5 业务对象公共接口的授权

5.1 业务对象公共接口

面向对象的三个基本特征之一就是“封装”。业务对象自身拥有的数据只有通过public的属性、过程、事件等方式才能被外部对象操作。

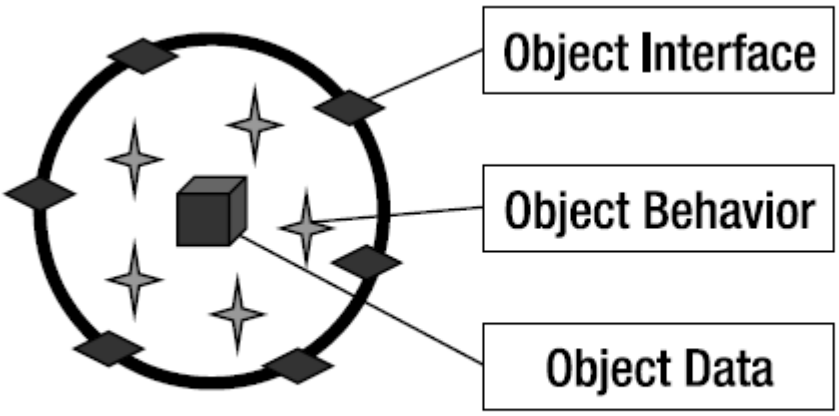


Figure 1-14. *A business object composed of state, implementation, and interface*

上图摘自CSLA作者的《Expert C# 2008 Business Objects》。

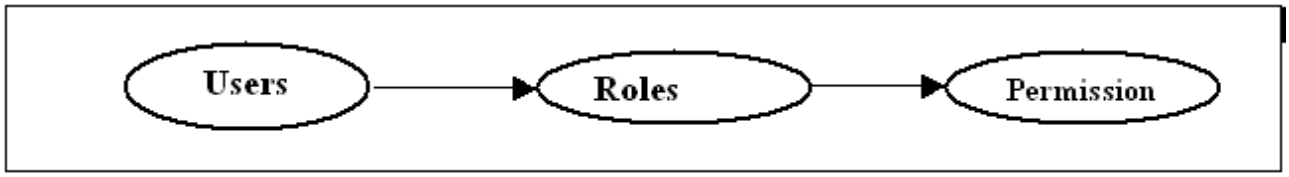
面向对象的“封装”落实到具体实现，就是业务对象公共接口的设计应该最小限度地将数据和操作暴露给外部对象，并禁止不可信的数据和操作，这个不可信的操作是需要被验证的，也就是业务对象公共接口的操作授权。

