DOI：10.13955/j.yzyj.2022.02.07.04

中国邮政个性化定制活动系统性能优化探析

刘小华

（中国邮政集团有限公司广东省信息技术中心，广东　广州　510898）

摘　要：针对业务需求增加对中国邮政个性化定制活动系统造成的运维压力，通过对微服务源代码进行优化，系统吞吐量由每秒 100 请求数提升到 1 400 左右请求数，平均响应时间在 500 毫秒以下，优化效果显著。同时，结合监控工具对系统性能进行实时监控，实现以最小的硬件资源投入满足系统性能需求。

关键词：微服务；压力测试；高并发；分布式追踪系统

中图分类号：F61　　**文献标识码：**A

中国邮政个性化定制活动系统主要向邮政营销人员提供线上营销活动配置、发布与分享传播，向用户提供活动参与及活动权益领取等功能，常用的活动模板包括资格抽奖、秒杀、线上表单、合伙助力等。系统主要由客户活动子系统、配置管理子系统、API 网关、活动微服务、后台处理子系统等组成。系统采用前后分离模式搭建，前端使用 SpringMVC 框架搭建，通过 API 网关访问后台微服务实现业务逻辑，后端使用主流的微服务框架 SpringCloud 搭建。各子系统均采用集群方式部署，系统的核心业务逻辑都集中在微服务，上线之初活动规则比较简单，用户访问时间点也比较分散，所以通过增加各子系统的节点数量就能轻松满足业务需求。但随着业务需求的不断增加，尤其是新冠肺炎疫情期间需要通过线上举办秒杀微信红包活动来增加用户黏性，并明确要求系统吞吐量达到每秒 10 000 左右请求数，平均响应时间在 1 000 毫秒以内。按照以往经验，通过系统横向扩容的方式已经很难实现需求，而且系统节点增加后，系统运维能力、数据库资源难以在短时间内同步跟上。本文采用先优化系统源代码与中间参数、再进行横向扩容的方式，提高系统扩容的性价比。

1　系统架构

中国邮政个性化定制活动系统 2.0 版本后台业务逻辑采用 SpringCloud 框架搭建，主要使用了三个组件：ZUUL、Eureka 与微服务，其中 ZUUL 与 Eureka 组件采用直接开箱即用模式，业务逻辑代码都集中在微服务上。前端展现采用传统的 SpringMVC 框架搭建，展现层的业务逻辑通过 API 网关调用后台微服务接口来实现，拿到数据后在前端进行渲染。系统总体结构见图 1。

由于采用了微服务架构，系统的横向扩展能力非常强，可以直接通过增加硬件资源来扩容前端、API 网关及微服务的节点数量，快速实现系统性能提升。但为了提高硬件资源的利用率，先优化各子系统的源代码，然后优化中间件参数与系统参数，让各子系统在单节点的情况下达到最优性能后再进行扩容，这样就能达到事半功倍的效果。

