<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

{

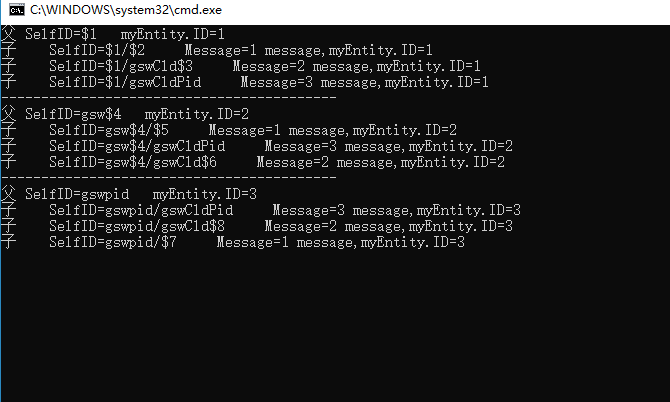
public string Message { get; set; }

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：



## 《通过C#学Proto.Actor模型》之PID

PID对象是代表Actor对象的进程，是能过Actor.Spawn(props)获取的；它有什么成员呢？既然代理Actor，首先有一个ID，标识自己是谁，Actor在Spawn时可以命名这个ID，否则会自动生成。还有三种向邮箱发消息的方法，Tell()，Request()，RequestAsync()，还有一个发送系统消息(Started,Stoping,Stoped等)方法SendSystemMessage()，还有一个停止的方法Stop()。

码友看代码：

using Proto;

using System;

using System.Threading;

using System.Threading.Tasks;

namespace P004\_PID

{

class Program

{

static void Main(string[] args)

{

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

var pid = Actor.Spawn(props);

while (true)

{

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Console.WriteLine("1、单向请求Tell 2、单向请求Request 3、双向请求RequestAsync");

switch (Console.ReadLine())

{

case "1":

Console.WriteLine("单向请求开始");

pid.Tell(new Request { Name = "单向请求 Tell", RequestType = "one-way", Time = DateTime.Now });

break;

case "2":

Console.WriteLine("单向请求开始");

//无法接回应签，与官网说法不一

pid.Request(new Request { Name = "单向请求 Request", RequestType = "two-way-1", Time = DateTime.Now }, pid);

break;

case "3":

Console.WriteLine("双向请求开始");

var response = pid.RequestAsync<Response>(new Request { Name = "双向请求 RequestAsync", RequestType = "two-way-2", Time = DateTime.Now }).Result;

Console.WriteLine(response.Time + ":" + response.Name);

break;

}

Thread.Sleep(2000);

}

}

}

public class MyActor : IActor

{

public Task ReceiveAsync(IContext context)

{

if (context.Message is Request request)

{

switch (request.RequestType)

{

case "one-way"://context.Sender为null

Console.WriteLine("接收到：" + request.RequestType + "," + request.Time + ":" + request.Name);

break;

case "two-way-1"://context.Sender= context.Self为自己

Console.WriteLine("接收到：" + request.RequestType + "," + request.Time + ":" + request.Name);

context.Respond(new Response() { Time = DateTime.Now, Name = "服务端应答 two-way-1" });

break;

case "two-way-2"://context.Sender!= context.Self为新实例

Console.WriteLine("接收到：" + request.RequestType + "," + request.Time + ":" + request.Name);

context.Respond(new Response() { Time = DateTime.Now, Name = "服务端应答 two-way-2" });

break;

}

}

return Actor.Done;

}

}

public class Request

{

public string Name

{ get; set; }

public string RequestType

{ get; set; }

public DateTime Time

{ get; set; }

}

public class Response

{

public string Name

{ get; set; }

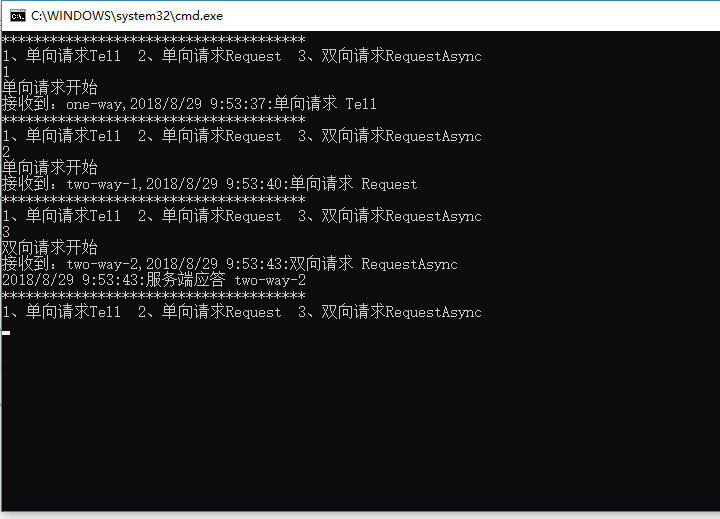
public DateTime Time

{ get; set; }

}

}

Actor中的Receive接到消息后，Context是中有两个PID对象，一个Self，一个Sender，Tell方法到达后，Sender对象为空；Request到达后，Sender=Self；而RequestAsync则Sender,Self都有对象，但不相同，这是一个区别。再有就是Tell和Request都是单向调用（我实测与官方文档说明有出入），RequestAsync是可以有返回值的，详见代码，如下是运行结果：



## 《通过C#学Proto.Actor模型》之Mailbox

邮箱是Actor模型的一个重要组成部分，负责接收发过来的消息，并保存起来，等待Actor处理。邮箱中维护着两种队列，一种是存系统消息，另一个是存用户消息，系统省是指Started,Stoping,Stoped之类的，用户当然指我们自定义的Actor。

