安全主体的提供可以通过自定AuthorizationPolicy或者ServiceAuthorizationManager来实现。

# **一、AuthorizationPolicy**

在WCF安全应用编程接口中，所有的AuthorizationPolicy实现了[IAuthorizationPolicy](http://msdn2.microsoft.com/ms583255.aspx" \o "IAuthorizationPolicy Interface)接口。如下面的代码所示，IAuthorizationPolicy继承自[IAuthorizationComponent](http://msdn2.microsoft.com/ms583250.aspx" \o "IAuthorizationComponent Interface)接口，本身具有一个ClaimSet类型的Issuer属性和一个Evaluate方法。关于ClaimSet，我们会在后续的部分继续介绍，这里我们只需要关注Evaluate方法。该方法的第一个参数的类型为System.IdentityModel.Policy.EvaluationContext，它具有一个字典类型的只读属性Properties。

1: public interface IAuthorizationPolicy : IAuthorizationComponent

2: {

3: bool Evaluate(EvaluationContext evaluationContext, ref object state);

4: ClaimSet Issuer { get; }

5: }

6: public abstract class EvaluationContext

7: {

8: //其他成员

9: public abstract IDictionary<string, object> Properties { get; }

10: }

如果我们需要通过自定义的方式来提供安全主体，我们只需要通过实现IAuthorizationPolicy接口创建自定义的AuthorizationPolicy，并在Evaluate方法中将创建安全主体对象添加到EvaluationContext的Properties字典中即可。在该字典中，用于存放安全主体条目对应的键值为“Principal”。

1: Public class CustomAuthorizationPolicy:IAuthorizationPolicy

2: {

3: //其他成员

4: bool Evaluate(EvaluationContext evaluationContext, ref object state)

5: {

6: //其他操作

7: evaluationContext. Properties[“Principal”] = customPrincipal;

8: return true;

9: }

10: }

那么自定义的AuthorizationPolicy通过怎样的方式被应用到WCF的授权运行时呢？这还是要借助于我们已经很熟悉的服务行为[ServiceAuthorizationBehavior](http://msdn2.microsoft.com/ms599547.aspx" \o "ServiceAuthorizationBehavior Class)。如下面给出的代码片断所示，ServiceAuthorizationBehavior具有一个类型为ReadOnlyCollection<IAuthorizationPolicy> 的ExternalAuthorizationPolicies属性，表示自定义AuthorizationPolicy的集合。

1: public sealed class ServiceAuthorizationBehavior : IServiceBehavior

2: {

3: //其他成员

4: public ReadOnlyCollection<IAuthorizationPolicy> ExternalAuthorizationPolicies { get; set; }

5: }

你可以通过编程的方式将自定义的AuthorizationPolicy添加到ServiceAuthorizationBehavior的ExternalAuthorizationPolicies集合中，也可以通过配置指定自定义AuthorizationPolicy的类型。如下面给出的配置片断所示，ServiceAuthorizationBehavior的ExternalAuthorizationPolicies集合对应的配置节点为<serviceAuthorization>/<authorizationPolicies>。

1: <configuration>

2: <system.serviceModel>

3: <behaviors>

4: <serviceBehaviors>

5: <behavior name="useCustomAuthorization">

6: <serviceAuthorization principalPermissionMode="Custom">

7: <authorizationPolicies >

8: <add policyType="AuthorizationPolicyType1" />

9: <add policyType="AuthorizationPolicyType2" />

10: ...

11: </authorizationPolicies>

12: </serviceAuthorization>

13: <serviceDebug includeExceptionDetailInFaults="true"/>

14: </behavior>

15: </serviceBehaviors>

16: </behaviors>

17: </system.serviceModel>

18: </configuration>

# **二、ServiceAuthorizationManager**

在ServiceAuthorizationBehavior选择Custom安全主体权限模式的情况下，除了自定义AuthorizationPolicy，你还可以通过自定义ServiceAuthorizationManager来提供当前的安全主体。下面给出了ServiceAuthorizationManager的定义，从中我们可以看出它具有两个CheckAccess方法用于实现授权。方法的返回值表示当前请求的服务操作是否被授权指定。实际上最终的授权判断实现在受保护方法CheckAccessCore中，并且在ServiceAuthorizationManager中该方法直接返回True。

1: public class ServiceAuthorizationManager

2: {

3: //其他成员

4: public virtual bool CheckAccess(OperationContext operationContext);

5: public virtual bool CheckAccess(OperationContext operationContext, ref Message message);

6: protected virtual bool CheckAccessCore(OperationContext operationContext);

7: }

当ServiceAuthorizationBehavior的PrincipalPermissionMode被设置成Custom的情况下，被设置的当前安全主体实际上是通过当前服务安全上下文（ServiceSecurityContext）获取的。具体来说，ServiceSecurityContext具有一个表示授权信息的AuthorizationContext对象。和EvaluationContext一样，AuthorizationContext也具有一个字典类型的Properties属性。实际上，通过AuthorizationPolicy添加到EvaluationContext中的属性，最终都会被转移到当前AuthorizationContext的Properties属性中。

1: public class ServiceSecurityContext

2: {

3: //其他成员

4: public AuthorizationContext AuthorizationContext { get; }

5: }

6: public abstract class AuthorizationContext : IAuthorizationComponent

7: {

8: //其他成员

9: public abstract IDictionary<string, object> Properties { get; }

10: }

所以只要我们能够在WCF从当前AuthorizationContext获取安全主体之前对其进行初始化，整个基于安全主体的授权体系就能正常运作，而这个工作可以通过自定义ServiceAuthorizationManager来实现。一般来讲，我们只需通过继承ServiceAuthorizationManager，重写虚方法CheckAccessCore进行安全主体的初始化。

1: public class CustomServiceAuthorizationManager : ServiceAuthorizationManager

2: {

3: protected override bool CheckAccessCore(OperationContext operationContext)

4: {

5: //其他操作

6: AuthorizationContext authorizationContext = operationContext.ServiceSecurityContext.AuthorizationContext;

7: authorizationContext.Properties["Principal"] = customPrincipal;

8: return true;

9: }

10: }

自定义的ServiceAuthorizationManager最终还是通过ServiceAuthorizationBehavior这个服务行为应用到WCF授权框架体系中。如下面给出的代码片断所示，在ServiceAuthorizationBehavior中依然具有相应属性定义的。而在ServiceAuthorizationBehavior的配置节中，ServiceAuthorizationManager对应的配置属性为serviceAuthorizationManager，你可以通过该配置属性将设置自定义ServiceAuthorizationManager的类型。

1: public sealed class ServiceAuthorizationBehavior: IServiceBehavior

2: {

3: //其他成员

4: public ServiceAuthorizationManager ServiceAuthorizationManager { get; set; }

5: }

如果两种默认的安全主体权限模式（UseWindowsGroup和UseAspNetRoles)不能满足你的要求，你需要自定义安全主体提供方式，自定义AuthorizationPolicy或者ServiceAuthorizationManager不失为一个很好的解决方案。为了让你对此有个深刻的认识，在《[下篇](http://www.cnblogs.com/artech/archive/2011/07/09/customauthorization02.html)》中我们提供一个完整的实例。