# Orleans

Orleans是一个框架，它提供了一种直接的方法来构建分布式得到规模计算应用程序，而无需学习和应用复杂的并发或其他扩展模式。

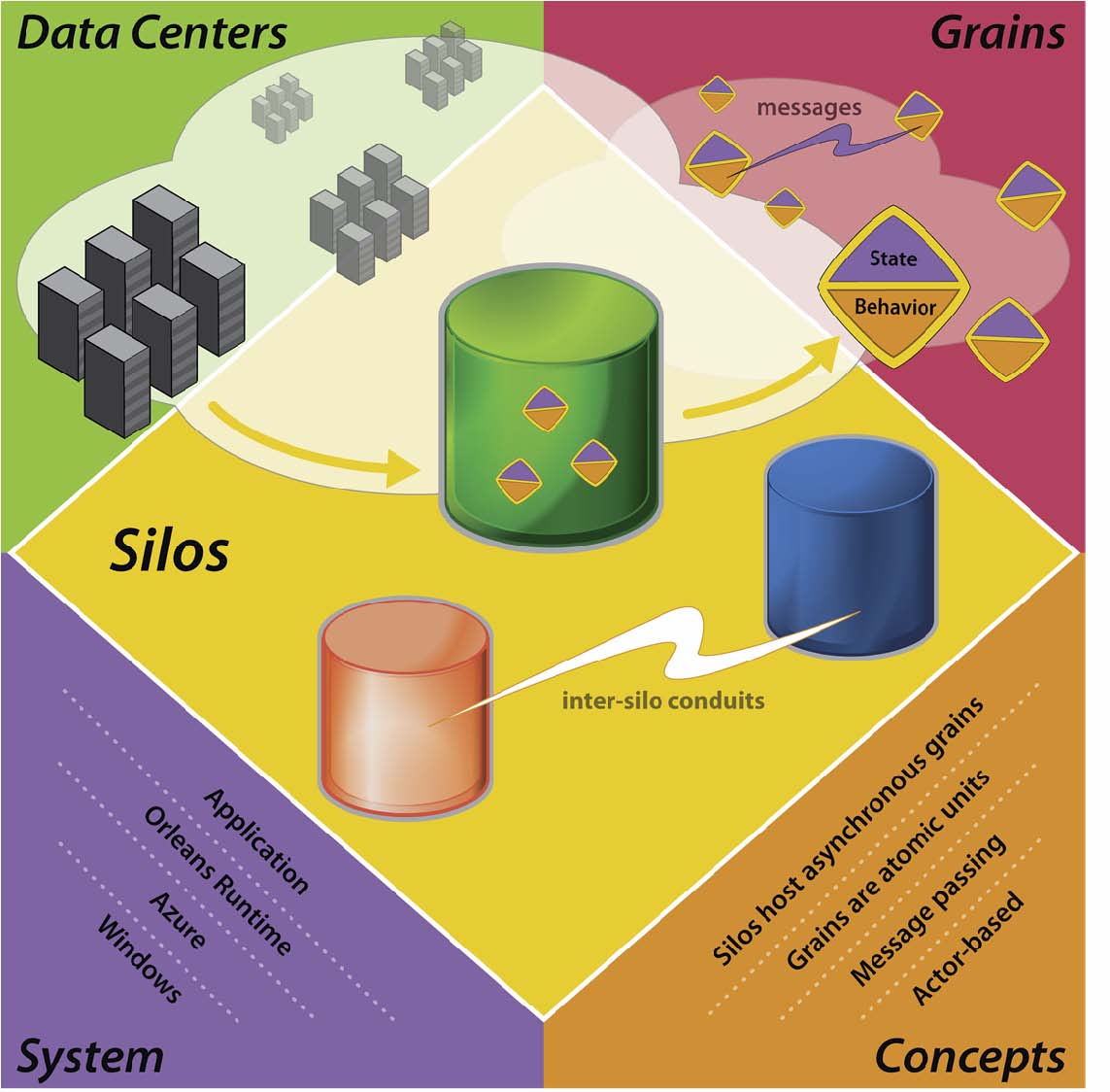
grain类型是一个简单的.NET类，它实现一个或多个应用程序定义的grain接口。单个grain实例是应用程序定义的grain类的实例，它们可以根据需要由服务器上的Orleans运行时自动创建，以处理对这些grain的请求。grains自然映射到大多数应用程序实体，例如用户，设备，会话，库存和订单。这使得构建面向对象的业务逻辑变得非常容易，但是在服务器集群中透明地扩展。每个grain在其应用逻辑选择的grain类型内具有稳定的逻辑标识（密钥），例如，用户电子邮件或设备ID或库存SKU代码。Orleans保证每个单独grain的单线程执行，从而保护应用程序逻辑免受并发和竞争的危险。

## 虚拟Actor

为了避免这些复杂性，我们构建了Orleans编程模型和运行时，它提高了actor抽象的级别。Orleans针对的不是分布式系统专家的开发人员，尽管我们的专家客户也发现它很有吸引力。它是基于actor的，但与现有的基于actor的平台不同，它将actor视为虚拟实体，而不是物理实体。首先，Orleans actor总是存在。它无法显式创建或销毁。它的存在超越了其任何内存实例化的生命周期，因此超越了任何特定服务器的生命周期。其次，Orleans actor被自动实例化：如果没有actor的内存实例，发送给actor的消息会导致在可用服务器上创建新实例。未使用的actor实例将作为运行时资源管理的一部分自动回收。actor永远不会失败：如果是服务器 崩溃时，发送给 故障服务器上运行的actor的下一条消息 会导致Orleans自动 在另一台服务器上重新实例化actor ，从而无需应用程序监督并明确重新创建失败的actor。第三，actor实例的位置对应用程序代码是透明的，这极大地简化了编程。第四，Orleans以自动创建同一个无状态actor的多个实例，无缝扩展热门actor。

总的来说，Orleans为开发人员提供了一个虚拟的“actor空间”，类似于虚拟内存，允许他们调用系统中的任何actor，无论它是否存在于内存中。虚拟化依赖于从虚拟actor映射到当前正在运行的物理实例的转接。这种转接性为运行时提供了解决许多硬分布式系统问题的机会，这些问题必须由开发人员解决，例如actor放置和负载平衡，停用未使用的actor，以及服务器故障后的actor恢复，让他们走向正确这是众所周知的困难。因此，虚拟actor方法显着简化了编程模型，同时允许运行时透明地平衡负载和从故障中恢复。

运行时通过分布式目录支持转接，该目录从actor身份映射到其当前物理位置。Orleans通过使用该映射的本地缓存来最小化间接的运行时成本。事实证明，这种策略非常有效。我们的生产服务通常会看到缓存命中率超过90％。



## Orleans优势

* **熟悉的面向对象编程（OOP）范例**

Grains是.NET类，它使用异步方法实现声明的.NET grain接口。因此，粒子对程序员来说似乎是可以直接调用远程对象的方法。这为程序员提供了熟悉的OOP范例，办法是将方法调用转换为消息，将它们路由到正确的端点，调用目标Grain的方法，以完全透明的方式处理故障和极端情况。

* **单线程执行Grain**

运行时保证Grain一次不会在多个线程上执行。结合与其他Grain的隔离，程序员永远不会在Grain级别来自行处理并发性，并且永远不需要使用锁或其他同步机制来控制对共享数据的访问。仅此功能使得分布式应用程序的开发易于为非专业程序员开发。

* **透明激活**

只有在有消息要处理时，运行时才会激活Grain。这清楚地区分了创建对应用程序代码可见和受其控制的Grain的引用以及内存中Grain的物理激活的概念，这对应用程序是透明的。在许多方面，这类似于虚拟内存，因为它决定何时“翻页”（停用）或“页面输入”（激活）Grain; 应用程序可以不间断地访问逻辑创建的Grain的完整“存储空间”，无论它们是否在任何特定时间点都在物理内存中。透明激活通过在硬件资源池中放置和迁移Grain来实现动态的自适应负载平衡。这个功能是对传统actor模型的重大改进

* **位置透明度**

程序员用来调用Grain方法或传递给其他组件的Grain引用（代理对象）只包含Grain的逻辑标识。Oleans运行时透明地完成了Grain的逻辑身份到其物理位置的转换以及相应的消息路由。应用程序代码与Grain通信，同时保持对其物理位置的遗忘，这可能由于故障或资源管理而导致随着时间的推移而改变，或者因为Grain在被调用时被停用。

* **与持久存储透明集成**

Orleans允许将Grain的内存状态声明性映射到持久性存储。它同步更新，透明地保证调用者只有在成功更新持久状态后才能收到结果。扩展和/或定制可用的现有持久存储提供程序集是直截了当的。

* **自动传播错误**

运行时使用异步和分布式try / catch的语义自动在调用链上传播未处理的错误。因此，错误不会在应用程序中丢失。

## 默认情况下透明可伸缩性

Orlean编程模型旨在引导程序员走上可能成功扩展应用程序或服务几个数量级的道路。这是通过结合经过验证的最佳实践和模式以及提供低级系统功能的有效实现来完成的。以下是支持可伸缩性和性能的一些关键因素：

* **应用状态的隐式Grain细分**。

通过使用Grain作为可直接寻址的实体，程序员隐式地打破了应用程序的整体状态。虽然Orleans编程模型没有规定Grain应该有多大或多小，但在大多数情况下，有相对大量的Grain - 数百万或更多 - 是有意义的，每个Grain代表应用程序的自然实体，例如用户帐户或采购订单。由于Grain是可单独寻址的，并且它们的物理位置被运行时抽象掉，Orleans在以透明和通用的方式平衡负载和处理热点方面具有极大的灵活性，而无需应用程序开发人员的任何考虑

* **自适应资源管理**

当Grain与它们相互作用时，Grain不会假设其他Grain的位置。由于这种位置透明性，运行时可以动态方式管理和调整可用硬件资源的分配。运行时通过对计算群集中的Grain的放置和迁移做出细粒度的决策来实现这一点，以响应加载和通信模式 - 而不会使传入的请求失败。通过创建特定Grain的多个副本，运行时可以增加Grain的吞吐量，而无需对应用程序代码进行任何更改。

* **多路通信**

Orleans的Grain具有逻辑端点，并且它们之间的消息传递在一组固定的所有物理连接（TCP套接字）上进行多路复用。这允许运行时托管数百万个可寻址实体，每个Grain具有低OS开销。此外，激活和停用Grain不会产生注册/取消注册物理端点（例如TCP端口或HTTP URL）或甚至关闭TCP连接的成本。

* **高效的安排**。

运行时调度跨自定义线程池执行大量单线程Grains，每个物理处理器核心具有一个线程。使用以非阻塞，基于延续的方式（Orleans编程模型的要求）编写的Grain代码，应用程序代码以非常有效的“协作”多线程方式运行而没有争用。这使系统能够达到高吞吐量并以非常高的CPU利用率（高达90％+）运行，并具有极高的稳定性。事实上，系统中的Grain数量的增加和负载的增加不会导致额外的线程或其他OS原语，这有助于单个节点和整个系统的可扩展性。

* **显式异步**。

Orleans编程模型使分布式应用程序的异步特性明确，并指导程序员编写非阻塞异步代码。结合异步消息传递和高效调度，无需明确使用多线程即可实现大规模的分布式并行性和总体吞吐量。

# Grain

## Grain是什么

Grains是Orleans编程模型的关键原语。Grains是Orleans应用程序的构建块，它们是隔离，分发和持久性的原子单元。Grains是表示应用程序实体的对象。就像在经典的面向对象编程中一样，Grain封装了实体的状态并在代码逻辑中对其行为进行编码。Grains可以通过调用彼此通过接口公开的方法来保持彼此的引用并进行交互。

Orleans旨在大大简化构建可伸缩应用程序并消除大多数并发性挑战

* 在两个Grain之间不共享数据，除非通过消息传递。
* 通过为每个单独的Grain提供单线程执行保证。

典型的Grain封装单个实体（例如特定用户或设备或会话）的状态和行为。

## 命周期：

1. 一个Grain或一个客户端调用Grain内的一个方法（通过一个Grain引用）
2. Grain被激活（Grain没有在集群中被激活），并创建一个Grain对象，这个过程叫Grain激活：
   1. 通过依赖注入在构造中创建Grain对象
   2. 如果使用持久化，从存付中还原Grain对象
   3. 如果被重写，OnActivateAsync被调用
3. Grain处理传入请求
4. Grain闲置一段时间
5. Silo运行时决定何时停止Grain
6. 如果被重写，Silo运行时调用OnDeactivateAsync
7. Silo运行时从内存中移除Grain

## Orleans包

### 关键Orleans包

Microsoft.Orleans.Core.Abstractions

包含Orleans.Core.Abstractions.dll，它定义开发应用程序代码（Grain接口和类）所需的Orleans公共类型。任何Orleans项目都需要直接或间接引用此包。将它添加到定义grain接口和类的项目中。

Microsoft.Orleans.OrleansCodeGenerator.Build

为Grain界面和实施项目建立时间支持。将它添加到Grain接口和实现项目中，以启用Grain参考和序列化器的代码生成。

Microsoft.Orleans.Server

一个易于构建和启动Silo的元数据包。包括以下包：

* Microsoft.Orleans.Core.Abstractions
* Microsoft.Orleans.Core
* Microsoft.Orleans.OrleansRuntime
* Microsoft.Orleans.OrleansProviders

Microsoft.Orleans.Client

一个易于构建和启动Orleans客户端（前端）的元数据包。包括以下包：

* Microsoft.Orleans.Core.Abstractions
* Microsoft.Orleans.Core
* Microsoft.Orleans.OrleansProviders

Microsoft.Orleans.Core

包含应用程序代码和Orleans客户端（前端）使用的大多数Orleans公共类型的实现。引用它来构建使用Orleans类型但不处理托管或Silo的库和客户端应用程序。包含在Microsoft.Orleans.Client和Microsoft.Orleans.Server元数据包中，并由大多数其他包直接或间接引用。

### 主机

Microsoft.Orleans.OrleansRuntime

用于配置和启动Silo的库。在您的silo主机项目中引用它。包含在Microsoft.Orleans.Server元包中。

Microsoft.Orleans.Runtime.Abstractions

包含Microsoft.Orleans.OrleansRuntime中实现的类型的接口和抽象。

### 集群提供商

Microsoft.Orleans.Hosting.AzureCloudServices

包含用于托管silos和Orleans客户端的帮助程序类作为Azure云服务（工作者角色和Web角色）

Microsoft.Orleans.Hosting.ServiceFabric

包含用于托管Silo的辅助类作为无状态Service Fabric服务

Microsoft.Orleans.Clustering.AzureStorage

包含使用Azure表存储集群成员资格数据的插件。

Microsoft.Orleans.Clustering.AdoNet

包括使用ADO.NET在其中一个受支持的数据库中存储集群成员资格数据的插件。

Microsoft.Orleans.OrleansConsulUtils

包含使用Consul存储集群成员资格数据的插件。

Microsoft.Orleans.OrleansZooKeeperUtils

包含使用ZooKeeper存储集群成员资格数据的插件。

Microsoft.Orleans.Clustering.DynamoDB

包括使用AWS DynamoDB存储集群成员资格数据的插件。

### 提醒提供商

Microsoft.Orleans.Reminders.AzureStorage

包含使用Azure表存储集群成员资格数据的插件。

Microsoft.Orleans.Reminders.AdoNet

包括使用ADO.NET在其中一个受支持的数据库中存储提醒的插件。

Microsoft.Orleans.Reminders.DynamoDB

包括使用AWS DynamoDB存储提醒的插件。

### 粮食储存供应商

Microsoft.Orleans.Persistence.AzureStorage

包括使用Azure表或Azure Blob存储Grain状态的插件。

Microsoft.Orleans.Persistence.AdoNet

包含使用ADO.NET在其中一个受支持的数据库中存储Grain状态的插件。

Microsoft.Orleans.Persistence.DynamoDB

包括使用AWS DynamoDB存储Grain状态的插件。

### 流提供商

Microsoft.Orleans.OrleansServiceBus

包括Azure事件中心的流提供程序。

Microsoft.Orleans.Streaming.AzureStorage

包括Azure队列的流提供程序。

Microsoft.Orleans.Streaming.SQS

包括AWS SQS服务的流提供程序。

Microsoft.Orleans.OrleansGCPUtils

包括GCP PubSub服务的流提供程序。

### 附加包

Microsoft.Orleans.OrleansCodeGenerator

包括运行时代码生成器。

Microsoft.Orleans.EventSourcing

包含一组基类型，用于创建具有事件源状态的Grain类。

### 开发和测试

Microsoft.Orleans.OrleansProviders

包含一组持久性和流提供程序，用于将数据保存在内存中。用于测试。一般情况下，不推荐用于生产用途，除非丢失数据丢失是可以接受的。

Microsoft.Orleans.TestingHost

包括用于在测试项目中托管Silo和客户端的库。

### 串行器

Microsoft.Orleans.Serialization.Bond

包括对Bond序列化程序的支持。

Microsoft.Orleans.OrleansGoogleUtils

包括Google Protocol Buffers序列化程序。

Microsoft.Orleans.ProtobufNet

包括protobuf-net版本的Protocol Buffers序列化程序。

### 遥测

Microsoft.Orleans.OrleansTelemetryConsumers.Counters

Windows性能计数器实现了Orleans遥测API。

Microsoft.Orleans.OrleansTelemetryConsumers.AI

包括Azure Application Insights的遥测消费者。

Microsoft.Orleans.OrleansTelemetryConsumers.NewRelic

包括NewRelic的遥测消费者。

### 工具

Microsoft.Orleans.CounterControl

包括OrleansCounterControl.exe，它为Orleans统计信息和已部署的grain类注册Windows性能计数器类别。需要提升。可以在Azure中作为角色启动任务的一部分执行。

### 交易

Microsoft.Orleans.Transactions

包括对跨Grain交易（beta）的支持。

Microsoft.Orleans.Transactions.AzureStorage

包含用于在Azure Table（beta）中保留事务日志的插件。

## 定义Grain

定义Grain时要引入

Microsoft.Orleans.OrleansCodeGenerator.Build

定义只有返回值为Task方法的自定义Grain接口，同时自定义Grain类继承这个接口和Grain类

interface IMyGrain

{

Task Test();

}

class MyGrain:Grain,IMyGrain

{

public Task Test()

{

}  
}

## Grain标识

在分布式系统中，对象引用不能表示实例标识，因为引用通常仅限于单个地址空间。.NET引用肯定就是这种情况。此外，Grain必须具有同一性，无论它是否活跃，因此我们可以根据需要激活Grain。因此Grain有一个主键。主键可以是全局唯一标识符（GUID），长整数或字符串。

主键的范围是Grain类型。因此，Grain的完整身份是由Grain的类型及其关键形成的。

Grain的调用者决定应该使用哪种方案。选项是：

* long
* GUID
* string
* GUID +string
* long+string

在客户端代码中通过GUID引用粒度：

var grain = grainFactory.GetGrain<IExample>(Guid.NewGuid());

从Grain代码中检索主键：

public override Task OnActivateAsync()

{

Guid primaryKey = this.GetPrimaryKey();

return base.OnActivateAsync();

}

## 计时器和提醒

Orleans运行时提供了两种机制，称为计时器和提醒，使开发人员能够指定Grain的周期性行为。

### 计时器

#### 定时器说明

**定时器**用于创建周期性的Grain行为，不需要跨越多个激活（Grain的实例化）。它基本上与标准相同。**NET System.Threading.Timer**类。此外，它在运行的Grain激活中受到单线程执行保证。

每次激活可以具有与其相关联的零个或多个计时器。运行时在与其关联的激活的运行时上下文中执行每个计时器例程。

#### 定时器使用

要启动计时器，请使用**Grain.RegisterTimer**方法，该方法返回 **IDisposable**引用：

public IDisposable RegisterTimer(

Func<object, Task> asyncCallback, // function invoked when the timer ticks

object state, // object tp pass to asyncCallback

TimeSpan dueTime, // time to wait before the first timer tick

TimeSpan period) // the period of the timer

通过处理取消计时器。

如果激活被禁用或发生故障并且其筒仓崩溃，则计时器将停止触发。

重要考虑因素

* 启用激活收集后，执行计时器回调不会将激活状态从空闲状态更改为正在使用状态。这意味着计时器不能用于推迟停用其他空闲激活。
* 传递给**Grain.RegisterTimer**的时间段是从解析**asyncCallback**返回的任务到下一次调用**asyncCallback**的时刻所经过的时间。这不仅使得对**asyncCallback的**连续调用不可能重叠，而且还使得**asyncCallback**完成所需的时间长度影响调用**asyncCallback**的频率。这是与**System.Threading.Timer**的语义的重要偏差。
* 每次调用**asyncCallback**都会在单独的转弯中传递给激活，并且永远不会与同一激活的其他转弯同时运行。但请注意，**asyncCallback**调用不作为消息传递，因此不受消息交错语义的影响。这意味着应该认为**asyncCallback的**调用行为就像在相对于该粒度的其他消息的可重入粒度上运行一样。

### 提醒

#### 提醒说明

提醒与计时器类似，但有一些重要区别：

* 除非明确取消，否则提醒是持久性的，并且将在几乎所有情况下（包括部分或完全群集重新启动）继续触发。
* 提醒“定义”被写入存储。但是，具体时间的每个特定事件都不是。这具有副作用，即如果群集在特定提醒滴答时完全关闭，则将错过它并且仅发生提醒的下一个滴答。
* 提醒与Grain相关，而不是任何特定的活化。
* 如果Grain没有与之关联的激活并且提醒滴答，则将创建一个Grain。例如：如果激活变为空闲且被停用，则与相同Grain相关联的提醒将在下一个滴答时重新激活Grain。
* 提醒通过消息传递，并且与所有其他Grain方法具有相同的交错语义。
* 提醒不应用于高频计时器 - 它们的周期应以分钟，小时或天来衡量。

### 组态

持久的提醒依靠存储来运作。您必须在提醒子系统运行之前指定要使用的存储后备。这是通过UseXReminderService扩展方法配置其中一个提醒提供程序来完成的，其中X是提供程序的名称，例如UseAzureTableReminderService。

Azure表配置：

// TODO replace with your connection string

const string connectionString = "YOUR\_CONNECTION\_STRING\_HERE";

var silo = new SiloHostBuilder()

[...]

