SELECTOR 模型及其分布式系统

王宇晗
竺可桢学院
wang_yuhan@zju.edu.cn

蒋仕彪
竺可桢学院
3170102587@zju.edu.cn

章启航
竺可桢学院
qh_zhang@zju.edu.cn

李广林
竺可桢学院
3170104648@zju.edu.cn

2020 年 1 月 3 日

摘要

Selector 模型是一种分布式计算模型。他脱胎于上世纪七十年代提出的 Actor 模型 [1]。经典的 Actor 模型在处理很多分布式计算问题时会面临难以解决的一致性问题，因此多年来诞生了许多 Actor 模型的变种（如 Akka [2]），也有许多与 Actor 结构不同的模型出现（如 CSP [3]）。本文介绍的 Selector 模型是基于 Actor 模型的变种 [4]，它还实现了大规模的分布式系统集群的搭建 [5]。它解决了多个过往模型难以处理的分布式计算问题，并在多个 benchmark 问题上取得了更好的表现。

关键词 Actor 模型 · Selector 模型 · Selector 分布式系统 · Akka 框架 · CSP

1 Actor 模型

在介绍 **Selector 模型**前，我们首先以 **Actor 模型**作为引入。Actor 模型是最早也是最经典的分布式计算模型，它首次提出于 1973 年 [1]，它既是一个计算理论框架，也是很多分布式系统实现的理论基础。

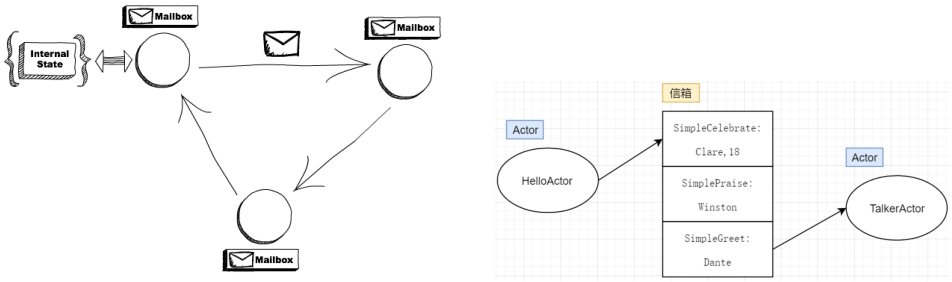
1.1 基本概念

Actor 模型将 Actor 作为分布式计算的单元，每一个单独的 Actor 可以修改自己的私有状态，但只能通过发送消息 (messaging) 来改变其他 Actor 的状态。因此 Actor 模型的基础就是消息传递。

每一个 Actor 由状态 (state)、行为 (behavior) 和邮箱 (mailbox) 三部分组成：

- 状态 (state)：如果用面向对象的视角来看待 Actor，状态可以更直观地理解为 Actor 对象的变量信息，状态由 Actor 自己管理，避免了并发环境下的原子性问题。
- 行为 (behavior)：行为指定 Actor 中的计算逻辑，通过 Actor 接收到的消息来改变 Actor 的状态。
- 邮箱 (mailbox)：邮箱是 Actor 和 Actor 之间的通信桥梁，邮箱内部通过 First-in-First-out 消息队列存储发送方 Actor 的消息，并从中获取消息用于指导 Actor 的下一步动作。

Actor 最大的特点在于消息的发送与处理是**异步**的。所以当 Actor 正在处理消息时，新来的消息应该存储到别的地方，也就是 mailbox 消息存储的地方。需要指明的一点是，尽管多个 Actor 同时运行，但是一个 Actor 只能顺序地处理消息。也就是说其它 Actor 发送多条消息给一个 Actor 时，这个 Actor 只能一次处理一条。如果需要并行的处理多条消息时，就需要将消息发送给多个 actor。



(a) (b)

图 1: 消息被处理的顺序是按照他们到达 mailbox 的顺序，而不是被发出的顺序。

一个 Actor 从邮箱中获取到一条消息后，他进行的反应可以有：

- 修改内部状态
- 创建新的 Actor
- 向其他 Actor 发送消息
- 决定如何反应下一个到来的消息

值得注意的是，在传统的 Actor 模型中，消息的到达顺序是没有办法被确定的。也就是说，**消息在 mailbox 中的顺序只取决于他们到达 mailbox 的时间，与他们什么时候被发出毫无关系**。在 Actor 模型的发展与完善过程中，这个原则始终是一个充满争议的点。有的研究者建议，应该增加一条限制，要求消息应该严格按照被发出的顺序进入 mailbox。但 Actor 模型开山之作的作者 Hewitt 明确反对添加这样的限制，因为这违背了设计 Actor 模型的初衷。下一节将对此详细介绍。

