大型平台分析与设计

刘宝

第二章：分布式架构

第一部分：

- 1、分布式系统概述
- 2、CAP经典理论
- 3、高并发
- 4、高可用
- 5、分布式事务

分布式系统概述

- 回顾：什么叫分布式系统？

相对单体架构或集中式架构而言，将相同或不同的功能模块运行在不同的机器上，互相之间通过网络通信。

思考，分布式系统具备什么特点？

分布式系统概述

- 分布式系统特点:

分布性:

分布式系统中的多节点计算机会在空间上随意分布，同时，机器的分布情况可能会发生动态变化。

分布式系统概述

- 分布式系统特点:

对等性:

分布式系统中的节点没有“严格”的主/从之分。“副本”是分布式系统常见概念之一，指在数据或者服务上提供的一种冗余方式，这是解决数据丢失以及局部服务异常的有效手段。

分布式系统概述

- 分布式系统特点:

并发性:

同一分布式系统下诸多节点会共享操作资源。

分布式系统概述

- 分布式系统特点:

全局时钟:

典型的分布式系统是一系列在空间上随意分布的诸多进程，且具备一定并发性。那么在资源使用和响应的过程中，两个事件谁先谁后如何定义呢?

在一搜向东行驶的飞机上。。。。。

分布式系统概述

- 分布式系统特点：

故障总是发生：

> 一个被大量工程实践锁检验过的黄金定理：任何在设计阶段考虑到的异常情况，一定会在系统实际运行过程中发生。且还会发生在设计时未考虑到的异常故障。
>
> 假设一台机器一年内发生故障的概率是1%，那么100台机器的集群一年内发生的概率是多少？

分布式系统概述

- 小结：分布式系统特点：

（1）分布性
（2）对等性
（3）并发性
（4）全局时钟
（5）故障总是发生

分布式系统概述

- 分布式面临的典型问题:

（1）解决高并发问题的分片策略、异步化任务调度等
（2）解决分布式事务场景下数据一致性问题
（3）解决数据分区或多副本场景下数据一致性问题
（4）多节点中的不稳定、容灾等问题
（5）高并发时的数据预期问题

等等等等。。

分布式系统概述

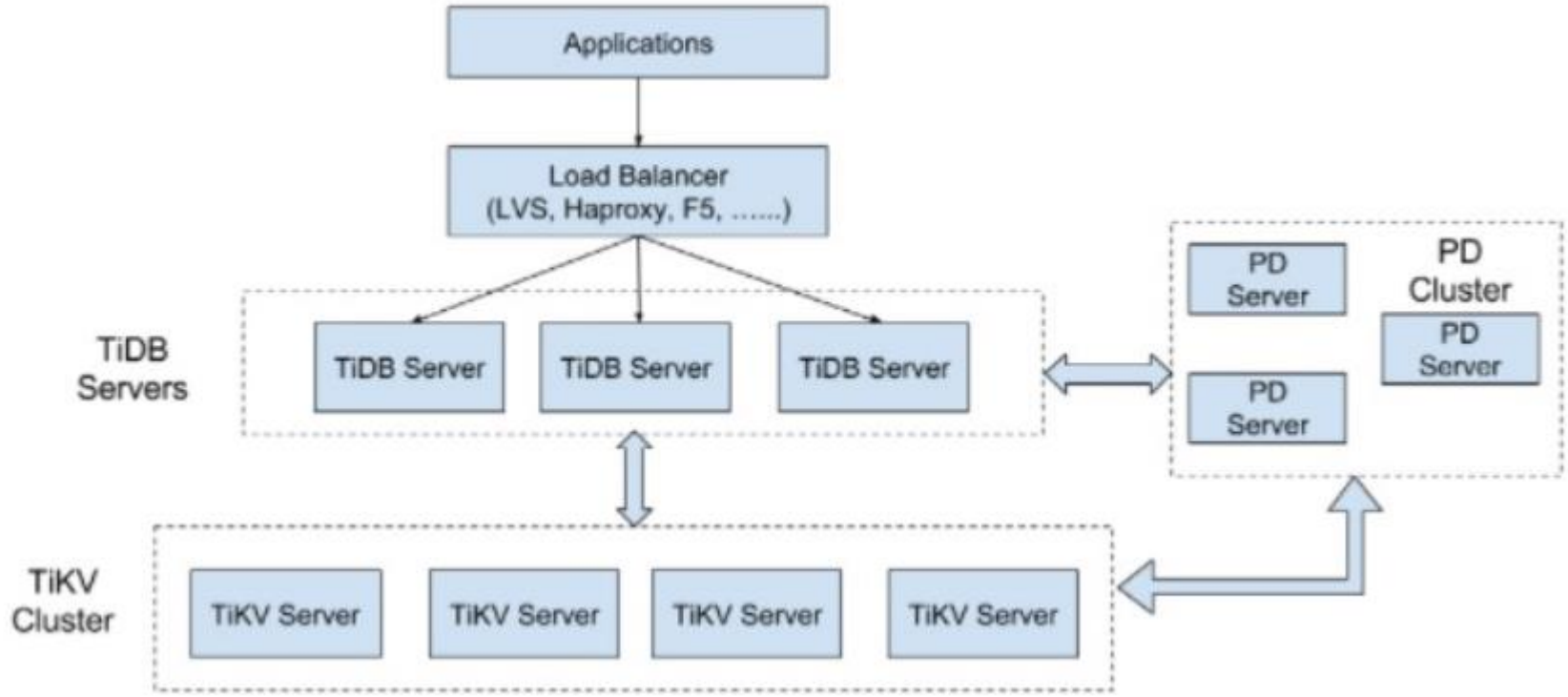
- 经典案例:

  HadoopHDFS（大数据分布式文件系统）
  Elastic search（搜索引擎）
  Kafka（消息中间件）
  TiDB（分布式关系型数据库）
  Redis-Cluster（集群模式）

- 列举其他案例?

分布式概述

- 案例剖析：TiDB



分布式概述

- TiDB Server：
- TiDB Server 负责接收 SQL 请求，处理 SQL 相关的逻辑，并通过 PD 找到存储计算所需数据的 TiKV 地址，与 TiKV 交互获取数据，最终返回结果。 TiDB Server 是无状态的，其本身并不存储数据，只负责计算，可以无限水平扩展，可以通过负载均衡组件（如LVS、HAProxy 或 F5）对外提供统一的接入地址。
- PD Server：
- Placement Driver (简称 PD) 是整个集群的管理模块，其主要工作有三个： 一是存储集群的元信息（某个 Key 存储在哪个 TiKV 节点）；二是对 TiKV 集群进行调度和负载均衡（如数据的迁移、Raft group leader 的迁移等）；三是分配全局唯一且递增的事务 ID。
- PD 是一个集群，需要部署奇数个节点，一般线上推荐至少部署 **3** 个节点。
- TiKV Server
- TiKV Server 负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是 Region（区域），每个 Region 负责存储一个 Key Range （从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region 。TiKV 使用 Raft 协议做复制，保持数据的一致性和容灾。副本以 Region 为单位进行管理，不同节点上的多个 Region 构成一个 Raft Group，互为副本。数据在多个 TiKV 之间的负载均衡由 PD 调度，这里也是以 Region 为单位进行调度。

分布式概述

- 核心特性

水平扩展

无限水平扩展是 TiDB 的一大特点，这里说的水平扩展包括两方面：计算能力和存储能力。TiDB Server 负责处理 SQL 请求，随着业务的增长，可以简单的添加 TiDB Server 节点，提高整体的处理能力，提供更高的吞吐。TiKV 负责存储数据，随着数据量的增长，可以部署更多的 TiKV Server 节点解决数据 Scale 的问题。PD 会在 TiKV 节点之间以 Region 为单位做调度，将部分数据迁移到新加的节点上。所以在业务的早期，可以只部署少量的服务实例，随着业务量的增长，按照需求添加 TiKV 或者 TiDB 实例。

分布式概述

高可用

高可用是 TiDB 的另一大特点，TiDB/TiKV/PD 这三个组件都能容忍部分实例失效，不影响整个集群的可用性。

1、TiDB

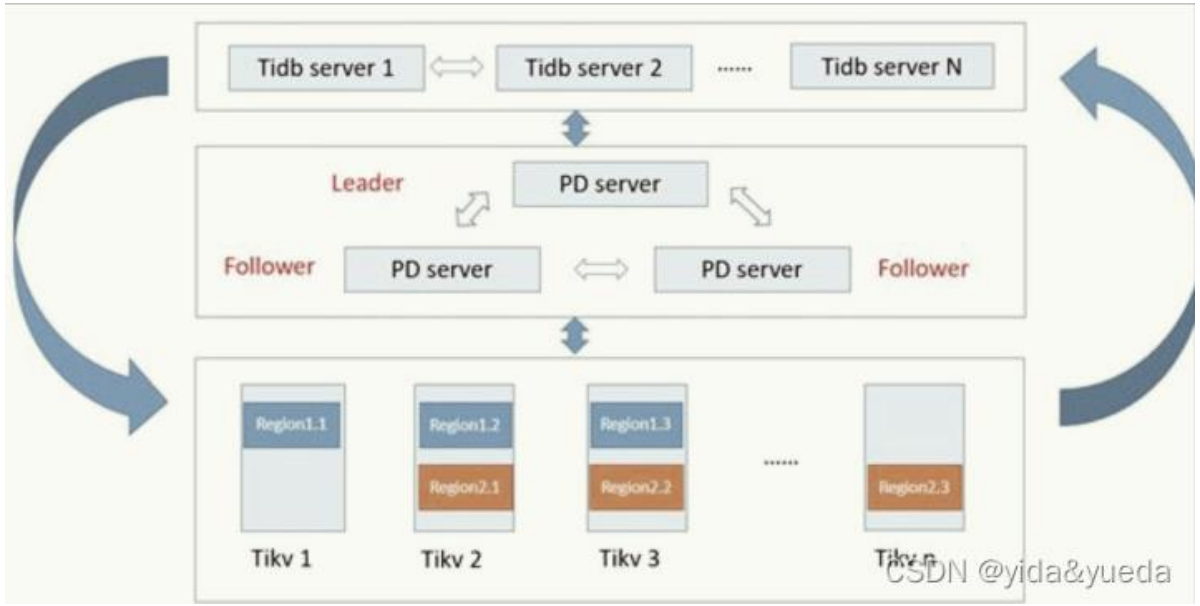
TiDB 是无状态的，推荐至少部署两个实例，前端通过负载均衡组件对外提供服务。当单个实例失效时，会影响正在这个实例上进行的 Session，从应用的角度看，会出现单次请求失败的情况，重新连接后即可继续获得服务。单个实例失效后，可以重启这个实例或者部署一个新的实例。