.UseAzureTableReminderService(options => options.ConnectionString = connectionString)

[...]

SQL：

// TODO replace with your connection string

const string connectionString = "YOUR\_CONNECTION\_STRING\_HERE";

const string invariant = "YOUR\_INVARIANT";

var silo = new SiloHostBuilder()

[...]

.UseAdoNetReminderService(options =>

{

options.ConnectionString = connectionString;

options.Invariant = invariant;

})

[...]

如果您只是想要一个占位符实现提醒而无需设置Azure帐户或SQL数据库，那么这将为您提供一个仅限开发的提醒系统实现：

var silo = new SiloHostBuilder()

[...]

.UseInMemoryReminderService()

[...]

### 提**醒用法**

使用提醒的Grain必须实现**IRemindable.RecieveReminder**方法。

Task IRemindable.ReceiveReminder(string reminderName, TickStatus status)

{

Console.WriteLine("Thanks for reminding me-- I almost forgot!");

return TaskDone.Done;

}

要启动提醒，请使用**Grain.RegisterOrUpdateReminder**方法，该方法返回**IOrleansReminder**对象：

protected Task<IOrleansReminder> RegisterOrUpdateReminder(string reminderName, TimeSpan dueTime, TimeSpan period)

* reminderName是一个字符串，必须唯一地标识上下文粒度范围内的提醒。
* dueTime指定在发出第一个计时器滴答之前等待的时间量。
* period指定计时器的周期。

由于提醒在任何单次激活的生命周期中都存在，因此必须明确取消它们（而不是被处置）。您通过调用**Grain.UnregisterReminder**取消提醒：

protected Task UnregisterReminder(IOrleansReminder reminder)

提醒是**Grain.RegisterOrUpdateReminder**返回的句柄对象。

**IOrleansReminder的**实例不保证在激活的生命周期之外有效。如果您希望以持续的方式识别提醒，请使用包含提醒名称的字符串。

如果您只有提醒名称并且需要相应的**IOrleansReminder**实例 ，请调用**Grain.GetReminder**方法：

protected Task<IOrleansReminder> GetReminder(string reminderName)

## 观察者

在某些情况下，简单的消息/响应模式是不够的，客户端需要接收异步通知。例如，用户可能希望在朋友发布新的即时消息时得到通知。

客户端观察者是一种允许异步通知客户端的机制。观察者是一个继承自的单向异步接口，IGrainObserver其所有方法都必须是无效的。Grain通过像Grain接口方法一样调用它向观察者发送通知，除了它没有返回值，因此Grain不需要依赖于结果。Orleans运行时将确保单向传递通知。发布此类通知的Grain应提供添加或删除观察者的API。此外，通常可以方便地公开允许取消现有订阅的方法。Grain开发者可以使用OrleansObserverSubscriptionManager<T>通用类来简化观察到的Grain类型的开发。

要订阅通知，客户端必须首先创建一个实现观察者接口的本地C＃对象。然后，它在观察器工厂调用静态方法CreateObjectReference()，将C＃对象转换为Grain引用，然后可以将其传递给通知粒度上的预订方法。

其他Grain也可以使用此模型来接收异步通知。与客户端订阅案例不同，订阅粒度只是将观察者接口实现为构面，并将引用传递给自身（例如this.AsReference<IMyGrainObserverInterface>）。

### 代码示例

让我们假设我们有一个定期向客户端发送消息的Grain。为简单起见，我们示例中的消息将是一个字符串。我们首先在客户端上定义将接收消息的接口。

界面看起来像这样

public interface IChat : IGrainObserver

{

void ReceiveMessage(string message);

}

唯一特别的事情是接口应该继承IGrainObserver。现在任何想要观察这些消息的客户端都应该实现一个实现的类IChat。

最简单的情况是这样的：

public class Chat : IChat

{

public void ReceiveMessage(string message)

{

Console.WriteLine(message);

}

}

现在在服务器上我们应该有一个Grain将这些聊天消息发送给客户端。Grain还应该有一种机制让客户订阅和取消订阅以接收通知。对于订阅，Grain可以使用实用程序类ObserverSubscriptionManager。OrleansException如果您尝试订阅已订阅的观察者（或取消订阅未订阅的观察者），则此类会抛出一个，因此通过使用该IsSubscribed()方法或处理以下内容来处理此情况非常重要OrleansException：

class HelloGrain : Grain, IHello

{

private ObserverSubscriptionManager<IChat> \_subsManager;

public override async Task OnActivateAsync()

{

// We created the utility at activation time.

\_subsManager = new ObserverSubscriptionManager<IChat>();

await base.OnActivateAsync();

}

// Clients call this to subscribe.

public Task Subscribe(IChat observer)

{

if (!\_subsManager.IsSubscribed(observer))

{

\_subsManager.Subscribe(observer);

}

return Task.CompletedTask;

}

//Also clients use this to unsubscribe themselves to no longer receive the messages.

public Task UnSubscribe(IChat observer)

{

if (\_subsManager.IsSubscribed(observer))

{

\_subsManager.Unsubscribe(observer);

}

return Task.CompletedTask;

}

}

要将消息发送到客户端，可以使用实例的Notify方法ObserverSubscriptionManager<IChat>。该方法采用Action<T>方法或lambda表达式（此处T为类型IChat）。您可以调用接口上的任何方法将其发送给客户端。在我们的例子中，我们只有一个方法ReceiveMessage，我们在服务器上的发送代码如下所示：

public Task SendUpdateMessage(string message)

{

\_subsManager.Notify(s => s.ReceiveMessage(message));

return Task.CompletedTask;

}

现在我们的服务器有一个方法向观察者客户端发送消息，两种方法用于订阅/取消订阅，客户端实现了一个类，以便能够观察Grain消息。最后一步是使用我们之前实现的Chat类在客户端上创建观察者引用，并让它在订阅后接收消息。

代码如下所示：

//First create the grain reference

var friend = GrainClient.GrainFactory.GetGrain<IHello>(0);

Chat c = new Chat();

//Create a reference for chat usable for subscribing to the observable grain.

var obj = await GrainClient.GrainFactory.CreateObjectReference<IChat>(c);

//Subscribe the instance to receive messages.

await friend.Subscribe(obj);

现在，只要服务器上的Grain调用该SendUpdateMessage方法，所有订阅的客户端都将收到该消息。在我们的客户端代码中，Chat变量中的实例c将接收消息并将其输出到控制台。

## 重入

Grain激活是单线程的，默认情况下，在下一个请求开始处理之前，从开始到完成处理每个请求。在某些情况下，当一个请求等待异步操作完成时，可能需要激活处理其他请求。由于这个原因和其他原因，Orleans为开发人员提供了对请求交错行为的一些控制。在以下情况下，可能会交错多个请求：

* Grain类标记为 [Reentrant]
* 接口方法标记为 [AlwaysInterleave]
* 同一个调用链中的请求
* grain的MayInterleave谓词返回true

以下各节将讨论每种情况。

### 折返Grain

Grain可以用[Reentrant]属性标记实现类，以指示可以自由地交错不同的请求。

换句话说，重入激活可以在先前请求尚未完成处理时开始执行另一请求。执行仍然限于单个线程，因此激活仍然一次执行一个回合，并且每个回合仅代表激活的一个请求执行。

重入的Grain代码永远不会并行运行多个Grain代码（Grain代码的执行将始终是单线程的），但是重入的Grain**可能会**看到不同请求交错的代码执行。也就是说，来自不同请求的延续转换可以交错。

例如，使用下面的伪代码，当Foo和Bar是同一Grain类的2个方法时：

Task Foo()

{

await task1; // line 1

return Do2(); // line 2

}

Task Bar()

{

await task2; // line 3

return Do2(); // line 4

}

如果标记[Reentrant]了这个Grain，则Foo和Bar的执行可能会交错。

例如，以下执行顺序是可能的：

第1行，第3行，第2行和第4行。即，来自不同请求的转弯交错。

如果Grain不是可重入的，则唯一可能的执行是：第1行，第2行，第3行，第4行OR：第3行，第4行，第1行，第2行（新请求无法在上一个完成之前开始）。

在折返和非折返Grain之间进行选择的主要权衡是使交错工作正确的代码复杂性以及推理它的难度。

在一个微不足道的情况下，当Grain是无状态的并且逻辑简单时，更少（但不是太少，以便使用所有硬件线），重入Grain应该通常稍微更有效。

如果代码更复杂，那么即使整体效率稍差一些，大量的非重入Grain也应该可以避免很多问题，从而解决非显而易见的交错问题。

最后的答案取决于应用程序的具体情况。

### 交错方法

[AlwaysInterleave]无论Grain是否是可重入的，标记的Grain界面方法都将交错。请考虑以下示例：

public interface ISlowpokeGrain : IGrainWithIntegerKey

{

Task GoSlow();

[AlwaysInterleave]

Task GoFast();

}

public class SlowpokeGrain : Grain, ISlowpokeGrain

{

public async Task GoSlow()

{

await Task.Delay(TimeSpan.FromSeconds(10));

}

public async Task GoFast()

{

await Task.Delay(TimeSpan.FromSeconds(10));

}

}

现在考虑以下客户端请求启动的呼叫流程：

var slowpoke = client.GetGrain<ISlowpokeGrain>(0);

// A) This will take around 20 seconds

await Task.WhenAll(slowpoke.GoSlow(), slowpoke.GoSlow());

// B) This will take around 10 seconds.

await Task.WhenAll(slowpoke.GoFast(), slowpoke.GoFast(), slowpoke.GoFast());

呼叫GoSlow不会交错，因此两次GoSlow()呼叫的执行大约需要20秒。另一方面，因为GoFast被标记[AlwaysInterleave]，对它的三次调用将同时执行并且将在大约10秒内完成，而不需要至少30秒完成。

### 呼叫链中的重入

为了避免死锁，调度程序允许在给定的调用链中重入。考虑下面两个具有相互递归方法的Grain的例子，IsEven并且IsOdd：

public interface IEvenGrain : IGrainWithIntegerKey

{

Task<bool> IsEven(int num);

}

public interface IOddGrain : IGrainWithIntegerKey

{

Task<bool> IsOdd(int num);

}

public class EvenGrain : Grain, IEvenGrain

{

public async Task<bool> IsEven(int num)

{

if (num == 0) return true;

var oddGrain = this.GrainFactory.GetGrain<IOddGrain>(0);

return await oddGrain.IsOdd(num - 1);

}

}

public class OddGrain : Grain, IOddGrain

{

public async Task<bool> IsOdd(int num)

{

if (num == 0) return false;

var evenGrain = this.GrainFactory.GetGrain<IEvenGrain>(0);

return await evenGrain.IsEven(num - 1);

}

}

现在考虑以下客户端请求启动的呼叫流程：

var evenGrain = client.GetGrain<IEvenGrain>(0);

await evenGrain.IsEven(2);

上面的代码调用IEvenGrain.IsEven(2)，调用IOddGrain.IsOdd(1)哪些调用IEvenGrain.IsEven(0)，它将true调用链返回给客户端。如果没有调用链重入，上面的代码将在IOddGrain调用时导致死锁IEvenGrain.IsEven(0)。但是，通过呼叫链重入，允许呼叫继续进行，因为它被认为是开发者的意图。

可以通过设置SchedulingOptions.AllowCallChainReentrancy为禁用此行为false。例如：

siloHostBuilder.Configure<SchedulingOptions>(

options => options.AllowCallChainReentrancy = false);

### 使用谓词重入

Grain类可以通过检查请求来指定用于在逐个呼叫的基础上确定交织的谓词。该[MayInterleave(string methodName)]属性提供此功能。该属性的参数是grain类中的静态方法的名称，该方法接受一个InvokeMethodRequest对象并返回一个bool指示是否应该交错的请求。

这是一个允许交换的示例，如果请求参数类型具有以下[Interleave]属性：

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]

public sealed class InterleaveAttribute : Attribute { }

// Specify the may-interleave predicate.

[MayInterleave(nameof(ArgHasInterleaveAttribute))]

public class MyGrain : Grain, IMyGrain

{

public static bool ArgHasInterleaveAttribute(InvokeMethodRequest req)

{

// Returning true indicates that this call should be interleaved with other calls.

// Returning false indicates the opposite.

return req.Arguments.Length == 1

&& req.Arguments[0]?.GetType().GetCustomAttribute<InterleaveAttribute>() != null;

}

public Task Process(object payload)

{

// Process the object.

}

}

## 持久化

### 粮食持久性目标

1. 允许不同的粒度类型使用不同类型的存储提供程序（例如，一个使用Azure表，一个使用ADO.NET）或相同类型的存储提供程序但具有不同的配置（例如，两者都使用Azure表，但一个使用存储帐户＃1和一个使用存储帐户＃2）
2. 允许交换存储提供程序实例的配置（例如，Dev-Test-Prod），只需更改配置文件，无需更改代码。
3. 提供一个框架，以便以后可以由Orleans团队或其他人编写其他存储提供程序。
4. 提供最小的生产级存储提供商
5. 存储提供商可以完全控制如何在持久性后备存储中存储Grain状态数据。推论：Orleans没有提供全面的ORM存储解决方案，但允许自定义存储提供商在需要时支持特定的ORM要求。

### Grain Persistence API

Grain类型可以通过以下两种方式之一声明：

* Grain如果它们没有任何持久状态，或者它们将自己处理所有持久状态，则扩展，或者
* Grain<T>如果它们具有他们希望Orleans运行时处理的持久状态，则扩展。换句话说，通过扩展Grain<T>grain类型自动选择加入Orleans系统管理的持久性框架。

对于本节的其余部分，我们将仅考虑选项＃2 / Grain<T>因为选项＃1粒将继续像现在一样运行而不会有任何行为更改。

### 粮食储存

继承的Grain类Grain<T>（其中T是需要持久化的特定于应用程序的状态数据类型）将从指定的存储中自动加载其状态。

粒度将标记一个[StorageProvider]属性，该属性指定用于读取/写入此粒度的状态数据的存储提供程序的命名实例。

[StorageProvider(ProviderName="store1")]

public class MyGrain<MyGrainState> ...

{

...

}

Orleans框架提供了一种机制来指定和注册不同的存储提供程序并使用它们进行配置ISiloHostBuilder，

var silo = new SiloHostBuilder()

.AddMemoryGrainStorage("DevStore")

.AddAzureTableGrainStorage("store1", options => options.ConnectionString = "DefaultEndpointsProtocol=https;AccountName=data1;AccountKey=SOMETHING1")

.AddAzureBlobGrainStorage("store2", options => options.ConnectionString = "DefaultEndpointsProtocol=https;AccountName=data2;AccountKey=SOMETHING2")

.Build();

### 配置IGrainStorage提供程序

Orleans本身支持一系列IGrainStorage实现，您可以将它们用于存储Grain状态的应用程序。在本节中，我们将在如何配置AzureTableGrainStorage，AzureBlobGrainStorage，DynamoDBGrainStorage，MemoryGrainStorage，和AdoNetGrainStorage在一个筒仓。其他IGrainStorage提供商的配置类似。

#### AzureTableGrainStorage提供程序

var silo = new SiloHostBuilder()

.AddAzureTableGrainStorage("TableStore", options => options.ConnectionString = "UseDevelopmentStorage=true")

...

.Build();

以下设置可用于配置AzureTableGrainStorage提供程序AzureTableGrainStorageOptions：

/// <summary>

/// Configuration for AzureTableGrainStorage

/// </summary>

public class AzureTableStorageOptions

{

/// <summary>

/// Azure table connection string

/// </summary>

[RedactConnectionString]

public string ConnectionString { get; set; }

/// <summary>

/// Table name where grain stage is stored

/// </summary>

public string TableName { get; set; } = DEFAULT\_TABLE\_NAME;

public const string DEFAULT\_TABLE\_NAME = "OrleansGrainState";

/// <summary>

/// Indicates if grain data should be deleted or reset to defaults when a grain clears it's state.

/// </summary>

public bool DeleteStateOnClear { get; set; } = false;

/// <summary>

/// Stage of silo lifecycle where storage should be initialized. Storage must be initialzed prior to use.

/// </summary>

public int InitStage { get; set; } = DEFAULT\_INIT\_STAGE;

public const int DEFAULT\_INIT\_STAGE = ServiceLifecycleStage.ApplicationServices;

#region json serialization

public bool UseJson { get; set; }

public bool UseFullAssemblyNames { get; set; }

public bool IndentJson { get; set; }

public TypeNameHandling? TypeNameHandling { get; set; }

#endregion json serialization

}

**注意：**状态大小不应超过64KB，这是Azure Table Storage强加的限制。

#### AzureBlobGrainStorage提供程序

var silo = new SiloHostBuilder()

.AddAzureBlobGrainStorage("BlobStore", options => options.ConnectionString = "UseDevelopmentStorage=true")

...

.Build();

以下设置可用于配置AzureBlobGrainStorage提供程序AzureBlobStorageOptions：

public class AzureBlobStorageOptions

{

/// <summary>

/// Azure connection string

/// </summary>

[RedactConnectionString]

public string ConnectionString { get; set; }

/// <summary>

/// Container name where grain stage is stored

/// </summary>

public string ContainerName { get; set; } = DEFAULT\_CONTAINER\_NAME;

public const string DEFAULT\_CONTAINER\_NAME = "grainstate";

/// <summary>

/// Stage of silo lifecycle where storage should be initialized. Storage must be initialzed prior to use.

/// </summary>

public int InitStage { get; set; } = DEFAULT\_INIT\_STAGE;

public const int DEFAULT\_INIT\_STAGE = ServiceLifecycleStage.ApplicationServices;

#region json serialization

public bool UseJson { get; set; }

public bool UseFullAssemblyNames { get; set; }

public bool IndentJson { get; set; }

public TypeNameHandling? TypeNameHandling { get; set; }

#endregion json serialization

}

#### DynamoDBGrainStorage提供程序

var silo = new SiloHostBuilder()

.AddDynamoDBGrainStorage("DDBStore", options =>

{

options.AccessKey = "MY\_ACCESS\_KEY";

options.SecretKey = "MY\_SECRET\_KEY";

options.Service = "us-wes-1";

})

...

.Build();

以下设置可用于配置DynamoDBGrainStorage提供程序DynamoDBStorageOptions：

public class DynamoDBStorageOptions

{

/// <summary>

/// Gets or sets a unique identifier for this service, which should survive deployment and redeployment.

/// </summary>

public string ServiceId { get; set; } = string.Empty;

/// <summary>

/// AccessKey string for DynamoDB Storage

/// </summary>

[Redact]

public string AccessKey { get; set; }

/// <summary>

/// Secret key for DynamoDB storage

/// </summary>

[Redact]

public string SecretKey { get; set; }

/// <summary>

/// DynamoDB Service name

/// </summary>

public string Service { get; set; }

/// <summary>

/// Read capacity unit for DynamoDB storage

/// </summary>

public int ReadCapacityUnits { get; set; } = DynamoDBStorage.DefaultReadCapacityUnits;

/// <summary>

/// Write capacity unit for DynamoDB storage

/// </summary>

public int WriteCapacityUnits { get; set; } = DynamoDBStorage.DefaultWriteCapacityUnits;

/// <summary>

/// DynamoDB table name.

/// Defaults to 'OrleansGrainState'.

/// </summary>

public string TableName { get; set; } = "OrleansGrainState";

/// <summary>

/// Indicates if grain data should be deleted or reset to defaults when a grain clears it's state.

/// </summary>

public bool DeleteStateOnClear { get; set; } = false;

/// <summary>

/// Stage of silo lifecycle where storage should be initialized. Storage must be initialzed prior to use.

