实时稳定高并发

——基于 *actor* 模型的另一种思考

王礼鹤
Wanglihe.programmer@gmail.com

我们正在经历的挑战

- 电信业向互联网靠拢的同时，互联网服务也在向电信级的服务靠拢
- 像能源，电力，交通一样，成为其他社会服务的基础平台
- 稳定，可靠， 7x24 小时持续服务
- 速度需要越来越快，近乎实时
- 同时向大量用户，大量任务提供服务
- 与此同时，用户在线时间越来越长，操作越来越复杂

我们正在经历的挑战

- 在线游戏
- 云存储
- 与在线教育类似的互动平台
- 协作平台

我们正在经历的挑战

- 所以：
- 在高并发的前提下，稳定性和实时性将变得越来越重要

并发的思考

- 单纯的并发，其实很简单：
- 只需要一个处理核心
- 足够的接受速度
- 配上足够容纳数据的内存

并发的思考

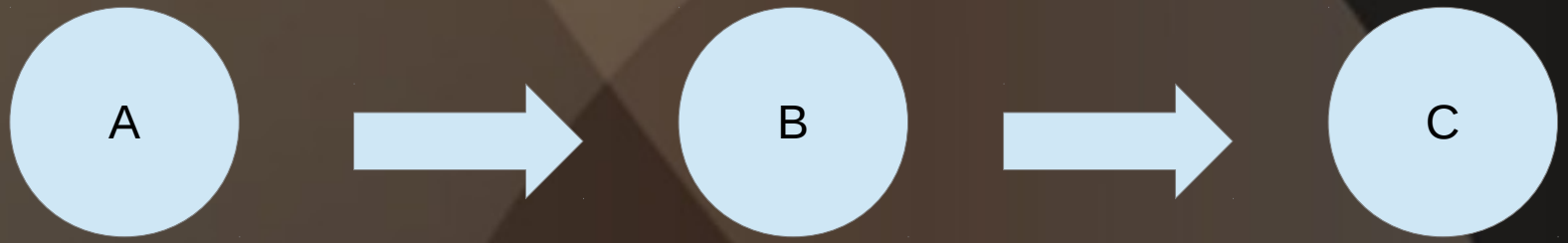
- 有效的并发：
- 在请求和新请求之间
- 有足够的计算能力和内存
- 处理请求而不堆积

并发的思考

- 并发实际需要解决的问题：
- 并发的基石，任务调度
- 任务调度与操作系统类似，需要可以打断和阻塞
- 任务类型更为单一，没有硬件驱动和总线，只有控制和内存

解决方案——线程（进程）

- 高并发架构模型 I ：管道模式（流水线模式）
- 由于数据共享的性能甚至诞生了很多技术：管道，命名管道，消息队列， socket ，共享内存
- 由于操作太复杂，性能太差，所以线程打败了进程

