Microsoft Composition Application Block（CAB）源码分析

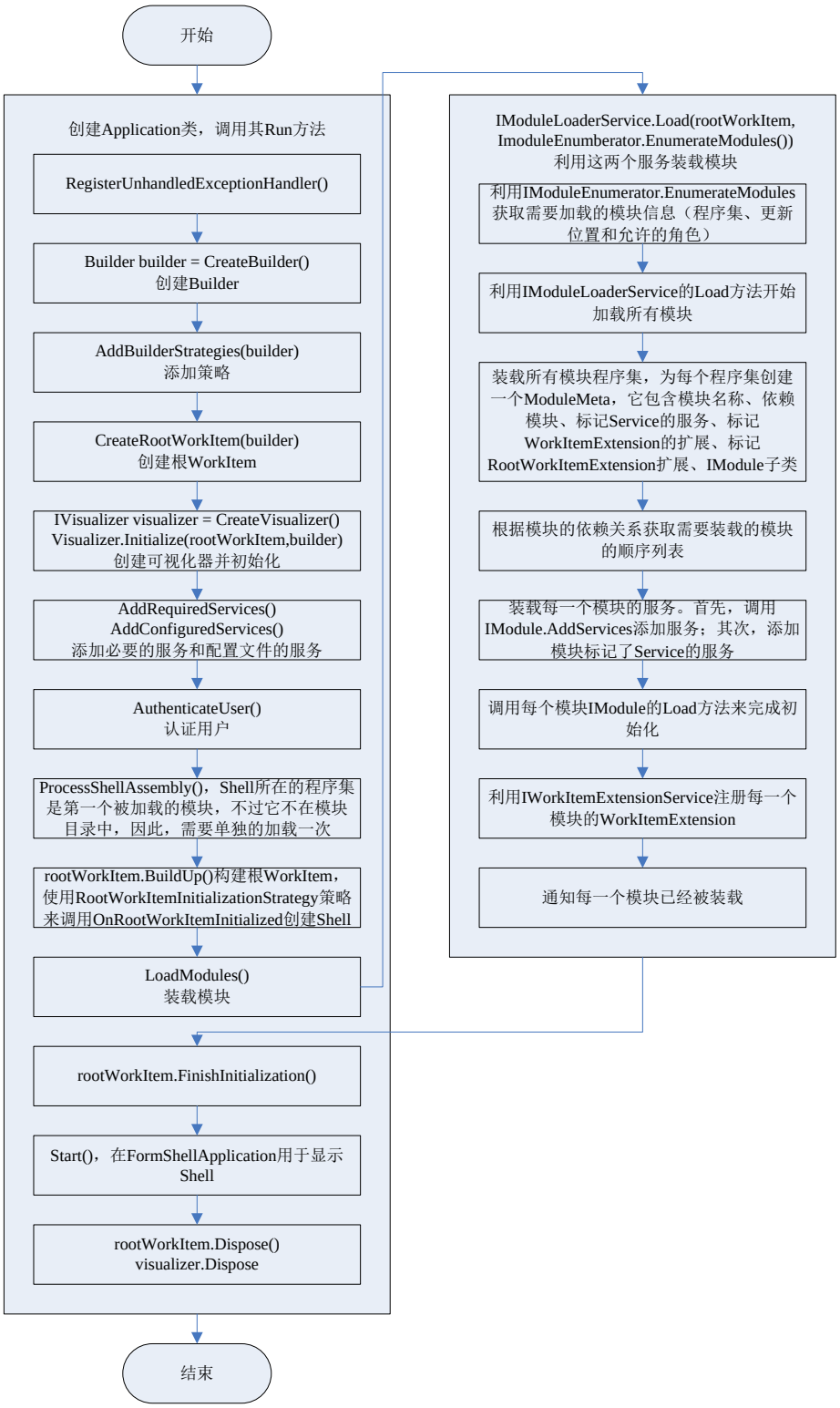
西安尤埃信息技术有限公司（http://www.uishell.com）成立于 2008 年 5 月份，专注于尤埃开放服务平台和尤埃 SaaS 引擎云计算产品开发。

尤埃开放服务平台（XAUI Open Service Platform，UIOSP）是一个移植了 **OSGi** 规范的**动态插件化与模块化平台**，支持插件化与模块化、SOA 和模块扩展。

尤埃 SaaS 引擎（XAUI SaaS Engine，XSE）是一个 SaaS 应用商店**开放平台**。该平台是面向 SaaS 运营商、SaaS 提供商和 SaaS 消费者三个角色的 **PaaS 云计算平台**，其模式为“SaaS 运营商负责平台运营，SaaS 提供商利用平台提供的开发工具包基于 VS2008SP1 开发 SaaS 应用并上传，SaaS 消费者在应用商店挑选、购买并使用 SaaS 应用”。该平台由应用商店网站、应用开发工具包和应用虚拟运行环境构成。

0 CAB 序

0.1 应用系统启动过程



五个大步骤：1 创建 Builder；2 添加服务（包括主程序所在的程序集）；3 构建根 WorkItem，创建 Shell；4 装载模块；5 启动系统。

0.2 WorkItem 相关

1 基础设施

1.1 特性（Attributes）

1.1.1 WorkItemExtensionAttribute

指示 WorkItem 扩展的特性。用于声明一个 WorkItemExtension 类。

```
using System;
using Microsoft.Practices.CompositeUI.Services;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple=true, Inherited=true)]
    public class WorkItemExtensionAttribute : Attribute
    {
        private Type workItemType;
        public WorkItemExtensionAttribute(Type workItemType)
        {
            Guard.ArgumentNotNull(workItemType, "workItemType");
            this.workItemType = workItemType;
        }
        //被扩展的WorkItem类型。
        public Type WorkItemType
        {
            get { return workItemType; }
```

```
        }
    }
}
```

1.1.2 OptionalDependencyAttribute

可选的依赖参数的特性的基类。

```
using System;
using Microsoft.Practices.ObjectBuilder;
namespace Microsoft.Practices.CompositeUI
{
    public abstract class OptionalDependencyAttribute : ParameterAttribute
    {
        private bool required = true;
        //该依赖是否必须。
        public bool Required
        {
            get { return required; }
            set { required = value; }
        }
        public override abstract IParameter CreateParameter(Type memberType);
    }
}
```

1.1.3 ComponentDependencyAttribute

用于声明属性、构造器或方法的参数是对父 WorkItem 的组件集合（WorkItem.Items）的组件的引用。

```
using System;
using Microsoft.Practices.CompositeUI.Utility;
using Microsoft.Practices.ObjectBuilder;
```

```
namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Parameter)]
    public sealed class ComponentDependencyAttribute : OptionalDependencyAttribute
    {
        private bool createIfNotFound = false;
        private string id;
        private Type type;
        //依赖的组件的ID。
        public ComponentDependencyAttribute(string id)
        {
            Guard.ArgumentNotNull(id, "id");
            this.id = id;
        }
        //注入的组件的ID。
        public string ID
        {
            get { return id; }
        }
        //如果CreateIfNotFound为true，则该属性用于指定构建的组件的类型。
        public Type Type
        {
            get { return type; }
            set { type = value; }
        }
        //指定如果该组件不存在，是否新建一个。
        public bool CreateIfNotFound
        {
            get { return createIfNotFound; }
            set { createIfNotFound = value; }
```

```
        }
        //创建参数。返回一个DependencyParameter参数。
        public override IParameter CreateParameter(Type memberType)
        {
            return new DependencyParameter(type ?? memberType, id, null,
                GetNotPresentBehavior(), SearchMode.Local);
        }
        private NotPresentBehavior GetNotPresentBehavior()
        {
            if (createIfNotFound)
                return NotPresentBehavior.CreateNew;
            if (Required)
                return NotPresentBehavior.Throw;
            return NotPresentBehavior.ReturnNull;
        }
    }
}
```

1.1.4 ServiceDenpendencyAttribute

用于指定一个属性或参数是一个对服务的依赖，并且在将所在类的实例添加到 WorkItem 时，必须解决依赖关系。

```
using System;
using Microsoft.Practices.ObjectBuilder;
namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Parameter)]
    public sealed class ServiceDependencyAttribute : OptionalDependencyAttribute
    {
        private Type type;
```

```
public ServiceDependencyAttribute()
{
}
//服务的类型。
public Type Type
{
    get { return type; }
    set { type = value; }
}
//创建参数。
public override IParameter CreateParameter(Type memberType)
{
    return new ServiceDependencyParameter(type ?? memberType, Required);
}
//服务依赖参数。
class ServiceDependencyParameter : IParameter
{
    Type serviceType;
    bool ensureExists;
    public ServiceDependencyParameter(Type serviceType, bool ensureExists)
    {
        this.serviceType = serviceType;
        this.ensureExists = ensureExists;
    }
    public Type GetParameterType(IBuilderContext context)
    {
        return serviceType;
    }
    //获取WorkItem，然后从服务集合中获取该服务。
    public object GetValue(IBuilderContext context)
```

```
            {
                WorkItem workItem = (WorkItem)context.Locator.Get(
                    new DependencyResolutionLocatorKey(typeof(WorkItem), null));
                return workItem.Services.Get(serviceType, ensureExists);
            }
        }
    }
}
```

1.1.5 StateAttribute

指定一个属性或参数必须从 WorkItem 的 State 的值注入。

```
using System;
using Microsoft.Practices.ObjectBuilder;
namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Parameter,
        AllowMultiple = false, Inherited = true)]
    public sealed class StateAttribute : ParameterAttribute
    {
        private string id;
        //注入State的第一项。
        public StateAttribute()
        {
        }
        //注入State中指定ID的对象。
        public StateAttribute(string id)
        {
            this.id = id;
        }
```

```
//获取参数。
public override IParameter CreateParameter(Type memberType)
{
    return new StateParameter(memberType, id);
}
//State参数。
private class StateParameter : KnownTypeParameter
{
    string id;
    public StateParameter(Type type, string id)
        : base(type)
    {
        this.id = id;
    }
    public override object GetValue(IBuilderContext context)
    {
        //获取WorkItem。
        DependencyResolutionLocatorKey key = new
            DependencyResolutionLocatorKey(typeof(WorkItem), null);
        WorkItem wi = (WorkItem)context.Locator.Get(key);
        //获取状态数据。
        if (id != null)
            return wi.State[id];
        //获取第一个匹配的项。
        foreach (string stateID in wi.State.Keys)
            if (type.IsAssignableFrom(wi.State[stateID].GetType()))
                return wi.State[stateID];
        return null;
    }
}
```

```
    }
}
```

1.1.6 ServiceAttribute

指定一个类为根 WorkItem 的服务。在装载模块时会被自动注入到 WorkItem 中。

```
using System;

namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public sealed class ServiceAttribute : Attribute
    {
        private Type registerAs;
        private bool addOnDemand;
        //服务注册的接口类型默认为本省。
        public ServiceAttribute()
        {
        }
        //指定服务注册的接口类型。
        public ServiceAttribute(Type registerAs)
        {
            this.registerAs = registerAs;
        }
        public Type RegisterAs
        {
            get { return registerAs; }
        }
        //声明是否按需创建服务。
        public bool AddOnDemand
        {
```

```
            get { return this.addOnDemand; }
            set { this.addOnDemand = value; }
        }
    }
}
```

1.1.7 StateChangedAttribute

用于声明一个方法订阅了 WorkItem 状态数据变化的事件主题。

```
using Microsoft.Practices.CompositeUI.EventBroker;

namespace Microsoft.Practices.CompositeUI
{
    //指定方法将处理状态数据变更。
    public class StateChangedAttribute : EventSubscriptionAttribute
    {
        public StateChangedAttribute(string topic, ThreadOption option)
            : base(StateChangedTopic.BuildStateChangedTopicString(topic))
        {
            this.Thread = option;
        }
    }
}
```

1.1.8 RootWorkItemAttriute

指定一个类扩展了根 WorkItem 类。这个类必须实现 IWorkItemExtension 接口。

```
using System;

using Microsoft.Practices.CompositeUI.Services;

namespace Microsoft.Practices.CompositeUI
{

    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = true)]
```

```
    public class RootWorkItemExtensionAttribute : Attribute
    {
    }
}
```

1.1.9 ModuleAttribute

声明一个程序集是一个指定名称的 Module。

```
using System;
namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Assembly, AllowMultiple = false)]
    public sealed class ModuleAttribute : Attribute
    {
        private string name;
        public ModuleAttribute(string name)
        {
            this.name = name;
        }
        public string Name
        {
            get { return name; }
        }
    }
}
```

1.1.20 ModuleDependencyAttribute

指定模块的依赖关系。一个程序集可以依赖多个模块，这些模块必须在该模块之前装载。

```
using System;
```

```
namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Assembly, AllowMultiple=true)]
    public sealed class ModuleDependencyAttribute : Attribute
    {
        private string name;
        public ModuleDependencyAttribute(string name)
        {
            this.name = name;
        }
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
    }
}
```

1.2 Application 基础类

Application 类用于管理整个应用系统的生命周期。

1.2.1 CabApplication

CAB 应用类，用于启动 CAB 应用系统。主要完成的工作：创建 Builder、创建根 WorkItem、添加服务、认证、处理程序集、构建根 WorkItem、装载模块、完成根 WorkItem 的初始化、运行根 WorkItem、启动系统、清理根 WorkItem 等。

```
using System;
using System.Configuration;
using System.Diagnostics;
using System.Globalization;
```

```
using System.Reflection;
using System.Security.Permissions;
using Microsoft.Practices.CompositeUI.BuilderStrategies;
using Microsoft.Practices.CompositeUI.Commands;
using Microsoft.Practices.CompositeUI.Configuration;
using Microsoft.Practices.CompositeUI.EventBroker;
using Microsoft.Practices.CompositeUI.Services;
using Microsoft.Practices.CompositeUI.UIElements;
using Microsoft.Practices.ObjectBuilder;
using Microsoft.Practices.CompositeUI.Utility;

namespace Microsoft.Practices.CompositeUI
{
    //处理应用系统和指定类型（IWorkItem）的根WorkItem的生命周期。
    public abstract class CabApplication<TWorkItem>
        where TWorkItem : WorkItem, new()
    {
        TWorkItem rootWorkItem;
        SettingsSection configuration;
        Type visualizerType = null;
        //可视化器的类型。如果为null，即使在App.config配置，都不支持可视化功能。
        protected Type VisualizerType
        {
            get { return visualizerType; }
            set
            {
                if (value != null)
                    Guard.TypeIsAssignableFromType(value, typeof(IVisualizer),
                    "VisualizerType");
```

```
            visualizerType = value;
        }
    }
    //启动应用系统。
    //1. 创建Builder。
    //2. 创建RootWorkItem。
    //3. 添加服务。
    //4. 认证。
    //5. 处理程序集。
    //6. 构建根WorkItem。
    //7. 装载模块。
    //8. 其它。
    public void Run()
    {
        //注册异常处理句柄。
        RegisterUnhandledExceptionHandler();
        //创建Builder。
        Builder builder = CreateBuilder();
        //添加策略。
        AddBuilderStrategies(builder);
        //创建根WorkItem，并制定Builder。
        CreateRootWorkItem(builder);
        //创建可视化器并初始化。
        IVisualizer visualizer = CreateVisualizer();
        if (visualizer != null)
            visualizer.Initialize(rootWorkItem, builder);
        //添加必须的服务。这些服务可以在AddConfigureServices被替换。
        AddRequiredServices();
        //添加配置文件的服务。根据配置文件Services节点添加或替换服务。
        AddConfiguredServices();
```

```
        //添加自定义的服务，可被重写。
        AddServices();
        //认证用户。
        AuthenticateUser();
        //装载程序集。
        ProcessShellAssembly();
        //构建根WorkItem，执行根WorkItem的依赖注入。
        rootWorkItem.BuildUp();
        //装载模块。利用IModuleEnumeratorService和IModuleLoaderService装载模块。
        LoadModules();
        //完成根WorkItem初始化。
        rootWorkItem.FinishInitialization();
        //运行根WorkItem。
        rootWorkItem.Run();
        //启动系统。由子类实现。如启动Windows主窗体、注册插件等。
        Start();
        //系统运行结束，则清理资源。
        rootWorkItem.Dispose();
        if (visualizer != null)
            visualizer.Dispose();
    }
    //添加自定义服务。由用户重写。
    protected virtual void AddServices()
    {
    }
    //创建可视化器。
    protected IVisualizer CreateVisualizer()
    {
        if (visualizerType == null || Configuration.Visualizer.Count == 0)
            return null;
```

```
        return (IVisualizer)Activator.CreateInstance(visualizerType);
    }
    //使用IModuleLoaderService装载IModuleEnumerator返回的模块列表。
    private void LoadModules()
    {
        IModuleLoaderService loader =
            rootWorkItem.Services.Get<IModuleLoaderService>(true);
        IModuleEnumerator modEnumerator =
            rootWorkItem.Services.Get<IModuleEnumerator>(true);
        if (modEnumerator != null)
            loader.Load(rootWorkItem, modEnumerator.EnumerateModules());
    }
    //装载程序集。
    private void ProcessShellAssembly()
    {
        IModuleLoaderService loader =
            rootWorkItem.Services.Get<IModuleLoaderService>(true);
        Assembly assembly = Assembly.GetEntryAssembly();
        if (assembly != null)
            loader.Load(rootWorkItem, assembly);
    }
    //使用IAuthenticationService认证用户。
    private void AuthenticateUser()
    {
        IAuthenticationService auth =
            rootWorkItem.Services.Get<IAuthenticationService>(true);
        auth.Authenticate();
    }
    //添加配置文件定义的服务。
    private void AddConfiguredServices()
```

```
{
    foreach (ServiceElement service in Configuration.Services)
    {
        //如果存在该服务，则删除。
        Type registerAs = service.ServiceType ?? service.InstanceType;
        if (rootWorkItem.Services.Get(registerAs) != null)
            rootWorkItem.Services.Remove(registerAs);
        //添加该服务。
        object serviceinstance = rootWorkItem.Services.AddNew(
            service.InstanceType, registerAs);

        IConfigurable tmpConfigurable = serviceinstance as IConfigurable;
        //如果服务实现了IConfigurable接口，则调用其Configure方法。
        if (tmpConfigurable != null)
            tmpConfigurable.Configure(service.Parameters);
    }
}
//配置节。
protected virtual SettingsSection Configuration
{
    get
    {
        if (configuration == null)
            configuration = LoadConfiguration();
        return configuration;
    }
}
//装载应用程序的配置节。即CompositeUI节点。
private static SettingsSection LoadConfiguration()
{
```

```
            object section = ConfigurationManager.GetSection(SettingsSection.SectionName);
            SettingsSection configSection = section as SettingsSection;

            if (section != null && configSection == null)
            {
                throw new ConfigurationErrorsException(String.Format(
                    CultureInfo.CurrentCulture,
                    Properties.Resources.InvalidConfigurationSectionType,
                    SettingsSection.SectionName, typeof(SettingsSection)));
            }
            //如果为空，则返回空。
            if (configSection == null)
                configSection = new SettingsSection();
            return configSection;
        }
        //添加必须的服务。有：ITraceSourceCatalogService、IWorkItemExtensionService、
        //IWorkItemTypeCatalogService、IWorkItemActivationService、
IAuthenticationService、
        //IModuleEnumeratorService、ICryptographyService、ICommandAdapterMapService、
        //IUIElementAdapterFactoryCatalog服务。
        private void AddRequiredServices()
        {
            rootWorkItem.Services.AddNew<TraceSourceCatalogService,
                ITraceSourceCatalogService>();
            rootWorkItem.Services.AddNew<WorkItemExtensionService,
                IWorkItemExtensionService>();
            rootWorkItem.Services.AddNew<WorkItemTypeCatalogService,
                IWorkItemTypeCatalogService>();
            rootWorkItem.Services.AddNew<SimpleWorkItemActivationService,
                IWorkItemActivationService>();
```

```
        rootWorkItem.Services.AddNew<WindowsPrincipalAuthenticationService,
            IAuthenticationService>();
        rootWorkItem.Services.AddNew<ModuleLoaderService, IModuleLoaderService>();
        rootWorkItem.Services.AddNew<FileCatalogModuleEnumerator,
            IModuleEnumerator>();
        rootWorkItem.Services.AddOnDemand<DataProtectionCryptographyService,
            ICryptographyService>();
        rootWorkItem.Services.AddNew<CommandAdapterMapService,
            ICommandAdapterMapService>();
        rootWorkItem.Services.AddNew<UIElementAdapterFactoryCatalog,
            IUIElementAdapterFactoryCatalog>();
    }
    //创建根WorkItem。
    private void CreateRootWorkItem(Builder builder)
    {
        rootWorkItem = new TWorkItem();
        rootWorkItem.InitializeRootWorkItem(builder);
    }
    //注册异常处理句柄。
    [SecurityPermission(SecurityAction.Demand, ControlAppDomain = true)]
    private void RegisterUnhandledExceptionHandler()
    {
        if (!Debugger.IsAttached && IsOnUnhandledExceptionOverridden())
            AppDomain.CurrentDomain.UnhandledException +=
            new UnhandledExceptionEventHandler(OnUnhandledException); ;
    }

    private bool IsOnUnhandledExceptionOverridden()
    {
        return GetType().GetMethod("OnUnhandledException").DeclaringType !=
```

```
            typeof(CabApplication<TWorkItem>);
        }
        //创建Builder并添加策略和政策。添加的策略有：EventBrokerStrategy、
CommandStrategy、
        //RootWorkItemInitializationStrategy和ObjectBuiltNotificationStrategy。
        //添加默认政策有：ISingletonPolicy、IBuilderTracePolicy和
        //ObjectBuiltNotificationPolicy。
        private Builder CreateBuilder()
        {
            Builder builder = new Builder();

            builder.Strategies.AddNew<EventBrokerStrategy>(BuilderStage.Initialization);
            builder.Strategies.AddNew<CommandStrategy>(BuilderStage.Initialization);
            builder.Strategies.Add(new RootWorkItemInitializationStrategy(
                this.OnRootWorkItemInitialized), BuilderStage.Initialization);
            builder.Strategies.AddNew<ObjectBuiltNotificationStrategy>(
                BuilderStage.PostInitialization);
            builder.Policies.SetDefault<ISingletonPolicy>(new SingletonPolicy(true));
            builder.Policies.SetDefault<IBuilderTracePolicy>(new BuilderTraceSourcePolicy(
                new TraceSource("Microsoft.Practices.ObjectBuilder")));
            builder.Policies.SetDefault<ObjectBuiltNotificationPolicy>(
                new ObjectBuiltNotificationPolicy());

            return builder;
        }
        //返回根WorkItem。
        public TWorkItem RootWorkItem
        {
            get { return rootWorkItem; }
        }
```

```
        //根节点初始化后处理。由子类重写。
        protected virtual void OnRootWorkItemInitialized()
        {
        }
        //添加自定义策略。
        protected virtual void AddBuilderStrategies(Builder builder)
        {
        }
        //启动应用系统。由子类实现。
        protected abstract void Start();
        //未处理异常处理，由子类重写。
        public virtual void OnUnhandledException(object sender,
            UnhandledExceptionEventArgs e)
        {
        }
    }
}
```

1.2.2 CabShellApplication

扩展 CabApplication。添加一个 Shell 属性，用于指定该类应用具有一个启动的 Shell。

```
namespace Microsoft.Practices.CompositeUI
{
    //指定根WorkItem类型和Shell的类型。
    public abstract class CabShellApplication<TWorkItem, TShell> :
        CabApplication<TWorkItem>
        where TWorkItem : WorkItem, new()
    {
        private TShell shell;
```

```
        //根节点初始化后，执行Shell创建。触发BeforeShellCreated和AfterShellCreated。
        //当前方法由RootItemInitiatilizationStrategy的CallBack调用。
        //创建Shell并添加到RootWorkItem中。
        protected sealed override void OnRootWorkItemInitialized()
        {
            BeforeShellCreated();
            shell = RootWorkItem.Items.AddNew<TShell>();
            AfterShellCreated();
        }
        //在Shell创建之前的操作。
        protected virtual void BeforeShellCreated()
        {
        }
        //Shell创建之后的操作。
        protected virtual void AfterShellCreated()
        {
        }
        //Shell的实例。
        protected TShell Shell
        {
            get { return shell; }
        }
    }
}
```

1.3 工作项（WorkItem）

1.3.1 WorkItemStatus

```
namespace Microsoft.Practices.CompositeUI
```

```
{
    //WorkItem状态。
    public enum WorkItemStatus
    {
        Inactive, //冻结
        Active, //激活
        Terminated, //终止
    }
}
```

1.3.2 WorkItem

WorkItem 代表一个用例，它是用一个容器，包含了完成该用例的各个组件。应用类在启动时首先创建一个 Builder，然后创建一个根 WorkItem 并调用其 InitRootWorkItem 方法，加载必要的服务后，会调用该 WorkItem 的 BuilderUp 方法，在加载完模块或，调用其 FinishInitialization 来完成初始化工作，然后调用其 Run 方法开始运行根 WorkItem。对于子 WorkItem，在调用父 WorkItem.WorkItems 集合的 AddNew 方法时，首先利用父 WorkItem 的 Builder 对其进行构建，然后再调用其 Run 方法运行。

（1）字段

●builder：对象构建器。对于根 WorkItem，由 InitializeRootWorkItem 方法指定，对于子 WorkItem，由于子 WorkItem 是父 WorkItem 的构建器构建的，它会调用注入方法 InitializeWorkItem，在该方法设置子的构建器与父构建器相同。

●buildUpFinished：WorkItem 构建完成标志，避免多次构建。

●locator：本 WorkItem 的 locator，它是子 locator 的父 locator。

●parent：父 WorkItem。

●smartPartInfos：SmartPart 和 Info 列表序列集合。一个 SmartPart 对象具有多个 Info 对象。

●state：本 WorkItem 的 State，由 IStatePersistenceService 管理。

●status：工作项状态，激活、冻结和终止状态。

●commandCollection：命令集合。

●smartPartCollection：SmartPart 集合。

●workItemCollection：子 WorkItem 集合。

●workspaceCollection：Workspace 集合。

●itemsCollection：存储杂项集合。

●eventTopicCollection：事件主题集合。

●serviceCollection：服务集合。

●uiExtensionSiteCollection：UI 扩展站点集合。

●traceSource：跟踪源。

（2）属性

A）公共属性

●commands：命令集合。

●ID：工作项状态的 ID，更改 ID 后，旧 State 会被清理并创建新 State。

●Items：杂项集合。

●Parent：父 WorkItem。

●RootWorkItem：根 WorkItem。

●Services：服务集合。

●SmartParts：SmartPart 集合。

●State：状态数据。

●Status：WorkItem 状态。

●TraceSource：跟踪源。

●UIExtensionSites：扩展站点集合。

●WorkItems：子 WorkItem 集合。

●Workspaces：Workspace 集合。

●EventTopics：事件主题集合。

B）保护属性

●InnerBuilder：内部 Builder。

●InnerLocator：内部 Locator。

C）私有属性

●ActivationService：从服务集合中获取 IWorkItemActivationService 实例。

（3）公共事件

●Activated：已激活事件。

●Activating：正在激活事件。

●Deactiving：正在冻结事件。

●Deactived：已冻结事件。

●Disposed：已清理事件。

●IdChanged：状态 ID 已改变事件。

●RunStarted：WorkItem 已开始运行事件。

●ObjectAdded：对象已添加事件。

●ObjectRemoved：对象删除事件。

（4）函数

A）构造器、清理和初始化

●Dispose：清理 WorkItem。调用保护方法 Dispose(true)，并使用垃圾收集。

●Run：运行 WorkItem。

●InitializeRootWorkItem：初始化根 WorkItem，由第一次调用 new 创建 WorkItem 的用户调用，目前在 Application 类中实现。子 WorkItem 则利用依赖注入完成初始化。该方法用于设置根 WorkItem 的对象构建器，创建 Locator，初始化字段、集合和状态。

●InitializeWorkItem：声明了 InjectionMethod 特性，由父 WorkItem 构建子 WorkItem 时自动调用，初始化字段、集合、状态和服务。

●InitializeFields：初始化字段，包括对象构建器、对象存储器、对象通知器、存储当前对象到当前 Locator，主键分别为 typeof(WorkItem)和 GetType()。

●LocateWorkItem：存储当前 WorkItem 到 Locator，指定的 Type 参数为主键。

●InitializeCollectionFacades：初始化集合字段。

●InitializeState：初始化状态，即新建一个 GUID 作为属性 ID 的值。

●CreateCommand：创建指定类型和名称命令。

●CreateEventTopic：创建指定类型和名称的事件主题。

●InitializeServices：虚方法，完成服务初始化。目前没有任何实现。

B）工作项状态

●Activate：更改当前工作项状态为激活。

●Deactivate：更改前工作项状态为冻结。

●Terminate：终止 WorkItem。如果已终止，则抛出异常，否则开始清理对象。

C）状态数据管理

●DeleteState：调用 IStatePersistenceService 的 Remove 方法删除当前 ID 的状态。

●Load：状态当前 WorkItem 对象的状态数据。

●Save：保存当前 WorkItem 对象的状态数据。

D）其它方法

●GetSmartPartInfo：获取某对象指定类型的 SmartPartInfo。所有的 SmartPartInfo 都存在根 WorkItem，且一个 SmartPart 可以有对应不同 Workspaces 的 Info。

●OnBuiltUp：对 IBuilderAware 的实现，设置 buildUpFinished 为 true 并调用 FinishInitialization 方法完成对 IWorkItemExtensionService 对该工作项的初始化，并触发 Initialized 事件。

●FinishInitialization：完成初始化步骤。

●OnTearingDown：对 IBuilderAware 的实现，删除通知策略的事件绑定。

●RegisterSmartPartInfo：在根 WorkItem 中注册 Info。

●RegisterSmartPartInfoWorkItem：静态方法，用于注册某一 Info。指定了 SmartPart 对象，SmartPartInfo 对象、重复判断的条件和 WorkItem。在一个 WorkItem 中，一个 SmartPart 可以对应对各 SmartPart。如果当前 WorkItem 不存在指定 SmartPart，则直接插入；若存在 SmartPart，则根据重复判定条件判定是否存在该类型的 Info，若存在则删除后再插入。

E）保护的方法

●BuildUp：由 CabApplication 类调用，用于给根 WorkItem 执行依赖注入。因为根 WorkItem 是使用 New 方法创建，因此，需要对其属性和方法执行依赖注入。所有的子 WorkItem 无需执行此方法。因为它们都是由父 WorkItem 的 WorkItems 集合的 AddNew 方法添加，该方法会利用父 WorkItem 的 Builder 构建实例。

●Dispose(bool)：清理 WorkItem。清理一个 WorkItem 需要完成一系列任务——更改状态为终止、TearDown 当前 WorkItem 的 Locator 对象、清理 UIExtensions

集合、注销对象通知事件、释放 WorkItem 状态数据、从父 Locator 删除当前 WorkItem 的实例、从根节点删除当前 WorkItem 对应的 SmartParts 的 SmartPartInfo、从声明周期容器中删除当前 WorkItem 的实例、TearDown 声明周期容器的对象包括其本身、释放声明周期容器、触发 OnDisposed 事件。

● OnActivated 、 OnActivating 、 OnDeactivated 、 OnDeactiving 、 OnDisposed 、 OnIdChanged、OnInitialized（触发公共事件）、OnObjectAdded 与 OnObjectRemoved（由对象添加通知政策订阅，并触发公共事件）、OnRunStarted（）、OnTerminated、OnTerminating（触发公共事件）等默认事件处理方法，可以被重写。

F）私有方法

●AttachState：修改状态数据。

●ReleaseState：删除状态数据。

●FindSmartPartInfo：为 SmartPart 查找指定类型的 Info。

●stateDataStateChanged：状态数据更新处理，引发发布的事件主题。

●ExtensionCallback：回调代理。

●ExtensionCallBackCancellable：可被取消的回调代理。

●ChangeStatus：处理状态变更。首先调用 On…ing 方法，如果该状态变更事件未被取消，则更改状态，并触发 On…ed 方法。

●FireIfActivating、FireIfDeactivating、FireIfTerminating、FireStatusEvents：On…ing 事件和 On…ed 事件。

