并行需求下的Scala和Erlang比较

QCon Beijing 2010.04

Caoyuan (NetBeans Dream Team Member)

Regal Finance 宏爵财经资讯（北京）有限公司

http://blogtrader.net
@dcaoyuan

需求

- 秒级数据的全市场实时扫描、预警(每秒计算上百万个指标)
- 同时支持上万用户实时长连接
- 人工智能（机器学习）等数据挖掘手段 – 聚类、图形模式匹配、趋势拟合

难点

- 2008年开始的转折点：
  - 单CPU计算能力提高有限 – 多核时代的到来
  - 多核、并发计算的难度
    - 从语言到计算模型
    - 高频数据的内存存贮和管理
    - 分布式缓存机制

- 我们是一个适时的进入者吗?

天河一号 - 千万亿次超级计算机

- 每秒钟1206万亿次的峰值速度
- 由103台机柜x1.5吨=155吨，耗资6亿
- 6144个CPU、5120个GPU
- 全系统运行情况下
  - 每小时耗电1280度
  - 消耗功率1280千瓦
  - 每天3万度：电费2万，16吨煤
- 但是，硬件CPU利用效率只有不到60%

我们能怎么做（没有天河一号）

- 并行计算的瓶颈主要在软件（语言和模型）
  - Fortress，Erlang，Scala => Scala
    - 并行运算
    - 更简洁的代码、快速的开发
    - DSL – 自定义指标、辅助开发、调试 – 集成开发环境
  - 模型-分解成：可并行、串行
    - Map-Reduce, Fork-Join
- 用近可能低的成本实现高性能计算

Actor模型 – Erlang和Scala的并行方案

简单说，Erlang和Scala针对并行需求的主要方案都是Actor模型

- Erlang
  - Erlang VM中的轻量级进程作为Actor单元
  - 内建语法支持
- Scala
  - Actor作为普通对象代理给JVM的线程池
  - 通过actor库引入

Actor模型 - 一切都是Actor

Actor是指这样一种计算实体，它能并行地在接收到消息时作出如下反应：

- 发送消息给其它Actors （交互）

```
actor ! ChangeYourPosition
Pid ! change_your_position
```

- 创建新的Actors（加入参与者）

```
val a = actor {
  def loop(i: Int): Unit =
    react {
      case i: Int => loop(i + 1)
      case Exit => println(i)
    }
  loop(0)
}
```

```
loop(I) ->
  receive
    I -> I, loop(I + 1);
    exit -> io:format("~p~n", [I])
  end
end
Pid = spawn(fun loop/1, [0])
```

- 指定接收到下一个消息时的行为（状态转换）
  - Actor维护自己的状态
  - 状态保持在loop函数调用栈中(Scala、Erlang)，或者在域变量中(Scala)
  - 改变状态的唯一方式是接收到其它Actors的消息

Actor模型 – 并行地

前述Actor的特点看上去像OO里的对象而已，但并行能力还要考虑：

- 更彻底的空间的解偶
  - OO中的对象一定程度上实现了空间的解偶
  - 理想的Actor之间除了不变量外不共享任何东西，彻底解偶
  - 不变量(immutable)不随时间变化，是并行安全的

