Actor模型是解决高并发的终极解决方案

架构师

写在开始

一般来说有两种策略用来在并发线程中进行通信：共享数据和消息传递。使用共享数据方式的并发编程面临的最大的一个问题就是数据条件竞争。处理各种锁的问题是让人十分头痛的一件事。

传统多数流行的语言并发是基于多线程之间的共享内存，使用同步方法防止写争夺，Actors使用消息模型，每个Actor在同一时间处理最多一个消息，可以发送消息给其他Actor，保证了单独写原则。从而巧妙避免了多线程写争夺。和共享数据方式相比，消息传递机制最大的优点就是不会产生数据竞争状态。实现消息传递有两种常见的类型：基于channel（golang为典型代表）的消息传递和基于Actor（erlang为代表）的消息传递。

Actor简介

> Actor模型(Actor model)首先是由Carl Hewitt在1973定义， 由Erlang OTP 推广，其 消息传递更加符合面向对象的原始意图。Actor属于并发组件模型，通过组件方式定义并发编程范式的高级阶段，避免使用者直接接触多线程并发或线程池等基础概念。Actor模型=数据+行为+消息。

Actor模型是一个通用的并发编程模型，而非某个语言或框架所有，几乎可以用在任何一门编程语言中，其中最典型的是erlang，在语言层面就提供了Actor模型的支持，杀手锏应用RabbitMQ 就是基于erlang开发的。

更加面向对象

Actor类似面向对象编程（OO）中的对象，每个Actor实例封装了自己相关的状态，并且和其他Actor处于物理隔离状态。举个游戏玩家的例子，每个玩家在Actor系统中是Player 这个Actor的一个实例，每个player都有自己的属性，比如Id，昵称，攻击力等，体现到代码级别其实和我们OO的代码并无多大区别，在系统内存级别也是出现了多个OO的实例

```
class PlayerActor
    {
        public int Id { get; set; }
```

```
        public string Name { get; set; }
    }
```

无锁

在使用Java/C# 等语言进行并发编程时需要特别的关注锁和内存原子性等一系列线程问题，而Actor模型内部的状态由它自己维护即它内部数据只能由它自己修改(通过消息传递来进行状态修改)，所以使用Actors模型进行并发编程可以很好地避免这些问题。Actor内部是以单线程的模式来执行的，类似于redis，所以Actor完全可以实现分布式锁类似的应用。

异步

每个Actor都有一个专用的MailBox来接收消息，这也是Actor实现异步的基础。当一个Actor实例向另外一个Actor发消息的时候，并非直接调用Actor的方法，而是把消息传递到对应的MailBox里，就好像邮递员，并不是把邮件直接送到收信人手里，而是放进每家的邮箱，这样邮递员就可以快速的进行下一项工作。所以在Actor系统里，Actor发送一条消息是非常快的。

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-vWwNZ5t0-1570964270753)(https://timgsa.baidu.com/timg...]

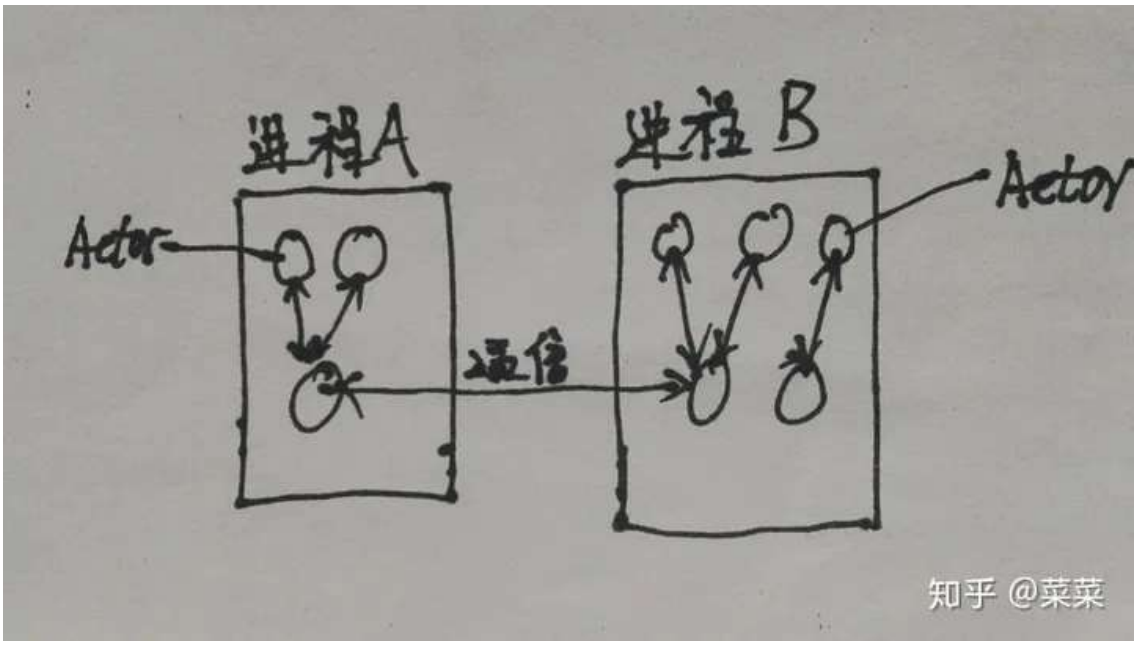
这样的设计主要优势就是解耦了Actor，数万个Actor并发的运行，每个actor都以自己的步调运行，且发送消息，接收消息都不会被阻塞。