/// </summary>

public int InitStage { get; set; } = DEFAULT\_INIT\_STAGE;

public const int DEFAULT\_INIT\_STAGE = ServiceLifecycleStage.ApplicationServices;

#region JSON Serialization

public bool UseJson { get; set; }

public bool UseFullAssemblyNames { get; set; }

public bool IndentJson { get; set; }

public TypeNameHandling? TypeNameHandling { get; set; }

#endregion

}

#### ADO.NET粮食存储提供商

ADO .NET Grain Storage提供程序允许您在关系数据库中存储grain状态。目前支持以下数据库：

* SQL Server
* MySQL的/ MariaDB的
* PostgreSQL的
* 神谕

首先，安装基础包：

Install-Package Microsoft.Orleans.Persistence.AdoNet

在项目的nuget包上还原后，您将找到支持的数据库供应商的不同SQL脚本，这些脚本将复制到项目目录\ OrleansAdoNetContent，其中每个受支持的ADO.NET扩展都有自己的目录。您也可以从中获取它们该[Orleans.Persistence.AdoNet库](https://github.com/dotnet/orleans/tree/master/src/AdoNet/Orleans.Persistence.AdoNet)。创建数据库，然后运行相应的脚本来创建表。

接下来的步骤是安装特定于所需数据库供应商的第二个NuGet包（参见下表），并以编程方式或通过XML配置配置存储提供程序。

| **数据库** | **脚本** | **NuGet包** | **AdoInvariant** | **备注** |
| --- | --- | --- | --- | --- |
| SQL Server | [SQLServer的-Persistence.sql](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/SQLServer-Persistence.sql) | [System.Data.SqlClient的](https://www.nuget.org/packages/System.Data.SqlClient/) | System.Data.SqlClient的 |  |
| MySQL / MariaDB | [在MySQL-Persistence.sql](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/MySQL-Persistence.sql) | [MySql.Data](https://www.nuget.org/packages/MySql.Data/) | MySql.Data.MySqlClient |  |
| PostgreSQL的 | [PostgreSQL相关Persistence.sql](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/PostgreSQL-Persistence.sql) | [Npgsql的](https://www.nuget.org/packages/Npgsql/) | Npgsql的 |  |
| 神谕 | [甲骨文Persistence.sql](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/Oracle-Persistence.sql) | [ODP.net](https://www.nuget.org/packages/Oracle.ManagedDataAccess/) | Oracle.DataAccess.Client | 没有.net核心支持 |

以下是如何通过以下方式配置ADO.NET存储提供程序的示例ISiloHostBuilder：

var siloHostBuilder = new SiloHostBuilder()

.AddAdoNetGrainStorage("OrleansStorage", options=>

{

options.Invariant = "<Invariant>";

options.ConnectionString = "<ConnectionString>";

options.UseJsonFormat = true;

});

实质上，您只需要设置特定于数据库供应商的连接字符串和Invariant标识供应商的 （参见上表）。您还可以选择保存数据的格式，可以是二进制（默认），JSON或XML。虽然二进制是最紧凑的选项，但它是不透明的，您将无法读取或处理数据。建议使用JSON。

您可以通过以下方式设置以下属性AdoNetGrainStorageOptions：

/// <summary>

/// Options for AdonetGrainStorage

/// </summary>

public class AdoNetGrainStorageOptions

{

/// <summary>

/// Connection string for AdoNet storage.

/// </summary>

[Redact]

public string ConnectionString { get; set; }

/// <summary>

/// Stage of silo lifecycle where storage should be initialized. Storage must be initialzed prior to use.

/// </summary>

public int InitStage { get; set; } = DEFAULT\_INIT\_STAGE;

/// <summary>

/// Default init stage in silo lifecycle.

/// </summary>

public const int DEFAULT\_INIT\_STAGE = ServiceLifecycleStage.ApplicationServices;

/// <summary>

/// The default ADO.NET invariant used for storage if none is given.

/// </summary>

public const string DEFAULT\_ADONET\_INVARIANT = AdoNetInvariants.InvariantNameSqlServer;

/// <summary>

/// The invariant name for storage.

/// </summary>

public string Invariant { get; set; } = DEFAULT\_ADONET\_INVARIANT;

#region json serialization related settings

/// <summary>

/// Whether storage string payload should be formatted in JSON.

/// <remarks>If neither <see cref="UseJsonFormat"/> nor <see cref="UseXmlFormat"/> is set to true, then BinaryFormatSerializer will be configured to format storage string payload.</remarks>

/// </summary>

public bool UseJsonFormat { get; set; }

public bool UseFullAssemblyNames { get; set; }

public bool IndentJson { get; set; }

public TypeNameHandling? TypeNameHandling { get; set; }

#endregion

/// <summary>

/// Whether storage string payload should be formatted in Xml.

/// <remarks>If neither <see cref="UseJsonFormat"/> nor <see cref="UseXmlFormat"/> is set to true, then BinaryFormatSerializer will be configured to format storage string payload.</remarks>

/// </summary>

public bool UseXmlFormat { get; set; }

}

ADO.NET持久性具有版本数据的功能，并使用任意应用程序规则和流定义任意（反）序列化程序，但目前没有方法将它们暴露给应用程序代码。[ADO.NET Persistence Rationale中的](https://dotnet.github.io/orleans/Documentation/grains/grain_persistence/index.html#ADONETPersistenceRationale)更多信息。

#### MemoryGrainStorage

MemoryGrainStorage是一个简单的Grain存储实现，它实际上并没有使用下面的持久数据存储。学习快速使用Grain Storages很方便，但不打算在生产场景中使用。

**注意：**此提供程序将状态保持为易失性内存，该内存将在silo关闭时被删除。仅用于测试。

以下是如何通过设置内存存储提供程序 ISiloHostBuilder

var siloHostBuilder = new SiloHostBuilder()

.AddMemoryGrainStorage("OrleansStorage", options=>options.NumStorageGrains = 10);

您可以通过以下方式设置以下配置属性 MemoryGrainStorageOptions

/// <summary>

/// Options for MemoryGrainStorage

/// </summary>

public class MemoryGrainStorageOptions

{

/// <summary>

/// Default number of queue storage grains.

/// </summary>

public const int NumStorageGrainsDefaultValue = 10;

/// <summary>

/// Number of store grains to use.

/// </summary>

public int NumStorageGrains { get; set; } = NumStorageGrainsDefaultValue;

/// <summary>

/// Stage of silo lifecycle where storage should be initialized. Storage must be initialzed prior to use.

/// </summary>

public int InitStage { get; set; } = DEFAULT\_INIT\_STAGE;

/// <summary>

/// Default init stage

/// </summary>

public const int DEFAULT\_INIT\_STAGE = ServiceLifecycleStage.ApplicationServices;

}

### 存储提供商的注意事项

如果没有[StorageProvider]为Grain<T>grain类指定属性，则将Default搜索名为的提供程序。如果未找到，则将其视为缺少存储提供程序。

如果粒度类引用的存储提供程序未在配置时添加到silo中，则该类型的粒度将无法在运行时激活，并且对它们的调用将失败，Orleans.Storage.BadProviderConfigException指出未加载粒度类型的错误。但其余的Grain类型不会受到影响。

不同的粒度类型可以使用不同的已配置存储提供程序，即使两者都是相同的类型：例如，两个不同的Azure表存储提供程序实例，连接到不同的Azure存储帐户（请参阅上面的配置文件示例）。

存储提供程序的所有配置详细信息都是通过ISiloHostBuilder定义的。目前没有提供动态更新或更改筒仓使用的存储提供程序列表的机制。但是，这是优先级/工作负载约束而不是基本设计约束。

### State Persitence API

状态持久性API有两个部分：粒状态API和存储提供程序API。

#### Grain状态API

Orleans Runtime中的Grain状态存储功能将提供读取和写入操作，以自动填充/保存该GrainStateGrain的数据对象。在幕后，这些函数将连接（在Orleans client-gen工具生成的代码中）到为该粒度配置的相应持久性提供程序。

### Grain状态读/写功能

谷粒被激活时将自动读取Grain状态，但Grain负责在必要时明确触发任何改变的Grain状态的写入。有关错误处理机制的详细信息，请参阅下面的“ [故障模式”](https://dotnet.github.io/orleans/Documentation/grains/grain_persistence/index.html#FailureModes)部分

GrainState将在为该激活调用该方法之前自动读取（使用等效的base.ReadStateAsync()）。 在任何方法调用该粒度之前不会刷新，除非为此调用激活了粒度。OnActivateAsync()GrainState

在任何Grain方法调用期间，Grain可以请求Orleans运行时通过调用将该激活的当前Grain状态数据写入指定的存储提供者base.WriteStateAsync()。当粒子对其状态数据进行重大更新时，粒子负责显式执行写操作。最常见的是，grain方法将返回从该grain方法返回base.WriteStateAsync() Task的最终结果Task，但不需要遵循此模式。在任何Grain方法之后，运行时不会自动更新存储的Grain状态。

在粒度中的任何粒度方法或计时器回调处理程序期间，粒度可以请求Orleans运行时通过调用从指定的存储提供程序重新读取当前粒度数据以进行该激活base.ReadStateAsync()。这将使用从持久存储中读取的最新值完全覆盖当前存储在grain状态对象中的任何当前状态数据。

一个不透明的提供者特定的Etag值（string）可以由存储提供商当读取状态填充晶状态元数据的一部分进行设定。某些提供商可能会选择将其保留为null不使用Etags。

从概念上讲，Orleans Runtime将在任何写入操作期间对粒度状态数据对象进行深层复制以供自己使用。在幕后，运行时可以使用优化规则和启发式来避免在某些情况下执行部分或全部深层复制，前提是保留了预期的逻辑隔离语义。

### Grain状态读/写操作的示例代码

Grain必须扩展Grain<T>该类才能参与OrleansGrain状态持久性机制。在T上述定义将由该Grain应用专用晶态类取代; 见下面的例子。

grain类还应该使用一个[StorageProvider]属性进行注释，该属性告诉运行时哪个存储提供程序（实例）与此类型的粒子一起使用。

public class MyGrainState

{

public int Field1 { get; set; }

public string Field2 { get; set; }

}

[StorageProvider(ProviderName="store1")]

public class MyPersistenceGrain : Grain<MyGrainState>, IMyPersistenceGrain

{

...

}

### 粮食国家库

在OnActivateAsync()调用grain 方法之前，Orleans运行时会自动初始读取粒状态; 没有应用程序代码来实现这一点。从那时起，粮食的状态将通过Grain<T>.State粮食类内的财产获得。

#### 粮食国家写

对谷粒的内存状态进行任何适当的更改后，Grain应该调用base.WriteStateAsync()方法，通过定义的存储提供程序将更改写入持久存储，以获得此Grain类型。此方法是异步的，并返回一个Task通常由grain方法返回的方法作为其自己的完成任务。

public Task DoWrite(int val)

{

State.Field1 = val;

return base.WriteStateAsync();

}

#### Grain状态刷新

如果Grain希望从后备商店明确重新读取此Grain的最新状态，Grain应该调用该base.ReadStateAsync()方法。这将通过此粒度类型的已定义存储提供程序从持久存储重新加载粒状态，并且在ReadStateAsync() Task完成时将覆盖并替换粒度状态的任何先前的内存中副本。

public async Task<int> DoRead()

{

await base.ReadStateAsync();

return State.Field1;

}

### Grain状态持久性操作的失败模式

#### Grain状态读取操作的故障模式

在初始读取该特定Grain的状态数据期间，存储提供者返回的失败将导致该Grain的激活操作失败; 在这种情况下，不会调用该粒子的OnActivateAsync()生命周期回调方法。引起激活的原始请求将以与Grain激活期间的任何其他故障相同的方式被故障返回给调用者。存储提供程序在读取特定Grain的状态数据时遇到的故障将导致ReadStateAsync() Task出现故障。Grain可以选择处理或忽略故障Task，就像TaskOrleans的其他任何一样。

由于缺少/错误的存储提供程序配置而导致无法在silo启动时加载粮食的任何尝试都将返回永久性错误Orleans.BadProviderConfigException。

#### Grain状态写入操作的故障模式

存储提供程序为特定Grain写入状态数据时遇到的故障将导致WriteStateAsync() Task出现故障。通常，这意味着如果将Grain调用WriteStateAsync() Task正确地链接Task到此Grain方法的最终返回，则Grain调用将被故障返回给客户端调用者。但是，某些高级方案可能会编写Grain代码来专门处理此类写入错误，就像它们可以处理任何其他错误一样Task。

执行错误处理/恢复代码的粒度必须捕获异常/故障WriteStateAsync() Tasks而不是重新抛出以表示它们已成功处理写入错误。

### 存储提供程序API

有一个服务提供程序API用于编写其他持久性提供程序 - IGrainStorage。

Persistence Provider API涵盖了GrainState数据的读写操作。

/// <summary>

/// Interface to be implemented for a storage able to read and write Orleans grain state data.

/// </summary>

public interface IGrainStorage

{

/// <summary>Read data function for this storage instance.</summary>

/// <param name="grainType">Type of this grain [fully qualified class name]</param>

/// <param name="grainReference">Grain reference object for this grain.</param>

/// <param name="grainState">State data object to be populated for this grain.</param>

/// <returns>Completion promise for the Read operation on the specified grain.</returns>

Task ReadStateAsync(string grainType, GrainReference grainReference, IGrainState grainState);

/// <summary>Write data function for this storage instance.</summary>

/// <param name="grainType">Type of this grain [fully qualified class name]</param>

/// <param name="grainReference">Grain reference object for this grain.</param>

/// <param name="grainState">State data object to be written for this grain.</param>

/// <returns>Completion promise for the Write operation on the specified grain.</returns>

Task WriteStateAsync(string grainType, GrainReference grainReference, IGrainState grainState);

/// <summary>Delete / Clear data function for this storage instance.</summary>

/// <param name="grainType">Type of this grain [fully qualified class name]</param>

/// <param name="grainReference">Grain reference object for this grain.</param>

/// <param name="grainState">Copy of last-known state data object for this grain.</param>

/// <returns>Completion promise for the Delete operation on the specified grain.</returns>

Task ClearStateAsync(string grainType, GrainReference grainReference, IGrainState grainState);

}

### 存储提供程序语义

当存储提供程序检测到Etag约束违规时，任何尝试执行写入操作都应该导致写入Task出现暂时性错误Orleans.InconsistentStateException并包装底层存储异常。

public class InconsistentStateException : AggregateException

{

/// <summary>The Etag value currently held in persistent storage.</summary>

public string StoredEtag { get; private set; }

/// <summary>The Etag value currently held in memory, and attempting to be updated.</summary>

public string CurrentEtag { get; private set; }

public InconsistentStateException(

string errorMsg,

string storedEtag,

string currentEtag,

Exception storageException

) : base(errorMsg, storageException)

{

this.StoredEtag = storedEtag;

this.CurrentEtag = currentEtag;

}

public InconsistentStateException(string storedEtag, string currentEtag, Exception storageException)

: this(storageException.Message, storedEtag, currentEtag, storageException)

{ }

}

写操作的任何其他故障条件都应该导致写入Task被包含底层存储异常的异常中断。

### 数据映射

各个存储提供商应该决定如何最好地存储Grain状态 - blob（各种格式/序列化形式）或每个字段列是明显的选择。

Azure Table的基本存储提供程序使用Orleans二进制序列化将状态数据字段编码为单个表列。

### ADO.NET持久性原理

ADO.NET支持的持久性存储的原则是：

1. 在数据，数据格式和代码的演变过程中，可以保护业务关键数据的安全。
2. 优先考虑供应商和存储特定功能。

实际上，这意味着遵循ADO.NET实现目标和ADO.NET特定存储提供程序中的一些添加的实现逻辑，这些逻辑允许改进存储中数据的形状。

除了通常的存储提供程序功能外，ADO.NET提供程序还具有内置功能

1. 在往返状态时，将存储数据格式从一种格式更改为另一种格式（例如，从JSON到二进制）。
2. 以任意方式塑造要保存或从存储中读取的类型。这有助于改进版本状态。
3. 从数据库中流出数据。

二者1.并2.可以在任意决定的参数，例如被应用于GrainID，粒型，有效载荷数据。

发生这种情况，以便选择一种格式，例如[简单二进制编码（SBE）](https://github.com/real-logic/simple-binary-encoding)并实现 [IStorageDeserializer](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/Storage/Provider/IStorageDeserializer.cs)和[IStorageSerializer](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/Storage/Provider/IStorageSerializer.cs)。内置（反）序列化器已使用此方法构建。该[OrleansStorageDefault（反）序列化](https://github.com/dotnet/orleans/tree/master/src/AdoNet/Orleans.Persistence.AdoNet/Storage/Provider) 可以用作如何实现其他格式的示例。

当（de）序列化器已经实现时，它们需要添加到[AdoNetGrainStorage中](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/Storage/Provider/AdoNetGrainStorage.cs)的StorageSerializationPicker属性中。这是[IStorageSerializationPicker](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/Storage/Provider/IStorageSerializationPicker.cs)的实现。默认情况下， 将使用[StorageSerializationPicker](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/Storage/Provider/StorageSerializationPicker.cs)。在[RelationalStorageTests中](https://github.com/dotnet/orleans/blob/master/test/TesterAdoNet/StorageTests/Relational/RelationalStorageTests.cs)可以看到更改数据存储格式或使用（de）序列化器的示例。

目前没有方法将此公开给Orleans应用程序消费，因为没有方法可以访问创建[AdoNetGrainStorage](https://github.com/dotnet/orleans/blob/master/src/AdoNet/Orleans.Persistence.AdoNet/Storage/Provider/AdoNetGrainStorage.cs)的框架。

## 事件溯源

事件采购提供了一种管理和维持Grain状态的灵活方式。事件来源的Grain比标准Grain具有许多潜在的优势。例如，它可以与许多不同的存储提供程序配置一起使用，并支持跨多个集群的地理复制。此外，它可以清晰地将Grain类与Grain状态（由Grain状态对象表示）和Grain更新（由事件对象表示）的定义分开。

### JournaledGrain基础知识

记录的Grain来自JournaledGrain<StateType,EventType>，具有以下类型参数：

* StateType代表Grain的状态类型。它必须是具有公共默认构造函数的类。
* EventType 是可以为这个Grain引发的所有事件的常见超类型，可以是任何类或接口。

所有状态和事件对象都应该是可序列化的（因为日志一致性提供程序可能需要保留它们，和/或将它们发送到通知消息中）。

对于事件为POCO（普通旧C＃对象）的Grain， JournaledGrain<StateType>可以用作简写JournaledGrain<StateType,Object>。

#### 读取Grain状态

要读取当前Grain状态并确定其版本号，JournaledGrain具有属性

GrainState State { get; }

int Version { get; }

版本号始终等于已确认事件的总数，并且状态是将所有已确认事件应用于初始状态的结果。初始状态具有版本0（因为没有应用任何事件），由GrainState类的默认构造函数确定。

重要说明：应用程序永远不应该直接修改返回的对象State。它仅供阅读。相反，当应用程序想要修改状态时，它必须通过引发事件间接地这样做。

#### 引发事件

通过调用RaiseEvent函数来完成事件的触发。例如，代表聊天的Grain可以引发一个PostedEvent表示用户提交帖子的内容：

RaiseEvent(new PostedEvent() { Guid = guid, User = user, Text = text, Timestamp = DateTime.UtcNow });

请注意，RaiseEvent启动对存储访问的写入，但不等待写入完成。对于许多应用程序，重要的是等到我们确认事件已被持久化。在这种情况下，我们总是通过等待ConfirmEvents：

RaiseEvent(new DepositTransaction() { DepositAmount = amount, Description = description });

await ConfirmEvents();

请注意，即使您没有明确调用ConfirmEvents，最终也会确认事件 - 它会在后台自动发生。

#### 状态过渡方法

每当引发事件时，运行时都会自动更新Grain状态。在引发事件后，应用程序无需显式更新状态。但是，应用程序仍然必须提供指定如何更新状态以响应事件的代码。这可以通过两种方式完成。

**（a）** GrainState类可以实现一个或多个Apply方法StateType。通常，会创建多个重载，并为事件的运行时类型选择最接近的匹配：

class GrainState {

Apply(E1 @event)

{

// code that updates the state

}

Apply(E2 @event)

{

// code that updates the state

}

}

**（b）** grain可以覆盖TransitionState函数：

protected override void TransitionState(State state, EventType @event)

{

// code that updates the state

}

假设转换方法除了修改状态对象之外没有副作用，并且应该是确定性的（否则，效果是不可预测的）。如果转换代码抛出异常，则捕获该异常并将其包含在由日志一致性提供程序发出的Orleans日志中的警告中。

确切地说，运行时调用转换方法取决于所选的日志一致性提供程序及其配置。除非日志一致性提供程序特别保证，否则最好不要依赖于特定时序的应用程序。

某些提供程序（如LogStorage日志一致性提供程序）会在每次加载Grain时重放事件序列。因此，只要仍然可以从存储中正确地反序列化事件对象，就可以从根本上修改GrainState类和转换方法。但对于其他提供程序（例如StateStorage日志一致性提供程序），只保留GrainState对象，因此开发人员必须确保从存储中读取时可以正确反序列化。

#### 引发多个事件

在调用ConfirmEvents之前，可以多次调用RaiseEvent：

RaiseEvent(e1);

RaiseEvent(e2);

await ConfirmEvents();

但是，这可能会导致两次连续的存储访问，并且在仅写入第一个事件后会产生Grain失败的风险。因此，通常使用一次性提高多个事件会更好

RaiseEvents(IEnumerable<EventType> events)

这保证了给定的事件序列以原子方式写入存储。请注意，由于版本号始终与事件序列的长度匹配，因此引发多个事件会使版本号一次增加一个以上。

#### 检索事件序列

基JournaledGrain类中的以下方法允许应用程序检索所有已确认事件序列的指定段：

Task<IReadOnlyList<EventType>> RetrieveConfirmedEvents(int fromVersion, int toVersion)

