



课程编号:

自定义单据（高级）

U9客户化开发及管理部

杨旭

2010年 3月25日

主题

- 课程概述
- 后台组件编程基础
- 前台组件编程基础
- 专题研究

主题

- 课程概述
- 后台组件编程基础
- 前台组件编程基础
- 专题研究

课程概述

课程目标

在《自定义单据（初级）》课程基础之上，理解前后台编程接口和编程方法，深入挖掘单据开发功能

课程内容

内容涵盖前后台模型开发，前后台编程基础；

听课基础

自定义单据（初级），OQL语言，C#项目开发

考查目标

主题

- 课程概述
- 后台组件编程基础
- 前台组件编程基础
- 专题研究

实体对外结构的基础组成

- ❏ 实体的Key（强类型的EntityKey）
- ❏ 实体的查询类Finder
- ❏ 实体的强类型集合EntityList
- ❏ 实体的资源属性和强类型访问属性的辅助类EntityRes
- ❏ 实体的OriginalData
- ❏ 实体的状态SysState

Session

Session的概念

- ➡ 在现在UBF中，Session的本意是work unit,即持久层的一个边界，非常轻，主要用作批量提交，并标识这次批量提交的边界，不涉及到事务等概念。
- ➡ 当前ISession可以通过Session的Current属性获得，每调用一次Session的Open方法，Current属性都会被更新

Session的使用

```
using (ISession session = Session.Open ()) {  
    ...  
    session.Commit();           //提交更改，但只是Session范围内的修改更新  
}
```


Session嵌套

 当发生session嵌套的情况时，每次提交都是真正提交

 示例：

```
using (ISession session1 = Session.Open ()) {  
    ...  
    using (ISession session2 = Session.Open ()) {  
        ...  
        session2.Commit();    //提交更改，但只是Session2范围内的修改更新  
    }  
    session1.Commit();        //提交更改，只处理Session1的修改更新  
}
```


实体新建

 新建实体实体的操作分为主实体的新建和非主实体的新建，如果是主实体新建，用不带参数的Create方法，如果是非主实体的，用带上级实例参数的Create方法。

 举例：


```
using (ISession session = Session.Open())
{
    A_Com1c1 objA = A_Com1c1.Create();
    objA.Code_A_Com1c1 = 1000;
    objA.Name_A_Com1c1 = "objA ";
    B_Com1c1 objB = B_Com1c1.Create(objA) //B为A的子实体
    objB.Code_B_Com1c1 = 1001;
    objB.Name_B_Com1c1 = "objB" ;
    session.Commit();
}
```

最好在Session
中创建

实体更新

```
using(ISession session = Session.Open ()) {  
    Customer obj = Customer.Finder.FindByID(id);  
    if(obj!=null)  
        obj.Name = "new name";  
    session.Modify(obj);  
    ... ..  
    session.Commit();  
}
```

实体删除

 对于组合关系，如果删除主实体，那么，主实体下面的所有子实体，孙实体会被级联删除，对于非主实体的删除，可以用子集合的RemoveAt和Remove（entity）方法,这2个方法等价，例如

```
using (ISession session = Session.Open())
{
    A_Com1cN objA = A_Com1cN.Finder.FindByID(aID);
    objA.B_Com1cN.RemoveAt(0);
    //或用下面的方式:
    B_Com1cN b = Com1cN.Finder.FindByID(bID);
    objA.B_Com1cN.Remove(b);
    session.Commit();
}
```

实体查询方式

- ❏ 实体的查询类Finder
- ❏ EntityDataQuery
- ❏ EntityQuery
- ❏ EntityViewQuery

实体查询

实体的查询类Finder

- ➡ 返回强类型的对象
- ➡ 常用方法：Find, FindByID, FindAll
- ➡ 举例

```
OqlParamList pList = new OqlParamList();  
string strOql = "Code=@Code";  
pList.Add(new OqlParam("Code", "11" ));  
Customer.EntityList list=Customer.Finder.FindAll(strOql, pList.ToArray());
```

实体查询

EntityDataQuery

➡ EntityDataQuery 主要用于希望返回结果是IDataReader, DataSet, 单值的情况

➡ 创建方式:

1、通过Entity对象:

```
EntityDataQuery q = Customer.Finder.CreateDataQuery();
```

2、通过EntityFullName:

```
EntityDataQuery q = new EntityDataQuery("UFIDA.U9.CB0.SCM.Customer.Customer");
```

```
EntityDataQuery q = new EntityDataQuery(Customer.EntityRes.BE_FullName);
```

➡ 返回DataSet: FindDataSet

1、全OQL方式

```
DataSet ds =
```

```
q.FindDataSet("select ID, Name, Code from UFIDA::U9::CB0::SCM::Customer::Customer");
```

2、条件OQL方式

```
DataSet ds = q.FindDataSet("ID>1");
```


实体查询

EntityDataQuery

➡ 返回IDataReader : FindDataReader

1、全OQL方式

```
IDataReader dr = q.FindDataReader ("select ID,Name,Code from  
UFIDA::U9::CB0::SCM::Customer::Customer");
```

2、条件OQL方式

```
IDataReader dr = q.FindDataReader ( "ID>1");
```

➡ 返回单值:

```
q.FindValue("select Max(ID) where id > 111");
```

➡ 分页查询:

```
DataSet ds = q.FindDataSetByPage(1, 2, " ID > @ID and Name = @mmm order by ID");
```

实体查询

EntityQuery

- ➡ 主要用于希望返回的结果是实体（弱类型）
- ➡ 以强类型的方式使用Customer.Finder，其对外接口和功能与EntityQuery类似，只是Customer.Finder返回强类型的对象，EntityQuery返回弱类型的对象。
- ➡ 创建方式：

1、通过Entity对象：

```
EntityQuery eq = Customer.Finder.CreateQuery();
```

2、通过EntityFullName：

```
EntityQuery q = new EntityQuery("UFIDA.U9.CB0.SCM.Customer.Customer");
```

```
EntityQuery q = new EntityQuery(Customer.EntityRes.BE_FullName);
```

实体查询

EntityQuery

返回实体: FindByID和Find

FindByID , 先从缓存加载对象, 如果缓存没有, 则从数据库加载对象;

```
Customer a = Customer.Finder.FindByID(a.ID);
```

```
Customer a = (Customer)eq.FindByID(a.ID);
```

Find方式, 直接从数据库加载, 然后查缓存, 如果缓存有, 用缓存对象替换;

```
Customer.Finder.Find(“ID = @ID”);
```

```
eq.Find(“ID = @ID”);
```

