APMCon

面向数据应用的 Reactive微服务架构设计与实践

恒丰银行科技开发部 曾光尧

需求背景

互联网+场景时代需求

APM

数据服务能力需求提升1-2个量级

客户行为预测

01.实时采集客户点击数据
02.被动响应变为主动预测
03.个性化产品与服务资讯

万物互联

01.手机与穿戴设备的实时海量数据
02.地理位置与行动轨迹预测需求
03.本行、第三方产品与服务推荐

场景化营销

01.尊重用户体验
02.兴趣需求搜索触发
03.资源整合+精准营销

金融安全需求

APM

稳定可靠的服务质量

应对突发访问压力

01.双十一购物狂欢交易请求冲击
02.应对金融突发事件带来的访问压力
03.始终稳定的最大服务能力

系统自我修复能力

01.规避程序缺陷累积的全局服务破坏
02.子系统问题的有效隔离
03.有监督的服务自治
04.电信级的服务可用能力

技术架构需求

APM

易开发、易部署、低成本

云部署

01.可弹性扩容和下线
02.支持Docker容器云
03.快速连接各类设备的能力

大数据生态集成

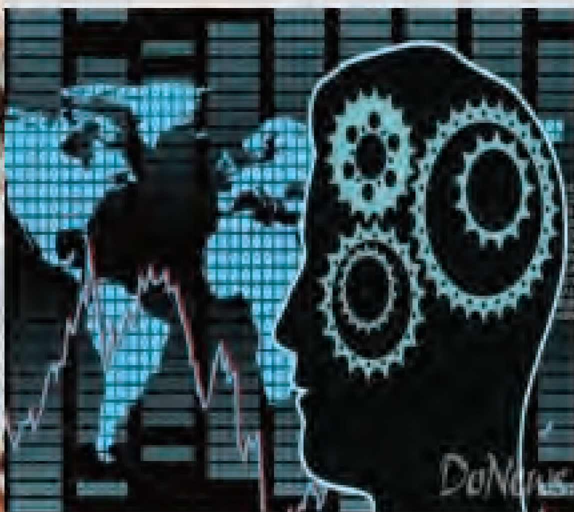
01.大数据处理分析平台
02.实时流式处理平台
03.数据服务的快速发布部署
04.简单开发，屏蔽复杂技术细节

研究概述

架构设计研究目标

APM

构建高性能可弹性扩展的大数据应用服务架构



高并发低延迟的服务能力

01.单节点支持1万以上设备接入
02.单节点数万Qps的处理能力
03.支持并行处理降低响应延迟

稳定高可用的服务质量

01.峰值压力下能提供稳定服务输出
02.局部缺陷问题隔离与快速恢复
03.硬件单点故障的容错能力

易开发、弹性部署

01.概念简单的开发工具支持
02.支持动态部署和扩容服务
03.支持服务质量差异化管控

主流应用服务架构的能力局限性

以Tomcat、Weblogic为代表的应用服务架构不能满足新时代要求

主流应用服务架构性能问题剖析

核心技术问题

主流的J2EE服务架构每节点吞吐量一般在1000-3000Qps，在多核硬件时代已不适应高并发低延时的用户需求。

1. 受内核调度成本限制，单节点同时支持的网络连接数有限
2. 每请求/连接对应一个线程，导致操作系统线程数过多；而线程占用的系统资源较多，切换代价太大
3. 高并发场景下线程间资源共享锁冲突，进一步降低系统资源利用率
4. 采用同步IO机制引起线程阻塞，导致服务吞吐量太低
5. 高容错的分布式并行编程实现困难，不能有效发挥多核资源优势

互联网企业高性能服务架构的技术特点

实现高并发低延迟服务能力的有效技术手段

提高网络连接数	epoll替代了传统的select/poll,实现更多的网络连接支持.
降低服务线程数	异步IO取代同步IO，通过异步回调机制单个线程服务多个客户端连接，避免过多线程上下文切换导致系统CPU消耗过大.
降低锁冲突	事件驱动的微服务架构。如使用Actor模型利用消息传递方式实现数据的共享和修改，避免了锁冲突
大任务切分	pipeline机制细分子任务，在分布式的管道线协作完成业务功能,提升计算并行度和吞吐量.

选择Reactive微服务技术架构

单体应用 v.s. 微服务

- 单体应用：大而全
  - 包含多种能力
  - 模块接口
  - 成熟技术和中间件
  - 模块级别变更导致应用更新
  - 垂直扩展
- 微服务应用：解耦、自治、协同
  - 只完成一个功能，达到最好
  - 服务接口
  - 允许尝试新技术，如不同的语言
  - 服务内部变更不影响其他服务
  - 水平扩展

微服务技术架构是正确方向

Reactive微服务架构特征

- **react to events(高并发）**
  对事件立即反应，通过消息传递机制避免共享资源的锁冲突，降低线程资源需求
- **react to failure(高容错)**
  对失败立即反应，服务自治，在任何级别都可实现失败快速恢复的弹性系统
- **弹性部署**
  功能级服务可以灵活打包部署在不同节点，构建位置透明的集群服务体系，实现弹性扩容和差异化的硬件资源配置

事件驱动、弹性容错、实时响应

AKKA作为微服务架构基础

优势与特点

- **成熟可信**
  Spark底层架构，Ebay、Amazon的架构选择，2015Jax创新技术大奖
- **高性能**
  内嵌NIO框架,支持更多客户端连接，单节点每秒5000万消息处理；1GB内存250万Actor，消息传递机制实现分布式微服务协同、数据共享，消除资源锁需求
- **稳定可靠**
  Actor模型实现多层级自治监管机制，构建安全运行的防火墙和沙箱，微秒级的故障恢复
- **弹性部署**
  多种集群部署模式，远程服务透明访问，多种可配置的负载均衡策略

选择Akka作为微服务基础软件框架

电信级服务架构，品质之选

尚待解决的技术问题与基本思路

Akka作为微服务软件框架提供了基础平台支持，但仍存在如下问题需要我们通过设计更完善的应用服务架构方案加以解决

问题

高并发低延迟服务能力

- 缺乏微服务异步并行调度编程工具支持，有效降低多个服务调用的处理延迟

稳定可靠的服务质量

- 峰值压力下提供稳定服务输出的能力

弹性部署

- 运行时差异化的服务质量管控

设计实现支持分布式并行计算的新编程语言，实现多种并行处理范式，支持微服务的异步并行调用

微服务组件化，实现服务组件容器，实现服务组件实例的弹性创建和销毁，服务请求的排队、超时清理、过载阻断.

