17 条件检索业务对象

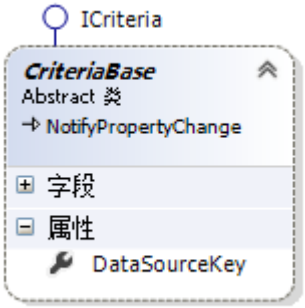
在“11.业务对象生命周期及其状态”的“Fetch 业务对象”章节中，仅罗列了 Fetch 的各种方法函数，本文将重点讨论这些 Fetch()函数中可提供的检索条件的方式。

17.1 ICriteria

条件接口 Phenix.Core.Mapping.ICriteria，仅含一个指定数据源键（用于指定 Fetch()的是哪个数据源，需要与数据库连接配置项 DataSourceKey 值保持一致）的属性之外别无其他属性，标识了当前的对象是条件对象：

17.1.1条件类

在应用系统的设计当中，具体使用的是 Phenix.Business.CriteriaBase，它实现了条件接口 ICriteria，只要继承了它，可被 Phenix 识别为条件对象：



Phenix.Business.CriteriaBase 的作用，主要是可以作为数据源对象与界面控件进行绑定，实现在界面上检索条件的手工输入，从而达到快速开发界面的目的。但这同时也衍生出一个问题，就是在业务逻辑层中为这些条件对象的条件属性赋值时，如何即刻反映到界面上？我们知道，从逻辑分层上，上一层调用下一层的服务，而下一层则以事件（/或推送消息）的方式影响到上一层。也就是说，在业务逻辑层上的条件对象，应该可以做到当条件属性发生变更时，能即时通知界面控制层的 BindingSource 组件刷新界面上绑定控件的数据显示。为此，它实现了 System.ComponentModel.INotifyPropertyChanged 接口，提供了 PropertyHasChanged()函数，可在属性的 set 语句中被调用，触发条件对象上的 PropertyChanged 事件，因为 BindingSource 组件是可以感知到这个事件的。

以下案例演示了如何调用 PropertyHasChanged()函数：

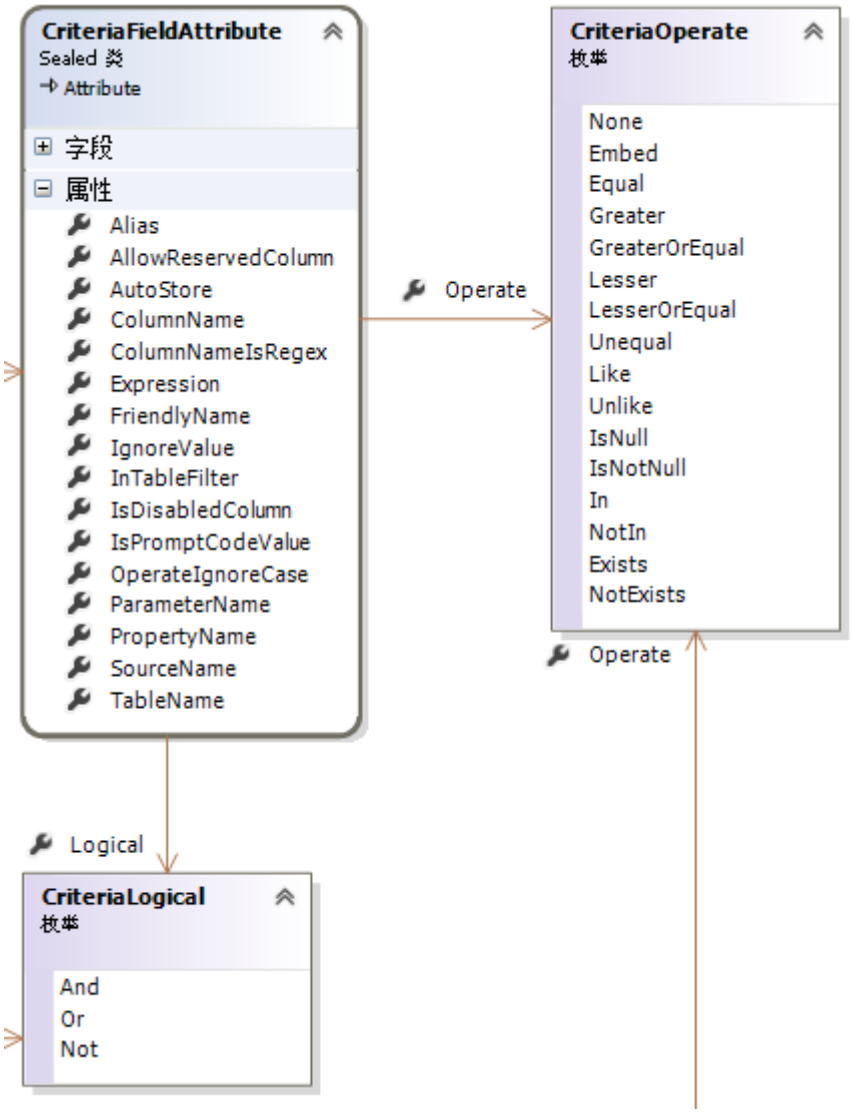
```
/// <summary>
/// 过程锁查询
/// </summary>
[System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("过程锁查询")]
public class ProcessLockCriteria : Phenix.Business.CriteriaBase
{
  /// <summary>
  /// 名称
  /// </summary>
  [Phenix.Core.Mapping.CriteriaField(Operate = Phenix.Core.Mapping.CriteriaOperate.Equal, Logical = Phenix.Core.Mapping.CriteriaLogical.And,
    FriendlyName = "名称", SourceName = "PH_PROCESSLOCK", Alias = "PL_NAME", TableName = "PH_PROCESSLOCK", ColumnName = "PL_NAME")]
  private string _name;
  /// <summary>
  /// 名称
  /// </summary>
  [System.ComponentModel.DisplayName("名称")]
  public string Name
  {
    get { return _name; }
    set
    {
      _name = value;
      PropertyHasChanged();
    }
  }

  /// <summary>
  /// 是否允许执行
  /// </summary>
  [Phenix.Core.Mapping.CriteriaField(Operate = Phenix.Core.Mapping.CriteriaOperate.Equal, Logical = Phenix.Core.Mapping.CriteriaLogical.And,
    FriendlyName = "是否允许执行", SourceName = "PH_PROCESSLOCK", Alias = "PL_ALLOWEXECUTE", TableName = "PH_PROCESSLOCK", ColumnName = "PL_ALLOWEXECUTE")]
  private int? _allowexecute;
  /// <summary>
  /// 是否允许执行
  /// </summary>
  [System.ComponentModel.DisplayName("是否允许执行")]
  public int? Allowexecute
  {
    get { return _allowexecute; }
```

```
        set
        {
          _allowexecute = value;
          PropertyHasChanged();
        }
      }
    }
  }
}
```