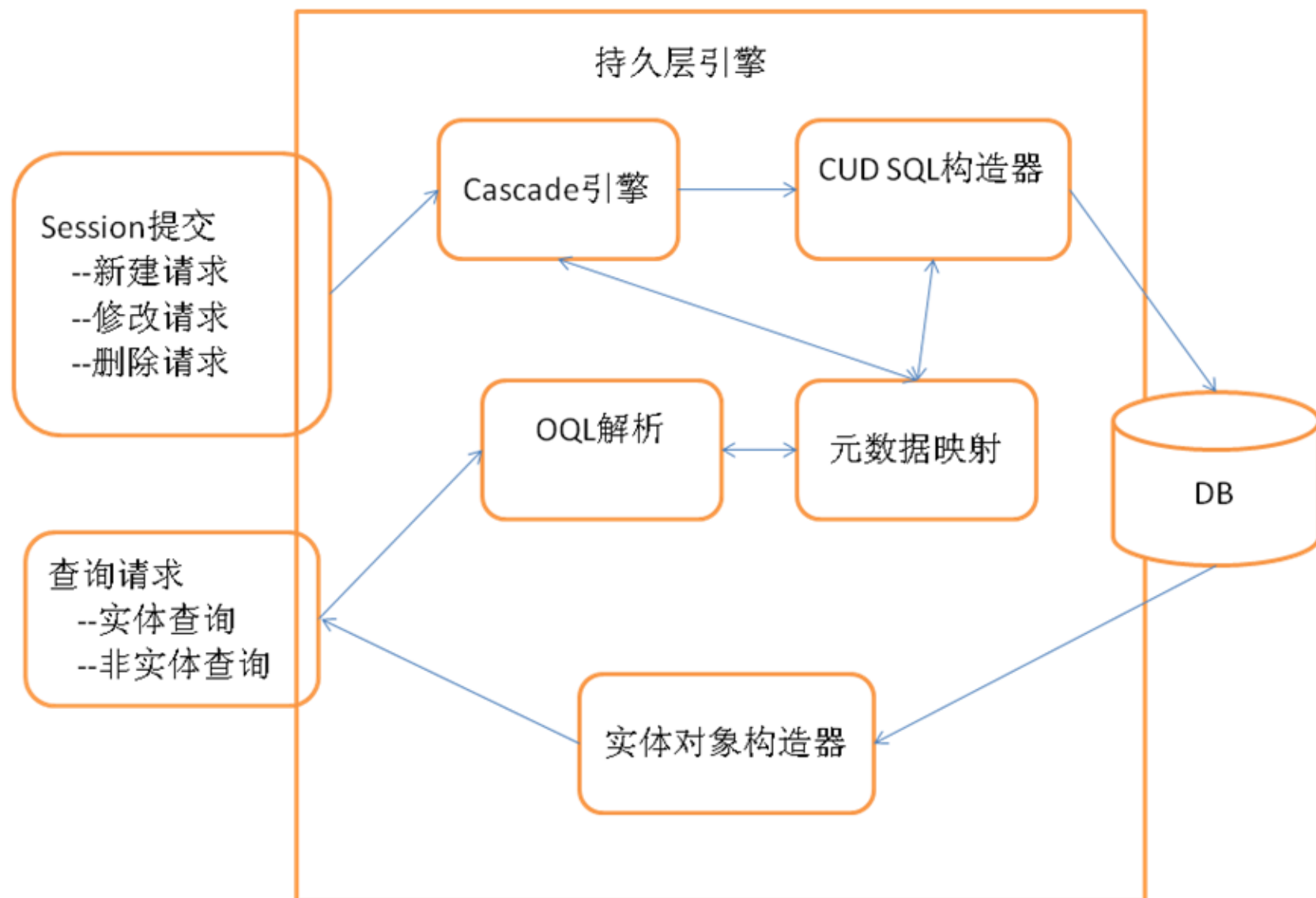
返回实体集合

条件OQL方式:

```
q = new EntityQuery(Customer.EntityRes.BE_FullName);
```

```
q.FindAll(“ID >1”);
```

实体操作→持久化引擎



实体资源

理解实体资源

实体资源的访问

//取实体ID的资源

```
string str = Customer.EntityRes.GetResource(Customer.EntityRes.ID);
```

//取实体全名的资源

```
str = Customer.EntityRes.GetResource(Customer.EntityRes.BE_Name);
```

//取枚举的资源

```
str = Customer.EntityRes.GetResource(Customer.EntityRes.Name);
```

```
str = Customer.EntityRes.GetResource(Customer.EntityRes.ExtEnum);
```

```
str = Customer.EntityRes.GetResource(Customer.EntityRes.ExtEnum_ExtNum1);
```

//取属性类型的资源

```
str = Customer.EntityRes.GetResource(Customer.EntityRes.PropertyType);
```

```
str =
```

```
Customer.EntityRes.GetResource(Customer.EntityRes.PropertyType_PropertyType  
1);
```

实体资源

为什么要使用实体资源类？

- ➡ 强类型的方式访问属性名；
- ➡ 减少手动的误差
- ➡ 提高程序健壮性
- ➡ 举例

```
EntityDataQuery q = new EntityDataQuery(Customer.EntityRes.BE_FullName);  
EntityDataQuery q = new EntityDataQuery(“UFIDA.U9.CB0.SCM.Customer.Customer”);  
q.Select(Customer.EntityRes.Name);  
q.Select(“Name”);  
q.Parameters.Add(new UFSOft.UBF.PL.OqlParam(“1”));  
object value = q.FindValue(Customer.EntityRes.Bargain + “=@” +  
Customer.EntityRes.Bargain);  
object value = q.FindValue(“Bargain =@ Bargain”);
```


实体校验

实体本身的业务校验

- ➡ 通过在实体Extend代码中的OnValidate方法中定义校验业务逻辑来实现

平台统一校验

- ➡ 非空校验
- ➡ 引用对象的存在性校验（关联实体，枚举对象）
- ➡ 敏感字段校验



事件扩展方法

实体的事件在子类上会自动生成事件扩展方法

`OnSetDefaultValue()`, `OnInserting()`, `OnInserted()`, `OnUpdating()`
`OnUpdated()`, `OnDeleting()`, `OnDeleted()`

这些方法提供了编写实体业务逻辑的入口

扩展方法的一些规则

-  实体的`OnSetDefaultValue`方法只关注对自身和下级的影响，如果有涉及到兄弟行或父实体的逻辑，需要移交到父实体的`OnSetDefaultValue`
-  在后事件（`OnInserted` / `OnUpdated` / `OnDeleted`）中不允许在当前session范围内对实体本身的属性再进行修改，但可以调用服务修改（慎用）

