Scala与反应式架构

王石冲

字节跳动大数据工程师

自我介绍

Scala程序员

《反应式设计模式》译者

字节跳动大数据工程师

目录

1. 为什么要反应式
2. 《反应式宣言》之演绎版
3. 基于Scala的反应式基础工具

目录

为什么要反应式


即时响应


Switchboard
Meeting speech
Broadcast speec
Switchboard
lular
itchboard
Microsoft
1993
2000
2006
2010
2017
应对失败


应对负载


这是一个很好的应用。
应对输入

即时响应

应对失败

应对负载

应对输入

目录

在京城外的西山上，

有一座远近闻名的寺庙，发生了三个小故事，非常值得程序员深思……

故事一：

方丈派和尚去挑水，结果一个和尚挑水吃，
两个和尚抬水吃，三个和尚没水吃……

启发

方丈为了让僧人能喝更多的水，派了更多和尚去挑水，但是为什么去挑水的和尚越多，吃的水会越少呢？

启发

方丈为了让僧人能喝更多的水，派了更多和尚去挑水，但是为什么去挑水的和尚越多，吃的水会越少呢？

- **症结**
  - **单点瓶颈 （只有一根扁担）**
  - **并发协调 （没有人对三个和尚进行资源管理和任务分配）**

启发

方丈为了让僧人能喝更多的水，派了更多和尚去挑水，但是为什么去挑水的和尚越多，吃的水会越少呢？

- 症结
  - 单点瓶颈 （只有一根扁担）
  - 并发协调 （没有人对三个和尚进行资源管理和任务分配）
- 解决办法
  - 减少依赖性（让和尚们各自挑水，不要抬水）
  - 增加必须的资源（给每个派过来的和尚配扁担和水桶）
  - 分片（在有资源的地方建分寺，将挂单的和尚派过去）

《反应式宣言》之弹性：

系统在不断变化的工作负载之下依然保持即时响应性。反应式系统可以对输入的速率变化做出反应，比如通过增加或者减少被分配用于服务这些输入的资源。 这意味着设计上并没有争用点和中央瓶颈， 得以进行组件的分片或者复制，并在它们之间分布负载。 通过提供相关的实时性能指标， 反应式系统能支持预测式以及反应式的伸缩算法。这些系统可以在常规的硬件以及软件平台上实现成本高效的弹性。

故事二：

方丈派小和尚接待来拜佛的程序员，小和尚因为前天被师兄欺负了，情绪很差，结果半路直接甩手走人，留下程序员呆若木鸡不知所措……

启发

遇到这种小和尚甩手不管的情况， 程序员应该怎么办？这种事情应该由程序员处理吗？

启发

遇到这种小和尚甩手不管的情况， 程序员应该怎么办？这种事情应该由程序员处理吗？

- 分清楚两种不同的概念
  - 检验错误 (Validation Error)，是模块之间正常协议的一部分
  - 失败(Failure)，则是正常协议无法继续履行的场景

启发

遇到这种小和尚甩手不管的情况， 程序员应该怎么办？这种事情应该由程序员处理吗？

- 分清楚两种不同的概念
  - 检验错误 (Validation Error)，是模块之间正常协议的一部分
  - 失败(Failure)，则是正常协议无法继续履行的场景
- 处理失败的办法
  - 模块化分割系统，隔离失败
  - 明确所有权和层级系统，由拥有失败发生的模块的所有者来处理异常
  - 引入监督者和监督策略，Let it crash.
  - 在多台硬件上进行复制，以防止硬件故障的时候，发生数据丢失和系统的不可恢复。

《反应式宣言》之回弹性：

指系统在出现失败时依然保持即时响应性的能力。 回弹性是通过复制、 遏制、 隔离以及委托来实现的。 失败的扩散被遏制在了每个组件内部， 与其他组件相互隔离， 从而确保系统某部分的失败不会危及整个系统，并能独立恢复。 每个组件的恢复都被委托给了另一个（外部的）组件， 此外，在必要时可以通过复制来保证高可用性。 组件的客户端不再承担组件失败的处理。

故事三：

国庆长假，来寺庙游玩、参观、拜佛的人越来越多，逐渐超过了寺庙的承载能力……

可控的流量

假设往来的游客可以由本寺分流给分寺，则通过协商可以控制流量。

- **拉取模式**
  - **告知上游自己的任务需求（比如，分寺告知本寺，要接待10名游客）**
  - **上游按照下游的需求给出任务（本寺分派少于10名的游客）**
  - **下游根据自己当前处理能力，继续告知上游自己的需求（在接待游客的同时，告知本寺自己还能接待5个游客）**
- **优点：**
  - **动态推拉模式**
  - **按照处理能力来分配下游负载**