查询条件类的构建，可参考“03.Addin 工具使用方法”的“初始化/编辑查询类”章节。

17.1.2条件属性

在条件类中，被打上 Phenix.Core.Mapping.CriteriaFieldAttribute 标签的才是条件字段及其属性，它申明了持久层引擎应该如何拼装条件语句。



属性	说明	备注

SourceName	指示该条件字段对应的数据源名	未标注时取所属类最近的标签 ClassAttribute.FetchScript(首选)、或者取使用类最近的标签 ClassAttribute.FetchScript(次选)、或者取 TableName(次选)；
Alias	指示该条件字段对应的别名；如果为空，则认为从属性名获取	当 ClassAttribute.FetchScript 中存在字段别名时，应该与之对应；
TableName	指示该条件字段对应的哪张表	如果为空，则认为表名=所属类的 ClassAttribute.TableName；
ColumnName	指示该条件字段对应的表列名	如果为空，则认为表列名从字段名获取，获取方法为：字段名第一个字符如果为'_'将被裁剪掉；
ColumnNameIsRegex	ColumnName 是正则表达式	缺省为 false；当为 true 时，将在被 Fetch 的类中匹配所有字段的 FieldAttribute.ColumnName、FieldLinkAttribute.ColumnName，匹配到的字段将被构建到条件表达式中；
Expression	指示该条件字段对应的条件表达式	如果为空，则认为从表列名获取；
ParameterName	指示该条件字段对应的参数名	未标注时从字段名获取，获取方法为：字段名第一个字符如果为'_'将被裁剪掉；
PropertyName	指示该条件字段对应的属性名	未标注时从字段名获取，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	指示该条件字段对应的友好名	
Logical	指示该条件字段的逻辑（条件运算符）	缺省为 CriteriaLogical.And；
Operate	指示该条件字段的操作（条件操作符号）	缺省为 CriteriaOperate.Equal；
OperateIgnoreCase	条件操作忽略大小写（仅针对字符串类型的字段）	缺省为 false；
AutoStore	指示该条件字段可自动保存和恢复	缺省为 true；
IgnoreValue	指示忽略加入条件项的值	缺省为 String.Empty；
AllowReservedColumn	是否允许作为保留字段使用	缺省为 true；
IsDisabledColumn	指示该字段是禁用字段	缺省为 false；当 ColumnName 包含 Phenix.Core.Mapping.CodingStandards.DefaultD

		isabledColumnName 时必定是禁用字段（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；
InTableFilter	用于表过滤器	缺省为 true；

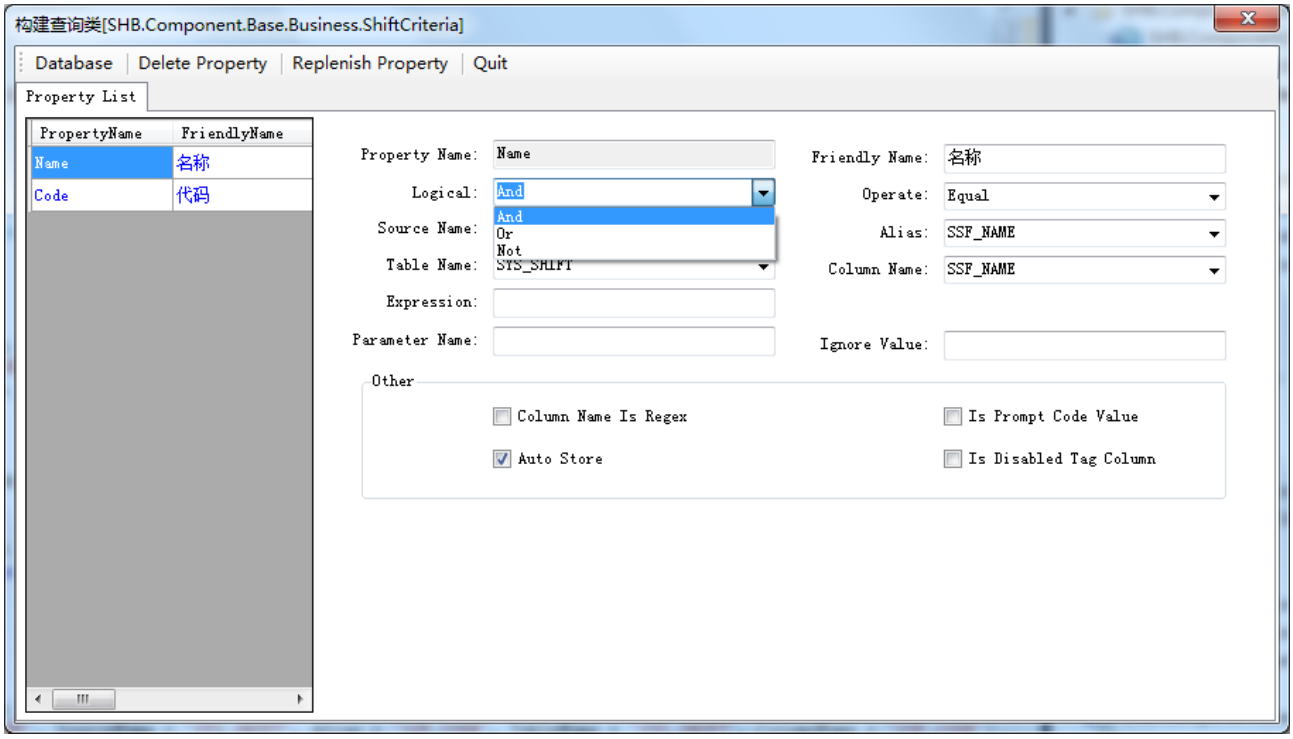
17.1.3条件语句

条件语句由“条件运算符”与“条件表达式”组成，而“条件表达式”则由条件操作符、所映射的表字段信息、属性值等组成。

如下示例中，黄底黑字部分为条件表达式，红字部分为条件运算符，括号表现了条件运算之间的嵌套关系：

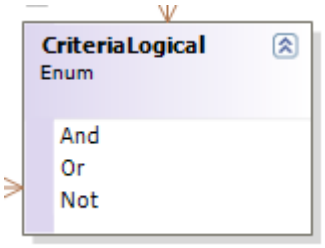
<Command Text="select CTI_ID, CTI_CHNNAME,CTI_ENGNAME, CTI_CODE,CTI_CHINESE_ADDRESS,CTI_ENGLISH_ADDRESS,CTI_CUSTOMER_STATE_FG from CSR_CUSTOMER_INFO where CTI_DISABLED='0' and ((CTI_CUSTOMER_TYPE_FG is null or CTI_CUSTOMER_TYPE_FG <> :P_CUSTOMER_TYPE_FGf48465da77ff) and (((CSR_CUSTOMER_INFO.CTI_ID is null or CSR_CUSTOMER_INFO.CTI_ID <> :customerId2c337226963b))))" /><Parameter Name="customerId2c337226963b" Value="0" /><Parameter Name="P_CUSTOMER_TYPE_FGf48465da77ff" Value="1" />