事件顺序

U9平台异常介绍

属性空效验异常：

AttrNotNullableException：显示为 **xxx 不能为空，请录入**；

例如：部门界面的[编码]字段不能为空，但是为空时保存了，则错误信息为‘**编码不能为空，请录入**’

权限异常：

AuthorityAddException：显示为**实体 xxx 未授权，不能新增记录**；

AuthorityDeleteException：**实体 xxx 未授权，不能删除记录**；

AuthorityUpdateException：**实体 xxx 未授权，不能修改记录**；

例如：**实体收款单未授权，不能新增记录**。

引用检查异常：

CascadeDeleteException：**当前xxx 已被使用，不能删除**；

CascadeModifyException：**当前xxx 已被使用，不能修改**；

例如：部门a已经被业务员甲引用，则删除部门a时，出错信息为‘**当前部门已被使用，不能删除**’。

业务主键重复异常：

SqlUniqueKeyException：**xxx 已存在，请重新录入**；

例如：部门档的主键是组织+编码，则出错信息为‘**组织+编码已存在，请重新录入**’。

引用有效性异常：

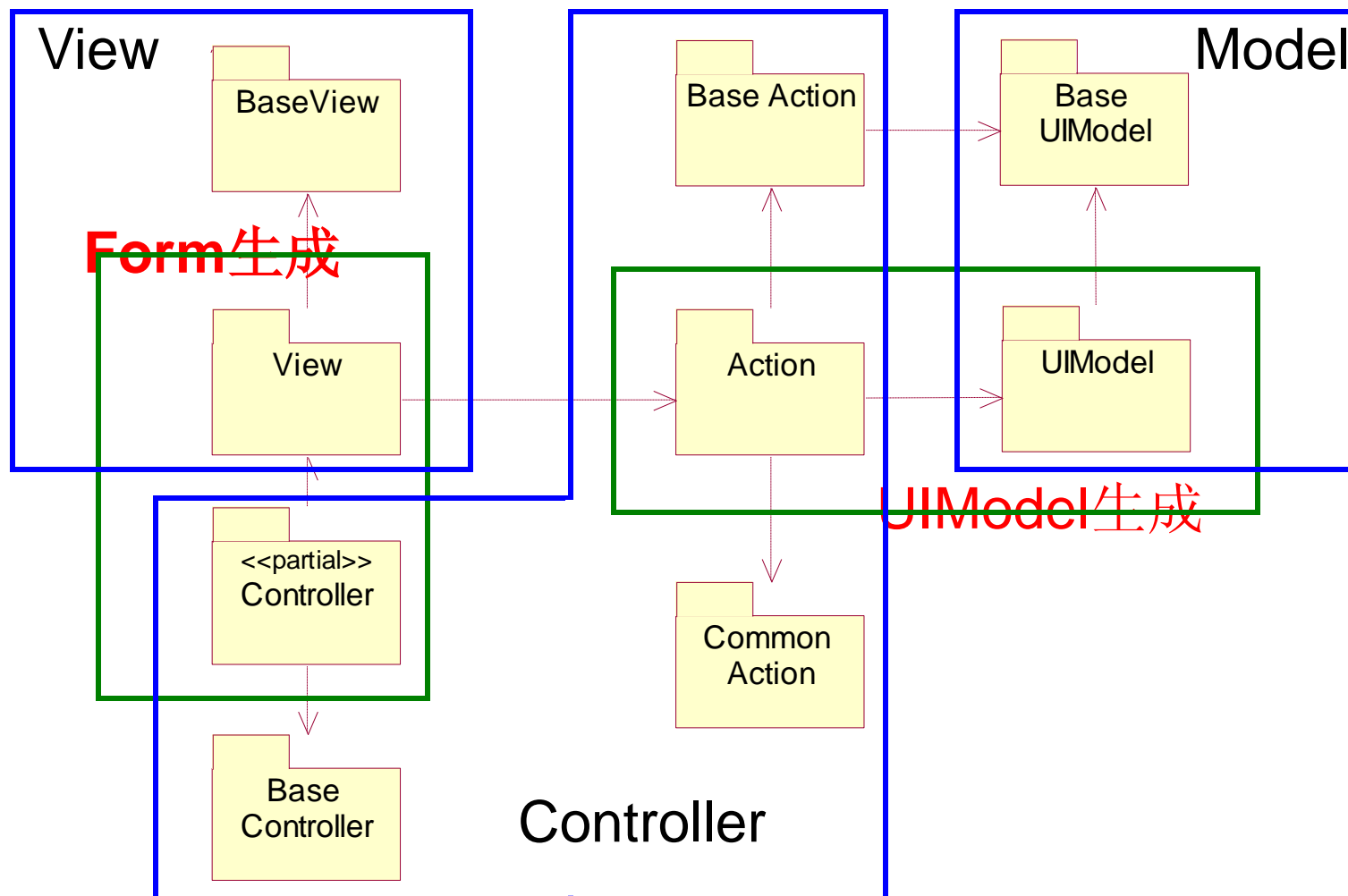
RefObjectInvalidException：**xxx 不存在，请重新录入**；

