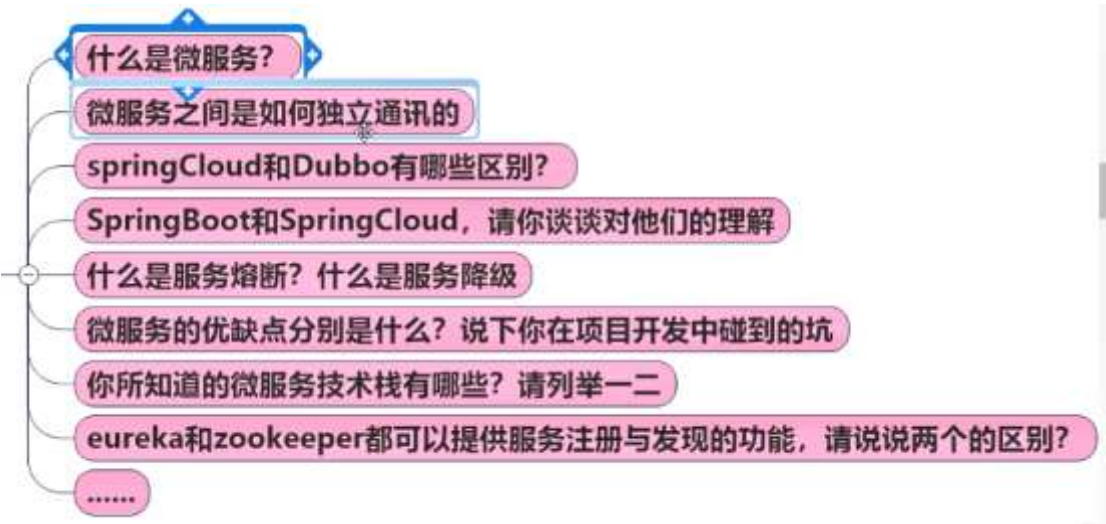


微服务概述与 springcloud



微服务之间是如何独立通讯的

总的微服务来说

微服务通信方式:

1.

1 | 同步: RPC, REST等

2.

1 | 异步: 消息队列。要考虑消息可靠传输、高性能, 以及编程模型的变化等。

消息队列中间件如何选型

1

1. 协议: AMQP, STOMP, MQTT, 私有协议等。

2

2. 消息是否需要持久化。

3

3. 吞吐量。

4

4. 高可用支持, 是否单点。

5

5. 分布式扩展能力。

6

6. 消息堆积能力和重放能力。

7

7. 开发便捷, 易于维护。

8

8. 社区成熟度。

针对 springcloud 来说

RestTemplate的三种使用方式

SpringCloud中服务之间的两种调用RESTful接口通信的方式:

1. RestTemplate

2. Feign

RestTemplate是一个Http客户端, 类似于HttpClient, org但比HttpClient更简单。我们通过RestTemplate来简单演示一下服务之间的调用, 我们使用两个服务来做演示。一个商品服务, 一个订单服务。首先创建一个商品服务工程:

你在微服务开发中碰到的坑

微服务

强调的是服务的大小, 它关注的是某一个点, 是具体解决某一个问题/提供落地对应服务的一个服务应用,

狭意的看, 可以看作 Eclipse 里面的一个个微服务工程/或者 Module

微服务化的核心就是将传统的一站式应用, 根据业务拆分成一个一个的服务, 彻底地去耦合, 每一个微服务提供单个业务功能的服务, 一个服务做一件事

微服务架构

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相协作（通常是基于 HTTP 协议的 RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

springcloud 的 5 大神兽

- 服务发现—NetFlix Eureka
- 客户端负载均衡—Netflix Ribbon
- 断路器—Netflix hystrix
- 服务网关—Netflix zuul
- 分布式配置—Spring cloud config

微服务的优缺点

微服务优缺点
<div><div>• 优点</div><div>每个服务足够内聚，足够小，代码容易理解这样能聚焦一个指定的业务功能或业务需求 开发简单、开发效率提高，一个服务可能就是专一的只干一件事。 微服务能够被小团队单独开发，这个小团队是2到5人的开发人员组成。 微服务是松耦合的，是有功能意义的服务，无论是在开发阶段或部署阶段都是独立的。 微服务能使用不同的语言开发。 易于和第三方集成，微服务允许容易且灵活的方式集成自动部署，通过持续集成工具，如Jenkins, Hudson, bamboo 。 微服务易于被一个开发人员理解，修改和维护，这样小团队能够更关注自己的工作成果。无需通过合作才能体现价值。 微服务允许你利用融合最新技术。 微服务只是业务逻辑的代码，不会和HTML,CSS 或其他界面组件混合。 每个微服务都有自己的存储能力，可以有自己的数据库。也可以有统一数据库。</div></div>
<div><div>• 缺点</div><div>开发人员要处理分布式系统的复杂性 多服务运维难度，随着服务的增加，运维的压力也在增大 系统部署依赖 服务间通信成本 数据一致性 系统集成测试 性能监控.....</div></div>

微服务的技术栈

微服务项目	相当于接口	落地技术	实现类
服务开发		Springboot, Spring, SpringMVC	
服务配置与管理		Netflix公司的Archaius, 阿里的Diamond等	
服务注册与发现		Eureka, Consul, Zookeeper等	
服务调用		Rest, RPC, gRPC	
服务熔断器		Hystrix, Envoy等	
负载均衡		Ribbon, Nginx等	
服务接口调用(客户端调用服务的简化工具)		Feign等	
消息队列		Kafka, RabbitMQ, ActiveMQ等	
服务配置中心管理		SpringCloudConfig, Chef等	
服务路由 (API网关)		Zuul等	
服务监控		Zabbix, Nagios, Metrics, Spectator等	
全链路追踪		Zipkin, Brave, Dapper等	
服务部署		Docker, OpenStack, Kubernetes等	
数据流操作开发包		SpringCloud Stream (封装与Redis, Rabbit, Kafka等发送接收消息)	
事件消息总线		Spring Cloud Bus	

springboot 和 springcloud 是什么关系

springboot 是微观的，（相当于医院的一个一个科室）
springcloud 是宏观的 （相当于对外的一个医院）
springcloud 依赖于 springboot.

SpringBoot 专注于快速、方便的开发单个微服务个体， SpringCloud 关注全局的服务治理框架。

dubbo 和 springcloud 的区别

对比结果

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务监控	Dubbo-monitor	Spring Boot Admin
断路器	不完善	Spring Cloud Netflix Hystrix
服务网关	无	Spring Cloud Netflix Zuul
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

最大区别：SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式。

严格来说，这两种方式各有优劣。虽然从一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题。而自REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更加合适。

品牌机与组装机区别

很明显，Spring Cloud的功能比DUBBO更加强大，涵盖面更广，而且作为Spring的拳头项目，它也能够与Spring Framework、Spring Boot、Spring Data、Spring Batch等其他Spring项目完美融合，这些对于微服务而言是至关重要的。使用Dubbo构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果你是一名高手，那这些都不是问题；而Spring Cloud就像品牌机，在Spring Source的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要在非原装组件外的东西，就需要对其基础有足够的了解。

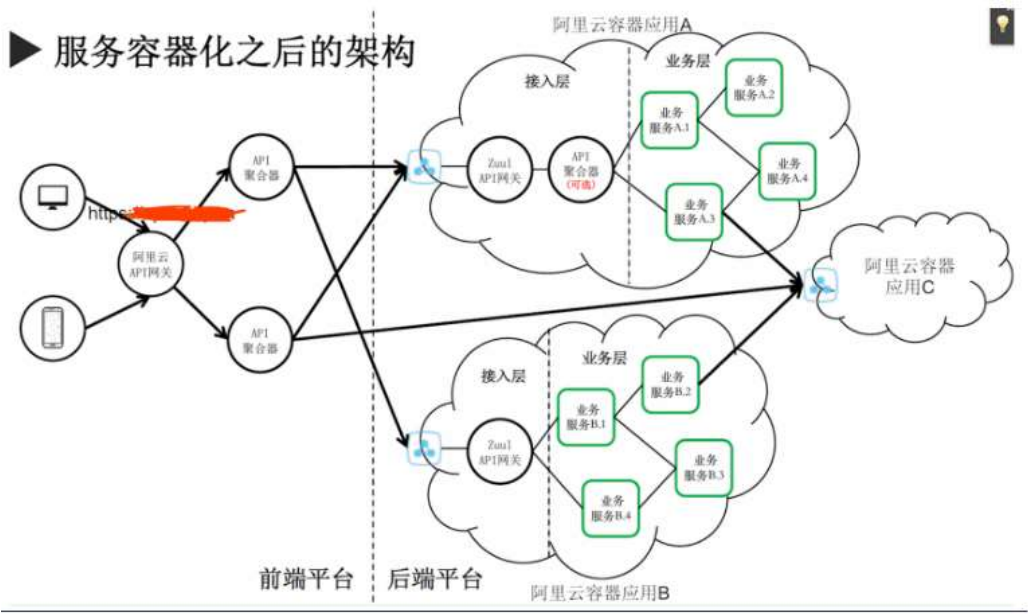
社区支持与更新力度

最为重要的是，DUBBO停止了5年左右的更新，虽然2017.7重启了。对于技术发展的新需求，需要由开发者自行拓展升级（比如当当网弄出了DubboX），这对于很多想要采用微服务架构的中小软件组织，显然是不太合适的，中小公司没有这么强大的技术能力去修改Dubbo源码+周边的一整套解决方案，并不是每一个公司都有阿里的大牛+真实的线上生产环境测试过。

总结

Dubbo 的定位始终是一款 RPC 框架，而 spring cloud 的目标是微服务架构下的一站式解决方案。再面临微服务基础框架选型时 Dubbot 与 Springcloud 只能二选一

springcloud 在阿里云的应用



Rest 微服务构建工程

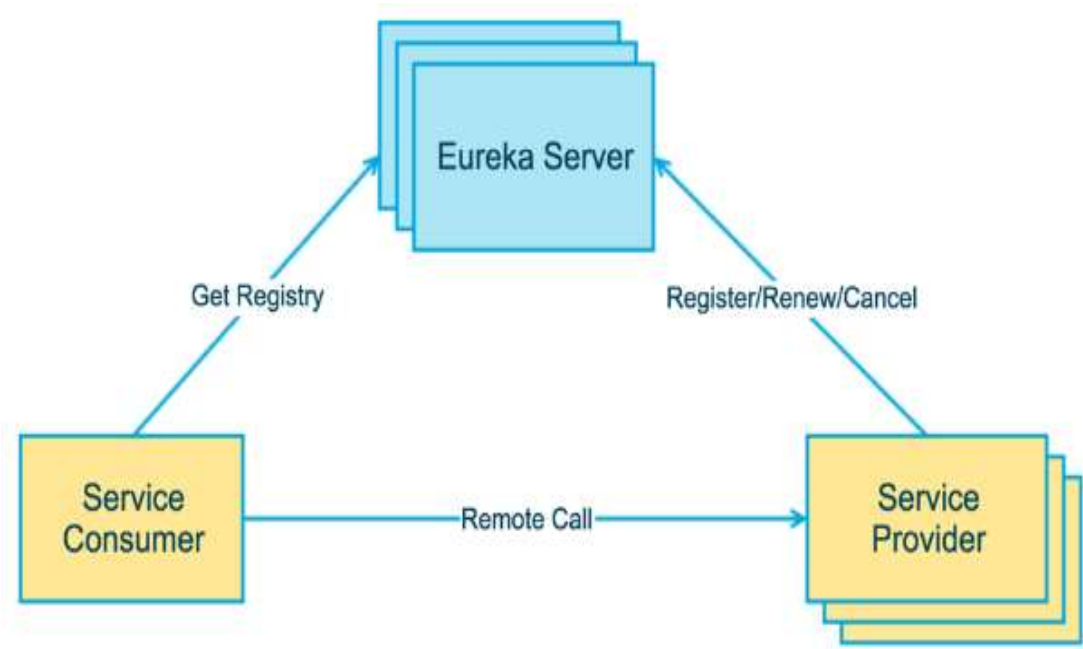
消费端的 RestTemplate

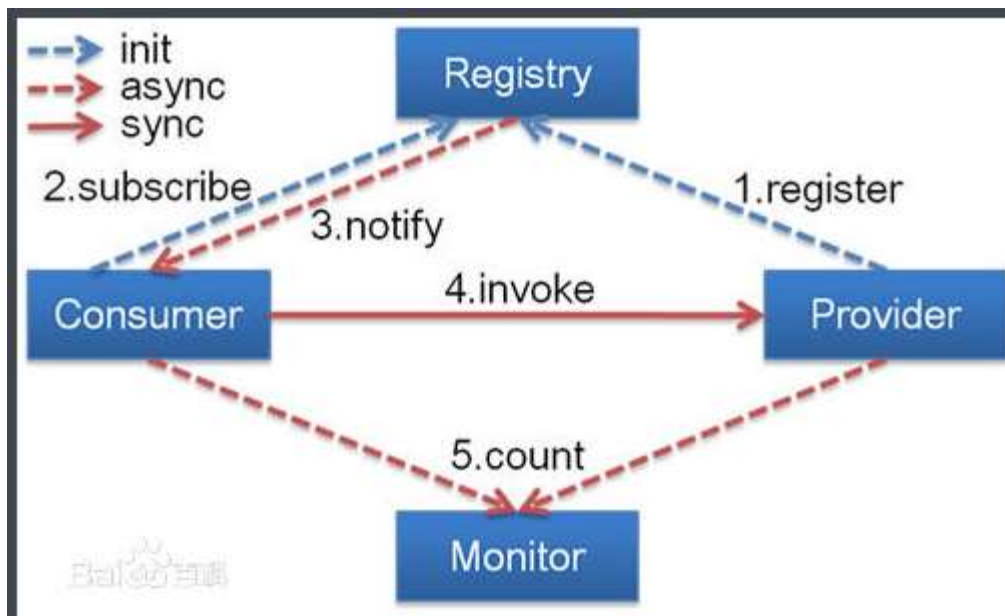
RestTemplate 提供了多种便捷访问远程 Http 服务的方法，
是一种简单便捷的访问 restful 服务模板类，是 Spring 提供的用于访问 Rest 服务的客户端模板工具集

Eureka 是什么

Eureka 采用了 C-S 的设计架构。Eureka Server（物业公司） 作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接（交物业费相当于维持心跳连接）。这样系统的维护人员就可以通过 Eureka Server 来监控系统各个微服务是否正常运行。SpringCloud 的一些其他模块（比如 Zuul）就可以通过 Eureka Server 来发现系统中的其他微服务，并执行相关的逻辑。
Eureka 包含两个组件：Eureka Server 和 Eureka Client
Eureka Server 提供服务注册服务
各个节点启动后，会在 EurekaServer 中进行注册，这样 EurekaServer 中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到
EurekaClient 是一个 Java 客户端，用于简化 Eureka Server 的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。
在应用启动后，将会向 Eureka Server 发送心跳(默认周期为 30 秒)。如果 Eureka Server 在多个心跳周期内没有接收到某个节点的心跳，EurekaServer 将会从服务注册表中把这个服务节点移除（默认 90 秒）

Eureka 与 dubbo 的区别





eureka 的自我保护

一句话：某时刻某一个微服务不可用了，eureka 不会立刻清理，依旧会对该微服务的信息进行保存

eureka 的服务发现

向外暴露你提供了那些微服务

对于注册进 Eureka 里面的服务，可以通过服务发现来获得该服务的信息

eureka 与 zookeeper 的区别

Eureka:AP

zookeeper:CP

2. Zookeeper保证CP 保证了一致性，牺牲了可用性

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

3. Eureka保证AP 不保证强一致性，但保证了高可用

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
 2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
 3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中
- 因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。

Ribbon 负载均衡

Spring Cloud Ribbon 是基于 Netflix Ribbon 实现的一套 **客户端负载均衡**的工具(比如说去麦当劳买汉堡，有 3 个窗口，1 号窗口 8 个人，2 号窗口 6 个人，3 号窗口 5 个人，我肯定去 3 号窗口)。

简单的说，Ribbon 是 Netflix 发布的开源项目，主要功能是提供**客户端的软件负载均衡算法**，将 Netflix 的中间层服务连接在一起。Ribbon 客户端组件提供一系列完善的配置项如连接超

时，重试等。简单的说，就是在配置文件中列出 Load Balancer（简称 LB）后面所有的机器，Ribbon 会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们也很容易使用 Ribbon 实现自定义的负载均衡算法。

nginx,ribbon,feign 的 区别

服务器端负载均衡 Nginx Nginx 基于C语言，快速，性能高5w/s。 Redis 5w/s，RibbatMQ 1.2w/s ApacheActiveMQ 0.6w/s 业务系统，kafka 20w~50w/s大数据，Zuul2.0 200w/s 负载均衡，反向代理，代理后端服务器。隐藏真实地址，防火墙。不能外网直接访问，安全性较高。属于服务器端负载均衡。既请求由 nginx 服务器端进行转发。
客户端负载均衡 Ribbon Ribbon 是从 eureka 注册中心服务器端上获取服务注册信息列表，缓存到本地，然后在本地实现轮询负载均衡策略。 既在客户端实现负载均衡。 应用场景的区别： Nginx 适合于服务器端实现负载均衡 比如 Tomcat，Ribbon 适合与在微服务中 RPC 远程调用实现本地服务负载均衡，比如 Dubbo、SpringCloud 中都是采用本地负载均衡。
Feign Feign 是一个声明web服务客户端，这使得编写web服务客户端更容易Spring Cloud Netflix 的微服务都是以 HTTP 接口的形式暴露的，所以可以用 Apache 的 HttpClient 或 Spring 的 RestTemplate 去调用，而 Feign 是一个使用起来更加方便的 HTTP 客户端，使用起来就像是调用自身工程的方法，而感觉不到是调用远程方法 Feign包含了ribbon。