操作授权是业务对象的职责之一。

5.2 什么是授权

狭义的授权就是指权限控制，权限是用户对应用系统中的数据或者用数据表示的其它资源进行访问的许可。

应用系统的权限管理是基于角色的系统安全控制模型：



基于角色控制的基本思想

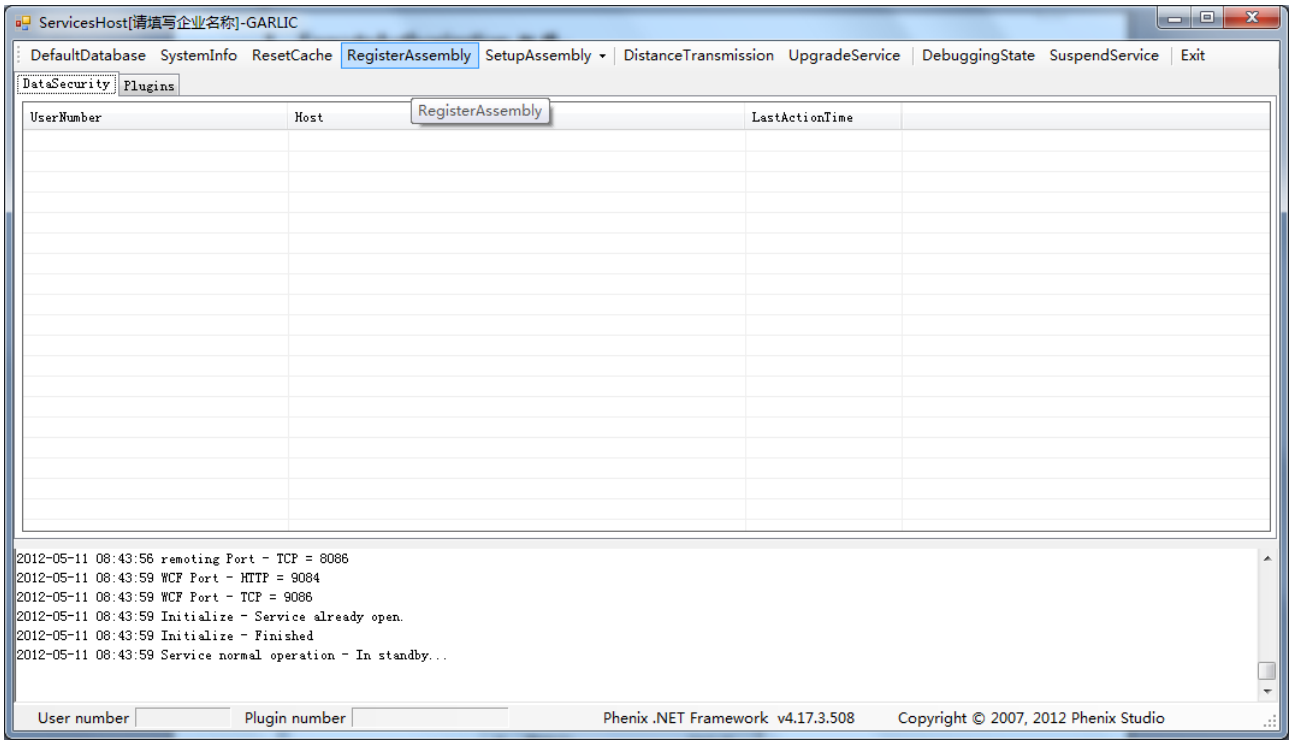
在这样的语境里，通常对权限的表述是：XX角色具有…的权限；如果再加上限定条件，则是：XX角色在…条件下具有…的权限，这就是广义的授权，而这个限定条件就是我们所谓的业务规则。所以，授