●ThrowIfActivationServiceNotPresent、ThrowIfWorkItemTerminated：抛出异常事件。

```
using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.IO;

using System.Reflection;

using Microsoft.Practices.CompositeUI.BuilderStrategies;

using Microsoft.Practices.CompositeUI.Collections;

using Microsoft.Practices.CompositeUI.Commands;

using Microsoft.Practices.CompositeUI.EventBroker;
```

```
using Microsoft.Practices.CompositeUI.Properties;
using Microsoft.Practices.CompositeUI.Services;
using Microsoft.Practices.CompositeUI.SmartParts;
using Microsoft.Practices.CompositeUI.UIElements;
using Microsoft.Practices.CompositeUI.Utility;
using Microsoft.Practices.ObjectBuilder;
using System.ComponentModel;
using System.Globalization;
using System.Threading;
namespace Microsoft.Practices.CompositeUI
{
    //定义一个WorkItem。
    public class WorkItem : IBuilderAware, IDisposable
    {
        #region 字段
        private Builder builder; //对象构建器。
        private bool buildUpFinished = false; //是否构建完毕，避免重复构建。
        private ILifetimeContainer lifetime = new LifetimeContainer();//生命周期容器。
        private IReadWriteLocator locator; //对象存储器，存储引用。
        private WorkItem parent; //父WorkItem。
        private ListDictionary<object, ISmartPartInfo> smartPartInfos =
            new ListDictionary<object, ISmartPartInfo>();
        private State state; //状态。
        private WorkItemStatus status; //工作项状态。
        //WorkItem存储的所有对象都存储在Locator中，这些管理对象集合用于管理不同类
        //型的对象。Command集合对象所有继承Command的对象的实例，SmartPart集合管理所有声明
        //了SmartPart特性的对象，WorkItem集合管理了所有继承了WorkItem的对象，Workspace集合
        //管理了所有实现IWorkspace接口的对象，Items管理所有对象，EventTopic集合管理所有
        //继承了EventTopic对象。
        //由于服务没有统一的接口，都是对象，因此只能用ID为空，主键为类型来管理，需要用
```

```
//另一个管理集合来管理服务。
private ManagedObjectCollection<Command> commandCollection; //命令集合。
private ManagedObjectCollection<object> smartPartCollection; //SmartParts集合。
private ManagedObjectCollection<WorkItem> workItemCollection; //子WorkItem集合。
private ManagedObjectCollection<IWorkspace> workspaceCollection; //Workspace集合。
private ManagedObjectCollection<object> itemsCollection; //存储项集合。
private ManagedObjectCollection<EventTopic> eventTopicCollection; //事件主题集合。
private ServiceCollection serviceCollection; //服务集合。
private UIExtensionSiteCollection uiExtensionSiteCollection; //UI扩展站点集合。

private TraceSource traceSource = null; //Trace。
#endregion

#region WorkItem Lifetime (ctor, Dispose, Initialize, etc.)
//创建一个WorkItem。
public WorkItem()
{
}
//清理WorkItem。
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
//运行WorkItem。
public void Run()
{
    OnRunStarted();
}
//初始化一个根WorkItem。由使用new创建第一个WorkItem的用户调用。
```

```
//默认是CabApplication。
//子WorkItem通过父WorkItem的注入依赖自动初始化。
//完成：设置Builder、新建Locator、初始化字段、集合和状态。
//在CabApplication类的Run方法中，它首先构建Builder，然后创建一个RootWorkItem，并将
//该Builder作为参数传递给该方法来初始化该RootWorkItem。
//紧接着添加必要的服务和初始化服务，再调用RootWorkItem的BuilderUp方法对根WorkItem
//进行依赖注入。完成注入根WorkItem后，则装载模块。最后，再调用RootWorkItem的
//FinishInitialization来装载WorkItemExtension服务，并调用。
//此时，一个根WorkItem准备完毕。可以调用Run方法运行该WorkItem。
protected internal void InitializeRootWorkItem(Builder builder)
{
    this.builder = builder; //设置Builder。
    this.locator = new Locator(); //新建Locator。
    InitializeFields();//初始化字段。
    InitializeCollectionFacades();//初始化集合。
    InitializeState();//初始化状态。
}
//构建WorkItem后的初始化。由父WorkItem构建子WorkItem时自动调用。
//完成：初始化字段、集合、状态和服务。
//该函数在WorkItem的初始化阶段运行。
//问题：为什么Parent属性能够正确设置为父WorkItem呢？
//答案：在Parent注入时，Builder从本地Locator获取WorkItem服务，
//此时，本地不存在WorkItem服务，于是便从父Locator获取父WorkItem服务。
[InjectionMethod]
public void InitializeWorkItem()
{
    if (buildUpFinished)
        throw new InvalidOperationException();
    InitializeFields();
    InitializeCollectionFacades();
```

```
        InitializeState();
        InitializeServices();
    }
    //初始化字段。并将当前对象保存到Locator，主键为WorkItem和GetType()。
    private void InitializeFields()
    {
        if (builder == null) //对于根WorkItem，通过InitRootWorkItem设置。
                             //对于子WorkItem，设置其父WorkItem的Builder。
            builder = parent.builder;
        if (locator == null) //Locator。
            locator = new Locator(parent.locator);
        if (!locator.Contains(typeof(ILifetimeContainer), SearchMode.Local))
            locator.Add(typeof(ILifetimeContainer), lifetime);
        ObjectBuiltNotificationPolicy policy =
            builder.Policies.Get<ObjectBuiltNotificationPolicy>(null, null);
        if (policy != null) //添加通知策略。
        {
            policy.AddedDelegates[this] = new
                ObjectBuiltNotificationPolicy.ItemNotification(OnObjectAdded);
            policy.RemovedDelegates[this] = new
                ObjectBuiltNotificationPolicy.ItemNotification(OnObjectRemoved);
        }
        LocateWorkItem(typeof(WorkItem)); //存储WorkItem到Locator。
        LocateWorkItem(GetType()); //存储当前WorkItem到Locator。
        status = WorkItemStatus.Inactive; //设置状态。
    }
    //将当前WorkItem存储到Locator。
    private void LocateWorkItem(Type workItemType)
    {
        DependencyResolutionLocatorKey key = new
```

```
            DependencyResolutionLocatorKey(workItemType, null);
        if (!locator.Contains(key, SearchMode.Local))
            locator.Add(key, this);
    }
    //初始化集合。
    private void InitializeCollectionFacades()
    {
        if (serviceCollection == null)
        {
            serviceCollection = new ServiceCollection(lifetime, locator, builder,
                parent == null ? null : parent.serviceCollection);
        }
        if (commandCollection == null)
        {
            commandCollection = new ManagedObjectCollection<Command>(
                lifetime, locator,
                builder, SearchMode.Up, CreateCommand, null,
                parent == null ? null : parent.commandCollection);
        }
        if (workItemCollection == null)
        {
            workItemCollection = new ManagedObjectCollection<WorkItem>(lifetime,
                locator, builder, SearchMode.Local, null, null,
                parent == null ? null : parent.workItemCollection);
        }
        if (workspaceCollection == null)
        {
            workspaceCollection = new ManagedObjectCollection<IWorkspace>(lifetime,
                locator, builder, SearchMode.Up, null, null,
                parent == null ? null : parent.workspaceCollection);
```

```
}
if (itemsCollection == null)
{
    itemsCollection = new ManagedObjectCollection<object>(lifetime, locator,
        builder, SearchMode.Local, null, null,
        parent == null ? null : parent.itemsCollection);
}
if (smartPartCollection == null)
{
    smartPartCollection = new ManagedObjectCollection<object>(lifetime,
        locator, builder, SearchMode.Local, null,
        delegate(object obj)
            {
                return obj.GetType().GetCustomAttributes(
                    typeof(SmartPartAttribute), true).Length > 0;
            },
        parent == null ? null : parent.smartPartCollection);
}
if (eventTopicCollection == null)
{
    if (parent == null)
        eventTopicCollection = new ManagedObjectCollection<EventTopic>(
            lifetime, locator, builder, SearchMode.Local,
                CreateEventTopic, null, null);
    else
        eventTopicCollection = RootWorkItem.eventTopicCollection;
}
if (uiExtensionSiteCollection == null)
{
    if (parent == null)
```

```
                uiExtensionSiteCollection = new UIExtensionSiteCollection(this);
            else
                uiExtensionSiteCollection = new UIExtensionSiteCollection(
                    parent.uiExtensionSiteCollection);
        }
    }
    //初始化状态。
    private void InitializeState()
    {
        ID = Guid.NewGuid().ToString();
    }
    //当命令不存在时，命令管理集合将用一个指定该函数的代理创建一个Command。
    protected virtual Command CreateCommand(Type t, string name)
    {
        return new Command();
    }
    //同上。
    protected virtual EventTopic CreateEventTopic(Type t, string topicName)
    {
        return new EventTopic();
    }
    //初始化服务。
    protected virtual void InitializeServices()
    {
    }
    #endregion

    #region 事件。SmartPartExtension将实现这些事件的默认处理来扩展WorkItem功能。
    //激活。
    public event EventHandler Activated;
```

```
//正在激活。
public event CancelEventHandler Activating;
//冻结。
public event EventHandler Deactivated;
//正在冻结。
public event CancelEventHandler Deactivating;
//清理。
public event EventHandler Disposed;
//状态ID改变。
public event EventHandler<DataEventArgs<string>> IdChanged;
//初始化。
public event EventHandler Initialized;
//开始运行。
public event EventHandler RunStarted;
//终止。
public event EventHandler Terminated;
//正在终止。
public event EventHandler Terminating;
//添加对象。
internal event EventHandler<DataEventArgs<object>> ObjectAdded;
//删除对象。
internal event EventHandler<DataEventArgs<object>> ObjectRemoved;
#endregion

#region 属性。
//命令。
public ManagedObjectCollection<Command> Commands
{
    get { return commandCollection; }
}
```

```
//获取和设置WorkItem的ID。该ID用于WorkItem的持久化。如果设置了新的ID，
//旧状态信息会被清空，取而代之的是一个新的空状态。
public string ID
{
    get
    {
        if (state == null)
            return null;
        return state.ID;
    }
    set
    {
        if (state != null)  //清空旧数据。
            ReleaseState();
        //付新状态。
        AttachState(new State(value));
        OnIdChanged();//触发ID变更事件。
    }
}
//存储项。
public ManagedObjectCollection<object> Items
{
    get { return itemsCollection; }
}
//父WorkItem。
[Browsable(false)]
[Dependency(NotPresentBehavior = NotPresentBehavior.ReturnNull)]
public WorkItem Parent
{
    get { return parent; }
```

```
        set
        {
            if (buildUpFinished)
                throw new InvalidOperationException();
            parent = value;
        }
    }
    //根WorkItem。
    [Browsable(false)]
    public WorkItem RootWorkItem
    {
        get
        {
            WorkItem result = this;
            while (result.Parent != null)
                result = result.Parent;
            return result;
        }
    }
    //服务。
    public ServiceCollection Services
    {
        get { return serviceCollection; }
    }
    //SmartParts。
    public ManagedObjectCollection<object> SmartParts
    {
        get { return smartPartCollection; }
    }
    //状态数据。
```

```
public State State
{
    get { return state; }
}
//WorkItem状态。
public WorkItemStatus Status
{
    get { return status; }
}
//跟踪信息。
[ClassNameTraceSource]
public TraceSource TraceSource
{
    set { traceSource = value; }
}
//UI扩展点。
public UIExtensionSiteCollection UIExtensionSites
{
    get { return uiExtensionSiteCollection; }
}
//子WorkItem。
public ManagedObjectCollection<WorkItem> WorkItems
{
    get { return workItemCollection; }
}
//Workspaces。
public ManagedObjectCollection<IWorkspace> Workspaces
{
    get { return workspaceCollection; }
}
```

```
//事件主题。
public ManagedObjectCollection<EventTopic> EventTopics
{
    get { return eventTopicCollection; }
}
#endregion

#region WorkItem状态。
//激活。更改状态为激活状态。
public void Activate()
{
    ChangeStatus(WorkItemStatus.Active);
}
//冻结。更改状态为冻结状态。
public void Deactivate()
{
    ChangeStatus(WorkItemStatus.Inactive);
}
#endregion

#region 状态数据管理。
//删除当前WorkItem的状态数据。
public void DeleteState()
{
    IStatePersistenceService service = Services.Get<IStatePersistenceService>();
    if (service == null)
    {
        throw new ServiceMissingException(typeof(IStatePersistenceService), this);
    }
    service.Remove(ID);
```

```
}
//装载WorkItem的状态数据。
public void Load()
{
    IStatePersistenceService service = Services.Get<IStatePersistenceService>();
    if (service == null)
    {
        throw new ServiceMissingException(typeof(IStatePersistenceService), this);
    }
    using (new WriterLock(state.syncRoot))
    {
        State newState = service.Load(ID);
        newState.syncRoot = ReleaseState();
        AttachState(newState);
    }
}
#endregion

#region 其它方法。
//获取SmartPart的Info。
public TSmartPartInfo GetSmartPartInfo<TSmartPartInfo>(object smartPart)
    where TSmartPartInfo : ISmartPartInfo
{
    Guard.ArgumentNotNull(smartPart, "smartPart");
    return FindSmartPartInfo<TSmartPartInfo>(smartPart);
}
//IBuilderWare.OnBuiltUp事件。设置构建标志。
public virtual void OnBuiltUp(string id)
{
    //阻止重复BuiltUp事件。
```

```
        if (buildUpFinished)
            throw new InvalidOperationException();
        buildUpFinished = true;
        if (parent != null)
            FinishInitialization();
    }
    //通过调用IWorkItemExtensionService和OnInitiatized代理完成WorkItem类的初始化。
    //对于根WorkItem，它由CabApplication类，在装载模块结束后调用。
    //对于子WorkItem，它在OnBuiltUp事件中被自动调用。
    protected internal void FinishInitialization()
    {
        IWorkItemExtensionService extensionsService =
            Services.Get<IWorkItemExtensionService>();
        if (extensionsService != null) //调用ExtensionService。
            extensionsService.InitializeExtensions(this);
        OnInitialized();//调用代理事件。
    }
    //清理对象。IBuildWare.OnTearDown实现。
    public virtual void OnTearingDown()
    {
        ObjectBuiltNotificationPolicy policy =
            builder.Policies.Get<ObjectBuiltNotificationPolicy>(null, null);
        //移除代理。
        if (policy != null)
        {
            policy.AddedDelegates.Remove(this);
            policy.RemovedDelegates.Remove(this);
        }
    }
    //注册SmartPartInfo。
```

```
public void RegisterSmartPartInfo(object smartPart, ISmartPartInfo info)
{
    Guard.ArgumentNotNull(smartPart, "smartPart");
    Guard.ArgumentNotNull(info, "info");
    //相同类型的SmartPartInfo查询条件。
    Predicate<ISmartPartInfo> findSPI = delegate(ISmartPartInfo match)
    {
        return match.GetType() == info.GetType();
    };
    //注册到根WorkItem。
    if (this != RootWorkItem)
    {
        RegisterSmartPartInfoInWorkItem(smartPart, info, findSPI, RootWorkItem);
    }
    RegisterSmartPartInfoInWorkItem(smartPart, info, findSPI, this);
}
//注册SmartPartInfo。
private static void RegisterSmartPartInfoInWorkItem(object smartPart,
    ISmartPartInfo info, Predicate<ISmartPartInfo> findSPI, WorkItem workItem)
{
    //不存在，直接注册。
    if (workItem.smartPartInfos.ContainsKey(smartPart) == false)
    {
        workItem.smartPartInfos.Add(smartPart);
        workItem.smartPartInfos[smartPart].Add(info);
    }
    else
    {
        //如果不存在同类型的，则注册。
        ISmartPartInfo registered =
```

```
                workItem.smartPartInfos[smartPart].Find(findSPI);
            if (registered != null)
            {
                workItem.smartPartInfos[smartPart].Remove(registered);
            }
            workItem.smartPartInfos[smartPart].Add(info);
        }
    }
    //保存WorkItem的状态信息。
    public void Save()
    {
        IStatePersistenceService service = Services.Get<IStatePersistenceService>();
        if (service == null)
            throw new ServiceMissingException(typeof(IStatePersistenceService), this);
        using (new WriterLock(state.syncRoot))
        {
            service.Save(state);
            state.AcceptChanges();
        }
    }
    //终止WorkItem。
    public void Terminate()
    {
        ThrowIfWorkItemTerminated();
        Dispose();
    }
    #endregion

    #region 保护的成员。
    //在创建一个根WorkItem并需要确保属性正确构建时使用。
```

```
        //注意：该方法仅能够在内部程序集使用，由CabShellApplication类调用来构建根
WorkItem。
        //根WorkItem采用New方法创建，需要调用BuildUp来执行依赖注入。这样做可以为继承
WorkItem
        //其它类型WorkItem为根的节点进行依赖注入。
        protected internal void BuildUp()
        {
            //使用一个新的ID来确保BuildUp可以完成。因为在此前已存在一个null作为ID的对象，
            //单件策略会阻止其构建。
            Type type = GetType();
            string temporaryID = Guid.NewGuid().ToString();
            PropertySetterPolicy propPolicy = new PropertySetterPolicy();
            //设置根WorkItem的Parent属性为null。
            propPolicy.Properties.Add("Parent", new PropertySetterInfo("Parent",
                new ValueParameter(typeof(WorkItem), null)));

            PolicyList policies = new PolicyList();
            policies.Set<ISingletonPolicy>(new SingletonPolicy(false), type, temporaryID);
            policies.Set<IPropertySetterPolicy>(propPolicy, type, temporaryID);

            builder.BuildUp(locator, type, temporaryID, this, policies);
        }
        //Builder。
        protected Builder InnerBuilder
        {
            get { return builder; }
        }
        //Locator。
        protected IReadWriteLocator InnerLocator
        {
```

```
        get { return locator; }
    }
    //清理。
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (status == WorkItemStatus.Terminated)
                return;
            //更改状态为终止。
            ChangeStatus(WorkItemStatus.Terminated);
            //执行清理工作。主要是从容器中注销组件。
            builder.TearDown(locator, this);
            uiExtensionSiteCollection.Dispose();
            //注销事件。
            ObjectBuiltNotificationPolicy policy =
                builder.Policies.Get<ObjectBuiltNotificationPolicy>(null, null);
            if (policy != null)
            {
                policy.AddedDelegates.Remove(this);
                policy.RemovedDelegates.Remove(this);
            }
            //注销状态数据。
            if (state != null)
                ReleaseState();
            if (parent != null)
            {
                List<object> ids = new List<object>();
                //从父WorkItem中删除该类的实例。
                foreach (KeyValuePair<object, object> pair in parent.locator)
```

```
                if (pair.Value == this)
                    ids.Add(pair.Key);
            foreach (object id in ids)
                parent.locator.Remove(id);
            //父类的Lifetime容器删除该实例。
            parent.lifetime.Remove(this);
            //删除SmartPartInfo。所有SmartPartInfo都保存在跟WorkItem。
            foreach (object smartpart in smartPartInfos.Keys)
                RootWorkItem.smartPartInfos.Remove(smartpart);
        }
        //删除当前实例。
        lifetime.Remove(this);
        List<object> lifetimeObjects = new List<object>();
        lifetimeObjects.AddRange(lifetime);
        //对Lifetime容器的实例，包括Lifetime容器本身执行清理工作。
        foreach (object obj in lifetimeObjects)
            if (lifetime.Contains(obj))
                builder.TearDown(locator, obj);
        //释放Lifetime实例。
        lifetime.Dispose();
        //触发事件。
        OnDisposed();
    }
}
//修改状态，并触发事件同时将该状态保存到Items集合。
private void AttachState(State state)
{
    this.state = state;
    this.state.StateChanged += new
        EventHandler<StateChangedEventArgs>(stateDataStateChanged);
```

```
        Items.Add(this.state);
    }
    //删除状态。
    private ReaderWriterLock ReleaseState()
    {
        ReaderWriterLock rwl = state.syncRoot;
        state.StateChanged -= new
            EventHandler<StateChangedEventArgs>(stateDataStateChanged);
        Items.Remove(state);
        state.Dispose();
        state = null;
        return rwl;
    }
    //激活处理。
    protected virtual void OnActivated()
    {
        if (this.Activated != null)
        {
            this.Activated(this, EventArgs.Empty);
        }
        if (traceSource != null)
            traceSource.TraceInformation(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.TraceWorkItemStatusChangedToActivated, ID));
    }
    //激活处理。
    protected virtual void OnActivating(CancelEventArgs args)
    {
        if (traceSource != null)
            traceSource.TraceInformation(String.Format(
```

```
                CultureInfo.CurrentCulture,
                Properties.Resources.TraceWorkItemStatusActivating, ID));
        if (this.Activating != null)
        {
            this.Activating(this, args);
        }
    }
    //冻结处理。
    protected virtual void OnDeactivated()
    {
        if (this.Deactivated != null)
        {
            this.Deactivated(this, EventArgs.Empty);
        }
        if (traceSource != null)
            traceSource.TraceInformation(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.TraceWorkItemStatusChangedToDeactivated, ID));
    }
    //冻结处理。
    protected virtual void OnDeactivating(CancelEventArgs args)
    {
        if (traceSource != null)
            traceSource.TraceInformation(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.TraceWorkItemStatusDeactivating, ID));
        if (this.Deactivating != null)
        {
            this.Deactivating(this, args);
        }
```

```
}
//清理处理。
protected virtual void OnDisposed()
{
    if (Disposed != null)
    {
        Disposed(this, EventArgs.Empty);
    }
    if (traceSource != null)
        traceSource.TraceInformation(String.Format(
            CultureInfo.CurrentCulture,
            Properties.Resources.TraceWorkItemStatusChangedToDisposed, ID));
}
//ID改变处理。
protected virtual void OnIdChanged()
{
    if (IdChanged != null)
        IdChanged(this, new DataEventArgs<string>(ID));
}
//初始化处理。
protected virtual void OnInitialized()
{
    if (this.Initialized != null)
    {
        this.Initialized(this, EventArgs.Empty);
    }
    if (traceSource != null)
        traceSource.TraceInformation(String.Format(
            CultureInfo.CurrentCulture,
            Properties.Resources.TraceWorkItemStatusChangedToInitialized, ID));
```

```
}
//添加对象。
protected virtual void OnObjectAdded(object item)
{
    if (ObjectAdded != null)
        ObjectAdded(this, new DataEventArgs<object>(item));
}
//删除对象。
protected virtual void OnObjectRemoved(object item)
{
    if (ObjectRemoved != null)
        ObjectRemoved(this, new DataEventArgs<object>(item));
}
//运行。
protected virtual void OnRunStarted()
{
    if (this.RunStarted != null)
    {
        this.RunStarted(this, EventArgs.Empty);
    }
    if (traceSource != null)
        traceSource.TraceInformation(String.Format(
            CultureInfo.CurrentCulture,
                Properties.Resources.TraceWorkItemRunStarted, ID));
}
//终止。
protected virtual void OnTerminated()
{
    if (this.Terminated != null)
    {
```

```
            this.Terminated(this, EventArgs.Empty);
        }
        if (traceSource != null)
            traceSource.TraceInformation(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.TraceWorkItemStatusChangedToTerminated, ID));
    }
    //正在终止。
    protected virtual void OnTerminating()
    {
        if (traceSource != null)
            traceSource.TraceInformation(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.TraceWorkItemStatusTerminating, ID));
        if (this.Terminating != null)
        {
            this.Terminating(this, EventArgs.Empty);
        }
    }
    #endregion

    #region 私有成员。
    //查找SmartPartInfo。
    private TSmartPartInfo FindSmartPartInfo<TSmartPartInfo>(object smartPart)
        where TSmartPartInfo : ISmartPartInfo
    {
        TSmartPartInfo result = default(TSmartPartInfo);
        if (RootWorkItem.smartPartInfos.ContainsKey(smartPart))
        {
            result = (TSmartPartInfo)RootWorkItem.
```

```
                smartPartInfos[smartPart].Find(
                delegate(ISmartPartInfo info)
                 {
                      return info.GetType() == typeof(TSmartPartInfo);
                 });
        }
        return result;
    }
    //激活服务。
    private IWorkItemActivationService ActivationService
    {
        get { return Services.Get<IWorkItemActivationService>(); }
    }
    //状态数据变更事件。
    private void stateDataStateChanged(object sender, StateChangedEventArgs e)
    {
        string topicKey = StateChangedTopic.BuildStateChangedTopicString(e.Key);
        EventTopic topic = EventTopics.Get(topicKey);
        if (topic != null)
        {
            topic.Fire(this, e, this, PublicationScope.WorkItem);
        }
    }
    private delegate void ExtensionCallback(WorkItem workItem);
    private delegate void ExtensionCallbackCancellable(
        WorkItem workItem, CancelEventArgs args);
    //更改状态。在Active方法、Deactive方法和Dispose方法被调用时，都将引起状态变更。
    private void ChangeStatus(WorkItemStatus newStatus)
    {
        //如果已经终止，则抛出异常。
```

```
        ThrowIfWorkItemTerminated();
        WorkItemStatus oldStatus = status;
        if (oldStatus != newStatus) //如果状态已更改
        {
            CancelEventArgs args = new CancelEventArgs();
            //触发正在更改状态事件，允许用户取消事件。
            FireIfActivating(newStatus, args);
            FireIfDeactivating(newStatus, args);
            FireIfTerminating(newStatus);
            if (args.Cancel == false)  //如果用户未取消更改状态
            {
                status = newStatus;
                if (ActivationService != null)  //调用激活服务的更改状态。
                {
                    //更改状态。
                    ActivationService.ChangeStatus(this);
                }
                //触发状态更改后的事件。
                FireStatusEvents();
            }
        }
    }
    //如果激活，则触发。
    private void FireIfActivating(WorkItemStatus newStatus, CancelEventArgs args)
    {
        if (newStatus == WorkItemStatus.Active)
        {
            OnActivating(args);
        }
    }
```

```
//如果冻结，则触发。
private void FireIfDeactivating(WorkItemStatus newStatus, CancelEventArgs args)
{
    if (newStatus == WorkItemStatus.Inactive)
    {
        OnDeactivating(args);
    }
}
//如果终止，则触发。
private void FireIfTerminating(WorkItemStatus newStatus)
{
    if (newStatus == WorkItemStatus.Terminated)
    {
        OnTerminating();
    }
}
//触发状态改变后事件。
private void FireStatusEvents()
{
    if (this.status == WorkItemStatus.Active)
    {
        OnActivated();
    }
    else if (this.status == WorkItemStatus.Inactive)
    {
        OnDeactivated();
    }
    else if (this.status == WorkItemStatus.Terminated)
    {
        OnTerminated();
```

```
            }
        }
        //如果激活服务不存在，则触发异常。
        private void ThrowIfActivationServiceNotPresent()
        {
            if (ActivationService == null)
            {
                throw new ServiceMissingException(
                    typeof(IWorkItemActivationService), this);
            }
        }
        //如果已终止，则触发异常。
        private void ThrowIfWorkItemTerminated()
        {
            if (this.Status == WorkItemStatus.Terminated)
            {
                throw new InvalidOperationException(Resources.WorkItemTerminated);
            }
        }
        #endregion
        #region 内部类。
        //跟踪特性。
        private class ClassNameTraceSourceAttribute : TraceSourceAttribute
        {
            public ClassNameTraceSourceAttribute() : base(typeof(WorkItem).FullName) { }
        }
        #endregion
    }
}
```

1.3.3 IWorkItemActivationService

WorkItem 激活服务，用于监控 WorkItem 的状态。

```
namespace Microsoft.Practices.CompositeUI
{
    public interface IWorkItemActivationService
    {
        //当WorkItem状态改变后，由WorkItem调用来执行状态更改。默认的实现是：在一个应用系统
        //中只有一个激活的WorkItem，如果参数item的状态为激活，且它是已激活的实例，则激活
item，
        //若已经激活的WorkItem没有终止，则变成冻结状态。
        void ChangeStatus(WorkItem item);
    }
}
```

1.3.4 SimpleWorkItemActivationService

默认实现。激活当前 WorkItem 并使前一个激活的 WorkItem 的状态变为冻结。

```
namespace Microsoft.Practices.CompositeUI
{
    public class SimpleWorkItemActivationService : IWorkItemActivationService
    {
        private object syncroot = new object();
        private WorkItem activeWorkItem;
        public SimpleWorkItemActivationService()
        {
        }
        public void ChangeStatus(WorkItem item)
        {
            lock (syncroot)
```

```
            {
                if (item != activeWorkItem && item.Status == WorkItemStatus.Active)
                {
                    if (activeWorkItem != null && activeWorkItem.Status !=
                        WorkItemStatus.Terminated)
                        activeWorkItem.Deactivate();
                    activeWorkItem = item;
                }
            }
        }
    }
}
```

1.3.5 IWorkItemExtension

WorkItem 扩展，用于监控和执行 WorkItem 生命周期中的事件。

```
using System.ComponentModel;

namespace Microsoft.Practices.CompositeUI
{
    public interface IWorkItemExtension
    {
        //使用需要被扩展的WorkItem进行初始化。如果该扩展实现IComponent接口，
        //该方法在Sited的时候，将会被调用。
        void Initialize(WorkItem workItem);
    }
}
```

1.3.6 WorkItemExtension

WorkItem 扩展的基类。处理了 WorkItem 的 5 个方法。

```
using System;
using Microsoft.Practices.CompositeUI.Utility;

namespace Microsoft.Practices.CompositeUI
{
    public abstract class WorkItemExtension : IWorkItemExtension
    {
        private WorkItem workItem;
        //添加事件处理。
        public void Initialize(WorkItem workItem)
        {
            Guard.ArgumentNotNull(workItem, "workItem");
            this.workItem = workItem;
            workItem.Activated += new EventHandler(OnActivated);
            workItem.Deactivated += new EventHandler(OnDeactivated);
            workItem.Initialized += new EventHandler(OnInitialized);
            workItem.RunStarted += new EventHandler(OnRunStarted);
            workItem.Terminated += new EventHandler(OnTerminated);
        }
        void OnActivated(object sender, EventArgs e)
        {
            OnActivated();
        }
        void OnDeactivated(object sender, EventArgs e)
        {
            OnDeactivated();
        }
        void OnInitialized(object sender, EventArgs e)
        {
            OnInitialized();
```

```
        }
        void OnRunStarted(object sender, EventArgs e)
        {
            OnRunStarted();
        }
        void OnTerminated(object sender, EventArgs e)
        {
            OnTerminated();
        }
        //扩展的WorkItem。
        public WorkItem WorkItem
        {
            get { return workItem; }
        }
        protected virtual void OnActivated()
        {
        }
        protected virtual void OnDeactivated()
        {
        }
        protected virtual void OnInitialized()
        {
        }
        protected virtual void OnRunStarted()
        {
        }
        protected virtual void OnTerminated()
        {
        }
}
```

```
}
```

1.3.7 IConfigurable

从配置文件 Name/Value 序列对接受配置信息的类的接口。

```
using System.Collections.Specialized;

namespace Microsoft.Practices.CompositeUI
{
    public interface IConfigurable
    {
        //根据配置数据来配置组件。
        void Configure(NameValueCollection settings);
    }
}
```

1.4 状态数据（State）

1.4.1 StateException

状态数据异常类。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Serialization;

namespace Microsoft.Practices.CompositeUI
{
    //由StateElement触发的异常类。
    [Serializable]
    public class StateException : Exception
    {
        public StateException() : base() { }
```

```
        public StateException(string message) : base(message) { }
        public StateException(string message, Exception innerException) :
            base(message, innerException) { }
        protected StateException(SerializationInfo info, StreamingContext context) :
            base(info, context) { }
    }
}
```

1.4.2 StateChangedArgs

状态数据变更事件参数。

```
using System;
using System.ComponentModel;
namespace Microsoft.Practices.CompositeUI
{
    //指定数据项ID、新值和旧值。
    public class StateChangedEventArgs : EventArgs
    {
        private object newValue;
        private object oldValue;
        private string key;
        public StateChangedEventArgs(string key, object newValue)
        {
            this.key = key;
            this.newValue = newValue;
        }
        public StateChangedEventArgs(string key, object newValue, object oldValue)
        {
            this.key = key;
            this.newValue = newValue;
```

```
            this.oldValue = oldValue;
        }
        public string Key
        {
            get { return key; }
            set { key = value; }
        }
        [DefaultValue(null)]
        public object NewValue
        {
            get { return newValue; }
            set { this.newValue = value; }
        }
        [DefaultValue(null)]
        public object OldValue
        {
            get { return oldValue; }
            set { oldValue = value; }
        }
    }
}
```

1.4.3 StateChangedTopic

状态变更主题的定义。

```
using System.Globalization;
namespace Microsoft.Practices.CompositeUI
{
    internal static class StateChangedTopic
    {
```

```
        private const string TOPIC_PREFIX = "topic://WorkitemStateChanged/{0}";
        internal static string BuildStateChangedTopicString(string stateName)
        {
            return string.Format(CultureInfo.InvariantCulture, TOPIC_PREFIX, stateName);
        }
    }
}
```

1.4.4 IChangeNotification

一个类将提供数据变更通知。

```
using System;
namespace Microsoft.Practices.CompositeUI
{
    public interface IChangeNotification
    {
        //通知订阅者状态的数据项被改变。
        event EventHandler Changed;
    }
}
```

1.4.5 StateElement

状态类的基类。以词典方式存储数据。当词典的数据发生变更时，将触发通知。能够准确触发变更的集合，需要满足以下条件：当使用 State 写一个项时，State 必须触发通知；当从 State 获取一个项，在 State 外对该项更改时，该项必须触发通知。

```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
```

```
using System.Runtime.Serialization;
using System.Security.Permissions;
using Microsoft.Practices.CompositeUI.Properties;
using Microsoft.Practices.CompositeUI.Utility;
using System.Threading;

namespace Microsoft.Practices.CompositeUI
{
    [Serializable]
    public class StateElement : NameObjectCollectionBase, IChangeNotification, IDisposable
    {
        internal ReaderWriterLock syncRoot = new ReaderWriterLock();
        //StateElemtn保存了（Key，Value）序列，valueIndexed保存了（Value，Key）序列
        //用于快速根据Key得到Value，根据Value得到Key。因此，在Add、Remove方法中，除了
        //维护本类保存的对象之外，还需要维护alueIndexed词典。
        IDictionary<object, string> valueIndexed = new Dictionary<object, string>();
        //状态数据项改变时触发该事件。
        public event EventHandler<StateChangedEventArgs> StateChanged;
        //初始化实例。
        public StateElement()
        {
        }
        //使用序列化信息初始化StateElement类的实例。
        //info：用于收集数据。
        //context：序列化的目的地。
        protected StateElement(SerializationInfo info, StreamingContext context)
            : base(info, context)
        {
        }
        //索引器。用于设置和获取指定Key的数据。
```

```
//继承的类必须使用该索引器来触发StateChanged事件。
protected object this[string key]
{
    get
    {
        Guard.ArgumentNotNull(key, "key");
        //加读锁，然后返回。
        using (new ReaderLock(syncRoot))
        {
            return BaseGet(key);
        }
    }
    set
    {
        Guard.ArgumentNotNull(key, "key");
        //加写锁。
        using (new WriterLock(syncRoot))
        {
            //删除旧数据。
            object oldvalue = BaseGet(key);
            if (oldvalue != null)
            {
                Remove(key);
            }
            //设置新值
            BaseSet(key, value);
            //设置索引
            if (value != null)
            {
                valueIndexed[value] = key;
```

```
                }
                //通过StateElemtn类更改数据项时，必须直接触发通知。
                RaiseStateChanged(key, value, oldvalue);
                //同时，还要为新加的项添加变更通知，以便当该项在State外被更改后
                //能够触发通知。
                if (value != null)
                {
                    HookChangeNotification(value as IChangeNotification);
                }
            }
        }
    }
    //如果该元素支持IChangeNotification，则触发更改通知。
    private void HookChangeNotification(IChangeNotification element)
    {
        if (element != null)
        {
            //首先删除事件通知，在添加，以避免当相同元素传递多次时重复挂载。
            element.Changed -= ChildChanged;
            element.Changed += ChildChanged;
        }
    }
    //卸载挂载的事件。
    private void UnHookChangeNotification(IChangeNotification element)
    {
        if (element != null)
        {
            element.Changed -= ChildChanged;
        }
    }
```

```
private void ChildChanged(object sender, EventArgs e)
{
    if (valueIndexed.ContainsKey(sender))
    {
        RaiseStateChanged(valueIndexed[sender], sender);
    }
}
//删除数据。
public void Remove(string key)
{
    Guard.ArgumentNotNull(key, "key");
    //加锁。
    using (new WriterLock(syncRoot))
    {
        object value = BaseGet(key);
        //卸载通知。
        UnHookChangeNotification(value as IChangeNotification);
        BaseRemove(key);
        //删除索引。
        if (value != null)
        {
            valueIndexed.Remove(value);
        }
    }
}
//触发数据项变更。
public void RaiseStateChanged(string key, object newValue, object oldValue)
{
    OnStateChanged(new StateChangedEventArgs(key, newValue, oldValue));
```

```
        }
        //触发更改。
        public void RaiseStateChanged(string key, object newValue)
        {
            RaiseStateChanged(key, newValue, null);
        }
        //StateChanged的处理。如果StateChanged句柄不为空，则调用。
        protected virtual void OnStateChanged(StateChangedEventArgs e)
        {
            EventHandler<StateChangedEventArgs> stateHandler = StateChanged;
            if (stateHandler != null)
            {
                stateHandler(this, e);
            }
            //该类实现了IChangedNotification接口。因此，需要出发该类的变更事件。
            EventHandler changedHandler = (EventHandler)handlers[changedKey];
            if (changedHandler != null)
            {
                changedHandler(this, EventArgs.Empty);
            }
        }

        #region IChangeNotification 接口的成员
        object changedKey = new object();//新建一个对象作为Key，仅允许一次添加和删除
Handler。
        //处理Handlers列表，实际上最多仅有一个Handler。
        System.ComponentModel.EventHandlerList handlers =
            new System.ComponentModel.EventHandlerList();
        //实现事件数据。
        event EventHandler IChangeNotification.Changed
```

```
        {
            add { handlers.AddHandler(changedKey, value); }
            remove { handlers.RemoveHandler(changedKey, value); }
        }
        #endregion

        #region IDisposable 成员
        //清理资源，并调用垃圾收集。
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        //清理资源
        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                handlers.Dispose();
            }
        }
        #endregion
    }
}
```

1.4.6 State

继承 StateElement。它随着 WorkItem 一起被保存和装载。

```
using System;
using System.ComponentModel;
```

```
using System.Runtime.Serialization;
using System.Security.Permissions;
using System.Diagnostics;
namespace Microsoft.Practices.CompositeUI
{
    [Serializable]
    public class State : StateElement, ISerializable
    {
        private bool hasChanges;
        private string id;
        private TraceSource traceSource = null;
        //跟踪信息。
        [ClassNameTraceSource]
        public TraceSource TraceSource
        {
            set { traceSource = value; }
        }
        //使用随机ID创建一个State。
        public State() : this(Guid.NewGuid().ToString())
        {
        }
        //创建一个State并制定ID。
        public State(string id)
        {
            this.id = id;
        }
        //使用提供的序列化信息初始化State类的实例。
        //SerializationInfo：提供数据收集相关信息。
        //StreamingContext：序列化的目的地。
        protected State(SerializationInfo info, StreamingContext context)
```

```
        : base(info, context)
{
        this.id = (string)info.GetValue("id", typeof(string));
}
//添加序列化属性。
[SecurityPermission(SecurityAction.Demand,
        Flags = SecurityPermissionFlag.SerializationFormatter)]
public override void GetObjectData(SerializationInfo info, StreamingContext context)
{
        base.GetObjectData(info, context);
        info.AddValue("id", this.id);
}
//设置和更改对象。
public new object this[string key]
{
        get { return base[key]; }
        set { base[key] = value; }
}
//接受变更。
public void AcceptChanges()
{
        hasChanges = false;
}
//是否有更新。
[DefaultValue(false)]
public bool HasChanges
{
        get { return hasChanges; }
}
//ID。
```

```
        public string ID
        {
            get { return this.id; }
        }
        //状态变更处理。调用基类方法，设置hasChanges为true。
        protected override void OnStateChanged(StateChangedEventArgs e)
        {
            if (traceSource != null)
                traceSource.TraceInformation(
                    Properties.Resources.TraceStateChangedMessage, id, e.Key);
            base.OnStateChanged(e);
            hasChanges = true;
        }
        //特性。
        class ClassNameTraceSourceAttribute : TraceSourceAttribute
        {
            public ClassNameTraceSourceAttribute() : base(typeof(State).FullName) { }
        }
    }
}
```

1.5 模块（Module）

1.5.1 IModule

模块接口。在模块加载时，会被 IModuleLoaderService 调用。

```
namespace Microsoft.Practices.CompositeUI
{
    public interface IModule
    {
```

```
        //允许在启动时向根WorkItem添加服务。
        void AddServices();
        //允许通知模块已经被加载。
        void Load();
    }
}
```

1.5.2 ModuleInit

IModule 的默认实现。都为空处理。