2　微服务子系统优化

在系统优化前，要先通过压力测试工具对系

基金项目：中国邮政集团有限公司广东省分公司科技项目（项目名称：O2O 个性化定制活动平台研究；项目编号：GDXXJ-P210304）。

作者简介：刘小华（1980 ～），男，湖南茶陵人，高级工程师，主要从事软件开发与项目管理研究。

收稿日期：2021—09—16　　**本刊网址：**yzyj.sjzpc.edu.cn

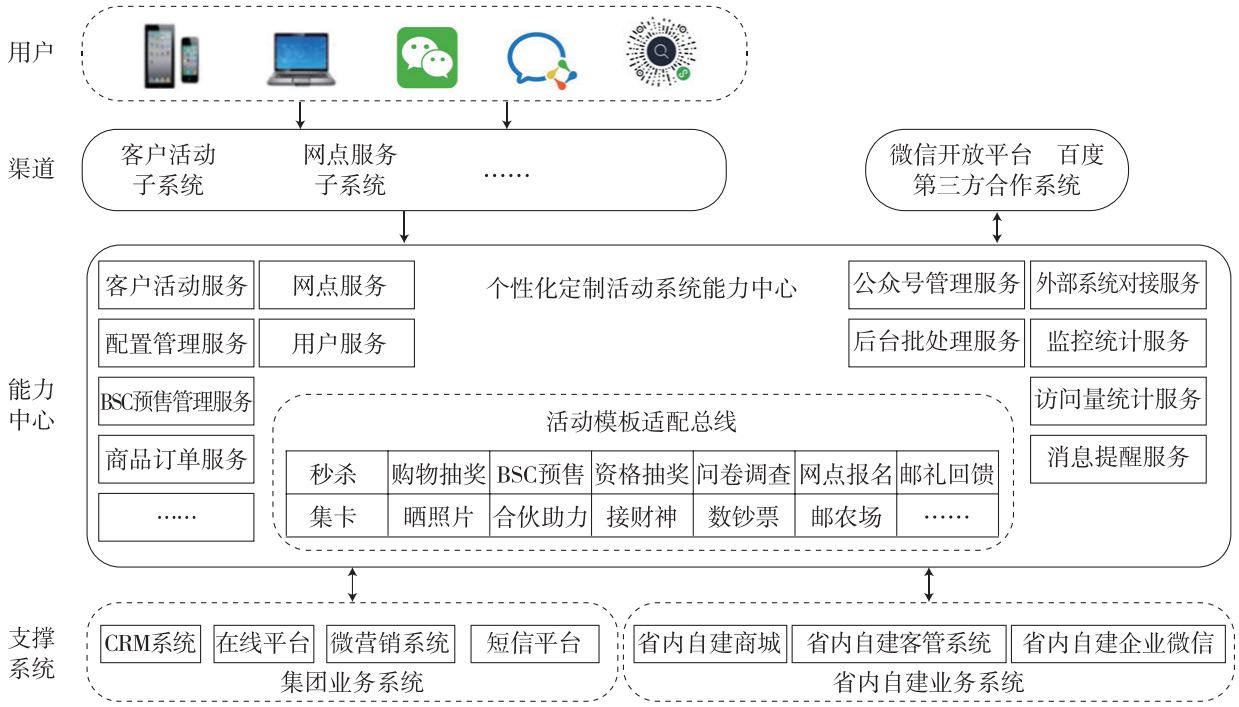

图 1　系统总体结构图

统进行压力测试，找到性能瓶颈后才好对症下药。开始直接使用压力测试工具模拟用户从前端发起请求进行全流程压力测试，试图找出系统的性能瓶颈。在各子系统服务器资源配置均为 4 核 8G 的情况下， 模拟用户发起一次简单的预约购买登记操作。客户活动子系统首先收到用户请求（压力测试时用户的请求直接发送到客户活动子系统，没有经过负载均衡设备），然后根据功能模块编号把请求转发到 API 网关，API 网关根据路由信息再把请求分发到具体微服务，微服务再根据业务逻辑对数据进行落地并向 RocketMQ 登记预约消息以便后续处理，登记完后返回登记结果给前端。一个客户端的请求至少要经过 3 个子系统、一次数据库操作与一次 RocketMQ 操作才能完成。对同一个信息查询功能按照不断增加并发用户数的方案进行压力测试，发现并发数达到 100 后，吞吐量就开始下降，并发数达到 150 后，吞吐量只有每秒 60 以上的请求数，平均响应时间在 3 000 毫秒以上，但前端服务器资源 CPU 利用率接近 90%，API 网关、微服务、数据库等服务器的 CPU 利用率只有 20% 左右，说明在高并发场景下系统访问压力堵在了前端服务器，后端服务器资源并没有发挥作用。

随后，调整压力测试方法，直接对最末端微服务的预约购买登记接口进行压力测试，测试时间为 120 秒，在并发数超过 150 后，系统吞吐量只有每秒 150 以上的请求数，平均响应时间达 1 300 毫秒以上，但 CPU 资源使用率一直居高不下，证明微服务子系统存在性能瓶颈。通过仔细分析此功能的源代码，发现业务逻辑并不复杂，相关数据库操作也都能使用索引，通过分析日志文件发现，各代码段都不存在明显耗时过多的情况。多方查找资料后发现，在压力测试过程中，可实时分析微服务进程中占用 CPU 过高线程的 dump 文件，通过 dump 文件可定位到消耗 CPU 资源过高的线程堆栈信息，发现堵塞都在 Log4j 写日志操作上，因为 Log4j 采用线程阻塞方式把日志内容写到文件中，即使开启 buffer，在高并发场景也没有效果。分析方法如下：