另外，我们可以通过实现IMailboxStatistics接口，来获取邮箱的状态变更，并且可以有多个IMailboxStatistics实现。

码友看代码：

using Proto;

using Proto.Mailbox;

using System;

using System.Threading.Tasks;

namespace P005\_Mailboxes

{

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())

//默认邮箱使用无界队列

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

// 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 = "this is 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; }

}

public class MyMailboxStatistics : IMailboxStatistics

{

public void MailboxEmpty()

{

Console.WriteLine("邮箱MailboxEmpty");

}

public void MailboxStarted()

{

Console.WriteLine("邮箱MailboxStarted");

}

public void MessagePosted(object message)

{

Console.WriteLine("邮箱MessagePosted:"+message);

}

public void MessageReceived(object message)

{

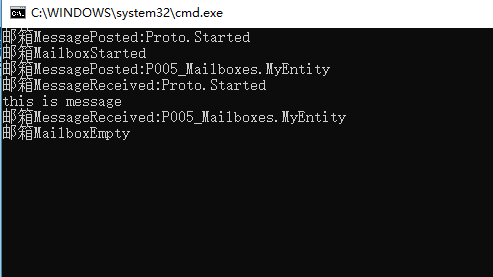
Console.WriteLine("邮箱MessageReceived:"+message);

}

}

}

当消息Posted时，Started时，Received时，邮箱为空时，这些方法会被先后调用，这里可对消息作处理。



## 《通过C#学Proto.Actor模型》之Supervision

Supervision，字面意思是监督，是父Actor发现子Actor有异常发生后，对子Actor产用保种策略处理的机制，如果父Actor不处理，则往上传递。

**子Actor发生异常后处理的策略有：**

Resume：立即恢复

Restart：恢复之前停止Actor并重新创建它

Stop：停止

Escalate：上报到父级

Supervisor的通过Props.WithChildSupervisorStarategy()来实施的，这个方法第一个参数是接收异常和Actor的PID，以便按一定方式处理异常，第二个参数是重启的次数，第三个参数是两次重启的间隔（按照官方文档，两个连续重启时间大于间隔时间，次数会复位，但实则只按次数走，间隔不起作用）达到最大次数据，子Actor将重新生成。

码友看代码：

using Proto;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace P006\_Supervision

{

class Program

{

static void Main(string[] args)

{

//设置监督策略，指向SupervisiorMode方法，重启次数为3，两次之间间隔1s

var props = Actor.FromProducer(() => new ShopingCatActor()).WithChildSupervisorStrategy(new OneForOneStrategy(SupervisorMode.Decide, 3, TimeSpan.FromSeconds(1)));

var pid = Actor.Spawn(props);

var user = new User { UserName = "gsw" };

var sn = 1;

while (true)

{

Console.WriteLine($"{sn++}--------------------begin-----------------");

foreach (var goods in user.ShopingCat.Goodses)

{

Console.WriteLine(goods);

}

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

Console.ReadLine();

pid.Request(user, pid);

}

}

}

/// <summary>

/// Supervisor 模式

/// </summary>

class SupervisorMode

{

/// <summary>

/// 用来处理异常发生的后续操作

/// </summary>

/// <param name="pid">PID</param>

/// <param name="reason">异常原因，Exception类型</param>

/// <returns></returns>

public static SupervisorDirective Decide(PID pid, Exception reason)

{

Console.WriteLine(" 异常发生：" + reason.Message + " " + pid);

switch (reason)

{

//重新开始的异常

case RecoverableException \_:

return SupervisorDirective.Restart;

//停止异常

case FatalException \_:

return SupervisorDirective.Stop;

//其他都上报

default:

return SupervisorDirective.Escalate;

}

}

}

/// <summary>

/// 购物车actor

/// </summary>

class ShopingCatActor : IActor

{

ShopingCat \_shopingCat;

public ShopingCatActor()

{

\_shopingCat = new ShopingCat();

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*actor ShopingCatActor\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

public Task ReceiveAsync(IContext context)

{

PID childPid;

if (context.Children == null || context.Children.Count == 0)

{

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

childPid = context.Spawn(props);

}

else

{

childPid = context.Children.First();

}

switch (context.Message)

{

case User user:

childPid.Request(\_shopingCat, childPid);

user.ShopingCat = \_shopingCat;

break;

}

return Actor.Done;

}

}

/// <summary>

/// 商品actor

/// </summary>

class GoodsActor : IActor

{

public GoodsActor()

{

Console.WriteLine(" \*\*\*\*\*\*\*ctor GoodsActor\*\*\*\*\*\* ");

}

public Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

case ShopingCat shopingCat:

var goods = new Goods { Name = "红茶", Price = 3.0m, Describe = "统一" };

//用来随机产生异常

var random = new Random();

goods.Quantity = random.Next(1, 3) - 1;

if (goods.Quantity <= 0)

{

throw new RecoverableException("数量不能小于等于0");

}

else

{

shopingCat.Goodses.Add(goods);

Console.WriteLine($"添加 {goods} 到购物车里");

}

break;

}

return Actor.Done;

}

}

/// <summary>

/// 用户

/// </summary>

class User

{

public ShopingCat ShopingCat { get; set; } = new ShopingCat();

public string UserName { get; set; }

}

/// <summary>

/// 购物车

/// </summary>

class ShopingCat

{

public List<Goods> Goodses

{ get; set; } = new List<Goods>();

}

/// <summary>

/// 商品

/// </summary>

class Goods

{

public string Name { get; set; }

public int Quantity { get; set; }

public decimal Price { get; set; }

public string Describe { get; set; }

public override string ToString()

{

return $"Name={Name}，Quantity={Quantity}，Price={Price}，Describe={Describe}";