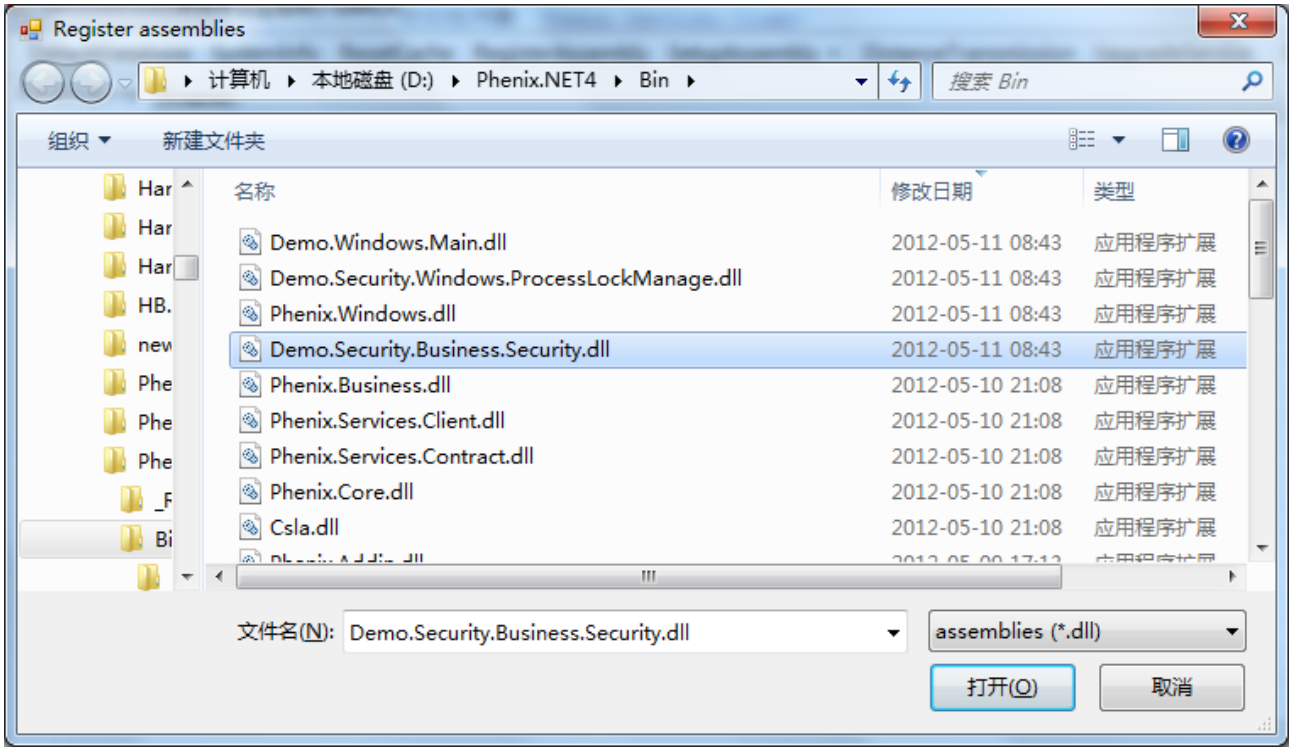
权是：权限+业务规则的综合体。

5.3 怎么授权

5.3.1 注册业务组件

为了实现业务对象公共接口的授权，必须事先通过 Phenix.Services.Host.exe 注册业务组件（业务类的程序集）：





找到业务组件的程序集，打开后注册的提示信息类似于：

2012-05-11 08:47:47 Initialize failure - If without registration procedures set components can be omitted the tip.：Demo.Security.Business.Security, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null未找到 IPlugin 接口

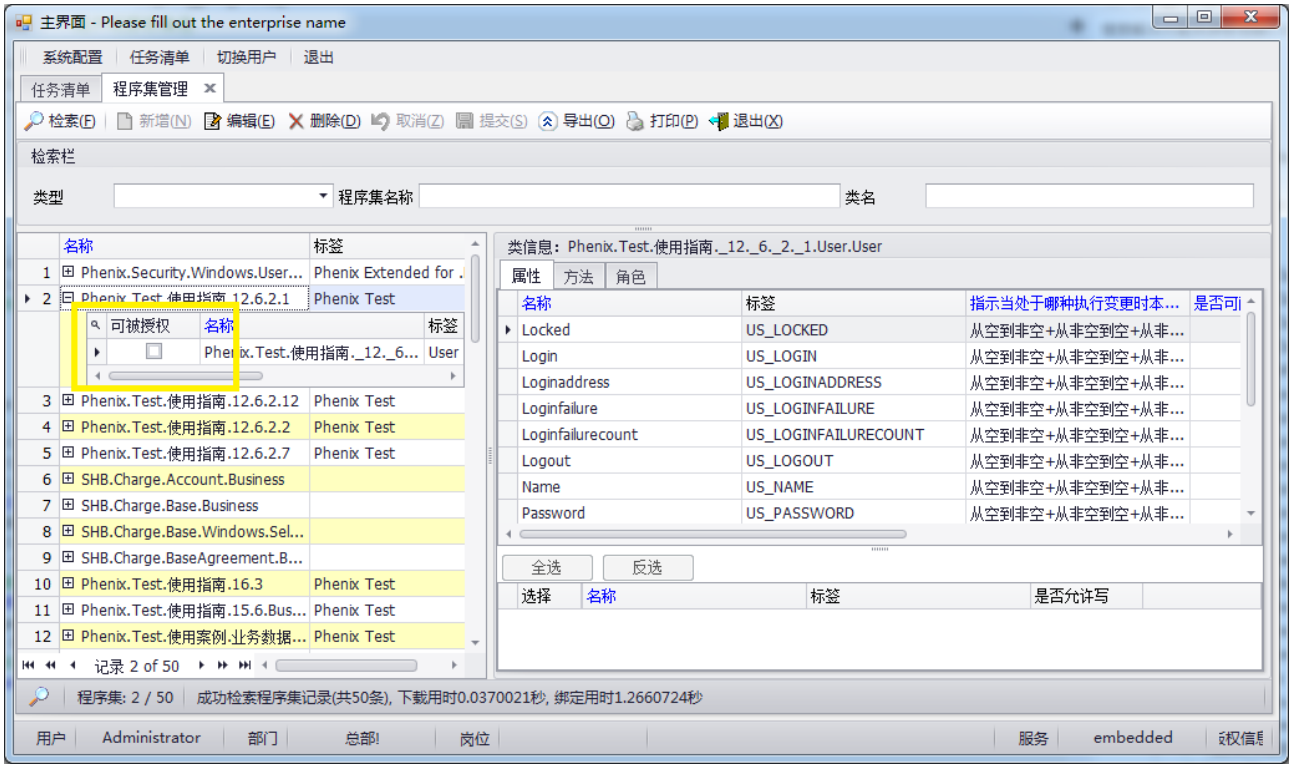
2012-05-11 08:47:49 Initialize - Register class: Demo.Security.Business.Security.ProcessLock