隔离

每个Actor的实例都维护这自己的状态，与其他Actor实例处于物理隔离状态，并非像 多线程+锁模式那样基于共享数据。Actor通过消息的模式与其他Actor进行通信，与OO式的消息传递方式不同，Actor之间消息的传递是真正物理上的消息传递。

天生分布式

每个Actor实例的位置透明，无论Actor地址是在本地还是在远程机器上对于代码来说都是一样的。每个Actor的实例非常小，最多几百字节，所以单机几十万的Actor的实例很轻松。如果你写过golang代码，就会发现其实Actor在重量级上很像Goroutine。由于位置透明性，所以Actor系统可以随意的横向扩展来应对并发，对于调用者来说，调用的Actor的位置就在本地，当然这也得益于Actor系统强大的路由系统。



生命周期

每个Actor实例都有自己的生命周期，就像C# java 中的GC机制一样，对于需要淘汰的Actor，系统会销毁然后释放内存等资源来保证系统的持续性。其实在Actor系统中，Actor的销毁完全可以手动干预，或者做到系统自动化销毁。

容错

说到Actor的容错，不得不说还是挺令人意外的。传统的编程方式都是在将来可能出现异常的地方去捕获异常来保证系统的稳定性，这就是所谓的防御式编程。但是防御式编程也有自己的缺点，类似于现实，防御的一方永远不能100%的防御住所有将来可能出现代码缺陷的地方。比如在java代码中很多地方充斥着判断变量是否为nil，这些就属于防御式编码最典型的案例。但是Actor模型的程序并不进行防御式编程，而是遵循“任其崩溃”的哲学，让Actor的管理者们来处理这些崩溃问题。比如一个Actor崩溃之后，管理者可以选择创建新的实例或者记录日志。每个Actor的崩溃或者异常信息都可以反馈到管理者那里，这就保证了Actor系统在管理每个Actor实例的灵活性。

劣势

天下无完美的语言，框架/模型亦是如此。Actor作为分布式下并发模型的一种，也有其劣势。

1. 由于同一类型的Actor对象是分散在多个宿主之中，所以取多个Actor的集合是个软肋。比如在电商系统中，商品作为一类Actor，查询一个商品的列表在多数情况下经过以下过程：首先根据查询条件筛选出一系列商品id，根据商品id分别取商品Actor列表（很可能会产生一个商品搜索的服务，无论是用es或者其他搜索引擎）。如果量非常大的话，有产生网络风暴的危险（虽然几率非常小）。在实时性要求不是太高的情况下，其实也可以独立出来商品Actor的列表，利用MQ接收商品信息修改的信号来处理数据一致性的问题。

2. 在很多情况下基于Actor模型的分布式系统，缓存很有可能是进程内缓存，也就是说每个Actor其实都在进程内保存了自己的状态信息，业内通常把这种服务成为有状态服务。但是每个Actor又有自己的生命周期，会产生问题吗？呵呵，也许吧。想想一下，还是拿商品作为例子， 如果环境是非Actor并发模型，商品的缓存可以利用LRU策略来淘汰非活跃的商品缓存，来保证内存不会使用过量，如果是基于Actor模型的进程内缓存呢，每个actor其实就是缓存本身，就不那么容易利用LRU策略来保证内存使用量了，因为Actor的活跃状态对于你来说是未知的。
3. 分布式事物问题，其实这是所有分布式模型都面临的问题，非由于Actor而存在。还是以商品Actor为例，添加一个商品的时候，商品Actor和统计商品的Actor（很多情况下确实被设计为两类Actor服务）需要保证事物的完整性，数据的一致性。在很多的情况下可以牺牲实时一致性用最终一致性来保证。
4. 每个Actor的mailBox有可能会出现堆积或者满的情况，当这种情况发生，新消息的处理方式是被抛弃还是等待呢，所以当设计一个Actor系统的时候mailBox的设计需要注意。

升华一下

1. 通过以上介绍，既然Actor对于位置是透明的，任何Actor对于其他Actor就好像在本地一样。基于这个特性我们可以做很多事情了，以前传统的分布式系统，A服务器如果想和B服务器通信，要么RPC的调用（http调用不太常用），要么通过MQ系统。但是在Actor系统中，服务器之间的通信都变的很优雅了，虽然本质上也属于RPC调用，但是对于编码者来说就好像在调用本地函数一样。其实现在比较时兴的是Streaming方式。
2. 由于Actor系统的执行模型是单线程，并且异步，所以凡是有资源竞争的类似功能都非常适合Actor模型，比如秒杀活动。
3. 基于以上的介绍，Actor模型在设计层面天生就支持了负载均衡，而且对于水平扩容支持的非常好。当然Actor的分布式系统也是需要服务注册中心的。
4. 虽然Actor是单线程执行模型，并不意味着每个Actor都需要占用一个线程，其实Actor上执行的任务就像Golang的goroutine一样，完全可以是一个轻量级的东西，而且一个宿主上所有的Actor可以共享一个线程池，这就保证了在使用最少线程资源的情况下，最大量化业务代码。

> **搜索公众号：架构师修行之路，领取福利，获取更多精彩内容**

发布于 2019-10-13 19:15

并发

高并发

actor 模型