

42 | Actor模型：面向对象原生的并发模型

2019-06-04 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 08:12 大小 7.52M



上学的时候，有门计算机专业课叫做面向对象编程，学这门课的时候有个问题困扰了我很久，按照面向对象编程的理论，对象之间通信需要依靠**消息**，而实际上，像 C++、Java 这些面向对象的语言，对象之间通信，依靠的是**对象方法**。对象方法和过程语言里的函数本质上没有区别，有入参、有出参，思维方式很相似，使用起来都很简单。那面向对象理论里的消息是否就等价于面向对象语言里的对象方法呢？很长一段时间里，我都以为对象方法是面向对象理论中消息的一种实现，直到接触到 Actor 模型，才明白消息压根不是这个实现法。

Hello Actor 模型


Actor 模型本质上是一种计算模型，基本的计算单元称为 Actor，换言之，**在 Actor 模型中，所有的计算都是在 Actor 中执行的**。在面向对象编程里面，一切都是对象；在 Actor

模型里，一切都是 Actor，并且 Actor 之间是完全隔离的，不会共享任何变量。

当看到“不共享任何变量”的时候，相信你一定会眼前一亮，并发问题的根源就在于共享变量，而 Actor 模型中 Actor 之间不共享变量，那用 Actor 模型解决并发问题，一定是相当顺手。的确是这样，所以很多人就把 Actor 模型定义为一种**并发计算模型**。其实 Actor 模型早在 1973 年就被提出来了，只是直到最近几年才被广泛关注，一个主要原因就在于它是解决并发问题的利器，而最近几年随着多核处理器的发展，并发问题被推到了风口浪尖上。

但是 Java 语言本身并不支持 Actor 模型，所以如果你想在 Java 语言里使用 Actor 模型，就需要借助第三方类库，目前能完备地支持 Actor 模型而且比较成熟的类库就是**Akka**了。在详细介绍 Actor 模型之前，我们就先基于 Akka 写一个 Hello World 程序，让你对 Actor 模型先有个感官的印象。

在下面的示例代码中，我们首先创建了一个 ActorSystem（Actor 不能脱离 ActorSystem 存在）；之后创建了一个 HelloActor，Akka 中创建 Actor 并不是 new 一个对象出来，而是通过调用 system.actorOf() 方法创建的，该方法返回的是 ActorRef，而不是 HelloActor；最后通过调用 ActorRef 的 tell() 方法给 HelloActor 发送了一条消息“Actor”。

 复制代码

```
1 // 该 Actor 当收到消息 message 后，
2 // 会打印 Hello message
3 static class HelloActor
4     extends UntypedActor {
5     @Override
6     public void onReceive(Object message) {
7         System.out.println("Hello " + message);
8     }
9 }
10
11 public static void main(String[] args) {
12     // 创建 Actor 系统
13     ActorSystem system = ActorSystem.create("HelloSystem");
14     // 创建 HelloActor
15     ActorRef helloActor =
16         system.actorOf(Props.create(HelloActor.class));
17     // 发送消息给 HelloActor
18     helloActor.tell("Actor", ActorRef.noSender());
19 }
```

通过这个例子，你会发现 Actor 模型和面向对象编程契合度非常高，完全可以用 Actor 类比面向对象编程里面的对象，而且 Actor 之间的通信方式完美地遵守了消息机制，而不是通过对象方法来实现对象之间的通信。那 Actor 中的消息机制和面向对象语言里的对象方法有什么区别呢？

消息和对象方法的区别

在没有计算机的时代，异地的朋友往往是通过写信来交流感情的，但信件发出去之后，也许会在寄送过程中弄丢了，也有可能寄到后，对方一直没有时间写回信.....这个时候都可以让邮局“背个锅”，不过无论如何，也不过是重写一封，生活继续。

Actor 中的消息机制，就可以类比这现实世界里的写信。Actor 内部有一个邮箱 (Mailbox)，接收到的消息都是先放到邮箱里，如果邮箱里有积压的消息，那么新收到的消息就不会马上得到处理，也正是因为 Actor 使用单线程处理消息，所以不会出现并发问题。你可以把 Actor 内部的工作模式想象成只有一个消费者线程的生产者 - 消费者模式。

所以，在 Actor 模型里，发送消息仅仅是把消息发出去而已，接收消息的 Actor 在接收到消息后，也不一定会立即处理，也就是说**Actor 中的消息机制完全是异步的**。而**调用对象方法**，实际上是**同步的**，对象方法 return 之前，调用方会一直等待。

除此之外，**调用对象方法**，需要持有对象的引用，**所有的对象必须在同一个进程中**。而在 Actor 中发送消息，类似于现实中的写信，只需要知道对方的地址就可以，**发送消息和接收消息的 Actor 可以不在一个进程中，也可以不在同一台机器上**。因此，Actor 模型不但适用于并发计算，还适用于分布式计算。

Actor 的规范化定义

通过上面的介绍，相信你应该已经对 Actor 有一个感官印象了，下面我们再看看 Actor 规范化的定义是什么样的。Actor 是一种基础的计算单元，具体来讲包括三部分能力，分别是：

1. 处理能力，处理接收到的消息。
2. 存储能力，Actor 可以存储自己的内部状态，并且内部状态在不同 Actor 之间是绝对隔离的。
3. 通信能力，Actor 可以和其他 Actor 之间通信。

