总的目录：

https://www.processon.com/view/link/60fb9421637689719d246739

功能分离

why

确保核心功能的高可用和高并发。

通过对系统业务的仔细分析，进行功能的分离

按照功能的重要程度

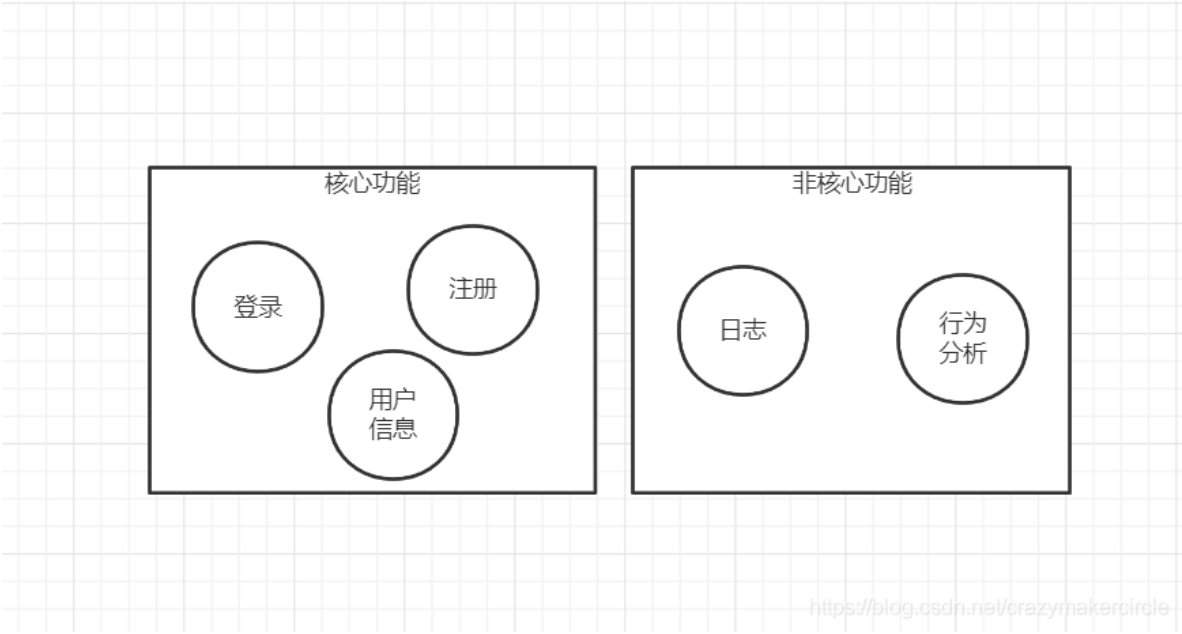
按照功能的重要程度，划分核心功能和非核心功能，将核心功能和非核心功能物理隔离

这里有两个关键点：

首先要区分核心功能和非核心功能。

例如，在亿级规模的用户中台系统中，假设有 注册、登录、用户信息、日志、行为分析 等功能。 问题是：哪一个更重要。

例如用户中台系统中，对于一个亿级用户，日活2000万的业务来说，平均每天注册用户可能是10万左右（假设2年），修改用户信息的可能还不到1万，但登录功能是2000万，很明显我们应该保证登录的才是核心。



登录是核心功能，注册、用户信息是非核心功能。登录功能一旦有问题，其他的业务系统，就不能登录了；而非核心功能即使有问题，暂时也不会立刻影响业务系统的使用。因此，优先保证核心功能正常，是我们首要的目标。

其次要核心功能和非核心功能，有不同的应对策略：

隔离策略、重试策略、功能降级策略

重要程度，仅仅是功能分离的一种维度。还可以按照其他维度进行划分，比如流量特点，

还可以按照其他维度进行划分，比如流量特点.

按照功能的流量特点.

在秒杀系统中，这里需要区分。可以区分为流量突发型、流量平缓型的功能，对突发流量的功能做好隔离。

电商平台： 秒杀功能、电商功能

流量突发功、流量平缓型的应对策略：

首先，做好隔离策略，

另外，对突发流量的功能做好独立的伸缩扩展策略。

功能分离之后的应对：

功能隔离

如何隔离：

单独的域名，单独的接入层、隔离的服务层、单独的缓存，单独的数据库

域名隔离、 代理隔离、微服务隔离、缓存隔离、数据库隔离

只要核心功能和非核心功能存在共享的资源，就有可能因为非核心功能影响核心功能。

举个最简单的例子，如果数据库共用一套，那么非核心功能如果出现了大量的整表查询（慢sql），核心功能同样受到影响。

只要流量突发型、流量平缓型功能存在共享的资源，就有可能影响流量平缓型功能。

假设：核心功能、非核心功能共享了缓存服务器，就可能会由于非核心功能的操作影响了缓存的性能，甚至出现问题.

解决方案：缓存物理隔离后，就更加保证了核心功能的安全，

功能降级：

当出现故障的时候，当出现瓶颈的时候，可以将非核心功能直接降级，保护核心功能不受影响

拆分为核心功能和非核心功能后，虽然物理上两者隔离了，但有的业务还是需要核心功能和非核心功能配合才能完成，这就存在了一定的风险。

比如说大量用户登录时，可以停止行为分析、登录日志 等非核心功能。以保证核心功能不受影响。

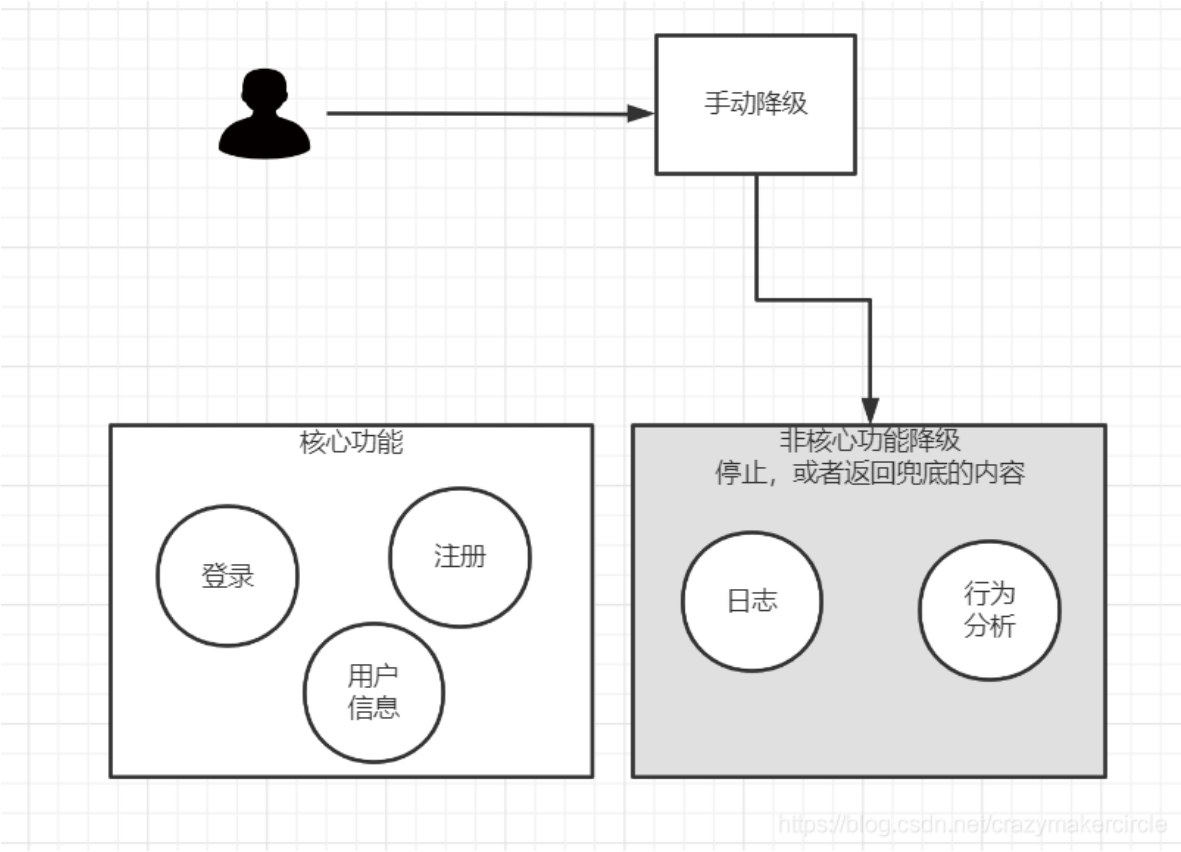
降级的实现方式通常有**手动**和**自动**
自动方式是程序调用发生问题时，自动降级，如调用某服务时，响应时间超过预订阀值，自动降级

微服务的熔断，就属于自动降级

手动方式是使用配置中心，对系统中可降级的服务都设置好开关项，当需要降级时，在配置中心中进行操作，配置中心进行下发变更通知

、

可以开发了一个后台**运维管理程序**，当需要停用某个功能的时候，只需要在后台上点击一个按钮就能够完成，花费时间只需要几秒钟。



系统分层

常见互联网分层架构

常见互联网分层架构，分为：

（1）**客户端层：**客户端层是浏览器browser或者手机应用APP；

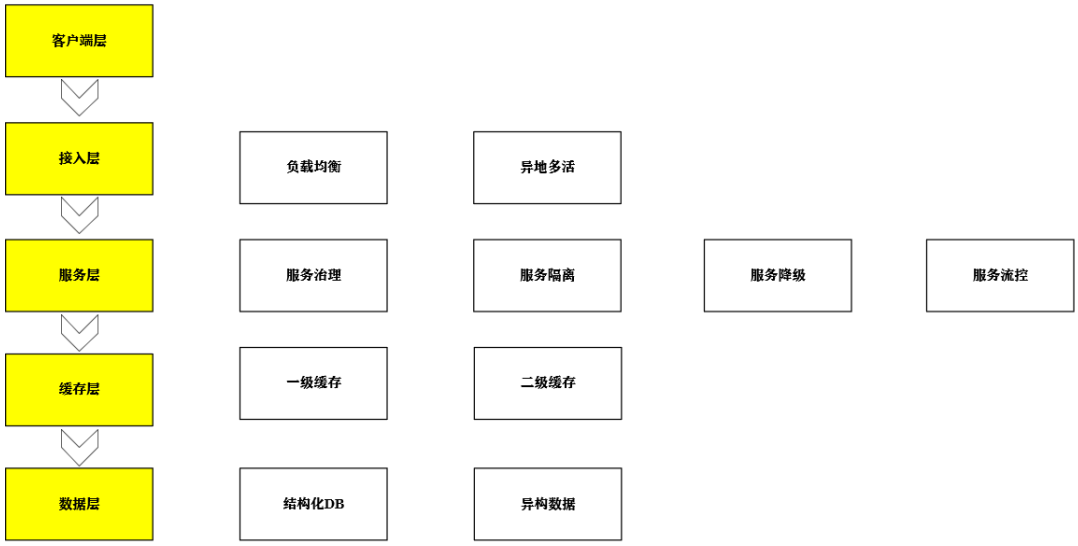
（2）**接入层**：系统入口，负载均衡、反向代理；web服务； dns、cdn