1.2 特点与利弊

在谈 Actor 模型的特点前，我们首先要回顾传统并发模式编程的实现方式。传统并发模式中，**共享内存**是倾向于**强一致性弱隔离性** [1] 的，例如悲观锁同步的方式就是使用强一致性的方式控制并发。但控制并发的代价是，程序逻辑内在的复杂性有可能将多核多线程的应用变成单线程的应用，或者导致工作线程之间存在高度竞争。

而 Actor 模型作为**消息传递**模式的一种形式，天然是**强隔离性且弱一致性**的，所以 Actor 模型在并发中有良好的性能，而且易于控制和管理 [6]。因此，Actor 模型适用于对一致性需求不是很高且对性能需求较高的场景，同时数据的一致性是使用 Actor 模型编程时的主要难点。

此刻我们再回看上一节中提到的关于原则的争议。部分学者建议增加限制，规定消息在 mailbox 中按照发出的顺序进行处理。显然这会给一致性的保障提供很大的便利。例如在转账模型中，假定 WK 有三个账户 A、B、C。考虑以下场景，B 原有 50 元，WK 先从 A 向 B 转账 100 元，之后再从 B 向 C 转账 100 元。考虑 B 账户的 Actor，如果没有添加这条限制，那么 Actor 可能会先收到向 C 转账 100 元的消息，再收到进账 100

[1] 一致性是让数据保持一致，例如银行转账的场景中，转账完成时双方账户必须是一方减少一方增加。而隔离性而可以理解为牺牲一部分一致性需求，从而获得性能的提升。

元的消息。此时只有后一条会被成功处理。若添加了这条限制，那么账户 B 的结果则是唯一的，余额只能为 50 元。

然而从另一个角度来看，这条限制牺牲的是 Actor 模型在并发编程场景下的巨大潜力。假设消息在 Actor 之间传递的过程中可能经过不同的路径。同时，Actor 之间传递消息的通道是**不可靠**的，即消息在传递时有概率丢失或被修改。那么这条额外的限制将很难得到满足：前者导致消息的到达时间与发出时间并没有必然联系，而后者导致消息的时序性仍然不能保证上下文的确定。但众所周知，这两个特点正是**网络**的固有特点：数据包根据路由算法的选择进行传递，没有一种信道是完全可靠的。而只有让通信由网络完成，才能让超大规模的分布式计算成为可能。

因此，在 Actor 的各种衍生模型中（例如本文的主角 Selector），**消息顺序无关**这条原则基本得到了保留，这也让 Actor 模型天然地支持了大规模的并发和更高的并发性。

2 Akka 模型

基于 Actor 模型，有很多应用于商业场景的框架应运而生，它们的出现为 Selector 模型提供了灵感和借鉴，Akka 就是其中的代表。Akka 是基于 Acotr 模型的用于在 JVM 上管理并发性和弹性的一个框架，它同时支持 Java 和 Scala[2]。

2.1 Akka 的优点

Akka 有如下优点 [7]：

- 对并发模型进行了更高的抽象
- 是异步、非阻塞、高性能的事件驱动编程模型
- 轻量级事件处理（1GB 内存可容纳百万级别个 Actor）
- 提供了一套容错机制，允许在 Actor 出现异常时进行一些恢复或重置操作
- 既可以在单机上构建高并发程序，也可以在网络中构建分布式程序，并提供位置透明的 Actor 定位服务

2.2 Akka 的具体应用

下面我们将以一个电商平台处理“秒杀”活动的实例来分析 Akka 框架的优势。

所谓的“秒杀”活动，简单点来说，就是把某个稀缺商品或促销商品，挂到页面，供大量客户抢购。这里有两个关键点，**商品数量不多，客户量非常大或抢购流量非常大**。客户量或抢购流量往往意味着并发量非常大，容易给服务器造成很大的瞬时压力。

