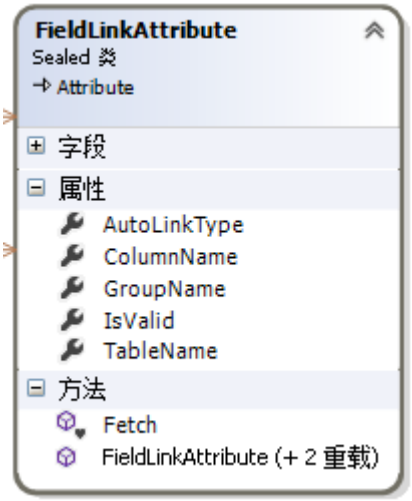
已经在数据库中建立的索引，不必显式声明，除非有意在类中将该索引约束条件置为失效（IsValid = false）状态。

以下是 FieldUniqueAttribute 标签中属性的说明：

属性	说明	备注
IndexName	索引名	应与数据库索引名保持一致；

IsValid	是否有效	缺省为 true；

12.6.1.5 FieldLinkAttribute



用于：

- 自动勾连主从业务对象的结构关系；
- Fetch 从业务对象时提供勾连关系；
- 为从业务对象初始化时，自动填充外键值；

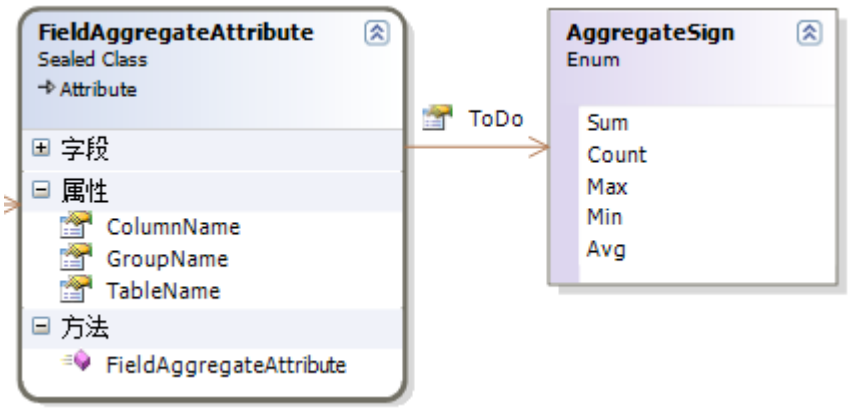
已经在数据库中建立的外键不必显式声明，除非：

- 有意在类中将该外键结构关系置为失效（IsValid = false）状态；
- 需自动关联类，即为 AutoLinkType 属性赋值；

以下是 FieldLinkAttribute 标签中属性的说明：

属性	说明	备注
AutoLinkType	自动关联类	必须所在类是映射视图，且关联类是本类的主业务类，框架会将主业务数据随本业务对象一起提交数据库；
TableName	关联表的表名	
ColumnName	关联表的表列名	
GroupName	分组名	用于区分存在多组外键关联表的情况；如果不赋值，当检索时指定分组名的话则匹配字段对应的属性名；
IsValid	是否有效	缺省为 true；

12.6.1.6 FieldAggregateAttribute



用于主从业务对象结构下，主业务对象聚合（Aggregate）字段的自动赋值，可在从业务类中申明需自动被聚合的字段：

- 通过从业务（集合）类提供的 ComputeMasterAggregate()函数，自动聚合所有从业务对象的被聚合字段值，为本标签所申明指向的主业务对象的聚合字段赋值；
- 在提交主从业务对象结构到数据库时，自动拼装聚合语句并提交到数据库，实现自动为主表聚合字段赋值的功能；

要能应用到本功能，主从业务类所映射的主从表必须由外键显式勾连，或者用 FieldLinkAttribute 在从业务类相关字段上显式申明它们的勾连关系。

以下是 FieldAggregateAttribute 标签中属性的说明：

属性	说明	备注
TableName	关联表的表名	
ColumnName	关联表的聚合表列名	
GroupName	分组名	用于区分存在多组外键关联表的情况；如果不赋值，当检索时指定分组名的话则匹配字段对应的属性名；
ToDo	要做...	缺省为 Sum；
ToDoOnUpdate	指示该字段在 Update 时要为主表做..	缺省为 true

FieldAggregateAttribute.ToDo 属性可设置的聚合功能如下：

```
/// <summary>
/// 聚合符号
```

```
/// </summary>
[KeyCaption(FriendlyName = "聚合符号"), Serializable]
public enum AggregateSign
{
    /// <summary>
    /// 合计
    /// </summary>
    [EnumCaption("合计", Key = "Sum")]
    Sum,

    /// <summary>
    /// 计数
    /// </summary>
    [EnumCaption("计数", Key = "Count")]
    Count,

    /// <summary>
    /// 最大值
    /// </summary>
    [EnumCaption("最大值", Key = "Max")]
    Max,

    /// <summary>
    /// 最小值
    /// </summary>
    [EnumCaption("最小值", Key = "Min")]
    Min,

    /// <summary>
    /// 平均值
    /// </summary>
    [EnumCaption("平均值", Key = "Avg")]
    Avg,
}
```

Phenix 在从业务（集合）类提供了如下的 ComputeMasterAggregate()函数：

```
/// <summary>
/// 计算聚合字段
/// </summary>
public void ComputeMasterAggregate()

/// <summary>
/// 计算聚合字段
```

```
    /// </summary>
    /// <param name="property">属性信息</param>
    public void ComputeMasterAggregate(IPropertyInfo property)


    /// <summary>
    /// 计算聚合字段
    /// </summary>
    /// <param name="propertyName">属性名</param>
    public void ComputeMasterAggregate(string propertyName)
```

在从业务类上声明被聚合字段的 FieldAggregateAttribute 案例如下：

```
        /// <summary>
        /// 出纳台账
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<long?> COD_COH_IDProperty =
RegisterProperty<long?>(c => c.COD_COH_ID);
        [Phenix.Core.Mapping.Field(FriendlyName = "出纳台账", TableName = "Cashier_OrderDetail",
ColumnName = "COD_COH_ID ", NeedUpdate = true)]
        [Phenix.Core.Mapping.FieldLink("Cashier_OrderHead", " COH_ID")]
        private long? _COD_COH_ID;
        /// <summary>
        /// 出纳台账
        /// </summary>
        [System.ComponentModel.DisplayName("COD_COH_ID")]
        public long? COD_COH_ID
        {
            get { return GetProperty(COD_COH_IDProperty, _COD_COH_ID); }
            set { SetProperty(COD_COH_IDProperty, ref _COD_COH_ID, value); }
        }


        /// <summary>
        /// 明细金额
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<decimal?> PriceProperty =
RegisterProperty<decimal?>(c => c.Price, () => 0);
        [Phenix.Core.Mapping.Field(FriendlyName = "明细金额", TableName = "Cashier_OrderDetail",
ColumnName = "COD_Price", NeedUpdate = true)]
        [Phenix.Core.Mapping.FieldAggregateAttribute("Cashier_OrderHead", "COH_Sum_Price")]
        private decimal? _price;
        /// <summary>
        /// 明细金额
        /// </summary>
        [System.ComponentModel.DisplayName("明细金额")]
```

```
        public decimal? Price
        {
          get { return GetProperty(PriceProperty, _price); }
          set { SetProperty(PriceProperty, ref _price, value); }
        }
```

如果需要在提交数据库之前，动态刷新主业务类的聚合字段值，以保证界面的实时动态显示的话，可在被聚合字段的属性 set 语句中添加 ComputeMasterAggregate()函数的调用：

```
        /// <summary>
        /// 明细金额
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<decimal?> PriceProperty =
RegisterProperty<decimal?>(c => c.Price, () => 0);
        [Phenix.Core.Mapping.Field(FriendlyName = "明细金额", TableName = "Cashier_OrderDetail",
ColumnName = "COD_Price", NeedUpdate = true)]
        [Phenix.Core.Mapping.FieldAggregateAttribute("Cashier_OrderHead", "COH_Sum_Price")]
        private decimal? _price;
        /// <summary>
        /// 明细金额
        /// </summary>
        [System.ComponentModel.DisplayName("明细金额")]
        public decimal? Price
        {
          get { return GetProperty(PriceProperty, _price); }
          set
          {
            SetProperty(PriceProperty, ref _price, value);
            ComputeMasterAggregate(PriceProperty);
          }
        }
```

注意，在删除从业务对象的时候，仍需调用它，以保证主业务对象聚合字段值的准确，建议重载从业务类的 RemoveItem()函数：

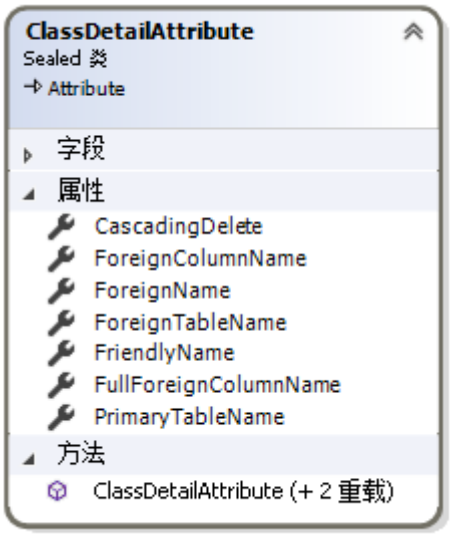
```
        /// <summary>
        /// 移除指定索引处的项
        /// </summary>
        /// <param name="index">索引</param>
        protected override void RemoveItem(int index)
        {
          base.RemoveItem(index);
```

```
            ComputeMasterAggregate(PriceProperty);
        }
```

