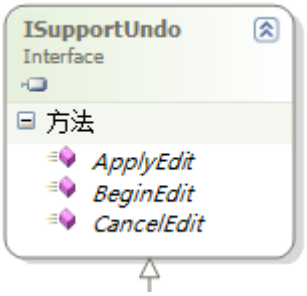
11 业务对象生命周期及其状态

11.5 Edit 业务对象及回滚机制

Phenix.Business.BusinessBase<T>和 Phenix.Business.BusinessListBase<T, TBusiness>都实现了回滚机制的接口 Csla.Core.ISupportUndo，提供了对多级撤销功能公有的和多态的访问，被界面控制对象或其他业务对象访问。

为规范编码及保证代码质量，Phenix丷规定了应用系统的开发，必须且仅以本接口提供的函数对业务对象、业务对象集合进行编辑操作。

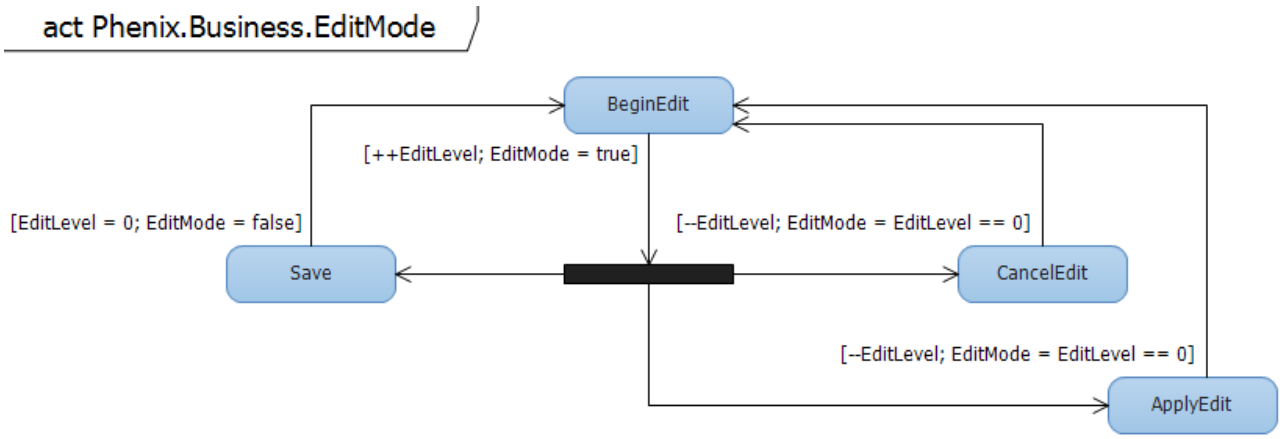


另外，业务对象、业务对象集合还提供了如下属性，以明晰当前是否处于编辑状态及编辑层级：

```
/// <summary>
/// 在编辑状态
/// </summary>
bool EditMode { get; }

/// <summary>
/// 编辑层级
/// </summary>
int EditLevel { get; }
```

它们与编辑操作函数之间的互动关系为：



鉴于回滚机制是影响到从业务对象的整个业务结构，所以在复杂的业务结构设计开发中思路需清晰，要明白业务对象之间的层级关系。一旦某个业务对象或业务对象集合启动了编辑模式（登记了快照，即被调用了 BeginEdit()函数），那么：

- 只有这个（登记了快照的）对象才能执行后续的操作：Save()、CancelEdit()、ApplyEdit()，否则 CSLA 会抛出 Csla.Core.UndoException 异常，提示信息大致是“编辑层级不属于第 X 层”；
- 之后再添加进这个业务结构里的 New 业务对象、Fetch 业务对象，都会被自动置为相同的 EditMode 与 EditLevel，纳入到回滚机制中；

总之，在整个编辑操作的生命周期中，Phenix丶会维持整个业务结构中的所有业务对象都处于相同的 EditLevel。

11.5.1案例

11.5.1.1 演示如何使用回滚机制

```
WorkingProcessLock.BeginEdit();
try
{
  WorkingProcessLock.Name = null;
  WorkingProcessLock.Save();
}
catch
{
  WorkingProcessLock.CancelEdit();
  throw;
}
```