为了简化问题，我们把秒杀活动中的概念也进行简化，分为库存和抢购请求。库存：待抢购商品的数量。抢购请求：客户为了抢购商品的点击动作，也就是一次请求。抢购请求分为成功和不成功两种，抢购成功会减少库存，否则不会。

在 akka 中，我们可以将库存抽象一个为一个 actor 集，每一个库存代表一个 actor，抢购请求就是 actor 收到的消息。由于 akka 中 actor 的轻量级特性，每个 actor 的内存占用大概在 4k，1G 的内存可以生成 26 万的 actor。26 万的商品对于大多数秒杀场景绰绰有余。

我们将抢购请求视做一个 actor 的消息。这里的需要实现的难点是消息的路由，我们需要让特定的用户的请求路由到特定的 actor 模型上，同时我们还需要实现均匀的路由，避免局部的热点的出现。akka 中解决这个问题的方法是又创建了一个 Router actor, 在每个抢购消息发送之前都需要经过该 actor，由该 actor 对消

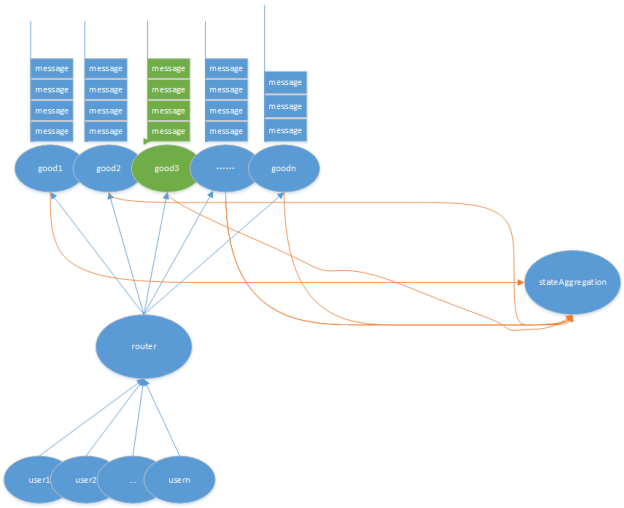


图 2: Akka 应用于“秒杀”场景中的架构

息进行分发。该 router 的作用，就是把同一个客户的抢购请求路由到固定的商品 actor。每个商品和用户都会有一个 HASH 值，Routre 总是把用户 ID 的 HASH 值跟商品 HASH 进行比较，把抢购消息路由给与用户 ID 的 HASH 值最接近的商品 actor 上就可以了。

另外商品 actor 的邮箱类型需要修改成有界队列。因为每个商品就是一个 actor，而每个商品只能买个一个客户，所以严格上来说，这个 actor 只能接收一个抢购请求。所以商品 actor 的邮箱类型必须是有界队列，避免消息过多撑爆内存，当然这个队列的长度必须是大于 1 的。超过这个队列长度的消息怎么办呢？在这个商品被抢购成功之前，其实可以直接丢弃，因为已经有其他客户占用这个商品了。当然，占用不意味着一定能够抢成功，也可能失败，比如触发了反薅羊毛策略，或数据库更新失败。因为队列中还有其他的抢购请求，前面的客户抢购失败，还是可以把该商品分配给其他客户的。

最后还需要计算当前商品的库存，这也需要一个 actor。每个商品 actor 启动时，给 stateActor 发消息；商品被抢购成功后，给 stateActor 发消息，同时 stop 掉自身。这样 stateActor 就可以异步的获取当前的库存了。

2.3 Akka 模型在一些场景下的劣势

同步请求-应答机制 (Synchronous Request-Reply Pattern) Actor 需要使用 stash 去阻塞在 reply 来之前的 request，而 Akka 系统实现了 become & unbecome 机制，抽象封装了 stash 阻塞的操作，然而这种机制仍然会因为阻塞、需要管理 stash 带来很大的开销。相比之下，selector 可以通过简单地多邮箱性质实现请求和相应消息的隔离。

流服务中的合并操作 (Join Patterns in Streaming Applications) Akka 提供了对聚合器模式的支持，其允许动态地添加或删除消息处理逻辑内部的参与者。但是，该实现不允许在聚合过程中匹配消息的发送者（源），然而这却是连接模式的关键部分。

3 CSP 模型

communicating sequential processes(CSP) 是用于描述并发系统中通信模式的编程语言。CSP 中，channel 是主体，process 通过 channel 来传递信息 [3]。

