# Orleans的Grain事件發送機制：Observer

Orleans的Grain除了用 *Request - Response* 的非同步RPC機制來溝通之外，還有另外兩個機制：Observer和Streaming，也就是事件(event)觸發功能和事件流(event stream)，來達到Grain與Grain、Grain與外界物件之間非同步、響應式執行動作或傳送訊息的目的。

和昨天描述的Timer與Reminder很像，使用Observer的時候不需要底層Silo額外配置，而Stream則需要在Silo額外配置Stream Provider。

但使用Observer架構的時候，Grain的應用邏輯開發者必須要自行負責與事件發送者之間的訂閱事件關係，而且事件發送者必須要額外實作事件發送的邏輯，其實要撰寫的程式碼比撰寫事件流的複雜。

而且很重要的是，Observer的事件訂閱架構無法讓Grain在Silo故障之後該Grain執行實體被轉移到其他Silo執行的情況下，繼續接收事件通知。

原本官方在 v3.0 正式發行時建議將Observer的API淘汰，全改為事件流的機制，但後來還是繼續支援，甚至在[v7.0 在新增一個管理訂閱者的Utility類別](https://github.com/dotnet/orleans/issues/7858) ObserverManager：

在 v7.0 的github分支有提供一個內建的ObserverManager類別： https://github.com/dotnet/orleans/blob/main/src/Orleans.Core/Utils/ObserverManager.cs

但 v3.x 沒有提供，不過可以從官方Github庫 v3.x 分支的測試程式碼拷貝過來使用： https://github.com/dotnet/orleans/blob/3.x/test/Grains/TestGrains/ObserverManager.cs

替代之前 Orleans v2.x 的 [ObserverSubscriptionManager<T>](https://learn.microsoft.com/dotnet/api/orleans.observersubscriptionmanager-1?view=orleans-2.0) 類別。

## Observer架構實作

Observer架構的實作，在訊息接收的 C# 介面定義上，需要繼承一個Orleans框架提供的 [IGrainObserver](https://learn.microsoft.com/dotnet/api/orleans.igrainobserver) 介面：

public interface IConsumer : IGrainObserver  
{  
 void ReceiveEvent(string message);  
}

介面 [IGrainObserver](https://learn.microsoft.com/dotnet/api/orleans.igrainobserver) 其實沒有定義任何方法，只有要求傳送訊息的自定義函式必須是不回傳值的 void ，所以上述的 void ReceiveEvent(string message); 是自行定義的接收訊息自定義函式，用來接收事件發送者Grain（以下簡稱Producer Grain）發送的訊息。

然後接收訊息的Consumer Grain，實務上還會多提供幾個相關的RPC方法，以便於從接收端來訂閱或取消訂閱事件：

public interface IConsumerGrain : IGrainWithIntegerKey, IConsumer  
{  
 Task SubscribeTo(ProducerInfo producerInfo);  
 Task Unsubscribe();  
}

其中那個 ProducerInfo 只是個自訂的Data Object，用來儲存當需要訂閱/取消訂閱事件時，需取得Producer Grain的資訊，例如，當Producer Grain是使用 [IGrainWithIntegerCompoundKey](https://learn.microsoft.com/dotnet/api/orleans.igrainwithintegercompoundkey) 這種 *數字 + 字串* 的Grain Identifier介面時，就可以定義 ProducerInfo 為：

public record ProducerInfo(int Id, string Namespace);

這邊使用C# 9之後提供的 [*record*](https://learn.microsoft.com/dotnet/csharp/whats-new/tutorials/records) 型態語法來定義Data Object。