当一个 Actor 接收的一条消息之后，这个 Actor 可以做以下三件事：

1. 创建更多的 Actor；
2. 发消息给其他 Actor；
3. 确定如何处理下一条消息。


其中前两条还是很好理解的，就是最后一条，该如何去理解呢？前面我们说过 Actor 具备存储能力，它有自己的内部状态，所以你也可以把 Actor 看作一个状态机，把 Actor 处理消息看作是触发状态机的状态变化；而状态机的变化往往要基于上一个状态，触发状态机发生变化的时刻，上一个状态必须是确定的，所以确定如何处理下一条消息，本质上不过是改变内部状态。

在多线程里面，由于可能存在竞态条件，所以根据当前状态确定如何处理下一条消息还是有难度的，需要使用各种同步工具，但在 Actor 模型里，由于是单线程处理，所以就不存在竞态条件问题了。

用 Actor 实现累加器

支持并发的累加器可能是最简单并且有代表性的并发问题了，可以基于互斥锁方案实现，也可以基于原子类实现，但今天我们要尝试用 Actor 来实现。

在下面的示例代码中，CounterActor 内部持有累计值 counter，当 CounterActor 接收到一个数值型的消息 message 时，就将累计值 counter += message；但如果是其他类型的消息，则打印当前累计值 counter。在 main() 方法中，我们启动了 4 个线程来执行累加操作。整个程序没有锁，也没有 CAS，但是程序是线程安全的。

 复制代码

```
1 // 累加器
2 static class CounterActor extends UntypedActor {
3     private int counter = 0;
4     @Override
5     public void onReceive(Object message){
6         // 如果接收到的消息是数字类型，执行累加操作，
7         // 否则打印 counter 的值
8         if (message instanceof Number) {
9             counter += ((Number) message).intValue();
10        } else {
11            System.out.println(counter);
12        }
13    }
```

```

14 }
15 public static void main(String[] args) throws InterruptedException {
16     // 创建 Actor 系统
17     ActorSystem system = ActorSystem.create("HelloSystem");
18     //4 个线程生产消息
19     ExecutorService es = Executors.newFixedThreadPool(4);
20     // 创建 CounterActor
21     ActorRef counterActor =
22         system.actorOf(Props.create(CounterActor.class));
23     // 生产 4*100000 个消息
24     for (int i=0; i<4; i++) {
25         es.execute()->{
26             for (int j=0; j<100000; j++) {
27                 counterActor.tell(1, ActorRef.noSender());
28             }
29         });
30     }
31     // 关闭线程池
32     es.shutdown();
33     // 等待 CounterActor 处理完所有消息
34     Thread.sleep(1000);
35     // 打印结果
36     counterActor.tell("", ActorRef.noSender());
37     // 关闭 Actor 系统
38     system.shutdown();
39 }

```

总结

Actor 模型是一种非常简单的计算模型，其中 Actor 是最基本的计算单元，Actor 之间是通过消息进行通信。Actor 与面向对象编程（OOP）中的对象匹配度非常高，在面向对象编程里，系统由类似于生物细胞那样的对象构成，对象之间也是通过消息进行通信，所以在面向对象语言里使用 Actor 模型基本上不会有违和感。

在 Java 领域，除了可以使用 Akka 来支持 Actor 模型外，还可以使用 Vert.x，不过相对来说 Vert.x 更像是 Actor 模型的隐式实现，对应关系不像 Akka 那样明显，不过本质上也是一种 Actor 模型。

Actor 可以创建新的 Actor，这些 Actor 最终会呈现出一个树状结构，非常像现实世界里的组织结构，所以利用 Actor 模型来对程序进行建模，和现实世界的匹配度非常高。Actor 模型和现实世界一样都是异步模型，理论上不保证消息百分百送达，也不保证消息送达的顺序和发送的顺序是一致的，甚至无法保证消息会被百分百处理。虽然实现 Actor 模型的厂

商都在试图解决这些问题，但遗憾的是解决得并不完美，所以使用 Actor 模型也是有成本的。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 41 | 案例分析（四）：高性能数据库连接池HiKariCP

精选留言 (9)

写留言



兔斯基

2019-06-04

2

这种并发模型现实应用多么？系统往往很少有可以接受丢失消息的吧？

展开 ▾

作者回复: 线程池也一样会丢数据，现在基本上都是靠数据库，mq这些支持事务的存储来搞定安全的异步处理



潭州太守

2019-06-04



请问老师，Actor是不是不适合低延迟场景，或者有没有策略保证低延迟。

展开 ∨



有铭

2019-06-04



Actor模型的最佳实践目前还是erlang，Java的akka有些不伦不类

展开 ∨



峰

2019-06-04



感觉核心就是通过消息对列实现消息的暂存，然后actor就可以一个接一个单线程处理消息。有点像redis，但不同的是不同actor的调用线程可能不一样，只要保证同一时刻最多只有一个线程处理某个actor就行，并且actor直接可以消息通信，意味着可以用多个actor去组织起来完成一次请求。



明天更美好

2019-06-04



遇到一个线程问题，我们有个业务要通过mq去通知第三方，但是第三方能力比较差，我们同步的时候mq堆积很多。后来改成用woker-thread模式，队列设置了2000线程用了64个机器是64核的，拒绝策论是当前线程执行该任务。结果发现队列很快就被放满了，一段时间后mq又堆积了。因为客户端没有及时签收消息，导致broker限流了直接不消费了，这种问题老师您有什么好的建议吗？

展开 ∨



往事随风, ...

2019-06-04



Scala 编写spark 内部实现也是用这个通信机制

展开 ∨





邱
2019-06-04



要是数据库io和纯cpu结合的大数据量高并发的实际就更好 😊

展开 ▾



周治慧
2019-06-04



万物皆是面向对象 早上好

展开 ▾



张三
2019-06-04



打卡!

展开 ▾