3.1 CSP 与 Actor 模型的区别

- 在 CSP 模型中，channel 是主体，process 是匿名的，而在 Actor 中，actor 是主体，而 mailbox 是匿名的。Actor 中，每个 Actor 只有一个对应的 Mailbox，Actor 之间的信息传递是通过发送到 mailbox 中实现的。在 CSP 中 Channel 和 Process 之间没有从属关系，Process 可以接受任意个 Channel 发送的信息。
- 在 CSP 模型中，channel 之间进行消息传递，而 Actor 中消息传递会将消息发送给指定的目标 Actor。CSP 模型可以让每个 process 对应一个 channel 来实现 Actor 模型，在 Actor 模型中，可以将 channel 对应转化成 Actor 来实现 CSP 模型。
- CSP 模型中，消息的传递基本上需要接收者与发送者之间的会合，就是说只有当接收者准备接收消息时，发送者才能将消息发送给接收者。而在 Actor 模型中，消息的传递时不同步的，消息的发送和接受可以不在同一时刻发生，发送者可以在接收者准备接收之前就把消息发送出去。同步的交流可以用缓存交流的机制实现非同步的信息系统，非同步的交流可以通过消息协议来实现接收者和发送者之间的同步。

3.2 CSP 中的基本数据类型

在 CSP 中有两种基本数据类型：

- Events 代表着交流或者互动。Events 被认为是不可分割且瞬时的。可以是原子操作 (on，off) 也可以是输入输出事件。
- Primitive processes 代表基本的行为，比如 STOP（不进行任何通信的进程，即死锁），和 SKIP（成功终止）。

3.3 CSP 中的基本运算

a 是 event，P 是 process

- Prefix: $a \to P$ 交流完 event a 之后做 process P。
- Deterministic Choice: $(a \to P) \Box (b \to Q)$ 交流完 event a 后转到 process P 或者交流完 event b 后转到 process Q。
- Nondeterministic Choice: $(a \to P) \sqcap (b \to Q)$ 交流完 event a 与 event b 后随机转到 process P 或 process Q。
- Interleaving Choice: $P|||Q$ process P 与 process Q 并行。
- Interface Parallel: $P|a|Q$ 只有 process P 与 process Q 都能交流 event a 才能处理下一个事件，否则死锁。
- Hiding: $(a \to P) \backslash \{a\}$ 将 event a 隐藏，相当于 P。

4 Selector 模型

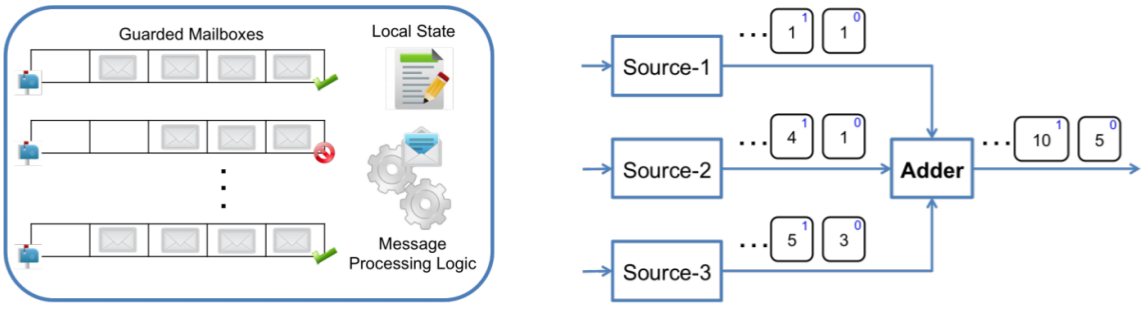
如上所述，Actor 模型是原生的异步模型，很难处理一些复杂的同步通信问题。Shams M Imam an 于 2014 年提出了 Selector 模型 [4]，用来针对性解决 Actor 模型的不足。

4.1 Selector 的基本概念

作为 Actor 模型的拓展，Selector 模型也是基本的消息处理单元。它最大的特点是：**每个 Selector 可以拥有多个不同优先级的邮箱**。每个邮件发送者可以选择投递到哪个邮箱；如果没有指定，接受者也可以自由地选择由哪个邮箱接收。