17.1.3.1 条件运算符



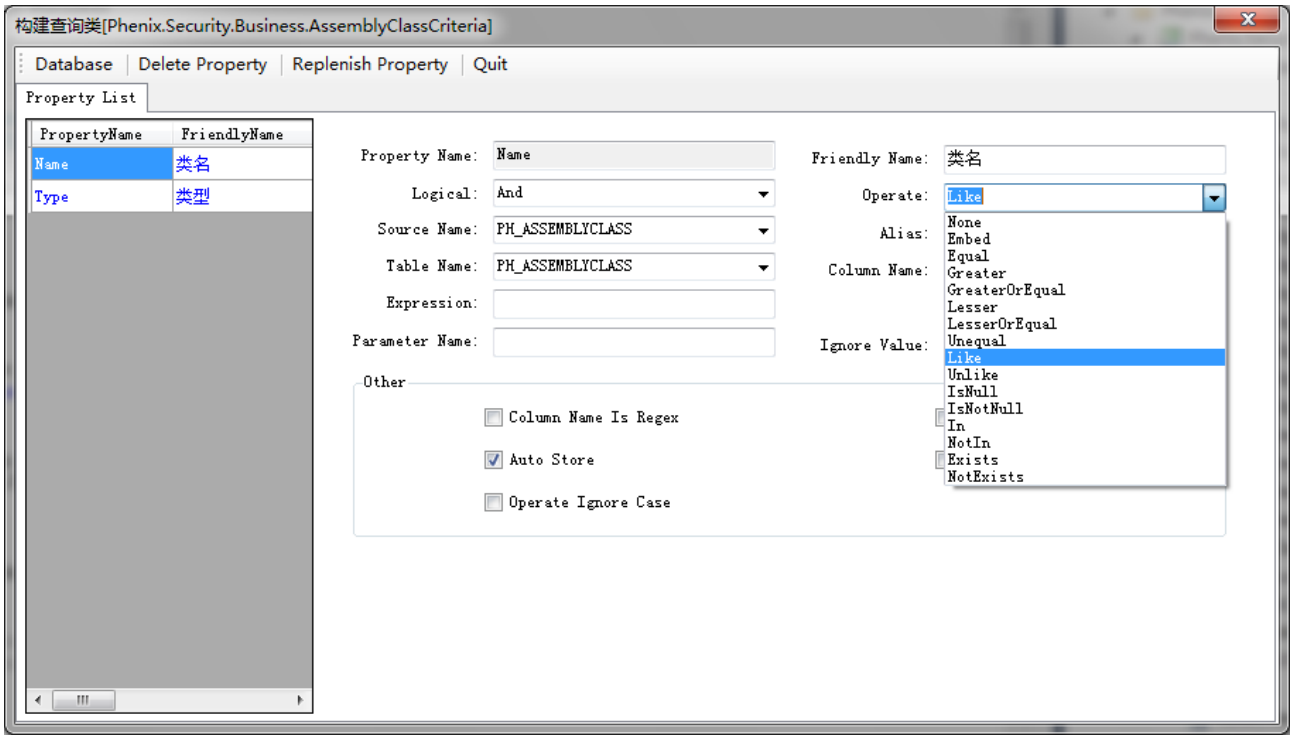
条件运算符，使用的是 Phenix.Core.Mapping.CriteriaLogical 枚举来表达，枚举内容如下图（望

文生义，不必解释）：

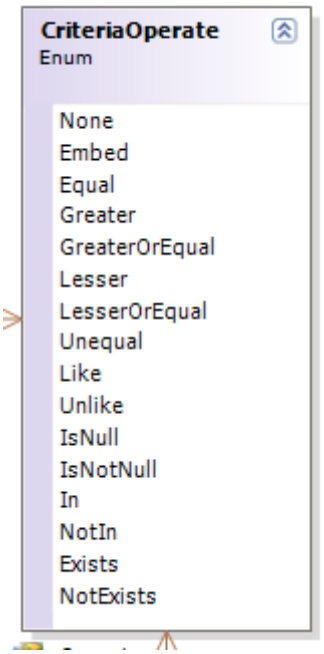


在一个条件类中，条件属性之间的条件运算顺序，是按照它在条件类中所定义的顺序来确定的。需提请注意的是，第一个条件属性所定义的条件运算符（Logical 值）是没有实质用处的，所以，随意定义它都不会对结果造成影响。

17.1.3.2 条件操作符



条件操作符（Operate 值），使用的是 Phenix.Core.Mapping.CriteriaOperate 枚举来表达，枚举内容如下图：



属性	说明	备注
None	无	不参与 SQL 的拼装，但参与参数定义和赋值；
Embed	嵌入	可实现嵌套条件；
Equal	=	
Greater	>	
GreaterOrEqual	>=	
Lesser	<	
LesserOrEqual	<=	
Unequal	<>	
Like	like	如果 Value 未含有%，则会在 Value 字符串的前后自动补上%；
Unlike	not like	如果 Value 未含有%，则会在 Value 字符串的前后自动补上%；
IsNull	is null	如果条件属性为 bool 型，当： ● Value = true，则拼装 Operate； ● Value = false，则拼装反相的 Operate； ● Value = null，则忽略本条件属性； 如果条件属性为非 bool 型，则无所谓 Value 取任何值（包括 null）都会拼装 Operate；
IsNotNull	is not null	
In	in	如果条件属性为 Array 型，例如：枚举类型数组、值类型数组、string[]类型等，则用’,’间隔数组项 ToString()的字符串拼
NotIn	not in	

		装到 Operate 中；否则，直接 ToString()被拼装到 Operate 中；
Exists	exists	适用于关联关系；
NotExists	not exists	

17.1.4忽略查询条件

一般情况下，可以将值类型的条件属性定义为 Nullable<T>类型，从而在需要的时候只需设置它的值为 null 就可以屏蔽掉这个条件项，不会被拼装到条件语句中了。