有时候有的项目会 2 个技术一起用在该项目中是因为 feign 是远程调用的,ribbon 是做负载均衡的,

负载均衡算法

轮询，随机，优化的轮询，带权重的轮询，超时重试的轮询，并发最小的访问，判断选择区域内最好的。

Feign 的原理

<p>Feign 是一个声明式的 Web 服务客户端，使得编写 Web 服务客户端变得非常容易，只需要创建一个接口，然后在上面添加注解即可。</p> <pre>@Mapper public interface DeptDao { // ... }</pre> <p>接口+注解 feign的原理和这个一样</p>	<p>Feign能干什么</p> <p>Feign旨在使编写Java Http客户端变得更容易。</p> <p>前面在使用Ribbon+RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模版化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用。所以，Feign在此基础上做了进一步封装，由他来帮助我们定义和实现依赖服务接口的定义。在Feign的实现下，我们只需创建一个接口并使用注解的方式来配置它(以前是Dao接口上面标注Mapper注解,现在是一个微服务接口上面标注一个Feign注解即可)，即可完成对服务提供方的接口绑定，简化了使用Spring cloud Ribbon时，自动封装服务调用客户端的开发量。</p> <p>开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用。所以，Feign在此基础上做了进一步封装，由他来帮助我们定义和实现依赖服务接口的定义。在Feign的实现下，我们只需创建一个接口并使用注解的方式来配置它(以前是Dao接口上面标注Mapper注解,现在是一个微服务接口上面标注一个Feign注解即可)，即可完成对服务提供方的接口绑定，简化了使用Spring cloud Ribbon时，自动封装服务调用客户端的开发量。</p> <p>Feign集成了Ribbon</p> <p>利用Ribbon维护了MicroServiceCloud-Dept的服务列表信息，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，通过feign只需要定义服务绑定接口且以声明式的方法，优雅而简单的实现了服务调用</p>
--	--

Hystrix 断路器

服务雪崩

多个微服务之间调用的时候，假设微服务 A 调用微服务 B 和微服务 C，微服务 B 和微服务 C 又调用其它的微服务，这就是所谓的“扇出”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务 A 的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“雪崩效应”。

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几秒钟内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障。这些都表示需要对故障和延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或系统。

hystrix 是什么

Hystrix 是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix 能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

服务熔断

熔断机制是应对雪崩效应的一种微服务链路保护机制。

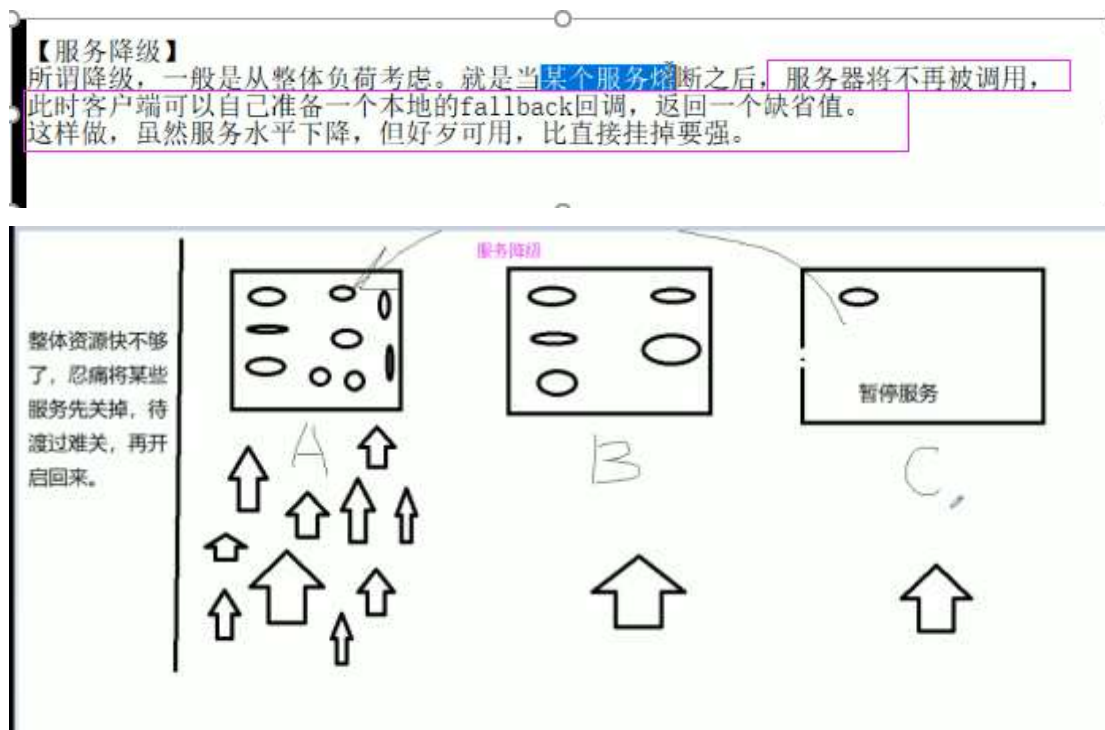
当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回“错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在 SpringCloud 框架里熔断机制通过 Hystrix 实现。Hystrix 会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是 5 秒内 20 次调用失败就会启动熔断机制。熔断机制的注解是 @HystrixCommand。

服务降级（服务降级处理是在客户端实现完成的，与服务端没有关系）

整体资源快不够了，忍痛将某些服务先关掉，待渡过难关，再开启回来。

服务降级处理是在客户端实现完成的，与服务端没有关系（比如说银行柜台，展厅服务，其他顾客会去找一个开着的柜台）

【服务熔断】
一般是某个服务故障或者异常引起，类似现实世界中的“保险丝”，当某个异常条件被触发，直接熔断整个服务，防止一直等待服务返回。



服务限流

在开发高并发系统时有三把利器用来保护系统：**缓存、降级和限流**。缓存的目的是提升系统访问速度和增大系统能处理的容量，可谓是抗高并发流量的银弹；而降级是当服务出问题或者影响到核心流程的性能则需要暂时屏蔽掉，待高峰或者问题解决后再打开；而有些场景并不能用缓存和降级来解决，比如稀缺资源（秒杀、抢购）、写服务（如评论、下单）、频繁的复杂查询（评论的最后几页），因此需有一种手段来限制这些场景的并发/请求量，即限流。

限流的目的是通过对并发访问/请求进行限速或者一个时间窗口内的的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务（定向到错误页或告知资源没有了）、排队或等待（比如秒杀、评论、下单）、降级（返回兜底数据或默认数据，如商品详情页库存默认有货）。

一般开发高并发系统常见的限流有：限制总并发数（比如数据库连接池、线程池）、限制瞬时并发数（如 nginx 的 limit_conn 模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如 Guava 的 RateLimiter、nginx 的 limit_req 模块，限制每秒的平均速率）；其他还有如限制远程接口调用速率、限制 MQ 的消费速率。另外还可以根据网络连接数、网络流量、CPU 或内存负载等来限流

限流算法

令牌桶、漏桶，计数器。

1、计数器算法

采用计数器实现限流有点简单粗暴，一般我们会限制一秒钟的能够通过的请求数，比如限流qps为100，算法的实现思路就是从第一个请求进来开始计时，在接下去的1s内，每来一个请求，就把计数加1，如果累加的数字达到了100，那么后续的请求就会被全部拒绝。等到1s结束后，把计数恢复成0，重新开始计数。

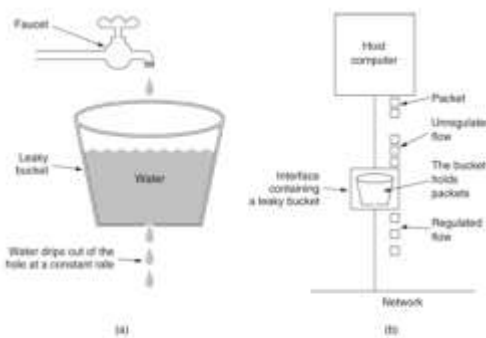
具体的实现可以是这样的：对于每次服务调用，可以通过 AtomicLong#incrementAndGet()方法来给计数器加1并返回最新值，通过这个最新值和阈值进行比较。

这种实现方式，相信大家都有一个弊端：如果我在单位时间1s内的前10ms，已经通过了100个请求，那后面的990ms，只能眼巴巴的把请求拒绝，我们把这种现象称为“突刺现象”

2、漏桶算法

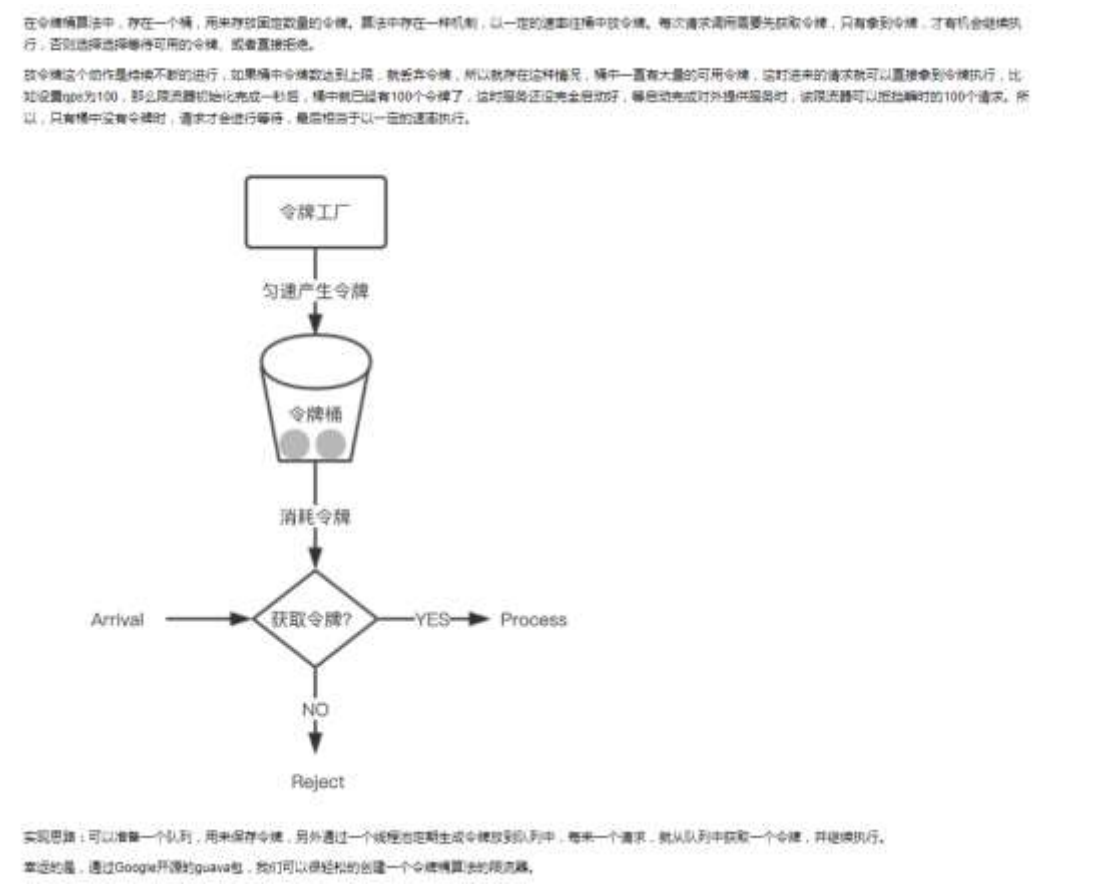
为了消除“突刺现象”，可以采用漏桶算法实现限流。漏桶算法这个名字就很形象，算法内部有一个容器，类似生活用到的漏桶，当请求进来时，相当于水倒入漏桶，然后从下边小口慢慢匀速的流出。不管上面流量多大，下面流出的速度始终保持不变。

不管服务调用方多么不稳定，通过漏桶算法进行限流，每10毫秒处理一次请求。因为处理的速度是固定的，请求进来的速度是未知的，可能突然进来很多请求，来不及处理的请求就先放在桶里，既然是个桶，肯定是有容量上限，如果桶满了，那么新进来的请求就丢弃。



在算法实现方面，可以准备一个队列，用来保存请求，另外通过一个线程定期从队列中获取请求并执行，可以一次性获取多个并发执行。

这种算法，在使用过后也存在弊端：无法应对短时间的突发流量。



hystrix 可以实时监控

除了隔离依赖服务的调用以外，Hystrix 还提供了**准实时的调用监控 (Hystrix Dashboard)**，Hystrix 会持续地记录所有通过 Hystrix 发起的请求的执行信息，**并以统计报表和图形的形式展示给用户**，包括每秒执行多少请求多少成功，多少失败等。

路由网关

Zuu1 包含了对请求的路由和过滤两个最主要的功能：

其中**路由功能负责将外部请求转发到具体的微服务实例上**，**是实现外部访问统一入口的基础而过滤器功能则负责对请求的处理过程进行干预**，是实现请求校验、服务聚合等功能的基础。

Zuu1 和 Eureka 进行整合，将 Zuu1 自身注册为 Eureka 服务治理下的应用，**同时从 Eureka 中获得其他微服务的消息**，也即以后的访问微服务都是通过 Zuu1 跳转后获得。

注意：Zuu1 服务最终还是会注册进 Eureka

提供=代理+路由+过滤三大功能

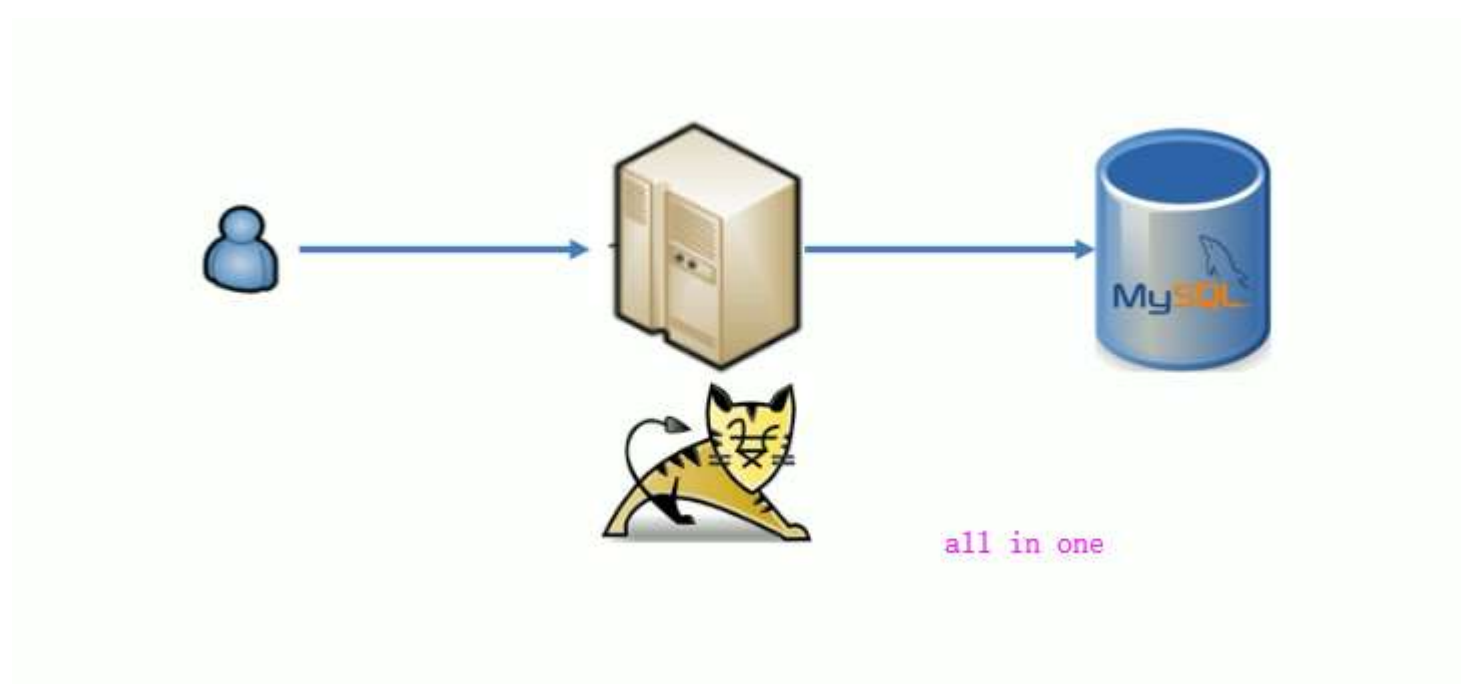
springcloudconfig 分布式配置中心

微服务意味着**要将单体应用中的业务拆分成一个个子服务**，每个服务的粒度相对较小，因此系统中会出现大量的服务。由于每个服务都需要必要的配置信息才能运行，所以一套集中式的、动态的配置管理设施是必不可少的。SpringCloud提供了ConfigServer来解决这个问题，我们每一个微服务自己带有一个application.yml，上百个配置文件的管理...../(T o T)/~~~

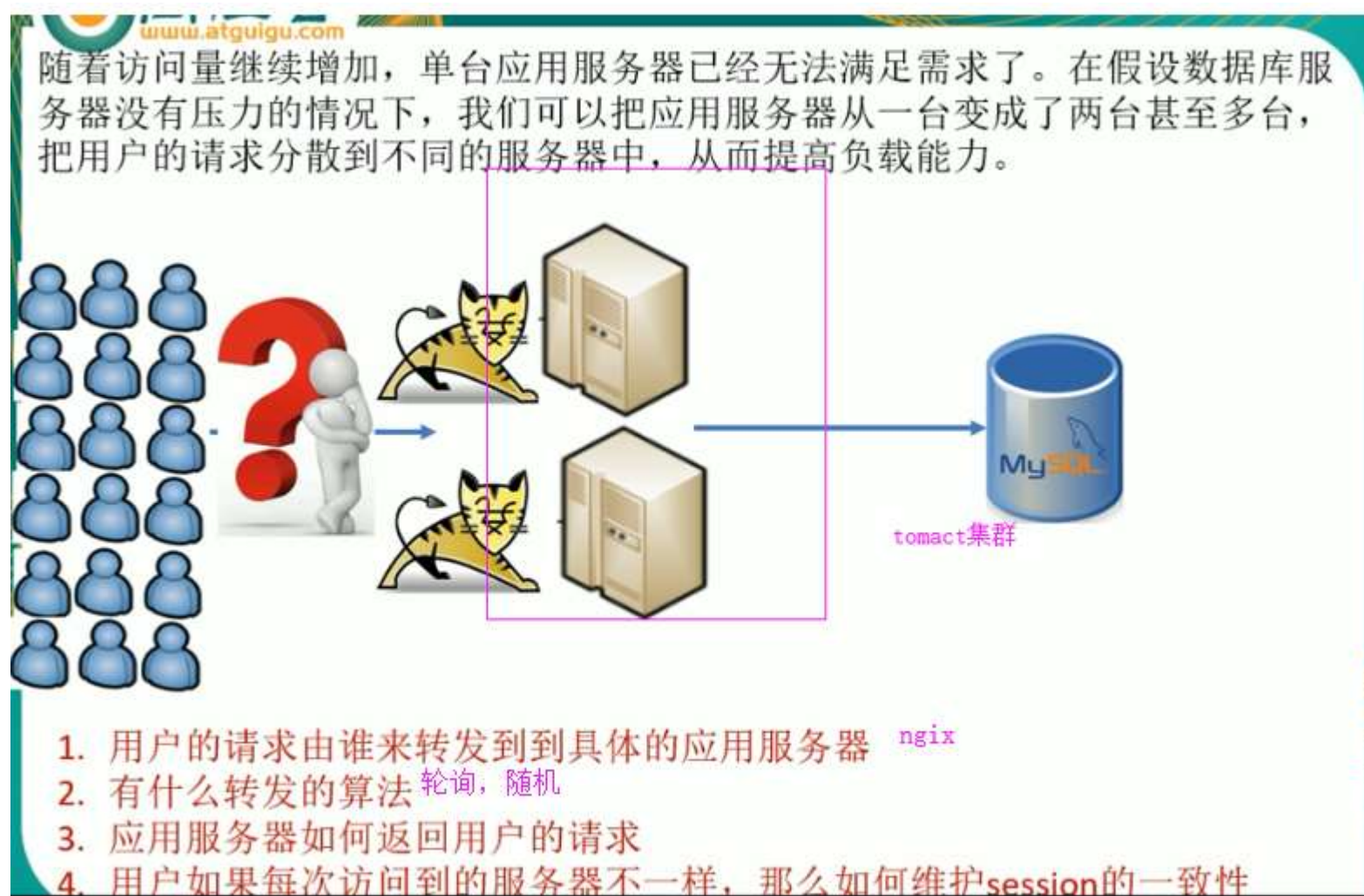
51SpringCloud 重要必看，第一季架构技术总结和第二季展望

<https://tlog.csdn.net/qgyb2000/article/details/80419592> (web 服务端架构演变)