第一步，使用 top 命令找到微服务进程 pid

第二步，对进程 pid 里所有线程按 CPU 使用率从高到底排序，找到排名靠前的 tid

#ps -mp pid -o THREAD，tid，time | sort -k2r

第三步，把排名靠前的 tid 转换为 16 进制

#printf "%x\n" tid

第四步，使用 jstack 显示线程 tid 堆栈信息

#jstack pid |grep tid -A 30

查到的 Log4j 线程堆栈信息大致如下：

java.lang.Thread.State: BLOCKED (on object monitor)

at org.apache.log4j.Category.callAppenders(Category.java:204)

- waiting to lock <0x00000000800371d0> (a org.apache.log4j.spi.RootLogger)

at org.apache.log4j.Category.forcedLog(Category.java:391)

at org.apache.log4j.Category.log(Category.java:856)

at org.slf4j.impl.Log4jLoggerAdapter.info(Log4jLoggerAdapter.java:368)

定位到 Log4j 存在性能问题后把日志级别调整为 WARN，再用 150 个并发对相同功能进行压力测试，压力测试 120 秒时系统吞吐量达每秒 1 600 请求数，平均响应时间在 100 毫秒左右，系统吞吐量提升了 10 倍以上，服务器的 CPU 资源使用率也达 80% 左右。从测试情况看，在高并发系统中采用 Log4j 框架写日志对系统的性能影响非常大，而且并发数越高越明显。

为了不影响生产环境，通过日志文件排查故障，把 Log4j 换成了 Log4j2，再用 150 个并发对相同功能进行压力测试，压力测试 120 秒时发现系统吞吐量达每秒 1 200 请求数，平均响应时间在 120 毫秒左右。证明 Log4j2 在高并发的情况下，INFO 级别与 WARN 级别对系统的吞吐量影响不太明显，所以在高并发系统中不能使用 Log4j 框架记录日志，否则无论如何优化业务逻辑代码，都达不到优化效果。通过同样的方法发现，System.currentTimeMillis()、UUID.randomUUID() 这两个方法也存在性能问题，对系统吞吐量影响较大，并发数越大影响越大，高并发场景下应尽量避免使用。

除了对上述源代码进行优化，还对 Tomcat 与 JVM 参数进行了简单优化。Tomcat 只优化了线程数与连接数，其他都使用默认参数，通过测试发现，在 4 核 8G 配置的服务器资源下最优化配置如下：

max-threads: 200 # 最大线程数

minSpareThreads: 200 # 最小空闲线程数

maxSpareThreads: 200 # 最大空闲线程数

accept-count: 500 # 队列长度

max-connections: 1000 # 最大链接数

connection-timeout: 8000 #http 超时时间，单位毫秒，默认 60 000 毫秒

在高并发场景下，minSpareThreads 参数非常重要，为了减少系统开销，建议 minSpareThreads 与 maxSpareThreads 配置相同参数，因为在系统低峰期，一个线程超过 60 秒没有请求就会自动回收，等下次业务高峰期时再重新创建新线程，所以系统运行一段时间后在日志中会发现线程 ID 号在不断增大。JVM 优化了堆内存及元数据内存大小参数，具体参考如下：

-Xms6g -Xmx6g -Xmn3g

-XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m

后台微服务子系统如果按未优化前的性能进行扩容，达到每秒 10 000 请求数的吞吐量，理论上至少要部署 100 个节点才能勉强满足需求，通过优化后，只要部署 10 个节点就能基本满足需求。由此可见，在高并发场景下，代码质量对系统的性能影响非常大。

3　API 网关优化

API 网关采用 SpringCloud 原生的 ZUUL 组件，只在开源项目的基础上增加两个简单过滤器：路由及鉴权功能，系统启动后相关数据都已初始化到内存中，其他功能都是开箱即用，因此业务逻辑代码优化空间不大。根据微服务的优化经验，先把 ZUUL 的 Log4j 日志级别调整为 WARN，并开启 Tomcat 中间件的 accesslog 功能，便于监控服务路由转发是否正常。然后参考微服务调整 Tomcat 与 JVM 的参数，通过压力测试工具发现，相同的功能从 API 网关端发起压力测试比从微服务端发起压力测试吞吐量下降约 20%. 经查询资料发现，可通过调整参数解决，但除了调整 max-semap-hores 参数可以解决高并发场景下 500 错误外，其他参数效果并不明显，具体内容参考如下：

zuul.semaphore. max-semaphores=3 000

ribbon.MaxTotalHttpConnections=1 000

ribbon.MaxConnectionsPerHost=200

通过深入了解发现，ZUUL 1.X 版本采用多线程阻塞模型实现，每个请求一个线程，虽然编程

模型简单，但对性能有一定影响，只能通过部署多个节点来提高系统性能。也可以把 Tomcat 换成 Undertown，提供基于 NIO 的阻塞和非阻塞 API，据说在高并发系统中吞吐量比 Tomcat 要高一些，但没有实际验证过。

