# 论微服务架构在图书馆数据中台的实践

## 一、摘要

2019年11月，本人所在信息技术部承接了馆内数据中台的建设，该项目为期半年。通过数据中台，实现了我馆数据的分层与水平解耦，沉淀了公共基础数据。通过数据建模实现跨域数据整合和知识沉淀，通过数据服务实现对于数据的封装和开放，快速、灵活满足上层应用的要求，通过数据开发工具满足个性化数据和应用的需要，为全市文旅数据融合工作提供了有力的数据支撑。该项目时间紧任务重，涉及人员组织多， 涉及馆内 10 余个部门，外部配合协作 10 多个厂商团队。该项目于 2020 年 5 月正式上线， 2020 年 6 月通过最终验收。本人在该项目中担任系统架构师，主要完成技术方案评估与实现，项目立项论证等工作。本文结合笔者实际经验，以该项目为例，讨论微服务架构及其应用，包括微服务的基本概念及特点，如何从单体架构迁移到微服务架构，以及在实现微服务架构的过程中遇到的问题及其解决方案等。

## 二、正文

近年来，随着我馆服务质量的不断提升，大量软硬件系统在我馆得到部署。硬件设备包括自助借还机、交互机器人、电子书借阅机、人脸识别门禁、人流量计数器，软件系统包括图书管理系统、微信小程序、移动端APP。软硬件系统的增加带来了大量用户数据的流入，如读者借阅数据、读者信息、阅读偏好、读者常见问题、入馆人次、入馆时间等海量的数据，但是这些数据缺乏统一的接入和分析。因此，需要通过数据技术，对海量数据进行采集、计算、存储、加工，同时统一标准和口径，并在此基础之上，对数据进行归纳分析，对外提供统一的馆情数据服务。

笔者带领团队于2019年11月承接馆内数据中台的建设，并担任系统架构设计师。主要负责项目整体技术方案评估实现、立项论证以及项目管理工作。该项目的架构工作于同年12月完成，整个项目耗时6个月，于2020年5月上线测试。于2020年6月完成验收。

笔者带领部门的同事经过一段时间的架构考察和评估，最终决定采用微服务架构建设新系统。采用微服务架构的目的是充分拆分庞大臃肿的系统，以促进软件的敏捷开发和部署。在此项目中，每个小组负责一个组件的完整生命周期，从开发到测试，再到运维以及后续迭代升级。最后，各个服务组件通过REST接口进行组装。

相比微服务，传统的单体软件架构在构建部署和扩展伸缩方面有很大的局限性，传统的单体架构一般分为表现层、业务逻辑层、数据层三个部分。系统中任何程序的变更都需要整个应用重新构建和部署新版本。另外传统的单体架构在进行水平扩展时也只能整个系统扩展，而不能针对某一个功能模块进行水平扩展。而微服务架构恰好可以完美地解决传统单体架构所遇到的种种问题。微服务架构将系统以组件化的方式分解为多个服务，服务之间相对独立，单个服务功能的改变只需要重新构建部署相应的服务即可。

但是从实践角度来说，微服务的可靠性也问题相比单体系统更加难以处理。除了服务自身的可靠性因素之外，还受到其它因素的影响，比如网络和其它相关联的微服务运行质量等。因此微服务的可靠性设计，需要考虑集群容错、故障隔离和流量控制。

集群容错包含路由容错、服务降级和熔断。利用路由容错机制，可以在底层实现微服务的自动容错处理，提升系统的可靠性。常用的容错策略包括：失败自动切换机制。微服务调用失败自动切换策略指的是当发生服务调用异常时，重新选路，查找下一个可用的微服务提供者。失败回调机制。微服务调用失败之后，提供异常回调接口，执行微服务消费者自定义的失败处理逻辑。快速失败机制：在业务高峰期，对于一些非核心的服务，希望只调用一次，失败也不再重试，为重要的核心服务节约宝贵的运行资源。此时，快速失败是个不错的选择。快速失败策略的设计比较简单，获取到服务调用异常之后，直接忽略异常，记录异常日志。

