<http://proto.actor/docs/actors>

在微服务中，数据最终一致性的一个解决方案是通过有状态的Actor模型来达到，那什么是Actor模型呢？

Actor是并行的计算模型，包含状态，行为，并且包含一个邮箱，来异步处理消息。

关于Actor的介绍可参考：

<https://www.jianshu.com/p/449850aa8e82>

<https://www.jianshu.com/p/db04cab86ab9>

对于.net下的Actor模型有akka.net, Microsoft Orleans，在这里我们介绍的是另外一个Actor模型Proto.Actor

这次要说一下Proto.Actor，关于Proto.Actor的资料较少，这里有一篇可以作简单入门

<https://studygolang.com/p/protoactor>

为了便于开码友们理解，这个系统就以代码为主来学习，通过代码来“意会”Proto.Actor，所以这个系列叫《通过C#代码学Proto.Actor模型》

1. **代码第一篇**HelloWorld：

https://github.com/axzxs2001/ProtoActorSample/tree/master/ProtoActorSample/P001\_HelloWorld

引用NuGet：Proto.Actor

using Proto;

using System;

using System.Threading.Tasks;

namespace P001\_HelloWorld

{

    class Program

    {

        static void Main(string[] args)

        {

            //Actor产生一个props（道具）

            var props = Actor.FromProducer(() => new HelloActor());

            //从props衍生pid，pid代理一个actor的地址

            var pid = Actor.Spawn(props);

            //把Hello对象交给HelloActor处理

            pid.Tell(new Hello

            {

                Who = "World"

            });

            Console.ReadLine();

        }

    }

    //传递对象

    class Hello

    {

        public string Who;

    }

    //actor

    class HelloActor : IActor

    {

        //被调用

        public Task ReceiveAsync(IContext context)

        {

            var msg = context.Message;

            if (msg is Hello hello)

            {

                Console.WriteLine($"Hello {hello.Who}");

            }

            return Actor.Done;

        }

    }

}

可能你觉得就是一个控制台输出个Hello World，转了这么多弯；不要小看它，这可是一个支持分布式的Hello World，来看看后面经历了或可能经历了什么？

Actor lifecycle

注意上图中的Event，我们在后面了解中会遇到他们。现在可以在Actor的ReceiveAsync方法中加一段代码，来作个小实验：

if(msg is Started)

 {

       Console.WriteLine("Started");

 }

调时的话，会发现ReceiveAsync会在pid.Tell后执行两次，第一次Context.Message就是Started对象，第二次才是Hello对象。

不防在Main中调用一下pid.Stop()，跟踪一下ReceiveAsync会有什么一现……

## 《通过C#学Proto.Actor模型》之Prpos

在第一篇Proto.Actor博文中，HelloWorld的第一行真正代码是：

var props = Actor.FromProducer(() => new HelloActor());

这个返回的变量props就是一个Props的对象，它是负责创Actor实例，以及配置Actor实例，并且产Actor上下文Context（类似asp.net中的Context）。

Props对象产生通常是用Actor.FromProducer或Actor.FromFunc产生，不过也可能实例化，实体例时可以给Actor做一系列配置，如下代码：

using Proto;

using Proto.Mailbox;

using System;

using System.Threading.Tasks;

namespace P002\_CustomProps

{

class Program

{

static void Main(string[] args)

{

var props = new Props()

//用道具代理返回一个IActor实例

.WithProducer(() => new MyActor())

//默认调度器用线程池，邮箱中最多300个消息吞吐量

.WithDispatcher(new ThreadPoolDispatcher { Throughput = 300 })

//默认邮箱使用无界队列

.WithMailbox(() => UnboundedMailbox.Create())

//默认策略在10秒的窗口内最多重新启动子Actor 10次

.WithChildSupervisorStrategy(new OneForOneStrategy((who, reason) =>

SupervisorDirective.Restart, 10, TimeSpan.FromSeconds(10)))