但是，并非所有日志一致性提供程序都支持它。如果不受支持，或者序列的指定段不再可用，NotSupportedException则抛出a。

要检索最新确认版本的所有事件，可以调用

await RetrieveConfirmedEvents(0, Version);

只能检索确认的事件：如果toVersion大于属性的当前值，则抛出异常Version。

由于确认的事件永远不会改变，即使存在多个实例或延迟确认，也无需担心比赛。但是，在这种情况下，属性值可能会在恢复时比调用Version时更大，因此建议将其值保存在变量中。另请参阅“并发保证”一节。awaitRetrieveConfirmedEvents

### ournaledGrain诊断

#### 监视连接错误

根据设计，日志一致性提供程序在连接错误（包括存储连接和集群之间的连接）下都具有弹性。但是只是容忍错误是不够的，因为应用程序通常需要监视任何此类问题，并且如果它们是严重的，则将它们引起操作员的注意。

JournaledGrain子类可以覆盖以下方法，以便在观察到连接错误时以及解决这些错误时接收通知：

protected override void OnConnectionIssue(ConnectionIssue issue)

{

/// handle the observed error described by issue

}

protected override void OnConnectionIssueResolved(ConnectionIssue issue)

{

/// handle the resolution of a previously reported issue

}

ConnectionIssue是一个抽象类，有几个常见字段描述问题，包括自上次连接成功以来观察了多少次。实际的连接问题类型由子类定义。连接问题被分类为类型，例如PrimaryOperationFailed或NotificationFailed，并且有时具有额外RemoteCluster缩小类别的额外键（例如）。

如果同一类别的问题多次发生（例如，我们不断获得NotificationFailed相同的目标RemoteCluster），则每次都会报告OnConnectionIssue。解决此类问题后（例如，我们最终成功向此发送通知RemoteCluster），然后OnConnectionIssueResolved调用一次，使用issue上次报告的相同对象OnConnectionIssue。独立类别的连接问题及其解决方案是独立报告的。

#### 简单统计

我们目前提供对基本统计的简单支持（将来，我们可能会用更标准的遥测机制取代它）。可以通过调用为JournaledGrain启用或禁用统计信息收集

void EnableStatsCollection()

void DisableStatsCollection()

可以通过调用检索统计信息

LogConsistencyStatistics GetStats()

### 立即确认

对于许多应用程序，我们希望确保立即确认事件，以便持久化版本不会落后于内存中的当前版本，并且如果Grain失败，我们不会冒失去最新状态的风险。我们可以遵循以下规则来保证：

1. 在grain方法返回之前RaiseEvent使用确认所有调用ConfirmEvents。
2. 确保RaiseConditionalEvent在grain方法返回之前完成返回的任务。
3. 避免 [Reentrant]或[AlwaysInterleave]属性，因此一次只能处理一个Grain调用。

如果我们遵循这些规则，则意味着在引发事件之后，在将事件写入存储之前，不会执行任何其他Grain代码。因此，无法观察内存中的版本与存储中的版本之间的不一致。虽然这通常正是我们想要的，但它也有一些潜在的缺点。

#### 潜在的缺点

* 如果与**远程集群或存储**的**连接暂时中断**，则Grain变得不可用：实际上，Grain在等待确认事件时无法执行任何代码，这可能需要无限期的时间（日志 -一致性协议保持重试，直到恢复存储连接）。
* 当处理**单个Grain实例的大量更新时，一次**确认一个**Grain实例**可能变得非常低效，即吞吐量很低。

### 延迟确认

为了提高上述情况下的可用性和吞吐量，Grain可以选择执行以下一项或两项操作：

* 允许Grain方法在不等待确认的情况下引发事件。
* 允许重入，所以Grain可以继续处理新的呼叫，即使之前的呼叫卡住等待确认。

这意味着在某些事件仍处于确认过程中时，可以执行Grain代码。该JournaledGrainAPI有一些特殊的规定，给予开发者在如何处理与当前“飞行中”未经证实的事件进行精确控制。

可以检查以下属性以找出当前未确认的事件：

IEnumerable<EventType> UnconfirmedEvents { get; }

此外，由于该State属性返回的状态不包括未确认事件的影响，因此存在替代属性

StateType TentativeState { get; }

返回“暂定”状态，通过应用所有未经证实的事件从“状态”获得。在所有未经证实的事件得到确认之后，暂定状态基本上是对可能成为下一个确认状态的“最佳猜测”。但是，不能保证它实际上会，因为粒子可能会失败，或者因为事件可能会与其他事件竞争而失败，导致它们被取消（如果它们是有条件的）或出现在序列中的后一个位置而不是预期（如果他们是无条件的）。

#### 并发保证

请注意，即使使用重入或延迟确认，Orleans基于回合制的调度（协作并发）保证始终适用。这意味着即使有几种方法正在进行中，只有一种方法可以主动执行---所有其他方法都处于等待状态，因此并行线程不会引发任何真正的竞争。

特别要注意：

* 的属性State，TentativeState，Version，和UnconfirmedEvents可以在方法的执行期间改变。
* 但这种变化只能在等待期间发生。

这些保证假定用户代码保持在关于任务和async / await 的[推荐实践中](https://dotnet.github.io/grains/external_tasks_and_grains.md)（特别是，不使用线程池任务，或仅将它们用于不调用Grain功能且正确等待的代码）。

### 通知

能够对状态变化作出反应通常很方便。所有回调均受Orleans回合制保证的约束; 另请参阅“并发保证”一节。

#### 跟踪确认状态

要通知已确认状态的任何更改，JournaledGrain子类可以覆盖此方法：

protected override void OnStateChanged()

{

// read state and/or event log and take appropriate action

}

OnStateChanged每当更新确认状态时调用，即版本号增加。这可能发生在

1. 从存储中加载了更新版本的状态。
2. 此实例引发的事件已成功写入存储。
3. 从其他一些实例收到通知消息。

请注意，由于所有Grain最初都具有零版本，因此在存储的初始加载完成之前，这意味着OnStateChanged只要初始加载以大于零的版本完成，就会调用它。

#### 跟踪暂定状态

要通知暂定状态的任何更改，JournaledGrain子类可以覆盖此方法：

protected override void OnTentativeStateChanged()

{

// read state and/or events and take appropriate action

}

OnTentativeStateChanged每当临时状态改变时调用，即组合序列（ConfirmedEvents + UnconfirmedEvents）改变。特别是，回调OnTentativeStateChanged()始终发生在RaiseEvent。

### 配置

#### 配置项目引用

##### Grain接口

和以前一样，接口仅依赖于Microsoft.Orleans.Core包，因为grain接口独立于实现。

##### Grain实现

JournaledGrains需要派生自JournaledGrain<S,E>或JournaledGrain<S>在Microsoft.Orleans.EventSourcing包中定义。

#### 日志一致性提供者

我们目前包括三个日志一致性提供程序（用于状态存储，日志存储和自定义存储）。这三个都包含在Microsoft.Orleans.EventSourcing包装中。因此，所有Journaled Grains都可以访问这些。有关这些提供程序的功能及其区别的说明，请参阅[包含的日志一致性提供程序](https://dotnet.github.io/orleans/Documentation/grains/event_sourcing/log_consistency_providers.html)。

#### 群集配置

日志一致性提供程序的配置与任何其他Orleans提供程序一样。例如，要包含所有三个提供程序（当然，您可能不需要全部三个），请将其添加到<Globals>配置文件的元素中：

<LogConsistencyProviders>

<Provider Type="Orleans.EventSourcing.StateStorage.LogConsistencyProvider" Name="StateStorage" />

<Provider Type="Orleans.EventSourcing.LogStorage.LogConsistencyProvider" Name="LogStorage" />

<Provider Type="Orleans.EventSourcing.CustomStorage.LogConsistencyProvider" Name="CustomStorage" />

</LogConsistencyProviders>

可以通过编程方式实现同​​样的目的。假设项目包含Microsoft.Orleans.EventSourcing包，并且config是一个ClusterConfiguration对象：

using Orleans.Runtime.Configuration; // pick up the necessary extension methods

config.AddLogStorageBasedLogConsistencyProvider("LogStorage");

config.AddStateStorageBasedLogConsistencyProvider("StateStorage");

config.AddCustomStorageBasedLogConsistencyProvider("CustomStorage");

#### Grain类属性

每个日志Grain类必须具有LogConsistencyProvider指定日志一致性提供者的属性。某些提供商还需要一个StorageProvider属性。

#### LogConsistencyProvider属性

要指定日志一致性提供程序，请将[LogConsistencyProvider(ProviderName=...)]属性添加到grain类，并根据Cluster Configuration配置提供提供程序的名称。例如：

[LogConsistencyProvider(ProviderName = "CustomStorage")]

public class ChatGrain : JournaledGrain<XDocument, IChatEvent>, IChatGrain, ICustomStorage { ... }

#### StorageProvider属性

某些日志一致性提供程序（包括LogStorage和StateStorage）使用标准StorageProvider与存储进行通信。使用单独的StorageProvider属性指定此提供程序，如下所示：

[LogConsistencyProvider(ProviderName = "LogStorage")]

[StorageProvider(ProviderName = "AzureBlobStorage")]

public class ChatGrain : JournaledGrain<XDocument, IChatEvent>, IChatGrain { ... }

#### 默认提供商

如果在配置中指定了默认值，则可以省略LogConsistencyProvider和/或StorageProvider属性。这是通过使用Default相应提供程序的特殊名称来完成的。例如：

<LogConsistencyProviders>

<Provider Type="Orleans.EventSourcing.LogStorage.LogConsistencyProvider" Name="Default" />

</LogConsistencyProviders>

<StorageProviders>

<Provider Type="Orleans.Storage.MemoryStorage" Name="Default" />

</StorageProviders>

### 内置日志一致性提供程序

该Microsoft.Orleans.EventSourcing软件包包括几个日志一致性提供程序，涵盖适合入门的基本方案，并允许一些可扩展性。

#### Orleans.EventSourcing。****StateStorage**** .LogConsistencyProvider

此提供程序使用可以独立配置的标准存储提供程序存储Grain状态快照。

存储中保存的数据是一个对象，它包含Grain状态（由第一个类型参数指定JournaledGrain）和一些元数据（版本号，以及用于避免存储时重复事件的特殊标记）访问失败）。

由于每次访问存储时都会读取/写入整个Grain状态，因此该提供程序不适用于Grain状态非常大的对象。

此提供程序不支持RetrieveConfirmedEvents：它无法从存储中检索事件，因为事件不会持久存在。

#### Orleans.EventSourcing.**LogStorage** .LogConsistencyProvider

此提供程序使用可以单独配置的标准存储提供程序将完整事件序列存储为单个对象。

存储中保存的数据是包含一个List<EventType> object和一些元数据的对象（一种特殊标记，用于避免在存储访问失败时重复事件）。

该提供商确实支持RetrieveConfirmedEvents。所有事件始终可用并保存在内存中。

由于每次访问存储时都会读取/写入整个事件序列，因此该提供程序不适合用于生产，除非事件序列保证非常短。此提供程序的主要目的是说明事件源和语言/测试环境的语义。

#### Orleans.EventSourcing。****CustomStorage****.LogConsistencyProvider

此提供程序允许开发人员插入自己的存储接口，然后在适当的时候通过一致性协议调用该接口。此提供程序不会对存储的内容是状态快照还是事件做出具体假设 - 程序员可以控制该选择（并且可以存储其中一个或两个）。

要使用此提供程序，Grain必须JournaledGrain<StateType,EventType>像以前一样派生，但另外还必须实现以下接口：

public interface ICustomStorageInterface<StateType, EventType>

{

Task<KeyValuePair<int,StateType>> ReadStateFromStorage();

Task<bool> ApplyUpdatesToStorage(IReadOnlyList<EventType> updates, int expectedversion);

}

一致性提供程序期望这些行为以某种方式运行。程序员应该意识到：

* 第一种方法，ReadStateFromStorage应该返回版本和状态读取。如果还没有存储，则它应该为版本返回零，并且匹配的状态对应于默认构造函数StateType。
* ApplyUpdatesToStorage 如果预期版本与实际版本不匹配，则必须返回false（这类似于电子标签检查）。
* 如果ApplyUpdatesToStorage失败并出现异常，则重试性提供程序将重试。这意味着如果抛出这样的异常，某些事件可能会重复，但事件实际上是持久的。开发人员负责确保这是安全的：例如，通过不抛出异常来避免这种情况，或者确保重复事件对应用程序逻辑无害，或者添加一些额外的机制来过滤重复项。

此提供商不支持RetrieveConfirmedEvents。当然，由于开发人员无论如何都要控制存储接口，因此他们不需要首先调用它，而是可以实现自己的事件检索。

### 复制Grain

有时，可能存在多个相同Grain活动的实例，例如在操作多个群集和使用该[OneInstancePerCluster]属性时。JournaledGrain旨在以最小的摩擦力支持复制的实例。它依赖于日志一致性提供程序来运行必要的协议，以确保所有实例在相同的事件序列上达成一致。特别是，它涉及以下几个方面：

* **一致版本**：所有版本的Grain状态（暂定版本除外）都基于相同的全局事件序列。特别是，如果两个实例看到相同的版本号，则它们会看到相同的状态。
* **赛事事件**：多个实例可以同时举办赛事。一致性提供程序解决此竞争并确保每个人都遵循相同的顺序。
* **通知/反应性**：在一个Grain实例上引发事件后，一致性提供程序不仅更新存储，还会通知所有其他Grain实例。

有关一致性模型的一般性讨论，请参阅我们的[TechReport](https://www.microsoft.com/en-us/research/publication/geo-distribution-actor-based-services/)和[GSP论文](https://www.microsoft.com/en-us/research/publication/global-sequence-protocol-a-robust-abstraction-for-replicated-shared-state-extended-version/)（全局序列协议）。

### 有条件的事件

竞争事件如果发生冲突可能会出现问题，即不应因某种原因而承诺。例如，当从银行账户中提取资金时，两个实例可以独立地确定有足够的资金用于提款，并发出提款事件。但两个事件的结合可能会透支。为避免这种情况，JournaledGrain API支持一种RaiseConditionalEvent方法。

bool success = await RaiseConditionalEvent(new WithdrawalEvent() { ... });

条件事件会仔细检查本地版本是否与存储中的版本匹配。如果不是，则意味着事件序列在此期间增长，这意味着该事件已经失去了与其他事件的竞争。在这种情况下，条件事件不会附加到日志中，并RaiseConditionalEvent返回false。

这是使用带有条件存储更新的电子标签的类比，同样提供了一种避免提交冲突事件的简单机制。

对同一粒子使用条件和无条件事件是可能和明智的，例如a DepositEvent和a WithdrawalEvent。存款不必是有条件的：即使DepositEvent输掉比赛，也不必取消，但仍可以附加到全局事件序列。

等待返回的任务RaiseConditionalEvent足以确认事件，即没有必要也打电话ConfirmEvents。

### 显式同步

有时，我们希望确保Grain完全符合最新版本。这可以通过调用来强制执行

await RefreshNow();

其中（1）确认所有未经证实的事件，（2）从存储装载最新版本。

## 外部任务和Grain

根据设计，从Grain代码中生成的任何子任务（例如，通过使用await或ContinueWith或Task.Factory.StartNew）将在与父任务相同的每个激活的[TPL任务调度程序上调度](https://msdn.microsoft.com/en-us/library/dd997402(v=vs.110).aspx)，因此继承与其余部分相同的单线程执行模型。Grain代码。这是[基于Grain转向的](http://dotnet.github.io/orleans/Tutorials/Concurrency)单一线程执行的主要观点。

在某些情况下，Grain代码可能需要“突破”Orleans任务调度模型并“做一些特殊的事情”，例如明确指向Task不同的任务调度程序或使用.NET线程池。这种情况的一个例子是当Grain代码必须执行同步远程阻塞调用（例如远程IO）时。在Grain环境中执行此操作将阻止Grain以及Orleans螺纹之一，因此永远不应该制造。相反，粒度代码可以在线程池线程上执行这段阻塞代码，并且join（await）完成该执行并继续在grain上下文中。我们希望从Orleans调度程序中逃避将是一种非常先进的，并且很少需要超出“正常”使用模式的使用场景。

### 基于任务的API：

1） ，await，Task.Factory.StartNew，Task.ContinuewWith，，Task.WhenAny 都尊重当前任务计划。这意味着以默认方式使用它们而不传递不同的TaskScheduler将导致它们在grain上下文中执行。Task.WhenAllTask.Delay

2）两者Task.Run和endMethod代表Task.Factory.FromAsync都不尊重当前的任务调度程序。它们都使用TaskScheduler.Default调度程序，它是.NET线程池任务调度程序。因此，里面的代码Task.Run和endMethod将始终为新OrleansGrain单线程执行模型外的.NET线程池中运行，[因为这里详述](http://blogs.msdn.com/b/pfxteam/archive/2011/10/24/10229468.aspx)。但是，任务之后的任何代码await Task.Run或await Task.Factory.FromAsync将在创建任务时在调度程序下运行，这是粒度调度程序。

3）configureAwait(false)是一个用于转义当前任务Scheduler的显式API。它将导致在等待的Task之后的代码在TaskScheduler.Default调度程序（即.NET线程池）上执行，从而打破了Orleans grain的单线程执行。通常，您应该**永远不要configureAwait(false)直接使用Grain代码。**

4）带签名的方法async void不应与Grain一起使用。它们适用于图形用户界面事件处理程序。

### 例：

下面是演示使用的示例代码TaskScheduler.Current，Task.Run以及一个特殊的自定义调度程序，以逃避Orlean grain上下文以及如何返回它。

public async Task MyGrainMethod()

{

// Grab the Orleans task scheduler

var orleansTs = TaskScheduler.Current;

await TaskDelay(10000);

// Current task scheduler did not change, the code after await is still running in the same task scheduler.

Assert.AreEqual(orleansTs, TaskScheduler.Current);

Task t1 = Task.Run( () =>

{

// This code runs on the thread pool scheduler, not on Orleans task scheduler

Assert.AreNotEqual(orleansTS, TaskScheduler.Current);

Assert.AreEqual(TaskScheduler.Default, TaskScheduler.Current);

} );

await t1;

// We are back to the Orleans task scheduler.

// Since await was executed in Orleans task scheduler context, we are now back to that context.

Assert.AreEqual(orleansTS, TaskScheduler.Current);

// Example of using ask.Factory.StartNew with a custom scheduler to escape from the Orleans scheduler

Task t2 = Task.Factory.StartNew(() =>

{

// This code runs on the MyCustomSchedulerThatIWroteMyself scheduler, not on the Orleans task scheduler

Assert.AreNotEqual(orleansTS, TaskScheduler.Current);

Assert.AreEqual(MyCustomSchedulerThatIWroteMyself, TaskScheduler.Current);

},

CancellationToken.None, TaskCreationOptions.None,

scheduler: MyCustomSchedulerThatIWroteMyself);

await t2;

// We are back to Orleans task scheduler.

Assert.AreEqual(orleansTS, TaskScheduler.Current);

}

### 高级示例 - 从在线程池上运行的代码进行粒度调用

更高级的场景是一块Grain代码需要“突破”Orleans任务调度模型并在线程池（或其他一些非Orleans上下文）上运行，但仍然需要调用另一个Grain。如果您尝试进行粒度调用但不在Orleans上下文中，则会出现一个异常，表示您“尝试在粒子上发送消息而不是从粒度内发送消息而不是在系统目标内发送消息（RuntimeContext不是设置为SchedulingContext）“。

下面的代码演示了如何通过在Grain内部而不是Grain上下文中运行的一段代码进行粒度调用。

public async Task MyGrainMethod()

{

// Grab the Orleans task scheduler

var orleansTs = TaskScheduler.Current;

Task<int> t1 = Task.Run(async () =>

{

// This code runs on the thread pool scheduler, not on Orleans task scheduler

Assert.AreNotEqual(orleansTS, TaskScheduler.Current);

// You can do whatever you need to do here. Now let's say you need to make a grain call.

Task<Task<int>> t2 = Task.Factory.StartNew(() =>

{

// This code runs on the Orleans task scheduler since we specified the scheduler: orleansTs.

Assert.AreEqual(orleansTS, TaskScheduler.Current);

return GrainFactory.GetGrain<IFooGrain>(0).MakeGrainCall();

}, CancellationToken.None, TaskCreationOptions.None, scheduler: orleansTs);

int res = await (await t2); // double await, unrelated to Orleans, just part of TPL APIs.

// This code runs back on the thread pool scheduler, not on the Orleans task scheduler

Assert.AreNotEqual(orleansTS, TaskScheduler.Current);

return res;

} );

int result = await t1;

// We are back to the Orleans task scheduler.

// Since await was executed in the Orleans task scheduler context, we are now back to that context.