阶段一：单机集中构建网站

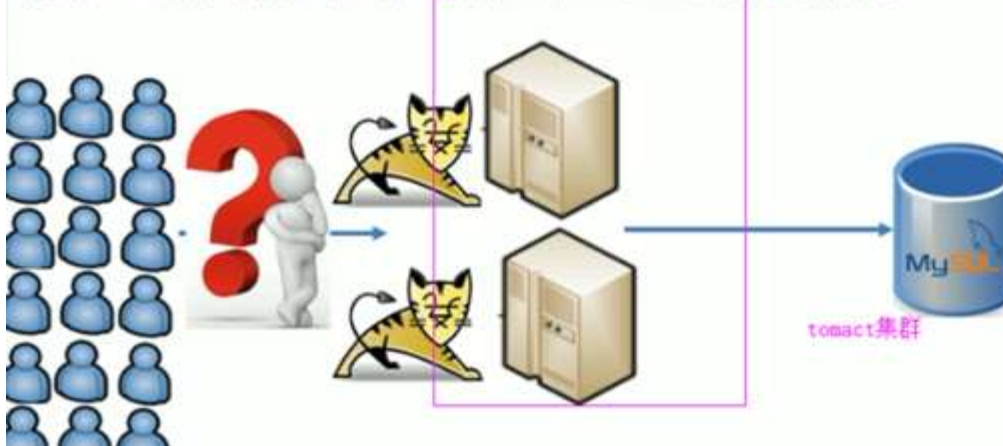


阶段二：应用服务配置集群



www.atguigu.com

随着访问量继续增加，单台应用服务器已经无法满足需求了。在假设数据库服务器没有压力的情况下，我们可以把应用服务器从一台变成了两台甚至多台，把用户的请求分散到不同的服务器中，从而提高负载能力。

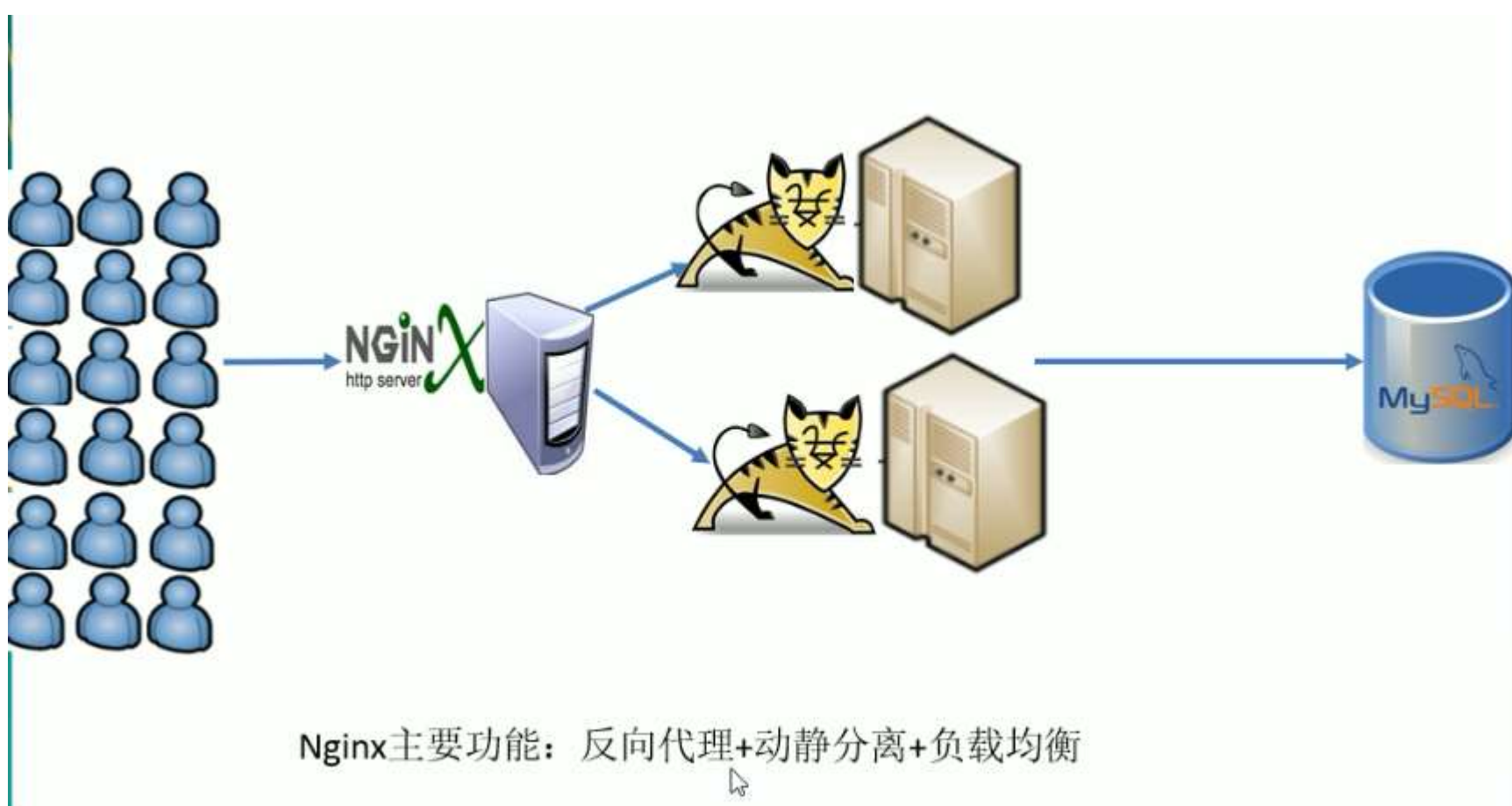


tomcat集群

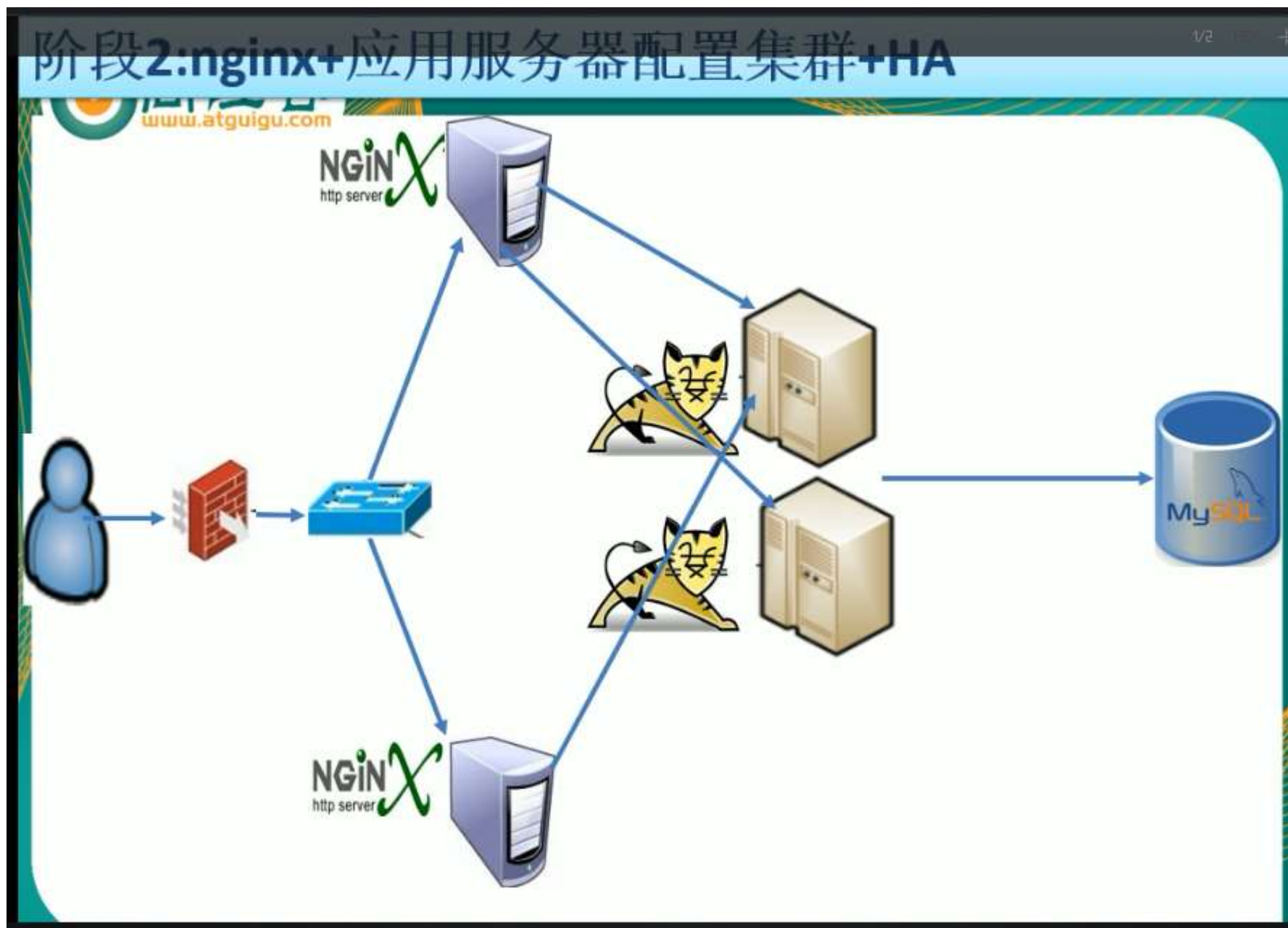
1. 用户的请求由谁来转发到具体的应用服务器 nginx
2. 有什么转发的算法 轮询, 随机
3. 应用服务器如何返回用户的请求 直连
4. 用户如果每次访问到的服务器不一样, 那么如何维护session的一致性

