**Лекция 17 Управление экземплярами сервиса WCF**

Суть управления экземплярами заключается в применении ряда методик, используемых в Windows Communication Foundation для связывания набора сообщений с экземпляром сервиса. Такие методики необходимы, поскольку приложения предъявляют совершенно разные требования к масштабируемости, производительности, пропускной способности, транзакциям и очередям вызовов, а единственного решения на все случаи жизни просто нет. При разработке масштабируемых унифицированных приложений, ориентированных на сервисы, не обойтись без знаний об управлении экземплярами. В статье даются наиболее оптимальные решения для разных режимов управления экземплярами, предлагаются принципы, определяющие, когда и как использовать эти режимы, рассматриваются некоторые смежные вопросы, такие как поведения (behaviors), контексты, граничные операции (demarcating operations) и деактивация экземпляров.

В целом, режим экземпляра сервиса - это особенность реализации сервиса, которая не должна проявляться на стороне клиента. С этой и некоторыми другими целями в Windows Communication Foundation определено понятие поведения. Поведение - это локальный атрибут сервиса, не влияющий на его шаблоны взаимодействия.

ServiceBehaviorAttribute определяет поведение, влияющее на все конечные точки сервиса, и применяется напрямую к классу реализации сервиса. Как показано в **листинге 1**, этот атрибут задает свойство InstanceContextMode перечислимого типа InstanceContextMode, значение которого управляет тем, какие режимы экземпляра используются сервисом.

**Листинг 1. InstanceContextMode**

public enum InstanceContextMode

{

PerCall, PerSession, Shareable, Single

}

[AttributeUsage(AttributeTargets.Class)]

public sealed class ServiceBehaviorAttribute : Attribute, ...

{

public InstanceContextMode InstanceContextMode {get;set;}

... // Прочие члены

}

**Экземпляры сервисов, создаваемые для каждого вызова**

"Экземпляры сервисов, создаваемые для каждого вызова" (per-call services) - режим создания экземпляров в Windows Communication Foundation по умолчанию. Когда тип сервиса настроен на активацию сервиса для каждого вызова, экземпляр сервиса является объектом общеязыковой исполняющей среды (CLR), который существует только во время выполнения клиентского вызова. Для каждого клиентского запроса создается выделенный экземпляр сервиса. На **рис. 1** показано, как происходит активация сервиса для каждого вызова.

## Экземпляры сервисов, создаваемые для каждого сеанса

Windows Communication Foundation может поддерживать закрытый сеанс между клиентом и определенным экземпляром сервиса. Когда клиент создает новый прокси для сервиса, поддерживающего информацию о сеансе, клиент получает новый выделенный экземпляр сервиса, независимый от остальных экземпляров того же сервиса. Этот экземпляр обычно существует до тех пор, пока не перестанет быть нужным клиенту. Каждый закрытый сеанс поддерживает уникальную связь прокси с определенным экземпляром сервиса. Заметьте: у клиентского сеанса имеется по одному экземпляру сервиса для каждого прокси. Если клиент создает другой прокси для той же или другой конечной точки, второй прокси будет связан с новыми экземпляром и сеансом.

Поскольку экземпляр сервиса остается в памяти на протяжении всего сеанса, он может хранить состояние в памяти; эта модель программирования очень схожа с классической клиент-серверной моделью. Следовательно, в ней возникают те же проблемы с масштабируемостью и транзакциями, что и в классической модели. Сервис, настроенный на закрытые сеансы, как правило, не может поддерживать более нескольких десятков (или, может быть, нескольких сотен) клиентов из-за издержек выполнения каждого такого выделенного экземпляра сервиса.

У поддержки сеанса два аспекта: контракт и поведение. Контракт необходим за пределами сервиса, поскольку клиентской части исполняющей среды Windows Communication Foundation должно быть известно, нужно ли использовать сеанс. У атрибута ServiceContract есть логическое свойство Session:

[AttributeUsage(

AttributeTargets.Interface|AttributeTargets.Class,

Inherited=false)]

public sealed class ServiceContractAttribute : Attribute

{

public bool Session {get;set;}

... // Прочие члены

}

По умолчанию Session имеет значение false. Для поддержки сеансов присвойте Session значение true на уровне контракта:

[ServiceContract(Session = true)]

interface IMyContract {...}

Чтобы завершить настройку, вы должны указать Windows Communication Foundation, что экземпляр сервиса нужно хранить на протяжении всего сеанса и что клиентские сообщения следует направлять ему. Это локальное поведение определяют, присваивая значение InstanceContextMode.PerSession свойству InstanceContextMode атрибута ServiceBehavior:

[ServiceBehavior(InstanceContextMode =

InstanceContextMode.PerSession)]

class MyService : IMyContract {...}

Обычно сеанс завершается, когда клиент закрывает прокси: при этом сервис уведомляется, что сеанс окончен. Если сервис поддерживает IDisposable, вызывается метод Dispose. В **листинге 3** показаны контракт и сервис, настроенные на использование закрытого сеанса, и их клиент. Как видно из вывода, клиент получил выделенный экземпляр.