我们可以在条件类中穷尽（最大范围）定义上条件属性，而在检索的时候，通过将某些条件属性赋值为 null 来忽略掉它们，达到动态检索的目的。

另外，我们也可以通过设置 CriteriaFieldAttribute 标签的 IgnoreValue 属性来告知 Phenix 丶如何忽略它，这对在希望采用非 null 值来屏蔽条件项的应用场景下是非常有用的。Phenix 丶的判断方法是：将条件属性的值 ToString()比对 IgnoreValue 属性值内容（注意：对于枚举类型的属性，IgnoreValue 值应该设置为枚举值的 Name 而不是 Flag），一旦匹配就屏蔽掉这个条件属性。

17.1.5判断空不空 Phenix.Core.Mapping.CriteriaOperate.IsNull/IsNotNull

如何实现在查询条件中判断字段是否为空（或不为空）？由于我们用 null 作为忽略查询条件的缺省判断条件，所以不能直接对条件属性赋值 null 来达到拼接 is null 条件表达式到条件语句中的目的。此时，我们可以通过使用 Phenix.Core.Mapping.CriteriaOperate.IsNull/IsNotNull 来实现：

```
/// <summary>
/// Customer查询条件(主要是用来查询有效客户信息)(CST_CLOSE_DTGreater初始值为当前日期)
/// </summary>
[Serializable()]
public class CustomerCriteriaOtherEqual : ICriteria
{
  [CriteriaField(Operate = CriteriaOperate.IsNull, Logical = CriteriaLogical.Or, ColumnName =
"CST_CLOSE_DT")]
  private DateTime _CST_CLOSE_DT0;

  [CriteriaField(Operate = CriteriaOperate.Greater, Logical = CriteriaLogical.Or, ColumnName =
"CST_CLOSE_DT")]
  private DateTime? _CST_CLOSE_DTGreater = DateTime.Parse(DateTime.Now.ToShortDateString());
  /// <summary>
  /// 关闭日期如果存在，则必须大于当前日期，否则此客户信息为无效(and 连接)
  /// </summary>
  public DateTime? CST_CLOSE_DTGreater
  {
```

```
      get { return _CST_CLOSE_DTGreater; }
      set { _CST_CLOSE_DTGreater = value; }
    }
  }
```

提交到数据库的 SQL 语句将类似于：

```
CST_CLOSE_DT is null or CST_CLOSE_DT = :CST_CLOSE_DT
```

具体细节，请见前文“条件操作符号”章节。

17.1.6嵌套条件 Phenix.Core.Mapping.CriteriaOperate.Embed

查询类可以作为子查询类被嵌套到另一个查询类中：

```
  /// <summary>
  /// 过程锁查询
  /// </summary>
  [System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("过程锁查询")]
  public class ProcessLockCriteria : Phenix.Business.CriteriaBase
  {
    /// <summary>
    /// 名称
    /// </summary>
    [Phenix.Core.Mapping.CriteriaField(Operate = Phenix.Core.Mapping.CriteriaOperate.Equal, Logical =
Phenix.Core.Mapping.CriteriaLogical.And,
      FriendlyName = "名称", SourceName = "PH_PROCESSLOCK", Alias = "PL_NAME", TableName =
"PH_PROCESSLOCK", ColumnName = "PL_NAME")]
    private string _name;
    /// <summary>
    /// 名称
    /// </summary>
    [System.ComponentModel.DisplayName("名称")]
    public string Name
    {
      get { return _name; }
      set
      {
        _name = value;
        PropertyHasChanged();
      }
    }
```

```
    /// <summary>
    /// 是否允许执行
    /// </summary>
    [Phenix.Core.Mapping.CriteriaField(Operate = Phenix.Core.Mapping.CriteriaOperate.Equal, Logical =
Phenix.Core.Mapping.CriteriaLogical.And,
      FriendlyName = "是否允许执行", SourceName = "PH_PROCESSLOCK", Alias = "PL_ALLOWEXECUTE", TableName
= "PH_PROCESSLOCK", ColumnName = "PL_ALLOWEXECUTE")]
    private int? _allowexecute;
    /// <summary>
    /// 是否允许执行
    /// </summary>
    [System.ComponentModel.DisplayName("是否允许执行")]
    public int? Allowexecute
    {
      get { return _allowexecute; }
      set
      {
        _allowexecute = value;
        PropertyHasChanged();
      }
    }

    /// <summary>
    /// 子条件
    /// </summary>
    [Phenix.Core.Mapping.CriteriaField(Operate = Phenix.Core.Mapping.CriteriaOperate.Embed, Logical =
Phenix.Core.Mapping.CriteriaLogical.Or, FriendlyName = "子条件")]
    private ProcessLockCriteria _subCriteria;
    /// <summary>
    /// 子条件
    /// </summary>
    [System.ComponentModel.DisplayName("名称")]
    public ProcessLockCriteria SubCriteria
    {
      get { return _subCriteria; }
      set
      {
        _subCriteria = value;
        PropertyHasChanged();
      }
    }
  }
```

如上述示例，嵌套条件务必在 Phenix.Core.Mapping.CriteriaFieldAttribute 标签中将属性 Operate = Phenix.Core.Mapping.CriteriaOperate.Embed，否则持久层引擎并不认可它为嵌套条件。

下面演示了使用嵌套条件的代码编写形式：

```
WorkingProcessLocks = ProcessLockList.Fetch(new ProcessLockCriteria()
  {
    Name = "1",
    SubCriteria = new ProcessLockCriteria()
      {
        Allowexecute = 1
      }
  });
```

可以在日志文件中找到拼装的 SQL 语句：

```
<Command Text="select PL_NAME,PL_ALLOWEXECUTE,PL_TIME,PL_USERNUMBER,PL_REMARK from PH_PROCESSLOCK where ( PH_PROCESSLOCK.PL_NAME = :namebe2dc0b6f961 or ( PH_PROCESSLOCK.PL_ALLOWEXECUTE = :Pteriaallowexecute0996ed241c3f))" /><Parameter Name="namebe2dc0b6f961" Value="1" /><Parameter Name="Pteriaallowexecute0996ed241c3f" Value="1" />
```