nginx
+redis

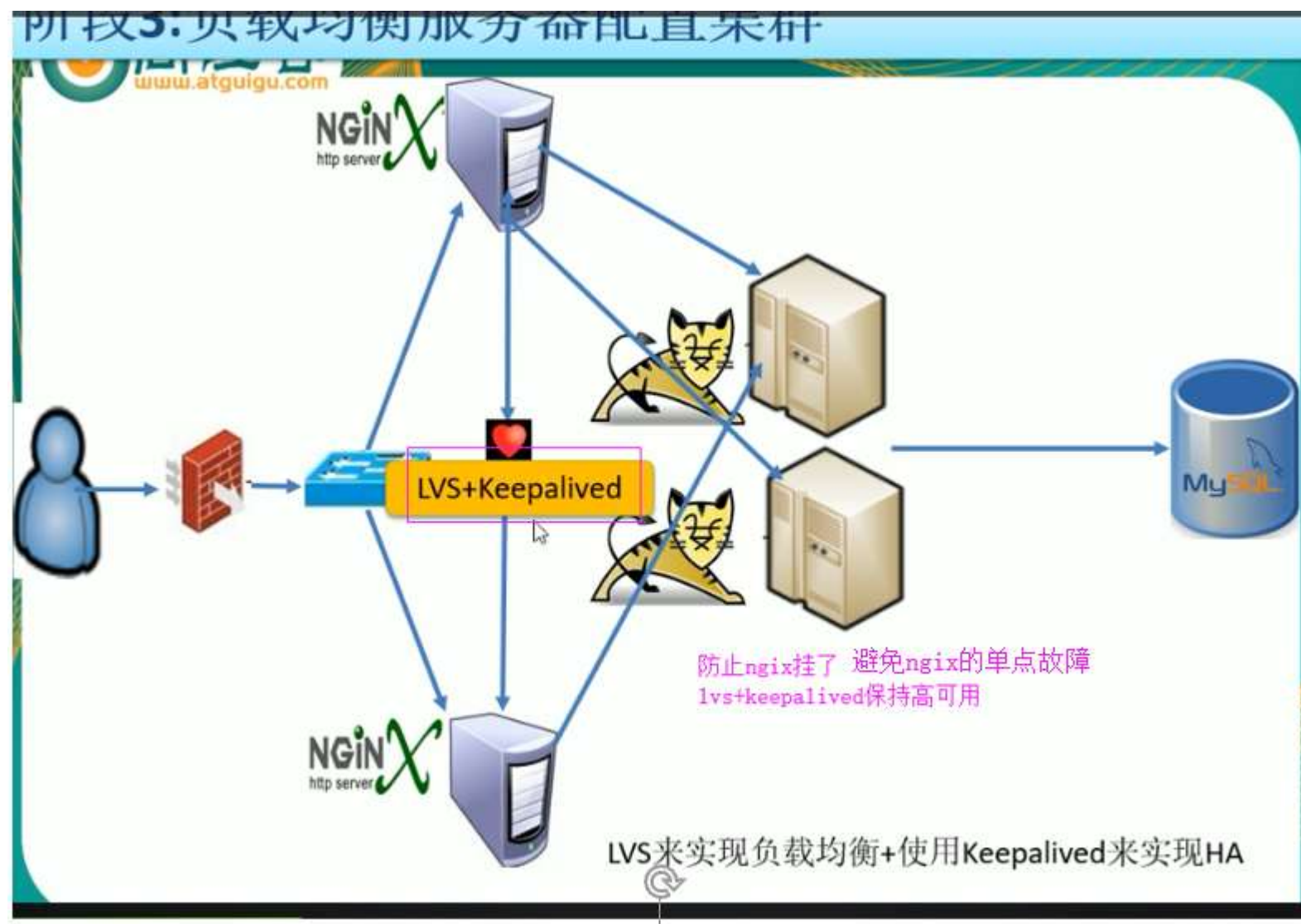
阶段二：nginx+应用服务器配置集群



阶段二：nginx+应用服务器配置集群+HA（避免单点故障）

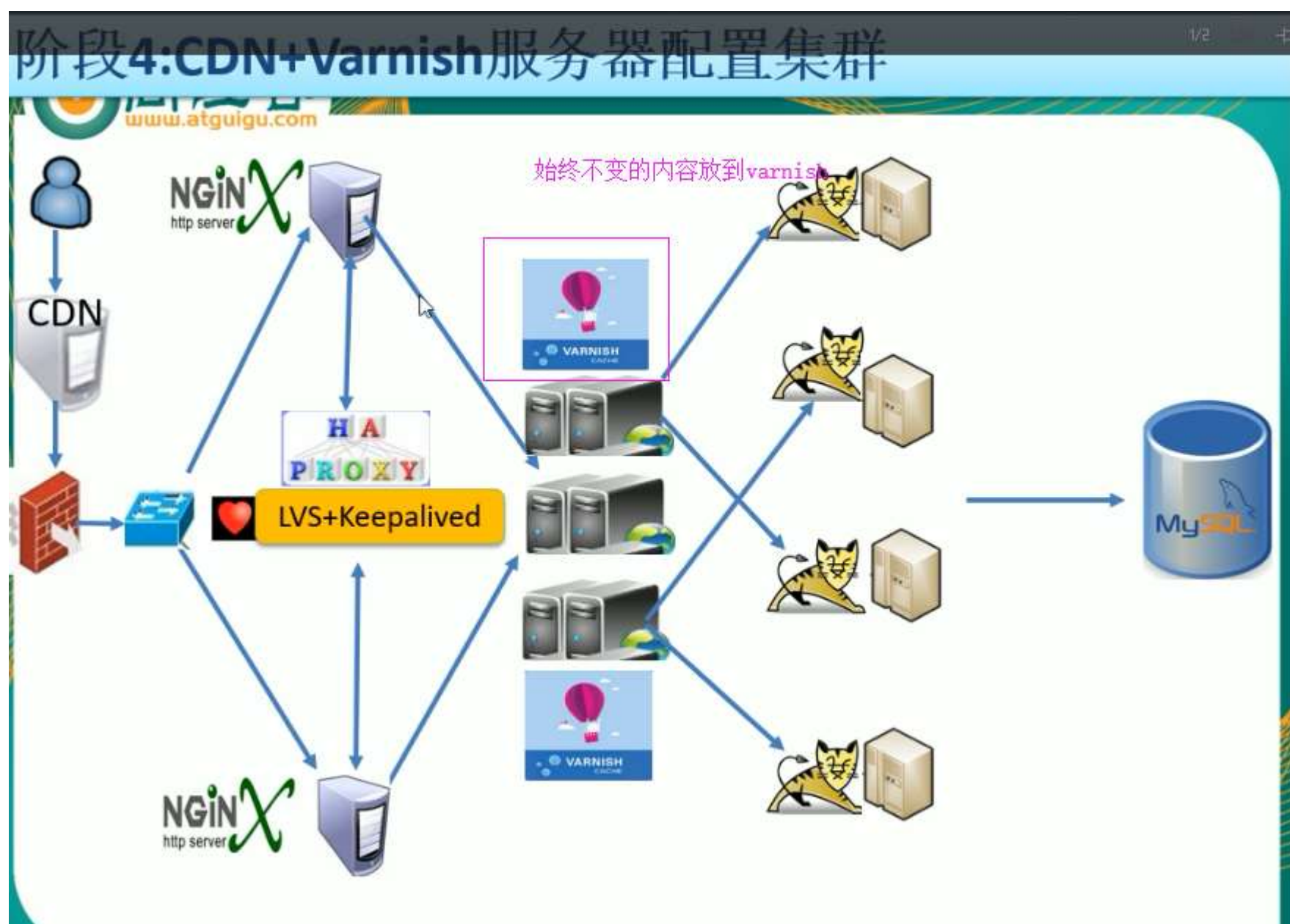


阶段三：负载均衡服务器配置集群



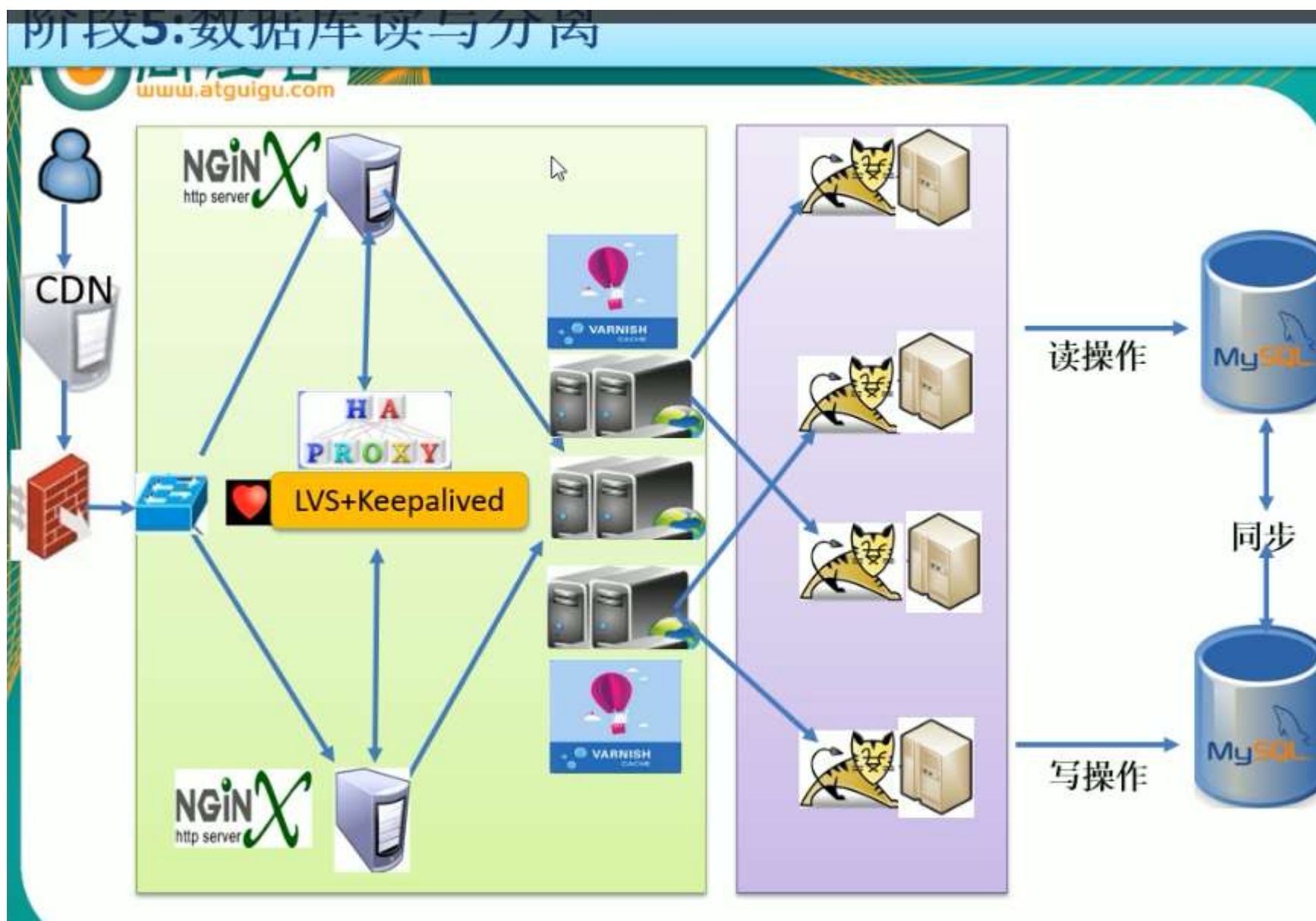
LVS (Linux Virtual Server) 即 Linux 虚拟服务器，是由章文嵩博士主导的开源负载均衡项目，目前 LVS 已经被集成到 Linux 内核模块中。