如果存在批量为被聚合字段的属性赋值的场景，可以在批量赋值之后，再调用从业务集合对象的ComputeMasterAggregate()函数，一次刷新主业务对象的聚合字段值，以保证系统性能。

另外，使用 ComputeMasterAggregate()函数，需保证主业务对象通过 GetDetail()方式获取的从业务对象的集合是个全集，否则算出的值是不正确的。但不管如何，提交到数据库中的值都是正确的。

12.6.1.7 ClassDetailAttribute



用于：

- 在删除（级联删除/Unlink）子表时的友好性提示；
- 申明要求自动删除（级联删除/Unlink）子表记录的功能；

如果不显式声明 ClassDetailAttribute 标签，也没有申明 GetDetail 定义（或申明了但在运行中又未调用到的话），在提交 Root 业务对象时，Phenix丶是不会采取任何动作的。 反之，如果显式声明 ClassDetailAttribute 标签，并将 CascadingDelete = true，被删除（IsDeleted = true）的业务对象在提交时会自动级联删除其子表记录（不管在数据库中有没有物理外键）；如果将 CascadingDelete = false，被删除（IsDeleted = true）的业务对象在提交时会自动 Unlink 其子表记录。

以下是 ClassDetailAttribute 标签中属性的说明：

属性	说明	备注
ForeignTableName	外键表名	
ForeignColumnName	外键字段	

ForeignName	外键名	
PrimaryTableName	主键表名	
FriendlyName	指示该关联的友好名	用于提示信息中；
CascadingDelete	是否级联删除	级联删除（CascadingDelete = true）或 Unlink（CascadingDelete = false）子表记录

12.6.2干预业务数据的提交

12.6.2.1 将完整的业务数据提交到服务端

缺省情况下，在跨物理域提交业务结构之前，Phenix丶会过滤掉非脏数据（IsDirty = false），以提高传输效率。但是，有时候在设计上，希望将一些非脏数据也传递到服务端，供服务端的代码使用，这可以通过覆写业务（/集合）对象的 EnsembleOnSaving 属性来实现：

```
/// <summary>
/// 保留非脏数据以提交到服务端的代码使用
/// 缺省为 false
/// </summary>
protected override bool EnsembleOnSaving
{
  get { return true; }
}
```

当业务结构中某个业务对象的 EnsembleOnSaving = true 时，这个对象及其 Parent 关系链、GetDetail()关系链、GetLink()关系链上的业务对象，都将被传递到服务端；如果是业务集合对象的 EnsembleOnSaving = true，则其全部的 Item 及其 Parent 关系链、GetDetail()关系链、GetLink()关系链上的业务对象，都被传递到服务端。

12.6.2.2 将服务端的业务数据回传到客户端

缺省情况下，如果在服务端上未对业务数据进行编辑的话，是不会被回传回来的。但是，刻意要将它们回传回来的话，也是可以的，通过覆写业务（/集合）对象的 NeedRefreshSelf 属性来实现：

```
/// <summary>
/// 需要刷新自己
/// </summary>
protected override bool NeedRefreshSelf
{
  get { return true; }
```

```
}
```

在业务结构中，只要有一个业务（/集合）对象的 NeedRefreshSelf = true，则整个业务结构都将被完整回传，并用这些回传数据刷新本地标记了 NeedRefreshSelf = true 的业务对象，及其 GetDetail()关系链、GetLink()关系链上的业务对象。

12.6.2.3 在一个事务中捆绑提交不相干的业务数据

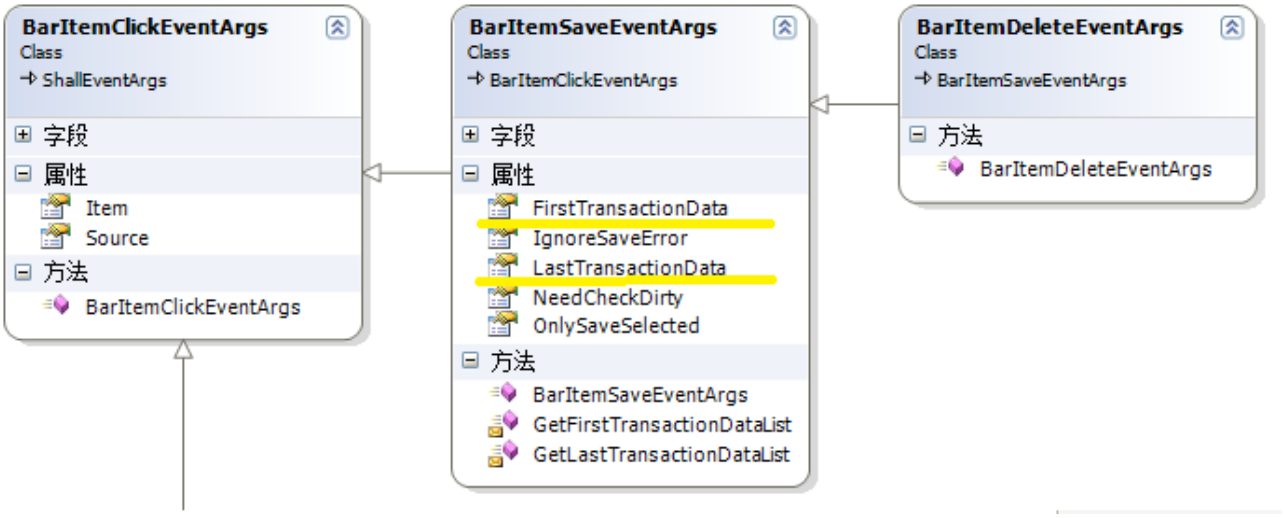
当 Root 对象被执行 Save()函数的时候，可以申明需要顺带一起提交的业务数据：

```
/// <summary>
/// 保存
/// </summary>
/// <param name="firstTransactionData">参与事务处理前端的业务队列</param>
/// <param name="lastTransactionData">参与事务处理末端的业务队列</param>
/// <returns>成功提交的业务对象</returns>
public T Save(IBusiness[] firstTransactionData, IBusiness[] lastTransactionData)

/// <summary>
/// 保存
/// 调用时运行在持久层的程序域里
/// </summary>
/// <param name="transaction">数据库事务，如果为空则将重启新事务</param>
/// <param name="firstTransactionData">参与事务处理前端的业务队列</param>
/// <param name="lastTransactionData">参与事务处理末端的业务队列</param>
/// <returns>成功提交的业务对象</returns>
public T Save(DbTransaction transaction, IBusiness[] firstTransactionData, IBusiness[]
lastTransactionData)
```

更多类似的函数见“11.业务对象生命周期及其状态”的“Save 业务对象”章节。

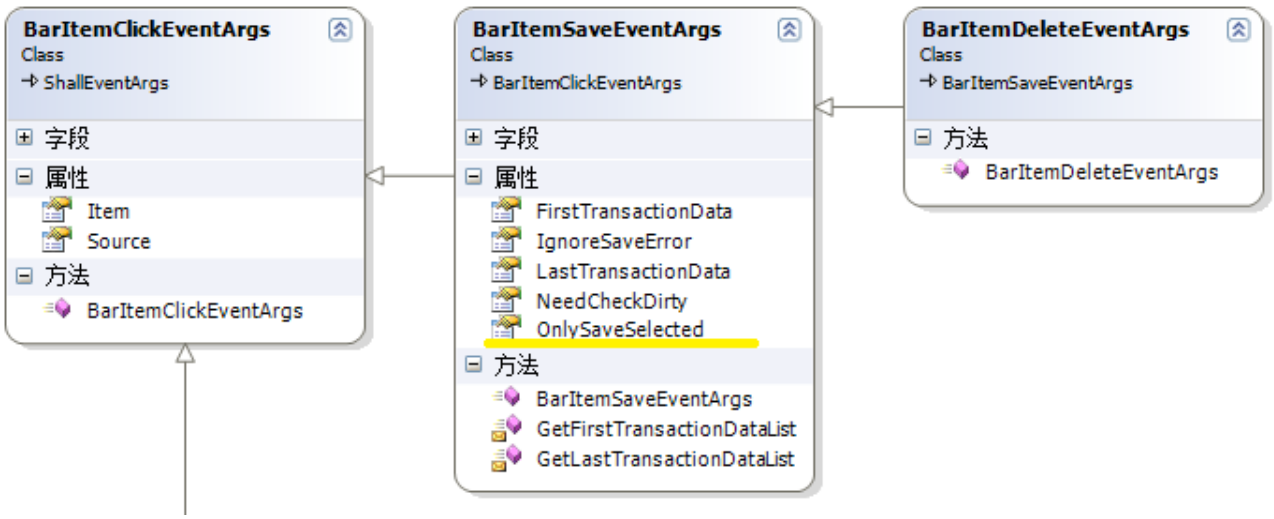
BarManager 组件在保存事件（Saving、Saved）、删除事件（Deleting、Deleted、DeleteCanceled）中提供了调用“提交数据”函数注入参数值的方法（见“10.BarManager 组件”的“ToolBar 事件”章节）：