此外，每个邮箱还有一个运行状态，我们称之为 **guard**。guard 分为“开”和“闭”两种，一个 Selector 单位每次只会从“开”状态的邮箱里（以一定规则）取邮件。值得注意的是，邮箱的开关状态不影响它接受邮件的能力，即“关”状态的邮箱依然能够接受信息。

除了在处理消息时手动转换某个邮箱的 guard 值之外，Selector 还可以在一开始为其指出一个确定的布尔表达式。在这种情况下，邮箱的开闭是“自动”的，完全受这个表达式的值控制。



(a) (b)

图 3: 左图：Select 模型。右图：合作流模式示例。

4.2 应用举例

4.2.1 合作流模式

任务 两个或者多个管道都会往当前 Selector 里发信息，要正确地接收信息并按组批量处理。右上的图展示了一个加法器的模型，它每次要从三个源里接收一个数字，将它们加起来并输出。

分析 现实中经常会遇到这种多源的问题。

- **actors**: 由于多个源的信息发送不是有序的，接收者不能直接挑出自己感兴趣的信息。actor 必须专门维护一个数据结构，用来保存来自不同源的数据。这样的实现十分复杂，增加了很多不必要的麻烦。
- **selectors 解决方案一**: 创建 n 个邮箱（假设源的数量是 n，并编号为 $0 \sim n-1$）一开始所有邮箱都是打开的，每当 selector 接收到源 i 发来的邮件就关闭邮箱 i 。接收完一组信息后（所有邮箱都关闭了），执行加法任务并重新打开所有邮箱。
- **selectors 解决方案二**: 依然是创建 n 个邮箱，一开始只有邮箱 0 is 是打开的。每当 Selector 收到源 i 的信息后，关掉邮箱 i 并打开邮箱 $i+1$。接收完一组信息后执行加法任务，并重新打开邮箱 0。

4.2.2 同步的发送-接收模式 [8]

任务 接受者从发送者那里收到消息，处理完之后再发送回去。在这里，发送者会有一个**阻塞**的过程，即只有当它收到该接受者处理完的结果后才能处理别的消息。

分析 这是一个最基本的同步任务。

- **actors**: 发送者在收到回复前需要阻塞所有消息，所以它必须暂时“冻结”这段时间收到的消息。我们可能需要额外的队列或者一些特殊的技巧来做到“阻塞”它，因为 actor 模型里不自带这种功能。
- **selectors**: 我们只需定义两个邮箱：REGULAR 和 REPLY。前者用来接收**普通的信息**，后者只接收**对应的回复**。一开始的时候只有 REGULAR 被打开，每当它想执行这个同步操作时，关闭 REGULAR 信箱并打开 REPLY 信箱。这样，尽管普通邮件会堆积在 REGULAR 信箱里，selector 会暂时屏蔽它们直到 REPLY 信箱的信件被处理。

4.2.3 生产者-消费者模式 [9]

任务 生产者会不断地传输生产出来的产品，消费者会不断地发送产品需求的信息，你要实现一个带一定容量仓库的中介，做到资源的合理配置。

分析 这是操作系统里很经典的模型，同时包含同步和互斥的要求。

- **actors**: 有很多细节需要考虑。1) 当仓库空了消费者无法获得产品，必须开一个额外的队列来存储这些请求；2) 当生产者想发送产品但是仓库已经满了，必须回绝这个请求，并额外记录一些信息。这些细节会加大设计难度。
- **selectors**: 当前 Selector 中介只需维护两个信箱，一个针对生产者，一个针对消费者。我们用以下**明确的布尔表达式**声明它们的开关状态：
  guard(PRODUCER , dataBuffer.size() < thresholdSize)
  guard(CONSUMER, !dataBuffer.isEmpty())
  由此可见，每当仓库满了生产者信箱会自动关闭，每当仓库空了消费者信箱会自动关闭。这些信箱会持续接受信息“缓存”起来但是不会决策。

5 Selector 分布式系统

上一章主要介绍了单个 Selector 模型的特点和应用，本章将介绍如何用很多 Selector 搭建大规模的分布式系统集群 [5]。该系统主要依靠 Java 来搭建。

5.1 基本概念

分布式系统的一个基本计算单元通常被称为 *place* 或 *node*。分布式系统由无数个计算单元组成，且用户可以指定一个 **Master Node**，用来执行整个系统的（全局）启动和注销操作。