2012-05-11 08:47:49 Initialize - Register class: Demo.Security.Business.Security.ProcessLockList

2012-05-11 08:47:49 Initialize - Registered assembly over.

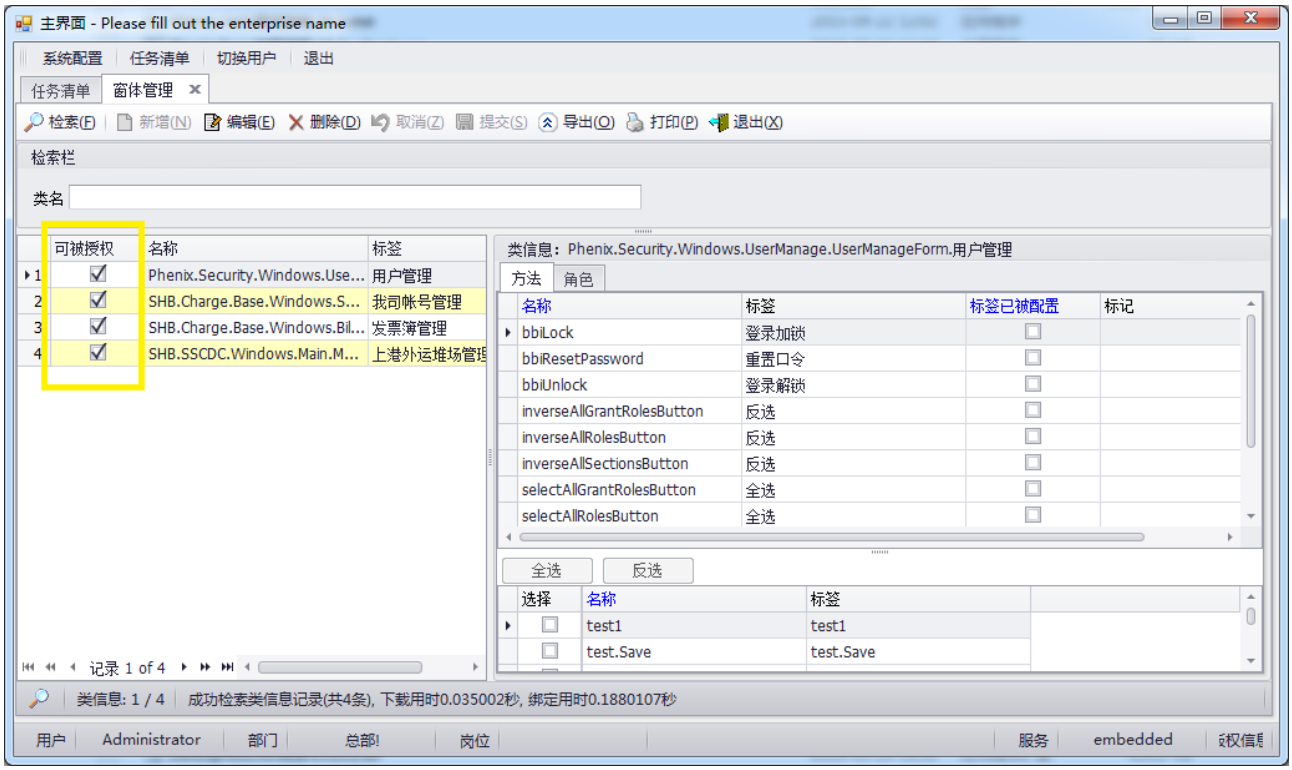
5.3.2 与角色的关联

这种关联是可以做到标准化、通用化的配置，具体由 Phenix.Security 权限管理功能组件包实现，见“Phenix.Security.Windows.AssemblyManage”、“Phenix.Security.Windows.RoleManage”、“Phenix.Security.Windows.UserManage”工程：