17.1.7数组条件 Phenix.Core.Mapping.CriteriaOperate.In/NotIn

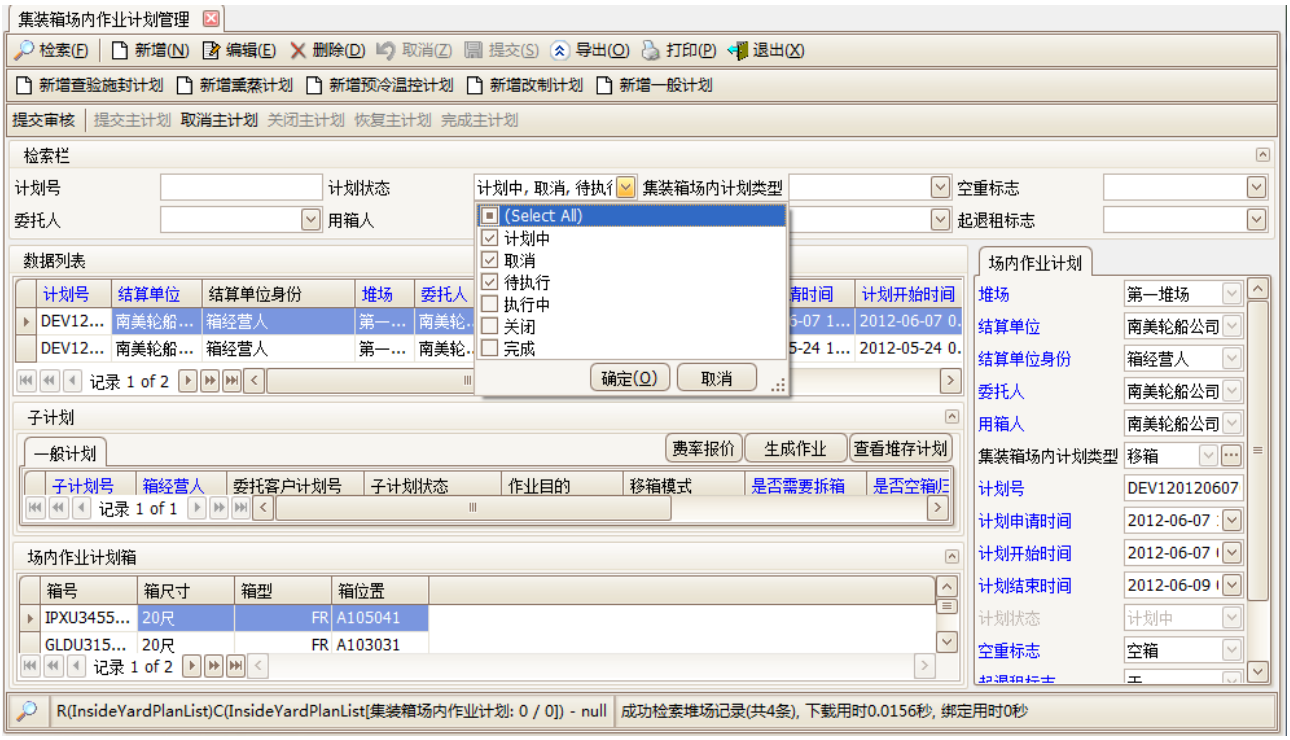
在“12.业务结构对象模型”的“枚举在业务结构中的使用方法”章节中，我们用到了枚举类型数组，可被拼装到 In/NotIn 语句中：

```
[CriteriaField(Operate = CriteriaOperate.In, Logical = CriteriaLogical.And, FriendlyName = "计划状态", ColumnName = "CYP_PLAN_STATUS_FG")]
private PlanStatus[] _planStatusIn;
/// <summary>
/// 计划状态
/// </summary>
public string PlanStatusIn
{
  get { return Phenix.Core.Code.Converter.EnumArrayToFlags<PlanStatus>(_planStatusIn); }
  set { _planStatusIn = Phenix.Core.Code.Converter.FlagsToEnumArray<PlanStatus>(value); }
}
```

当检索数据时，提交到数据库的 SQL 语句将类似于：

```
<Command Text="select CYP_ID,CYP_YRD_ID,CYP_CONSIGNOR_CTI_ID,CYP_CTN_USER_CTI_ID,CYP_CYT_ID,CYP_PLAN_SERIAL,CYP_APPLY_TIME from CWP_INSIDE_YARD_PLAN where ( ( CWP_INSIDE_YARD_PLAN.CYP_PLAN_STATUS_FG in (0,1,2)))" />
```

条件属性的字段类型是枚举数组，而属性类型则是 string，是为了能够绑定到界面控件上，实现如下多选项的业务场景：



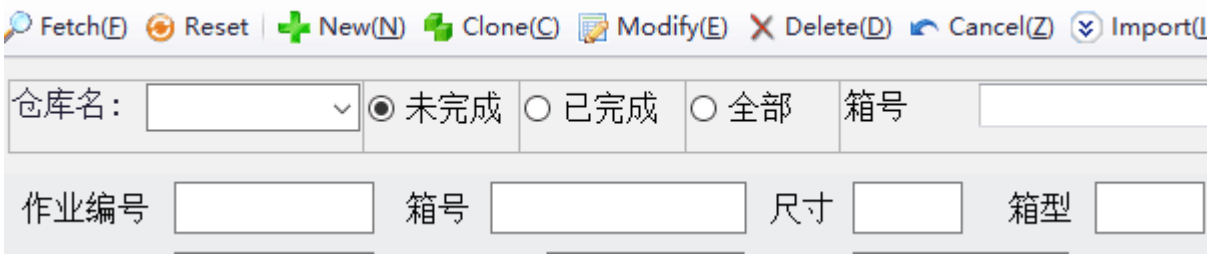
除了枚举数组类型，值数组类型、string 数组类型等都能被拼装到 In/NotIn 语句中：

```
        [CriteriaField(Operate = CriteriaOperate.In, Logical = CriteriaLogical.And, FriendlyName = "ID", ColumnName = "CPI_ID")]
        private long?[] _IDRange;
        /// <summary>
        /// ID
        /// </summary>
        public long?[] IDRange
        {
            get { return _IDRange; }
            set { _IDRange = value; }
        }

        [CriteriaField(Operate = CriteriaOperate.In, Logical = CriteriaLogical.And, FriendlyName = "箱位置", ColumnName = "CPI_LOCATION")]
        private string[] _locationIn;
        /// <summary>
        /// 箱位置
        /// </summary>
        public string[] LocationIn
        {
```

```
            get { return _locationIn; }
            set { _locationIn = value; }
        }
```