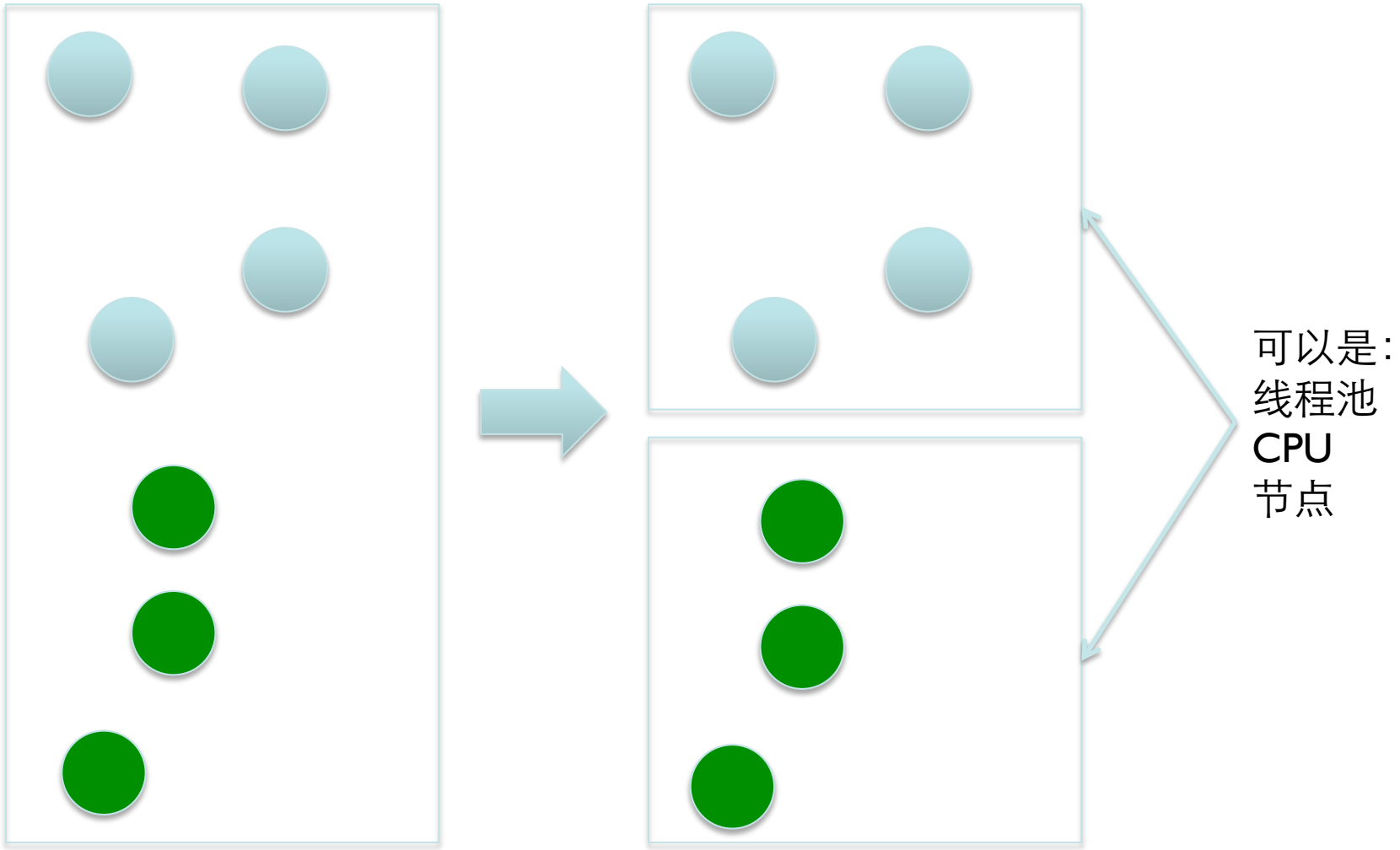
- 更彻底的时间的解偶
  - 线程一定程度上实现了时间的解偶
  - Actor采用异步消息机制，“并行”接收消息，但“串行”处理，可以避免同时修改内部状态所引发的冲突
  - Actor的状态是自己所有内部状态的集合，只取决于自己的当前状态和下一个消息，实际上整个Actor的状态也是“串行”改变的
  - 消息机制使得“并行”到了Actor这一级别全部转换成“串行”，不再需要Actor内部及Actor级别的锁

Actor模型 – 最小的计算颗粒?

Actor模型的目的：寻找能并行、分布计算的最小颗粒

- 时间与空间的解偶
- 最小的可并行或分布计算的颗粒
- 调度器只需在这个粒度上调度，或者说，Actor可以分配、分布、调度到不同的时间片 、不同的CPU、乃至不同的节点
- 越细的粒度越容易充分利用资源