4　客户活动子系统优化

客户活动子系统采用 SSH 框架搭建，部署在 Tomcat 容器中。系统功能相对比较简单，业务逻辑通过 API 网关调用微服务来实现。能够快速优化的地方不多，参照微服务优化经验把 Log4j 换成 Log4j2，把调用 API 网关的 HttpClient 组件开启连接池模式。把 Tomcat 的 context.xml 与 server.xml 配置文件中的 Access Log Valve 功能注释掉，并且修改了 Tomcat 的连接器参数，具体内容参考如下：

<Connector port="8080"protocol="org.apache.coyote.http11.Http11AprProtocol"maxHttpHeaderSize="8192"maxThreads="200"minSpareThreads="200"maxSpareThreads="200"enableLookups="false"redirectPort="8443"acceptCount="1500"maxConnections="1500"connectionTimeout="10000"disableUploadTimeout="true"userSendfile="false" />

客户活动子系统除了对代码与中间件参数优化外，还购买了 CDN 服务，对系统中的图片、视频等多媒体资源进行加速处理，减轻机房带宽压力，提升用户体验。

5　微服务性能监控

系统优化后，还要不断监控优化成果是否持续发挥作用。随着业务需求不断叠加，很难保证新增功能不会存在性能问题，也不可能迭代一次需求就做一次压力测试，还有大量需求在排队等着迭代开发。为了不影响业务需求开发进度，又能提前发现性能问题，采用监控工具对系统性能进行实时监控，发现性能问题及时解决。通过压力测试发现，对系统性能影响最大的是微服务子系统，所以重点监控微服务的性能指标，就能掌控系统整体性能。市面上监控微服务性能的开源工具很多，如 Zipkin、Pinpoint、SkyWalking、CAT 等，活动系统采用分布式追踪系统 SkyWalking，因为是基于字节码注入的调用链分析，支持多种插件，UI 功能也较强，关键是对现有项目无代码侵入，不但可以监控每个微服务接口的耗时情况，而且还能监控微服务接口中操作 Redis、数据库、消息队列等耗时情况，方便实时掌控微服务性能。

结语

中国邮政个性化定制活动系统按上述步骤对各子系统调优完成后，单个节点的吞吐量由原来的每秒 70 左右请求数提升到 1 000 左右请求数，平均响应时间在 1 000 毫秒以下。根据优化后的压力测试数据，在生产环境对客户活动子系统、API 网关与微服务子系统分别扩容到 10 个节点，理论上能够满足业务性能需求。优化版本上线后，通过第三方访问量统计工具监测到，同时在线人数达 5 400 人左右时，系统能够平滑支撑，微服务接口响应时间都在 1 000 毫秒以内，服务器 CPU 资源利用率在 20% 左右，初步判断已达到优化目标。此次优化经验还在集团微营销系统与省内微商城得到了进一步验证，特别是 Log4j 框架与 Tomcat 的 minSpareThreads 参数，固定最小空闲线程数量后，在业务高峰期有效缓解了 CPU 的消耗情况。

目前的优化工作主要集中在应用层，但随着访问量不断增加，数据库中的数据量不断增加，数据库性能优化将是一个新的挑战。分布式缓存技术 Redis 可以缓解一些数据查询操作的访问压力，但解决不了数据库写操作的性能问题。互联网应用系统优化是一个持续的过程，任重道远，只有持续跟进业界主流技术，才能在出现性能瓶颈时，快速满足业务发展需要。

参　考　文　献

［1］许进，叶志远，钟尊发，等．重新定义 Spring Cloud 实战［M］．北京：机械工业出版社，2018

［2］CHARLIE HUNT，BINU JOHN，著．柳飞，陆明刚，译．Java 性能优化权威指南［M］．北京：人民邮电出版社，2014

［3］吴晟，高洪涛，赵禹光，等．Apache SkyWalking 实战［M］．北京：机械工业出版社，2020

［4］周志明．深入理解 Java 虚拟机：JVM 高级特性与最佳实践（第 3 版）［M］．北京：机械工业出版社，2019