另外，也可以直接将这些业务数据添加到被提交（调用 Save()函数的）对象的 FirstTransactionData、LastTransactionData 属性队列中。但请注意，在添加之前，应该先清理掉（调用 Clear()函数）以往可能被添加进去的数据。

12.6.2.4 仅提交被选中的业务对象

Phenix 提供了业务对象的可被勾选能力（见前文），一个具体应用就是，用户对业务数据清单进行勾选，然后仅提交这些被选中的业务对象，比如批量删除等操作。那么，接下来我们可以通过 BarManager 组件在保存事件(Saving、Saved)、删除事件(Deleting、Deleted、DeleteCanceled)中，为调用业务集合对象的“提交数据”函数注入参数值（见“10.BarManager 组件”的“ToolBar 事件”章节）的方法来实现：



也可以直接调用 Phenix.Business.BusinessListBase<T, TBusiness>提供的函数，参数 onlySaveSelected = true，这适用于在业务逻辑层中直接操作业务对象的应用场景：

```
    /// <summary>
    /// 保存
    /// </summary>
    /// <param name="needCheckDirty">校验数据库数据在下载到提交期间是否被更改过，一旦发现将报错:CheckDirtyException</param>
    /// <param name="onlySaveSelected">仅提交被勾选的业务对象</param>
    /// <param name="firstTransactionData">参与事务处理前端的业务队列</param>
    /// <param name="lastTransactionData">参与事务处理末端的业务队列</param>
    /// <returns>成功提交的业务对象集合</returns>
    public T Save(bool needCheckDirty, bool? onlySaveSelected, IBusiness[] firstTransactionData, IBusiness[] lastTransactionData)
```

12.6.2.5 禁止持久化脏数据

业务对象虽然在操作过程中被修改了（IsSelfDirty = true）或被删除了（IsDeleted = true），但并不希望被持久化。

12.6.2.5.1 禁止持久化单个业务对象

如果是单个对象的话，可以调用 Phenix.Business.BusinessBase<T>的函数：

```
    /// <summary>
    /// 标为 IsNew = false & IsSelfDirty = false
    /// </summary>
    protected override void MarkOld()
```

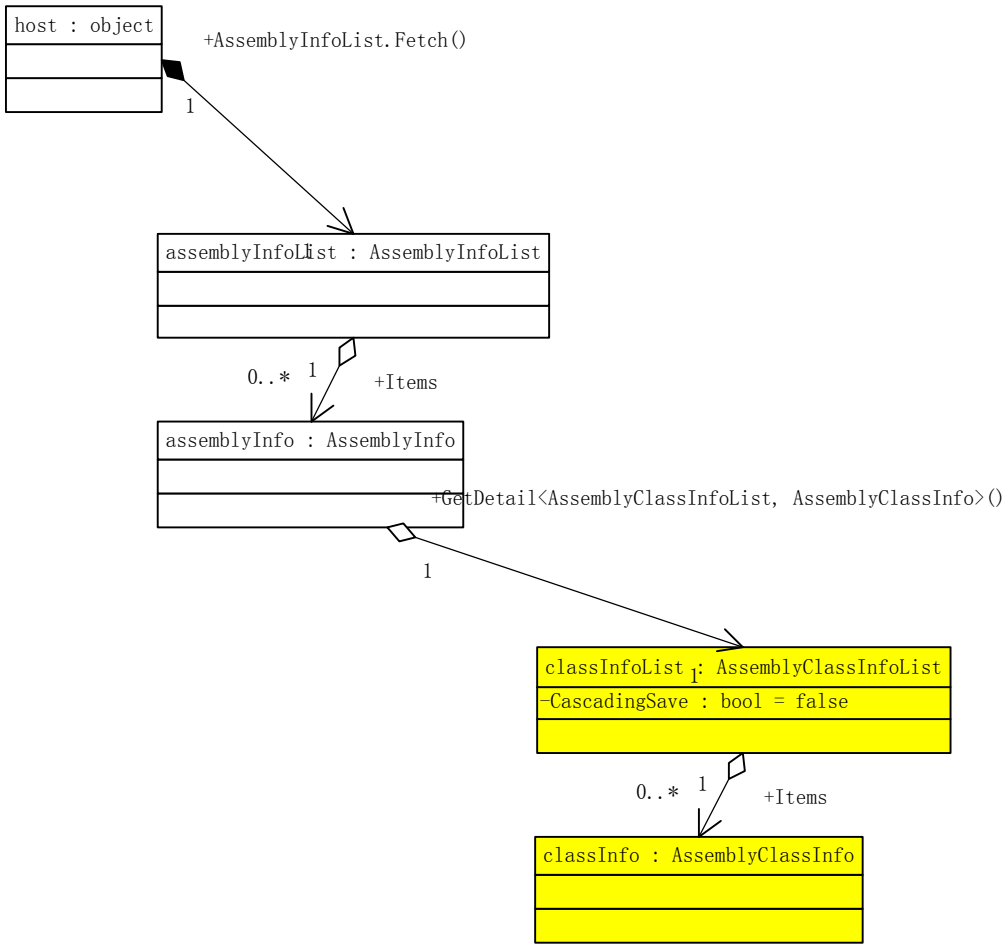
12.6.2.5.2 禁止级联保存从业务对象

如果希望从业务对象集合在其业务结构里不被 Root 对象捆绑一起持久化到数据库，我们可以在获取从业务集合对象的时候，通过传入参数 cascadingSave = false 来实现：

```
    /// 取从业务对象集合
    /// 条件类的字段映射关系请用Phenix.Core.Mapping.CriteriaFieldAttribute标注
    /// </summary>
    /// <param name="criteria">从业务条件对象</param>
    /// <param name="groupName">分组名</param>
    /// <param name="cascadingSave">是否级联保存?</param>
    /// <param name="cascadingDelete">是否级联删除?</param>
    /// <param name="lazyFetch">是否惰性Fetch(</param>
    /// <param name="orderByInfos">数据排列顺序队列</param>
    public TDetail GetDetail<TDetail, TDetailBusiness>(ICriteria criteria, string groupName, bool cascadingSave, bool cascadingDelete, bool lazyFetch, params OrderByInfo[] orderByInfos)
      where TDetail : BusinessListBase<TDetail, TDetailBusiness>
      where TDetailBusiness : BusinessBase<TDetailBusiness>
```

```
    /// </summary>
    /// <param name="criteriaExpression">从业务条件表达式</param>
    /// <param name="groupName">分组名</param>
    /// <param name="cascadingSave">是否级联保存?</param>
    /// <param name="cascadingDelete">是否级联删除?</param>
    /// <param name="lazyFetch">是否惰性Fetch(</param>
    /// <param name="orderByInfos">数据排列顺序队列</param>
    public TDetail GetDetail<TDetail, TDetailBusiness>(CriteriaExpression criteriaExpression, string
groupName, bool cascadingSave, bool cascadingDelete, bool lazyFetch, params OrderByInfo[] orderByInfos)
      where TDetail : BusinessListBase<TDetail, TDetailBusiness>
      where TDetailBusiness : BusinessBase<TDetailBusiness>
```

这样，下图示例中 classInfoList 里的业务对象都不会被持久化：



要知道本业务集合对象是否允许被级联保存，可通过 Phenix.Business.BusinessListBase<T, TBusiness>提供的属性感知：

属性	说明	备注

CascadingSave	是否级联 Save?

12.6.2.6 CompositionDetail/AggregationDetail

业务类提供 GetCompositionDetail()和 GetAggregationDetail()两套方法以获取其从业务对象，分别对应到业务结构的组合关系和聚合关系。

组合关系和聚合关系，表现在持久化数据处理过程，是有所不同的。就是当删除主表记录时，一个是可级联删除子表记录，一个是断开子表记录的外键（字段赋值为 null，需允许为空）。

业务类提供的 GetDetail()方法,默认下等同于 GetCompositionDetail()方法。如果传入的参数 cascadingDelete = false 时，才等同于 GetAggregationDetail()方法：

```
    /// 取从业务对象集合
    /// 条件类的字段映射关系请用Phenix.Core.Mapping.CriteriaFieldAttribute标注
    /// </summary>
    /// <param name="criteria">从业务条件对象</param>
    /// <param name="groupName">分组名</param>
    /// <param name="cascadingSave">是否级联保存?</param>
    /// <param name="cascadingDelete">是否级联删除?</param>
    /// <param name="lazyFetch">是否惰性Fetch(</param>
    /// <param name="orderByInfos">数据排列顺序队列</param>
    public TDetail GetDetail<TDetail, TDetailBusiness>(ICriteria criteria, string groupName, bool cascadingSave, bool cascadingDelete, bool lazyFetch, params OrderByInfo[] orderByInfos)
      where TDetail : BusinessListBase<TDetail, TDetailBusiness>
      where TDetailBusiness : BusinessBase<TDetailBusiness>


    /// </summary>
    /// <param name="criteriaExpression">从业务条件表达式</param>
    /// <param name="groupName">分组名</param>
    /// <param name="cascadingSave">是否级联保存?</param>
    /// <param name="cascadingDelete">是否级联删除?</param>
    /// <param name="lazyFetch">是否惰性Fetch(</param>
    /// <param name="orderByInfos">数据排列顺序队列</param>
    public TDetail GetDetail<TDetail, TDetailBusiness>(CriteriaExpression criteriaExpression, string groupName, bool cascadingSave, bool cascadingDelete, bool lazyFetch, params OrderByInfo[] orderByInfos)
      where TDetail : BusinessListBase<TDetail, TDetailBusiness>
      where TDetailBusiness : BusinessBase<TDetailBusiness>
```