例如：业务员甲引用了部门a，但是在保存业务员甲时，部门a已经被删除了，这时出错信息是‘**部门不存在，请重新录入**’。

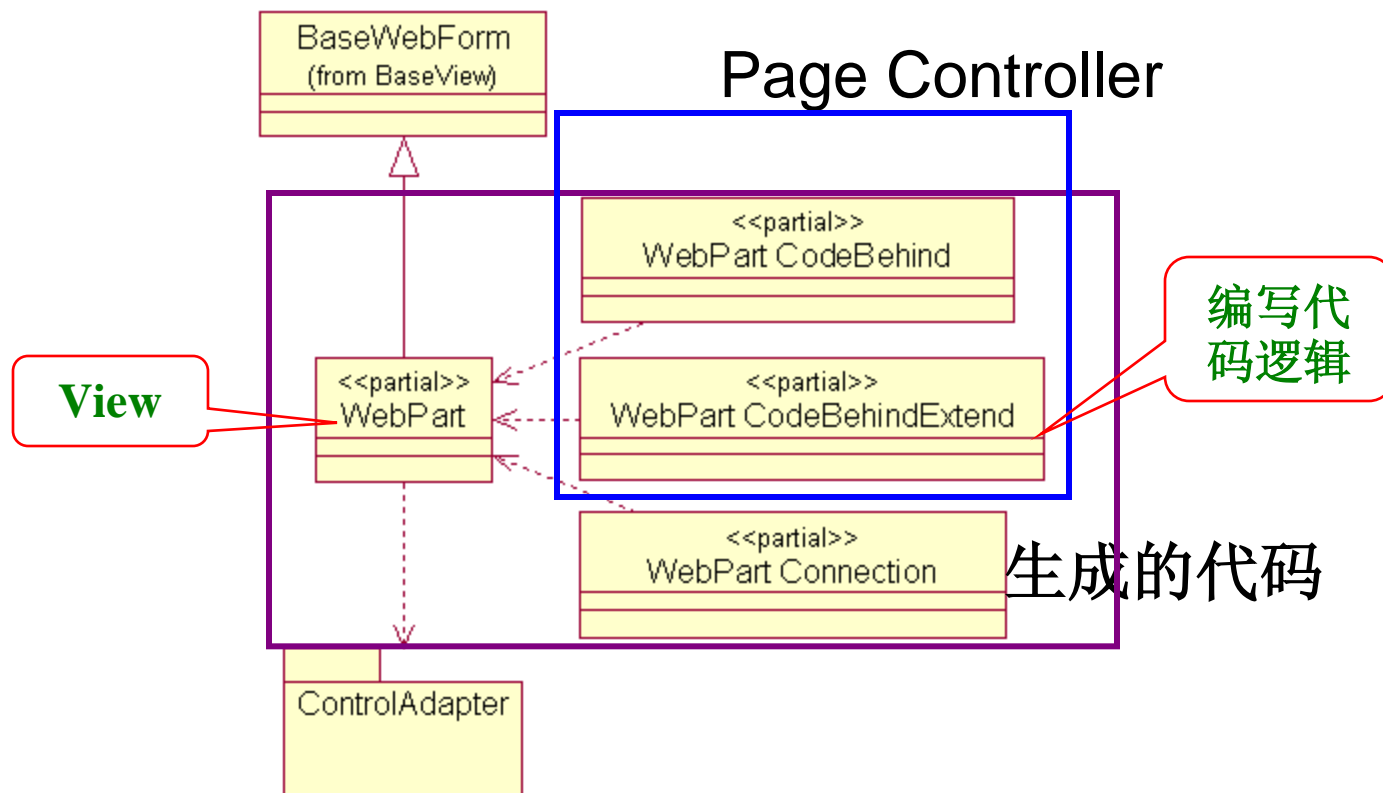
主题

- 课程概述
- 后台组件编程基础
- 前台组件编程基础
- 专题研究

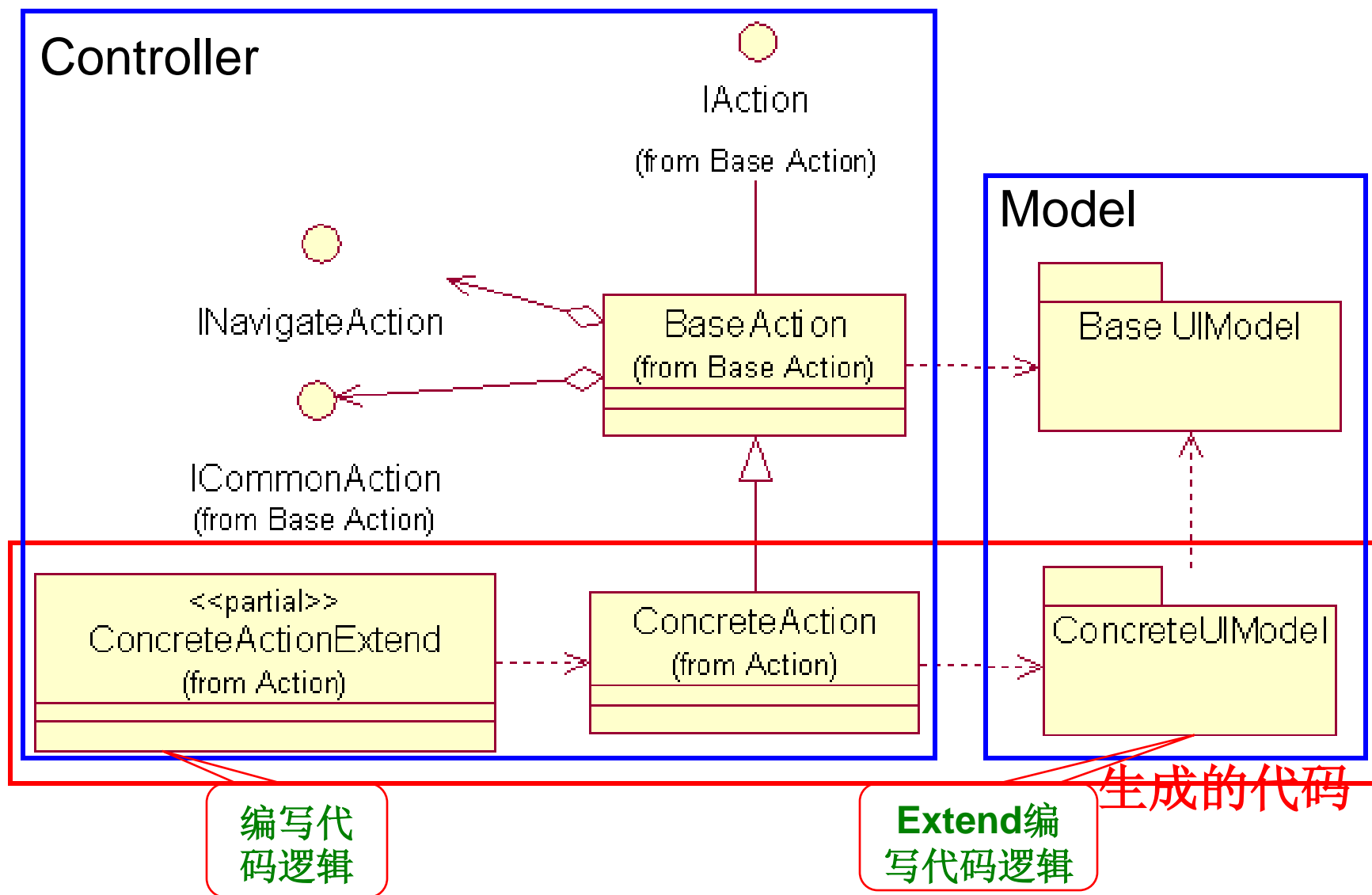
UI生成代码框架与MVC的关系



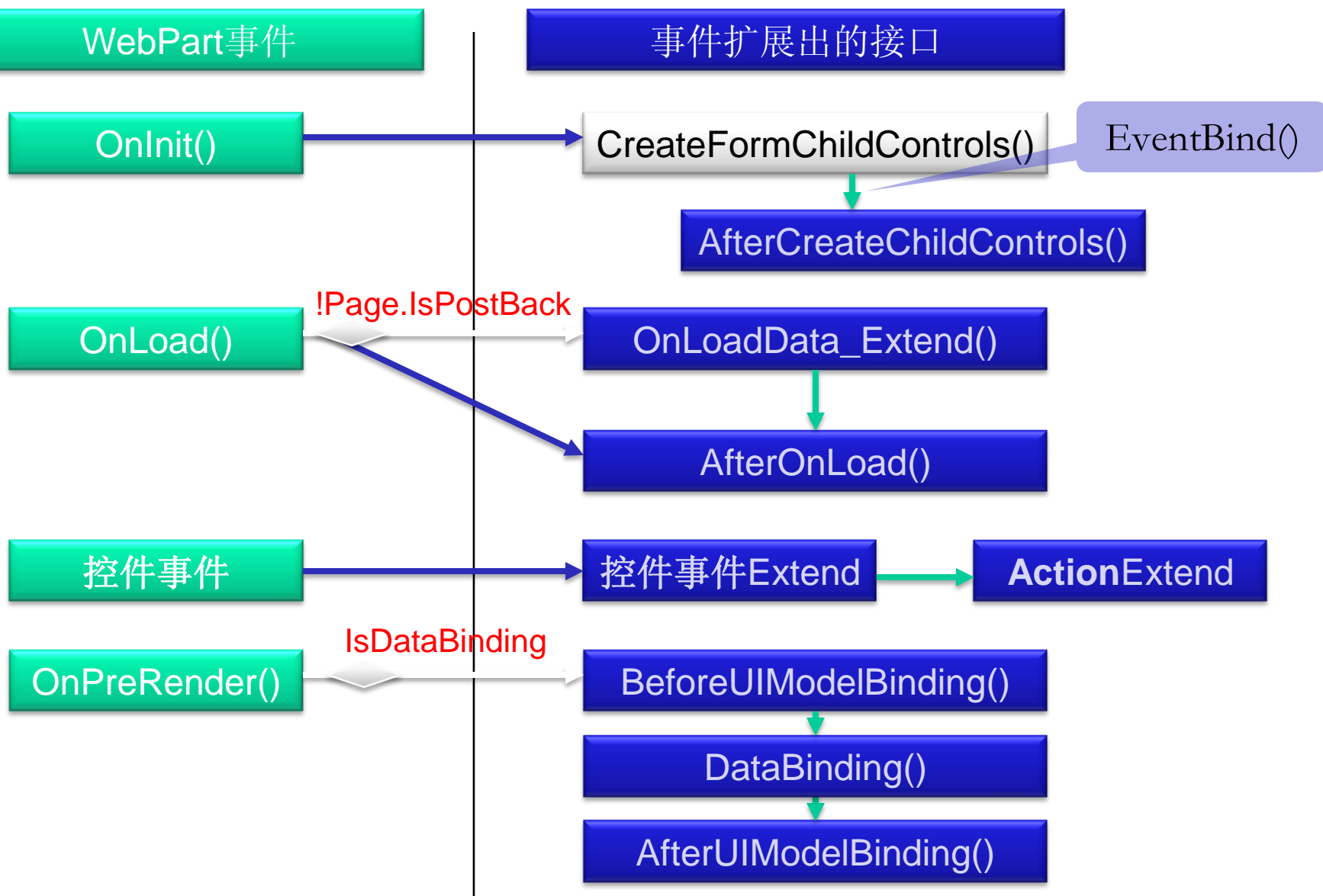
Form生成的代码框架结构



UIModel生成的代码框架结构



UI代码写入时机



UI代码写入时机

UIForm扩展方法	可写的逻辑
AfterCreateChildControls()	创建控件树，创建关联控件
OnLoadData_Extend()	非PostBack状态下创建控件树，加载数据
AfterOnLoad()	创建控件树的最后机会，也可以加载数据，修改控件属性
事件Extend()	事件响应业务逻辑
BeforeUIModelBinding()	修改模型数据最后的机会，否则不会反映到控件上，修改控件属性
AfterUIModelBinding()	修改控件属性
AfterEventBind()	绑定事件
ActionExtend()	默认通用逻辑，可扩展
ModelExtend()	对Model中视图的修改，如默认值

UI Model 编程

如何访问UIModel?

- ❏ 在Action基类中，有CurrentModel属性，可以在某个ActionExtend方法中这样访问UIModel: `this.CurrentModel`;
- ❏ 在WebPart的CodeBehind中访问`this.Model`，得到强类型的UIModel;
- ❏ 通过UIModel，即可以以强类型或弱类型的方式访问得到UIView和Link集合；可以访问UIModel的属性、UIModel下UIView的属性、以及UIField属性；