```
using System;
namespace Microsoft.Practices.CompositeUI
{
    public abstract class ModuleInit : IModule
    {
        public virtual void AddServices()
        {
        }
        public virtual void Load()
        {
        }
    }
}
```

1.5.3 ModuleDependencySolver

模块依赖处理器。ModuleLoaderService 使用该类根据模块间依赖关系获取模块装载顺序。

```
using System;
using System.Collections.Generic;
using System.Globalization;
```

```
using Microsoft.Practices.CompositeUI.Properties;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class ModuleDependencySolver
    {
        //模块依赖矩阵。
        private ListDictionary<string, string> dependencyMatrix =
            new ListDictionary<string, string>();
        //装载的模块列表。
        private List<string> knownModules = new List<string>();
        //添加需要加载的模块的名称。
        public void AddModule(string name)
        {
            Guard.ArgumentNotNullOrEmptyString(name, "name");

            AddToDependencyMatrix(name);
            AddToKnownModules(name);
        }
        //添加模块依赖关系。
        public void AddDependency(string dependingModule, string dependentModule)
        {
            Guard.ArgumentNotNullOrEmptyString(dependingModule, "dependingModule");
            Guard.ArgumentNotNullOrEmptyString(dependentModule, "dependentModule");

            if (!knownModules.Contains(dependingModule))
                throw new ArgumentException(string.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.DependencyForUnknownModule, dependingModule));

            AddToDependencyMatrix(dependentModule);
```

```
        dependencyMatrix.Add(dependentModule, dependingModule);
    }

    private void AddToDependencyMatrix(string module)
    {
        if (!dependencyMatrix.ContainsKey(module))
        {
            dependencyMatrix.Add(module);
        }
    }

    private void AddToKnownModules(string module)
    {
        if (!knownModules.Contains(module))
        {
            knownModules.Add(module);
        }
    }
    //解决依赖关系。
    public string[] Solve()
    {
        List<string> skip = new List<string>();
        while (skip.Count < dependencyMatrix.Count)
        {
            List<string> leaves = this.FindLeaves(skip);
            if (leaves.Count == 0 && skip.Count < dependencyMatrix.Count)
            {
                //抛出循环依赖异常。
                throw new CyclicDependencyFoundException();
            }
```

```
            skip.AddRange(leaves);
        }
        skip.Reverse();

        if (skip.Count > knownModules.Count)
        {
            //抛出缺少模块异常。
            throw new ModuleLoadException(String.Format(CultureInfo.CurrentCulture,
                Resources.DependencyOnMissingModule,FindMissingModules(skip)));
        }

        return skip.ToArray();
    }

    private string FindMissingModules(List<string> skip)
    {
        string missingModules = "";

        foreach (string module in skip)
        {
            if (!knownModules.Contains(module))
            {
                missingModules += ", ";
                missingModules += module;
            }
        }

        return missingModules.Substring(2);
    }
    //模块数量。
```

```
public int ModuleCount
{
    get { return dependencyMatrix.Count; }
}

private List<string> FindLeaves(List<string> skip)
{
    List<string> result = new List<string>();

    foreach (string precedent in dependencyMatrix.Keys)
    {
        if (skip.Contains(precedent))
        {
            continue;
        }

        int count = 0;
        foreach (string dependent in dependencyMatrix[precedent])
        {
            if (skip.Contains(dependent))
            {
                continue;
            }
            count++;
        }
        if (count == 0)
        {
            result.Add(precedent);
        }
    }
```

```
            return result;
        }
    }
}
```

1.5.4 CyclicDependencyFoundException

循环依赖异常。

```
using System;
using System.Runtime.Serialization;
namespace Microsoft.Practices.CompositeUI
{
    [Serializable]
    public class CyclicDependencyFoundException : Exception
    {
        public CyclicDependencyFoundException() : base() { }
        public CyclicDependencyFoundException(string message) : base(message) { }
        public CyclicDependencyFoundException(string message, Exception innerException) :
            base(message, innerException) { }
        protected CyclicDependencyFoundException(SerializationInfo info,
            StreamingContext context) : base(info, context) { }
    }
}
```

1.6 其它类

1.6.1 Controller

控制器和表示器的基类。用于访问 WorkItem 和 State。

```
using Microsoft.Practices.ObjectBuilder;
namespace Microsoft.Practices.CompositeUI
```

```
{
    public class Controller
    {
        private WorkItem workItem;
        public Controller()
        {
        }
        [Dependency(NotPresentBehavior = NotPresentBehavior.ReturnNull)]
        public WorkItem WorkItem
        {
            get { return workItem; }
            set { workItem = value; }
        }
        public State State
        {
            get
            {
                return WorkItem == null ? null : WorkItem.State;
            }
        }
    }
}
```

1.6.2 DesignerCodeDomSerializer

自定义序列化器的辅助类。

```
sing System;
using System.ComponentModel.Design.Serialization;
using System.CodeDom;
using System.Collections.Generic;
```

```
using System.ComponentModel.Design;
using System.ComponentModel;

namespace Microsoft.Practices.CompositeUI
{
    internal class DesignerCodeDomSerializer : CodeDomSerializer
    {
        //查找成员。
        protected static TMember FindMember<TMember>(
            CodeTypeDeclaration declaration, string memberName)
            where TMember : CodeObject
        {
            foreach (CodeTypeMember member in declaration.Members)
            {
                TMember method = member as TMember;
                if (method != null && member.Name == memberName)
                {
                    return method;
                }
            }
            return null;
        }
        //查找InitializeComponent方法的代码。
        protected static CodeMemberMethod FindInitializeComponent(
            CodeTypeDeclaration declaration)
        {
            return FindMember<CodeMemberMethod>(declaration, "InitializeComponent");
        }
        //添加代码到InitializeComponent方法。
        protected static void AddStatementToInitializeComponent(
```

```
        CodeTypeDeclaration declaration, CodeStatement statement)
    {
        CodeMemberMethod init = FindInitializeComponent(declaration);
        if (init != null)
        {
            init.Statements.Add(statement);
        }
    }
    //获取在设计器的组件名称列表。
    protected static List<string> GetComponentsOnDesignerSurface(
        IDesignerSerializationManager manager, object rootComponent)
    {
        //通过设计器宿主，我们可以查找设计器的组件。
        IDesignerHost host = (IDesignerHost)manager.GetService(typeof(IDesignerHost));
        List<string> names = new List<string>(host.Container.Components.Count);
        foreach (IComponent component in host.Container.Components)
        {
            //注意：根节点组件也在列表中。
            if (component != rootComponent)
            {
                names.Add(component.Site.Name);
            }
        }
        return names;
    }
    //根据查询条件，删除代码语句。
    protected static void RemoveFromStatements<TCodeObject>(
        CodeStatementCollection statements, Predicate<TCodeObject> shouldRemove)
        where TCodeObject : CodeObject
    {
```

```
        List<CodeStatement> toRemove = new List<CodeStatement>();
        //查找删除列表。
        foreach (CodeStatement statement in statements)
        {
            TCodeObject typedStatement = statement as TCodeObject;
            CodeExpressionStatement expression = statement as CodeExpressionStatement;
            if (typedStatement == null && expression != null)
            {
                typedStatement = expression.Expression as TCodeObject;
            }

            if (typedStatement != null && shouldRemove(typedStatement))
            {
                toRemove.Add(statement);
            }
        }
        //执行删除。
        foreach (CodeStatement statement in toRemove)
        {
            statements.Remove(statement);
        }
    }
    //从InitializeComponent方法删除代码。
    protected static void RemoveFromInitializeComponent<TCodeObject>(
        CodeTypeDeclaration declaration, Predicate<TCodeObject> shouldRemove)
        where TCodeObject : CodeObject
    {
        CodeMemberMethod initialize = FindInitializeComponent(declaration);

        if (initialize != null)
```

```
        {
            RemoveFromStatements<TCodeObject>(initialize.Statements, shouldRemove);
        }
    }
    //查找代码。
    protected static List<TCodeObject> FindCode<TCodeObject>(
        CodeStatementCollection statements, Predicate<TCodeObject> matchesFilter)
        where TCodeObject : CodeObject
    {
        List<TCodeObject> values = new List<TCodeObject>();
        foreach (CodeStatement statement in statements)
        {
            TCodeObject typedStatement = statement as TCodeObject;
            CodeExpressionStatement expression = statement as CodeExpressionStatement;
            if (typedStatement == null && expression != null)
            {
                typedStatement = expression.Expression as TCodeObject;
            }

            if (typedStatement != null && matchesFilter(typedStatement))
            {
                values.Add(typedStatement);
            }
        }

        return values;
    }
    //查找第一个匹配的代码。
    protected static TCodeObject FindFirstCode<TCodeObject>(
        CodeStatementCollection statements, Predicate<TCodeObject> matchesFilter)
```

```
            where TCodeObject : CodeObject
        {
            foreach (CodeStatement statement in statements)
            {
                TCodeObject typedStatement = statement as TCodeObject;
                CodeExpressionStatement expression = statement as CodeExpressionStatement;
                if (typedStatement == null && expression != null)
                {
                    typedStatement = expression.Expression as TCodeObject;
                }

                if (typedStatement != null && matchesFilter(typedStatement))
                {
                    return typedStatement;
                }
            }

            return null;
        }
    }
}
```

1.6.3 DictionaryEventArgs

词典事件参数类。

```
using System;
using System.Collections.Generic;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI
{
```

```
    public class DictionaryEventArgs : DataEventArgs<Dictionary<string, object>>
    {
        public DictionaryEventArgs() : base(new Dictionary<string, object>())
        {
        }
        public override string ToString()
        {
            string[] values = new string[Data.Count];
            int i = 0;
            foreach (KeyValuePair<String, object> pair in Data)
                values[i++] = pair.Key + ": " + pair.Value;
            return String.Join(", ", values);
        }
    }
}
```

2 策略（Strategies）

2.1 BuilderStrategyException

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Serialization;

namespace Microsoft.Practices.CompositeUI.BuilderStrategies
{
    [Serializable]
    public class BuilderStrategyException : Exception
    {
```

```
        public BuilderStrategyException() : base() { }
        public BuilderStrategyException(string message) : base(message) { }
        public BuilderStrategyException(string message, Exception innerException) :
            base(message, innerException) { }
        protected BuilderStrategyException(SerializationInfo info,
            StreamingContext context) : base(info, context) { }
    }
}
```

2.2 CommandStrategy

命令策略。用于将声明了 CommandHandler 特性的方法作为命令注册到 WorkItem 的 Commands 集合。命令处理分为 2 个步骤：

（1）一个类实现了命令的处理句柄，并声明了 CommandHandlerAttribute 特性。

（2）调用 WorkItem.Commands[“CommandName”].AddInvoker(item, “Click”)来添加调用者。即 item 的 Click 事件由指定的命令来处理。

```
using System;
using System.Reflection;
using Microsoft.Practices.CompositeUI.Commands;
using Microsoft.Practices.ObjectBuilder;
using System.Globalization;

namespace Microsoft.Practices.CompositeUI.BuilderStrategies
{
    public class CommandStrategy : BuilderStrategy
    {
        public override object BuildUp(
            IBuilderContext context, Type t, object existing, string id)
        {
            //获取WorkItem。
```

```
        WorkItem workItem = GetWorkItem(context, existing);
        //如果为空，则不处理。
        if (workItem != null)
        {
            Type targetType = existing.GetType();
            //注册命令句柄。
            foreach (MethodInfo methodInfo in targetType.GetMethods())
                RegisterCommandHandlers(context, workItem, existing, id, methodInfo);
        }

        return base.BuildUp(context, t, existing, id);
    }
    public override object TearDown(IBuilderContext context, object item)
    {
        WorkItem workItem = GetWorkItem(context, item);
        if (workItem != null)
        {
            Type targetType = item.GetType();
            //注销命令句柄。
            foreach (MethodInfo methodInfo in targetType.GetMethods())
                UnregisterCommandHandlers(context, workItem, item, methodInfo);
        }
        return base.TearDown(context, item);
    }
    //为一个声明了CommandHandler特性的方法声明句柄。
    private void RegisterCommandHandlers(
        IBuilderContext context, WorkItem workItem, object target,
        string targetID, MethodInfo methodInfo)
    {
        //获取CommandHandler特性
```

```
        foreach (CommandHandlerAttribute attr in
            methodInfo.GetCustomAttributes(typeof(CommandHandlerAttribute), true))
        {
            //如果在Locator中不存在该实例，会使用indexedCreateDelegate
            //创建一个Command实例并构建。
            Command cmd = workItem.Commands[attr.CommandName];
            //注册命令。
            if (!cmd.IsHandlerRegistered(target, methodInfo))
            {
                //创建该命令对应的EventHandler。
                //即创建了声明CommandHandler特性的方法的代理。
                cmd.ExecuteAction += CreateCommandHandler(target, methodInfo);
                TraceBuildUp(context, target.GetType(), targetID,
                    Properties.Resources.CommandInjectionBuildUp,
                    methodInfo.Name, attr.CommandName);
            }
        }
    }
    private void UnregisterCommandHandlers(
        IBuilderContext context, WorkItem workItem, object target, MethodInfo methodInfo)
    {
        foreach (CommandHandlerAttribute attr in
            methodInfo.GetCustomAttributes(typeof(CommandHandlerAttribute), true))
        {
            //注销命令。
            if (workItem.Commands.Contains(attr.CommandName))
            {
                workItem.Commands[attr.CommandName].ExecuteAction -=
                    CreateCommandHandler(target, methodInfo);
                TraceTearDown(context, target,
```

```
                    Properties.Resources.CommandInjectionTearDown,
                    methodInfo.Name, attr.CommandName);
            }
        }
    }
    //创建一个到target对象，指定方法methodinfo的代理。
    private EventHandler CreateCommandHandler(object target, MethodInfo methodInfo)
    {
        Delegate handler = null;
        if (methodInfo.IsStatic == false)
        {
            handler = Delegate.CreateDelegate(
                typeof(EventHandler), target, (MethodInfo)methodInfo);
        }
        else
        {
            throw new CommandException(
                string.Format(CultureInfo.CurrentCulture,
                Properties.Resources.StaticCommandHandlerNotSupported,
                methodInfo.Name));
        }
        return (EventHandler)handler;
    }
    //获取WorkItem。
    private static WorkItem GetWorkItem(IBuilderContext context, object item)
    {
        if (item is WorkItem)
            return item as WorkItem;
        return context.Locator.Get<WorkItem>(new
            DependencyResolutionLocatorKey(typeof(WorkItem), null));
```

```
        }
    }
}
```

2.3 EventBrokerStrategy

事件代理策略。用于处理标记了 EventPublication 和 EventSubscription 特性的成员。

```
using System;
using Microsoft.Practices.CompositeUI.EventBroker;
using Microsoft.Practices.ObjectBuilder;

namespace Microsoft.Practices.CompositeUI.BuilderStrategies
{
    public class EventBrokerStrategy : BuilderStrategy
    {
        public override object BuildUp(IBuilderContext context, Type t, object existing,
            string id)
        {
            WorkItem workItem = GetWorkItem(context, existing);
            if (workItem != null)  //注册发布和订阅。
                EventInspector.Register(existing, workItem);
            return base.BuildUp(context, t, existing, id);
        }
        public override object TearDown(IBuilderContext context, object item)
        {
            WorkItem workItem = GetWorkItem(context, item);
            if (workItem != null)  //卸载发布和订阅。
                EventInspector.Unregister(item, workItem);
            return base.TearDown(context, item);
        }
```

```
        //获取WorkItem。
        private static WorkItem GetWorkItem(IBuilderContext context, object item)
        {
            if (item is WorkItem)
                return item as WorkItem;
            return context.Locator.Get<WorkItem>(new DependencyResolutionLocatorKey(
                typeof(WorkItem), null));
        }
    }
}
```

2.4 ObjectNotificationPolicy

对象通知政策。

```
using System;
using System.Collections.Generic;
using Microsoft.Practices.ObjectBuilder;
namespace Microsoft.Practices.CompositeUI.BuilderStrategies
{
    public class ObjectBuiltNotificationPolicy : IBuilderPolicy
    {
        //项目操作通知代理，如OnAdded，OnRemoved等。
        public delegate void ItemNotification(object item);
        //添加操作的通知代理。
        public Dictionary<WorkItem, ItemNotification> AddedDelegates = new
            Dictionary<WorkItem, ItemNotification>();
        //删除操作的通知代理。
        public Dictionary<WorkItem, ItemNotification> RemovedDelegates = new
            Dictionary<WorkItem, ItemNotification>();
    }
```

```
}
```

2.5 ObjectNotificationStrategy

```
using System;
using Microsoft.Practices.ObjectBuilder;

namespace Microsoft.Practices.CompositeUI.BuilderStrategies
{
    public class ObjectBuiltNotificationStrategy : BuilderStrategy
    {
        //通知政策。
        ObjectBuiltNotificationPolicy policy;
        public override object BuildUp(IBuilderContext context, Type typeToBuild,
            object existing, string idToBuild)
        {
            //获取WorkItem。
            WorkItem workItem = context.Locator.Get<WorkItem>(
                new DependencyResolutionLocatorKey(typeof(WorkItem), null));
            ObjectBuiltNotificationPolicy.ItemNotification notification;
            //获取对象通知政策。
            if (policy == null)
                policy = context.Policies.Get<ObjectBuiltNotificationPolicy>(null, null);
            //通知一个项添加到WorkItem。
            if (workItem != null && !Object.ReferenceEquals(workItem, existing) &&
                policy.AddedDelegates.TryGetValue(workItem, out notification))
                notification(existing);

            return base.BuildUp(context, typeToBuild, existing, idToBuild);
        }
```

```
        //从WorkItem对象删除时通知。
        public override object TearDown(IBuilderContext context, object item)
        {
            WorkItem workItem = context.Locator.Get<WorkItem>(
                new DependencyResolutionLocatorKey(typeof(WorkItem), null));
            ObjectBuiltNotificationPolicy.ItemNotification notification;
            //获取对象删除通知代理。
            if (workItem != null && policy.RemovedDelegates.TryGetValue(
                workItem, out notification))
                notification(item);

            return base.TearDown(context, item);
        }
    }
}
```

2.6 RootWorkItemInitializationStrategy

```
using System;
using Microsoft.Practices.ObjectBuilder;

namespace Microsoft.Practices.CompositeUI.BuilderStrategies
{
    internal class RootWorkItemInitializationStrategy : BuilderStrategy
    {
        //根节点初始化回调。
        public delegate void RootWorkItemInitializationCallback();
        //代理。
        RootWorkItemInitializationCallback callback;
        public RootWorkItemInitializationStrategy(
```

```
            RootWorkItemInitializationCallback callback)
        {
            this.callback = callback;
        }
        public override object BuildUp(IBuilderContext context, Type typeToBuild,
            object existing, string idToBuild)
        {
            WorkItem wi = existing as WorkItem;

            if (wi != null && wi.Parent == null)
                callback();

            return base.BuildUp(context, typeToBuild, existing, idToBuild);
        }
    }
}
```

3 集合类（Collections）

3.1 ManagedObjectCollection

WorkItem 的 Locator 的对象的过滤集合。

在 WorkItem 中的 Commands、SmartParts、WorkItems、Workspaces、Items 和 EventTopics 都是 ManagedObjectCollection 类的实例。SmartParts 集合对应的过滤条件是声明了 SmartPart 特性，其它都不设置过滤条件。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using Microsoft.Practices.CompositeUI.Utility;
```

```
using Microsoft.Practices.ObjectBuilder;
using Microsoft.Practices.CompositeUI.Commands;
using Microsoft.Practices.CompositeUI.EventBroker;

namespace Microsoft.Practices.CompositeUI.Collections
{
    public class ManagedObjectCollection<TItem> :
        ICollection, IEnumerable<KeyValuePair<string, TItem>>
    {
        private ILifetimeContainer container;  //声明周期容器。
        private IReadWriteLocator locator;  //Locator。
        private IBuilder<BuilderStage> builder;  //Builder。
        private SearchMode searchMode;  //查找模式。
        private IndexerCreationDelegate indexerCreationDelegate;  //创建代理。当对象不存
在，
                                                //便使用其创建一个对象。
        private Predicate<TItem> filter;  //过滤条件。
        private ManagedObjectCollection<TItem> parentCollection;  //父管理对象集合。
        private WorkItem workItem;   //所属WorkItem。
        //该代理用于自动创建一个对象，当这个对象不存在但在索引中。
        public delegate TItem IndexerCreationDelegate(Type t, string id);
        //初始化一个管理对象集合。父管理对象集合为null。
        public ManagedObjectCollection(
            ILifetimeContainer container, IReadWriteLocator locator,
            IBuilder<BuilderStage> builder, SearchMode searchMode,
            IndexerCreationDelegate indexerCreationDelegate, Predicate<TItem> filter)
            : this(
            container, locator, builder, searchMode, indexerCreationDelegate, filter, null)
        {
        }
```

```
public ManagedObjectCollection(ILifetimeContainer container,
    IReadWriteLocator locator, IBuilder<BuilderStage> builder,
    SearchMode searchMode, IndexerCreationDelegate indexerCreationDelegate,
    Predicate<TItem> filter, ManagedObjectCollection<TItem> parentCollection)
{
    this.container = container;
    this.locator = locator;
    this.builder = builder;
    this.searchMode = searchMode;
    this.indexerCreationDelegate = indexerCreationDelegate;
    this.filter = filter;
    this.parentCollection = parentCollection;
    this.workItem = locator.Get<WorkItem>(
        new DependencyResolutionLocatorKey(typeof(WorkItem), null));
    if (this.workItem != null)
    {
        this.workItem.ObjectAdded += new EventHandler<DataEventArgs<object>>(
            WorkItem_ItemAdded);
        this.workItem.ObjectRemoved += new EventHandler<DataEventArgs<object>>(
            WorkItem_ItemRemoved);
    }
}
//对象添加到集合时触发。
public event EventHandler<DataEventArgs<TItem>> Added;
//对象被删除或清理时触发。
public event EventHandler<DataEventArgs<TItem>> Removed;
//按ID获取一个对象。如果不存在时，若indexerCreationDelegate不为空，则利用
//该代理创建一个对象、将该对象添加到集合，并返回。
public TItem this[string id]
{
```

```
        get
        {
            TItem result = Get(id);
            if ((object)result == null && indexerCreationDelegate != null &&
                !Contains(id, searchMode, false))
            {
                result = indexerCreationDelegate(typeof(TItem), id);
                Add(result, id);
            }

            return result;
        }
    }
    //Locator中，某类对象TItem的对象符合过滤条件的对象的数目。
    //如果指定SearchMode为Up，则包括父管理对象的集合。
    public int Count
    {
        get
        {
            int result = 0;
            //当前Locator符合条件的数目。
            foreach (KeyValuePair<object, object> pair in locator)
            {
                DependencyResolutionLocatorKey key =
                    pair.Key as DependencyResolutionLocatorKey;

                if (key != null && key.ID != null && pair.Value is TItem &&
                    PassesFilter(pair.Value))
                    result++;
            }
```

```
            //父Locator符合条件的数目。
            if (searchMode == SearchMode.Up && parentCollection != null)
                result += parentCollection.Count;
            return result;
        }
    }
    //添加一项，ID为空，主键为TItem的类型。
    public void Add(TItem item)
    {
        Add(item, null);
    }
    //添加一项。
    public void Add(TItem item, string id)
    {
        Guard.ArgumentNotNull(item, "item");
        Build(item.GetType(), id, item);
    }
    //创建并添加。ID为null。
    public TItem AddNew(Type typeToBuild)
    {
        return AddNew(typeToBuild, null);
    }
    //添加并创建一项，即直接使用Builder来构建一个对象。
    public TItem AddNew(Type typeToBuild, string id)
    {
        return Build(typeToBuild, id, null);
    }
    public TTypeToBuild AddNew<TTypeToBuild>()
        where TTypeToBuild : TItem
    {
```

```
        return (TTypeToBuild)AddNew(typeof(TTypeToBuild), null);
    }
    public TTypeToBuild AddNew<TTypeToBuild>(string id)
        where TTypeToBuild : TItem
    {
        return (TTypeToBuild)AddNew(typeof(TTypeToBuild), id);
    }
    //是否包含指定ID的对象。类型为TItem。
    public bool Contains(string id)
    {
        return Get(id) != null;
    }
    //是否包含指定ID的对象。类型为ITem。
    private bool Contains(string id, SearchMode searchMode, bool filtered)
    {
        return Get(id, searchMode, filtered) != null;
    }
    //判断在声明周期容器中，是否包含该项。
    public bool ContainsObject(TItem item)
    {
        return container.Contains(item);
    }
    //查找所有继承了TSearchType的对象。
    public ICollection<TSearchType> FindByType<TSearchType>()
        where TSearchType : TItem
    {
        List<TSearchType> result = new List<TSearchType>();
        foreach (object obj in container)
            if (typeof(TSearchType).IsAssignableFrom(obj.GetType()))
                result.Add((TSearchType)obj);
```

```
        if (searchMode == SearchMode.Up && parentCollection != null)
            result.AddRange(parentCollection.FindByType<TSearchType>());
        return result;
    }
    public ICollection<TItem> FindByType(Type searchType)
    {
        List<TItem> result = new List<TItem>();

        foreach (object obj in container)
            if (searchType.IsAssignableFrom(obj.GetType()))
                result.Add((TItem)obj);

        if (searchMode == SearchMode.Up && parentCollection != null)
            result.AddRange(parentCollection.FindByType(searchType));

        return result;
    }
    public TItem Get(string id)
    {
        return Get(id, searchMode, true);
    }
    //查找指定ID的对象。
    private TItem Get(string id, SearchMode searchMode, bool filtered)
    {
        Guard.ArgumentNotNull(id, "id");
        //在本地查找，如果找到，则返回。
        foreach (KeyValuePair<object, object> pair in locator)
        {
            if (pair.Key is DependencyResolutionLocatorKey)
```

```
            {
                DependencyResolutionLocatorKey depKey =
                    (DependencyResolutionLocatorKey)pair.Key;

                if (object.Equals(depKey.ID, id))
                {
                    TItem result = (TItem)pair.Value;
                    if (!filtered || filter == null || filter(result))
                        return result;
                }
            }
        }
        //在父容器查找，如果找到，返回。
        if (searchMode == SearchMode.Up && parentCollection != null)
            return parentCollection.Get(id);
        //返回空值。
        return default(TItem);
    }
    public TTypeToGet Get<TTypeToGet>(string id)
        where TTypeToGet : TItem
    {
        return (TTypeToGet)Get(id);
    }
    //删除一个对象。
    public void Remove(TItem item)
    {
        if (container.Contains(item))
        {
            ThrowIfItemRemovalIsNotPermitted(item);
            //利用Builder清理该对象。
```

```
            builder.TearDown(locator, item);
            //从声明周期容器中删除。
            container.Remove(item);
            //从Locator中删除。
            List<object> keysToRemove = new List<object>();
            foreach (KeyValuePair<object, object> pair in locator)
                if (pair.Value.Equals(item))
                    keysToRemove.Add(pair.Key);
            foreach (object key in keysToRemove)
                locator.Remove(key);
        }
    }
    //如果不允许删除，则抛出异常。WorkItem、Command和EventTopic不能删除。
    private static void ThrowIfItemRemovalIsNotPermitted(TItem item)
    {
        if (item is WorkItem)  //WorkItem不能删除。
            throw new ArgumentException(
            Properties.Resources.NoRemoveWorkItemFromManagedObjectCollection, "item");
        if (item is Command)
        {
            Command cmd = item as Command;
            throw new ArgumentException(String.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.RemoveCommandFromWorkItemIsNotPermitted,
                    cmd.Name), "item");
        }
        if (item is EventTopic)
        {
            EventTopic topic = item as EventTopic;
            throw new ArgumentException(String.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.RemoveEventTopicFromWorkItemIsNotPermitted,
```

```
                    topic.Name), "item");
        }
    }
    //构建一个Item。
    private TItem Build(Type typeToBuild, string idToBuild, object item)
    {
        //如果重复，则抛出异常。
        if (idToBuild != null && Contains(idToBuild, SearchMode.Local, true))
            throw new ArgumentException(String.Format(
            CultureInfo.CurrentCulture, Properties.Resources.DuplicateID, idToBuild));
        //如果当前item是否WorkItem，则抛出异常。
        if (item != null && Object.ReferenceEquals(item, locator.Get(
            new DependencyResolutionLocatorKey(typeof(WorkItem), null))))
            throw new ArgumentException(
            Properties.Resources.CannotAddWorkItemToItself, "item");
        //构建该项目。如果该项目在声明周期容器中存在，则不需要再构建，仅添加项。
        if (item == null)
            item = BuildFirstTimeItem(typeToBuild, idToBuild, null);
        else if (!container.Contains(item))
            item = BuildFirstTimeItem(typeToBuild, idToBuild, item);
        else
            BuildRepeatedItem(typeToBuild, idToBuild, item);
        return (TItem)item;
    }
    //使用Builder从新构建。
    private object BuildFirstTimeItem(Type typeToBuild, string idToBuild, object item)
    {
        return builder.BuildUp(locator, typeToBuild, NormalizeID(idToBuild), item);
    }
    //添加到Locator。
```

```
private void BuildRepeatedItem(Type typeToBuild, string idToBuild, object item)
{
    locator.Add(new DependencyResolutionLocatorKey(typeToBuild,
        NormalizeID(idToBuild)), item);
}
//获取迭代器，返回子Locator到根Locator所有满足条件的TItem。
//注意，如果在子Locator包含与父Locator相同ID的TItem，则仅返回子Locator的对象。
public IEnumerator<KeyValuePair<string, TItem>> GetEnumerator()
{
    IReadableLocator currentLocator = locator;
    Dictionary<string, object> seenNames = new Dictionary<string, object>();
    do
    {
        foreach (KeyValuePair<object, object> pair in currentLocator)
            if (pair.Key is DependencyResolutionLocatorKey)
            {
                DependencyResolutionLocatorKey depKey =
                    (DependencyResolutionLocatorKey)pair.Key;
                if (depKey.ID != null && pair.Value is TItem)
                {
                    string id = depKey.ID;
                    TItem value = (TItem)pair.Value;

                    if (!seenNames.ContainsKey(id) &&
                        (filter == null || filter(value)))
                    {
                        seenNames[id] = String.Empty;
                        yield return new KeyValuePair<string, TItem>(id, value);
                    }
                }
```

```
                }
            }
            while (searchMode == SearchMode.Up &&
                (currentLocator = currentLocator.ParentLocator) != null);
        }
        //迭代器。
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
        //是否满足过滤条件。
        private bool PassesFilter(object item)
        {
            if (filter != null)
                return filter((TItem)item);
            else
                return true;
        }
        //如果idToBuild为null，则创建一个新的GUID。
        private string NormalizeID(string idToBuild)
        {
            return idToBuild ?? Guid.NewGuid().ToString();
        }
        //当添加一个Item时，如果在当前Locator存在该对象，且满足过滤条件，则触发Added事件。
        private void WorkItem_ItemAdded(object sender, DataEventArgs<object> e)
        {
            if (Added != null && e.Data is TItem)
            {
                TItem value = (TItem)e.Data;
                DependencyResolutionLocatorKey key = FindObjectInLocator(value);
```

```
            if (key != null && key.ID != null && PassesFilter(value))
                Added(this, new DataEventArgs<TItem>(value));
        }
    }
    //删除一个对象时，如果是删除指定条件的TItem对象，则触发事件。
    private void WorkItem_ItemRemoved(object sender, DataEventArgs<object> e)
    {
        if (Removed != null && e.Data is TItem)
        {
            TItem value = (TItem)e.Data;
            DependencyResolutionLocatorKey key = FindObjectInLocator(value);

            if (key != null && key.ID != null && PassesFilter(value))
                Removed(this, new DataEventArgs<TItem>(value));
        }
    }
    //查找对象。
    private DependencyResolutionLocatorKey FindObjectInLocator(object value)
    {
        foreach (KeyValuePair<object, object> pair in locator)
            if (pair.Key is DependencyResolutionLocatorKey && pair.Value == value)
                return ((DependencyResolutionLocatorKey)pair.Key);

        return null;
    }
    //不支持拷贝。
    void ICollection.CopyTo(Array array, int index)
    {
        throw new NotImplementedException();
    }
```

```
        //个数。
        int ICollection.Count
        {
            get { return Count; }
        }
        //不同步。
        bool ICollection.IsSynchronized
        {
            get { return false; }
        }
        //不支持。
        object ICollection.SyncRoot
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