（3）**服务层**：实现核心应用逻辑，返回json或者html

（4）**缓存层**：缓存加速访问存储；

（5）**数据库层**：结构化db和异构db；

（6）**中间件：** zk、xxl-job，rocketmq



参考：亿级流量秒杀的分层设计方案

https://www.processon.com/view/link/61148c2b1e08536191d8f92f

客户层

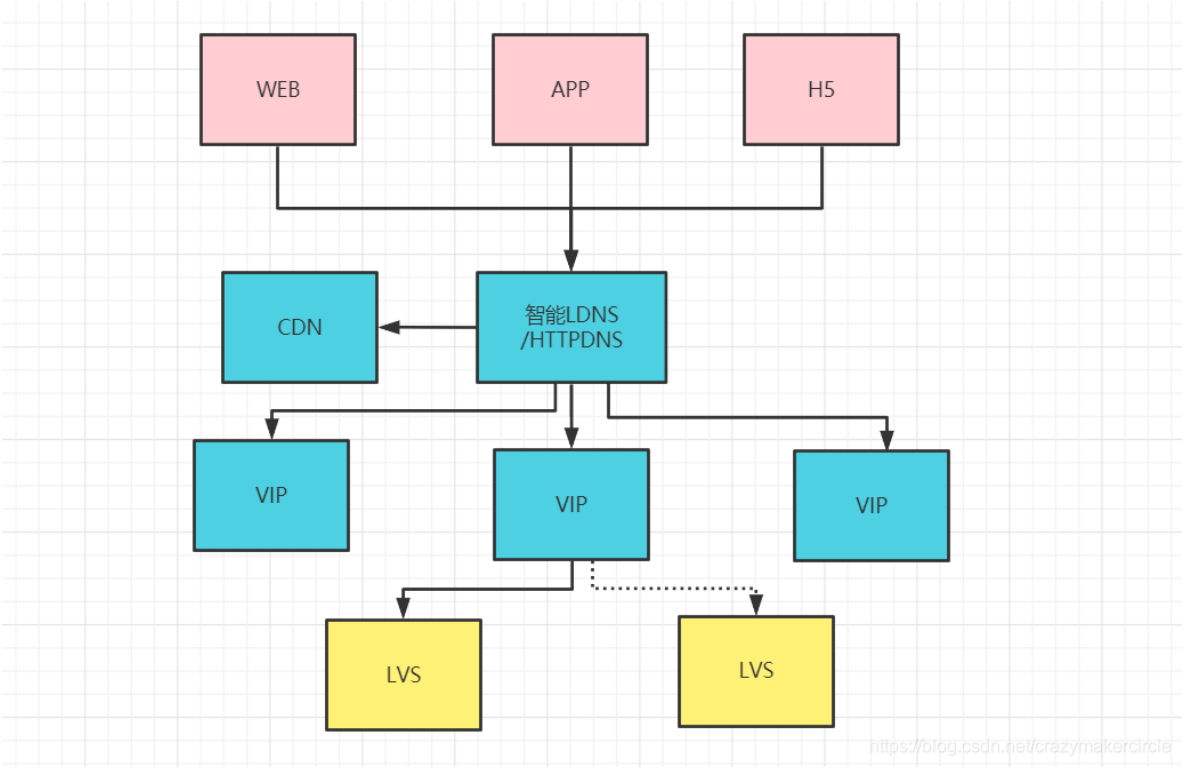
客户层：支持PC浏览器、手机APP、H5页面。

差别是手机APP可以直接访问通过IP访问（而不仅仅是域名）访问接入层服务器。

接入层

流量分发、负载均衡

按照用户规模，流量规模（吞吐量规模），接入层的架构方案不一样



服务层

服务层：提供公用服务，比如用户服务，订单服务，支付服务等；

公共基础能力

服务治理，统一配置，统一监控

缓存层：

> 请求大致分为两类： 读请求、写请求

多级缓存：整个系统架构的不同系统层级进行数据缓存，以提升读取的效率。

Tomcat堆缓存（一级）、分布式缓存（二级）、Nginx本地缓存（三级）。

HTTP缓存：根据服务器端返回的缓存设置响应头将响应内容缓存到浏览器。减少浏览器端和服务器端之间来回传输数据量，节省带宽。

数据层：

包含结构化数据（关系型）数据库集群（支持读写分离）

还包含异构数据（非关系型、NOSQL）集群，

还包含分布式文件系统集群；

1. 大数据存储层：支持应用层和服务层的日志数据收集，关系数据库和NOSQL数据库的结构化和半结构化数据收集；
2. 大数据处理层：通过Mapreduce进行离线数据分析或Storm实时数据分析，并将处理后的数据存入关系型数据库。

亿级用户的请求分层过滤模型

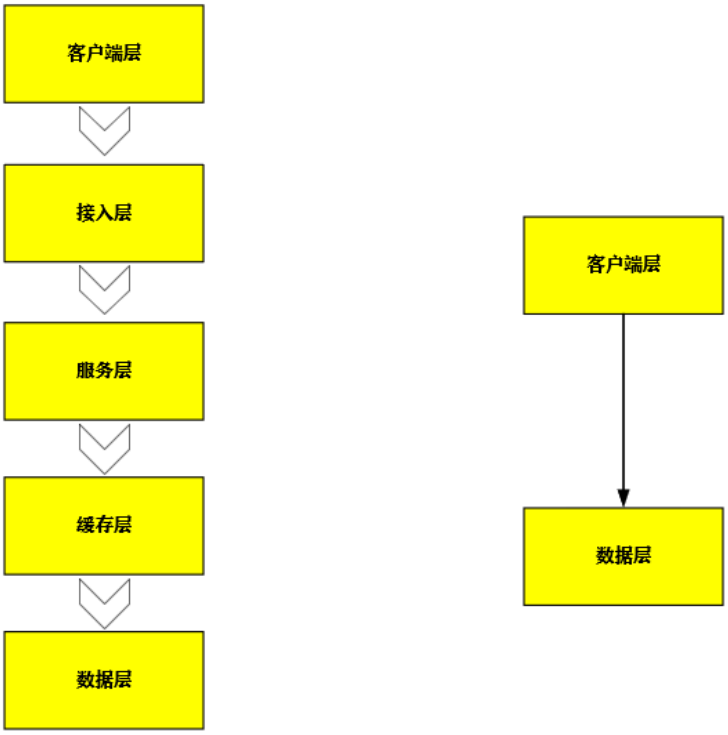
基础：请求处理模型

直筒型

直筒型请求处理模型，指的是用户请求 1:1 的洞穿到 db 层，如下图所示。

在比较简单的业务中，才会采用这个模型。

传统的 低并发、低性能、低可用项目。



直筒型请求处理模型的适用场景：

企业级应用。

特点：

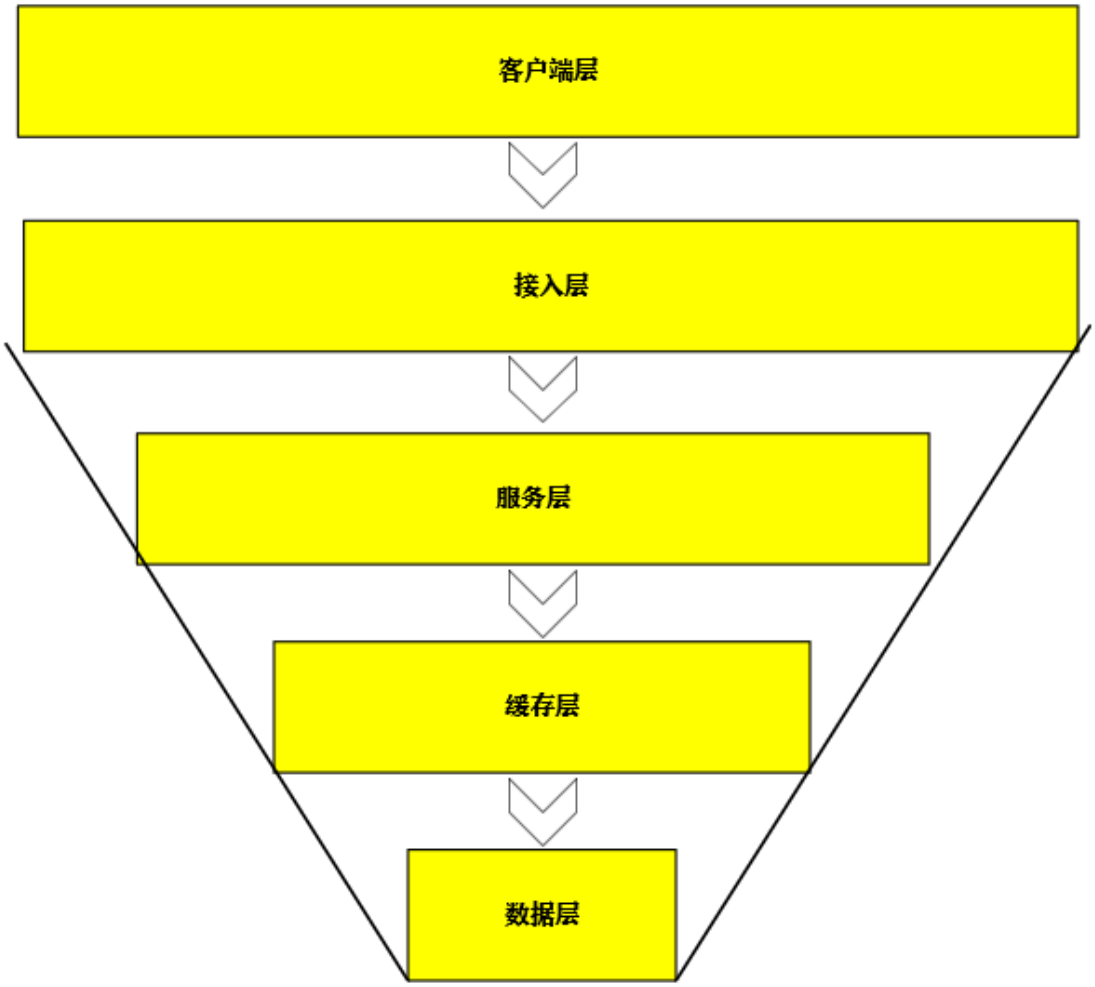
1. 用户规模较小

2. 请求峰值和平均值相差不大
3. 请求峰值不会超过数据层的处理能力

漏斗型

漏斗型业务，指的是，用户的请求，从客户端到 db 层，层层递减，递减的程度视业务而定。例如当10w 人去抢 1 00个物品时，db 层的请求在个位数量级 1000以内，这就是比较理想的模型。

