# Protoactor学习笔记

**什么是Actor？**

Actor是通过消息通讯的并发对象，就是在面向对象编程里面凡是需要通过消息通讯的对象都可以定义成actor，然后就可以采用actor机制操作了。

在GO语言中只要对象实现了Actor接口就定义为actor对象了，即实现了(\*object)Receive(c actor.Context)方法就是一个actor对象了

Actor就是一个对象，这些对象之间互相独立，互相隔离，异步通讯

**Actor的异步消息通讯**

消息是发送道Actor的邮箱，Actor消息发送之后不会阻塞，而是继续执行

Actor在处理消息时，可以执行三种操作：

创建新的Actor，或者说创建新的actor对象；发送消息到别的actor，修改自己对象的状态

最基本的用法，一个actor对象给另外一个发送hello消息，然后响应：

/\* 首先，定义actor通讯的消息格式，这里定义一个最简单的TLV格式的消息\*/  
**type** helloMessage **struct** {  
 messageType int  
 length int  
 value string  
}  
  
**const** (  
 *Hello* = *iota* //0  
 *HelloResponse* //1  
)  
  
/\* 然后，定义一个Actor对象，用于接收HellO消息\*/  
**type** helloActor **struct** {  
}  
  
/\* 然后了，实现Actor接口的方法\*/  
**func** (\*helloActor) Receive(c actor.Context) {  
 **switch** msg := c.Message().(**type**) {  
 **case** \*helloMessage:  
 **if** msg.messageType == *Hello* {  
 fmt.Println(c.Sender(), "say to ", c.Self().String(), ":", msg.value)  
 c.Respond(&helloMessage{*HelloResponse*, 10, "You are welcome!"})  
 } **else if** msg.messageType == *HelloResponse* {  
 fmt.Println(c.Sender(), "say to ", c.Self().String(), ":", msg.value)  
 }  
 }  
}  
  
/\* 演示给一个actor发送hello消息，然后收到一个响应\*/  
**func** DemoHelloActor() {  
 helloActorSrc := actor.Spawn(actor.FromInstance(&helloActor{}))  
 helloActorDst := actor.Spawn(actor.FromInstance(&helloActor{}))  
   
 helloActorDst.Request(&helloMessage{*Hello*, 10, "Hello, every one!"},helloActorSrc)  
   
 time.Sleep(2 \* time.*Second*)  
}

Props是Actor的配置参数，业务代码通过props来对Actor进行控制

Props中必须配置Actor，就是制定props属于哪个actor

actor.FromProducer(actorProducer Producer)

actor.FromFunc(f ActorFunc)

actor.FromInstance(template Actor)

框架对象的配置不是必须的，如果不配置采用缺省配置

Spawn，Actor的创建

PID是操纵Actor的标识，创建成果后就返回一个PID指针，如：

Func Spawn(props \*Props) \*PID

Func SpawnNamed(props \*Props, name string) (\*PID,err)

Actor Context

Context保存了Actor处理消息的上下文，如

待处理的消息，消息发送方，自己的PID，父PID，子PID切片

Context是Actor和框架交互的中间人，如：

监视其他Actor：func （c \*context）watch(pid \*PID)

修改消息处理函数 func c \*context）SetBehavior(behavior ActorFunc) //换一个消息处理函数，缺省是Receive(c context)

应答消息：func (c \*context)Respond(responseMessage interface{})

Context保证了Actor之间的隔离性

小贴士：Variable.(type)的用法，第一个返回值表示值，第二参数表示是否是type类型，通常先判断类型是否正确，再使用值。

that1, ok := that.(\*PoisonPill) //如果括弧里面事type，则ok表示。。。  
**if** !ok {  
 that2, ok := that.(PoisonPill)  
 **if** ok {  
 that1 = &that2  
 } **else** {  
 **return** *false* }  
}

PID是一个对象，用于Actor发送消息，对象定义如下所示：



Address：没有什么实际意义，通常就是nonhost

ID：全局唯一的一个标识，用于唯一标识一个PID对象，其实也就是一个字符串，在创建PID的时候指定，如果不明确指定就自动生成，也可以用SpawnName方法明确指定名称，但是如果名称不唯一会创建失败。同时创建PID时，PID会与一个Actor对象一一绑定，此后就可以通过PID给这个Actor对象发送消息了。

p：process，PID对象在发送消息的时候实际是通过process对象来执行的。在创建PID对象的时候，会通过processRegistry对象将PID和process映射起来。

发送消息有三种方式：

Tell：异步发送，发后就不管了，而且不知道发送着是谁

Request：也是异步发哦是那个，发后就不管了，但是可以通过actor的context.Sender知道发送着，而且通常是需要回响应消息的，通过context.respond方法回相应消息；

RequestFuture：同步发送消息，发送后需要等响应，有超时处理；

Marshal：就是把Address和ID转成ascii码

String：就是Address+”/”+ID