一个计算单元是这么构成的：

- 一个 *System Actor*: 用来维护这个计算单元内部的状态，同时与外界产生交流。
- 一个 *Proxy Actor*: 用来协调与本地和远方的 selector 进行通信。
- 一个 *local registy*: 由 Proxy Actor 维护，用来存储在本地的 selector 的地址。
- 若干个 *Selectors*: 该计算单元真正的消息处理结构。

5.2 协作机制

一些计算单元结合在一起可以构成庞大的系统，让我们分析一下它们是怎么进行协作的。

初始化 Master Node 的 System Actor 会用 SSH 连接所有远程的计算单元。每个计算单元会在初始化时建立与 Master Node 之间的 SSH 连接。然后它创建一个配置文件，用来记录本地的配置信息和 Master Node 的信

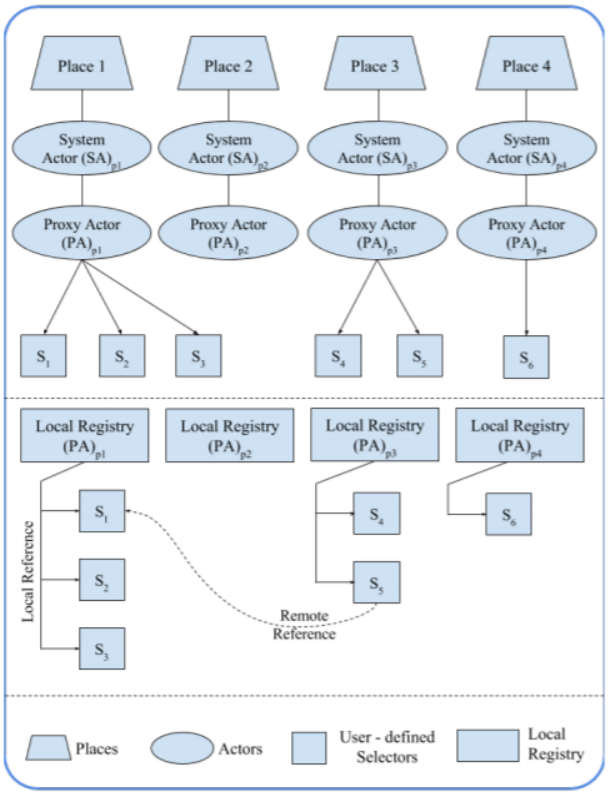


图 4: 每个计算单元的结构

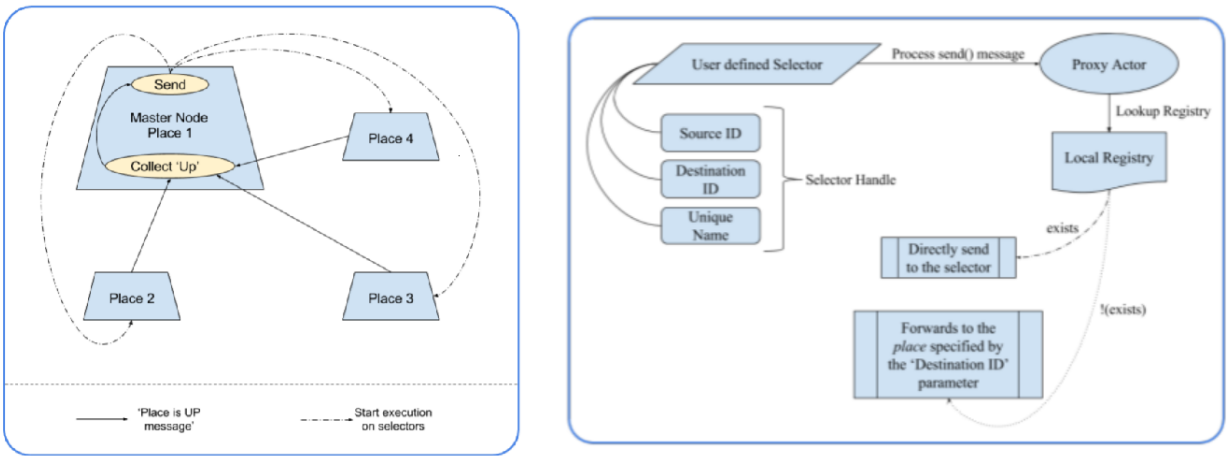


图 5: 分布式系统的初始化和交流

息。初始化成功后，这些 system actor 会把它们的成功状态发送给 Master Node。Master Node 收集到所有的创建信息后，再发送“可以正式启动”的命令给每个计算单元。自此，每个计算单元的 proxy actor 会唤醒底下的 Selector 进行消息处理。