如下图所示



漏斗型请求处理模型的适用场景：

互联网应用（如秒杀）。

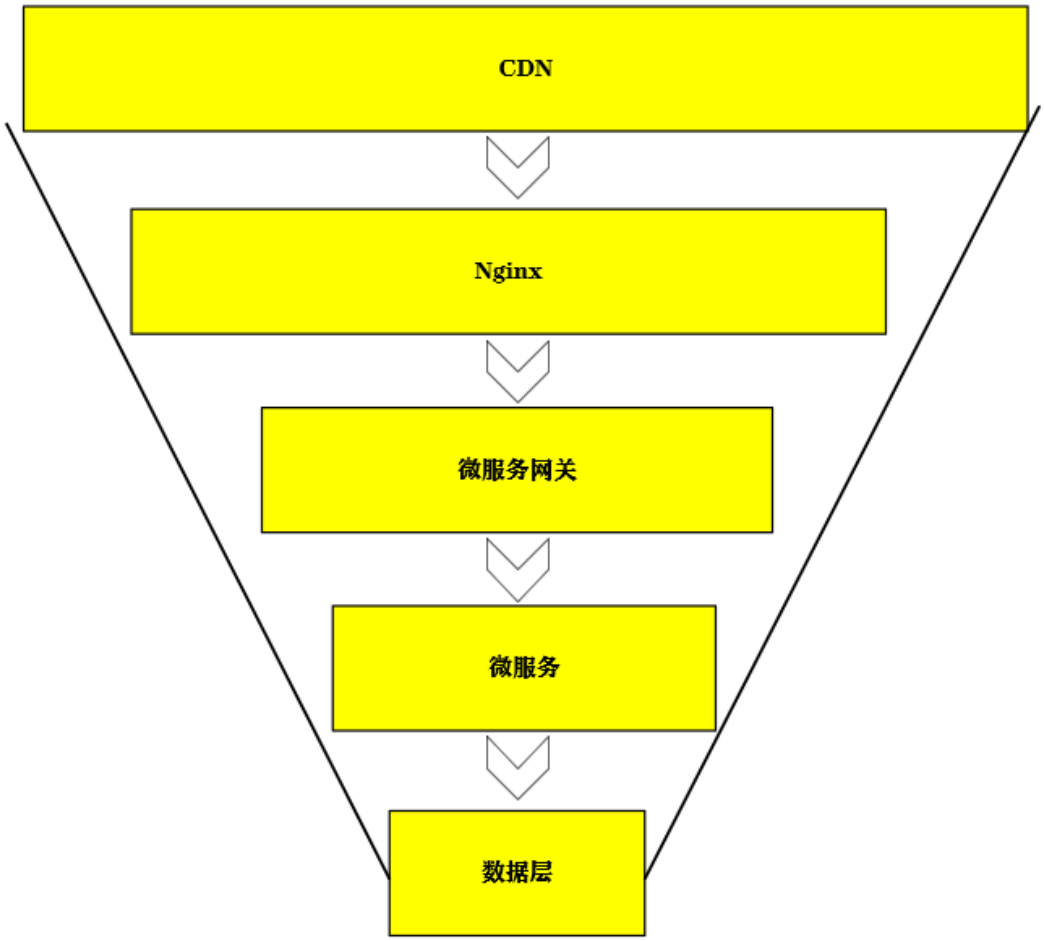
特点：

1. 用户规模大
2. 请求峰值和平均值相差巨大
3. 请求峰值远远超出最后一层（数据层）的处理能力

漏斗模型中的请求分层过滤

漏斗型请求处理模型的核心策略，对请求进行 **分层过滤。**

而针对漏斗型请求处理模型（如秒杀场景）一 种核心策略，就是对请求进行分层过滤，从而过滤掉一些无效的请求。



案例：秒杀系统的分层过滤

比如,在秒杀系统中，请求分别经过 CDN、Nginx（商品详情）、微服务（如交seckill）和数据库 这几层，那么：

1 大部分数据和流量在用户浏览器或者 CDN 上获取，这一层可以拦截大部分静态资源的读取；

2、经过第二层 Nginx（商品详情）时，尽量得走 Nginx Cache，过滤一些可以直接访问Nginx缓存的请求；

经过第二层 Nginx（商品详情）时，也可以进行流控，还可以进行黑名单过滤，拦截掉一些无效的流量；

3、再到服务层，进入微服务网关时，可以做用户的授权检验，对系统做好保护和限流，这样数据量和请求就 进一步减少；

4、业务层，还可以进行数据的有效性、一致性过滤，这里又减少了一些流量。

这样就像漏斗一样，尽量把数据量和请求量一层一层地过滤和减少了。

分层过滤的核心思想是：在不同的层次尽可能地过滤掉无效请求，让“漏斗”最末端的才是有效 请求。而要达到这种效果，我们就必须对数据做分层的校验。

分层过滤的基本原则是：

通过在不同的层次尽可能地过滤掉无效请求， 尽早处理掉请求。

- 通过CDN过滤掉大量的图片，静态资源的请求。
- 读请求尽量命中缓存，不要穿透到数据库；
- 尽量将动态读数据请求，命中在三级缓存,或者二级缓存，过滤掉无效的数据读；
- 对写入操作进行削峰，争取批量写入，提高写入的吞吐量；
- 分层限流：防止系统雪崩

写入操作的流量削峰方案(降级方案)

削峰从本质上来说，就是更多地延缓用户请求，以及层层过滤用户的访问需求，遵从【最后落地到数据库的请求数要尽量少】的原则。

同步的直接调用转换成异步的间接推送

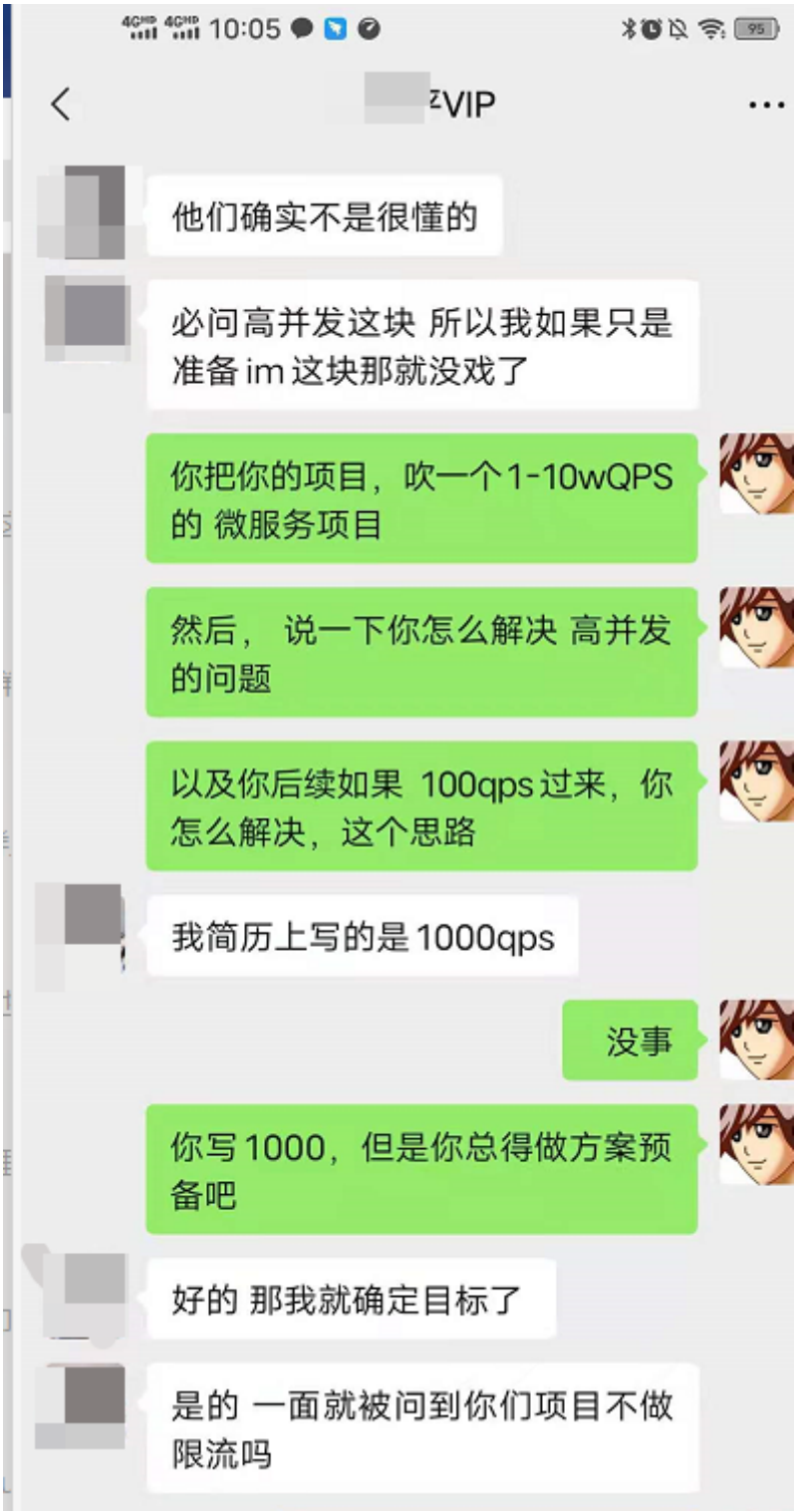
要对流量进行削峰，最容易想到的解决方案就是用消息队列来缓冲瞬时流量，把同步的直接调用转换成异步的间接推送，中间通过一个队列在一端承接瞬时的流量洪峰，在另一端平滑地将消息推送出去。