2、PD

PD 是一个集群，通过 Raft 协议保持数据的一致性，单个实例失效时，如果这个实例不是 Raft 的 leader，那么服务完全不受影响；如果这个实例是 Raft 的 leader，会重新选出新的 Raft leader，自动恢复服务。PD 在选举的过程中无法对外提供服务，这个时间大约是3秒钟。推荐至少部署三个 PD 实例，单个实例失效后，重启这个实例或者添加新的实例。

3、TiKV

TiKV 是一个集群，通过 Raft 协议保持数据的一致性（副本数量可配置，默认保存三副本），并通过 PD 做负载均衡调度。单个节点失效时，会影响这个节点上存储的所有 Region。对于 Region 中的 Leader 节点，会中断服务，等待重新选举；对于 Region 中的 Follower 节点，不会影响服务。当某个 TiKV 节点失效，并且在一段时间内（默认 30 分钟）无法恢复，PD 会将其上的数据迁移到其他的 TiKV 节点上。



分布式概述

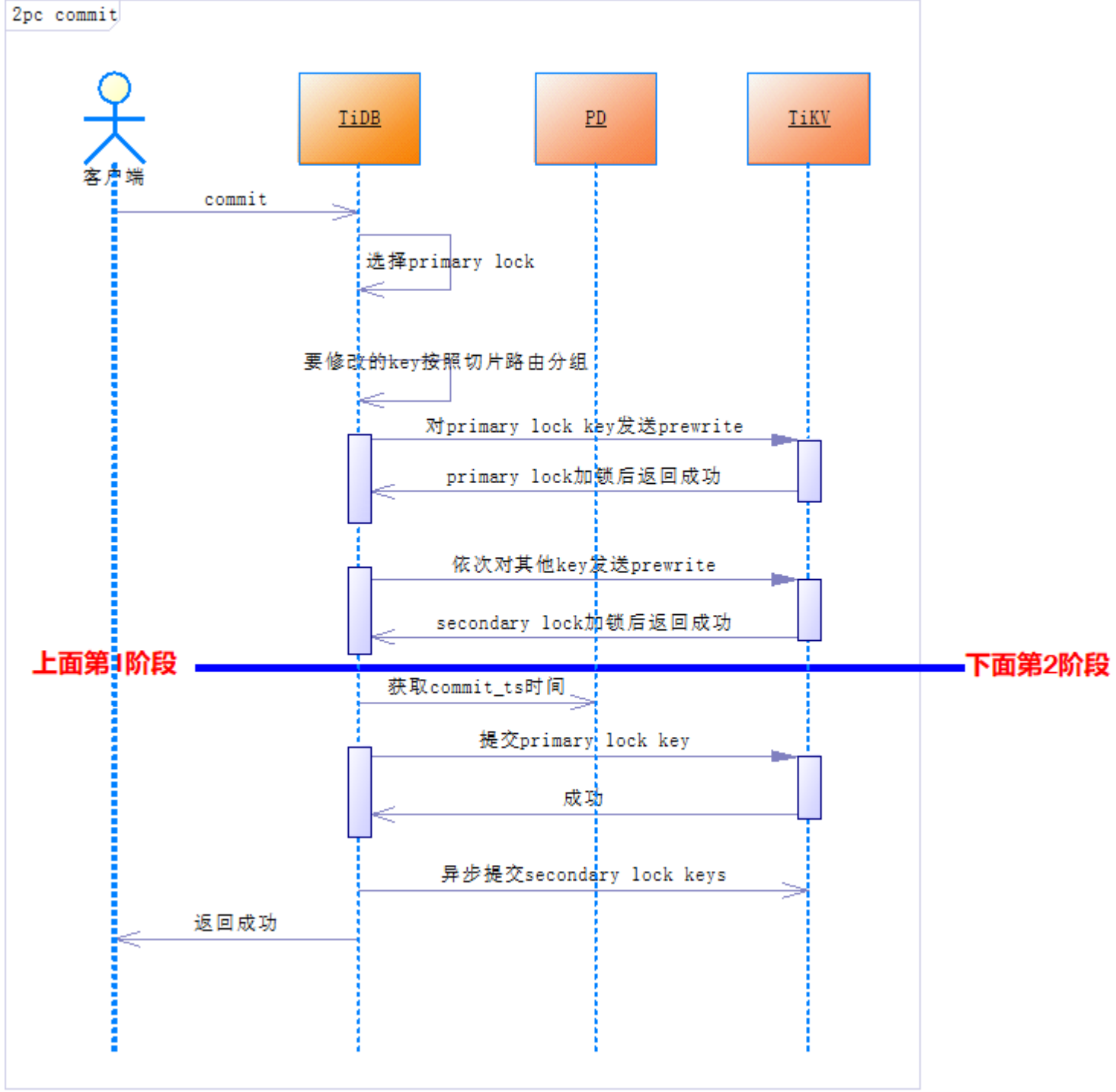
TiDB分布式事务模型简介:

Percolator模型

第一阶段，TiDB收到客户端请求后，首先会从缓存的待修改key中找出第一个发送prewrite请求，这个key加primary lock后返回成功。然后TiDB会对这个事务其他的所有的key发送prewrite请求，这些key加secondary lock后返回成功。

第二阶段，prewrite成功后，TiDB首先会从PD获取一个时间戳作为当前事务的commit_ts，然后向primary lock key发送commit请求，primary lock key提交数据成功后清理掉primary lock返回成功。TiDB收到primary lock key的成功消息后给客户端返回成功。

乐观事务的冲突检测主要是在prewrite阶段，如果检测到当前的key已经加锁，会有一个等待时间，这个时间过后如果还没有获取到锁，就返回失败。因此当多个事务修改同一个key时，必然导致大量的锁冲突。



分布式概述

- 案例分析思考:
- 思考1：相比于传统的分库分表，你认为TIDB有哪些优势?
- 思考2：尝试总结一些TIDB的使用场景。
- 思考3：你在TIDB的架构和原理中，看到哪些优秀的设计思想。

分布式概述

- 案例剖析：Kafka

Producer：Producer 即生产者，消息的产生者，是消息的入口
Broker：Broker 是 kafka 一个实例，每个服务器上有一个或多个 kafka 的实例，简单的理解就是一台 kafka 服务器，kafka cluster表示集群的意思
Topic：消息的主题，可以理解为消息队列，kafka的数据就保存在topic。在每个 broker 上都可以创建多个 topic 。
Partition：Topic的分区，每个 topic 可以有多个分区，分区的作用是做负载，提高 kafka 的吞吐量。**同一个 topic 在不同的分区的数据是不重复的。**
Replication：每一个分区都有多个副本，副本的作用是做备胎，主分区（Leader）会将数据同步到从分区（Follower）。当主分区（Leader）故障的时候会选择一个备胎（Follower）上位，成为 Leader。在kafka中默认副本的最大数量是10个，且副本的数量不能大于Broker的数量，follower和leader绝对是在不同的机器，同一机器对同一个分区也只可能存放一个副本
Message：每一条发送的消息主体。
Consumer：消费者，即消息的消费方，是消息的出口。
Consumer Group：我们可以将多个消费组组成一个消费者组，**在 kafka 的设计中同一个分区的数据只能被消费者组中的某一个消费者消费。同一个消费者组的消费者可以消费同一个topic的不同分区的数据，这也是为了提高kafka的吞吐量**!
Zookeeper：kafka 集群依赖 zookeeper 来保存集群的的元信息，来保证系统的可用性。