Assert.AreEqual(orleansTS, TaskScheduler.Current);

}

### 处理Libs

您的代码正在使用的一些外部库可能在ConfigureAwait(false)内部使用。事实上，ConfigureAwait(false) [在实现通用库时](https://msdn.microsoft.com/en-us/magazine/jj991977.aspx)，使用.NET是一种很好的正确做法。这在Orleans不是问题。只要调用库方法await的grain中的代码正在等待使用常规的库调用，Grain代码就是正确的。结果将完全符合要求 - 库代码将在Default调度程序上运行continuation（恰好ThreadPoolTaskScheduler但它不能保证continuation肯定会在ThreadPool线程上运行，因为continuation通常在前一个线程中内联） ，而Grain代码将在Orleans调度程序上运行。

另一个常见问题是是否需要执行库调用Task.Run- 也就是说，是否需要将库代码显式卸载到ThreadPool（用于执行Grain代码Task.Run(()=> myLibrary.FooAsync())）。答案是否定的。没有必要将任何代码卸载到ThreadPool，除了进行阻塞同步调用的库代码的情况。通常，任何编写良好且正确的.NET异步库（返回Task并以Async后缀命名的方法）都不会进行阻塞调用。因此，除非您怀疑异步库是错误的或者您故意使用同步阻塞库，否则无需将任何内容卸载到ThreadPool。

### 摘要

| **你想做什么？** | **怎么做** |
| --- | --- |
| 在.NET线程池线程上运行后台工作。不允许Grain代码或Grain调用。 | Task.Run |
| 粮食界面电话 | 方法返回类型= Task或Task<T> |
| 使用Orleans回合制并发保证从Grain代码运行工作任务。 | Task.Factory.StartNew |
| 执行工作项的超时时间 | Task.Delay + Task.WhenAny |
| 与async/一起使用await | 正常的.NET任务 - 异步编程模型。支持和推荐 |
| ConfigureAwait(false) | 不要使用内部Grain代码。仅允许在库内部。 |
| 调用异步库 | await 图书馆电话 |

## Grain呼叫过滤器

Grain呼叫过滤器提供拦截Grain呼叫的方法。过滤器可以在Grain调用之前和之后执行代码。可以同时安装多个过滤器。过滤器是异步的，可以修改RequestContext，参数和被调用方法的返回值。过滤器还可以检查在MethodInfograin类上调用的方法，并可用于抛出或处理异常。

Grain调用过滤器的一些示例用法是：

* 授权：过滤器可以检查被调用的方法以及中的参数或某些授权信息，RequestContext以确定是否允许调用继续进行。
* 记录/遥测：过滤器可以记录信息并捕获定时数据和有关方法调用的其他统计信息。
* 错误处理：过滤器可以拦截方法调用抛出的异常，并将其转换为另一个异常，或在异常通过过滤器时处理异常。

过滤器有两种形式：

* 来电过滤器
* 传出呼叫过滤器

接收呼叫时执行来电过滤。拨打电话时会执行拨出过滤。

### 调用过滤器

传入的Grain调用过滤器实现了IIncomingGrainCallFilter接口，它有一个方法：

public interface IIncomingGrainCallFilter

{

Task Invoke(IIncomingGrainCallContext context);

}

IIncomingGrainCallContext传递给该Invoke方法的参数具有以下形状：

public interface IIncomingGrainCallContext

{

/// <summary>

/// Gets the grain being invoked.

/// </summary>

IAddressable Grain { get; }

/// <summary>

/// Gets the <see cref="MethodInfo"/> for the interface method being invoked.

/// </summary>

MethodInfo InterfaceMethod { get; }

/// <summary>

/// Gets the <see cref="MethodInfo"/> for the implementation method being invoked.

/// </summary>

MethodInfo ImplementationMethod { get; }

/// <summary>

/// Gets the arguments for this method invocation.

/// </summary>

object[] Arguments { get; }

/// <summary>

/// Invokes the request.

/// </summary>

Task Invoke();

/// <summary>

/// Gets or sets the result.

/// </summary>

object Result { get; set; }

}

该IIncomingGrainCallFilter.Invoke(IIncomingGrainCallContext)方法必须等待或返回IIncomingGrainCallContext.Invoke()执行下一个已配置过滤器的结果，并最终执行grain方法本身。Result等待该Invoke()方法后，可以修改该属性。该ImplementationMethod属性返回MethodInfo实现类的内容。所述MethodInfo接口方法的可使用的被访问InterfaceMethod属性。对Grain的所有方法调用都会调用Grain调用过滤器，这包括对GrainIGrainExtension中安装的Grain扩展（实现）的调用。例如，grain扩展用于实现Streams和Cancellation Tokens。因此，应该预期值ImplementationMethod并不总是粮食类本身的方法。

### 配置传入的Grain调用过滤器

IIncomingGrainCallFilter可以通过依赖注入将实现注册为筒仓范围的过滤器，也可以通过IIncomingGrainCallFilter直接实现的粒度注册为粒度级过滤器。

### 筒仓式Grain呼叫过滤器

委托可以使用依赖注入注册为silo范围的粒度调用过滤器，如下所示：

siloHostBuilder.AddIncomingGrainCallFilter(async context =>

{

// If the method being called is 'MyInterceptedMethod', then set a value

// on the RequestContext which can then be read by other filters or the grain.

if (string.Equals(context.InterfaceMethod.Name, nameof(IMyGrain.MyInterceptedMethod)))

{

RequestContext.Set("intercepted value", "this value was added by the filter");

}

await context.Invoke();

// If the grain method returned an int, set the result to double that value.

if (context.Result is int resultValue) context.Result = resultValue \* 2;

});

类似地，可以使用AddIncomingGrainCallFilter辅助方法将类注册为粒度调用过滤器。这是一个Grain调用过滤器的示例，它记录每个Grain方法的结果：

public class LoggingCallFilter : IIncomingGrainCallFilter

{

private readonly Logger log;

public LoggingCallFilter(Factory<string, Logger> loggerFactory)

{

this.log = loggerFactory(nameof(LoggingCallFilter));

}

public async Task Invoke(IIncomingGrainCallContext context)

{

try

{

await context.Invoke();

var msg = string.Format(

"{0}.{1}({2}) returned value {3}",

context.Grain.GetType(),

context.InterfaceMethod.Name,

string.Join(", ", context.Arguments),

context.Result);

this.log.Info(msg);

}

catch (Exception exception)

{

var msg = string.Format(

"{0}.{1}({2}) threw an exception: {3}",

context.Grain.GetType(),

context.InterfaceMethod.Name,

string.Join(", ", context.Arguments),

exception);

this.log.Info(msg);

// If this exception is not re-thrown, it is considered to be

// handled by this filter.

throw;

}

}

}

然后可以使用AddIncomingGrainCallFilter扩展方法注册此过滤器：

siloHostBuilder.AddIncomingGrainCallFilter<LoggingCallFilter>();

或者，可以在没有扩展方法的情况下注册过滤器：

siloHostBuilder.ConfigureServices(

services => services.AddSingleton<IIncomingGrainCallFilter, LoggingCallFilter>());

### 每粒Grain呼叫过滤器

粒度类可以将自身注册为粒度调用过滤器，并通过IIncomingGrainCallFilter如此实现过滤对其进行的任何调用：

public class MyFilteredGrain : Grain, IMyFilteredGrain, IIncomingGrainCallFilter

{

public async Task Invoke(IIncomingGrainCallContext context)

{

await context.Invoke();

// Change the result of the call from 7 to 38.

if (string.Equals(context.InterfaceMethod.Name, nameof(this.GetFavoriteNumber)))

{

context.Result = 38;

}

}

public Task<int> GetFavoriteNumber() => Task.FromResult(7);

}

在上面的示例中，对GetFavoriteNumber方法的所有调用都将返回38而不是7，因为返回值已被过滤器更改。

过滤器的另一个用例是访问控制，如下例所示：

[AttributeUsage(AttributeTargets.Method)]

public class AdminOnlyAttribute : Attribute { }

public class MyAccessControlledGrain : Grain, IMyFilteredGrain, IIncomingGrainCallFilter

{

public Task Invoke(IIncomingGrainCallContext context)

{

// Check access conditions.

var isAdminMethod = context.ImplementationMethod.GetCustomAttribute<AdminOnlyAttribute>();

if (isAdminMethod && !(bool) RequestContext.Get("isAdmin"))

{

throw new AccessDeniedException($"Only admins can access {context.ImplementationMethod.Name}!");

}

return context.Invoke();

}

[AdminOnly]

public Task<int> SpecialAdminOnlyOperation() => Task.FromResult(7);

}

在上面的例子中，SpecialAdminOnlyOperation如果只能被称为方法"isAdmin"被设置为true在RequestContext。通过这种方式，Grain呼叫过滤器可用于授权。在此示例中，调用者有责任确保"isAdmin"正确设置值并正确执行身份验证。请注意，该[AdminOnly]属性是在grain类方法中指定的。这是因为ImplementationMethod属性返回的MethodInfo是实现，而不是接口。过滤器还可以检查InterfaceMethod属性。

### Grain调用过滤器的订购

Grain调用过滤器遵循定义的顺序：

1. IIncomingGrainCallFilter 在依赖项注入容器中按照它们的注册顺序配置的实现。
2. Grain过滤器，如果粮食实施IIncomingGrainCallFilter。
3. Grain方法实现或grain扩展方法实现。

每次调用都会IIncomingGrainCallContext.Invoke()封装下一个定义的过滤器，以便每个过滤器都有机会在链中的下一个过滤器之前和之后执行代码，最终执行grain方法本身。

### 传出调用过滤器

传出的粒子调用过滤器类似于传入的粒子调用过滤器，主要区别在于它们在调用者（客户端）而不是被调用者（粒度）上调用。

传出的粒度调用过滤器实现了IOutgoingGrainCallFilter接口，它有一个方法：

public interface IOutgoingGrainCallFilter

{

Task Invoke(IOutgoingGrainCallContext context);

}

IOutgoingGrainCallContext传递给该Invoke方法的参数具有以下形状：

public interface IOutgoingGrainCallContext

{

/// <summary>

/// Gets the grain being invoked.

/// </summary>

IAddressable Grain { get; }

/// <summary>

/// Gets the <see cref="MethodInfo"/> for the interface method being invoked.

/// </summary>

MethodInfo InterfaceMethod { get; }

/// <summary>

/// Gets the arguments for this method invocation.

/// </summary>

object[] Arguments { get; }

/// <summary>

/// Invokes the request.

/// </summary>

Task Invoke();

/// <summary>

/// Gets or sets the result.

/// </summary>

object Result { get; set; }

}

该IOutgoingGrainCallFilter.Invoke(IOutgoingGrainCallContext)方法必须等待或返回IOutgoingGrainCallContext.Invoke()执行下一个已配置过滤器的结果，并最终执行grain方法本身。Result等待该Invoke()方法后，可以修改该属性。所述MethodInfo接口方法的调用可以使用来访问InterfaceMethod属性。对Grain的所有方法调用都会调用派出Grain调用过滤器，这包括对Orleans制作的系统方法的调用。

### 配置传出Grain调用过滤器

IOutgoingGrainCallFilter可以使用依赖注入在两个孤岛和客户端上注册实现。

代理可以注册为调用过滤器，如下所示：

builder.AddOutgoingGrainCallFilter(async context =>

{

// If the method being called is 'MyInterceptedMethod', then set a value

// on the RequestContext which can then be read by other filters or the grain.

if (string.Equals(context.InterfaceMethod.Name, nameof(IMyGrain.MyInterceptedMethod)))

{

RequestContext.Set("intercepted value", "this value was added by the filter");

}

await context.Invoke();

// If the grain method returned an int, set the result to double that value.

if (context.Result is int resultValue) context.Result = resultValue \* 2;

});

在上面的代码，builder可以是的一个实例ISiloHostBuilder或IClientBuilder。

类似地，可以将类注册为传出Grain调用过滤器。这是一个Grain调用过滤器的示例，它记录每个Grain方法的结果：

public class LoggingCallFilter : IOutgoingGrainCallFilter

{

private readonly Logger log;

public LoggingCallFilter(Factory<string, Logger> loggerFactory)

{

this.log = loggerFactory(nameof(LoggingCallFilter));

}

public async Task Invoke(IOutgoingGrainCallContext context)

{

try

{

await context.Invoke();

var msg = string.Format(

"{0}.{1}({2}) returned value {3}",

context.Grain.GetType(),

context.InterfaceMethod.Name,

string.Join(", ", context.Arguments),

context.Result);

this.log.Info(msg);

}

catch (Exception exception)

{

var msg = string.Format(

"{0}.{1}({2}) threw an exception: {3}",

context.Grain.GetType(),

context.InterfaceMethod.Name,

string.Join(", ", context.Arguments),

exception);

this.log.Info(msg);

// If this exception is not re-thrown, it is considered to be

// handled by this filter.

throw;

}

}

}

然后可以使用AddOutgoingGrainCallFilter扩展方法注册此过滤器：

builder.AddOutgoingGrainCallFilter<LoggingCallFilter>();

或者，可以在没有扩展方法的情况下注册过滤器：

builder.ConfigureServices(

services => services.AddSingleton<IOutgoingGrainCallFilter, LoggingCallFilter>());

与委托调用过滤器示例一样，builder可以是ISiloHostBuiler或者的实例IClientBuilder。

### 用例

### 异常转换

当从服务器抛出的异常在客户端上反序列化时，有时可能会得到以下异常而不是实际异常： TypeLoadException: Could not find Whatever.dll.

如果包含异常的程序集对客户端不可用，则会发生这种情况。例如，假设您在粮食实施中使用实体框架; 然后有可能EntityException抛出一个。另一方面，客户端不会（也不应该）引用，EntityFramework.dll因为它不了解底层数据访问层。

当客户端尝试反序列化时EntityException，由于缺少DLL，它将失败; 结果TypeLoadException是抛出原来的东西EntityException。

有人可能会说这很好，因为客户永远不会处理EntityException; 否则它将不得不参考EntityFramework.dll。

但是如果客户想要至少记录异常呢？问题是原始错误消息丢失。解决此问题的一种方法是拦截服务器端异常，Exception如果异常类型在客户端可能是未知的，则将其替换为类型的普通异常。

但是，我们必须记住一件重要的事情：**如果调用者是粒度客户端，**我们只想替换异常。如果调用者是另一个Grain（或者正在进行Grain调用的Orleans基础设施，例如在GrainBasedReminderTableGrain上），我们不想替换异常。

在服务器端，这可以通过筒仓级拦截器来完成：

public class ExceptionConversionFilter : IIncomingGrainCallFilter

{

private static readonly HashSet<string> KnownExceptionTypeAssemblyNames =

new HashSet<string>

{

typeof(string).Assembly.GetName().Name,

"System",

"System.ComponentModel.Composition",

"System.ComponentModel.DataAnnotations",

"System.Configuration",

"System.Core",

"System.Data",

"System.Data.DataSetExtensions",

"System.Net.Http",

"System.Numerics",

"System.Runtime.Serialization",

"System.Security",

"System.Xml",

"System.Xml.Linq",

"MyCompany.Microservices.DataTransfer",

"MyCompany.Microservices.Interfaces",

"MyCompany.Microservices.ServiceLayer"

};

public async Task Invoke(IIncomingGrainCallContext context)

{

var isConversionEnabled =

RequestContext.Get("IsExceptionConversionEnabled") as bool? == true;

if (!isConversionEnabled)

{

// If exception conversion is not enabled, execute the call without interference.

await context.Invoke();

return;

}

RequestContext.Remove("IsExceptionConversionEnabled");

try

{

await context.Invoke();

}

catch (Exception exc)

{

var type = exc.GetType();

if (KnownExceptionTypeAssemblyNames.Contains(type.Assembly.GetName().Name))

{

throw;

}

// Throw a base exception containing some exception details.

throw new Exception(

string.Format(

"Exception of non-public type '{0}' has been wrapped."

+ " Original message: <<<<----{1}{2}{3}---->>>>",

type.FullName,

Environment.NewLine,

exc,

Environment.NewLine));

}

}

}

然后可以在筒仓上注册此过滤器：

siloHostBuilder.AddIncomingGrainCallFilter<ExceptionConversionFilter>();

通过添加传出呼叫过滤器为客户端进行的呼叫启用过滤器：

clientBuilder.AddOutgoingGrainCallFilter(context =>

{

RequestContext.Set("IsExceptionConversionEnabled", true);

return context.Invoke();

});

这样客户端告诉服务器它想要使用异常转换。

### 从拦截器调用Grain

通过注入IGrainFactory拦截器类可以从拦截器进行粒子调用：

private readonly IGrainFactory grainFactory;

public CustomCallFilter(IGrainFactory grainFactory)

{

this.grainFactory = grainFactory;

}

public async Task Invoke(IIncomingGrainCallContext context)

{

// Hook calls to any grain other than ICustomFilterGrain implementations.

// This avoids potential infinite recursion when calling OnReceivedCall() below.

if (!(context.Grain is ICustomFilterGrain))

{

var filterGrain = this.grainFactory.GetGrain<ICustomFilterGrain>(context.Grain.GetPrimaryKeyLong());

// Perform some grain call here.

await filterGrain.OnReceivedCall();

}

// Continue invoking the call on the target grain.

await context.Invoke();

}

## GrainServices

GrainService是一种特殊的Grain; 一个没有身份的人，并且在从启动到关闭的每个孤岛中运行。

### 创建GrainService

**步骤1.**创建界面。GrainService的界面使用与构建任何其他Grain的界面完全相同的原则构建。

public interface IDataService : IGrainService {

Task MyMethod();

}

**步骤2.**创建DataService数组本身。如果可能，请使GrainService可重入以获得更好的性能。注意必要的基础构造函数调用。很高兴知道您也可以注入一个，IGrainFactory这样您就可以从GrainService进行Grain调用。

关于流的说明：GrainService无法写入Orleans流，因为它在Grain任务调度程序中不起作用。如果您需要GrainService为您写入流，那么您必须将对象发送到另一种Grain以写入流。

[Reentrant]

public class LightstreamerDataService : GrainService, IDataService {

readonly IGrainFactory GrainFactory;

public LightstreamerDataService(IServiceProvider services, IGrainIdentity id, Silo silo, ILoggerFactory loggerFactory, IGrainFactory grainFactory) : base(id, silo, loggerFactory) {

GrainFactory = grainFactory;

}

public override Task Init(IServiceProvider serviceProvider) {

return base.Init(serviceProvider);

}

public override async Task Start() {

await base.Start();

}

public override Task Stop() {

return base.Stop();

}

public Task MyMethod() {

}

}

**步骤3.**为GrainServiceClient创建一个接口，供其他grain使用，以连接到GrainService。

public interface IDataServiceClient : IGrainServiceClient<IDataService>, IDataService {

}

**步骤4.**创建实际的Grain服务客户端。它几乎只是作为数据服务的代理。不幸的是，您必须手动输入所有方法映射，这只是简单的单行映射。

public class DataServiceClient : GrainServiceClient<IDataService>, IDataServiceClient {

public DataServiceClient(IServiceProvider serviceProvider) : base(serviceProvider) {

}

public Task MyMethod() => GrainService.MyMethod();

}

**步骤5.**将Grain服务客户注入需要它的其他Grain中。请注意，GrainServiceClient不保证访问本地silo上的GrainService。您的命令可能会在群集中的任何Silo上发送到GrainService。

public class MyNormalGrain: Grain<NormalGrainState>, INormalGrain {

readonly IDataServiceClient DataServiceClient;

public MyNormalGrain(IGrainActivationContext grainActivationContext, IDataServiceClient dataServiceClient) {

DataServiceClient = dataServiceClient;

}

}

**步骤6.**将Grain服务注入筒仓本身。您需要这样做才能使筒仓启动GrainService。

(ISiloHostBuilder builder) => builder .ConfigureServices(services => { services.AddSingleton<IDataService, DataService>(); });

## 补充说明

### 注1

有一个扩展方法ISiloHostBuilder: AddGrainService<SomeGrainService>()。类型约束是：where T : GrainService。最后调用这个位：**orleans / src / Orleans.Runtime / Services / GrainServicesSiloBuilderExtensions.cs**

return services.AddSingleton<IGrainService>(sp => GrainServiceFactory(grainServiceType, sp));

基本上，silo IGrainService在启动时从服务提供者获取类型：**orleans / src / Orleans.Runtime / Silo / Silo.cs**var grainServices = this.Services.GetServices<IGrainService>();

该Orleans.RuntimeNuGet包应由Grainservice项目引用。

### 笔记2

为了使其工作，您必须注册服务及其客户端。代码看起来像这样：

var builder = new SiloHostBuilder()

.AddGrainService<DataService>() // Register GrainService

.ConfigureServices(s =>

{

// Register Client of GrainService

s.AddSingleton<IDataServiceClient, DataServiceClient>();

})

## 无国籍工人Grain

默认情况下，Orleans运行时在群集中创建的Grain激活不超过一次。这是Virtual Actor模型的最直观的表达，其中每个颗粒对应于具有唯一类型/身份的实体。但是，也存在应用程序需要执行与系统中的特定实体无关的功能无状态操作的情况。例如，如果客户端发送具有压缩有效负载的请求，需要将其解压缩，然后才能将其路由到目标颗粒进行处理，则此类解压缩/路由逻辑不会绑定到应用程序中的特定实体，并且可以轻松扩展。

当该[StatelessWorker]属性应用于Grain类时，它向Orleans运行时指示该类的Grain应被视为**无状态工人**Grain。 **无国籍工人**Grain具有以下属性，使其执行与正常Grain类别的执行非常不同。

1. Orleans运行时可以并将在群集的不同孤岛上创建多个无状态工作者粒度的激活。
2. 对无状态工人粮食的请求总是在当地执行，即在发出请求的同一个仓库中，由在粮仓上运行的粮食或由筒仓的客户网关接收。因此，从其他Grain或客户网关呼叫无国籍工人Grain从不会产生远程信息。
3. 如果现有的Grain忙碌，Orleans Runtime会自动创建额外的无状态工作者Grain激活。除非由可选maxLocalWorkers参数明确指定，否则运行时为每个孤岛创建的无状态工作者粒度的最大激活次数默认受限于计算机上的CPU核心数。
4. 由于2和3，无国籍工人的粮食激活不能单独解决。随后对无国籍工人粮食的两次请求可能会通过不同的激活来处理。

无状态工人Grain提供了一种直接的方式来创建自动管理的Grain激活池，该池自动根据实际负载进行放大和缩小。运行时始终以相同的顺序扫描可用的无状态工作者颗粒激活。因此，它总是将请求分派给它可以找到的第一个空闲本地激活，并且只有在所有先前的激活都忙的情况下才会到达最后一个。如果所有激活都忙，并且尚未达到激活限制，则会在列表末尾再创建一次激活，并将请求发送给它。这意味着当对无状态工作者粒度的请求率增加，并且现有激活当前都处于忙碌状态时，运行时会将其激活池扩展到极限。相反，当负载下降时，它可以通过少量无状态工人粮食的激活来处理，列表尾部的激活不会收到发送给他们的请求。它们将变为空闲状态，并最终通过标准激活收集过程停用。因此，激活池最终将缩小以匹配负载。

以下示例MyStatelessWorkerGrain使用默认的最大激活数限制定义Stateless Worker粒度类。

[StatelessWorker]

public class MyStatelessWorkerGrain : Grain, IMyStatelessWorkerGrain

{

...

