# Orleans Grain的RPC設計進階技巧：例外處理、呼叫取消、重新進入(Reentrant)

## 例外處理(Exception Handling)

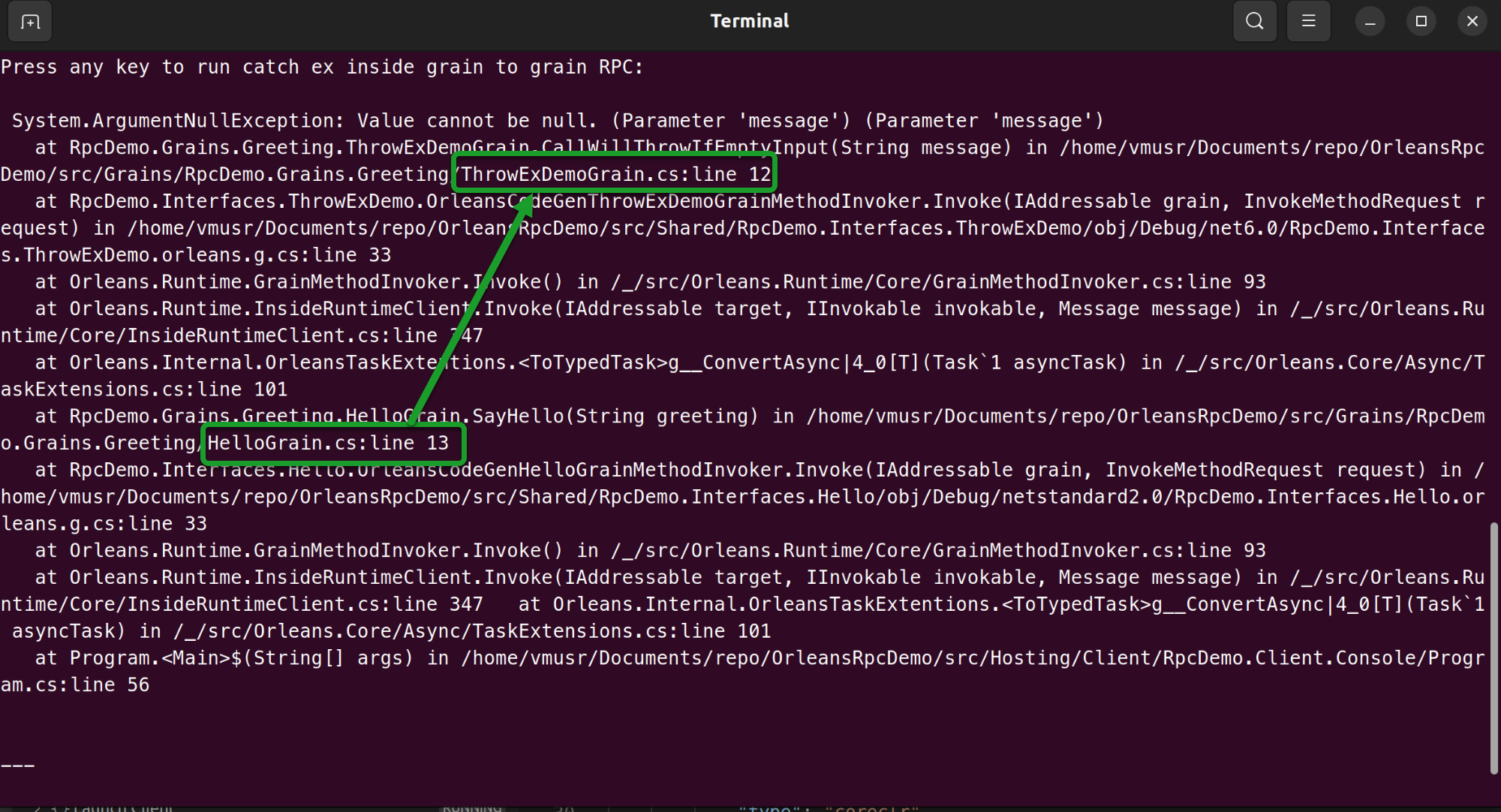
Grain的RPC方法可以拋出例外，如下範例：

public Task<string> CallWillThrowIfEmptyInput(string message)  
{  
 if (string.IsNullOrEmpty(message))  
 {  
 throw new ArgumentNullException(nameof(message));  
 }  
  