注意上述截屏中黄框内的“可被授权”选项，这个开关决定了当前类是否确实被授权管理上。

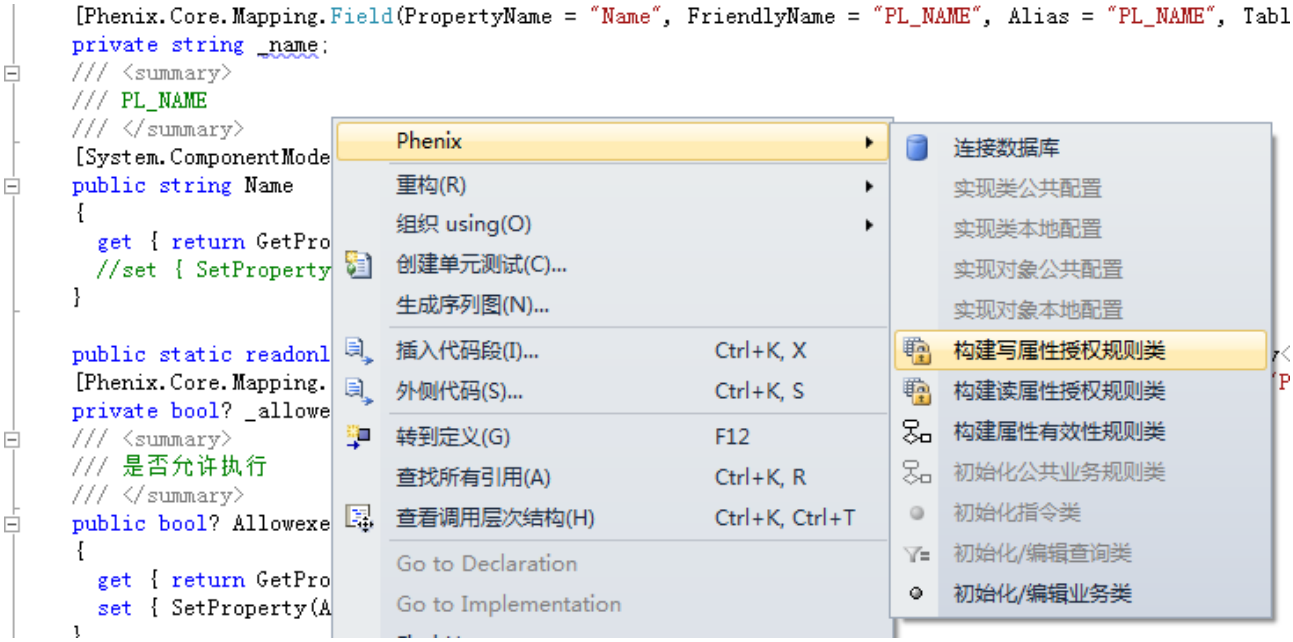
默认下，仅有窗体类是被授权管理上的：



之所以这么做，主要考虑的是在一般应用场景下，我们不需要将权限控制细化具体到某个属性的读写控制上，只要控制到界面的菜单和功能按钮即可。

5.3.3 与业务规则的关联

具体方法见“03.Addin 工具使用方法”中“构建读/写属性授权规则类”、“构建执行过程授权规则类”章节，摘录如下：



5.3.4 要点注意

通过上述这两种方式，Phenix丶可实现全面的业务对象公共接口授权，并通过界面控制层上的ReadWriteAuthorization 组件、ExecuteAuthorization 组件体现出这些管控能力。

需注意的是，业务对象公共接口的命名在配置过后，如无必要不得修改。否则，不仅要在 IDE 中彻底检索、替换界面组件代码，还要通过 Phenix.Services.Host.exe 重新注册程序集。