消息队列中间件主要解决应用耦合、异步消息、流量削峰等问题。常用的消息队列系统有ActiveMQ、RabbitMQ、ZeroMQ、Kafka、Me'taMQ和RocketMQ等。

在这里，消息队列就像是水库一样，拦截上游的洪水，削减进入下游河道的洪峰流量，从而达到减免洪水灾害的目的。

分层限流：防止系统雪崩的无奈策略

> 分层限流：接入层、服务层 限流，根据各层的能力，对系统进行保护



分层架构的幂等性原则

幂等性的定义：

所谓幂等性通俗的将就是一次请求和多次请求同一个资源产生相同的副作用。

通俗点说：

对于以相同的请求调用这个接口一次或多次，需要给调用方返回一致的结果时，就要考虑将这个接口设计成幂等接口。

用数学语言表达就是`f(x)=f(f(x))`。

- 维基百科的幂等性定义如下：

> 幂等（idempotent、idempotence）是一个数学与计算机学概念，常见于抽象代数中。
> 在编程中一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。幂等函数，或幂等方法，是指可以使用相同参数重复执行，并能获得相同结果的函数。这些函数不会影响系统状态，也不用担心重复执行会对系统造成改变。例如，“setTrue()”函数就是一个幂等函数,无论多次执行，其结果都是一样的，更复杂的操作幂等保证是利用唯一交易号(流水号)实现.

为什么需要幂等性

在系统高并发的环境下，很有可能因为网络，阻塞等等问题导致客户端或者调用方并不能及时的收到服务端的反馈甚至是调用超时的问题。总之，就是请求方调用了你的服务，但是没有收到任何的信息，完全懵逼的状态。比如订单的问题，可能会遇到如下的几个问题：

1. 创建订单时，第一次调用服务超时，再次调用是否产生两笔订单？
2. 订单创建成功去减库存时，第一次减库存超时，是否会多扣一次？

幂等性要求，幂等性通俗的将就是一次请求和多次请求同一个资源产生相同的副作用。

所以： 创建订单时，第一次调用服务超时，再次调用是否产生两笔订单？ 当然不会。

所以： 创建订单时，同一个订单的两次扣减的操作， 是否会多扣一次？ 当然不会。

需要幂等场景：

可能会发生重复请求或消费的场景，在分布式、微服务架构中是随处可见的。

- 网络波动：因网络波动，可能会引起重复请求
- 分布式消息消费：任务发布后，使用分布式消息服务来进行消费
- 用户重复操作：用户在使用产品时，可能无意地触发多笔交易，甚至没有响应而有意触发多笔交易
- 未关闭的重试机制：因开发人员、测试人员或运维人员没有检查出来，而开启的重试机制（如Nginx重试、RPC通信重试或业务层重试等）

做到幂等性的几个例子

服务方提供一个查询操作是否成功的api，第一次超时之后，调用方调用查询接口，如果查到了就走成功的流程，失败了就走失败的流程。

1. 用户礼包领取

我们都知道一个用户新注册的时候，系统都会送该用户一份新用户大礼包，当我们点击领取这个礼包之后，我们就相当于收下了这份礼包，以后无论你怎么点击领取，结果还是一样，系统只会提示你已经领取过该礼包了，不会再让你重复领取一次。

2. 抢红包

当我们在抢一份红包的时候，我们点击了抢，抢到就有，没抢到就没有，之后，无论我们重复点击多少次，红包都会提示你已经抢过该红包了。

3. 账单付款

当我们要结账的时候，支付平台会生成唯一的支付连接，不会再次生成另外的支付连接。(不能因为这个支付接口被调了两次就创建两个一样的订单。)

微服务幂等性

分布式系统中的幂等性概念：用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。

CRUD操作的天然幂等性分析

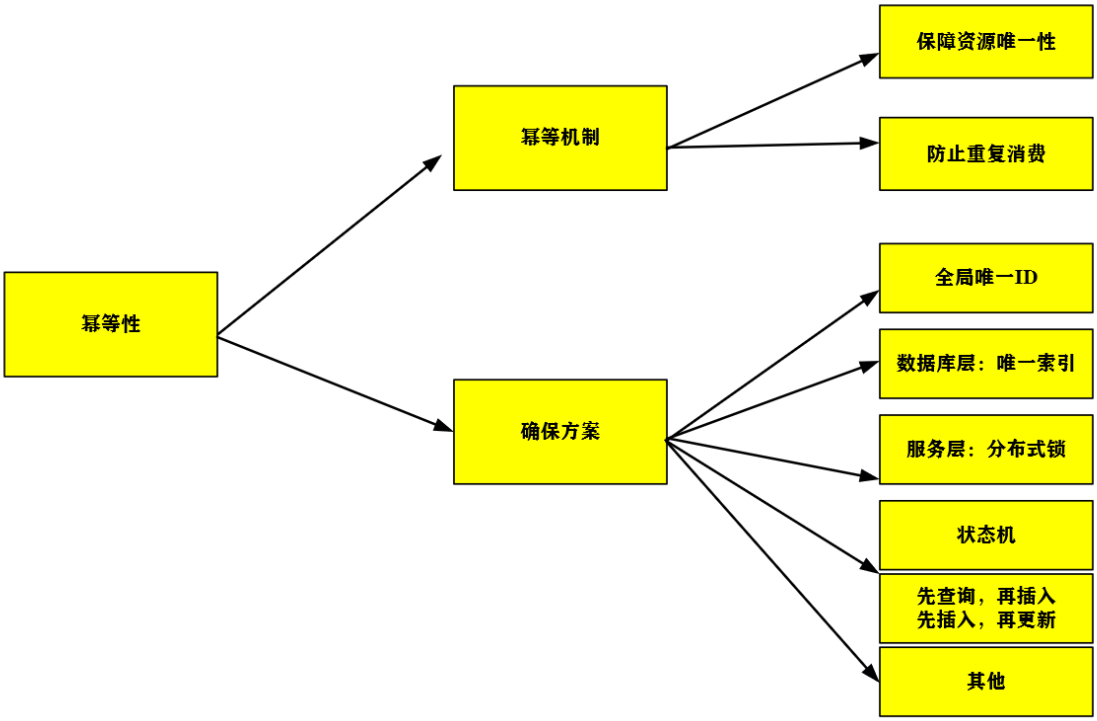
- 新增类请求：不具备幂等性
- 查询类动作：重复查询不会产生或变更新的数据，查询具有天然幂等性
- 更新类请求：
  - 基于主键的计算式Update，不具备幂等性，即`UPDATE goods SET number=number-1 WHERE id=1`
  - 基于主键的非计算式Update：具备幂等性，即`UPDATE goods SET number=newNumber WHERE id=1`
  - 基于条件查询的更新，不一定具备幂等性（需要根据实际情况进行分析判断）
- 删除类请求：
  - 基于主键的Delete具备幂等性
  - 一般业务层面都是逻辑删除（即update操作），而基于主键的逻辑删除操作也是具有幂等性的

幂等性的重要性

针对一个微服务架构，如果不支持幂等操作，那将会出现以下情况：

- 电商超卖现象
- 重复转账、扣款或付款
- 重复增加金币、积分或优惠券

如何保证幂等呢？



幂等性解决方案

- **全局唯一ID**

如果使用全局唯一ID，就是**根据业务的操作和内容生成一个全局ID**，在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。如果不存在则把全局ID，存储到存储系统中，比如数据库、Redis等。如果存在则表示该方法已经执行。

使用全局唯一ID是一个通用方案，可以支持插入、更新、删除业务操作。但是这个方案看起来很美但是实现起来比较麻烦，下面的方案适用于特定的场景，但是实现起来比较简单。

一般情况下，对分布式的全局唯一id，可以参考以下几种方式：

- UUID
- Snowflake
- 数据库自增ID
- 业务本身的唯一约束
- 业务字段+时间戳拼接

- **唯一索引(去重表)**

这种方法适用于在业务中有唯一标识的插入场景中，比如在以上的支付场景中，如果一个订单只会支付一次，所以**订单ID可以作为唯一标识。**这时，我们就可以建一张去重表，并且把唯一标识作为唯一索引，在我们实现时，把创建支付单据写入去重表，放在一个事务中，**如果重复创建，数据库会抛出唯一约束异常**，操作就会回滚。

- **插入或更新（upsert）**

这种方法插入并且有唯一索引的情况，比如我们要关联商品品类，其中商品的ID和品类的ID可以构成唯一索引，并且在数据表中也增加了唯一索引。这时就可以使用InsertOrUpdate操作。

- **多版本控制**

这种方法适合在更新的场景中，比如我们要更新商品的名字，这时我们就可以在更新的接口中增加一个版本号，来做幂等：`boolean updateGoodsName(int id,String newName,int version);`

在实现时可以如下：`update goods set name=#{newName},version=#{version} where id=#{id} and version<${version}`

- **状态机控制**

这种方法适合在有状态机流转的情况下，比如就会订单的创建和付款，订单的付款肯定是在之前，这时我们可以通过在设计状态字段时，使用int类型，并且通过值类型的大小来做幂等，比如订单的创建为0，付款成功为100，付款失败为99。在做状态机更新时，我们就这可以这样控制：`update goods_order set status=#{status} where id=#{id} and status<#{status}`