被拼装的 string 数组项值会被打上引号，拼接到 in 条件表达式中（具体细节，请见前文“条件操作符号”章节）。

如果界面设计风格是要绑定到一个个选择框的话：



查询类可以这样写：

```
[System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("查询进场计划")]
public class InPlanViewCriteria : Phenix.Business.CriteriaBase
{
  [Phenix.Core.Mapping.CriteriaField(Operate = Phenix.Core.Mapping.CriteriaOperate.In,
    Logical = Phenix.Core.Mapping.CriteriaLogical.And,
    FriendlyName = "计划状态", SourceName = "WDP_IN_PLAN_V", Alias = "IPL_PLAN_STATUS_FG", TableName
= "WIP_IN_PLAN",
    ColumnName = "IPL_PLAN_STATUS_FG")]
  private PlanStatus[] _planStatusIn;

  /// <summary>
  /// 包含未完成计划
  /// </summary>
  [System.ComponentModel.DisplayName("包含未完成计划")]
  public bool ContainNotFinish
  {
    get { return _planStatusIn.Contains(PlanStatus.NotFinish); }
    set
    {
      List<PlanStatus> planStatusIn = _planStatusIn.ToList();
      if (value)
      {
        if (!planStatusIn.Contains(PlanStatus.NotFinish))
          planStatusIn.Add(PlanStatus.NotFinish);
      }
      else
```

```
        planStatusIn.Remove(PlanStatus.NotFinish);
      _planStatusIn = planStatusIn.ToArray();
      PropertyHasChanged();
    }
  }

  /// <summary>
  /// 包含完成计划
  /// </summary>
  [System.ComponentModel.DisplayName("包含完成计划")]
  public bool ContainFinish
  {
    get { return _planStatusIn.Contains(PlanStatus.Finish); }
    set
    {
      List<PlanStatus> planStatusIn = _planStatusIn.ToList();
      if (value)
      {
        if (!planStatusIn.Contains(PlanStatus.Finish))
          planStatusIn.Add(PlanStatus.Finish);
      }
      else
        planStatusIn.Remove(PlanStatus.Finish);
      _planStatusIn = planStatusIn.ToArray();
      PropertyHasChanged();
    }
  }

  /// <summary>
  /// 包含全部状态
  /// </summary>
  [System.ComponentModel.DisplayName("包含全部状态")]
  public bool ContainAll
  {
    get { return ContainNotFinish && ContainFinish; }
    set
    {
      ContainNotFinish = value;
      ContainFinish = value;
      PropertyHasChanged();
    }
  }
}
```

以上示例用到的 PlanStatus 枚举，代码如下：

```
/// <summary>
/// 计划状态
/// </summary>
[Phenix.Core.Operate.KeyCaptionAttribute(FriendlyName = "计划状态"), System.SerializableAttribute()]
public enum PlanStatus
{
  /// <summary>
  /// 未完成
  /// </summary>
  [Phenix.Core.Rule.EnumCaptionAttribute("未完成")]
  NotFinish,

  /// <summary>
  /// 完成
  /// </summary>
  [Phenix.Core.Rule.EnumCaptionAttribute("完成")]
  Finish,
 }
```

17.1.8子查询条件 Phenix.Core.Mapping.CriteriaOperate.Exists/NotExists

业务类也可以被嵌入到条件类中作为子查询条件，此时这个业务类必须与被查询的业务类存在着关联关系。

17.1.8.1 通过表结构关系嵌入子查询条件

通过表结构关系，可以将主、子查询条件之间的关联关系自动勾连、拼接为关联条件，这样可以不必在代码中特别申明，只要这两个业务类之间符合：

- 互为主从表关系；
- 拥有共同的主表；

```
[CriteriaField(Operate = CriteriaOperate.Exists, Logical = CriteriaLogical.And)]
SHB.Component.Customer.Business.CustomerIdentity _customerIdentity;
/// <summary>
/// 客户身份类型
/// </summary>
public SHB.Component.Customer.Rule.CustomerIdentity? CustomerIdentity
{
    get { return _customerIdentity == null ? null : _customerIdentity.CustIdentity; }
    set
    {
        if (_customerIdentity == null)
        {
            _customerIdentity = new SHB.Component.Customer.Business.CustomerIdentity ();
```

```
                }
                _customerIdentity.CustIdentity = value;
            }
        }
```

本案例中，CustomerIdentity 是 Customer 的从业务类，在 CustomerIdentity 业务类中应该定义上外键字段及其映射关系：

```
    /// <summary>
    /// 客户身份类别
    /// </summary>
    [System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("客户身份类别"),
Phenix.Core.Mapping.Class("CSR_CUSTOMER_IDENTITY", FriendlyName = "客户身份类别")]
    public abstract class CustomerIdentity<T> : Phenix.Business.BusinessBase<T> where T :
CustomerIdentity<T>
    {
        /// <summary>
        /// 客户身份类别
        /// </summary>
      public static readonly
Phenix.Business.PropertyInfo<SHB.Component.Customer.Rule.CustomerIdentity?> CustomerIdentityProperty =
RegisterProperty<SHB.Component.Customer.Rule.CustomerIdentity?>(c => c.CustIdentity);
        [Phenix.Core.Mapping.FieldUniqueAttribute("PK_CSR_CUSTOMER_IDENTITY")]
        [Phenix.Core.Mapping.Field(PropertyName = "CustomerIdentity", FriendlyName = "客户身份类别",
Alias = "CCI_CUSTOMER_IDENTITY_FG", TableName = "CSR_CUSTOMER_IDENTITY", ColumnName =
"CCI_CUSTOMER_IDENTITY_FG", IsPrimaryKey = true, NeedUpdate = true)]
      private SHB.Component.Customer.Rule.CustomerIdentity? _customerIdentity;
        /// <summary>
        /// 客户身份类别
        /// </summary>
        [System.ComponentModel.DisplayName("客户身份类别")]
        public SHB.Component.Customer.Rule.CustomerIdentity? CustIdentity
        {
            get { return GetProperty(CustomerIdentityProperty, _customerIdentity); }
            set { SetProperty(CustomerIdentityProperty, ref _customerIdentity, value); }
        }
        /// <summary>
        /// 客户身份类别
        /// </summary>
        [System.ComponentModel.DisplayName("客户身份类别")]
        public string CustIdentityCaption
        {
            get { return Phenix.Core.Rule.EnumKeyCaption.GetCaption(_customerIdentity); }
        }
```

```
        /// <summary>
        /// 客户信息
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<long?> CCI_CTI_IDProperty =
RegisterProperty<long?>(c => c.CCI_CTI_ID);
        [Phenix.Core.Mapping.FieldLinkAttribute("CSR_CUSTOMER_INFO", "CTI_ID")]
        [Phenix.Core.Mapping.Field(PropertyName = "CCI_CTI_ID", FriendlyName = "客户信息", TableName =
"CSR_CUSTOMER_IDENTITY", ColumnName = "CCI_CTI_ID", IsPrimaryKey = true, NeedUpdate = true)]
        private long? _CCI_CTI_ID;
        /// <summary>
        /// 客户信息
        /// </summary>
        [System.ComponentModel.DisplayName("客户信息")]
        public long? CCI_CTI_ID
        {
            get { return GetProperty(CCI_CTI_IDProperty, _CCI_CTI_ID); }
            set { SetProperty(CCI_CTI_IDProperty, ref _CCI_CTI_ID, value); }
        }

        [System.ComponentModel.Browsable(false)]
        [System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
        public override string PrimaryKey
        {
            get { return String.Format("{0},{1},", CustIdentity, CCI_CTI_ID); }
        }
    }
```