5.4 可扩展性

毋庸置疑，Phenix丶的权限控制具备了完全的可扩展性，所以下文讨论的是授权的业务规则如何做到可扩展性。

我们知道，应用系统的可扩展性是设计出来的，它不是一蹴而就的，而是需要随着你对业务领域理解的深入而不断重构获得；同时，在系统演化过程中时刻准备着保持对复杂性的关注，不能盲目为了扩展而扩展，把系统搞得极其复杂来应对几乎不可能发生的变化（所谓“过度开发”）。

一套软件产品，我们要将其业务规则区分为基础规则和扩展规则两大类：基础规则的逻辑结构是固化的、不便替换的；而扩展规则是可以在产品实施时做二次开发的插拔、替换。

5.4.1 基础规则的可扩展性

基础规则的扩展方法，是从业务逻辑的算法中抽取出参数并将它重构为规则对象的可配置化属性。具体如何实现？请见“03.Addin 工具使用方法”的“可配置化对象”章节。

5.4.2 扩展规则的可插拔性

业务基类 Phenix.Business.BusinessBase<T>为授权规则的可扩展性提供了以下的静态事件：

```
/// <summary>
/// 授权规则注册中事件
/// 可配置化：当应用程序初始化时，可通过本事件来添加额外的授权规则库
/// </summary>
public static event AuthorizationRuleRegisteringEventHandler AuthorizationRuleRegistering;
```

以“执行解锁授权规则”类为例，黄底代码部分是为方便动态注册授权规则而额外编写的：

```
/// <summary>
/// 执行解锁授权规则
/// </summary>
public class ProcessLockUnlockExecuteAuthorizationRule :
Phenix.Business.Rules.ExecuteAuthorizationRule
{
```

```
    /// <summary>
    /// 初始化
    /// </summary>
    public ProcessLockUnlockExecuteAuthorizationRule()
      : base(ProcessLock.UnlockMethod) { }

    #region Register

    /// <summary>
    /// 注册授权规则
    /// </summary>
    public static void Register()
    {
      ProcessLock.AuthorizationRuleRegistering += new
Phenix.Business.Core.AuthorizationRuleRegisteringEventHandler(ProcessLock_AuthorizationRuleRegisterin
g);
    }

    private static void
ProcessLock_AuthorizationRuleRegistering(Phenix.Business.Rules.AuthorizationRules authorizationRules)
    {
      authorizationRules.AddRule(new ProcessLockUnlockExecuteAuthorizationRule());
    }

    #endregion

    /// <summary>
    /// 执行
    /// </summary>
    protected override void Execute(Phenix.Business.Rules.AuthorizationContext context)
    {
      context.HasPermission =
        Phenix.Business.Security.UserPrincipal.User.Identity.UserNumber ==
((ProcessLock)context.Target).Usernumber;
    }
  }
```

这样，只要在应用程序初始化时，调用如下代码，就可以将上述规则添加进业务类的授权规则库中：

```
    ProcessLockUnlockExecuteAuthorizationRule.Register();
```

那么，以下写死在业务类中的代码（黄底代码）就可删除掉了：

```
    /// <summary>
```

```
/// 注册授权规则
/// </summary>
protected override void AddAuthorizationRules()
{
  AuthorizationRules.AddRule(new ProcessLockUnlockExecuteAuthorizationRule());
  base.AddAuthorizationRules();
}
```

5.4.3 具体应用

软件产品的业务逻辑组件具体形式可分为基础程序集和扩展程序集，业务类及其基础业务规则类被编译为一个基础程序集、而外围的扩展规则类被编译成各个独立的扩展程序集。将这些程序集发布给实施团队，在产品的实施阶段就可以按需组装成实际的应用系统，实施团队也可以自行在此基础上二次开发规则库，动态注册到应用系统里。

5.5 业务类上与授权验证规则相关的接口

5.5.1 Phenix.Business.BusinessBase<T>提供的功能

属性	说明	备注
CanFetch	是否允许Fetch	
CanCreate	是否允许 Create	
CanEdit	是否允许 Edit	
CanDelete	是否允许 Delete	