## Совместно используемые сервисы

Windows Communication Foundation не позволяет передавать ссылки на объекты через границы сервисов. Объекты являются сущностями, специфичными для технологий, и совместное использование объектов нарушает принципы взаимодействия, ориентированного на сервисы и не зависящего от технологий. Тем не менее иногда один клиент может использовать совместно с другим клиентом текущее состояние своего сеанса. Совместно используемый (общий) сервис ведет себя во многом аналогично сервису, экземпляр которого создается для каждого сеанса, но имеет одну важную особенность: у экземпляра есть уникальный идентификатор, и, создавая сеанс с общим сервисом, клиент может передать логическую ссылку на этот экземпляр другому клиенту. Второй клиент может создать независимый сеанс, но работать с тем же экземпляром. В каждом из этих сеансов также можно определить свои периоды ожидания при отсутствии активности, которые могут истекать независимо от других сеансов.

Для настройки совместно используемого сервиса присвойте значение InstanceContextMode.Sharable свойству InstanceContextMode:

[ServiceBehavior(InstanceContextMode =

InstanceContextMode.Sharable)]

class MyService : IMyContract {...}

Есть два способа применения общих сервисов, являющиеся вариантами одной и той же методики. Первый - простое дублирование прокси в том же домене приложения, а второй - передача адреса конечной точки другому клиенту через Windows Communication Foundation.

При передаче ссылки на экземпляр возникает интересная проблема работы с Windows Communication Foundation: как быть, если клиент сначала создаст экземпляр, затем передаст ссылку второму клиенту, а потом закроет прокси до того, как второй клиент создаст свой прокси по ссылке? Чтобы решить эту проблему, в Windows Communication Foundation используется дополнительный тайм-аут, по умолчанию равный минуте. Windows Communication Foundation отслеживает сеансы, связанные с одним и тем же экземпляром. При закрытии прокси одним из клиентом экземпляр сервиса уничтожается, только когда завершается последний сеанс и истекает дополнительный тайм-аут. Кроме того, Windows Communication Foundation позволяет настраивать этот дополнительный тайм-аут. Добавьте в элемент service конфигурационного файла хоста сервиса свойство behaviorConfiguration, указывающее на раздел behavior. В этом разделе задайте требуемый дополнительный тайм-аут в свойстве instanceContextIdleTimeout:

<services>

<service type="MySharableService"

behaviorConfiguration="ShortLingerBehaviour">

...

</service>

</services>

<behaviors>

<behavior name="ShortLingerBehaviour"

instanceContextIdleTimeout="00:05:00" />

</behaviors>

Если дополнительный тайм-аут не слишком подходит для вашего приложения, то клиентам, работающим с сервисом, придется координировать свои действия, определяя, когда можно закрывать прокси.

## Singleton-сервисы

Singleton-сервисы в конечном счете являются общими. Когда сервис сконфигурирован как Singleton-сервис, все клиенты подключаются к одному и тому же единственному экземпляру сервиса независимо друг от друга и от того, с какой конечной точкой сервиса они соединяются. Singleton-сервис существует постоянно и уничтожается, только когда хост завершает работу. Singleton-сервис создается ровно в то же время, когда создается хост.

Применяя Singleton-сервис, не требуется создавать сеанс с экземпляром сервиса или использовать привязку, которая поддерживает сеанс транспортного уровня. Если у контракта, используемого клиентом, имеется сеанс, то при закрытии прокси клиентом завершается только сеанс, но не Singleton-экземпляр. Кроме того, сеанс никогда не устаревает. Если Singleton-сервис поддерживает контракты без сеанса, эти контракты не будут создаваться для каждого вызова - они также будут соединены с одним и тем же экземпляром. При работе с Singleton-сервисом нельзя вызвать ResolveInstance. Причина очевидна: по своей природе Singleton является общим, и каждый клиент просто должен создавать для него свои прокси.

Чтобы настроить сервис как Singleton-сервис, присвойте значение InstanceContextMode.Single свойству InstanceContextMode:

[ServiceBehavior(InstanceContextMode =

InstanceContextMode.Single)]

class MySingleton : ...

{...}

В **листинге 5** показан Singleton-сервис с двумя контрактами, один из которых требует сеанс, а другой - нет. Как видно по клиентскому вызову, вызовы двух конечных точек направляются одному экземпляру, а закрытие прокси не приводит к завершению Singleton-сервиса.

**Листинг 5. Singleton-сервис и клиент**

Код сервиса

[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()");

}

}

Клиентский код

MyContractProxy proxy1 = new MyContractProxy();

proxy1.MyMethod();

proxy1.Close();

MyOtherContractProxy proxy2 = new MyOtherContractProxy();

proxy2.MyOtherMethod();

proxy2.Close();

Вывод

MyService.MyService()

Counter = 1

Counter = 2