服务组件容器支持不同微服务的并发实例数、使用线程资源池、任务队列长度、超时时间、部署物理节点等动态配置项，以支持差异化的服务质量.

Skyline软件架构平台设计思路

APM

高并发低延迟服务

1. Zebra分布式并行服务语言通过增强语法支持异步并行服务调用，通过数据并行、指令并行、Pipeline等三种方式提升程序的运行速度，降低响应延迟时间
2. Zeroutine协程调度框架减少线程切换成本，构建完全无阻塞的程序运行机制，有效提高服务吞吐量

稳定可靠服务输出

1. 服务组件容器技术实现微服务实例的按需自适应调整和峰值压力下的排队缓存、超时清理和过载阻断机制
2. 组件容器隔离程序缺陷影响，实现异常崩溃后的快速重启
3. 契约式编程范式提升代码质量，隔离不同模块和应用系统的错误影响

弹性部署

1. 可动态配置的服务组件部署机制，支持多种集群部署模式
2. 从并发处理能力、单位时间最大吞吐量、线程资源等多个维度实现弹性的服务部署策略
3. 与Docker容器云技术结合实现硬件资源的弹性扩容

自主研发Skyline 软件架构平台，在Akka微服务框架的基础上，针对企业应用场景实现了上述能力增强

设计与实现

名称由来

自主研发的软件架构平台

Skyline-地平线

Zebra-斑马

自研企业级应用服务平台架构

01. 坚实的平台基础
02. 化繁为简的能力
03. 聚焦商业逻辑

原创分布式并行计算编程语言

01. 创新需要打破常规
02. 分布式并行计算为核心
03. 让服务协同更容易

Skyline平台技术架构

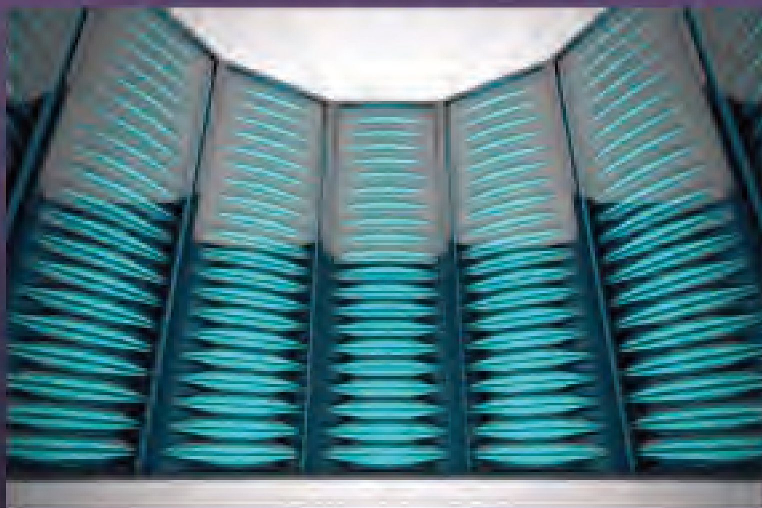
APM

Zebra分布式并行服务语言

APM□

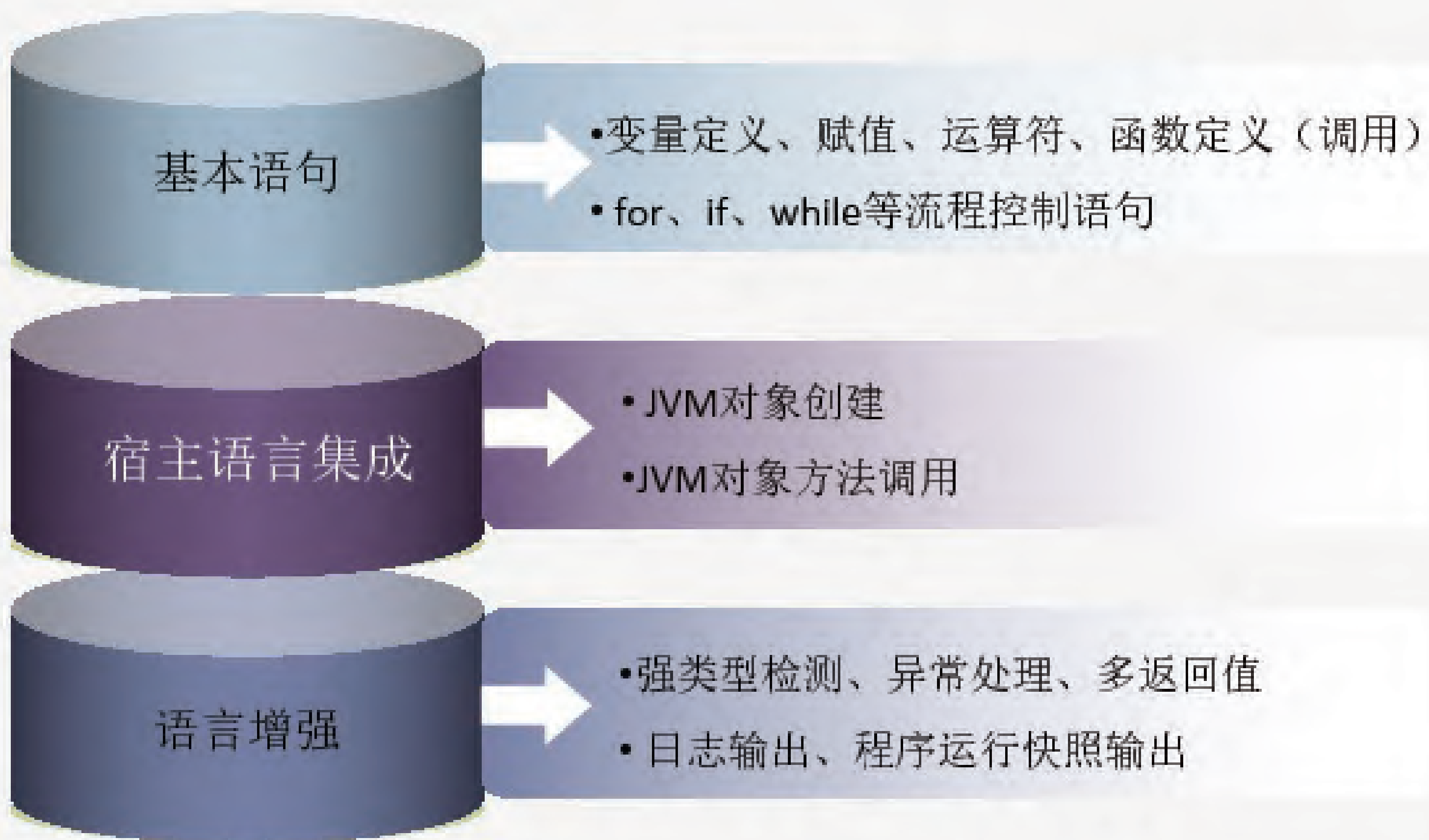
1. 简洁完整的过程式脚本语言
2. 同时支持数据并行和任务并行
3. 契约式编程范式确保软件质量
4. 强类型检查与STM软事务