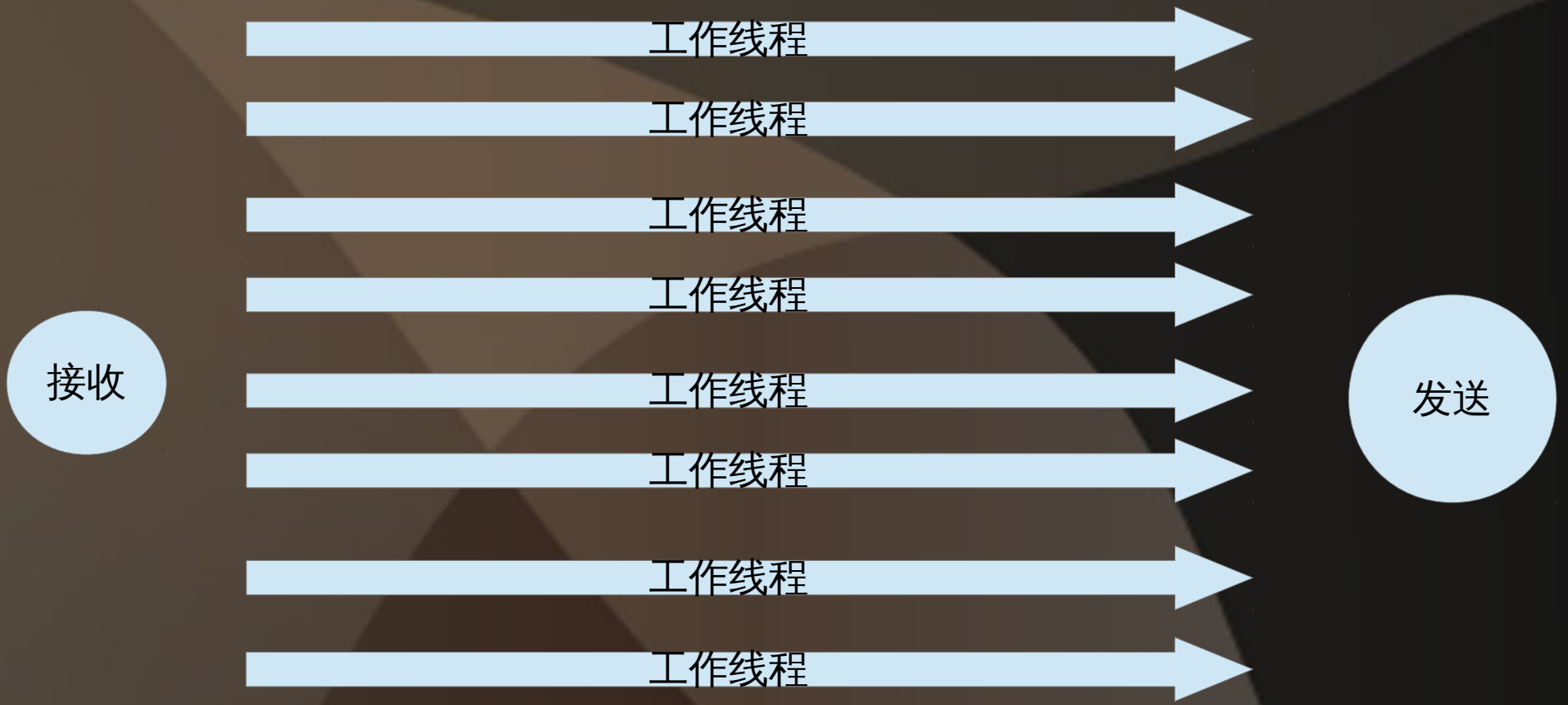

A
B
C

解决方案——线程（进程）

- 管道模式的弊端：
- 随着可用的计算资源增多，操作系统的调度不够特化
- 为利用多核，更多级的流水线，弹性丧失
- 为利用多核，更多的高计算需求的线程，出现了内部的任务分配和调度，难度大
- 整体上来说，需要开发人员特化的理解计算内容并合理化分，超过了正常人类的智力极限

解决方案——线程（进程）

- 高并发架构模型 II ：平行模式
- 每请求一线程
- 生活从未如此美好：
- 直接使用操作系统调度
- 代码简单清晰
- 数据大部分独立


工作线程
工作线程
工作线程
工作线程
接收
工作线程
发送
工作线程
工作线程
工作线程

解决方案——线程（进程）

- 平行模式的弊端：
- 内存占用过大
- 并没有完全摆脱共享数据的问题

解决方案——线程（进程）

- 问题：
- 假设平行进程需要多次客户端交互，由 ID 标识，接受线程如何将后续消息交给正确的线程处理
- 接收线程自己计算？
- 如果接收线程是性能瓶颈怎么办？
- 共享数据，所有的平行线程共同访问
- 这就没有性能问题了么？

共享数据和锁

- 终于说到恶梦般的共享数据：
- c++ 和 java 一类的对象语言，均有隐含的共享数据——成员变量，入门选手根本无从理解
- 搞清楚共享数据后，加锁（数据同步）
- 诞生了各种锁：
- 悲观的——信号量，互斥
- 自私的——自旋锁（又一个超越人类智力的，每个锁时长都需要了解计算类型）
- 乐观的以及无锁化—— CAS ， STM