 return Task.FromResult(message);  
}

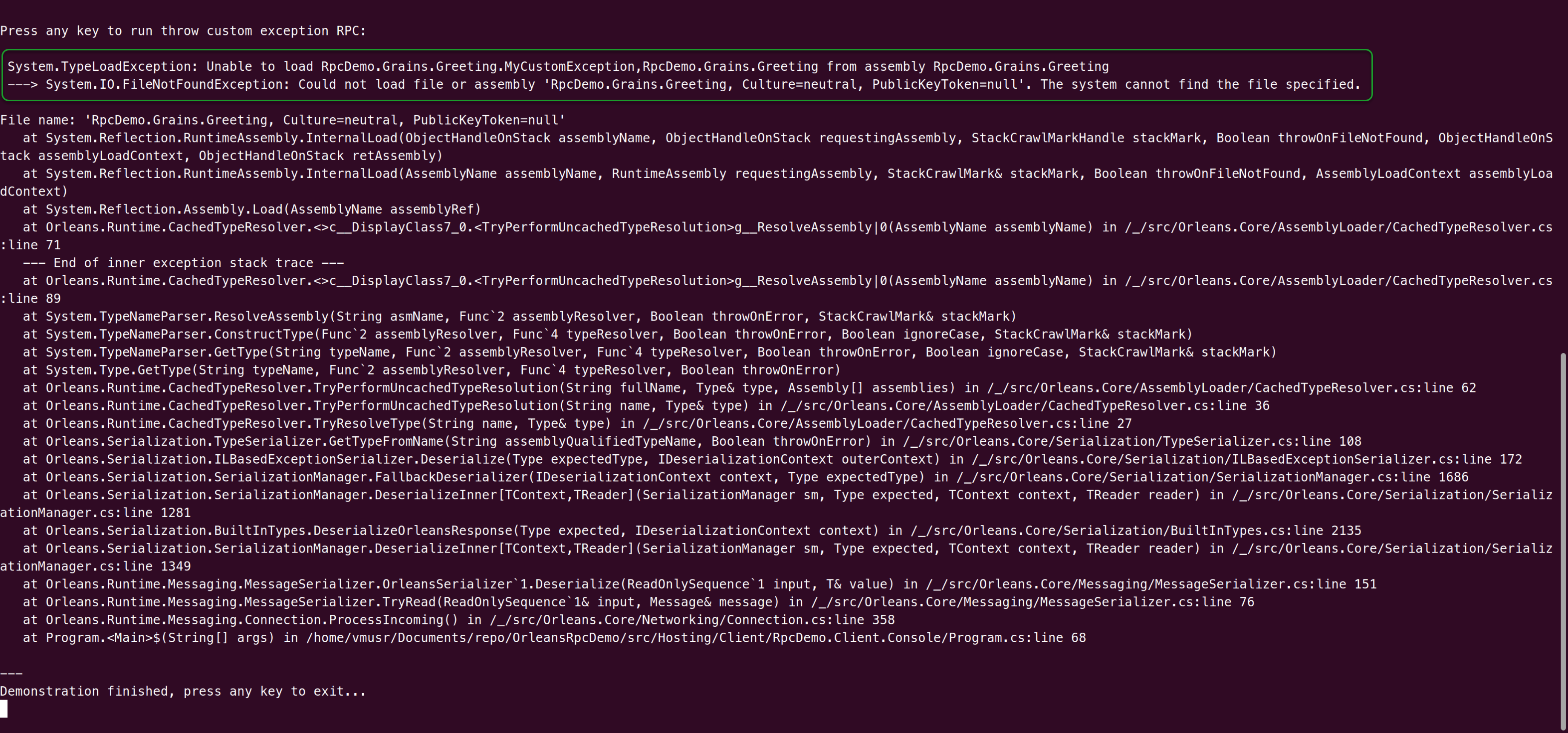
而在呼叫端，無論是使用 [IClusterClient](https://learn.microsoft.com/dotnet/api/orleans.iclusterclient) 的客戶端程式還是由另一個Grain用 [GrainFactory](https://learn.microsoft.com/dotnet/api/orleans.grain.grainfactory) 屬性取得的RPC呼叫實體，都可以使用標準的 C# try...catch 語法來捕捉例外：