5. 自然的语法简化异步通讯和并行计算编程难度
6. 支持动态部署和灰度测试
7. 可直接使用JVM兼容语言的生态软件资源
8. Zeroutine实现更高效的协程调度架构



Zebra基本语法介绍

APM



Zebra异步并行处理语法

异步并行处理语法是Zebra语言区分于其他语言的重要特征

原创技术优势

- **适用场景**

  客户信用评估并行调用多个外部数据接口；
  可视化仪表板应用并行调用多个数据库SQL；
  复杂业务逻辑的数据集合计算需求分拆多个数据子集实现并行计算

- **作用**

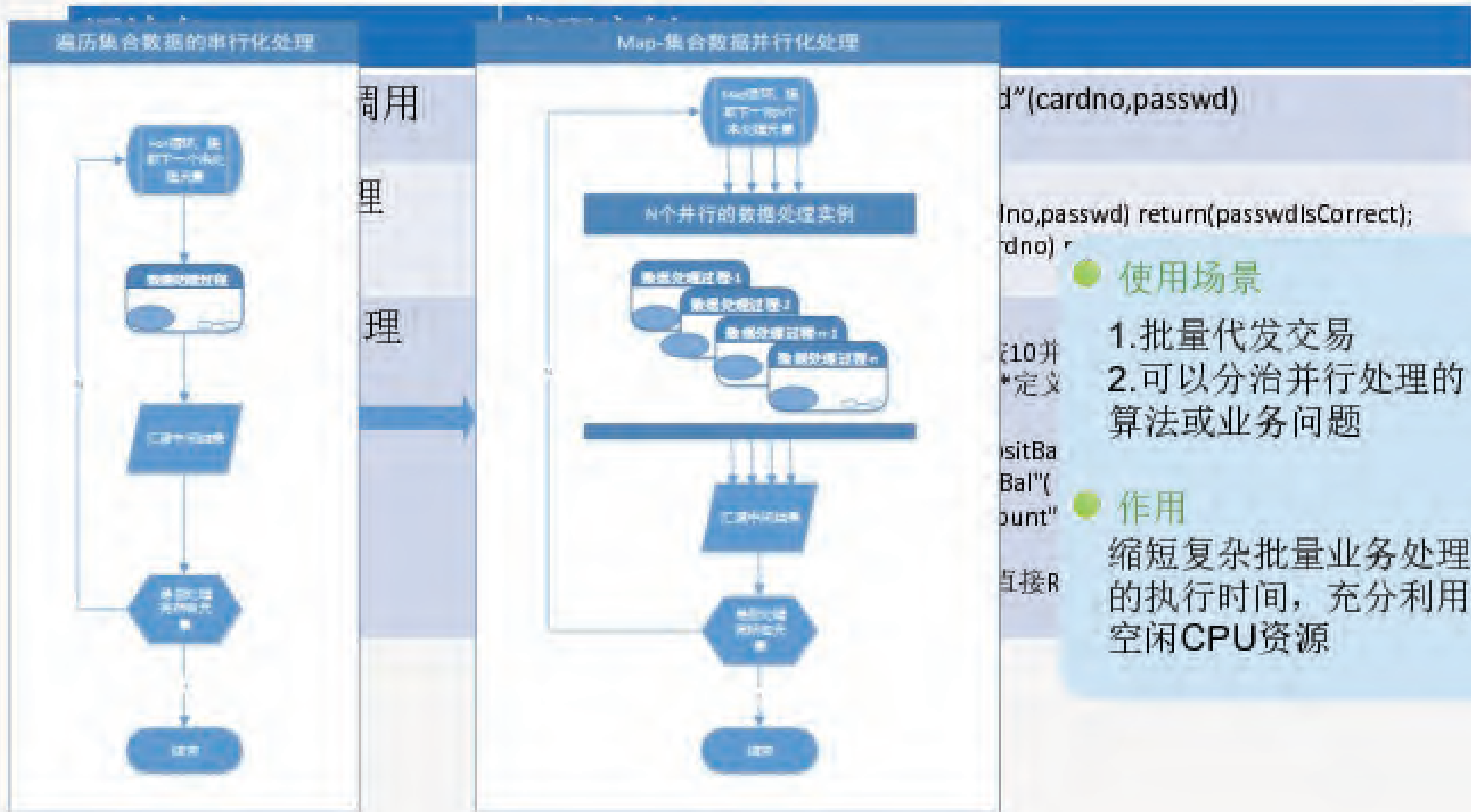
  通过异步服务调用和并行处理两种机制，缩短多个微服务组合调用的执行时间，实现数据集的分布式并行计算，提升服务吞吐量和降低响应延迟时间