- ❏ UIView是数据的具体对象与接口：OQLString、DefaultFilter、对应EntityFullName、ParentLink、Childlinks、ParentView、FocusedRecord、FocusedIndex、FocusRecord、Records等；
- ❏ UIView可以强名访问字段，也可以弱类型访问字段；

UIModel数据收集与界面绑定

- ❏ 对于Action操作，可以在设计期设置该Action操作是否要收集数据和绑定数据；
- ❏ BaseWebForm实现IPart接口，有DataCollect（）、DataBanding（）方法和IsDataBanding属性，负责收集数据、界面绑定；
- ❏ 可以在WebPart的CodeBehindExtend代码的相应方法中编程，调用this.DataCollect（）、this.DataBanding（）等方法，完成数据收集与绑定；

操作UIModel

UIModel赋值、赋默认值和其他属性

➡ 在*ModelExtend.cs文件中AfterInitModel()方法中设置字段的defaultValue

➡ 在AfterInitModel()中完成对UIModel的默认值的设置：

```
public override void AfterInitModel()  
{
```

```
    //默认值
```

```
    this.Organization.FieldEffective_EffectiveDate.DefaultValue =  
PlatformContext.Current.DateTime;
```

```
    this.Organization.FieldIsLegacyOrg.DefaultValue = true;
```

```
➡ }
```