上述代码中，CCI_CTI_ID 属性所映射的 CSR_CUSTOMER_IDENTITY 表 CCI_CTI_ID 字段是个外键字段，关联到的是主表 CSR_CUSTOMER_INFO（映射为 CustomerInfo 类）的 CTI_ID 字段。黄底黑字部分是它的 Link 标记，之所以打上了双划线，是因为这个关联关系使用了物理外键（在数据库中实际构建了外键），我们无需再在代码中标记 Phenix.Core.Mapping.FieldLinkAttribute（Phenix 在初始化这个业务类的时候会为它自动打上这个标记）。

这样，在检索时是这么写的：

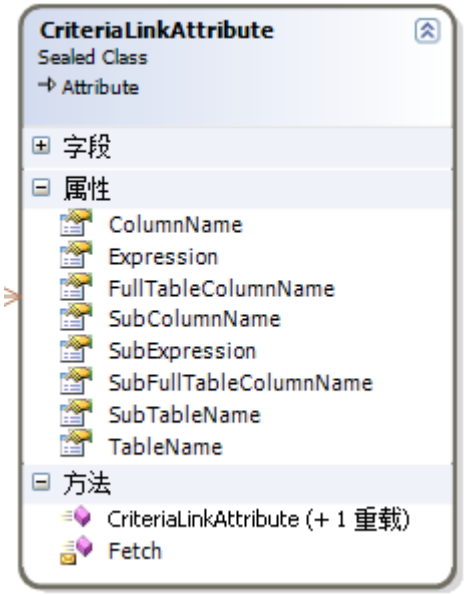
```
    return CustomerInfoList.Fetch(new CustomerCriteria
      {
         CustomerState = InformationStatus.Formal,
         CustomerIdentity = identity
      });
```

提交到数据库的 SQL 语句将类似于：

<Command Text="select CTI_ID,CTI_ORIGINAL_CTI_ID,CTI_CTC_ID,CTI_CSC_ID,CTI_SCR_ID,CTI_PARENT_CTI_ID,CTI_CHNNAME,CTI_ENGNAME,CTI_SHORT_CHNNAME,CTI_SHORT_ENGNAME,CTI_CODE,CTI_CHINESE_ADDRESS,CTI_ENGLISH_ADDRESS from CSR_CUSTOMER_INFO where CTI_CUSTOMER_STATE_FG=:PCTI_CUSTOMER_STATE_FG and (exists (select * from CSR_CUSTOMER_IDENTITY where CCI_CTI_ID=CTI_ID and CCI_CUSTOMER_IDENTITY_FG=:PTITY_CCI_CUSTOMER_IDENTITY_FG))" /><Parameter Name="PCTI_CUSTOMER_STATE_FG" Value="5" /><Parameter Name="PTITY_CCI_CUSTOMER_IDENTITY_FG" Value="0" />

17.1.8.2 通过申明关联关系嵌入子查询条件

除了上述方法之外，我们也可以通过显式地申明关联关系，实现子查询条件的嵌入。申明关联关系的标签类为 Phenix.Core.Mapping.CriteriaLinkAttribute：



属性	说明	备注
TableName	关联表的表名	
ColumnName	关联表的表列名	
Expression	指示该关联表字段对应的表达式	如果为空，则认为从 FullTableColumnName 获取
SubTableName	关联子条件表的表名	
SubColumnName	关联子条件表的表列名	
SubExpression	指示该关联子条件表字段对应的表达式	如果为空，则认为从 SubFullTableColumnName 获取

以前文中的示例，可改写为：

```
[CriteriaField(Operate = CriteriaOperate.Exists, Logical = CriteriaLogical.And)]
```

```
[CriteriaLink("CSR_CUSTOMER_INFO", "CTI_ID", "CSR_CUSTOMER_IDENTITY ", "CCI_CTI_ID")]
SHB.Component.Customer.Business.CustomerIdentity _customerIdentity;
/// <summary>
/// 客户身份类型
/// </summary>
public SHB.Component.Customer.Rule.CustomerIdentity? CustomerIdentity
{
    get { return _customerIdentity == null ? null : _customerIdentity.CustIdentity; }
    set
    {
        if (_customerIdentity == null)
        {
            _customerIdentity = new SHB.Component.Customer.Business.CustomerIdentity ();
        }
        _customerIdentity.CustIdentity = value;
    }
}
```

得到的效果和前文中的是一样的。当然这仅仅是个示例，实际编写时这行代码可以省略掉，因为是主从表结构，是允许不显式申明的。

17.2 CriteriaExpression

Phenix.Core.Mapping.CriteriaExpression 是 Phenix 为 Fetch 业务对象提供的一种输入条件的方式，它与 Phenix.Business.PropertyInfo<T>组合，可实现类似于 LINQ 一样的语法：

```
ProcessLockList processLocks = ProcessLockList.Fetch(ProcessLock.AllowexecuteProperty == true & ("a" + ProcessLock.NameProperty).Like("a%"));
```

可以在日志文件中找到拼装的 SQL 语句：

```
<Command Text="select PL_NAME,PL_ALLOWEXECUTE,PL_TIME,PL_USERNUMBER,PL_REMARK from PH_PROCESSLOCK where ( PL_ALLOWEXECUTE = :P4eebf63317eb and :P7a0f3d3107ee || PL_NAME like :P83f67c33e223 )" /><Parameter Name="P4eebf63317eb" Value="1" /><Parameter Name="P7a0f3d3107ee" Value="a" /><Parameter Name="P83f67c33e223" Value="a%" />
```

上述示例中，也顺带演示了 Like 操作符的用法。如果传入的值含有%，则自动拼装到 where 条件语句中（这样就不会被自动在值的前后补上%了）。

我们在 Criteria 条件对象中也可以做到这点，示例如下：

```
[System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("类信息查询")]
```

```
public class AssemblyClassCriteria : Phenix.Business.CriteriaBase
{
  [Phenix.Core.Mapping.CriteriaField(FriendlyName = "类名", Logical =
Phenix.Core.Mapping.CriteriaLogical.And, Operate = Phenix.Core.Mapping.CriteriaOperate.Like, TableName
= "PH_ASSEMBLYCLASS", ColumnName = "AC_NAME")]
  private string _name;
  /// <summary>
  /// 类名
  /// </summary>
  [System.ComponentModel.DisplayName("类名")]
  public string Name
  {
    get { return _name != null ? _name.TrimEnd('%') : null; }
    set { _name = value != null ? String.Format("{0}%", value) : null; PropertyHasChanged(); }
  }
```