}

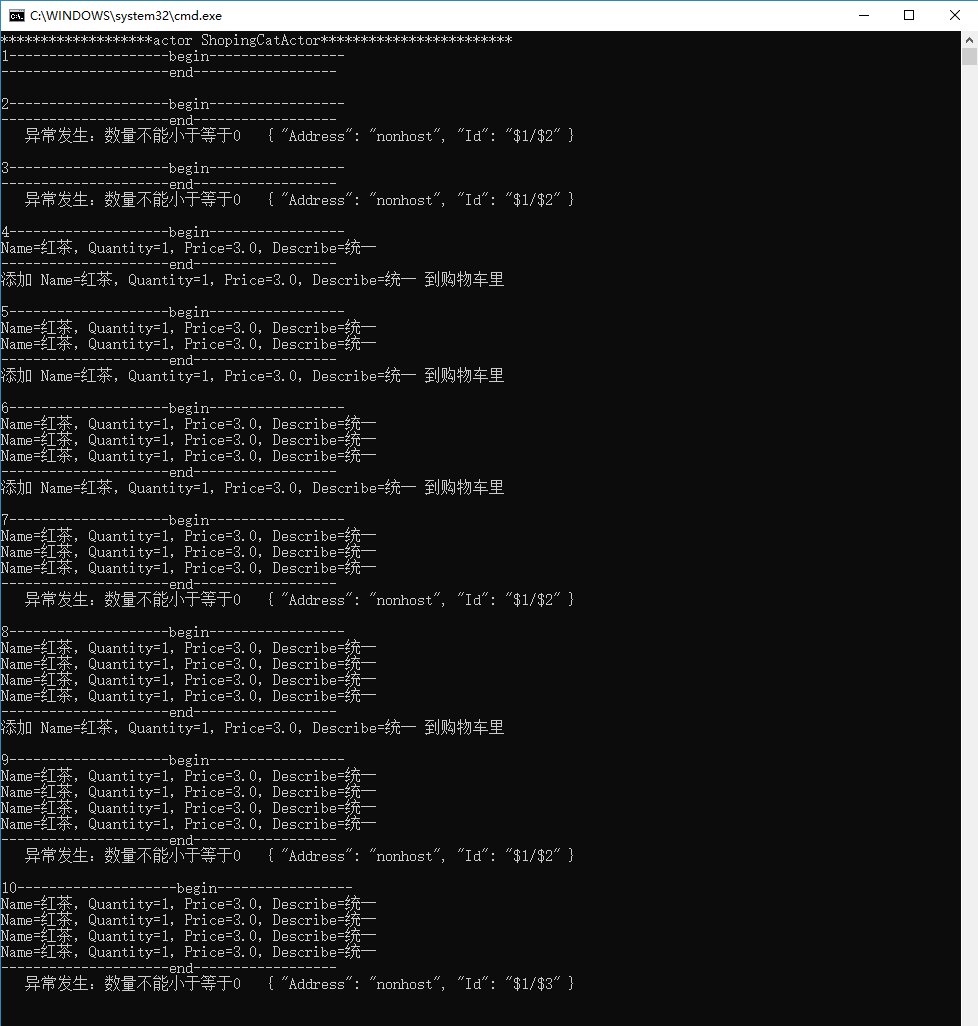
}

}

Demo中有三个实体类，用户类，购物车类，商品类，他们的关系一目了解，聚合关系。有两个Actor，购物车Actor（ShopingCatActor），商品Actor（GoodsActor），商品Actor是购物车Actor子Actor；ShopingCatActor下有监督策略，在

SupervisorMode下的Decide方法中，处理不同的异常，采用不同的Actor策略。GoodsActor中，添加商品时根据数量来决定是否产生异常。

看运行结果：



注意看ID $1/$2，是在发生四次异常后才生成$1/$3的（这个与文档说法有出入，也许是我理解有问题，是重启3次，不是启动3次）

## 《通过C#学Proto.Actor模型》之Behaviors

Behaviors就是Actor接收到消息后可以改变处理的方法，相同的Actor，每次调用，转到不同的Actor内方法执行，非常适合按流程进行的场景。Behaviors就通过在Actor内部实例化一个Behavior对象，然后通过这个对象的Become来切换执行的方法的。

码友看码：

using Proto;

using System;

using System.Threading.Tasks;

namespace P007\_Behaviors

{

    class Program

    {

        static void Main(string[] args)

        {

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

            var pid = Actor.Spawn(props);

            while (true)

            {

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

                Console.WriteLine("按开关");

                Console.ReadLine();

                var message = pid.RequestAsync<string>(new PressSwitch()).Result;

                Console.WriteLine(message);

                message = pid.RequestAsync<string>(new Touch()).Result;

                Console.WriteLine(message);

            }

        }

    }

    public class LightBulb : IActor

    {

        private readonly Behavior \_behavior;

        public LightBulb()

        {

            \_behavior = new Behavior();

            //把Off方法放入栈

            \_behavior.BecomeStacked(Off);

        }

        public Task ReceiveAsync(IContext context)

        {

            //切换到behavior指定的方法,来充当ReceiveAsync

            return \_behavior.ReceiveAsync(context);

        }

        /// <summary>

        /// 关

        /// </summary>

        /// <param name="context"></param>

        /// <returns></returns>

        private Task Off(IContext context)

        {

            switch (context.Message)

            {

                case PressSwitch \_:

                    context.Respond("打开");

                    \_behavior.Become(On);

                    break;

                case Touch \_:

                    context.Respond("凉的");

                    break;

            }

            return Actor.Done;

        }

        /// <summary>

        /// 开

        /// </summary>

        /// <param name="context"></param>

        /// <returns></returns>

        private Task On(IContext context)