但是直接在代码中引入路由容错，往往需要引入健康检测机制，这无疑会提高代码设计的复杂性。在实践中，笔者采用借助业内著名的中间件Consult 对微服务进行治理，通过和 Nginx 相互配合可以完美实现服务的容错。Consult 提供健康检测功能，只要在服务注册的同时，配置相关的检测条件如 HTTP 请求为200,并指定在故障出现多久后自动将服务从注册中心注销。对于那些需要暴露在外网的服务，笔者通过 Consult-Template 实例自动监测 Consult 集群的变化，若发现服务实例被注销，则按照事先定义好的Nginx 配置模板重新生成nginx.conf并reload本节点的Nginx，使得失效的服务从Nginx中删除，避免开放接口出现故障。对于处于系统内部的微服务，直接通过 Consult 提供的服务域名进行互相访问，对删除的服务实例不存在感知。故代码中也无需添加路由容错逻辑，降低了编码的复杂度。

在服务降级方面。在业务高峰时，为了保证核心服务的SLA，往往需要停掉一些不太重要的业务。在数据中台的数据开发层上，晚间需要处理大量报表计算。为了进一步释放资源，在夜间21点到凌晨6点，暂停所有的数据接入服务，仅开放部分数据开放接口，其余资源全部用于报表生成。通过这种形式的服务降级，确保复杂的业务报表的正常生成。

在熔断方面。当一个微服务使用另一个微服务提供的功能时，需要通过 RPC、HTTP等方式与其通信并通常需要等待其返回响应。但请求可能因为网络故障导致丢失、或是被请求的服务异常而无法响应。为避免因为这些异常导致请求方的资源耗尽，需要引入断路器。断路器向下游系统发送心跳，或是监控对每一个下游系统的请求，一旦察觉这些下游服务出现故障，后续的请求将不会发向此实例，直到下游系统恢复正常。笔者使用开源的 Hytrix 框架进行熔断配置，通过设置熔断参数，当请求出错达到一定阈值时，就会变成开启状态，所有请求短路，一段时间以后，断路器变成半开状态，如果下一个请求成功了，就关闭断路器，反之继续开启断路器。Hytrix 将每个微服务的请求放在线程池里，进行资源隔离，即使一个依赖服务出现问题，也不会影响应用程序其他部分。笔者在项目中，对于关键服务都使用熔断机制进行保护，以免出现大规模超时等待的情况。

在故障隔离和流量控制方面，本系统也做了一些探索。微服务鼓励软件开发者将整个软件解耦为各个功能单一的服务，并且这些服务能够独立部署、升级和扩容。利用Docker容器部署微服务，可以实现高密度的微服务部署，实现服务之间的资源层隔离。在流量控制方面，当资源成为瓶颈时，服务框架需要对消费者做限流，启动流控保护机制。流量控制有多种策略，比较常用的有针对访问速率的静态流控和针对资源占用的动态流控等。为了防止开放接口被频繁外部频繁调用，本系统采用了针对访问速率的静态流控。静态流控主要针对客户端访问速率进行控制，它通常根据服务质量等级协定SLA中约定的QPS做全局流量控制，一旦设置静态流控阈值为某个QPS，则无论集群有服务实例，它们总的处理速率之和就不能超过这个数值。

微服务架构不是银弹，除了可靠性，它同样也还有一些其他可大可小的问题。比如定位故障困难，以往的单体结构只要日志完备就能快速定位问题。而在微服务中，需要引入链路跟踪才能方便定位问题。其次，一旦微服务数量扩大，服务数量非常多，部署、管理的工作量很大，这对团队来说也会是不小的挑战。因此在后续过程中，需要引入更多自动化部署、服务监控等工具来节约部署和监控的成本。所以，在拥抱微服务之前，也需要认清它所带来的挑战。需要避免为了“微服务”而“微服务”。对类似图书馆这样的传统行业而言，开始时可以考虑引入部分合适的微服务架构原则，对已有系统进行改造或新建微服务应用，逐步探索及积累微服务架构经验，而非全盘实施微服务架构。