不可控的流量

但是对于本寺，由于游客来自四面八方，流量无法控制

- **托管队列模式与丢弃模式**
  - **维护一条队列（让游客排队）**
  - **根据下游汇报的处理能力，分配任务（游客）过去**
  - **当队列超过一定长度的时候，概率性入队；超过极限时，拒绝入队。**
- **优点：**
  - **平滑流量**
  - **监控队列信息，并作出反应**

《反应式宣言》之及时响应性：

只要有可能， 系统就会及时地做出响应。 即时响应是可用性和实用性的基石， 而更加重要的是，即时响应意味着可以快速地检测到问题并且有效地对其进行处理。 即时响应的系统专注于提供快速而一致的响应时间， 确立可靠的反馈上限， 以提供一致的服务质量。 这种一致的行为转而将简化错误处理、 建立最终用户的信任并促使用户与系统作进一步的互动。

《反应式宣言》之消息驱动：

反应式系统依赖异步的消息传递，从而确保了松耦合、隔离、位置透明的组件之间有着明确边界。 这一边界还提供了将失败作为消息委托出去的手段。 使用显式的消息传递，可以通过在系统中塑造并监视消息流队列， 并在必要时应用回压， 从而实现负载管理、 弹性以及流量控制。 使用位置透明的消息传递作为通信的手段， 使得跨集群或者在单个主机中使用相同的结构成分和语义来管理失败成为了可能。 非阻塞的通信使得接收者可以只在活动时才消耗资源， 从而减少系统开销。

目录

函数式编程

函数是一等公民

不可变数据结构 | 引用透明和副作用
完善的类型系统

函数式编程

函数是一等公民

```scala
def fibonacci(n: Int): BigDecimal = {

  @tailrec
  def loop(n: Int, lastAcc: BigDecimal, acc: BigDecimal):BigDecimal =
    if (n < 2) acc
    else loop(n - 1, acc, lastAcc + acc)

  loop(n, lastAcc = 1, acc = 1)
}

val fibValue: Int => BigDecimal = fibonacci

case class ImmutableData(value: String, fun: Int => Int)
```

不可变数据结构 | 引用透明和副作用
完善的类型系统

Future & Promise

并发利器

异步和非阻塞 | 避免回调地狱
Monad

Future & Promise

并发利器

```scala
val bigNumber = 1000000
val result: BigDecimal = fibonacci(bigNumber)
val start: Long = System.currentTimeMillis()

def heavyTask(): BigDecimal = fibonacci(bigNumber)

val taskA: Future[BigDecimal] = Future(heavyTask())
val taskB: Future[BigDecimal] = Future(heavyTask())
val taskC: Future[BigDecimal] = Future(heavyTask())

def assetEqual(calculated: BigDecimal, expected: BigDecimal, step: Int): Future[Unit] = {
  if (calculated == expected) Future.successful(())
  else Future.failed(new Exception(s"第${step}计算结果不正确！"))
}

val calculation = for {
  rA <- taskA
  _ <- assetEqual(rA, result, step = 1)
  rB <- taskB
  _ <- assetEqual(rB, result, step = 2)
  rC <- taskC
  _ <- assetEqual(rC, result, step = 3)
  sum = rA + rB + rC
  _ <- assetEqual(sum, result + result + result, step = 4)
} yield {
  sum
}

Await.result(calculation, 1.minute)
```

异步和非阻塞 | 避免回调地狱

Monad

Akka

JVM上最好的Actor模型实现

消息驱动 | 层级结构 | 监督策略 | 位置透明性 | Actor内部无并发 | Akka Stream

Akka

JVM上最好的Actor模型实现

```
class FibonacciActor extends Actor {
  override def receive: Receive = {
    case n: Int if n < 0 =>
      throw new IllegalArgumentException("非法参数")
    case n: Int =>
      sender() ! fibonacci(n)
  }
}

class Supervisor extends Actor {
  var child = context.actorOf(Props(new FibonacciActor))

  override def receive: Receive = {
    case "calculate" =>
      child ! bigNumber
    case "restartResult" =>
      child ! -1
    case result: BigDecimal =>
      println(result)
  }

  override def supervisorStrategy: SupervisorStrategy = OneForOneStrategy() {
    case _: IllegalArgumentException =>
      Restart
  }
}

val supervisor = system.actorOf(Props(new Supervisor), name = "s")
supervisor ! "calculate"
supervisor ! "restartResult"
supervisor ! "calculate"
supervisor ! "restartResult"
supervisor ! "calculate"
```