        {

            switch (context.Message)

            {

                case PressSwitch \_:

                    context.Respond("关闭");

                    \_behavior.Become(Off);

                    break;

                case Touch \_:

                    context.Respond("烫手");

                    break;

            }

            return Actor.Done;

        }

    }

    class PressSwitch

    { }

    class Touch

    { }

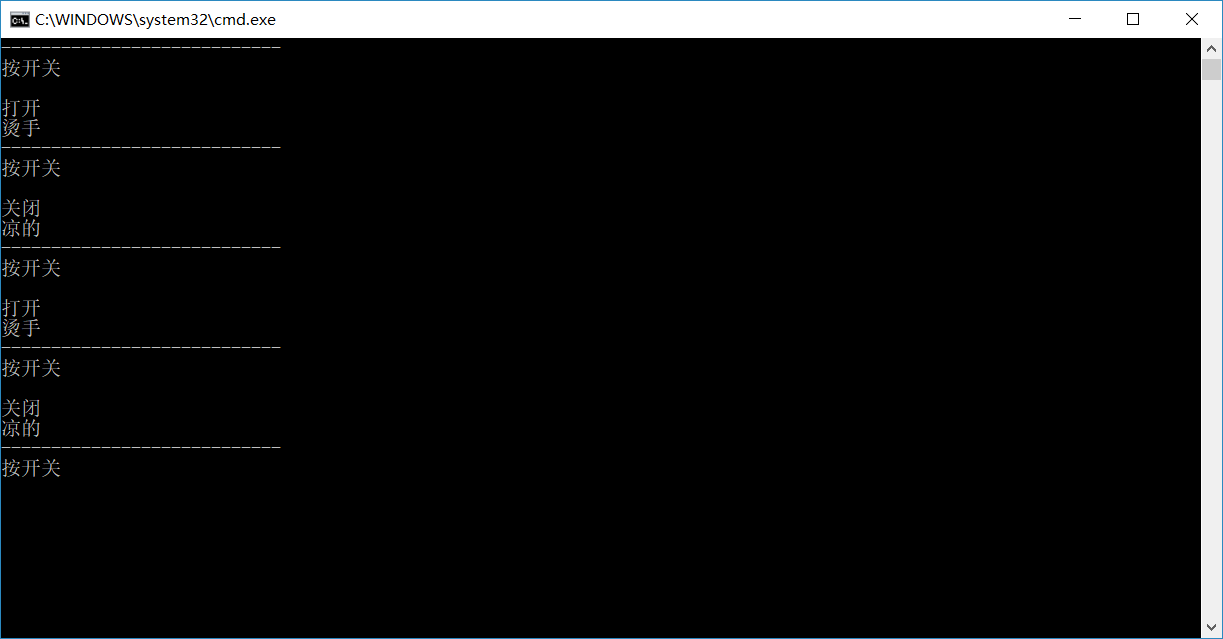
}

几个要点，在Actor构造中把Off放入方法栈，在Receive中调用Behavior的Receive，同时可以按自己业务逻辑切换方法，这样在一定程度上增加了Actor的灵活性，让一个Actor可以按设定流程实现不同行为。

Behavior提供了三个切换方法相关的方法：

* Become只需将传递的Receive方法设置为当前方法，替换默认方法。
* BecomeStacked将传递的Receive方法推送到方法堆栈，但保留以前的方法。
* UnbecomeStacked 恢复到以前使用的方法。

看结果：



## 《通过C#学Proto.Actor模型》之Persistence

Actor是有状态的，当每一步执行失败后，返回失败地方继续执行时，希望此时的状态是正确的，为了保证这一点，持久化就成了必要的环节了。

Proto.Actor提供了三种方式执久化：

* Event Sourcing事件溯源
* Snapshotting快照
* Event Sourcing with Snapshotting带快照的事件溯源

不管是那种持久化方式，首先要构造一个持久化的提供者，这个提者是内存也好，数据库也罢，本例中用Sqlite作为持久化的载体；在Actor中，实现持久化，首先要创建一个Persistence对象，用来将快照或事件保存起来，最重要的一点是，我们用事件溯源或快照，是帮我们保留住Actor在某刻的状，保留下来，以便我们再次启动时能延续这个状态，所以Persistence有一个很关键的作用就是能从持久化的载体中把原来的状态回复过来，这里，Event Source的把原来的状态步骤走再走一次，到达当前流程的点，但快照不然，直接取的是最后时刻的状态；带快照的事件溯源则是两者的结合。

码友看码：

NuGet安装

Proto.Actor

Proto.Persistence

Proto.Persistence.Sqlite

using Microsoft.Data.Sqlite;

using Proto;

using Proto.Persistence;

using Proto.Persistence.SnapshotStrategies;

using Proto.Persistence.Sqlite;

using System;

using System.Threading.Tasks;

namespace P008\_Persistence

{

    class Program

    {

        static void Main(string[] args)

        {

            //用sqlite持久化后

            var actorid = "myactorid";

            var dbfile = @"C:\MyFile\Source\Repos\ProtoActorSample\ProtoActorSample\P008\_Persistence\data.sqlite";

            var sqliteProvider = new SqliteProvider(new SqliteConnectionStringBuilder() { DataSource = dbfile });

            while (true)

            {

                Console.WriteLine("1、事件溯源   2、快照   3、带快照的事件溯源  4、退出");

                switch (Console.ReadLine())

                {

                    case "1":

                        CallEventSource(actorid, sqliteProvider);

                        break;

                    case "2":

                        CallSnapShoot(actorid, sqliteProvider);

                        break;

                    case "3":

                        CallSnapShootEventSource(actorid, sqliteProvider);