➡ 也可以在Action方法中直接对UIModel的字段赋值、赋默认值；方式同上；

操作UIModel

- ❏ 在卡片界面做“新增”等操作时，需要增加一条空Record，这条空记录与界面完成绑定，达到清空界面的效果，例如，清空主Form的Card容器界面的处理：

```
int index = this.MainView.FocusedIndex;  
IUIRecord newRecord = this.MainView.NewUIRecord();  
if(this.MainView.RecordCount == 0 || index < 0)  
    index = 0;  
this.MainView.Records.Insert(index, newRecord);  
this.MainView.FocusedRecord = newRecord;
```

操作UIModel

UIView与Record的处理:

```
//UIView从DataTable、UIView中复制
this.MainView.FromDataTable(DataTable table);
this.MainView.FromUIView(UIView view);
//复制到DataTable
this.MainView.ToDataTable(DataTable table);
//UIRecord与DataRow复制
UIRecord.FromDataRow(DataRow dataRow);
UIRecord.ToDataRow(DataRow dataRow);
//获取焦点记录索引
int index = this.MainView.FocusedIndex;
//获取焦点记录
IUIRecord record = this.MainView.FocusedRecord;
//设置焦点记录
this.MainView.FocusedRecord = record;
//获取被选择的记录集合
IUIRecordCollection records = this.MainView.SelectRecords;
```

操作UIModel

UIView与Record的处理:

//获取\设置焦点字段

this.MainView.FocusedRecord;{支持get、set}

//设置记录删除标志

UIRecord.Delete();

//移出记录

UIRecord.Remove();

//记录复制到UIRecord

UIRecord.CopyTo(UIRecord);

//UIRecord与EntityData复制

UIRecord.FromEntityData(object entityData);

UIRecord.WriteToEntityData(object entityData);

//UIRecord与DataRow复制

UIRecord.FromDataRow(DataRow dataRow);

UIRecord.ToDataRow(DataRow dataRow);

//判断该记录是否为焦点、被选择的记录

UIRecord.IsFocused;

UIRecord.IsSelected;

加载UIModel

修改视图中的过滤条件

直接修改Opath语句

修改条件语句值，增加Order By ， Group By语句

修改过滤条件中的参数

UIForm编程

UI控件

- ❏ 容器控件
- ❏ Label控件
- ❏ 文本框控件（含数字控件）
- ❏ 日期控件
- ❏ 参照控件
- ❏ 弹性域控件
- ❏ 枚举控件（含动态枚举）
- ❏ 关联控件
- ❏ 向导控件

数据绑定控件的查找方法

IPart接口中的查找控件的方法定义如下:

/// 通过名称得到抽象容器.

IUIViewBindingControl GetContainerByName(string name);

/// 通过UIView名称得到其绑定的容器控件.

IUIViewBindingControl GetViewContainerByUIViewName(string viewName);

/// 通过UIView绑定的容器, 和UIField的名字, 查找绑定UIField的控件

IUIFieldBindingControl GetUIFieldBindingControl(IUIViewBindingControl viewContainer, string fieldName);

/// 在指定容器中查找控件

IUFControl GetUFControlByName(IContainer container, string name);

- (1) 在ActionExtend中通过IPart的接口在指定容器中查找控件
- (2) 查找到指定控件后, 然后修改属性即可。

通用的CRUD的Action

 接口见

[\通用CRUD接口.txt](#)

 可以在Action中调用通用CRUD Action，不过目前都是在Action模板写好了通用CRUD Action调用，除非有特殊需求，否则还是使用现成的通用CRUD Action调用模板；

在状态条中显示信息

做法:

在UFSOft.UBF.UI.Engine.Runtime.dll位于UFSOft.UBF.UI.Engine命名空间的UIEngineHelper类下有静态方法:

```
/// <summary>  
/// 在客户端的状态栏中显示指定的信息  
/// </summary>  
/// <param name="page">用户控件页面</param>  
/// <param name="message">在客户端状态栏中要显示的信息</param>  
public static void ShowWindowStatus(IPart form, string message)
```

示例:

在ActionExtend中, 需要引用UFSOft.UBF.UI.Engine.Runtime.dll, 代码如下:

```
UIEngineHelper.ShowWindowStatus(this.CurrentPart, Resource);  
Resource代表资源信息;
```

在ActionExtend中调用专用BP Agent

做法:

在ActionExtend所在的项目中引用BP Agent的dll

请注意，按照我们的架构设计，在ActionExtend只能访问BP Agent

示例:

在ActionExtend所在的项目中引用BP Agent的dll

增加命名空间的引用，如

```
using UFSOFT.UBF.Alert.Proxy;
```

写调用BP Agent的代码，如下

```
StartAlertBPProxy bp = new StartAlertBPProxy();
```

```
bp.AlertKey = 。。。
```

```
bp.Do();
```

状态编程

在Action的基类中有接口:

//取当前的State.

```
public UIState CurrentState{ get; }
```

在ActionExtend中可以访问CurrentState, 并可以给CurrentState赋值和取值

示例:

比如说Form1弹出Form2, 如果传递比较大的数据时, 如传Form1把自己的整个UIModel传递给Form2, 可以使用CurrentState进行传递, 做法如下:

在Form1弹出Form2的ActionExtend中, 给CurrentState赋值:

```
this.CurrentState["MyUIModel"] = this.CurrentModel;
```

在Form2的某个ActionExtend中取出传递的UIModel:

```
IUIModel uiModel=this.CurrentState["MyUIModel"];
```

```
if(uiModel !=null)
```

```
{ ..... }
```

这种方式最后要自己手工清除掉缓存的数据, 比较麻烦。

推荐的做法: 在UIModel中添加一个属性, 来缓存, 系统会在适当的时机自动清除缓存。

弹出Form

弹出的对话框页面和被弹出的页面共享UIModel

比如说Form1弹出Form2，Form1和Form2共享UIModel，只要在Form1中调用如下代码就能弹出Form2，并且把Form1当前的UIModel传过去，假如在ActionExtend中写：

```
this.CurrentPart.ShowModalDialog(PartID, "Form2", "400", "500",  
this.CurrentPart.TaskId.ToString());
```

弹出的对话框页面和被弹出的页面不共享UIModel

比如说Form1弹出Form2，Form1和Form2不共享UIModel，只要在Form1中调用如下代码就能弹出Form2，Form2自会加载自己的UIModel，假如在ActionExtend中写：

```
this.CurrentPart.ShowModalDialog(PartID, "Form2", "400", "500", null);
```