try  
{  
 var throwDemoGrain = client.GetGrain<IThrowExDemoGrain>(0);  
 \_ = await throwDemoGrain.CallWillThrowIfEmptyInput(string.Empty);  
}  
catch (Exception e)  
{  
 System.WriteLine(e);  
}

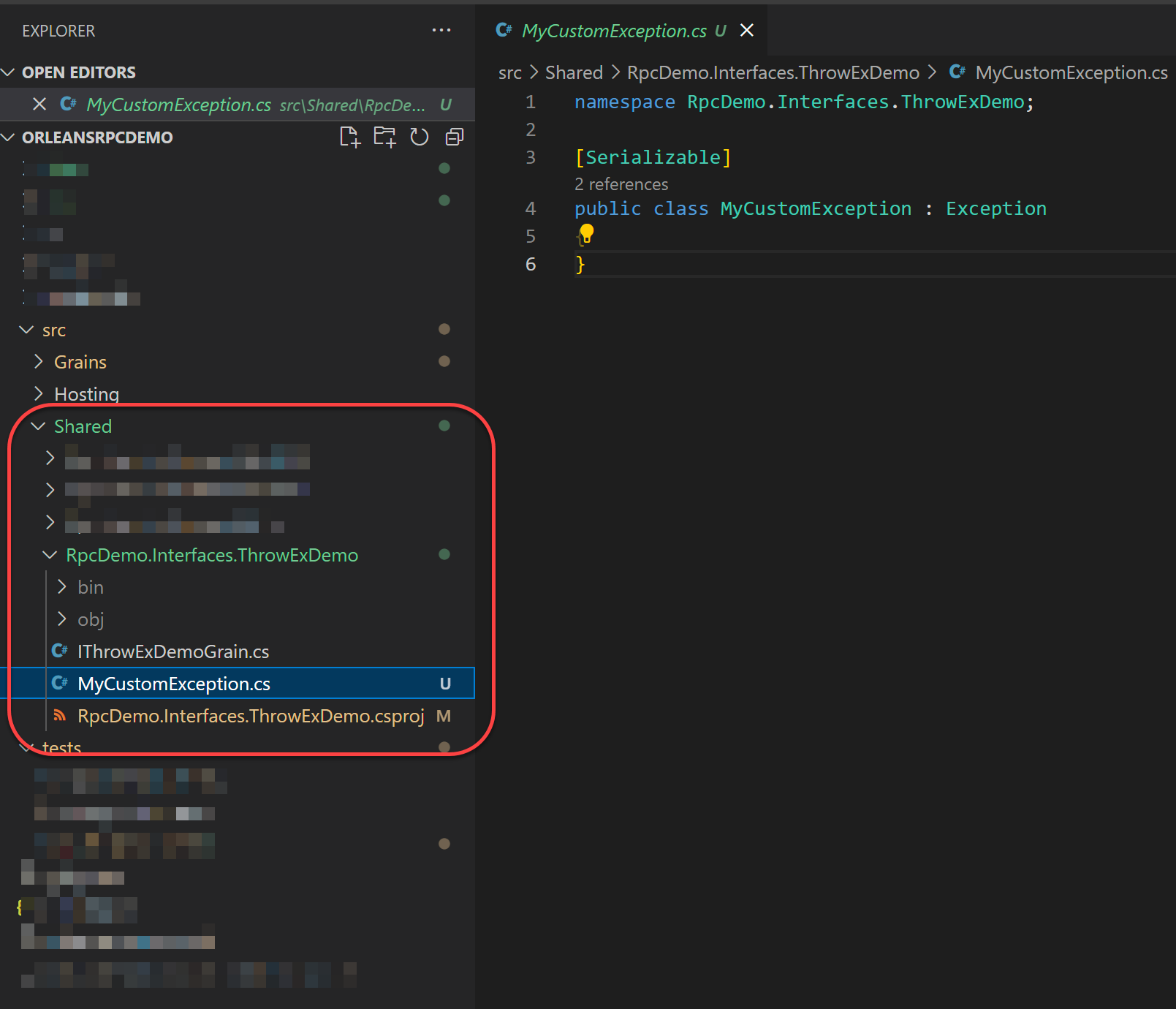
即使拋出例外的位置是當Grain去呼叫另一個Grain的RPC方法時，在最初始RPC呼叫的客戶端也可以正常捕捉到例外（假設呼叫另一個Grain的RPC之Grain內部實作沒捕捉例外的話），而且可藉由執行堆疊(Stacktrace)紀錄來查找例外觸發的執行路徑：