阶段四：CDN+Varnish 服务器配置集群（动静分离）

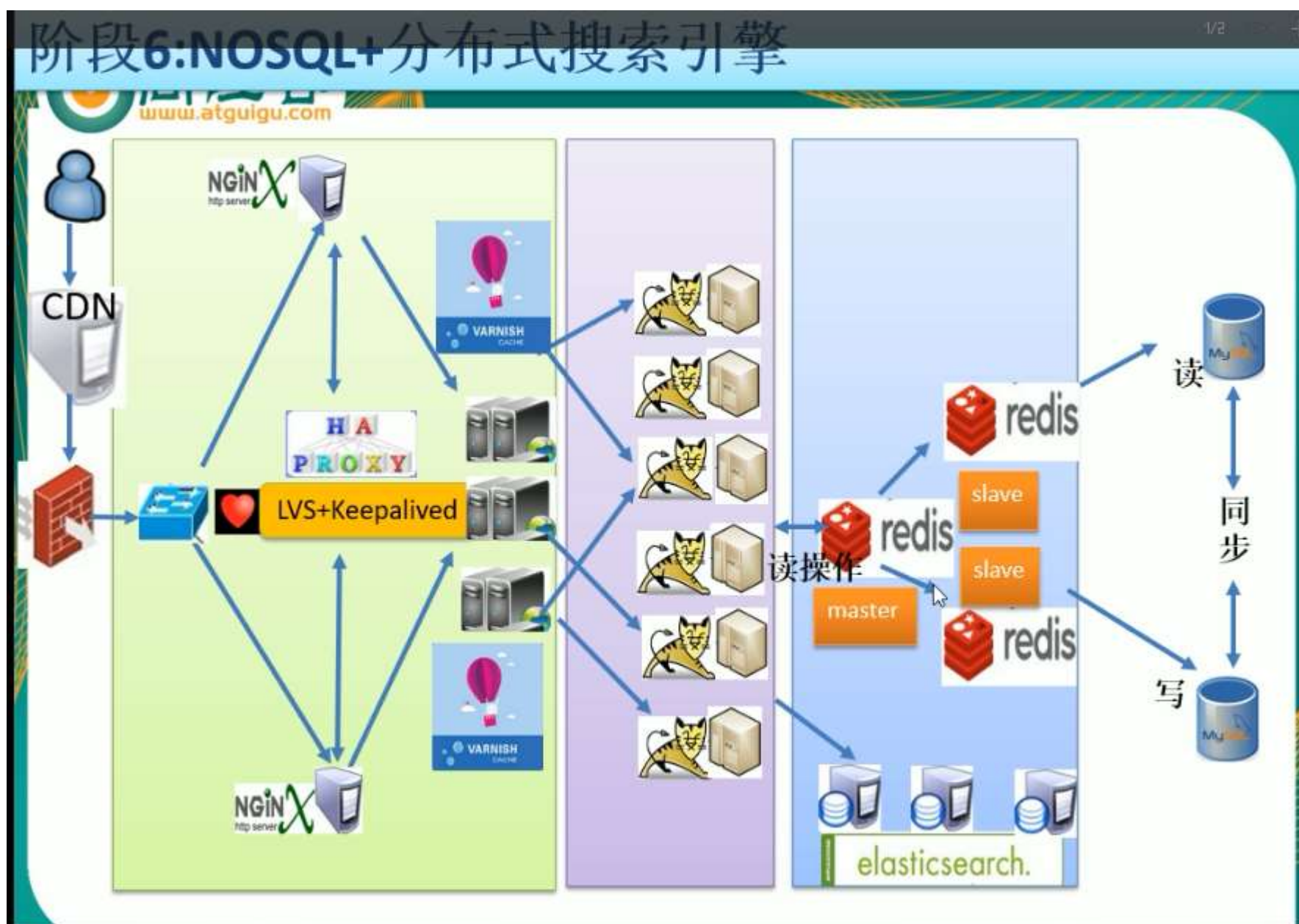


百年不变的内容放到 varnish 中（网站首页等） （varnish 反向代理服务器的加速器） 有些内容不用去找 tomact 直接去找 varnish 获得

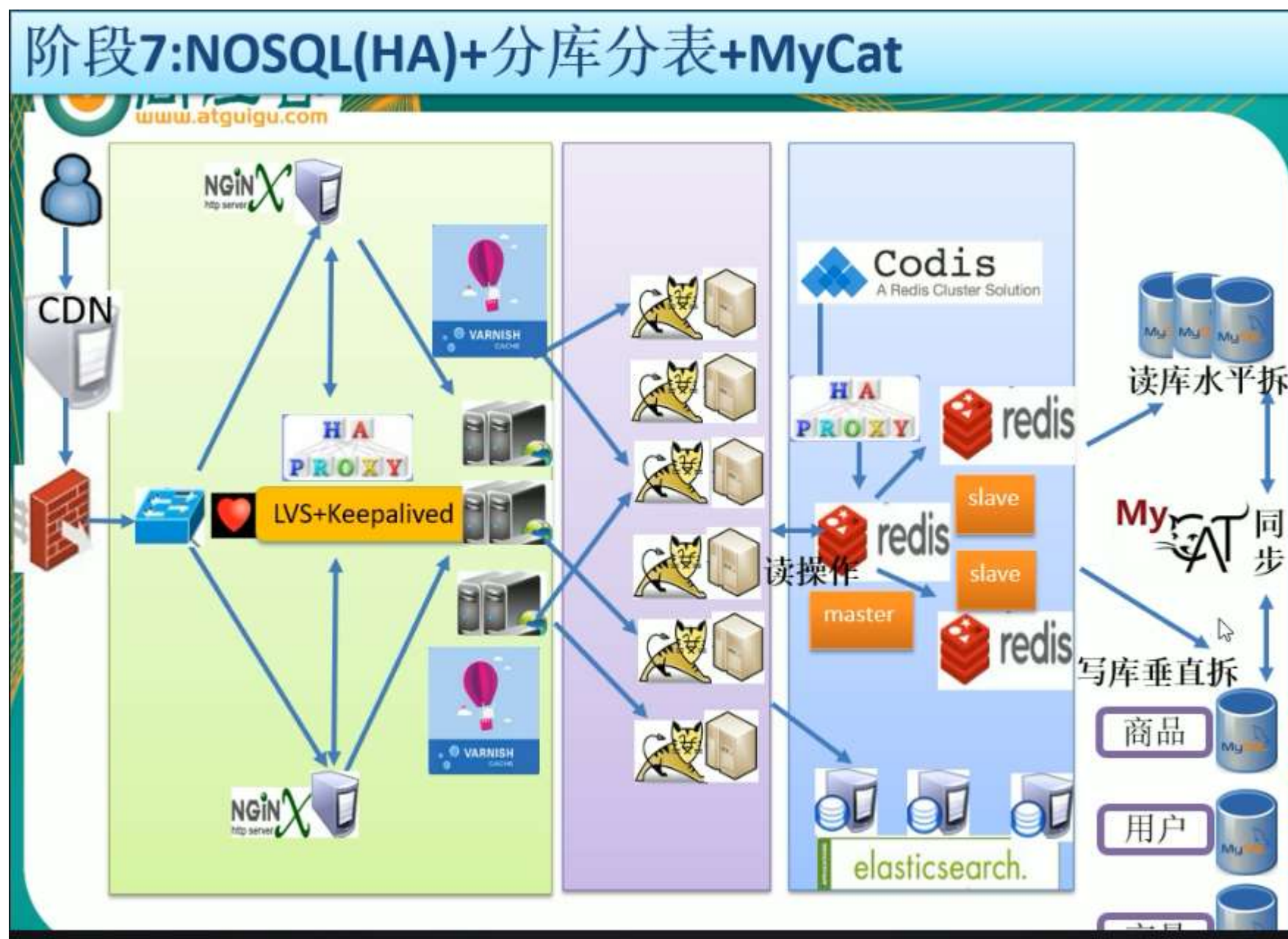
阶段五:数据库读写分离



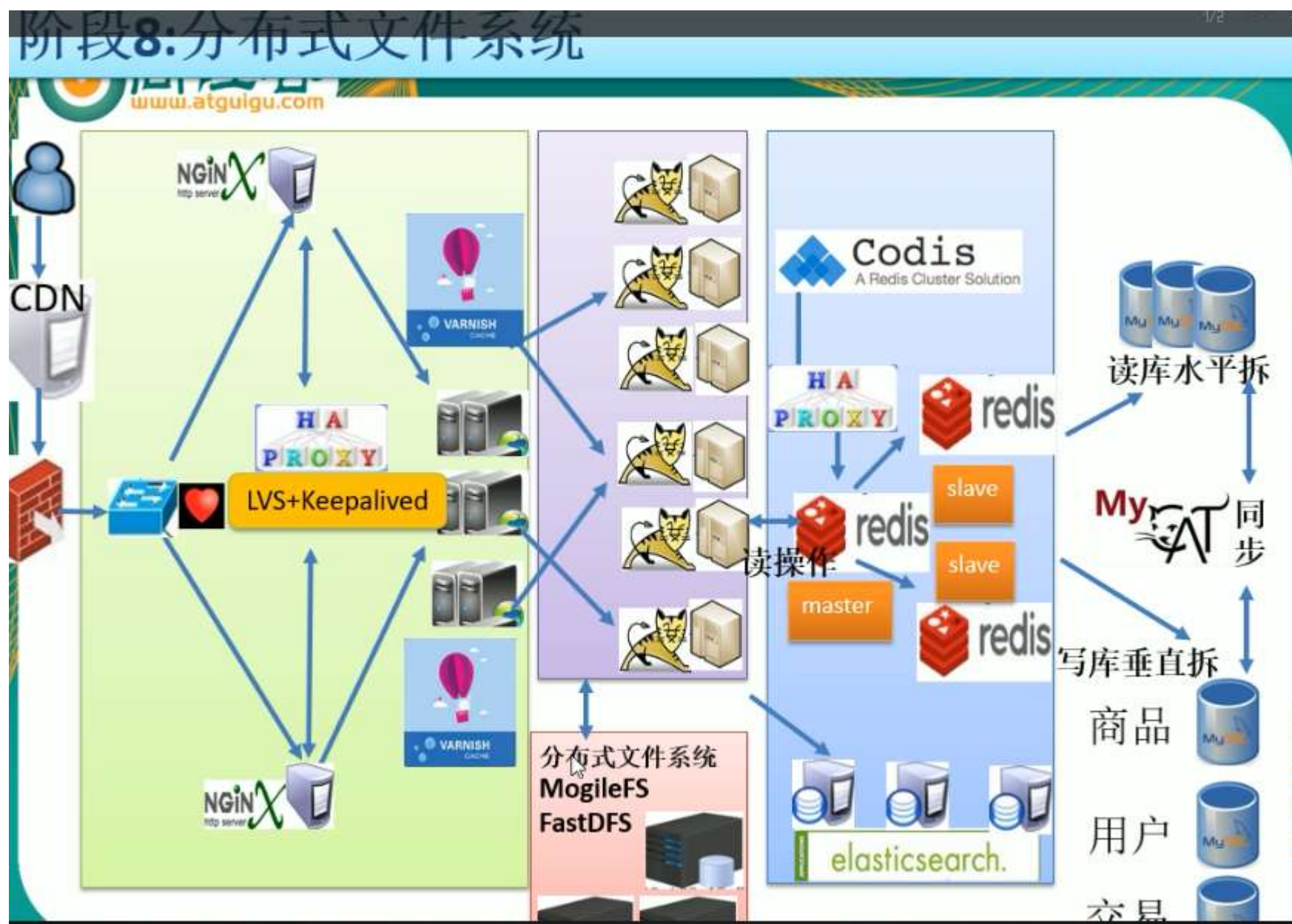
阶段 6: NOSQL+分布式搜索引擎



阶段七:NOSQL (HA) +分库分表+myCat



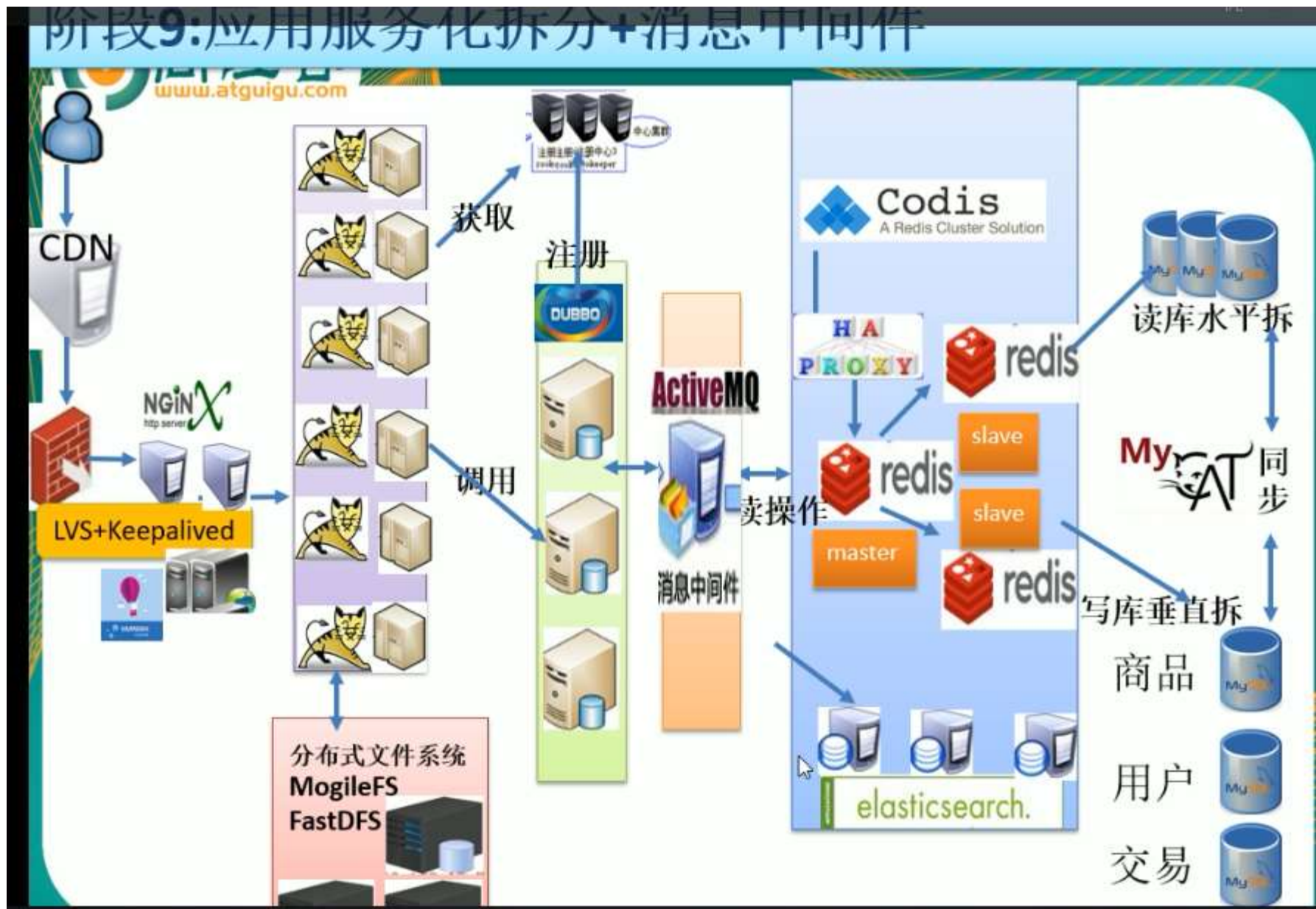
阶段八：分布式文件系统



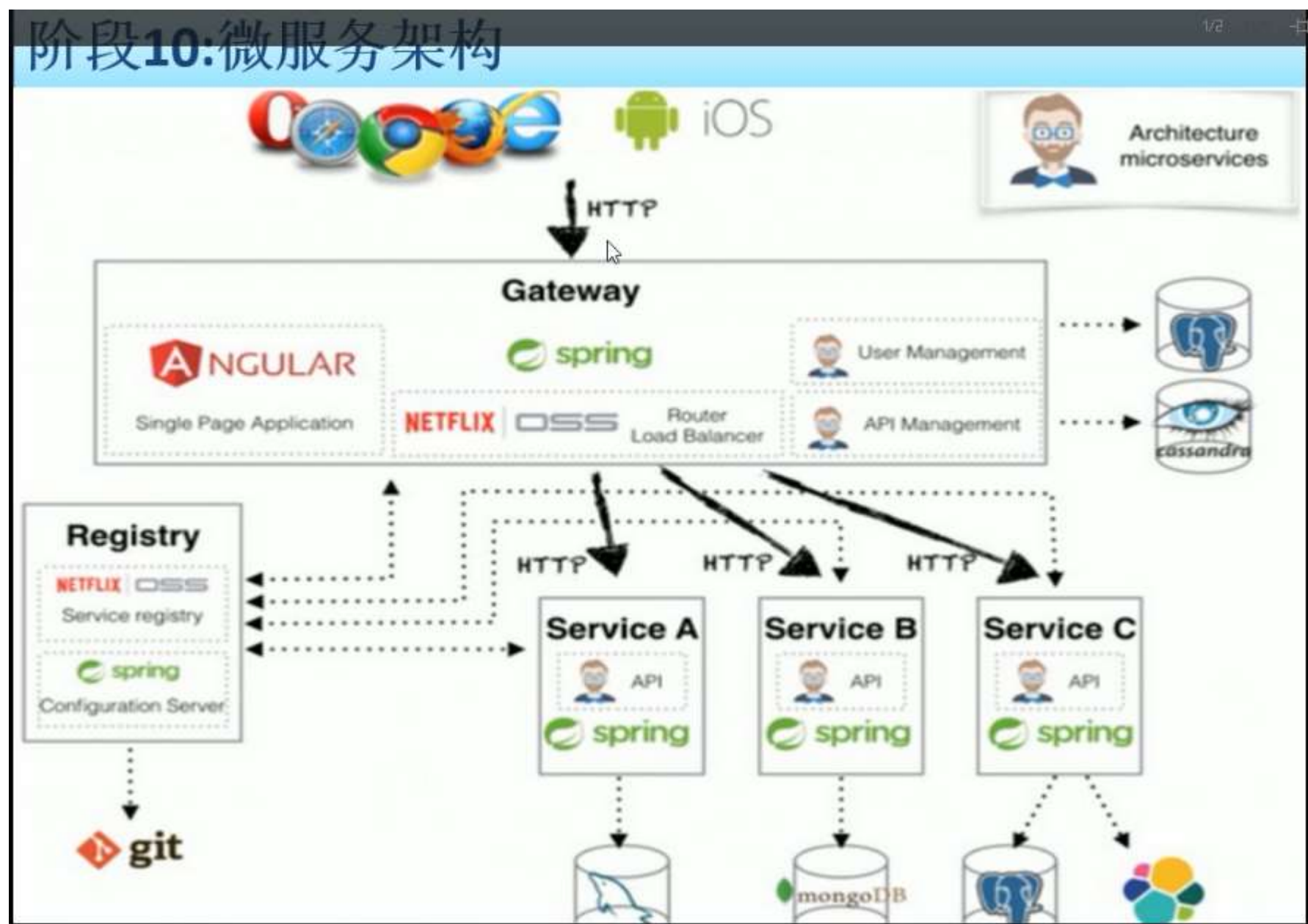
京东 2017618 交易平台目前的架构体系



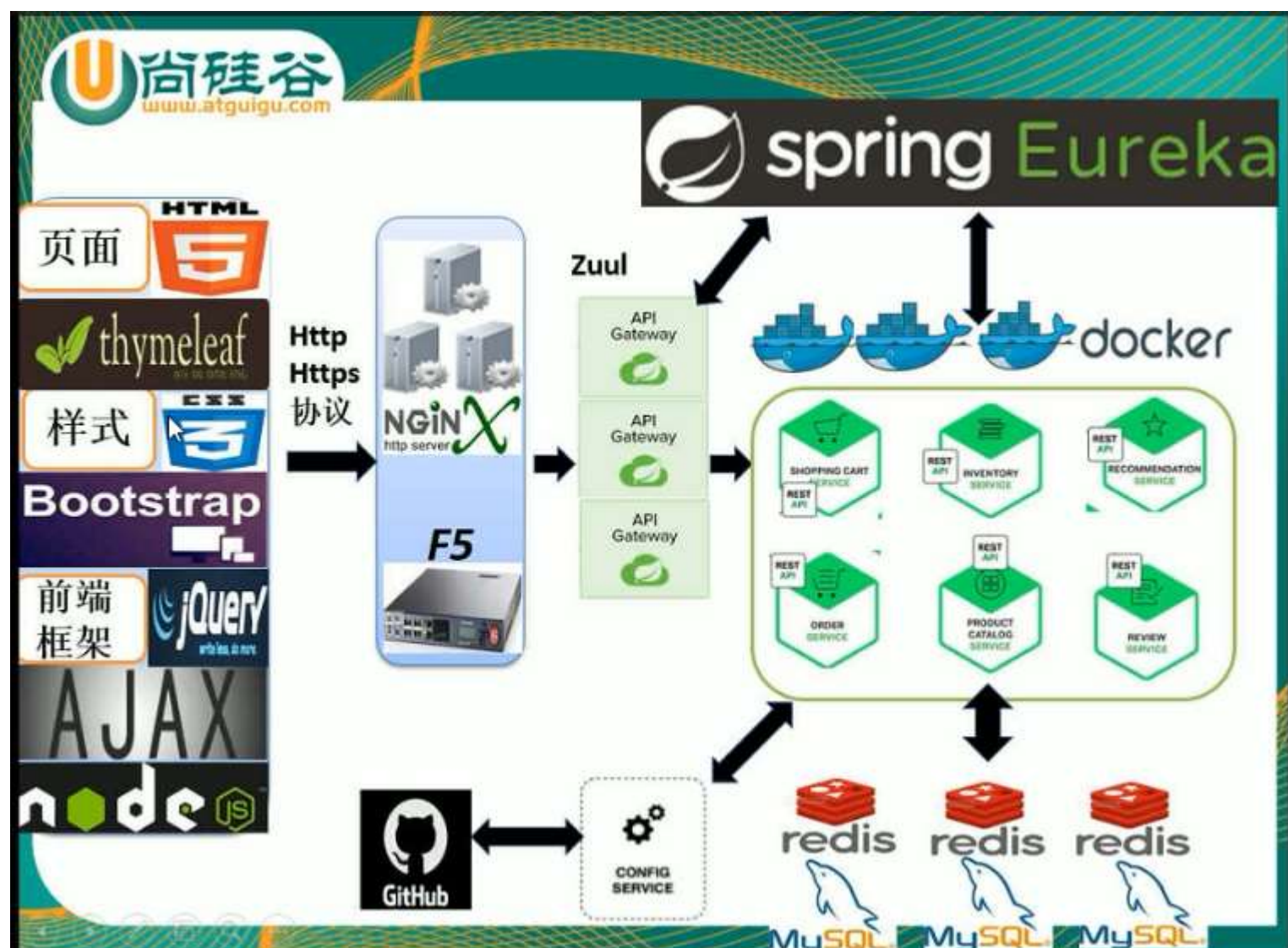
阶段九：应用服务化拆分+消息中间件



阶段十微服务架构



第一季度内容技术梳理与架构





分布式存在的问题

1.分布式 **session 问题**：因为在分布式系统中，服务器集群，同一服务通常会放在几台不同的服务器中，当浏览器第一次发来请求或原 session 已经失效时，会在服务器端创建 session，并将 sessionId 放在响应头中返回浏览器保存在 cookie。当浏览器第二次访问时，会带着 sessionId 在服务器中查找 session，虽然两次访问的网址相同，但是请求可能打到两个不同的 tomcat 上，这样第二次请求找不到之前的 session。这个问题有以下 5 种解决方法：

使用数据库进行 session 共享

使用 redis 进行 session，k，v 分别是 sessionId，session，该方法已经集成在 springboot 中，在 redis 配置类上加@EnableRedisHttpSession

使用 token 加 session，token 和 session 都是临时且唯一的，redis 的 k，v 分别是 token，session

使用 tomcat 自带的 session 同步工具，但是效果不好有延时

在网关层做 ip hash 操作，确保相同 ip 的请求可以打到相同的 tomcat

2.分布式跨域问题(从 A 系统调用 B 系统页面)：前端页面的 ajax 请求在访问当前页面所在的系统的服务接口时，不会发生任何问题，但是在访问

系统的页面时，可以将请求成功发出去，也可以成功返回，但浏览器不会允许展示，这个问题有 4 种解决办法：

在响应头中加“access-control-allow-origin”

用 nginx 或 zuul 反向代理，请求全都打到反向代理服务器，在通过反向代理服务器发请求，并返回给 ajax，在浏览器看来都是同一个系统

用 jsonp，将 ajax 的 type 设置 jsonp，但是 jsonp 只能支持 get 请求

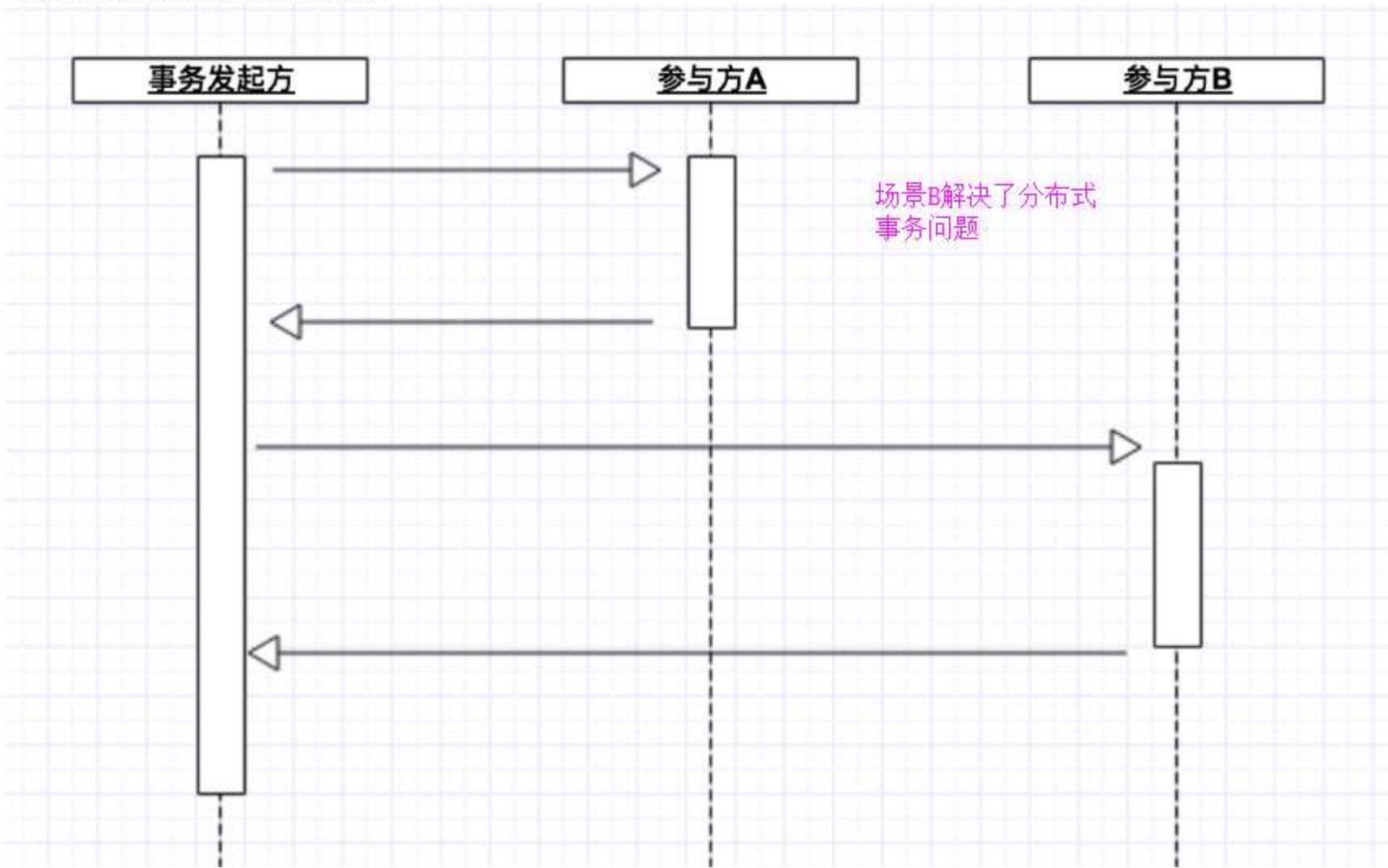
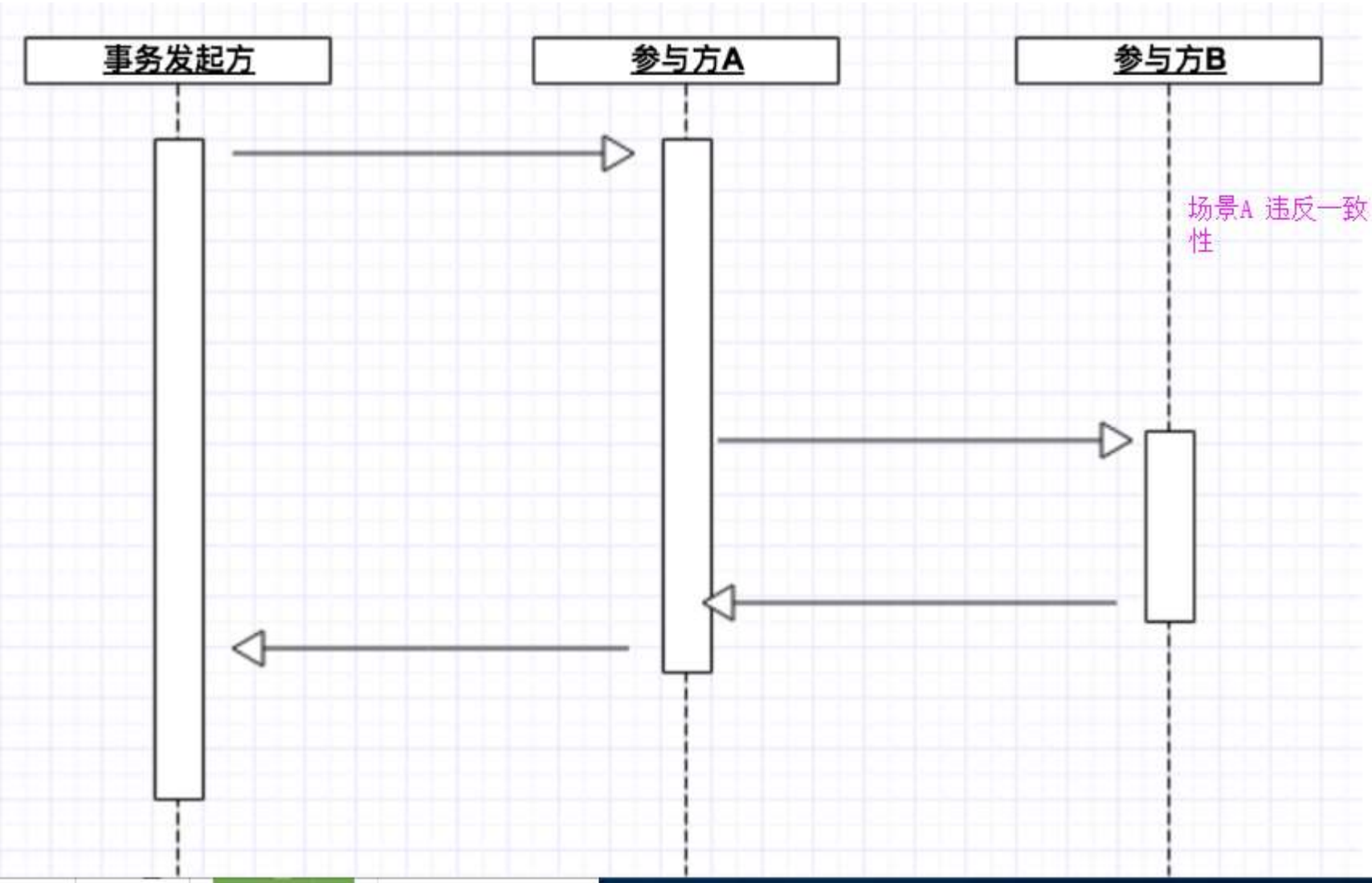
ajax 打到本系统的某接口，在通过这个接口封装 httpclient 或其他 rpc 框架发请求