以上就是保证接口幂等性的一些方法。

总结

幂等性设计不能脱离业务来讨论，

单元化

单元化, 顾名思义, 用多AZ(可用区）,IDC（数据中心）机房 的实例来提供服务。

什么是可用区

区域（Region）和可用区（Availability Zone），主要来描述数据中心（IDC）的位置，您可以在特定的区域、可用区创建资源。

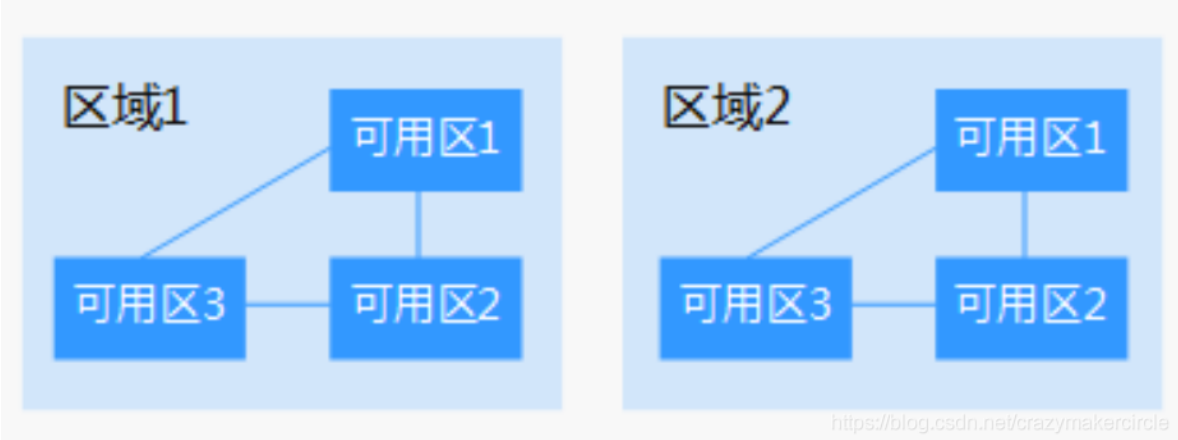
区域指物理的数据中心。

每个区域完全独立，这样可以实现最大程度的容错能力和稳定性。资源创建成功后不能更换区域。

可用区是同一区域内，电力和网络互相隔离的物理区域，一个可用区不受其他可用区故障的影响。

一个区域内可以有多个可用区，不同可用区之间物理隔离，但内网互通，既保障了可用区的独立性，又提供了低价、低时延的网络连接。

区域（Region）和可用区（Availability Zone）之间的关系，如下图：



比如，华为云已在全球多个地域开放云服务，可以根据需求选择适合自己的区域和可用区。

为什么要有单元化

现在稍微有点体量的公司都在做单元化，那为什么要做单元化的呢，总结而言有以下几个原因

1. 多AZ(可用区) 容灾。

2. 服务器体量太大，单一IDC没有足够的机器。

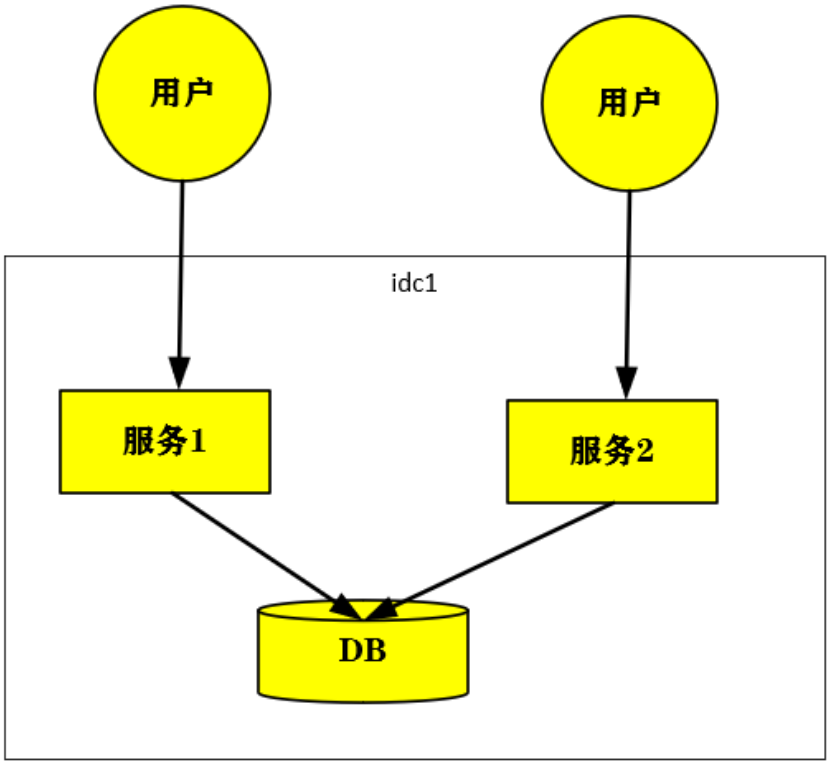
3. 提升用户请求访问速度。

单元化演进：

容灾是单元化最重要的应用场景之一，这里我们先聊聊容灾的演进史。借此引入单元化的话题。

单IDC时代

当一个网站刀耕火种之际，往往是多机服务 + 单mysql DB运行，这样我们一个服务迅速上线了。



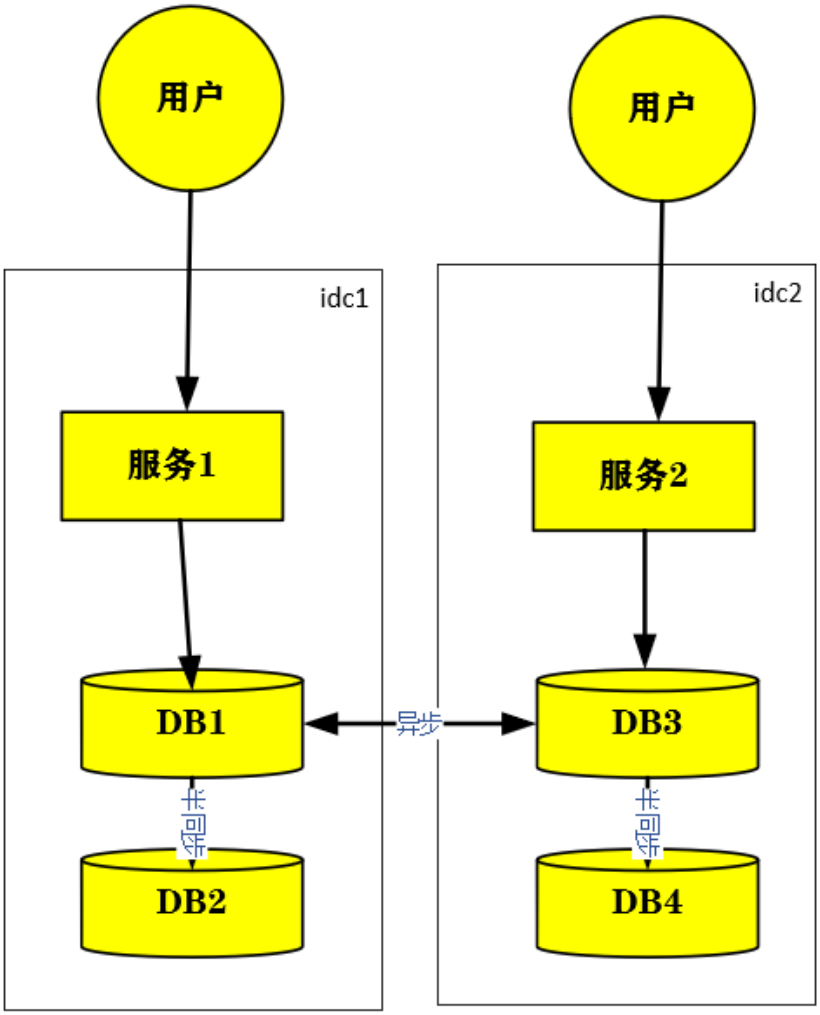
多机房容灾

直到有一天，网站IDC的光纤被挖断了 ，网站停止了服务，连续可用天数戛然而止。在开了无数次会议之后，网站决定开启多机房，实现机房级容灾。

同一个idc机房的DB，开启了mysql 半同步机制，并设置半同步超时报警。

不同一个idc机房的主DB，开启了异步复制。

对于跨idc机房的数据访问，至少有一个数据异步复制完成，才能进行。牺牲一部分可用性换取强一致性

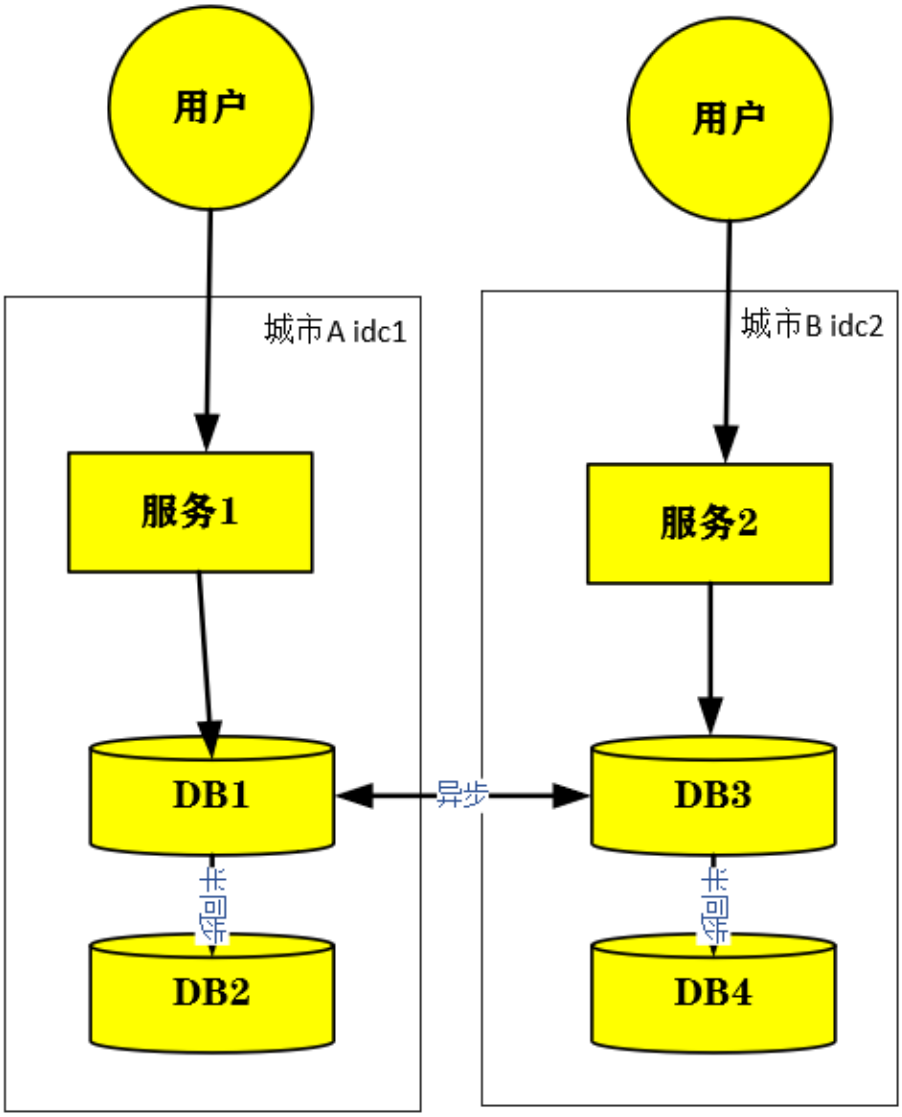


IDC进行数据异步同步。如果DB1出问题，两数据库瞬间切到IDB3，对于idc1 来说，如果数据没有复制完， 中间有一段时间不可用。

当DB1恢复时, 再把没同步的数据和DB3进行合并，按最后发生为准。(update 可能被覆盖，delete 未被更改的话有效，insert判断有无覆盖，当然也少不了人工对账)