如果想要知道本业务集合对象在持久化时，是被级联删除（CompositionDetail）还是被断开链接（AggregationDetail），可判断 Phenix.Business.BusinessListBase<T, TBusiness>提供的属性：

属性	说明	备注

CascadingDelete	是否级联 Delete?	CompositionDetail: CascadingDelete = true AggregationDetail: CascadingDelete = false

在此提醒的是，如果希望在有从业务对象的情况下不允许删除主对象的业务规则，应覆写主业务类的 AllowDelete 属性，控制界面上的删除功能按钮：

```
/// <summary>
/// 是否允许删除本对象
/// </summary>
[System.ComponentModel.Browsable(false)]
[System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
public override bool AllowDelete
{
  get
  {
    if (判断有从业务对象)
      return false;
    return base.AllowDelete;
  }
 }
```

12.6.2.7 申明级联删除或 Unlink 子表记录

如果代码设计时，没有 GetDetail()子表记录到业务结构中，默认下是不会被级联删除（CompositionDetail）的（除非数据库上配置了级联删除功能），也不会被被断开链接（AggregationDetail）的。那么，我们可以在主业务类上打上 ClassDetailAttribute 标签，显式设置 CascadingDelete = true/false：

以下案例是被级联删除（CompositionDetail）：

```
/// <summary>
/// 角色
/// </summary>
[Phenix.Core.Mapping.ClassAttribute("PH_ROLE", FriendlyName = "角色"), System.SerializableAttribute(),
System.ComponentModel.DisplayNameAttribute("角色")]
[Phenix.Core.Mapping.ClassDetail("PH_USER_ROLE", "UR_RL_ID", null, CascadingDelete = true,
FriendlyName = "用户角色")]
[Phenix.Core.Mapping.ClassDetail("PH_USER_GRANT_ROLE", "GR_RL_ID", null, CascadingDelete = true,
FriendlyName = "用户可授权角色")]
[Phenix.Core.Mapping.ClassDetail("PH_ASSEMBLYCLASS_ROLE", "AR_RL_ID", null, CascadingDelete = true,
FriendlyName = "类-角色")]
```

```
[Phenix.Core.Mapping.ClassDetail("PH_ASSEMBLYCLASSPROPERTY_ROLE", "AR_RL_ID", null, CascadingDelete = true, FriendlyName = "类属性-角色")]
[Phenix.Core.Mapping.ClassDetail("PH_ASSEMBLYCLASSMETHOD_ROLE", "AR_RL_ID", null, CascadingDelete = true, FriendlyName = "类方法-角色")]
public abstract class Role<T> : Phenix.Business.BusinessBase<T> where T : Role<T>
```

以下案例是被断开链接（AggregationDetail）：

```
/// <summary>
/// 岗位
/// </summary>
[Phenix.Core.Mapping.ClassAttribute("PH_POSITION", FriendlyName = "岗位"), System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("岗位")]
[Phenix.Core.Mapping.ClassDetail("PH_USER", "US_PT_ID", null, CascadingDelete = false, FriendlyName = "用户")]
[Phenix.Core.Mapping.ClassDetail("PH_DEPARTMENT", "DP_PT_ID", null, CascadingDelete = false, FriendlyName = "部门")]
public abstract class Position<T> : Phenix.Business.BusinessBase<T> where T : Position<T>
```

如果有用到 GetDetail()方法，则本配置无效。

12.6.2.8 乐观锁

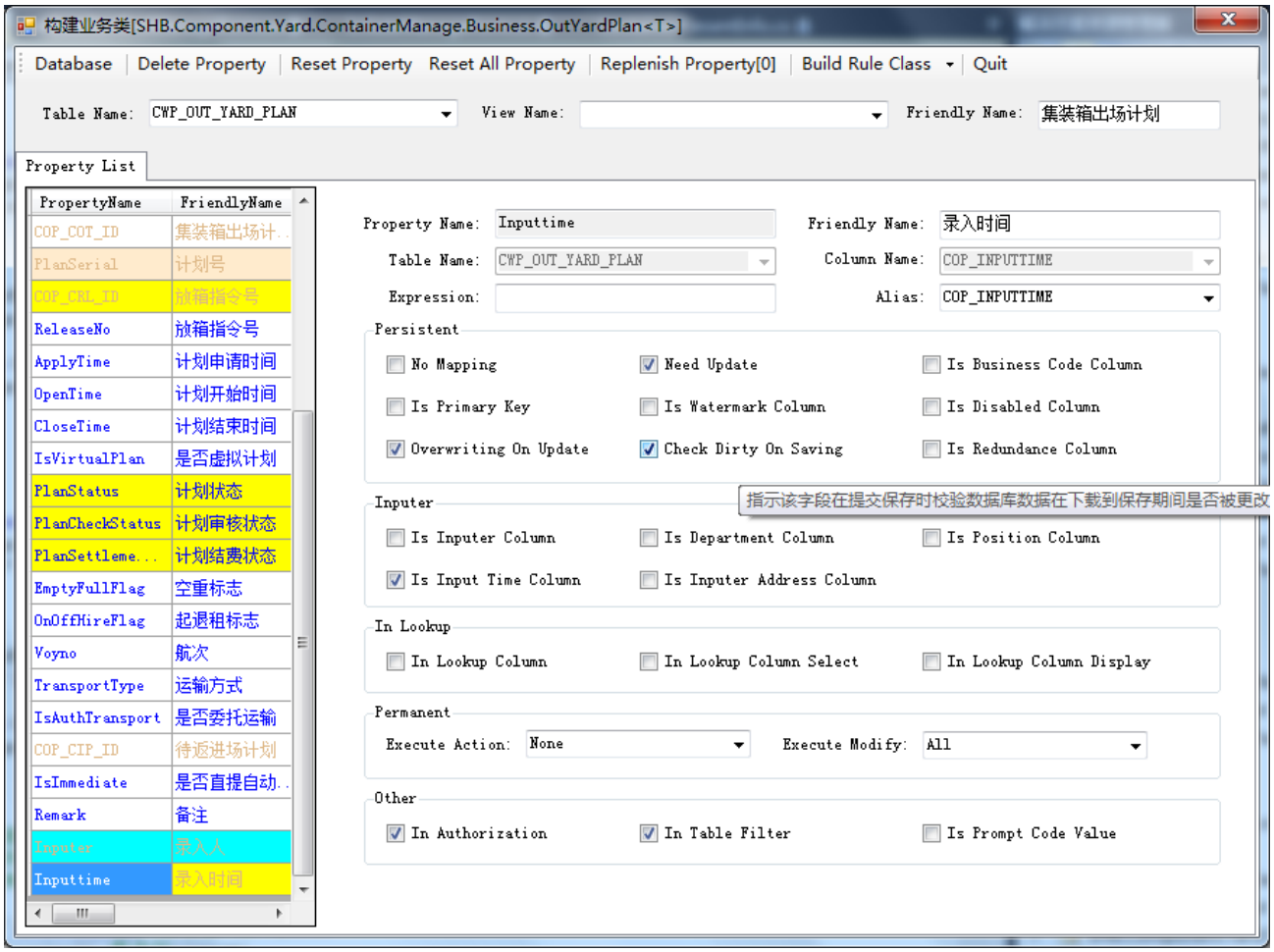
基于分布式多层物理架构下应用系统的设计和开发，也要应对排他性的并发数据处理要求，但直接使用数据库的锁机制（即悲观锁）往往无法适应这样的环境，需变相地实现类似的功能。比如在编辑记录之前，在某个字段上打上占位标记，提交后再消去占位标记的方法。不过，除了这种预先锁定记录的方法外，也可以采取在提交的时候才处理并发问题，这就是乐观锁验证机制。

乐观锁验证机制，就是在数据提交的时候，才判断它在“下载->处理->提交”这前后时间段内对应的数据库数据是否已发生了变化，一旦发现已被其他用户或应用修改了的话，就抛出异常。然后在界面控制层上拦截这个异常，通知本用户选择是否覆盖还是重新下载再处理；必要的话，也可以直接禁止掉强制覆盖的行为。

12.6.2.8.1 乐观锁异常及触发方法

乐观锁异常类型为 Phenix.Core.Mapping.CheckDirtyException：

要对业务对象应用乐观锁验证机制的话，只需选择业务类上适合于做脏数据判断的映射字段（即“乐观锁字段”），在其 FieldAttribute 标签上设定 CheckDirtyOnSaving = true 即可：