}

打电话给无国籍工人的粮食与任何其他粮食一样。唯一的区别是在大多数情况下使用单个颗粒ID，0或Guid.Empty。当具有多个无状态工作者粒度池（每个ID一个）是可取的时，可以使用多个粒度ID。

var worker = GrainFactory.GetGrain<IMyStatelessWorkerGrain>(0);

await worker.Process(args);

这个定义了一个无状态工人粮食类别，每个筒仓只有一个Grain激活。

[StatelessWorker(1)] // max 1 activation per silo

public class MyLonelyWorkerGrain : ILonelyWorkerGrain

{

...

}

请注意，[StatelessWorker]属性不会更改目标粒度类的重入。就像任何其他Grain一样，无国籍工人Grain默认是不可重入的。通过向[Reentrant]grain类添加属性，可以明确地将它们重新定义。

### 状态

“无国籍工人”的“无国籍”部分并不意味着无国籍工人不能拥有国家，只能执行职能运作。与任何其他Grain一样，无国籍工人Grain可以加载并保持其所需的任何状态。这只是因为可以在集群的相同和不同的孤岛上创建无状态工作者粒度的多次激活，没有简单的机制来协调不同激活所持有的状态。

有几种有用的模式涉及无国籍工人持有国家。

### 扩展热缓存项目

对于经历高吞吐量的热缓存项目，将每个此类项目保存在无状态工作者粒度中使其a）在筒仓内自动扩展并跨越集群中的所有孤岛; 和b）使数据始终在通过其客户端网关接收客户端请求的孤岛上本地可用，以便可以在没有额外网络跳转到另一个孤岛的情况下回答请求。

### 减少样式聚合

在某些情况下，应用程序需要计算群集中特定类型的所有粒度的某些度量标准，并定期报告聚合。示例是报告每个游戏地图的玩家数量，VoIP呼叫的平均持续时间等。如果数千或数百万个Grain中的每一个都要将其指标报告给单个全局聚合器，则聚合器将立即过载而无法处理大量的报道。另一种方法是将此任务转换为2（或更多）步骤减少样式聚合。第一层聚合是通过报告Grain将其指标发送给无状态工作者预聚合Grain来完成的。Orleans运行时将自动为每个筒仓创建无状态工作者Grain的多个激活。由于所有这些呼叫都将在本地处理而没有远程呼叫或消息的序列化，因此这种聚合的成本将远远低于远程情况。现在，每个预聚合无状态工作者Grain激活，独立地或与其他本地激活协调，可以将其聚合报告发送到全局最终聚合器（或必要时发送到另一个减少层）而不会使其超载。

## Grain取消令牌

Orleans运行时提供了称为Grain取消令牌的机制，使开发人员能够取消正在执行的Grain操作。

### 描述

**GrainCancellationToken**是标准**.NET System.Threading.CancellationToken**的包装器，它支持线程，线程池工作项或Task对象之间的协作取消，并且可以作为grain方法参数传递。

甲**GrainCancellationTokenSource**是通过其令牌属性提供了一个取消标记，并通过调用它的发送取消消息目的Cancel方法。

### 用法

* 实例化CancellationTokenSource对象，该对象管理并向各个取消令牌发送取消通知。

var tcs = new GrainCancellationTokenSource();

* 将GrainCancellationTokenSource.Token属性返回的标记传递给侦听取消的每个grain方法。

var waitTask = grain.LongIoWork(tcs.Token, TimeSpan.FromSeconds(10));

* 可取消的Grain操作需要处理**GrainCancellationToken的**基础**CancellationToken**属性，就像在任何其他.NET代码中一样。

public async Task LongIoWork(GrainCancellationToken tc, TimeSpan delay)

{

while(!tc.CancellationToken.IsCancellationRequested)

{

await IoOperation(tc.CancellationToken);

}

}

* 调用该GrainCancellationTokenSource.Cancel方法以启动取消。

await tcs.Cancel();

* Dispose完成**GrainCancellationTokenSource**对象后调用该方法。

tcs.Dispose();

#### 重要注意事项：

* 该GrainCancellationTokenSource.Cancel方法返回**Task**，为了确保取消，必须在瞬态通信失败的情况下重试取消调用。
* 在底层**System.Threading.CancellationToken**中注册的回调是在其注册的粒子激活中的单线程执行保证的主题。
* 每个**GrainCancellationToken**都可以通过多个方法调用进行传递。

## Orleans交易（测试版）

**Orleans交易处于beta状态，不应在生产系统中使用。**

Orleans支持针对持久Grain状态的分布式事务。

### 建立

Orleans交易是选择加入的。必须将筒仓配置为使用事务。如果不是，对事务Grain的任何调用都将收到OrleansTransactionsDisabledException。

#### Transaction Manager

Orleans交易需要一个对交易状态具有权威性的交易管理者。目前，事务管理器在群集中运行，必须进行配置。可以使用UseInClusterTransactionManager和TransactionsConfiguration在silo主机构建器上以编程方式配置它。

TransactionsConfiguration包含以下可配置值：

* TransactionIdAllocationBatchSize - 为了避免每次事务启动时的存储调用，事务ID都是批量分配的。这是每个事务ID生成请求（存储调用）分配的新事务ID的数量。
* AvailableTransactionIdThreshold - 如果可用ID低于此阈值，将自动分配一批新的事务ID。
* TransactionRecordPreservationDuration - 事务完成后在TM内存中保留事务记录的时间。这用于回答有关交易结果的查询。

建议您使用这些值的默认值，除非它们表明自己不够。

#### Transaction Log

事务管理器必须将事务结果保留为可恢复性目的。为了支持一系列后端存储解决方案，已经为日志存储提供了抽象。我们目前支持日志存储的azure存储版本，以及用于开发和测试目的的内存中版本。可以使用UseAzureTransactionLog或UseInMemoryTransactionLog在silo主机构建器上以编程方式配置它。

### Transaction状态

为了支持各种事务存储模式，面向接口的接口和支持逻辑是可插拔的。我们目前只支持ITransactionalState。可以使用UseTransactionalState在silo主机构建器上以编程方式启用此功能。要使用事务状态，还需要配置存储提供程序。

例：

var builder = new SiloHostBuilder()

[...]

.AddAzureTableGrainStorageAsDefault(options => options.ConnectionString = ”YOUR\_STORAGE\_CONNECTION\_STRING”)

.UseInClusterTransactionManager()

.UseAzureTransactionLog(options => options.ConnectionString = ”YOUR\_STORAGE\_CONNECTION\_STRING”)

.UseTransactionalState();

var host = builder.Build();

await host.StartAsync();

### 编程模型

对于Grain来支持交易，Grain上的交易操作必须使用“交易”属性标记为交易的一部分。呼叫可以标记为“必需”，这意味着呼叫可以参与多个其他Grain之间的交易，或者“RequiresNew”表示它已经开始自己的交易。

例：

public interface IATMGrain : IGrainWithIntegerKey

{

[Transaction(TransactionOption.RequiresNew)]

Task Transfer(Guid fromAccount, Guid toAccount, uint amountToTransfer);

}

上述atm grain中的Transfer操作将始终启动一个涉及两个引用帐户的新事务。

public interface IAccountGrain : IGrainWithGuidKey

{

[Transaction(TransactionOption.Required)]

Task Withdraw(uint amount);

[Transaction(TransactionOption.Required)]

Task Deposit(uint amount);

[Transaction(TransactionOption.Required)]

Task<uint> GetBalance();

}

上述账户中的提款和存款操作可以参与交易操作，例如ATMGrain中的转移操作，以及其他交易Grain。

要在粒度中使用事务状态，只需要定义一个可序列化的状态类来保持它并在grain构造函数中声明状态。

例：

public class AccountGrain : Grain, IAccountGrain

{

private readonly ITransactionalState<Balance> balance;

public AccountGrain(

[TransactionalState("balance")] ITransactionalState<Balance> balance)

{

this.balance = balance ?? throw new ArgumentNullException(nameof(balance));

}

Task IAccountGrain.Deposit(uint ammount)

{

this.balance.State.Value += ammount;

this.balance.Save();

return Task.CompletedTask;

}

Task IAccountGrain.Withdrawal(uint ammount)

{

this.balance.State.Value -= ammount;

this.balance.Save();

return Task.CompletedTask;

}

Task<uint> IAccountGrain.GetBalance()

{

return Task.FromResult(this.balance.State.Value);

}

}

在上面的示例中，属性TransactionalState用于声明'balance'构造参数应该与名为“balance”的事务状态相关联。有了这个声明，Orleans将连接一个ITransactionalState具有从默认存储提供程序加载的状态的实例，以供使用的grain。可以通过State属性访问和修改州。需要通过调用'Save（）'来记录对状态的任何更改。交易基础设施将确保作为交易的一部分执行的任何此类更改，即使在分布在Orleans集群上的多个Grain中，也将全部提交或还原。

## 奥尔良配置指南

本配置指南介绍了关键配置参数以及它们应如何用于大多数典型使用场景。

Orleans可以用于各种适合不同使用场景的配置，例如用于开发和测试的本地单节点部署，服务器集群，多实例Azure工作者角色等。

本指南提供了使Orleans在其中一个目标方案中运行所必需的关键配置参数的说明。还有其他配置参数主要帮助微调奥尔良以获得更好的性能。

筒仓和客户端经由编程配置SiloHostBuilder和ClientBuilder分别和多个补充选项类。Orleans中的选项类遵循[ASP.NET选项](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options/)模式，可以通过文件，环境变量等加载。有关更多信息，请参阅[选项模式文档](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options/)。

如果要为本地开发配置silo和客户端，请查看“ [本地开发配置”](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/local_development_configuration.html)部分。本指南的“ [服务器配置”](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/server_configuration.html)和“ [客户端配置”](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/client_configuration.html)部分分别包括配置筒仓和客户端。

“ [典型配置](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/typical_configurations.html) ”部分提供了一些常见配置的摘要。

可以在[此部分](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/list_of_options_classes.html)找到可配置的重要核心选项列表。

**重要**：确保正确配置.NET垃圾收集，如[配置.NET垃圾收集中所述](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/configuring_.NET_garbage_collection.html)。

### 本地开发配置

对于针对Orleans 2.0的工作示例应用程序，请参阅：[https](https://github.com/dotnet/orleans/tree/master/Samples/2.0/HelloWorld)：[//github.com/dotnet/orleans/tree/master/Samples/2.0/HelloWorld](https://github.com/dotnet/orleans/tree/master/Samples/2.0/HelloWorld) 该示例托管客户端和.NET Core控制台应用程序中的孤岛平台，但同样可以使用.NET Framework 4.6.1+控制台应用程序（仅适用于Windows）。

#### 筒仓配置

有关本地开发，请参阅以下示例，了解如何为该案例配置筒仓。它配置并启动一个孤岛监听“loopback”地址，11111和30000分别作为silo和gateway端口。

将Microsoft.Orleans.ServerNuGet元包添加到项目中。熟悉API之后，您可以选择Microsoft.Orleans.Server实际需要的精确软件包，然后引用它们。

PM> Install-Package Microsoft.Orleans.Server

您需要ClusterOptions通过ISiloBuilder.Configure方法配置，指定您希望DevelopmentClustering作为群集选择，并将此silo作为主要，然后配置silo端点。

ConfigureApplicationPartscall显式将带有grain类的程序集添加到应用程序设置中。由于WithReferences扩展，它还添加了任何引用的程序集。完成这些步骤后，构建筒仓主机并启动筒仓。

您可以创建一个针对.NET Framework 4.6.1或更高版本的空控制台应用程序项目，用于托管silo以及.NET Core控制台应用程序。

以下是如何启动本地孤岛的示例：

public class Program

{

public static async Task Main(string[] args)

{

try

{

var host = await StartSilo();

Console.WriteLine("Press Enter to terminate...");

Console.ReadLine();

await host.StopAsync();

return;

}

catch (Exception ex)

{

Console.WriteLine(ex);

return;

}

}

private static async Task<ISiloHost> StartSilo()

{

var builder = new SiloHostBuilder()

// Use localhost clustering for a single local silo

.UseLocalhostClustering()

// Configure ClusterId and ServiceId

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "dev";

options.ServiceId = "MyAwesomeService";

})

// Configure connectivity

.Configure<EndpointOptions>(options => options.AdvertisedIPAddress = IPAddress.Loopback)

// Configure logging with any logging framework that supports Microsoft.Extensions.Logging.

// In this particular case it logs using the Microsoft.Extensions.Logging.Console package.

.ConfigureLogging(logging => logging.AddConsole());

var host = builder.Build();

await host.StartAsync();

return host;

}

}

#### 客户端配置

对于本地开发，请参阅以下示例，了解如何为该案例配置客户端。它配置一个连接到loopback孤岛的客户端。

将Microsoft.Orleans.ClientNuGet元包添加到项目中。熟悉API之后，您可以选择Microsoft.Orleans.Client实际需要的精确软件包，然后引用它们。

PM> Install-Package Microsoft.Orleans.Client

您需要配置ClientBuilder与您为本地silo指定的集群ID匹配的集群ID，并指定静态集群作为集群选择，将其指向集装箱的网关端口

ConfigureApplicationParts call显式将带有grain接口的程序集添加到应用程序设置中。

完成这些步骤后，我们可以Connect()在其上构建客户端和方法以连接到群集。

您可以创建一个针对.NET Framework 4.6.1或更高版本的空控制台应用程序项目，以运行客户端或重用您为托管孤岛而创建的控制台应用程序项目。

以下是客户端如何连接到本地silo的示例：

client = new ClientBuilder()

// Use localhost clustering for a single local silo

.UseLocalhostClustering()

// Configure ClusterId and ServiceId

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "dev";

options.ServiceId = "MyAwesomeService";

})

.ConfigureLogging(logging => logging.AddConsole())

var client = builder.Build();

await client.Connect();

### 客户端配置

用于连接到筒仓群并向谷物发送请求的客户端通过一个ClientBuilder和多个补充选项类以编程方式配置。与silo选项一样，客户机选项类遵循[ASP.NET选项](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options)。

客户端配置有几个关键方面：

* 奥尔良聚类信息
* 群集提供者
* 应用部分

客户端配置示例：

var client = new ClientBuilder()

// Clustering information

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "my-first-cluster";

options.ServiceId = "MyAwesomeOrleansService";

})

// Clustering provider

.UseAzureStorageClustering(options => options.ConnectionString = connectionString)

// Application parts: just reference one of the grain interfaces that we use

.ConfigureApplicationParts(parts => parts.AddApplicationPart(typeof(IValueGrain).Assembly))

.Build();

我们将分析此示例中使用的步骤：

#### 奥尔良聚类信息

[...]

// Clustering information

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "orleans-docker";

options.ServiceId = "AspNetSampleApp";

})

[...]

这里我们设置两件事：

* 在ClusterId到"my-first-cluster"：这是为新奥尔良集群的唯一ID。使用此ID的所有客户端和孤岛将能够直接相互通信。例如，有些人会选择ClusterId为每个部署使用不同的。
* 在ServiceId到"AspNetSampleApp"：这是你的应用程序，将被一些供应商使用（例如用于持久性提供者）的唯一ID。**跨部署，此ID应该是稳定的（不会更改）**。

#### 群集提供者

[...]

// Clustering provider

.UseAzureStorageClustering(options => options.ConnectionString = connectionString)

[...]

客户端将使用此提供程序发现群集中可用的所有网关。有几个提供程序可用，在此示例中我们使用Azure表提供程序。

要获得更多详细信息，请查看“服务器配置”页面中的匹配部分。

#### 应用部分

[...]

// Application parts: just reference one of the grain interfaces that we use

.ConfigureApplicationParts(parts => parts.AddApplicationPart(typeof(IValueGrain).Assembly)).WithReferences())

[...];

要获得更多详细信息，请查看“服务器配置”页面中的匹配部分。

### 服务器配置

通过一个SiloHostBuilder和一些补充选项类以编程方式配置一个silo 。Orleans中的选项类遵循[ASP.NET选项](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options)模式，可以通过文件，环境变量等加载。有关更多信息，请参阅[选项模式文档](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/(https:/docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options))。

筒仓配置有几个关键方面：

* 奥尔良聚类信息
* 群集提供者
* 用于筒仓到筒仓和客户端到筒仓通信的端点
* 应用部分

定义集群信息的silo配置示例，使用Azure集群并配置应用程序部分：

var silo = new SiloHostBuilder()

// Clustering information

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "my-first-cluster";

options.ServiceId = "MyAwesomeOrleansService";

})

// Clustering provider

.UseAzureStorageClustering(options => options.ConnectionString = connectionString)

// Endpoints

.ConfigureEndpoints(siloPort: 11111, gatewayPort: 30000)

// Application parts: just reference one of the grain implementations that we use

.ConfigureApplicationParts(parts => parts.AddApplicationPart(typeof(ValueGrain).Assembly).WithReferences())

// Now create the silo!

.Build();

我们将分析此示例中使用的步骤：

#### 奥尔良聚类信息

[...]

// Clustering information

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "orleans-docker";

options.ServiceId = "AspNetSampleApp";

})

[...]

这里我们设置两件事：

* 在ClusterId到"my-first-cluster"：这是为新奥尔良集群的唯一ID。使用此ID的所有客户端和孤岛将能够直接相互通信。例如，有些人会选择ClusterId为每个部署使用不同的。
* 在ServiceId到"AspNetSampleApp"：这是你的应用程序，将被一些供应商使用（例如用于持久性提供者）的唯一ID。**跨部署，此ID应该是稳定的（不会更改）**。

#### 群集提供者

[...]

// Clustering provider

.UseAzureStorageClustering(options => options.ConnectionString = connectionString)

[...]

通常，构建在Orleans上的服务部署在节点集群上，可以在专用硬件上，也可以在Azure中。对于开发和基本测试，Orleans可以部署在单节点配置中。当部署到节点集群时，Orleans在内部实现了一组协议，以发现和维护集群中Orleans孤岛的成员资格，包括检测节点故障和自动重​​新配置。

为了可靠地管理集群成员资格，Orleans使用Azure Table，SQL Server或Apache ZooKeeper进行节点同步。

在此示例中，我们使用Azure Table作为成员资格提供程序。

#### 端点

var silo = new SiloHostBuilder()

[...]

// Endpoints

.ConfigureEndpoints(siloPort: 11111, gatewayPort: 30000)

[...]

奥尔良仓库通常配置两个端点：

* silo-to-silo端点，用于同一集群中的silo之间的通信
* 客户端到孤岛端点（或网关），用于同一群集中的客户端和孤岛之间的通信。

在该示例中，我们使用辅助方法.ConfigureEndpoints(siloPort: 11111, gatewayPort: 30000)设置用于筒仓到筒仓通信的端口以及用于网关的端口11111和30000。此方法将检测哪个接口侦听。

在大多数情况下，此方法应该足够了，但如果需要，可以进一步自定义。例如，如果要将外部IP地址与某些端口转发一起使用，请执行以下操作：

[...]

.Configure<EndpointOptions>(options =>

{

// Port to use for Silo-to-Silo

options.SiloPort = 11111;

// Port to use for the gateway

options.GatewayPort = 30000;

// IP Address to advertise in the cluster

options.AdvertisedIPAddress = IPAddress.Parse("172.16.0.42");

// The socket used for silo-to-silo will bind to this endpoint

options.GatewayListeningEndpoint = new IPEndPoint(IPAddress.Any, 40000);

// The socket used by the gateway will bind to this endpoint

options.SiloListeningEndpoint = new IPEndPoint(IPAddress.Any, 50000);

})

[...]

在内部，孤岛将会收听0.0.0.0:40000，0.0.0.0:50000但会员提供商发布的价值将是172.16.0.42:11111和172.16.0.42:30000。

#### 应用部分

[...]

