**Лекція 17 Управління екземплярами сервісу WCF**

Суть управління екземплярами полягає в застосуванні ряду методик, використовуваних в Windows Communication Foundation для зв'язування набору повідомлень з примірником сервісу. Такі методики необхідні, оскільки додатки пред'являють абсолютно різні вимоги до масштабованості, продуктивності, пропускної здатності, транзакциям і черг викликів, а єдиного рішення на всі випадки життя просто немає. При розробці масштабованих уніфікованих додатків, орієнтованих на сервіси, не обійтися без знань про управління екземплярами. У статті даються найбільш оптимальні рішення для різних режимів управління екземплярами, пропонуються принципи, що визначають, коли і як використовувати ці режими, розглядаються деякі суміжні питання, такі як поведінки (behaviors), контексти, граничні операції (demarcating operations) та деактивація примірників.

В цілому, режим примірника сервісу - це особливість реалізації сервісу, яка не повинна проявлятися на стороні клієнта. З цією і деякими іншими цілями в Windows Communication Foundation визначено поняття поведінки. Поведінка - це локальний атрибут сервісу, що не впливає на його шаблони взаємодії.

ServiceBehaviorAttribute визначає поведінку, що впливає на всі кінцеві точки сервісу, і застосовується безпосередньо до класу реалізації сервісу. Як показано в лістингу 1, цей атрибут задає властивість InstanceContextMode перечислимого типу InstanceContextMode, значення якого управляє тим, які режими примірника використовуються сервісом.

Лістинг 1. InstanceContextMode

public enum InstanceContextMode

{

     PerCall, PerSession, Shareable, Single

}

[AttributeUsage (AttributeTargets.Class)]

public sealed class ServiceBehaviorAttribute: Attribute, ...

{

     public InstanceContextMode InstanceContextMode {get; set;}

     ... // Інші члени

}

Примірники сервісів, створювані для кожного виклику

"Примірники сервісів, створювані для кожного виклику" (per-call services) - режим створення екземплярів в Windows Communication Foundation за замовчуванням. Коли тип сервісу налаштований на активацію сервісу для кожного виклику, екземпляр сервісу є об'єктом загальномовного виконуючого середовища (CLR), який існує тільки під час виконання клієнтського виклику. Для кожного клієнтського запиту створюється виділений екземпляр сервісу. //На рис. 1 показано, як відбувається активація сервісу для кожного виклику.//

Примірники сервісів, створювані для кожного сеансу

Windows Communication Foundation може підтримувати закритий сеанс між клієнтом і певним примірником сервісу. Коли клієнт створює новий проксі для сервісу, що підтримує інформацію про сеанс, клієнт отримує новий виділений екземпляр сервісу, незалежний від інших екземплярів того ж сервісу. Цей екземпляр зазвичай існує до тих пір, поки не перестане бути потрібним клієнту. Кожен закритий сеанс підтримує унікальну зв'язок проксі з певним примірником сервісу. Зауважте: у клієнтського сеансу є по одному екземпляру сервісу для кожного проксі. Якщо клієнт створює інший проксі для тієї ж або іншої кінцевої точки, другий проксі буде пов'язаний з новими примірником та сеансом.

Оскільки екземпляр сервісу залишається в пам'яті протягом всього сеансу, він може зберігати стан в пам'яті; ця модель програмування дуже схожа на класичну клієнт-серверної моделлю. Отже, в ній виникають ті ж проблеми з масштабованість і транзакціями, що і в класичній моделі. Сервіс, налаштований на закриті сеанси, як правило, не може підтримувати більше декількох десятків (або, можливо, декількох сотень) клієнтів через витрат виконання кожного такого виділеного примірника сервісу.

У підтримки сеансу два аспекти: контракт і поведінку. Контракт необхідний за межами сервісу, оскільки клієнтської частини виконуючого середовища Windows Communication Foundation має бути відомо, чи потрібно використовувати сеанс. У атрибута ServiceContract є логічне властивість Session:

[AttributeUsage (

     AttributeTargets.Interface | AttributeTargets.Class,

     Inherited = false)]

public sealed class ServiceContractAttribute: Attribute

{

     public bool Session {get; set;}

     ... // Інші члени

}

За замовчуванням Session має значення false. Для підтримки сеансів надайте Session значення true на рівні контракту:

[ServiceContract (Session = true)]

interface IMyContract {...}

Щоб завершити настройку, ви повинні вказати Windows Communication Foundation, що екземпляр сервісу потрібно зберігати протягом усього сеансу і що клієнтські повідомлення слід направляти йому. Це локальне поведінка визначають, привласнюючи значення InstanceContextMode.PerSession властивості InstanceContextMode атрибута ServiceBehavior:

[ServiceBehavior (InstanceContextMode =

     InstanceContextMode.PerSession)]

class MyService: IMyContract {...}

Зазвичай сеанс завершується, коли клієнт закриває проксі: при цьому сервіс повідомляється, що сеанс закінчено. Якщо сервіс підтримує IDisposable, викликається метод Dispose. У лістингу 3 показані контракт і сервіс, налаштовані на використання закритого сеансу, і їхній клієнт. Як видно з висновку, клієнт отримав виділений екземпляр.