//可以将中间件链接起来以拦截传入和传出消息

//接收中间件在Actor接收消息之前被调用

//发送者中间件在消息发送到目标PID之前被调用

.WithReceiveMiddleware(

next => async c =>

{

Console.WriteLine($"Receive中间件 1 开始，{c.Message.GetType()}:{c.Message}");

await next(c);

Console.WriteLine($"Receive中间件 1 结束");

},

next => async c =>

{

Console.WriteLine($"Receive中间件 2 开始，{c.Message.GetType()}:{c.Message}");

await next(c);

Console.WriteLine($"Receive中间件 2 结束");

})

.WithSenderMiddleware(

next => async (c, target, envelope) =>

{

Console.WriteLine($"Sender中间件 1 开始, {c.Message.GetType()}:{c.Message}");

await next(c, target, envelope);

Console.WriteLine($"Sender中间件 1 结束");

},

next => async (c, target, envelope) =>

{

Console.WriteLine($"Sender中间件 2 开始，{c.Message.GetType()}:{c.Message}");

await next(c, target, envelope);

Console.WriteLine($"Sender中间件 2 结束");

})

// 默认的 spawner 构造 Actor, Context 和 Process

.WithSpawner(Props.DefaultSpawner);

//从props衍生pid，pid代理一个actor的地址

var pid = Actor.Spawn(props);

//把Hello对象交给HelloActor处理

pid.Tell(new MyEntity

{

Message = "我是MyEntity的Message"

});

Console.ReadLine();

}

}

public class MyActor : IActor

{

public Task ReceiveAsync(IContext context)

{

if (context.Message is MyEntity myEntity)

{

Console.WriteLine(myEntity.Message);

}

return Actor.Done;

}

}

public class MyEntity

{

public string Message { get; set; }

}

}

你会发现，Demo中，总有一个实例类+一个Actor类，这是构成Actor模型的必备，Actor类就是主体，实体类就是Actor类运算的载体，所以它们总是如影随形。

第 行到第 行代码含意以后会解释。

**这里说Recevie和Sender的中间件，这里很像一个AOP，可以在调用某个行为前后作处理统一的处理，比如日志，权限等统一的规则**，也和asp.net core里的中件间如出一辙【如里你是一个asp.net core码友，熟悉的Context，熟悉的next()，还有熟悉的Middleware】。

现在可以**自己**运行一下分析一下结果吧

图片包含 文字

已生成极高可信度的说明

## 《通过C#学Proto.Actor模型》之Spawning

Props是配置Actor和实例化Actor，那实例化后，就应该访问了，Props.Actor提供了Actor.Spawn(),Actor.SpawnPrefix(),Actor.SpawnNamed()三个方法，来获取Actor实例，需要注意的是，这些方法返回的并不是真正的Actor对象，而是一个ProgressID，一个代表Actor对象的进程ID，缩写PID。

上代码：

using Proto;

using System;

using System.Threading;

using System.Threading.Tasks;

namespace P003\_SpawningActors

{

class Program

{

static void Main(string[] args)

{

var props = Actor.FromProducer(() => new MyActor());

//产生一个自定义名称的PID

var pid1 = Actor.Spawn(props);

pid1.Tell(new MyEntity { ID = 1 });

Thread.Sleep(1000);

Console.WriteLine("------------------------------------------");

//产生一个有gsw前缀，跟自动生成的名称的PID

var pid2 = Actor.SpawnPrefix(props, "gsw");

pid2.Tell(new MyEntity { ID = 2 });

Thread.Sleep(1000);

Console.WriteLine("------------------------------------------");

//产生一个名称为gswpid的PID

var pid3 = Actor.SpawnNamed(props, "gswpid");

pid3.Tell(new MyEntity { ID = 3 });

Console.ReadLine();

}

}

public class MyActor : IActor

{

public Task ReceiveAsync(IContext context)

{

if (context.Message is MyEntity myEntity)

{

Console.WriteLine($"父 SelfID={context.Self.Id} myEntity.ID={myEntity.ID}");

var cldProps = Actor.FromProducer(() => new MyChildActor());