信息交流 所有 Selector 会对应一个唯一的 32 位编码[2]。如果使用 Kryo serialization framework, 一个 selector 可以发送和接受任何可以序列化的信息（即可以 *java.serializable* 的信息）. 当信息在传递时，如果目标计算单元就在本地, Proxy Actor 会检查 local registry 来找到对应的 selector 然后直接传递消息；否则，说明目标计算单元在远处，那么 Proxy Actor 必须先连接远方的 Proxy Actor，再进行进一步的消息传递。

5.3 一个分布式系统的例子: N 皇后的前 K 小解

问题描述 要在 $N \times N$ 的棋盘里放置正好 N 个皇后，使得每个皇后不能互相攻击。要求用搜索找到字典序前 K 小的解。

采用分布式 Actor 如果直接用基于 Actor 协作的分布式系统，解决该问题的效率极其低下。主要是因为，Actor 通信时没有良好的同步机制，各计算单元在搜索时很难有效地把消息汇总起来。而且由于 Actor 模型没有消息优先级的机制，父 Actor 只能在一开始把状态平分再传递下去，很难及时知道是否可以停止搜索（即使目前的部分解字典序已经超过 K 了，它无法获知，必须基于搜索下去）。所以如果我们使用分布式 Actor 系统来搜索前 K 小的解，要不不能起到很好的并行效果，要不会几乎搜出所有的解、不能按字典序剪枝。

采用分布式 Selector Selectors 能很完美地并行解决这个问题。每当一个计算单元在一个未完成的解里成功放置了一个新的皇后，它就会立即汇报给 Master。如果该解还还没有放满，Master 会以轮转的方式把这些解重新分配给底下的 Selector，。如何保证是按字典序从小到大搜的呢？Selector 的优先级机制可以很好地保证这一点。因为信箱支持优先级，我们只需给字典序更小的解赋予更大的优先级，总体上就会按照字典序的大小进行搜索了。

6 总结

本综述报告主要基于这篇论文 [5]，系统介绍了 **Selector 模型及其在分布式系统上的应用**。为了详细阐述该模型诞生的前因后果，我们参考了很多引文并对相关的内容做了介绍，让读者有一个清晰自然的知识脉络。下面对本文的叙述做一个总结。

1. 本文首先从 Actor 模型出发，介绍了它的基本概念和特点，由此探讨了 Actor 模型的一些缺点，为之后 Selector 模型的改进方法做了铺垫。
2. 紧接着，本文深入分析了基于 Actor 模型的并发框架——Akka 框架。Akka 对并发模型进行了更高的抽象，对 Actor 模型做了一些有意义的改进和补充（如轻量级实现，提供容错机制等），能很好地应用在某些商业场景下。然而 Akka 在部分场景下存在劣势，这启发我们设计更有效的并发处理框架。
3. 再后来，本文介绍了并发系统中用于描述通信模式的 CSP 语言。它和 Actor 模型有一些本质的区别，为之后的 Selector 模型提供了创意来源。
4. 最后，受益于上述材料的启发，本文从“单模型”和“分布式系统”两个角度深入剖析了 Selector 模型及其应用。Selector 分布式系统很好地解决了以上场景遇到的问题，是一种新的优秀的并发框架。

[2]前 8 位记录了创建它的计算单元编号，中间 8 位记录了存储它的计算单元编号，最后 16 位是这个 Selector 的编号

参考文献

[1] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[2] Typesafe Inc. Actors - akka documentation, 2014.

[3] C.A.R.Hoare. Communicating sequential processes. commun. acm, 21(8):666–677, aug. 1978.

[4] Shams M Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 1–14. ACM, 2014.

[5] Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam, and Vivek Sarkar. A distributed selectors runtime system for java applications. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, page 3. ACM, 2016.

[6] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[7] Typesafe Inc. Akka cluster documentation, 2014.

[8] G Hohpe and B Woolf. Enterprise integration patterns-request-reply, 2003. *URL http://www. eaipatterns. com/RequestReply. html.[Online*.

[9] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. In *The origin of concurrent programming*, pages 272–294. Springer, 1974.