11.5.1.2 演示回滚机制中当前属性值及各编辑层级中的属性值之间的变化过程

```
object s;
int editlevel;
WorkingProcessLock.Usernumber = "A";
WorkingProcessLock.BeginEdit();
WorkingProcessLock.Usernumber = "B";
editlevel = ((Phenix.Business.IBusiness)WorkingProcessLock).EditLevel; //=1
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty); //=最原始值
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 0); //=最原始值
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 1); //=A
WorkingProcessLock.Usernumber = "C";
WorkingProcessLock.BeginEdit();
WorkingProcessLock.Usernumber = "D";
editlevel = ((Phenix.Business.IBusiness)WorkingProcessLock).EditLevel; //=2
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty); //=最原始值
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 0); //=最原始值
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 1); //=A
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 2); //=C
WorkingProcessLock.Usernumber = "E";
WorkingProcessLock.CancelEdit();
s = WorkingProcessLock.Usernumber; //=C
editlevel = ((Phenix.Business.IBusiness)WorkingProcessLock).EditLevel; //=1
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty); //=最原始值
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 0); //=最原始值
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 1); //=A
WorkingProcessLock.CancelEdit();
s = WorkingProcessLock.Usernumber; //=A
editlevel = ((Phenix.Business.IBusiness)WorkingProcessLock).EditLevel; //=0
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty); //=最原始值
s = WorkingProcessLock.GetOldValue(ProcessLock.UsernumberProperty, 0); //=最原始值
```

11.5.2业务数据属性的状态

上述代码中用到了业务类中与业务数据属性状态相关的函数，下面详细罗列。

11.5.2.1 取编辑层级上的旧属性值

```
/// <summary>
/// 取编辑层级上的旧属性值
/// </summary>
/// <param name="propertyInfo">属性信息</param>
/// <param name="editLevel">编辑层级，小于等于0则等同于取最原始值</param>
public object GetOldValue(Phenix.Core.Mapping.IPropertyInfo propertyInfo, int editLevel)
```

11.5.2.2 判断属性是否已是脏属性

Phenix.Business.BusinessBase<T>提供有以下函数：

```
/// <summary>
/// 是否脏属性?(如果写入时的新值与旧值相同则认为未被赋值过)
/// ignoreCompare = false
/// </summary>
/// <param name="propertyInfo">属性信息</param>
public bool IsDirtyProperty(Phenix.Core.Mapping.IPropertyInfo propertyInfo)

/// <summary>
/// 是否脏属性? (如果写入时的新值与旧值相同则认为未被赋值过)
/// </summary>
/// <param name="propertyInfo">属性信息</param>
/// <param name="ignoreCompare">忽略比较新旧值</param>
public bool IsDirtyProperty(Phenix.Core.Mapping.IPropertyInfo propertyInfo, bool ignoreCompare)
```

至于整个业务对象是否有脏属性，可调取其属性值：

属性	说明	备注
PropertyValueChanged	属性值被赋值过	如果写入时的新值与旧值相同则认为未被赋值过

11.5.3不参与多级撤销

不是所有的业务场景都必须应用到多级撤销功能，毕竟这套机制是蛮耗内存的，对于大数据的处理也蛮耗时的。为此，Phenix丫在业务对象、业务对象集合里提供了如下属性，用来申明其自身是不参与的，且其 Detail 对象也不会参与：

```
/// <summary>
/// 不参与多级撤销并阻断Detail对象的多级撤销
/// 缺省为 false
/// </summary>
protected virtual bool NotUndoable { get; }
```

默认是 false，如果我们在自己的业务对象、业务对象集合类中覆写这个属性并返回 true，就可以申明不再参与多级撤销功能。此时的 BeginEdit()函数是不会再起作用了，而 CancelEdit()函数的作用变成了将业务对象内容恢复到之前 Fetch 来的原始值,ApplyEdit()函数的作用变成了直接将业务对象当前的内容固化为原始值（相当于是 Fetch 来的）。还有，即使不覆写 NotUndoable 属性，在未调用