分布式概述

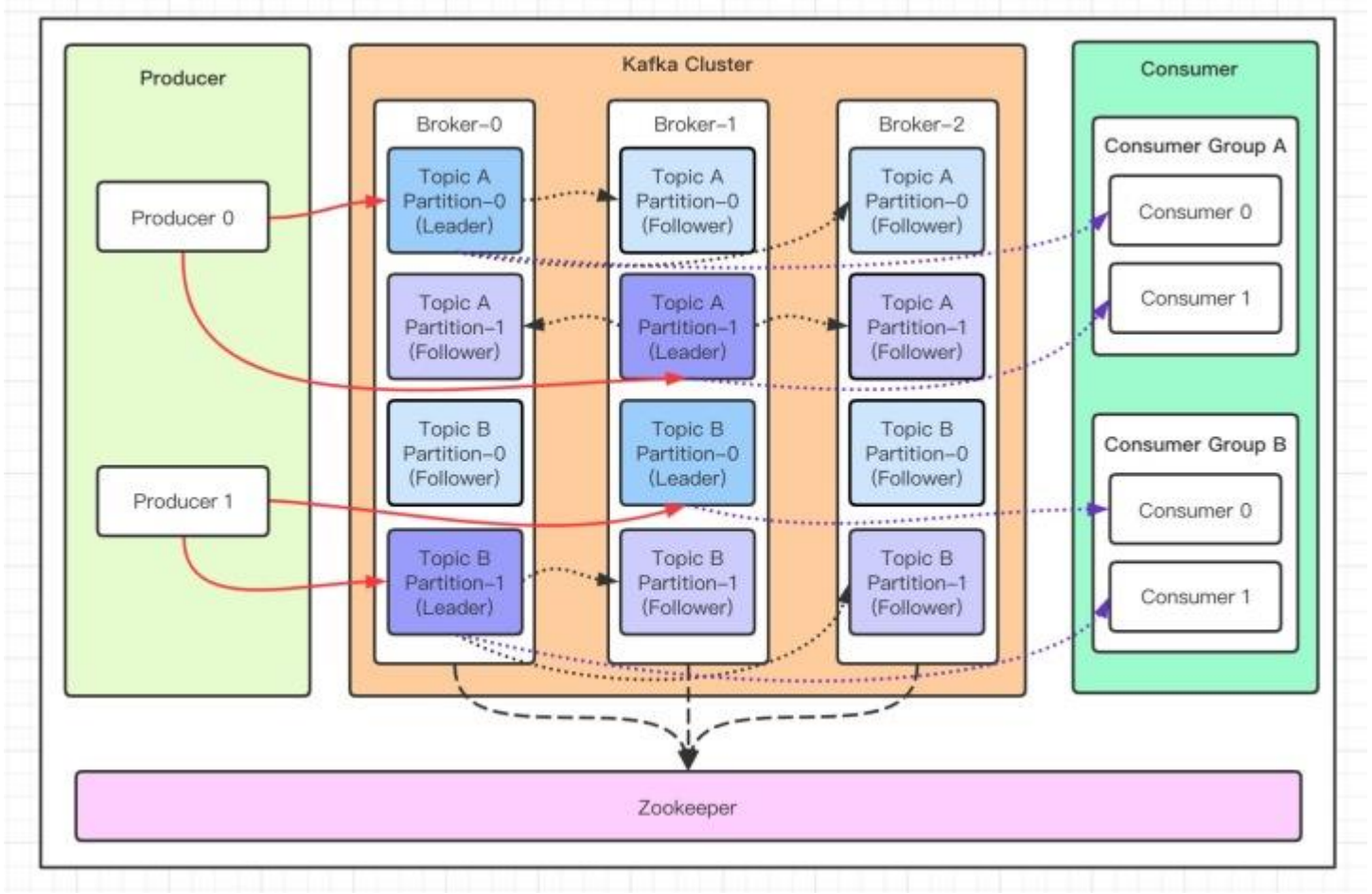
- 案例剖析：Kafka

kafka 本质就是一个消息系统，与大多数的消息系统一样，主要的特点如下：
使用推拉模型将生产者和消费者分离

为消息传递系统中的消息数据提供持久性，以允许多个消费者

提供高可用集群服务，主从模式，同时支持横向水平扩展

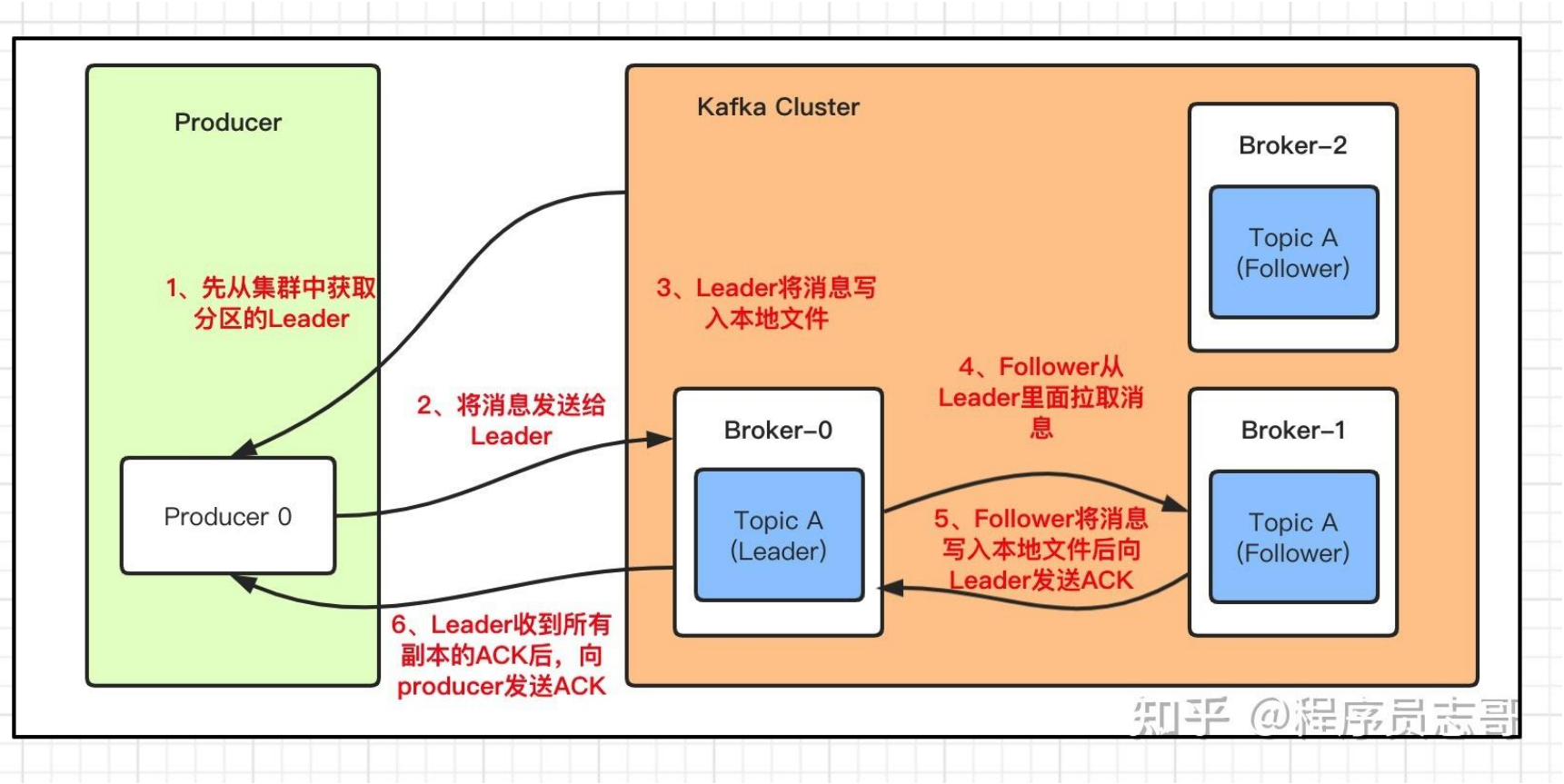
重点关注分区和副本



分布式概述

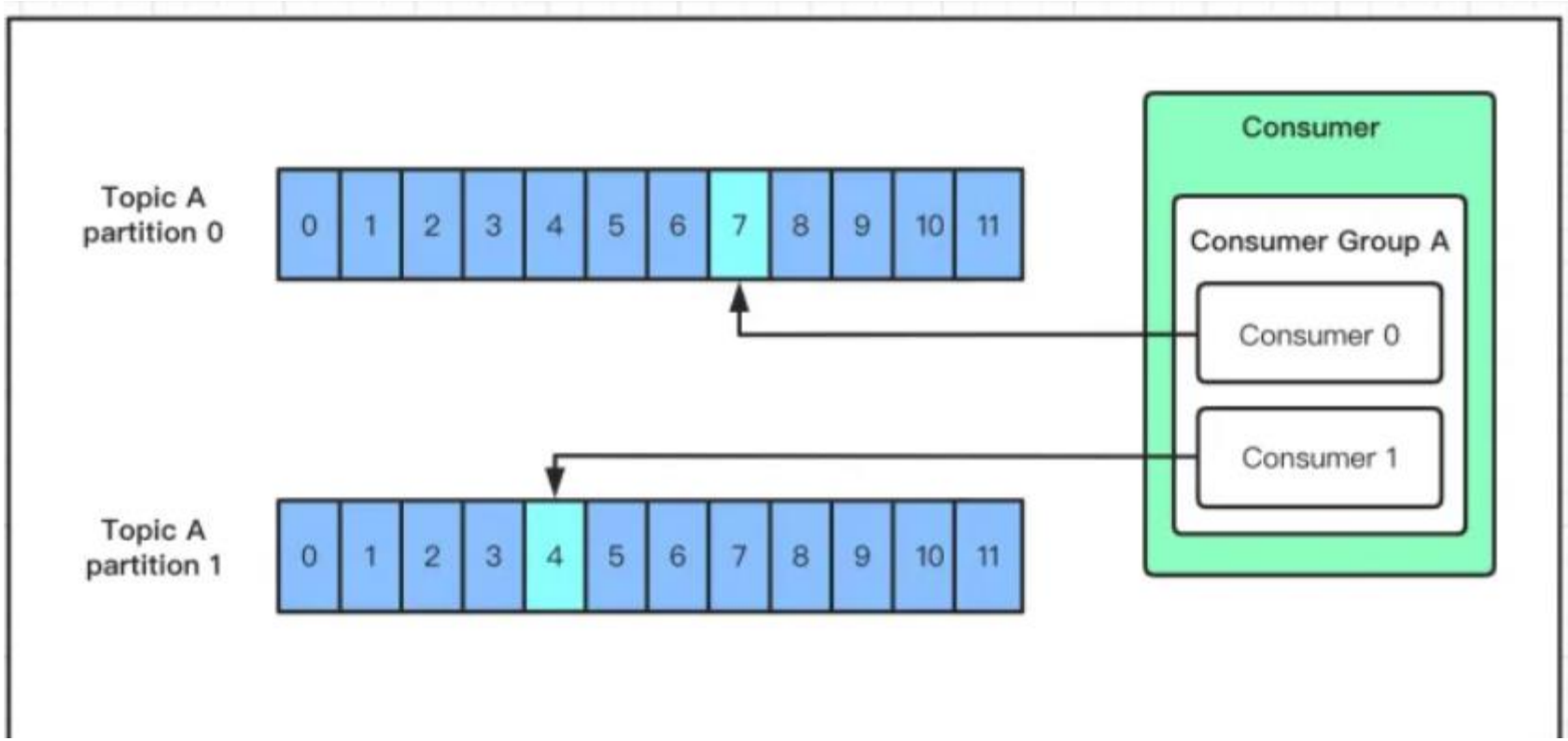
kafka 每次发送数据都是向Leader分区发送数据，并顺序写入到磁盘，然后Leader分区会将数据同步到各个从分区Follower，即使主分区挂了，也不会影响服务的正常运行。

副本保证高可用



分布式概述

依靠分区保证了消费高并发



分布式概述

- 案例分析思考:
- 思考一：能否总结一下Kafka的适用场景。
- 思考二：你从Kafka的设计思路中学到了什么。

小结回顾

- 1、什么是分布式系统。分布式系统：相对单体架构或集中式架构而言，将相同或不同的功能模块运行在不同的机器上，互相之间通过网络通信。

- 2、分布式系统的特点。（1）分布性、（2）对等性、（3）并发性、（4）全局时钟、（5）故障总是发生

- 3、分布式系统面临的难题。（1）解决高并发问题的分片策略、异步化任务调度等、（2）解决分布式事务场景下数据一致性问题、（3）解决数据分区或多副本场景下数据一致性问题、（4）多节点中的不稳定、容灾等问题、（5）高并发时的数据预期问题

- 4、案例分析。
- 小组讨论：针对你们曾经做过的课程设计，可否使用分布式的思想做性能上的提升。

CAP经典理论

- 2000年，来自加州大学伯克利分校的Eric Brewer教授首次提出CAP猜想。2年后，麻省理工学院的Seth Gilbert和Nancy Lynch从理论上证明了猜想的可行性。从此CAP理论成为了分布式计算领域的公认定理，并深深地影响了分布式计算的发展。

- CAP理论，它说：

一个分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容错性（P：Partition tolerance）这三个基本需求。

CAP经典理论

- 一致性:

指数据的多个副本间能否保持一致的特性。

若：针对某一数据项的修改成功后，可使所有用户或业务场景立刻读取或感知到最新的值，则该系统就可以认为具有强一致性。

CAP经典理论

- 可用性:

系统提供的服务必须一直处于可用的状态，对于用户的所有请求都能够在“有限的时间”内“返回结果”。

思考？如何定义有限时间？

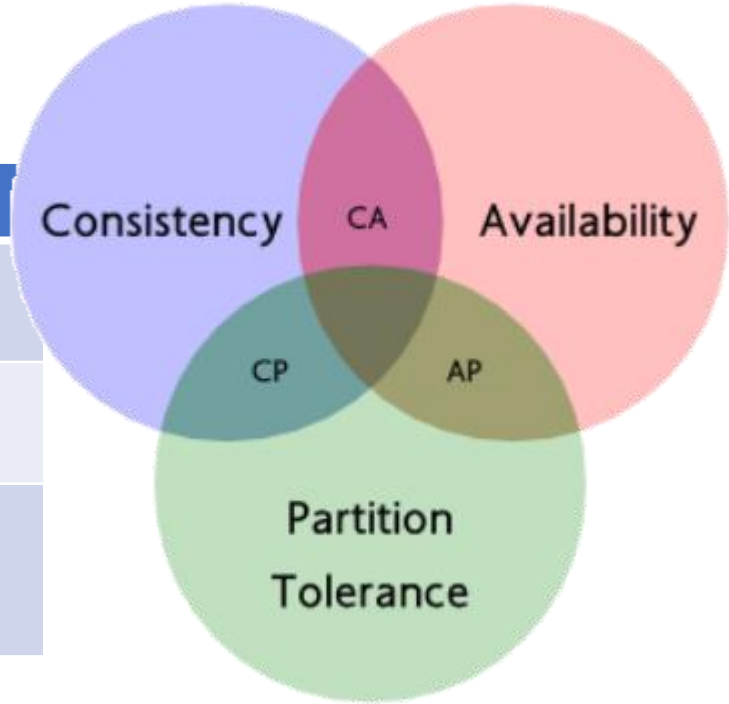
CAP经典理论

- 分区容错性:

系统遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障。

CAP经典理论

放弃CAP定理	说明
放弃P	放弃分区容错，意味着要把数据都搁置在一个节点上。也意味着放弃了分布式的可扩展性
放弃A	意味着当网络或者分区发生故障时，通过停止服务，等待问题恢复再重新提供服务的方式保证分区和数据一致性。
放弃C	弹性空间很大。 放弃一致性一般来说都是放弃强一致性，数据只要在可接受的时间内逐渐达成最终一致性，都是可以接受的。



最终一致性：系统中的所有副本，在经过一段时间的同步后，最终能达到一致的状态。

小组讨论：为什么CAP不能同时达成？

高并发读写设计原理

构建分布式系统的核心目的之一：支持高并发

场景一：高并发读

场景二：高并发写

高并发读

- 典型场景一：搜索场景

搜索引擎、电商、自媒体发布平台等等

高并发读

策略1：动静分离与CND加速

策略2：加缓存

1）缓存雪崩　　2）缓存击穿　　3）大量的热Key过期

策略3：异步化

策略4：转批量（预读）

策略5：重写轻读

案例解析

- 某博

关注人员关系表 follow

自增id	User_id关注人	Followings被关注人

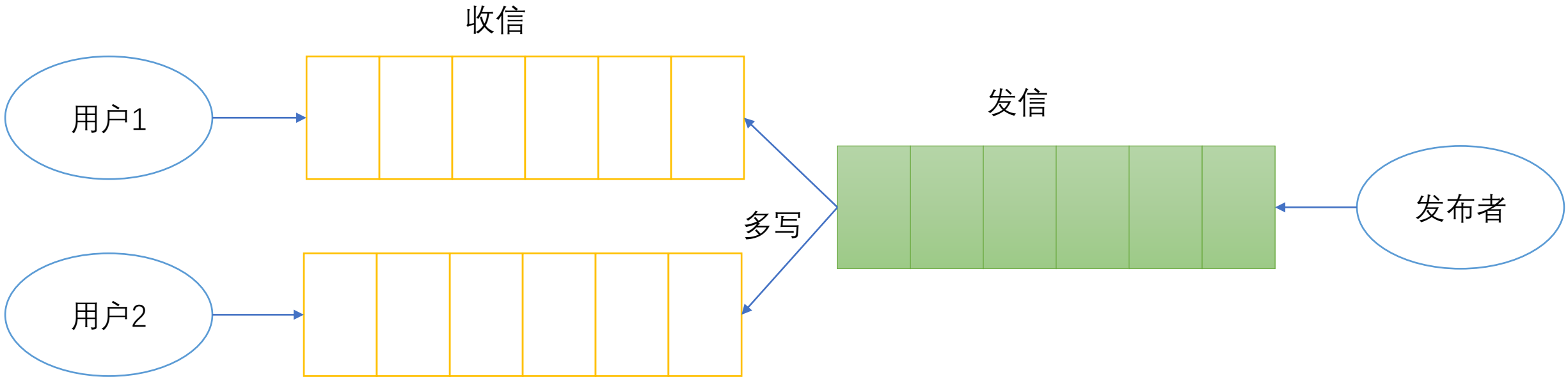
消息表

自增id	User_Id发布人	Msg_id微博Id

Select msg_id from msg where user_id in （select followings from follow where user_id=#{user_id}）

