# 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。

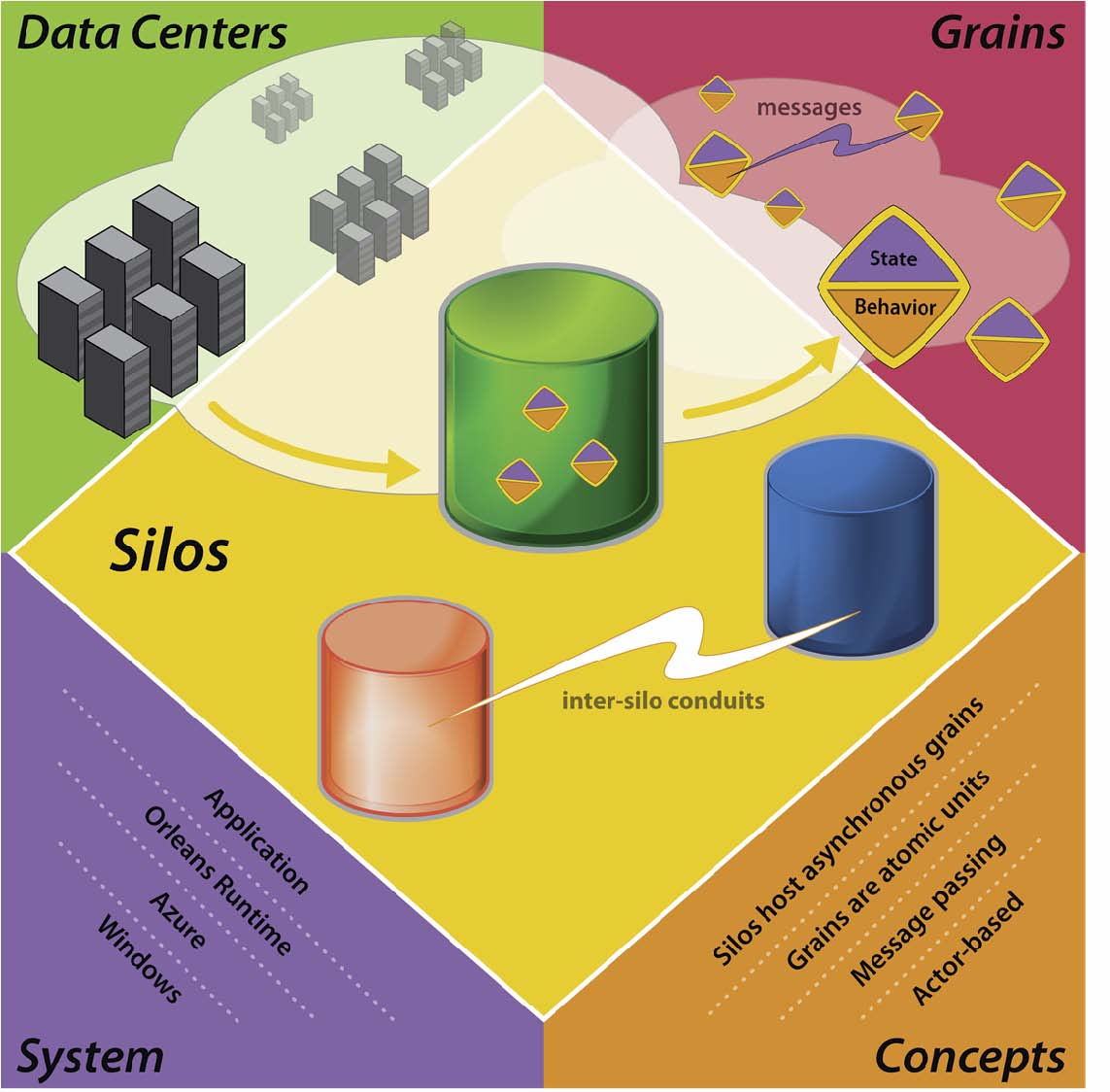
grain类型是一个简单的.NET类，它实现一个或多个应用程序定义的grain接口。单个grain实例是应用程序定义的grain类的实例，它们可以根据需要由服务器上的Orleans运行时自动创建，以处理对这些grain的请求。grains自然映射到大多数应用程序实体，例如用户，设备，会话，库存和订单。这使得构建面向对象的业务逻辑变得非常容易，但是在服务器集群中透明地扩展。每个grain在其应用逻辑选择的grain类型内具有稳定的逻辑标识（密钥），例如，用户电子邮件或设备ID或库存SKU代码。Orleans保证每个单独grain的单线程执行，从而保护应用程序逻辑免受并发和竞争的危险。

## 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。奥尔良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，奥尔良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谷物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，谷物的完整身份是由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());

从谷物代码中检索主键：

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谷物的周期性行为。

### 计时器

#### 定时器说明

**定时器**用于创建周期性的颗粒行为，不需要跨越多个激活（颗粒的实例化）。它基本上与标准相同。**NET System.Threading.Timer**类。此外，它在运行的颗粒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。
* 提醒与谷物相关，而不是任何特定的活化。
* 如果谷物没有与之关联的激活并且提醒滴答，则将创建一个谷物。例如：如果激活变为空闲且被停用，则与相同颗粒相关联的提醒将在下一个滴答时重新激活颗粒。
* 提醒通过消息传递，并且与所有其他谷物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**

使用提醒的谷物必须实现**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

## 重入

## 持久化

## 事件溯源

## 配置

[...]

// Clustering information

.Configure<ClusterOptions>(options =>

{

options.ClusterId = "orleans-docker";

options.ServiceId = "AspNetSampleApp";

})

[...]

* 在ClusterId：这是为Orleans集群的唯一ID。使用此ID的所有Client和Silo将能够直接相互通信。例如，有些人为每个部署使用不同的ClusterId。
* 在ServiceId：这是你的应用程序的唯一ID，将被一些Provider使用（例如用于persistence providers）。在整个部署中，此ID应该是稳定的（而不能更改）。

.ConfigureEndpoints(siloPort: 11111, gatewayPort: 30000)

Orleans silo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
* client-to-silo端点（或网关），用于同一群集中的客户端和孤岛之间的通信。
* [...]
* // 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。

## 序列化

Orleans两个重要特性

**动态类型和任意多态** ：Olearns对Grain调用中的传递类型没有任何限制，并保持原数据类型的运态特性，如接口方法中声明的是IDictionary，但实参是SortedDictionary，接收到的也是SortedDictionary类型。

**维护对象标识：**如过同一对象在Grain间传递多次，Orleans只序列化一次，在接收方，Orleans将正确恢复所有引用。

## 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。奥尔良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。