                        break;

                    case "4":

                        return;

                }

            }

        }

        /// <summary>

        /// 事件溯源

        /// </summary>

        /// <param name="actorid"></param>

        /// <param name="sqliteProvider"></param>

        private static void CallEventSource(string actorid, SqliteProvider sqliteProvider)

        {

            var props = Actor.FromProducer(() => new EventSourceDataActor(sqliteProvider, actorid));

            var pid = Actor.Spawn(props);

            var result = true;

            while (result)

            {

                Console.WriteLine("1、Tell  2、删除持久化  3、退出");

                switch (Console.ReadLine())

                {

                    case "1":

                        var random = new Random();

                        var no = random.Next(5, 15);

                        Console.WriteLine($"随机产生的数字：{no}");

                        pid.Tell(new Data { Amount = no });

                        break;

                    case "2":

                        //完成处理后清理持久化的操作

                        sqliteProvider.DeleteEventsAsync(actorid, 100).Wait();

                        break;

                    case "3":

                        result = false;

                        break;

                }

            }

        }

        /// <summary>

        /// 快照

        /// </summary>

        /// <param name="actorid"></param>

        /// <param name="sqliteProvider"></param>

        private static void CallSnapShoot(string actorid, SqliteProvider sqliteProvider)

        {

            var props = Actor.FromProducer(() => new SnapShootDataActor(sqliteProvider, actorid));

            var pid = Actor.Spawn(props);

            var result = true;

            while (result)

            {

                Console.WriteLine("1、Tell  2、删除持久化  3、退出");

                switch (Console.ReadLine())

                {

                    case "1":

                        var random = new Random();

                        var no = random.Next(5, 15);

                        Console.WriteLine($"随机产生的数字：{no}");

                        pid.Tell(new Data { Amount = no });

                        break;

                    case "2":

                        //完成处理后清理持久化的操作

                        sqliteProvider.DeleteEventsAsync(actorid, 100).Wait();

                        break;

                    case "3":

                        result = false;

                        break;

                }

            }

        }

        /// <summary>

        /// 快照事件溯源

        /// </summary>

        /// <param name="actorid"></param>

        /// <param name="sqliteProvider"></param>

        private static void CallSnapShootEventSource(string actorid, SqliteProvider sqliteProvider)

        {

            var props = Actor.FromProducer(() => new SnapShootEventSourceDataActor(sqliteProvider, sqliteProvider, actorid));

            var pid = Actor.Spawn(props);

            var result = true;

            while (result)

            {

                Console.WriteLine("1、Tell  2、删除持久化  3、退出");

                switch (Console.ReadLine())

                {

                    case "1":

                        var random = new Random();

                        var no = random.Next(5, 15);

                        Console.WriteLine($"随机产生的数字：{no}");

                        pid.Tell(new Data { Amount = no });

                        break;

                    case "2":

                        //完成处理后清理持久化的操作

                        sqliteProvider.DeleteEventsAsync(actorid, 100).Wait();

                        sqliteProvider.DeleteSnapshotsAsync(actorid, 100).Wait();

                        break;

                    case "3":

                        result = false;

                        break;

                }

            }

        }

    }

    public class Data

    {

        public long Amount { get; set; }

    }

    #region 事件溯源

    public class EventSourceDataActor : IActor

    {

        private long \_value = 0;

        private readonly Persistence \_persistence;

        public EventSourceDataActor(IEventStore eventStore, string actorId)

        {

            //事件溯源持久化方式

            \_persistence = Persistence.WithEventSourcing(eventStore, actorId, ApplyEvent);

        }

        private void ApplyEvent(Proto.Persistence.Event @event)

        {

            switch (@event.Data)

            {

                case Data msg:

                    \_value = \_value + msg.Amount;

                    Console.WriteLine($"累计：{\_value}");

                    break;

            }

        }

        public async Task ReceiveAsync(IContext context)

        {

            switch (context.Message)

            {

                case Started \_:

                    await \_persistence.RecoverStateAsync();

                    break;

                case Data msg:

                    await \_persistence.PersistEventAsync(new Data { Amount = msg.Amount });

                    break;

            }

        }

    }

    #endregion

    #region 快照

    public class SnapShootDataActor : IActor

    {

        private long \_value = 0;

        private readonly Persistence \_persistence;

        public SnapShootDataActor(ISnapshotStore snapshotStore, string actorId)

        {

            //快照持久化方式

            \_persistence = Persistence.WithSnapshotting(snapshotStore, actorId, ApplySnapshot);

        }

        private void ApplySnapshot(Proto.Persistence.Snapshot snapshot)

        {

            switch (snapshot.State)

            {

                case long value:

                    \_value = value;

                    Console.WriteLine($"累计：{\_value}");

                    break;

            }

        }

        public async Task ReceiveAsync(IContext context)

        {

            switch (context.Message)

            {

                case Started \_:

                    await \_persistence.RecoverStateAsync();

                    break;

                case Data msg:

                    \_value = \_value + msg.Amount;

                    await \_persistence.DeleteSnapshotsAsync(100);

                    await \_persistence.PersistSnapshotAsync(\_value);

                    break;

            }

        }

    }

    #endregion

    #region 事件溯源and快照

    public class SnapShootEventSourceDataActor : IActor

    {

        private long \_value = 0;

        private readonly Persistence \_persistence;

        public SnapShootEventSourceDataActor(IEventStore  eventStore, ISnapshotStore snapshotStore, string actorId)

        {

            //注释快照策略

            //\_persistence = Persistence.WithEventSourcingAndSnapshotting(eventStore, snapshotStore, actorId, ApplyEvent, ApplySnapshot, new IntervalStrategy(5), () => { return \_value; });