案例解析：某博

常用做法：



高并发读

策略1：动静分离与CND加速

策略2：加缓存

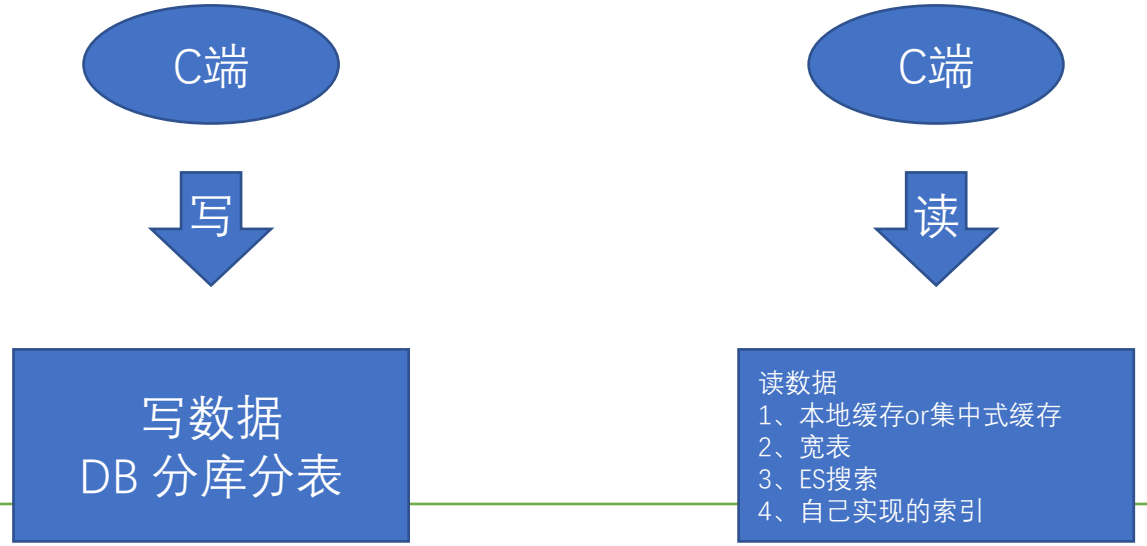
1）缓存雪崩　　2）缓存击穿　　3）大量的热Key过期

策略3：异步化

策略4：转批量

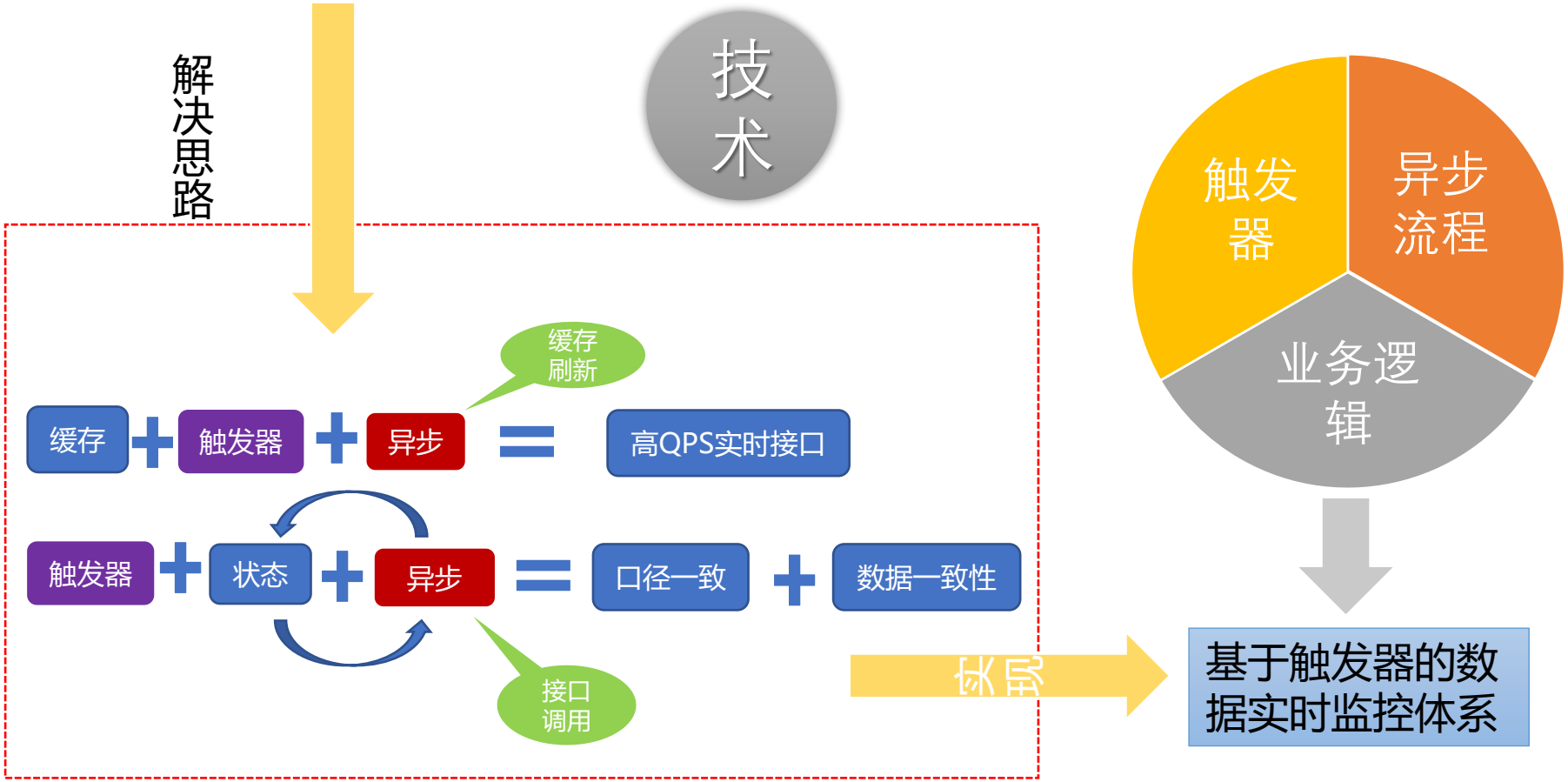
策略5：重写轻读

总结：读写分离

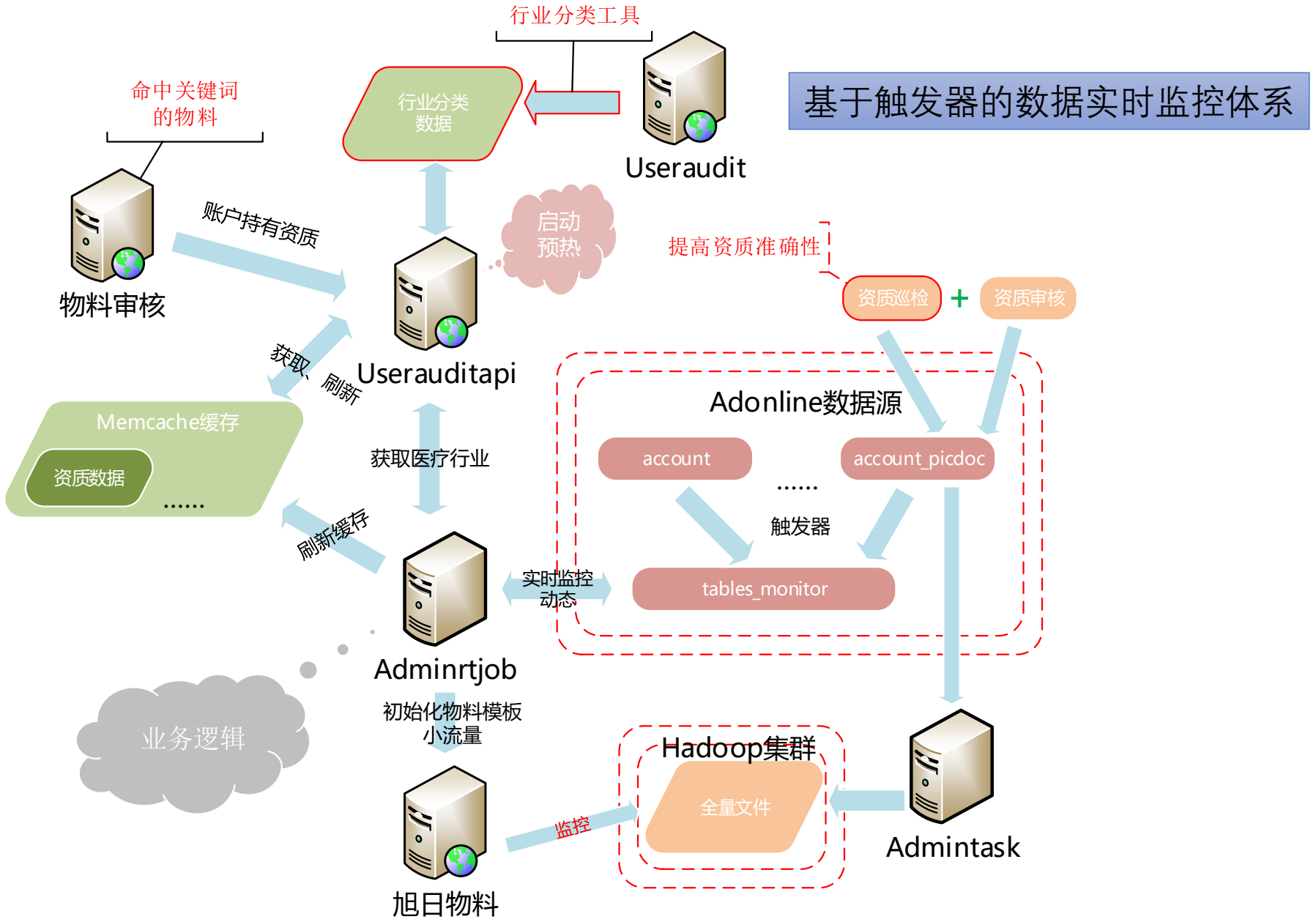


真实案例分析一：

问题：
1，物料审核需要获取客户资质列表信息，量级较大，预估QPS为几万，且有较高的实时性要求。
2，客户行业变更时，需要调用产品线接口创建物料模板，功能入口较多且需考虑数据的最终一致性。



真实案例分析一：


行业分类工具
命中关键词
的物料
行业分类
数据
Useraudit
基于触发器的数据实时监控体系
账户持有资质
物料审核
启动
预热
提高资质准确性
资质巡检
+
资质审核
获取、刷新
Useraudit api
Adonline数据源
Memcache缓存
资质数据
.....
获取医疗行业
account
account_picdoc
.....
触发器
刷新缓存
实时监控
动态
tables_monitor
Adminrtjob
初始化物料模板
小流量
业务逻辑
Hadoop集群
全量文件
监控
Admintask
旭日物料

高并发写

策略1：数据分片

DB的分库分表、JDK的currentHashMap、kafka的分区

以及：大家给出的case

策略2：任务分片

多线程、归并思路（map\reduce)

策略3：异步化(LSM树 log structured merged）

策略4：转批量

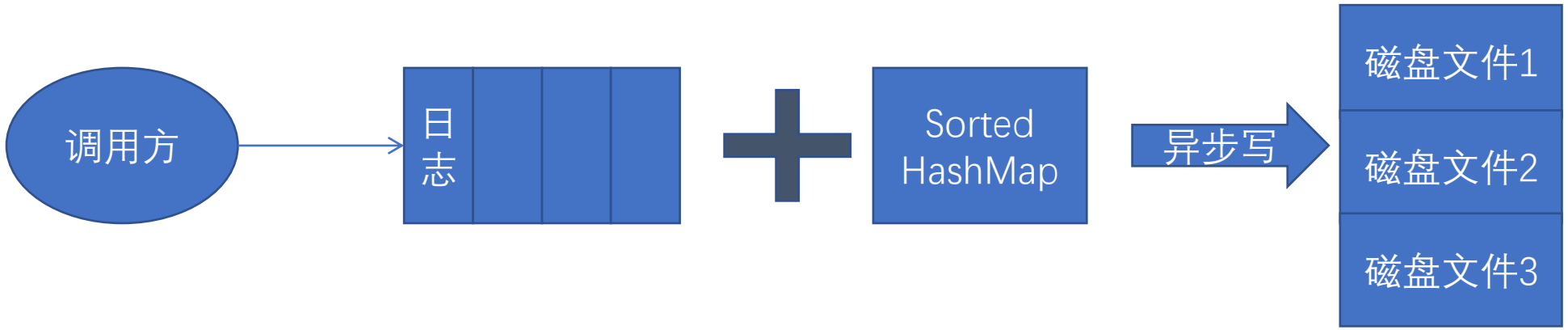
高并发写

LSM树：

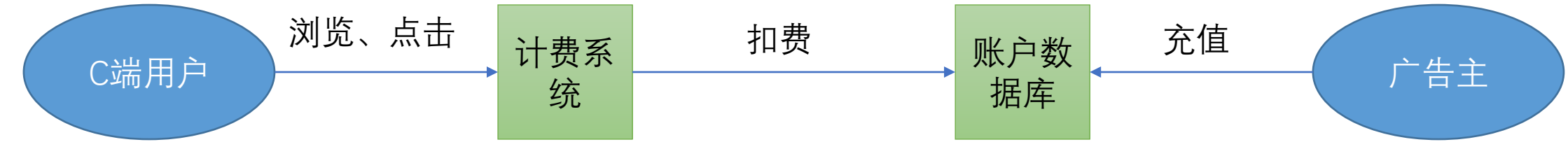
核心思想：异步写。LSM支撑的是KV存储，插入时，key是无序的；但在磁盘上又需要按照key的大小顺序存储，也就是在磁盘上实现Sorted HashMap。按key排序是为了快速检索，但不可能在写入磁盘的同时排序，效率太低。

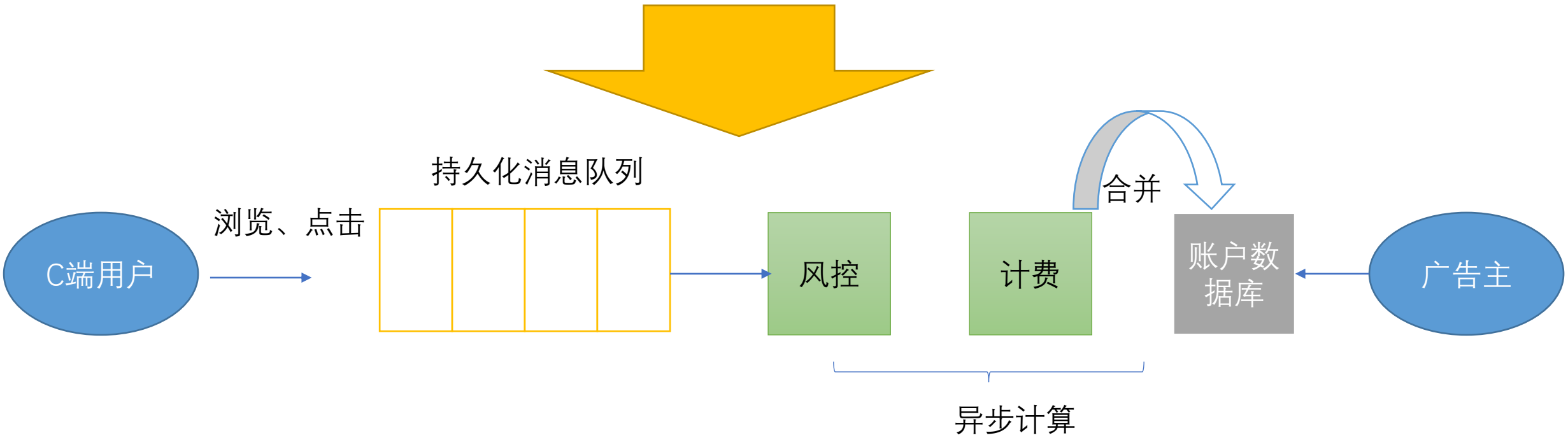
解决思路：把sorted HashMap放入内存中维护。每写一次数据，先预写一条顺序日志，再写入内存，这样即使重启，也可以从日志中恢复数据。因为日志时顺序写入，写入的效率得以大幅度提升。