3.分布式任务调度：系统中都会有定时任务，当同一服务被部署到多台不同服务器后，同一定时任务就会被被执行多次，比如定时进行 redis 备份，那么 redis 就会被备份多份，这显然是不对的。

通过使用 xxl-job 调度中心进行统一调度，保证同一时间相同任务只有一个被执行。

4.分步式事物：一个方法 A 改了自己的数据库，成功后，又通过 rpc 发请求给另一个服务 B 改 B 的数据库，也成功，返回 true，结果网断了，A 没收到 B 的响应，超时之后，A 数据库回滚，B 不动，这样就违反一致性。

用 lcn 框架解决，lcn 是国人开发，核心思想是“不生产事物，只做本地事物搬运工”，采用 2pc 协议，也就是当两个数据库都操作完成并成功后，都会给 lcn 发信号，这时 lcn 在同一 commit，否则 lcn 统一 rollback



5.分布式幂等性：例如表单重提交问题，应该提交多次只有一次起作用，在传统系统中，多次提交会打到统一接口，可以再接口内部进行请求去重，但是在分布式系统中，多次提交可能打到不同服务器上，不能再按照之前的方法去重

解决方法：用 aop 做请求拦截，在前置通知中，判断过来的请求头中是否有 token，如果有放行，如果没有，采用 uuid+时间戳的方式，基于分布式锁生成全局唯一 token，加到请求头中，在将 token 存放到 redis，放行。请求到达环绕通知，首先判断请求头中的 token 在 redis 能否查到，如果能，删除 redis 的 token，执行请求接口。如果 redis 中没有对应的 token，说明请求是第二次或多次调用，拦截。

6.分布式锁：当多个线程同时对一个变量进行读写操作，就会产生线程安全问题，但是传统的 synchronized 对其他系统的线程是不可见的，所以需要有一个对所有服务所有线程都可见的锁。

三中解决方法：基于数据库，基于 redission，基于 zookeeper。在 zookeeper 约定好的路径下建临时节点，同一时间只有一个线程能建立成功，并且在删除之前都不能创建相同路径的节点，定义 watch 可监听节点的删除操作，删除后其他节点能及时得到通知而不用轮巡的尝试创建锁。在这提一下 java 的一个基础知识，synchronized 和 lock 的区别，lock 有 tryLock 方法，可以尝试创建锁，并拿到返回值，lock 能够知道当前有没有锁，并进行处理，而 synchronized 在不能加锁的时候阻塞住，直到能加锁为止。zookeeper 分布式锁类似 lock，在创建临时节点的时候，创建成功和失败都会有返回值。

redission 方式：redission 和 ReentrantLock 是唯二的两个实现了 lock 接口的类，reentrantlock 是重入锁，redission 是分布式锁，用法和 lock 相同，是基于 redis 实现的。

token 的作用及实现原理

每次访问服务器资源的时候需要带着这个 token，然后怎么判断是否有呢？就要用拦截器来实现过滤，用拦截器去判断这个 ticket 当前的状态是

什么样的？有没有过期？身份状态是不是有效的？然后根据这个来判断应该赋予什么样的权限？

1：首先，先了解一下 request 和 session 的区别
request 指在一次请求的全过程中有效，即从 http 请求到服务器处理结束，返回响应的整个过程，存放在 HttpServletRequest 对象中。在这个过程中可以使用 forward 方式跳转多个 jsp。在这些页面里你都可以使用这个变量。request 是用户请求访问的当前组件，以及和当前 web 组件共享同一用户请求的 web 组件。如：被请求的 jsp 页面和该页面用<include>指令包含的页面以及<forward>标记包含的其它 jsp 页面；
Session 是用户全局变量，在整个会话期间都有效。只要页面不关闭就一直有效（或者直到用户一直未活动导致会话过期，默认 session 过期时间为 30 分钟，或调用 HttpSession 的 invalidate()方法）。存放在 HttpSession 对象中，同一个 http 会话中的 web 组件共享它。

2：token 主要有两个作用：①：防止表单重复提交②：用来作身份验证

(防止表单重复提交一般还是使用前后端都限制的方式，比如：在前端点击提交之后，将按钮置为灰色，不可再次点击，然后客户端和服务端的 token 各自独立存储，客户端存储在 Cookie 或者 Form 的隐藏域（放在 Form 隐藏域中的时候，需要每个表单）中，服务端存储在 Session（单机系统中可以使用）或者其他缓存系统（分布式系统可以使用）中。)

4：使用基于 Token 的身份验证方法，在服务端不需要存储用户的登录记录。大概的流程是这样的：

客户端使用用户名跟密码请求登录
服务端收到请求，去验证用户名与密码
验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端
客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里
客户端每次向服务端请求资源的时候需要带着服务端签发的 Token
服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据

分布式锁的三种实现方式

<https://www.jianshu.com/p/8bddd381de06>

分布式锁一般有三种实现方式：

- 1、数据库锁
- 2、基于Redis的分布式锁
- 3、基于ZooKeeper的分布式锁

分布式锁应该是怎么样的

- 互斥性 可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。
- 这把锁要是一把可重入锁（避免死锁）
- 不会发生死锁：有一个客户端在持有锁的过程中崩溃而没有解锁，也能保证其他客户端能够加锁
- 这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）
- 有高可用的获取锁和释放锁功能
- 获取锁和释放锁的性能要好

数据库锁

- 要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。
- 当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

```
1
2 CREATE TABLE `methodLock` (
3   `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
4   `method_name` varchar(64) NOT NULL DEFAULT '' COMMENT '锁定的方法名',
5   `desc` varchar(1024) NOT NULL DEFAULT '备注信息',
6   `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
7   ON UPDATE CURRENT_TIMESTAMP COMMENT '保存数据时间，自动生成',
8   PRIMARY KEY (`id`),
9   UNIQUE KEY `uidx_method_name` (`method_name`) USING BTREE
10
```

image

当我们想要锁住某个方法时，执行以下SQL：

```
1 insert into methodLock(method_name,desc) values ('method_name','desc')
```

image

因为我们对method_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为

操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下Sql:

```
1 delete from methodLock where method_name ='method_name'
```

image

数据库锁存在的问题

- 上面这种简单的实现有以下几个问题：

- 1、这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。
- 2、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得锁。
- 3、这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。
- 4、这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

- 当然，我们也可以有其他方式解决上面的问题。

- 数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。
- 没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。
- 非阻塞的？搞一个while循环，直到insert成功再返回成功。

- 非重入的？在数据库表中加个字段，记录当前获得锁的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前主机的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

- 基于数据库的排它锁

除了可以通过增删操作数据表中的记录以外，其实还可以借助数据库中自带的锁来实现分布式的锁。

我们还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。

在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：

```
1 public void unlock(){
2     connection.commit();
3 }
```

通过connection.commit()操作来释放锁。

这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。

- 阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。
- 锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。

- 总结：

- 总结一下使用数据库来实现分布式锁的方式，这两种方式都是依赖数据库的一张表，一种是通过表中的记录的存在情况确定当前是否有锁存在，另外一种是通过数据库的排他锁来实现分布式锁。

- 数据库实现分布式锁的优点: 直接借助数据库，容易理解。
- 数据库实现分布式锁的缺点: 会有各种各样的问题，在解决问题的过程中会使整个方案变得越来越复杂。
- 操作数据库需要一定的开销，性能问题需要考虑。

基于 Redis 的分布式锁

2、基于redis的分布式锁

- 相比较于基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现的更好一点。而且很多缓存是可以集群部署的，可以解决单点问题。

-首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

- 互斥性。在任意时刻，只有一个客户端能持有锁。
- 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
- 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
- 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

可以看到，我们加锁就一行代码：`jedis.set(String key, String value, String nxxx, String expx, int time)`，这个`set()`方法一共有五个形参：

- 第一个为key，我们使用key来当锁，因为key是唯一的。
- 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用`UUID.randomUUID().toString()`方法生成。
- 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
- 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
- 第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的`set()`方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

加锁代码满足我们可靠性里描述的三个条件。首先，`set()`加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

- 解锁：
 - 首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）
- 使用缓存实现分布式锁的优点
 - 性能好，实现起来较为方便。
- 使用缓存实现分布式锁的缺点
 - 通过超时时间来控制锁的失效时间并不是十分的靠谱。
- 总结：
 - 可以使用缓存来代替数据库来实现分布式锁，这个可以提供更好的性能，同时，很多缓存服务都是集群部署的，可以避免单点问题。并且很多缓存服务都提供了可以用来实现分布式锁的方法，比如redis的setnx方法等。并且，这些缓存服务也都提供了对数据的过期自动删除的支持，可以直接设置超时时间来控制锁的释放。

基于 Zookeeper 的分布式锁

- 基于zookeeper临时有序节点可以实现的分布式锁。大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的临时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个临时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。
- 完成业务流程后，删除对应的子节点释放锁。
- 来看下Zookeeper能不能解决前面提到的问题。
 - 锁无法释放？使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。
 - 非阻塞锁？使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端。客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。
 - 不可重入？使用Zookeeper也可以有效的解决不可重入的问题，客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。
 - 单点问题？使用Zookeeper可以有效的解决单点问题，ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

三种方式对比

4、三种方案的比较

- 从理解的难易程度角度（从低到高）：数据库 > 缓存 > Zookeeper
- 从实现的复杂性角度（从低到高）：Zookeeper >= 缓存 > 数据库
- 从性能角度（从高到低）：缓存 > Zookeeper >= 数据库
- 从可靠性角度（从高到低）：Zookeeper > 缓存 > 数据库

在实践中，当然是从以可靠性为主。所以首推Zookeeper。

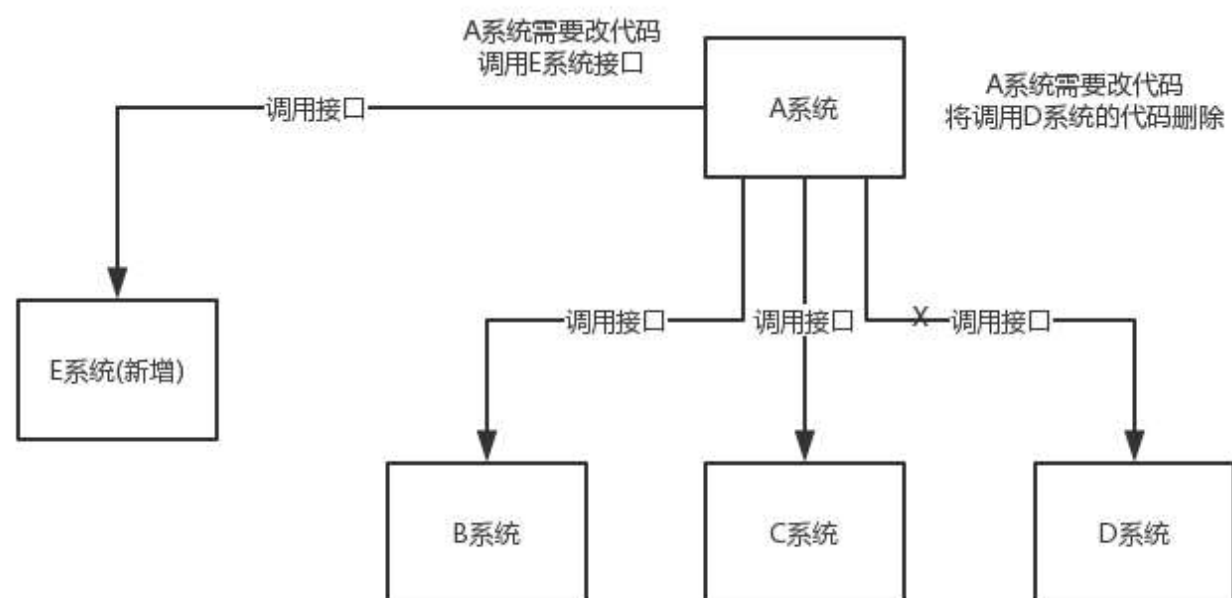
消息队列

消息队列的使用场景

场景有很多，但是比较核心的有 3 个：解耦、异步、削峰。

解耦

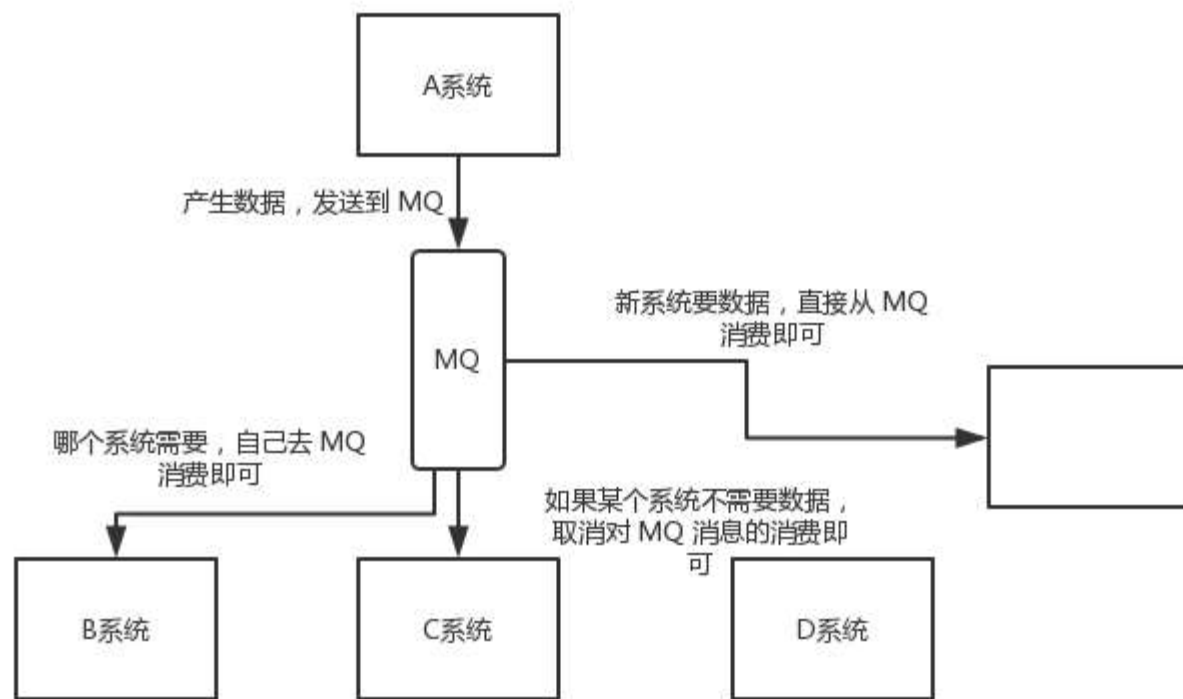
看这么个场景。A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃...



https://blog.csdn.net/baidu_26954825

在这个场景中，A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A 系统要时时刻刻考虑 BCDE 四个系统如果挂了该咋办？要不要重发，要不要把消息存起来，头发都白了啊！

如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。



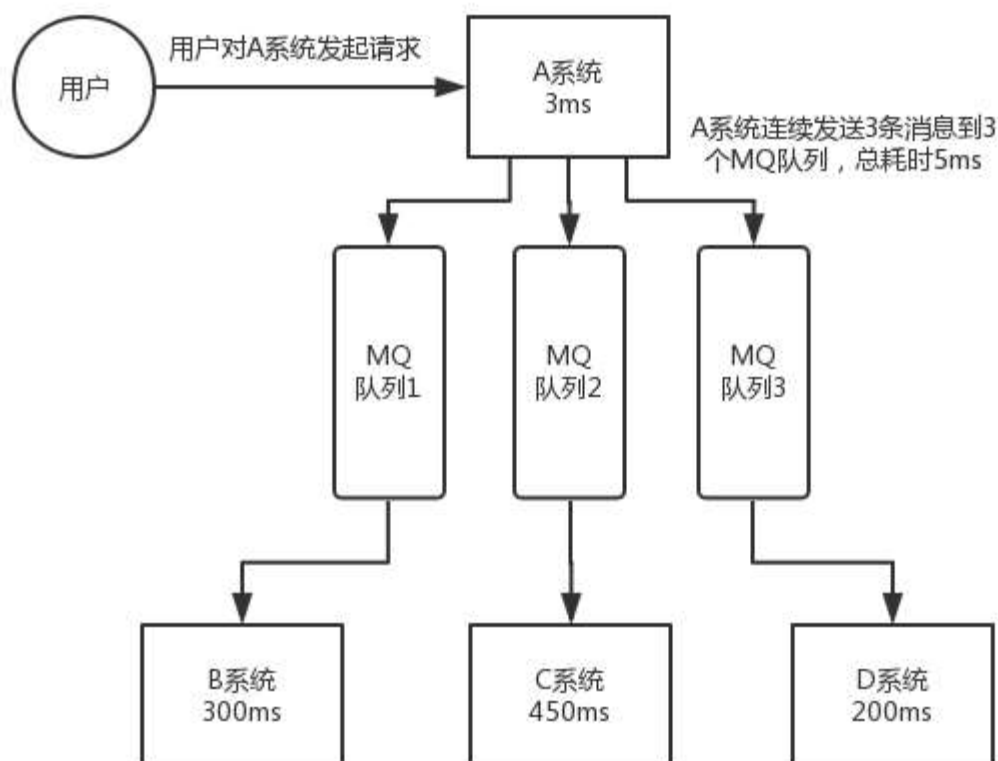
https://blog.csdn.net/baidu_26954625/

总结：通过一个 MQ, Pub/Sub 发布订阅消息这么一个模型，A 系统就跟其它系统彻底解耦了。

面试技巧：你需要去考虑一下你负责的系统中是否有类似的场景，就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦，也是可以的，你就需要去考虑在你的项目里，是不是可以运用这个 MQ 去进行系统的解耦。在简历中体现出来这块东西，用 MQ 作解耦。