3.2 ServiceCollection

服务集合。服务是主键为类型的集合。服务集合用于负责对 Locator 中服务的操作。

```
using System;
using System.Collections.Generic;
using Microsoft.Practices.CompositeUI.Services;
using Microsoft.Practices.CompositeUI.Utility;
using Microsoft.Practices.ObjectBuilder;
using System.Collections;
using System.Globalization;
namespace Microsoft.Practices.CompositeUI.Collections
```

```
{
    //包含在WorkItem中的服务。
    public class ServiceCollection : ICollection, IEnumerable<KeyValuePair<Type, object>>
    {
        private IBuilder<BuilderStage> builder;
        private IReadWriteLocator locator;
        private ILifetimeContainer container;
        private ServiceCollection parent;  //父集合。
        //初始化一个服务集合实例。
        public ServiceCollection(
            ILifetimeContainer container, IReadWriteLocator locator,
            IBuilder<BuilderStage> builder, ServiceCollection parent)
        {
            this.builder = builder;
            this.container = container;
            this.locator = locator;
            this.parent = parent;
        }
        //当服务添加时，将触发事件。对于Demand-add服务，只有在真正创建时才触发。
        public event EventHandler<DataEventArgs<object>> Added;
        //仅当删除时才触发。清理对象不会引发。
        public event EventHandler<DataEventArgs<object>> Removed;
        //添加服务并构建。
        public void Add<TService>(TService serviceInstance)
        {
            Add(typeof(TService), serviceInstance);
        }
        public void Add(Type serviceType, object serviceInstance)
        {
            Guard.ArgumentNotNull(serviceType, "serviceType");
```

```
        Guard.ArgumentNotNull(serviceInstance, "serviceInstance");
        Build(serviceInstance.GetType(), serviceType, serviceInstance);
    }
    //添加按需的服务。
    public void AddOnDemand<TService>()
    {
        AddOnDemand(typeof(TService), null);
    }
    public void AddOnDemand<TService, TRegisterAs>()
    {
        AddOnDemand(typeof(TService), typeof(TRegisterAs));
    }
    public void AddOnDemand(Type serviceType)
    {
        AddOnDemand(serviceType, null);
    }
    //添加按需服务。
    public void AddOnDemand(Type serviceType, Type registerAs)
    {
        Guard.ArgumentNotNull(serviceType, "serviceType");
        if (registerAs == null)
            registerAs = serviceType;
        DependencyResolutionLocatorKey key = new
            DependencyResolutionLocatorKey(registerAs, null);
        if (locator.Contains(key, SearchMode.Local))
            throw new ArgumentException(String.Format(
            CultureInfo.CurrentCulture,
            Properties.Resources.DuplicateService, registerAs.FullName));
        //直接添加一个DemandAddPlaceholder对象到Locator。
        DemandAddPlaceholder placeholder = new DemandAddPlaceholder(serviceType);
```

```
        locator.Add(key, placeholder);
        container.Add(placeholder);
    }
    //添加并创建服务。
    public TService AddNew<TService>()
    {
        return AddNew<TService, TService>();
    }
    public TService AddNew<TService, TRegisterAs>()
        where TService : TRegisterAs
    {
        return (TService)AddNew(typeof(TService), typeof(TRegisterAs));
    }
    public object AddNew(Type serviceType)
    {
        return AddNew(serviceType, serviceType);
    }
    public object AddNew(Type serviceType, Type registerAs)
    {
        Guard.ArgumentNotNull(serviceType, "serviceType");
        Guard.ArgumentNotNull(registerAs, "registerAs");

        return Build(serviceType, registerAs, null);
    }
    //包含服务判断。
    public bool Contains<TService>()
    {
        return Contains(typeof(TService));
    }
    public bool Contains(Type serviceType)
```

```
{
    return Contains(serviceType, SearchMode.Up);
}
//调用Locator.Contains返回。
private bool Contains(Type serviceType, SearchMode searchMode)
{
    Guard.ArgumentNotNull(serviceType, "serviceType");
    return locator.Contains(
        new DependencyResolutionLocatorKey(serviceType, null), searchMode);
}
//本地是否存在。
public bool ContainsLocal(Type serviceType)
{
    return Contains(serviceType, SearchMode.Local);
}
//获取服务。
public TService Get<TService>()
{
    return (TService)Get(typeof(TService), false);
}
public TService Get<TService>(bool ensureExists)
{
    return (TService)Get(typeof(TService), ensureExists);
}
public object Get(Type serviceType)
{
    return Get(serviceType, false);
}
public object Get(Type serviceType, bool ensureExists)
{
```

```
        Guard.ArgumentNotNull(serviceType, "serviceType");
        //如果本地存在。
        if (Contains(serviceType, SearchMode.Local))
        {
            //获取结果。
            object result = locator.Get(
                new DependencyResolutionLocatorKey(serviceType, null));
            //如果是按需服务，则创建该对象。
            DemandAddPlaceholder placeholder = result as DemandAddPlaceholder;
            if (placeholder != null)
            {
                Remove(serviceType);
                result = Build(placeholder.TypeToCreate, serviceType, null);
            }

            return result;
        }
        //如果Parent不为空，则从父Locator读取。
        if (parent != null)
            return parent.Get(serviceType, ensureExists);
        //如果确保存在却不存在，则抛出异常。
        if (ensureExists)
            throw new ServiceMissingException(serviceType);
        //返回空。
        return null;
    }
    //删除服务。
    public void Remove<TService>()
    {
        Remove(typeof(TService));
```

```
}
//删除指定类型的服务。如果是最后一个服务实例，则从声明周期容器删除。
public void Remove(Type serviceType)
{
    Guard.ArgumentNotNull(serviceType, "serviceType");
    DependencyResolutionLocatorKey key =
        new DependencyResolutionLocatorKey(serviceType, null);
    //如果本地存在。
    if (locator.Contains(key, SearchMode.Local))
    {
        object serviceInstance = locator.Get(key, SearchMode.Local);
        bool isLastInstance = true;
        //从集合移除。
        locator.Remove(new DependencyResolutionLocatorKey(serviceType, null));
        //是否有其它实例。
        foreach (KeyValuePair<object, object> kvp in locator)
        {
            if (ReferenceEquals(kvp.Value, serviceInstance))
            {
                isLastInstance = false;
                break;
            }
        }
        //如果不存在其他实例
        if (isLastInstance)
        {
            //清理并从声明周期容器删除。
            builder.TearDown(locator, serviceInstance);
            container.Remove(serviceInstance);
            //如果服务已经创建，则触发OnRemoved事件。
```

```
                if (!(serviceInstance is DemandAddPlaceholder))
                    OnRemoved(serviceInstance);
            }
        }
    }
    //OnAdded事件。
    protected virtual void OnAdded(object service)
    {
        if (Added != null)
        {
            Added(this, new DataEventArgs<object>(service));
        }
    }
    //OnRemoved。
    protected virtual void OnRemoved(object service)
    {
        if (Removed != null)
        {
            Removed(this, new DataEventArgs<object>(service));
        }
    }
    //构建服务。
    private object Build(Type typeToBuild, Type typeToRegisterAs, object serviceInstance)
    {
        Guard.TypeIsAssignableFromType(typeToBuild, typeToRegisterAs, "typeToBuild");

        if (locator.Contains(new DependencyResolutionLocatorKey(
            typeToRegisterAs, null), SearchMode.Local))
            throw new ArgumentException(string.Format(CultureInfo.CurrentCulture,
            Properties.Resources.DuplicateService, typeToRegisterAs.FullName));
```

```
        //第一次构建
        if (serviceInstance == null)
            serviceInstance = BuildFirstTimeItem(typeToBuild, typeToRegisterAs, null);
        else if (!container.Contains(serviceInstance))
            serviceInstance = BuildFirstTimeItem(
                typeToBuild, typeToRegisterAs, serviceInstance);
        else  //重复构建。
            BuildRepeatedItem(typeToRegisterAs, serviceInstance);

        return serviceInstance;
    }
    //第一次构建。
    private object BuildFirstTimeItem(
        Type typeToBuild, Type typeToRegisterAs, object item)
    {
        item = builder.BuildUp(locator, typeToBuild, null, item);
        if (typeToRegisterAs != typeToBuild)
        {
            locator.Add(new DependencyResolutionLocatorKey(
                typeToRegisterAs, null), item);
            locator.Remove(new DependencyResolutionLocatorKey(typeToBuild, null));
        }
        //OnAdded事件。
        OnAdded(item);
        return item;
    }
    //重复构建。
    private void BuildRepeatedItem(Type typeToRegisterAs, object item)
    {
        locator.Add(new DependencyResolutionLocatorKey(typeToRegisterAs, null), item);
```

```
}
//相当于代理对象。
private class DemandAddPlaceholder
{
    Type typeToCreate;
    public DemandAddPlaceholder(Type typeToCreate)
    {
        this.typeToCreate = typeToCreate;
    }
    public Type TypeToCreate
    {
        get { return typeToCreate; }
    }
}
//返回所有主键为Type + Null ID的对象。
public IEnumerator<KeyValuePair<Type, object>> GetEnumerator()
{
    IReadableLocator currentLocator = locator;
    Dictionary<Type, object> seenTypes = new Dictionary<Type, object>();
    do
    {
        foreach (KeyValuePair<object, object> pair in currentLocator)
            if (pair.Key is DependencyResolutionLocatorKey)
            {
                DependencyResolutionLocatorKey depKey =
                    (DependencyResolutionLocatorKey)pair.Key;
                if (depKey.ID == null)
                {
                    Type type = depKey.Type;
                    if (!seenTypes.ContainsKey(type))
```

```
                        {
                            seenTypes[type] = String.Empty;
                            yield return new KeyValuePair<Type, object>(
                                type, pair.Value);
                        }
                    }
                }
            }
            while ((currentLocator = currentLocator.ParentLocator) != null);
        }
        //迭代器。
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
        //没有实现。
        void ICollection.CopyTo(Array array, int index)
        {
            throw new NotImplementedException();
        }
        //没有实现。
        int ICollection.Count
        {
            get { throw new NotImplementedException(); }
        }
        //不同步。
        bool ICollection.IsSynchronized
        {
            get { return false; }
        }
```

```
        //没有实现。
        object ICollection.SyncRoot
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

3.3 UIExtensionSite

UI 元素的扩展站点。

```
using System;
using System.Collections;
using System.Collections.Generic;
using Microsoft.Practices.CompositeUI.UIElements;

namespace Microsoft.Practices.CompositeUI
{
    public class UIExtensionSite : IEnumerable<object>
    {
        //站点的Adapter。
        IUIElementAdapter adapter;
        //UI元素。
        List<object> items = new List<object>();
        //设置Adapter，并创建站点。
        public UIExtensionSite(IUIElementAdapter adapter)
        {
            this.adapter = adapter;
        }
        //站点元素的数量。
```

```
public int Count
{
    get { return items.Count; }
}
//添加一个界面元素，并使用Adapter添加到UI上。
public TElement Add<TElement>(TElement uiElement)
{
    TElement element = (TElement)adapter.Add(uiElement);
    items.Add(element);
    return element;
}
//清空元素。
public void Clear()
{
    foreach (object obj in items)
        adapter.Remove(obj);

    items.Clear();
}
//是否包含。
public bool Contains(object uiElement)
{
    return items.Contains(uiElement);
}
//删除Item。
public void Remove(object uiElement)
{
    if (items.Contains(uiElement))
    {
        adapter.Remove(uiElement);
```

```
                items.Remove(uiElement);
            }
        }
        //复制。
        internal UIExtensionSite Duplicate()
        {
            return new UIExtensionSite(this.adapter);
        }
        //获取迭代器。
        IEnumerator<object> IEnumerable<object>.GetEnumerator()
        {
            return items.GetEnumerator();
        }
        //迭代器。
        IEnumerator IEnumerable.GetEnumerator()
        {
            return items.GetEnumerator();
        }
    }
}
```

3.4 UIExtensionSiteCollection

可扩展站点的集合。这些站点不使用 Builder 构建。因此需要一个新的集合来存储。

```
using System;
using System.Collections;
using System.Collections.Generic;
using Microsoft.Practices.CompositeUI.UIElements;
using Microsoft.Practices.CompositeUI.Utility;
```

```
using System.Globalization;
namespace Microsoft.Practices.CompositeUI
{
    public class UIExtensionSiteCollection : ICollection, IEnumerable<string>, IDisposable
    {
        UIExtensionSiteCollection parent;
        WorkItem rootWorkItem;  //根WorkItem。
        //站点集合。
        Dictionary<string, UIExtensionSite> sites =
            new Dictionary<string, UIExtensionSite>();
        //Adapter集合。
        List<IUIElementAdapter> createdAdapters = new List<IUIElementAdapter>();
        public UIExtensionSiteCollection()
            : this(null, null)
        {
        }
        //根WorkItem的站点集合。
        public UIExtensionSiteCollection(WorkItem rootWorkItem)
            : this(rootWorkItem, null)
        {
        }
        //子WorkItem的站点集合。
        public UIExtensionSiteCollection(UIExtensionSiteCollection parent)
            : this(parent.rootWorkItem, parent)
        {
        }
        //设置根WorkItem和父站点。
        private UIExtensionSiteCollection(WorkItem rootWorkItem,
            UIExtensionSiteCollection parent)
        {
```

```
        this.parent = parent;
        this.rootWorkItem = rootWorkItem;
    }
    //获取站点。
    public UIExtensionSite this[string siteName]
    {
        get
        {
            if (sites.ContainsKey(siteName))
                return sites[siteName];
            if (parent != null)
            {
                //从父站点查找。如果没有找到，则抛出异常。
                UIExtensionSite childSite = parent[siteName].Duplicate();
                sites.Add(siteName, childSite);
                return childSite;
            }
            //抛出异常。
            throw new ArgumentException(string.Format(CultureInfo.CurrentCulture,
                Properties.Resources.ExtensionSiteDoesNotExist, siteName),
                "siteName");
        }
    }
    //站点的数量。
    public int Count
    {
        get
        {
            int result = 0;
            foreach (string siteName in this)
```

```
                result++;
            return result;
        }
    }
    //是否包含。
    public bool Contains(string siteName)
    {
        if (sites.ContainsKey(siteName))
            return true;
        if (parent != null)
            return parent.Contains(siteName);
        return false;
    }
    //使用UI元素注册站点。
    public void RegisterSite(string siteName, object uiElement)
    {
        Guard.ArgumentNotNullOrEmptyString(siteName, "siteName");
        Guard.ArgumentNotNull(uiElement, "uiElement");
        //获取AdapterFactory。
        IUIElementAdapterFactory factory = FactoryCatalog.GetFactory(uiElement);
        //获取Adapter。
        IUIElementAdapter adapter = factory.GetAdapter(uiElement);
        createdAdapters.Add(adapter);
        //注册站点。
        RegisterSite(siteName, adapter);
    }
    //注册站点。
    public void RegisterSite(string siteName, IUIElementAdapter adapter)
    {
        Guard.ArgumentNotNullOrEmptyString(siteName, "siteName");
```

```
        Guard.ArgumentNotNull(adapter, "adapter");
        //不能重复。
        if (Contains(siteName))
            throw new ArgumentException(string.Format(CultureInfo.CurrentCulture,
                Properties.Resources.DuplicateUIExtensionSite, siteName),
                "siteName");
        sites.Add(siteName, new UIExtensionSite(adapter));
    }
    //卸载。
    public void UnregisterSite(string siteName)
    {
        Guard.ArgumentNotNullOrEmptyString(siteName, "siteName");
        //不能卸载包含在父集合的站点。
        if (parent != null && parent.Contains(siteName))
            throw new ArgumentException(string.Format(CultureInfo.CurrentCulture,
                Properties.Resources.CannotUnregisterSiteRegisteredWithParent,
                siteName), "siteName");
        sites.Remove(siteName);
    }
    //AdapterFactory目录。
    private IUIElementAdapterFactoryCatalog FactoryCatalog
    {
        get { return rootWorkItem.Services.Get<IUIElementAdapterFactoryCatalog>(); }
    }
    //遍历所有站点。
    IEnumerator<string> IEnumerable<string>.GetEnumerator()
    {
        Set<string> seenNames = new Set<string>();
        for (UIExtensionSiteCollection current = this; current != null;
            current = current.parent)
```

```
    {
        foreach (string siteName in current.sites.Keys)
        {
            if (!seenNames.Contains(siteName))
            {
                seenNames.Add(siteName);
                yield return siteName;
            }
        }
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return ((IEnumerable<string>)this).GetEnumerator();
}

void ICollection.CopyTo(Array array, int index)
{
    throw new NotImplementedException();
}

int ICollection.Count
{
    get { return Count; }
}

bool ICollection.IsSynchronized
{
    get { return false; }
```

```
        }

        object ICollection.SyncRoot
        {
            get { throw new NotImplementedException(); }
        }
        //垃圾收集。
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        //清理资源。
        protected virtual void Dispose(bool disposing)
        {
            foreach (UIExtensionSite site in sites.Values)
                site.Clear();
            foreach (IUIElementAdapter adapter in createdAdapters)
            {
                IDisposable disp = adapter as IDisposable;
                if (disp != null)
                    disp.Dispose();
            }
            sites.Clear();
            createdAdapters.Clear();
        }
    }
}
```

4 命令（Commands）

Invoker 的某个事件(如 Clicked)与 Adapter 的 ExecuteCommand 句柄关联。Adapter 的 ExecuteCommand 与 Command 的 ExecuteAction 关联，Command 的 ExecuteAction 在 Builder 构建一个包含 CommandHanlderAttribute 声明的方法的对象时，会与其关联。因此，Invoker 的事件发生时，会自动调用对应 Command 的 Hanlder。

4.1 CommandException

```
using System;
using System.Runtime.Serialization;
namespace Microsoft.Practices.CompositeUI.Commands
{
    [Serializable]
    public class CommandException : Exception
    {
        public CommandException() : base() { }
        public CommandException(string message) : base(message) { }
        public CommandException(string message, Exception innerException) :
            base(message, innerException) { }
        protected CommandException(SerializationInfo info, StreamingContext context) :
            base(info, context) { }
    }
}
```

4.2 CommandHandlerAttribute

声明一个方法为命令句柄。

```
using System;
```

```
namespace Microsoft.Practices.CompositeUI.Commands
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
    public class CommandHandlerAttribute : Attribute
    {
        private string commandName;
        public CommandHandlerAttribute(string commandName)
        {
            this.commandName = commandName;
        }
        public string CommandName
        {
            get { return this.commandName; }
        }
    }
}
```

4.3 CommandStatus

命令状态。

```
using System;
namespace Microsoft.Practices.CompositeUI.Commands
{
    public enum CommandStatus
    {
        //当命令执行时会触发Command.ExecuteAction或直接调用
        //CommandAdapter.ExecuteCommand时。Adapter将Enabled界面元素。
        Enabled,
        //不触发任何动作，Adapter需要将UI元素值为Disabled。
        Disabled,
```

```
            //不触发任何动作，Adapter需要设置UI元素为隐藏状态。
            Unavailable,
        }
}
```

4.4 CommandAdapterMapServiceException

```
using System;
using System.Runtime.Serialization;

namespace Microsoft.Practices.CompositeUI.Commands
{
    [Serializable]
    public class AdapterMapServiceException : Exception
    {
        public AdapterMapServiceException() : base() { }
        public AdapterMapServiceException(string message) : base(message) { }
        public AdapterMapServiceException(string message, Exception innerException) :
            base(message, innerException) { }
        protected AdapterMapServiceException(SerializationInfo info,
            StreamingContext context) : base(info, context) { }
    }
}
```

4.5 CommandAdapter

提供一个对象调用 Command 的逻辑。

Invoker-N----1->CommandAdapter-M----1->Command-1----M->Target。

```
using System;
using System.Reflection;
using Microsoft.Practices.ObjectBuilder;
```

```
using Microsoft.Practices.CompositeUI.Utility;

namespace Microsoft.Practices.CompositeUI.Commands
{
    public abstract class CommandAdapter : IDisposable
    {
        //绑定的命令。
        private Command boundCommand;
        //执行命令。
        public event EventHandler ExecuteCommand;
        //绑定命令。
        public virtual void BindCommand(Command command)
        {
            Guard.ArgumentNotNull(command, "command");
            if (boundCommand != null)
            {
                throw new InvalidOperationException(
                    Properties.Resources.AdapterAlreadyBoundToACommand);
            }
            command.Changed += CommandChangedHandler;
            boundCommand = command;
        }
        //解除命令绑定。
        public virtual void UnbindCommand(Command command)
        {
            Guard.ArgumentNotNull(command, "command");
            if (boundCommand != command)
            {
                throw new InvalidOperationException(
                    Properties.Resources.AdapterNotBoundToGivenCommand);
```

```
        }
        command.Changed -= CommandChangedHandler;
        boundCommand = null;
    }
    //添加Invoker。
    public abstract void AddInvoker(object invoker, string eventName);
    //删除Invoker。
    public abstract void RemoveInvoker(object invoker, string eventName);
    //Invoker数量。
    public abstract int InvokerCount { get; }
    //垃圾收集。
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
    //垃圾收集
protected virtual void Dispose(bool disposing)
{
}
    //是否包括Invoker。
    public abstract bool ContainsInvoker(object invoker);
    //命令属性变化。
    protected virtual void OnCommandChanged(Command command)
    {
    }
    //执行命令。
    protected virtual void FireCommand()
    {
        if (ExecuteCommand != null)
```

```
            {
                ExecuteCommand(this, EventArgs.Empty);
            }
        }
        //命令句柄发生变化。
        private void CommandChangedHandler(object sender, EventArgs e)
        {
            OnCommandChanged((Command)sender);
        }
    }
}
```

4.6 EventCommandAdapter

事件命令 Invoker。即一个 Invoker 的某个事件将关联到 Command。Invoker 的事件处理会直接连接到 InvokeEventHandler，它会调用父类的 ExecuteCommand 方法。当 Invoker 的事件发生时，父类 ExecuteCommand 会被调用。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using Microsoft.Practices.CompositeUI.Utility;
using System.Globalization;

namespace Microsoft.Practices.CompositeUI.Commands
{
    public class EventCommandAdapter<TInvoker> : CommandAdapter
    {
        //Invoker与事件列表（List<string>）序列。
        private ListDictionary<TInvoker, string> invokers =
```

```
        new ListDictionary<TInvoker, string>();
    public EventCommandAdapter()
    {
    }
    public EventCommandAdapter(TInvoker invoker, string eventName)
    {
        WireUpInvoker(invoker, eventName);
    }
    public ReadOnlyDictionary<TInvoker, List<string>> Invokers
    {
        get { return new ReadOnlyDictionary<TInvoker, List<string>>(invokers); }
    }
    //调用事件处理句柄。直接触发命令。
    public void InvokerEventHandler(object sender, EventArgs e)
    {
        FireCommand();
    }
    //添加Invoker。
    public override void AddInvoker(object invoker, string eventName)
    {
        Guard.ArgumentNotNull(invoker, "invoker");
        ThrowIfWrongType(invoker);
        WireUpInvoker((TInvoker)invoker, eventName);
    }
    //删除调用。
    public override void RemoveInvoker(object invoker, string eventName)
    {
        UnwireInvoker(invoker, eventName);
        RemoveInvokerIfZeroEvents(invoker);
    }
```

```
//清理。
protected override void Dispose(bool disposing)
{
   if (disposing)
   {
       foreach (KeyValuePair<TInvoker, List<string>> pair in invokers)
       {
           List<string> events = pair.Value;
           for (int i = events.Count - 1; i >= 0; i--)
           {
               UnwireInvoker(pair.Key, events[i]);
           }
       }

       invokers.Clear();
   }
   base.Dispose(disposing);
}
//是否包含Invoker。
public override bool ContainsInvoker(object invoker)
{
    if (typeof(TInvoker).IsAssignableFrom(invoker.GetType()) == false)
    {
        return false;
    }
    else
    {
        return invokers.ContainsKey((TInvoker)invoker);
    }
}
```

```
//Invoker数量。
public override int InvokerCount
{
    get
    {
        return invokers.Count;
    }
}
//注册调用者。
private void WireUpInvoker(TInvoker invoker, string eventName)
{
    Guard.ArgumentNotNull(invoker, "invoker");
    Guard.ArgumentNotNull(eventName, "evenName");
    ThrowIfWrongType(invoker);

    if (invokers.ContainsKey(invoker) == false)
    {
        HookInvokerEvent(invoker, eventName);
        invokers.Add(invoker, eventName);
    }
    else if (invokers[invoker].Contains(eventName) == false)
    {
        invokers[invoker].Add(eventName);
        HookInvokerEvent(invoker, eventName);
    }
}
//注销一个调用者。
private void UnwireInvoker(object invoker, string eventName)
{
    Guard.ArgumentNotNull(invoker, "invoker");
```

```
        Guard.ArgumentNotNullOrEmptyString(eventName, "eventName");
        ThrowIfWrongType(invoker);

        TInvoker typedInvoker = (TInvoker)invoker;
        if (invokers.ContainsKey(typedInvoker) == true)
        {
            UnhookInvokerEvent(invoker, eventName);
            invokers[typedInvoker].Remove(eventName);
        }
    }
    //如果没有其它事件与Command关联，则直接删除。
    private void RemoveInvokerIfZeroEvents(object invoker)
    {
        Guard.ArgumentNotNull(invoker, "invoker");
        ThrowIfWrongType(invoker);

        TInvoker typedInvoker = (TInvoker)invoker;
        if (invokers.ContainsKey(typedInvoker) && invokers[typedInvoker].Count == 0)
        {
            invokers.Remove(typedInvoker);
        }
    }
    //注册一个代理。
    private void HookInvokerEvent(object invoker, string eventName)
    {
        EventInfo eventInfo = GetEventInfo(invoker, eventName);
        Delegate handlerDelegate = Delegate.CreateDelegate(
            eventInfo.EventHandlerType, this, InvokerEventHandle);
        eventInfo.AddEventHandler(invoker, handlerDelegate);
    }
```

```
//删除一个代理。
private void UnhookInvokerEvent(object invoker, string eventName)
{
    EventInfo eventInfo = GetEventInfo(invoker, eventName);
    Delegate handlerDelegate = Delegate.CreateDelegate(
        eventInfo.EventHandlerType, this, InvokerEventHandle);
    eventInfo.RemoveEventHandler(invoker, handlerDelegate);
}
//获取事件信息。
private EventInfo GetEventInfo(object invoker, string eventName)
{
    Type invokerType = invoker.GetType();
    EventInfo eventInfo = invokerType.GetEvent(
        eventName, BindingFlags.Instance|BindingFlags.Static|
        BindingFlags.Public);
    if (eventInfo == null)
    {
        throw new CommandException(String.Format(CultureInfo.CurrentCulture,
            Properties.Resources.EventNotPresentOnInvoker, eventName,
            invokerType.FullName));
    }
    if (eventInfo.GetAddMethod().IsStatic ||
        eventInfo.GetRemoveMethod().IsStatic)
    {
        throw new CommandException(
            string.Format(CultureInfo.CurrentCulture,
            Properties.Resources.StaticCommandPublisherNotSupported,
            eventInfo.Name));
    }
    return eventInfo;
```

```
        }
        private MethodInfo InvokerEventHandle
        {
            get
            {
                return this.GetType().GetMethod("InvokerEventHandler",
                    BindingFlags.Instance | BindingFlags.Public);
            }
        }
        private void ThrowIfWrongType(object invoker)
        {
            if (typeof(TInvoker).IsAssignableFrom(invoker.GetType()) == false)
            {
                throw new ArgumentException(
                    string.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.EventCommandAdapterInvokerNotType,
                    typeof(TInvoker)));
            }
        }
    }
}
```

4.7 ICommandAdapterMapService

存储 Invoker 与 CommandAdapter 的映射。由 Command 调用来创建 Invoker 对应的 Adapter。

```
using System;
namespace Microsoft.Practices.CompositeUI.Commands
{
    public interface ICommandAdapterMapService
```

```
    {
        void Register(Type invokerType, Type adapterType);
        CommandAdapter CreateAdapter(Type invokerType);
        void UnRegister(Type invokerType);
    }
}
```

4.8 CommandAdapterMapService

```
using System;
using System.Collections.Generic;
using Microsoft.Practices.CompositeUI.Properties;
using Microsoft.Practices.CompositeUI.Utility;

namespace Microsoft.Practices.CompositeUI.Commands
{
    public class CommandAdapterMapService : ICommandAdapterMapService
    {
        //映射序列。
        private Dictionary<Type, Type> map = new Dictionary<Type, Type>();
        //添加映射序列。
        public void Register(Type invokerType, Type adapterType)
        {
            Guard.ArgumentNotNull(invokerType, "invokerType");
            Guard.ArgumentNotNull(adapterType, "adapterType");
            //判断adaterType是否继承与CommandAdapter。
            if (!typeof (CommandAdapter).IsAssignableFrom(adapterType))
            {
                throw new AdapterMapServiceException(Resources.InvalidAdapter);
            }
```

```
        map[invokerType] = adapterType;
    }
    //创建Adapter。
    public CommandAdapter CreateAdapter(Type invokerType)
    {
        Guard.ArgumentNotNull(invokerType, "invokerType");
        CommandAdapter adapter = null;
        //如果找到该Invoker对应的Adapter，则直接创建。
        if (map.ContainsKey(invokerType))
        {
            adapter = (CommandAdapter)Activator.CreateInstance(map[invokerType]);
        }
        else  //否则，使用父类的Adapter创建。
        {
            adapter = FindAssignableAdapter(invokerType);
        }
        return adapter;
    }

    private CommandAdapter FindAssignableAdapter(Type type)
    {
        CommandAdapter adapter = null;
        foreach (KeyValuePair<Type, Type> pair in map)
        {
            if (pair.Key.IsAssignableFrom(type))
            {
                adapter = (CommandAdapter)Activator.CreateInstance(pair.Value);
                break;
            }
        }
```

```
            return adapter;
        }
        //卸载。
        public void UnRegister(Type invokerType)
        {
            map.Remove(invokerType);
        }
    }
}
```

4.9 Command

代表一个命令。一个指定名称的命令可以处理多个对象的方法。当 Builder 构建一个 Command 时，其 ExecuteAction 会自动挂载到声明了 CommandHandlerAttribute 的方法上。

I Invoker-N----1->CommandAdapter-M----1->Command-1----M->Target。

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using Microsoft.Practices.CompositeUI.Properties;
using Microsoft.Practices.CompositeUI.Utility;
using System.Globalization;
using System.Reflection;
using System.Diagnostics;
using Microsoft.Practices.ObjectBuilder;

namespace Microsoft.Practices.CompositeUI.Commands
{
    public class Command : IDisposable, IBuilderAware
    {
```

```
private TraceSource traceSource;
//命令状态。
private CommandStatus status = CommandStatus.Enabled;
//命令适配器。
private List<CommandAdapter> adapters = new List<CommandAdapter>();
//所属的命令适配器。在添加一个Invoker时，需要创建一个Adapter，因此
//需要记录下来，以便在删除该Invoker时，将Adpater删除。
private List<CommandAdapter> ownedAdapters = new List<CommandAdapter>();
private string name;
//保存了Invoker与Adapter的类型的映射关系。
private ICommandAdapterMapService mapService;
//Invoker与Adapter的类型的映射服务。
[ServiceDependency]
public ICommandAdapterMapService MapService
{
    set { mapService = value; }
}
[ClassNameTraceSource]
public TraceSource TraceSource
{
    set { traceSource = value; }
    protected get { return traceSource; }
}
//当命令属性变化是触发。
public event EventHandler Changed;
//执行命令。该事件处理在Builder的CommandStrategy中，会自动的挂载到声明了
//CommandHadler特性的方法上。
public event EventHandler ExecuteAction;
public Command() { }
//命令名称。
```

```
public string Name
{
    get { return name; }
}
//状态。
public CommandStatus Status
{
    get { return this.status; }
    set
    {
        CommandStatus oldValue = this.status;
        this.status = value;
        //触发事件。
        if (oldValue != value)
        {
            OnChanged();
            if (TraceSource != null)
                TraceSource.TraceInformation(
                Properties.Resources.TraceCommandStatusChanged, name, status);
        }
    }
}
//当前命令的适配器。
public virtual void AddCommandAdapter(CommandAdapter adapter)
{
    //将Adapter的ExecuteCommand句柄与ExecuteAction关联。
    adapter.ExecuteCommand += this.OnExecuteAction;
    adapter.BindCommand(this);
    adapters.Add(adapter);
}
```

```
//删除命令适配器。
public virtual void RemoveCommandAdapter(CommandAdapter adapter)
{
    adapter.ExecuteCommand -= this.OnExecuteAction;
    adapter.UnbindCommand(this);
    adapters.Remove(adapter);
}
//执行命令。直接触发OnExecuteAction事件。这个事件由所有Adapter订阅。
public virtual void Execute()
{
    if (status == CommandStatus.Enabled)
    {
        OnExecuteAction(this, EventArgs.Empty);
        if (TraceSource != null)
            TraceSource.TraceInformation(
            Properties.Resources.TraceCommandExecuted, name);
    }
}
//返回所有Adapters。
public virtual ReadOnlyCollection<CommandAdapter> Adapters
{
    get { return new ReadOnlyCollection<CommandAdapter>(adapters); }
}
//查找指定类型的Adapter。
public virtual ReadOnlyCollection<TAdapter> FindAdapters<TAdapter>()
    where TAdapter : CommandAdapter
{
    List<TAdapter> found = new List<TAdapter>();
    adapters.ForEach(delegate(CommandAdapter adapter)
        {
```

```
                if (adapter is TAdapter)
                {
                    found.Add((TAdapter)adapter);
                }
            });
        return new ReadOnlyCollection<TAdapter>(found);
    }
    //添加某个Invoker的某个事件对该Command的调用。
    public virtual void AddInvoker(object invoker, string eventName)
    {
        if (mapService == null)
        {
            throw new CommandException(String.Format(CultureInfo.CurrentCulture,
                Resources.CannotGetMapService, name));
        }
        //获取命令适配器。
        CommandAdapter adapter = mapService.CreateAdapter(invoker.GetType());
        if (adapter == null)
        {
            throw new CommandException(String.Format(CultureInfo.CurrentCulture,
                Resources.CannotGetCommandAdapter, invoker.GetType()));
        }
        //添加Invoker。
        adapter.AddInvoker(invoker, eventName);
        //添加Adapter。
        AddCommandAdapter(adapter);
        //添加OwnedAdapter。
        ownedAdapters.Add(adapter);
    }
    //删除调用。
```

```
public virtual void RemoveInvoker(object invoker, string eventName)
{
    foreach (CommandAdapter adapter in adapters.ToArray())
    {
        if (adapter.ContainsInvoker(invoker) == true)
        {
            adapter.RemoveInvoker(invoker, eventName);
            if (adapter.InvokerCount == 0)
            {
                RemoveCommandAdapter(adapter);
                if (ownedAdapters.Contains(adapter))
                {
                    ownedAdapters.Remove(adapter);
                    adapter.Dispose();
                }
            }
        }
    }
}
//处理OnExecuteAction。
protected virtual void OnExecuteAction(object sender, EventArgs e)
{
    if (Status == CommandStatus.Enabled && ExecuteAction != null)
    {
        ExecuteAction(this, e);
    }
}
//触发OnChanged事件。
protected virtual void OnChanged()
{
```

```
        if (Changed != null)
        {
            Changed(this, EventArgs.Empty);
        }
    }
    //Handler是否已经注册。
    internal bool IsHandlerRegistered(object target, MethodInfo methodInfo)
    {
        if (ExecuteAction != null)
        {
            foreach (Delegate dlg in ExecuteAction.GetInvocationList())
            {
                if (dlg.Target == target && dlg.Method == methodInfo)
                {
                    return true;
                }
            }
        }
        return false;
    }
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    //清理。
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
```

```
                RemoveCommandAdapters();
            }
        }
        //删除Adapter。
        private void RemoveCommandAdapters()
        {
            foreach (CommandAdapter adapter in adapters.ToArray())
            {
                RemoveCommandAdapter(adapter);
            }
            foreach (CommandAdapter adapter in ownedAdapters.ToArray())
            {
                adapter.Dispose();
            }
        }
        public virtual void OnBuiltUp(string id)
        {
            this.name = id;
        }
        public virtual void OnTearingDown()
        {
        }

        class ClassNameTraceSourceAttribute : TraceSourceAttribute
        {
            public ClassNameTraceSourceAttribute() : base(typeof(Command).FullName) { }
        }
    }
}
```

4.10 EventTopicCommand

定以一个触发 EventTopic 的 Command。它处理完命令后，还会发布一个事件主题。

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI.Utility;
using Microsoft.Practices.CompositeUI.EventBroker;
using System.Globalization;

namespace Microsoft.Practices.CompositeUI.Commands
{
    public class EventTopicCommand : Command
    {
        private PublicationScope scope = PublicationScope.Descendants;
        private WorkItem workItem;
        private const string TopicNameFormat = "topic://EventTopicCommand/{0}";
        //WorkItem。
        [ServiceDependency]
        public WorkItem WorkItem
        {
            set { workItem = value; }
        }
        //执行命令并且触发一个Topic。当指定名称的命令被执行时，会发布一个Topic。
        //用户可以通过订阅topic://EventTopicCommand/{CommandName}的主题来订阅
        //其处理。
        protected override void OnExecuteAction(object sender, EventArgs e)
        {
            base.OnExecuteAction(sender, e);
```

```
            EventTopic topic = workItem.EventTopics.Get(
                String.Format(CultureInfo.InstalledUICulture, TopicNameFormat, Name));
            if (topic != null)
            {
                topic.Fire(sender, e, workItem, scope);
            }
        }
        //发布范围。
        public PublicationScope Scope
        {
            get { return scope; }
            set { scope = value;  }
        }
    }
}
```

5 事件代理（EventBroker）

首先，在一个组件声明一个 EventPublication 来发布一个事件，指定订阅的范围。

```
[EventPublication("event://UpdatesAvailable",
                  PublicationScope. Global/Descendants/WorkItem)]
public event SomeEventHandler UpdatesAvailable;
```

其次，在另一个组件声明一个 EventSubscription 特性来订阅一个事件，指定线程模式。

```
[EventSubscription("event://UpdatesAvailable")]
public void NewUpdates(object sender, SomeEventArgs numUpdates)
{
   MessageBox.Show(numUpdates.ToString(), "Updates available");
}
```

一个 Publication 是由一个发布者和一个发布事件组成，发布事件必须继承 EventHandler。一个 Subscription 由一个订阅者和订阅处理函数组成，订阅处理函数由两个参数组成。事件主题是 Publication 和 Subscription 的通讯点。

5.1 EventBrokerException

```
using System;
using System.Runtime.Serialization;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    [Serializable]
    public class EventBrokerException : Exception
    {
        public EventBrokerException() : base() { }
        public EventBrokerException(string message) : base(message) { }
        public EventBrokerException(string message, Exception innerException) :
            base(message, innerException) { }
        protected EventBrokerException(SerializationInfo info, StreamingContext context) :
            base(info, context) { }
    }
}
```

5.2 EventTopicException

```
using System;
using System.Collections.ObjectModel;
using Microsoft.Practices.CompositeUI.Properties;
using System.Runtime.Serialization;
using System.Globalization;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
```

```
[Serializable]
public class EventTopicException : Exception
{
    private ReadOnlyCollection<Exception> exceptions;
    private EventTopic topic;
    public EventTopicException() : base()
    {
    }
    public EventTopicException(string message) : base(message)
    {
    }
    public EventTopicException(string message, Exception innerException)
        : base(message, innerException)
    {
    }
    protected EventTopicException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
    }
    //初始化实例，指定主题和异常列表。
    public EventTopicException(EventTopic topic,
        ReadOnlyCollection<Exception> exceptions)
        : base(String.Format(CultureInfo.CurrentCulture,
            Resources.EventTopicFireException, topic.Name))
    {
        this.topic = topic;
        this.exceptions = exceptions;
    }
    //订阅处理的异常列表。
    public ReadOnlyCollection<Exception> Exceptions
```

```
        {
            get { return exceptions; }
        }
        //引起错误的事件主题。
        public EventTopic Topic
        {
            get { return topic; }
        }
    }
}
```

5.3 EventPublicatonAttribute

发布事件特性。

```
using System;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    [AttributeUsage(AttributeTargets.Event, AllowMultiple=true)]
    public sealed class EventPublicationAttribute : Attribute
    {
        private string topic;
        private PublicationScope scope;
        public EventPublicationAttribute(string topic)
            : this(topic, PublicationScope.Global)
        {
        }
        public EventPublicationAttribute(string topic, PublicationScope scope)
        {
            this.topic = topic;
            this.scope = scope;
```

```
        }
        //主题名称。
        public string Topic
        {
            get { return topic; }
        }
        //发布范围。
        public PublicationScope Scope
        {
            get { return this.scope; }
        }
    }
}
```