Спільно використовувані сервіси

Windows Communication Foundation не дозволяє передавати посилання на об'єкти через кордони сервісів. Об'єкти є сутностями, специфічними для технологій, і спільне використання об'єктів порушує принципи взаємодії, орієнтованого на сервіси і не залежного від технологій. Проте іноді один клієнт може використовувати спільно з іншим клієнтом поточний стан свого сеансу. Спільно використовуваний (загальний) сервіс поводиться багато в чому аналогічно до сервісу, примірник якого створюється для кожного сеансу, але має одну важливу особливість: у примірника є унікальний ідентифікатор, і, створюючи сеанс із загальним сервісом, клієнт може передати логічну посилання на цей екземпляр іншому клієнту . Другий клієнт може створити незалежний сеанс, але працювати з тим же примірником. У кожному з цих сеансів також можна визначити свої періоди очікування за відсутності активності, які можуть спливати незалежно від інших сеансів.

Для налаштування спільно використовуваного сервісу надайте значення InstanceContextMode.Sharable властивості InstanceContextMode:

[ServiceBehavior (InstanceContextMode =

     InstanceContextMode.Sharable)]

class MyService: IMyContract {...}

Є два способи застосування загальних сервісів, які є варіантами однієї і тієї ж методики. Перший - просте дублювання проксі в тому ж домені додатки, а другий - передача адреси кінцевої точки іншому клієнту через Windows Communication Foundation.

При передачі посилання на екземпляр виникає цікава проблема роботи з Windows Communication Foundation: як бути, якщо клієнт спочатку створить екземпляр, потім передасть посилання другу клієнту, а потім закриє проксі до того, як другий клієнт створить свій проксі за посиланням? Щоб вирішити цю проблему, в Windows Communication Foundation використовується додатковий тайм-аут, за замовчуванням рівний хвилині. Windows Communication Foundation відстежує сеанси, пов'язані з одним і тим же примірником. При закритті проксі одним з клієнтом екземпляр сервісу знищується, тільки коли завершується останній сеанс і закінчується додатковий тайм-аут. Крім того, Windows Communication Foundation дозволяє настроювати цей додатковий тайм-аут. Додайте в елемент service конфігураційного файлу хоста сервісу властивість behaviorConfiguration, яке вказує на розділ behavior. У цьому розділі задайте необхідний додатковий тайм-аут у властивості instanceContextIdleTimeout:

<services>

     <service type = "MySharableService"

      behaviorConfiguration = "ShortLingerBehaviour">

     ...

     </ service>

</ services>

<behaviors>

     <behavior name = "ShortLingerBehaviour"

      instanceContextIdleTimeout = "00:05:00" />

</ behaviors>

Якщо додатковий тайм-аут не дуже підходить для вашої програми, то клієнтам, що працюють з сервісом, доведеться координувати свої дії, визначаючи, коли можна закривати проксі.

Singleton-сервіси

Singleton-сервіси в кінцевому рахунку є загальними. Коли сервіс налаштований як Singleton-сервіс, всі клієнти підключаються до одного і того ж єдиного екземпляру сервісу незалежно один від одного і від того, з якою кінцевою точкою сервісу вони з'єднуються. Singleton-сервіс існує постійно і знищується, тільки коли хост завершує роботу. Singleton-сервіс створюється рівно в той же час, коли створюється хост.

Застосовуючи Singleton-сервіс, не потрібно створювати сеанс з примірником сервісу або використовувати прив'язку, яка підтримує сеанс транспортного рівня. Якщо у контракту, використовуваного клієнтом, мається сеанс, то при закритті проксі клієнтом завершується тільки сеанс, але не Singleton-екземпляр. Крім того, сеанс ніколи не застаріває. Якщо Singleton-сервіс підтримує контракти без сеансу, ці контракти не будуть створюватися для кожного виклику - вони також будуть з'єднані з одним і тим же примірником. При роботі з Singleton-сервісом не можна викликати ResolveInstance. Причина очевидна: за своєю природою Singleton є загальним, і кожен клієнт просто повинен створювати для нього свої проксі.

Щоб налаштувати сервіс як Singleton-сервіс, надайте значення InstanceContextMode.Single властивості InstanceContextMode:

[ServiceBehavior (InstanceContextMode =

     InstanceContextMode.Single)]

class MySingleton: ...

{...}

У лістингу 5 показаний Singleton-сервіс з двома контрактами, один з яких вимагає сеанс, а інший - ні. Як видно з клієнтського викликом, виклики двох кінцевих точок направляються одному примірнику, а закриття проксі не приводить до завершення Singleton-сервісу.

Лістинг 5. Singleton-сервіс і клієнт

код сервісу

[ServiceContract (Session = true)]

interface IMyContract

{

     [OperationContract]

     void MyMethod ();

}

[ServiceContract]

interface IMyOtherContract

{

     [OperationContract]

     void MyOtherMethod ();

}

[ServiceBehavior (InstanceContextMode =

     InstanceContextMode.Single)]

class MySingleton: IMyContract, IMyOtherContract, IDisposable

{

     int m\_Counter = 0;

     public MySingleton ()

     {

         Trace.WriteLine ("MyService.MyService ()");

     }

     public void MyMethod ()

     {

         m\_Counter ++;

         Trace.WriteLine ("Counter =" + m\_Counter);

     }

     public void MyOtherMethod ()

     {

         m\_Counter ++;

         Trace.WriteLine ("Counter =" + m\_Counter);

     }

     public void Dispose ()

     {

         Trace.WriteLine ("MyService.Dispose ()");

     }

}

клієнтський код

MyContractProxy proxy1 = new MyContractProxy ();

proxy1.MyMethod ();

proxy1.Close ();

MyOtherContractProxy proxy2 = new MyOtherContractProxy ();

proxy2.MyOtherMethod ();

proxy2.Close ();

висновок

MyService.MyService ()

Counter = 1

Counter = 2