异步

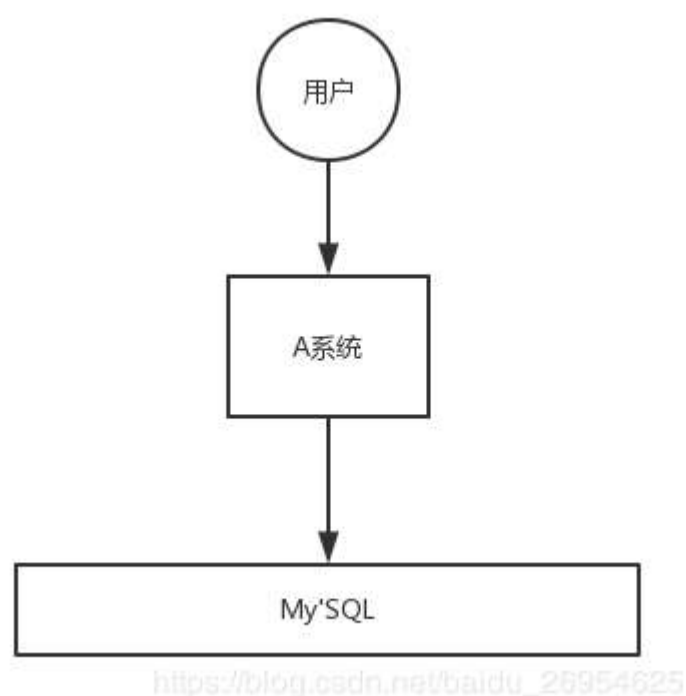
再来看一个场景，A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 $3 + 300 + 450 + 200 = 953\text{ms}$ ，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求，等待个 1s，这几乎是不可接受的。



https://blog.csdn.net/baidu_26954625/

一般互联网类的企业，对于用户直接的操作，一般要求是每个请求都必须在 200 ms 以内完成，对用户几乎是无感知的。如果使用 MQ，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 $3 + 5 = 8\text{ms}$ ，对于用户而言，其实感觉上就是点个按钮，8ms 以后就直接返回了，爽！网站做得真好，真快！

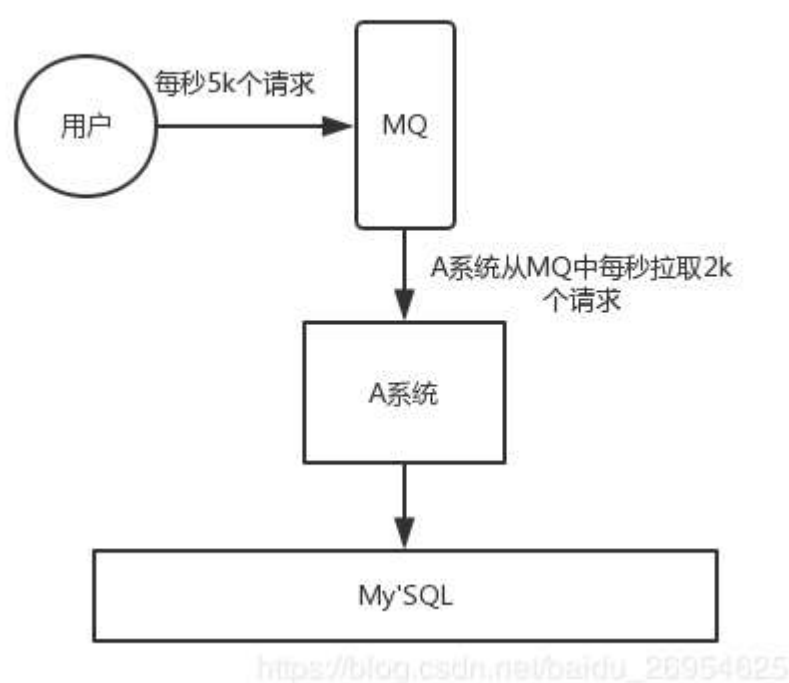
削峰



每天 0:00 到 12:00，A 系统风平浪静，每秒并发请求数量就 50 个。结果每次一到 12:00 ~ 13:00，每秒并发请求数量突然会暴增到 5k+ 条。但是系统是直接基于 MySQL 的，大量的请求涌入 MySQL，每秒钟对 MySQL 执行约 5k 条 SQL。

一般的 MySQL，扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。

但是高峰期一过，到了下午的时候，就成了低峰期，可能也就 1w 的用户同时在网站上操作，每秒中的请求数量可能也就 50 个请求，对整个系统几乎没有任何的压力。



如果使用 MQ，每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉。而 MQ 每秒钟 5k 个请求进来，就 2k 个请求出去，结果就导致在中午高峰期（1 个小时），可能有几十万甚至几百万的请求积压在 MQ 中。

消息队列如何确保可用性，一致性，处理消息丢失，消息传递的顺序性

<https://www.cnblogs.com/juforg/p/10213470.html>

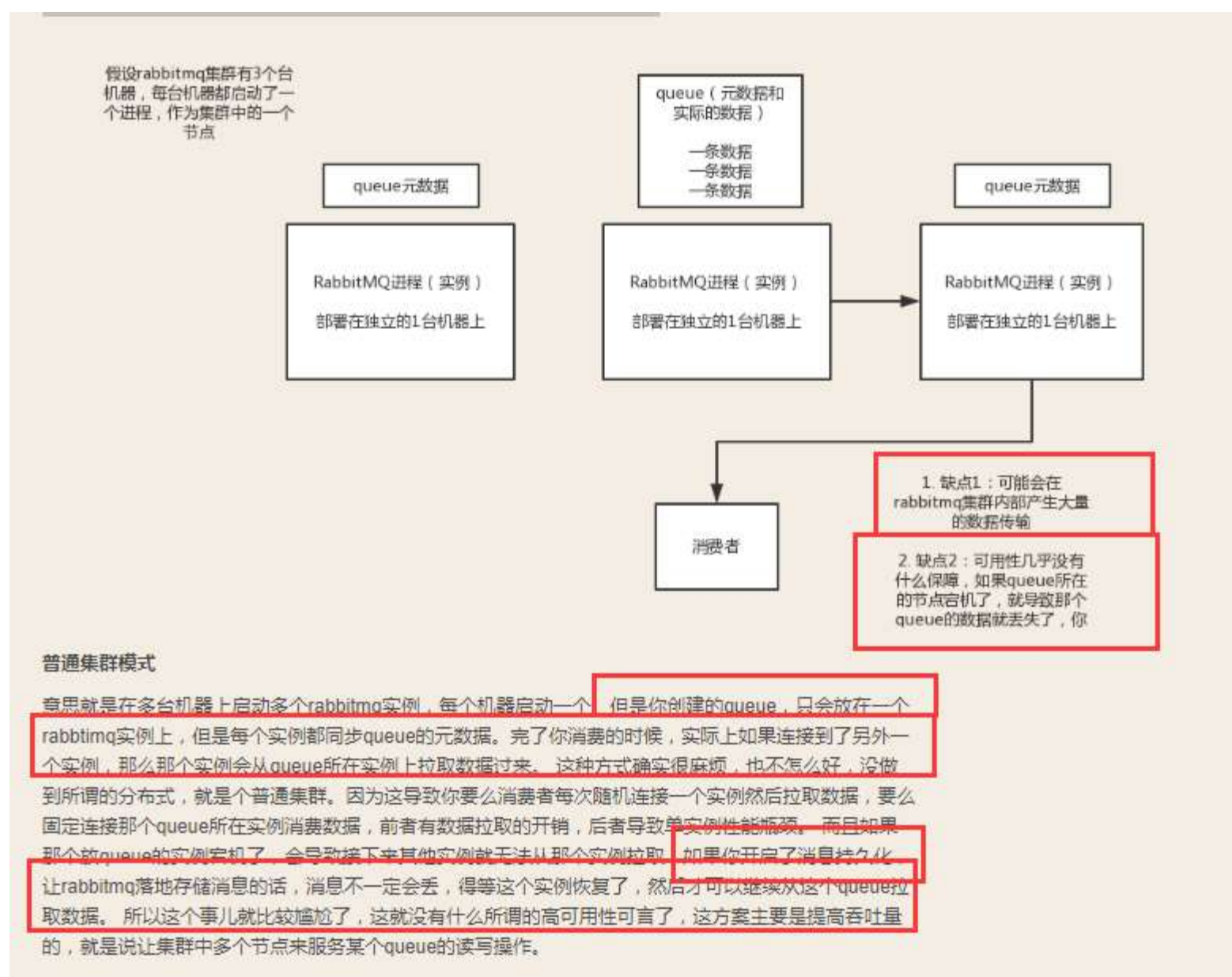
rabbitmq

rabbitmq 的高可用

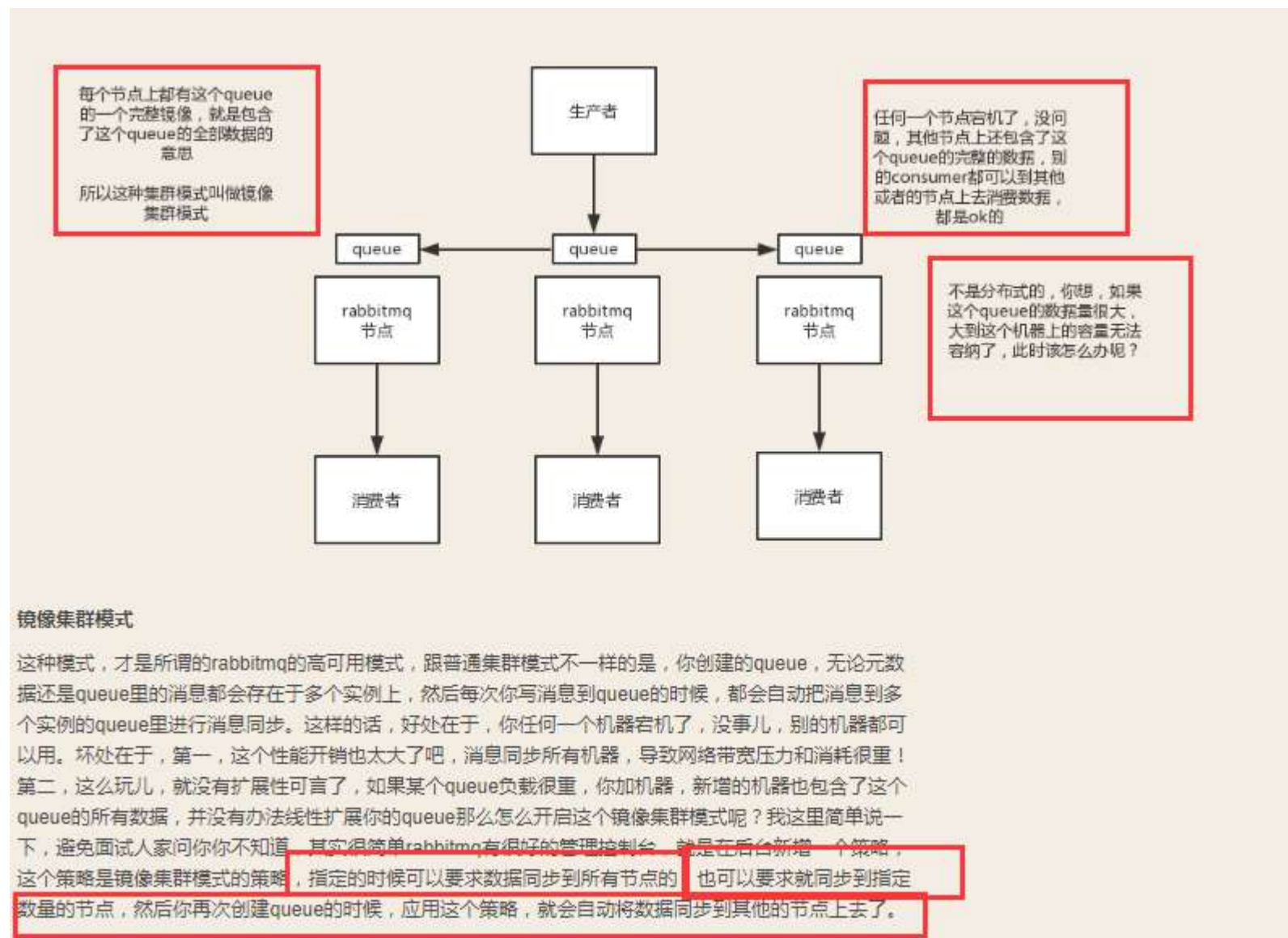
单机模式

就是 demo 级别的，一般就是你本地启动了玩儿的，**没人生产用单机模式**

普通集群模式



镜像集群模式（推荐）



rabbitmq 的数据丢失

生产者弄丢了数据

生产者将数据发送到rabbitmq的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。此时可以选择用rabbitmq提供的事务功能，就是生产者发送数据之前开启rabbitmq事务（channel.txSelect），然后发送消息，如果消息没有成功被rabbitmq接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息，如果收到了消息，那么可以提交事务（channel.txCommit）。但是问题是，rabbitmq事务机制一搞，基本上吞吐量会下来，因为太耗性能。所以一般来说，如果你要确保读写rabbitmq的消息到手，可以开启confirm模式。在生产者那里设置开启confirm模式之后，你每次写的消息都会分配一个唯一的id，然后如果写入了rabbitmq中，rabbitmq会给你回传一个ack消息，告诉你这个消息ok了。如果rabbitmq没能处理这个消息，会回调你一个nack接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息id的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。事务机制和confirm机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是confirm机制是异步的，你发送这个消息之后就可以发送下一个消息，然后那个消息rabbitmq接收了之后会异步回调你一个接口通知你这个消息接收到了。所以一般在生产者这块避免数据丢失，都是用confirm机制的。#####rabbitmq弄丢了数据就是rabbitmq自己弄丢了数据，这个你必须开启rabbitmq的持久化，就是消息写入之后会持久化到磁盘，哪怕是rabbitmq自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，rabbitmq还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。设置持久化有两个步骤，第一个是创建queue的时候将其设置为持久化的，这样就可以保证rabbitmq持久化queue的元数据，但是不会持久化queue里的数据；第二个是发送消息的时候将消息的deliveryMode设置为2，就是将消息设置为持久化的，此时rabbitmq就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，rabbitmq哪怕是挂了，再次重启，也会从磁盘上重启恢复queue，恢复这个queue里的数据。而且持久化可以跟生产者那边的confirm机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者ack了，所以哪怕是在持久化到磁盘之前，rabbitmq挂了，数据丢了，生产者收不到ack，你也是可以自己重发的。哪怕是你给rabbitmq开启了持久化机制，也有可能，就是这个消息写到了rabbitmq中，但是还没来得及持久化到磁盘上，结果不巧，此时rabbitmq挂了，就会导致内存里的一点点数据会丢失。

消费端弄丢了数据

rabbitmq如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了。rabbitmq认为你都消费了，这数据就丢了。这个时候得用rabbitmq提供的ack机制，简单来说，就是你关闭rabbitmq自动ack，可以通过一个api来调用就行，然后每次你自己代码里确保处理完的时候，再程序里ack一把。这样的话，如果你还没处理完，不就没有ack？那rabbitmq就认为你还没处理完，这个时候rabbitmq会把这个消费分配给别的consumer去处理，消息是不会丢的。

rabbitmq 保证数据的顺序性

rabbitmq保证数据的顺序性

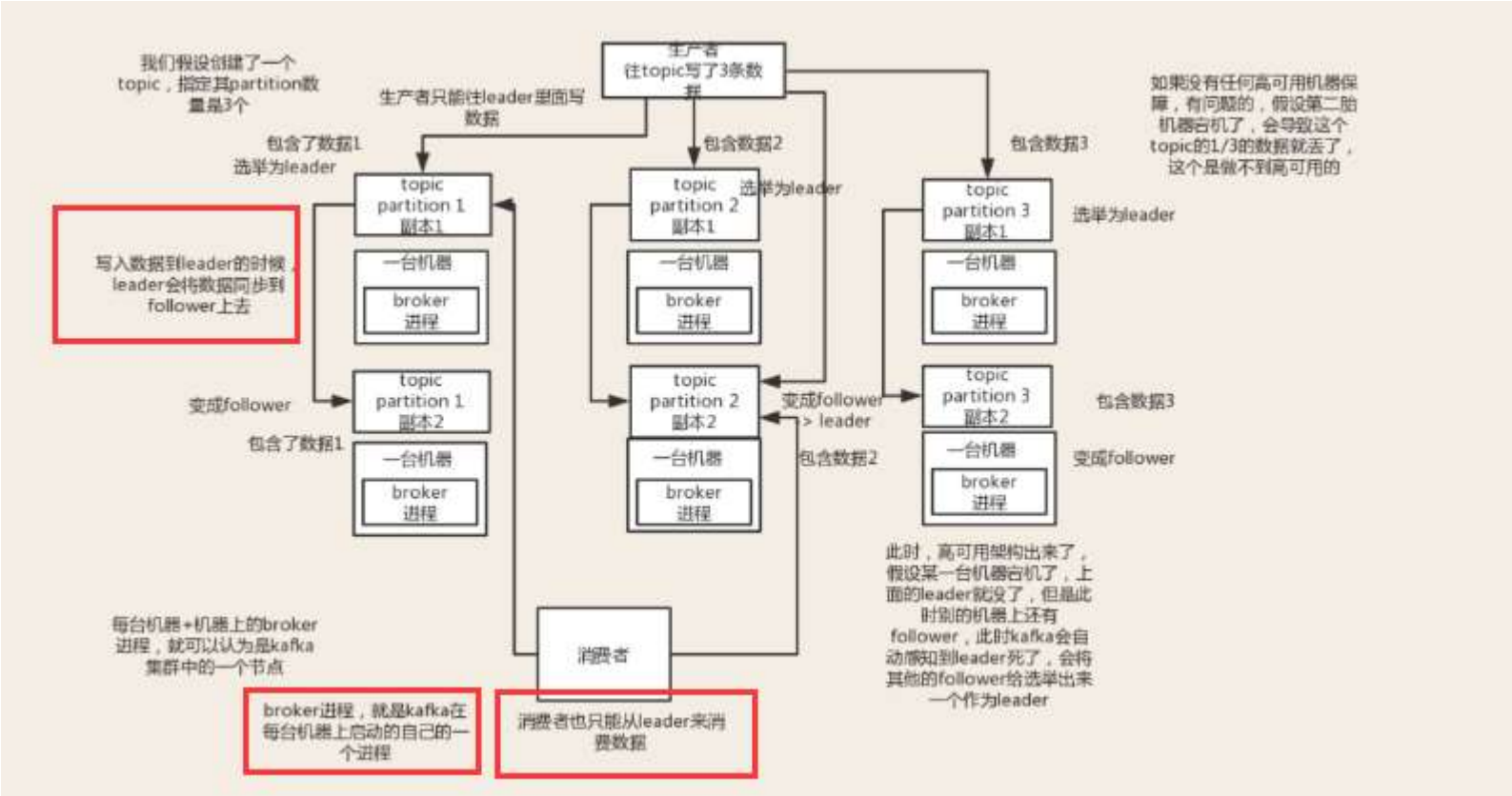
如果存在多个消费者，那么就让每个消费者对应一个queue，然后把要发送 的数据全都放到一个queue，这样就能保证所有的数据只到达一个消费者从而保证每个数据到达数据库都是顺序的。