5.4 EventSubscriptionAttribute

事件订阅声明特性。

```
using System;
using System.ComponentModel;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
    public class EventSubscriptionAttribute : Attribute
    {
        private string topic;
        private ThreadOption threadOption;
        public EventSubscriptionAttribute(string topic)
            : this(topic, ThreadOption.Publisher)
        {
        }
```

```
        public EventSubscriptionAttribute(string topic, ThreadOption threadOption)
        {
            this.topic = topic;
            this.threadOption = threadOption;
        }
        //事件主题名称。
        public string Topic
        {
            get { return topic; }
        }
        //订阅线程模式。
        [DefaultValue(ThreadOption.Publisher)]
        public ThreadOption Thread
        {
            set { threadOption = value; }
            get { return threadOption; }
        }
    }
}
```

5.5 PublicationScope

发布范围。

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    public enum PublicationScope
    {
```

```
            Global,
            WorkItem,
            Descendants,
        }
}
```

5.6 Publication

一个发布。一个发布由发布者、发布事件和发布主题组成。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using System.Globalization;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    internal class Publication : IDisposable
    {
        //发布主题。
        private EventTopic topic;
        //发布者。
        private WeakReference wrPublisher;
        //所在WorkItem。
        private WeakReference wrWorkItem;
        //发布范围。
        private PublicationScope scope;
        //发布事件名称。
        private string eventName;
        //新建实例。
        public Publication(EventTopic topic, object publisher, string eventName,
```

```
        WorkItem workItem, PublicationScope scope)
    {
        this.topic = topic;
        this.wrPublisher = new WeakReference(publisher);
        this.scope = scope;
        this.eventName = eventName;
        this.wrWorkItem = new WeakReference(workItem);
        //处理发布事件。
        EventInfo publishedEvent =
            publisher.GetType().GetEvent(eventName, BindingFlags.Public |
            BindingFlags.Instance | BindingFlags.Static);
        if (publishedEvent == null)
        {
            throw new EventBrokerException(String.Format(CultureInfo.CurrentCulture,
                Properties.Resources.CannotFindPublishedEvent, eventName));
        }
        //添加这个发布事件的处理句柄。
        ThrowIfInvalidEventHandler(publishedEvent);
        ThrowIfEventIsStatic(publishedEvent);
        Delegate handler = Delegate.CreateDelegate(publishedEvent.EventHandlerType,
            this, this.GetType().GetMethod("PublicationHandler"));
        publishedEvent.AddEventHandler(publisher, handler);
    }
    //如果发布的事件是静态，则触发异常。
    private void ThrowIfEventIsStatic(EventInfo publishedEvent)
    {
        if (publishedEvent.GetAddMethod().IsStatic ||
            publishedEvent.GetRemoveMethod().IsStatic)
        {
            throw new EventBrokerException(String.Format(CultureInfo.CurrentCulture,
```

```
                Properties.Resources.StaticPublisherNotAllowed, EventName));
        }
    }
    //如果事件声明的处理句柄不是继承EventHandler的句柄，则触发异常。
    private void ThrowIfInvalidEventHandler(EventInfo info)
    {
        if (typeof(EventHandler).IsAssignableFrom(info.EventHandlerType) ||
            (info.EventHandlerType.IsGenericType &&
                typeof(EventHandler<>).IsAssignableFrom(
                info.EventHandlerType.GetGenericTypeDefinition())))
        {
            return;
        }
        throw new EventBrokerException(String.Format(CultureInfo.CurrentCulture,
            Properties.Resources.InvalidPublicationSignature,
            info.DeclaringType.FullName, info.Name));
    }
    //发布处理句柄，直接调用EventTopic的Fire方法。
    public void PublicationHandler(object sender, EventArgs e)
    {
        topic.Fire(this, sender, e);
    }
    //发布对象。
    public object Publisher
    {
        get { return wrPublisher.Target; }
    }
    //WorkItem。
    public WorkItem WorkItem
    {
```

```
        get { return (WorkItem)wrWorkItem.Target; }
    }
    //事件名称。
    public string EventName
    {
        get { return eventName; }
    }
    //发布范围。
    public PublicationScope Scope
    {
        get { return scope; }
    }
    //清理。
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    //清理资源。
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            object publisher = Publisher;
            //清理发布对象时，必须移除发布对象的事件处理。
            if (publisher != null)
            {
                EventInfo publishedEvent = publisher.GetType().GetEvent(eventName);
                    publishedEvent.RemoveEventHandler(publisher,
                    Delegate.CreateDelegate(publishedEvent.EventHandlerType, this,
```

```
                        this.GetType().GetMethod("PublicationHandler")));
                }
            }
        }
    }
}
```

5.7 ThreadOption

```
using System.Threading;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    public enum ThreadOption
    {
        Publisher,
        Background,
        UserInterface,
    }
}
```

5.8 EventTopicFireDelegate

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    //代表订阅对象从事件主题获取调用的方法。
    //sender：发布者。
    //e：参数。
    //exceptions：异常列表。
```

```
    internal delegate void EventTopicFireDelegate(object sender, EventArgs e,
        List<Exception> exceptions, TraceSource traceSource);
}
```

5.9 WorkItemSubscriptons

管理一个 WorkItem 的订阅。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    internal class WorkItemSubscriptions : IEnumerable<Subscription>
    {
        //WorkItem引用。
        private WeakReference wrWorkItem;
        //订阅列表。
        private List<Subscription> subscriptions = new List<Subscription>();
        //锁对象。
        private object lockObject = new object();
        //初始化实例。
        public WorkItemSubscriptions(WorkItem workItem)
        {
            this.wrWorkItem = new WeakReference(workItem);
        }
        //添加一个订阅处理。
        public void AddSubscription(object subscriber, string handlerMethodName,
            ThreadOption threadOption)
        {
```

```
        AddSubscription(subscriber, handlerMethodName, null, threadOption);
    }
    //添加一个订阅处理。
    public void AddSubscription(object subscriber, string handlerMethodName,
        Type[] parameterTypes, ThreadOption threadOption)
    {
        lock (lockObject)
        {
            if (FindSubscription(subscriber, handlerMethodName) == null)
            {
                subscriptions.Add(new Subscription(
                this, subscriber, handlerMethodName, parameterTypes, threadOption));
            }
        }
    }
    //删除一个订阅处理。
    public void RemoveSubscription(object subscriber, string handlerMethodName)
    {
        lock (lockObject)
        {
            Subscription subscription = FindSubscription(
                subscriber, handlerMethodName);
            if (subscription != null)
            {
                RemoveSubscription(subscription);
            }
        }
    }
    //删除订阅处理。
    public void RemoveSubscription(Subscription subscription)
```

```
{
    lock (lockObject)
    {
        subscriptions.Remove(subscription);
    }
}
//订阅处理所在的WorkItem。
public WorkItem WorkItem
{
    get { return (WorkItem)wrWorkItem.Target; }
}
//获取事件主题的处理句柄。
public EventTopicFireDelegate[] GetHandlers()
{
    //句柄列表。
    List<EventTopicFireDelegate> result = new List<EventTopicFireDelegate>();
    lock (lockObject)
    {
        foreach (Subscription subscription in subscriptions.ToArray())
        {
            //利用Subscription获取处理句柄。
            EventTopicFireDelegate handler = subscription.GetHandler();
            if (handler != null)
            {
                result.Add(handler);
            }
        }
        return result.ToArray();
    }
}
```

```
//订阅数量。
public int SubscriptionCount
{
    get { return subscriptions.Count; }
}
//查找订阅处理。
public Subscription FindSubscription(object subscriber, string handlerMethodName)
{
    lock (lockObject)
    {
        Clean();
        return subscriptions.Find(delegate(Subscription match)
        {
            return match.Subscriber == subscriber &&
                match.HandlerMethodName == handlerMethodName;
        });
    }
}
//清理集合。
public void Clean()
{
    lock (lockObject)
    {
        //如果WorkItem为空或者已经终止，则删除所有的订阅对象。
        if (WorkItem == null || (
            WorkItem != null && WorkItem.Status == WorkItemStatus.Terminated))
        {
            subscriptions.Clear();
        }
        else
```

```
                {
                    //如果订阅者为空，则删除该订阅者的订阅。
                    foreach (Subscription subscription in subscriptions.ToArray())
                    {
                        if (subscription.Subscriber == null)
                        {
                            RemoveSubscription(subscription);
                        }
                    }
                }
            }
        }
        public IEnumerator<Subscription> GetEnumerator()
        {
            return subscriptions.GetEnumerator();
        }
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }
}
```

5.10 Subscription

订阅对象。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
```

```
using System.ComponentModel;
using System.Threading;
using Microsoft.Practices.CompositeUI.Utility;
using System.Globalization;
using System.Diagnostics;

namespace Microsoft.Practices.CompositeUI.EventBroker
{
    internal class Subscription
    {
        //订阅对象。
        private WeakReference wrSubscriber;
        //处理方法函数。
        private string handlerMethodName;
        //处理方法句柄。
        private RuntimeMethodHandle methodHandle;
        private ThreadOption threadOption;
        //同步上下文。
        private SynchronizationContext syncContext;
        //在本类没有使用。
        private WorkItemSubscriptions workItemSubscriptions;
        //处理方法事件参数类型。
        private Type handlerEventArgsType = null;
        //订阅对象类型句柄。
        private RuntimeTypeHandle typeHandle;
        //新建实例。
        internal Subscription(WorkItemSubscriptions workItemSubscriptions,
            object subscriber, string handlerMethodName, ThreadOption threadOption)
            : this(workItemSubscriptions, subscriber, handlerMethodName, null, threadOption)
        {
```

```
}
//新建实例。
internal Subscription(
    WorkItemSubscriptions workItemSubscriptions,
    object subscriber,
    string handlerMethodName,
    Type[] parameterTypes,
    ThreadOption threadOption)
{
    this.wrSubscriber = new WeakReference(subscriber);
    this.handlerMethodName = handlerMethodName;
    this.threadOption = threadOption;
    this.workItemSubscriptions = workItemSubscriptions;
    //获取处理方法的信息。
    MethodInfo miHandler = GetMethodInfo(subscriber, handlerMethodName,
        parameterTypes);
    if (miHandler == null)  //方法不存在。
    {
        throw new EventBrokerException(String.Format(CultureInfo.CurrentCulture,
            Properties.Resources.SubscriberHandlerNotFound,
            handlerMethodName, subscriber.GetType().ToString()));
    }
    else if (miHandler.IsStatic)  //方法是静态。
    {
        throw new EventBrokerException(String.Format(CultureInfo.CurrentCulture,
            Properties.Resources.CannotRegisterStaticSubscriptionMethods,
            miHandler.DeclaringType.FullName, miHandler.Name));
    }
    //订阅者类型句柄。
    this.typeHandle = subscriber.GetType().TypeHandle;
```

```
            //订阅处理方法句柄。
            this.methodHandle = miHandler.MethodHandle;
            ParameterInfo[] parameters = miHandler.GetParameters();
            if (IsValidEventHandler(parameters))  //参数必须2个，且第二个需要继承
EventArgs。
            {
                ParameterInfo pInfo = miHandler.GetParameters()[1];
                Type pType = pInfo.ParameterType;
                handlerEventArgsType = typeof(EventHandler<>).MakeGenericType(pType);
            }
            else
            {
                throw new EventBrokerException(String.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.InvalidSubscriptionSignature,
                    miHandler.DeclaringType.FullName, miHandler.Name));
            }
            if (threadOption == ThreadOption.UserInterface)
            {
                //如果处理线程是UI线程，则需要获取当前上下文，将处理句柄传递给它。
                if (SynchronizationContext.Current != null)
                {
                    syncContext = SynchronizationContext.Current;
                }
            }
        }
        //处理句柄的参数是否有效。
        private static bool IsValidEventHandler(ParameterInfo[] parameters)
        {
            return parameters.Length == 2 &&
                typeof(EventArgs).IsAssignableFrom(parameters[1].ParameterType);
```

```
}
//获取处理方法信息。
private MethodInfo GetMethodInfo(object subscriber, string handlerMethodName,
    Type[] parameterTypes)
{
    if (parameterTypes != null)
    {
        return subscriber.GetType().GetMethod(handlerMethodName, parameterTypes);
    }
    else
    {
        return subscriber.GetType().GetMethod(handlerMethodName);
    }
}
//订阅对象。
public object Subscriber
{
    get { return wrSubscriber.Target; }
}
//处理方法名称。
public string HandlerMethodName
{
    get { return handlerMethodName; }
}
//线程处理模式。
public ThreadOption ThreadOption
{
    get { return threadOption; }
}
//事件处理句柄。
```

```
internal EventTopicFireDelegate GetHandler()
{
    return EventTopicFireHandler;
}
//事件处理。
private void EventTopicFireHandler(object sender, EventArgs e,
    List<Exception> exceptions, TraceSource traceSource)
{
    object subscriber = wrSubscriber.Target;
    if (subscriber == null)
    {
        return;
    }
    switch (threadOption)
    {
        case ThreadOption.Publisher:
            CallOnPublisher(sender, e, exceptions);
            break;
        case ThreadOption.Background:
            CallOnBackgroundWorker(sender, e, traceSource);
            break;
        case ThreadOption.UserInterface:
            CallOnUserInterface(sender, e, exceptions);
            break;
        default:
            break;
    }
}
//在发布者的线程上处理。直接调用订阅代理。
private void CallOnPublisher(object sender, EventArgs e, List<Exception> exceptions)
```

```
{
    try
    {
        Delegate handler = CreateSubscriptionDelegate();
        if (handler != null)
        {
            handler.DynamicInvoke(sender, e);
        }
    }
    catch (TargetInvocationException ex)
    {
        exceptions.Add(ex.InnerException);
    }
}
//在后台线程处理。
private void CallOnBackgroundWorker(object sender, EventArgs e,
    TraceSource traceSource)
{
    Delegate handler = CreateSubscriptionDelegate();
    if (handler != null)
    {
        //将句柄的调用作为回调处理传递给线程池。
        ThreadPool.QueueUserWorkItem(new WaitCallback(delegate(object state)
        {
            CallInBackgroundArguments args = (CallInBackgroundArguments)state;
            try
            {
                args.Handler.DynamicInvoke(args.Sender, args.EventArgs);
            }
            catch (Exception ex)
```

```
                {
                    args.TraceSource.TraceInformation(
                        Properties.Resources.BackgroundSubscriberException,
                        ex.ToString());
                    throw;
                }
            }), new CallInBackgroundArguments(sender, e, handler, traceSource));
        }
    }
    //在UI线程处理。
    private void CallOnUserInterface(object sender, EventArgs e,
        List<Exception> exceptions)
    {
        Delegate handler = CreateSubscriptionDelegate();
        if (handler != null)
        {
            if (syncContext != null)
            {
                //发送处理代理，以handler为参数。
                syncContext.Send(delegate(object data)
                {
                    try
                    {
                        ((Delegate)data).DynamicInvoke(sender, e);
                    }
                    catch (TargetInvocationException ex)
                    {
                        exceptions.Add(ex.InnerException);
                    }
                }, handler);
```

```
            }
            else //尝试直接调用。
            {
                try
                {
                    handler.DynamicInvoke(sender, e);
                }
                catch (TargetInvocationException ex)
                {
                    exceptions.Add(ex.InnerException);
                }
            }
        }
    }
    //创建订阅代理。
    private Delegate CreateSubscriptionDelegate()
    {
        object subscriber = wrSubscriber.Target;
        if (subscriber != null)
        {
            return Delegate.CreateDelegate(this.handlerEventArgsType, subscriber,
                (MethodInfo)MethodInfo.GetMethodFromHandle(
                methodHandle, typeHandle));
        }
        return null;
    }
    //回调参数。
    private struct CallInBackgroundArguments
    {
        public Delegate Handler;
```

```
            public object Sender;
            public EventArgs EventArgs;
            public TraceSource TraceSource;
            public CallInBackgroundArguments(object sender, EventArgs eventArgs,
                Delegate handler, TraceSource traceSource)
            {
                Sender = sender;
                EventArgs = eventArgs;
                Handler = handler;
                TraceSource = traceSource;
            }
        }
    }
}
```

5.11 EventTopic

代表某一个在发布者和订阅者之间的主题的通讯点。

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI.Utility;
using System.ComponentModel;
using System.Collections.ObjectModel;
using System.Diagnostics;
using Microsoft.Practices.ObjectBuilder;
namespace Microsoft.Practices.CompositeUI.EventBroker
{
    public class EventTopic : IDisposable, IBuilderAware
    {
```

```
//主题名称。
private string name;
private bool enabled = true;
//发布者。
private List<Publication> publications = new List<Publication>();
//订阅者，按WorkItem组织。
private List<WorkItemSubscriptions>
    workItemSubscriptions = new List<WorkItemSubscriptions>();
private TraceSource traceSource = null;
[ClassNameTraceSource]
public TraceSource TraceSource
{
    set { traceSource = value; }
    protected get { return traceSource; }
}
//主题名称。
public string Name
{
    get { return name; }
}
//主题的Enabled属性。
public bool Enabled
{
    get { return enabled; }
    set { enabled = value; }
}
//添加发布。参数有：发布对象、发布事件、所在WorkItem和发布范围。
public virtual void AddPublication(object publisher, string eventName,
    WorkItem workItem, PublicationScope scope)
{
```

```
        Guard.ArgumentNotNull(publisher, "publisher");
        Guard.ArgumentNotNullOrEmptyString(eventName, "eventName");
        Guard.ArgumentNotNull(workItem, "workItem");
        Guard.EnumValueIsDefined(typeof(PublicationScope), scope, "scope");
        Clean();
        //不能发布重复。
        ThrowIfRepeatedPublication(publisher, eventName);
        //新建一个发布。
        Publication publication = new Publication(this, publisher, eventName,
            workItem, scope);
        publications.Add(publication);

        if (traceSource != null)
            traceSource.TraceInformation(
                Properties.Resources.EventTopicTracePublicationAdded,
                name, eventName, publisher.GetType().ToString());
    }
    //是否包含发布。
    public virtual bool ContainsPublication(object publisher, string eventName)
    {
        Guard.ArgumentNotNull(publisher, "publisher");
        Guard.ArgumentNotNull(eventName, "eventName");
        Clean();
        return FindPublication(publisher, eventName) != null;
    }
    //删除发布。
    public virtual void RemovePublication(object publisher, string eventName)
    {
        Guard.ArgumentNotNull(publisher, "publisher");
        Guard.ArgumentNotNull(eventName, "eventName");
```

```
        Clean();
        Publication publication = FindPublication(publisher, eventName);
        if (publication != null)
        {
            publications.Remove(publication);
            publication.Dispose();
            if (traceSource != null)
                traceSource.TraceInformation(
                    Properties.Resources.EventTopicTracePublicationRemoved,
                    name, eventName, publisher.GetType().ToString());
        }
    }
    //发布的数量。
    public virtual int PublicationCount
    {
        get
        {
            Clean();
            return publications.Count;
        }
    }
    //触发事件主题。
    public virtual void Fire(object sender, EventArgs e,
        WorkItem workItem, PublicationScope scope)
    {
        Guard.EnumValueIsDefined(typeof(PublicationScope), scope, "scope");
        if (enabled)
        {
            if (traceSource != null)
                traceSource.TraceInformation(
```

```
                Properties.Resources.EventTopicTraceFireStarted, name);
            Clean();

            switch (scope)
            {
                case PublicationScope.WorkItem:
                    CallSubscriptionHandlers(
                        sender, e, GetWorkItemHandlers(workItem));
                    break;
                case PublicationScope.Global:
                    CallSubscriptionHandlers(sender, e, GetAllHandlers());
                    break;
                case PublicationScope.Descendants:
                    CallSubscriptionHandlers(
                        sender, e, GetDescendantsHandlers(workItem));
                    break;
                default:
                    throw new ArgumentException(
                        Properties.Resources.InvalidPublicationScope);
            }
            if (traceSource != null)
                traceSource.TraceInformation(
                    Properties.Resources.EventTopicTraceFireCompleted, name);
        }
    }
    //添加一个订阅。
    public virtual void AddSubscription(object subscriber, string handlerMethodName,
          WorkItem workItem, ThreadOption threadOption)
    {
        AddSubscription(subscriber, handlerMethodName, null, workItem, threadOption);
```

```
}
//添加一个订阅。
public virtual void AddSubscription(object subscriber, string handlerMethodName,
    Type[] parameterTypes, WorkItem workItem, ThreadOption threadOption)
{
    Guard.ArgumentNotNull(subscriber, "subscriber");
    Guard.ArgumentNotNullOrEmptyString(handlerMethodName, "handlerMethodName");
    Guard.EnumValueIsDefined(typeof(ThreadOption), threadOption, "threadOption");
    Clean();
    WorkItemSubscriptions wis = FindWorkItemSubscription(workItem);
    if (wis == null)
    {
        wis = new WorkItemSubscriptions(workItem);
        workItemSubscriptions.Add(wis);
    }
    wis.AddSubscription(subscriber, handlerMethodName,
        parameterTypes, threadOption);
    if (traceSource != null)
        traceSource.TraceInformation(
            Properties.Resources.EventTopicTraceSubscriptionAdded,
            name, handlerMethodName, subscriber.GetType().ToString());
}
 //删除一个订阅。
public virtual void RemoveSubscription(object subscriber, string handlerMethodName)
{
    Guard.ArgumentNotNull(subscriber, "subscriber");
    Guard.ArgumentNotNull(handlerMethodName, "handlerMethodName");
    Clean();
    foreach (WorkItemSubscriptions wis in workItemSubscriptions)
    {
```

```
            wis.RemoveSubscription(subscriber, handlerMethodName);
        }
        if (traceSource != null)
            traceSource.TraceInformation(
                Properties.Resources.EventTopicTraceSubscriptionRemoved,
                name, handlerMethodName, subscriber.GetType().ToString());
    }
    //是否包含订阅。
    public virtual bool ContainsSubscription(object subscriber, string handlerMethodName)
    {
        Guard.ArgumentNotNull(subscriber, "subscriber");
        Guard.ArgumentNotNull(handlerMethodName, "handlerMethodName");
        Clean();
        return FindSubscription(subscriber, handlerMethodName) != null;
    }
    //订阅数量。
    public virtual int SubscriptionCount
    {
        get
        {
            Clean();
            int count = 0;
            foreach (WorkItemSubscriptions wis in workItemSubscriptions)
                count += wis.SubscriptionCount;
            return count;
        }
    }
    //清理。
    private void Clean()
    {
```

```
        //清理订阅。
        foreach (WorkItemSubscriptions wis in workItemSubscriptions.ToArray())
        {
            wis.Clean();
            if (wis.SubscriptionCount == 0)
            {
                workItemSubscriptions.Remove(wis);
            }
        }
        //清理发布。
        foreach (Publication publication in publications.ToArray())
        {
            if (publication.Publisher == null ||
                publication.WorkItem == null ||
                (publication.WorkItem != null &&
                publication.WorkItem.Status == WorkItemStatus.Terminated))
            {
                publications.Remove(publication);
                publication.Dispose();
            }
        }
    }
    //清理。
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    //清理。
    protected virtual void Dispose(bool disposing)
```

```
{
    if (disposing)
    {
        foreach (Publication publication in publications)
        {
            publication.Dispose();
        }
        publications.Clear();
        workItemSubscriptions.Clear();
    }
}
//调用订阅句柄。
private void CallSubscriptionHandlers(object sender, EventArgs e,
    EventTopicFireDelegate[] handlers)
{
    List<Exception> exceptions = new List<Exception>();
    foreach (EventTopicFireDelegate handler in handlers)
    {
        handler(sender, e, exceptions, traceSource);
    }
    if (exceptions.Count > 0)
    {
        TraceExceptions(exceptions);
        throw new EventTopicException(
            this, new ReadOnlyCollection<Exception>(exceptions));
    }
}
//显示异常信息。
private void TraceExceptions(List<Exception> exceptions)
{
```

```
        if (traceSource != null)
        {
            traceSource.TraceInformation(
                Properties.Resources.EventTopicTraceFireExceptions, name);
            foreach (Exception ex in exceptions)
                traceSource.TraceInformation(ex.ToString());
        }
    }
    //获取本WorkItem和子孙WorkItem的订阅句柄。
    private EventTopicFireDelegate[] GetDescendantsHandlers(WorkItem workItem)
    {
        List<WorkItem> descendants = new List<WorkItem>();
        List<EventTopicFireDelegate> handlers = new List<EventTopicFireDelegate>();
        descendants.Add(workItem);
        for (int i = 0; i < descendants.Count; i++)
        {
            //获取当前节点的处理句柄。
            handlers.AddRange(GetWorkItemHandlers(descendants[i]));
            //获取子孙节点。
            foreach (KeyValuePair<string, WorkItem> pair in descendants[i].WorkItems)
            {
                descendants.Add(pair.Value);
            }
        }
        return handlers.ToArray();
    }
    //获取当前节点的句柄。
    private EventTopicFireDelegate[] GetWorkItemHandlers(WorkItem workItem)
    {
        WorkItemSubscriptions wis = FindWorkItemSubscription(workItem);
```

```
            if (wis != null)
            {
                return wis.GetHandlers();
            }
            //返回空数组。
            return new EventTopicFireDelegate[0];
        }
        //获取所有句柄。
        private EventTopicFireDelegate[] GetAllHandlers()
        {
            List<EventTopicFireDelegate> handlers = new List<EventTopicFireDelegate>();
            foreach (WorkItemSubscriptions wis in workItemSubscriptions)
            {
                handlers.AddRange(wis.GetHandlers());
            }
            return handlers.ToArray();
        }
        //触发一个发布。
        internal void Fire(Publication publication, object sender, EventArgs e)
        {
            WorkItem workItem = publication.WorkItem;
            if (workItem == null || (
                workItem != null && workItem.Status == WorkItemStatus.Terminated))
            {
                publications.Remove(publication);
                publication.Dispose();
            }
            else
            {
                Fire(sender, e, publication.WorkItem, publication.Scope);
```

```
        }
    }
    //如果重复则触发。
    private void ThrowIfRepeatedPublication(object publisher, string eventName)
    {
        if (FindPublication(publisher, eventName) != null)
        {
            throw new EventBrokerException(
                Properties.Resources.OnlyOnePublicationIsAllowed);
        }
    }
    //查找发布。
    private Publication FindPublication(object publisher, string eventName)
    {
        Publication publication = publications.Find(delegate(Publication match)
        {
            return match.Publisher == publisher && match.EventName == eventName;
        });
        return publication;
    }
    //查找一个WorkItem的订阅。
    private WorkItemSubscriptions FindWorkItemSubscription(WorkItem workItem)
    {
        return workItemSubscriptions.Find(delegate(WorkItemSubscriptions match)
        {
            return match.WorkItem == workItem;
        });
    }
    //查找订阅。
    private Subscription FindSubscription(object subscriber, string handlerMethodName)
```

```
        {
            foreach (WorkItemSubscriptions wis in workItemSubscriptions)
            {
                Subscription subscription = wis.FindSubscription(
                    subscriber, handlerMethodName);
                if (subscription != null)
                {
                    return subscription;
                }
            }
            return null;
        }
        class ClassNameTraceSourceAttribute : TraceSourceAttribute
        {
            public ClassNameTraceSourceAttribute() : base(typeof(EventTopic).FullName) { }
        }
        //设置主题名称为ID名称。
        public virtual void OnBuiltUp(string id)
        {
            this.name = id;
        }
        public virtual void OnTearingDown()
        {
        }
    }
}
```

5.12 EventInspector

用于检查事件发布和订阅，注册和卸载发布者和订阅者。

```
using System;
using System.Reflection;
using Microsoft.Practices.CompositeUI.Utility;

namespace Microsoft.Practices.CompositeUI.EventBroker
{
    public static class EventInspector
    {
        //注册当前对象。
        public static void Register(object item, WorkItem workItem)
        {
            Guard.ArgumentNotNull(item, "item");
            Guard.ArgumentNotNull(workItem, "workItem");
            //注册发布。
            ProcessPublishers(item, item.GetType(), workItem, true);
            //注册订阅。
            ProcessSubscribers(item, item.GetType(), workItem, true);
        }
        //卸载。
        public static void Unregister(object item, WorkItem workItem)
        {
            Guard.ArgumentNotNull(item, "item");
            Guard.ArgumentNotNull(workItem, "workItem");

            ProcessPublishers(item, item.GetType(), workItem, false);
            ProcessSubscribers(item, item.GetType(), workItem, false);
        }
        //处理发布。
        private static void ProcessPublishers(object item, Type itemType,
            WorkItem workItem, bool register)
```

```
{
    foreach (EventInfo info in itemType.GetEvents())
    {
        foreach (EventPublicationAttribute attr in info.GetCustomAttributes(
            typeof(EventPublicationAttribute), true))
        {
            HandlePublisher(item, register, info, attr, workItem);
        }
    }
}

private static void HandlePublisher(object item, bool register, EventInfo info,
    EventPublicationAttribute attr, WorkItem workItem)
{
    //如果不存在当前EventTopic，则创建一个。
    EventTopic topic = workItem.EventTopics[attr.Topic];
    if (register)
    {
        //如果是注册，则添加发布者。
        topic.AddPublication(item, info.Name, workItem, attr.Scope);
    }
    else
    {
        //如果是卸载，则删除发布者。
        topic.RemovePublication(item, info.Name);
    }
}
//处理订阅。
private static void ProcessSubscribers(object item, Type itemType,
    WorkItem workItem, bool register)
```

```
{
    foreach (MethodInfo info in itemType.GetMethods())
    {
        foreach (EventSubscriptionAttribute attr in info.GetCustomAttributes(
            typeof(EventSubscriptionAttribute), true))
        {
            HandleSubscriber(item, register, info, attr, workItem);
        }
    }
}
//处理订阅。
private static void HandleSubscriber(object item, bool register, MethodInfo info,
    EventSubscriptionAttribute attr, WorkItem workItem)
{
    EventTopic topic = workItem.EventTopics[attr.Topic];
    if (register == true)
    {
        Type[] paramTypes = GetParamTypes(info);
        topic.AddSubscription(item, info.Name, paramTypes, workItem, attr.Thread);
    }
    else
    {
        topic.RemoveSubscription(item, info.Name);
    }
}
//获取参数类型。
private static Type[] GetParamTypes(MethodInfo info)
{
    ParameterInfo[] paramInfos = info.GetParameters();
    Type[] paramTypes = new Type[paramInfos.Length];
```

```
                for (int i = 0; i < paramTypes.Length; i++)
                {
                    paramTypes[i] = paramInfos[i].ParameterType;
                }
                return paramTypes;
            }
        }
    }
```

6 界面元素（UIElements）

6.1 IUIElementAdapter

Shell 中命名扩展点 UIElement 的操作支持的接口。

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Microsoft.Practices.CompositeUI.UIElements
{
    public interface IUIElementAdapter
    {
        //添加UIElement。
        object Add(object uiElement);
        //删除UIElement。
        void Remove(object uiElement);
    }
}
```

6.2 IUIElementAdapterFactory

适配器工厂。

```
using System;

namespace Microsoft.Practices.CompositeUI.UIElements
{
    public interface IUIElementAdapterFactory
    {
        //根据一个UIElement获取Adapter。
        IUIElementAdapter GetAdapter(object uiElement);
        //判断是否支持该UIElement。
        bool Supports(object uiElement);
    }
}
```

6.3 IUIElementAdapterFactoryCatalog

适配器工厂目录。

```
using System;
using Microsoft.Practices.CompositeUI.Utility;
using System.Collections.Generic;
namespace Microsoft.Practices.CompositeUI.UIElements
{
    public interface IUIElementAdapterFactoryCatalog
    {
        //工厂列表。
        IList<IUIElementAdapterFactory> Factories { get; }
        //根据UIElement对象获取一个工厂。
        IUIElementAdapterFactory GetFactory(object element);
        //注册一个工厂。
```

```
        void RegisterFactory(IUIElementAdapterFactory factory);
    }
}
```

6.4 UIElementAdapter

某类 UIElement 适配器抽象类。

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI.UIElements;
using Microsoft.Practices.CompositeUI.Utility;
using System.Globalization;

namespace Microsoft.Practices.CompositeUI.UIElements
{
    public abstract class UIElementAdapter<TUIElement> : IUIElementAdapter
    {
        protected abstract TUIElement Add(TUIElement uiElement);
        protected abstract void Remove(TUIElement uiElement);
        public object Add(object uiElement)
        {
            Guard.ArgumentNotNull(uiElement, "uiElement");
            TUIElement element = GetTypedElement(uiElement);
            return Add(element); ;
        }
        public void Remove(object uiElement)
        {
            Guard.ArgumentNotNull(uiElement, "uiElement");
            TUIElement element = GetTypedElement(uiElement);
```

```
            Remove(element);
        }
        private TUIElement GetTypedElement(object uiElement)
        {
            if (!(uiElement is TUIElement))
            {
                throw new ArgumentException(
                    string.Format(
                        CultureInfo.CurrentCulture,
                        Properties.Resources.UIElementNotCorrectType,
                        typeof(TUIElement).ToString()));
            }
            return (TUIElement)uiElement;
        }
    }
}
```

6.5 UIElementAdapterFactoryCatalog

默认目录。

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI.Utility;
using System.Globalization;

namespace Microsoft.Practices.CompositeUI.UIElements
{
    public class UIElementAdapterFactoryCatalog : IUIElementAdapterFactoryCatalog
    {
```

```
        //列表。
        private List<IUIElementAdapterFactory> factories =
            new List<IUIElementAdapterFactory>();
        //列表。
        public IList<IUIElementAdapterFactory> Factories
        {
            get { return factories.AsReadOnly(); }
        }
        //注册工厂。
        public void RegisterFactory(IUIElementAdapterFactory adapterFactory)
        {
            Guard.ArgumentNotNull(adapterFactory, "adapterFactory");
            factories.Add(adapterFactory);
        }
        //获取工厂。
        public IUIElementAdapterFactory GetFactory(object element)
        {
            foreach (IUIElementAdapterFactory factory in factories)
            {
                if (factory.Supports(element))
                    return factory;
            }
            throw new ArgumentException(String.Format(CultureInfo.CurrentCulture,
                                        Properties.Resources.NoRegisteredUIElementFactory,
                                        element.GetType().ToString()));
        }
    }
}
```

7 智能控件（SmartParts）

7.1 IWorkspace

Workspace 接口，提供了显示 SmartPart 的界面。

```
using System;
using System.Collections.ObjectModel;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public interface IWorkspace
    {
        //正在关闭SmartPart触发。
        event EventHandler<WorkspaceCancelEventArgs> SmartPartClosing;
        //SmartPart激活时触发。
        event EventHandler<WorkspaceEventArgs> SmartPartActivated;
        //SmartParts集合。
        ReadOnlyCollection<object> SmartParts { get; }
        //当前激活SmartPart。
        object ActiveSmartPart { get; }
        //激活一个SmartPart。
        void Activate(object smartPart);
        //应用SmartPartInfo。
        void ApplySmartPartInfo(object smartPart, ISmartPartInfo smartPartInfo);
        //关闭SmartPart。
        void Close(object smartPart);
        //隐藏SmartPart。
        void Hide(object smartPart);
        //显示SmartPart。
        void Show(object smartPart, ISmartPartInfo smartPartInfo);
```

```
        void Show(object smartPart);
    }
}
```

7.2 ISmartPartInfo

提供有关 SmartPart 的信息。

```
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public interface ISmartPartInfo
    {
        //描述信息。
        string Description { get; set; }
        //标题。
        string Title { get; set; }
    }
}
```

7.3 ISmartPartInfoProvider

信息提供商。

```
using System;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public interface ISmartPartInfoProvider
    {
        //根据SmartPart类型获取Info对象。
        ISmartPartInfo GetSmartPartInfo(Type smartPartInfoType);
    }
}
```

7.4 ISmartPartPlaceHolder

运行时存储 SmartPart 的控件接口。

```
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public interface ISmartPartPlaceholder
    {
        //SmartPart名称。
        string SmartPartName { get; set; }
        //显示的SmartPart。
        object SmartPart { get; set; }
    }
}
```

7.5 IComposableWorkspace

为不能继承于 WorkspaceComposer 但要使用其功能的类提供的接口。

```
using System;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public interface IComposableWorkspace<TSmartPart, TSmartPartInfo> : IWorkspace
        where TSmartPartInfo : ISmartPartInfo, new()
    {
        //当重写父类时，用于激活Workspace的SmartPart。
        void OnActivate(TSmartPart smartPart);
        //应用SmartPartInfo。
        void OnApplySmartPartInfo(TSmartPart smartPart, TSmartPartInfo smartPartInfo);
        //显示SmartPart。
        void OnShow(TSmartPart smartPart, TSmartPartInfo smartPartInfo);
        //隐藏SmartPart。
```