//第一个子Actor

var pidCld1 = context.Spawn(cldProps);

pidCld1.Tell(new MyChildEntity { Message = "1 message,myEntity.ID=" + myEntity.ID });

//第二个子Actor

var pidCld2 = context.SpawnPrefix(cldProps, "gswCld");

pidCld2.Tell(new MyChildEntity { Message = "2 message,myEntity.ID=" + myEntity.ID });

//第三个子Actor

var pidCld3 = context.SpawnNamed(cldProps, "gswCldPid");

pidCld3.Tell(new MyChildEntity { ID = 3, Message = "3 message,myEntity.ID=" + myEntity.ID });

}

return Actor.Done;

}

}

public class MyChildActor : IActor

{

public Task ReceiveAsync(IContext context)

{

if (context.Message is MyChildEntity myChildEntity)

{

Console.WriteLine($"子 SelfID={context.Self.Id} Message={myChildEntity.Message}");

}

return Actor.Done;

}

}

public class MyEntity

{

public int ID { get; set; }

}

public class MyChildEntity

{

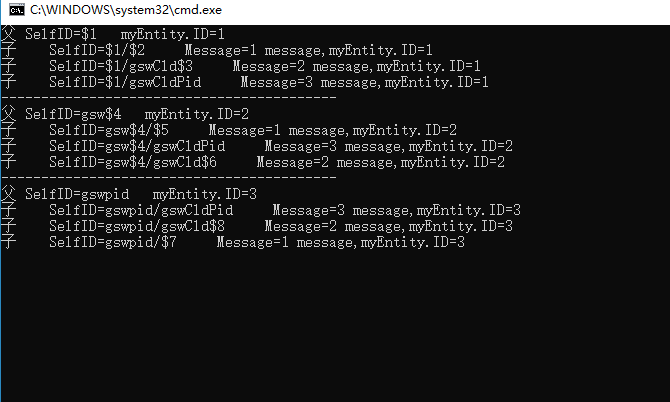
public string Message { get; set; }

public int ID { get; set; }

}

}

这个例子很简单，说明了三个Spawn的使用方式和Self.Id的特征，包括产生子Actor后，子Actor的Self.Id会带有父ID，结果如下：



## Actors

actor是State（状态），Behavior（行为），一个MailBox，Children 和一个Supervisor Strategy（主管策略）的容器，所有这些都封装在ActorRef中。

## PID-进行ID

如下所述，为了从Actor模型中受益，就需要将Actor对象封装起来。因此Actor使用PID表示外部，关且它可以无限制的传递对象。它分为内部和外部对象，可以透明的操作：重启acto也不需要重新引用，把actor放在远程主机上，向其他应用的actor发送信息。Actor内部状态是封闭的，除非actor不明智的对外发布这些信息。

## State

Actor里所含许多状态变理。可能是显式状态机（如FSM模块），也可能是计数器，监听器集合，待处理请求等。这些状态对于actor是最重要的，不通用受其他actor操作干扰。好消息是，Proto.Actor中所有actor都有自己的轻理级线程，是完全隔离的，这意味着，不需要使有锁同步访问，只需要编写自己的actor，不用担心并发性。

在后台，Proto.Actor将在真实线程上运行多组actor，有时会多个actor其享一个线程，并且一个actor的后续调用最终可能会在不同的线程上进行处理。Proto.Actor确保此实现细节不会影响处理actor状态的单线程。

因为actor的内部状态非常关键，所以肯有不一致的状态是致命的。因此，当actor失败并由其主管重启它时，将从头创建状态，就像首次创建actor一样。这是为了实现自我修改的能力。