            //无快照策略

            \_persistence = Persistence.WithEventSourcingAndSnapshotting(eventStore, snapshotStore, actorId, ApplyEvent, ApplySnapshot);

        }

        private void ApplyEvent(Proto.Persistence.Event @event)

        {

            switch (@event.Data)

            {

                case Data msg:

                    \_value = \_value + msg.Amount;

                    Console.WriteLine($"事件溯源累计：{\_value}");

                    break;

            }

        }

        private void ApplySnapshot(Proto.Persistence.Snapshot snapshot)

        {

            switch (snapshot.State)

            {

                case long value:

                    \_value = value;

                    Console.WriteLine($"快照累计：{\_value}");

                    break;

            }

        }

        public async Task ReceiveAsync(IContext context)

        {

            switch (context.Message)

            {

                case Started \_:

                    await \_persistence.RecoverStateAsync();

                    break;

                case Data msg:

                    await \_persistence.PersistEventAsync(new Data { Amount = msg.Amount });

                    //无快照策略时启用

                    await \_persistence.PersistSnapshotAsync(\_value);

                    break;

            }

        }

    }

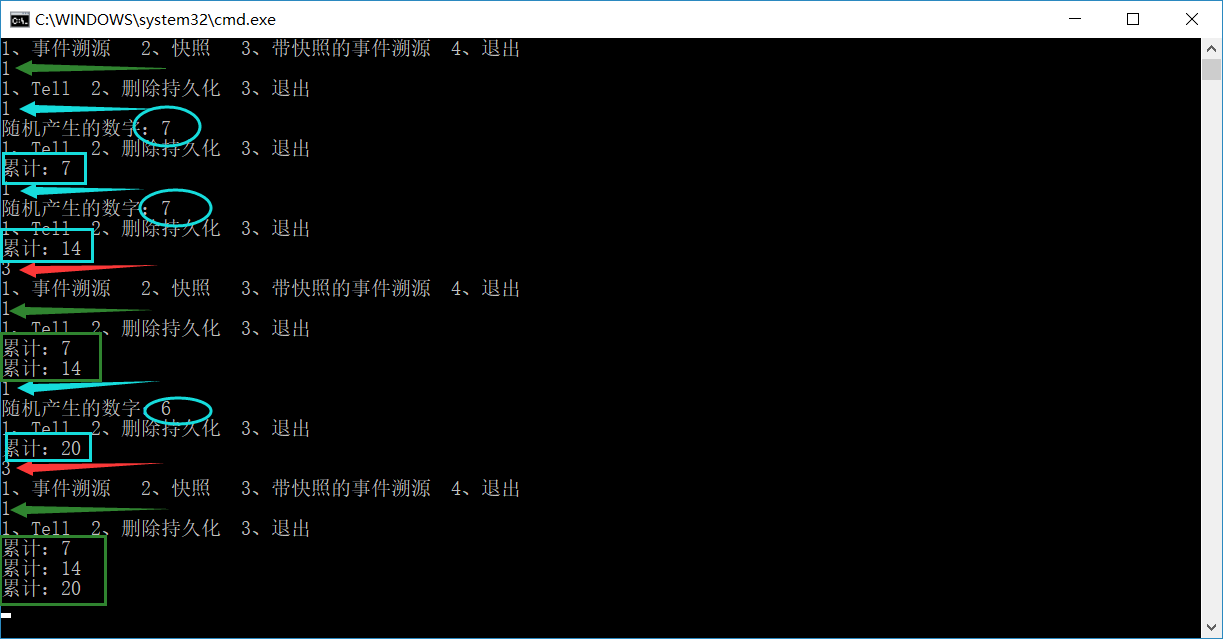
    #endregion

}

通过代码看到，持久化是通过在Actor中定义Persistence时，关联一个参数为，Event或Snapshot的方法，并且Actor的Receive方法在Stared到达是恢复（从持久载体中读取数据来恢复），在具体消息到达时，调用Persistence.PersistEventAsync或Persistence.PersisSnapshotAsync来持久化状态数据，这两个方法，都会把调用似递到Persistence产生是关联的那个方法，并把消息实体类通过Event.Data或Snapshot.State传递进去。