Actor模型 – 并行和分布的一致

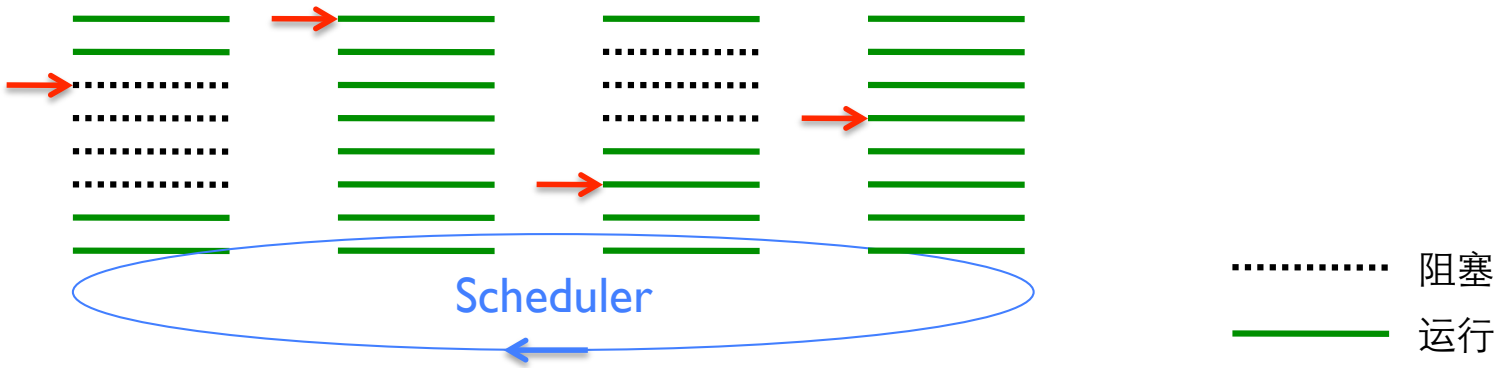


Actor模型 – 理想 vs 现实

理想	现实
每个对象都是Actor	直接调用效率要快的多，消息传递需要付出性能代价
每个Actor都独享自己的CPU	Actors只能共享有限个CPU的时间片
不共享除常量外的任何东西，行为完全由自己的私有状态和下一个消息决定。	拷贝需要创建新的实例，付出性能代价
完全没有副作用，彻底与赋值顺序（时间）无关	完全没有副作用就是没有现实作用

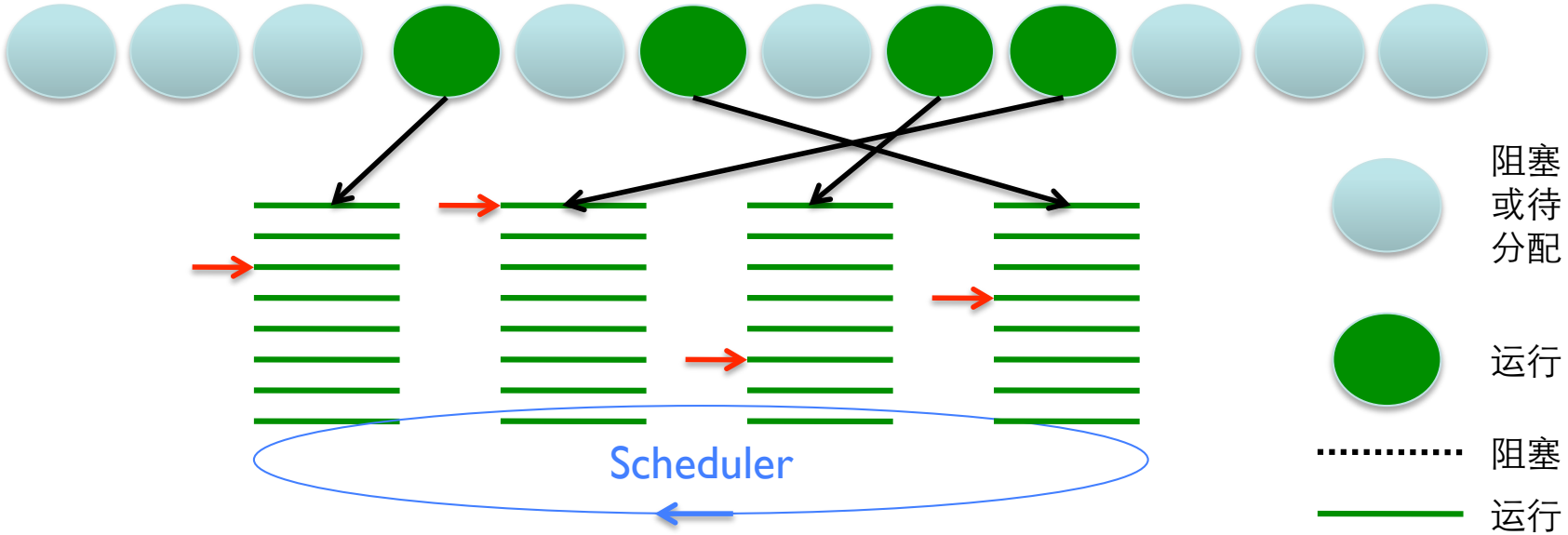
现实情况下如何做到资源的有效分时利用?