```
        void OnHide(TSmartPart smartPart);
        //管理SmartPart。
        void OnClose(TSmartPart smartPart);
        //触发SmartPartActivated事件。
        void RaiseSmartPartActivated(WorkspaceEventArgs e);
        //触发SmartPartClosing事件。
        void RaiseSmartPartClosing(WorkspaceCancelEventArgs e);
        //转换SmartPart。
        TSmartPartInfo ConvertFrom(ISmartPartInfo source);
    }
}
```

7.6 SmartPartAttribute

用于声明一个控件为 SmartPart。

```
using System;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    [AttributeUsage(AttributeTargets.Class)]
    public sealed class SmartPartAttribute : Attribute
    {
    }
}
```

7.7 SmartPartInfo

描述 SmartPart 的信息。

```
using System;
using System.ComponentModel;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.SmartParts
```

```
{
    ublic partial class SmartPartInfo : ISmartPartInfo
    {
        //描述。
        private string description = string.Empty;
        //标题。
        private string title = string.Empty;

        public SmartPartInfo()
        {
        }
        public SmartPartInfo(string title, string description)
        {
            this.title = title;
            this.description = description;
        }
        //SmartPart的描述。
        [Category("Layout")]
        public string Description
        {
            get { return description; }
            set { description = value; }
        }
        //SmartPart的标题。
        [Category("Layout")]
        public string Title
        {
            get { return title; }
            set { title = value; }
        }
```

```
        //转换成另一个SmartPart。
        public static TSmartPartInfo ConvertTo<TSmartPartInfo>(ISmartPartInfo source)
            where TSmartPartInfo : ISmartPartInfo, new()
        {
            Guard.ArgumentNotNull(source, "source");
            TSmartPartInfo info = new TSmartPartInfo();
            info.Description = source.Description;
            info.Title = source.Title;
            return info;
        }
    }
}
```

7.8 SmartPartInfoProvider

SmartPartInfo 提供商。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    [DesignTimeVisible(false)]
    public class SmartPartInfoProvider : Component, ISmartPartInfoProvider
    {
        //SmartPart信息集合。
        private List<ISmartPartInfo> items = new List<ISmartPartInfo>();
        //SmartPart信息集合。
        [DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)]
        [Browsable(false)]
        public ICollection<ISmartPartInfo> Items
```

```
        {
            get { return items; }
        }
        //根据SmartPartInfo的类型获取一个SmartPartInfo。
        public ISmartPartInfo GetSmartPartInfo(Type smartPartInfoType)
        {
            return items.Find(delegate(ISmartPartInfo info)
            {
                return info != null && info.GetType() == smartPartInfoType;
            });
        }
    }
}
```

7.9 SmartPartPlaceHolderEventArgs

```
using System;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public class SmartPartPlaceHolderEventArgs : EventArgs
    {
        private object smartPart;
        public SmartPartPlaceHolderEventArgs(object smartPart)
        {
            Guard.ArgumentNotNull(smartPart, "smartPart");
            this.smartPart = smartPart;
        }
        public object SmartPart
        {
```

```
            get { return smartPart; }
        }
    }
}
```

7.10 WorkspaceEventArgs

```
using System;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public class WorkspaceEventArgs : EventArgs
    {
        private object smartPart;
        public WorkspaceEventArgs(object smartPart)
        {
            this.smartPart = smartPart;
        }
        public object SmartPart
        {
            get { return smartPart; }
        }
    }
}
```

7.11 WorkspaceCancelEventArgs

```
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public class WorkspaceCancelEventArgs : WorkspaceEventArgs
    {
        private bool cancel;
```

```
        public WorkspaceCancelEventArgs(object smartPart)
            : base(smartPart)
        {
        }
        public bool Cancel
        {
            get { return cancel; }
            set { cancel = value; }
        }
    }
}
```

7.12 Workspace

IWorkspace 接口的默认实现。实现 Workspace 的公共行为，包括激活 SmartPart，获取注册的组件，触发事件。

注意：

（1）该类是 Workspace 的模板类，它实现了 IWorkspace 接口，但不实现具体实现。如 Activate 方法，该类首先检查需要激活的 SmartPart 是否与 Workspace 声明的 TsmartPart 兼容，如果不是抛出异常；然后检查当前 SmartPart 是否已经存在，如果不是则抛出异常；其次，调用子类实现的抽象方法 OnActivate；最后触发激活事件。对于不同的 Workspace，它只需要继承 Workspace 并在 OnActivate 方法中实现具体的激活逻辑。

（2）一些 Workspace 集成于控件，如 TabWorkspace，它不能继承 Workspace。因此引入的 IComposableWorkspace 接口和 ComposableWorkspace。IComposableWorkspace 定义了 Workspace 需要被具体实现的方法的接口，它继承了 IWorkspace。ComposableWorkspace 继承了 Workspace 并实现 IComposableWorkspace 接口，其构造器传入一个实现 IComposableWorkspace 接口的类。在 TabWorkspace，它继承了 IComposableWorkspace 接口并创建了

ComposableWorkspace 的实例，在它对 IWorkspace 接口的实现中，直接调用 ComposableWorkspace 实例的相应方法。举个例子。

```
Interface IActivator
{
    void Activate(object smartPart);
}

public abstract class Activator
{
    public void Activate(object smartPart)
    {
        ......
        OnActivate(smartPart);
        ......
    }
    abstract void OnActivate(object smartPart);
}

Interface IComposableActivator : IActivator
{
    OnActivate(object smartPart);
}

class ComposableActivator : Activator, IComposableActivator
{
    private IComposableActivator composedActivator;

    public ComposableActivator(IComposableActivator composedActivator)
    {
```

```
        this.composedActivator = composedActivator;
    }

    public void OnActivate(object smartPart)
    {
        composedActivator.OnActivate(smartPart);
    }
}

class CustomActivator : SomeClas, IComposableActivator
{
    private IComposableActivator composedActivator;

    public CustomActivator()
    {
        composedActivator = new ComposableActivator(this);
    }

    public void Activate(object smartPart)
    {
        //Call the Activate method implemented in Activator.
        //And then it will call the OnActivate of CustomActivator.
        composedActivator.Activate(smartPart);
    }

    Public void OnActivate(object smartPart)
    {
        //Do actual activation for this activator.
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Globalization;
using Microsoft.Practices.CompositeUI.Utility;
using System.ComponentModel;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    //TSmartPart：Workspace管理的SmartPart类型。
    //TSmartPartInfo：Workspace使用的SmartPart信息类的类型。
    public abstract class Workspace<TSmartPart, TSmartPartInfo> : IWorkspace
        where TSmartPartInfo : ISmartPartInfo, new()
    {
        //所在WorkItem。
        private WorkItem workItem;
        //当前激活的SmartPart。
        private object activeSmartPart;
        //所有SmartPart。
        private List<TSmartPart> smartParts = new List<TSmartPart>();
        //自动注入创建Workspace的WorkItem。
        [ServiceDependency]
        public WorkItem WorkItem
        {
            set { workItem = value; }
        }

        #region IWorkspace成员。
        //正在关闭SmartPart事件。
```

```
public event EventHandler<WorkspaceCancelEventArgs> SmartPartClosing;
//SmartPart激活事件。
public event EventHandler<WorkspaceEventArgs> SmartPartActivated;
//当前激活的SmartPart。
[Browsable(false)]
[DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)]
public object ActiveSmartPart
{
    get { return activeSmartPart; }
}
//设置激活的SmartPart。
protected void SetActiveSmartPart(object value)
{
    if (value != null)
    {
        ThrowIfUnsupportedSP(value);
        ThrowIfSmartPartNotShownPreviously((TSmartPart)value);
    }
    activeSmartPart = value;
}
//获取SmartPart集合。
[Browsable(false)]
[DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)]
public ReadOnlyCollection<object> SmartParts
{
    get
    {
        return new ReadOnlyCollection<object>(smartParts.ConvertAll<object>(
            delegate(TSmartPart smartPart)
            {
```

```
                return (object)smartPart;
            }));
        }
    }
    //激活SmartPart。
    //OnActive和RaiseSmartPartActived函数在激活一个未激活的SmartPart时被调用。
    public void Activate(object smartPart)
    {
        Guard.ArgumentNotNull(smartPart, "smartPart");
        ThrowIfUnsupportedSP(smartPart);
        ThrowIfSmartPartNotShownPreviously((TSmartPart)smartPart);
        //如果未被激活。
        if (activeSmartPart != smartPart)
        {
            OnActivate((TSmartPart)smartPart);
            activeSmartPart = smartPart;
            RaiseSmartPartActivated(smartPart);
        }
    }
    //应用SmartPartInfo到SmartPart。
    public void ApplySmartPartInfo(object smartPart, ISmartPartInfo smartPartInfo)
    {
        Guard.ArgumentNotNull(smartPart, "smartPart");
        Guard.ArgumentNotNull(smartPartInfo, "smartPartInfo");
        ThrowIfUnsupportedSP(smartPart);
        ThrowIfSmartPartNotShownPreviously((TSmartPart)smartPart);

        TSmartPartInfo typedInfo = GetSupportedSPI(smartPartInfo);
        //调用OnApplySmartPartInfo函数，来实现应用。它不会触发激活。
        OnApplySmartPartInfo((TSmartPart)smartPart, typedInfo);
```

```
}
//显示指定的SmartPart，并使用Info。
public void Show(object smartPart, ISmartPartInfo smartPartInfo)
{
    Guard.ArgumentNotNull(smartPart, "smartPart");
    Guard.ArgumentNotNull(smartPartInfo, "smartPartInfo");
    ThrowIfUnsupportedSP(smartPart);

    TSmartPartInfo typedInfo = GetSupportedSPI(smartPartInfo);
    TSmartPart typedSmartPart = (TSmartPart)smartPart;
    //如果已经包含了当前SmartPart。
    if (smartParts.Contains(typedSmartPart))
    {
        //应用Info并激活。
        ApplySmartPartInfo(smartPart, smartPartInfo);
        Activate(smartPart);
    }
    else
    {
        //添加并调用OnShow。
        smartParts.Add(typedSmartPart);
        OnShow(typedSmartPart, typedInfo);
    }
}
//显示一个SmartPart，获取相应的Info类。
public void Show(object smartPart)
{
    Guard.ArgumentNotNull(smartPart, "smartPart");
    ThrowIfUnsupportedSP(smartPart);
    TSmartPart typedSmartPart = (TSmartPart)smartPart;
```

```
//如果已经包含，则直接激活。
if (smartParts.Contains(typedSmartPart))
{
    Activate(smartPart);
}
else
{
    //1.获取Info类。
    ISmartPartInfo info = null;
    ISmartPartInfoProvider provider = smartPart as ISmartPartInfoProvider;
    if (workItem != null)
    {
        //首先从WorkItem获取TSmartPartInfo的Info实例。
        info = workItem.GetSmartPartInfo<TSmartPartInfo>(smartPart);
        //从Provider获取TSmartPartInfo的Info实例。
        if (info == null && provider != null)
        {
            info = provider.GetSmartPartInfo(typeof(TSmartPartInfo));
        }
        //从WorkItem获取SmartPartInfo的实例。
        if (info == null)
        {
            info = workItem.GetSmartPartInfo<SmartPartInfo>(smartPart);
        }
        //从Provide获取SmartPartInfo的实例。
        if (info == null && provider != null)
        {
            info = provider.GetSmartPartInfo(typeof(SmartPartInfo));
        }
    }
```

```
            else if (provider != null)
            {
                info = provider.GetSmartPartInfo(typeof(TSmartPartInfo));
                if (info == null)
                {
                    info = provider.GetSmartPartInfo(typeof(SmartPartInfo));
                }
            }
            //如果Info为空，则创建默认的SmartPartInfo。
            if (info == null)
            {
                info = CreateDefaultSmartPartInfo(typedSmartPart);
            }
            //调用Show方法，执行显示。
            Show(typedSmartPart, info);
        }
    }
    //隐藏SmartPart，调用OnHide实现隐藏。
    public void Hide(object smartPart)
    {
        Guard.ArgumentNotNull(smartPart, "smartPart");
        ThrowIfUnsupportedSP(smartPart);
        ThrowIfSmartPartNotShownPreviously((TSmartPart)smartPart);

        if (activeSmartPart == smartPart) activeSmartPart = null;
        OnHide((TSmartPart)smartPart);
    }
    //关闭SmartPart，调用CloseInternal方法实现。
    public void Close(object smartPart)
    {
```

```
        Guard.ArgumentNotNull(smartPart, "smartPart");
        ThrowIfUnsupportedSP(smartPart);
        ThrowIfSmartPartNotShownPreviously((TSmartPart)smartPart);
        //触发SmartPartClosing事件。
        WorkspaceCancelEventArgs cancelArgs = RaiseSmartPartClosing(smartPart);
        if (cancelArgs.Cancel == false)
        {
            //关闭。
            CloseInternal((TSmartPart)smartPart);
        }
    }
    #endregion

    #region Private methods
    //获取TSmartPartInfo的转换。
    private TSmartPartInfo GetSupportedSPI(ISmartPartInfo smartPartInfo)
    {
        if (typeof(TSmartPartInfo).IsAssignableFrom(smartPartInfo.GetType()) == false)
        {
            return ConvertFrom(smartPartInfo);
        }
        else
        {
            return (TSmartPartInfo)smartPartInfo;
        }
    }
    //判断当前SmartPart是否与TSmartPart兼容。
    //当前smartPart对象的类必须是TSmartPart的子类。
    private void ThrowIfUnsupportedSP(object smartPart)
    {
```

```
        if (typeof(TSmartPart).IsAssignableFrom(smartPart.GetType()) == false)
        {
            throw new ArgumentException(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.UnsupportedSmartPartType,
                smartPart.GetType(), typeof(TSmartPart)), "smartPart");
        }
    }
    //如果当前smartPart未被显示过，抛出异常。
    private void ThrowIfSmartPartNotShownPreviously(TSmartPart smartPart)
    {
        if (smartParts.Contains(smartPart) == false)
        {
            throw new ArgumentException(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.SmartPartNotInWorkspace,
                smartPart, this));
        }
    }
    //触发SmartPartActivated事件。
    protected void RaiseSmartPartActivated(object smartPart)
    {
        if (this.SmartPartActivated != null)
        {
            WorkspaceEventArgs args = new WorkspaceEventArgs(smartPart);
            this.SmartPartActivated(this, args);
        }
    }
    //触发关闭事件。
    protected WorkspaceCancelEventArgs RaiseSmartPartClosing(object smartPart)
```

```
{
    WorkspaceCancelEventArgs cancelArgs = new WorkspaceCancelEventArgs(smartPart);
    RaiseSmartPartClosing(cancelArgs);

    return cancelArgs;
}
//触发关闭事件。
protected void RaiseSmartPartClosing(WorkspaceCancelEventArgs e)
{
    if (this.SmartPartClosing != null)
    {
        this.SmartPartClosing(this, e);
    }
}
#endregion

#region Abstract members
//激活SmartPart，由子类实现。
protected abstract void OnActivate(TSmartPart smartPart);
//应用SmartPart，由子类实现。
protected abstract void OnApplySmartPartInfo(TSmartPart smartPart,
    TSmartPartInfo smartPartInfo);
//显示SmartPart，由子类实现。
protected abstract void OnShow(TSmartPart smartPart, TSmartPartInfo smartPartInfo);
//隐藏SmartPart，由子类实现。
protected abstract void OnHide(TSmartPart smartPart);
//关闭SmartPart，由子类实现。
protected abstract void OnClose(TSmartPart smartPart);
#endregion
```

```
#region Protected Mehtods
//SmartPart集合。
protected ICollection<TSmartPart> InnerSmartParts
{
    get { return smartParts; }
}
//转换SmartPartInfo。
protected virtual TSmartPartInfo ConvertFrom(ISmartPartInfo source)
{
    return SmartPartInfo.ConvertTo<TSmartPartInfo>(source);
}
//创建默认Info。当没有Info类可用时，自动创建。
protected virtual TSmartPartInfo CreateDefaultSmartPartInfo(
    TSmartPart forSmartPart)
{
    return new TSmartPartInfo();
}
//关闭SmartPart。
protected void CloseInternal(TSmartPart smartPart)
{
    OnClose((TSmartPart)smartPart);
    //如果当前激活对象为关闭对象，则需要设置激活SmartPart为空。
    if (activeSmartPart != null && activeSmartPart.Equals(smartPart))
    {
        activeSmartPart = null;
    }
    //删除该SmartPart。
    smartParts.Remove((TSmartPart)smartPart);
}
#endregion
```

```
    }
}
```

7.13 WorkspaceComposer

WorkspaceComposer 类允许你通过实现 IWorkspaceComposer 接口不直接继承 Workspace 而使用 Workspace 的功能。该类继承于 Workspace，实现了其抽象接口。它是自定义 WorkspaceComposer 的包装器。

```
using System;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.SmartParts
{
    public class WorkspaceComposer<TSmartPart, TSmartPartInfo> :
        Workspace<TSmartPart, TSmartPartInfo>
        where TSmartPartInfo : ISmartPartInfo, new()
    {
        //ComposableWorkspace接口的实例。
        IComposableWorkspace<TSmartPart, TSmartPartInfo> composedWorkspace;
        //初始化一个WorkspaceComposer，即Workspace组合器。
        public WorkspaceComposer(
            IComposableWorkspace<TSmartPart, TSmartPartInfo> composedWorkspace)
        {
            Guard.ArgumentNotNull(composedWorkspace, "composedWorkspace");
            this.composedWorkspace = composedWorkspace;
            //事件的默认实现。
            base.SmartPartActivated += new EventHandler<WorkspaceEventArgs>(
                OnSmartPartActivatedEvent);
            base.SmartPartClosing += new EventHandler<WorkspaceCancelEventArgs>(
                OnSmartPartClosingEvent);
        }
```

```
//设置激活的SmartPart。
public void SetActiveSmartPart(TSmartPart smartPart)
{
    base.SetActiveSmartPart((object)smartPart);
}
//强制关闭SmartPart而不触发SmartPartClosing事件。
public void ForceClose(TSmartPart smartPart)
{
    base.CloseInternal(smartPart);
}
//触发关闭事件。
void OnSmartPartClosingEvent(object sender, WorkspaceCancelEventArgs e)
{
    composedWorkspace.RaiseSmartPartClosing(e);
}
//触发激活事件。
void OnSmartPartActivatedEvent(object sender, WorkspaceEventArgs e)
{
    composedWorkspace.RaiseSmartPartActivated(e);
}
//激活SmartPart。
protected override void OnActivate(TSmartPart smartPart)
{
    composedWorkspace.OnActivate(smartPart);
}
//应用SmartPartInfo。
protected override void OnApplySmartPartInfo(TSmartPart smartPart,
    TSmartPartInfo smartPartInfo)
{
    composedWorkspace.OnApplySmartPartInfo(smartPart, smartPartInfo);
```

```
        }
        //显示SmartPart。
        protected override void OnShow(TSmartPart smartPart, TSmartPartInfo smartPartInfo)
        {
            composedWorkspace.OnShow(smartPart, smartPartInfo);
        }
        //隐藏SmartPart。
        protected override void OnHide(TSmartPart smartPart)
        {
            composedWorkspace.OnHide(smartPart);
        }
        //关闭SmartPart。
        protected override void OnClose(TSmartPart smartPart)
        {
            composedWorkspace.OnClose(smartPart);
        }
        //转换SmartPartInfo。
        protected override TSmartPartInfo ConvertFrom(ISmartPartInfo source)
        {
            return composedWorkspace.ConvertFrom(source);
        }
    }
}
```

8 服务（Services）

8.1 IAuthenticationService

认证服务接口。为当前用户执行认证。

```
namespace Microsoft.Practices.CompositeUI.Services
```

```
{
    public interface IAuthenticationService
    {
        //该方法希望用于设置一个IPrincipal接口实现类的实例的属性，并且可能调用
        //System.Appdomian.SetPrincipalPolicy和System.AppDomain.SetThreadPrincipal。
        void Authenticate();
    }
}
```

8.2 WindowsPrincipalAuthenticationService

Windows 标识认证服务。

```
using System;
using System.Security.Principal;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class WindowsPrincipalAuthenticationService : IAuthenticationService
    {
        public void Authenticate()
        {
            //设置当前标识为Windows标识。
            AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
        }
    }
}
```

8.3 ICryptographyService

提供加密和解密功能的服务接口。

```
namespace Microsoft.Practices.CompositeUI.Services
{
```

```
    public interface ICryptographyService
    {
        //使用指定对称加密算法提供商加密。
        byte[] EncryptSymmetric(byte[] plainText);
        //解密。
        byte[] DecryptSymmetric(byte[] cipherText);

    }
}
```

8.4 DataProctectionCryptographyService

使用数据保护 API 提供加密服务。

```
using System;
using System.Collections.Specialized;
using System.Security.Cryptography;
using Microsoft.Practices.CompositeUI.Utility;

namespace Microsoft.Practices.CompositeUI.Services
{
    public class DataProtectionCryptographyService : ICryptographyService, IConfigurable
    {
        //加密的熵。
        private byte[] entropy = null;
        private DataProtectionScope scope = DataProtectionScope.CurrentUser;
        //使用数据保护API加密数据。
        public byte[] EncryptSymmetric(byte[] plainText)
        {
            return ProtectedData.Protect(plainText, entropy, scope);
        }
```

```
        //解密。
        public byte[] DecryptSymmetric(byte[] cipherText)
        {
            return ProtectedData.Unprotect(cipherText, entropy, scope);
        }

        #region IConfigurable Members
        //配置信息。
        //加密的熵，加密范围。
        public void Configure(NameValueCollection settings)
        {
            Guard.ArgumentNotNull(settings, "settings");
            if (!String.IsNullOrEmpty(settings["Entropy"]))
            {
                entropy = Convert.FromBase64String(settings["Entropy"]);
            }
            if (!String.IsNullOrEmpty(settings["Scope"]))
            {
                scope = (DataProtectionScope) Enum.Parse(
                    typeof (DataProtectionScope), settings["Scope"]);
            }
            //删除配置信息。
            settings.Clear();
        }

        #endregion
    }
}
```

8.5 IModuleEnumerator

模块枚举服务。用于列举所有的模块。

```
using Microsoft.Practices.CompositeUI.Configuration;

namespace Microsoft.Practices.CompositeUI.Services
{
    public interface IModuleEnumerator
    {
        //获取所有模块信息。
        IModuleInfo[] EnumerateModules();
    }
}
```

8.6 ModuleEnumeratorException

```
using System;
using System.Runtime.Serialization;

namespace Microsoft.Practices.CompositeUI.Services
{
    [Serializable]
    public class ModuleEnumeratorException : Exception
    {
        public ModuleEnumeratorException() : base() { }
        public ModuleEnumeratorException(string message) : base(message) { }
        public ModuleEnumeratorException(string message, Exception innerException) :
            base(message, innerException) { }
        protected ModuleEnumeratorException(SerializationInfo info, StreamingContext context) :
            base(info, context) { }
    }
}
```

8.7 ReflectionModuleEnumerator

获取应用系统工作目录所有 DLL 文件的模块。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection;
using Microsoft.Practices.CompositeUI.Configuration;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class ReflectionModuleEnumerator : IModuleEnumerator
    {
        //工作目录。
        private string basePath;
        //初始化。
        public ReflectionModuleEnumerator()
        {
            basePath = AppDomain.CurrentDomain.BaseDirectory;
        }
        //工作目录。
        public string BasePath
        {
            get { return basePath; }
            set { basePath = value; }
        }
        //获取模块。
        public IModuleInfo[] EnumerateModules()
        {
            List<ModuleInfo> modules = new List<ModuleInfo>();
            List<string> directories = new List<string>();
```

```
            //目录列表。用于遍历。
            directories.Add(BasePath);
            while (directories.Count > 0)
            {
                string directory = directories[0];
                directories.Remove(directory);
                //获取DLL文件。
                foreach (string filename in Directory.GetFiles(directory, "*.dll"))
                {
                    //装载Assembly。
                    Assembly assm = Assembly.LoadFile(filename);
                    if (AssemblyHasModuleAttribute(assm))
                    {
                        //获取模块Info。
                        ModuleInfo mi = new ModuleInfo();
                        mi.SetAssemblyFile(filename);
                        modules.Add(mi);
                    }
                }
                directories.AddRange(Directory.GetDirectories(directory));
            }

            return modules.ToArray();
        }
        //是否声明了ModuleAttribut。
        private static bool AssemblyHasModuleAttribute(Assembly assm)
        {
            return assm.GetCustomAttributes(typeof (ModuleAttribute), false).Length > 0;
        }
    }
```

```
}
```

8.8 FileCatalogModuleEnumerator

文件目录模块枚举器。从 SolutionProfile.xml 文件读取模块配置。

```
using Microsoft.Practices.CompositeUI.Configuration;
using Microsoft.Practices.CompositeUI.Utility;
using System;
using System.Globalization;
using System.Collections.Specialized;
using Microsoft.Practices.CompositeUI.Configuration.Xsd;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class FileCatalogModuleEnumerator : IModuleEnumerator, IConfigurable
    {
        //目录文件路径。
        private string catalogFilePath = SolutionProfileReader.DefaultCatalogFile;
        public FileCatalogModuleEnumerator()
        {
        }
        public FileCatalogModuleEnumerator(string catalogFilePath)
        {
            Guard.ArgumentNotNullOrEmptyString(catalogFilePath, "catalogFilePath");
            this.catalogFilePath = catalogFilePath;
        }
        //获取模块信息。
        public IModuleInfo[] EnumerateModules()
        {
            try
            {
```

```
            SolutionProfileReader reader = new SolutionProfileReader(catalogFilePath);
            SolutionProfileElement profile = reader.ReadProfile();
            return CreateModuleInfos(profile);
        }
        catch (Exception ex)
        {
            throw new ModuleEnumeratorException(String.Format(CultureInfo.CurrentCulture,
            Properties.Resources.ErrorEnumeratingModules,
            this), ex);
        }
    }

    public string CatalogFilePath
    {
        get { return catalogFilePath; }
        set { catalogFilePath = value; }
    }
    //读取配置文件。
    public void Configure(NameValueCollection settings)
    {
        Guard.ArgumentNotNull(settings, "settings");

        if (!String.IsNullOrEmpty(settings["filePath"]))
        {
            catalogFilePath = settings["filePath"];
        }
    }

    private static IModuleInfo[] CreateModuleInfos(SolutionProfileElement solutionProfile)
    {
```

```
                ModuleInfo[] mInfos = new ModuleInfo[solutionProfile.Modules.Length];
                for (int i = 0; i < solutionProfile.Modules.Length; i++)
                {
                    ModuleInfoElement xsdModule = solutionProfile.Modules[i];
                    ModuleInfo mInfo = new ModuleInfo(xsdModule.AssemblyFile);
                    mInfo.SetUpdateLocation(xsdModule.UpdateLocation);
                    if (xsdModule.Roles != null && xsdModule.Roles.Length > 0)
                    {
                        foreach (RoleElement role in xsdModule.Roles)
                        {
                            mInfo.AddRoles(role.Allow);
                        }
                    }
                    mInfos[i] = mInfo;
                }
                return mInfos;
            }
        }
}
```

8.9 IModuleLoaderService

执行模块装载的服务。

```
using System;
using System.Collections.Generic;
using System.Reflection;
using Microsoft.Practices.CompositeUI.Configuration;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.Services
{
```

```
    public interface IModuleLoaderService
    {
        //已装载的模块列表。
        IList<LoadedModuleInfo> LoadedModules { get; }
        //装载指定的模块。
        void Load(WorkItem workItem, params IModuleInfo[] modules);
        //装载指定的作为模块的程序集。
        void Load(WorkItem workItem, params Assembly[] assemblies);
        //模块已装载事件。
        event EventHandler<DataEventArgs<LoadedModuleInfo>> ModuleLoaded;
    }
}
```

8.10 LoadedModuleInfo

装载的模块信息。

```
using System;
using System.Collections.Generic;
using System.Reflection;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class LoadedModuleInfo
    {
        //所在程序集。
        private Assembly assembly;
        //依赖的模块列表。
        private List<string> dependencies;
        //模块名称。
        private string name;
        //授权角色名称列表。
```

```
private List<string> roles;


public LoadedModuleInfo(Assembly assembly, string name,
        IEnumerable<string> roles, IEnumerable<string> dependencies)
{
        this.assembly = assembly;
        this.name = name;
        this.roles = new List<string>();
        if (roles != null)
                this.roles.AddRange(roles);
        this.dependencies = new List<string>();
        if (dependencies != null)
                this.dependencies.AddRange(dependencies);
}
//拷贝一个模块。
public LoadedModuleInfo(LoadedModuleInfo toCopy)
{
        this.assembly = toCopy.assembly;
        this.name = toCopy.name;
        this.roles = new List<string>(toCopy.roles);
        this.dependencies = new List<string>(toCopy.dependencies);
}
//程序集。
public Assembly Assembly
{
        get { return assembly; }
}
//名称。
public string Name
{
```

```
            get { return name; }
        }
        //依赖列表。
        public IList<string> Dependencies
        {
            get { return dependencies.AsReadOnly(); }
        }
        //角色列表。
        public IList<string> Roles
        {
            get { return roles.AsReadOnly(); }
        }
    }
}
```

8.11 ModuleLoadException

模块装载异常类。

```
using System;
using System.Globalization;
using System.Runtime.Serialization;
using Microsoft.Practices.CompositeUI.Properties;
namespace Microsoft.Practices.CompositeUI.Services
{
    [Serializable]
    public class ModuleLoadException : Exception
    {
        public ModuleLoadException()
            : base()
        {
```

```
        }
        public ModuleLoadException(string message)
            : base(message)
        {
        }
        public ModuleLoadException(string message, Exception exception)
            : base(message, exception)
        {
        }
        public ModuleLoadException(string moduleAssembly, string message)
            : base(String.Format(CultureInfo.CurrentCulture,
            Resources.FailedToLoadModule, moduleAssembly, message))
        {
        }
        public ModuleLoadException(string moduleAssembly, string message, Exception innerException)
            : base(String.Format(CultureInfo.CurrentCulture,
            Resources.FailedToLoadModule, moduleAssembly, message), innerException)
        {
        }
        protected ModuleLoadException(SerializationInfo info, StreamingContext context)
            : base(info, context)
        {
        }
    }
}
```

8.12 ModuleLoaderService

模块装载器服务。将模块装载到应用系统。

