## Grain生命周期：

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

## 定义Grain

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()

{

}  
}

## 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。这个功能是对传统演员模型的重大改进

* **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。

### 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。