本方案需约定所有涉及到本表的应用系统，一旦修改了这个表的记录，都应该改写这条记录上的乐观锁字段。

我们往往选用 Inputer（IsInputerColumn = true）字段、InputTime（IsInputTimeColumn = true）字段作为乐观锁字段组合。因为这两个字段，业务对象在持久化的时候，都会自动将当前用户和当前时间覆写到对应表记录的字段上：

```
        /// <summary>
        /// 录入人
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<string> InputerProperty =
RegisterProperty<string>(c => c.Inputer);
        [Phenix.Core.Mapping.Field(FriendlyName = "录入人", Alias = "COP_INPUTER", TableName =
"CWP_OUT_YARD_PLAN", ColumnName = "COP_INPUTER", NeedUpdate = true, OverwritingOnUpdate = true,
```

```
IsInputerColumn = true, CheckDirtyOnSaving = true)]
        private string _inputer;
        /// <summary>
        /// 录入人
        /// </summary>
        [System.ComponentModel.DisplayName("录入人")]
        public string Inputer
        {
            get { return GetProperty(InputerProperty, _inputer); }
            set { SetProperty(InputerProperty, ref _inputer, value); }
        }


        /// <summary>
        /// 录入时间
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<DateTime?> InputtimeProperty =
RegisterProperty<DateTime?>(c => c.Inputtime);
        [Phenix.Core.Mapping.Field(FriendlyName = "录入时间", Alias = "COP_INPUTTIME", TableName =
"CWP_OUT_YARD_PLAN", ColumnName = "COP_INPUTTIME", NeedUpdate = true, OverwritingOnUpdate = true,
IsInputTimeColumn = true, CheckDirtyOnSaving = true)]
        private DateTime? _inputtime;
        /// <summary>
        /// 录入时间
        /// </summary>
        [System.ComponentModel.DisplayName("录入时间")]
        public DateTime? Inputtime
        {
            get { return GetProperty(InputtimeProperty, _inputtime); }
            set { SetProperty(InputtimeProperty, ref _inputtime, value); }
        }
```

在有些业务场景里，我们仅需跟踪业务对象的状态变化即可，那么，这些状态字段（往往是枚举类型）也可以作为乐观锁字段来使用。

12.6.2.8.2 如何处理乐观锁异常

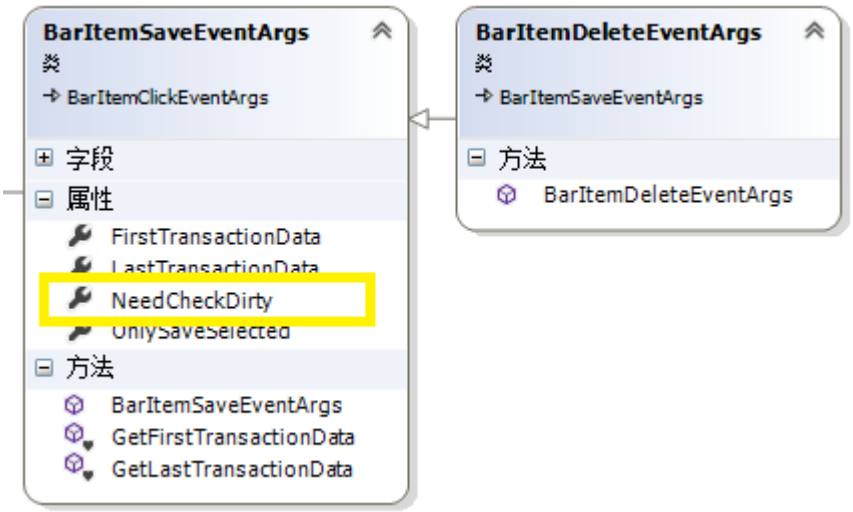
可参考 BarManager 组件，BarManager 组件在界面上对乐观锁验证机制实现了标准的响应及交互处理：

```
    private bool DoSave(bool needPrompt, bool needCheckDirty, bool? onlySaveSelected, IBusiness[]
firstTransactionData, IBusiness[] lastTransactionData)
    {
        while (true)
          try
          {
```

```
            try
            {
              using (DevExpress.Utils.WaitDialogForm waitDialog =
                new DevExpress.Utils.WaitDialogForm(String.Format(Properties.Resources.DataSaving,
CurrentBusinessRoot.Caption,
                  AppConfig.Debugging || UserPrincipal.User == null || UserPrincipal.User.IsAdminRole ?
CurrentBusinessRoot.GetType().Name : null),
                  Phenix.Core.Properties.Resources.PleaseWait))
              {
                CurrentBusinessRoot.Save(needCheckDirty, onlySaveSelected, firstTransactionData,
lastTransactionData);
              }
              string hint = String.Format(Properties.Resources.DataSaveSucceed,
CurrentBusinessRoot.Caption);
              ShowHint(hint);
              if (needPrompt)
                MessageBox.Show(hint, Properties.Resources.DataSave, MessageBoxButtons.OK,
MessageBoxIcon.Information);
              return true;
            }
            catch (Exception ex)
            {
              ExceptionEventArgs e = new ExceptionEventArgs(CurrentBusinessRoot, ex);
              OnSaveFailed(e);
              if (e.Applied)
                return false;
              throw;
            }
          }
          catch (CheckDirtyException ex)
          {
            string hint = String.Format(Properties.Resources.DataSaveForcibly,
              CurrentBusinessRoot.Caption,
              AppConfig.Debugging || UserPrincipal.User == null || UserPrincipal.User.IsAdminRole ?
CurrentBusinessRoot.GetType().FullName : null,
              AppUtilities.GetErrorHint(ex));
            if (MessageBox.Show(hint, Properties.Resources.DataSave, MessageBoxButtons.YesNo,
MessageBoxIcon.Question) != DialogResult.Yes)
              return false;
            needCheckDirty = false;
            continue;
          }
          catch (Exception ex)
          {
            string hint = String.Format(Properties.Resources.DataSaveAborted,
              CurrentBusinessRoot.Caption,
```

```
                AppConfig.Debugging || UserPrincipal.User == null || UserPrincipal.User.IsAdminRole ?
CurrentBusinessRoot.GetType().FullName : null,
                AppUtilities.GetErrorHint(ex, typeof(Csla.DataPortalException),
typeof(Csla.Reflection.CallMethodException)));
                ShowHint(hint);
                MessageBox.Show(hint, Properties.Resources.DataSave, MessageBoxButtons.OK,
MessageBoxIcon.Error);
                LocateInvalidItem(MasterBindingSource ?? BindingSource, false);
                return false;
            }
        }
```

在BarManager组件的保存事件(Saving、Saved)、删除事件(Deleting、Deleted、DeleteCanceled)中提供了调用“提交数据”函数注入参数值的方法（见“10.BarManager 组件”的“ToolBar 事件”章节）：




```
private void barManager_Saving(object sender, Phenix.Windows.BarItemSavingEventArgs e)
{
  e.NeedCheckDirty = false;
}
```

上述代码可以在有条件的（业务类上的 ClassAttribute 标签上设定 AllowIgnoreCheckDirty = false，见下一章节）前提下关闭乐观锁验证机制。

也可以直接调用 Phenix.Business.BusinessBase<T>、Phenix.Business.BusinessListBase<T, TBusiness>提供的函数，参数 needCheckDirty = false：

```
/// <summary>
/// 保存
```

```
    /// </summary>
    /// <param name="needCheckDirty">校验数据库数据在下载到提交期间是否被更改过，一旦发现将报错：CheckDirtyException；如果ClassAttribute.AllowIgnoreCheckDirty = false本功能无效，必定报错：CheckSaveException
    /// <param name="onlySaveSelected">仅提交被勾选的业务对象</param>
    /// <param name="firstTransactionData">参与事务处理前端的业务队列</param>
    /// <param name="lastTransactionData">参与事务处理末端的业务队列</param>
    /// <returns>成功提交的业务对象集合</returns>
    public T Save(bool needCheckDirty, bool? onlySaveSelected, IBusiness[] firstTransactionData, IBusiness[] lastTransactionData)
```

更多类似的函数见“11.业务对象生命周期及其状态”的“Save 业务对象”章节。

12.6.2.8.3 禁止关闭乐观锁验证机制

BarManager 组件上对 CheckDirtyException 的处理方法，及通过调用 Save()函数时传入参数 needCheckDirty = true，虽然都能关闭乐观锁验证机制，但这必须在业务类上的 ClassAttribute 标签上设定 AllowIgnoreCheckDirty = true 的前提下才能起作用，而它缺省情况下是为 false 值的。所以，如无显式设定它为 true 的话，Phenix 会直接抛出带 InnerException 属性值为 CheckDirtyException 异常的 CheckSaveException 异常。

我们可以拦截 CheckSaveException，只要它的异常 InnerException 属性值为 CheckDirtyException 异常，即可判断为发生了禁止关闭乐观锁验证机制的异常。此时我们可以尝试重新获取最新的数据库数据，然后处理后重新提交，实现悲观锁一样的效果。

以下是提交关联表的合计字段时发生了这些字段已被其他用户更新过时的处理方法：

```
        private void DoSave()
        {
          try
          {
            Save();
          }
          catch (CheckSaveException ex)
          {
            //拦截到禁止关闭乐观锁机制的异常
            if (ex.InnerException != null &&
ex.InnerException.GetType().Equals(typeof(CheckDirtyException)))
            {
              //先处理其他异常条件
              if (Fetch((T)this).LadingState == DeliveryWork.Rule.LadingState.Complete)
                throw new CheckSaveException(LadingStateProperty.FriendlyName + "已完成作业，请重新检索.");
              //清除内存，重新提交时会重新计算关联表的合计字段
```

```
                DeliveryPlan = null;
                DoSave();
            }
        }
    }
```