// Application parts: just reference one of the grain implementations that we use

.ConfigureApplicationParts(parts => parts.AddApplicationPart(typeof(ValueGrain).Assembly).WithReferences())

[...];

虽然此步骤在技术上并不需要（如果未配置，Orleans将扫描当前文件夹中的所有程序集），建议开发人员配置此步骤。此步骤将帮助Orleans加载用户程序集和类型。这些程序集称为应用程序部件。使用应用程序部件发现所有Grains，Grain Interfaces和Serializer。

应用部件使用的是配置IApplicationPartsManager，这可以使用被访问ConfigureApplicationParts扩展方法上IClientBuilder和ISiloHostBuilder。该ConfigureApplicationParts方法接受委托，Action<IApplicationPartManager>。

IApplicationPartManager支持常用的以下扩展方法：

* AddApplicationPart(assembly) 可以使用此扩展方法添加单个程序集。
* AddFromAppDomain()添加当前加载的所有程序集AppDomain。
* AddFromApplicationBaseDirectory()加载并添加当前基本路径中的所有程序集（请参阅参考资料AppDomain.BaseDirectory）。

通过上述方法添加的程序集可以使用以下扩展方法对其返回类型进行补充IApplicationPartManagerWithAssemblies：

* WithReferences()从添加的部分添加所有引用的程序集。这会立即加载任何可传递引用的程序集。装配加载错误将被忽略。
* WithCodeGeneration()为添加的零件生成支持代码并将其添加到零件管理器。请注意，这需要Microsoft.Orleans.OrleansCodeGenerator安装包，通常称为运行时代码生成。

类型发现要求提供的应用程序部件包含特定属性。将构建时代码生成包（Microsoft.Orleans.OrleansCodeGenerator.Build）添加到包含Grains，Grain Interfaces或Serializers的每个项目是建议的方法，以确保存在这些属性。构建时代码生成仅支持C＃。对于F＃，Visual Basic和其他.NET语言，可以通过上述WithCodeGeneration()方法在配置时生成代码。

### 典型配置

以下是可用于开发和生产部署的典型配置示例。

#### 地方发展

请参阅[本地开发配置](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/local_development_configuration.html)

#### 使用Azure进行可靠的生产部署

要使用Azure进行可靠的生产部署，您需要使用Azure表选项来实现集群成员身份。此配置是部署到内部部署服务器，容器或Azure虚拟机实例的典型配置。

DataConnection字符串的格式是 "DefaultEndpointsProtocol=https;AccountName=<Azure storage account>;AccountKey=<Azure table storage account key>"

筒仓配置：

// TODO replace with your connection string

const string connectionString = "YOUR\_CONNECTION\_STRING\_HERE";

var silo = new SiloHostBuilder()

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "Cluster42";

options.ServiceId = "MyAwesomeService";

})

.UseAzureStorageClustering(options => options.ConnectionString = connectionString)

.ConfigureEndpoints(siloPort: 11111, gatewayPort: 30000)

.ConfigureLogging(builder => builder.SetMinimumLevel(LogLevel.Warning).AddConsole())

.Build();

客户端配置：

// TODO replace with your connection string

const string connectionString = "YOUR\_CONNECTION\_STRING\_HERE";

var client = new ClientBuilder()

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "Cluster42";

options.ServiceId = "MyAwesomeService";

})

.UseAzureStorageClustering(options => options.ConnectionString = connectionString)

.ConfigureLogging(builder => builder.SetMinimumLevel(LogLevel.Warning).AddConsole())

.Build();

#### 使用SQL Server进行可靠的生产部署

对于使用SQL Server的可靠生产部署，需要提供SQL Server连接字符串。

筒仓配置：

// TODO replace with your connection string

const string connectionString = "YOUR\_CONNECTION\_STRING\_HERE";

var silo = new SiloHostBuilder()

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "Cluster42";

options.ServiceId = "MyAwesomeService";

})

.UseAdoNetClustering(options =>

{

options.ConnectionString = connectionString;

options.Invariant = "System.Data.SqlClient";

})

.ConfigureEndpoints(siloPort: 11111, gatewayPort: 30000)

.ConfigureLogging(builder => builder.SetMinimumLevel(LogLevel.Warning).AddConsole())

.Build();

客户端配置：

// TODO replace with your connection string

const string connectionString = "YOUR\_CONNECTION\_STRING\_HERE";

var client = new ClientBuilder()

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "Cluster42";

options.ServiceId = "MyAwesomeService";

})

.UseAdoNetClustering(options =>

{

options.ConnectionString = connectionString;

options.Invariant = "System.Data.SqlClient";

})

.ConfigureLogging(builder => builder.SetMinimumLevel(LogLevel.Warning).AddConsole())

.Build();

#### 专用服务器群集上的不可靠部署

要在可靠性不是问题时在专用服务器集群上进行测试，您可以利用MembershipTableGrain并避免依赖Azure Table。您只需将其中一个节点指定为主节点。

在筒仓上：

var primarySiloEndpoint = new IPEndpoint(PRIMARY\_SILO\_IP\_ADDRESS, 11111);

var silo = new SiloHostBuilder()

.UseDevelopmentClustering(primarySiloEndpoint)

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "Cluster42";

options.ServiceId = "MyAwesomeService";

})

.ConfigureEndpoints(siloPort: 11111, gatewayPort: 30000)

.ConfigureLogging(logging => logging.AddConsole())

.Build();

在客户端：

var gateways = new IPEndPoint[]

{

new IPEndPoint(PRIMARY\_SILO\_IP\_ADDRESS, 30000),

new IPEndPoint(OTHER\_SILO\_\_IP\_ADDRESS\_1, 30000),

[...]

new IPEndPoint(OTHER\_SILO\_\_IP\_ADDRESS\_N, 30000),

};

var client = new ClientBuilder()

.UseStaticClustering(gateways)

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "dev";

options.ServiceId = "AdventureApp";

})

.ConfigureLogging(logging => logging.AddConsole())

.Build();

### 配置ADO.NET提供程序

奥尔良的任何可靠部署都需要使用持久存储来保持系统状态，特别是奥尔良集群成员资格表和提醒。其中一个可用选项是通过ADO.NET提供程序使用SQL数据库。

为了将ADO.NET用于持久性，群集或提醒，需要将ADO.NET提供程序配置为silo配置的一部分，并且在群集的情况下，还需要将其配置为客户端配置的一部分。

silo配置代码应如下所示：

var siloHostBuilder = new SiloHostBuilder();

var invariant = "System.Data.SqlClient"; // for Microsoft SQL Server

var connectionString = "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Orleans;Integrated Security=True;Pooling=False;Max Pool Size=200;Asynchronous Processing=True;MultipleActiveResultSets=True";

//use AdoNet for clustering

siloHostBuilder.UseAdoNetClustering(options =>

{

options.Invariant = invariant;

options.ConnectionString = connectionString;

});

//use AdoNet for reminder service

siloHostBuilder.UseAdoNetReminderService(options =>

{

options.Invariant = invariant;

options.ConnectionString = connectionString;

});

//use AdoNet for Persistence

siloHostBuilder.AddAdoNetGrainStorage("GrainStorageForTest", options =>

{

options.Invariant = invariant;

options.ConnectionString = connectionString;

});

客户端配置代码应如下所示：

var siloHostBuilder = new SiloHostBuilder();

var invariant = "System.Data.SqlClient";

var connectionString = "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Orleans;Integrated Security=True;Pooling=False;Max Pool Size=200;Asynchronous Processing=True;MultipleActiveResultSets=True";

//use AdoNet for clustering

siloHostBuilder.UseAdoNetClustering(options =>

{

options.Invariant = invariant;

options.ConnectionString = connectionString;

});

其中ConnectionString设置为有效的AdoNet Server连接字符串。

为了使用ADO.NET提供程序进行持久性，提醒或集群，有一些用于创建数据库工件的脚本，将托管Orleans孤岛的所有服务器都需要访问这些工件。缺乏对目标数据库的访问是我们看到的开发人员犯的典型错误。

在AdoNet扩展nugets上安装或执行nuget还原后，脚本将被复制到项目目录\ OrleansAdoNetContent，其中每个受支持的ADO.NET扩展都有自己的目录。我们将AdoNet nugets拆分为每个功能点：Microsoft.Orleans.Clustering.AdoNet用于群集，Microsoft.Orleans.Persistence.AdoNet持久性和Microsoft.Orleans.Reminders.AdoNet提醒。

### 序列化和编写自定义序列化程序

Orleans拥有先进且可扩展的序列化框架。Orleans序列化谷物请求和响应消息中传递的数据类型以及谷物持久状态对象。作为此框架的一部分，Orleans会自动为这些数据类型生成序列化代码。除了为已经.NET可序列化的类型生成更有效的序列化/反序列化之外，Orleans还尝试为粒度接口中使用的类型生成序列化器，这些类型不是.NET可序列化的。该框架还包括一组用于常用类型的高效内置序列化器：列表，字典，字符串，基元，数组等。

Orleans的序列化程序有两个重要特性，它将它与许多其他第三方序列化框架区分开来：动态类型/任意多态和对象标识。

1. **动态类型和任意多态** - 奥尔良对粒度调用中可传递的类型没有任何限制，并维护实际数据类型的动态特性。这意味着，例如，如果粒度接口中的方法被声明为接受IDictionary但在运行时传递者通过SortedDictionary，接收器确实会得到SortedDictionary（尽管“静态契约”/粒度接口没有指定此行为）。
2. **维护对象标识** - 如果同一个对象在粒度调用的参数中传递多个类型，或者从参数中间接指向多次，则Orleans只会序列化一次。在接收方，Orleans将正确恢复所有引用，因此在反序列化之后，指向同一对象的两个指针仍然指向同一个对象。对象标识对于保存在以下场景中非常重要。想象一下，演员A正在向演员B发送一个包含100个条目的字典，并且字典中的10个键指向A侧的同一个对象obj。在不保留对象标识的情况下，B将接收100个条目的字典，其中10个键指向10个不同的obj克隆。保留对象标识后，B侧的字典看起来与A'完全相同

上述两种行为是由标准的.NET二进制序列化程序提供的，因此我们在奥尔良也支持这种标准和熟悉的行为非常重要。

#### 生成的序列化程序

Orleans使用以下规则来决定生成哪些序列化程序。规则是：

1）扫描所有引用核心奥尔良库的程序集中的所有类型。

2）在这些程序集中：为粒度接口方法签名或状态类签名中直接引用的类型或使用[Serializable]属性标记的任何类型生成序列化程序 。

3）此外，粒度接口或实现项目可以指向生成序列化的任意类型，方法是添加一个[KnownType]或[KnownAssembly]汇编级别属性，以告知代码生成器为程序集中的特定类型或所有符合条件的类型生成序列化程序。

#### 序列化提供商

Orleans支持使用提供者模型与第三方序列化程序集成。这需要实现IExternalSerializer本文档的自定义序列化部分中描述的类型。一些常见的序列化器的集成与奥尔良一起维护，例如：

* [协议缓冲区](https://developers.google.com/protocol-buffers/)：Orleans.Serialization.ProtobufSerializer来自[Microsoft.Orleans.OrleansGoogleUtils](https://www.nuget.org/packages/Microsoft.Orleans.OrleansGoogleUtils/)NuGet包。
* [Bond](https://github.com/microsoft/bond/)：Orleans.Serialization.BondSerializer来自[Microsoft.Orleans.Serialization.Bond](https://www.nuget.org/packages/Microsoft.Orleans.Serialization.Bond/) NuGet包。
* [Newtonsoft.Json AKA Json.NET](http://www.newtonsoft.com/json)：Orleans.Serialization.OrleansJsonSerializer来自奥尔良核心图书馆。

自定义实现IExternalSerializer在下面的“编写自定义序列化器”部分中描述。

#### 组态

确保所有客户端和孤岛上的序列化配置完全相同非常重要。如果配置不一致，则可能发生序列化错误。

IExternalSerializer可以使用代码的SerializationProviders属性ClientConfiguration和GlobalConfiguration代码来指定实现的序列化提供程序：

var cfg = new ClientConfiguration();

cfg.SerializationProviders.Add(typeof(FantasticSerializer).GetTypeInfo());

var cfg = new GlobalConfiguration();

cfg.SerializationProviders.Add(typeof(FantasticSerializer).GetTypeInfo());

或者，可以在以下<SerializationProviders />属性的XML配置中指定它们<Messaging>：

<Messaging>

<SerializationProviders>

<Provider type="GreatCompany.FantasticSerializer, GreatCompany.SerializerAssembly"/>

</SerializationProviders>

</Messaging>

在这两种情况下，都可以配置多个提供程序。该集合是有序的，这意味着如果提供者可以序列化类型A并且B在提供者之前指定只能序列化类型B，那么后者将不会被使用。

#### 编写自定义序列化器

除了自动序列化生成之外，应用程序代码还可以为其选择的类型提供自定义序列化。Orleans建议对大多数应用程序类型使用自动序列化生成，并且只有在您认为可以通过手动编码序列化程序来提高性能时，才会编写自定义序列化程序。本说明介绍了如何执行此操作，并在可能有用的情况下确定了一些特定情况。

应用程序可以通过3种方式自定义序列化：

1. 添加序列化方法，以你的类型和相应的属性（其标记CopierMethod，SerializerMethod，DeserializerMethod）。此方法适用于应用程序拥有的类型，即可以添加新方法的类型。
2. IExternalSerializer在配置期间实施并注册它。此方法对于集成外部序列化库非常有用。
3. 编写一个单独的静态类，使用[Serializer(typeof(YourType))]其中的3个序列化方法和与上面相同的属性进行注释。此方法对于应用程序不拥有的类型很有用，例如，应用程序无法控制的其他库中定义的类型。

以下各节详细介绍了这些方法中的每一种。

##### 介绍

奥尔良序列化分三个阶段进行：对象立即被深度复制以确保隔离; 在被戴上电线之前; 对象被序列化为消息字节流; 当传送到目标激活时，从接收的字节流中重新创建（反序列化）对象。可以在消息中发送的数据类型 - 即可以作为方法参数或返回值传递的类型 - 必须具有执行这三个步骤的关联例程。我们将这些例程统称为数据类型的序列化器。

一种类型的复印机是独立的，而串行器和解串器是一对可以协同工作的。您可以只提供自定义复印机，或只提供自定义序列化程序和自定义反序列化程序，或者您可以提供这三种自定义复制程序的自定义实现。

在筒仓启动时以及加载程序集时，都会为每种受支持的数据类型注册序列化程序。对于要使用的类型的自定义序列化程序例程，必须进行注册。序列化程序选择基于要复制或序列化的对象的动态类型。因此，不需要为抽象类或接口创建序列化器，因为它们永远不会被使用。

#### 何时考虑编写自定义序列化程序

手工制作的序列化程序例程很少能比生成的版本更好地执行。如果您想这样做，首先应考虑以下选项：

如果数据类型中的字段或属性不必序列化或复制，则可以使用该NonSerialized属性标记它们。这将导致生成的代码在复制和序列化时跳过这些字段。尽可能使用Immutable<T>＆[Immutable]避免复制不可变数据。有关详细信息，请参阅下面的优化复制部分。如果您避免使用标准通用集合类型，请不要这样做。Orleans运行时包含通用集合的自定义序列化程序，它们使用集合的语义来优化复制，序列化和反序列化。这些集合在序列化字节流中也有特殊的“缩写”表示，从而带来更多的性能优势。例如，Dictionary<string, string>List<Tuple<string, string>>。

自定义序列化程序可以提供显着性能增益的最常见情况是，在数据类型中编码的重要语义信息通过简单地复制字段值而不可用。例如，通过将数组视为索引/值对的集合，即使应用程序将数据保持为完全实现的数组以提高操作速度，通常可以更有效地序列化稀疏填充的数组。

在编写自定义序列化程序之前要做的一件事是确保生成的序列化程序确实会损害您的性能。分析在这里有所帮助，但更有价值的是运行端到端应力测试，使用不同的序列化负载来衡量系统级影响，而不是序列化的微观影响。例如，构建一个不传递参数或从grain方法得到的测试版本，只需使用两端的固定值，就可以放大序列化和复制对系统性能的影响。

#### 方法1：向类型添加序列化方法

所有序列化程序例程都应该实现为它们所操作的类或结构的静态成员。此处显示的名称不是必需的; 注册基于相应属性的存在，而不是基于方法名称。请注意，序列化程序方法不必是公共的。

除非您实现所有三个序列化例程，否则应使用该Serializable属性标记您的类型，以便为您生成缺少的方法。

##### 复印机

复制器方法使用以下Orleans.CopierMethod属性进行标记：

[CopierMethod]

static private object Copy(object input, ICopyContext context)

{

...

}

复制程序通常是最简单的序列化程序。它们采用一个对象，保证与定义复印机的类型相同，并且必须返回对象的语义等效副本。

如果，作为复制对象的一部分，需要复制子对象，最好的方法是使用SerializationManager的DeepCopyInner例程：

var fooCopy = SerializationManager.DeepCopyInner(foo, context);

使用DeepCopyInner而不是DeepCopy非常重要，以便维护完整复制操作的对象标识上下文。

##### 维护对象标识

复制例程的一个重要职责是维护对象标识。Orleans运行时为此提供了一个辅助类。在“手动”复制子对象之前（即，不通过调用DeepCopyInner），检查它是否已被引用如下：

var fooCopy = context.CheckObjectWhileCopying(foo);

if (fooCopy == null)

{

// Actually make a copy of foo

context.RecordObject(foo, fooCopy);

}

最后一行，即调用RecordObject，是必需的，以便将来可以正确找到与foo引用相同的对象的可能CheckObjectWhileCopying。

请注意，这只应该针对类实例，而不是结构实例或.NET基元（字符串，Uris，枚举）。

如果您使用DeepCopyInner复制子对象，则会为您处理对象标识。

##### 串行

序列化方法使用以下SerializerMethod属性进行标记：

[SerializerMethod]

static private void Serialize(object input, ISerializationContext context, Type expected)

{

...

}

与复制器一样，传递给序列化器的“输入”对象保证是定义类型的实例。“预期”类型可以忽略; 它基于有关数据项的编译时类型信息，并在更高级别用于在字节流中形成类型前缀。

序列化的子对象，使用SerializationManager的SerializeInner程序：

SerializationManager.SerializeInner(foo, context, typeof(FooType));

如果foo没有特定的预期类型，那么您可以为期望的类型传递null。

的BinaryTokenStreamWriter类提供了各种各样的用于写入数据到字节流的方法。可以通过context.StreamWriter属性获取类的实例。请参阅课程以获取文档。

##### 解串器

反序列化方法使用以下DeserializerMethod属性标记：

[DeserializerMethod]

static private object Deserialize(Type expected, IDeserializationContext context)

{

...

}

“预期”类型可以忽略; 它基于有关数据项的编译时类型信息，并在更高级别用于在字节流中形成类型前缀。要创建的对象的实际类型始终是定义反序列化器的类的类型。

反序列化的子对象，使用SerializationManager的DeserializeInner程序：

var foo = SerializationManager.DeserializeInner(typeof(FooType), context);

或者，或者：

var foo = SerializationManager.DeserializeInner<FooType>(context);

如果foo没有特定的预期类型，请使用非泛型DeserializeInner变量并传递null预期类型。

的BinaryTokenStreamReader类提供了各种各样的方法从字节流读取数据。可以通过context.StreamReader属性获取类的实例。请参阅课程以获取文档。

#### 方法2：编写序列化程序提供程序

在此方法中，您可以在客户端和孤岛上实现Orleans.Serialization.IExternalSerializer并将其添加到SerializationProviders属性中。配置详见上面的序列化提供程序部分。ClientConfigurationGlobalConfiguration

实现IExternalSerializer遵循Method 1上面针对序列化方法描述的模式，添加了Orleans用于确定序列化器是否支持给定类型的Initialize方法和IsSupportedType方法。这是接口定义：

public interface IExternalSerializer

{

/// <summary>

/// Initializes the external serializer. Called once when the serialization manager creates

/// an instance of this type

/// </summary>

void Initialize(Logger logger);

/// <summary>

/// Informs the serialization manager whether this serializer supports the type for serialization.

/// </summary>

/// <param name="itemType">The type of the item to be serialized</param>

/// <returns>A value indicating whether the item can be serialized.</returns>

bool IsSupportedType(Type itemType);

/// <summary>

/// Tries to create a copy of source.

/// </summary>

/// <param name="source">The item to create a copy of</param>

/// <param name="context">The context in which the object is being copied.</param>

/// <returns>The copy</returns>

object DeepCopy(object source, ICopyContext context);

/// <summary>

/// Tries to serialize an item.

/// </summary>

/// <param name="item">The instance of the object being serialized</param>

/// <param name="context">The context in which the object is being serialized.</param>

/// <param name="expectedType">The type that the deserializer will expect</param>

void Serialize(object item, ISerializationContext context, Type expectedType);

/// <summary>

/// Tries to deserialize an item.