- **相对其他语言和框架的技术优势**

  非常自然的语法，代码可读性强、调试更容易，完全屏蔽分布式并行计算的复杂技术细节。与此对照，大多数JAVA并行计算框架只支持本机多线程并行计算，类似Erlang、Go这类语言依赖于程序员对运行时序的控制，难以编写复杂的分布式并行处理程序

最简单的实现方式压榨机器资源潜力

Zebra异步并行处理语法



一个简化的业务编码示例

```
Function    withdrawal (cardno,accno,amt,pwd)
        par{/* 考虑到大部分客户取款能成功，使用并行审核机制*/
         async @"ws://sechost/checkpassword"(cardno,passwd) return(passwdIsCorrect) ;
         async @"ws://cardhost/checkCardState"(cardno) return(cardStateIsOk);
         async @"jdbc://depositDB/get"("select * from depositinfo where accno=?",accno) return(accinfo);
        }/*语言执行引擎处理所有的异步服务调用都返回后再往下执行 */
         /*集中做审核逻辑判断*/
        if(passwdIsCorrect == false) return InvokeResult(false,"密码检查错误");
        if(cardStateIsOk == false) return InvokeResult(false,"卡状态异常");
        if(accinfo.state != "0") return InvokeResult(false,"帐户状态异常");
        if(accinfo.balance<amt) return InvokeResult(false,"帐户余额不足") ;
        async @"jdbc://depositDB/update"("update depositinfo set balance = balance - ? where
accno=?",amt,accno);
        return InvokeResult(true,"交易成功") ;
End Function
```

Zebra契约式编程语法

语法名	代码实例
assert断言语句	assert(v1>10," v1>10约束条件不符");

precond前置条件	{ // 执行所有precond前置条件检查
	precond(p1 > 0 && p2<100，“step1 前置检查不通过，p1=${p1},p2=${p2}”);
	…… /* 其他处理语句*/
	}

postcond后置条件	{ // 开始前执行所有precond前置条件检查
	precond(p1 > 0 && p2<100);
	postcond(rltval>p1，“后置条件检查异常错误，rltval=${rltva},p1=${p1}”);
	…… /* 其他处理语句*/
	}// 结束前执行所有postcond后置条件检查

invariant不变式	{
	P1 = 10;
	P2 = 50;
	invariant(p1+p2>0,”p1+p2<=0”);
	……. //其后的每一条语句执行后触发检查所有不变式
	}//不变式效用终止

原子事务支持

APM

- 作用域外部变量保留执行前快照
- rollback可回滚全部外部变量值

- 事务启动前先获取提前声明的全部分布式资源锁权限
- 事务回滚或提交后自动释放事务锁

- 分布式事务管理器在服务端将分布式事务分为预处理、预提交、最终提交三个阶段
- 与两阶段提交协议相比，进一步提升了对单点硬件故障的容错能力

交易场景代码示例

APM

```
credit_accno="440011"; //贷方帐号
debit_accno="440022"; //借方帐号
amount=10000;  //发生额
succ_txcnt=0;
def credit_accinf,debit_accinf:AccountInfo; //帐户信息
atomic("accinfo/${credit_accno}","accinfo/${debit_accno
}"){ //获取帐户记录锁
  par{//并行查询
async@"shard://accinfo/query"(accno=credit_accno,
_id_= credit_accno)
return(credit_accinf =accinf);

async@"shard://accinfo/query"(accno=debit_accno,_id_
=debit_accno)
return(debit_accinf =accinf);
      }
  log"info","credit_bal=${credit_accinf.balance},
     debit_bal=${debit_accinf.balance}";
  balance_sum= credit_accinf.balance+
debit_accinf.balance; //求借贷账户余额
p_debit_bal = debit_accinf.balance;
  if(debit_accinf.balance>amount){
    precond(debit_accinf.stat=="0" &&
credit_accinf.stat=="0","账户状态不正常"); //前置条件
    postcond(debit_accinf.balance==p_debit_bal-amount,"
借方余额不正确"); //后置条件

invariant(debit_accinf.balance+credit_accinf.balance==
balance_sum,"借贷不相等");//不变式校验
    par{//并行更新
     async@"shard://accinfo/credit"(accno= credit_accno,
amount=amount,_id_= credit_accno)return
(credit_accinf =accinf);
     async@"shard://accinfo/debit"(accno= debit _accno,
amount=amount,_id_= debit _accno)return (debit
_accinf =accinf);
        }
  }else{
     rollback;
  }
  succ_txcnt=succ_txcnt+1;

}//执行三阶段事务提交，解锁
```

