# 论系统可靠性在图书馆馆情数据中台的实践

## 一、摘要

2019年11月，本人所在信息技术部承接了馆情数据中台的建设，该项目为期半年。该项目于 2020 年 5 月正式上线， 2020 年 6 月通过最终验收。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。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，讨论系统可靠性的基本概念，常见的可靠性保障方案，以及在系统可靠性实践中遇到的问题及解决方案。

## 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，如读者画像、读者偏好、读者借阅行为、读者行动轨迹等海量的数据，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。

为此，笔者带领团队于2019年11月承接馆情数据中台的建设，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。

在架构设计之初，笔者就考虑到了可靠性这一质量属性对系统质量的影响。软件可靠性是指系统从故障中恢复的能力，可靠性越高的系统，运行更加稳定。然而在软件的世界中，故障无处不在。硬盘可能损坏，网络可能中断，消息可能丢失，内存可能跳变。在基于微服务的分布式软件系统中，各种各样的故障更是会层出不穷。很多时候我们期望能够完美预防所有故障的出现，但事实上这几乎是一个不可能实现的梦想。因此很多企业开始转变思路，从避免故障出现转向使故障出现时对系统不产生影响。现在很多IT公司都参考对系统进行故障注入测试，比如虚拟机复位，压力测试等。这些故障注入甚至会在生产环境中进行，因此所有开发人员很清楚他们开发出的应用必须能够应对这些故障，而不是祈祷一切正常。为了应对可能的故障，确保系统的可靠性，常见的方法有恢复块设计、N 版本程序设计、冗余设计。

恢复块设计是指选择一组软件操作作为容错设计单元，把普通的程序块变成恢复块。一个恢复块包含有若干个功能相同、设计差异的程序块。其中，一个运行块，多个备份块，构成“动态冗余”，一旦运行块出现故障，检错机制检测到之后，就使用备用模块进行替换。N 版本程序设计是一种静态故障屏蔽技术。N 版本程序的核心是通过设计出多个模块或不同版本，对于相同初始条件和相同输入的操作结果，实现多数表决，防止其中某一软件模块/版本的故障提供错误的服务，以实现软件容错。这种技术实现成本非常高，往往需要调用不同的人员采用不同的技术实现相同的功能，一般用在对可靠性要求较高的军事、航天领域。冗余设计是指在一套完整的软件系统之外，启用相同的模块或系统作为备份，在出现故障时可以使用冗余的部分进行替换，从而维持软件系统的正常运行。

针对数据中台的应用场景，为尽可能达到软件复用与业务水平扩展的目的，笔者采用层次化的架构风格来进行系统架构。与一般的业务系统不同，数据中台本质上是面向数据的，笔者根据自己的实践经验，从数据处理的维度上对数据中台进行划分，将其分成数据接入层、数据模型层、数据服务层，数据开发层，各个层次只与相邻的两层发生交互。在不同的层次上，用户对数据利用的维度也从基础维度到跨域维度甚至是自定义维度，层层递进。其中在数据服务层笔者采用微服务架构进行设计，在实践中笔者发现，基于微服务架构设计的数据中台的可靠性问题，需要从集群容错、故障隔离和流量控制三方面入手。

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。

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Consul 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。Consul 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 Consul-Template 实例自动监测 Consul 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 Consul 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。

实践证明，关注可靠性以后，架构设计中引入了许多中间件，带来了大量的运维和部署问题，但是对于数据中台这样的核心系统来说，在可靠性方面的投入是值得的，通过微服务的集群容错、故障容错和流量控制,增强了系统的可靠性，系统上线半年以来运行稳定，没有出现业务中断现象。但是目前在可靠性监控方面没有一套自动化的监控工具，因此笔者团队很难及时发现系统中存在的问题，有时候可能系统出现重大故障才去进行定位排错。因此，在项目二期决定建立一整套的故障监控告警平台，从主机、数据库、服务等三个方面入手，打造一个自动发现及时告警的监控平台，持续提升数据中台的运行质量。