rabbitmq：拆分多个queue，每个queue一个consumer，就是多一些queue而已，确实是麻烦点；或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理

kafaka

kafaka 的高可用

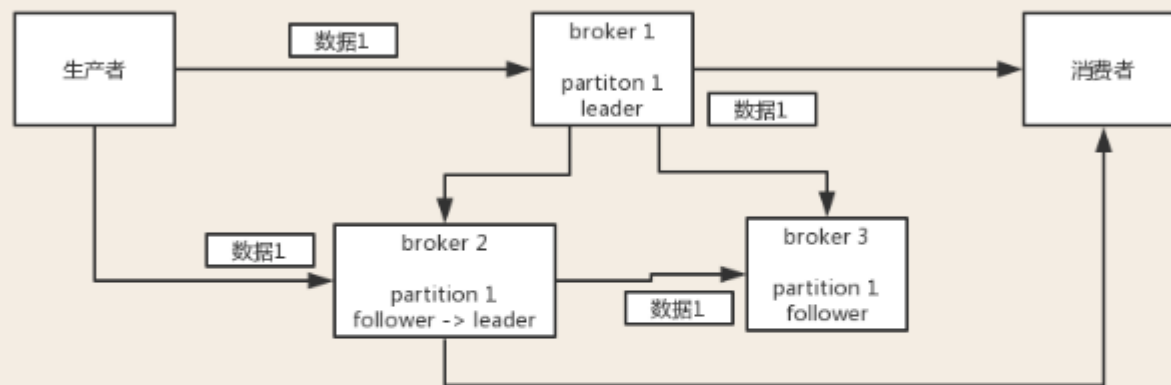
kafka一个最基本的架构认识：多个broker组成，每个broker是一个节点；你创建一个topic，这个topic可以划分为多个partition，每个partition可以存在于不同的broker上，每个partition就放一部分数据。这就是天然分布式消息队列，就是说一个topic的数据，是分散放在多个机器上的，每个机器就放一部分数据。实际上rabbitmq之类的，并不是分布式消息队列，他就是传统的消息队列，只不过提供了一些集群、HA的机制而已，因为无论怎么玩儿，rabbitmq一个queue的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个queue的完整数据。kafka 0.8以前，是没有HA机制的，就是任何一个broker宕机了，那个broker上的partition就废了，没法写也没法读，没有什么高可用性可言。kafka 0.8以后，提供了HA机制，就是replica副本机制。每个partition的数据都会同步到其他机器上，形成自己的多个replica副本。然后所有replica会选举一个leader出来，那么生产和消费都跟这个leader打交道，然后其他replica就是follower。写的时候，leader会负责把数据同步到所有follower上去，读的时候就直接读leader上数据即可。只能读写leader？很简单，要是你可以随意读写每个follower，那么就要care数据一致性的问题，系统复杂度太高，很容易出问题。kafka会均匀的将一个partition的所有replica分布在不同的机器上，这样才可以提高容错性。这么搞，就所谓的高可用性了，因为如果某个broker宕机了，没事儿，那个broker上面的partition在其他机器上都有副本的，如果这上面有某个partition的leader，那么此时会重新选举一个新的leader出来，大家继续读写那个新的leader即可。这就所谓的高可用性了。写数据的时候，生产者就写leader，然后leader将数据落地写本地磁盘，接着其他follower自己主动从leader来pull数据。一旦所有follower同步好数据了，就会发送ack给leader，leader收到所有follower的ack之后，就会返回写成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）消费的时候，只会从leader去读，但是只有一个消息已经被所有follower都同步成功返回ack的时候，这个消息才会被消费者读到。



kafaka 的消息丢失

消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你那个消费到了这个消息，然后消费者那边自动提交了offset，让kafka以为你已经消费好了这个消息，其实你刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。这不是一样么，大家都知道kafka会自动提交offset，那么只要关闭自动提交offset，在处理完之后自己手动提交offset，就可以保证数据不会丢。但是此时确实还是会重复消费，比如你刚处理完，还没提交offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。生产环境碰到的一个问题，就是说我们的kafka消费者消费到了数据之后是写到一个内存的queue里先缓冲一下，结果有的时候，你刚把消息写入内存queue，然后消费者会自动提交offset。然后此时我们重启了系统，就会导致内存queue里还没来得及处理的数据就丢失了



kafka弄丢了数据

这块比较常见的一个场景，就是kafka某个broker宕机，然后重新选举partition的leader时。大家想想，要是此时其他的follower刚好还有些数据没有同步，结果此时leader挂了，然后选举某个follower成leader之后，他不就少了一些数据？这就丢了一些数据啊。生产环境也遇到过，我们也是，之前kafka的leader机器宕机了，将follower切换为leader之后，就会发现说这个数据就丢了所以此时一般是要求起码设置如下4个参数：给这个topic设置replication.factor参数：这个值必须大于1，要求每个partition必须有至少2个副本在kafka服务端设置min.insync.replicas参数：这个值必须大于1，这个要求一个leader至少感知到有至少一个follower还跟自己保持联系，没掉队，这样才能确保leader挂了还有一个follower吧在producer端设置acks=all；这个是要求每条数据，必须是写入所有replica之后，才能认为是写成功了在producer端设置retries=MAX（很大很大很大的一个值，无限次重试的意思）；这个是要求一旦写入失败，就无限重试，卡在这里了我们生产环境就是按照上述要求配置的，这样配置之后，至少在kafka broker端可以保证在leader所在broker发生故障，进行leader切换时，数据不会丢失 3）生产者会不会弄丢数据如果按照上述的思路设置了ack=all，一定不会丢，要求是，你的leader接收到消息，所有的follower都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

kafka 保证消息的顺序性

kafka保证数据的顺序性

kafka 写入partition时指定一个key，例如订单id，那么消费者从partition中取出数据的时候肯定是有序的，当开启多个线程的时候可能导致数据不一致，这时候就需要内存队列，将相同的hash过的数据放在一个内存队列里，这样就能保证一条线程对应一个内存队列的数据写入数据库的时候顺序性的，从而可以开启多条线程对应多个内存队列（2）kafka：一个topic，一个partition，一个consumer，内部单线程消费，写N个内存queue，然后N个线程分别消费一个内存queue即可

保证消息的幂等性

kafka来举个例子，说说怎么重复消费吧。kafka实际上有个offset的概念，就是每个消息写进去，都有一个offset，代表他的序号，然后consumer消费了数据之后，每隔一段时间，会把自己消费过的消息的offset提交一下，代表我已经消费过了，下次我要是重启啥的，你就让我继续从上次消费到的offset来继续消费吧。但是凡事总有意料之外，比如我们之前生产经常遇到的，就是你有时候重启系统，看你怎么重启了，如果碰到点着急的，直接kill进程了，再重启。这会导致consumer有些消息处理了，但是没来得及提交offset，尴尬了。重启之后，少数消息会再次消费一次。其实重复消费不可怕，可怕的是你没考虑到重复消费之后，怎么保证幂等性。给你举个例子吧。假设你有个系统，消费一条往数据库里插入一条，要是你一条消息重复两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下已经消费过了，直接扔了，不就保留了一条数据？一条数据重复出现两次，数据库里就只有一条数据，这就保证了系统的幂等性幂等性，我通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，不能出错。

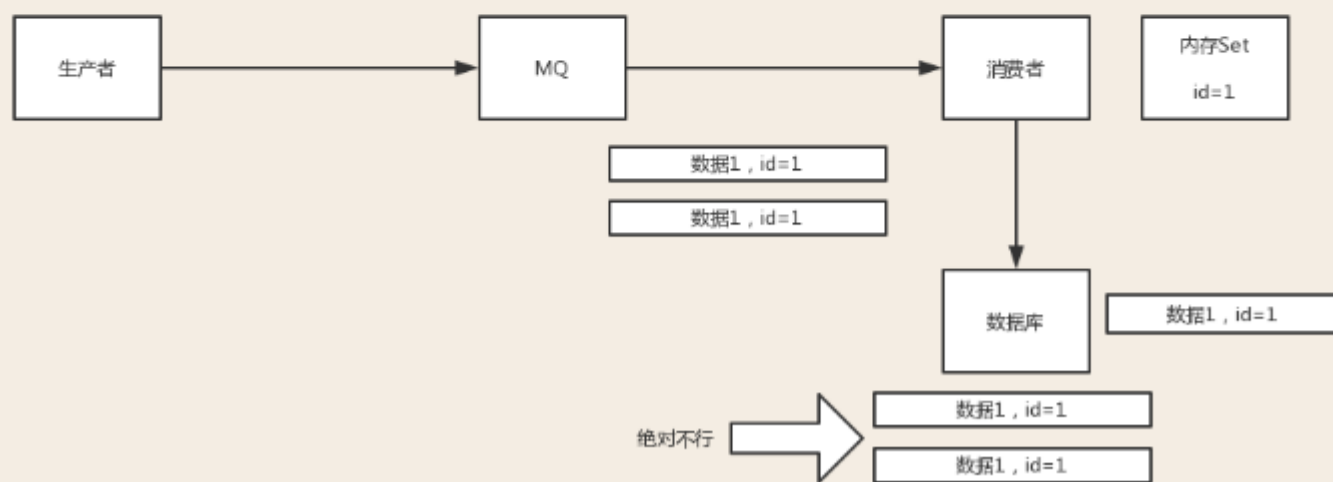
保证幂等性的思路

其实还是得结合业务来思考，我这里给几个思路：

1. 比如你拿个数据要写库 你先根据主键查一下，如果这数据都有了，你就别插入了，update一下好吧
2. 比如你是写redis，那没问题了 反正每次都是set，天然幂等性
3. 比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的id，类似订单id之类的东西，然后你这里消费到了之后，先根据这个id去比如redis里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个id写redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。

还有比如基于数据库的唯一键来保证重复数据不会重复插入多条，我们之前线上系统就有这个问题，就是拿到数据的时候，每次重启可能会有重复，因为kafka消费者还没来得及提交offset，重复数据拿到了以后我们插入的时候，因为有唯一键约束了，所以重复数据只会插入报错，不会导致数据库中出现脏数据

如何保证MQ的消费是幂等性的，需要结合具体的业务来看

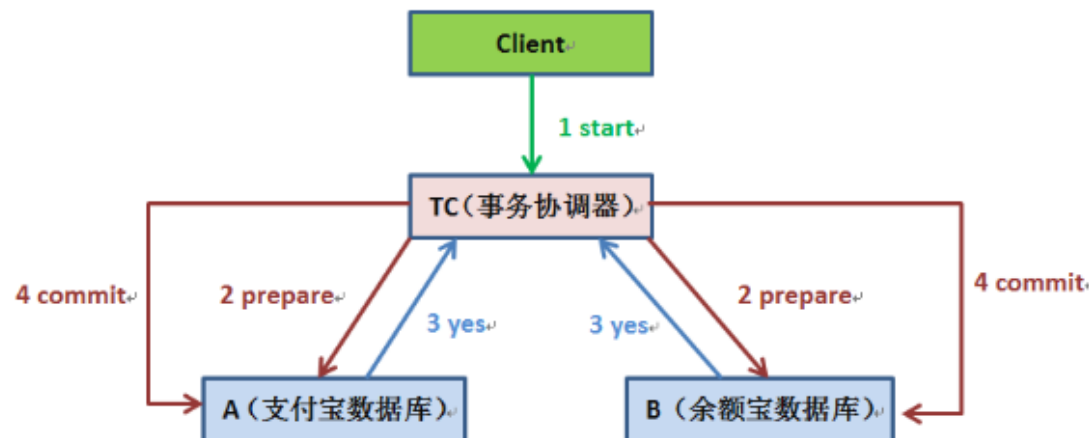


消息队列解决数据的最终一致性

两阶段提交协议

2 分布式事务—两阶段提交协议

两阶段提交协议（Two-phase Commit, 2PC）经常被用来实现分布式事务。一般分为协调器C和若干事务执行者Si两种角色，这里的事务执行者就是具体的数据库，协调器可以和事务执行器在一台机器上。



1) 我们的应用程序 (client) 发起一个开始请求到TC;

2) TC先将<prepare>消息写到本地日志, 之后向所有的Si发起<prepare>消息。以支付宝转账到余额宝为例, TC给A的prepare消息是通知支付宝数据库相应账目扣款1万, TC给B的prepare消息是通知余额宝数据库相应账目增加1w。为什么在执行任务前需要先写本地日志, 主要是为了故障后恢复用, 本地日志起到现实生活中凭证 的效果, 如果没有本地日志 (凭证), 容易死无对证;

3) Si收到<prepare>消息后, 执行具体本机事务, 但不会进行commit, 如果成功返回<yes>, 不成功返回<no>。同理, 返回前都应把要返回的消息写到日志里, 当作凭证。

4) TC收集所有执行器返回的消息, 如果所有执行器都返回yes, 那么给所有执行器发送commit消息, 执行器收到commit后执行本地事务的commit操作; 如果有任一个执行器返回no, 那么给所有执行器发送abort消息, 执行器收到abort消息后执行事务abort操作。

注: TC或Si把发送或接收到的消息先写到日志里, 主要是为了故障后恢复用。如某一Si从故障中恢复后, 先检查本机的日志, 如果已收到<commit>, 则提交, 如果<abort>则回滚。如果是<yes>, 则再向TC询问一下, 确定下一步。如果什么都没有, 则很可能在<prepare>阶段Si就崩溃了, 因此需要回滚。

现如今实现基于两阶段提交的分布式事务也没那么困难了, 如果使用java, 那么可以使用开源软件atomikos(<http://www.atomikos.com/>)来快速实现。

不过但凡使用过的上述两阶段提交的同学都可以发现性能实在是太差, 根本不适合高并发的系统。为什么?

1) 两阶段提交涉及多次节点间的网络通信, 通信时间太长!

2) 事务时间相对于变长了, 锁定的资源的时间也变长了, 造成资源等待时间也增加好多!

正是由于分布式事务存在很严重的性能问题, 大部分高并发服务都在避免使用, 往往通过其他途径来解决数据一致性问题。

使用消息队列来避免分布式事务

3 使用消息队列来避免分布式事务

如果仔细观察生活的话, 生活的很多场景已经给了我们提示。

比如在北京很有名的姚记炒肝点了炒肝并付了钱后, 他们并不会直接把你点的炒肝给你, 往往是给你一张小票, 然后让你拿着小票到出货区排队去取。为什么他们要将付钱和取货两个动作分开呢? 原因很多, 其中一个很重要的原因是为了使他们接待能力增强 (并发量更高)。

还是回到我们的问题, 只要这张小票在, 你最终是能拿到炒肝的。同理转账服务也是如此, 当支付宝账户扣除1万后, 我们只要生成一个凭证 (消息) 即可, 这个凭证 (消息) 上写着 “让余额宝账户增加 1万”, 只要这个凭证 (消息) 能可靠保存, 我们最终是可以拿着这个凭证 (消息) 让余额宝账户增加1万的, 即我们能依靠这个凭证 (消息) 完成最终一致性。

坑

这里我们假设再来第二个坑

假设你用的是rabbitmq，rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间就会被rabbitmq给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在mq里，而是大量的数据会直接搞丢。这个情况下，就不是说要增加consumer消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入mq里面去，把白天丢的数据给他补回来。也只能是这样了。假设1万个订单积压在mq里面，没有处理，其中1000个订单都丢了，你只能手动写程序把那1000个订单给查出来，手动发到mq里去再补一次

然后我们再来假设第三个坑

如果走的方式是消息积压在mq里，那么如果你很长时间都没处理掉，此时导致mq都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

Kafka，ActiveMQ,RabbitMQ,ROcketMQ 的优缺点

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比RocketMQ、Kafka 低一个数量级	同ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	基本不丢	经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

https://blog.csdn.net/baidu_2695452

经验

2、服务编排是个好东西，主要的作用是减少项目中的相互依赖。比如现在有项目a调用项目b，项目b调用项目c...一直到h，是一个调用链，那么项目上线的时候需要先更新最底层的h再更新g...更新c更新b最后是更新项目a。这只是这一个调用链，在复杂的业务中有非常多的调用，如果要记住每一个调用链对开发运维人员来说就是灾难。

有这样一个好办法可以尽量的减少项目的相互依赖，就是服务编排。一个核心的业务处理项目，负责和各个微服务打交道。比如之前是a调用b，b调用c，c调用d，现在统一在一个核心项目W中来处理，W服务使用a的时候去调用b，使用b的时候W去调用c，举个例子：在第三方支付业务中，有一个核心支付项目是服务编排，负责处理支付的业务逻辑，W项目使用商户信息的时候就去调用“商户系统”，需要校验设备的时候就去调用“终端系统”，需要风控的时候就调用“风控系统”，各个项目需要的依赖参数都由W来做主控。以后项目部署的时候只需要最后启动服务编排项目即可。

3、不要为了追求技术而追求技术，确定进行微服务架构改造之前，需要考虑以下几方面的因素：

- 1) 团队的技术人员是否已经具备相关技术基础。
- 2) 公司业务是否适合进行微服务化改造，并不是所有的平台都适合进行微服务化改造，比如：传统行业有很多复杂垂直的业务系统。
- 3) Spring Cloud生态的技术有很多，并不是每一种技术方案都需要用上，适合自己的才是最好的。