Zeroutine协程运行时架构

APM

服务组件容器-关键问题域

聚焦服务稳定可靠和弹性部署

解决之道

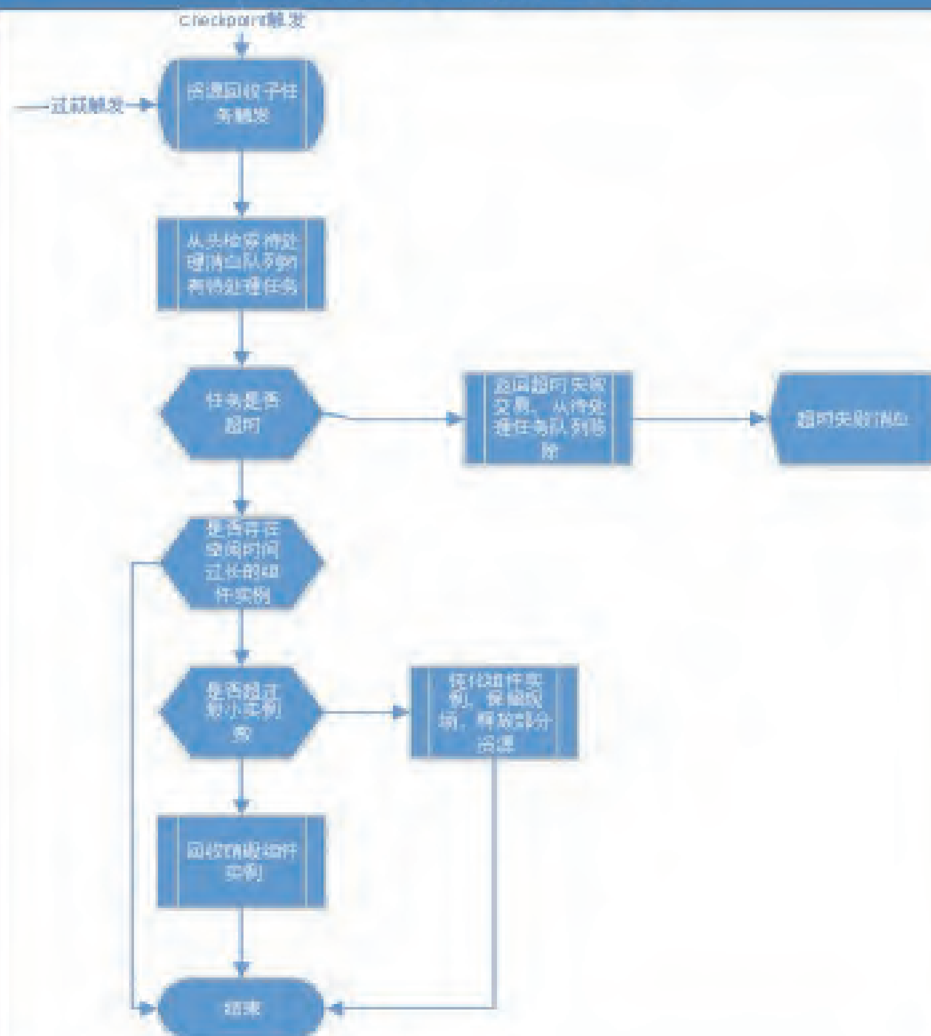
问题	描述	解决之道
问题1	怎么不被突发的瞬间交易峰值击垮	任务缓存队列、超时清理机制、过载阻断
问题2	怎么避免程序缺陷引起整个系统的瘫痪	组件容器承担对组件实例的监管职责，捕获异常，快速重启实例
问题3	如何针对不同业务优先级实施差异化的服务质量	差异化配置线程资源池、服务组件实例数、任务超时时间
问题4	服务器资源能否动态调整	Akka集群管理协议支持服务节点动态加入和离线；动态配置调整组件容器参数