可选项：通过持久保存收到的消息并在重新启动是重置它们，可以自动将actor的状态恢复到重启前的状态。

## Behavior

每次处理消息时，都会根据actor的当前行为进行匹配。Behavior是指定义在该 时间点对消息作出反应在的函数，如，客户端授权则转发，无授权则拒绝。此行为可能会随着时间的推移而发生变化，如，因为不同的客户端会随着时间的推移获得授权，或者因为参与者可能会进入“out-of-service”模式并稍后返回。这些更改是通过在State变量中对它们进行编码来实现的，而这些State是从Behavior的逻辑中读取的，或函数本身可以在运行时被换出，请参阅become和unbecome操作。但是，在构造actor对象期间定义的初始Behavior是特殊的，因为重新启动actor会将其Behavior重置为初始Behavior。

## Mailbox

Actor的目标是处理Message，这些Message是从其他Actor(或外部Actor系统)发送过来的，将发送者和接收者的关联起来的是Mailbox，每个Actor只有一个邮箱，所有发件人都将其Message排入队列。入队以发送操作的时间顺序发生，这意味着由于跨线程分配Actor的明显随机性，从不同的actor发送的消息可能在运行时没有定义的顺序。另一方面，从同一个actor向同一目标发送多个消息将以相同的顺序排列它们。

有不同的Emailbox实现可供选择，默认为FIFO：由actor处理的Message顺序与它们排队的顺序相匹配。这通常是一个很好的默认设置，但应用程序可能需要优先处理某些Message。在这种情况下，优先级Emailbox会按优先级来处理Message，在使用这种队列时，处理的Message的顺序会有算法定义，而不是FIFO。

Proto.Actor与其他一些Actor模型实现不同之处理在于，当前行为必须始终处理下一个出队列的Message，没有扫描Emailbox以查找下一个匹配的Message，无法处理通常视为失败，除非覆盖些Behavor。

## Children

每个Actor都是一个主管：如果它创建子任务，它将自动监督它们。子项列表保存在actor的上下文中，并且actor可以访问它。通过创建Context.Spawn()或停止child.Stop()子项来完成对列表的修改，产立即反映这些操作。实际的创建和终止操作以异步方式在幕后发生，因此它们不会“阻止”他们的主管。

## Supervisor Strategy

Actor最后处理子任务的故障。然后Proto.Actor透明的完成故障处理，对每个传入的故障应用监督和监控中描述的策略之一。由于此策略是Actor系统结构的基础，因此一旦创建了Actor，就无法更改。

考虑到每个Actor只有一个这样的策略，这就意味着如果不同的策略适用于一个参与者的各个子任务，那么这些子任务应该被分组在具有匹配策略的中间监督者之下，再次优先考虑Actor系统的结构，。将任务拆分为子任务。

## When an Actor Terminates

一旦一个actor终止，即以一种未被重启处理的方式失败，自行停止或被其主管停止，它将释放其资源，将所有剩余的消息从其邮箱中排放到系统的“死信邮箱”中。将他们转发给EventStream 作为DeadLetters，然后，使用系统Emailbox在Actor引用中替换Emailbox，将所有新邮件重定向到EventStream作为DeadLetters。这是在尽力而为的基础上完成的，但不要依赖它来构建“保证交付”

<http://proto.actor/docs/messages>

## Messages

消息是发送到特定目标的数据项。事件是组件在达到给定状态时发出的信号。在消息驱动的系统中，可寻址的接收者等待消息的到达并对它们作出反应，否则处于休眠状态。

## How are messages defined?

Proto.Actor允许您自动将这些消息传递给任何actor，无论它是在应用程序本地进程内运行的actor还是在不同机器上运行的远程actor。Proto.Actor可以自动序列化并将您的邮件路由到其预期的收件人（s。）

## Persistence

您可以选择让actor通过使用Proto.Persistence模块来保持其状态。这允许actor在启动时恢复它的状态并支持三种操作模式：

* 事件溯源
* 快照
* 带快照的事件采购

事件溯源

在使用事件溯源时，状态的每次变更都会被建模成恢复和运行时的actor事件，Persistence插件将Action<Event> applyEvent方法作为参数在每次保存事件时调用，或者在恢复期间从底层存储加载。重要的是，在此ApplyEvent方法中定义所有状态更改，包括转换为不同的行为。