所以Consumer Grain的範例實作內容：

public class EventConsumerGrain : Grain, IConsumerGrain  
{  
 private readonly ILogger<EventConsumerGrain> \_logger;  
  
 private ProducerInfo? subscribed\_producerInfo;  
  
 public EventConsumerGrain(ILogger<EventConsumerGrain> logger)  
 {  
 \_logger = logger;  
 }  
  
 public void ReceiveEvent(string message)  
 {  
 \_logger.LogInformation("Subscriber {id} Received message: {message}", this.GetPrimaryKey(), message);  
 }  
  
 public async Task SubscribeTo(ProducerInfo producerInfo)  
 {  
 if (subscribed\_producerInfo is not null)  
 {  
 throw new Exception("Already subscribed to a producer");  
 }  
  
 var producer =  
 GrainFactory.GetGrain<ISubscribeProducerGrain>(producerInfo.Id, producerInfo.Namespace);  
  
 await producer.Subscribe(this.AsReference<IConsumerGrain>());  
 subscribed\_producerInfo = producerInfo;  
 }  
  
 public async Task Unsubscribe()  
 {  
 if (subscribed\_producerInfo is not null)  
 {  
 var producer =  
 GrainFactory.GetGrain<ISubscribeProducerGrain>(subscribed\_producerInfo.Id,subscribed\_producerInfo.Namespace);  
  
 await producer.Unsubscribe(this.AsReference<IConsumerGrain>());  
 }  
 }  
}

當Consumer Grain需要訂閱/取消訂閱事件時，需要使用 [AsReference<T>()](https://learn.microsoft.com/dotnet/api/orleans.grainextensions.asreference) 的擴充方法來取得自身對於其RPC介面的轉型，才能符合下面Producer Grain的訂閱方法中的參數型態，而且，這個擴充方法是將Grain實體物件轉成一個『弱參考([WeakReference](https://learn.microsoft.com/dotnet/api/system.weakreference-1))』再傳入Producer Grain的方法參數內，避免當Consumer物件結束時無法被Garbage collection，造成記憶體洩漏。

Consumer Grain的訂閱/取消訂閱的實作，實際上就是取得Producer Grain的參考來呼叫其訂閱/取消訂閱的RPC，介面宣告如下：

public interface ISubscribeProducerGrain : IProducerGrain  
{  
 public Task Subscribe(IConsumer consumer);  
   
 public Task Unsubscribe(IConsumer consumer);  
}

這兩個方法的輸入參數都是前述繼承 [IGrainObserver](https://learn.microsoft.com/dotnet/api/orleans.igrainobserver) 的 IConsumer 介面。 而繼承的自訂 IProducerGrain 介面，是Producer Grain需要有可以觸發執行發送事件動作的方法宣告，等下會提到。

所以Producer Grain有關於這兩個介面的實作內容如下：

public class EventProducerGrain : Grain, ISubscribeProducerGrain  
{  
 private readonly ILogger<EventProducerGrain> \_logger;  
 private IDisposable? \_timer;  
 private int \_counter = 0;  
 private readonly ObserverManager<IConsumer> \_observers;  
  
 public EventProducerGrain(ILogger<EventProducerGrain> logger)  
 {  
 \_logger = logger;  
 \_observers = new ObserverManager<IConsumer>(TimeSpan.FromMinutes(5), \_logger, "event\_demo");  
 }  
  
 /\*   
 other method implementation ...   
 \*/  
  
 public Task Subscribe(IConsumer consumer)  
 {  
 \_observers.Subscribe(consumer, consumer);  
 return Task.CompletedTask;  
 }  
  
 public Task Unsubscribe(IConsumer consumer)  
 {  
 \_observers.Unsubscribe(consumer);  
 return Task.CompletedTask;  
 }  
}

用來負責管理訂閱者物件參考的是前述講的在 Orleans v7 之後會提供的，目前在 Orleans v3.x 可從其框架原始碼的單元測試專案抓 ObserverManager<T> 類別程式碼來用，所以就完成了Producer Grain與Consumer Grain之間的訂閱/取消訂閱的實作。