```
using System;
```

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Globalization;
using System.IO;
using System.Reflection;
using Microsoft.Practices.CompositeUI.Configuration;
using Microsoft.Practices.CompositeUI.Utility;
using Microsoft.Practices.ObjectBuilder;

namespace Microsoft.Practices.CompositeUI.Services
{
    public class ModuleLoaderService : IModuleLoaderService
    {
        //装载的模块列表。
        Dictionary<Assembly, ModuleMetadata> loadedModules =
            new Dictionary<Assembly, ModuleMetadata>();
        TraceSource traceSource = null;
        //注入构造器。
        [InjectionConstructor]
        public ModuleLoaderService([ClassNameTraceSource] TraceSource traceSource)
        {
            this.traceSource = traceSource;
        }
        //模块装载事件。
        public event EventHandler<DataEventArgs<LoadedModuleInfo>> ModuleLoaded;
        //装载的模块信息集合。
        public IList<LoadedModuleInfo> LoadedModules
        {
            get
            {
```

```
            List<LoadedModuleInfo> result = new List<LoadedModuleInfo>();

            foreach (ModuleMetadata module in loadedModules.Values)
                result.Add(module.ToLoadedModuleInfo());

            return result.AsReadOnly();
        }
    }
    //装载。
    public void Load(WorkItem workItem, params IModuleInfo[] modules)
    {
        Guard.ArgumentNotNull(workItem, "workItem");
        Guard.ArgumentNotNull(modules, "modules");

        InnerLoad(workItem, modules);
    }
    //装载。
    public void Load(WorkItem workItem, params Assembly[] assemblies)
    {
        Guard.ArgumentNotNull(workItem, "workItem");
        Guard.ArgumentNotNull(assemblies, "assemblies");

        List<IModuleInfo> modules = new List<IModuleInfo>();

        foreach (Assembly assembly in assemblies)
            modules.Add(new ModuleInfo(assembly));

        InnerLoad(workItem, modules.ToArray());
    }
    //模块装载处理。
```

```
protected virtual void OnModuleLoaded(LoadedModuleInfo module)
{
    if (ModuleLoaded != null)
        ModuleLoaded(this, new DataEventArgs<LoadedModuleInfo>(module));
}
//执行模块装载。
private void InnerLoad(WorkItem workItem, IModuleInfo[] modules)
{
    if (modules.Length == 0)
        return;
    //装载程序集。
    LoadAssemblies(modules);
    //模块装载顺序。
    List<ModuleMetadata> loadOrder = GetLoadOrder();
    //先装载每一个模块的服务。
    foreach (ModuleMetadata module in loadOrder)
        module.LoadServices(workItem);
    //初始化每一个模块。
    foreach (ModuleMetadata module in loadOrder)
        module.InitializeModuleClasses(workItem);
    //初始化每一个模块的扩展。
    foreach (ModuleMetadata module in loadOrder)
        module.InitializeWorkItemExtensions(workItem);
    //通知模块装载。
    foreach (ModuleMetadata module in loadOrder)
        module.NotifyOfLoadedModule(OnModuleLoaded);
}
//获取模块装载顺序列表。
private List<ModuleMetadata> GetLoadOrder()
{
```

```
        Dictionary<string, ModuleMetadata> indexedInfo = new Dictionary<string, ModuleMetadata>();
        ModuleDependencySolver solver = new ModuleDependencySolver();
        List<ModuleMetadata> result = new List<ModuleMetadata>();

        foreach (ModuleMetadata data in loadedModules.Values)
        {
            if (indexedInfo.ContainsKey(data.Name))
                throw new ModuleLoadException(String.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.DuplicatedModule, data.Name));

            indexedInfo.Add(data.Name, data);
            solver.AddModule(data.Name);

            foreach (string dependency in data.Dependencies)
                solver.AddDependency(data.Name, dependency);
        }

        if (solver.ModuleCount > 0)
        {
            string[] loadOrder = solver.Solve();

            for (int i = 0; i < loadOrder.Length; i++)
                result.Add(indexedInfo[loadOrder[i]]);
        }

        return result;
    }
    //装载程序集。
    private void LoadAssemblies(IModuleInfo[] modules)
    {
```

```
        foreach (IModuleInfo module in modules)
        {
            GuardLegalAssemblyFile(module);
            Assembly assembly = LoadAssembly(module.AssemblyFile);

            if (loadedModules.ContainsKey(assembly))
                continue;

            loadedModules.Add(assembly, new ModuleMetadata(assembly, traceSource));
        }
    }
    //装载一个程序集。
    private Assembly LoadAssembly(string assemblyFile)
    {
        Guard.ArgumentNotNullOrEmptyString(assemblyFile, "assemblyFile");

        assemblyFile = GetModulePath(assemblyFile);

        FileInfo file = new FileInfo(assemblyFile);
        Assembly assembly = null;

        try
        {
            assembly = Assembly.LoadFrom(file.FullName);
        }
        catch (Exception ex)
        {
            throw new ModuleLoadException(assemblyFile, ex.Message, ex);
        }
```

```
        if (traceSource != null)
            traceSource.TraceInformation(Properties.Resources.LogModuleAssemblyLoaded,
                file.FullName);

        return assembly;
    }
    //获取模块路径。
    private string GetModulePath(string assemblyFile)
    {
        if (Path.IsPathRooted(assemblyFile) == false)
            assemblyFile = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, assemblyFile);

        return assemblyFile;
    }

    #region Guards
    //如果模块信息的程序集文件路径不对，则抛出异常。
    private void GuardLegalAssemblyFile(IModuleInfo modInfo)
    {
        Guard.ArgumentNotNull(modInfo, "modInfo");
        Guard.ArgumentNotNull(modInfo.AssemblyFile, "modInfo.AssemblyFile");

        string assemblyFilePath = GetModulePath(modInfo.AssemblyFile);

        if (File.Exists(assemblyFilePath) == false)
        {
            throw new ModuleLoadException(
                string.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.ModuleNotFound, assemblyFilePath));
        }
```

```
}

#endregion

#region Helper classes
//模块元素据。
class ModuleMetadata
{
    //程序集。
    Assembly assembly;
    //是否需要装载服务。
    bool loadedServices = false;
    //是否扩展初始化。
    bool extensionsInitialized = false;
    //模块是否已初始化。
    bool modulesInitialzed = false;
    //名称。
    string name = null;
    //是否通知。
    bool notified = false;
    //依赖列表。
    List<string> dependencies = new List<string>();
    //实现IModule类型列表。
    List<Type> moduleTypes = new List<Type>();
    //实现IModule的类实例列表。
    List<IModule> moduleClasses = new List<IModule>();
    //使用角色。
    List<string> roles = new List<string>();
    //服务元素据列表。
    List<ServiceMetadata> services = new List<ServiceMetadata>();
```

```
//WorkItem扩展列表。
List<KeyValuePair<Type, Type>> workItemExtensions = new List<KeyValuePair<Type, Type>>();
//WorkItem根节点扩展列表。
List<Type> workItemRootExtensions = new List<Type>();

TraceSource traceSource;

public ModuleMetadata(Assembly assembly, TraceSource traceSource)
{
    this.assembly = assembly;
    this.traceSource = traceSource;
    //获取模块名称。
    foreach (ModuleAttribute attr in
        assembly.GetCustomAttributes(typeof(ModuleAttribute), true))
        name = attr.Name;
    //获取模块依赖列表。
    foreach (ModuleDependencyAttribute attr in
        assembly.GetCustomAttributes(typeof(ModuleDependencyAttribute), true))
        dependencies.Add(attr.Name);
    //遍历模块所有的类型。
    foreach (Type type in assembly.GetExportedTypes())
    {
        //是否声明了服务特性。
        foreach (ServiceAttribute attr in type.GetCustomAttributes(
            typeof(ServiceAttribute), true))
            services.Add(new ServiceMetadata(type,
            attr.RegisterAs ?? type, attr.AddOnDemand));
        //是否声明了WorkItemExtension特性。
        foreach (WorkItemExtensionAttribute attr in
            type.GetCustomAttributes(typeof(WorkItemExtensionAttribute), true))
```

```
                workItemExtensions.Add(new KeyValuePair<Type, Type>(
                    attr.WorkItemType, type));
            //是否声明了根节点扩展。
            foreach (RootWorkItemExtensionAttribute attr in
                type.GetCustomAttributes(typeof(RootWorkItemExtensionAttribute), true))
                workItemRootExtensions.Add(type);
            //IModule兼容的类。
            if (!type.IsAbstract && typeof(IModule).IsAssignableFrom(type))
                moduleTypes.Add(type);
        }
    }
    //依赖。
    public IEnumerable<string> Dependencies
    {
        get { return dependencies; }
    }
    //名称，如果没有指定模块名称，则为程序集名称。
    public string Name
    {
        get
        {
            if (name == null)
                name = assembly.FullName;

            return name;
        }
        set { name = value; }
    }
    //装载服务。
    //1. 调用实现IModule接口的类的AddServices方法。
```

```
//2. 装载声明了Service特性的服务。
public void LoadServices(WorkItem workItem)
{
    if (loadedServices)
        return;

    loadedServices = true;
    EnsureModuleClassesExist(workItem);

    try
    {
        //调用IModule.AddServices添加服务。
        foreach (IModule moduleClass in moduleClasses)
        {
            moduleClass.AddServices();
            if (traceSource != null)
                traceSource.TraceInformation(
                Properties.Resources.AddServicesCalled, moduleClass.GetType());
        }
        //装载声明了Service特性的服务。
        foreach (ServiceMetadata svc in services)
        {
            if (svc.AddOnDemand)
            {
                workItem.Services.AddOnDemand(
                    svc.InstanceType, svc.RegistrationType);

                if (traceSource != null)
                traceSource.TraceInformation(
                Properties.Resources.ServiceAddedOnDemand, Name, svc.InstanceType);
```

```
                }
                else
                {
                    workItem.Services.AddNew(svc.InstanceType, svc.RegistrationType);

                    if (traceSource != null)
                        traceSource.TraceInformation
                        (Properties.Resources.ServiceAdded, Name, svc.InstanceType);
                }
            }
        }
        catch (Exception ex) { ThrowModuleLoadException(ex); }
    }
    //创建IModule类型的实例。
    private void EnsureModuleClassesExist(WorkItem workItem)
    {
        if (moduleClasses.Count == moduleTypes.Count)
            return;

        try
        {
            foreach (Type moduleType in moduleTypes)
            {
                IModule module = (IModule)workItem.Items.AddNew(moduleType);
                moduleClasses.Add(module);
                if (traceSource != null)
                    traceSource.TraceInformation(
                    Properties.Resources.LogModuleAdded, moduleType);
            }
        }
```

```
        catch (FileNotFoundException ex) { ThrowModuleReferenceException(ex); }
        catch (Exception ex) { ThrowModuleLoadException(ex); }
    }
    //初始化IModule的类的实例。调用每一个IModule类实例的Load方法。
    public void InitializeModuleClasses(WorkItem workItem)
    {
        if (modulesInitialzed)
            return;

        modulesInitialzed = true;
        EnsureModuleClassesExist(workItem);

        try
        {
            foreach (IModule module in moduleClasses)
            {
                //调用Load方法。
                module.Load();

                if (traceSource != null)
                    traceSource.TraceInformation(
                    Properties.Resources.ModuleStartCalled, module.GetType());
            }
        }
        catch (FileNotFoundException ex) { ThrowModuleReferenceException(ex); }
        catch (Exception ex) { ThrowModuleLoadException(ex); }
    }
    //初始化WorkItemExtensions。
    public void InitializeWorkItemExtensions(WorkItem workItem)
    {
```

```
        if (extensionsInitialized)
            return;

        extensionsInitialized = true;
        //获取扩展服务。
        IWorkItemExtensionService svc = workItem.Services.Get<IWorkItemExtensionService>();

        if (svc == null)
            return;
        //注册Extensions。
        foreach (KeyValuePair<Type, Type> kvp in workItemExtensions)
            svc.RegisterExtension(kvp.Key, kvp.Value);
        //注册根节点Extensions。
        foreach (Type type in workItemRootExtensions)
            svc.RegisterRootExtension(type);
    }
    //装载模块通知。Action是通知处理代理。
    public void NotifyOfLoadedModule(Action<LoadedModuleInfo> action)
    {
        if (notified)
            return;

        notified = true;
        action(ToLoadedModuleInfo());
    }
    //获取通知的ModuleInfo。
    public LoadedModuleInfo ToLoadedModuleInfo()
    {
        return new LoadedModuleInfo(assembly, Name, roles, dependencies);
    }
```

```
        //抛出模块装载异常。
        private void ThrowModuleLoadException(Exception innerException)
        {
            throw new ModuleLoadException(Name,
                        String.Format(CultureInfo.CurrentCulture,
                        Properties.Resources.FailedToLoadModule,
                        assembly.FullName, innerException.Message), innerException);
        }
        //抛出模块引用异常。
        private void ThrowModuleReferenceException(Exception innerException)
        {
            throw new ModuleLoadException(Name,
                        Properties.Resources.ReferencedAssemblyNotFound,
                        innerException);
        }
    }
    //服务元素据。
    class ServiceMetadata
    {
        //按需创建服务。
        public bool AddOnDemand = false;
        //服务实例类型。
        public Type InstanceType = null;
        //服务注册类型。
        public Type RegistrationType = null;
        public ServiceMetadata(Type instanceType, Type registrationType, bool addOnDemand)
        {
            this.InstanceType = instanceType;
            this.RegistrationType = registrationType;
            this.AddOnDemand = addOnDemand;
```

```
            }
        }

        class ClassNameTraceSourceAttribute : TraceSourceAttribute
        {
            public ClassNameTraceSourceAttribute() : base(typeof(ModuleLoaderService).FullName)
            {
            }
        }

        #endregion
    }
}
```

8.13 IStatePersistenceService

状态持久化服务。用于在存储区存储和取出数据。提供状态保存、获取、删除和包含操作。

```
using System;

namespace Microsoft.Practices.CompositeUI.Services
{
    public interface IStatePersistenceService
    {
        //存储状态。
        void Save(State state);
        //获取状态。
        State Load(string id);
        //删除状态。
        void Remove(string id);
        //是否包括状态。
```

```
        bool Contains(string id);
    }
}
```

8.14 StatePersistenceException

状态持久化异常。

```
using System;
using System.Runtime.Serialization;
namespace Microsoft.Practices.CompositeUI.Services
{
    [Serializable]
    public class StatePersistenceException : Exception
    {
        public StatePersistenceException()
            : base()
        {
        }
        public StatePersistenceException(string message)
            : base(message)
        {
        }
        public StatePersistenceException(string message, Exception innerException)
            : base(message, innerException)
        {
        }
        protected StatePersistenceException(SerializationInfo info, StreamingContext context)
            : base(info, context)
        {
        }
```

```
    }
}
```

8.15 StreamStatePersistenceService

使用 BinaryFormatter 格式化器将 State 数据存储到二进制文件。

```
using System;
using System.Collections.Specialized;
using System.Globalization;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using Microsoft.Practices.CompositeUI.Properties;

namespace Microsoft.Practices.CompositeUI.Services
{
    public abstract class StreamStatePersistenceService : IStatePersistenceService, IConfigurable
    {
        private bool useCryptography = false;
        private ICryptographyService cryptoSvc = null;
        //加密服务。
        [ServiceDependency(Required = false)]
        public ICryptographyService CryptographyService
        {
            get
            {
                if (cryptoSvc == null)
                {
                    throw new StatePersistenceException(String.Format(
                        CultureInfo.CurrentCulture,
                        Properties.Resources.ServiceMissingExceptionMessage,
```

```
                    typeof(ICryptographyService),
                    this.GetType()));
            }

            return cryptoSvc;
        }
        set { cryptoSvc = value; }
    }
    //加密配置属性名称。
    public const string UseCryptographyAttribute = "UseCryptography";
    //保存状态。
    public void Save(State state)
    {
        if (state == null)
        {
            throw new ArgumentNullException("state");
        }
        try
        {
            //如果已存在，则删除。
            if (Contains(state.ID))
            {
                Remove(state.ID);
            }
            if (useCryptography)
            {
                BinaryFormatter fmt = new BinaryFormatter();
                using (MemoryStream stream = new MemoryStream())
                {
                    //获取State的二进制信息。
```

```
                    fmt.Serialize(stream, state);
                    //加密。
                    byte[] data = CryptographyService.EncryptSymmetric(stream.GetBuffer());
                    //利用实际的Stream对象，执行流的操作。
                    OpenStream(state.ID, delegate(Stream fs)
                        {
                            fs.Write(data, 0, data.Length);
                        });
                }
            }
            else
            {
                BinaryFormatter fmt = new BinaryFormatter();
                OpenStream(state.ID, delegate(Stream stream)
                    {
                        fmt.Serialize(stream, state);
                    });
            }
        }
        catch (Exception ex)
        {
            throw new StatePersistenceException(String.Format(CultureInfo.CurrentCulture,
                Resources.CannotSaveState, state.ID),
                ex);
        }
    }
    //装载，实现方式类似Save，只不过把序列化编程反序列化。
    public State Load(string id)
    {
        if (Contains(id) == false)
```

```
{
    throw new StatePersistenceException(String.Format(
        CultureInfo.CurrentCulture,
        Resources.StateDoesNotExist, id));
}
try
{
    if (useCryptography)
    {
        byte[] cipherData = null;
        //读取数据。
        OpenStream(id, delegate(Stream stream)
                    {
                        cipherData = new byte[stream.Length];
                        stream.Read(cipherData, 0, (int)stream.Length);
                    });
        //解密数据并反序列化。
        using (MemoryStream ms =
            new MemoryStream(CryptographyService.DecryptSymmetric(cipherData)))
        {
            BinaryFormatter fmt = new BinaryFormatter();
            return (State)fmt.Deserialize(ms);
        }
    }
    else
    {
        //直接反序列化数据。
        BinaryFormatter fmt = new BinaryFormatter();
        State state = null;
        OpenStream(id, delegate(Stream stream)
```

```
                                {
                                    state = (State)fmt.Deserialize(stream);
                                });
                    return state;
                }
            }
            catch (Exception ex)
            {
                throw new StatePersistenceException(String.Format(CultureInfo.CurrentCulture,
                    Resources.CannotLoadState, id), ex);
            }
        }
        //删除操作。
        public void Remove(string id)
        {
            try
            {
                RemoveStream(id);
            }
            catch (Exception ex)
            {
                throw new StatePersistenceException(String.Format(CultureInfo.CurrentCulture,
                    Resources.CannotLoadState, id),
                    ex);
            }
        }
        //是否包含状态，由子类实现。
        public abstract bool Contains(string id);
        //如果Stream无效，则抛出异常。
        private static void ThrowIfInvalidStream(Stream stm)
```

```
{
        if (stm == null || stm.CanRead == false)
        {
                throw new StatePersistenceException(Resources.InvalidStateStream);
        }
}
//删除指定ID的Stream。由子类实现。
public abstract void RemoveStream(string id);
//获取实际操作的Stream对象。由子类实现。
protected abstract Stream GetStream(string id);
//获取实际操作的Stream对象。
protected virtual Stream GetStream(string id, out bool shouldDispose)
{
        shouldDispose = true;
        return GetStream(id);
}
//获取配置信息。
public virtual void Configure(NameValueCollection settings)
{
        if (settings == null)
        {
                throw new ArgumentNullException("settings");
        }
        if (!String.IsNullOrEmpty(settings[UseCryptographyAttribute]))
        {
                if (Boolean.TryParse(settings[UseCryptographyAttribute], out useCryptography) == false)
                {
                        throw new StatePersistenceException(
                                Resources.InvalidCryptographyValue);
                }
```

```
            }
        }
        //流操作代理类。
        private delegate void OpenStreamDelegate(Stream openedStream);
        //执行实际的流操作。
        private void OpenStream(string stateId, OpenStreamDelegate operation)
        {
            bool dispose;
            //获取当前状态ID的流处理类。
            Stream stream = GetStream(stateId, out dispose);
            try
            {
                //执行操作。
                ThrowIfInvalidStream(stream);
                operation(stream);
            }
            finally
            {
                if (dispose && stream != null)
                {
                    ((IDisposable)stream).Dispose();
                }
            }
        }
    }
}
```

8.16 FileStatePersistenceService

文件持久化服务。

```
using System;
using System.ComponentModel;
using System.Globalization;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class FileStatePersistenceService : StreamStatePersistenceService
    {
        //状态文件父目录。
        private string basePath;
        //状态文件格式。
        private readonly string fileNameFormat = "{0}.state";

        public FileStatePersistenceService() : this(AppDomain.CurrentDomain.BaseDirectory) { }
        public FileStatePersistenceService(string basePath)
        {
            BasePath = basePath;
        }
        //存储目录。
        public string BasePath
        {
            get { return basePath; }
            set { basePath = new DirectoryInfo(value).FullName; }
        }
        //是否包含。
        public override bool Contains(string id)
        {
            return File.Exists(GetFileName(id));
        }
```

```
        //直接删除文件。
        public override void RemoveStream(string id)
        {
            string filename = GetFileName(id);
            if (File.Exists(filename))
            {
                File.Delete(filename);
            }
        }
        //获取FileStream。
        protected override Stream GetStream(string id)
        {
            string filename = GetFileName(id);
            return new FileStream(filename,
                File.Exists(filename) ? FileMode.Open : FileMode.Create, FileAccess.ReadWrite);
        }
        //获取文件名称。
        private string GetFileName(string id)
        {
            return  Path.Combine(basePath,  String.Format(CultureInfo.InvariantCulture,  fileNameFormat,
id));
        }
    }
}
```

8.17 IsolatedStorageStatePersistenceService

独立存储状态持久化服务。

```
using System;
using System.Collections.Generic;
```

```
using System.Collections.Specialized;
using System.Globalization;
using System.IO;
using System.IO.IsolatedStorage;
using Microsoft.Practices.CompositeUI.Properties;

namespace Microsoft.Practices.CompositeUI.Services
{
    public class IsolatedStorageStatePersistenceService : StreamStatePersistenceService
    {
        //独立存储范围属性。范围如Assembly| User或Roaming | User | Assembly | Domain，与
        public const string ScopeAttribute = "Scope";
        IsolatedStorageScope scope;
        //默认。
        public IsolatedStorageStatePersistenceService()
            : this(IsolatedStorageScope.Roaming | IsolatedStorageScope.User |
                    IsolatedStorageScope.Assembly | IsolatedStorageScope.Domain)
        {
        }
        public IsolatedStorageStatePersistenceService(IsolatedStorageScope scope)
        {
            this.scope = scope;
        }
        //是否包含。
        public override bool Contains(string id)
        {
            //获取独立存储文件。
            IsolatedStorageFile store = IsolatedStorageFile.GetStore(scope, null, null);
            //获取当前ID的存储文件。
            string[] files = store.GetFileNames(GetFileName(id));
```

```
        return files.Length == 1;
}
//删除。
public override void RemoveStream(string id)
{
        if (Contains(id))
        {
                IsolatedStorageFile store = IsolatedStorageFile.GetStore(scope, null, null);
                store.DeleteFile(GetFileName(id));
        }
}
//获取指定ID的状态流。
protected override Stream GetStream(string id)
{
        IsolatedStorageFile store = IsolatedStorageFile.GetStore(scope, null, null);
        string fileName = GetFileName(id);
        if (Contains(id) == false)
        {
                return new IsolatedStorageFileStream(fileName, FileMode.CreateNew, store);
        }
        else
        {
                return new IsolatedStorageFileStream(fileName, FileMode.Open, store);
        }
}
//配置Scope属性。
public override void Configure(NameValueCollection settings)
{
        base.Configure(settings);
```

```
if (!String.IsNullOrEmpty(settings[ScopeAttribute]))
{
    string[] values = settings[ScopeAttribute].Split(
        new char[] {'|'}, StringSplitOptions.RemoveEmptyEntries);
    List<IsolatedStorageScope> scopes = new List<IsolatedStorageScope>(values.Length);
    foreach (string value in values)
    {
        string trimmed = value.Trim();
        if (trimmed.Length > 0)
        {
            if (Enum.IsDefined(typeof (IsolatedStorageScope), trimmed) == false)
            {
                throw new StatePersistenceException(
                    String.Format(CultureInfo.CurrentCulture,
                    Resources.InvalidIsolatedStorageStatePersistanceScope,
                     settings[ScopeAttribute], trimmed));
            }
            scopes.Add((IsolatedStorageScope) Enum.Parse(
                typeof (IsolatedStorageScope), trimmed));
        }
    }
    scope = IsolatedStorageScope.None;
    foreach (IsolatedStorageScope tmpscope in scopes)
    {
        scope |= tmpscope;
    }
    // 尝试使用所声明的范围。
    try
    {
        IsolatedStorageFile store = IsolatedStorageFile.GetStore(scope, null, null);
```

```
            }
            catch (ArgumentException aex)
            {
                throw new StatePersistenceException(
                    String.Format(CultureInfo.CurrentCulture,
                    Resources.InvalidIsolatedStorageStatePersistanceParsedScope,
                     settings[ScopeAttribute]), aex);
            }
        }
    }
    //删除所有的存储。
    public void Clear()
    {
        IsolatedStorageFile store = IsolatedStorageFile.GetStore(scope, null, null);
        string[] files = store.GetFileNames("*");
        foreach (string file in files)
        {
            store.DeleteFile(file);
        }
    }
    public IsolatedStorageScope Scope
    {
        get { return scope; }
        set { scope = value; }
    }
    //获取指定ID的状态文件。
    private string GetFileName(string id)
    {
        return String.Format(CultureInfo.InvariantCulture, "{0}.state", id);
    }
```

```
    }
}
```

8.18 IWorkItemExtensionService

用于管理 IWorkItemExtension 注册的服务。

```
using System;
using System.Collections.Generic;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.Services
{
    public interface IWorkItemExtensionService
    {
        //注册的WorkItemExtension集合。
        ReadOnlyDictionary<Type, IList<Type>> RegisteredExtensions { get; }
        //根WorkItem的扩展。
        IList<Type> RegisteredRootExtensions { get; }
        //初始化扩展。
        void InitializeExtensions(WorkItem workItem);
        //注册WorkItemExtension。
        void RegisterExtension(Type workItemType, Type extensionType);
        //注册根WorkItem的扩展。
        void RegisterRootExtension(Type extensionType);
    }
}
```

8.19 WorkItemExtensionService

默认的服务。

```
using System;
using System.ComponentModel;
```

```
using System.Globalization;
using Microsoft.Practices.CompositeUI.Properties;
using Microsoft.Practices.CompositeUI.Utility;
using System.Collections.Generic;
using System.Diagnostics;
using Microsoft.Practices.ObjectBuilder;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class WorkItemExtensionService : IWorkItemExtensionService
    {
        //所有扩展。
        ListDictionary<Type, Type> extensions = new ListDictionary<Type, Type>();
        //根节点扩展。
        List<Type> rootExtensions = new List<Type>();
        TraceSource traceSource;

        public WorkItemExtensionService()
            : this(null)
        {
        }
        [InjectionConstructor]
        public WorkItemExtensionService([ClassNameTraceSource] TraceSource traceSource)
        {
            this.traceSource = traceSource;
        }
        //注册的扩展。
        public ReadOnlyDictionary<Type, IList<Type>> RegisteredExtensions
        {
            get
            {
```

```
            Dictionary<Type, IList<Type>> results = new Dictionary<Type, IList<Type>>();
            foreach (KeyValuePair<Type, List<Type>> pair in extensions)
                results[pair.Key] = pair.Value.AsReadOnly();
            return new ReadOnlyDictionary<Type, IList<Type>>(results);
        }
    }
    //根WorkItem的扩展。
    public IList<Type> RegisteredRootExtensions
    {
        get { return rootExtensions.AsReadOnly(); }
    }
    //注册扩展。
    public void RegisterExtension(Type workItemType, Type extensionType)
    {
        Guard.ArgumentNotNull(workItemType, "workItemType");
        Guard.ArgumentNotNull(extensionType, "extensionType");
        Guard.TypeIsAssignableFromType(workItemType, typeof(WorkItem), "workItemType");
        Guard.TypeIsAssignableFromType(extensionType,
            typeof(IWorkItemExtension), "extensionType");
        ThrowIfAlreadyAdded(workItemType, extensionType);

        extensions.Add(workItemType, extensionType);

        if (traceSource != null)
            traceSource.TraceInformation(String.Format(
                CultureInfo.CurrentCulture,
                Properties.Resources.WorkItemExtensionRegistered,
                extensionType, workItemType));
    }
    private void ThrowIfAlreadyAdded(Type workItemType, Type extensionType)
```

```
{
    if (extensions.ContainsKey(workItemType) &&
        extensions[workItemType].Contains(extensionType))
    {
        throw new ArgumentException(
            string.Format(CultureInfo.CurrentCulture,
            Properties.Resources.WorkItemExtensionTypeAlreadyRegistered,
            extensionType.ToString(), workItemType.ToString()));
    }
}

public void RegisterRootExtension(Type extensionType)
{
    Guard.ArgumentNotNull(extensionType, "extensionType");
    Guard.TypeIsAssignableFromType(extensionType, typeof(IWorkItemExtension),
        "extensionType");

    rootExtensions.Add(extensionType);

    if (traceSource != null)
        traceSource.TraceInformation(String.Format(
            CultureInfo.CurrentCulture,
            Properties.Resources.WorkItemExtensionRootRegistered,
            extensionType));
}
//初始化WorkItem。
public void InitializeExtensions(WorkItem workItem)
{
    Guard.ArgumentNotNull(workItem, "workItem");
    InnerInitialize(workItem, GetRegisteredExtensions(workItem.GetType()));
```

```
        if (workItem.Parent == null)
            InnerInitialize(workItem, rootExtensions);
    }
    //实际执行。将WorkItem的事件实现与扩展关联。
    private void InnerInitialize(WorkItem workItem, IEnumerable<Type> extensions)
    {
        foreach (Type extensionType in extensions)
        {
            IWorkItemExtension extension =
                (IWorkItemExtension)workItem.Items.AddNew(extensionType);
            extension.Initialize(workItem);

            if (traceSource != null)
                traceSource.TraceInformation(String.Format(
                    CultureInfo.CurrentCulture,
                    Properties.Resources.WorkItemExtensionInitialized,
                    extension, workItem));
        }
    }
    //获取扩展。
    private IEnumerable<Type> GetRegisteredExtensions(Type workItemType)
    {
        return extensions.FindAllValuesByKey(
            delegate(Type key)
            {
                return key.IsAssignableFrom(workItemType);
            });
    }

    private class ClassNameTraceSourceAttribute : TraceSourceAttribute
```

```
        {
            public ClassNameTraceSourceAttribute()
                : base(typeof(WorkItemExtensionService).FullName)
            {
            }
        }
    }
}
```

8.20 IWorkItemTypeCatalogService

WorkItem 类型目录服务。

```
using System;
using System.Collections.Generic;
namespace Microsoft.Practices.CompositeUI.Services
{
    public interface IWorkItemTypeCatalogService
    {
        //注册的WorkItem类型。
        ICollection<Type> RegisteredWorkItemTypes { get; }
        //创建指定类型的WorkItems，并对创建的WOrkItem执行操作。
        void CreateEachWorkItem<TWorkItem>(WorkItem parentWorkItem, Action<TWorkItem> action);
        //创建指定类型的WorkItems，并对创建的WOrkItem执行操作。
        void CreateEachWorkItem(Type workItemType, WorkItem parentWorkItem,
            Action<WorkItem> action);
        //注册WorkItem。
        void RegisterWorkItem<TWorkItem>()
            where TWorkItem : WorkItem;
        //注册WorkItem。
        void RegisterWorkItem(Type workItemType);
```

```
    }
}
```

8.21 WorkItemTypeCatalogService

```
using System;
using Microsoft.Practices.CompositeUI.Utility;
using System.Collections.Generic;
namespace Microsoft.Practices.CompositeUI.Services
{
    public class WorkItemTypeCatalogService : IWorkItemTypeCatalogService
    {
        //注册的WorkItem类型集合。
        private Set<Type> registeredTypes = new Set<Type>();
        public ICollection<Type> RegisteredWorkItemTypes
        {
            get { return registeredTypes.AsReadOnly(); }
        }
        //创建指定类型的WorkItem。
        public void CreateEachWorkItem<TWorkItem>(WorkItem parentWorkItem, Action<TWorkItem>
action)
        {
            Guard.ArgumentNotNull(parentWorkItem, "parentWorkItem");
            Guard.ArgumentNotNull(action, "action");

            foreach (Type type in registeredTypes)
                if (typeof(TWorkItem).IsAssignableFrom(type))
                    action((TWorkItem)parentWorkItem.Items.AddNew(type));
        }
        //创建并调用action。
```

```
        public void CreateEachWorkItem(Type workItemType, WorkItem parentWorkItem,
            Action<WorkItem> action)
        {
            Guard.ArgumentNotNull(parentWorkItem, "parentWorkItem");
            Guard.ArgumentNotNull(action, "action");
            foreach (Type type in registeredTypes)
                if (workItemType.IsAssignableFrom(type))
                    action((WorkItem)parentWorkItem.Items.AddNew(type));
        }
        //注册WorkItem。
        public void RegisterWorkItem<TWorkItem>()
            where TWorkItem : WorkItem
        {
            RegisterWorkItem(typeof(TWorkItem));
        }
        public void RegisterWorkItem(Type workItemType)
        {
            Guard.TypeIsAssignableFromType(workItemType, typeof(WorkItem), "workItemType");
            registeredTypes.Add(workItemType);
        }
    }
}
```

8.22 ServiceMissingException

```
using System;
using System.Globalization;
using System.Runtime.Serialization;
using Microsoft.Practices.CompositeUI.Properties;
```

```
namespace Microsoft.Practices.CompositeUI.Services
{
    [Serializable]
    public class ServiceMissingException : Exception
    {
        public ServiceMissingException() : base()
        {
        }
        public ServiceMissingException(string message)
            : base(message)
        {
        }
        public ServiceMissingException(string message, Exception exception)
            : base(message, exception)
        {
        }
        public ServiceMissingException(Type serviceType)
            : base(String.Format(CultureInfo.CurrentCulture,
                                 Resources.ServiceMissingExceptionSimpleMessage,
                                 serviceType))
        {
        }
        public ServiceMissingException(Type serviceType, object component)
            : base(String.Format(CultureInfo.CurrentCulture,
                                 Resources.ServiceMissingExceptionMessage,
                                 serviceType, component))
        {
        }
        public ServiceMissingException(Type serviceType, object component, Exception innerException)
            : base(String.Format(CultureInfo.CurrentCulture,
```

```
                                        Resources.ServiceMissingExceptionMessage,
                                        serviceType, component), innerException)
            {
            }
            protected ServiceMissingException(SerializationInfo info, StreamingContext context)
                : base(info, context)
            {
            }
        }
}
```

9 仪表（Instrumentation）

9.1 TraceSourceAttribute

指定一个属性或参数将注入一个 TraceSource。

```
using System;
using System.Diagnostics;
using Microsoft.Practices.ObjectBuilder;

namespace Microsoft.Practices.CompositeUI
{
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Parameter, AllowMultiple = false,
        Inherited = true)]
    public class TraceSourceAttribute : ParameterAttribute
    {

        private string sourceName;
        //跟踪信息源名称。
        public string SourceName
```

```
{
        get { return sourceName; }
}
//信息源名称。
public TraceSourceAttribute(string sourceName)
{
        this.sourceName = sourceName;
}
//直接创建一个TraceSourceParamter。
public override IParameter CreateParameter(Type memberType)
{
        return new TraceSourceParameter(memberType, sourceName);
}
private class TraceSourceParameter : KnownTypeParameter
{
        private string sourceName;
        public TraceSourceParameter(Type memberType, string sourceName)
                : base(memberType)
        {
                this.sourceName = sourceName;
        }

        public override object GetValue(IBuilderContext context)
        {
                //获取TraceSource目录。
                DependencyResolutionLocatorKey key =
                        new DependencyResolutionLocatorKey(typeof(ITraceSourceCatalogService), null);
                ITraceSourceCatalogService svc = (ITraceSourceCatalogService)context.Locator.Get(key);
                //如果目录服务存在，则获取其TraceSource。
                return svc != null ? svc.GetTraceSource(sourceName) : null;
```

```
                }
            }
        }
}
```

9.2 ITraceSourceCatalogService

TraceSource 目录。由 TraceSource 名称和 TraceSource 序列组成。

```
using System;
using System.Diagnostics;
using System.Security.Permissions;
using Microsoft.Practices.CompositeUI.Utility;

namespace Microsoft.Practices.CompositeUI
{
    public interface ITraceSourceCatalogService
    {
        //添加一个TraceSource的事件。
        event EventHandler<DataEventArgs<TraceSource>> TraceSourceAdded;
        //获取指定名称的TraceSouce。
        TraceSource GetTraceSource(string sourceName);
        //所有TraceSources。
        ReadOnlyDictionary<string, TraceSource> TraceSources { get; }
    }
}
```

9.3 TraceSourceCatalogService

管理 TraceSource 的服务。

```
using System;
using System.Collections.Generic;
```

```
using System.Diagnostics;
using Microsoft.Practices.CompositeUI.Utility;
using System.Security.Permissions;
namespace Microsoft.Practices.CompositeUI
{
    public class TraceSourceCatalogService : ITraceSourceCatalogService
    {
        private Dictionary<string, TraceSource> sources = new Dictionary<string, TraceSource>();
        public TraceSourceCatalogService()
        {
        }
        public event EventHandler<DataEventArgs<TraceSource>> TraceSourceAdded;
        //获取TraceSource。
        [SecurityPermission(SecurityAction.Demand, Flags = SecurityPermissionFlag.UnmanagedCode)]
        public TraceSource GetTraceSource(string sourceName)
        {
            Guard.ArgumentNotNull(sourceName, "sourceName");
            if (sources.ContainsKey(sourceName))
                return sources[sourceName];
            //如果不存在，则新建。
            TraceSource source = new TraceSource(sourceName);
            sources.Add(source.Name, source);
            if (source.Listeners.Count == 1 && source.Listeners[0] is DefaultTraceListener)
                source.Listeners.Clear();
            source.Listeners.AddRange(Trace.Listeners);
            //触发事件。
            if (TraceSourceAdded != null)
                TraceSourceAdded.Invoke(this, new DataEventArgs<TraceSource>(source));
            return source;
        }
```

```
        //为所有的TraceSource添加TraceListener。
        public void AddSharedListener(TraceListener listener)
        {
            Guard.ArgumentNotNull(listener, "listener");

            AddSharedListener(listener, string.Empty);
        }
        [SecurityPermission(SecurityAction.Demand, Flags = SecurityPermissionFlag.UnmanagedCode)]
        public void AddSharedListener(TraceListener listener, string name)
        {
            Guard.ArgumentNotNull(listener, "listener");
            Guard.ArgumentNotNull(name, "name");
            if (name.Length != 0)
            {
                listener.Name = name;
            }
            foreach (KeyValuePair<string, TraceSource> pair in sources)
            {
                pair.Value.Listeners.Add(listener);
            }
        }
        //TraceSource集合。
        public ReadOnlyDictionary<string, TraceSource> TraceSources
        {
            get
            {
                return new ReadOnlyDictionary<string, TraceSource>(sources);
            }
        }
}
```

```
}
```

10 可视化器

10.1 IVisualizer

可视化器接口。

```
using System;

using Microsoft.Practices.ObjectBuilder;


namespace Microsoft.Practices.CompositeUI

{

    public interface IVisualizer : IDisposable

    {

        //构建器。

        Builder CabBuilder { get; }

        //根WorkItem。

        WorkItem CabRootWorkItem { get; }

        //初始化可视化器。

        void Initialize(WorkItem cabRootWorkItem, Builder cabBuilder);

    }

}
```

10.2 CabVisualizer

```
using System;

using Microsoft.Practices.CompositeUI.BuilderStrategies;

using Microsoft.Practices.CompositeUI.Commands;

using Microsoft.Practices.CompositeUI.EventBroker;

using Microsoft.Practices.CompositeUI.Services;
```

```
using Microsoft.Practices.CompositeUI.UIElements;
using Microsoft.Practices.ObjectBuilder;
using Microsoft.Practices.CompositeUI.Configuration;
using System.Configuration;
using System.Globalization;
using System.Collections.Generic;

namespace Microsoft.Practices.CompositeUI
{
    public class CabVisualizer : IVisualizer
    {
        //CAB应用系统的Builder。
        Builder cabBuilder;
        //CAB应用系统的根WorkItem。
        WorkItem cabRootWorkItem;
        //可视化器根WorkItem。
        WorkItem rootWorkItem;
        //可视化器配置。
        VisualizationElementCollection configuration;
        //可视化对象。
        List<object> visualizations = new List<object>();
        //可视化对象集合。
        protected ICollection<object> Visualizations
        {
            get
            {
                return visualizations.AsReadOnly();
            }
        }
        //配置。
```

```
protected virtual VisualizationElementCollection Configuration
{
    get
    {
        if (configuration == null)
            configuration = LoadConfiguration();

        return configuration;
    }
}
//装载配置文件。
private VisualizationElementCollection LoadConfiguration()
{
    object section = ConfigurationManager.GetSection(SettingsSection.SectionName);
    SettingsSection configSection = section as SettingsSection;

    if (section != null && configSection == null)
    {
        throw new ConfigurationErrorsException(String.Format(
            CultureInfo.CurrentCulture,
            Properties.Resources.InvalidConfigurationSectionType,
            SettingsSection.SectionName, typeof(SettingsSection)));
    }

    if (configSection == null)
        configSection = new SettingsSection();

    return configSection.Visualizer;
}
//CAB应用Builder。
```

```
public Builder CabBuilder
{
    get { return cabBuilder; }
}
//CAB根WorkItem。
public WorkItem CabRootWorkItem
{
    get { return cabRootWorkItem; }
}
//可视化器根WorkItem。
protected WorkItem RootWorkItem
{
    get { return rootWorkItem; }
}
//初始化可视化器。
public void Initialize(WorkItem cabRootWorkItem, Builder cabBuilder)
{
    if (this.cabRootWorkItem != null)
        throw new InvalidOperationException(Properties.Resources.VisualizerAlreadyInitialized);

    this.cabRootWorkItem = cabRootWorkItem;
    this.cabBuilder = cabBuilder;
    //创建可视化器的Builder。
    Builder builder = CreateBuilder();
    //添加可视化策略。
    AddBuilderStrategies(builder);
    //创建可视化根WorkItem。
    CreateRootWorkItem(builder);
    //添加必须服务。
    AddRequiredServices();
```

```
        //添加服务。
        AddServices();
        //构建根WorkItem。
        rootWorkItem.BuildUp();
        //创建Shell。
        CreateVisualizationShell();
        //装载插件。
        LoadVisualizerPlugins();
        //运行根WorkItem。
        rootWorkItem.Run();
    }
    //由子类重写。
    protected virtual void CreateVisualizationShell()
    {
    }
    //子类重写。
    protected virtual void AddBuilderStrategies(Builder builder)
    {
    }
    //子类重写。
    protected virtual void AddServices()
    {
    }
    //创建Builder。
    private Builder CreateBuilder()
    {
        Builder builder = new Builder();

        builder.Strategies.AddNew<EventBrokerStrategy>(BuilderStage.Initialization);
        builder.Strategies.AddNew<CommandStrategy>(BuilderStage.Initialization);
```

```
        builder.Policies.SetDefault<ISingletonPolicy>(new SingletonPolicy(true));


        return builder;
}
//创建根WorkItem。
private void CreateRootWorkItem(Builder builder)
{
        rootWorkItem = new WorkItem();
        RootWorkItem.InitializeRootWorkItem(builder);
}
//添加必须服务。
private void AddRequiredServices()
{
        RootWorkItem.Services.AddNew<TraceSourceCatalogService, ITraceSourceCatalogService>();
        RootWorkItem.Services.AddNew<
                WorkItemTypeCatalogService, IWorkItemTypeCatalogService>();
        RootWorkItem.Services.AddNew<
                SimpleWorkItemActivationService, IWorkItemActivationService>();
        RootWorkItem.Services.AddNew<
                CommandAdapterMapService, ICommandAdapterMapService>();
        RootWorkItem.Services.AddNew<
                UIElementAdapterFactoryCatalog, IUIElementAdapterFactoryCatalog>();
        RootWorkItem.Services.Add<IVisualizer>(this);
}
//状态可视化插件。
private void LoadVisualizerPlugins()
{
        foreach (VisualizationElement elt in Configuration)
                AddNewVisualization(elt.Type);
}
```

```
        //回收。
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        //清理。
        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                rootWorkItem.Services.Remove<IVisualizer>();
                rootWorkItem.Dispose();
            }
        }
        //添加可视化器。
        protected TVisualization AddNewVisualization<TVisualization>()
        {
            return (TVisualization)AddNewVisualization(typeof(TVisualization));
        }
        protected object AddNewVisualization(Type type)
        {
            object result = RootWorkItem.Items.AddNew(type);
            visualizations.Add(result);
            return result;
        }
    }
}
```