Name 属性绑定的输入控件，不会显示出含%的内容，输入时也不必刻意要求带上%，因为在 get/set 时，以上代码已做了自动裁剪。

17.2.1条件运算符

CriteriaExpression 的条件运算符，采取的是重载类的条件运算符方式：

条件运算符	说明
!	not
&	and
|	or

17.2.2条件操作符

CriteriaExpression 的条件操作符，部分采取的是重载类的条件操作符方式：

条件操作符	说明
==	=
>	>
>=	>=
<	<
<=	<=
!=	<>

另外一部分则采取的是函数方式:

函数	说明
Like()	like
Unlike()	not like
LikeIgnoreCase()	like
UnlikeIgnoreCase()	not like
IsNull	is null
IsNotNull	is not null
In()	in
NotIn()	not in
Exists()	exists
NotExists()	not exists

17.2.3计算符

CriteriaExpression 允许在条件表达式内存在计算符，并被拼装到条件语句中：

计算符	说明
+	适用于数值和 string 类型 string 类型的计算符在 SQL 语句中会被转换为"||"
-	适用于数值类型
*	适用于数值类型
/	适用于数值类型

以前文中的例子为例（注意黄底黑字部分）：

```
ProcessLockList processLocks = ProcessLockList.Fetch(ProcessLock.AllowexecuteProperty == true &
("a" + ProcessLock.NameProperty).Like("a%"));
```

可以在日志文件中找到拼装的 SQL 语句：

```
<Command Text="select PL_NAME,PL_ALLOWEXECUTE,PL_TIME,PL_USERNUMBER,PL_REMARK from PH_PROCESSLOCK
where ( PL_ALLOWEXECUTE = :P4eebf63317eb and :P7a0f3d3107ee || PL_NAME like :P83f67c33e223 )"
/><Parameter Name="P4eebf63317eb" Value="1" /><Parameter Name="P7a0f3d3107ee" Value="a" /><Parameter
```

Name="P83f67c33e223" Value="a%" />

另外一部分，则采取的是函数方式:

函数	说明	备注
Length	字符数	适用于 string 类型
		在 SQL 语句中会被转换为"Length()"/"Len()"
ToLower()	转换为小写形式	适用于 string 类型
		在 SQL 语句中会被转换为"Lower()"
ToUpper()	转换为大写形式	适用于 string 类型
		在 SQL 语句中会被转换为"Upper()"
TrimStart()	去除字符串左边的空格	适用于 string 类型
		在 SQL 语句中会被转换为" LTrim()"
TrimEnd()	去除字符串右边的空格	适用于 string 类型
		在 SQL 语句中会被转换为" RTrim()"
Trim()	去除字符串左右两边的空格	适用于 string 类型
		在 SQL 语句中会被转换为"LTrim(RTrim())"
Substring()	截取字符串	适用于 string 类型
		在 SQL 语句中会被转换为"Substr()"/"Substring()"

例如：

```
WorkingProcessLocks = ProcessLockList.Fetch(ProcessLock.NameProperty.Trim() == "A");
```

可以在日志文件中找到拼装的 SQL 语句：

```
<Command Text="select PL_NAME,PL_ALLOWEXECUTE,PL_TIME,PL_USERNUMBER,PL_REMARK from PH_PROCESSLOCK where LTrim(RTrim( PL_NAME )) = :Pb3d928d0a6ef " /><Parameter Name="Pb3d928d0a6ef" Value="A" />
```

17.2.4子查询条件

CriteriaExpression 所提供的子查询条件中关联条件的编码方式非常简单，仅需在子查询条件表达式后加上 Where 函数即可：

```
WorkerList workerList = WorkerList.Fetch(
  Worker.NameProperty == "a" &
  WorkerWorkTypeList.Exists(WorkerWorkType.RemarkProperty.IsNull).
  Where(WorkerWorkType.WWT_WOK_IDProperty == Worker.WOK_IDProperty));
```

可以在日志文件中找到拼装的 SQL 语句：

```
<Command Text="select WOK_ID,WOK_NAME,WOK_CODE,WOK_OPEN_DT,WOK_CLOSE_DT,WOK_REMARK,WOK_INPUTER from VHL_WORKER where ( WOK_NAME = :P4e3ae0880947 and ( exists (select * from VHL_WORKER_WORK_TYPE where WWT_WOK_ID = WOK_ID and ( WWT_REMARK is null ))))" /><Parameter Name="P4e3ae0880947" Value="a" />
```

由于 Where 函数传入的参数也是一个条件表达式，所以关联条件的复杂度可以和普通的条件表达式相当。当然，Phenix 并不会检查编码的逻辑合理性，所以这是由开发者自行把握的：

```
WorkerList workerList = WorkerList.Fetch(
  Worker.NameProperty == "a" &
  WorkerWorkTypeList.Exists(WorkerWorkType.RemarkProperty.IsNull).
  Where(WorkerWorkType.WWT_WOK_IDProperty == Worker.WOK_IDProperty + 1));
```

可以在日志文件中找到拼装的 SQL 语句：

```
<Command Text="select WOK_ID,WOK_NAME,WOK_CODE,WOK_OPEN_DT,WOK_CLOSE_DT,WOK_REMARK,WOK_INPUTER,WOK_INPUTTIME from VHL_WORKER where ( WOK_NAME = :P4e3ae0880947 and ( exists (select * from VHL_WORKER_WORK_TYPE where WWT_WOK_ID = WOK_ID + :Pe5a7becd97fb and ( WWT_REMARK is null ))))" /><Parameter Name="P4e3ae0880947" Value="a" /><Parameter Name="Pe5a7becd97fb" Value="1" />
```

上述代码肯定是不合逻辑的，这只是一个示例而已。

17.3 Expression<Func<TBusiness, bool>>

本方法借用了.NET 的 LINQ 表达式机制，但有一定的局限：

- 仅限于业务类自身的属性定义参与到表达式的构建；
- 仅使用了部分的 LINQ 表达式运算符，是它的一个子集；

请尽量使用 Phenix.Core.Mapping.CriteriaExpression 与 Phenix.Business.PropertyInfo<T>组合构建条件的方法，本方法仅做作为一种构建条件的辅助手段。