以下代码是在提交过程中如何计算关联表的合计字段：

```
    protected override void OnSavingSelf(System.Data.Common.DbTransaction transaction)
    {
        if (!IsDeleted)
        {
            //出仓计划
            if (DeliveryPlan == null || DeliveryPlan.DLP_ID != DJT_DLP_ID)
            {
                DeliveryPlan = Warehouse.DeliveryPlan.Business.DeliveryPlan.Fetch(p => p.DLP_ID ==
DJT_DLP_ID);
            }
            //当实际件数不为0时，修改提货单状态为发货中
            if (LadingState == DeliveryWork.Rule.LadingState.None
                && DeliveryGoodsList.Any(p => p.ActualCount > 0))
            {
                LadingState = DeliveryWork.Rule.LadingState.Receiving;
            }
            //生成了作业单计划即在执行中
            if (DeliveryPlan.PlanStatus == Base.Rule.PlanStatus.PlanStatus.WaitExecute)
            {
                DeliveryPlan.PlanStatus = Base.Rule.PlanStatus.PlanStatus.Executing;
            }
            #region 同步修改WPL_DELIVERY_PLAN中的预配货位数据及库存量
            if (LadingState == DeliveryWork.Rule.LadingState.Complete)
            {
                //库存货物
                var storeGoodsCache = new Dictionary<long?, Store.Business.StoreGoods>();
                foreach (var deliverPickingGoods in DeliveryPickingGoodsList)
                {
                    //出仓计划货物
                    DeliveryPlanGoods deliveryPlanGoods = null;
                    foreach (var planDetail in DeliveryPlan.DeliveryPlanDetails)
                    {
                        //出仓计划货物
                        deliveryPlanGoods = planDetail.DeliveryPlanGoods.FirstOrDefault(p =>
p.DPG_ID == deliverPickingGoods.DPG_DPG_ID);
                        if (deliveryPlanGoods == null) continue;
                        break;
                    }
```

```
                    #region 修改时同步库存
                    //对出仓计划货物下的预配货位进行比较，如果实际出货货位在计划预配货位中不存在则新增一条计划预配货位
                    foreach (var pickingLocation in deliverPickingGoods.DeliveryPickingLocations)
                    {
                        //库存货物
                        Store.Business.StoreGoods storeGoods = null;
                        if (storeGoodsCache.ContainsKey(deliveryPlanGoods.DPG_SGS_ID))
                        {
                            storeGoods = storeGoodsCache[deliveryPlanGoods.DPG_SGS_ID];
                        }
                        else
                        {
                            storeGoods = Store.Business.StoreGoods.Fetch(transaction,
                                p => p.SGS_ID == deliveryPlanGoods.DPG_SGS_ID);
                            storeGoodsCache.Add(storeGoods.SGS_ID, storeGoods);
                        }

                        var planLocation = deliveryPlanGoods.DeliveryPlanLocations
                            .FirstOrDefault(p => p.LocationNo == pickingLocation.LocationNo);
                        var storeGoodsLocation = storeGoods.StoreGoodsLocations.FirstOrDefault(p => p.LocationNo == pickingLocation.LocationNo);
                        if (storeGoodsLocation == null)
                        {
                            throw new Exception("[" + storeGoods.ProductName + "]没有堆放在货位[" + pickingLocation.LocationNo + "]上");
                        }

                        int? count = pickingLocation.ActualCount;
                        decimal? volume = pickingLocation.ActualVolume;
                        decimal? weight = pickingLocation.ActualWeight;
                        decimal? gaugeCount = pickingLocation.ActualGaugeCount;
                        storeGoodsLocation.Count = (storeGoodsLocation.Count ?? 0) - count;
                        storeGoodsLocation.Weight = (storeGoodsLocation.Weight ?? 0) - weight;
                        storeGoodsLocation.Volume = (storeGoodsLocation.Volume ?? 0) - volume;
                        storeGoodsLocation.GaugeCount = (storeGoodsLocation.GaugeCount ?? 0) - gaugeCount;

                        //实际出货货位在计划预配货位中不存在则新增一条计划预配货位,并将库存预占量增加
                        if (planLocation == null)
                        {
                            planLocation = deliveryPlanGoods.DeliveryPlanLocations.AddNew();
                            planLocation.LocationNo = pickingLocation.LocationNo;
                        }
```

```
                        planLocation.ActualCount = pickingLocation.ActualCount;
                        planLocation.ActualVolume = pickingLocation.ActualVolume;
                        planLocation.ActualWeight = pickingLocation.ActualWeight;
                        planLocation.ActualGaugeCount = pickingLocation.ActualGaugeCount;
                    }
                    #endregion
                }

                foreach (var storeGoods in storeGoodsCache)
                {
                    storeGoods.Value.Save(transaction);
                }
            }
            #endregion

            if (DeliveryPlan != null && DeliveryPlan.IsDirty)
            {
                DeliveryPlan.Save(transaction);
            }
        }
        base.OnSavingSelf(transaction);
    }
```

通过拦截 BarManager 组件的 SaveFailed 事件也可以处理乐观锁异常：

```
private void barManager_SaveFailed(object sender, Phenix.Core.ExceptionEventArgs e)
{
  //拦截到禁止关闭乐观锁机制的异常
  if (e.Error.GetType().Equals(typeof(Phenix.Business.CheckSaveException)) &&
e.Error.InnerException != null &&
e.Error.InnerException.GetType().Equals(typeof(Phenix.Business.CheckDirtyException)))
  {
    //先处理其他异常条件
    if (Fetch((T)this).LadingState == DeliveryWork.Rule.LadingState.Complete)
      throw new CheckSaveException(LadingStateProperty.FriendlyName + "已完成作业，请重新检索.");
    //清除内存，重新提交时会重新计算关联表的合计字段
    DeliveryPlan = null;
    DoSave();
    //已成功处理
    e.Succeed = true;
  }
}
```

12.6.2.8.4 处理并发或脏数据的方法

脏数据有两种不同的类型，并应该有针对性地采取不同的处理方式：

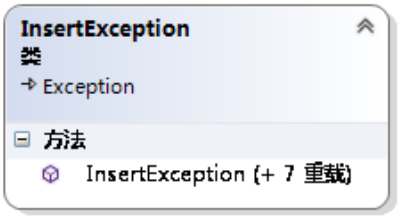
- 类似合计数等计算字段，需要严格保证数据完整性的，请根据前文拦截 CheckSaveException，判断它的 InnerException is CheckDirtyException 来处理，以保证提交成功且是数据完整的，此类问题不应该抛给用户处理；
- 如果是类似两个用户先后修改提交同一计划内容的使用场景，则允许交给用户判断是否覆盖提交，不会造成数据完整性损害的，那么方法是：在业务类上的ClassAttribute标签上设定 AllowIgnoreCheckDirty = true；这样，通过BarManager的Save按钮提交第一次失败时，会提示用户选择是否覆盖，点击确认的话，就用本地数据覆盖数据库的数据。

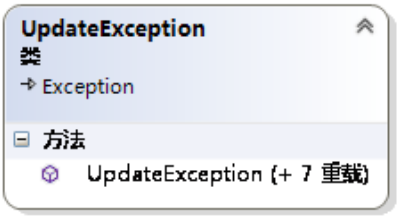
12.6.2.9 在提交过程中抛出的底层异常

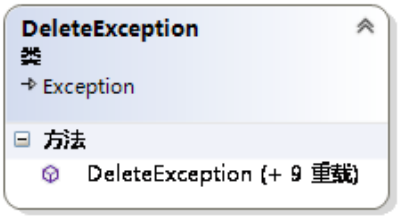
12.6.2.9.1 增删改的底层异常

缺省情况下，只要在提交过程中发生任何异常，都会直接回滚事务，并将这底层异常包装到：

- Phenix.Core.Mapping.InsertException
- Phenix.Core.Mapping.UpdateException
- Phenix.Core.Mapping.DeleteException

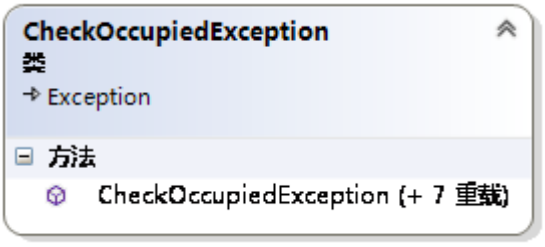






即代入它们的 InnerException 属性里，再次抛出给 Root 对象 Save()函数的调用方。

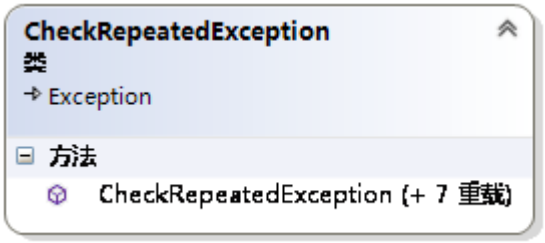
12.6.2.9.2 检测到被占用记录



Phenix.Core.Mapping.CheckOccupiedException 异常，是在删除记录时，因有子表记录仍然挂着而无法删除本记录的时候被抛出的。其实，这个异常完全可以避免，具体做法，请参考前文中有关级联删除的内容。