日志顺序写入+内存Sorted HashMap排序+后台任务定期将内存数据合并到磁盘



案例剖析一：广告计费


C端用户
浏览、点击
计费系统
扣费
账户数据库
充值
广告主


持久化消息队列
浏览、点击
C端用户
风控
计费
合并
账户数据库
广告主
异步计算

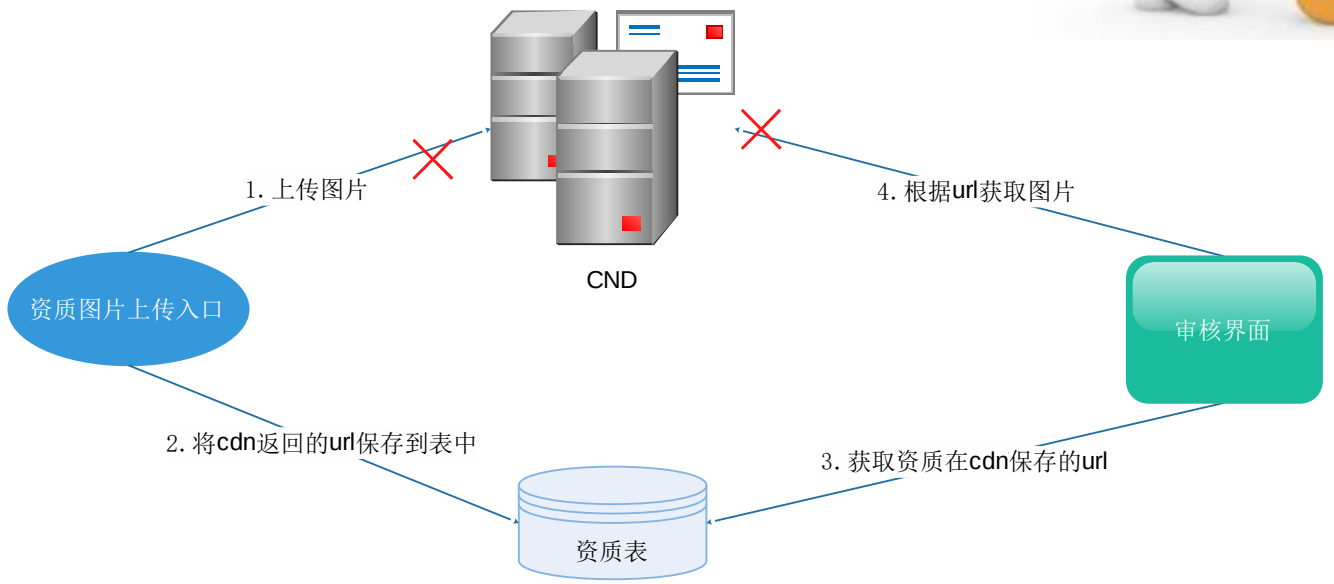
案例剖析二：资质核心数据可用性

问题：

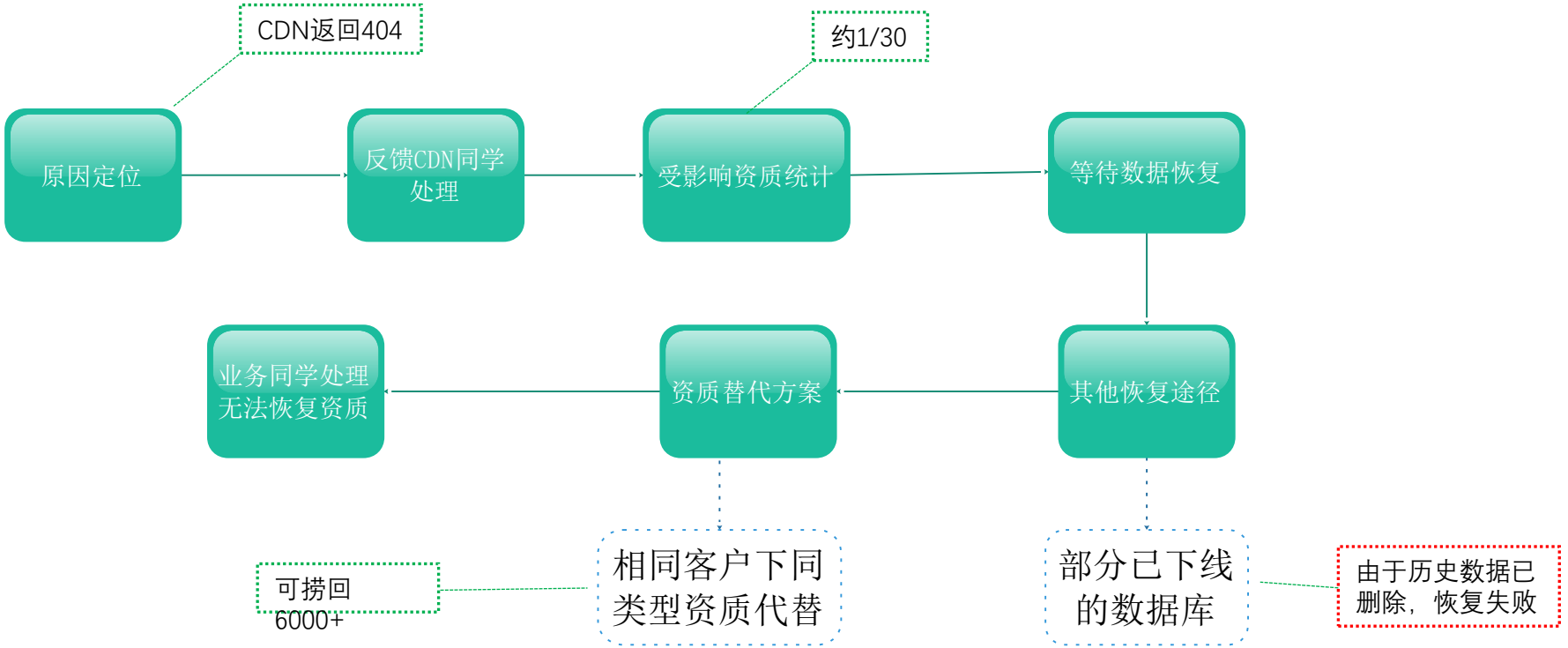
客户资质无法查看，阻塞审核流程！

原因：

CDN异常！！资质链接返回404！！



案例剖析二：资质核心数据可用性



效果：

1.cdn恢复近期资质信息，解除了审核流程的阻塞

2.对于cdn无法恢复的历史资质，由于数据库历史备份数据不全无法完全恢复。资质替代方案风险、不确定性较高，属于下策。

结论：现有的依赖架构、备份机制下，对于CDN故障对应用造成的影响对我们无计可施。

案例剖析二：资质核心数据可用性

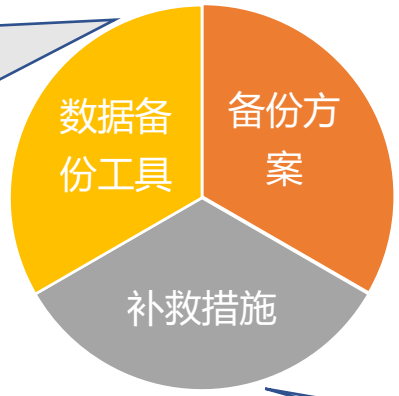
安全性反思优化

原因：单一依赖、无替补措施、数据备份不足

风险弥补：需解决三个问题

最终方案：资质双写双读策略——采用DFS存储备份，同步备份，实时补救的

数据库:数据备份代码实现维护简单但查询速度慢。
DFS:数据备份代码实现、维护较复杂但查询速度快。

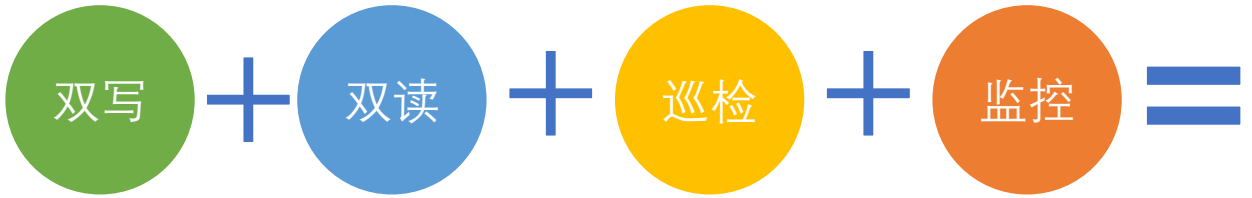


异步备份：定时从cdn抓取已经保存的图片保存到备份存储器。实现简单，但仍会有丢失数据的风险。
同步备份：同时向备份存储器保存图片。实现较复杂，但可保证数据完全备份。

实时补救：cdn和备份存储器其中一个故障可即刻切换到另一个。应用安全性高后期维护成本低，但实现较为复杂。
阻塞式补救：应用依赖CDN，CDN某图片故障时，备份图片重传到CDN。开发成本低，但遇到CDN故障时处理成本较高。