服务组件容器-架构示意

APM

服务组件容器的处理逻辑

APM



实时流处理组件技术

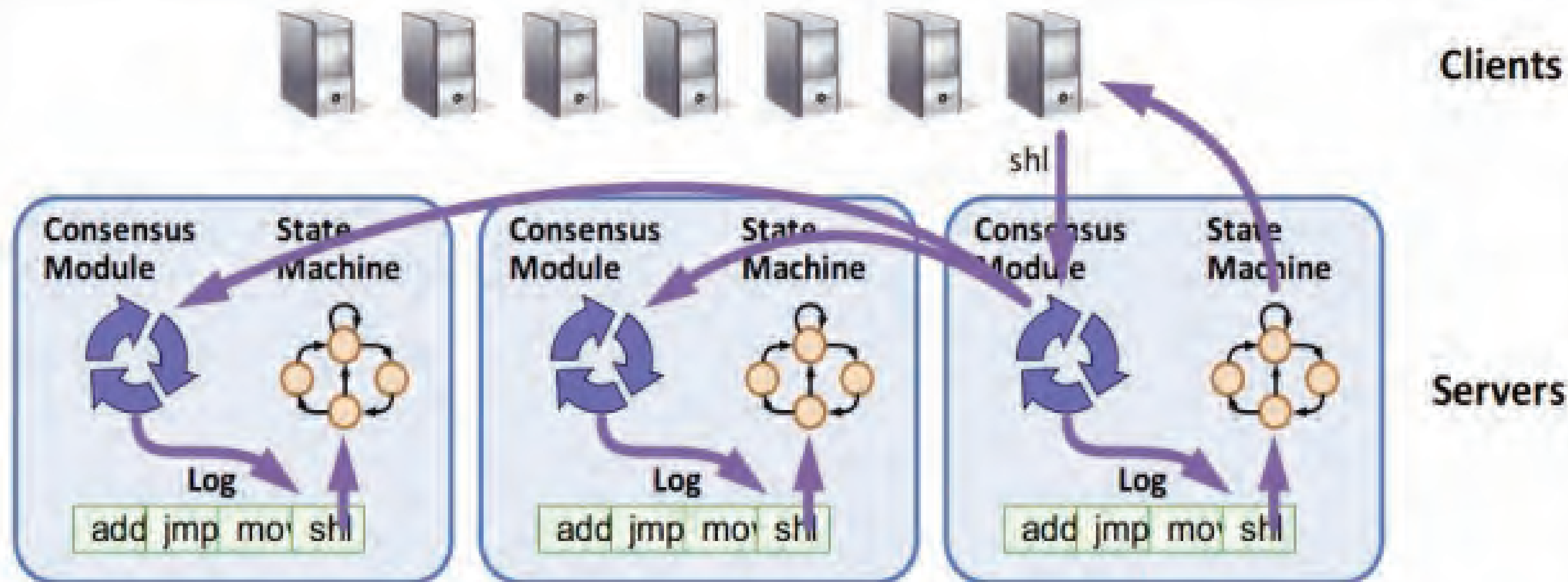
APM

分布式实时消息(事件)总线

与其他技术方案的对比：

统一的发布订阅原语，易扩展实现多种消息设施的集成；可插入式的消息编码解码算法；
支持Pub-Sub、RPC、OneWay，可直接调用Zebra函数实现分布式并行数据处理；
可在客户端与客户端、客户端与服务器以及多个服务器之间实现一致性的即时消息通讯开发

Raft协议组件



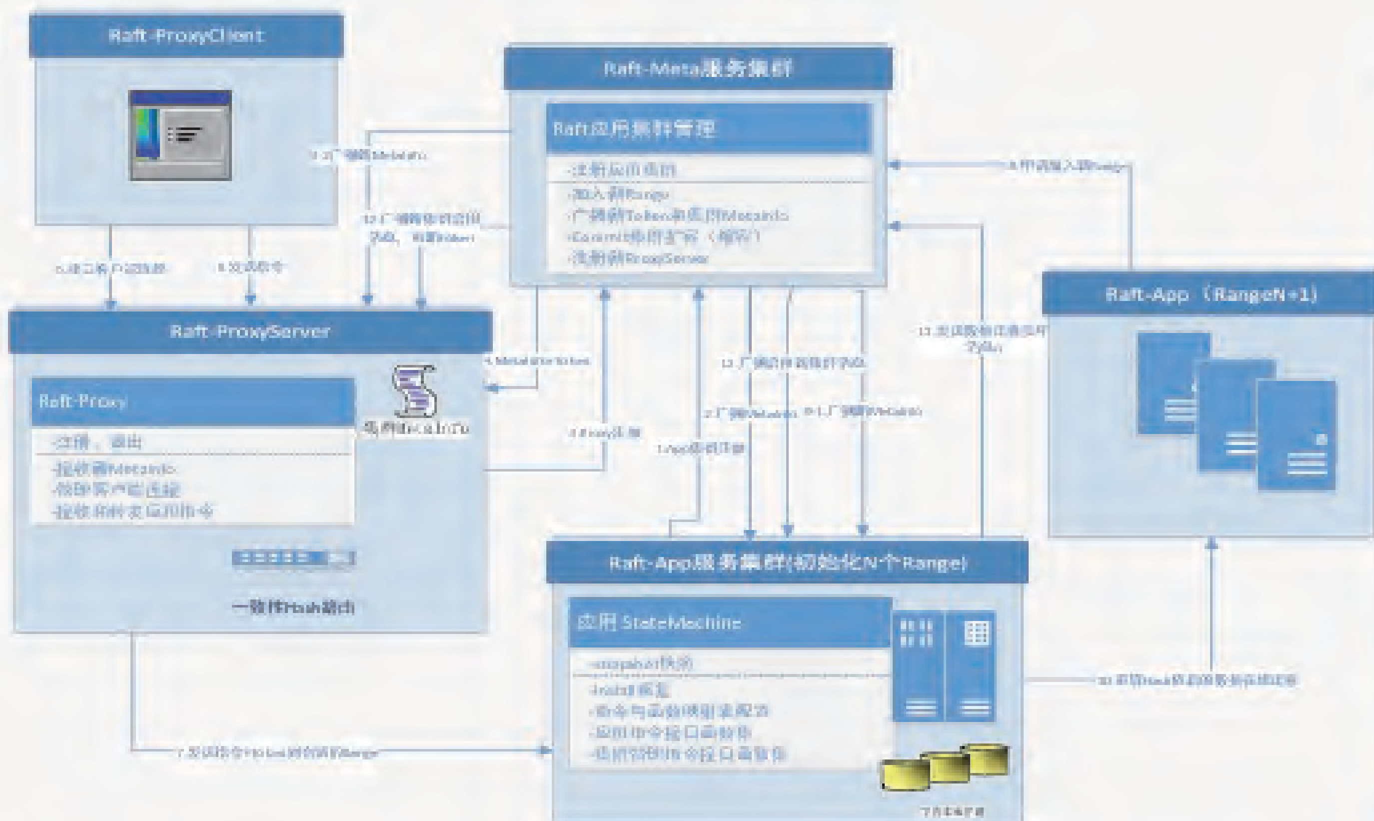
解决问题:分布式一致性状态复制

适用场景：机器故障造成部分交易失败；跨中心网络闪断引起的服务异常。

实现机制：Raft协议和Paxos类似，通过选主算法、命令日志数据存储和同步复制、过半提交等主要机制实现分布式一致性状态复制，作为构建跨中心高容错分布式系统的技术基础。