本异常，会被作为底层异常包装到 DeleteException 中再次抛出。

12.6.2.9.3 检测到重复的记录



Phenix.Core.Mapping.CheckRepeatedException 异常，在增删改的提交过程中都会发生，只要提交的记录与数据库表中其他记录存在相同的唯一键值时就会被抛出。

本异常，会被作为底层异常分别包装到 InsertException、UpdateException、DeleteException 中再次抛出。

12.6.2.10 使用独立事务

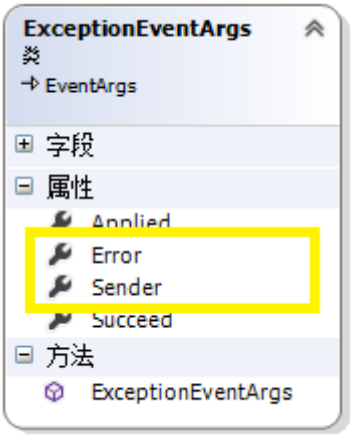
缺省情况下，业务结构中的所有业务对象在提交过程中都将在一个事务里被持久化。但是，在有些应用场景里，集合中的每个业务对象是需要单开自己的事务，在自己的事务里被持久化（其 Detail 业务对象将会被裹挟在 Master 业务对象的这个事务里）。

要实现独立的事务，我们可以通过覆写 Master 业务对象集合类的 AloneTransaction 属性来实现：

```
public class UserList : Phenix.Business.BusinessListBase<UserList, User>
{
  /// <summary>
  /// 是否业务对象各自使用独立事务
  /// 缺省为 false
  /// </summary>
  protected override bool AloneTransaction
  {
    get { return true; }
  }
}
```

这样，在 Master 集合对象的提交过程中（本例为调用 UserList 对象的 Save()函数），未能正常持久化（即未能 Commit 成功）的 Master 业务对象（本例为 User 对象）所抛出的异常，都会被 Phenix 包装在 Phenix.Business.SaveException 异常对象的 SaveErrors 属性里，最终抛给 Master 集合对象 Save()函数的调用方。

Phenix.Business.SaveException 类的 SaveErrors 属性，是个 Phenix.Core.ExceptionEventArgs 对象的队列：



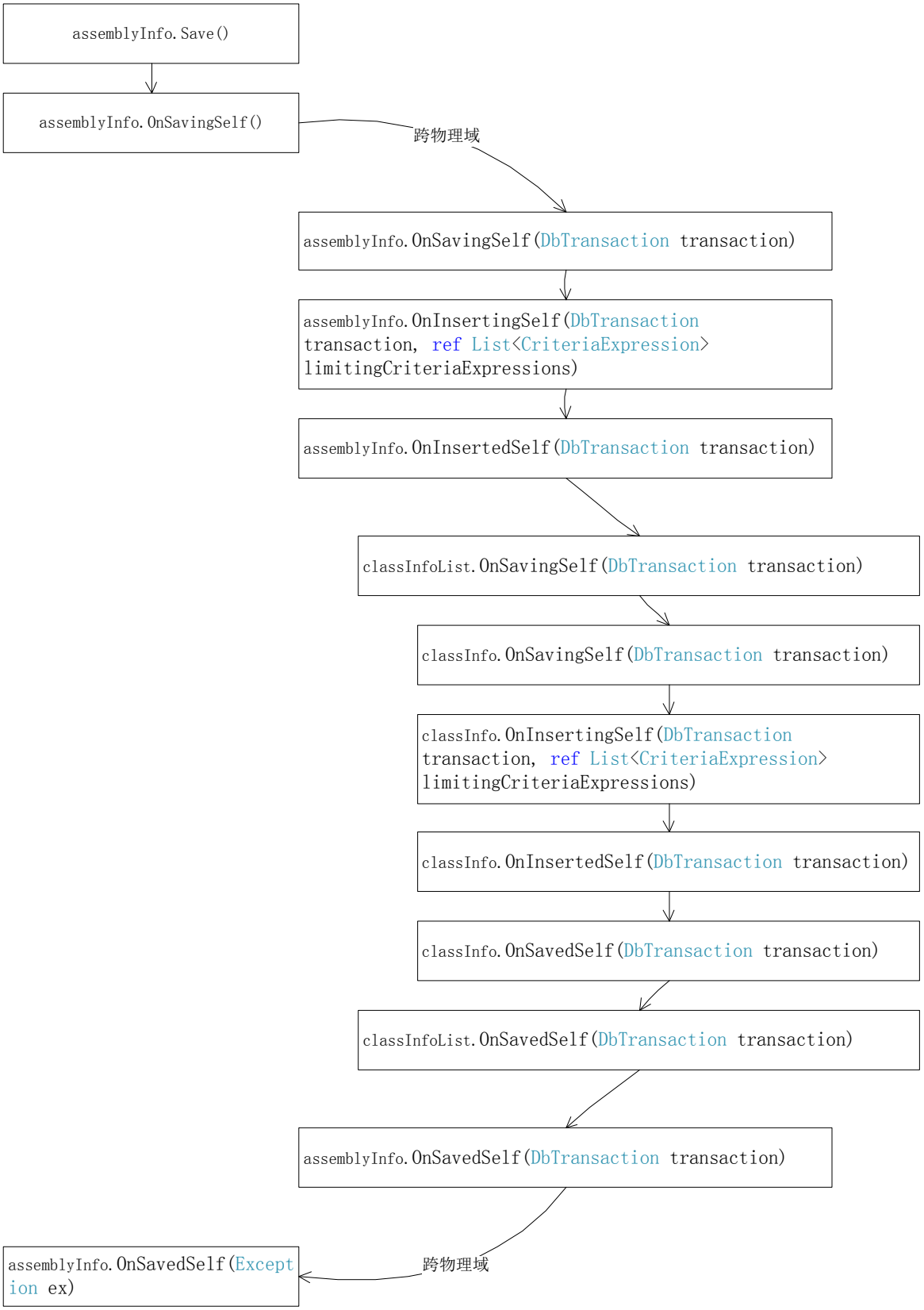
其 Sender 属性就是未能正常持久化的 Master 业务对象，而 Error 属性则是其抛出的的异常对象。这样，我们可以通过拦截 SaveException 异常并遍历其 SaveErrors 属性的队列内容，来获取到具体的异常细节。

12.6.2.11 覆写数据提交过程中的“On”函数

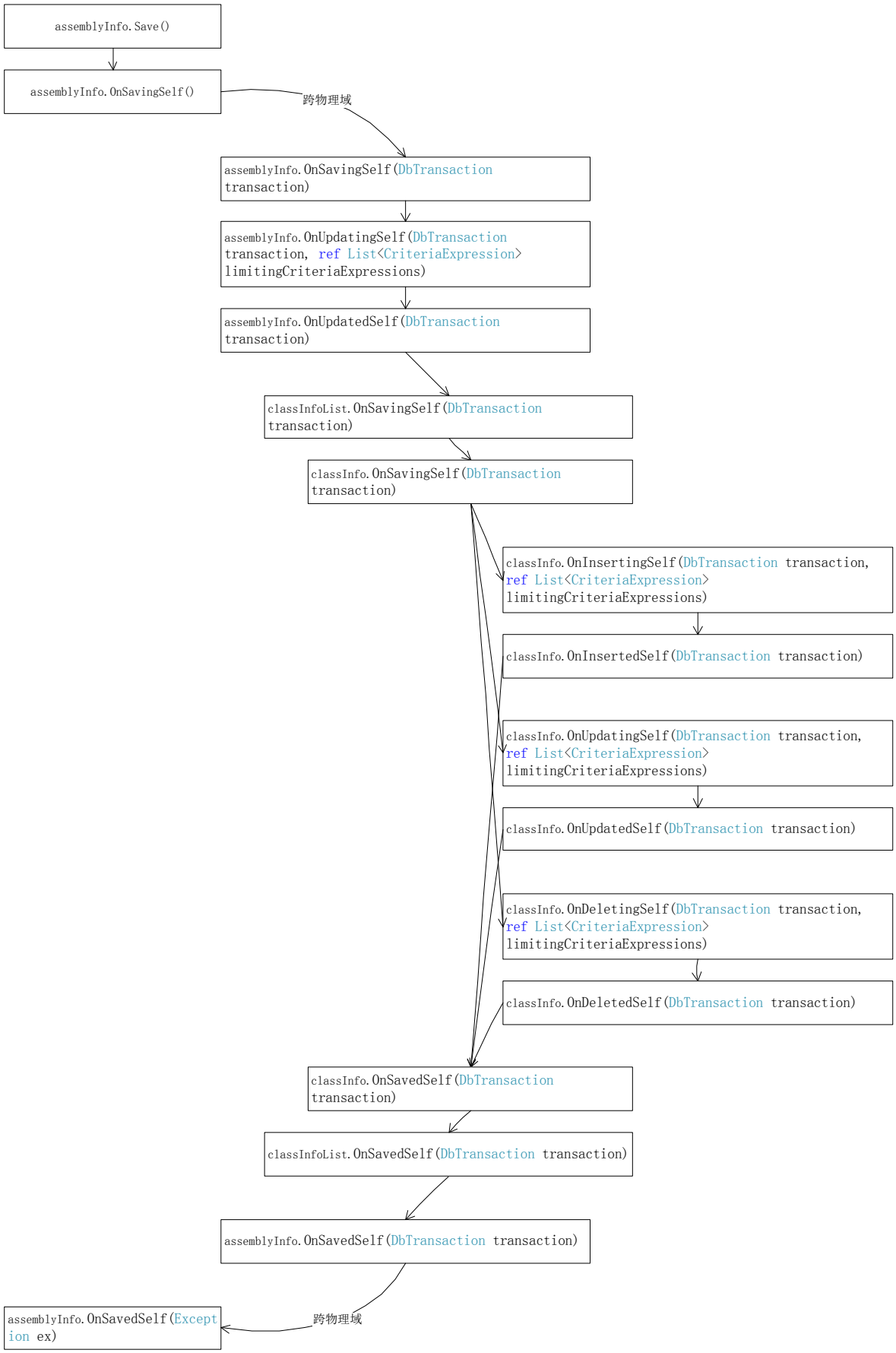
Phenix.Business.BusinessBase<T>和 Phenix.Business.BusinessListBase<T, TBusiness>都提供了调用 Root 对象的 Save()函数时干预业务数据提交过程的 virtual 函数，以“On”为函数名的前缀，我们可按需在这些函数里嵌入自己的业务逻辑代码。