此例分别演示了事件溯源，快照，带快照事件溯源，例子很简单，就是把每次产生的随机数累加起来

1. 事件溯源



三个绿色箭头，意思是进了三次“1、事件溯源”这个选项

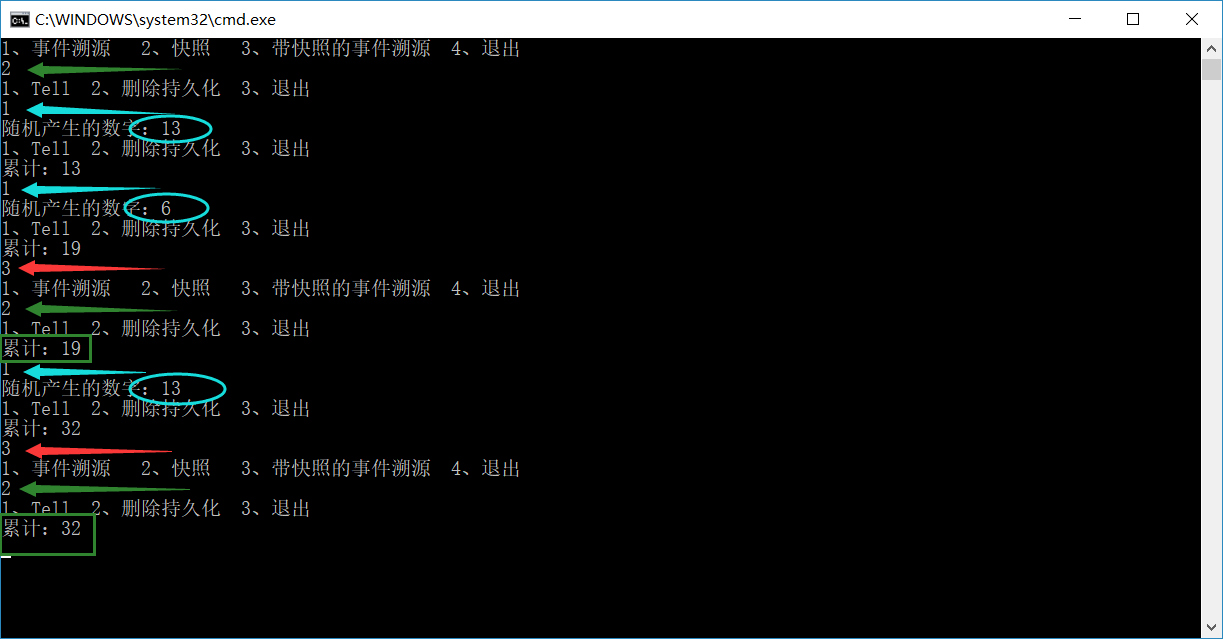
三次蓝色箭头，意思是调用了三次Tell方法，用来获取三次随机数，蓝色椭圆是产生的三个数字，分别是7，7，6，蓝色方框是累计结果，从上往下，第一次是(0+7)7，第二次是(7+7)14，第三次是(14+6)20

红色箭头是退出事件溯源的方法，返回上一级

绿色方框是绿色箭头再次进入，自动恢复，事件溯源后的结果（即从持久化载体中把之前的所有事件重新走一次）还是之前退出时的结果，累计20，所以不管这个Actor在什么地方退出，再次运行，都会把之前的补运行回来。