异地多活

炎热的夏季让整个**城市停止了电力供应**。网站又陷入了停滞，为了应对新的问题，城市级容灾的方案出现了。



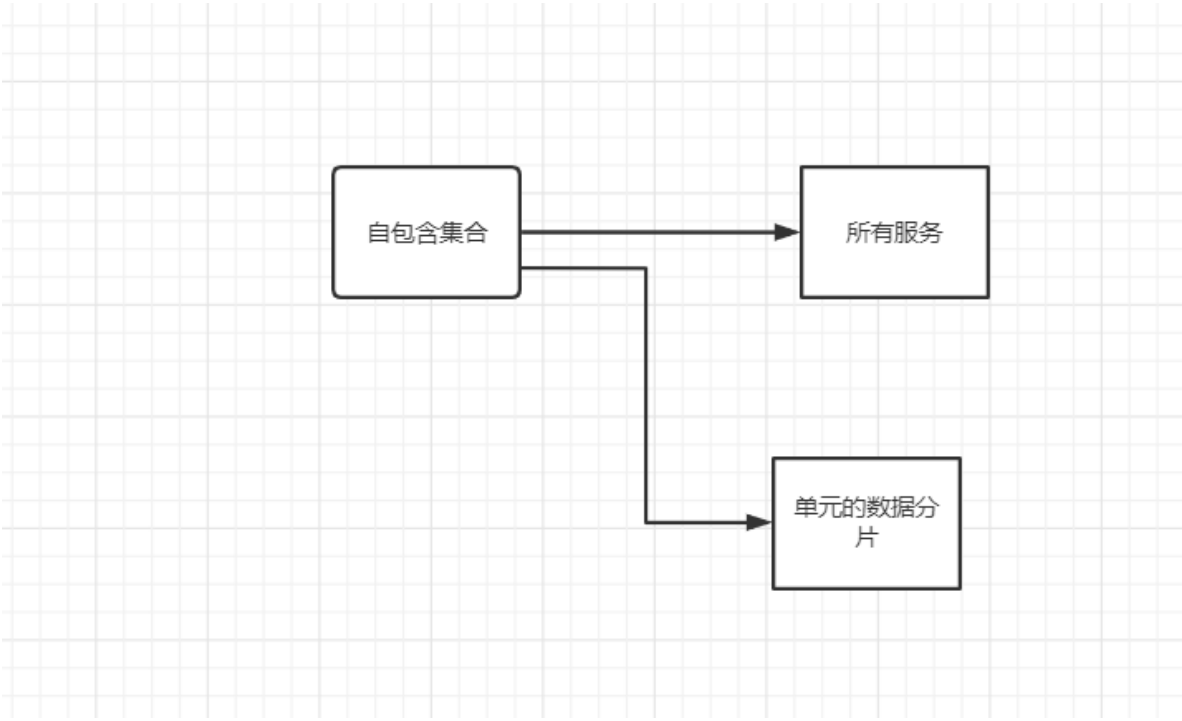
在这样的架构设计下,城市A停电了，可以切流到城市B。

对于跨idc机房的数据访问，至少有一个数据异步复制完成，才能进行。牺牲一部分可用性换取强一致性

只是，数据复制的时间，更长了，切流的时候，不可用的时间，更长了。

单元化的定义

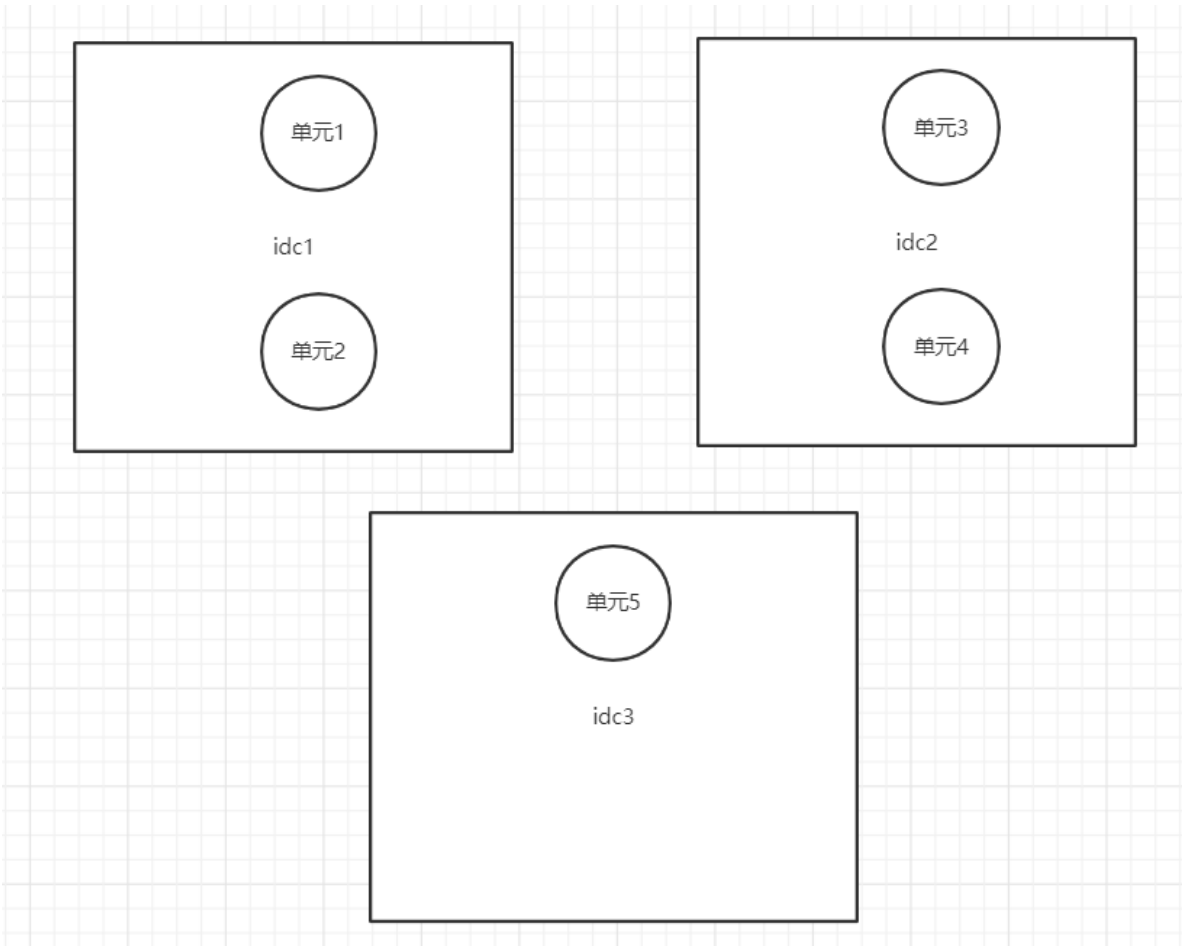
所谓单元，是指一个能完成所有业务操作的**自包含集合**，在这个集合中包含了**所有业务所需的所有服务**，以及**单元的数据分片**。



单元化架构就是把单元作为系统部署的基本单位，在全站所有idc机房中部署数个单元.