共享数据和锁

- 锁的性能越来越高？
- 魔咒：单核极限
- 排他性临界区和自私派：同时只有一个线程在运行
- 乐观派：只有一个线程的计算有效

共享数据和锁

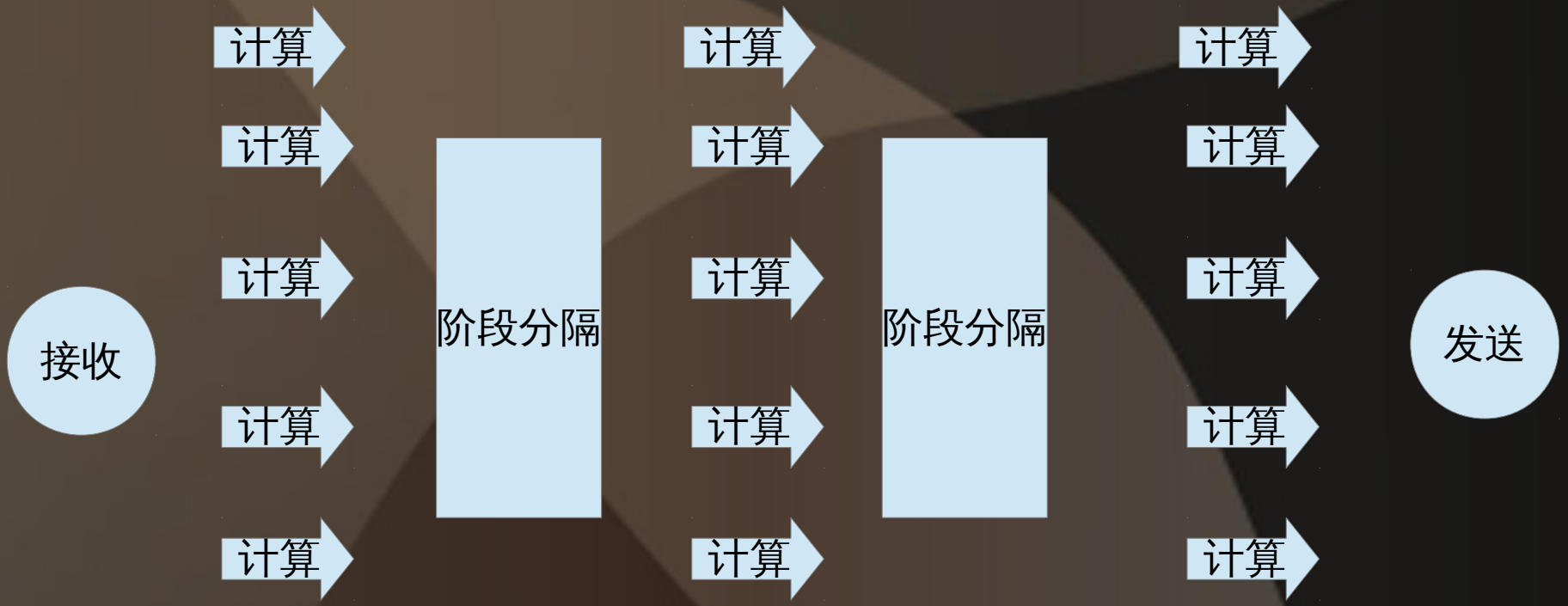
- 高并发的一下步需求：高并行
- 对比操作系统，加锁后，理应变成异步调用，让其他计算继续，但是线程的调度粒度不够

同步和异步

- 思想精髓：
- 同步是指发出请求，直到得到回复后才能进行下一步计算，所有的函数调用都是同步的。
- 异步是指，不需要
- 异步相当于建立一个并行任务，同时将调度外推至外部系统，比如操作系统
- 特别说明，阻塞和同步没关系，两个范畴的概念