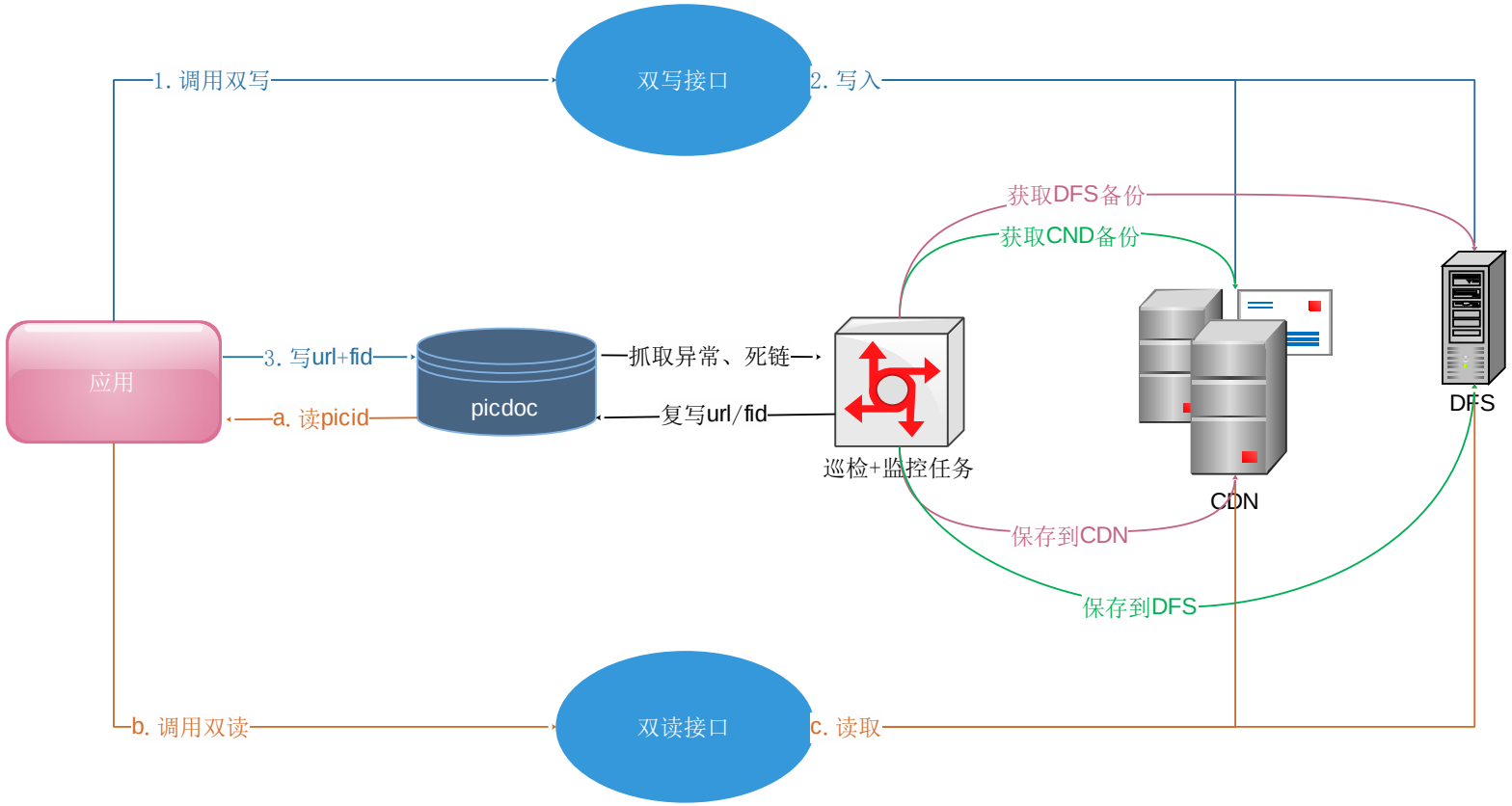
案例剖析二： 资质核心数据可用性

方案设计：



待解决：

1. **解决核心数据安全性**
2. **实现备份数据自动切换**
3. **易接入性**



案例剖析二：资质核心数据可用性

经验总结

服务依赖数据备份	对于核心业务对第三方服务的依赖，在享用服务带来便利的同时要有自己保底的灾备措施。核心数据一定要做好备份，预防万一事件。
监控机制	监控机制要完善，数据异常时能够及时发现，甚至利用备份数据自动恢复。
展望	学会未雨绸缪，及早对单点依赖做容灾预防准备。

高并发读写设计原理

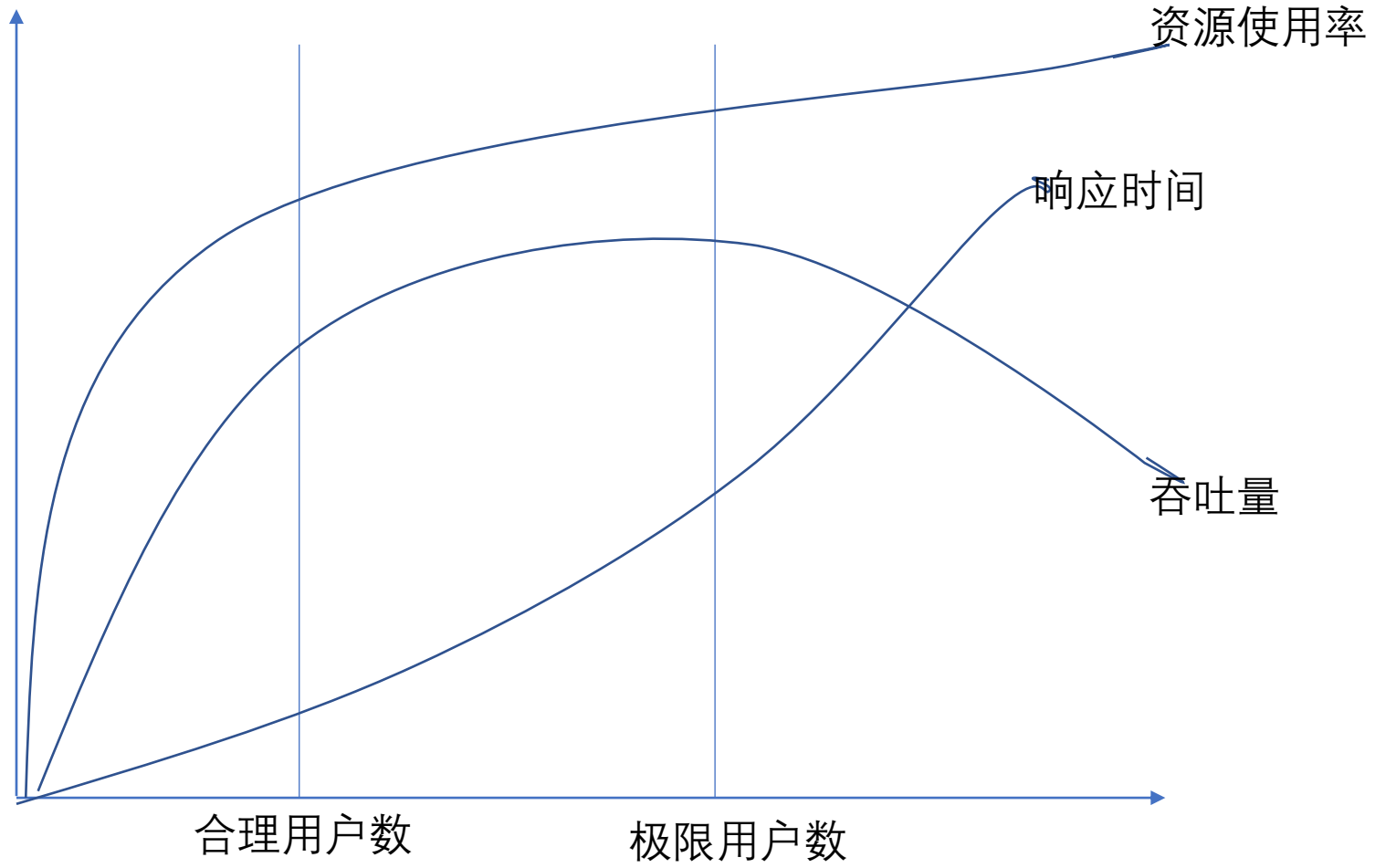
如何考虑系统承载量的评估与规划?

常用基本概念：吞吐量、响应时间、并发数
吞吐量：单位时间内可处理的请求数量。QPS、TPS。
响应时间：处理每个请求所需时间。TP95、TP99
并发数：服务器能同时处理的请求个数。

吞吐量 * 响应时间 = 并发数（单cpu）

并行系统?

高并发读写设计原理



并发系统模型比串行系统更复杂

高并发读写设计原理

- 架构设计时做好容量规划

机器数 = 总请求预估/单机能承载的最大请求

总请求如何预估？平均值还是最高峰值？

高并发读写设计原理

方法一：利用IO和CPU耗时比例估算

200ms = 30ms（cpu）+120ms（io）+50ms（内存）

充分利用cpu的方法：开多线程、异步IO

方法二：压力测试

准备好环境

核心读写接口压力测试

全链路压力测试与单机压力

思考：假设A应用（web系统）压力测试下，单机QPS200,为了承载1000QPS，于是部署了5个实例，是否合理？

预估好了QPS，接下来做什么？

高并发读写设计原理

限流：通过限制单位时间内请求次数的方式来实现保护服务应用的一种策略。(类似景区的限流)

单机限流、中央总限流、按场景限流

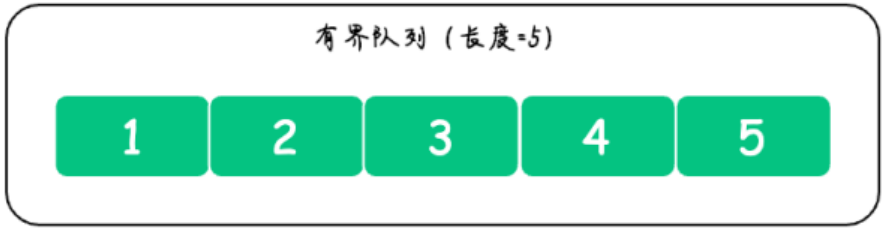
高并发读写设计原理

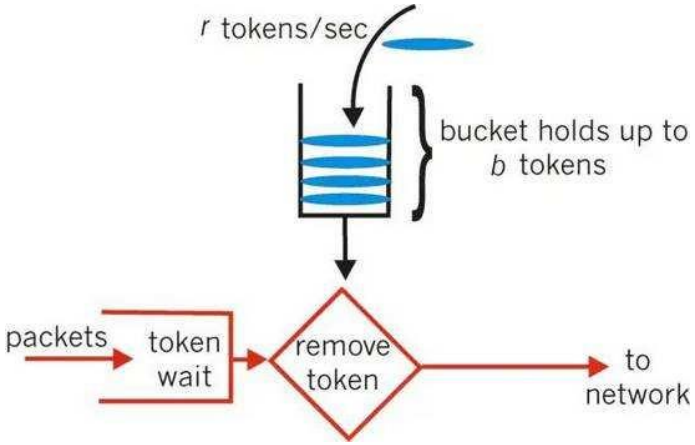
单机限流:

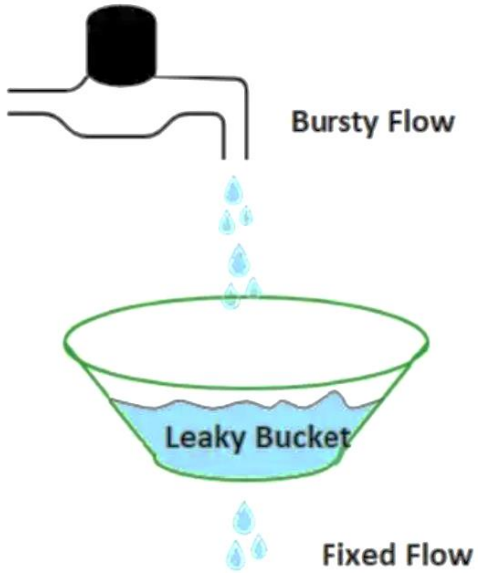
1、有界的任务队列

2、令牌桶算法

3、漏桶算法







高并发读写设计原理

思考一个问题：

一个桥承重5吨，大象重5.1吨，问大象要过桥，怎么过？

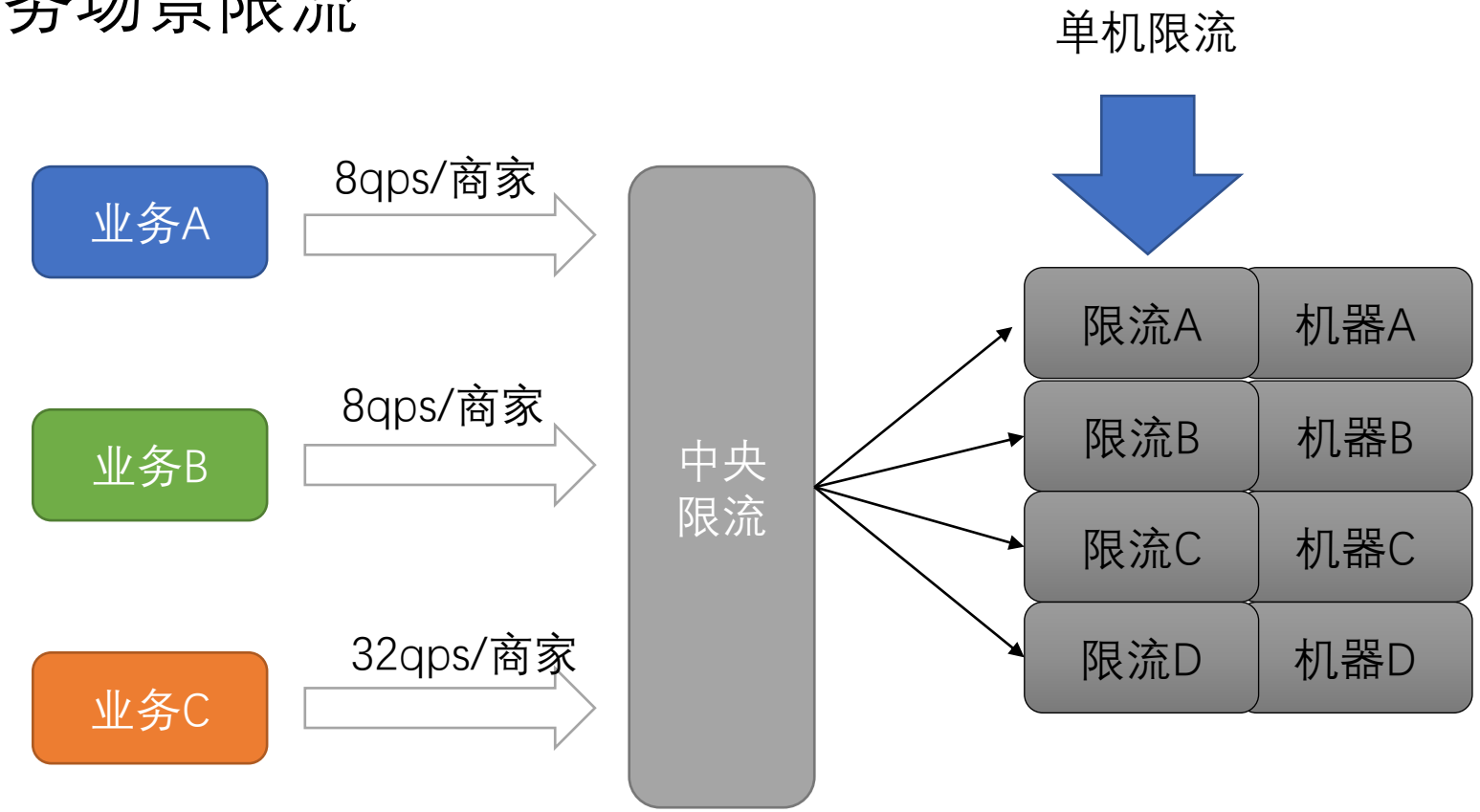
高并发读写设计原理

中央限流：用来控制总流量。

比如单机可承载100QPS，10台的集群即可控制在1000。但你如何保证每台机器恰好平均分到1/10的总流量？

高并发读写设计原理

按业务场景限流



高并发读写设计原理

熔断与降级

熔断一般是服务调用者对自我体系的保护机制。“熔断”概念最常见于电路中，当电路发生异常有烧毁全部电路风险时，熔断器会自动熔断，切换电路以达到保护全局的目的。

1）基于请求失败率为基准做熔断策略

2）基于请求耗时为基准做熔断策略

高并发读写设计原理

1）基于请求失败率为基准做熔断策略

以Histrix为例：

circuitBreaker.requestVolumeThreshold = 20；//请求批次数

circuitBreaker.seepWindowInmilliseconds = 5000；//熔断再检测间隔

circuitBreaker.errorThresholdPercentage = 50；//失败率阈值

每20个请求中，当存在50%的概率失败时，熔断器就会开启。n秒后，重新检测触发条件，若不再符合，则关闭熔断器，否则继续开启。

高并发读写设计原理

2）基于请求耗时为基准做熔断策略

以Sentinel为例：

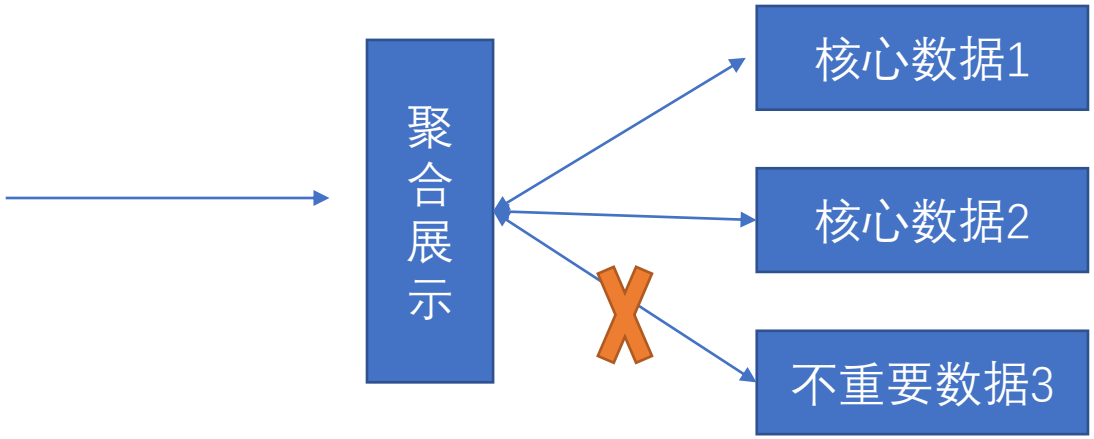
```
DegradeRule rule  = new DegradeRule();
Rule.setResource(“xxx”);
Rule.setCount(50);
Rule.setGrade(RuleConstant.DEGRADE_GRADE_RT);
Rule.setTimeWindow(5000);
```

当平均耗时大于50ms，且接下来的连续n个请求耗时都超过50ms，那在接下来 的5000ms内，熔断器都会开启。

高并发读写设计原理

熔断与降级

降级往往是服务调用者为了保证自身核心服务和数据的稳定，对非核心功能和流程采取的“忽略”行为。比如：



高并发读写设计原理

- 熔断与降级

1.触发条件不同

服务熔断一般是某个服务挂掉了引起的，一般是下游服务，而服务降级一般是从整体的负荷考虑，主动降级；

2.管理目标的层次不同

熔断其实是一个框架级的处理，每个微服务都需要。

降级一般需要对业务有层级之分。

高并发读写设计原理

- 熔断与降级-补充

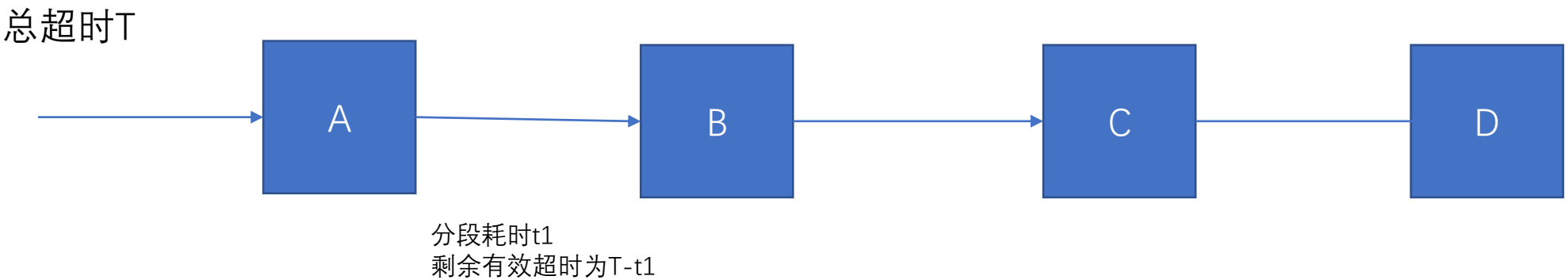
降级不单单是接口请求级别的降级，也可以是功能主动降级和服务主动降级。

比如之前某电商在大促前夜，为了降低DB读写压力，会主动关闭评论相关功能，以保证下单交易的顺利进行。

高并发读写设计原理

- 请求重试

- 单次rpc设置500ms超时，总接口1s超时是否合理?
- 请求重试应该注意哪些问题？（分享一个慢SQL引发的雪崩）



高并发读写设计原理-小结

- 高并发读:

动静分离与CND加速、加缓存、异步化、转批量、重写轻读

- 高并发写:

数据分片、任务分片、异步化(LSM树 log structured merged）、转批量

熔断与降级的分析