消息驱动 | 层级结构 | 监督策略 | 位置透明性 | Actor内部无并发 | Akka Stream

其他设计模式

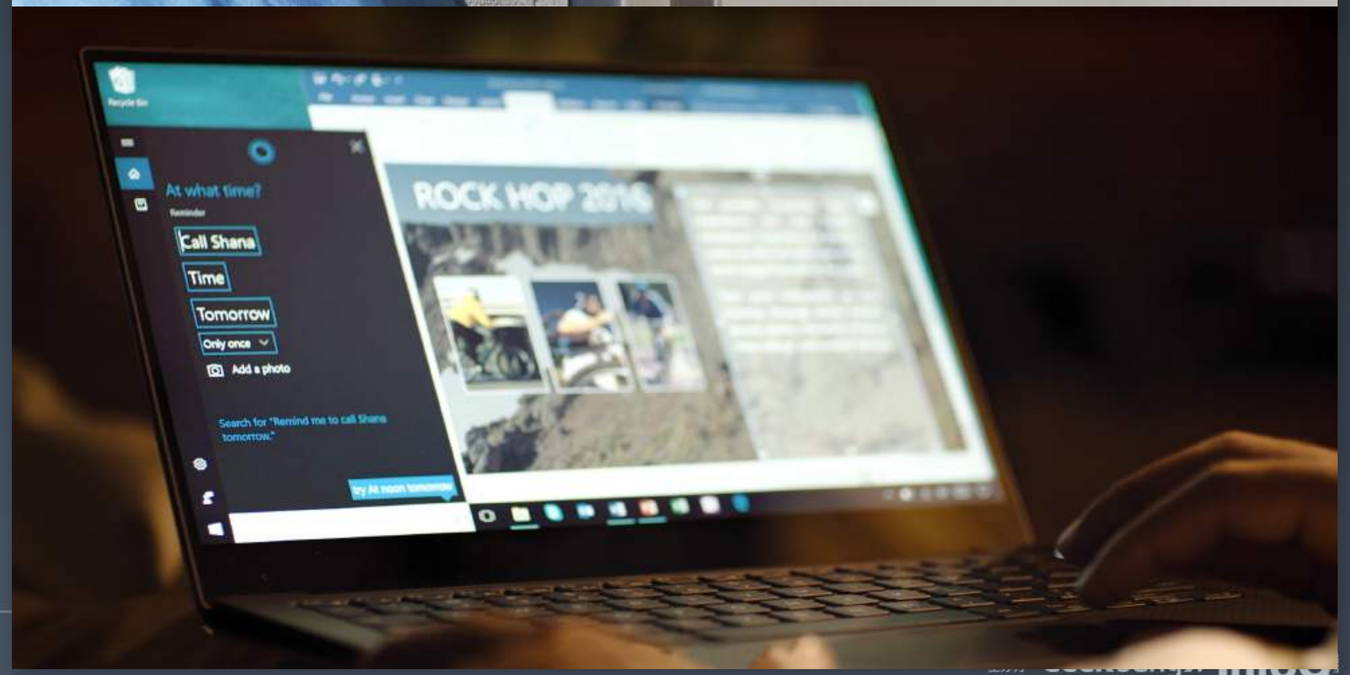
反应式设计模式

容错与恢复模式	简单组件模式 / 错误内核模式 / 放任崩溃模式 / 断路器模式
复制模式	主动-被动复制模式 / 多主复制模式 / 主动-主动动复制模式
资源管理模式	资源封装模式 / 资源借贷模式 / 复杂命令模式 / 资源池模式 / 托管阻塞模式
消息流模式	请求-响应模式 / 消息自包含模式 / 询问模式 / 转发流模式 / 聚合器模式 / 事务序列模式 / 可靠投递模式

反应式设计模式

流量控制模式	**拉取模式** **托管队列模式** **丢弃模式** **限流模式**
状态管理和持久化模式	**领域对象模式** **分片模式** **事件溯源模式** **事件流模式**

Q & A


At what time?
Reminder
Call Shana
Time
Tomorrow
Only once
Add a photo
Search for "Remind me to call Shana tomorrow."
ROCK HOP 2016

Geekbang 极客邦科技 | InfoQ 极客传媒

InfoQ官网 全新改版上线

促进软件开发领域知识与创新的传播

关注InfoQ网站
第一时间浏览原创IT新闻资讯

免费下载迷你书
阅读一线开发者的技术干货

THANKS!

QCon 10th

ByteDance 字节跳动