可扩展的分布式一致性解决方案

APM



实现概要

1.由Raft-meta集群管理应用集群的注册和结构元数据信息

2.客户端通过Raft代理服务上联，减少管理状态广播报文

3.Token值校验确保集群结构变化后分片数据寻址正确

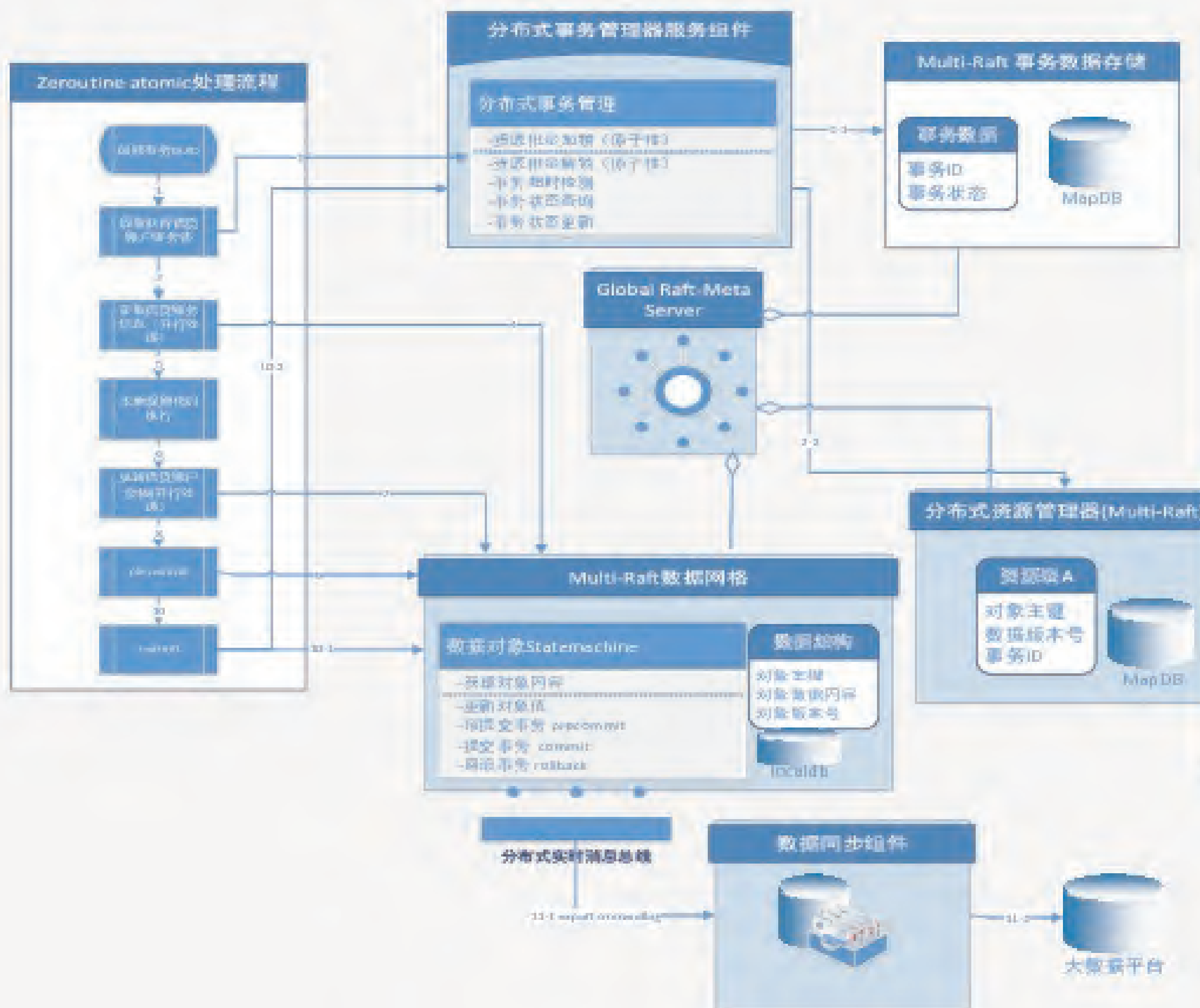
4.与docker容器技术结合做到更细粒度的扩容和缩容

解决问题: Raft集群的弹性扩容

多Range实现集群的线性扩展能力：集群在线扩容和缩容

atomic语义的实现方案

APM



方案优势

1.三阶段协议增强分布式事务数据提交的容错能力

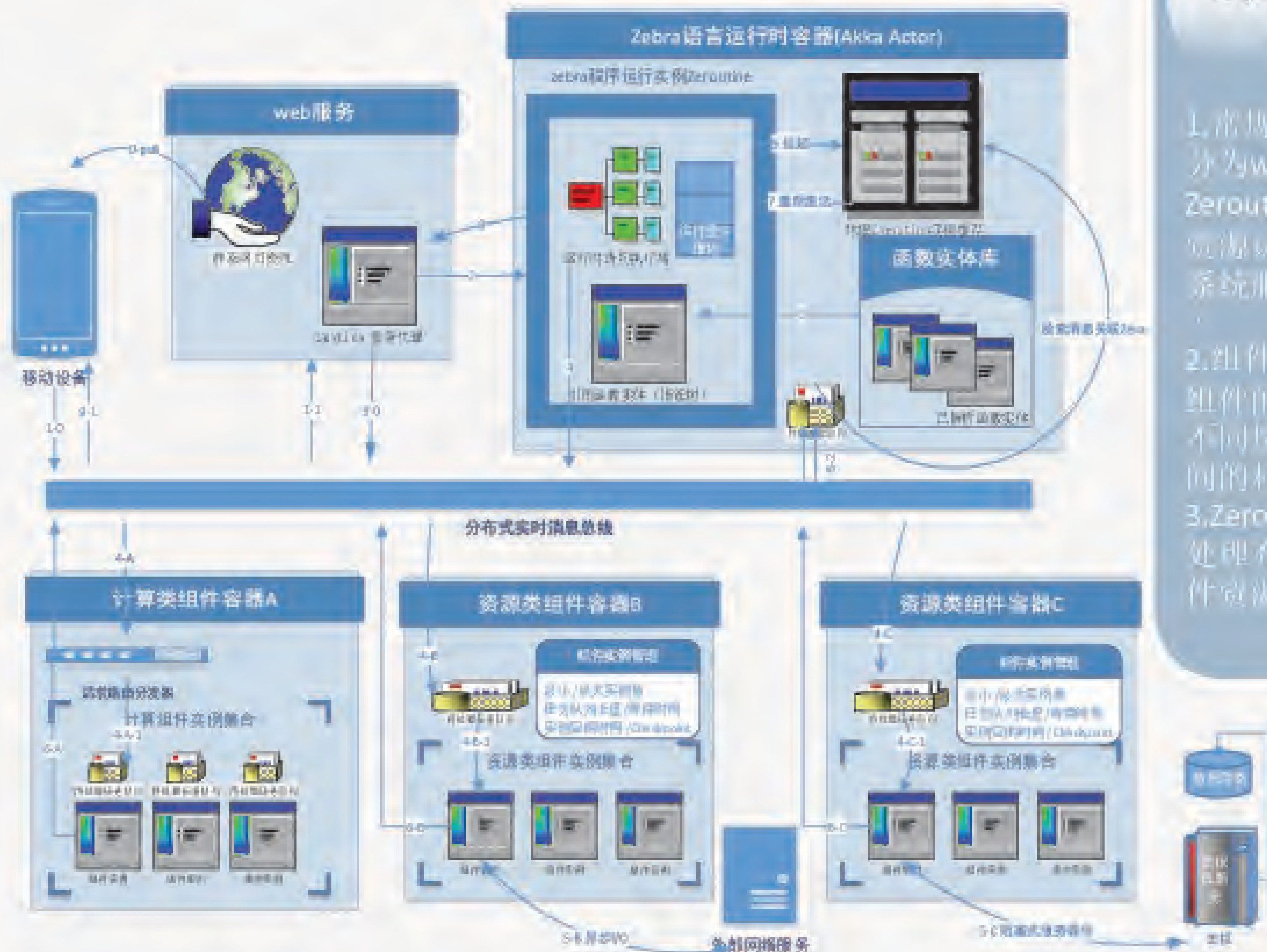
2.分布式资源管理器减少死锁概率，提升事务处理吞吐量

3.数据准实时同步方案可实现交易场景和查询场景的分离，提升分析应用的时效性

4.高度弹性的在线扩容能力

应用分层部署架构

APM



实现弹性部署和高效运行

1.常规应用逻辑上可大致分为web服务层、Zeroutine业务逻辑层、资源访问层（包括外部系统服务和数据库服务）。

2.组件容器技术实现各类组件的分层部署，隔离不同场景组件之间的相互影响

3.Zeroutine异步并行处理充分利用多节点硬件资源，降低响应延迟

计算类组件编写


```
class MyEchoService(conf:WorkConfig)
 extends skyline.service.reactive.akka.ComputeTaskActor(conf){
 def excute(rdata:AnyRef){  /*处理消息*/
       rdata match {
      case pmap:RequestParameters => {  /*处理客户端传来的调用参数列表*/
       val msg = pmap("message")
        this.response(Vector("respmsg"->rlt)) //返回结果集
       }
      case _ =>


 }
/* 实现组件初始化*/
 def init(workConf:java.util.Map[String,AnyRef]):Future[Unit]  {
      ...
       Future{
         /*根据配置需要做的类初始化，主要考虑存在异步IO的操作*/
          ...
       }
  }
}
```

资源管理类组件编写


```
class MySqlService(conf:WorkConfig) extends skyline.service.reactive.akka.AsyncOneTaskResourceActor(conf){
 def excute(rdata:AnyRef){ /*处理消息*/
      rdata match {
    case pmap:RequestParameters => { /*处理客户端传来的调用参数列表*/
     val sql = pmap("sql")
     mydbProxy ! Query(sql)  /*向后台资源服务发请求*/
        }
    case prlt: SqlResultSet =>{
        this.response(Vector("dataSet"->prlt)) //返回结果集
        }
    case _ =>