每个idc机房里的单元数目不定，任意一个单元都**部署了系统所需的所有的服务。**

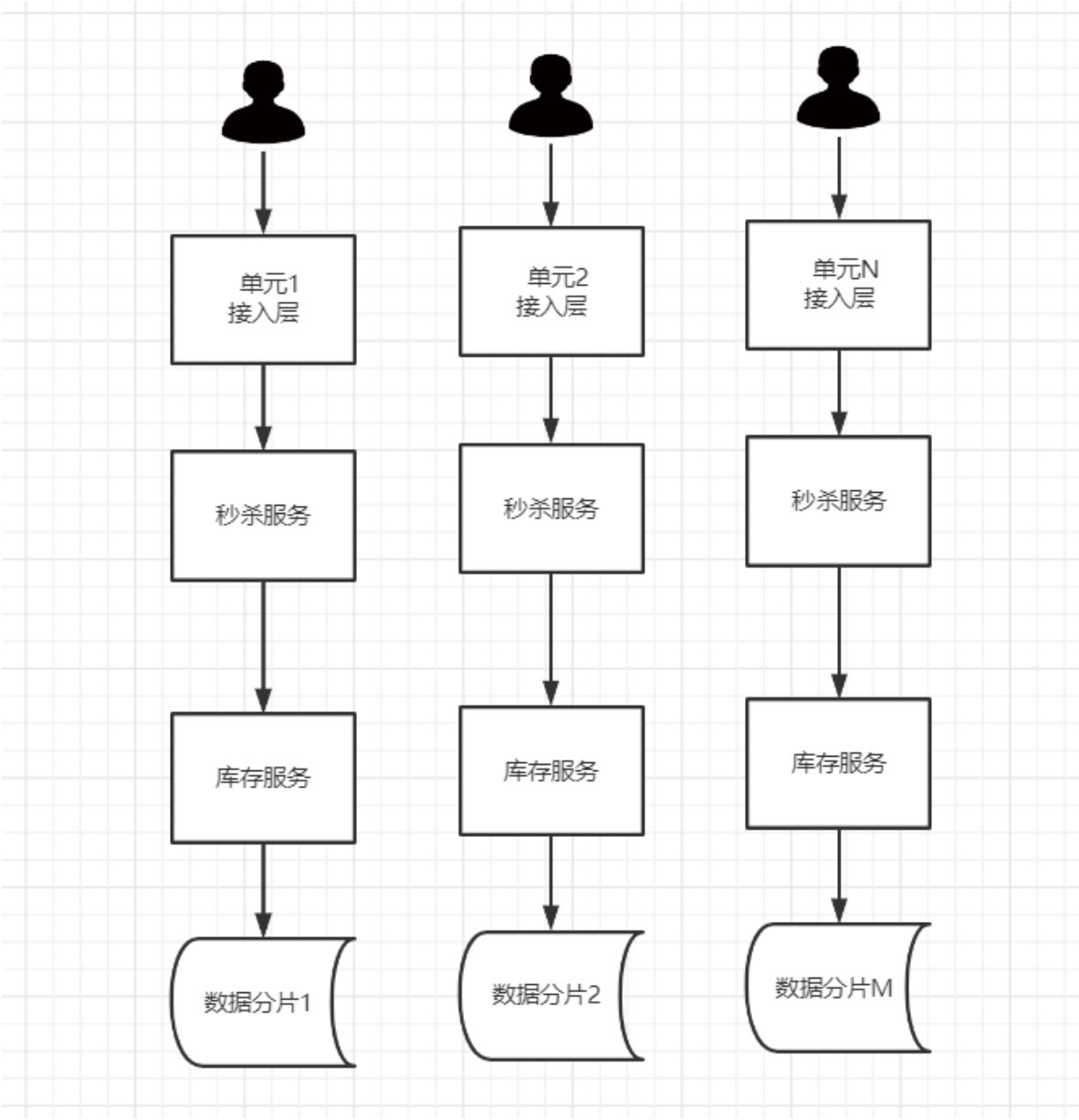
任意一个单元的数据是**首先拥有分片数据**，但是为了切流的方便， 最终需要拥有**全量数据。**



传统意义上的 SOA 化（服务化）架构，服务是分层的，每层的节点数量不尽相同，上层调用下层时，随机选择节点。

单元化架构下，服务仍然是分层的，

> 不同的是每一层中的任意一个节点都属于且仅属于某一个单元，上层调用下层时，仅会选择本单元内的节点。



而要做到单元化，必须要满足以下要求：

- 业务必须是可分片的，如 淘宝按照用户分片， 饿了么按照地理位置分片
- 单元内的业务是自包含的，调用尽量封闭

单元化的基础技术组件

为了实现单元化，需要由以下基础技术组件做支撑。

全局路由网关

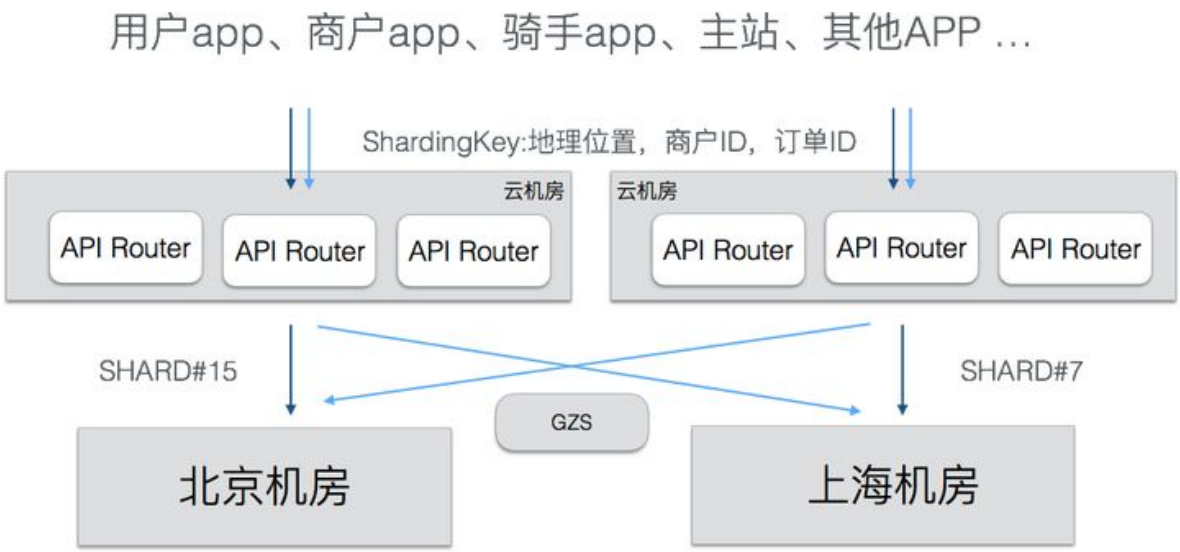
由于实施单元化后，整个交易链路从前到后的分片规则都是一致的，因此需要在入口处识别用户请求的所属单元，直接将请求路由至目标单元处理。这就使得必须有一套机制或系统来专门完成在这项工作，而又因为是在网络入口处处理，因此需要一个全局路由网关。

此时，需要**所有交易尽可能的带上分片键**，以便全局路由网关判断当前交易属于哪个单元。然而实际应用过程中，并不是所有交易都能带上分片键，这种情况就需要**应用跨单元交易转发**组件来处理了。

饿了么的 APIRouter： 路由分发服务

> 这一层负责对客户端过来的 API 调用进行路由，把流量导向到正确的 ezone。
>
> API Router 部署在多个公有云机房中，用户就近接入到公有云的API Router，还可以提升接入质量。

API Router是一个HTTP反向代理和负载均衡器，部署在公有云中作为HTTP API流量的入口，它能识别出流量的归属 shard ，并根据 shard 将流量转发到对应的 ezone。API Router 支持多种路由键，可以是地理位置，也可以是商户ID，订单ID等等，最终由 API Router 映射为统一的 Sharding ID。

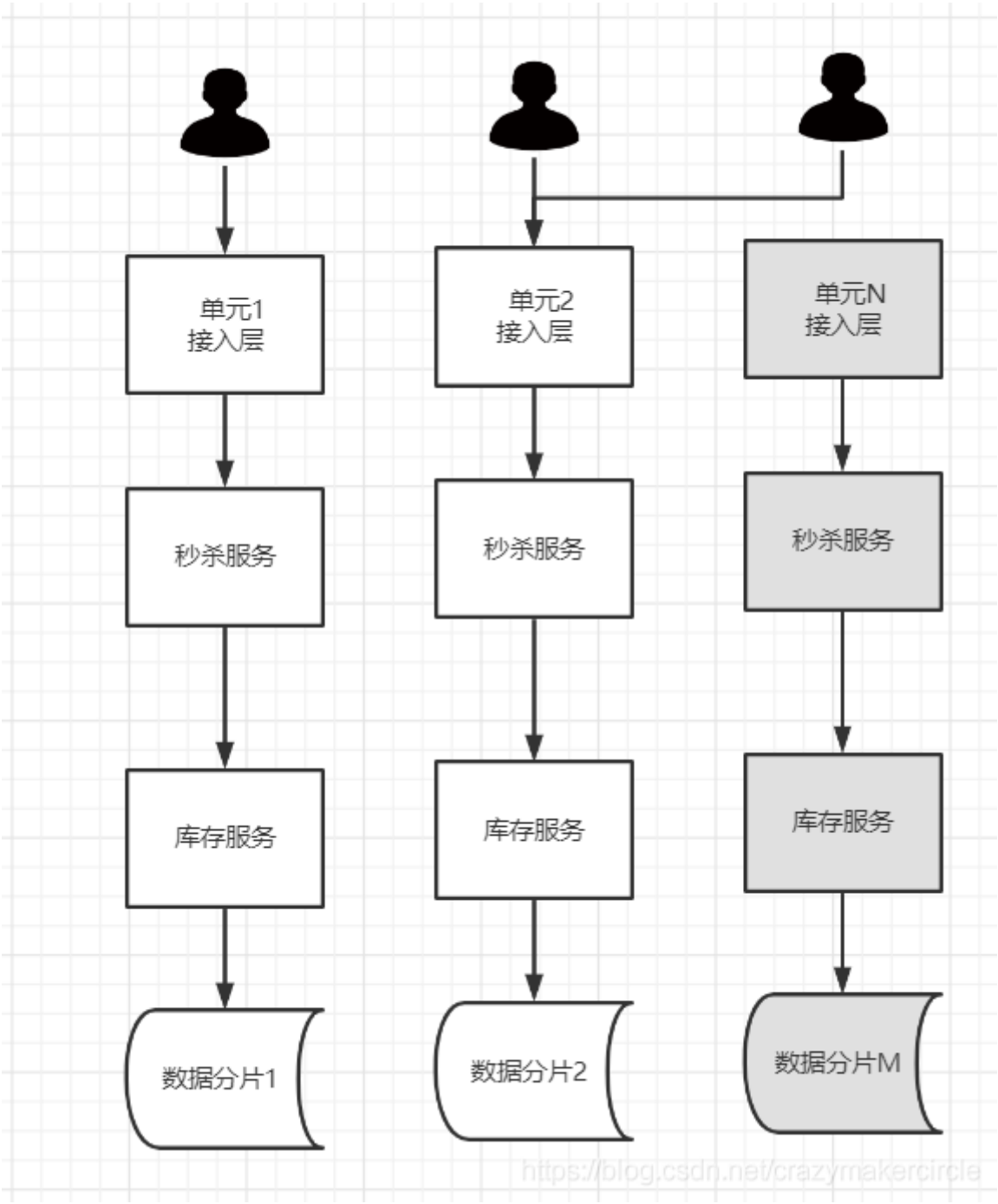


切流操作

把流量从一个单元，切到另一个单元

需要能随时按比例进行切流。

切流是一个日常性的操作，例如对机房进行的大规模运维、升级等操作，都会先将流量切走



目标单元的数据库必须有全量的数据

如果我们采用类似这样的架构，目标单元各有一套数据库，但他们之间的数据毫无重叠，那当需要切流的时候，再去迁移数据吗？

这个显然是不对的，所以这一点实际上要求**每个单元的数据库必须有全量的数据**，这样才有切流的基础。

跨单元数据同步组件

每个单元平时都要承担业务流量，处于“活”的状态。同时，每个单元要能按照百分比承担业务流量，并且能够动态的进行调整.