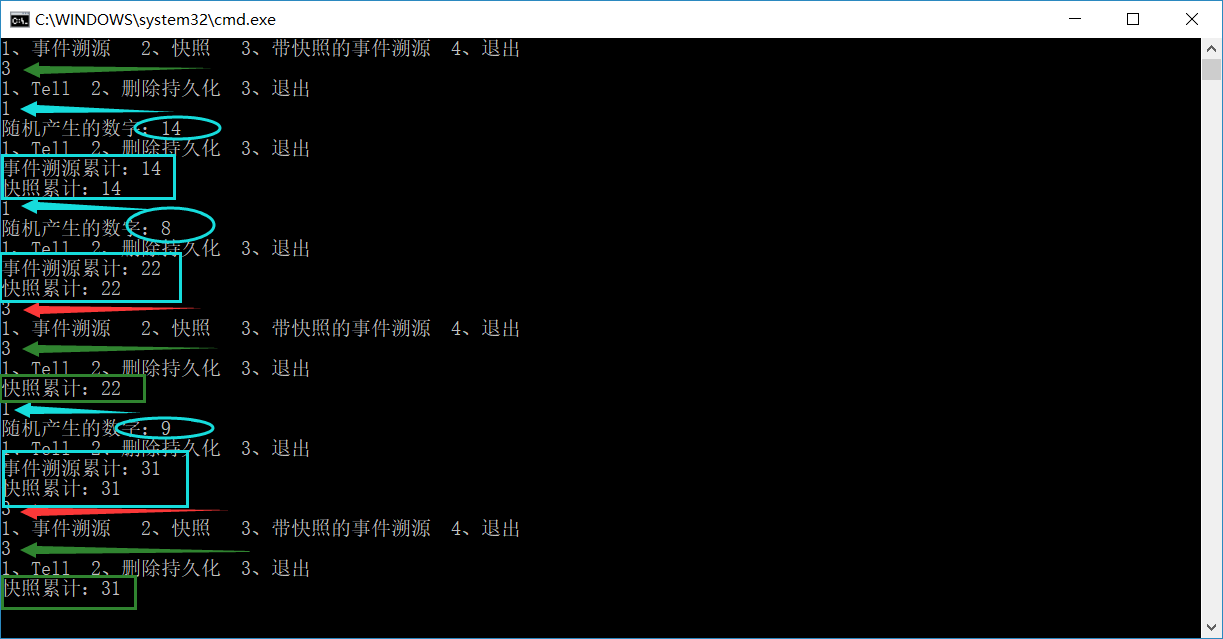
也可以打开sqlite库进行查看保存的事件结果

1. 快照



快照与事件溯源类似，差别在于每次再次进来，只取上次退出时的结果，同时，在数据里，只保存了最后一次的结果。

3、带快照的事件溯源



与快照类似，上面代码我们是一个事件，一个快照。

官方给出带快照的事件可以通过快照策略来保存快照

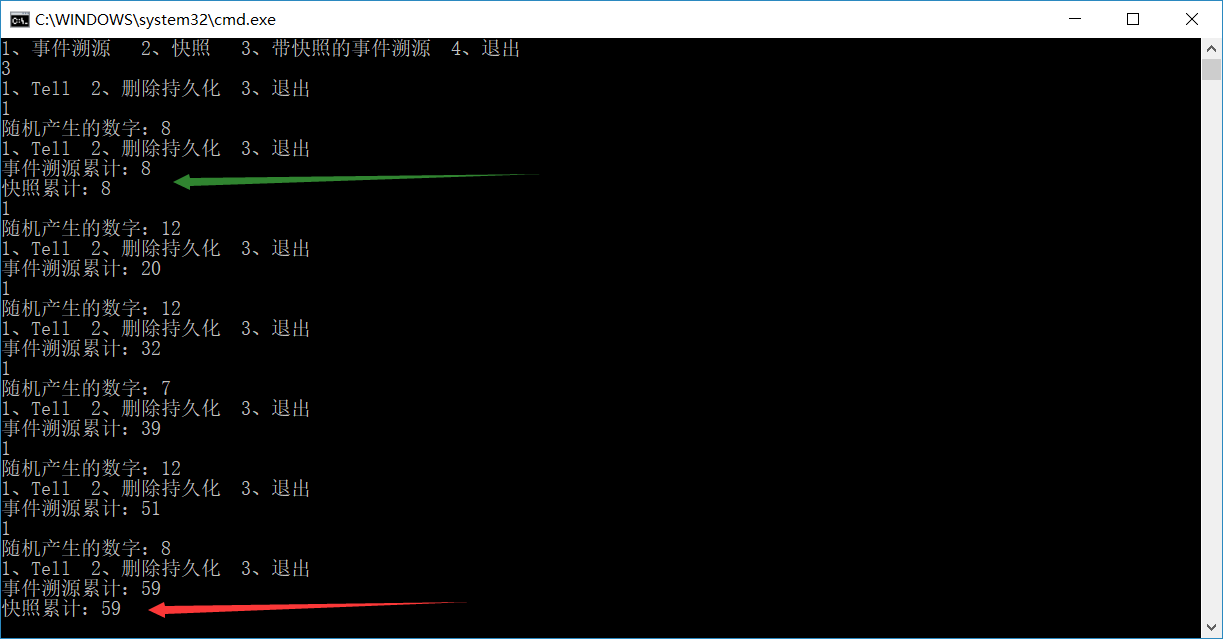
在创建持久化对象时，可以添加快照策略

\_persistence = Persistence.WithEventSourcingAndSnapshotting(eventStore, snapshotStore, actorId, ApplyEvent, ApplySnapshot, new IntervalStrategy(5), () => { return \_value; });

您可以选择ISnapshotStrategy在保存事件时指定自动保存快照。提供的策略是：

* EventTypeStrategy - 根据保存的事件类型保存快照
* IntervalStrategy - 根据保存的事件数量，即每100个事件，定期保存快照
* TimeStrategy - 根据时间以固定间隔保存快照，即在快照之间等待至少6小时

同时要在Actor的Receive把保存快照注释掉，Demo中我用的是5个事件后保存一次快照，如下图结果



绿色是第一次，要保存一下快照，然后之后第五个事件过来后保存第二次快照，如果在第四个事件后程序就退出，那快照保存的只有第一次的，不有担心，当再次调用时，因为记录下了所有事件，Actor会取出最后一次快照，再支执行快照后的事件，这是因为在保存快照和事件时，会把他们的索引保存起来，索引是一样的，就能用最后的快照+这个快照索引后的事件，恢复到退出的地方。

记得执行后查看Sqlite数据，有助于你更好的了解Proto.Actor的Persistence机制哦！

## 《通过C#学Proto.Actor模型》之Remote

Proto.Actor中提供了基于tcp/ip的通迅来实现Remote，可以通过其Remot实现对Actor的调用。

先来看一个极简单片的远程调用。

码友看码：

引用NuGet包

Proto.Actor

Proto.Remote

Proto.Serialization.Wire

共享库：

namespace P009\_Lib

{

    public class HelloRequest

    {

        public string Message

        {

            get; set;

        }

    }

    public class HelloResponse

    {

        public string Message

        { get; set; }

    }

}

服务端：

using P009\_Lib;

using Proto;

using Proto.Remote;

using Proto.Serialization.Wire;

using System;

using System.Threading;

using System.Threading.Tasks;

namespace P009\_Server

{

    class Program

    {

        static void Main(string[] args)

        {

            Console.Title = "服务端";

            Console.WriteLine("回车开始");

            Console.ReadLine();

            //设置序列化类型并注册

            var wire = new WireSerializer(new[] { typeof(HelloRequest), typeof(HelloResponse) });

            Serialization.RegisterSerializer(wire, true);

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

            //注册一个为hello类别的

            Remote.RegisterKnownKind("hello", props);

            //服务端监控端口5001

            Remote.Start("127.0.0.1", 5001);

            Console.WriteLine("服务端开始……");

            Console.ReadLine();

        }

    }

    class HelloQuestActor : IActor

    {

        public Task ReceiveAsync(IContext context)

        {

            switch (context.Message)

            {

                case HelloRequest msg:

                    Console.WriteLine(msg.Message);

                    context.Respond(new HelloResponse

                    {

                        Message = $"回应：我是服务端【{DateTime.Now}】",

                    });

                    break;

            }

            return Actor.Done;

        }

    }

}

客户端：

using P009\_Lib;

using Proto;

using Proto.Remote;

using Proto.Serialization.Wire;

using System;

using System.Threading.Tasks;

namespace P009\_Client

{

    class Program

    {

        static void Main(string[] args)

        {

            Console.Title = "客户端";

            Console.WriteLine("回车开始");

            Console.ReadLine();

            //设置序列化类型并注册

            var wire = new WireSerializer(new[] { typeof(HelloRequest), typeof(HelloResponse) });

            Serialization.RegisterSerializer(wire, true);

            //设置自己监控端口5002

            Remote.Start("127.0.0.1", 5002);

            //连接服务端5001

            var pid = Remote.SpawnNamedAsync("127.0.0.1:5001", "clientActor", "hello", TimeSpan.FromSeconds(50)).Result.Pid;

            while (true)

            {

                var res = pid.RequestAsync<HelloResponse>(new HelloRequest { Message = $"请求：我是客户端 【{DateTime.Now}】" }).Result;

                Console.WriteLine(res.Message);

                Console.ReadLine();

            }

        }

    }

}

代码很简单，看注释就够了。

……

## 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。