JVM的线程



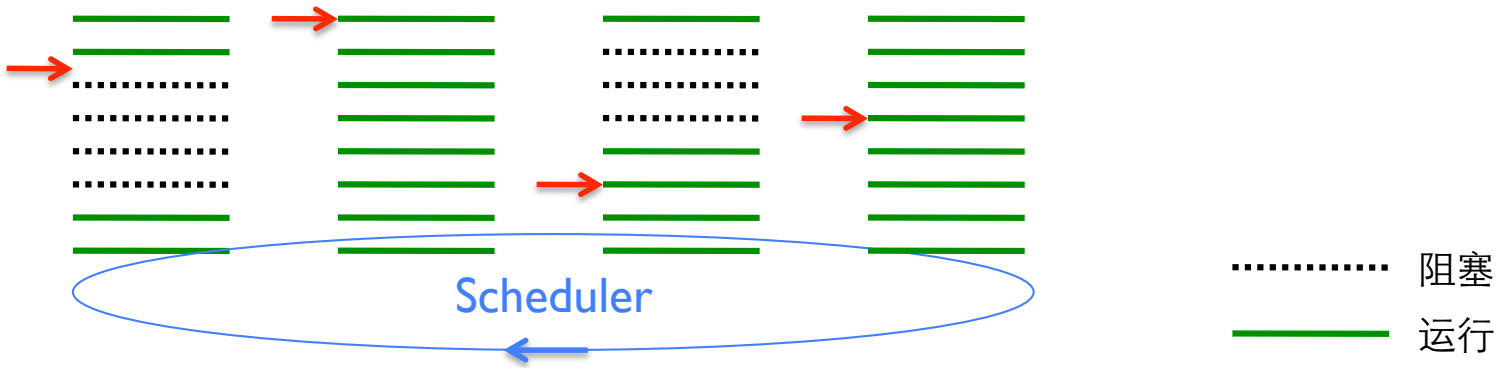
- 线程的数据结构较大，要分配足够大的栈以保存运行时上下文，包括寄存器值、当前指令位置、方法调用链的变量（其中包括了对共享堆中的对象的引用）等，是珍贵资源（数千个）
- 线程在被IO阻塞时，是一种被动阻塞，需要具体处理
- 线程在阻塞时不会释放除CPU外所占用的资源，但又不处在运行状态，是一种浪费

Scala在JVM上的Actor



- Actor是普通的Object，可以非常轻量（上百万个）
- Actor被代理给线程运行
- Actor在等待接收下一个消息时主动阻塞
- Actor在主动阻塞时会主动释放所占线程，释放的线程可以立即调度给其它处在尚未获得线程的等候队列中的Actor使用
- 阻塞的是Actors而不再是线程。线程交替调度给不同的Actors，总是繁忙地工作中（处在运行状态），从而得到了充分利用
- 但目前Scala中的IO阻塞仍然是被动阻塞，除非特别处理，否则遇到IO阻塞的Actor不会释放线程

Erlang中的Actor – 轻量Process



- Erlang中的Actor是轻量Process，本身占用资源少（上百万个）
- Process在等待接收下一个消息时，主动阻塞
- Process之间几乎不共享可变量，创建、释放和切换的代价非常小
- Process在主动阻塞时不需释放所占有资源，因为无所谓
- Erlang的IO库从一开始就经过仔细设计，都被代理给 Process，并使用异步消息机制通知IO完成，从而成为主动阻塞

异步消息机制的优点

- 异步消息机制使得“并行”到了Actor级别转换为内部的“串行”
  - “并行”体现在Actors各行其事及互发消息，但对单个Actor而言，则是顺序处理接收到的消息，避免了状态修改的冲突
- 异步消息机制是一种主动阻塞机制，可以主动释放资源
  - 将阻塞都转化为消息等待，实现了阻塞的一致性和主动性（对比线程的各种sleep阻塞、IO阻塞、锁阻塞、条件阻塞等）
  - 多个并行主体的协作如果只通过异步消息传递实现，也就自然实现了主动阻塞，从而实现资源最有效的分时利用
- 并行与分布现在行为完全一致（都是通过消息传递实现互操作）