页面导航（非封装模式）

导航页面和被导航页面共享UIModel

比如说Page1导航到Page2，Page1和Page2共享UIModel，只要在Page1中调用如下代码就能导航Page2，并且把Page1当前的UIModel传过去，假如在ActionExtend中写：

```
NameValueCollection query = new NameValueCollection();  
query.Add("ParentTaskID", this.CurrentPart.TaskId.ToString());  
this.CurrentPart.Navigate(PageURI, query);
```

导航页面和被导航页面不共享UIModel

比如说Page1导航到Page2，Page1和Page2不共享UIModel，只要在Page1中调用如下代码就能导航Page2，并且Page1自会加载自己的UIModel去，假如在ActionExtend中写：

```
this.CurrentPart.Navigate(PageURI, null);
```

页面导航（封装模式）

使用UFIDA.U9.UI.PDHelper.NavigateManager类完成Form的弹出和页面切换导航

与非封装的方式类似，省去了不必要的参数设置，如：

`NavigateManager.NavigatePage(IPart part, string pageId),`

`NavigateManager.NavigatePage(IPart part, string pageId , NaviteParamter param)`

主题

- 课程概述
- 后台组件编程基础
- 前台组件编程基础
- 专题研究

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- CallBack函数调用

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- Callback函数调用

弱类型实体

弱类型实体新建

```
/// <summary>
/// 根据名称建实体
/// </summary>
[Test]
public void TestCrtEntity()
{
    using (ISession s = Session.Open())
    {
        string fullname = "UFIDA.U9.CB0.SCM.Customer.Customer";
        //新建
        Entity entity = Entity.Create(fullname, null);
        entity.SetValue("Name", "1234567");
        //提交
        s.Commit();
    }
}
```

弱类型实体

弱类型实体删除

```
/// <summary>
/// 根据名称删除实体
/// </summary>
[Test]
public void TestDelEntity()
{
    using (ISession s = Session.Open())
    {
        string fullname = "UFIDA.U9.CB0.SCM.Customer.Customer";
        //查询
        EntityQuery query = new EntityQuery(fullname);
        Entity entity = query.FindByID ((long)111);
        //删除
        s.Remove(entity);
        //提交
        s.Commit();
    }
}
```

弱类型实体

弱类型实体查询修改

```
/// <summary>
/// 查询
/// </summary>
[Test]
public void FindAndMod()
{
    using (ISession s = Session.Open())
    {
        string fullname = "UFIDA.U9.CBO.SCM.Customer.Customer";
        //查询
        EntityQuery query = new EntityQuery(fullname);
        Entity entity = query.FindByID((long)111);
        //修改
        entity.SetValue("Name", "forTest");
        s.Modify(entity);
        //提交
        s.Commit();
    }
}
```

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- CallBack函数调用

异常信息的扩展

- ❏ 在一些特殊的业务情况下，可能不够详细，如果需要扩展平台的异常信息，可以在实体的Extend的文件中，重载基类方法DealException (Exception e)，然后捕获自己期待的异常，封装为规范的异常显示

- ❏ 示例：

```
public override void DealException(Exception e)
{
    if (e is RefObjectInvalidException)
    {
        RefObjectInvalidException re = (RefObjectInvalidException)e;
        throw new ApplicationExceptionBase("实体["+re.EntityFullName +"]的属性[" + re.AttrName + "] 引用对象不合法，值为[" + this.Values[re.AttrName] + "]");
    }
    base.DealException(e);
}
```

异常绑定

业务异常绑定到实体

➡ 需要给异常赋值 EntityID 和EntityFullName, 如:

➡ `XXXException.EntityID = EntityInstanceID;`

➡ `XXXException.EntityFullName = EntityFullName;`

业务异常绑定到属性

➡ 需要给异常赋EntityID, EntityFullName, AttributeName, 如:

➡ `XXXException.EntityID = EntityInstanceID;`

➡ `XXXException.EntityFullName = EntityFullName`

➡ `XXXException.AttributeName = AttributeName`

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- CallBack函数调用

实体缓存

实体缓存简介

为提高性能，避免频繁的访问数据库，后台引擎提供了实体缓存的机制

典型的特殊场景

开发在服务中嵌入部分存储过程以致修改了实体的数据，而这个数据在该服务的后续场景还要访问到被存储过程更新的新值，这是很特殊的情况，需要将2层缓存的数据全部清除，强制下次访问直接从数据库加载新值。

实体缓存

清除缓存中的实体

可以通过下面的方式访问2层缓存，引用UFSOft.UBF.PL.dll：

线程级数据缓存是：UFSOft.UBF.PL.Tool.[ConfigParm](#).EntityCache；

服务级引用缓存是：UFSOft.UBF.PL.Engine.[StateManager](#).EntityStateCache；

两者返回的对象都是[ICache](#)类型，[ICache](#)有下面2个对外方法可以清除缓存：

```
/// <summary>
/// 从Cache中删除指定的对象
/// </summary>
/// <param name="key"></param>
void Remove(object key);
/// <summary>
/// 清除所有的Cache内容
/// </summary>
void Flush();
```

注意

- 1、清缓存时这2级缓存都要清除。
- 2、`void Remove(object key)`，参数key是实体的EntityKey

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- CallBack函数调用

直接运行SQL和存储过程

 引用UFSOft.UBF.Util.DataAccess.dll

 DataAccessor类

➡ 方法RunSQL：直接运行SQL

➡ 方法RunSP：直接运行存储过程

 示例（ RunSQL ）

```
StringBuilder ExecuteSql = new StringBuilder(256);
ExecuteSql.Append(" DECLARE @StartSN bigint ; ");
ExecuteSql.Append(" DECLARE @EmpCount int ; ");
ExecuteSql.Append(" Set @EmpCount= " + this.EmpPayrolls.Count * 10 + " ");
ExecuteSql.Append(" EXEC [dbo].[AllocSerials] @EmpCount OUTPUT, @StartSN OUTPUT ;");
ExecuteSql.Append(InsertSql.ToString()); //其它Insert语句
ExecuteSql.Append(UpdatePayrollDocSQL.ToString()); //其他Update语句
ExecuteSql.Append(UpdatePayrollSupplySQL.ToString());
//进行提交SQL
DataAccessor.RunSQL(UFSOft.UBF.Sys.Database.DatabaseManager.GetCurrentConnection(),
ExecuteSql.ToString(), new DataParamList(), true);
```