在設計例外處理的架構時有一點要注意，例外的類別必須是在Client端可解析的，否則會在擲回例外給Client端時發生錯誤，如下圖：



解決方法：將例外的類別定義原始碼放在Client端也會加入專案對專案參考的類別專案中，如RPC介面的專案，並將該例外類別加上 [[Serializable] 屬性](https://learn.microsoft.com/dotnet/api/system.serializableattribute)。



而由於目前 Orleans 3.x單元測試用的 [TestCluster](https://learn.microsoft.com/dotnet/api/orleans.testinghost.testcluster) 其底層實作是Client和Server共用同個appDomain，所以Client端和Server端的載入的型別兩邊都載入得到，如此導致用單元測試專案來跑Grain時，無法抓到這個錯誤，要特別注意。

## 使用Cancellation Token取消Grain RPC呼叫

RPC方法如果有定義 [CancellationToken](https://learn.microsoft.com/dotnet/api/system.threading.cancellationtoken) 輸入參數，並且在實作內容上有根據該參數的值來判斷是否要取消執行，就可以在呼叫端使用 [CancellationTokenSource](https://learn.microsoft.com/dotnet/api/system.threading.cancellationsource) 來取消呼叫：

//RPC方法定義  
public interface ILongJobGrain : IGrainWithStringKey  
{  
 public Task<string>? ProcessString(string input, CancellationToken cancellationToken);  
}  
  
//Grain實作  
public Task<string>? ProcessString(string input, CancellationToken cancellationToken)  
{  
 if (cancellationToken.IsCancellationRequested)  
 {  
 \_logger.LogInformation("Cancellation requested");  
 throw new OperationCanceledException();  
 }  
  
 // Pretend to do some work  
 var i = 0;  
 var result = string.Empty;  
 while (!cancellationToken.IsCancellationRequested)  
 {  
 \_logger.LogInformation("i = {i}", i);  
 i++;  
 Task.Delay(1000, cancellationToken).Wait(cancellationToken);  
 if (i > 20)  
 {  
 result = $"{input} done";  
 break;  
 }  
 }  
  
 if (cancellationToken.IsCancellationRequested)  
 {  
 \_logger.LogInformation("Cancellation requested");  
 throw new OperationCanceledException();  
 }  
  
 // Result gets sent back to the caller  
 return Task.FromResult(result);  
}

但實際上跑起來會發現這個方法只有在Silo內被其他Grain實體呼叫的情況下會有效，如果是在Client端呼叫Grain的RPC方法，則無法正常取消，因為Client端的呼叫是透過Grain的代理proxy物件來呼叫，而Grain的代理proxy物件是在Client端的AppDomain中，而非Silo內的AppDomain中，所以在Silo內的Grain實作中，是無法正常取得Client端的CancellationToken的值，因此無法正常取消呼叫。

官方有提供一個 [GrainCancellationToken](https://learn.microsoft.com/dotnet/api/orleans.graincancellationtoken) 類別的替代標準C#的 [CancellationToken](https://learn.microsoft.com/dotnet/api/system.threading.cancellationtoken)，但 [*不建議使用*](https://github.com/dotnet/orleans/issues/5299#issuecomment-452455112)，而且一樣沒有辦法由Client端遠端要求取消Silo端的Grain RPC方法執行。