方法	说明	参数类型	参数	参数说明
CanReadProperty	属性可读	Csla.Core.IPropertyInfo	property	属性信息
		bool	throwOnFalse	不可读则抛出异常
		bool	returns	
CanWriteProperty	属性可写	Csla.Core.IPropertyInfo	property	属性信息
		bool	throwOnFalse	不可写则抛出异常
		bool	returns	
CanExecuteMethod	过程可执行	Csla.Core.IMemberInfo	method	过程信息
		bool	throwOnFalse	不可执行则抛出异常
		params object[]	arguments	参数
		bool	returns	

5.5.2 Phenix.Business.BusinessListBase<T, TBusiness>提供的功能

属性	说明	备注
CanFatch	是否允许 Fetch	
CanEdit	是否允许 Edit	

方法	说明	参数类型	参数	参数说明
CanExecuteMethod	过程可执行	Csla.Core.IMemberInfo	method	过程信息
		bool	throwOnFalse	不可执行则抛出异常
		params object[]	arguments	参数
		bool	returns	

5.5.3 Phenix.Business.CommandBase<T>提供的功能

属性	说明	备注
CanExecute	是否允许 Execute	

5.5.4 要点注意

针对具体的业务类，上述功能是否有效，取决于该业务类是否已被配置为“可被授权”（见前文“与角色的关联”章节）。

5.6 前提条件

上述验证方法是否能得到正确的结果，是建立在业务类是否按照以下的编码规范要求进行设计。

5.6.1 注册需纳入到授权验证规则的属性

```
        /// <summary>
        /// 名称
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<string> NameProperty =
RegisterProperty<string>(c => c.Name);
        [Phenix.Core.Mapping.FieldUniqueAttribute("I_SSF_NAME")]
        [Phenix.Core.Mapping.FieldRuleAttribute(StringOnImeMode = true, PinyinCodeProperty = "Code")]
        [Phenix.Core.Mapping.Field(FriendlyName = "名称", Alias = "SSF_NAME", TableName = "SYS_SHIFT",
ColumnName = "SSF_NAME", NeedUpdate = true, InLookUpColumn = true, InLookUpColumnDisplay = true)]
        private string _name;
```

```
        /// <summary>
        /// 名称
        /// </summary>
        [System.ComponentModel.DisplayName("名称")]
        public string Name
        {
            get { return GetProperty(NameProperty, _name); }
            set { SetProperty(NameProperty, ref _name, value); }
        }
```

5.6.2 注册需纳入到授权验证规则的过程

```
    /// <summary>
    /// 解锁
    /// </summary>
    public static readonly Phenix.Business.MethodInfo UnlockMethod = RegisterMethod(c => c.Unlock());
    /// <summary>
    /// 解锁
    /// </summary>
    [Phenix.Core.Mapping.Method(FriendlyName = "解锁")]
    public void Unlock()
    {
      CanExecuteMethod(UnlockMethod, true);
      //以下代码为具体的业务逻辑
    }
```

5.7 绕开属性读写的授权验证

属性读写授权规则的验证，对系统性能是有一定损耗的，特别是在一组业务对象集合中逐个读取业务对象属性等应用场景，效果较为明显。不过，这些场景往往不是界面交互，而是业务逻辑层中较为复杂的逻辑运算过程。所以，在这种无需验证授权规则的应用场景下，可以暂时屏蔽掉属性读写的授权验证，以提高系统性能。

只要 using 业务对象上的 BypassPropertyChecks 属性，框定在 using(BypassPropertyChecks){ } 内的属性读写，就可以绕开授权验证：

```
  /// <summary>
  /// 过程锁
  /// </summary>
  [Serializable]
  public class ProcessLockReadOnly : ProcessLock<ProcessLockReadOnly>
```

```
  {
    /// <summary>
    /// 拷贝
    /// </summary>
    protected ProcessLockReadOnly Copy()
    {
      ProcessLockReadOnly result = new ProcessLockReadOnly();
      using (BypassPropertyChecks)
      {
        result.Name = Name;
        result.Allowexecute = Allowexecute;
        result.Time = Time;
      }
      return result;
    }
  }
```

需注意的是，BypassPropertyChecks 属性是 protected 的，以防范被无故滥用。所以，必须在业务类里面使用它，通过具体函数暴露它应用到的业务功能。