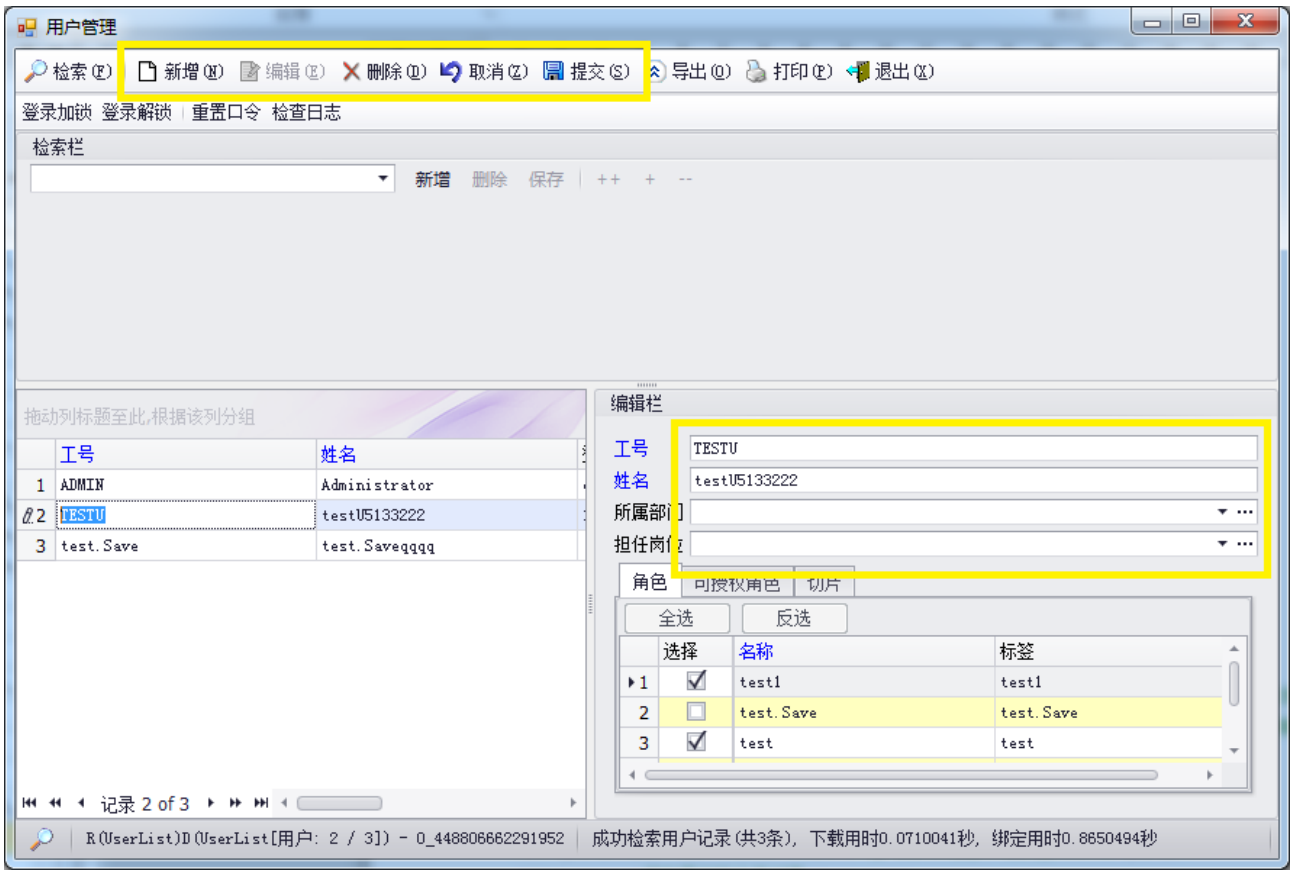
BeginEdit()函数的前提下，CancelEdit()函数和 ApplyEdit()函数也是可以被调用的，效果同上。

只要 root 对象的 NotUndoable 属性被覆写为返回 True，BarManager 组件的界面逻辑将会有微小的变化，即点击检索功能按钮后，将直接进入到可编辑状态，编辑功能按钮已无必要（将不可操作），而取消功能按钮、提交功能按钮都始终保持着可操作状态。

以下我们利用 Phenix.Security.Windows.UserManage 工程来做一下演示，这只要向 root 对象即 UserList 类中添加如下代码：

```
/// <summary>
/// 用户清单
/// </summary>
[Serializable]
public class UserList : Phenix.Business.BusinessListBase<UserList, User>
{
  /// <summary>
  /// 不参与回滚机制并阻断Detail的回滚处理
  /// 缺省为 false
  /// </summary>
  [System.ComponentModel.Browsable(false)]
  [System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
  protected override bool NotUndoable
  {
    get { return true; }
  }
}
```

编译后运行 UserManage，点击检索功能按钮后：



就会发现有几处很明显的变化：

- 编辑功能按钮，始终是不能操作的；
- 取消功能按钮，始终可以操作：新增后删除可以被恢复（内容维持现状），修改内容后可以被恢复为初始值；
- 提交功能按钮，始终可以操作；
- 业务对象的编辑栏，始终都可以输入内容；

界面逻辑的演示就是这样。在此基础上，我们可以自行添加上基于自己业务规则的界面逻辑控制（见“07.ExecuteAuthorization 组件”章节）。