Scala还需要一个Actor和NIO完美结合的库

Erlang vs Scala – 调度

- Erlang对Process采取公平调度策略
  - 对Process来者不拒，立即开始服务，获得平均分配时间片
  - 同时服务大量的Process可能导致服务质量下降，最终谁也服务不好
  - 单个Process获得的CPU能力可能随系统处理能力和Process的数量变化而动荡，导致很难估算其处理能力
    - 案例：异步log
      - 2-core时，能处理140个请求，process数为200，每个process平均获得CPU能力为1 / 200 * 2 Core = 1%
      - 8-core时，能处理700个请求，process数为980，每个process平均获得CPU能力为1 / 980 * 8 Core = 0.82%

Erlang vs Scala – 调度

- Scala的Actor代理给线程池
  - 对Actor来者不拒，但只代理给有限数量的线程，尚未获得线程的Actors先放在等候队列中
  - 线程池的大小由运行中系统的资源能力和正在服务的Actor任务的完成效率而动态调节
  - 优先保证正在进行的服务
  - 单例Actor可以代理给固定线程

Erlang vs Scala – 计算还是切换

- Erlang的Process切换非常快
  - 不共享可变量，独立的内存空间
  - 释放时杀掉即可(GC可以立即完成)
  - 重起一个也很简单(没有共享状态)
  - VM的计算能力很弱，尤其处理文本时(二进制数据尚可)
  - 适合于计算时间与上下文切换时间相近时，比如实时大规模短消息处理
  - 案例：某银行手机银行服务(基于消息转发)

Erlang vs Scala – 计算还是切换

- Scala的Actor代理给线程，上下文切换比Erlang慢一个数量级左右
  - 要自己保证不共享可变量
  - 释放时线程要做上下文切换
  - VM的计算能力很强，目前最好的工业VM
  - 适合于计算时间比上下文切换时间更显著时
  - 现代JVM的线程上下文切换时间在微秒级
  - 案例：金融市场实时指标并行计算

Erlang vs Scala – 计算还是切换

```
object ThreadRing {
  val (nTokens, nProcs) = (20000, 10000)

  def main(args: Array[String]) {
    val last= Ring(null, 1)
    val h = Range(nProcs, 2, -1).foldLeft(last){(acc, i) => Ring(acc, i)}
    last.next = h
    h ! nTokens
  }

  case class Ring(var next: Ring, id: Int) extends Actor {
    start
    def act = loop {
      react {
        case 1 =>
          println("Done")
          println(id)
          System.exit(0)
        case token: Int =>
          next ! token - 1
          println(token)
          //fib(N_FIB)
      }
    }
  }
}
```

Erlang vs Scala – 计算还是切换

```
-module(threadring).
-export([main/0, ring/2]).
-define(N_TOKENS, 20000).
-define(N_PROCS, 10000).

main() ->
   start(?N_TOKENS).

start(NToken) ->
   H = lists:foldl(fun(Id, Pid) ->
                           spawn(threadring, ring, [Id, Pid])
                   end, self(), lists:seq(?N_PROCS, 2, -1)),
   H ! NToken,
   ring(1, H).

ring(Id, Pid) ->
   receive
      1 ->
         io:fwrite(”~p~n", [done]),
         io:fwrite("~b~n", [Id]),
         erlang:halt();
      Token ->
         Pid ! Token - 1,
         io:fwrite("~b~n", [Token]),
         //fib(?N_FIB),
         ring(Id, Pid)
   end.
```

Erlang vs Scala –FP还是OO+FP

- Erlang是动态类型、函数式语言
  - 函数式语言保证代码质量
  - Process可以模拟对象，但调用效率是大问题
  - 缺少静态类型检查给重构带来困难，对业务逻辑复杂的应用未必合适

Erlang vs Scala –FP还是OO+FP

- Scala是静态类型、OO+FP融合语言
  - 函数式语言提供另一种抽象能力
  - Actor是普通对象，可以直接作为业务对象，也可以一个业务对象持有一个或多个Actors
  - 通过继承和偏函数，子类可以重载和级联Actor的行为
  - 静态类型检查带来的优点，尤其是业务逻辑复杂的应用

Q & A

是的，我在写一本Scala的书

谢谢！