同步和异步

- 多级同步需要状态保存
- 异步没有状态
- 异步架构下，似是而非的流水线架构
- 考虑一下对用户的请求和反馈，架构时我们可以利用用户的内存资源，让用户保存状态，同步，而我们的系统异步响应（君子善假于物），减少我们的系统消耗并提供更好的服务


接收
计算
计算
计算
计算
计算
阶段分隔
计算
计算
计算
计算
计算
阶段分隔
计算
计算
计算
计算
计算
发送

再吐槽一下线程

- 线程的内存管理无法特异化：
- C++/C 等简单粗暴的内存管理，性能差
- Java 一类的垃圾收集管理到针尖
- 高稳定性的隐忧：异常与共享运行时栈
- 你防得住自已，防不住别人
- 实时（过会儿再谈）

总结一下我们的技术需求

- 调度
- 异步（无状态）
- 快速的数据交换（来自线程）
- 超高并发（大内存，堆）
- 全速并行（多核，无锁）
- 高效及特化的内存管理
- 稳定
- 实时

Actor

- Actor 给了一切我们想要的
- erlang 做了一切我们该做的

Actor

- Actor 可以认为是一个独立对象，拥有自己的状态
- actor 有自己的运行时栈，和独立的内存（意义上，同时也是实现上）
- actor 彼此独立互不共享状态，一个死了其他的还活着
- Actor 靠消息交互
- erlang 的 link 机制，稳定性加强，“别人知道我死了，死得有多惨”

Actor

- 并发：多个 actor 即是并发
- 并行： actor 被分配到多个线程上即是并行
- 调度：采用平均化调度，因为大多数 actor 身份是等价的，平均化是合理的，同时提供的优先级做特别安排
- 数据交互：概念上的消息在内存中实际上是少量复制和引用，非常高效
- 锁：消息排队， actor 最坏情况下与锁同级
- 内存管理： actor 私有内存， actor 自己运行垃圾收集，并在退出时整块的回收。

回到刚才的问题

- 单 actor 自己计算，生成新的 actor 任务处理
- 性能极限：单核，顺序计算
- 管理的数据：所有
- 与平行多线程等价

回到刚才的问题

- 单 actor 接收，生成多个 actor 计算及查找
- 性能极限：单核顺序写，多核并行查找
- 管理数据量：全部
- 比平行多线程查找快，数据管理同级
- actor 数量超过核数，调度需求多

回到刚才的问题

- 单个用于接收的 actor ，与核数相对应的多个用于分配生成的 actor
- 接收 actor 只转发，由所有 actor 确定是否该由自己管理
- 分配算法计算量是 n 倍（其实很少，求个余），查找成本 1/n ，时间成本整体下降
- 我认为数学上能证明已经是时间最优了

actor 额外的好处，超越基本线程

- 稳定性：每个 actor 都可以异常退出，但是其他任务不受影响，整体上更稳定， link 机制可以还可以验尸，优化更容易，系统更稳定。
- 资源收集：基于 link 机制，无论资源使用者如何退出，管理者都可以将资源收回
- 实时，以及服务稳定性：设定超时消息，超时即向用户发送内部错误的回复，超时计算以及内部崩溃变成同线，内部错误对用户也友好
- 节省计算资源：最节省的方式不是更好的算法，而是“兄弟，你干点别的吧，用户不需要了”

实时性与服务的闲话

- 如何确定系统已经饱和？
- 资源上限？ cpu 阀值？内存告警？
- 遇到只占并发，不发数据的攻击怎么办？
- 用户不是攻击，只是网络太慢了怎么办？
- 我认为时间阀值是最基本合理的统一方案，超时即是：能力饱和，或是外部原因不值得继续服务。

终于跟 *lisp* 有点关系

- erlang 作为一个平台，已经像 java 一样有其他语言运行在其虚拟机上
- 基本的前端实现： lfe
- 完全的新实现： joxa

谢谢，有没有问题