11 配置（Configuration）

11.1 SettingsSection

读取 CompositeUI 配置节。

```
using System.Configuration;

namespace Microsoft.Practices.CompositeUI.Configuration
{
    public class SettingsSection : ConfigurationSection
    {
        public const string SectionName = "CompositeUI";
        //读取services配置集合。
        [ConfigurationProperty("services", IsRequired = true)]
        public ServiceElementCollection Services
        {
            get { return (ServiceElementCollection) this["services"]; }
        }
        //读取visualizer配置集合。
        [ConfigurationProperty("visualizer", IsRequired = false)]
        public VisualizationElementCollection Visualizer
        {
            get { return (VisualizationElementCollection) this["visualizer"]; }
        }
    }
}
```

11.2 ServiceElementCollection

```
using System;
using System.Collections.Generic;
```

```
using System.Configuration;


namespace Microsoft.Practices.CompositeUI.Configuration
{
    [ConfigurationCollection(typeof (ServiceElement))]
    public class ServiceElementCollection : ConfigurationElementCollection, IEnumerable<ServiceElement>
    {
        //重写
        protected override ConfigurationElement CreateNewElement()
        {
            return new ServiceElement();
        }
        //获取Key。
        protected override object GetElementKey(ConfigurationElement element)
        {
            return ((ServiceElement) element).ServiceType;
        }
        //获取ServiceElement。
        public ServiceElement this[Type serviceType]
        {
            get { return (ServiceElement) base.BaseGet(serviceType); }
            set
            {
                if (base.BaseGet(serviceType) != null)
                {
                    base.BaseRemove(serviceType);
                }
                base.BaseAdd(value);
            }
        }
```

```
        public void Add(ServiceElement service)
        {
            base.BaseAdd(service);
        }
        public void Remove(Type serviceType)
        {
            base.BaseRemove(serviceType);
        }

        #region IEnumerable<ServiceElement> Members
        public new IEnumerator<ServiceElement> GetEnumerator()
        {
            int count = base.Count;
            for (int i = 0; i < count; i++)
            {
                yield return (ServiceElement) base.BaseGet(i) ;
            }
        }
        #endregion
    }
}
```

11.3 ServiceElement

Service 节点。

```
using System;
using System.ComponentModel;
using System.Configuration;

namespace Microsoft.Practices.CompositeUI.Configuration
```

```
{
    public class ServiceElement : ParametersElement
    {
        //服务类型。
        [ConfigurationProperty("serviceType", IsKey = true, IsRequired = true)]
        [TypeConverter(typeof (TypeNameConverter))]
        public Type ServiceType
        {
            get { return (Type) this["serviceType"]; }
            set { this["serviceType"] = value; }
        }
        //实例类型。
        [ConfigurationProperty("instanceType", IsRequired = true)]
        [TypeConverter(typeof (TypeNameConverter))]
        public Type InstanceType
        {
            get { return (Type) this["instanceType"]; }
            set { this["instanceType"] = value; }
        }
    }
}
```

11.4 VisualizationElementCollection

```
using System;
using System.ComponentModel;
using System.Configuration;
using System.Collections.Generic;

namespace Microsoft.Practices.CompositeUI.Configuration
```

```
{
    [ConfigurationCollection(typeof(VisualizationElement))]
    public class VisualizationElementCollection :
        ConfigurationElementCollection, IEnumerable<VisualizationElement>
    {
        protected override ConfigurationElement CreateNewElement()
        {
            return new VisualizationElement();
        }
        protected override object GetElementKey(ConfigurationElement element)
        {
            return ((VisualizationElement)element).Type;
        }
        public new IEnumerator<VisualizationElement> GetEnumerator()
        {
            int count = base.Count;
            for (int i = 0; i < count; i++)
            {
                yield return (VisualizationElement)base.BaseGet(i);
            }
        }
        public void Add(VisualizationElement element)
        {
            base.BaseAdd(element);
        }
    }
}
```

11.5 VisualizationElement

可视化器类型。

```
using System;

using System.Collections.Generic;

using System.Text;

using System.Configuration;

using System.ComponentModel;


namespace Microsoft.Practices.CompositeUI.Configuration

{

    public class VisualizationElement : ConfigurationElement

    {

        [ConfigurationProperty("type", IsKey = true, IsRequired = true)]

        [TypeConverter(typeof(TypeNameConverter))]

        public Type Type

        {

            get { return (Type)this["type"]; }

            set { this["type"] = value; }

        }

    }

}
```

11.6 ConfigurationAttribute

配置特性。

```
using System;

using System.Collections.Generic;

using System.Configuration;

using System.Globalization;
```

```
using Microsoft.Practices.ObjectBuilder;

namespace Microsoft.Practices.CompositeUI.Configuration
{
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Parameter, AllowMultiple = false,
        Inherited = true)]
    public sealed class ConfigurationAttribute : ParameterAttribute
    {
        //对应配置节的主键。
        private string settingsKey;
        public ConfigurationAttribute()
        {
        }
        public ConfigurationAttribute(string settingsKey)
        {
            this.settingsKey = settingsKey;
        }
        //获取参数。
        //memberType：配置节参数。
        public override IParameter CreateParameter(Type memberType)
        {
            if (!typeof(ApplicationSettingsBase).IsAssignableFrom(memberType))
                throw new ArgumentException(String.Format(
                        CultureInfo.CurrentCulture,
                        Properties.Resources.ConfigurationAttributeOnValueWithIncompatibleType,
                        memberType));

            return new ConfigurationParameter(memberType, settingsKey);
        }
        //配置参数。
        private class ConfigurationParameter : KnownTypeParameter
```

```
{
    static Dictionary<string, ApplicationSettingsBase> cache =
        new Dictionary<string, ApplicationSettingsBase>();
    string settingsKey;

    public ConfigurationParameter(Type type, string settingsKey)
        : base(type)
    {
        this.settingsKey = settingsKey;
    }

    public override object GetValue(IBuilderContext context)
    {
        lock (cache)
        {
            string cacheKey = type.ToString();
            if (settingsKey != null)
                cacheKey += "-" + settingsKey;
            ApplicationSettingsBase settings = cache[cacheKey];

            if (settings == null)
            {
                //创建一个配置节。
                settings = (ApplicationSettingsBase)context.HeadOfChain.BuildUp(
                    context, type, null, null);
                if (settingsKey != null)
                    settings.SettingsKey = settingsKey;
                cache.Add(cacheKey, settings);
            }
```

```
                    return settings;
                }
            }
        }
    }
}
```

11.7 ConfigurationException

```
using System;
using System.Runtime.Serialization;
namespace Microsoft.Practices.CompositeUI.Configuration
{
    [Serializable]
    public class ConfigurationException : Exception
    {
        public ConfigurationException() : base() { }
        public ConfigurationException(string message) : base(message) { }
        public ConfigurationException(string message, Exception innerException) :
            base(message, innerException) { }
        protected ConfigurationException(SerializationInfo info, StreamingContext context) :
            base(info, context) { }

    }
}
```

11.8 IModuleInfo

Module 信息，包括程序集文件、更新 Location 和使用角色集合。

```
using System.Collections.Generic;
namespace Microsoft.Practices.CompositeUI.Configuration
```

```
{
    public interface IModuleInfo
    {
        string AssemblyFile { get; }
        string UpdateLocation { get; }
        IList<string> AllowedRoles { get; }
    }
}
```

11.9 ModuleInfo

```
using System;
using System.Collections.Generic;
using System.Reflection;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.Configuration
{
    public class ModuleInfo : IModuleInfo
    {
        private string assemblyFile;
        private string updateLocation;
        private List<string> allowedRoles = new List<string>();
        public ModuleInfo()
        {
        }
        public ModuleInfo(string assemblyFile)
        {
            Guard.ArgumentNotNullOrEmptyString(assemblyFile, "assemblyFile");

            this.assemblyFile = assemblyFile;
```

```
}
public ModuleInfo(Assembly assembly)
{
    Guard.ArgumentNotNull(assembly, "assembly");
    this.assemblyFile = assembly.CodeBase.Replace("file:///", "").Replace('/', '\\');
}
public string AssemblyFile
{
    get { return assemblyFile; }
}
public void SetAssemblyFile(string assemblyFile)
{
    this.assemblyFile = assemblyFile;
}
public string UpdateLocation
{
    get { return updateLocation; }
}
public void SetUpdateLocation(string updateLocation)
{
    this.updateLocation = updateLocation;
}
public IList<string> AllowedRoles
{
    get { return allowedRoles.AsReadOnly(); }
}
public void AddRoles(params string[] roles)
{
    Guard.ArgumentNotNull(roles, "roles");
```

```
            foreach (string role in roles)
                if (role != null && role.Length > 0)
                    allowedRoles.Add(role);
        }
        public void ClearRoles()
        {
            allowedRoles.Clear();
        }
    }
}
```

11.10 ParametersElement

支持通过 Configuration 接收任意 Name-Value 对的配置元素的基类。

```
using System.Collections.Specialized;
using System.Configuration;
namespace Microsoft.Practices.CompositeUI.Configuration
{
    public abstract class ParametersElement : ConfigurationElement
    {
        //存储不认识的配置。
        private ConfigurationPropertyCollection dynamicProperties = new ConfigurationPropertyCollection();
        private NameValueCollection configAttributes = new NameValueCollection();
        protected ParametersElement()
        {
        }
        //配置属性集合。
        protected override ConfigurationPropertyCollection Properties
        {
            get
```

```
            {
                ConfigurationPropertyCollection baseprops = base.Properties;
                foreach (ConfigurationProperty dynprop in dynamicProperties)
                {
                    baseprops.Add(dynprop);
                }
                return baseprops;
            }
        }
        //参数集合。
        public NameValueCollection Parameters
        {
            get { return configAttributes; }
        }
        //将不认输的配置添加到dynamicProperties中。
        protected override bool OnDeserializeUnrecognizedAttribute(string name, string value)
        {
            ConfigurationProperty dynprop = new ConfigurationProperty(name, typeof (string), value);
            dynamicProperties.Add(dynprop);
            this[dynprop] = value;
            configAttributes.Add(name, value);
            return true;
        }
    }
}
```

11.11 SolutionProfileReaderException

```
using System;
using System.Runtime.Serialization;
```

```
namespace Microsoft.Practices.CompositeUI.Configuration
{
    [Serializable]
    public class SolutionProfileReaderException : Exception
    {
        public SolutionProfileReaderException() : base() { }
        public SolutionProfileReaderException(string message) : base(message) { }
        public SolutionProfileReaderException(string message, Exception innerException) :
            base(message, innerException) { }
        protected SolutionProfileReaderException(SerializationInfo info, StreamingContext context) :
            base(info, context) { }
    }
}
```

11.12 SolutionProfileReader

SolutionProfile 配置文件解析器。

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Reflection;
using System.Security.Principal;
using System.Threading;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;
using Microsoft.Practices.CompositeUI.Configuration.Xsd;
using Microsoft.Practices.CompositeUI.Properties;
using Microsoft.Practices.CompositeUI.Utility;
```

```
namespace Microsoft.Practices.CompositeUI.Configuration
{
    public class SolutionProfileReader
    {
        //配置文件名称。
        private string catalogFilePath = DefaultCatalogFile;
        //默认配置文件。
        public const string DefaultCatalogFile = "ProfileCatalog.xml";
        public SolutionProfileReader()
        {
        }
        public SolutionProfileReader(string catalogFilePath)
        {
            this.catalogFilePath = GetFullPathOrThrowIfInvalid(catalogFilePath);
        }
        //配置文件路径。
        public string CatalogFilePath
        {
            get { return catalogFilePath; }
        }
        //读取配置文件。
        public SolutionProfileElement ReadProfile()
        {
            SolutionProfileElement solution = new SolutionProfileElement();
            if (File.Exists(catalogFilePath))
            {
                try
                {
                    XmlSerializer serializer = new XmlSerializer(typeof(SolutionProfileElement));
                    using (XmlReader reader = GetValidatingReader())
```

```
                {
                    //使用XmlReader反序列化。
                    solution = (SolutionProfileElement)serializer.Deserialize(reader);
                    ProcessProfileRoles(solution);
                    return solution;
                }
            }
            catch (Exception ex)
            {
                throw new SolutionProfileReaderException(String.Format(
                    CultureInfo.CurrentCulture,
                    Resources.ErrorReadingProfile,
                    catalogFilePath), ex);
            }
        }
        else
        {
            ThrowIfCatalogPathNotDefault();
        }
        return solution;
    }
    //创建一个XmlReader。
    private XmlReader GetValidatingReader()
    {
        Stream stream = Assembly.GetExecutingAssembly().
            GetManifestResourceStream("Microsoft.Practices.CompositeUI.SolutionProfile.xsd");
        XmlTextReader schemaReader = new XmlTextReader(stream);
        stream.Dispose();

        XmlReaderSettings settings = new XmlReaderSettings();
```

```
        settings.Schemas.Add("http://schemas.microsoft.com/pag/cab-profile", schemaReader);
        XmlReader catalogReader = XmlReader.Create(catalogFilePath, settings);

        return catalogReader;
    }
    //处理SolutionProfile元素。
    private static void ProcessProfileRoles(SolutionProfileElement profile)
    {
        //获取当前标识。
        IPrincipal principal = Thread.CurrentPrincipal;
        //授权的模块。
        List<ModuleInfoElement> permittedModules = new List<ModuleInfoElement>();
        if (profile.Modules != null)
        {
            //遍历ModuleInfoElement元素。
            foreach (ModuleInfoElement modInfo in profile.Modules)
            {
                //需要授权。
                if (modInfo.Roles != null)
                {
                    //是否已经认证。
                    if (principal.Identity.IsAuthenticated)
                    {
                        //执行授权。
                        foreach (RoleElement role in modInfo.Roles)
                        {
                            if (principal.IsInRole(role.Allow))
                            {
                                permittedModules.Add(modInfo);
                                break;
```

```
                        }
                    }
                }
            }
            else
            {
                permittedModules.Add(modInfo);
            }
        }
        profile.Modules = permittedModules.ToArray();
    }
}

private void ThrowIfCatalogPathNotDefault()
{
    string defaultFullPath = Path.Combine(
        AppDomain.CurrentDomain.BaseDirectory, DefaultCatalogFile);
    if (String.Compare(catalogFilePath, DefaultCatalogFile, true, CultureInfo.CurrentCulture) == 0 ||
        String.Compare(catalogFilePath, defaultFullPath, true, CultureInfo.CurrentCulture) == 0)
    {
        return;
    }
    throw new SolutionProfileReaderException(String.Format(CultureInfo.CurrentCulture,
        Resources.SolutionProfileNotFound,
        catalogFilePath));
}
//判断指定文件是否正确。
private string GetFullPathOrThrowIfInvalid(string catalogFilePath)
{
    Guard.ArgumentNotNullOrEmptyString(catalogFilePath, "catalogFilePath");
```

```
            string fullPath = catalogFilePath;

            if (!Path.IsPathRooted(fullPath))
            {
                fullPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, fullPath);
            }
            fullPath = new FileInfo(fullPath).FullName;

            if (!fullPath.StartsWith(AppDomain.CurrentDomain.BaseDirectory, true,
                CultureInfo.CurrentCulture))
            {
                throw new SolutionProfileReaderException(String.Format(
                    CultureInfo.CurrentCulture,
                    Resources.InvalidSolutionProfilePath,
                    catalogFilePath,
                    AppDomain.CurrentDomain.BaseDirectory));
            }

            return fullPath;
        }
    }
}
```

11.13 SolutionProfileElement

由 Xsd 生成。

```
namespace Microsoft.Practices.CompositeUI.Configuration.Xsd {
    using System.Xml.Serialization;
```

```
        //SolutionProfileElement节对应的元素。
        [System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.26")]
        [System.SerializableAttribute()]
        [System.Diagnostics.DebuggerStepThroughAttribute()]
        [System.ComponentModel.DesignerCategoryAttribute("code")]
        [System.Xml.Serialization.XmlTypeAttribute(Namespace="http://schemas.microsoft.com/pag/cab-profile")]
        [System.Xml.Serialization.XmlRootAttribute("SolutionProfile",
Namespace="http://schemas.microsoft.com/pag/cab-profile", IsNullable=false)]
        public partial class SolutionProfileElement {

            private ModuleInfoElement[] modulesField;

            //模块元素集合。
            [System.Xml.Serialization.XmlArrayItemAttribute("ModuleInfo", IsNullable=false)]
            public ModuleInfoElement[] Modules {
                get {
                    return this.modulesField;
                }
                set {
                    this.modulesField = value;
                }
            }
        }

        //ModuleInfo元素。
        [System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.26")]
        [System.SerializableAttribute()]
        [System.Diagnostics.DebuggerStepThroughAttribute()]
        [System.ComponentModel.DesignerCategoryAttribute("code")]
        [System.Xml.Serialization.XmlTypeAttribute(Namespace="http://schemas.microsoft.com/pag/cab-profile")]
```

```
public partial class ModuleInfoElement {

    private RoleElement[] rolesField;

    private string assemblyFileField;

    private string updateLocationField;
      //使用角色。
    [System.Xml.Serialization.XmlArrayItemAttribute("Role", IsNullable=false)]
    public RoleElement[] Roles {
        get {
            return this.rolesField;
        }
        set {
            this.rolesField = value;
        }
    }
      //程序集文件。
    [System.Xml.Serialization.XmlAttributeAttribute()]
    public string AssemblyFile {
        get {
            return this.assemblyFileField;
        }
        set {
            this.assemblyFileField = value;
        }
    }
      //更新Location。
    [System.Xml.Serialization.XmlAttributeAttribute(DataType="anyURI")]
    public string UpdateLocation {
```

```
            get {
                return this.updateLocationField;
            }
            set {
                this.updateLocationField = value;
            }
        }
    }

    //角色Element。
    [System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.26")]
    [System.SerializableAttribute()]
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Xml.Serialization.XmlTypeAttribute(Namespace="http://schemas.microsoft.com/pag/cab-profile")]
    public partial class RoleElement {
        private string allowField;
        [System.Xml.Serialization.XmlAttributeAttribute()]
        public string Allow {
            get {
                return this.allowField;
            }
            set {
                this.allowField = value;
            }
        }
    }
}
```

11.14 SolutionProfileElement

```
namespace Microsoft.Practices.CompositeUI.Configuration.Xsd
{
    public partial class SolutionProfileElement
    {
        public SolutionProfileElement()
        {
            modulesField = new ModuleInfoElement[0];
        }
    }
}
```

12 辅助（Utility）

12.1 LockBase

只读、只写锁的基类。

```
using System;
using System.Threading;
namespace Microsoft.Practices.CompositeUI
{
    //请求加锁。
    internal delegate void AcquireIntTimeoutMethod(int millisecondsTimeout);
    internal delegate void AcquireTimeSpanTimeoutMethod(TimeSpan timeout);
    //释放锁。
    internal delegate void ReleaseMethod();

    public abstract class LockBase : IDisposable
    {
```

```
ReleaseMethod release;
bool timedOut = false;

internal LockBase(AcquireIntTimeoutMethod acquire, ReleaseMethod release, int timeout)
{
    this.release = release;

    try
    {
        acquire(timeout);
    }
    catch (ApplicationException)
    {
        timedOut = true;
    }
}

internal LockBase(AcquireTimeSpanTimeoutMethod acquire, ReleaseMethod release,
    TimeSpan timeout)
{
    this.release = release;

    try
    {
        acquire(timeout);
    }
    catch (ApplicationException)
    {
        timedOut = true;
    }
```

```
        }

        public bool TimedOut
        {
            get { return timedOut; }
        }
        //释放锁。
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        protected virtual void Dispose(bool disposing)
        {
            if (disposing && !timedOut)
            {
                release();
            }
        }
    }
}
```

12.2 ReaderLock

只读锁。

```
using System;
using System.Threading;
namespace Microsoft.Practices.CompositeUI
{
    //使用例子如下：
```

```
        //ReaderWriterLock mylock = new ReaderWriterLock();
        //using (new ReaderLock(rwLock))
        //{
        //    使用资源
        //}
        //
        public class ReaderLock : LockBase
        {
            //不指定Timeout。
            public ReaderLock(ReaderWriterLock rwLock)
                : base(new AcquireIntTimeoutMethod(rwLock.AcquireReaderLock),
                new ReleaseMethod(rwLock.ReleaseReaderLock), -1) { }
            //指定Timeout。
            public ReaderLock(ReaderWriterLock rwLock, int millisecondsTimeout)
                : base(new AcquireIntTimeoutMethod(rwLock.AcquireReaderLock),
                new ReleaseMethod(rwLock.ReleaseReaderLock), millisecondsTimeout) { }
            public ReaderLock(ReaderWriterLock rwLock, TimeSpan timeout)
                : base(new AcquireTimeSpanTimeoutMethod(rwLock.AcquireReaderLock),
                new ReleaseMethod(rwLock.ReleaseReaderLock), timeout) { }
        }
}
```

12.3 WriterLock

```
using System;
using System.Threading;
namespace Microsoft.Practices.CompositeUI
{
        public class WriterLock : LockBase
        {
```

```
        //不指定Timeout。
        public WriterLock(ReaderWriterLock rwLock)
            : base(new AcquireIntTimeoutMethod(rwLock.AcquireWriterLock),
            new ReleaseMethod(rwLock.ReleaseWriterLock), -1) { }
        //指定Timeout。
        public WriterLock(ReaderWriterLock rwLock, int millisecondsTimeout)
            : base(new AcquireIntTimeoutMethod(rwLock.AcquireWriterLock),
            new ReleaseMethod(rwLock.ReleaseWriterLock), millisecondsTimeout) { }
        public WriterLock(ReaderWriterLock rwLock, TimeSpan timeout)
            : base(new AcquireTimeSpanTimeoutMethod(rwLock.AcquireWriterLock),
            new ReleaseMethod(rwLock.ReleaseWriterLock), timeout) { }
    }
}
```

12.4 ListDictionary

Key 与 List 系列集合的字典。

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI.Utility;
namespace Microsoft.Practices.CompositeUI.Utility
{
    public class ListDictionary<TKey, TValue> : IDictionary<TKey, List<TValue>>
    {
        Dictionary<TKey, List<TValue>> innerValues = new Dictionary<TKey, List<TValue>>();

        #region Public Methods
        //添加一个Key，如果不存在对应的List对象，则新建一个。
        public void Add(TKey key)
```

```
{
        Guard.ArgumentNotNull(key, "key");

        CreateNewList(key);
}
//添加Value。
public void Add(TKey key, TValue value)
{
        Guard.ArgumentNotNull(key, "key");
        Guard.ArgumentNotNull(value, "value");

        if (innerValues.ContainsKey(key))
        {
                innerValues[key].Add(value);
        }
        else
        {
                List<TValue> values = CreateNewList(key);
                values.Add(value);
        }
}
//创建List。
private List<TValue> CreateNewList(TKey key)
{
        List<TValue> values = new List<TValue>();
        innerValues.Add(key, values);

        return values;
}
//删除所有对象。
```

```
public void Clear()
{
    innerValues.Clear();
}
//是否包含Value。
public bool ContainsValue(TValue value)
{
    foreach (KeyValuePair<TKey, List<TValue>> pair in innerValues)
    {
        if (pair.Value.Contains(value))
        {
            return true;
        }
    }
    return false;
}
//是否包含Key。
public bool ContainsKey(TKey key)
{
    Guard.ArgumentNotNull(key, "key");
    return innerValues.ContainsKey(key);
}
//对Key进行过滤。
public IEnumerable<TValue> FindAllValuesByKey(Predicate<TKey> keyFilter)
{
    foreach (KeyValuePair<TKey, List<TValue>> pair in this)
    {
        if (keyFilter(pair.Key))
        {
            foreach (TValue value in pair.Value)
```

```
                {
                    yield return value;
                }
            }
        }
    }
    //对Value进行过滤。
    public IEnumerable<TValue> FindAllValues(Predicate<TValue> valueFilter)
    {
        foreach (KeyValuePair<TKey, List<TValue>> pair in this)
        {
            foreach (TValue value in pair.Value)
            {
                if (valueFilter(value))
                {
                    yield return value;
                }
            }
        }
    }
    //删除对象。
    public bool Remove(TKey key)
    {
        Guard.ArgumentNotNull(key, "key");
        return innerValues.Remove(key);
    }
    //删除一个Value。
    public void Remove(TKey key, TValue value)
    {
        Guard.ArgumentNotNull(key, "key");
```

```
        Guard.ArgumentNotNull(value, "value");

        if (innerValues.ContainsKey(key))
        {
            innerValues[key].RemoveAll(delegate(TValue item)
            {
                return value.Equals(item);
            });
        }
    }
    //删除Value。
    public void Remove(TValue value)
    {
        foreach (KeyValuePair<TKey, List<TValue>> pair in innerValues)
        {
            Remove(pair.Key, value);
        }
    }

    #endregion

    #region Properties
    //所有的Value。
    public List<TValue> Values
    {
        get
        {
            List<TValue> values = new List<TValue>();
            foreach (IEnumerable<TValue> list in innerValues.Values)
            {
```

```
                values.AddRange(list);
            }

            return values;
        }
    }
    //所有Key。
    public ICollection<TKey> Keys
    {
        get { return innerValues.Keys; }
    }
    //所引器。
    public List<TValue> this[TKey key]
    {
        get
        {
            if (innerValues.ContainsKey(key) == false)
            {
                innerValues.Add(key, new List<TValue>());
            }
            return innerValues[key];
        }
        set { innerValues[key] = value; }
    }
    //字典数目。
    public int Count
    {
        get { return innerValues.Count; }
    }
```

```
#endregion

#region IDictionary<TKey,List<TValue>> Members
void IDictionary<TKey, List<TValue>>.Add(TKey key, List<TValue> value)
{
    Guard.ArgumentNotNull(key, "key");
    Guard.ArgumentNotNull(value, "value");
    innerValues.Add(key, value);
}
bool IDictionary<TKey, List<TValue>>.TryGetValue(TKey key, out List<TValue> value)
{
    value = this[key];
    return true;
}
ICollection<List<TValue>> IDictionary<TKey, List<TValue>>.Values
{
    get { return innerValues.Values; }
}

#endregion

#region ICollection<KeyValuePair<TKey,List<TValue>>> Members
void ICollection<KeyValuePair<TKey, List<TValue>>>.Add(KeyValuePair<TKey, List<TValue>> item)
{
    ((ICollection<KeyValuePair<TKey, List<TValue>>>)innerValues).Add(item);
}
bool ICollection<KeyValuePair<TKey, List<TValue>>>.Contains(KeyValuePair<TKey, List<TValue>> item)
{
    return ((ICollection<KeyValuePair<TKey, List<TValue>>>)innerValues).Contains(item);
}
```

```
        void ICollection<KeyValuePair<TKey, List<TValue>>>.CopyTo(
            KeyValuePair<TKey, List<TValue>>[] array, int arrayIndex)
        {
            ((ICollection<KeyValuePair<TKey, List<TValue>>>)innerValues).CopyTo(array, arrayIndex);
        }
        bool ICollection<KeyValuePair<TKey, List<TValue>>>.IsReadOnly
        {
            get { return ((ICollection<KeyValuePair<TKey, List<TValue>>>)innerValues).IsReadOnly; }
        }
        bool ICollection<KeyValuePair<TKey, List<TValue>>>.Remove(KeyValuePair<TKey, List<TValue>> item)
        {
            return ((ICollection<KeyValuePair<TKey, List<TValue>>>)innerValues).Remove(item);
        }
        #endregion

        #region IEnumerable<KeyValuePair<TKey,List<TValue>>> Members
        IEnumerator<KeyValuePair<TKey, List<TValue>>> IEnumerable<
            KeyValuePair<TKey, List<TValue>>>.GetEnumerator()
        {
            return innerValues.GetEnumerator();
        }
        #endregion

        #region IEnumerable Members
        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return innerValues.GetEnumerator();
        }
        #endregion
    }
```

```
}
```

12.5 ReadOnlyDictionary

只读 Dictionary。对 IDictionary 实现的包装。

```
using System;

using System.Collections.Generic;

using System.Text;

using System.Collections;

namespace Microsoft.Practices.CompositeUI.Utility

{

    public class ReadOnlyDictionary<TKey, TValue> : IDictionary<TKey, TValue>

    {

        IDictionary<TKey, TValue> inner;

        //指定管理的Dictionary。

        public ReadOnlyDictionary(IDictionary<TKey, TValue> innerDictionary)

        {

            Guard.ArgumentNotNull(innerDictionary, "innerDictionary");


            inner = innerDictionary;

        }


        #region IDictionary<TKey,TValue> Members

        //不支持写操作。

        void IDictionary<TKey,TValue>.Add(TKey key, TValue value)

        {

            throw new NotSupportedException( Properties.Resources.DictionaryIsReadOnly );

        }

        public bool ContainsKey(TKey key)

        {
```

```
        return inner.ContainsKey(key);
}
public ICollection<TKey> Keys
{
        get { return inner.Keys; }
}
//不支持删除。
bool IDictionary<TKey, TValue>.Remove(TKey key)
{
        throw new NotSupportedException(Properties.Resources.DictionaryIsReadOnly);
}
public bool TryGetValue(TKey key, out TValue value)
{
        return inner.TryGetValue(key, out value);
}
public ICollection<TValue> Values
{
        get { return inner.Values; }
}
public TValue this[TKey key]
{
        get { return inner[key]; }
        set
        {
                throw new NotSupportedException(Properties.Resources.DictionaryIsReadOnly);
        }
}

#endregion
```

```
#region ICollection<KeyValuePair<TKey,TValue>> Members
//不支持。
void ICollection<KeyValuePair<TKey, TValue>>.Add(KeyValuePair<TKey, TValue> item)
{
    throw new NotSupportedException(Properties.Resources.DictionaryIsReadOnly);
}
//不支持。
void ICollection<KeyValuePair<TKey, TValue>>.Clear()
{
    throw new NotSupportedException(Properties.Resources.DictionaryIsReadOnly);
}
bool ICollection<KeyValuePair<TKey, TValue>>.Contains(KeyValuePair<TKey, TValue> item)
{
    return ((IDictionary<TKey, TValue>)inner).Contains(item);
}
void ICollection<KeyValuePair<TKey, TValue>>.CopyTo(
    KeyValuePair<TKey, TValue>[] array, int arrayIndex)
{
    ((IDictionary<TKey, TValue>)inner).CopyTo(array, arrayIndex);
}
public int Count
{
    get { return inner.Count; }
}
//返回True。
bool ICollection<KeyValuePair<TKey, TValue>>.IsReadOnly
{
    get { return true; }
}
//不支持。
```

```
        bool ICollection<KeyValuePair<TKey, TValue>>.Remove(KeyValuePair<TKey, TValue> item)
        {
            throw new NotSupportedException(Properties.Resources.DictionaryIsReadOnly);
        }

        #endregion

        #region IEnumerable<KeyValuePair<TKey,TValue>> Members
        IEnumerator<KeyValuePair<TKey, TValue>> IEnumerable<
            KeyValuePair<TKey, TValue>>.GetEnumerator()
        {
            return inner.GetEnumerator();
        }
        #endregion

        #region IEnumerable Members
        IEnumerator IEnumerable.GetEnumerator()
        {
            return ((IEnumerable)inner).GetEnumerator();
        }
        #endregion
    }
}
```

12.6 Set

集合，不允许有一个类的多个实例。。

```
using System;
using System.Collections;
using System.Collections.Generic;
```

```
namespace Microsoft.Practices.CompositeUI.Utility
{
    public class Set<TItem> : ICollection<TItem>
    {
        bool isReadOnly;
        //对象类型和实例列表。对一个类仅保存一个实例。
        Dictionary<TItem, object> dictionary;
        public Set()
        {
            dictionary = new Dictionary<TItem, object>();
            isReadOnly = false;
        }
        private Set(Set<TItem> innerSet)
        {
            dictionary = innerSet.dictionary;
            isReadOnly = true;
        }
        //如果只读，则抛出异常。
        public void Add(TItem item)
        {
            if (IsReadOnly)
                throw new NotSupportedException();
            dictionary[item] = String.Empty;
        }
        public Set<TItem> AsReadOnly()
        {
            return new Set<TItem>(this);
        }
        //清楚所有对象。
        public void Clear()
```

```
{
        if (IsReadOnly)
                throw new NotSupportedException();

        dictionary.Clear();
}
//是否当前类型的实例。
public bool Contains(TItem item)
{
        return dictionary.ContainsKey(item);
}
//不支持拷贝。
public void CopyTo(TItem[] array, int arrayIndex)
{
        throw new NotImplementedException();
}
public int Count
{
        get { return dictionary.Count; }
}
public bool IsReadOnly
{
        get { return isReadOnly; }
}
//删除。
public bool Remove(TItem item)
{
        if (IsReadOnly)
                throw new NotSupportedException();
        return dictionary.Remove(item);
```

```
        }
        public IEnumerator<TItem> GetEnumerator()
        {
            return dictionary.Keys.GetEnumerator();
        }
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }
}
```

12.7 DataEventArgs

数据事件参数。

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Microsoft.Practices.CompositeUI.Utility
{
    //TData：数据类型。
    public class DataEventArgs<TData> : EventArgs
    {
        TData data;
        public DataEventArgs(TData data)
        {
            if (data == null)
            {
                throw new ArgumentNullException("data");
            }
```

```
            this.data = data;
        }
        public TData Data
        {
            get { return data; }
        }
        public override string ToString()
        {
            return data.ToString();
        }
    }
}
```

12.8 Guard

验证类。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Globalization;
namespace Microsoft.Practices.CompositeUI.Utility
{
    public static class Guard
    {
        //参数不为空或空字符串。
        public static void ArgumentNotNullOrEmptyString(string argumentValue, string argumentName)
        {
            ArgumentNotNull(argumentValue, argumentName);
            if (argumentValue.Length == 0)
                throw new ArgumentException(
```

```
                    string.Format(CultureInfo.CurrentCulture,
                    Properties.Resources.StringCannotBeEmpty, argumentName));
            }
            //参数不为空。
            public static void ArgumentNotNull(object argumentValue, string argumentName)
            {
                if (argumentValue == null)
                    throw new ArgumentNullException(argumentName);
            }
            //检查参数为某类型枚举。
            public static void EnumValueIsDefined(Type enumType, object value, string argumentName)
            {
                if (Enum.IsDefined(enumType, value) == false)
                    throw new ArgumentException(String.Format(CultureInfo.CurrentCulture,
                        Properties.Resources.InvalidEnumValue,
                        argumentName, enumType.ToString()));
            }
            //判断providedType是否为assignee的基类。
            public static void TypeIsAssignableFromType(Type assignee, Type providedType,
                string argumentName)
            {
                if (!providedType.IsAssignableFrom(assignee))
                    throw new ArgumentException(string.Format(CultureInfo.CurrentCulture,
                        Properties.Resources.TypeNotCompatible, assignee, providedType), argumentName);
            }
        }
    }
```