直接运行SQL和存储过程

示例（RunSP）

```
DataParamList lst;
```

```
lst = new DataParamList();
```

```
lst.Add(DataParamFactory.CreateInput("PayrollDoc", this.ID.ToString(),  
System.Data.DbType.StringFixedLength));
```

```
lst.Add(DataParamFactory.CreateInput("PayrollCalculate", this.PayrollCalculate.ID.ToString(),  
System.Data.DbType.StringFixedLength));
```

```
DataAccessor.RunSP(DatabaseManager.GetCurrentConnection(), "PA_ReWritepayrollresult", lst);
```

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- CallBack函数调用

实体锁

- ❏ 离线悲观锁
- ❏ 加锁操作（读锁、写锁）
- ❏ 解锁操作
- ❏ 查询某个实体当前的锁状态
- ❏ 查询某个用户当前的加锁信息

实体锁的使用

引用UFSOft.UBF.Business.dll

主要接口

- UFSOft.UBF.Business.Lock.EntityLockManager提供实体的加锁、解锁、以及查询接口API。
- LockForWrite(entity.Key)//加写锁
- LockForRead(entity.Key)//加读锁
- Unlock(eLock)//解锁
- QueryLockOwners(entity.Key)//查询特定实体的加锁信息
- QueryLocks()//查询当前用户的加锁信息

实体锁的使用

使用样例

```
using (ISession session = Session.Open())
{
    //获得实体entity
    .....
    //对实体entity加写锁
    EntityLock eLock = EntityLockManager.LockForWrite(entity.Key);
    if(eLock!=null)
    {
        //如果加锁成功
        .....
        session.Commit();
        EntityLockManager.Unlock(eLock);// 对实体entity解锁
    }
}
```

加锁操作与解锁操作一定要成对的进行。或者使用using语句。

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- Callback函数调用

系统日志的配置说明

配置路径

Porta\UBFConfig\UFIDA.UBF.Log.Config

UBF的Log

目前UBF的LOG 是UBF自身定义的一套LOG接口，但其实现是基于LOG4NET的框架上包装而成，所以，目前基本配置及用法，和LOG4NET是一样的。

Log输出类型

LOG输出类型共有5种：从级别低到高依次为

Debug	输出调试信息
Info	输出提示信息
Warn	输出警告信息
Error	输出错误信息
Fatal	输出严重错误信息

配置文件中如果配置的为Info，那Info及Info以上级别的都会输出

系统日志的使用

添加代码

使用举例：

```
ILogger logger = LogManager.GetLogger(typeof(OrderDoc));  
if (logger.IsInfoEnabled)  
    logger.Info(“Info级别的信息”);  
if (logger.IsDebugEnabled)  
    logger.Debug(“Debug级别的信息”);
```

使用Is**Enabled的方式主要用于，当logger的信息是经过运算或者有一定的处理消耗，那可以用该判断来避免不输出Log时，这些方法的无谓的消耗。

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- Callback函数调用

UI界面调整

UI界面调整的意义

剖析界面布局

-  容器
-  功能区
-  工具栏

布局尺寸的计算

流式布局

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- Callback函数调用

实体扩展字段

 又称为描述性弹性域

 描述性弹性域包含30个公共扩展字段（全局段）和30个实体私有扩展字段（全局段和上下文相关段）

 实现：

 后台：

- 添加“实体扩展字段”

 前台：

- 卡片处理
- Grid处理

描述性弹性域

卡片处理

1)、拖一个弹性域控件，并设置其类型为**Description**。

2)、在AfterCreateChildControls()里面调用：

```
FlexFieldHelper.SetDescFlexField(this.FlexFieldPicker0, this.Model.Territory);
```

其中：`this.FlexFieldPicker0`为要设置的描述性弹性域控件，`this.Model.Territory`为描述性弹性域所在的View，把红色的**Territory**换成你的View就可以了。

Grid处理

1)、在Grid的末尾加一文本列，不需要绑定任何字段。

2)、在AfterCreateChildControls()里面调用：

```
FlexFieldHelper.SetDescFlexField(this.DataGrid3, 2);
```

其中：`this.DataGrid3`为要设置的描述性弹性域的Grid，2为描述性弹性域列（即上面添加的文本列）在Grid中的列索引。

注意事项

如果一个Form只有一处用到描述性弹性域，可以使用上面面的方法，但是如果一个Form有多处用到描述性弹性域（包括卡片和列表），请一定用下面的方式，提高效率：

```
FlexFieldHelper.SetDescFlexField(DescFlexFieldParameter[] parameters);
```

其中parameters为整个Form上所有要设置的描述性弹性域的相关参数的数组（包括卡片和列表），new DescFlexFieldParameter的时候请使用你想要的构造函数构造实例。

专题目录

- 弱类型实体
- 异常扩展、异常绑定
- 实体缓存
- 直接运行SQL和存储过程
- 实体锁
- 系统日志
- UI界面调整
- 实体扩展字段
- CallBack函数调用

CallBack函数调用

CallBack VS PostBack

CallBack控件无刷新，PostBack是回发到服务端进行处理，页面要刷新

示例：

在AfterCreateChildControls中注册CallBack函数，函数定义如下

```
private void SchemaCallBack ()
{
    AssociationControl objectTypeASC = new AssociationControl();//交互关联控件实例
    objectTypeASC.SourceServerControl = this.QCSchemaHead172;//触发源控件
    objectTypeASC.SourceControl.EventName = "onchange";触发事件
    ClientCallBackFrm CBF = new ClientCallBackFrm();
    CBF.DoCustomerAction += new ClientCallBackFrm.ActionCustomer( CBF_DoCustomerAction_SetRef );//定义服务器端的处理方法
    CBF.ParameterControls.Add( this.QCSchemaHead172 );//添加传送到服务器端的控件值
    CBF.Add( objectTypeASC );
}
object CBF_DoCustomerAction_SetRef (CustomerActionEventArgs args)
{
    UFWebClientTextBoxAdapter SrcRef = new UFWebClientTextBoxAdapter( this.Version0 );
    UFIDA.U9.CBO.QCBP.Proxy.GetSchemaVersionProxy tmpproxy = new UFIDA.U9.CBO.QCBP.Proxy.GetSchemaVersionProxy();
    tmpproxy.QCSchemaID = long.Parse( args.ArgsHash[this.QCSchemaHead172.ClientID].ToString() );//获取客户端控件值
    SrcRef.Value = tmpproxy.Do();//赋值
    args.ArgsResult.Add( SrcRef.ClientInstanceWithValue );//生成前台JS脚本更新控件值
    return args;
}
```

主题

- 课程概述
- 后台组件编程基础
- 前台组件编程基础
- 专题研究



用友大学

学以致用 信而益友