建議使用的設計架構是將呼叫方法多包裝設計成由一個ProxyGrain來呼叫，而ProxyGrain再呼叫真正的Grain實作，例如以下範例：

using Orleans;  
using Orleans.Concurrency;  
  
//Proxy方法定義  
public interface ILongJobProxy : IGrainWithStringKey  
{  
 Task StartAsync(string input);  
 Task<string> GetResultAsync();  
   
 [AlwaysInterleave]  
 Task CancelAsync();  
}  
  
//ProxyGrain定義  
[Reentrant]  
public class LongJobProxy : Grain, ILongJobProxy  
{  
 private GrainCancellationTokenSource? tokenSource;  
 private Task<string>? processStringTask = null;  
 private ILongJobGrain? grain;  
  
 public Task StartAsync(string input)  
 {  
 tokenSource = new GrainCancellationTokenSource();  
 grain = GrainFactory.GetGrain<ILongJobGrain>(this.GetPrimaryKeyString());  
 processStringTask = grain.ProcessString(input, tokenSource.Token.CancellationToken);  
 return Task.CompletedTask;  
 }  
  
 public async Task<string> GetResultAsync()  
 {  
 if (processStringTask is null)  
 {  
 throw new InvalidOperationException("StartAsync must be called before GetResultAsync");  
 }  
  
 return await processStringTask;  
 }  
  
 public async Task CancelAsync()  
 {  
 if(tokenSource is null)  
 {  
 throw new InvalidOperationException("StartAsync must be called before CancelAsync");  
 }  
 if(tokenSource.IsCancellationRequested)  
 {  
 \_logger.LogWarning("CancelAsync called but cancellation has already been requested");   
 }  
 await tokenSource.Cancel();  
 }  
}

這在RPC方法定義中，用來取消執行方法的 CancelAsync() 加掛的 [[AlwaysInterleave]](https://learn.microsoft.com/dotnet/api/orleans.concurrency.alwaysinterleaveattribute) 屬性，是用來告訴Orleans這個方法可以被同時多個來源呼叫的，因為有可能網路或是Silo正重新載入Grain等問題導致單次呼叫 CancelAsync() 時，Silo端的Grain沒有收到，所以在架構設計上要能夠多次呼叫 CancelAsync() 確保Silo端的Grain能收到取消請求。

而為了要讓該方法的實作的確可以同時被多次呼叫，需要在實作RPC方法的Grain類別上加掛 [[Reentrant]](https://learn.microsoft.com/dotnet/api/orleans.concurrency.reentrantattribute) 屬性，宣告該Grain可以 **“重新進入(Reentrant)”** ：讓該Grain的RPC方法可以不必約束於原本的Actor模型定義，可以在grain正在執行中一個RPC方法時，容許有 [[AlwaysInterleave]](https://learn.microsoft.com/dotnet/api/orleans.concurrency.alwaysinterleaveattribute) 屬性的RPC方法也可被呼叫而不需等前一個RPC方法結束執行。

在命令列的Client端程式呼叫時就可以寫成這樣，在呼叫RPC方法之後，可按 **Ctrl+C** 來取消執行：

var longJobProxy = client.GetGrain<ILongJobProxy>("job\_proxy");  
Console.CancelKeyPress += async (source, args) =>  
{  
 args.Cancel = true;  
 WriteLine("Cancelling ProcessString()...");  
 await longJobProxy.CancelAsync();  
};  
  
Console.WriteLine("Start ProcessString()..., press Ctrl+C to cancel");  
try  
{  
 await longJobProxy.StartAsync("long job demo");  
 var longJobResult = await longJobProxy.GetResultAsync();  
  
 Console.WriteLine($"Call LongJobGrain.ProcessString(\"Cancellable RPC Demo\") = {longJobResult}");  
}  
catch (Exception e)  
{  
 //RPC method cancelled  
 WriteLine(e);  
}

此範例程式GitHub專案在：  
<https://github.com/windperson/OrleansRpcDemo/tree/day15>

明天繼續討論Reentrant的用途，包括如何解決Grain RPC呼叫的死結問題。