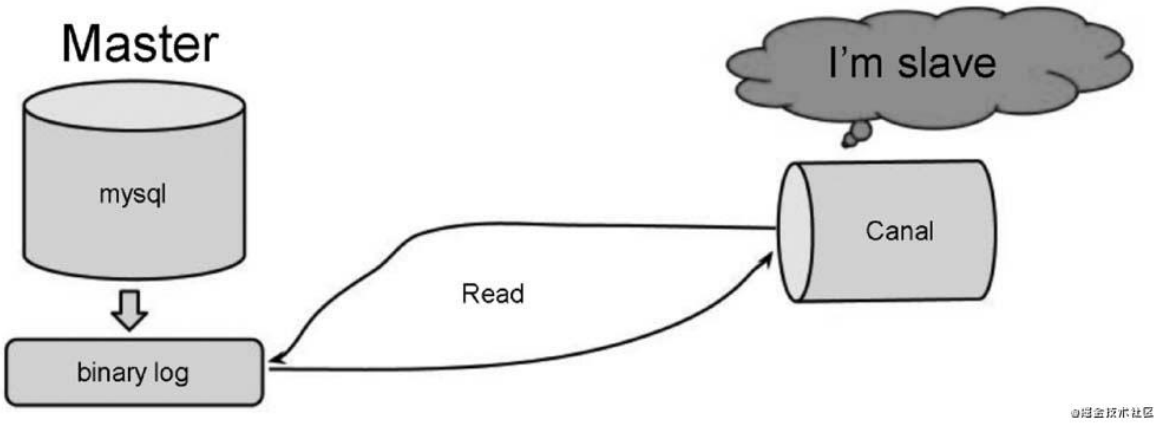
比方说，不同idc机房，mysql之间的数据异步复制

数据库如何同步

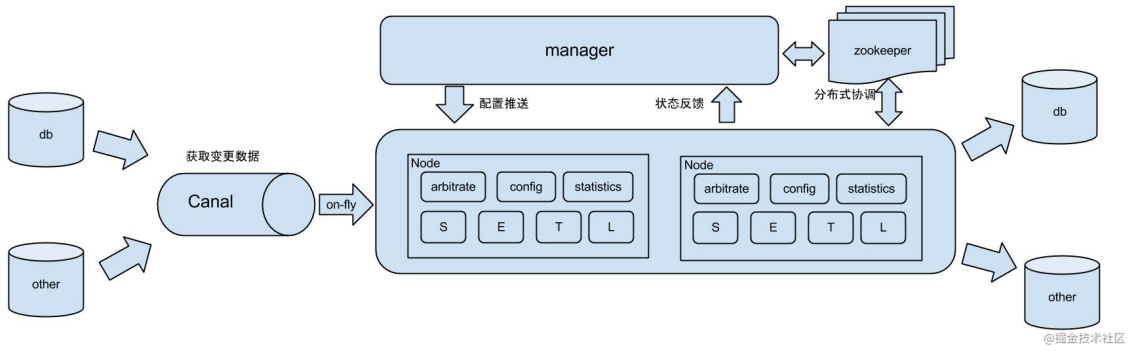
应用的双活和中间件的双活，最终都依赖于数据如何存放。如果两个机房中各部署一个数据库，那么机房间的数据如何同步呢。

以mysql为例，业界最常用的做法就是利用binlog做数据同步，最具代表性的就是阿里开源的Canal+Otter数据同步方案。

Canal可以伪装成一个Mysql Slave，接收binlog文件，获取到Mysql Master的数据变更，如图：



Otter可以将Canal获取的数据，同步到目标数据库，如图：



Canal+Otter不仅可以实现同构数据的同步，还能实现异构数据的同步，同时会简化压缩要传输的binlog，减少网络压力，传输速度更快。

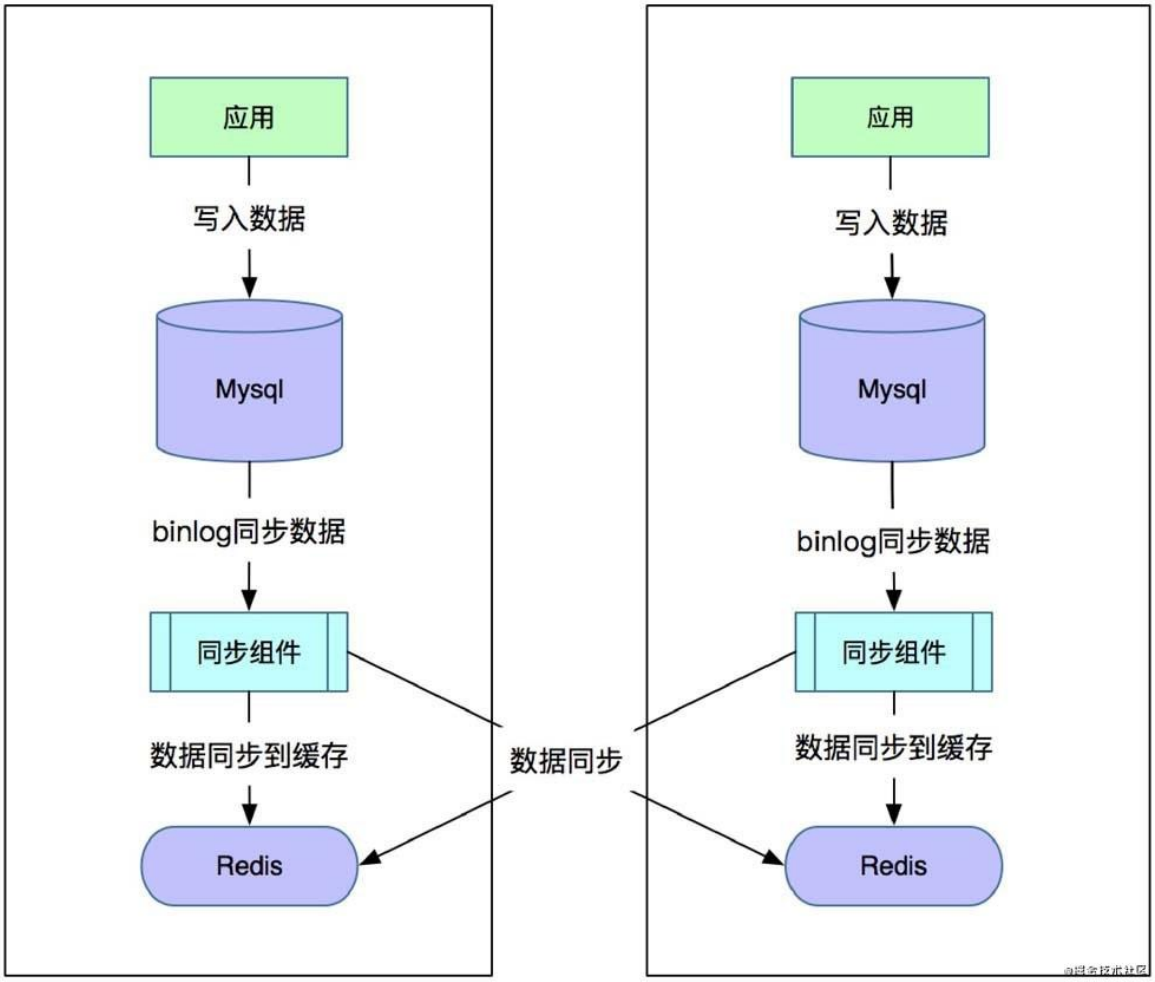
中间件如何进行数据同步

对于常用的中间件，如redis、kafka、ZooKeeper等，需要考虑双机房如何进行数据同步。

以redis为例，官方并没有提供跨机房的主主同步机制。如果仅利用redis的主从数据同步机制的话，需要将主节点与从节点部署在不同的机房。当主节点所在机房出现故障时，从节点可以升级为主节点，应用可以持续对外提供服务。但这种模式下，若要写数据，则只能通过主节点写，写请求有一半还是会跨机房访问。

若要实现redis的主主同步，需自己研发相应的插件，例如可以通过订阅mysql的binlog日志来做缓存数据的同步。通过实现同步组件，监听mysql的binlog并解析，将数据同步到两个机房的redis集群中。

如下图：



该方案看起来还不错，但是它具有以下弊端：

- 由于跨机房，数据同步会有几十到上百毫秒的延时。
- 同步组件将数据写入到本地redis和远程redis，由于没有事务的约束，不能保证两边都写成功，因此有可能会出现不一致。
- redis的数据可以同步，但数据的过期时间无法同步。
- redis具备5种数据结构，需要根据业务提前约定好使用哪种数据结构，业务侵入到了数据同步组件。

参考资料

https://blog.csdn.net/wuzhiwei549/article/details/80692278
https://www.i3geek.com/archives/841
https://www.cnblogs.com/seesun2012/p/9214653.html
https://github.com/yangliu0/DistributedLock

https://www.cnblogs.com/liuyang0/p/6744076.html
https://www.cnblogs.com/liuyang0/p/6800538.html
https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/
https://www.infoq.cn/article/cap-twelve-years-later-how-the-rules-have-changed
https://www.cnblogs.com/bluemiaomiao/p/11216380.html

https://www.jianshu.com/p/d909dbaa9d64

https://book.douban.com/subject/26292004/

https://segmentfault.com/a/1190000004468442

https://blog.csdn.net/universsky2015/article/details/105727244/

https://www.cnblogs.com/wudimanong/p/10340948.html

https://blog.csdn.net/kusedexingfu/article/details/103484198

https://www.jianshu.com/p/bfb619d3eea2

http://seata.io/zh-cn/docs/dev/mode/at-mode.html

https://blog.csdn.net/kusedexingfu/article/details/103484198

https://blog.csdn.net/wsdc0521/article/details/108223310