Producer Grain需要有機制可以執行實際發送事件的動作，我們定義這些觸發動作的方法在 IProducerGrain 介面裡：

public interface IProducerGrain : IGrainWithIntegerCompoundKey  
{  
 public Task StartProducing();  
   
 public Task StopProducing();  
}

在Producer Grain的實作，使用Grain的計時器(Timer)機制來每秒發送事件，內容如下：

public class EventProducerGrain : Grain, ISubscribeProducerGrain  
{  
 private readonly ILogger<EventProducerGrain> \_logger;  
 private IDisposable? \_timer;  
 private int \_counter = 0;  
 private readonly ObserverManager<IConsumer> \_observers;  
  
 /\*   
 other method implementation ...   
 \*/  
  
 public Task StartProducing()  
 {  
 if (\_timer is not null)  
 {  
 throw new Exception("Already producing");  
 }  
  
 //Register a timer that produce an event every second  
 var period = TimeSpan.FromSeconds(1);  
 \_timer = RegisterTimer(TimerTick, null, period, period);  
  
 \_logger.LogInformation("I will produce a new event every {Period}", period);  
 return Task.CompletedTask;  
 }  
  
 private Task TimerTick(object \_)  
 {  
 var msg = $"Producing event {\_counter++}";  
 \_logger.LogInformation(msg);  
   
 //Notify all the observers  
 \_observers.Notify(o => o.ReceiveEvent(msg));  
   
 return Task.CompletedTask;  
 }  
  
 public Task StopProducing()  
 {  
 if (\_timer is not null)  
 {  
 \_timer.Dispose();  
 \_timer = null;  
 }  
  
 return Task.CompletedTask;  
 }  
  
 /\*   
 other method implementation ...   
 \*/  
  
}

由於有套用那個 ObserverManager<T> ，所以在發送事件的時候，只要呼叫 \_observers.Notify(o => o.ReceiveEvent(msg)) 就可以通知所有訂閱者了。

實際套用在 .NET 6 Minimal API 的程式碼如下：

var builder = WebApplication.CreateBuilder(args);  
  
builder.Host.UseOrleans(siloBuilder =>  
{  
 siloBuilder.UseLocalhostClustering();  
 siloBuilder.ConfigureApplicationParts(parts =>  
 {  
 parts.AddApplicationPart(typeof(EventProducerGrain).Assembly).WithReferences();  
 parts.AddApplicationPart(typeof(EventConsumerGrain).Assembly).WithReferences();  
 });  
});  
  
var app = builder.Build();  
  
var producerInfo = new ProducerInfo(0, "producer\_demo");  
  
app.MapGet("/consumer/subscribe/{id}", async (IClusterClient clusterClient, int id) =>  
{  
 var consumerGrain = clusterClient.GetGrain<IConsumerGrain>(id);  
 await consumerGrain.SubscribeTo(producerInfo);  
 return Results.Ok($"Consumer {id} subscribed");  
});  
app.MapGet("/consumer/unsubscribe/{id}", async (IClusterClient clusterClient, int id) =>  
{  
 var consumerGrain = clusterClient.GetGrain<IConsumerGrain>(id);  
 await consumerGrain.Unsubscribe();  
 return Results.Ok($"Consumer {id} unsubscribed");  
});  
  
app.MapGet("/producer\_start", async (IClusterClient clusterClient) =>  
{  
 var producerGrain = clusterClient.GetGrain<ISubscribeProducerGrain>(producerInfo.Id, producerInfo.Namespace);  
 await producerGrain.StartProducing();  
 return Results.Ok("Producer started");  
});  
app.MapGet("/producer\_stop", async (IClusterClient clusterClient) =>  
{  
 var producerGrain = clusterClient.GetGrain<ISubscribeProducerGrain>(producerInfo.Id, producerInfo.Namespace);  
 await producerGrain.StopProducing();  
 return Results.Ok("Producer stopped");  
});

完整範例，含一個單純展示Timer的程式碼在：  
<https://github.com/windperson/OrleansRpcDemo/tree/day18/src/Hosting/Server#rpcdemohostingminimalapi>

明天來繼續介紹Orleans的另一個非同步程式設計模式：事件流(Streams)。