 }
 /* 实现组件初始化*/
 def init(workConf:java.util.Map[String,AnyRef]):Future[Unit] {
    ...
      Future{
       /*根据配置需要做的类初始化，主要考虑存在异步IO的操作*/
        ...
      }
  }
 /* 组件资源释放*/
 def release() {
      /*执行资源释放操作 */
    ...
  }

}
```

流处理组件编写

APM

```
class StreamCalculator(conf:WorkConfig) extends
skyline.service.reactive.akka.OneTaskStreamActor(conf) {
    def execute(rdata:AnyRef) {
     rdata match {
      case (a:Integer,b:Integer) =>   this.push(a+b)  //得到计算结果数据并向后传递
      case _ => this.acceptNextTask()      //错误的输入,忽略并转入下一条数据接收处理，
     }


    }
}
```

```
##流组件配置
 kafka2akka{
        worker = "skyline.service.reactive.akka.ZebraFunTaskStreamActor"
        type = "local"
                props{
                        listen-points =["#userInfo@skyline.kafka1"]
                        next-points =["#userInfo"]
                        func-name = "replicateMessage"


                }
  }
```

应用性能测试

APM

测试场景

1	访问20kb网页，http请求
2	Zebra编写的数值计算应用，WebSocket请求
3	Zebra编写的数值计算应用，WebSocket请求
4	Zebra应用调用HBase单表查询API，1千万条数据样本，每条记录1.2kb，WebSocket请求
5	Zebra应用调用HBase单表查询，1千万条数据样本，每条记录1.2kb，WebSocket请求

测试结果

场景序号	并发用户数	持续时间(s)	每秒处理请求数	最小响应时间(ms)	最大响应时间(ms)	95%响应时间范围(ms)
1	12000	610	12000	1	233	19
2	3000	2105	73472	0	746	99
3	10000	350	58538	0	4127	100
4	3000	3730	24320	1	7065	190
5	10000	600	17305	0	9905	1400

应用服务器一台，做Docker虚拟化

部件类型	参数	数量
CPU	E5-2650v3/2.3G/10C	2
内存	16G/DDR4/2133/ER	8
网卡	Intel I350/1000M	1
硬盘	1T/SATA/7200转/3.5寸	2

测试结果证明Skyline在高负载场景下运行平稳，相比传统应用服务架构处理能力提升10倍以上。

APMCon

THANK YOU