以下图示了在各种业务结构下，这些函数被调用的先后次序，以及被调用的时候，代码所运行的物理域：

Root业务对象数据的Insert过程

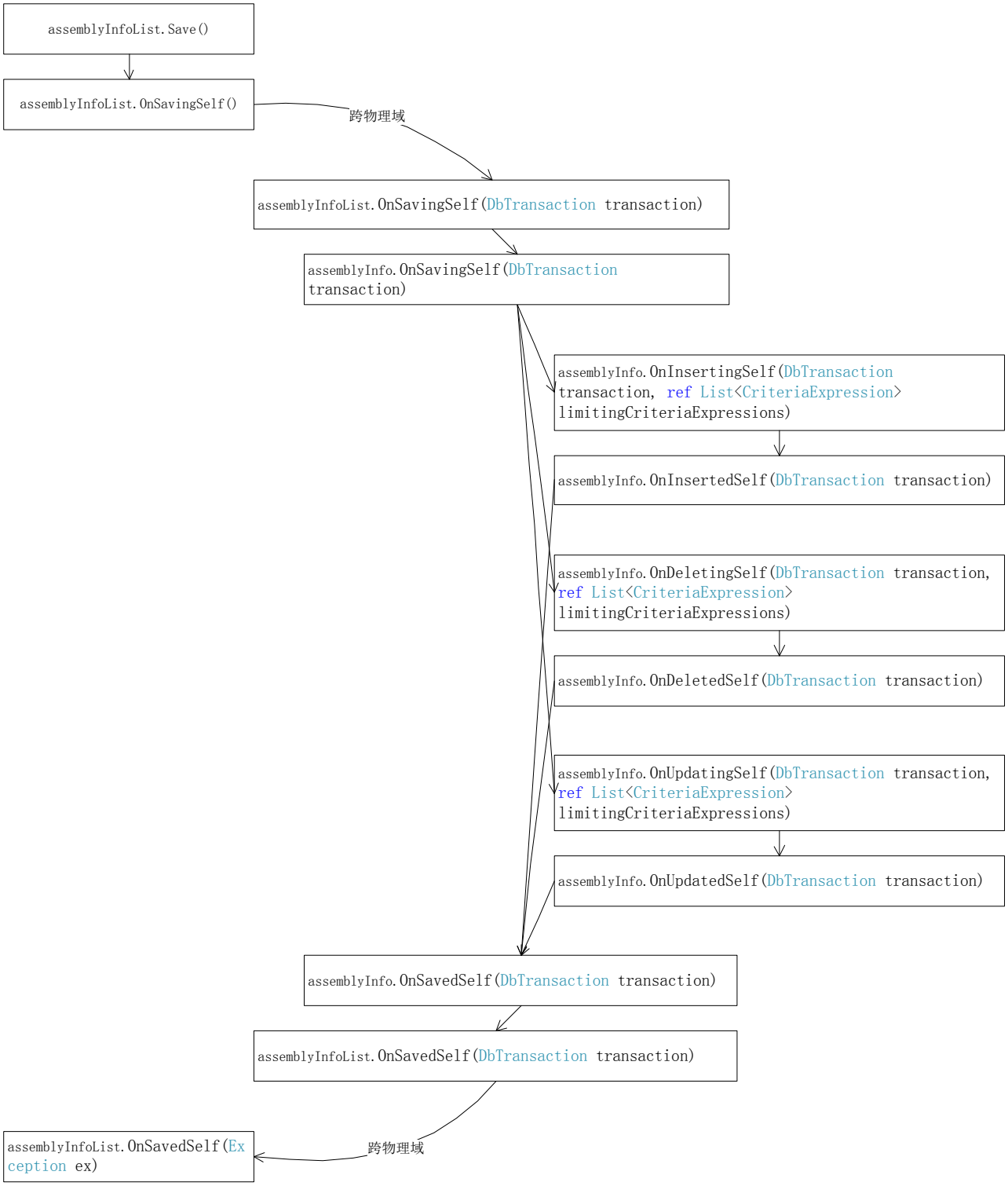


Root业务对象数据的Update过程



Root业务对象数据的Delete过程

Root业务集合对象数据的提交过程



另外，在业务对象 Phenix.Business.BusinessBase<T>级联删除子表记录的时候，我们可通过重载以下函数达到拦截处理的目的：

```
    /// <summary>
    /// 删除本从业务对象数据之前
    /// 在运行持久层的程序域里被调用
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    protected virtual void OnDeletingDetails(DbTransaction transaction)
    {
    }

    /// <summary>
    /// 删除本从业务对象数据之后
    /// 在运行持久层的程序域里被调用
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    protected virtual void OnDeletedDetails(DbTransaction transaction)
    {
    }
```

12.6.2.12 在最后时刻限制保存

上文中调用 Root 对象 Save()函数的提交过程中，都可以在正式持久化数据之前嵌入“限制保存的条件”：

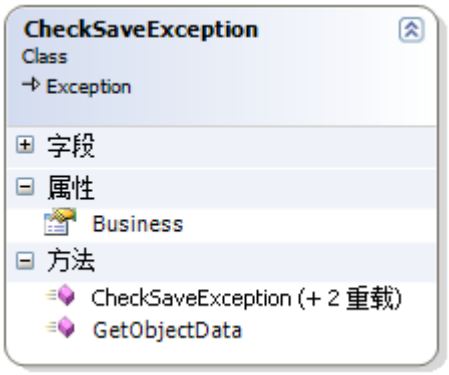
```
    /// <summary>
    /// 新增本对象数据之前
    /// 在运行持久层的程序域里被调用
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    /// <param name="limitingCriteriaExpressions">限制保存的条件(not exists 条件语句)</param>
    protected virtual void OnInsertingSelf(DbTransaction transaction, ref List<CriteriaExpression> limitingCriteriaExpressions)

    /// <summary>
    /// 更新本对象数据之前
    /// 在运行持久层的程序域里被调用
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    /// <param name="limitingCriteriaExpressions">限制保存的条件(not exists 条件语句)</param>
    protected virtual void OnUpdatingSelf(DbTransaction transaction, ref List<CriteriaExpression> limitingCriteriaExpressions)

    /// <summary>
    /// 删除本对象数据之前
```

```
    /// 在运行持久层的程序域里被调用
    /// </summary>
    /// <param name="transaction">数据库事务</param>
    /// <param name="limitingCriteriaExpressions">限制保存的条件(not exists 条件语句)</param>
    protected virtual void OnDeletingSelf(DbTransaction transaction, ref List<CriteriaExpression> limitingCriteriaExpressions)
```

“限制保存的条件”是以条件表达式的方式传递给持久层引擎，由持久层引擎以“not exists”语句拼接到增删改 SQL 语句的 where 条件里。当保存完数据之后，其返回的保存记录数为 0 时，则认为限制条件起了作用，并抛出 Phenix.Business.CheckSaveException 异常，调用方可拦截此异常并做后续处理：



12.6.2.13 在 OnSavedSelf 里处理异常

上文中调用 Root 对象 Save()函数的提交过程中，最后一个可干预的“On”函数都是 OnSavedSelf 函数，在这个函数里可以处理持久化中被抛出的异常：

```
    /// <summary>
    /// 保存业务对象集合之后
    /// 在执行Save()的程序域里被调用
    /// </summary>
    /// <param name="ex">错误信息</param>
    ///<returns>发生错误时的友好提示信息，缺省为 null 直接抛出原异常，否则会抛出SaveException并打包本信息及原异常</returns>
    protected virtual string OnSavedSelf(Exception ex)
    {
      return null;
    }
```

如果覆写这个函数，在代码中返回非 null 字符串的话，这个字符串会被包装到 Phenix.Business.SaveException（即代入它们的 Message 属性里）抛出给 Root 对象的 Save()函数调用方。


SaveException
Class
Exception
方法
SaveException (+ 3 重载)