/// </summary>

/// <param name="context">The context in which the object is being deserialized.</param>

/// <param name="expectedType">The type that should be deserialized</param>

/// <returns>The deserialized object</returns>

object Deserialize(Type expectedType, IDeserializationContext context);

}

#### 方法3：为单个类型编写序列化程序

在此方法中，您将编写一个使用属性注释的新类[SerializerAttribute(typeof(TargetType))]，其中TargetType是要序列化的类型，并实现3个序列化例程。如何编写这些例程的规则与方法1相同。奥尔良使用它[SerializerAttribute(typeof(TargetType))]来确定此类是序列化程序TargetType，如果该属性能够序列化多个类型，则可以在同一个类上多次指定该属性。以下是此类的示例：

public class User

{

public User BestFriend { get; set; }

public string NickName { get; set; }

public int FavoriteNumber { get; set; }

public DateTimeOffset BirthDate { get; set; }

}

[Orleans.CodeGeneration.SerializerAttribute(typeof(User))]

internal class UserSerializer

{

[CopierMethod]

public static object DeepCopier(object original, ICopyContext context)

{

var input = (User) original;

var result = new User();

// Record 'result' as a copy of 'input'. Doing this immediately after construction allows for

// data structures which have cyclic references or duplicate references.

// For example, imagine that 'input.BestFriend' is set to 'input'. In that case, failing to record

// the copy before trying to copy the 'BestFriend' field would result in infinite recursion.

context.RecordCopy(original, result);

// Deep-copy each of the fields.

result.BestFriend = (User)context.SerializationManager.DeepCopy(input.BestFriend);

result.NickName = input.NickName; // strings in .NET are immutable, so they can be shallow-copied.

result.FavoriteNumber = input.FavoriteNumber; // ints are primitive value types, so they can be shallow-copied.

result.BirthDate = (DateTimeOffset)context.SerializationManager.DeepCopy(input.BirthDate);

return result;

}

[SerializerMethod]

public static void Serializer(object untypedInput, ISerializationContext context, Type expected)

{

var input = (User) untypedInput;

// Serialize each field.

SerializationManager.SerializeInner(input.BestFriend, context);

SerializationManager.SerializeInner(input.NickName, context);

SerializationManager.SerializeInner(input.FavoriteNumber, context);

SerializationManager.SerializeInner(input.BirthDate, context);

}

[DeserializerMethod]

public static object Deserializer(Type expected, IDeserializationContext context)

{

var result = new User();

// Record 'result' immediately after constructing it. As with with the deep copier, this

// allows for cyclic references and de-duplication.

context.RecordObject(result);

// Deserialize each field in the order that they were serialized.

result.BestFriend = SerializationManager.DeserializeInner<User>(context);

result.NickName = SerializationManager.DeserializeInner<string>(context);

result.FavoriteNumber = SerializationManager.DeserializeInner<int>(context);

result.BirthDate = SerializationManager.DeserializeInner<DateTimeOffset>(context);

return result;

}

}

#### 序列化通用类型

例如，TargetType参数[Serializer(typeof(TargetType))]可以是开放泛型类型MyGenericType<>。在这种情况下，序列化程序类必须具有与目标类型相同的通用参数。Orleans将在运行时为序列化的每个具体MyGenericType<T>类型创建一个具体版本的序列化程序，例如，每个MyGenericType<int>和的一个类型MyGenericType<string>。

#### 编写序列化器和反序列化器的提示

通常，编写序列化器/解串器对的最简单方法是通过构造字节数组并将数组长度写入流，然后是数组本身进行序列化，然后通过反转进程进行反序列化。如果数组是固定长度的，则可以从流中省略它。当您拥有一个可以紧凑地表示并且没有可能重复的子对象的数据类型时，这种方法很有效（因此您不必担心对象标识）。

另一种方法是Orleans运行时对字典等集合采用的方法，适用于具有重要和复杂内部结构的类：使用实例方法访问对象的语义内容，序列化该内容，并通过设置语义来反序列化内容而不是复杂的内部状态。在这种方法中，内部对象使用SerializeInner编写并使用DeserializeInner读取。在这种情况下，通常也会编写自定义复印机。

如果您编写自定义序列化程序，并且它看起来像是对类中每个字段的SerializeInner调用序列，则不需要为该类定制序列化程序。

#### 后备序列化

Orleans支持在运行时传输任意类型，因此内置代码生成器无法确定将提前传输的整个类型集。此外，某些类型不能为它们生成序列化程序，因为它们不可访问（例如private）或具有不可访问的字段（例如readonly）。因此，需要及时地序列化类型，这些类型是意外的或者不能提前生成的序列化器。负责这些类型的序列化程序称为回退序列化程序。奥尔良附带两个后备序列发生器：

* Orleans.Serialization.BinaryFormatterSerializer它使用.NET的[BinaryFormatter](https://msdn.microsoft.com/en-us/library/system.runtime.serialization.formatters.binary.binaryformatter) ; 和
* Orleans.Serialization.ILBasedSerializer它在运行时发出[CIL](https://en.wikipedia.org/wiki/Common_Intermediate_Language)指令以创建序列化器，利用Orleans的序列化框架来序列化每个字段。这意味着如果不可访问类型MyPrivateType包含MyType具有自定义序列化程序的字段，则将使用该自定义序列化程序对其进行序列化。

可以使用客户端和孤岛上的FallbackSerializationProvider属性来配置回退序列化程序。ClientConfigurationGlobalConfiguration

var cfg = new ClientConfiguration();

cfg.FallbackSerializationProvider = typeof(FantasticSerializer).GetTypeInfo();

var cfg = new GlobalConfiguration();

cfg.FallbackSerializationProvider = typeof(FantasticSerializer).GetTypeInfo();

或者，可以在XML配置中指定回退序列化提供程序：

<Messaging>

<FallbackSerializationProvider type="GreatCompany.FantasticFallbackSerializer, GreatCompany.SerializerAssembly"/>

</Messaging>

BinaryFormatterSerializer 是默认的回退序列化程序。

#### 异常序列化

使用[回退序列化程序序列化](https://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/serialization.html#fallback-serialization)异常。使用默认配置BinaryFormatterSerializer是回退序列化程序，因此必须遵循[ISerializable模式](https://docs.microsoft.com/en-us/dotnet/standard/serialization/custom-serialization)，以确保在异常类型中正确序列化所有属性。

以下是具有正确实现的序列化的异常类型的示例：

[Serializable]

public class MyCustomException : Exception

{

public string MyProperty { get; }

public MyCustomException(string myProperty, string message)

: base(message)

{

this.MyProperty = myProperty;

}

public MyCustomException(string transactionId, string message, Exception innerException)

: base(message, innerException)

{

this.MyProperty = transactionId;

}

// Note: This is the constructor called by BinaryFormatter during deserialization

public MyCustomException(SerializationInfo info, StreamingContext context)

: base(info, context)

{

this.MyProperty = info.GetString(nameof(this.MyProperty));

}

// Note: This method is called by BinaryFormatter during serialization

public override void GetObjectData(SerializationInfo info, StreamingContext context)

{

base.GetObjectData(info, context);

info.AddValue(nameof(this.MyProperty), this.MyProperty);

}

}

#### 使用不可变类型优化复制

Orleans有一个功能可以用来避免与序列化包含不可变类型的消息相关的一些开销。本节介绍该功能及其应用，从相关位置的上下文开始。

#### 奥尔良的序列化

当调用grain方法时，Orleans运行时会生成方法参数的深层副本，并从副本中形成请求。这可以防止调用代码在将数据传递给被调用的grain之前修改参数对象。

如果被调用的grain位于不同的silo上，则副本最终被序列化为字节流并通过网络发送到目标silo，在那里它们被反序列化为对象。如果被调用的粒子在同一个筒仓上，则副本将直接传递给被调用的方法。

返回值的处理方式相同：首先复制，然后可能序列化和反序列化。

请注意，所有3个进程（复制，序列化和反序列化）都遵循对象标识。换句话说，如果您传递两次具有相同对象的列表，则在接收端您将获得一个列表，其中包含相同的对象两次，而不是两个具有相同值的对象。

#### 优化复制

在许多情况下，深度复制是不必要的。例如，一种可能的场景是Web前端，它从其客户端接收一个字节数组，并将该请求（包括字节数组）传递给一个谷粒进行处理。一旦将数组传递给粒子，前端进程就不对数组做任何事情; 特别是，它不会重用该数组来接收未来的请求。在粒度内部，解析字节数组以获取输入数据，但不进行修改。grain返回它创建的另一个字节数组，以传递回Web客户端; 它一返回就丢弃数组。Web前端将结果字节数组传递回其客户端，无需修改。

在这种情况下，不需要复制请求或响应字节数组。不幸的是，奥尔良运行时无法自行解决这个问题，因为它无法判断数组是否会在以后由Web前端或谷物进行修改。在所有可能的世界中，我们都有某种.NET机制来指示不再修改某个值; 由于缺乏这一点，我们为此添加了Orleans特定的机制：Immutable<T>包装类和[Immutable]属性。

#### 运用 Immutable<T>

该Orleans.Concurrency.Immutable<T>包装类被用来表示一个值可以被认为是不变的 也就是说，不会修改基础值，因此安全共享不需要复制。请注意，使用Immutable<T>意味着值的提供者和值的接收者都不会在将来修改它; 这不是片面的承诺，而是双方的共同承诺。

使用Immutable<T>很简单：在你的谷物界面中，而不是传递T，传递Immutable<T>。例如，在上述场景中，谷物方法是：

Task<byte[]> ProcessRequest(byte[] request);

变为：

Task<Immutable<byte[]>> ProcessRequest(Immutable<byte[]> request);

要创建一个Immutable<T>，只需使用构造函数：

Immutable<byte[]> immutable = new Immutable<byte[]>(buffer);

要获取不可变的值，请使用以下.Value属性：

byte[] buffer = immutable.Value;

#### 运用 [Immutable]

对于用户定义的类型，[Orleans.Concurrency.Immutable]可以将属性添加到类型中。这指示Orleans的序列化程序避免复制此类型的实例。以下代码段演示[Immutable]用于表示不可变类型。传输过程中不会复制此类型。

[Immutable]

public class MyImmutableType

{

public MyImmutableType(int value)

{

this.MyValue = value;

}

public int MyValue { get; }

}

#### 奥尔良的不变性

对于奥尔良的目的，不变性是一个相当严格的陈述：数据项的内容不会以任何可能改变项的语义含义的方式被修改，或者会干扰同时访问该项的另一个线程。确保这一点的最安全的方法是根本不修改项目：按位不变性，而不是逻辑不变性。

在某些情况下，放宽逻辑不变性是安全的，但必须注意确保变异代码是正确的线程安全的; 因为处理多线程是复杂的，并且在奥尔良的上下文中不常见，我们强烈建议不要采用这种方法，并建议坚持按位不变性。

### 序列化最佳实践

序列化在奥尔良有两个主要目的：

1. 作为在运行时在粒度和客户端之间传输数据的线格式。
2. 作为一种存储格式，用于持久存储长期数据以供日后检索。

奥尔良生成的序列化器因其灵活性，性能和多功能性而适用于第一个目的。它们不适合第二个目的，因为它们不是明确版本容忍的。建议用户为持久数据配置容易版本的序列化程序，如[协议缓冲区](https://developers.google.com/protocol-buffers/)。协议缓冲器经由支持Orleans.Serialization.ProtobufSerializer从[Microsoft.Orleans.OrleansGoogleUtils](https://www.nuget.org/packages/Microsoft.Orleans.OrleansGoogleUtils/) NuGet包。应使用所选特定序列化程序的最佳实践，以确保版本容差。可以使用SerializationProviders如上所述的配置属性配置第三方序列化程序。

### 启动任务

在许多情况下，一旦仓库变得可用，就需要自动执行某些任务。启动任务提供此功能。

一些用例包括但不限于：

* 启动后台计时器以执行定期内务处理任务
* 使用从外部后备存储下载的数据预加载一些缓存颗粒

启动期间从启动任务抛出的任何异常都将在孤岛日志中报告，并将停止孤岛。

这种快速失败的方法是Orleans处理孤岛启动问题的标准方法，旨在允许在测试阶段轻松检测到孤岛配置和/或引导逻辑的任何问题，而不是被静默忽略并在以后导致意外问题在筒仓生命周期中。

#### 配置启动任务

可以使用ISiloHostBuilder注册要在启动期间调用的委托或通过注册实现来配置启动任务IStartupTask。

#### 示例：注册代理

siloHostBuilder.AddStartupTask(

async (IServiceProvider services, CancellationToken cancellation) =>

{

// Use the service provider to get the grain factory.

var grainFactory = services.GetRequiredService<IGrainFactory>();

// Get a reference to a grain and call a method on it.

var grain = grainFactory.GetGrain<IMyGrain>(0);

await grain.Initialize();

});

#### 示例：注册IStartupTask实现

首先，我们必须定义一个实现IStartupTask：

public class CallGrainStartupTask : IStartupTask

{

private readonly IGrainFactory grainFactory;

public CallGrainStartupTask(IGrainFactory grainFactory)

{

this.grainFactory = grainFactory;

}

public async Task Execute(CancellationToken cancellationToken)

{

var grain = this.grainFactory.GetGrain<IMyGrain>(0);

await grain.Initialize();

}

}

然后，该实现必须注册ISiloHostBuilder：

siloHostBuilder.AddStartupTask<CallGrainStartupTask>();

### 关机

本文档介绍了如何在应用程序退出之前正常关闭Orleans孤岛，首先作为Console应用程序，然后作为Docker容器应用程序。

#### 优雅关机 - 控制台应用程序

以下代码显示如何正常关闭Orleans silo控制台应用程序以响应用户按Ctrl + C（生成Console.CancelkeyPress事件）。

通常，当该事件处理程序返回时，应用程序将立即退出，从而导致灾难性的奥尔良孤岛崩溃并丢失内存状态。但是在下面的示例代码中，我们设置a.Cancel = true;为在Orleans silo完成正常关闭之前阻止应用程序关闭。

using Microsoft.Extensions.Logging;

using Orleans.Configuration;

using Orleans.Hosting;

using System;

using System.Net;

using System.Threading;

using System.Threading.Tasks;

namespace MySiloHost {

class Program {

static readonly ManualResetEvent \_siloStopped = new ManualResetEvent(false);

static ISiloHost silo;

static bool siloStopping = false;

static readonly object syncLock = new object();

static void Main(string[] args) {

SetupApplicationShutdown();

silo = CreateSilo();

silo.StartAsync().Wait();

/// Wait for the silo to completely shutdown before exiting.

\_siloStopped.WaitOne();

}

static void SetupApplicationShutdown() {

/// Capture the user pressing Ctrl+C

Console.CancelKeyPress += (s, a) => {

/// Prevent the application from crashing ungracefully.

a.Cancel = true;

/// Don't allow the following code to repeat if the user presses Ctrl+C repeatedly.

lock (syncLock) {

if (!siloStopping) {

siloStopping = true;

Task.Run(StopSilo).Ignore();

}

}

/// Event handler execution exits immediately, leaving the silo shutdown running on a background thread,

/// but the app doesn't crash because a.Cancel has been set = true

};

}

static ISiloHost CreateSilo() {

return new SiloHostBuilder()

.Configure(options => options.ClusterId = "MyTestCluster")

/// Prevent the silo from automatically stopping itself when the cancel key is pressed.

.Configure<ProcessExitHandlingOptions>(options => options.FastKillOnProcessExit = false)

.UseDevelopmentClustering(options => options.PrimarySiloEndpoint = new IPEndPoint(IPAddress.Loopback, 11111))

.ConfigureLogging(b => b.SetMinimumLevel(LogLevel.Debug).AddConsole())

.Build();

}

static async Task StopSilo() {

await silo.StopAsync();

\_siloStopped.Set();

}

}

}

当然，还有许多其他方法可以实现相同的目标。下面显示了一种方式，流行的在线和误导性，不能正常工作。它不起作用，因为它在尝试首先退出的两个方法之间设置竞争条件：Console.CancelKeyPress事件处理程序方法和static void Main(string[] args)方法。当事件处理程序方法首先完成时（至少发生一半时间），应用程序将挂起而不是顺利退出。

class Program {

static readonly ManualResetEvent \_siloStopped = new ManualResetEvent(false);

static ISiloHost silo;

static bool siloStopping = false;

static readonly object syncLock = new object();

static void Main(string[] args) {

Console.CancelKeyPress += (s, a) => {

Task.Run(StopSilo);

/// Wait for the silo to completely shutdown before exiting.

\_siloStopped.WaitOne();

/// Now race to finish ... who will finish first?

/// If I finish first, the application will hang! :(

};

silo = CreateSilo();

silo.StartAsync().Wait();

/// Wait for the silo to completely shutdown before exiting.

\_siloStopped.WaitOne();

/// Now race to finish ... who will finish first?

}

static async Task StopSilo() {

await silo.StopAsync();

\_siloStopped.Set();

}

}

#### 优雅的关机 - Docker应用程序

要完成的。

## Orleans流

Orleans v.1.0.0增加了对编程模型的流扩展的支持。流扩展提供了一组抽象和API，使得思考和使用流更简单，更健壮。流扩展允许开发人员编写以结构化方式对一系列事件进行操作的响应式应用程序。流提供程序的可扩展性模型使编程模型与各种现有排队技术兼容并可在其中移植，例如[Event Hub](http://azure.microsoft.com/en-us/services/event-hubs/)，[ServiceBus](http://azure.microsoft.com/en-us/services/service-bus/)，[Azure Queues](http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-queues/)和[Apache Kafka](http://kafka.apache.org/)。无需编写特殊代码或运行专用进程来与这些队列进行交互。

### 编程模型

Orleans流程编程模型背后有许多原则。

1. Orleans流是虚拟的。也就是说，流始终存在。它没有明确地创建或销毁，它永远不会失败。
2. 流由流ID 标识，流ID只是由GUID和字符串组成的逻辑名称。
3. Orleans Streams允许在时间和空间上将数据的生成与其处理分离。这意味着流生产者和流消费者可能在不同的服务器上，在不同的时间，并将承受失败。
4. Orleans流轻盈而充满活力。Orleans Streaming Runtime旨在处理大量来自高速率的流。
5. Orleans流绑定是动态的。Orleans Streaming Runtime旨在处理Grain以高速连接和断开流的情况。
6. Orleans Streaming Runtime 透明地管理流消耗的生命周期。在应用程序订阅流之后，从那时起它将接收流的事件，即使存在故障。
7. Orleans流在Grain和Orleans客户之间均匀运作。

### 新要求

我们确定了流处理系统的4个基本要求，使其能够针对上述方案。

1. 灵活的流处理逻辑
2. 支持高度动态的拓扑
3. 细粒度的流粒度
4. 分配

#### 灵活的流处理逻辑

我们希望系统支持表达流处理逻辑的不同方式。我们上面提到的现有系统要求开发人员编写声明性数据流计算图，通常遵循函数式编程风格。这限制了处理逻辑的表现力和灵活性。Orleans流对处理逻辑的表达方式无动于衷。它可以表示为数据流（例如，通过[在.NET中](https://msdn.microsoft.com/en-us/data/gg577609.aspx)使用[Reactive Extensions（Rx）](https://msdn.microsoft.com/en-us/data/gg577609.aspx)）; 作为一个功能计划; 作为声明性查询; 或者在一般的命令逻辑中。逻辑可以是有状态的或无状态的，可能有也可能没有副作用，并且可以触发外部动作。所有权力都交给开发者。

#### 支持动态拓扑

我们希望系统允许动态演变的拓扑。我们上面提到的现有系统通常仅限于在部署时固定且不能在运行时进化的静态拓扑。在下面的数据流表达式示例中，一切都很简单，直到您需要更改它。

Stream.GroupBy(x=> x.key).Extract(x=>x.field).Select(x=>x+2).AverageWindow(x, 5sec).Where(x=>x > 0.8) \*

更改Where过滤器中的阈值条件，添加其他Select语句或在数据流图中添加另一个分支并生成新的输出流。在现有系统中，如果不拆除整个拓扑并从头开始重新启动数据流，这是不可能的。实际上，这些系统将检查现有计算，并能够从最新检查点重新启动。尽管如此，这种重新启动对于实时产生结果的在线服务来说是破坏性的并且代价高昂。当我们谈论大量这样的表达式以相似但不同的（每用户，每个deveice等）参数执行并且不断变化时，这种重新启动变得特别不切实际。

我们希望系统允许在运行时通过向计算图添加新链接或节点，或通过更改计算节点内的处理逻辑来演化流处理图。

#### 细粒度的流粒度

在现有系统中，最小的抽象单元通常是整个流（拓扑）。但是，我们的许多目标场景要求拓扑中的单个节点/链路本身是逻辑实体。这样，每个实体都可以独立管理。例如，在包括多个链路的大流拓扑中，不同的链路可以具有不同的特性，并且可以在不同的物理传输上实现。一些链接可以通过TCP套接字，而其他链接可以通过可靠的队列。不同的链接可以有不同的交付保证。不同的节点可以具有不同的检查点策略，并且它们的处理逻辑可以用不同的模型或甚至不同的语言来表达。在现有系统中通常不可能具有这种灵活性。

抽象和灵活性参数的单位类似于SoA（面向服务的体系结构）与Actors的比较。Actor系统允许更大的灵活性，因为每个系统本质上都是一个独立管理的“小型服务”。同样，我们希望系统允许这种细粒度控制。

#### 分配

当然，我们的系统应该具有**“良好的分布式系统”的**所有属性。那包括：

1. 可伸缩性 - 支持大量流和计算元素。
2. 弹性 - 允许根据负载添加/删除资源以增长/缩小。
3. 可靠性 - 适应故障
4. 效率 - 有效利用基础资源
5. 响应能力 - 实现近实时场景。